

PostgreSQL 9.2.4文書

PostgreSQLグローバル開発グループ

製作著作 © 1996-2013 PostgreSQL グローバル開発グループ

目次

[はじめに](#)

[PostgreSQLとは?](#)
[PostgreSQL小史](#)
[規約](#)
[より進んだ情報](#)
[バグレポートガイドライン](#)

I. [チュートリアル](#)

[1章さあ始めましょう](#)
[2章SQL言語](#)
[3章高度な諸機能](#)

II. [SQL言語](#)

[4章SQLの構文](#)
[5章データ定義](#)
[6章データ操作](#)
[7章問い合わせ](#)
[8章データ型](#)
[9章関数と演算子](#)
[10章型変換](#)
[11章インデックス](#)
[12章全文検索](#)
[13章同時実行制御](#)
[14章性能に関するヒント](#)

III. [サーバの管理](#)

[15章ソースコードからインストール](#)
[16章Windowsにおけるソースコードからのインストール](#)
[17章サーバの準備と運用](#)
[18章サーバの設定](#)
[19章クライアント認証](#)
[20章データベースロール](#)
[21章データベース管理](#)
[22章多言語対応](#)
[23章定常的なデータベース保守作業](#)
[24章バックアップとリストア](#)
[25章高可用性、負荷分散およびレプリケーション](#)
[26章リカバリの設定](#)
[27章データベース活動状況の監視](#)
[28章ディスク使用量の監視](#)
[29章信頼性とログ先行書き込み](#)
[30章リグレッションテスト](#)

IV. [クライアントインタフェース](#)

[31章libpq - C ライブラリ](#)

- 32章 [ラージオブジェクト](#)
- 33章 [ECPG - C言語による埋め込みSQL](#)
- 34章 [情報スキーマ](#)

V. [サーバプログラミング](#)

- 35章 [SQLの拡張](#)
- 36章 [トリガ](#)
- 37章 [ルールシステム](#)
- 38章 [手続き言語](#)
- 39章 [PL/pgSQL - SQL手続き言語](#)
- 40章 [PL/Tcl - Tcl手続き言語](#)
- 41章 [PL/Perl - Perl手続き言語](#)
- 42章 [PL/Python - Python手続き言語](#)
- 43章 [サーバプログラミングインタフェース](#)

VI. [リファレンス](#)

- I. [SQLコマンド](#)
- II. [PostgreSQLクライアントアプリケーション](#)
- III. [PostgreSQLサーバアプリケーション](#)

VII. [内部情報](#)

- 44章 [PostgreSQL内部の概要](#)
- 45章 [システムカタログ](#)
- 46章 [フロントエンド/バックエンドプロトコル](#)
- 47章 [PostgreSQLコーディング規約](#)
- 48章 [多言語サポート](#)
- 49章 [手続き言語ハンドラの作成](#)
- 50章 [外部データラップの作成](#)
- 51章 [遺伝的問い合わせ最適化](#)
- 52章 [インデックスアクセスメソッドのインタフェース定義](#)
- 53章 [GiSTインデックス](#)
- 54章 [SP-GiSTインデックス](#)
- 55章 [GINインデックス](#)
- 56章 [データベースの物理的な格納](#)
- 57章 [BKIバックエンドインタフェース](#)
- 58章 [プランナが統計情報をどのように使用するか](#)

VIII. [付録](#)

- A. [PostgreSQLエラーコード](#)
- B. [日付/時刻のサポート](#)
- C. [SQLキーワード](#)
- D. [SQLへの準拠](#)
- E. [リリースノート](#)
- F. [追加で提供されるモジュール](#)
- G. [追加で提供されるプログラム](#)
- H. [外部プロジェクト](#)
- I. [ソースコードリポジトリ](#)
- J. [ドキュメント作成](#)
- K. [頭字語](#)

[参考文献](#)

[目次](#)

法的告知

Copyright © 1996-2013 PostgreSQLはPostgreSQLグローバル開発チームが著作権を有します。

Copyright © 1994-1995 Postgres95はカリフォルニア大学評議員が著作権を有します。

Permission to use, copy, modify, and distribute this software and its documentation for any purpose, without fee, and without a written agreement is hereby granted, provided that the above copyright notice and this paragraph and the following two paragraphs appear in all copies. [訳注：日本語は参考程度と解釈してください。] 上記の著作権表示、および 本段落と続く2つの段落を全てのコピーに含めることを条件として、無料かつ 書面による許可なしに、このソフトウェアとドキュメントの使用、複製、改変、再配布をどのような目的にでも許可します。

IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, INCLUDING LOST PROFITS, ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF THE UNIVERSITY OF CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. カリフォルニア大学は、いかなる当事者に対しても、利益の喪失を含む、直接的、間接的、特別、偶然あるいは必然的にかかわらず生じた 損害について、たとえカリフォルニア大学がこれらの損害の可能性について 知らされていたとしても、一切の責任を負いません。

THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE SOFTWARE PROVIDED HEREUNDER IS ON AN "AS-IS" BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATIONS TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS. カリフォルニア大学は、商用目的における暗黙の保証と、特定目的での 適合性に関してはもとより、これらに限らず、いかなる保証もしません。 以下に用意されたソフトウェアは「そのまま」を基本原理とし、 カリフォルニア大学はそれを維持、支援、更新、改良あるいは修正する 義務を負いません。

[ホーム](#)

はじめに

目次

[PostgreSQLとは?](#)[PostgreSQL小史](#)[規約](#)[より進んだ情報](#)[バグレポートガイドライン](#)

本書はPostgreSQLのオフィシャルドキュメントです。 PostgreSQLソフトウェアの開発と並行してPostgreSQL開発者とそれ以外のボランティアにより書かれてきました。 現在のバージョンのPostgreSQLが公式にサポートする全ての機能を網羅しています。

PostgreSQLの膨大な情報を取り扱いやすくするため、本書はいくつかの部分で構成されています。 それぞれの部分は異なるユーザクラスを対象としており、ユーザに対してもPostgreSQLの経験に異なる段階があることを意識しています。

- [パートI](#)は新規ユーザ向けの形式ばらない入門編です。
- [パートII](#)はユーザレベルのパフォーマンスチューニング、データ型や関数などのSQL問い合わせ言語環境についてを文書化したものです。 PostgreSQLユーザは皆、一読すべきです。
- [パートIII](#)はサーバのインストールと管理について記載しています。 PostgreSQLを実行させているユーザは全て、個人用やその他用途などの目的にかかわらず、この部分を読むべきです。
- [パートIV](#)はPostgreSQLのクライアントプログラム用プログラミングインタフェースについて記載しています。
- [パートV](#)にはサーバの拡張機能について上級ユーザ向けに情報を提供します。 話題にはユーザ定義型やユーザ定義関数があります。
- [パートVI](#)にはSQLコマンド、クライアントおよびサーバプログラムに関する情報があります。 この部分では、他で構成された情報を補助するために、コマンドやプログラムを順番に説明します。
- [パートVII](#)にはPostgreSQL開発者が利用可能な類別された情報があります。

PostgreSQLとは?

PostgreSQLは、カリフォルニア大学バークレイ校のコンピュータサイエンス学科で開発された[POSTGRES, Version 4.2](#)をベースにしたオブジェクトリレーショナルデータベース管理システム（ORDBMS）です。POSTGRESは後からいくつかの商用データベースで利用できるようになった、多くの概念についての先駆となりました。

PostgreSQLはオリジナルのバークレイ校のソースコードを引き継ぐオープンソースのデータベースで、標準SQLの大部分やその他の最新の機能をサポートしています。

- 複雑な問い合わせ
- 外部キー
- トリガ
- ビュー
- トランザクションの一貫性（完全性）
- 多版同時実行制御

またPostgreSQLは、例えば新規に以下のものを付け加えることで、いろいろな方法でユーザが拡張することができます。

- データ型
- 関数
- 演算子
- 集約関数
- インデックスメソッド
- 手続き言語

さらに自由主義的ライセンス条件により、PostgreSQLは誰にでも、その使用、変更、配布を個人使用、商用、学術など目的に限らず無償で可能です。

PostgreSQL小史

現在PostgreSQLとして知られるオブジェクト指向リレーショナルデータベース管理システムは、カリフォルニア大学バークレイ校で作成されたPOSTGRESパッケージから派生しています。20年以上にわたる開発の背景を持ち、PostgreSQLは現在最も先端的な、どこからでも入手可能なオープンソースデータベースです。

バークレイ校POSTGRESプロジェクト

Michael Stonebraker教授率いるPOSTGRESプロジェクトにはその後援者としてDefense Advanced Research Projects Agency (DARPA)、National Science Foundation (NSF)、そしてESL, Inc.が名を連ねていました。POSTGRESの実装は1986年から始まりました。当初のシステムに対する概念は [The design of POSTGRES](#) で発表され、最初のデータモデルの定義は [The POSTGRES data model](#) で紹介されました。当時のルールシステムの設計は [The design of the POSTGRES rules system](#) で説明されました。ストレージ管理の理論や構造は [The design of the POSTGRES storage system](#) で詳しく示されました。

Postgresはそれ以来いくつかの主要なリリースを重ねてきました。最初の"デモウェア"システムが1987年に使用可能になり、1988年のACM-SIGMODコンファレンスで紹介されました。 [The implementation of POSTGRES](#) で説明されているバージョン1は、1989年6月に一部の外部ユーザにリリースされ、最初のルールシステムに対する批評の結果 ([A commentary on the POSTGRES rules system](#)) を基にルールシステムは再設計 ([On Rules, Procedures, Caching and Views in Database Systems](#)) され、バージョン2が1990年6月に新しいルールシステムを実装してリリースされました。バージョン3は1991年に登場し、複数ストレージ管理機構、改善された問い合わせエクゼキュータ、書き直しされたルールシステムのサポートが追加されました。Postgres95まで引き続いた各リリース (下記を参照) のほとんどの部分では、移植性と信頼性に焦点を合わせていました。

POSTGRESは様々な研究用、そして実際の業務アプリケーションを実装するために使われてきています。その中には、金融データ分析システム、ジェットエンジン性能分析パッケージ、小惑星追跡データベース、医療情報データベース、いくつかの地図情報処理システム (GIS) などがあります。POSTGRESはさらに、いくつかの大学で教材としても使われています。最後に、Illustra Information Technologies社 (後に、[Informix](#)社に吸収合併され、現在はIBM社所有) がコードを整理し商用化しました。POSTGRESは1992年後半から始まった [Sequoia 2000 scientific computing project](#) の主要なデータ管理システムになりました。

1993年には外部ユーザコミュニティの大きさは約2倍に膨れました。データベースの研究に費されるべき時間がプロトタイプコードの保守とサポートに取られていることが次第に明らかになってきました。このサポートの重荷を減らすために、バークレイPOSTGRESプロジェクトはバージョン4.2をもって公式に終了しました。

Postgres95

1994年、Andrew YuとJolly ChenがPOSTGRESにSQL言語インタプリタを追加しました。引き続いてPostgres95がWeb上でリリースされ、オリジナルのPOSTGRESバークレイコードのオープンソースによる後続として世界への独自の道を歩み始めました。

Postgres95のコードは全てANSI C準拠となるように書き直され、これまでに膨れ上がったコードの25%を整理することで身軽になりました。多くの内部改造によって性能と保守性が改善されました。Postgres95リリース1.0.xは、POSTGRESバージョン4.2に比べWisconsin Benchmarkで約30~50%速く動作しました。バグ修正以外では、下記の大きな改善がありました。

- (サーバに実装された) SQLが問い合わせ言語PostQUELに取って代わりました。PostgreSQLになる以前は副問い合わせはサポートされていませんでしたが (下記を参照)、Postgres95ではユーザ定義SQL関数で模倣できました。集約は再実装されました。GROUP BY問い合わせ句のサポートも追加されました。
- GNUのReadLineを使った新しいプログラム (psql) が、対話式SQL問い合わせのために用意されました。これは古いmonitorプログラムにほぼ取って代わるものになりました。
- 新しいフロントエンドライブラリ、libpgtclがTclベースのクライアントをサポートしました。サンプルシェルスクリプトpgtclshはTclとPostgres95サーバとインタフェースをとる新規Tclコマンドを提供しました。
- ラージオブジェクトインタフェースがオーバーホールされました。転置ラージオブジェクトが唯一のラージオブジェクト格納機構でした (転置ファイルシステムは削除されました)。
- インスタンスレベルのルールシステムが削除されました。ルールは書き換えルールとしてまだ利用できました。
- 標準SQLの機能やPostgres95の機能を紹介したチュートリアルがソースコードとともに配布されました。

- GNU makeが（BSD makeの代わりに）構築に使われました。また、Postgres95はパッチの当たっていないGCCでコンパイルできました（doubleにおけるデータ整列が修正されたおかげです）。

PostgreSQL

1996年になると"Postgres95"という名前が時代の試練に耐えられなくなったことが明らかになりました。そこで、オリジナルのPOSTGRESとSQLの能力を持つ、より最近のバージョンとの関係を反映する、PostgreSQLという新しい名前を選びました。同時に、元々バークレイPOSTGRESプロジェクトで始まった連番に戻す番号の6.0で始まるバージョン番号を設定しました。

歴史的な理由か発音しやすいためか、多くの人々はPostgreSQLを"Postgres"と参照しつづけています。（いまではすべて大文字と記載することは稀です。）こうした使い方は愛称や別名として広く受け入れられています。

Postgres95開発で重視されたのは、サーバのコードに内在する問題点を特定し、原因を理解することでした。PostgreSQLにおいては、全ての分野に目を留めているとしても、保守作業を続けつつ特徴や能力を強化することに重点が移りました。

その後PostgreSQLがたどった足跡の詳細は[付録E](#)を参照してください。

[前のページ](#)

PostgreSQLとは？

[ホーム](#)

[上に戻る](#)

[次のページ](#)

規約

規約

本書ではテキストのある部分を特色付けるために次のような印字上の規約を使用します。新規の語彙、外来の語法、そしてその他の重要なセンテンスはイタリックで強調されます。特にコマンド、プログラムコード、そして画面出力などですが、コンピュータの入出力を表現するものはすべて固定幅フォント（例題）で示します。こうした一節の中のイタリック（例）はプレースホルダを意味しており、そのプレースホルダの場所に実際の値を挿入することになります。場合によって、もし前に引用した事例に加えられたり変更されたりしたプログラムコードの一部は、太文字（例）で強調されます。

以下の規約はコマンドの概要に対して使用されます。大括弧（`[`および`]`）はオプションである部分を示します。（Tcl コマンドの概要では疑問符（`?`）をTclのしきたりとして代わりに使います。）中括弧（`{`および`}`）そして縦線（`|`）はいずれか1つを選択しなければならないことを意味します。点々（`...`）は前にある要素が繰り返されるという意味です。

明確性を強調する場面では、SQLコマンドの前には`=>`を付け、シェルコマンドの前には`$`を付けます。とは言っても通常プロンプトは示しません。

一般的に管理者とは、サーバのインストールと運営の責にあたる人のことです。ユーザとは、PostgreSQLシステムの一部でも活用したり、もしくはこれから使用する誰もがその対象です。これらの用語は厳密に解釈されるものではありません。本書はシステム管理の手順について特定の固定観念を想定していません。

より進んだ情報

ドキュメント、つまり本書の他に、PostgreSQLについてのその他のリソースがあります。

Wiki

PostgreSQL [wiki](#)はプロジェクトの[FAQ](#)（よくある質問）リスト、[TODO](#)リスト、およびさらに多くの話題の情報を収容しています。

Webサイト

PostgreSQLの[Webサイト](#)には、最新のリリースに関する詳細についてや、PostgreSQLの生産性をより高めて利用・操作する上でのその他の情報があります。

メーリングリスト

メーリングリストは、ユーザが質問の答えを得ること、他のユーザと経験を共有すること、開発者に連絡することに適した所です。 PostgreSQLのWebサイトで調べてください。

あなた自身！

PostgreSQLはオープンソースプロジェクトです。つまり、ユーザコミュニティの継続的なサポートに依存しています。 PostgreSQLを使い始めてからは、ドキュメントやメーリングリストを通じて他の人からの助けに頼ることになるでしょう。 知識を得たらそれを今度は貢献することを考えてください。 メーリングリストを講読し質問に回答してください。 もし、ドキュメントに載っていないことを学んだら、それを書いて寄稿してください。 もし、コードに機能を追加したら、それを寄稿してください。

バグレポートガイドライン

PostgreSQLに関してバグを発見した場合、ぜひ我々に連絡してください。最大限の注意を払っても、全てのプラットフォーム、全ての環境でPostgreSQLの機能全てが正常に動くことは保証できませんので、バグレポートはPostgreSQLをより信頼性の高いものにするために、大変重要になります。

下記の助言は、有効に活用され得るバグレポートを作成する際に、作成者を支援することを狙ったものです。これに従う義務はありませんが、沿った方がより有益なものとなるでしょう。

私たちは、すべてのバグを直ちに修正することを約束することはできません。そのバグが明確で、重大で、あるいは他の多くのユーザにも影響を与えるものであれば、誰かがすぐに調査する可能性が高くなるでしょう。また、より新しいバージョンに変えて、そこでも同じようなことが起こるかを確認してもらうように伝える場合もあります。あるいは、現在計画中の大きな変更が終了するまで、バグを修正できないと判断する場合もあります。また、単に修正が非常に困難であり、より重要な他の事項があることもあります。早急に処置が必要な場合は、商用サポートの購入を検討してください。

バグの特定

バグ報告を行う前に、ドキュメントを読み、もう一度読み返し、実行しようとしている処理が実行可能かどうか確認してください。実行可能かどうか不明な場合は、その旨を報告してください。それはドキュメントのバグです。また、ドキュメントに書かれていることと実際の結果が異なる場合にはバグとなります。以下のような状況が考えられます。但し、これらに限定しているわけではありません。

- プログラムが致命的なシグナル、またはオペレーティングシステムのエラーメッセージで終了し、それがプログラム内部の問題を指摘している場合（逆に、"disk full"のようなメッセージはプログラムの問題ではありませんから、この場合は自分で修正してください）。
- あるプログラムで、入力された値に対して間違った結果を返す場合。
- （ドキュメントで定義されている）有効な値を入力してもプログラムで受け付けない場合。
- プログラムが、無効な入力値を通知やエラーメッセージなどを表示させず受け入れる場合。但し、無効な入力と思われるものでも、拡張、あるいは過去の経緯による互換性と考えられている可能性があることに注意してください。
- サポートされているプラットフォームで、PostgreSQLが手順通りにコンパイル、ビルド、インストールできない場合。

ここでは、"プログラム"とはバックエンドプロセスだけではなく、すべての実行可能ファイルを意味します。

プログラムの実行が遅かったり、リソースを大量に使用するのは必ずしもバグではありません。アプリケーションを改善するためには、ドキュメントを読んだり、どこかのメーリングリストで尋ねてみたりしてください。標準SQLの要求に応じない場合も、その機能の互換性を明確にうたっていない限り、バグとは言えません。

以降に進む前に、TODOリストやFAQを参照して、そのバグが既知のものかどうか確認してください。もしTODOリストの情報を読み取ることができなければ、問題を報告してください。少なくともTODOリストを分かりやすくすることができます。

報告すべきこと

バグ報告で最も重要なことは、全ての事実を、そして事実のみを明確に記述することです。何が起こったのか、または、プログラムのどこが問題か、"何々が起きているようだ"などの憶測や推測を記述しないでください。実装にさほど詳しくない方の推測は正しくない場合があり、有効なバグ報告になりません。実装に精通している方の場合であっても、根拠のある説明は参考情報となりますが、やはり正しい事実が一番役に立ちます。バグを修正するためには、まず開発者自身がそのバグを再現する必要があります。ありのままの事実を報告することは、単刀直入（多くの場合は画面からメッセージをコピー&ペーストを行うのみ）ですが、えてして、重要でないだろうと想像したり、省いても理解してもらえらるだろうという思い込みによって、重要な情報が洩れてしまう場合がかなり多くあります。

全てのバグ報告では、下記の内容が記述されていなければいけません。

- 問題を再現できるように、プログラムの起動から行った作業を順序通りに記述してください。例えば、出力がテーブルのデータに依存するならば、単にSELECT文を記述していても、それ以前に行われた、CREATE TABLEやINSERT文が記述されていなければ十分とはいえません。我々は、ユーザのデータベーススキーマをリバースエンジニアリングするための時間を取ることができませんし、推測してデータを構築したとしても、おそらく間違えることになるでしょう。

SQL関連の問題のテストケースの最適な書式は、psqlフロントエンドに直接読み込ませて問題を再現することができる

ファイルを用意することです（`~/.psqlrc`の起動ファイルに何も書かれていないことを確認してください）。このファイルを簡単に作成するには、`pg_dump`を使ってテーブル定義とその状況を再現させるために必要なデータを取り出し、問題の起こった問い合わせを追加します。サンプルデータの量を減らすことは、推奨されますが必ずしも必要ではありません。どのような方法であれ、バグが再現できればよいのです。

アプリケーションがPHPなど何か別のクライアントインタフェースを使用している場合、問題となる問い合わせを切り出してください。問題を再現させるために我々がWebサーバをセットアップすることは、おそらくないでしょう。どのような場合においても、正確な入力ファイルを提供することを忘れないでください。"大規模ファイル"や"中規模データベース"で発生する問題である、といった推測は行わないでください。こうした情報は不正確過ぎて役に立ちません。

- 得られた出力そのもの。"うまくいかなかった"、あるいは"クラッシュした"といった報告はしないでください。エラーメッセージがあるならば、たとえ意味が理解できなくてもそれを報告してください。オペレーティングシステムのエラーでプログラムが強制終了してしまったら、どのエラーでそうなったのかを報告してください。何も起こらない場合も、その旨を報告してください。たとえテストケースの結果がプログラムのクラッシュなど明確な場合でも、我々のプラットフォームで再現できない場合があります。最も容易な方法は、出力をターミナルからコピーすることです。

注意: エラーメッセージを報告する場合、そのメッセージを最大限詳細に取得してください。psqlでは、前もって`\set VERBOSITY verbose`を指定してください。サーバログからメッセージを取り出す場合は、全ての詳細をログに取得できるように[log_error verbosity](#)実行時パラメータを`verbose`に設定してください。

注意: 致命的なエラーが起こった場合、クライアント側で報告されるエラーメッセージには得られる情報が全て書かれているとは限りません。データベースサーバのログも見てみてください。もしログを取っていないならば、取る習慣を付けるいいタイミングです。

- どのような出力を望んでいたのかを記述することも非常に重要です。ただ単に"このコマンドはこのような出力を返した"や、"期待していた結果ではない"だけでは、再現して結果を検証した際、開発者は、これは期待した通りの正しい結果である、と考えるかもしれません。送られてきたコマンドの背後にある文脈を全て把握することはできません。また、特に"SQLではこう書かれていない/Oracleではこのようにならない"というコメントはご遠慮願います。SQLの正確な動作を探し出すのは楽しい作業ではありませんし、また、世にある他のリレーショナルデータベースの動作全てをPostgreSQLの開発者が把握しているわけでもありません（問題がプログラムのクラッシュである場合、この内容は言うまでもなく省略できます）。
- すべてのコマンドラインオプションと起動時のオプション、デフォルトから変更した関連する環境変数や設定ファイル。繰り返しますが、正確な情報を提供してください。OSの起動時にデータベースサーバを起動するようにパッケージされたディストリビューションを使用している場合は、それらがどのように実行されているかを確認する必要があります。
- インストールの手順書から変更して実行したすべての内容。
- PostgreSQLのバージョン。 `SELECT version();` で、接続しているサーバのバージョンがわかります。多くの実行可能なプログラムでは`--version`オプションも使用できます。少なくとも`postgres --version`と`psql --version`は実行できるはずです。これらの関数やオプションが使用できない場合、アップグレードが保証されているものよりも、さらに古いバージョンです。RPMなどパッケージ化されたものを使用している場合は、その旨を連絡し、パッケージに付加されたバージョン番号があれば、それも記載してください。Git版に対するバグ報告の場合は、その旨も記載し、コミットハッシュの情報も含めてください。

9.2.4よりもバージョンが古い場合、アップグレードすることをお勧めします。新しいリリースでは多くのバグ修正や改良がなされているからです。ですので、古めのPostgreSQLのリリースを使用していて遭遇した不具合が修正されている可能性がかなりあります。古いPostgreSQLのリリースを使用しているサイトに対して、我々は限定されたサポートしか提供することができません。それ以上のサポートが必要であれば、商用サポート契約を結ぶことを検討してください。

- プラットフォーム情報。カーネル名とバージョン、Cライブラリ、プロセッサ、メモリ情報なども含めて報告してください。多くの場合、ベンダ名とそのバージョンを明記するだけで十分ですが、"Debian"の正確な構成要素を全ての人間が把握している、であるとか、全ての人間がi386系を使用しているなどの思い込みは止めてください。インストールに関する問題の場合は、マシンのツール群（コンパイラやmakeなど）の情報も必要となります。

バグ報告が長文になってもそれは仕方がないことなので、気にしないでください。一度に全ての情報を入手できる方が、開発者から情報を催促するよりも手間暇がかかりません。その一方、ファイルが大きいならば、その情報に誰か興味があるかを最初に尋ねるのが得策かもしれません。記事には、バグ報告に関するその他のTIPSの概要があります。

問題を解決させる入力を見つけ出すための試行錯誤に時間をかけないでください。これはおそらく問題解決の助けにはなりません。バグが即座に修正されない場合、その時間を利用すれば、あなた自身のワークアラウンドを探して共有することができます。繰り返しになりますが、バグがなぜあるのかを解明するのに余計な時間をかける必要はありません。開発者の方が十分速くそれを見つけ出します。

バグ報告をする際、理解しやすい用語を使用してください。このソフトウェアパッケージ全体は"PostgreSQL"と呼ばれていますが、略して"Postgres"とも呼ばれます。特にバックエンドプロセスに関して述べる時は、そのように明記し、"PostgreSQLがクラッシュする"とは記述しないでください。1つのバックエンドプロセスのクラッシュと、その親プロセ

ス"postgres"のクラッシュとはかなり異なります。1つのバックエンドがダウンしてしまったことを、"サーバがクラッシュした"とは記述しないでください。その逆の場合にも当てはまります。また、"psql"対話式フロントエンドなどのクライアントプログラムはバックエンドとは完全に分離されています。問題がクライアント側かサーバ側かの切り分けを試みてください。

バグ報告先

一般論として、バグ報告は<pgsql-bugs@postgresql.org>というバグ報告用メーリングリストに送ってください。バグ報告の題名には、エラーメッセージの一部分などわかりやすいものを使ってください。

その他の方法として、プロジェクトの[Webサイト](#)にあるバグ報告フォームの項目を埋める方法があります。この方法で入力したバグ報告は、<pgsql-bugs@postgresql.org>メーリングリストに送信されます。

バグ報告にセキュリティが関連する場合や公開アーカイブからすぐに関連できることを好まない場合、`pgsql-bugs`には送信しないでください。セキュリティの問題については、非公開で<security@postgresql.org>に報告することができます。

<pgsql-sql@postgresql.org>や<pgsql-general@postgresql.org>などのユーザ向けのメーリングリストには決してバグ報告を送らないでください。これらのメーリングリストはユーザからの質問に答えるためのもので、ほとんどの購読者はバグ報告を受け取りたくないと思われます。さらに重要なのは、これらのリストの購読者によってバグが修正されることはほとんどないということです。

また、開発者向けの<pgsql-hackers@postgresql.org>にもバグ報告を送らないでください。ここはPostgreSQLの開発に関して議論する場で、バグ報告とは切り離している方が良くとされています。もしその問題により多くのレビューが必要な場合は、そのバグ報告を`pgsql-hackers`で議論することになります。

ドキュメントに関して問題がある場合は、ドキュメント用のメーリングリスト、<pgsql-docs@postgresql.org>が最適な報告先です。その際、問題になった箇所がどの部分かを明記してください。

また、サポートされていないプラットフォームへの移植に関係するバグ報告である場合は<pgsql-hackers@postgresql.org>に報告してください。そのプラットフォームへPostgreSQLを移植するように（報告者と一緒に）最善の努力をします。

注意: spamメールを防止するために、残念なことに、これらのメーリングリストは非公開となっています。つまり、これらのメーリングリストに投稿するには、購読していなければいけません（但し、Webフォームによるバグ報告の場合には、メーリングリストを購読している必要はありません）。メーリングリストからのメールを受け取らずに単にメールを送りたい場合は、購読登録を行い、購読オプションを`nomail`に設定してください。詳細な情報については<majordomo@postgresql.org>宛に、`help`とだけ本文に書いてメールを送ってください。

[前のページ](#)
より進んだ情報

[ホーム](#)
[上に戻る](#)

[次のページ](#)
チュートリアル

I. チュートリアル

PostgreSQLのチュートリアルによるこそ。本書は数章から構成され、ここで、PostgreSQLやリレーショナルデータベースの概念、SQL言語を初めて使用する方向けに、簡単に紹介します。ここでは、コンピュータの一般的な使い方についての知識だけを前提としています。Unixやプログラミングに関する経験は必要ありません。ここでは、主にPostgreSQLシステムの重要なポイントについて実践的な経験を得ることを目的としています。扱っているトピックについての完全で詳細な処理を記述しているものではありません。

このチュートリアルの内容を読んだ後、SQL言語のより形式的な知識を学習したいのであれば、[パートII](#)を、PostgreSQL用のアプリケーションの開発に関する情報を学習したいのであれば、[パートIV](#)を続いて読んでください。また、サーバをセットアップおよび管理される方は、[パートIII](#)も参照してください。

目次

1章 [さあ始めましょう](#)

- 1.1. [インストール](#)
- 1.2. [構造的な基本事項](#)
- 1.3. [データベースの作成](#)
- 1.4. [データベースへのアクセス](#)

2章 [SQL言語](#)

- 2.1. [序文](#)
- 2.2. [概念](#)
- 2.3. [新しいテーブルの作成](#)
- 2.4. [テーブルに行を挿入](#)
- 2.5. [テーブルへの問い合わせ](#)
- 2.6. [テーブル間を結合](#)
- 2.7. [集約関数](#)
- 2.8. [更新](#)
- 2.9. [削除](#)

3章 [高度な諸機能](#)

- 3.1. [はじめに](#)
- 3.2. [ビュー](#)
- 3.3. [外部キー](#)
- 3.4. [トランザクション](#)
- 3.5. [ウィンドウ関数](#)
- 3.6. [継承](#)
- 3.7. [まとめ](#)

第 1 章 さあ始めましょう

目次

- 1.1. [インストール](#)
- 1.2. [構造的な基本事項](#)
- 1.3. [データベースの作成](#)
- 1.4. [データベースへのアクセス](#)

1.1. インストール

PostgreSQLを使用できるようにするためには、当然ながらインストールしなければなりません。 使用しているオペレーティングシステム配布物内に含まれていたり、システム管理者がインストールしていたりしますので、PostgreSQLが既にサイトにインストールされている可能性があります。 そのような場合、オペレーティングシステムの文書やシステム管理者からPostgreSQLへのアクセス方法に関する情報を得なければなりません。

PostgreSQLが既に使用可能かどうかや実験的に使用できるかどうかがわからなければ、独自にインストールすることができます。 インストールは難しくありません。 良い経験になるでしょう。 PostgreSQLは、非特権ユーザによって、つまり、スーパーユーザ (root) 権限を必要とすることなく、インストールできます。

自身でPostgreSQLをインストールする場合は、インストール手順を[第15章](#)で確認してください。 そしてインストールが完了してから、本書に戻ってきてください。 適切な環境変数の設定に関する節に正確に従っていることを確認してください。

システム管理者がデフォルトの方法で設定していなかった場合、他にも多少の作業をすることになります。 例えば、データベースサーバマシンがリモートマシンの場合、`PGHOST`環境変数をデータベースサーバマシンの名前に設定する必要があります。 また、`PGPORT`環境変数の設定も行わなければならないかもしれません。 要するに、アプリケーションプログラムを起動しようとして、データベースに接続できないというエラーが発生する場合には、サイト管理者と相談し、自分が管理者であれば、文書を読んで、環境が適切に設定されていることを確認してください。 これまでの内容を理解できない場合は、次の節を読んでください。

1.2. 構造的な基本事項

先に進む前に、PostgreSQLシステム構成の基礎を理解すべきです。PostgreSQLの各部分がどのように相互作用しているかを理解することにより、本章の内容がわかりやすくなります。

データベースの用語で言うと、PostgreSQLはクライアント/サーバモデルを使用しています。PostgreSQLのセッションは以下の協調動作するプロセス（プログラム）から構成されます。

- サーバプロセス。これは、データベースファイルを管理し、クライアントアプリケーションからのデータベースの接続を受け付け、クライアントに代わってデータベースに対する処理を行います。データベースサーバプログラムは`postgres`と呼ばれています。
- ユーザの、データベース操作を行うクライアント（フロントエンド）アプリケーション。クライアントアプリケーションはその性質上非常に多様性があります。テキスト指向のツール、グラフィカルなアプリケーション、データベースにアクセスしWebページを表示するWebサーバ、あるいはデータベースに特化した保守ツールなどがあります。PostgreSQLの配布物では、いくつかのクライアントアプリケーションを用意しています。そのほとんどはユーザによって開発されました。

典型的なクライアント/サーバアプリケーションでは、クライアントとサーバはホストが異なっても構いません。その場合、クライアントとサーバはTCP/IPネットワーク接続経由で通信を行います。このことには注意してください。なぜなら、クライアントマシンからアクセスできるファイルは、データベースサーバマシンではアクセスできない（または、異なるファイル名でアクセスできるだけである）可能性があるからです。

PostgreSQLサーバはクライアントから複数の同時接続を取り扱うことができます。これを達成するため、サーバは接続ごとに新しいプロセスを開始（"fork"）します。その時点から、クライアントと新しいサーバプロセスは元の`postgres`プロセスによる干渉がない状態で通信を行います。こうして、マスターサーバは常に稼働し、クライアントとの接続を待ち、一方、クライアントと関連したサーバプロセスは通信を行います（もちろんこれは全てユーザからはわかりません。ここでの説明は完全性のみを目的としています）。

1.3. データベースの作成

データベースサーバにアクセスできるかどうか分かる最初の試験は、データベースの作成を試みることです。稼働中のPostgreSQLサーバは多くのデータベースを管理することができます。典型的には、プロジェクトやユーザごとにデータベースは分割されます。

サイト管理者により使用できるデータベースが既に作成されている場合もあります。その場合、管理者は使用できるデータベースの名前をユーザに通知しなければなりません。この場合、この段階を飛ばして、次の節まで進んでください。

新しいデータベースを作成するには、以下のコマンドを使用してください。この例ではmydbという名前です。

```
$ createdb mydb
```

この手順で何も応答がなければ、この段階は成功し、本節の残りは飛ばすことができます。

以下のようなメッセージが現れた場合、PostgreSQLが正しくインストールされていません。

```
createdb: command not found
```

インストールがまったくされていないか使用しているシェルの検索パスが盛り込むように設定されていないのいずれかです。代わりに絶対パスでそのコマンドを実行してみてください。

```
$ /usr/local/pgsql/bin/createdb mydb
```

このパスはサイトによって異なるかもしれません。この問題を解決するには、サイト管理者に連絡するか、状況を修正するためインストール取扱説明書を調べてください。

他の応答として以下もあります。

```
createdb: could not connect to database postgres: could not connect to server: No such file or directory
Is the server running locally and accepting
connections on Unix domain socket "/tmp/.s.PGSQL.5432"?
```

これは、サーバが起動していないか、createdbが想定している状態でサーバが起動していないかを示しています。こちらも、インストール手順を点検するか管理者に相談してください。

他の応答として以下もあります。

```
createdb: could not connect to database postgres: FATAL:  role "joe" does not exist
```

今回はログインした時のユーザ名が記載されています。これは、管理者がそのユーザ用のPostgreSQLユーザアカウントを作成していない時に起こります（PostgreSQLユーザアカウントは、オペレーティングシステムのユーザアカウントとは異なります）。管理者であれば、アカウントの作成方法に関して[第20章](#)を参照してください。最初のユーザアカウントを作成するためには、PostgreSQLをインストールした時のオペレーティングシステムのユーザ（通常postgres）になる必要があります。PostgreSQLユーザアカウントがオペレーティングシステムのユーザ名と異なる名前で用意されているかもしれません。その場合は、PostgreSQLのユーザ名を指定するために、-Uスイッチを使用するか、PGUSER環境変数を設定する必要があります。

ユーザアカウントを持っているが、データベースを作成するために必要な権限を持っていない場合は、以下のメッセージが現れます。

```
createdb: database creation failed: ERROR:  permission denied to create database
```

全てのユーザがデータベースを新規に作成できる権限を持っているわけではありません。PostgreSQLがデータベースの作成を拒否した場合、サイト管理者はデータベースを作成する権限をユーザに付与する必要があります。これが発生した場合はサイト管理者に相談してください。自身でPostgreSQLをインストールしたのであれば、このチュートリアルのも目的のために、サーバを起動したユーザアカウントでログインしなければなりません。[1]

他の名前のデータベースを作成することもできます。PostgreSQLは、与えられたサイトでいくつでもデータベースを作成することを許可しています。データベース名は、先頭がアルファベット文字から始まる、63バイトまでの長さでなければなりません。簡単な選択は、現在のユーザ名と同じ名前のデータベースを作成することです。多くのツールでは、データベース名のデフォルトとしてそれを仮定していますので、これにより入力数を減らすことができます。このデータベースを作成するには、単純に以下を実行します。

```
$ createdb
```

データベースを使用しなくなったら、削除することができます。例えば、mydbデータベースの所有者（作成者）であれば、以下のコマンドでそれを廃棄することができます。

```
$ dropdb mydb
```

(このコマンドでは、データベース名のデフォルトはユーザアカウント名ではありません。常に指定しなければなりません。) この動作は、そのデータベースに関する全てのファイルを物理的に削除しますので、取り消すことはできません。事前に熟考した場合にのみこれを実施してください。

`createdb`や`dropdb`の詳細は、それぞれ[createdb](#)と[dropdb](#)にあります。

注意

[1]

何故これで上手くいくのかについての説明：PostgreSQLのユーザ名はオペレーティングシステムのユーザアカウントとは分離されています。データベースに接続する場合、接続に利用するPostgreSQLのユーザ名を選択します。選択しなければ、デフォルトで現在のオペレーティングシステムのアカウントと同じ名前となります。これにより、サーバを起動するオペレーティングシステムのユーザと同じ名前のPostgreSQLユーザアカウントは、常に存在します。そしてまた、このユーザは常にデータベースを作成する権限を持ちます。そのユーザとしてログインする代わりに、`-U`オプションを毎回使用して、接続するPostgreSQLのユーザ名を選択することができます。

[前のページ](#)

構造的な基本事項

[ホーム](#)

[上に戻る](#)

[次のページ](#)

データベースへのアクセス

1.4. データベースへのアクセス

データベースを作成した後、以下によってアクセスできます。

- `psql`というPostgreSQL対話式端末プログラムを実行。これにより、対話式にSQLコマンドの入力、編集、実行を行うことができます。
- データベースの作成や操作をサポートするpgAdminや、ODBCもしくはJDBCを備えたオフィススイートなどの既存のグラフィカルなフロントエンドツールを使用。これらについてはこのチュートリアルでは取り上げません。
- 複数の使用可能言語の1つを使用した、独自のアプリケーションの作成。これについては、[パートIV](#)で詳しく説明します。

このチュートリアル例を試すには、`psql`から始めることを勧めます。以下のコマンドを入力することによって、`mydb`データベースをアクティブにすることができます。

```
$ psql mydb
```

データベース名を与えなかった場合、データベース名はデフォルトでユーザアカウント名となります。この仕組みについては前節で`createdb`を使って既に説明しています。

`psql`では、始めに以下のメッセージが表示されます。

```
psql (9.2.4)
Type "help" for help.

mydb=>
```

最後の行は以下のようになっているかもしれません。

```
mydb=#
```

これは、データベーススーパーユーザであることを示します。自身でPostgreSQLをインストールした場合に最もよく現れます。スーパーユーザであることは、アクセス制御の支配を受けないことを意味します。チュートリアル段階では、これは重要ではありません。

`psql`の起動に問題が発生した場合は、前節に戻ってください。`createdb`の診断と`psql`の診断方法は似ており、前者が動作すれば後者も同様に動作するはずです。

`psql`が最後に出力する行はプロンプトで、`psql`が入力を監視していること、`psql`が管理する作業領域にSQL問い合わせを入力できることを示しています。以下のコマンドを試してください。

```
mydb=> SELECT version();
               version
-----
PostgreSQL 9.2.4 on i586-pc-linux-gnu, compiled by GCC 2.96, 32-bit
(1 row)

mydb=> SELECT current_date;
      date
-----
2002-08-31
(1 row)

mydb=> SELECT 2 + 2;
?column?
-----
4
(1 row)
```

`psql`プログラムは、SQLコマンドではない、多くの内部コマンドを持っています。それらはバックスラッシュ文字"`\`"から始まります。例えば、各種PostgreSQL SQLコマンドの構文に関するヘルプを以下のようにして得ることができます。

```
mydb=> \h
```

`psql`を終了するには、以下を入力します。

```
mydb=> \q
```

`psql`は終了し、コマンドシェルに戻ります（他の内部コマンドについては`psql`のプロンプトで"`\?`"を入力してください）。`psql`の完全な能力については[psql](#)で説明されています。このチュートリアルではこれらの機能は明示的に使用しませんが、便利

な場合これらを使用しても構いません。

[前のページ](#)
データベースの作成

[ホーム](#)
[上に戻る](#)

[次のページ](#)
SQL言語

第 2章SQL言語

目次

- 2.1. [序文](#)
- 2.2. [概念](#)
- 2.3. [新しいテーブルの作成](#)
- 2.4. [テーブルに行を挿入](#)
- 2.5. [テーブルへの問い合わせ](#)
- 2.6. [テーブル間を結合](#)
- 2.7. [集約関数](#)
- 2.8. [更新](#)
- 2.9. [削除](#)

2.1. 序文

本章では、SQLを使用した簡単な操作方法について、その概要を説明します。このチュートリアルは単なる入門用であり、SQLについての完全な教科書ではありません。[Understanding the New SQL](#)や[A Guide to the SQL Standard](#)など、SQLを説明した書籍は多くあります。PostgreSQL言語が持つ機能の中には標準を拡張したものがあることには注意してください。

以下で示す例では、前章で説明したmydbという名前のデータベースを作成し、psqlを起動できるようになっていることを前提としています。

このマニュアルで示す例は、PostgreSQLソース配布物に含まれており、src/tutorial/以下に展開されます。（PostgreSQLのバイナリ配布物ではこのファイルはコンパイルされていないかも知れません。）このファイルを使用するためには、以下に示すように、まずこのディレクトリに移動し、makeを実行してください。

```
$ cd .... /src/tutorial
$ make
```

これによりスクリプトが作成され、そして、ユーザ定義の関数と型を含むCのファイルがコンパイルされます。その後、以下を行うことで、チュートリアルを始めることができます。

```
$ cd .... /tutorial
$ psql -s mydb
...

mydb=> \i basics.sql
```

\i は、指定したファイルからコマンドを読み込みます。psqlの-sオプションによって、それぞれの文をサーバに送る前に一時停止する、シングルステップモードとなります。本節で使用するコマンドはbasics.sourceファイル内にあります。

2.2. 概念

PostgreSQL はリレーショナルデータベースマネジメントシステム (RDBMS) です。これはリレーションの中に格納されたデータを管理するシステムであることを意味しています。リレーションは基本的にはテーブルを表す数学用語です。テーブルにデータを格納することは今日では平凡なことです。わかりきったものだと思われるかもしれませんが、データベースを構成する方法には他にも多くの方式があります。Unix 互換のオペレーティングシステムのファイルとディレクトリは、階層型データベースの一種と言えます。より近代的な成果はオブジェクト指向データベースです。

各テーブルは、行の集合に名前を付けたものです。あるテーブルの各行は、名前を付けた列の集合ということができます。各列は特定のデータ型を持ちます。列は行において固定の順番を持ちますが、SQL はテーブルにある行の順番をまったく保証しないことを覚えておくことは重要です（しかし、表示用に明示的にソートさせることは可能です）。

テーブルはデータベースとしてまとめられ、1 つの PostgreSQL サーバインスタンスで管理されるデータベースの集合はデータベースクラスタを構成します。

2.3. 新しいテーブルの作成

テーブル名と、テーブルの全ての列の名前と型を指定することで、新しいテーブルを作成することができます。

```
CREATE TABLE weather (
    city          varchar(80),
    temp_lo       int,          -- 最低気温
    temp_hi       int,          -- 最高気温
    prcp          real,         -- 降水量
    date          date
);
```

上のコマンドを複数の行に分けて `psql` に入力することができます。 `psql` は、セミコロンで終わるまでそのコマンドは継続するものと認識します。

SQL コマンドでは空白文字（つまり空白、タブ、改行）を自由に使用することができます。つまり、上で示したコマンドとは異なる形で入力することができることを意味します。全てを 1 行で入力することさえできます。連続した 2 つのハイフン ("--") はコメントの始まりです。その後に入力したものは、行末まで無視されます。SQL はキーワードと識別子に対して大文字小文字を区別しません。ただし、（上では行っていませんが）識別子が二重引用符で括られていた場合は大文字小文字を区別します。

`varchar(80)` は、80 文字までの任意の文字列を格納できるデータ型を指定しています。 `int` は一般的な整数用の型です。 `real` は単精度浮動小数点数値を格納する型です。 `date`（日付）はその名前からわかるでしょう（わかると思いますが、`date` 型の列は `date` と名付けられています。これはわかりやすいかもしれませんが、逆に混乱を招くかもしれません。これは好みによります）。

PostgreSQL は標準 SQL のデータ型、`int`、`smallint`、`real`、`double precision`、`char(N)`、`varchar(N)`、`date`、`time`、`timestamp` や `interval` をサポートします。また、一般的なユーティリティ用の型や高度な幾何データ型もサポートします。任意の数のユーザ定義のデータ型を使用して、PostgreSQL をカスタマイズすることができます。したがって、標準 SQL における特殊な場合をサポートするために必要な場所を除き、型名は構文内でキーワードではありません。

以下に示す 2 番目の例では、都市とその地理的な位置情報を格納します。

```
CREATE TABLE cities (
    name          varchar(80),
    location       point
);
```

`point` 型は、PostgreSQL 独自のデータ型の一例です。

最後に、テーブルが不要になった場合や別のものに作り直したい場合、以下のコマンドを使用して削除できることを示します。

```
DROP TABLE tablename;
```

2.4. テーブルに行を挿入

以下のように、INSERT 文を使用して、テーブルに行を挿入します。

```
INSERT INTO weather VALUES ('San Francisco', 46, 50, 0.25, '1994-11-27');
```

全てのデータ型でどちらかといえばわかりやすい入力書式を使用していることに注意してください。通常、単純な数値以外の定数は、この例のように、単一引用符 (') で括らなければなりません。date 型で受け付けられるものは実際はかなり柔軟です。しかし、このチュートリアル段階では、曖昧さが無い書式にこだわることにします。

point 型では、入力として次のような座標の組み合わせが必要です。

```
INSERT INTO cities VALUES ('San Francisco', '(-194.0, 53.0)');
```

ここまでの構文では、列の順番を覚えておく必要がありました。以下に示す他の構文では、列のリストを明示的に与えることができます。

```
INSERT INTO weather (city, temp_lo, temp_hi, prcp, date)
VALUES ('San Francisco', 43, 57, 0.0, '1994-11-29');
```

リスト内の列は好きな順番で指定できます。また、一部の列を省略することもできます。例えば、降水量がわからない場合は以下のようにすることができます。

```
INSERT INTO weather (date, city, temp_hi, temp_lo)
VALUES ('1994-11-29', 'Hayward', 54, 37);
```

多くの開発者は、暗黙的な順番に依存するよりも、列のリストを明示的に指定する方を好んで使用します。

次節でもデータを使用しますので、上のコマンドを全て入力してください。

また、COPY を使用して大量のデータを平文テキストファイルからロードすることもできます。COPY コマンドは INSERT よりも柔軟性はありませんが、この目的に特化していますので、通常、より高速になります。以下に例を示します。

```
COPY weather FROM '/home/user/weather.txt';
```

ここで元となるファイルを表すファイル名は、クライアントではなく、バックエンドプロセスを動かしているマシンで利用できるものでなければなりません。バックエンドプロセスがこのファイルを直接読み込むからです。[COPY](#) には COPY コマンドについてのより詳しい説明があります。

2.5. テーブルへの問い合わせ

テーブルからデータを取り出すために、テーブルへ問い合わせをします。このためにSQLのSELECT文が使用されます。この文は選択リスト（返される列のリスト部分）とテーブルリスト（データを取り出すテーブルのリスト部分）、および、省略可能な条件（制限を指定する部分）に分けることができます。例えば、weatherの全ての行を取り出すには、以下を入力します。

```
SELECT * FROM weather;
```

ここで*は"全ての列"の省略形です。[1] したがって、以下のようにしても同じ結果になります。

```
SELECT city, temp_lo, temp_hi, prcp, date FROM weather;
```

出力は、以下のようになります。

city	temp_lo	temp_hi	prcp	date
San Francisco	46	50	0.25	1994-11-27
San Francisco	43	57	0	1994-11-29
Hayward	37	54		1994-11-29

(3 rows)

選択リストには、単なる列参照だけではなく任意の式を指定することができます。例えば、以下を行うことができます。

```
SELECT city, (temp_hi+temp_lo)/2 AS temp_avg, date FROM weather;
```

この結果は次のようになります。

city	temp_avg	date
San Francisco	48	1994-11-27
San Francisco	50	1994-11-29
Hayward	45	1994-11-29

(3 rows)

AS句を使用した出力列の再ラベル付けの部分に注意してください（AS句は省略することができます）。

必要な行が何かを指定するWHERE句を追加して問い合わせに"条件付け"することができます。WHERE句は論理（真値）式を持ち、この論理式が真となる行のみを返します。よく使われる論理演算子（AND、OR、NOT）を条件付けに使用することができます。例えば以下は、San Franciscoの雨天時の気象データを取り出します。

```
SELECT * FROM weather
WHERE city = 'San Francisco' AND prcp > 0.0;
```

結果は次のようになります。

city	temp_lo	temp_hi	prcp	date
San Francisco	46	50	0.25	1994-11-27

(1 row)

問い合わせの結果をソートして返すように指定することができます。

```
SELECT * FROM weather
ORDER BY city;
```

city	temp_lo	temp_hi	prcp	date
Hayward	37	54		1994-11-29
San Francisco	43	57	0	1994-11-29
San Francisco	46	50	0.25	1994-11-27

この例では、ソート順は十分に指定されていません。ですので、San Franciscoの行は順序が異なるかも知れません。しかし、次のようにすれば常に上記の結果になります。

```
SELECT * FROM weather
ORDER BY city, temp_lo;
```

問い合わせの結果から重複行を除くように指定することができます。

```
SELECT DISTINCT city
FROM weather;
```

city
Hayward

```
San Francisco
(2 rows)
```

繰り返しますが、結果行の順序は変動するかもしれません。 `DISTINCT`と`ORDER BY`を一緒に使用することで確実に一貫した結果を得ることができます。 [\[2\]](#)

```
SELECT DISTINCT city
FROM weather
ORDER BY city;
```

もちろん、`DISTINCT`および`ORDER BY`は単体でも使用することができます。

注意

[\[1\]](#)

`SELECT *`は即興的な問い合わせで有用ですが、製品レベルのコードでは、テーブルに列を追加することにより結果が異なってしまいますので、通常好まれません。

[\[2\]](#)

PostgreSQLの古めのバージョンを含む一部のデータベースシステムでは、`DISTINCT`の実装に行の自動順序付けが含まれており、`ORDER BY`は不要です。しかし、これは標準SQLにおける要求ではなく、現在のPostgreSQLでは`DISTINCT`句が行の順序付けを行うことを保証していません。

[前のページ](#)

テーブルに行を挿入

[ホーム](#)

[上に戻る](#)

[次のページ](#)

テーブル間を結合

2.6. テーブル間を結合

ここまでの問い合わせは、一度に1つのテーブルにのみアクセスするものでした。問い合わせは、一度に複数のテーブルにアクセスすることも、同時にテーブル内の複数の行の処理を行う場合など、同じテーブルにアクセスすることも可能です。一度に同一のテーブルまたは複数のテーブルの複数の行にアクセスする問い合わせは、結合問い合わせと呼ばれます。例として、関連する都市の位置情報を気象データと一緒に表示したい場合が挙げられます。それを行うためには、`weather` テーブルの各行の `city` 列と、`cities` テーブルの全ての行の `name` 列を比較し、両者の値が一致する行の組み合わせを選択しなければなりません。

注意: これは概念的なモデルでしかありません。実際の結合は通常、1つひとつの行の組み合わせを比べるのではなく、もっと効率的な方法で行われます。しかし、これはユーザからはわかりません。

これは、以下の問い合わせによって行うことができます。

```
SELECT *
FROM weather, cities
WHERE city = name;
```

city	temp_lo	temp_hi	prcp	date	name	location
San Francisco	46	50	0.25	1994-11-27	San Francisco	(-194,53)
San Francisco	43	57	0	1994-11-29	San Francisco	(-194,53)

(2 rows)

この結果から2つのことがわかります。

- Hayward 市についての結果行はありません。 `cities` テーブルには Hayward に一致する項目がなく、結合の際に `weather` テーブル内の一致しない行は無視されるからです。これがどのようになされるのかを、簡単に見てみましょう。
- 都市名を持つ2つの列があります。 `weather` テーブルと `cities` テーブルからの列のリストが連結されているためこのようになります。しかし実際には、これは望ましい結果ではないため、*を使わずに、明示的に出力列のリストを指定することになります。

```
SELECT city, temp_lo, temp_hi, prcp, date, location
FROM weather, cities
WHERE city = name;
```

練習: `WHERE` 句を省略した場合のこの問い合わせの意味を決定してください。

列はそれぞれ異なる名前ですので、パーサは自動的にどのテーブルの列かを見つけます。2つのテーブルで列名が重複していた場合は、以下のようにどちらの列を表示させたいかを示すために列名を修飾しなければなりません。

```
SELECT weather.city, weather.temp_lo, weather.temp_hi,
       weather.prcp, weather.date, cities.location
FROM weather, cities
WHERE cities.name = weather.city;
```

結合問い合わせではすべての列名を修飾する方式が優れているとよく考えられています。テーブルのいずれかに後で重複する名前を持つ列が追加された場合に、問い合わせが失敗するからです。

ここで示すような結合問い合わせは、以下のように別の形で表すことができます。

```
SELECT *
FROM weather INNER JOIN cities ON (weather.city = cities.name);
```

この構文は先の例よりも一般的に使用されるものではありませんが、以降の話題の理解を助けるためにここで示しています。

ここで、どのようにすれば Hayward のレコードを得ることができるようになるのでしょうか。実行したい問い合わせは、`weather` をスキャンし、各行に対して、`cities` 行に一致するかを判断するものです。一致しない行があった場合、`cities` テーブルの列の部分は何らかの"空の値"に置き換えたいのです。この種の問い合わせは外部結合と呼ばれます（これまで示してきた結合は内部結合です）。以下のようなコマンドになります。

```
SELECT *
FROM weather LEFT OUTER JOIN cities ON (weather.city = cities.name);
```

city	temp_lo	temp_hi	prcp	date	name	location
Hayward	37	54		1994-11-29		
San Francisco	46	50	0.25	1994-11-27	San Francisco	(-194,53)
San Francisco	43	57	0	1994-11-29	San Francisco	(-194,53)

(3 rows)

この問い合わせは左外部結合と呼ばれます。結合演算子の左側に指定したテーブルの各行が最低でも一度出力され、一方で、右側のテーブルでは左側のテーブルの行に一致するもののみが出力されるからです。右側のテーブルに一致しない、左側のテーブルの行を出力する時、右側のテーブルの列は空の値（NULL）で置換されます。

練習：右外部結合や完全外部結合も存在します。これらが何を行うかを考えてください。

テーブルを自分自身に対して結合させることができます。これは自己結合と呼ばれます。例として、他の気象データの気温範囲内にある気象データを全て取り出すことを考えます。weather各行のtemp_loとtemp_hiを、他のweather行のtemp_loとtemp_hi列とを比較しなければなりません。以下の問い合わせを使用して行うことができます。

```
SELECT W1.city, W1.temp_lo AS low, W1.temp_hi AS high,
       W2.city, W2.temp_lo AS low, W2.temp_hi AS high
FROM weather W1, weather W2
WHERE W1.temp_lo < W2.temp_lo
AND W1.temp_hi > W2.temp_hi;
```

city	low	high	city	low	high
San Francisco	43	57	San Francisco	46	50
Hayward	37	54	San Francisco	46	50

(2 rows)

ここで、結合の左側と右側を区別することができるように、weatherテーブルにw1とw2というラベルを付けています。また、入力量を省くために、他の問い合わせでもこの種の別名を使用することができます。以下に例を示します。

```
SELECT *
FROM weather w, cities c
WHERE w.city = c.name;
```

こういった形の省略はかなりよく行われます。

2.7. 集約関数

他のほとんどのリレーショナルデータベース製品同様、PostgreSQLは集約関数をサポートします。集約関数は複数の入力行から1つの結果を計算します。例えば、行の集合に対して、count（総数）、sum（総和）、avg（平均）、max（最大）、min（最小）といった演算を行う集約があります。

例として、次のように全ての都市の最低気温から最も高い気温を求めることができます。

```
SELECT max(temp_lo) FROM weather;

max
-----
 46
(1 row)
```

どの都市のデータなのかを知りたいとしたら、下記のような問い合わせを試行するかもしれません。

```
SELECT city FROM weather WHERE temp_lo = max(temp_lo);
```

間違い

しかし、max集約をWHEREで使うことができませんので、このコマンドは動作しません（WHERE句はどの行を集約処理に渡すのかを決定するものであり、したがって、集約関数の演算を行う前に評価されなければならないことは明らかなです。このためにこの制限があります）。しかしたいいていの場合、問い合わせを書き直すことで、意図した結果が得られます。これには以下のような副問い合わせを使用します。

```
SELECT city FROM weather
       WHERE temp_lo = (SELECT max(temp_lo) FROM weather);

city
-----
San Francisco
(1 row)
```

副問い合わせは、外側の問い合わせで起こることとは別々に集約を計算する独立した演算ですので、この問い合わせは問題ありません。

また、GROUP BY句と組み合わせた集約は非常に役に立ちます。例えば、以下のコマンドで都市ごとに最低気温の最大値を求めることができます。

```
SELECT city, max(temp_lo)
FROM weather
GROUP BY city;

city      | max
-----+-----
Hayward   | 37
San Francisco | 46
(2 rows)
```

ここには都市ごとに1行の出力があります。それぞれの集約結果は都市に一致するテーブル行全体に対する演算結果です。以下のように、HAVINGを使用すると、グループ化された行にフィルタをかけることができます。

```
SELECT city, max(temp_lo)
FROM weather
GROUP BY city
HAVING max(temp_lo) < 40;

city      | max
-----+-----
Hayward   | 37
(1 row)
```

このコマンドは上と同じ計算を行うものですが、全てのtemp_loの値が40未満の都市のみを出力します。最後になりますが、"s"から始まる名前の都市のみを対象にしたい場合は、以下を行います。

```
SELECT city, max(temp_lo)
FROM weather
WHERE city LIKE 's%' (1)
GROUP BY city
HAVING max(temp_lo) < 40;
```

(1) LIKE演算子はパターン一致を行います。これについては[項9.7](#)で説明します。

集約とSQLのWHEREとHAVING句の間の相互作用を理解することが重要です。WHEREとHAVINGの基本的な違いを以下に記します。

WHERE は、グループや集約を演算する前に入力行を選択します（したがって、これはどの行を使用して集約演算を行うかを制御します）。一方、HAVING は、グループと集約を演算した後に、グループ化された行を選択します。したがって、WHERE 句は集約関数を持つことはできません。集約を使用して、どの行をその集約の入力にするのかを決定することは意味をなしません。一方で、HAVING 句は常に集約関数を持ちます（厳密に言うと、集約を使用しないHAVING 句を書くことはできますが、これは有用となることはほぼありません。同じ条件はWHERE の段階でもっと効率良く使用できます）。

前の例ではWHERE 内に都市名制限を適用することができます。集約を行う必要がないからです。WHERE の検査で失敗する全ての行に対するグループ化や集約演算が行われませんので、HAVING に制限を追加するよりも効率的です。

2.8. 更新

UPDATE コマンドを使用して、既存の行を更新することができます。 11月28日以降の全ての気温の読み取りが2度高くなっていることがわかったとします。 その場合、以下のコマンドによって、データを修正することができます。

```
UPDATE weather
  SET temp_hi = temp_hi - 2, temp_lo = temp_lo - 2
 WHERE date > '1994-11-28';
```

データの更新後の状態を確認します。

```
SELECT * FROM weather;
```

city	temp_lo	temp_hi	prcp	date
San Francisco	46	50	0.25	1994-11-27
San Francisco	41	55	0	1994-11-29
Hayward	35	52		1994-11-29

(3 rows)

2.9. 削除

DELETE コマンドを使用してテーブルから行を削除することができます。Hayward の気象を対象としなくなったとします。その場合、以下のコマンドを使用して、テーブルから行を削除することができます。

```
DELETE FROM weather WHERE city = 'Hayward';
```

Hayward に関する気象データは全て削除されました。

```
SELECT * FROM weather;
```

city	temp_lo	temp_hi	prcp	date
San Francisco	46	50	0.25	1994-11-27
San Francisco	41	55	0	1994-11-29

(2 rows)

以下の形式の文には注意しなければなりません。

```
DELETE FROM tablename;
```

条件がない場合、DELETE は指定したテーブルの全ての行を削除し、テーブルを空にします。システムは削除前に確認を求めるようなことは行いません！

第 3章 高度な諸機能

目次

- 3.1. [はじめに](#)
- 3.2. [ビュー](#)
- 3.3. [外部キー](#)
- 3.4. [トランザクション](#)
- 3.5. [ウィンドウ関数](#)
- 3.6. [継承](#)
- 3.7. [まとめ](#)

3.1. はじめに

前章では、PostgreSQLでSQLを使用してデータを保存したりアクセスしたりする基本について説明しました。ここでは、管理を単純化しデータの喪失や破壊を防止するSQLのいくつかのより高度な機能について論議します。最後にPostgreSQLが拡張しているいくつかの機能に触れます。

本章では折々[第2章](#)にある例題に変更や改善を加えますので、その章を読んでおくことは役立ちます。本章にあるいくつかの例題は、配付物を解凍した時にできるsrc/tutorialディレクトリのadvanced.sqlに入っています。ここでは繰り返しません、このファイルにはロードして使ってみることができるサンプルデータもあります（ファイルの使い方は[項2.1](#)を参照してください）。

3.2. ビュー

[項2.6](#)の問い合わせをもう一度参照してください。天候の記録と都市の所在場所の組み合わせ一覧を得ることが今作っているアプリケーションにとって特に重要なのですが、この組み合わせの一覧を必要とする度に問い合わせを打ち込みたくはないとしましょう。結果を求める問い合わせに対してビューを作成し、名前を付けると、通常のテーブル参照のように結果を入手できるようになります。

```
CREATE VIEW myview AS
  SELECT city, temp_lo, temp_hi, prcp, date, location
     FROM weather, cities
     WHERE city = name;

SELECT * FROM myview;
```

SQLデータベースを設計する上で押さえておかなければならないのは、ビューを自由に使えるようにするという項目です。ビューはテーブル構造の詳細をカプセル化しますので、アプリケーションによる更新でテーブル構造が変わったとしても一貫したインタフェースを保てます。

ビューは実テーブルが使用できるほとんどの場合で使えます。他のビューに対するビューの作成も珍しくはありません。

3.3. 外部キー

[第2章](#)のweatherテーブルとcitiesテーブルを思い出してください。次のような問題点を考えてみましょう。citiesテーブルに一致する項目がない行は絶対にweatherテーブルに挿入できなくしたいとします。これをデータの参照整合性の保全と呼びます。最も単純なデータベースシステムではcitiesテーブルに一致する行が存在するかどうかを最初に検証してからweatherテーブルに新規レコードを追加するか否かを実装しなければなりません。この手法には多くの問題があること、そしてとても不便であることから、PostgreSQLに代わって作業させることができます。

新規のテーブルの宣言は以下のようになります。

```
CREATE TABLE cities (  
    city      varchar(80) primary key,  
    location  point  
);  
  
CREATE TABLE weather (  
    city      varchar(80) references cities(city),  
    temp_lo   int,  
    temp_hi   int,  
    prcp      real,  
    date      date  
);
```

では無効なレコードを挿入してみましょう。

```
INSERT INTO weather VALUES ('Berkeley', 45, 53, 0.0, '1994-11-28');  
  
ERROR:  insert or update on table "weather" violates foreign key constraint "weather_city_fkey"  
DETAIL:  Key (city)=(Berkeley) is not present in table "cities".
```

外部キーによる振舞いでアプリケーションが見事にチューニングされました。このチュートリアルではこの簡単な例題より先には進みませんが、さらに情報がほしい方は[第5章](#)をご覧ください。外部キーを正しく使用するようにすると、間違いなくデータベースアプリケーションの質を向上させますので身に付くように励んでください。

3.4. トランザクション

トランザクションは全てのデータベースシステムで基礎となる概念です。トランザクションの基本的要点は複数の手順を単一の「全てかなし」の操作にまとめ上げることです。手順の進行途中の状態は他の動いているトランザクションからは見えません。そして、何らかのエラーが起こればトランザクションは完結しません。ですからデータベースはエラーの原因となった手順によってまったく影響されることはありません。

例を挙げましょう。ある銀行のデータベースでそこに多数の顧客の口座の残高と支店の総預金残高が記録されているとします。アリスの口座からボブの口座に\$100.00の送金があったことを記録したいとします。ちょっと乱暴に単純化すると、このSQLは次のようになります。

```
UPDATE accounts SET balance = balance - 100.00
WHERE name = 'Alice';
UPDATE branches SET balance = balance - 100.00
WHERE name = (SELECT branch_name FROM accounts WHERE name = 'Alice');
UPDATE accounts SET balance = balance + 100.00
WHERE name = 'Bob';
UPDATE branches SET balance = balance + 100.00
WHERE name = (SELECT branch_name FROM accounts WHERE name = 'Bob');
```

書かれているSQLコマンドの詳しいことについて、ここでは重要ではありません。重要な点は、この単純な操作の目的を果たすため、複数の独立した更新手続きが関わっていることです。銀行職員としてはこれら全ての更新が行われるかもしくはまったく行われないのかいずれかの確証が必要です。\$100.00がアリスの口座から引き落とされずにボブの口座に振り込まれるようなシステムの不備があってはなりません。一方、\$100.00がボブに振り込まれないでアリスの口座から引き落とされたとしたら、アリスはこの銀行のお得意様ではなくなるでしょうね。操作の途中で一部不都合が発生した場合、結果に影響を与えるいかなる手続きも実行されないという確証が必要です。更新手続きをトランザクションにグループ化すると、その確証が得られます。あるトランザクションは他のトランザクションから見て完結するかまったく起こらなかったかという見方から原子的と呼ばれます。

もう一方で、いったんトランザクションが完結しデータベースシステムに承認された場合は、確実に恒久的に保存され、たとえ直後にクラッシュが起こったとしても記録は失われないという確証も必要です。例えばボブが自分の口座から現金を引き落とし店舗から立ち去った直後にボブの口座からの引き落とし記録がシステムのクラッシュで消えてしまうことは受け入れられません。トランザクションが実装されているデータベースでは、あるトランザクションによる全ての更新がそのトランザクションを完結したと通知を行う前に永続的記録装置（すなわちディスク上）にログを書き込むことで保証しています。

他にもトランザクション実装のデータベースの重要な特性は、原子的更新という概念に深く関係しています。複数のトランザクションが同時に動作している時、それぞれのトランザクションは別のトランザクションが行っている未完了の変更を見ることができてはなりません。例えば、あるトランザクションがすべての支店の残高を集計する作業を行なう場合、アリスの口座がある支店からの引き落としを勘定に入れませんし、ボブの口座がある支店への振り込みも勘定に入れません。もしくは両方とも勘定に入れます。つまり、データベース上での恒久的効果という意味のみならず、一連の操作の過程で可視性ということにおいてもトランザクションは「すべて」か「なし」かでなければなりません。作業中のトランザクションによる更新は、他のトランザクションからはトランザクションが完結するまで不可視です。そのトランザクションが完結したその時点で、トランザクションが行った更新の全てが見えるようになります。

PostgreSQLではトランザクションを構成するSQLコマンドをBEGINとCOMMITで囲んで設定します。そうすると、この銀行取引引きのトランザクションの実際は次のようになります。

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
WHERE name = 'Alice';
-- etc etc
COMMIT;
```

トランザクション処理の途中でコミットを行わない（アリスの口座残高が足りなかったような場合）と判断した場合は、COMMITではなくROLLBACKを使用して行った全ての更新を破棄します。

PostgreSQLは実際全てのSQL文をトランザクション内で実行するようになっています。BEGINを発行しないでも、それぞれの文は暗黙的にBEGINが付いているとみなし、（成功すれば）COMMITで囲まれているものとします。BEGINとCOMMITで囲まれた文のグループはトランザクションブロックと呼ばれることもあります。

注意: いくつかのクライアントライブラリは自動的にBEGINとCOMMITコマンドを発行し、警告なしにトランザクションブロックが有効になります。使用しているインタフェースのドキュメントで確認してください。

セーブポイントを使用することで、トランザクション内で文を、より粒度を細かく制御することが可能になります。セーブポイントは、トランザクションを構成するある部分を選択的に破棄する一方、破棄されない残りの部分をコミットします。SAVEPOINTコマンドでセーブポイントを定義した後、必要であればROLLBACK TOコマンドによりセーブポイントまでロールバックできます。定義されたセーブポイントとロールバックするポイントとの間の全てのトランザクションのデータベースの変更

更には破棄されますが、セーブポイント以前の変更は保持されます。

セーブポイントまでロールバックした後でもセーブポイントは定義されたままです。このため何度でもそこにロールバックできます。逆に再度特定のセーブポイントにロールバックする必要がないのであれば、それを解除しシステムリソースを多少とも解放することができます。あるセーブポイントを解除したりセーブポイントにロールバックすることにより、自動的にその後定義されたすべてのセーブポイントが解除されることに注意してください。

これら全てはトランザクションブロック内で起こるので、他のデータベースセッションからは何も見えません。トランザクションブロックをコミットした場合、他のセッションからはコミットされた行為が1つの単位として見えるようになりますが、ロールバックの行為は決して可視になりません。

銀行のデータベースを思い出してください。アリスの口座から\$100.00を引き出してボブの口座に振り込むとします。後になってボブではなくウィリーの口座に振り込むべきだったと気が付きました。この場合セーブポイントを次のように使います。

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
  WHERE name = 'Alice';
SAVEPOINT my_savepoint;
UPDATE accounts SET balance = balance + 100.00
  WHERE name = 'Bob';
-- おっと、忘れるところだった。ウィリーの口座を使わなければ。
ROLLBACK TO my_savepoint;
UPDATE accounts SET balance = balance + 100.00
  WHERE name = 'Wally';
COMMIT;
```

この例はもちろん極端に単純化していますが、セーブポイントの使用を通じてトランザクションブロック内で多くの可能な操作を行えることがわかります。さらには何らかのエラーでシステムがトランザクションブロックを中断した場合、完全にロールバックして再び開始するのを除いて、`ROLLBACK TO` コマンドがトランザクションブロックの制御を取り戻す唯一の手段です。

[前のページ](#)
外部キー

[ホーム](#)
[上に戻る](#)

[次のページ](#)
ウィンドウ関数

3.5. ウィンドウ関数

ウィンドウ関数は現在の行に何らかとも関係するテーブル行の一纏まり全般の計算を行います。これは集約関数により行われる計算の形式と似たようなものです。とは言っても、通常の集約関数とは異なり、ウィンドウ関数の使用は単一出力行に行をグループ化しません。行はそれぞれ個別の身元を維持します。裏側では、ウィンドウ関数は問い合わせ結果による現在行だけでなく、それ以上の行にアクセスすることができます。

これはその部署の平均給与とそれぞれの従業員の給与をどのように比較するかを示した例です。

```
SELECT depname, empno, salary, avg(salary) OVER (PARTITION BY depname) FROM empsalary;
```

depname	empno	salary	avg
develop	11	5200	5020.0000000000000000
develop	7	4200	5020.0000000000000000
develop	9	4500	5020.0000000000000000
develop	8	6000	5020.0000000000000000
develop	10	5200	5020.0000000000000000
personnel	5	3500	3700.0000000000000000
personnel	2	3900	3700.0000000000000000
sales	3	4800	4866.6666666666666667
sales	1	5000	4866.6666666666666667
sales	4	4800	4866.6666666666666667

(10 rows)

最初の3つの出力列は、テーブルempsalaryから直接もたらされ、テーブル内のそれぞれの行に対し1つの出力行が存在します。4番目の列は、現行の行と同じdepnameの値を持つ全てのテーブル行に渡って取得した平均値を表わしています。（これは実際、通常のavg集約関数と同じですが、OVER句によりウィンドウ関数として扱われ、行の適切な集合に渡り計算されます。）

ウィンドウ関数呼び出しは常に、ウィンドウ関数名と引数に直接伴われたOVER句を含みます。これが通常関数、または集約関数と構文的に区別されるところです。OVER句は、ウィンドウ関数により処理のため問い合わせの行がどのように分解されるかを厳密に決定します。OVER内のPARTITION BYリストは、行をPARTITION BY式の同じ値を共有するグループまたはパーティションに分割する指定を行います。それぞれの行に対し、ウィンドウ関数は現在行と同じパーティションに分類される行に渡って計算されます。

OVER内でORDER BYを使用することによりウィンドウ関数で処理される行の順序を制御することもできます。（ウィンドウのORDER BYは行が出力される順序に一致する必要すらありません。）ここに例をあげます。

```
SELECT depname, empno, salary, rank() OVER (PARTITION BY depname ORDER BY salary DESC) FROM empsalary;
```

depname	empno	salary	rank
develop	8	6000	1
develop	10	5200	2
develop	11	5200	2
develop	9	4500	4
develop	7	4200	5
personnel	2	3900	1
personnel	5	3500	2
sales	1	5000	1
sales	4	4800	2
sales	3	4800	2

(10 rows)

ここで示されたように、rank関数は、それぞれの明確なORDER BYの値に対する現在行のパーティション内における順位を、ORDER BY句で定義された順序で生成します。rankは明示的なパラメータを必要としません。この動作はOVER句により完全に決定されるためです。

ウィンドウ関数で考慮される行は、もし存在するのであればそのWHERE、GROUP BY、およびHAVING句でフィルターをかけられた問い合わせのFROM句によって生成された"仮想テーブル"です。例えば、WHERE条件に一致しないため削除された行はウィンドウ関数では見つけることができません。異なったOVER句を用いて、異なった方法によりデータを分割する複数のウィンドウ関数を問い合わせが含んでも構いません。しかし、この仮想テーブルで定義された行の同一の集まり上で全てが作動します。

ORDER BYは、行の順序付けが重要でない場合割愛可能であることを見てきました。PARTITION BYも同様に割愛することができます。この場合、全ての行を含むたった一つのパーティションが存在します。

ウィンドウ関数に関連したその他の重要な概念があります。それぞれの行に対して、そのウィンドウフレームと呼ばれる、そのパーティション内の行の集合が存在します。多くの（しかしすべてではありません）ウィンドウ関数は、パーティション全体ではなく、ウィンドウフレームの行のみに対して作用します。デフォルトでは、ORDER BYが指定されると、フレームは、パーティションの始めから現在の行までのすべての行と、その後にあるORDER BY句に従って現在の行と等しい行から構成されます。

ORDER BYが省略された場合、デフォルトのフレームはそのパーティション内のすべての行を含みます。[1] sumを使用した例を

示します。

```
SELECT salary, sum(salary) OVER () FROM empsalary;
```

salary	sum
5200	47100
5000	47100
3500	47100
4800	47100
3900	47100
4200	47100
4500	47100
4800	47100
6000	47100
5200	47100

(10 rows)

上では、OVER 句内にORDER BYが存在しませんので、ウィンドウフレームはパーティションと同一です。またパーティションはPARTITION BYもありませんのでテーブル全体となります。言い換えると、総和はそれぞれ、テーブル全体に対して行われ、その結果、各出力行で同じ結果を得ることになります。しかし以下のように、ORDER BY句を加えると、非常に異なる結果を得ます。

```
SELECT salary, sum(salary) OVER (ORDER BY salary) FROM empsalary;
```

salary	sum
3500	3500
3900	7400
4200	11600
4500	16100
4800	25700
4800	25700
5000	30700
5200	41100
5200	41100
6000	47100

(10 rows)

ここで、sumは最初の（最も低い）salaryから現在の行まで、現在のものと重複する全てを含んで、計算されます（重複するsalaryに対する結果に注意してください）。

ウィンドウ関数は問い合わせのSELECTリストとORDER BY句に限って許可されます。GROUP BY、HAVING、およびWHEREのような句の中では禁止されています。その理由は、ウィンドウ関数は論理的に、ここに挙げたような句が処理された後に実行されるからです。またウィンドウ関数は通常の集約関数の後に実行されます。これが意味する所は、ウィンドウ関数の引数に集約関数呼び出しを含めても有効ですが、その逆は成り立たないということです。

ウィンドウ演算が行われた後、行にフィルタ処理を行ったりグループ化を行う必要が生じた場合、副問い合わせを使用します。例をあげます。

```
SELECT depname, empno, salary, enroll_date
FROM
  (SELECT depname, empno, salary, enroll_date,
    rank() OVER (PARTITION BY depname ORDER BY salary DESC, empno) AS pos
    FROM empsalary
  ) AS ss
WHERE pos < 3;
```

上記問い合わせは3より小さいrankを持った内部問い合わせからの行のみを表示します。

問い合わせが複数のウィンドウ関数を含む場合、各ウィンドウ関数に異なるOVER句を記述することができます。しかし複数の関数で同じウィンドウ処理動作が必要な場合は重複となり、またエラーを招きがちです。代わりにWINDOW句でウィンドウ処理動作に名前を付け、これをOVER内で参照することができます。以下に例を示します。

```
SELECT sum(salary) OVER w, avg(salary) OVER w
FROM empsalary
WINDOW w AS (PARTITION BY depname ORDER BY salary DESC);
```

ウィンドウ関数についてより詳細は、[項4.2.8](#)、[項9.21](#)、[項7.2.4](#)、および [SELECT](#) マニュアルページにあります。

注意

[1] ほかの方法でウィンドウフレームを定義するいくつかのオプションがありますが、このチュートリアルでは扱いません。詳細は、[項4.2.8](#)を参照してください。

3.6. 継承

継承とはオブジェクト指向データベースの概念です。 データベース設計においてこれまでになかった興味深い可能性を広げてくれます。

2つのテーブルを作ってみましょう。cities (都市) テーブルとcapitals (行政府所在地) テーブルです。行政府所在地は本来同時に都市でもありますので、全ての都市をリストする時は何もしなくても行政府所在地も表示する何らかの方法が必要です。賢い人なら次のような案を工夫するでしょう。

```
CREATE TABLE capitals (
    name      text,
    population real,
    altitude  int,    -- (フィート単位)
    state     char(2)
);

CREATE TABLE non_capitals (
    name      text,
    population real,
    altitude  int    -- (フィート単位)
);

CREATE VIEW cities AS
    SELECT name, population, altitude FROM capitals
    UNION
    SELECT name, population, altitude FROM non_capitals;
```

問い合わせを続ける分には問題はありませんが、たった1つ、いくつか複数の行を更新する時に醜くなります。

より良い解決策は次のような構文です。

```
CREATE TABLE cities (
    name      text,
    population real,
    altitude  int    -- (フィート単位)
);

CREATE TABLE capitals (
    state     char(2)
) INHERITS (cities);
```

この例では、capitals (行政府所在地) テーブルの行は親のcities (都市) テーブルから全ての列、すなわちname (都市名)、population (人口) そしてaltitude (標高) を継承します。 name列のデータ型は、可変長文字列のためにPostgreSQLが初めから備えているtext型です。 州の行政府所在地のテーブルは、これに加えて州を示すstate列を持ちます。 PostgreSQLでは、テーブルは関連付けられたテーブルがあればそれぞれから属性を継承することができます。

以下の問い合わせの例は、行政府所在地を含む標高500フィート以上に位置する全ての都市を求めるものです。

```
SELECT name, altitude
FROM cities
WHERE altitude > 500;
```

これは以下を返します。

name	altitude
Las Vegas	2174
Mariposa	1953
Madison	845

(3 rows)

その一方、行政府所在地ではない標高500フィート以上に位置する都市を見つけ出したい時は次のような問い合わせになります。

```
SELECT name, altitude
FROM ONLY cities
WHERE altitude > 500;
```

name	altitude
Las Vegas	2174
Mariposa	1953

(2 rows)

ここでcities (都市) の前に置かれたONLYは、継承階層においてcities (都市) テーブルの下層にあるテーブルではな

く、cities（都市）テーブルのみを参照することを意味します。既に説明したSELECT、UPDATEおよびDELETEなど数多くのコマンドは、このONLY表記をサポートしています。

注意: 継承は便利ではありますが、一意性制約もしくは外部キーと一緒に使えないので万能ではありません。詳細は[項5.8](#)を参照してください。

[前のページ](#)
ウィンドウ関数

[ホーム](#)
[上に戻る](#)

[次のページ](#)
まとめ

3.7. まとめ

SQL 初心者向けのこのチュートリアル入門では触れていない多くの機能が、PostgreSQL にはあります。これらの機能は、本書の残りで詳しく説明します。

もっと多くの入門資料がお望みであれば、PostgreSQL の [Web サイト](#) により多くのリソースがリンクされています。

II. SQL言語

ここでは、PostgreSQLでSQL言語を使用する方法を説明します。まず最初にSQL構文全般について述べ、データを保持する構造の作成方法、データベースに登録する方法、そして、データベースへの問い合わせを行う方法について説明していきます。本パートの中盤では、SQLコマンドで使用可能なデータ型と関数を紹介します。そして残りの部分では、最適な性能を実現するためにデータベースを調整する際に重要となるいくつかの点について説明します。

ここでの内容は、初心者ユーザでも他の入門書を参照することなく、最初から最後まで全てのトピックを理解できるよう構成になっています。各章ごとに内容が独立していますので、上級ユーザは必要な章だけを選んで読むことができます。ここではトピックに関する説明が中心となっていますので、特定のコマンドの完全な記述が必要なユーザは[パートVI](#)を参照してください。

対象読者は、PostgreSQLデータベースへの接続およびSQLコマンド発行に慣れ親しんでいるユーザです。まだこれらについて熟知していないユーザは、本書の前に[パートI](#)をお読みになることをお勧めします。SQLコマンドは通常PostgreSQLの対話式端末psqlを使用して入力しますが、同様の機能を備えた他のプログラムも使用することができます。

目次

4章 [SQLの構文](#)

- 4.1. [語量の構成](#)
- 4.2. [評価式](#)
- 4.3. [関数呼び出し](#)

5章 [データ定義](#)

- 5.1. [テーブルの基本](#)
- 5.2. [デフォルト値](#)
- 5.3. [制約](#)
- 5.4. [システム列](#)
- 5.5. [テーブルの変更](#)
- 5.6. [権限](#)
- 5.7. [スキーマ](#)
- 5.8. [継承](#)
- 5.9. [パーティショニング](#)
- 5.10. [外部データ](#)
- 5.11. [その他のデータベースオブジェクト](#)
- 5.12. [依存関係の追跡](#)

6章 [データ操作](#)

- 6.1. [データの挿入](#)
- 6.2. [データの更新](#)
- 6.3. [データの削除](#)

7章 [問い合わせ](#)

- 7.1. [概要](#)
- 7.2. [テーブル式](#)
- 7.3. [選択リスト](#)
- 7.4. [問い合わせの結合](#)
- 7.5. [行の並び替え](#)
- 7.6. [LIMITとOFFSET](#)
- 7.7. [VALUESリスト](#)
- 7.8. [WITH問い合わせ（共通テーブル式）](#)

8章 [データ型](#)

- 8.1. [数値データ型](#)
- 8.2. [通貨型](#)
- 8.3. [文字型](#)
- 8.4. [バイナリ列データ型](#)
- 8.5. [日付/時刻データ型](#)
- 8.6. [論理値データ型](#)

- 8.7. [列挙型](#)
- 8.8. [幾何データ型](#)
- 8.9. [ネットワークアドレス型](#)
- 8.10. [ビット列データ型](#)
- 8.11. [テキスト検索に関する型](#)
- 8.12. [UUID型](#)
- 8.13. [XML型](#)
- 8.14. [JSONデータ型](#)
- 8.15. [配列](#)
- 8.16. [複合型](#)
- 8.17. [範囲型](#)
- 8.18. [オブジェクト識別子データ型](#)
- 8.19. [疑似データ型](#)

9章 [関数と演算子](#)

- 9.1. [論理演算子](#)
- 9.2. [比較演算子](#)
- 9.3. [算術関数と演算子](#)
- 9.4. [文字列関数と演算子](#)
- 9.5. [バイナリ文字列関数と演算子](#)
- 9.6. [ビット文字列関数と演算子](#)
- 9.7. [パターンマッチ](#)
- 9.8. [データ型書式設定関数](#)
- 9.9. [日付/時刻関数と演算子](#)
- 9.10. [列挙型サポート関数](#)
- 9.11. [幾何関数と演算子](#)
- 9.12. [ネットワークアドレス関数と演算子](#)
- 9.13. [テキスト検索関数と演算子](#)
- 9.14. [XML関数](#)
- 9.15. [JSON関数](#)
- 9.16. [シーケンス操作関数](#)
- 9.17. [条件式](#)
- 9.18. [配列関数と演算子](#)
- 9.19. [範囲関数と演算子](#)
- 9.20. [集約関数](#)
- 9.21. [ウィンドウ関数](#)
- 9.22. [副問い合わせ式](#)
- 9.23. [行と配列の比較](#)
- 9.24. [集合を返す関数](#)
- 9.25. [システム情報関数](#)
- 9.26. [システム管理関数](#)
- 9.27. [トリガ関数](#)

10章 [型変換](#)

- 10.1. [概要](#)
- 10.2. [演算子](#)
- 10.3. [関数](#)
- 10.4. [値の格納](#)
- 10.5. [UNION、CASEおよび関連する構文](#)

11章 [インデックス](#)

- 11.1. [序文](#)
- 11.2. [インデックスの種類](#)
- 11.3. [複数列インデックス](#)
- 11.4. [インデックスとORDER BY](#)
- 11.5. [複数のインデックスの組み合わせ](#)
- 11.6. [一意インデックス](#)
- 11.7. [式に対するインデックス](#)
- 11.8. [部分インデックス](#)
- 11.9. [演算子クラスと演算子族](#)
- 11.10. [インデックスと照合順序](#)
- 11.11. [インデックス使用状況の検証](#)

12章 [全文検索](#)

- 12.1. [導入](#)
- 12.2. [テーブルとインデックス](#)
- 12.3. [テキスト検索の制御](#)
- 12.4. [追加機能](#)
- 12.5. [パーサ](#)
- 12.6. [辞書](#)
- 12.7. [設定例](#)
- 12.8. [テキスト検索のテストとデバッグ](#)
- 12.9. [GiSTおよびGINインデックス種類](#)
- 12.10. [psqlサポート](#)
- 12.11. [制限事項](#)
- 12.12. [8.3以前のテキスト検索からの移行](#)

13章 [同時実行制御](#)

- 13.1. [序文](#)
- 13.2. [トランザクションの分離](#)
- 13.3. [明示的ロック](#)
- 13.4. [アプリケーションレベルでのデータの一貫性チェック](#)
- 13.5. [ロックとインデックス](#)

14章 [性能に関するヒント](#)

- 14.1. [EXPLAINの利用](#)
- 14.2. [プランナで使用される統計情報](#)
- 14.3. [明示的なJOIN句でプランナを制御する](#)
- 14.4. [データベースへのデータ投入](#)
- 14.5. [永続性がない設定](#)

[前のページ](#)
まとめ

[ホーム](#)

[次のページ](#)
SQLの構文

第 4章SQLの構文

目次

4.1. [語彙の構成](#)

- 4.1.1. [識別子とキーワード](#)
- 4.1.2. [定数](#)
- 4.1.3. [演算子](#)
- 4.1.4. [特殊文字](#)
- 4.1.5. [コメント](#)
- 4.1.6. [演算子の優先順位](#)

4.2. [評価式](#)

- 4.2.1. [列の参照](#)
- 4.2.2. [位置パラメータ](#)
- 4.2.3. [添字](#)
- 4.2.4. [フィールド選択](#)
- 4.2.5. [演算子の呼び出し](#)
- 4.2.6. [関数呼び出し](#)
- 4.2.7. [集約式](#)
- 4.2.8. [ウィンドウ関数呼び出し](#)
- 4.2.9. [型キャスト](#)
- 4.2.10. [照合順序式](#)
- 4.2.11. [スカラ副問い合わせ](#)
- 4.2.12. [配列コンストラクタ](#)
- 4.2.13. [行コンストラクタ](#)
- 4.2.14. [式の評価規則](#)

4.3. [関数呼び出し](#)

- 4.3.1. [位置表記の使用](#)
- 4.3.2. [名前付け表記の使用](#)
- 4.3.3. [混在表記の利用](#)

本章ではSQLの構文について説明します。本章の内容は、データの定義や変更のためにSQLコマンドを適用する方法について詳しく説明する以後の章を理解する上での基礎となります。

この章はSQLデータベース間で異なって実装されたり、またはPostgreSQLに固有な幾つかの規則と概念を含んでいるので、SQLについて熟知しているユーザも本章を注意深く読むことをお勧めします。

4.1. 語彙の構成

SQLの入力は、ひと続きのコマンドからなります。コマンドはトークンが繋がったもので構成され、最後はセミコロン (";") で終わります。入力ストリームの終了もやはりコマンドを終わらせます。どのトークンが有効かは特定のコマンドの構文によります。

トークンはキーワード、識別子、引用符で囲まれた識別子、リテラル（もしくは定数）、特別な文字シンボルです。トークンは通常空白（スペース、タブ、改行）で区切られますが、曖昧さがなければ（一般的には特別な文字が他のトークン型と隣接している場合のみ）必要ありません。

例えば、以下のものは（構文的に）正しいSQLの入力です。

```
SELECT * FROM MY_TABLE;
UPDATE MY_TABLE SET A = 5;
INSERT INTO MY_TABLE VALUES (3, 'hi there');
```

この例は1行に1つのコマンドを記述した、3つのコマンドが連続しています（必ずしも1つのコマンドを1行で書く必要はありません。1行に複数のコマンドを入力することも可能ですし、1つのコマンドを複数行に分けて記述することも可能です）。

さらに、入力されたSQLにコメントが付いていても構いません。コメントはトークンではなく、その効果は空白と同じです。

SQL構文は、どのトークンがコマンドを識別し、どれがオペランドでどれがパラメータかに関してはさほど首尾一貫していません。最初のいくつかのトークンは一般にコマンド名です。したがって、上記の例において"SELECT"、"UPDATE"、"INSERT"コマンドについて通常説明することになります。しかし、例えばUPDATEコマンドでは、SETトークンが特定の位置に常に記述されなければなりませんし、この例で使われているINSERTコマンドを完結するためにはVALUESトークンが必要です。それぞれのコマンドの正確な構文規則は[パートVI](#)で説明されています。

4.1.1. 識別子とキーワード

上記の例に出てくるSELECT、UPDATE、もしくはVALUESのようなトークンは、キーワードの一例です。キーワードとは、SQL言語で決まった意味を持っている単語です。MY_TABLEトークンやAトークンは識別子の一例です。これらは、使われるコマンドによって、テーブル、列、他のデータベースオブジェクトの名前を識別します。したがって、単に"名前"と呼ばれることもあります。キーワードと識別子は同じ語彙の構造を持つため、言語を知らなくてはトークンが識別子なのかキーワードなのかかわからないということになります。全てのキーワードのリストは[付録C](#)にあります。

SQL識別子とキーワードは、文字（a～zおよび発音区別符号付き文字と非Latin文字）、アンダースコア（_）で始まらなければいけません。識別子またはキーワードの中で続く文字は、文字、アンダースコア、数字（0～9）あるいはドル記号(\$)を使用することができます。標準SQLの記述に従うと、ドル記号は識別子内では使用できないことに注意してください。ですから、これを使用するとアプリケーションの移植性は低くなる可能性があります。標準SQLでは、数字を含む、あるいはアンダースコアで始まったり終わったりするキーワードは定義されていません。したがって、この形式の識別子は標準の今後の拡張と競合する可能性がないという意味で安全と言えます。

システムはNAMEDATALEN-1バイトより長い識別子の文字数を使いません。より長い名前をコマンドで書くことはできますが、短く切られてしまいます。デフォルトではNAMEDATALENは64なので、識別子は最長で63バイトです。この制限が問題になる場合は、src/include/pg_config_manual.h内のNAMEDATALEN定数の値を変更して増やすことができます。

キーワードと引用符付きでない(クォートされていない)識別子は大文字と小文字を区別しません。したがって、

```
UPDATE MY_TABLE SET A = 5;
```

は、以下の文と同じ意味になります。

```
uPDaTE my_TabLe SeT a = 5;
```

慣習的には、キーワードを大文字で、名前を小文字で書きます。例えば下記のようになります。

```
UPDATE my_table SET a = 5;
```

識別子には副次的な種類もあります。区切り識別子あるいは引用符付き識別子です。任意の文字の連なりを二重引用符(")で囲んだものです。区切り識別子は常に識別子であって、キーワードではありません。ですから、"select"は"select"という名前の列あるいはテーブルを問い合わせるために使えますが、引用符の付かないselectはキーワードとして理解されるので、テーブルもしくは列名が期待される部分では解析エラーを起こします。引用符付き識別子は下記の例のように書くことができます。

```
UPDATE "my_table" SET "a" = 5;
```

引用符付き識別子は、文字コード0以外であればどのような文字でも使えます（二重引用符を含めたい場合は、二重引用符

を2つ入力します)。この決まりがあることによって、普段使えない空白やアンパサンド (&) を含むテーブル名や列名を作ることが可能です。この場合においても長さの制限は適用されます。

引用符付き識別子の変異体はそれらのコード番号で識別されるエスケープされたUnicode文字を含むことができます。この変異体は、二重引用符開始の直前にU& (後にアンパサンドが付いた大文字・小文字のU) で始まります。例えば、U&"foo"となります。(これにより演算子&との不明確性が生じることに注意してください。この問題を回避するには空白を演算子の前後に入れます。) 引用符の中で、Unicode文字はバックスラッシュとそれに続く4桁16進数の文字コード番号で、またはもう1つの方法として、バックスラッシュに続いてプラス符号、そして続いた6桁16進数の文字コード番号によりエスケープ形式で指定されます。例えば、識別子"data"は次のように書かれます。

```
U&"d¥0061t¥+000061"
```

次のより一般的でない例はロシア語の"slon" (象) をキリル語で書いたものです。

```
U&"¥0441¥043B¥043E¥043D"
```

バックスラッシュではない異なるエスケープ文字を使用したい場合、文字列の後にUESCAPE句を使用して指定することが可能です。例をあげます。

```
U&"d!0061t!+000061" UESCAPE '!'
```

エスケープ文字には、16進表記用の文字、プラス記号、単一引用符、二重引用符、空白文字以外の任意の単一文字を使用することができます。エスケープ文字には二重引用符ではなく単一引用符で記述していることに注意してください。

識別子内にエスケープ文字をそのまま含めるためには、それを2つ記述してください。

Unicodeエスケープ構文はサーバの符号化方式がUTF8の場合のみ有効です。他のサーバ符号化方式が用いられている場合、ASCII範囲 (¥007Fまで) のコードポイントのみ指定できます。技術的には6桁形式の機能によりこれは不要になりますが、4桁または6桁の形式は、U+FFFFより大きなコードポイントを持つ文字を構成するUTF-16サロゲートペアを指定するために使用することができます。(サロゲートペアは直接格納されませんが、UTF8に符号化される場合には一つのコードポイントに結合されます。)

引用符が付かない名前は常に小文字に解釈されますが、識別子を引用符で囲むことによって大文字と小文字が区別されるようになります。例えば、識別子FOO、foo、"foo"はPostgreSQLによれば同じものとして解釈されますが、"Foo"と"FOO"は、これら3つとも、またお互いに違ったものとして解釈されます (PostgreSQLが引用符の付かない名前を小文字として解釈することは標準SQLと互換性がありません。標準SQLでは引用符の付かない名前は大文字に解釈されるべきだとされています。したがって標準SQLによれば、fooは"FOO"と同じであるべきで、"foo"とは異なるはずなのです。もし移植可能なアプリケーションを書きたいならば、特定の名前は常に引用符で囲むか、あるいはまったく囲まないかのいずれかに統一することをお勧めします)。

4.1.2. 定数

PostgreSQLには、3つの暗黙に型付けされる定数があります。文字列、ビット文字列、そして数字です。定数は明示的な型で指定することもでき、その場合はシステムによる、より正確な表現と効率の良い操作が可能になります。こうした他の方法については後ほど説明します。

4.1.2.1. 文字列定数

SQLにおける文字列定数は、単一引用符 (') で括られた任意の文字の並びです。例えば、'This is a string'です。文字列定数内の単一引用符の記述方法は、2つ続けて単一引用符を記述することです。例えば、'Dianne''s horse'です。二重引用符 (") とは同一ではない点に注意してください。

2つの文字列定数が、少なくとも1つの改行を含んだ空白で区切られている場合は、2つの定数は連結され、実質的に1つの定数として書かれたように処理されます。例を示します。

```
SELECT 'foo'
       'bar';
```

は、

```
SELECT 'foobar';
```

と同じです。しかし、

```
SELECT 'foo'      'bar';
```

は有効な構文ではありません (このちょっとした奇妙な振舞いはSQLで決められているもので、PostgreSQLではこの標準に従っています)。

4.1.2.2. C形式エスケープでの文字列定数

PostgreSQLでは、また、"エスケープ"文字定数を受け付けます。これは標準SQLの拡張です。エスケープ文字定数は、開始単一引用符の直前に`␣`(大文字でも小文字でもかまいません)を記述することで指定されます。例えば`␣'foo'`です。(複数行に渡るエスケープ文字定数では、最初の開始引用符の前にのみ`␣`を記述してください。) エスケープ文字列の中では、バックスラッシュ文字(`¥`)によりC言語のようなバックスラッシュシーケンスが開始し、その中でバックスラッシュとそれに続く文字の組み合わせが(表4-1で示したように)特別なバイト値を表現します。

表 4-1. バックスラッシュエスケープシーケンス

Backslash Escape Sequence	解釈
¥b	後退
¥f	改ページ
¥n	改行
¥r	復帰
¥t	タブ
¥o, ¥oo, ¥ooo (o = 0 - 7)	8進数バイト値
¥xh, ¥xhh (h = 0 - 9, A - F)	16進数バイト値
¥uxxxx, ¥Uxxxxxxxx (x = 0 - 9, A - F)	16もしくは32ビットの16進数 Unicode 文字値

バックスラッシュの後のそのほかの全ての文字はそのまま扱われます。従って、バックスラッシュ文字を含ませるときは2つのバックスラッシュ(`¥¥`)を記載します。同時に、エスケープ文字列の中の1つの単一引用符は、通常の方法の`'`に加え、`¥'`として含むことができます。

特に10進数や16進数エスケープを用いて作成されるバイトシーケンスが、サーバ文字セット符号化方式において有効な文字で構成されていることはコードを書く人の責任です。サーバ符号化方式がUTF-8の場合、Unicodeエスケープか、項4.1.2.3で説明したもう一方のUnicodeエスケープ構文を代わりとして使用しなければなりません。(代案は手作業でUTF-8符号化を行い、書き出さなくてはならないのでとても厄介です。)

Unicodeエスケープ構文は、サーバの符号化方式がUTF8である場合のみ、完全に動作します。他のサーバ符号化方式が使用されている場合、ASCII範囲(`¥u007F`まで)のコードポイントのみを指定することができます。技術的には8桁形式の機能によりこれは不要になりますが、4桁または8桁の形式は、U+FFFFより大きなコードポイントを持つ文字を構成するUTF-16サロゲートペアを指定するために使用することができます。(サーバの符号化方式がUTF8の場合にサロゲートペアが使用される時、まず単一のコードポイントに組み合わせられ、その後にUTF-8に符号化されます。)

注意

設定パラメータ[standard_conforming_strings](#)が `off` の場合、PostgreSQLはバックスラッシュエスケープを通常の文字定数とエスケープ文字定数の両方で認識します。しかし、PostgreSQL9.1からデフォルトは`on`になりました。これはバックスラッシュエスケープはエスケープ文字列でのみ認識されるということになります。この振る舞いはSQL標準仕様に即していますが、バックスラッシュエスケープを常に認識するという歴史的な動作に依存しているアプリケーションは動作しなくなるでしょう。回避策として、このパラメータを`off`にすることはできますが、バックスラッシュエスケープの使用を避けるよう移植するのが良いでしょう。特殊文字を表現するためにバックスラッシュを使用する必要がある場合、`␣`をつけて文字列定数を記述してください。

`standard_conforming_strings`の他に、設定パラメータ[escape_string_warning](#)および[backslash_quote](#)が文字定数内のバックスラッシュの動作を決定します。

コードゼロの文字は文字列定数の中に入れられません。

4.1.2.3. Unicodeエスケープがある文字列定数

PostgreSQLは同時に、文字コード番号で任意のUnicode文字を指定可能な別の文字列に対するエスケープ構文を提供します。Unicodeエスケープ文字列定数は、開始引用符の直前に`U&`(後にアンバサンドが付いた大文字・小文字のU)で始まります。例えば、`U&'foo'`となります。(これにより演算子`&`との不明確性が生じることに注意してください。この問題を回避するには空白を演算子の前後に入れます。) 引用符の中で、Unicode文字はバックスラッシュとそれに続く4桁16進数の文字コード番号で、またはもう1つの方法として、バックスラッシュに続いてプラス符号、そして続いた6桁16進数の文字コード番号によりエスケープ形式で指定されます。例えば、文字列`'data'`は次のように書かれます。

```
U&'d¥0061t¥+000061'
```

次のより一般的でない例はロシア語の"slon"(象)をキリル文字で書いたものです。

```
U&'¥0441¥043B¥043E¥043D'
```

バックスラッシュではない異なるエスケープ文字を使用したい場合、文字列の後に`UESCAPE`句を使用して指定することが可能です。例をあげます。

```
U&'d!0061t!+000061' UESCAPE '!'
```

エスケープ文字には、16進表記用の文字、プラス記号、単一引用符、二重引用符、空白文字以外の任意の単一文字を使用することができます。

Unicodeエスケープ構文はサーバの符号化方式が`UTF8`の場合のみ有効です。他のサーバ符号化方式が用いられている場合、ASCII範囲（`¥007F`まで）のコードポイントのみ指定できます。技術的には6桁形式の機能によりこれは不要になりますが、4桁または6桁の形式は、`U+FFFF`より大きなコードポイントを持つ文字を構成するUTF-16サロゲートペアを指定するために使用することができます。（サーバの符号化方式が`UTF8`の場合にサロゲートペアが使用される時、まず単一のコードポイントに組み合わせられ、その後にUTF-8に符号化されます。）

同様に、文字列定数に対するユニコードエスケープ構文は設定パラメータ`standard_conforming_strings`が有効なときのみ動作します。そうでないとこの構文は、SQLインジェクションと、類似したセキュリティ問題に繋がる時点までSQL文を構文解釈するクライアントを惑わすことになるからです。

文字列の中に、文字どおりにエスケープ文字を包括するにはエスケープ文字を2回書きます。

4.1.2.4. ドル記号で引用符付けされた文字列定数

SQLにおける文字列定数の指定構文はたいいていの場合便利ですが、対象とする文字列内に多くの単一引用符やバックスラッシュがあると、それらを全て二重にしなければなりませんので理解しづらくなります。こうした状況においても問い合わせの可読性をより高めるためにPostgreSQLは、"ドル引用符付け"という他の文字列定数の指定方法を提供します。ドル引用符付けされた文字列定数は、ドル記号（`$`）、省略可能な0個以上の文字からなる"タグ"、ドル記号、文字列定数を構成する任意の文字の並び、ドル記号、この引用符付けの始めに指定したものと同一タグ、ドル記号から構成されます。例えば、"Dianne's horse"という文字列をドル引用符付けを使用して指定する方法には、以下の2つがあります。

```
$$Dianne's horse$$
$SomeTag$Dianne's horse$SomeTag$
```

ドル引用符付けされた文字列は既にエスケープされていますので、文字列定数は全てそのまま記述することができます。その並びが開始タグに一致しない限り、バックスラッシュもドル記号も特別なものではありません。

各入れ子レベルに異なるタグを付けることで、ドル引用符付けされた文字列を入れ子にすることができます。これは、関数定義を作成する時に非常によく使用されます。以下に例を示します。

```
$function$
BEGIN
    RETURN ($1 ~ $q$[¥t¥r¥n¥v¥¥¥]¥q$);
END;
$function$
```

ここで、`$q$[¥t¥r¥n¥v¥¥¥]¥q$` は、ドル引用符付けされた`[¥t¥r¥n¥v¥¥¥]` リテラル文字列を表し、PostgreSQLがこの関数本体を実行する時に認識されます。しかし、この並びは、外側のドル引用符用の区切り文字`$function$`に一致しませんので、外側の文字列を対象としている場合は単なる文字の並びとなります。

もしあれば、ドル引用符付けされた文字列のタグは、引用符付けされていない識別子と同じ規則に従います。ただし、タグにはドル記号を含めることはできません。タグは大文字小文字を区別します。したがって、`$tag$string content$tag$`は正しいのですが、`$TAG$string content$tag$`は間違いです。

キーワードや識別子の後にドル引用符付けされた文字列を続ける場合は、空白でそれを区切らなければなりません。さもないと、ドル引用符の区切り文字は、直前の識別子の一部として解釈されます。

ドル引用符付けは、標準SQLで定義されていません。しかし、複雑な文字列リテラルを記述する場合は標準準拠の単一引用符構文よりも便利ながよくあります。特に、他の定数の内部に文字列定数を記述するような場合は役に立ちます。こうした状況は手続き関数の定義でよく必要とされます。単一引用符構文では、上の例のバックスラッシュはそれぞれ、4個のバックスラッシュで記述しなければなりません。この4つのバックスラッシュは、元の文字列定数を解析する際に2つに減少され、そして、関数を実行する際に内部の文字列定数が再解析され1つに減少します。

4.1.2.5. ビット文字列定数

ビット文字列定数は`B`（大文字もしくは小文字）が引用符の始まりの前に付いている（間に空白はありません）通常の文字列定数のように見えます。例えば`B'1001'`のようになります。ビット文字列定数の中で許可される文字は`0`と`1`のみです。

その他にも、ビット文字列定数は`X'1FF'`といった具合に、先頭に`X`（大文字または小文字）を使用して16進表記で指定すること

もできます。この表記は、各16進数値をそれぞれ4つの2進数値に置き換えたビット文字列定数と同等です。

どちらの形式のビット文字列定数でも、通常の文字列定数と同じように複数行にわたって続けて書くことができます。ドル引用符付けはビット文字列定数では使用できません。

4.1.2.6. 数値定数

数値定数は下記の一般的な形で受け付けられます。

```
digits
digits.[digits][e[+-]digits]
[digits].digits[e[+-]digits]
digitse[+-]digits
```

ここで`digits`は1つ以上の10進数字（0～9）です。小数点を使用する場合は、少なくとも1つの数字が小数点の前か後になくなくてはなりません。指数記号`e`の付く形式を使う場合には`e`の後に少なくとも1つの数字がなければいけません。空白や他の文字は、定数の中に埋め込むことはできません。プラスまたはマイナスの符号を先頭につけても、定数の一部とはみなされないことに注意してください。これらの符号は定数に適用される演算子とみなされます。

下記は有効な数値定数のいくつかの例です。

```
42
3.5
4.
.001
5e2
1.925e-3
```

小数点も指数も含まない数値定数の場合、まずその値が`integer`型（32ビット）に適合すれば`integer`型であるとみなされます。そうでない場合、`bigint`型（64ビット）で適合すれば`bigint`型とみなされます。どちらでもない場合は、`numeric`型とみなされます。定数が小数点または指数あるいはその両方を含む場合は、常に最初に`numeric`型であるとみなされます。

数値定数に最初に割り振られるデータ型は、型解決アルゴリズムの開始点に過ぎません。ほとんどの場合、定数には文脈に基づいて自動的に正しい型が割り振られます。必要であれば、特定のデータ型をキャストして、数値がそのデータ型として解釈されるように強制することができます。例えば、以下のようにして数値を`real`型（`float4`）として処理することができます。

```
REAL '1.23' -- 文字列書式
1.23::REAL  -- (歴史的な) PostgreSQL 書式
```

実のところ、これらは以下で説明する一般的なキャスト記法の特別な場合です。

4.1.2.7. 他の型の定数

任意の型の定数は下記の表記のいずれかを使って入力することができます。

```
type 'string'
'string'::type
CAST ( 'string' AS type )
```

文字列定数のテキストは`type`と呼ばれる型の入力変換ルーチンへと渡されます。結果は指示された型の定数です。明示的な型キャストは、定数がどの型でなければならないかについて曖昧な点がなければ（例えば定数が直接テーブル列に代入されている場合）省略しても構いません。その場合自動的に型強制されます。

文字列定数は通常のSQL記法でもドル引用符付けでも記述することができます。

関数のような構文を使って型強制を指定することも可能です。

```
typename ( 'string' )
```

しかし、全ての型の名前でのこの方法は使用できるというわけではありません。詳細は[項4.2.9](#)を参照してください。

`::`、`CAST()` や関数呼び出し構文も、[項4.2.9](#)で説明する通り、任意の式の実行時の型変換を指定するために使うことができます。しかし、構文的なあいまいさをなくすために、`type 'string'`という形式は単なるリテラル定数を指定する場合にのみ使うことができます。この他`type 'string'`構文には、配列型では動作しないという制限があります。配列型の定数を指定する場合は`::`か`CAST()`を使用してください。

`CAST()` 構文はSQLに従っています。`type 'string'`構文は、標準を一般化したものです。SQLでは、この構文を数個のデータ型でのみ規定しています。しかし、PostgreSQLではすべての型で使うことができます。`::`付きの構文は、関数呼び出し構文と同様に歴史的にPostgreSQLで使用されてきました。

4.1.3. 演算子

演算子はNAMEDATALEN-1までの（デフォルトは63）長さの、以下に示すリストに含まれる文字の並びです。

+ - * / < > = ~ ! @ # % ^ & | ` ?

しかし、演算子の名前にはいくつかの制約があります。

- --と/*は演算子名の中に使うことができません。なぜならこれらはコメントの始まりと解釈されるからです。
- 複数文字の演算子名は、その名前が少なくとも下記の文字の1つ以上を含まない限り、+や-で終わることができません。

~ ! @ # % ^ & | ` ?

例えば、@-は演算子名として認められていますが、*-は認められていません。この制限によりPostgreSQLは、SQLに準拠する問い合わせをトークン同士の間に空白を要求せず、解析することができます。

非SQL標準の演算子名を使う場合、通常は曖昧さを回避するために、隣り合った演算子を空白で区切る必要があります。例えば@という左単項演算子を定義した場合、X*@Yとは書けません。PostgreSQLがこれを確実に1つではなく2つの演算子名として解釈できるように、X* @Yと書く必要があります。

4.1.4. 特殊文字

英数字ではないいくつかの文字は、演算子であることとは異なる特殊な意味を持っています。使用法の詳細はそれぞれの構文要素についてのところで説明します。本節では、単にその存在を知らせ、これらの文字の目的をまとめるに留めます。

- 直後に数字が続くドル記号 (\$) は、関数定義の本体またはプリペアド文中の位置パラメータを表すために使われます。他の文脈ではドル記号は演算子名の一部であるかもしれませんが、ドル引用符付けされた文字列定数の一部とすることができます。
- 括弧 () は、通常通り式をまとめ優先するという意味を持ちます。場合によっては括弧は、特定のSQLコマンドの固定構文の一部として要求されることがあります。
- 大括弧 [] は、配列要素を選択するために使われます。配列に関する詳しい情報は[項8.15](#)を参照してください。
- カンマ (,) は、リストの要素を区切るために構文的構造体で使われることがあります。
- セミコロン (;) は、SQLコマンドの終わりを意味します。文字列定数または引用符付き識別子以外では、コマンドの途中では使うことができません。
- コロン (:) は、配列から"一部分"を取り出すために使われます ([項8.15](#)を参照してください)。いくつかのSQL方言（埋め込みSQLなど）では、コロンは変数名の接頭辞として使われます。
- アスタリスク (*) は、いくつかの文脈において、テーブル行や複合型の全てのフィールドを表現するために使用されます。また、集約関数の引数として使われる場合も特殊な、つまり、その集約が明示的なパラメータをまったく必要としないという意味を持ちます。
- ピリオド (.) は数値定数の中で使われます。また、スキーマ、テーブル、列名を区切るためにも使われます。

4.1.5. コメント

コメントは二重ハイフンで始まる文字の並びで、行の終わりまで続きます。例えば以下ようになります。

```
-- これは標準SQLのコメントです
```

他にも、C言語様式のブロックコメントも使用できます。

```
/* ネストされた複数行にわたる
 * コメント */
*/
```

コメントは/*で始まり、対応する*/で終わります。これらのブロックコメントはC言語とは異なり、標準SQLで規定されているように入れ子にすることができます。したがって、既存のブロックコメントを含む可能性のある大きなコードのブロックをコメントアウトすることができます。

コメントは、その後の構文解析が行われる前に入力ストリームから取り去られ、空白によって適切に置き換えられます。

4.1.6. 演算子の優先順位

表4-2は、PostgreSQLの演算子の優先順位と結合性を示しています。ほとんどの演算子は同じ優先順位を持ち、左結合します。演算子の優先順位と結合性はパーサに組み込まれています。これは非直感的な動作を導く可能性があります。例えば、ブーリアン演算子<>は、ブーリアン演算子<=>とは違った優先順位を持ちます。さらに、二項演算子と単項演算子を組み合わせる場合は括弧を加える必要がある場合があります。例えば下記のような場合です。

```
SELECT 5 ! - 6;
```

は、

```
SELECT 5 ! (- 6);
```

と解析されます。なぜならば、パーサは!が中置ではなく接尾演算子として定義されていることに最後まで気が付かないためです。この場合、求める結果を得るためには下記のように書く必要があります。

```
SELECT (5 !) - 6;
```

これが拡張性を求める故の代償です。

表 4-2. 演算子の優先順位（強いものから）

演算子/要素	結合性	説明
.	左	テーブル/列名の区切り文字
::	左	PostgreSQL方式の型キャスト
[]	左	配列要素選択
+ -	右	単項加算、単項減算
^	左	累乗
* / %	左	掛け算、割り算、剰余
+ -	左	加算、減算
IS		IS TRUE, IS FALSE, IS NULL, その他
ISNULL		NULLかどうかを試す
NOTNULL		非NULLかどうかを試す
(その他)	左	その他全ての組み込み、あるいはユーザ定義の演算子
IN		メンバシップを設定する
BETWEEN		範囲内に含有
OVERLAPS		時間間隔の重複
LIKE ILIKE SIMILAR		文字列パターンの一致
< >		小なり、大なり
=	右	等しい、代入
NOT	右	論理否定
AND	左	論理積
OR	左	論理和

演算子優先順位の規則は、上記で触れた組み込み演算子と同じ名前を持つユーザ定義演算子にも当てはまります。例えばもし"+"演算子のある独自のデータ型に定義すると、新しい演算子は何をするかにかかわらず、"+"組み込み演算子と同じ優先順位を持つようになります。

次の例のように、OPERATOR構文でスキーマで修飾された演算子名を使用する場合、

```
SELECT 3 OPERATOR(pg_catalog.+) 4;
```

OPERATOR構文は、表4-2の"その他"演算子で示されているデフォルトの優先順位を持つとみなされます。これは、OPERATOR()にどの特定の演算子が入る場合でも変わりません。

4.2. 評価式

評価式は、例えば `SELECT` コマンドの目的リストとして、`INSERT` や `UPDATE` の新しい列の値として、もしくはいくつかのコマンドの検索条件として様々な文脈の中で使われます。評価式の結果は、テーブル式の結果（つまりテーブル）から区別するために、スカラと呼ばれることもあります。したがって、評価式はスカラ式（またはもっと簡単に式）とも呼ばれます。式の構文によって、算術、論理、集合などの演算を行う基本的な部分で値の計算を行うことができます。

評価式は下記のうちのいずれかです。

- 定数あるいはリテラル値
- 列の参照
- 関数宣言の本体やプリペアド文における、位置パラメータ参照
- 添字付きの式
- フィールド選択式
- 演算子の呼び出し
- 関数呼び出し
- 集約式
- ウィンドウ関数呼び出し
- 型キャスト
- 照合順序(collation)式
- スカラ副問い合わせ
- 配列コンストラクタ
- 行コンストラクタ
- （副式をグループ化したり優先順位を変更するのに使用される）括弧で囲まれた別の評価式

これ以外にも、式として分類されるけれども一般的な構文規約には従わない、いくつかの構成要素があります。これらは一般的に関数あるいは演算子の意味を持ちます。第9章の該当部分で説明されています。例を挙げると `IS NULL` 句があります。

項4.1.2で既に定数については説明しました。続く節では残りのオプションについて説明します。

4.2.1. 列の参照

列は、下記のような形式で参照することができます。

`correlation.columnname`

`correlation` は、テーブル名（スキーマで修飾されている場合もあります）、あるいは `FROM` 句で定義されたテーブルの別名です。相関名と区切り用のドットは、もし列名が現在の問い合わせで使われる全てのテーブルを通して一意である場合は省略することができます。（第7章も参照してください）。

4.2.2. 位置パラメータ

位置パラメータ参照は、外部から SQL 文に渡される値を示すために使用されます。パラメータは SQL 関数定義およびプリペアド問い合わせの中で使用されます。また、クライアントライブラリの中には、SQL コマンド文字列とデータ値を分離して指定できる機能をサポートするものもあります。この場合、パラメータは行外データ値を参照するために使用されます。パラメータ参照の形式は以下の通りです。

`$number`

例えば、関数 `dept` の定義が以下のようにされたとします。


```
CREATE FUNCTION dept(text) RETURNS dept
AS $$ SELECT * FROM dept WHERE name = $1 $$
LANGUAGE SQL;
```

ここで\$1は関数が呼び出される時に最初の関数引数の値を参照します。

4.2.3. 添字

配列の値を与える式の場合、特定の配列要素の値は以下のように記述することで展開されます。

```
expression[subscript]
```

また、複数の要素をまたがる ("配列の一部") 場合は以下のように記述することで展開されます。

```
expression[lower_subscript:upper_subscript]
```

(ここで大括弧[]はリテラルとして現れています。) 各subscriptは自身が式であり、整数値を生成しなければなりません。

一般的には、配列expressionは括弧で括らなければなりませんが、添字が付いた式が単なる列参照や位置パラメータであった場合、その括弧を省略することができます。 以下に例を示します。

```
mytable.arraycolumn[4]
mytable.two_d_column[17][34]
$l[10:42]
(arrayfunction(a,b))[42]
```

最後の例では括弧が必要です。 配列の詳細は[項8.15](#)を参照してください。

4.2.4. フィールド選択

式が複合型 (行型) の値を生成する場合、行の特定のフィールドは以下のように記述することで展開できます。

```
expression.fieldname
```

一般的には、行expressionは括弧で括らなければなりません。 しかし、選択元となる式が単なるテーブル参照や位置パラメータの場合、括弧を省略することができます。 以下に例を示します。

```
mytable.mycolumn
$l.somecolumn
(rowfunction(a,b)).col3
```

(したがって、修飾された列参照は実際のところ、単なるこのフィールド選択構文の特殊な場合です。) 重要となる特殊な場合としては、複合型のテーブル列からフィールドを抽出するときです。

```
(compositecol).somefield
(mytable.compositecol).somefield
```

compositecolがテーブル名でなく列名であること、または2番目の場合のmytableがスキーマ名でなくテーブル名であることを示すため丸括弧が要求されます。

SELECT列のリスト([項7.3](#)参照)に.*を記述することで、複合型の全ての値を問い合わせることが可能です。

```
(compositecol).*
```

4.2.5. 演算子の呼び出し

演算子の呼び出しには以下の3構文が可能です。

```
expression operator expression (二項中置演算子)
operator expression (単項前置演算子)
expression operator (単項後置演算子)
```

ここでoperatorトークンは、[項4.1.3](#)構文規則に従うもの、もしくはキーワードAND、OR、NOTのいずれか、または以下の形式の修飾された演算子名です。

```
OPERATOR(schema.operatorname)
```

具体的にどんな演算子が存在し、それが単項か二項かどうかは、システムやユーザによってどんな演算子が定義されたかに依存します。 [第9章](#)にて、組み込み演算子について説明します。

4.2.6. 関数呼び出し

関数呼び出しの構文は、関数名（スキーマ名で修飾されている場合があります）に続けてその引数を丸括弧で囲んで列挙したものです。

```
function_name ([expression [, expression ... ]])
```

例えば、以下のものは2の平方根を計算します。

```
sqrt(2)
```

組み込み関数の一覧は[第9章](#)にあります。他の関数はユーザが追加できます。

引数には名前を任意で付与することができます。詳細は[項4.3](#)を見て下さい。

注意: 複合型の単一引数をとる関数はフィールド選択構文をオプショナルに使うことで呼び出すことができます。反対に関数形式でフィールド選択を記述することもできます。この記述方法は`col(table)`や`table.col`となり、どちらを使っても変わりありません。PostgreSQLでは"計算された領域"のエミュレートをする関数の利用が可能なため、これはSQL標準の振る舞いではなくPostgreSQL独自機能となります。詳しくは[項35.4.3](#)を参照してください。

4.2.7. 集約式

集約式は、問い合わせによって選択される行に対して集約関数が適用されることを意味します。集約関数は、例えば入力合計や平均などのように、複数の入力を単一の出力値にします。集約式の構文は下記のうちのいずれかです。

```
aggregate_name (expression [, ... ] [ order_by_clause ] )
aggregate_name (ALL expression [, ... ] [ order_by_clause ] )
aggregate_name (DISTINCT expression [, ... ] [ order_by_clause ] )
aggregate_name ( * )
```

ここで、`aggregate_name`は事前に定義された集約（スキーマ名で修飾された場合もあります）、`expression`はそれ自体に集約式またはウィンドウ関数呼び出しを含まない任意の値評価式、`order_by_clause`は後述の省略可能な`ORDER BY`です。

集約式の最初の構文は、それぞれの入力行に対して1回ずつ集計を呼び出します。`ALL`はデフォルトなので、2つ目の形式は最初の形式と同じです。3番目の形式は、入力行の中にある式の、全ての重複しない値（複数式では重複しない値集合）の集約を呼び出します。最後の形式はそれぞれの入力行に対して1回ずつ集約を呼び出します。具体的な入力値が指定されていないため、これは一般的に`count(*)`集約関数でのみ役に立ちます。

1つ以上の式がNULLを返した場合にその行を破棄するように、ほとんどの集約関数はNULL入力を無視します。これは、特記されていない限り、すべての組み込み集約で正しいと仮定することができます。

例えば、`count(*)`は入力行の合計数を求めます。`count`はNULLを無視しますので、`count(f1)`は`f1`が非NULLである入力行の数を求めます。`count(distinct f1)`は`f1`の重複しない非NULL値の数を求めます。

通常、入力行は順序を指定されずに集計関数に与えられます。多くの場合では問題になりません。たとえば`min`は入力順序に関係なく同一の結果を返します。しかし一部の集約関数(`array_agg`および`string_agg`など)は入力行の順序に依存した結果を返します。こうした集約関数を使用する際は、省略可能な`order_by_clause`を使用して必要とする順序を指定できます。`order_by_clause`は、[項7.5](#)で説明する問い合わせレベルの`ORDER BY`句と同じ構文を取りますが、その式は常に単なる式であり、出力列名や序数とすることはできません。以下に例を示します。

```
SELECT array_agg(a ORDER BY b DESC) FROM table;
```

複数の引数を取る集約関数を扱う場合、`ORDER BY`句はすべての集約引数の後に指定することに注意してください。例えば、

```
SELECT string_agg(a, ',' ORDER BY a) FROM table;
```

であり、

```
SELECT string_agg(a ORDER BY a, ',') FROM table; -- incorrect
```

ではありません。後者は構文的には有効なものですが、2つの`ORDER BY`キーを持つ単一引数の集約関数の呼び出しを表しています。(2つ目のキーは定数なので役には立ちません。)

`order_by_clause`に加え`DISTINCT`が指定された場合、すべての`ORDER BY`式が集約関数の通常の引数に一致しなければなりません。つまり、`DISTINCT`リストに含まれない式でソートすることはできません。

注意: 集計関数における`DISTINCT`と`ORDER BY`の指定機能はPostgreSQL独自の拡張です。

定義済みの集約関数は[項9.20](#)で説明されています。ユーザは他の集約関数を追加することができます。

集約式は、SELECT コマンドの結果リストもしくはHAVING 句内でのみ記述することができます。 WHERE などの他の句では許されません。これらの句は理論上集計結果が形成される前に評価されるためです。

集約式が副問い合わせ（[項4.2.11](#)と[項9.22](#)を参照）内に現れた場合、通常集約は副問い合わせの行全体に対して評価します。しかし、その集約の引数が上位レベルの変数のみを持つ場合は例外です。その場合、上位問い合わせの行全体に対して評価します。全体として、その集約式は、その後、その集約を含む副問い合わせでは外部参照となり、その副問い合わせにおける評価に対しては定数として動作します。結果リストもしくはHAVING 句にのみ現れるという制約は、その集約が属する問い合わせレベルに関連して適用されます。

4.2.8. ウィンドウ関数呼び出し

ウィンドウ関数呼び出しは、問い合わせにより選択された行のある部分に渡って集約のような機能を実現することを表します。通常の集約関数呼び出しと異なり、これは選択された行を1つの行にグループ化することに束縛されず、各行は別途問い合わせ出力に残ります。しかしウィンドウ関数は、ウィンドウ関数呼び出しのグループ化指定（PARTITION BY リスト）に従った、現在の行のグループの一部となる行をすべてスキャンすることができます。ウィンドウ関数呼び出しの構文は以下のいずれかです。

```
function_name ([expression [, expression ...]]) OVER ( window_definition )
function_name ([expression [, expression ...]]) OVER window_name
function_name ( * ) OVER ( window_definition )
function_name ( * ) OVER window_name
```

ここで、window_definition は以下の構文になります。

```
[ existing_window_name ]
[ PARTITION BY expression [, ...] ]
[ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ] [, ...] ]
[ frame_clause ]
```

オプションのframe_clause は次の中の1つです。

```
[ RANGE | ROWS ] frame_start
[ RANGE | ROWS ] BETWEEN frame_start AND frame_end
```

ここでframe_start およびframe_end は以下のいずれかです。

```
UNBOUNDED PRECEDING
value PRECEDING
CURRENT ROW
value FOLLOWING
UNBOUNDED FOLLOWING
```

ここで、expression はそれ自身ウィンドウ関数呼び出しを含まないいかなる値式を表わします。 PARTITION BY およびORDER BY リストは基本的に、それらの式は常に式であって、出力列名または数値でない限り、問い合わせ全体のGROUP BY とORDER BY と同じ構文と動作です。 window_name は、問い合わせのWINDOW 句で定義された名前付きウィンドウ仕様への参照です。名前付きウィンドウ仕様は通常単にOVER window_name で参照されますが、同時に丸括弧の内部にウィンドウ名を書くことができ、そして順序付け句とフレーム句の両方またはいずれか一方を供給します（ここでこれらの句が供給されると、参照されたウィンドウにこれらの句があってはなりません）。後者の構文はWINDOW 句の中の既存のウィンドウ名を変更するのと同じ規則に従います（詳細は[SELECT](#) マニュアルページを見てください）。

frame_clause は、全てのパーティションの代わりにフレーム上で作動するウィンドウ関数に対して、window frame を構成する行の集合を指定します。frame_end を省略した場合のデフォルトはCURRENT ROW です。制限は、frame_start をUNBOUNDED FOLLOWING とすることができない点、frame_end をUNBOUNDED PRECEDING とすることができない点、およびframe_end の選択は上のリストの中でframe_start の選択より先に記述することができない点です。例えば、RANGE BETWEEN CURRENT ROW AND value PRECEDING は許されません。デフォルトのフレーム化オプションはRANGE UNBOUNDED PRECEDING で、RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW と同じです。これはパーティションの開始からORDER BY による順序付けに沿った現在行と同等となる最後の行までのフレームをセットします。（ORDER BY が無い場合は全ての行であることを意味します。）一般的に、UNBOUNDED PRECEDING はパーティションの最初の行をフレームの開始点にすることを意味し、同様にUNBOUNDED FOLLOWING はパーティションの最後の行をフレームの終わりにすることを意味します。（RANGE あるいはROWS のモードに関わらずです。）ROWS モードでは、CURRENT ROW は現在行をフレームの開始もしくは終了とすることを意味します。しかしRANGE モードでは、ORDER BY による順序付けに沿った現在行の最初か最後と同等な行がフレームの開始もしくは終了となります。value PRECEDING とvalue FOLLOWING の形式は、現在のところROWS モードでのみ許可されています。これは、フレームの開始もしくは終了点となる現在行の前や後に、大量の行があることを意味します。value は一切の変数、集計関数、あるいはウィンドウ関数を含まない整数式でなければなりません。またNULL や負数も許可されませんが、現在行を選択することになる0は指定可能です。

組み込みウィンドウ関数は[表9-47](#)に記載されています。その他のウィンドウ関数をユーザが追加することが可能です。また、全ての組み込み、またはユーザ定義の集約関数もウィンドウ関数として使用できます。

を使用した構文は、例えばcount() OVER (PARTITION BY x ORDER BY y) のように、ウィンドウ関数として呼び出しパラメータのない集約関数を使用します。アスタリスク(*)は習慣的に非集約ウィンドウ関数には使われません。通常の集約関数と異なり、集約ウィンドウ関数は、関数引数リストの中でDISTINCT やORDER BY が使われることを許可しません。

ウィンドウ関数呼び出しは問い合わせのSELECTリストとORDER BY句の中でのみ許可されます。

更なるウィンドウ関数についての情報は [項3.5](#), [項9.21](#), [項7.2.4](#)にあります。

4.2.9. 型キャスト

型キャストは、あるデータ型から他のデータ型への変換を指定します。 PostgreSQLは型キャストに2つの等価な構文を受け付けます。

```
CAST ( expression AS type )
expression::type
```

CAST構文はSQLに準拠したものです。 ::を使用する構文は、PostgreSQLで伝統的に使用されている方法です。

キャストが既知の型の評価式に適用された場合、それは実行時型変換を表します。 このキャストは、適切な型変換操作が定義されている場合のみ成功します。 [項4.1.2.7](#)で示すように、これと定数のキャストの使用との微妙な違いに注意してください。 修飾されていない文字列リテラルに対するキャストは、リテラル定数値の初期に割り当てられる型を表します。 ですから、これは（文字列リテラル定数の内容がそのデータ型の入力構文で受け付けられるのであれば）全ての型で成功します。

通常（例えばテーブル列への代入時など）、評価式が生成しなければならない型に曖昧さが無い場合、明示的な型キャストは省略することができます。 その場合、システムは自動的に型キャストを適用します。 しかし、自動キャストは、システムカタログに"暗黙的に適用しても問題なし"と示されている場合にのみ実行されます。 その他のキャストは明示的なキャスト構文で呼び出す必要があります。 この制限は、知らないうちに変換が実行されてしまうことを防ぐためのものです。

また、関数のような構文を使用して型キャストを指定することもできます。

```
typename ( expression )
```

しかし、これはその型の名前が関数の名前としても有効な場合にのみ動作します。 例えば、double precision はこの方式で使用できませんが、同等のfloat8は使用できます。 また、interval、time、timestampという名前は、構文が衝突するため、二重引用符で括った場合にのみこの方式で使用できます。 このように、この関数のようなキャスト構文は一貫性がなくなりがちです。 おそらくアプリケーションでは使用すべきではありません

注意: この関数のような構文は、実際には単なる関数呼び出しです。 2つの標準的なキャスト構文のうちの1つが実行時変換で使用されると、この構文は登録済みの関数を内部的に呼び出して変換を実行します。 慣習的に、これらの変換関数は自身の出力型と同じ名前を持ち、これにより、"関数のような構文"は背後にある変換用関数を直接呼び出す以上のことを行いません。 移植性を持つアプリケーションが依存すべきものでないことは明確です。 詳細については[CREATE CAST](#)を参照してください。

4.2.10. 照合順序式

COLLATE句は式の照合順序規則を上書きします。 適用するため次の様に式に加えます。

```
expr COLLATE collation
```

collationはおそらくスキーマ部を含む識別子です。 COLLATE句は演算子よりも優先度が低いため、必要に応じて括弧で囲います。

もし照合順序が何も指定されなければ、データベースシステムは式にある列から照合順序を取得します。 もし列に関する照合順序が式になければ、データベースのデフォルトの照合順序を使います。

例として、よく使われるCOLLATE句による並び替え順序の上書きを挙げます。 ORDER BY句を使う場合には、

```
SELECT a, b, c FROM tbl WHERE ... ORDER BY a COLLATE "C";
```

とします。 ロケール依存を伴った関数や演算子の呼び出しに対しての上書き方法例は

```
SELECT * FROM tbl WHERE a > 'foo' COLLATE "C";
```

とします。 後者の場合、COLLATE句が、処理対象と想定している入力演算子の引数に対して付与されることに注意してください。 演算子や関数の呼び出しのどの引数に対してCOLLATE句が付与されるかは問題ではありません。 演算子や関数により適用される照合順序は対象となる全ての引数を考慮して引き出され、そして明示的に指定されたCOLLATE句がその他の全ての引数に対しての照合順序を上書きするからです。 (しかし、複数の引数に対して一致しないCOLLATE句の付与はエラーとなります。 詳細は[項22.2](#)を参照してください)。 このため、前述の例と同じ結果を次の様にして取得することができます。

```
SELECT * FROM tbl WHERE a COLLATE "C" > 'foo';
```

ただし、これはエラーになります。 正しくは次の様にします。

```
SELECT * FROM tbl WHERE (a > 'foo') COLLATE "C";
```


なぜなら、照合順序を与えることができないboolean型となる>演算子の結果に対して。照合順序が適用されようとするからです。

4.2.11. スカラ副問い合わせ

スカラ副問い合わせは、正確に1行1列を返す、括弧内の通常のSELECT問い合わせです（問い合わせの記述方法については[第7章](#)を参照してください）。そのSELECT問い合わせは実行され、返される単一の値はその値の前後の評価式で使用されます。1行を超える行や1列を超える列がスカラ副問い合わせ用の問い合わせとして使用された場合はエラーになります（しかし、ある実行時に、副問い合わせが行を返さない場合はエラーとはなりません。そのスカラ結果はNULLとして扱われます）。副問い合わせは、その周りの問い合わせ内の値を参照することができます。その値は副問い合わせの評価時には定数として扱われます。副問い合わせに関する他の式については[項9.22](#)も参照してください。

例えば、以下は各州で最も人口の多い都市を検索します。

```
SELECT name, (SELECT max(pop) FROM cities WHERE cities.state = states.name)
FROM states;
```

4.2.12. 配列コンストラクタ

配列コンストラクタは、メンバー要素に対する値を用いて配列値を構築する式です。単純な配列コンストラクタの構成は、ARRAY キーワード、左大括弧[、（カンマで区切った）配列要素値用の式のリストで、最後に右大括弧]です。以下に例を示します。

```
SELECT ARRAY[1,2,3+4];
      array
-----
{1,2,7}
(1 row)
```

デフォルトで配列要素型は、メンバ式の型と同じで、UNIONやCASE構文と同じ規則を使用して決定されます（[項10.5](#)を参照してください）。これを明示的に配列コンストラクタを希望する型にキャストすることで書き換えることができます。例をあげます。

```
SELECT ARRAY[1,2,22.7]::integer[];
      array
-----
{1,2,23}
(1 row)
```

これはそれぞれの式を配列型に個別にキャストするのと同じ効果があります。キャストについてより多くは[項4.2.9](#)を参照してください。

多次元配列値は、配列コンストラクタを入れ子にすることで構築できます。内側のコンストラクタではARRAY キーワードは省略可能です。例えば、以下は同じ結果になります。

```
SELECT ARRAY[ARRAY[1,2], ARRAY[3,4]];
      array
-----
{{1,2},{3,4}}
(1 row)

SELECT ARRAY[[1,2],[3,4]];
      array
-----
{{1,2},{3,4}}
(1 row)
```

多次元配列は四角形配列でなければなりませんので、同一レベルの内部コンストラクタは同次元の副配列を生成しなければなりません。外部ARRAYコンストラクタに適用される全てのキャストは自動的に全ての内部コンストラクタに伝播します。

多次元配列コンストラクタの要素は、副ARRAY構文だけでなく、適切な種類の配列を生成するものをとることができます。以下に例を示します。

```
CREATE TABLE arr(f1 int[], f2 int[]);

INSERT INTO arr VALUES (ARRAY[[1,2],[3,4]], ARRAY[[5,6],[7,8]]);

SELECT ARRAY[f1, f2, '{{9,10},{11,12}}'::int[]] FROM arr;
      array
-----
{{{1,2},{3,4}},{5,6},{7,8}},{9,10},{11,12}}}
(1 row)
```

空配列を構築できますが、型を所有しない配列を持つことは不可能なので、空配列を望まれる型に明示的にキャストしなければ

なりません。例をあげます。

```
SELECT ARRAY[]::integer[];
      array
-----
{ }
(1 row)
```

また、副問い合わせの結果から配列を構成することも可能です。この形式の場合、配列コンストラクタは`ARRAY`キーワードの後に括弧（大括弧ではない）で括られた副問い合わせとして記述されます。以下に例を示します。

```
SELECT ARRAY(SELECT oid FROM pg_proc WHERE proname LIKE 'bytea%');
      array
-----
{2011,1954,1948,1952,1951,1244,1950,2005,1949,1953,2006,31,2412,2413}
(1 row)
```

副問い合わせは単一の列を返さなければなりません。その結果である一次元配列は、副問い合わせの出力列と一致する型を要素型とした、副問い合わせの結果内の各行を要素として持ちます。

`ARRAY`で構築された配列値の添字は、常に1から始まります。配列についての詳細は[項8.15](#)を参照してください。

4.2.13. 行コンストラクタ

行コンストラクタは、そのメンバフィールドに対する値を用いて行値（複合値とも呼ばれます）を構築する式です。行コンストラクタは、`ROW`キーワード、左括弧、行のフィールド値用の0個以上の式（カンマ区切り）、最後に右括弧からなります。以下に例を示します。

```
SELECT ROW(1,2.5,'this is a test');
```

`ROW`キーワードは、1つ以上の式がリスト内にある場合は省略することができます。

行コンストラクタには`rowvalue.*`構文を含めることができます。これは、`SELECT`リストの最上位レベルで`.*`構文が使用された時とまったく同様に、行値の要素の列挙に展開されます。たとえば、テーブル`t`が`f1`列と`f2`列を持つ場合、以下は同一です。

```
SELECT ROW(t.*, 42) FROM t;
SELECT ROW(t.f1, t.f2, 42) FROM t;
```

注意: PostgreSQL 8.2より前では、`.*`構文は展開されませんでした。`ROW(t.*, 42)`と記述すると、1つ目のフィールドに別の行値を持つ、2つのフィールドからなる行が作成されました。たいていの場合、新しい動作はより使いやすくなっています。入れ子状の行値という古い動作が必要であれば、`.*`を使用せずに、たとえば`ROW(t, 42)`といった、内部行値を記述してください。

デフォルトでは、`ROW`式により作成される値は匿名レコード型になります。必要に応じて、名前付きの複合型、テーブルの行型、もしくは`CREATE TYPE AS`で作成された複合型にキャストすることができます。明示的なキャストは曖昧性を防止するために必要となることもあります。以下に例を示します。

```
CREATE TABLE mytable(f1 int, f2 float, f3 text);

CREATE FUNCTION getf1(mytable) RETURNS int AS 'SELECT $1.f1' LANGUAGE SQL;

-- getf1() が1つしか存在しないためキャスト不要。
SELECT getf1(ROW(1,2.5,'this is a test'));
      getf1
-----
1
(1 row)

CREATE TYPE myrowtype AS (f1 int, f2 text, f3 numeric);

CREATE FUNCTION getf1(myrowtype) RETURNS int AS 'SELECT $1.f1' LANGUAGE SQL;

-- ここでは、どの関数を呼び出すのかを示すためにキャストが必要。
SELECT getf1(ROW(1,2.5,'this is a test'));
ERROR:  function getf1(record) is not unique

SELECT getf1(ROW(1,2.5,'this is a test')::mytable);
      getf1
-----
1
(1 row)

SELECT getf1(CAST(ROW(11,'this is a test',2.5) AS myrowtype));
      getf1
-----
11
(1 row)
```

行コンストラクタは、複合型のテーブル列に格納する複合型の値を構築するため、あるいは複合型のパラメータを受け付ける関数に渡すために使用することができます。 また、以下の例のように、2つの行値を比較することも、`IS NULL`もしくは`IS NOT NULL`で行を検査することも可能です。

```
SELECT ROW(1,2.5,'this is a test') = ROW(1, 3, 'not the same');
SELECT ROW(table.*) IS NULL FROM table;  -- すべてがNULLの行を検出します。
```

詳細は[項9.23](#)を参照してください。 行コンストラクタは、[項9.22](#)で説明した、副問い合わせと一緒に使用することもできます。

4.2.14. 式の評価規則

副式の評価の順序は定義されていません。 特に演算子や関数の入力は、必ずしも左から右などの決まった順序で評価されるわけではありません。

さらに、その式の一部を評価しただけで式の結果を判断できる場合には、他の副式がまったく評価されないこともあります。 例えば、

```
SELECT true OR somefunc();
```

では、（おそらく）`somefunc()` は呼び出されないでしょう。 以下の場合も同様です。

```
SELECT somefunc() OR true;
```

これは一部のプログラミング言語に見られる、ブーリアン演算子での左から右への"ショートサーキット"とは異なることに注意してください。

そのため、副次作用がある関数を複雑な式の一部として使用することは推奨されません。 特に、`WHERE` 句および`HAVING` 句で副次作用や評価順に依存するのは危険です。 これらの句は、実行計画を作成する過程で頻繁に再処理されるからです。 これらの句のブール式（`AND/OR/NOT`の組み合わせ）は、ブール代数の規則で許されるあらゆる方式で再編成することができます。

評価の順序を強制することが重要であれば、`CASE` 構文（[項9.17](#)を参照）を使用できます。 例えば、次の式は`WHERE` 句で0除算を避ける方法としては信頼性の低いものです。

```
SELECT ... WHERE x > 0 AND y/x > 1.5;
```

しかし、次のようにすれば安全です。

```
SELECT ... WHERE CASE WHEN x > 0 THEN y/x > 1.5 ELSE false END;
```

このような方法で使用する`CASE` 構文は最適化を妨げるものなので、必要な場合にのみ使用してください（特に、この例では、`y > 1.5*x`と代わりに記述することが問題を回避するより優れた方法です。）。

4.3. 関数呼び出し

PostgreSQLは位置と名前付け表記による名前付けパラメータを関数に持たせることが可能です。特に名前付け表記は、多数のパラメータを持つ関数においてパラメータと引数の関連をより明確・確実にするのに有用です。位置表記では、関数定義の際に定義されたものと同じ並び順の引数が、関数呼び出しに記述されることになります。名前付け表記では、引数と関数パラメータの名前を一致させることになり、引数はどのような並び順でも構いません。

どちらの表記でも、パラメータが関数定義時にデフォルト値を与えられていれば呼び出し時に記述される必要はありません。これは、名前付け表記では、パラメータ群の任意の組み合わせを省略することができるので、特殊なケースでは有用です。一方、位置表記のパラメータは右から左へ省略していくことしかできません。

PostgreSQLでは、名前付け表記と位置表記の混在表記もサポートしています。この場合、位置表記のパラメータが最初に記述され、その後に名前付け表記のパラメータが記述されることになります。

下記は3つの表記方法を使った関数定義の例です。

```
CREATE FUNCTION concat_lower_or_upper(a text, b text, uppercase boolean DEFAULT false)
RETURNS text
AS
$$
    SELECT CASE
        WHEN $3 THEN UPPER($1 || ' ' || $2)
        ELSE LOWER($1 || ' ' || $2)
    END;
$$
LANGUAGE SQL IMMUTABLE STRICT;
```

`concat_lower_or_upper`関数は、`a`と`b`の指定必須となる2つのパラメータを持ちます。加えて、`uppercase`というデフォルトが`false`となっているオプションなパラメータの一つを持ちます。`a`と`b`で入力された文字列が結合され、`uppercase`パラメータにより大文字か小文字に変換されます。他のこの関数定義についての詳細は、ここでは重要ではありません。(詳細は[第35章](#)を参照して下さい。)

4.3.1. 位置表記の使用

位置表記は、PostgreSQLの引数を関数に渡す伝統的な仕組みです。例を挙げます。

```
SELECT concat_lower_or_upper('Hello', 'World', true);
concat_lower_or_upper
-----
HELLO WORLD
(1 row)
```

すべての引数を順番通りに指定します。`uppercase`が`true`と指定されていますので、結果は大文字です。別の例を示します。

```
SELECT concat_lower_or_upper('Hello', 'World');
concat_lower_or_upper
-----
hello world
(1 row)
```

ここでは`uppercase`パラメータが省略されていますので、そのデフォルト値である`false`を受け取ることとなり、結果は小文字になります。位置表記では引数がデフォルト値を持つ限り右側から左の方向で、引数を省略することができます。

4.3.2. 名前付け表記の使用

名前付け表記では、各引数の名前は`:=`を使用し引数の表現と分けて指定されます。例を挙げます。

```
SELECT concat_lower_or_upper(a := 'Hello', b := 'World');
concat_lower_or_upper
-----
hello world
(1 row)
```

繰り返しになりますが、`uppercase`引数が省略されていますので、暗黙的に`false`に設定されます。名前付け表記の使用の利点の1つとして、引数を任意の順序で指定できる点があります。以下に例を示します。

```
SELECT concat_lower_or_upper(a := 'Hello', b := 'World', uppercase := true);
concat_lower_or_upper
-----
HELLO WORLD
```



```
(1 row)

SELECT concat_lower_or_upper(a := 'Hello', uppercase := true, b := 'World');
concat_lower_or_upper
-----
HELLO WORLD
(1 row)
```

4.3.3. 混在表記の利用

混在表記は名前付け表記と位置表記を組み合わせたものです。しかし既に述べたように、名前付けされた引数は位置づけされたパラメータの前に記述することはできません。例を挙げます。

```
SELECT concat_lower_or_upper('Hello', 'World', uppercase := true);
concat_lower_or_upper
-----
HELLO WORLD
(1 row)
```

上記の問い合わせでは、`a`と`b`が位置で指定され、`uppercase`は名前で指定されています。この例では文書化の目的以外ほとんど意味がありません。デフォルト値が割り当てられた多くのパラメータを持つ、もっと複雑な関数では、名前付けもしくは混在表記により記述量を大きく減らすことができ、かつ、エラーが紛れ込む可能性を抑えることができます。

[前のページ](#)
評価式

[ホーム](#)
[上に戻る](#)

[次のページ](#)
データ定義

第 5 章 データ定義

目次

- 5.1. [テーブルの基本](#)
- 5.2. [デフォルト値](#)
- 5.3. [制約](#)
 - 5.3.1. [検査制約](#)
 - 5.3.2. [非NULL制約](#)
 - 5.3.3. [一意性制約](#)
 - 5.3.4. [プライマリキー](#)
 - 5.3.5. [外部キー](#)
 - 5.3.6. [排他制約](#)
- 5.4. [システム列](#)
- 5.5. [テーブルの変更](#)
 - 5.5.1. [列の追加](#)
 - 5.5.2. [列の削除](#)
 - 5.5.3. [制約の追加](#)
 - 5.5.4. [制約の削除](#)
 - 5.5.5. [列のデフォルト値の変更](#)
 - 5.5.6. [列のデータ型の変更](#)
 - 5.5.7. [列名の変更](#)
 - 5.5.8. [テーブル名の変更](#)
- 5.6. [権限](#)
- 5.7. [スキーマ](#)
 - 5.7.1. [スキーマの作成](#)
 - 5.7.2. [publicスキーマ](#)
 - 5.7.3. [スキーマ検索パス](#)
 - 5.7.4. [スキーマおよび権限](#)
 - 5.7.5. [システムカタログスキーマ](#)
 - 5.7.6. [使用パターン](#)
 - 5.7.7. [移植性](#)
- 5.8. [継承](#)
- 5.9. [パーティショニング](#)
 - 5.9.1. [概要](#)
 - 5.9.2. [パーティショニングの実装](#)
 - 5.9.3. [パーティションの管理](#)
 - 5.9.4. [パーティショニングと制約による除外](#)
 - 5.9.5. [他のパーティショニング方法](#)
 - 5.9.6. [警告](#)
- 5.10. [外部データ](#)
- 5.11. [その他のデータベースオブジェクト](#)
- 5.12. [依存関係の追跡](#)

本章では、データを保持するためのデータベース構造を作成する方法を説明します。リレーショナルデータベースでは生データはテーブルに格納されます。したがって、本章ではテーブルの作成と変更の方法や、テーブルにどのようなデータを格納するかを制御するための機能について重点的に解説します。さらに、テーブルをスキーマに編成する方法、およびテーブルへの権限の割り当てについても説明します。そして最後に、継承、ビュー、関数、およびトリガなど、データの格納に影響する機能について簡単に説明します。

5.1. テーブルの基本

リレーショナルデータベースのテーブルは、紙に書く表によく似ています。テーブルは行と列からできています。列の数と順序は固定されており、それぞれの列に名前が付けられています。行の数は可変です。つまり行の数とは、その時点でどれだけのデータが格納されているのかを示すものです。SQLではテーブル内の行の順序は保証されません。テーブルを読み込むと、明示的に並び替えが要求されない限り、行は不特定な順序で返されます。これについては[第7章](#)を参照してください。さらに、SQLでは行に固有の識別子が割り当てられないので、テーブル内にまったく同一の行がいくつも存在することがあり得ます。これは、SQLの基礎をなす算術モデルの結果ですが、通常は好ましいことではありません。この問題の対処法については、本章で後述します。

それぞれの列にデータ型があります。データ型によって、列に割り当てられる値が制限されます。また、列に格納されているデータに意味が割り当てられ、データを計算に使用できるようになります。例えば、数値型と宣言された列は任意のテキスト文字列は受け付けません。そして、数値型の列に格納されているデータは算術計算に使用できます。これに対して、文字列型と宣言された列はほとんど全ての種類のデータを受け付けます。しかし、文字列の結合といった演算には使用できますが、算術計算には使用できません。

PostgreSQLには、様々なアプリケーションに対応した多数のデータ型の集合が組み込まれています。またユーザが独自のデータ型を定義することも可能です。組み込みデータ型のほとんどにはわかりやすい名前と意味が付けられているので、詳しい説明はここでは行わず、[第8章](#)で行います。よく使用されるデータ型としては、整数を表す `integer`、小数も表すことができる `numeric`、文字列を表す `text`、日付を表す `date`、時刻を表す `time`、そして日付と時間の両方を含む `timestamp` があります。

テーブルを作成するには、その名の通り `CREATE TABLE` コマンドを使用します。このコマンドで最低限指定する必要があるのは、新規テーブル名、列名、各列のデータ型です。例を示します。

```
CREATE TABLE my_first_table (  
    first_column text,  
    second_column integer  
);
```

これで2列からなる `my_first_table` という名前のテーブルが作成されます。最初の列の名前は `first_column` で、そのデータ型は `text` です。2番目の列の名前は `second_column` で、そのデータ型は `integer` です。テーブル名および列名は、[項4.1.1](#)で説明した識別子の構文に従います。通常、型名は識別子でもあります。例外もあります。列リストはカンマで区切り、括弧で囲むことに注意してください。

先ほどの例は、説明が目的であるため現実的ではありません。通常、テーブルおよび列の名前は、どのようなデータが格納されているかわかるような名前にします。以下に、より現実的な例を示します。

```
CREATE TABLE products (  
    product_no integer,  
    name text,  
    price numeric  
);
```

(`numeric` 型は小数を格納することができ、金額を扱う場合はこれが一般的です。)

ティップ: 相関するテーブルを数多く作成する場合は、テーブルと列の名前を一貫させるのが賢明です。例えば、テーブル名に単数形あるいは複数形どちらの名詞を使用するかという選択肢があります（これは論者によって好み分かれています）。

テーブルに含めることができる列の数には制限があります。制限は、列の型に応じて250～1600の間となります。しかし、これほど多くの列を使用することは稀ですし、そのような場合は設計に問題があることも多いのです。

必要のないテーブルができた場合は、`DROP TABLE` コマンドを使用してそのテーブルを削除できます。例を示します。

```
DROP TABLE my_first_table;  
DROP TABLE products;
```

存在しないテーブルを削除しようとすると、エラーになります。もっともテーブルが存在するかどうか関係なくスクリプト全体を動作させることができるように、テーブルを作成する前に、エラーメッセージを無視して無条件に削除操作を行うことは、SQLスクリプトファイルではよく行われることです。（この操作を行いたければ、エラーメッセージの出力を防ぐ `DROP TABLE IF EXISTS` という構文を使用することができます。しかし、これは標準SQLではありません。）

既に存在するテーブルを変更する方法については、本章で後述する[項5.5](#)を参照してください。

これまでに説明したツールを使用して、十分に機能するテーブルを作成できます。本章の残りでは、テーブル定義に機能を追加して、データの整合性、安全性、利便性を確実にする方法について述べていきます。この時点でテーブルにデータを入力したければ、本章の残りを後回しにして[第6章](#)に進んでも構いません。

5.2. デフォルト値

列にはデフォルトの値を割り当てることができます。新しく作成された行のいくつかの列に値が指定されていない場合、そうした空欄にはそれぞれの列のデフォルト値が入ります。データ操作コマンドを使用して、列を（どのような値かを把握する必要なく）デフォルト値に設定するように明示的に要求することもできます。（データ操作コマンドの詳細については[第6章](#)を参照してください。）

明示的に宣言されたデフォルト値がない場合は、デフォルト値はNULL値になります。NULL値は不明のデータを表すものとみなすことができるので、通常はこの方法で問題ありません。

テーブル定義では、デフォルト値は列データ型の後に列挙されています。例を示します。

```
CREATE TABLE products (  
    product_no integer,  
    name text,  
    price numeric DEFAULT 9.99  
);
```

デフォルト値を式にすることが可能で、それはデフォルト値が挿入される時はいつでも（テーブルが作成されたときではありません）評価されます。よくある例として、`timestamp`列が挿入時の時刻に設定されるように、その列はデフォルトの`CURRENT_TIMESTAMP`を持つことができます。もう1つの例としては、各行に"通番"を割り振る場合です。PostgreSQLでは、典型的に以下のように記述することにより生成されます。

```
CREATE TABLE products (  
    product_no integer DEFAULT nextval('products_product_no_seq'),  
    ...  
);
```

ここで、`nextval()` 関数が、シーケンスオブジェクトから連続した値を生成します（[項9.16](#)を参照してください）。特に省略した形として以下のようにも記述できます。

```
CREATE TABLE products (  
    product_no SERIAL,  
    ...  
);
```

省略形である`SERIAL`は[項8.1.4](#)で詳しく述べられています。

5.3. 制約

データ型は、テーブルに格納するデータの種類を限定するための方法です。しかし、多くのアプリケーションでは、型が提供する制約では精密さに欠けます。例えば、製品の価格が入る列には、おそらく正数のみを受け入れるようにする必要があります。しかし、正数のみを受け入れるという標準のデータ型はありません。また、他の列や行に関連して列データを制約したい場合もあります。例えば、製品の情報が入っているテーブルでは、1つの製品番号についての行が2行以上あってはなりません。

このような問題を解決するため、SQLでは列およびテーブルに対する制約を定義することができます。制約によってテーブル内のデータを自由に制御することができます。制約に違反するデータを列に格納しようとすると、エラーとなります。このことは、デフォルト値として定義された値を格納する場合にも適用されます。

5.3.1. 検査制約

検査制約は最も一般的な制約の種類です。これを使用して、特定の列の値が論理値の式を満たす（真の値）ように指定できます。例えば、製品価格を必ず正数にするには以下のようにします。

```
CREATE TABLE products (  
    product_no integer,  
    name text,  
    price numeric CHECK (price > 0)  
);
```

このように、制約の定義はデフォルト値の定義と同様に、データ型の後にきます。デフォルト値と制約は任意の順序で列挙できます。検査制約はCHECKキーワードの後に続く括弧で囲まれた式で構成されます。検査制約式には、制約される列を含む必要があります。そうしないと、制約はあまり意味のないものになります。

制約に個別に名前を付けることもできます。名前を付けることで、エラーメッセージがわかりやすくなりますし、変更したい制約を参照できるようになります。構文は以下の通りです。

```
CREATE TABLE products (  
    product_no integer,  
    name text,  
    price numeric CONSTRAINT positive_price CHECK (price > 0)  
);
```

上記のように、名前付き制約の指定はCONSTRAINTキーワードで始め、これに識別子、制約定義と続きます。（この方法で制約名を指定しない場合は、システムにより名前が付けられます。）

検査制約では複数の列を参照することもできます。例えば、通常価格と割引価格を格納する場合に、必ず割引価格が通常価格よりも低くなるようにしたいとします。

```
CREATE TABLE products (  
    product_no integer,  
    name text,  
    price numeric CHECK (price > 0),  
    discounted_price numeric CHECK (discounted_price > 0),  
    CHECK (price > discounted_price)  
);
```

最初の2つの制約は上で説明した通りです。3つ目の制約では新しい構文を使っています。これは特定の列に付加されるのではなく、カンマで区切られた列リスト内の別個の項目として現れます。列定義およびこれらの制約定義は、任意の順序で列挙することができます。

最初の2つの制約を列制約と言います。これに対し、3つ目の制約は列定義とは別個に書かれるので、テーブル制約と言います。列制約をテーブル制約として書くことはできますが、その逆はできる場合とできない場合があります。なぜなら列制約は、制約に関連付けられている列のみを参照するためです（PostgreSQLはこの規則を強制しません。しかし他のデータベースシステムと協調して稼働させたい場合はこの規則に従ってください）。上の例は、以下のように書くこともできます。

```
CREATE TABLE products (  
    product_no integer,  
    name text,  
    price numeric,  
    CHECK (price > 0),  
    discounted_price numeric,  
    CHECK (discounted_price > 0),  
    CHECK (price > discounted_price)  
);
```

あるいは、次のようにもできます。

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CHECK (price > 0),
    discounted_price numeric,
    CHECK (discounted_price > 0 AND price > discounted_price)
);
```

どのようにするかは好みの問題です。

列制約と同様に、テーブル制約に名前を割り当てることができます。

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    CHECK (price > 0),
    discounted_price numeric,
    CHECK (discounted_price > 0),
    CONSTRAINT valid_discount CHECK (price > discounted_price)
);
```

検査制約では、検査式が真またはNULL値と評価された場合に、条件が満たされることに注意して下さい。ほとんどの式は、演算項目に一つでもNULLがあればNULLと評価されるので、検査制約では制約対象の列にNULL値が入るのを防げません。列がNULL値を含まないようにするために、次節で説明する非NULL制約を使用することができます。

5.3.2. 非NULL制約

非NULL制約は単純に、列がNULL値を取らないことを指定します。構文の例は以下の通りです。

```
CREATE TABLE products (
    product_no integer NOT NULL,
    name text NOT NULL,
    price numeric
);
```

非NULL制約は常に列制約として記述されます。非NULL制約はCHECK (column_name IS NOT NULL) という検査制約と機能的には同等ですが、PostgreSQLでは、明示的に非NULL制約を作成する方がより効果的です。このように作成された非NULL制約に明示的な名前を付けられないのが欠点です。

もちろん、1つの列に複数の制約を適用することもできます。そのためには、次々と制約を書いていくだけです。

```
CREATE TABLE products (
    product_no integer NOT NULL,
    name text NOT NULL,
    price numeric NOT NULL CHECK (price > 0)
);
```

順序は関係ありません。書かれた順序と検査される順序は必ずしも同じではありません。

NOT NULL制約に対し、逆のパターンであるNULL制約があります。これは、列がNULLでなければならないということではありません。そのような制約は意味がありません。この制約は、列がNULLであってもよいというデフォルトの振舞いを選択するだけのものです。NULL制約は標準SQLには存在しませんので、移植予定のアプリケーションでは使用すべきではありません。（これは、PostgreSQLと他の一部のデータベースシステムとの互換性のために追加された機能に過ぎません。）もっとも、スク립トファイルでの制約の切り替えが簡単であるという理由でこの機能を歓迎するユーザもいます。例えば、最初に

```
CREATE TABLE products (
    product_no integer NULL,
    name text NULL,
    price numeric NULL
);
```

と書いてから、必要な場所にNOTキーワードを挿入することができます。

ティップ: ほとんどのデータベース設計において、列の大多数をNOT NULLとマークする必要があります。

5.3.3. 一意性制約

一意性制約によって、列あるいは列のグループに含まれるデータが、テーブル内の全ての行で一意であることを確実にします。列制約の場合の構文は以下の通りです。

```
CREATE TABLE products (
    product_no integer UNIQUE,
    name text,
    price numeric
);
```


また、テーブル制約の場合の構文は

```
CREATE TABLE products (  
    product_no integer,  
    name text,  
    price numeric,  
    UNIQUE (product_no)  
);
```

となります。

一意性制約が列のグループを参照する場合、各列はカンマで区切って列挙します。

```
CREATE TABLE example (  
    a integer,  
    b integer,  
    c integer,  
    UNIQUE (a, c)  
);
```

これは、指定された列の値の組み合わせがテーブル全体で一意であることを指定しています。しかし、列の片方が一意である必要はありません（通常一意ではありません）。

一意性制約には、いつものように名前を割り当てることもできます。

```
CREATE TABLE products (  
    product_no integer CONSTRAINT must_be_different UNIQUE,  
    name text,  
    price numeric  
);
```

一意性制約を追加すると、制約で使われる列または列のグループに対して一意的なbtreeインデックスが自動的に作られます。

一般に、制約の対象となる列について同じ値を持つ行が、テーブル内に1行を上回る場合は、一意性制約違反になります。しかし、この比較では2つのNULL値は等価とはみなされません。つまり、一意性制約があったとしても、制約対象の列の少なくとも1つにNULL値を持つ行を複数格納することができるということです。この振舞いは標準SQLに準拠していますが、この規則に従わないSQLデータベースがあることを聞いたことがあります。ですから、移植する予定のアプリケーションを開発する際には注意してください。

5.3.4. プライマリキー

技術的には、プライマリキー制約は単純に一意性制約と非NULL制約を組み合わせたものです。つまり、次の2つのテーブル定義は同じデータを受け入れます。

```
CREATE TABLE products (  
    product_no integer UNIQUE NOT NULL,  
    name text,  
    price numeric  
);
```

```
CREATE TABLE products (  
    product_no integer PRIMARY KEY,  
    name text,  
    price numeric  
);
```

プライマリキーでも複数の列を制約することができ、その構文は一意性制約に似ています。

```
CREATE TABLE example (  
    a integer,  
    b integer,  
    c integer,  
    PRIMARY KEY (a, c)  
);
```

プライマリキーは、列または列のグループを、テーブル内で行の一意の識別子として使用できることを示します（このことは、プライマリキーの定義からの直接の帰結です。一意性制約ではNULL値を除外しないため、一意の識別子が提供されないことに注意してください。）これは文書化、および、クライアントアプリケーションの両方の面で役に立ちます。例えば、行値の変更が可能なGUIアプリケーションが行を一意的に特定するためには、おそらくテーブルのプライマリキーを知る必要があります。

プライマリキーを追加すると、プライマリキーで使われる列または列のグループに対して一意的なbtreeインデックスが自動的に作られます。

1つのテーブルは最大1つのプライマリキーを持つことができます。（一意性制約および非NULL制約には個数の制限はありません）

ん。 機能的には同じものですが、プライマリキーとして識別される制約は1つのみです。） リレーショナルデータベース理論では、全てのテーブルにプライマリキーが1つ必要とされています。 この規則はPostgreSQLでは強制されませんが、たいいていの場合にはこれに従うことが推奨されます。

5.3.5. 外部キー

外部キー制約は、列（または列のグループ）の値が、他のテーブルの行の値と一致しなければならないことを指定します。 これによって関連する2つのテーブルの参照整合性が維持されます。

これまで何度か例に使用したproductsテーブルについて考えてみます。

```
CREATE TABLE products (  
    product_no integer PRIMARY KEY,  
    name text,  
    price numeric  
);
```

また、これらの製品に対する注文を格納するテーブルも作成済みだとしましょう。 この注文のordersテーブルには実際に存在する製品の注文のみを格納したいと思っています。 そこで、productsテーブルを参照するordersテーブルに外部キー制約を定義します。

```
CREATE TABLE orders (  
    order_id integer PRIMARY KEY,  
    product_no integer REFERENCES products (product_no),  
    quantity integer  
);
```

これで、productsテーブルに存在しない非NULLのproduct_no項目を使用して注文を作成することはできなくなります。

このような場合に、ordersテーブルのことを参照テーブル、productテーブルのことを被参照テーブルと呼びます。 同様に、参照列と被参照列もあります。

上記のコマンドは、次のように短縮することもできます。

```
CREATE TABLE orders (  
    order_id integer PRIMARY KEY,  
    product_no integer REFERENCES products,  
    quantity integer  
);
```

列リストがないため、被参照テーブルのプライマリキーが被参照列（複数可）として使用されます。

外部キーでも、列のグループを制約したり参照したりすることもできます。 これもまた、テーブル制約の形式で記述する必要があります。 以下は、説明のための非現実的な例です。

```
CREATE TABLE t1 (  
    a integer PRIMARY KEY,  
    b integer,  
    c integer,  
    FOREIGN KEY (b, c) REFERENCES other_table (c1, c2)  
);
```

もちろん、制約される列数および型は、被参照列の数および型と一致しなければなりません。

外部キーには、以下のようにして名前を割り当てることもできます。

テーブルには複数の外部キー制約を含めることができます。 このことはテーブル間の多対多関係を実装するために使用されます。 例えば、製品と注文に関するそれぞれのテーブルがある場合に、複数の製品にまたがる注文を可能にしたいとします（上の例の構造では不可能です）。 この場合、次のテーブル構造を使用できます。

```
CREATE TABLE products (  
    product_no integer PRIMARY KEY,  
    name text,  
    price numeric  
);  
  
CREATE TABLE orders (  
    order_id integer PRIMARY KEY,  
    shipping_address text,  
    ...  
);  
  
CREATE TABLE order_items (  
    product_no integer REFERENCES products,  
    order_id integer REFERENCES orders,  
    quantity integer,  
    PRIMARY KEY (product_no, order_id)
```

```
);
```

最後のテーブルで、プライマリキーと外部キーが重なっていることに注目してください。

外部キーが製品に関連付けられていない注文の作成を許可しないことは、既に説明した通りです。しかし、ある注文で参照していた製品が、注文後に削除されたらどうなるでしょう。SQLではこのような場合も扱うことができます。直感的に、いくつかのオプションが考えられます。

- 参照される製品の削除を許可しない
- 注文も一緒に削除する
- 他にもありますか？

具体例として、上の例の多対多関係に次のポリシーを実装してみましょう。（`order_items`によって）注文で参照されたままの製品を削除しようとしても、この操作を行えないようにします。注文が削除されると、注文項目も削除されます。

```
CREATE TABLE products (  
    product_no integer PRIMARY KEY,  
    name text,  
    price numeric  
);  
  
CREATE TABLE orders (  
    order_id integer PRIMARY KEY,  
    shipping_address text,  
    ...  
);  
  
CREATE TABLE order_items (  
    product_no integer REFERENCES products ON DELETE RESTRICT,  
    order_id integer REFERENCES orders ON DELETE CASCADE,  
    quantity integer,  
    PRIMARY KEY (product_no, order_id)  
);
```

削除の制限およびカスケードという2つは、最も一般的なオプションです。`RESTRICT`は、被参照行が削除されるのを防ぎます。`NO ACTION`は、制約が検査された時に参照行がまだ存在していた場合に、エラーとなることを意味しています。これは、何も指定しない場合のデフォルトの振舞いとなります（これらの本来の違いは、`NO ACTION`では検査をトランザクション中で後回しにすることができるのに対し、`RESTRICT`では後回しにできないということです）。`CASCADE`は被参照行が削除された時、それを参照する行（複数可）も同様に削除されなければならないことを指定します。他にも2つのオプションがあります。`SET NULL`と`SET DEFAULT`です。これらは、被参照行が削除された際に、参照列がそれぞれNULLかデフォルトに設定されるようになります。これらは制約を守ることを免除することではない、ということに注意してください。例えば、動作に`SET DEFAULT`を指定したとしても、デフォルト値が外部キー制約を満たさない場合には操作は失敗します。

`ON DELETE`に似たもので、被参照列が変更（更新）された時に呼び出される`ON UPDATE`もあります。これらが行えるアクションは同じです。

被参照テーブルからの行の`DELETE`や被参照行の`UPDATE`は、古い値と一致する行に対して参照テーブルのスキャンを要求しますので、参照行にインデックスを付けるのは大抵は良い考えです。これは常に必要という訳ではなく、また、インデックスの方法には多くの選択肢がありますので、外部キー制約の宣言では参照列のインデックスが自動的に作られるということはありません。

データの更新および削除について詳しくは、[第6章](#)を参照してください。

最後に、外部キーはプライマリキーまたは一意性制約のいずれかである列を参照している必要があります。外部キーが一意性制約を参照している場合、NULL値をどのように対応させるかに関する他の方法がいくつかあります。これらについては、リファレンス文書の[CREATE TABLE](#)に説明があります。

5.3.6. 排他制約

排他制約によって、2つの行に関して指定された列もしくは式を指定された演算子を利用して比較した場合に、少なくとも演算子の比較の1つが偽もしくはnullを返すことを確実にします。構文は以下の通りです。

```
CREATE TABLE circles (  
    c circle,  
    EXCLUDE USING gist (c WITH &&)  
);
```

詳細は[CREATE TABLE ... CONSTRAINT ... EXCLUDE](#)を参照して下さい。

排他制約を追加すると、制約宣言で指定された種類のインデックスが自動的に作られます。

5.4. システム列

全てのテーブルには、システムによって暗黙的に定義されたシステム列がいくつかあります。そのため、システム列の名前はユーザ定義列の名前として使うことはできません。（これらの制約は名前がキーワードであるかどうかとは関係ありません。つまり、名前を引用符で囲んでもこの制約を回避することはできません。）システム列については、あまり意識する必要はありません。これらが存在することを知っていれば十分です。

oid

行のオブジェクト識別子（オブジェクトID）です。この列はWITH OIDSを付けた場合と、その時にdefault with oidsが設定されていた場合に作成されます。この列の型はoid（列名と同じ）です。この型についての詳細は[項8.18](#)を参照してください。

tableoid

この行を含むテーブルのOIDです。この列は特に、継承階層からの選択問い合わせでは便利です（[項5.8](#)を参照してください）。この列がないと、どのテーブルからその行が来たのかわかりにくいからです。tableoidはテーブル名を得るためにpg_classのoid列に結合することができます。

xmin

この行バージョンの挿入トランザクションの識別情報（トランザクションID）です。（行のバージョンとは、行の個別の状態です。行が更新される度に、同一の論理的な行に対する新しいバージョンの行が作成されます。）

cmin

挿入トランザクション内の（0から始まる）コマンド識別子です。

xmax

削除トランザクションの識別情報（トランザクションID）です。削除されていない行ではゼロです。可視のバージョンの行でこの列が非ゼロの場合があります。これは通常、削除トランザクションがまだコミットされていないこと、または、削除の試行がロールバックされたことを意味しています。

cmax

削除トランザクション内のコマンド識別子、もしくはゼロです。

ctid

テーブル内における、行バージョンの物理的位置を表します。ctidは行バージョンを素早く見つけるために使うことができますが、行のctidは更新されたり、VACUUM FULLにより移動させられたりすると変わります。したがって、ctidは長期の行識別子としては使えません。論理行を識別するためには、OID、あるいはさらに良いのはユーザ定義の通番数を使うべきです。

OIDは32ビット数であり、クラスタ全体で1つのカウンタです。大規模、もしくは長期間使用するデータベースでは、カウンタが一周してしまう可能性があります。そのため、一意性を確保するための手順を踏んでいない限り、OIDが一意であると仮定してはなりません。もしテーブル内の行を同定する必要がある場合は、シーケンスジェネレータを使用することを強く推奨します。しかし、OIDはいくつかの注意を払うことによりシーケンスと同様に使用することができます。

- 一意性制約は、OIDが行を同定するために使用されるようなテーブルのOID列に作成されなくてははいけません。このような一意性制約（もしくは一意インデックス）が存在する場合は、システムは既存の行に一致するようなOIDを生成しません。（もちろん、これはテーブルの行数が 2^{32} （40億）より少ない場合に可能となります。性能を考慮すると、実際には行数はそれよりずっと少ない方がよいでしょう。）
- OIDは複数のテーブルをまたがって一意であると仮定してはなりません。データベース全体での識別子が必要な場合はtableoidと行OIDを組み合わせて使用してください。
- もちろん、問題にしているテーブルは、WITH OIDSを使用して作成されなくてははいけません。PostgreSQL 8.1からWITHOUT OIDSがデフォルトです。

トランザクション識別子も32ビット量です。長期間使用するデータベースでは、トランザクションIDが一周してしまう可能性があります。これは、適切な保守作業を行うことで、致命的な問題にはなりません。詳細は[第23章](#)を参照してください。しかし、長期（10億トランザクション以上）にわたってトランザクションIDの一意性に依存することは賢明ではありません。

コマンド識別子もまた、32ビット量です。このため、単一トランザクション内のコマンド数には 2^{32} （40億）個までという制限が発生します。実際、この制限は問題にはなりません。これはSQLコマンド数に対する制限であり、処理される行数に対す

る制限ではないことに注意してください。 またPostgreSQL 8.3から、データベースの内容を実際に変更するコマンドのみがコマンド識別子を消費します。

[前のページ](#)
制約

[ホーム](#)
[上に戻る](#)

[次のページ](#)
テーブルの変更

5.5. テーブルの変更

テーブルの作成後に間違いに気付いたり、あるいはアプリケーションの要件が変わったりした場合には、テーブルをいったん削除して再度作成することができます。しかし、テーブルにデータを入力済みの場合、あるいはそのテーブルが他のデータベースオブジェクト（例えば外部キー制約）によって参照されている場合、これは良い方法ではありません。そのため、PostgreSQL では既存のテーブルに変更を加えるための一連のコマンドが用意されています。テーブル内のデータを変更するという概念ではないことに注意してください。ここでは、テーブルの定義や構造を変更することに焦点を合わせます。

次のことができます。

- 列の追加
- 列の削除
- 制約の追加
- 制約の削除
- デフォルト値の変更
- 列のデータ型の変更
- 列名の変更
- テーブル名の変更

これらの操作は全て [ALTER TABLE](#) コマンド（本節の説明範囲を超えますので詳細はこちらを参照してください）を使用して行うことができます。

5.5.1. 列の追加

列を追加するには、次のようにコマンドを使用します。

```
ALTER TABLE products ADD COLUMN description text;
```

新しい列にはデフォルト値が初期値として入ります（`DEFAULT` 句を指定しない場合は `NULL` 値が入ります）。

次の構文を使用すると、列の制約も同時に定義することができます。

```
ALTER TABLE products ADD COLUMN description text CHECK (description <> '');
```

実際には `CREATE TABLE` 内の列の記述に使用されている全てのオプションが、ここで使用できます。ただしデフォルト値は与えられている制約を満足するものでなくてはならないことに注意してください。満足しない場合は `ADD` が失敗します。一方で、新規の列に正しく値を入れた後で制約を追加することができます（後述）。

ティップ: デフォルトを持つ列を追加するためには、（新しい列値を格納するために）テーブルの各行を更新しなければなりません。しかし、デフォルトが指定されていない場合は、PostgreSQL は物理的な更新を避けることが可能です。ですので、ほとんどがデフォルト以外の値を持つ列をテーブルに格納するつもりであれば、デフォルトを指定せずに列を追加し、`UPDATE` を使用して正しい値を挿入することをお勧めします。その後、後述のようにして設定したいデフォルト値を付与してください。

5.5.2. 列の削除

列を削除するには、次のようにコマンドを使用します。

```
ALTER TABLE products DROP COLUMN description;
```

列内にある、どんなデータであれ消去します。また列を含むテーブルの制約も消去されます。しかし、もし列が他のテーブルの外部キー制約として参照されている場合は、PostgreSQL は暗黙のうちに制約を消去しません。`CASCADE` を追加することにより列に依存する全てを消去することができます。

```
ALTER TABLE products DROP COLUMN description CASCADE;
```

この背後にある一般的な仕組みに関する説明については [項 5.12](#) を参照してください。

5.5.3. 制約の追加

制約を追加するには、テーブル制約の構文が使用されます。

```
ALTER TABLE products ADD CHECK (name <> '');
ALTER TABLE products ADD CONSTRAINT some_name UNIQUE (product no);
ALTER TABLE products ADD FOREIGN KEY (product_group_id) REFERENCES product_groups;
```

テーブル制約として記述できない非NULL制約を追加するには、次の構文を使用します。

```
ALTER TABLE products ALTER COLUMN product_no SET NOT NULL;
```

制約は即座に検査されますので、制約を追加する前にテーブル内のデータがこれに従っている必要があります。

5.5.4. 制約の削除

制約を削除するには、その制約名を知る必要があります。自分で名前を付けた場合は簡単です。しかし、自分で名前を付けていない場合はシステム生成の名前が割り当てられているので、それを探さなくてはなりません。それにはpsqlの`\d tablename`コマンドを使用すると便利です。他のインタフェースにもテーブルの詳細を調べる方法があるかもしれませんが。コマンドは以下の通りです。

```
ALTER TABLE products DROP CONSTRAINT some_name;
```

(自動生成された`some_name`といった制約名を扱う場合は、有効な識別子となるように二重引用符で括弧を付ける必要があることを忘れないでください。)

列の削除に関して、何かが依存している制約を削除する場合には`CASCADE`を付ける必要があります。例として、参照されている列に付いている一意またはプライマリキー制約に依存している外部キー制約を削除する場合です。

これは、非NULL制約以外の全ての制約型に適用できます。非NULL制約を削除するには、次のようにします。

```
ALTER TABLE products ALTER COLUMN product_no DROP NOT NULL;
```

(非NULL制約には名前がないことを想起してください。)

5.5.5. 列のデフォルト値の変更

列に新しいデフォルトを設定するには、以下のようなコマンドを使用します。

```
ALTER TABLE products ALTER COLUMN price SET DEFAULT 7.77;
```

これはテーブル内の既存の行には何も影響を与えないことに注意してください。これは将来の`INSERT`コマンドのために単純にデフォルトを変えるだけです。

デフォルト値を削除するには次のようにします。

```
ALTER TABLE products ALTER COLUMN price DROP DEFAULT;
```

これは、デフォルトをNULLに設定することと同等です。そのため、定義されていないデフォルト値を削除してもエラーにはなりません。なぜなら NULL値が暗黙的にデフォルトとなっているからです。

5.5.6. 列のデータ型の変更

列を異なるデータ型に変換するには以下のようなコマンドを使用してください。

```
ALTER TABLE products ALTER COLUMN price TYPE numeric(10,2);
```

これは、その列の既存の項目が新しい型に暗黙的キャストにより変換できる場合にのみ成功します。より複雑な変換が必要な場合、古い値から新しい値をどのように計算するかを指定する`USING`句を付けることができます。

PostgreSQLは、(もしあれば)列のデフォルト値を新しい型に、同時に、その列に関連する全ての制約も新しい型に変換しようとします。しかし、こうした変換は失敗するかもしれませんし、予想を超えた結果になってしまうかもしれません。型を変更する前にその列に関する制約を全て削除し、後で適切に変更した制約を付け直すことが最善な場合がよくあります。

5.5.7. 列名の変更

列名を変更するには、次のようにします。

```
ALTER TABLE products RENAME COLUMN product_no TO product_number;
```

5.5.8. テーブル名の変更

テーブル名を変更するには、次のようにします。

```
ALTER TABLE products RENAME TO items;
```

[前のページ](#)
システム列

[ホーム](#)
[上に戻る](#)

[次のページ](#)
権限

5.6. 権限

オブジェクトが作成されると、所有者が割り当てられます。通常、所有者は作成する文を実行したロールです。ほとんどの種類のオブジェクトについて、初期状態では所有者(またはスーパーユーザ)だけがそのオブジェクトを使用できます。他のユーザがこのオブジェクトを使用するには、権限が付与されていなければなりません。

権限にはいくつかの種類があります。すなわちSELECT、

INSERT、UPDATE、DELETE、TRUNCATE、REFERENCES、TRIGGER、CREATE、CONNECT、TEMPORARY、EXECUTE、USAGEです。特定のオブジェクトに適用する特権は、オブジェクトの型(テーブル、関数など)により変わります。PostgreSQLがサポートする様々な権限の詳細については[GRANT](#)リファレンスページを参照してください。以降の節および章でもこれらの権限の使用方法についての説明があります。

オブジェクトの変更や削除の権限は常に所有者のみに与えられるものです。

[ALTER TABLE](#)のような適切な種類のALTER コマンドにより、あるオブジェクトに新しい所有者を割り当てることができます。スーパーユーザはいつでも所有者を変更できます。通常のロールは、対象オブジェクトの現在の所有者(または所有者ロールのメンバー)であり、かつ新しい所有者ロールのメンバーである場合に限り、所有者を変更できます。

権限を割り当てるには、GRANT コマンドを使用します。例えば、joeという既存のユーザとaccountsという既存のテーブルがある場合、このテーブルを更新する権限を付与するには以下のようにします。

```
GRANT UPDATE ON accounts TO joe;
```

特定の権限名を指定する代わりにALLを指定すると、その種類のオブジェクトに関連する全ての権限が付与されます。

システム内の全てのユーザに権限を付与するには、特別な"ユーザ"名であるPUBLICを使用することができます。また、"グループ"ロールを使用すれば、データベース内に多くのユーザが存在する場合に権限の管理が簡単になります。詳細は[第20章](#)を参照してください。

権限を取り消す(revoke)には、それに相応しい名前のREVOKE コマンドを使用します。

```
REVOKE ALL ON accounts FROM PUBLIC;
```

オブジェクト所有者の特別の権限(DROP、GRANT、REVOKEを行う権限など)は、所有者であることを前提とした権限なので、付与したり取り消したりすることはできません。しかしオブジェクト所有者は、テーブルを他のユーザ同様に自分自身に対しても読み取り専用にしたい時などに、自分の通常の権限を取り消すことができます。

普通はオブジェクトの所有者(またはスーパーユーザ)は、オブジェクトにおける権限の付与や剥奪ができます。しかし"with grant option"を付けることで、権限を与えられたユーザが、所有者と同様に他のユーザに権限を付与することが可能になります。もし後になってグラントオプションが剥奪されると、剥奪されたユーザから(直接もしくは権限付与の連鎖により)権限を与えられていたユーザはすべて、その権限が剥奪されます。詳細は、[GRANT](#)と[REVOKE](#)を参照してください。

5.7. スキーマ

PostgreSQL データベースクラスタには、複数の名前付きデータベースが含まれます。ユーザおよびユーザのグループはクラスタ全体で共有されますが、他のデータは複数のデータベース間で共有されません。サーバに接続しているクライアントは、単一のデータベース、つまり接続要求で指定したデータベース内のデータにしかアクセスできません。

注意: クラスタのユーザは、クラスタ内の全てのデータベースへのアクセス権限を持っているとは限りません。ユーザ名を共有するということは、例えば `joe` という同じユーザ名を持つ異なるユーザが同じクラスタ内の2つのデータベースに存在することはできないということです。しかし、`joe` が一部のデータベースにのみアクセスできるようにシステムを構成することはできます。

データベースには、複数の名前付きスキーマが含まれ、スキーマにはテーブルが含まれます。スキーマには、データ型、関数および演算子などの他の名前付きオブジェクトも含まれます。同じオブジェクト名を異なるスキーマで使用しても矛盾は起こりません。例えば、`schema1` と `myschema` の両方のスキーマに `mytable` というテーブルが含まれていても構いません。スキーマはデータベースとは異なり厳格に分離されていないので、ユーザは、権限さえ持っていれば接続しているデータベース内のどのスキーマのオブジェクトにでもアクセスすることができます。

スキーマの使用が好まれる理由はいくつかあります。

- 1つのデータベースを多数のユーザが互いに干渉することなく使用できるようにするため。
- 管理しやすくなるよう、データベースオブジェクトを論理グループに編成するため。
- サードパーティのアプリケーションを別々のスキーマに入れることにより、他のオブジェクトの名前と競合しないようにするため。

スキーマは、入れ子にできないという点を除き、オペレーティングシステムのディレクトリと似ています。

5.7.1. スキーマの作成

スキーマを作成するには、[CREATE SCHEMA](#) コマンドを使用します。スキーマに自由に名前を付けます。例を示します。

```
CREATE SCHEMA myschema;
```

スキーマ内にオブジェクトを作成したりこれにアクセスするには、スキーマ名とテーブル名をドットで区切った修飾名を書きます。

```
schema.table
```

この方法は、後の章で説明するテーブル変更コマンドやデータアクセスコマンドなど、テーブル名を必要とする場合すべてに使用できます。（簡単に、テーブルについてのみ述べます。しかし型や関数といった名前付きのオブジェクトの全種類について同様の考え方が適用できます。）

実際には、より一般的な以下の構文

```
database.schema.table
```

を使用することもできますが、現在ではこの構文は形式上標準SQLに準拠するためにのみ存在しています。記述されるデータベース名は、接続しているデータベースと同じ名前であればなりません。

ですから、新しいスキーマにテーブルを作成するには次のようにします。

```
CREATE TABLE myschema.mytable (  
    ...  
);
```

空のスキーマ（全てのオブジェクトが削除されたスキーマ）を削除するには次のようにします。

```
DROP SCHEMA myschema;
```

オブジェクトを含むスキーマを削除する場合には次のようにします。

```
DROP SCHEMA myschema CASCADE;
```

この背後にある一般的な機構についての詳細は[項5.12](#)を参照してください。

他のユーザが所有するスキーマを作成したい場合があります（これは他のユーザの活動を明確に定義された名前空間内に制限す

る方法の1つです)。 そのための構文は次の通りです。

```
CREATE SCHEMA schemaname AUTHORIZATION username;
```

スキーマ名は省略することもでき、その場合スキーマ名はユーザ名と同じになります。 この構文の便利な使用方は[項5.7.6](#)に記載されています。

`pg_`で始まるスキーマ名は、システム上の使用のため予約されており、ユーザが作成することはできません。

5.7.2. publicスキーマ

これまでの節ではスキーマ名を指定せずにテーブルを作成してきました。 デフォルトでは、このようなテーブル（および他のオブジェクト）は自動的に"public"という名前のスキーマに入れられます。 新しいデータベースには全てこのようなスキーマが含まれています。 そのため、以下の2つの構文は同等です。

```
CREATE TABLE products ( ... );
```

および

```
CREATE TABLE public.products ( ... );
```

5.7.3. スキーマ検索パス

修飾名を書くのは手間がかかりますし、どちらにしても、アプリケーションに特定のスキーマ名を書き込まない方が良いことも多いのです。 そのため、テーブルは多くの場合、テーブル名しか持たない非修飾名として参照されます。 システムは、検索するスキーマのリストである検索パスに従って、どのテーブルを指しているのかを判別します。 検索パスで最初に一致したテーブルが、該当テーブルだと解釈されます。 検索パス内に一致するテーブルがないと、データベースの他のスキーマ内に一致するテーブルがある場合でもエラーが報告されます。

検索パスの最初に列挙されているスキーマは、「現在のスキーマ」と呼ばれます。 現在のスキーマは、検索される最初のスキーマであると同時に、スキーマ名を指定せずに`CREATE TABLE` コマンドでテーブルを作成した場合に新しいテーブルが作成されるスキーマでもあります。

現行の検索パスを示すには次のコマンドを使用します。

```
SHOW search_path;
```

デフォルトの設定では次のように返されます。

```
search_path
-----
"$user",public
```

最初の要素は、現行ユーザと同じ名前のスキーマを検索することを指定しています。 そのようなスキーマが存在していない場合、この項目は無視されます。 2番目の要素は、先ほど説明したpublicスキーマを参照しています。

検索パス内で最初に存在するスキーマが、新規オブジェクトが作成されるデフォルトの場所になります。 これが、デフォルトでオブジェクトがpublicスキーマに作成される理由です。 オブジェクトがスキーマ修飾なしで別の文脈で参照される場合（テーブル変更、データ変更、あるいは問い合わせコマンドなど）、一致するオブジェクトが見つかるまで検索パス内で探索されます。 そのためデフォルト構成では、非修飾のアクセスはpublicスキーマしか参照できません。

新しいスキーマをパスに追加するには次のようにします。

```
SET search_path TO myschema,public;
```

（`$user`はまだ必要ないので、ここでは省略しています。） そして、次のようにしてスキーマ修飾なしでテーブルにアクセスします。

```
DROP TABLE mytable;
```

また、`myschema`はパス内の最初の要素なので、新しいオブジェクトはデフォルトでここに作成されます。

以下のように書くこともできます。

```
SET search_path TO myschema;
```

このようにすると、今後は修飾名なしでpublicスキーマにアクセスすることができなくなります。 publicスキーマはデフォルトで存在するという以外に特別な意味はありません。 他のスキーマと同様に削除することもできます。

スキーマ検索パスに操作する他の方法については[項9.25](#)を参照してください。

検索パスはデータ型名、関数名、演算子名についても、テーブル名の場合と同じように機能します。データ型および関数の名前は、テーブル名とまったく同じように修飾することができます。式で修飾演算子名を書く場合には、特別な決まりがあります。それは以下の通りです。

```
OPERATOR (schema.operator)
```

この規則は構文が曖昧になることを防ぐためのものです。以下に例を示します。

```
SELECT 3 OPERATOR(pg_catalog.+) 4;
```

実際の場面ではこのような見づらい構文を書かなくて済むように、演算子についても検索パスが使用されています。

5.7.4. スキーマおよび権限

ユーザは、デフォルトでは所有していないスキーマのオブジェクトにアクセスすることはできません。アクセスするためには、そのスキーマの所有者からスキーマの `USAGE` 権限を付与してもらわなければなりません。そのスキーマ内のオブジェクトに対して操作を行うには、そのオブジェクトに応じて、さらに追加の権限が必要となる場合があります。

他のユーザのスキーマ内でオブジェクトを作成することも可能です。それには、スキーマ上で `CREATE` 権限が付与されていなければなりません。デフォルトでは、`public` スキーマに関しては全てのユーザが `CREATE` と `USAGE` 権限を持っていることに注意してください。つまり、全てのユーザは、そのユーザが接続できる任意のデータベース上の `public` スキーマにオブジェクトを作成できるということです。これが好ましくない場合は、以下のように権限を取り消すことができます。

```
REVOKE CREATE ON SCHEMA public FROM PUBLIC;
```

(最初の "public" はスキーマです。2番目の "public" は "全てのユーザ" を意味します。最初の public は識別子で、2番目の public はキーワードなので、それぞれ小文字、大文字を使用しています。[項4.1.1](#)のガイドラインを思い出してください。)

5.7.5. システムカタログスキーマ

各データベースには、`public` およびユーザ作成のスキーマの他に `pg_catalog` スキーマが含まれています。このスキーマにはシステムテーブルと全ての組み込みデータ型、関数および演算子が含まれています。`pg_catalog` は常に検索パスに含まれています。パスに明示的にリストされていない場合は、パスのスキーマを検索する前に暗黙的に検索されます。これにより組み込みの名前が常に検索されることが保証されます。しかし、ユーザ定義の名前で組み込みの名前を上書きする場合は、`pg_catalog` を明示的にパスの最後に置くことができます。

PostgreSQL 7.3より前のバージョンでは、`pg_` で始まるテーブル名は予約されていました。しかし現在では変更され、システムスキーマ以外のスキーマにも `pg_` で始まる名前を付けられるようになりました。ただし、このような名前は使用しないのが得策と言えます。今後のバージョンでユーザのテーブルと同じ名前のシステムカタログが定義され、競合する事態を避けるためです。(その結果、デフォルトの検索パスでは、ユーザのテーブル名への非修飾の参照はシステムカタログとして解決されることになります。) システムカタログは今後も `pg_` で始まる規則に従うので、ユーザが `pg_` という接頭辞を使わない限り、非修飾のユーザ定義テーブル名がシステムカタログと競合することはありません。

5.7.6. 使用パターン

スキーマは様々な方法でデータの編成に使用できます。デフォルト構成で簡単にサポートできるお勧めの使用パターンがいくつかあります。

- スキーマを作成しない場合は、全てのユーザが暗黙に `public` スキーマにアクセスします。これはスキーマがまったく使用できない状況と同じです。この構成は主に、データベースで作業するのが1人または2、3人しかいない場合に推奨されます。またこの構成では、スキーマを認識しない状況からの移行を容易に行えます。
- それぞれのユーザに、ユーザ名と同じ名前のスキーマを作成することができます。デフォルトの検索パスが、ユーザ名として解決される `$user` で始まることを思い出してください。つまり、各ユーザが個別のスキーマを持っていると、デフォルトでそれぞれのスキーマにアクセスすることになります。

この設定を使用する場合は、`public` スキーマへのアクセス権を取り消して（またはスキーマを削除して）、ユーザが完全に自分のスキーマにしかアクセスできないようにすることもできます。

- 共有アプリケーション（共有で使うテーブル、サードパーティ提供の追加関数など）をインストールする際には、それぞれ別のスキーマに入れるようにしてください。また他のユーザがこれらにアクセスできるように適切な権限を付与することを忘れないようにしてください。それによって他のユーザは、追加のオブジェクトをスキーマ名で修飾することによって参照したり、希望するならば、スキーマをそれぞれの検索パスに追加したりすることができます。

5.7.7. 移植性

標準SQLでは、1つのスキーマ内の異なるユーザが所有するオブジェクトという概念は存在しません。それどころか、実装によっては所有者と異なる名前のスキーマを作成することが許可されていない場合もあります。実際、標準で規定されている基本スキーマサポートのみを実装しているデータベースシステムでは、スキーマという概念とユーザという概念はほとんど同じなのです。そのため、修飾名とは`username.tablename`のことであると思っているユーザはたくさんいます。PostgreSQLにおいても、ユーザごとに1つのスキーマを作成すると、このようになります。

また、標準SQLには、`public`スキーマという概念也没有。標準に最大限従うためには、`public`スキーマは使用すべきではありません（おそらく削除した方が良いでしょう）。

もちろん、スキーマをまったく実装していなかったり、または、データベース間アクセスを（場合によっては制限付きで）許可することによって名前空間の使用をサポートしているSQLデータベースもあります。このようなシステムで作業する必要がある場合は、スキーマをまったく使わないようにすることで最大限の移植性を実現できます。

[前のページ](#)
権限

[ホーム](#)
[上に戻る](#)

[次のページ](#)
継承

5.8. 継承

PostgreSQLは、データベース設計者にとって便利なテーブルの継承を実装しています。（SQL:1999以降は型の継承を定義していますが、ここで述べられている継承とは多くの点で異なります。）

まず例から始めましょう。市（cities）のデータモデルを作成しようとしていると仮定してください。それぞれの州にはたくさんの市がありますが、州都（capitals）は1つのみです。特定の州から州都を素早く検索したいとします。これは、2つのテーブルを作成することにより実現できます。1つは州都のテーブルで、もう1つは州都ではないテーブルです。しかし、州都であるか否かに関わらない市に対するデータを問い合わせたときには何が起るのでしょうか？ 継承はこの問題を解決できます。citiesから継承されるcapitalsテーブルを定義するのです。

```
CREATE TABLE cities (
    name          text,
    population     float,
    altitude       int      -- in feet
);

CREATE TABLE capitals (
    state         char(2)
) INHERITS (cities);
```

この場合、capitalsテーブルは、その親テーブルであるcitiesテーブルの列をすべて継承します。州都は1つの余分な列stateを持ち、州を表現します。

PostgreSQLでは、1つのテーブルは、0以上のテーブルから継承することが可能です。また、問い合わせはテーブルのすべての行、またはテーブルのすべての行と継承されたテーブルのすべての行のいずれかを参照できます。後者がデフォルトの動作になります。例えば次の問い合わせは、500フィートより上に位置している州都を含んだ、すべての市の名前を検索します。

```
SELECT name, altitude
FROM cities
WHERE altitude > 500;
```

PostgreSQLチュートリアルからのサンプルデータ（[項2.1](#)を参照してください）に対して、この問い合わせは、以下の結果を出力します。

name	altitude
Las Vegas	2174
Mariposa	1953
Madison	845

一方、次の問い合わせは、州都ではなく500フィートより上に位置しているすべての市を検索します。

```
SELECT name, altitude
FROM ONLY cities
WHERE altitude > 500;
```

name	altitude
Las Vegas	2174
Mariposa	1953

ここでONLYキーワードは、問い合わせがcitiesテーブルのみを対象にしcities以下の継承の階層にあるテーブルは対象としないことを意味します。これまで議論したコマンドの多く—SELECT、UPDATEそしてDELETE—がONLYキーワードをサポートしています。

また、明示的に子孫テーブルが含まれていることを示すために、テーブル名の後ろに*を書くこともできます：

```
SELECT name, altitude
FROM cities*
WHERE altitude > 500;
```

*の指定は、その動作がデフォルトである（あなたが[sql_inheritance](#)構成オプションの設定を変更していない限り）ため必須ではありません。しかしながら、*を指定することは、追加のテーブルが検索されることを強調するのに有効でしょう。

場合によっては、ある特定の行がどのテーブルからきたものか知りたいということがあるかもしれません。それぞれのテーブルにはtableoidという、元になったテーブルを示すシステム列があります。

```
SELECT c.tableoid, c.name, c.altitude
FROM cities c
WHERE c.altitude > 500;
```


出力は以下の通りです。

tableoid	name	altitude
139793	Las Vegas	2174
139793	Mariposa	1953
139798	Madison	845

(この例をそのまま実行しても、おそらく異なる数値OIDが得られるでしょう。) `pg_class`と結合することで、テーブルの実際の名前が分かります。

```
SELECT p.relname, c.name, c.altitude
FROM cities c, pg_class p
WHERE c.altitude > 500 AND c.tableoid = p.oid;
```

出力は以下の通りです。

relname	name	altitude
cities	Las Vegas	2174
cities	Mariposa	1953
capitals	Madison	845

継承はINSERTまたはCOPYにより、継承の階層にある他のテーブルにデータを自動的に伝播することはありません。この例では、次のINSERT文は失敗します。

```
INSERT INTO cities (name, population, altitude, state)
VALUES ('New York', NULL, NULL, 'NY');
```

データが、どうにかしてcapitalsテーブルまで到達できればいいのですが、そのようにはなりません。INSERTは、いつも指定されたテーブルに対してデータを挿入します。ルール(詳細は第37章を参照してください)を使用して挿入を中継できる場合があります。しかし、ルールを使用しても上記のような場合には適用できません。なぜなら、citiesテーブルにはstate列は含んでいませんし、ルールが適用される前にコマンドは拒否されてしまうからです。

親テーブル上の検査制約と非NULL制約はその子テーブルに自動的に継承されます。他の種類の制約(一意性制約、プライマリキー、外部キー制約)は継承されません。

テーブルは1つ以上の親テーブルから継承可能です。この場合、テーブルは親テーブルで定義された列の和になります。子テーブルで宣言された列は、これらの列に追加されることになります。もし親テーブルに同じ名前の列がある場合、もしくは、親テーブルと子テーブルに同じ名前の列がある場合は、列が"統合"されて子テーブルではただ1つの列となります。統合されるには列は同じデータ型を持っている必要があります。異なるデータ型の場合にはエラーとなります。統合された列は、元となったテーブルで列定義が持つすべての検査制約のコピーを持ちます。そして、列定義のいずれかが非NULL制約を持てば、非NULLという印が付きます。

テーブル継承は、通常、CREATE TABLE文のINHERITS句を使用して、子テーブルを作成する時に確立します。他にも、互換性を持つ方法で定義済みのテーブルに新しく親子関係を付けることも可能です。これにはALTER TABLEのINHERIT形式を使用します。このためには、新しい子テーブルは親テーブルと同じ名前の列を持ち、その列の型は同じデータ型でなければなりません。また、親テーブルと同じ名前、同じ式の検査制約を持っていなければなりません。ALTER TABLEのNO INHERIT形式を使用して、同様に継承関係を子テーブルから取り除くことも可能です。このような継承関係の動的追加、動的削除は、継承関係をテーブル分割(項5.9を参照)に使用している場合に有用です。

後で子テーブルとする予定の、互換性を持つテーブルを簡単に作成する方法の1つは、CREATE TABLEでLIKE句を使用することです。これは、元としたテーブルと同じ列を持つテーブルを新しく作成します。新しい子テーブルが必ず親テーブルと一致する制約を持ち、互換性があるものとみなされるように、元となるテーブルでCHECK制約が存在する場合は、LIKEにINCLUDING CONSTRAINTSオプションを指定すべきです。

子テーブルが存在する場合親テーブルを削除することはできません。また、子テーブルでは、親テーブルから継承した列、または検査制約を削除することも変更することもできません。テーブルとそのすべての子テーブルを削除したければ、CASCADEオプションを付けて親テーブルを削除することが簡単な方法です。

ALTER TABLEは、列データ定義と検査制約の変更を継承の階層にあるテーブルに伝えます。繰り返しになりますが、他のテーブルに依存する列の削除はCASCADEオプションを使用したときのみ可能となります。ALTER TABLEは、重複列の統合時に適用される規則とCREATE TABLE時に適用される拒否の規則に従います。

テーブルアクセス権限がどのように扱われるかに注意して下さい。親テーブルへの問い合わせでは、それ以上アクセス権限を検査することなく自動的に子テーブルのデータにアクセスできます。これによりデータが親テーブルに(も)あるように見えることが保たれます。しかし、子テーブルへの直接のアクセスは自動的に許可されることはなく、追加の権限の付与が要求されるでしょう。

5.8.1. 警告

すべてのSQLコマンドが継承階層に対して動作できるとは限らないことに注意してください。データの検索、データの変更、スキーマの変更のために使用されるコマンド（例えばSELECT、UPDATE、DELETE、ALTER TABLEのほとんどの構文です。しかしINSERTやALTER TABLE ... RENAMEは除きます。）は通常、デフォルトで子テーブルを含み、また、それを除外するためのONLY記法をサポートします。データベース保守およびチューニング（例えばREINDEX、VACUUM）を行うコマンドは通常、個々の物理テーブルに対してのみ動作し、継承階層に対する再帰をサポートしません。個々のコマンドのそれぞれの動作はそのマニュアルページ（[参照I, SQLコマンド](#)）に記載されています。

継承機能の厳しい制限として、（一意性制約を含む）インデックス、および外部キーは、そのテーブルのみに適用され、それを継承した子テーブルには適用されないことがあります。これは外部キーの参照側、被参照側でも同様に適用されません。したがって、上の例では

- もし、cities.nameをUNIQUEまたはPRIMARY KEYと宣言しても、citiesテーブルの行と重複した行をcapitalsテーブル内に持つことを禁止することにはなりません。さらに、これらの重複した行はデフォルトでcitiesテーブルへの問い合わせで現れるでしょう。事実として、capitalsテーブルはデフォルトで一意性制約を持っていませんし、同一の名前の複数の行を持つことがあります。capitalsテーブルに一意性制約を追加できますが、これはcitiesテーブルと比較して重複を禁止することにはなりません。
- 同じように、もしcities.name REFERENCES他のテーブルとしても、この制約は自動的にcapitalsに引き継がれるわけではありません。この場合はcapitalsテーブルに同一のREFERENCES制約を手動で追加しなければいけません。
- 他のテーブルの列REFERENCES cities(name)を指定することは、他のテーブルが市の名前を持つことを許可しますが、州都の名前を持つことを許可しません。この場合は次善策はありません。

これらの機能の不足は、今後のリリースでおそらく改善されるでしょう。しかし当面、継承がアプリケーションにとって有用であるかどうかを判断する際に十分注意してください。

推奨しない設定: 7.1より前のリリースのPostgreSQLでは、問い合わせにおいて子テーブルを含まないことがデフォルトでした。これはエラーとなりやすく、また標準SQLにも抵触しています。7.1より前の古い動作を行うようにするには、[sql_inheritance](#)設定オプションを無効にしてください。

[前のページ](#)
スキーマ

[ホーム](#)
[上に戻る](#)

[次のページ](#)
パーティショニング

5.9. パーティショニング

PostgreSQLは基本的なテーブルのパーティショニング（分割）をサポートしています。この節では、データベース設計において、なぜそしてどのようにしてパーティショニングを実装するのかを解説します。

5.9.1. 概要

パーティショニングは論理的に1つの大きなテーブルを、物理的に小さなパーティションに分けることを指します。パーティショニングはいくつかの利点があります。

- 特定の状況において、問い合わせのパフォーマンスが劇的に改善されます。具体的には、アクセスが多いテーブル内の行を単一パーティションまたは一部のパーティション内に収めることです。パーティショニングはインデックスの先頭の列を置き換え、インデックスサイズが減少します。このため、インデックス内のよく使用される部分がメモリ内により収まりやすくなります。
- 問い合わせや更新が単一パーティションの大部分にアクセスする場合、インデックススキャンとテーブル全体に渡って散乱したランダムアクセス読み取りの代わりに、そのパーティションのシーケンシャルスキャンとすることができ、性能は改善します。
- その指定がパーティション設計に合うように計画されていれば、一括でのロードや削除が、パーティションの追加や削除によってなされる可能性があります。ALTER TABLE NO INHERITやDROP TABLEは両方とも一括操作よりもずっと高速です。また、これらのコマンドは大量のDELETEによって発生するVACUUMのオーバーヘッドを完全に防ぎます。
- めったに使用されないデータは、安価で遅い記憶メディアに移行できます。

この利点は、テーブルのサイズがとても大きくなる場合に価値が出てきます。テーブルのパーティショニングによる利点はアプリケーションに依存しますが、経験的にテーブルのサイズがデータベースサーバの物理メモリを超えるかどうかということがポイントになります。

現状ではPostgreSQLは、パーティショニングをテーブルの継承によりサポートしています。それぞれのパーティションは1つの親テーブルの子テーブルとして作成されなくてはなりません。親テーブル自身は通常、空のテーブルとなり、全体のデータを代表するために存在します。パーティショニングを設定する前に、継承（[項5.8](#)を参照してください）について詳しく知っておく必要があります。

パーティショニングについて次の種類がPostgreSQLに実装されています。

範囲分割

テーブルは、キーとなる列もしくは列のセットにより定義される"範囲"に分割されます。異なるパーティションに割り当てられた値の範囲は重なることはありません。例えば、日付の範囲により分割されたり、特定のビジネスオブジェクトの識別子の範囲により分割されたりします。

リスト分割

キー値がそれぞれのパーティションに現れるような明示的なリストにより、テーブルが分割されます。

5.9.2. パーティショニングの実装

テーブルのパーティショニングを実装するには、以下を行ってください。

- 1 11 すべてのパーティションが継承することになる、"マスタ"テーブルを作成してください。

このテーブルはデータを格納しません。このテーブルにはすべてのパーティションに対して適用されるつもりでなければ検査制約は定義しないでください。同様にインデックスや一意制約を定義することも意味がありません。

- 1 11 マスタテーブルから継承された、いくつかの"子"テーブルを作成します。通常、これらの子テーブルはマスタから継承された列以外には列を追加しないようにします。

子テーブルは、あらゆる点でPostgreSQLの普通のテーブルですが、子テーブルをパーティションとして参照することになります。

- 1 11 それぞれのパーティションでのキー値を定義するために、分割されたテーブルにテーブル制約を追加してください。

典型的な例は、

```
CHECK ( x = 1 )
CHECK ( county IN ( 'Oxfordshire', 'Buckinghamshire', 'Warwickshire' ))
CHECK ( outletID >= 100 AND outletID < 200 )
```

制約が、異なるパーティションにおいて許可されているキー値の間で重なりが無いことを保証してください。よくある間違いは範囲制約を次のように設定してしまうことです。

```
CHECK ( outletID BETWEEN 100 AND 200 )
CHECK ( outletID BETWEEN 200 AND 300 )
```

キー値の200がどちらのパーティションに属するのが明確になっていないため、これは間違いになります。

範囲分割とリスト分割の間に構文の違いは無いことに注意してください。これらの字句は記述上のものだけです。

- 1 11 それぞれのパーティションにおいて、他のインデックスと同様にキーとなる列（列の集合）にインデックスを作成してください。（キーのインデックスは必ずしも必要ではありませんが、たいいていの場合に役立ちます。もしキー値が一意であることを意図するのであればいつでも、一意もしくは主キー制約をそれぞれのパーティションに作成してください。）
- 1 11 また、マスタテーブルに挿入されたデータを適当なパーティションに中継するためにトリガもしくはルールを定義してください。
- 1 11 [constraint exclusion](#)設定パラメータが`postgresql.conf`内で無効になっていないことを確認してください。無効であると、問い合わせは思ったとおりに最適化されません。

例えば、大規模なアイスクリーム会社のデータベースを構築すると仮定してください。会社は、それぞれの地方のアイスクリームの売上と同様に毎日の最高気温を計測しています。概念的に、次のようなテーブルが必要になります。

```
CREATE TABLE measurement (
    city_id          int not null,
    logdate          date not null,
    peaktemp         int,
    unitsales        int
);
```

ほとんどの問い合わせが先週、先月もしくは半年前のデータを問い合わせるものであることが分かっています。その理由は、このテーブルが経営に関するオンラインのレポートを作成することに主に使用されるからです。必要な過去のデータ量を減らすために、過去3ヶ月のデータのみを保存することにします。月の始めに過去のデータを削除します。

このような場合、measurementテーブルに対する異なる要求をすべて満たすようにパーティショニングを利用できます。上記で述べた方法で、パーティショニングを次のように設定します。

- 1 11 マスタテーブルは、上記で宣言されたmeasurementテーブルです。

- 1 11 次にそれぞれの月に対して1つのパーティションを作成します。

```
CREATE TABLE measurement_y2006m02 ( ) INHERITS (measurement);
CREATE TABLE measurement_y2006m03 ( ) INHERITS (measurement);
...
CREATE TABLE measurement_y2007m11 ( ) INHERITS (measurement);
CREATE TABLE measurement_y2007m12 ( ) INHERITS (measurement);
CREATE TABLE measurement_y2008m01 ( ) INHERITS (measurement);
```

それぞれのパーティションは、完結したテーブルですがmeasurementテーブルからの定義を継承しています。

これは古いデータの削除という問題を解決します。毎月、最も古い子テーブルをDROP TABLE し新規の月に対しては子テーブルを作成するだけでいいのです。

- 1 11 重なりの無いテーブル制約を付ける必要があります。上記のようにパーティションを単に作成するよりも、以下のようにすべきでしょう。

```
CREATE TABLE measurement_y2006m02 (
    CHECK ( logdate >= DATE '2006-02-01' AND logdate < DATE '2006-03-01' )
) INHERITS (measurement);
CREATE TABLE measurement_y2006m03 (
    CHECK ( logdate >= DATE '2006-03-01' AND logdate < DATE '2006-04-01' )
) INHERITS (measurement);
...
CREATE TABLE measurement_y2007m11 (
    CHECK ( logdate >= DATE '2007-11-01' AND logdate < DATE '2007-12-01' )
) INHERITS (measurement);
CREATE TABLE measurement_y2007m12 (
    CHECK ( logdate >= DATE '2007-12-01' AND logdate < DATE '2008-01-01' )
) INHERITS (measurement);
CREATE TABLE measurement_y2008m01 (
    CHECK ( logdate >= DATE '2008-01-01' AND logdate < DATE '2008-02-01' )
```

```
) INHERITS (measurement);
```

1 11 キーとなる列にインデックスが必要になるでしょう。

```
CREATE INDEX measurement_y2006m02_logdate ON measurement_y2006m02 (logdate);
CREATE INDEX measurement_y2006m03_logdate ON measurement_y2006m03 (logdate);
...
CREATE INDEX measurement_y2007m11_logdate ON measurement_y2007m11 (logdate);
CREATE INDEX measurement_y2007m12_logdate ON measurement_y2007m12 (logdate);
CREATE INDEX measurement_y2008m01_logdate ON measurement_y2008m01 (logdate);
```

今回は、これ以上のインデックスを付けないことにします。

1 11 アプリケーションでINSERT INTO measurement ...を実行したら、適切なパーティションテーブルにデータが中継されるようにしたいとします。そのような場合は、適切なトリガ関数をマスタテーブルに設定することにより可能となります。もしデータが最新のパーティションにのみ追加される場合は、非常に簡単なトリガ関数を使用できます。

```
CREATE OR REPLACE FUNCTION measurement_insert_trigger()
RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO measurement_y2008m01 VALUES (NEW.*);
    RETURN NULL;
END;
$$
LANGUAGE plpgsql;
```

関数を作成した後で、このトリガ関数を呼ぶトリガを作成します。

```
CREATE TRIGGER insert_measurement_trigger
BEFORE INSERT ON measurement
FOR EACH ROW EXECUTE PROCEDURE measurement_insert_trigger();
```

毎月、トリガ関数を再定義しなくてははいけません。それにより現在のパーティションをいつも指すこととなります。しかし、トリガ定義は更新する必要はありません。

データを挿入したら、サーバが行を追加すべきパーティションを自動的に決定するようにしたいかもしれません。これは以下のようなもっと複雑なトリガ関数を作成することにより可能です。

```
CREATE OR REPLACE FUNCTION measurement_insert_trigger()
RETURNS TRIGGER AS $$
BEGIN
    IF ( NEW.logdate >= DATE '2006-02-01' AND
        NEW.logdate < DATE '2006-03-01' ) THEN
        INSERT INTO measurement_y2006m02 VALUES (NEW.*);
    ELSIF ( NEW.logdate >= DATE '2006-03-01' AND
        NEW.logdate < DATE '2006-04-01' ) THEN
        INSERT INTO measurement_y2006m03 VALUES (NEW.*);
    ...
    ELSIF ( NEW.logdate >= DATE '2008-01-01' AND
        NEW.logdate < DATE '2008-02-01' ) THEN
        INSERT INTO measurement_y2008m01 VALUES (NEW.*);
    ELSE
        RAISE EXCEPTION 'Date out of range. Fix the measurement_insert_trigger() function!';
    END IF;
    RETURN NULL;
END;
$$
LANGUAGE plpgsql;
```

トリガ定義は前と同じです。それぞれのIFテストをパーティションのCHECK制約と正確に一致させなければならないことに注意してください。

この関数は単一月の場合より複雑になりますが、頻繁に更新する必要はありません。なぜなら条件分岐を前もって追加しておくことが可能だからです。

注意: 実際には、ほとんどの挿入が一番新しいパーティションに入る場合は、そのパーティションを最初に検査することが最善です。簡単に、この例でのほかの部分と同じ順番でトリガのテストを示しました。

以上のように、複雑なパーティション化の計画はたくさんのDDLが必要となります。上記の例では、毎月新しいパーティションを作成することになりますが、必要となるDDLを自動的に生成するスクリプトを書くのが賢明です。

5.9.3. パーティションの管理

通常、初期定義でテーブルを静的なままにするつもりではない場合、一群のパーティションを作ることになります。古いデータのパーティションの削除や新規データ向けの定期的な新規パーティションの追加という要求はよくあります。こうしたそうでなければ苦痛を伴う作業を、大量のデータを物理的に移動させずに、パーティション構造を操作することでほとんど瞬間的に行うことができます。パーティショニングの最も重要な利点の一つはまさにこの点です。

古いデータを削除する最も簡単な方法は、単に不要となったパーティションを削除することです。

```
DROP TABLE measurement_y2006m02;
```

レコードごとに削除する必要がありませんので、これは高速に何百万ものレコードを削除することができます。

この他、よく使用される方法は、テーブル自体へのアクセス権限をそのまま残したまま、パーティション付けされたテーブルからパーティションを削除することです。

```
ALTER TABLE measurement_y2006m02 NO INHERIT measurement;
```

これにより、削除前にデータ操作をさらに行うことができます。例えば、COPY、pg_dumpなどのツールを使用してデータをバックアップすることはよくあります。また、データをより小さな書式に集約したり、他のデータ操作を行ったり、報告を作成したりすることもよくあります。

同様に、新しいデータを扱うために新しいパーティションを追加することもできます。上で元々のパーティションを作成した時と同じように、パーティション付けテーブルに空のパーティションを作成することができます。

```
CREATE TABLE measurement_y2008m02 (
    CHECK ( logdate >= DATE '2008-02-01' AND logdate < DATE '2008-03-01' )
) INHERITS (measurement);
```

この他の方法として、パーティション構造の外部で新しいテーブルを作成し、後で適切にパーティションとすることが便利な場合もあります。これにより、パーティション付けしたテーブルに追加する前に、データをロードし、検査、変換することができます。

```
CREATE TABLE measurement_y2008m02
    (LIKE measurement INCLUDING DEFAULTS INCLUDING CONSTRAINTS);
ALTER TABLE measurement_y2008m02 ADD CONSTRAINT y2008m02
    CHECK ( logdate >= DATE '2008-02-01' AND logdate < DATE '2008-03-01' );
\copy measurement_y2008m02 from 'measurement_y2008m02'
-- 何らかの準備操作を行うことができます。
ALTER TABLE measurement_y2008m02 INHERIT measurement;
```

5.9.4. パーティショニングと制約による除外

制約による除外は、上記の方法で定義された分割されたテーブルに対するパフォーマンスを向上させる問い合わせの最適化技術です。例えば、

```
SET constraint_exclusion = on;
SELECT count(*) FROM measurement WHERE logdate >= DATE '2008-01-01';
```

制約による除外が有効になっているとき、プランナはそれぞれのパーティションの制約を検証し、パーティションが問い合わせのWHEREに一致する行を含んでいないためにスキャンされる必要が無いことを証明しようとします。プランナはこれを証明すると、問い合わせ計画からそのパーティションを除外します。

constraint_exclusionを有効とした時の計画と無効にした時の計画の違いを明らかにするため、EXPLAINコマンドを使用できます。この種類のテーブル設定に対する典型的な最適化されない計画は以下のようになります。

```
SET constraint_exclusion = off;
EXPLAIN SELECT count(*) FROM measurement WHERE logdate >= DATE '2008-01-01';

               QUERY PLAN
-----
Aggregate  (cost=158.66..158.68 rows=1 width=0)
-> Append  (cost=0.00..151.88 rows=2715 width=0)
    -> Seq Scan on measurement  (cost=0.00..30.38 rows=543 width=0)
        Filter: (logdate >= '2008-01-01'::date)
    -> Seq Scan on measurement_y2006m02 measurement  (cost=0.00..30.38 rows=543 width=0)
        Filter: (logdate >= '2008-01-01'::date)
    -> Seq Scan on measurement_y2006m03 measurement  (cost=0.00..30.38 rows=543 width=0)
        Filter: (logdate >= '2008-01-01'::date)
    ...
    -> Seq Scan on measurement_y2007m12 measurement  (cost=0.00..30.38 rows=543 width=0)
        Filter: (logdate >= '2008-01-01'::date)
    -> Seq Scan on measurement_y2008m01 measurement  (cost=0.00..30.38 rows=543 width=0)
        Filter: (logdate >= '2008-01-01'::date)
```

一部のパーティション、もしくはすべてのパーティションで、テーブル全体に対するシーケンシャルスキャンではなく、インデックススキャンが使用される可能性があります。しかしここでのポイントは、この問い合わせに対する回答のために古いパーティションをスキャンする必要はまったく無いということです。制約による除外を有効にしたとき、大幅により安価な同じ回答を返す計画を得ることができます。

```
SET constraint_exclusion = on;
EXPLAIN SELECT count(*) FROM measurement WHERE logdate >= DATE '2008-01-01';

               QUERY PLAN
-----
Aggregate  (cost=0.00..0.00 rows=1 width=0)
-> Seq Scan on measurement  (cost=0.00..0.00 rows=1 width=0)
    Filter: (logdate >= '2008-01-01'::date)
```



```
-----
Aggregate  (cost=63.47..63.48 rows=1 width=0)
-> Append  (cost=0.00..60.75 rows=1086 width=0)
      -> Seq Scan on measurement  (cost=0.00..30.38 rows=543 width=0)
            Filter: (logdate >= '2008-01-01'::date)
      -> Seq Scan on measurement_y2008m01 measurement  (cost=0.00..30.38 rows=543 width=0)
            Filter: (logdate >= '2008-01-01'::date)
```

制約による除外はCHECK制約のみで動作し、インデックスの有無では動作しないことに注意してください。よってキー列のインデックスを定義することは必要ではありません。あるパーティションでインデックスが必要かどうかは、パーティションをスキャンする問い合わせがパーティションの大部分もしくは小さな部分をスキャンするのにかによります。前者ではなく後者において、インデックスは役立ちます。

実のところ、[constraint exclusion](#)のデフォルト（かつ推奨）の設定は、onでもoffでもなく、partitionという中間の設定です。これによりこの技法は、パーティショニングされたテーブルに対して動作することになる問い合わせのみに適用されるようになります。on設定により、プランナはすべての問い合わせでCHECK制約を検証するようになります。これは単純な問い合わせでも行われ、プラスになりそうではありません。

5.9.5. 他のパーティショニング方法

適当なパーティションテーブルにリダイレクトする別の方法は、トリガの代わりにルールをマスターテーブル上に作成することです。例えば

```
CREATE RULE measurement_insert_y2006m02 AS
ON INSERT TO measurement WHERE
    ( logdate >= DATE '2006-02-01' AND logdate < DATE '2006-03-01' )
DO INSTEAD
    INSERT INTO measurement_y2006m02 VALUES (NEW.*);
...
CREATE RULE measurement_insert_y2008m01 AS
ON INSERT TO measurement WHERE
    ( logdate >= DATE '2008-01-01' AND logdate < DATE '2008-02-01' )
DO INSTEAD
    INSERT INTO measurement_y2008m01 VALUES (NEW.*);
```

ルールはトリガよりもかなりオーバーヘッドがありますが、行ごとよりもむしろ問い合わせごとのオーバーヘッドとなります。よってこの方法は、一括挿入といった場合に有効となります。しかし多くの場合、トリガを使用する方が、良い性能となります。

COPYはルールを無視することに注意してください。もしデータを挿入するのにCOPYを使用したい場合は、マスターテーブルより正しいパーティションテーブルにコピーすることが必要となるでしょう。COPYはトリガを起動します。よってトリガを使用する方法をとれば、普通にCOPYを使用することができます。

ルールを使用する方法のもうひとつの欠点は、一連のルールが挿入日付を扱わないときにエラーを強制する簡単な方法がないことです。データは代わりにマスターテーブルに暗黙のうちに挿入されます。

パーティショニングは、テーブルの継承の代わりにUNION ALLビューを使用することによっても使用できます。例えば、

```
CREATE VIEW measurement AS
    SELECT * FROM measurement_y2006m02
UNION ALL SELECT * FROM measurement_y2006m03
...
UNION ALL SELECT * FROM measurement_y2007m11
UNION ALL SELECT * FROM measurement_y2007m12
UNION ALL SELECT * FROM measurement_y2008m01;
```

しかしビューを再作成することで、データセットの個別のパーティションを追加したり削除したりする余分な手順が増えます。実際には、この方法は継承を使用する方法と比較してお勧めする点はほとんどありません。

5.9.6. 警告

以下の警告がパーティション付けテーブルに対して適用されます。

- すべてのCHECK制約が相互に排他であるかどうか自動で確認する方法はありません。パーティションを生成し、関連付けられたオブジェクトを作成もしくは修正するコードを作成する方が、それらをいちいち手書きするよりも安全です。
- ここで示すスキーマは、行のパーティションキー列が変更しない、もしくは少なくとも他のパーティションに移動することが要求されるほどの変更がないことを仮定しています。別のパーティションに移るようなUPDATEは、CHECK制約により失敗するでしょう。このような場合を扱うには、適切な更新トリガをパーティションテーブルに設定します。しかしこれにより管理構造がさらに複雑になります。
- もし手動のVACUUMもしくはANALYZEコマンドを使用している場合は、それぞれのパーティションで個別に実行することを

忘れないでください。 以下のようなコマンドは

```
ANALYZE measurement;
```

マスタテーブルのみ処理することになります。

以下の警告が制約による除外に適用されます。

- 制約による除外は問い合わせのWHERE句が定数(または外部から供給されたパラメータ)を含んでいたときにのみ動作します。例えば、CURRENT_TIMESTAMPのような非immutable関数に対する比較は、関数の結果値がどのパーティションに該当するかを実行時にプランナが知ることが出来ないため、最適化できません。
- パーティション制約を簡単にしておいてください。そうしないとプランナは、パーティションを使う必要がないことを立証できません。 前述の例で示したとおり、リスト分割のために簡単な等号条件を使用してください。また範囲分割のために簡単な範囲テストを使用してください。 手っ取り早い良い方法は、パーティショニングの制約がパーティション列とB-treeインデックス作成可能な演算子を用いた定数の比較のみを含んでいることです。
- マスタテーブルのすべてのパーティションのすべての制約は、制約による除外で試験されます。 よってパーティションの数が多くなれば問い合わせ計画の時間がかなり増加します。 これらの技術を使用したパーティショニングは、おそらく100個までのパーティションでうまく動作します。 何千ものパーティションを使用することは避けてください。

[前のページ](#)
継承

[ホーム](#)
[上に戻る](#)

[次のページ](#)
外部データ

5.10. 外部データ

PostgreSQLはSQL/MED仕様を部分的に実装しており、PostgreSQLの外部にあるデータに対して標準的なSQLクエリでアクセスできます。このようなデータは外部データと呼ばれます。(この用語をデータベース内の制約である外部キーと混同しないように気をつけてください。)

外部データは外部データラッパの手助けによりアクセスされます。外部データラッパは外部データソースと通信できるライブラリであり、接続確立やデータ取得といった詳細を隠蔽します。`contrib`モジュールとして、サーバにある単純なデータファイルを読み込む外部データラッパが利用できます。その他の種類の外部データラッパは外部製品として見つかるでしょう。既存の外部データラッパがあなたの用途に合わない場合、独自のラッパを書くことができます。[第50章](#)を参照してください。

外部データにアクセスするには、特定の外部データラッパが使用するオプションの組み合わせで外部データソースへの接続方法を定義する外部サーバオブジェクトを作成する必要があります。その後、外部データの構造を定義する外部テーブルを少なくともひとつ作成する必要があります。外部テーブルは通常のテーブルと同様にクエリの中で使用できますが、外部テーブルはPostgreSQLサーバには格納領域を持ちません。外部テーブルが使われるたびに、PostgreSQLは外部データラッパに外部ソースからデータを取得するように依頼します。

外部データへのアクセスは外部データソースでの認証を必要とする場合があります。この情報は、現在のPostgreSQLロールに基づいて追加のオプションを提供することができるユーザマッピングによって提供することができます。

現在、外部テーブルは読み取り専用です。この制限は将来のリリースで緩和されるかもしれません。

5.11. その他のデータベースオブジェクト

テーブルにはデータが保持されていますので、リレーショナルデータベース構造ではテーブルが中心オブジェクトとなります。しかし、データベースにはテーブルの他にもオブジェクトが存在します。様々なオブジェクトを作成して、データの使用および管理をより効果的に行うことができます。本章ではこれらのオブジェクトについては説明しませんが、どのようなものがあるかをここに列挙します。

- ビュー
- 関数および演算子
- データ型およびドメイン
- トリガおよび書き換えルール

これらのトピックに関する詳細な情報は[パート V](#)にあります。

5.12. 依存関係の追跡

外部キー制約や、ビュー、トリガ、関数などを使ったテーブルが多数含まれるような複雑なデータベース構造を作成すると、ユーザはそれらのオブジェクト間の暗黙的な依存関係のネットワークも作成していることになります。例えば、外部キー制約を持つテーブルは、参照するテーブルに依存しています。

データベース構造全体の整合性を保つため、PostgreSQLは、他のオブジェクトと依存関係にあるオブジェクトの削除を許可しません。例えば、[項5.3.5](#)で作成したproductsテーブルを削除しようとしても、ordersテーブルがこのテーブルに依存しているので、以下のようなエラーメッセージが現れます。

```
DROP TABLE products;  
  
NOTICE:  constraint orders_product_no_fkey on table orders depends on table products  
ERROR:  cannot drop table products because other objects depend on it  
HINT:   Use DROP ... CASCADE to drop the dependent objects too.
```

エラーメッセージには役に立つヒントが含まれています。以下のようにすると、依存する全てのオブジェクトを1つずつ削除する手間を省けます。

```
DROP TABLE products CASCADE;
```

これで全ての依存オブジェクトが削除されます。この場合、ordersテーブルは削除されずに外部キー制約のみが削除されます（`DROP ... CASCADE`が何を行うかを知りたい場合は、`CASCADE`を指定せずに`DROP`を実行して`NOTICE`メッセージを読んでください）。

PostgreSQLでは、全ての削除用コマンドに`CASCADE`を指定することができます。もちろん、どのような依存関係が存在するかは、オブジェクトの種類によって異なります。また、`CASCADE`ではなく`RESTRICT`と記述すると、他のオブジェクトが依存しているオブジェクトの削除を禁止するというデフォルトの振舞いを指定することもできます。

注意: 標準SQLでは、`RESTRICT`または`CASCADE`のいずれかを指定する必要があります。実際にこの決まり通りのデータベースシステムはありませんが、デフォルトが`RESTRICT`であるか、`CASCADE`であるかは、システムによって異なります。

注意: PostgreSQLの7.3より前のバージョンにおける外部キー制約の依存関係とシリアル列の依存関係は、アップグレード処理において維持も作成もされません。その他の種類の依存関係は全て、7.3より前のデータベースからのアップグレード中に適切に作成されます。

第 6 章 データ操作

目次

6.1. [データの挿入](#)

6.2. [データの更新](#)

6.3. [データの削除](#)

前章では、データを保持するためのテーブルやその他の構造の作成方法について説明しました。いよいよ、テーブルにデータを挿入してみましょう。本章では、テーブルのデータの挿入、更新、削除の方法について説明します。そして本章以降において、ずっと見つけれなかったデータをデータベースから抽出する方法について説明します。

[前のページ](#)

依存関係の追跡

[ホーム](#)[上に戻る](#)[次のページ](#)

データの挿入

6.1. データの挿入

テーブルは、作成時にはデータを含んでいません。データベースを利用価値のあるものにするには、まずデータを挿入する必要があります。理論的には、データは一度に1行ずつ挿入されます。もちろんユーザは複数行に挿入することもできますが、1行未満を挿入することはできません。列の値が一部しかわかっていない場合でも、1行全体を作成しなければなりません。

新規の行を作成するには、[INSERT](#) コマンドを使用します。このコマンドは、テーブル名と列の値を必要とします。例えば、[第5章](#)の products テーブルの例で考えてみましょう。

```
CREATE TABLE products (  
    product_no integer,  
    name text,  
    price numeric  
);
```

この場合、行を挿入するためのコマンドは以下のようになります。

```
INSERT INTO products VALUES (1, 'Cheese', 9.99);
```

データ値は、テーブルで列が表示される順序に従ってカンマで区切って列挙します。通常、データ値はリテラル（定数）ですが、スカラー式も使用できます。

上記の構文には、テーブル内の列の順序を知っていなければならないという難点があります。これを避けるには、列を明示的に列挙する方法があります。例えば、以下の2つのどちらのコマンドでも上記のコマンドと同等の効果が得られます。

```
INSERT INTO products (product_no, name, price) VALUES (1, 'Cheese', 9.99);  
INSERT INTO products (name, price, product_no) VALUES ('Cheese', 9.99, 1);
```

多くのユーザは常に列名を列挙する方法が優れていると考えています。

値がわからない列については、省略することができます。省略した列には、デフォルト値が挿入されます。以下に例を示します。

```
INSERT INTO products (product_no, name) VALUES (1, 'Cheese');  
INSERT INTO products VALUES (1, 'Cheese');
```

後者は PostgreSQL の拡張機能です。これによって、列には左から順に指定されただけの値が挿入され、残りにはデフォルト値が挿入されます。

明確にするため、列ごと、あるいは行全体についてデフォルト値を明示的に要求することもできます。

```
INSERT INTO products (product_no, name, price) VALUES (1, 'Cheese', DEFAULT);  
INSERT INTO products DEFAULT VALUES;
```

単一コマンドで複数行を挿入することができます。

```
INSERT INTO products (product_no, name, price) VALUES  
    (1, 'Cheese', 9.99),  
    (2, 'Bread', 1.99),  
    (3, 'Milk', 2.99);
```

ティップ: 一度に大量のデータを挿入する場合は [COPY](#) コマンドの使用を検討してください。 [INSERT](#) コマンドほどの柔軟性はありませんが、より効率的です。大量のデータをロードする性能を向上するためについて、詳細は [項14.4](#) を参照してください。

6.2. データの更新

既にデータベースに入っているデータを修正することを「更新する」と言います。個別の行、テーブル内の全ての行、あるいは全ての行のサブセットを更新することができます。各列は、他の列に影響を及ぼすことなく個別に更新することができます。

既存の行の更新を行うには[UPDATE](#)コマンドを使用してください。その際には3つの情報が必要となります。

1 11 更新するテーブルと列の名前

1 11 更新後の列の値

1 11 更新する行（複数も可）

[第5章](#)で説明した、一般にSQLでは行に対して一意のIDを指定しないことを思い出してください。従って、どの行を更新するかを直接指定することができない場合があります。その場合、更新される行が満たすべき条件を指定します。テーブルにプライマリキーを設定している場合に限り（ユーザが宣言したのかどうかには依存しないで）、プライマリキーと一致する条件を選択することで確実に個別の行を指定することができます。グラフィカルなデータベースアクセスツールは、この方法を使用して行を個別に更新することを可能にしています。

例えば、値段が5である全ての商品の値段を10に更新するには、以下のコマンドを使用します。

```
UPDATE products SET price = 10 WHERE price = 5;
```

これによって更新される行の数はゼロであるかもしれませんが、1つ、あるいは多数であるかもしれません。一致する行がない条件を指定して更新しようとしてもエラーにはなりません。

では、上記のコマンドの詳細を見てみましょう。最初はUPDATEキーワードで、これにテーブル名が続きます。いつも通り、テーブル名はスキーマで修飾することもできます。修飾しない場合はパス内から検索されます。次にSETキーワードがあり、これに列名、等号、そして更新後の列値が続きます。更新後の列値は、定数だけでなく任意のスカラ式で表すことができます。例えば、全ての商品の価格を10%上げるには以下のようにします。

```
UPDATE products SET price = price * 1.10;
```

このように、新しい値を表す式で行の中の古い値（複数可）を参照することもできます。ここでは、WHERE句を省略しました。WHERE句を省略すると、テーブル内の全ての行が更新されます。省略しない場合は、WHERE条件に一致する行のみが更新されます。SET句内の等号が代入を表すのに対し、WHERE句内の等号は比較を表すことに注意してください。ただし、これによって曖昧さが生じることはありません。もちろん、必ずしもWHERE条件が等式でなければならないということはありません。その他にも様々な演算子を使用することができます（[第9章](#)を参照）。ただし、式の評価結果は論理値でなければなりません。

UPDATEコマンドのSET句に複数の値を代入して、複数の列を更新することもできます。例を示します。

```
UPDATE mytable SET a = 5, b = 3, c = 1 WHERE a > 0;
```


6.3. データの削除

これまで、テーブルにデータを追加する方法と、データを変更する方法について説明してきました。これから取り上げるのは不要になったデータを削除する方法です。データの追加が行単位でしか行えないのと同様、削除の場合も、行全体をテーブルから削除するしかありません。前節で、SQLでは個々の行を直接指定する方法がないということを説明しました。ですから行の削除の場合も、削除対象となる行の条件を指定する必要があります。テーブルにプライマリキーが設定されている場合は、その行を正確に指定できます。しかし、条件を満たす複数の行、あるいは、テーブル内の全ての行を一度に削除することもできます。

行を削除するには、[DELETE](#) コマンドを使用します。構文は [UPDATE](#) コマンドによく似ています。例えば、products テーブルから価格が 10 である全ての行を削除するには以下のようにします。

```
DELETE FROM products WHERE price = 10;
```

単に次のようにすると、テーブル内の全ての行が削除されますので注意してください！

```
DELETE FROM products;
```

プログラマに対する警告です。

第 7章問い合わせ

目次

- 7.1. [概要](#)
- 7.2. [テーブル式](#)
 - 7.2.1. [FROM句](#)
 - 7.2.2. [WHERE句](#)
 - 7.2.3. [GROUP BYとHAVING句](#)
 - 7.2.4. [ウィンドウ関数処理](#)
- 7.3. [選択リスト](#)
 - 7.3.1. [選択リスト項目](#)
 - 7.3.2. [列ラベル](#)
 - 7.3.3. [DISTINCT](#)
- 7.4. [問い合わせの結合](#)
- 7.5. [行の並び替え](#)
- 7.6. [LIMITとOFFSET](#)
- 7.7. [VALUESリスト](#)
- 7.8. [WITH問い合わせ（共通テーブル式）](#)
 - 7.8.1. [WITH内のSELECT](#)
 - 7.8.2. [WITH内のデータ変更文](#)

前章までで、テーブルを作成し、これにデータを挿入し、さらに挿入したデータを操作する方法について説明しました。本章では、データベースからデータを取り出す方法について説明します。

7.1. 概要

データベースからデータを取り出す処理、または、取り出すためのコマンドを問い合わせと言います。SQLでは、[SELECT](#)コマンドを、問い合わせを指定するために使います。SELECTコマンドの一般的な構文は次の通りです。

```
[WITH with_queries] SELECT select_list FROM table_expression [sort_specification]
```

以降の節では、選択リスト、テーブル式、並び替えの仕様について詳細に説明します。WITH問い合わせはより進んだ機能のため最後に扱います。

単純な問い合わせの形式は次のようなものです。

```
SELECT * FROM table1;
```

table1というテーブルがあるとして、このコマンドはtable1からすべての行のすべてのユーザ定義の列を取り出します（抽出する手段はクライアントアプリケーションによって異なります。例えば、psqlプログラムでは、アスキー文字によるテキスト図で画面上にテーブルを表示し、クライアントライブラリでは個々の行と列を取り出す機能を提供します。）選択リストの指定における*は、テーブル式が持つすべての列を提供することを意味します。選択リストでは、選択可能な列の一部を選択することも、選択可能な列を使用して計算することもできます。例えば、table1にa、b、cという名前の列がある場合（他の列があっても構いません）、以下のような問い合わせができます。

```
SELECT a, b + c FROM table1;
```

（ここではbおよびcは数値型のデータであると仮定しています。）詳細については[項7.3](#)を参照してください。

FROM table1は、単純な形のテーブル式で、読み込むテーブルは1つだけです。一般にテーブル式は基本テーブルや結合そして副問い合わせなどで複雑に構成されることがあります。しかし、以下のように、テーブル式をすべて省略し、SELECTコマンドを電卓として使用することもできます。

```
SELECT 3 * 4;
```

選択リストの式が様々な結果を返す場合、これはさらに有用です。例えば、関数を次のように呼び出すことができます。

```
SELECT random();
```

7.2. テーブル式

テーブル式はテーブルを計算するためのものです。テーブル式にはFROM句が含まれており、その後ろにオプションとしてWHERE句、GROUP BY句、HAVING句を付けることができます。単純なテーブル式は、単にディスク上のいわゆる基本テーブルと呼ばれるテーブルを参照するだけです。しかし、様々な方法で基本テーブルを修正したり、組み合わせたりするためにより複雑な式を使用することができます。

テーブル式のオプションWHERE句、GROUP BY句、およびHAVING句は、FROM句で派生したテーブル上に対して次々に変換を実行するパイプラインを指定します。これらの変換によって仮想テーブルが1つ生成されます。そしてこの仮想テーブルの行が選択リストに渡され、問い合わせの出力行が計算されます。

7.2.1. FROM句

FROM句は、カンマで分けられたテーブル参照リストで与えられる1つ以上のテーブルから、1つのテーブルを派生します。

```
FROM table_reference [, table_reference [, ...]]
```

テーブル参照は、テーブル名（スキーマで修飾することもできます）、または、副問い合わせ、テーブル結合、それらの複雑な組み合わせなどから派生されたテーブルを取ることができます。FROM句に複数のテーブル参照がある場合、それらは、WHERE句、GROUP BY句、およびHAVING句で変換できる中間的な仮想テーブルを作るためにクロス結合（下記を参照）され、最終的にはテーブル式全体の結果となります。

テーブル参照で、テーブルの継承階層の親テーブルの名前を指定すると、テーブル名の前にONLYキーワードがない場合は、テーブル参照はそのテーブルだけでなくその子テーブルに継承されたすべての行を生成します。しかし、この参照は名前を指定したテーブルに現れる列のみを生成し、子テーブルで追加された列は無視されます。

テーブル名の前にONLYを記述する代わりに、テーブル名の後に*を記述して、子テーブルが含まれることを明示的に指定することができます。この動作は（[sql_inheritance](#)設定オプションの設定を変更していない限り）デフォルトですので、*を記述することは必要ありません。しかし*を記述してさらにテーブルが検索されることを強調することができますので有用かもしれません。

7.2.1.1. 結合テーブル

結合テーブルは、2つの（実または派生）テーブルから、指定した結合種類の規則に従って派生したテーブルです。内部結合、外部結合、およびクロス結合が使用可能です。

結合の種類

クロス結合

```
T1 CROSS JOIN T2
```

T1およびT2からのすべての可能な行の組み合わせ（つまりデカルト積）に対し、結合されたテーブルはT2のすべての列が続くT1のすべての列から成る行を含みます。テーブルがそれぞれN行とM行で構成されているとすると、結合されたテーブルの行数はN * M行となります。

FROM T1 CROSS JOIN T2はFROM T1, T2と同じです。また（後述の）FROM T1 INNER JOIN T2 ON TRUEとも同じです。

修飾付き結合

```
T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2 ON boolean_expression
T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2 USING ( join column list )
T1 NATURAL { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2
```

INNERやOUTERは省略可能です。INNERがデフォルトとなります。LEFT、RIGHT、FULLは外部結合を意味します。

結合条件は、ON句かUSING句で指定するか、またはNATURAL記述で暗黙的に指定します。結合条件は、以下で詳しく説明するように、2つの元となるテーブルのどの行が"一致するか"を決めます。

ON句は最も一般的な結合条件であり、WHERE句で使われるものと同じ論理値評価式となります。ON式の評価が真となる場合、T1およびT2の対応する行が一致します。

USINGは略記法です。それは、結合テーブルが共通で持つカンマで区切られた列名のリストから、各々の列の組み合わせの等価性を結合条件として形成します。さらに、JOIN USINGの出力は、入力列で等価判定された列の組み合わせそれぞれに対する1列と、その後に各テーブルの残った列が続きます。つまり、USING (a, b, c)はON (t1.a = t2.a AND t1.b

= t2.b AND t1.c = t2.c) と等価です。ただし、ONを使った場合は、結果においてa、b、cはそれぞれ2つの列になりますが、USINGを使うとそれぞれ1つの列になるという例外があります（そしてSELECT *が使われているとそれらは最初に現れます）。

最後に、NATURALはUSINGの略記形式です。2つの入力テーブルの両方に含まれているすべての列名で構成されるUSINGリストを形成します。USINGと同様、これらの列は出力テーブルに一度だけ現れます。共通する列が存在しない場合、NATURALはCROSS JOINと同様に動作します。

修飾付き結合には次のものがあります。

INNER JOIN（内部結合）

T1の行R1に対して、T2においてR1との結合条件を満たしている行が、結合されたテーブルに含まれます。

LEFT OUTER JOIN（左外部結合）

まず、内部結合が行われます。その後、T2のどの行との結合条件も満たさないT1の各行については、T2の列をNULL値として結合行が追加されます。したがって、連結されたテーブルは常にT1の行それぞれに少なくとも1つの行があります。

RIGHT OUTER JOIN（右外部結合）

まず、内部結合が行われます。その後、T1のどの行の結合条件も満たさないT2の各行については、T1の列をNULL値として結合行が追加されます。これは左結合の反対です。結果のテーブルは、T2の行が常に入りります。

FULL OUTER JOIN（完全外部結合）

まず、内部結合が行われます。その後、T2のどの行の結合条件も満たさないT1の各行については、T2の列をNULL値として結合行が追加されます。さらに、T1のどの行でも結合条件を満たさないT2の各行に対して、T1の列をNULL値として結合行が追加されます。

すべての結合は、互いに結び付けたり、あるいは入れ子にしたりすることができます。T1とT2のどちらか、あるいは両方が、結合テーブルになることがあります。括弧は結合の順序を制御するためにJOIN句を括弧に使うことができます。括弧がない場合、JOIN句は左から右に入れ子にします。

まとめとして、以下のテーブルt1

num	name
1	a
2	b
3	c

および、テーブルt2

num	value
1	xxx
3	yyy
5	zzz

を想定すると、以下のように様々な結合に関する結果が得られます。

```
=> SELECT * FROM t1 CROSS JOIN t2;
```

num	name	num	value
1	a	1	xxx
1	a	3	yyy
1	a	5	zzz
2	b	1	xxx
2	b	3	yyy
2	b	5	zzz
3	c	1	xxx
3	c	3	yyy
3	c	5	zzz

(9 rows)

```
=> SELECT * FROM t1 INNER JOIN t2 ON t1.num = t2.num;
```

num	name	num	value
1	a	1	xxx
3	c	3	yyy

(2 rows)

```
=> SELECT * FROM t1 INNER JOIN t2 USING (num);
```

```

num | name | value
-----+-----+-----
  1 | a    | xxx
  3 | c    | yyy
(2 rows)

=> SELECT * FROM t1 NATURAL INNER JOIN t2;
num | name | value
-----+-----+-----
  1 | a    | xxx
  3 | c    | yyy
(2 rows)

=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num;
num | name | num | value
-----+-----+-----
  1 | a    |    1 | xxx
  2 | b    |    | 
  3 | c    |    3 | yyy
(3 rows)

=> SELECT * FROM t1 LEFT JOIN t2 USING (num);
num | name | value
-----+-----+-----
  1 | a    | xxx
  2 | b    | 
  3 | c    | yyy
(3 rows)

=> SELECT * FROM t1 RIGHT JOIN t2 ON t1.num = t2.num;
num | name | num | value
-----+-----+-----
  1 | a    |    1 | xxx
  3 | c    |    3 | yyy
    |     |    5 | zzz
(3 rows)

=> SELECT * FROM t1 FULL JOIN t2 ON t1.num = t2.num;
num | name | num | value
-----+-----+-----
  1 | a    |    1 | xxx
  2 | b    |    | 
  3 | c    |    3 | yyy
    |     |    5 | zzz
(4 rows)

```

`ON`で指定される結合条件には、結合に直接関係しない条件も含めることができます。これは一部の問い合わせにおいては便利ですが、使用の際には注意が必要です。例を示します。

```

=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num AND t2.value = 'xxx';
num | name | num | value
-----+-----+-----
  1 | a    |    1 | xxx
  2 | b    |    | 
  3 | c    |    | 
(3 rows)

```

`WHERE`句の中に制約を記述すると異なる結果になることに注意してください。

```

=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num WHERE t2.value = 'xxx';
num | name | num | value
-----+-----+-----
  1 | a    |    1 | xxx
(1 row)

```

この理由は`ON`句の中の制約は結合の前に処理され、一方`WHERE`句の中の制約は結合の後に処理されることによります。

7.2.1.2. テーブルと列の別名

テーブルや複雑なテーブル参照は、問い合わせの後の方で派生テーブルを参照するために一時的な名前を与えることができます。これをテーブルの別名と呼びます。

テーブルの別名を作成するには以下のようにします。

```
FROM table_reference AS alias
```

もしくは

```
FROM table_reference alias
```

`AS`キーワードはなくても構わないノイズです。`alias`は任意の識別子になります。

テーブルの別名の一般的な適用法は、長いテーブル名に短縮した識別子を割り当てて結合句を読みやすくすることです。 例を示します。

```
SELECT * FROM some_very_long_table_name s JOIN another_fairly_long_name a ON s.id = a.num;
```

別名は、現在の問い合わせに関してはテーブル参照をする時の新しい名前になります。 問い合わせの他の場所で元々の名前でテーブルを参照することはできなくなります。 よって、これは有効ではありません。

```
SELECT * FROM my_table AS m WHERE my_table.a > 5;      -- wrong
```

テーブルの別名は主に表記を簡単にするためにあります。 しかし次のように、1つのテーブルが自分自身と結合する場合は、必須となります。

```
SELECT * FROM people AS mother JOIN people AS child ON mother.id = child.mother_id;
```

さらに、テーブル参照が副問い合わせ（[項7.2.1.3](#)を参照）の場合に別名が必要になります。

括弧は曖昧さをなくすために使われます。 次の例では、最初の文で2つ目のmy_tableのインスタンスにbという別名を付与し、一方、2つ目の文では結合結果に対して別名を付与しています。

```
SELECT * FROM my_table AS a CROSS JOIN my_table AS b ...
SELECT * FROM (my_table AS a CROSS JOIN my_table) AS b ...
```

次のような形式でテーブル別名を付けて、テーブル自身と同様にテーブルの列に一時的な名前を付けることができます。

```
FROM table_reference [AS] alias ( column1 [, column2 [, ...]] )
```

もし、実際のテーブルが持つ列よりも少ない数の列の別名が与えられる場合、残りの列は改名されません。 この構文は、自己結合あるいは副問い合わせで特に役立ちます。

別名がJOIN句の結果に適用される場合、別名はJOIN内で参照される元々の名を隠します。 以下に例を示します。

```
SELECT a.* FROM my_table AS a JOIN your_table AS b ON ...
```

は有効なSQLですが、

```
SELECT a.* FROM (my_table AS a JOIN your_table AS b ON ...) AS c
```

は有効ではありません。 テーブルの別名aは、別名cの外側では参照することができません。

7.2.1.3. 副問い合わせ

派生テーブルを指定する副問い合わせは括弧で囲む必要があります。 また、必ずテーブル別名が割り当てられている必要があります。（[項7.2.1.2](#)を参照してください。） 例を示します。

```
FROM (SELECT * FROM table1) AS alias_name
```

この例はFROM table1 AS alias_nameと同じです。 さらに興味深いケースとして、副問い合わせがグループ化や集約を含んでいる場合、単純結合にまとめることはできないということがあります。

また、副問い合わせをVALUESリストとすることもできます。

```
FROM (VALUES ('anne', 'smith'), ('bob', 'jones'), ('joe', 'blow'))
      AS names(first, last)
```

繰り返しますが、テーブルの別名が必要です。 VALUESリストの列に別名を付与することは省略することもできますが、付与することを勧めます。 [項7.7](#)を参照してください。

7.2.1.4. テーブル関数

テーブル関数は、基本データ型（スカラ型）、もしくは複合データ型（テーブル行）からなる行の集合を生成する関数です。 これらは、テーブル、ビュー、問い合わせのFROM句内の副問い合わせのように使用されます。 テーブル関数から返される列は、テーブル、ビュー、副問い合わせ列と同様の手順で、SELECT、JOIN、WHEREの中に含めることができます。

テーブル関数が基本データ型を返す場合、単一の結果列名は関数名に一致します。 関数が複合型を返す場合は、結果列はその型の個々の属性と同じ名前になります。

FROM句でテーブル関数に別名を付けることも、別名を付けずにそのまま使用することもできます。 別名を付けずにFROM句で関数を使用した場合、関数名が出力テーブル名として使用されます。

以下に数例示します。

```
CREATE TABLE foo (fooid int, foosubid int, fooname text);
```



```
CREATE FUNCTION getfoo(int) RETURNS SETOF foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT * FROM getfoo(1) AS t1;

SELECT * FROM foo
    WHERE foosubid IN (
        SELECT foosubid
        FROM getfoo(foo.fooid) z
        WHERE z.fooid = foo.fooid
    );

CREATE VIEW vw_getfoo AS SELECT * FROM getfoo(1);

SELECT * FROM vw_getfoo;
```

呼び出し方法に応じて異なる列集合を返すテーブル関数を定義することが役に立つ場合があります。これをサポートするには、テーブル関数をrecord仮定型を返すものと宣言します。こうした関数を問い合わせで使用する場合、システムがその問い合わせをどのように解析し計画を作成すればよいのかが判断できるように、想定した行構造を問い合わせ自身内に指定しなければなりません。次の例で考えてみましょう。

```
SELECT *
    FROM dblink('dbname=mydb', 'SELECT proname, prosrc FROM pg_proc')
    AS t1(proname name, prosrc text)
    WHERE proname LIKE 'bytea%';
```

[dblink](#)関数（[dblink](#)モジュールの一部）は遠隔問い合わせを実行します。これは任意の問い合わせで使えるように、recordを返すものと宣言されています。実際の列集合は、パーサが例えば*がどのように展開されるかを理解できるように、呼び出した問い合わせ内で指定されなければなりません。

7.2.2. WHERE句

[WHERE句](#)の構文は以下の通りです。

```
WHERE search_condition
```

ここで、search_conditionにはboolean型を返すどのような評価式（[項4.2](#)を参照）も指定できます。

FROM句の処理が終わった後、派生した仮想テーブルの各行は検索条件と照合されます。条件の結果が真の場合、その行は出力されます。そうでない（すなわち結果が偽またはNULLの）場合は、その行は捨てられます。一般的に検索条件は、FROM句で生成されたテーブルの最低1列を参照します。これは必須ではありませんが、そうしないとWHERE句はまったく意味がなくなります。

注意: 内部結合の結合条件は、WHERE句でもJOIN句でも記述することができます。例えば、以下のテーブル式は等価です。

```
FROM a, b WHERE a.id = b.id AND b.val > 5
```

および

```
FROM a INNER JOIN b ON (a.id = b.id) WHERE b.val > 5
```

また、以下でも同じです。

```
FROM a NATURAL JOIN b WHERE b.val > 5
```

どれを使うかは、主にスタイルの問題です。FROM句でJOIN構文を使用すると、SQL標準であってもおそらく他のSQLデータベース管理システムに移植できません。外部結合については、FROM句以外に選択の余地はありません。外部結合のON句またはUSING句は、WHERE条件とは等しくありません。なぜなら、最終結果での行を除去すると同様に、（一致しない入力行に対する）行の追加となるからです。

WHERE句の例を以下に示します。

```
SELECT ... FROM fdt WHERE c1 > 5

SELECT ... FROM fdt WHERE c1 IN (1, 2, 3)

SELECT ... FROM fdt WHERE c1 IN (SELECT c1 FROM t2)

SELECT ... FROM fdt WHERE c1 IN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10)

SELECT ... FROM fdt WHERE c1 BETWEEN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10) AND 100

SELECT ... FROM fdt WHERE EXISTS (SELECT c1 FROM t2 WHERE c2 > fdt.c1)
```

`fdt`はFROM句で派生されたテーブルです。WHERE句の検索条件を満たさなかった行は、`fdt`から削除されます。評価式としてのスカラー副問い合わせの使い方に注目してください。他の問い合わせのように、副問い合わせは複雑なテーブル式を使うことができます。副問い合わせの中でどのように`fdt`が参照されるかにも注意してください。`c1`を`fdt.c1`のように修飾することは、`c1`が副問い合わせの入力テーブルから派生した列名でもある時にだけ必要です。列名の修飾は、必須の場合ではなくても、明確にするために役立ちます。この例は、外側の問い合わせの列名の有効範囲を、どのようにして内側の問い合わせまで拡張するかを示します。

7.2.3. GROUP BYとHAVING句

WHEREフィルタを通した後、派生された入力テーブルをGROUP BY句でグループ化し、また、HAVING句を使用して不要なグループを取り除くことができます。

```
SELECT select_list
FROM ...
[WHERE ...]
GROUP BY grouping_column_reference [, grouping_column_reference]...
```

GROUP BY句は、テーブル内で選択された全列で同じ値を所有する行をまとめてグループ化するために使用されます。列の列挙順は関係ありません。これは共通する値を持つそれぞれの行の集合をグループ内のすべての行を代表する1つのグループ行にまとめる効果があります。これにより、出力の冗長度を排除し、さらにまた、これらのグループに適用される集約が計算されます。以下に例を示します。

```
=> SELECT * FROM test1;
 x | y
---+---
 a | 3
 c | 2
 b | 5
 a | 1
(4 rows)

=> SELECT x FROM test1 GROUP BY x;
 x
---
 a
 b
 c
(3 rows)
```

2番目の問い合わせでは、`SELECT * FROM test1 GROUP BY x`と書くことはできません。各グループに関連付けられる列`y`の値がないからです。グループごとに単一の値を持つので、選択リストでGROUP BYで指定した列を参照することができます。

一般的に、テーブルがグループ化されている場合、GROUP BYでリストされていない列は集約式を除いて参照することはできません。集約式の例は以下の通りです。

```
=> SELECT x, sum(y) FROM test1 GROUP BY x;
 x | sum
---+-----
 a | 4
 b | 5
 c | 2
(3 rows)
```

上記で`sum()`は、グループ全体について単一の値を計算する集約関数です。使用可能な集約関数の詳細については、[項9.20](#)を参照してください。

ティップ: 集約式を使用しないグループ化は、列内の重複しない値の集合を効率良く計算します。これはDISTINCT句 ([項7.3.3](#)を参照) の使用で同じように達成することができます。

別の例を示します。これは各製品の総売上を計算します（全製品の総売上ではありません）。

```
SELECT product_id, p.name, (sum(s.units) * p.price) AS sales
FROM products p LEFT JOIN sales s USING (product_id)
GROUP BY product_id, p.name, p.price;
```

この例では、`product_id`列、`p.name`列、`p.price`列は必ずGROUP BY句で指定する必要があります。なぜなら、これらは問い合わせ選択リスト（後述）の中で使われているからです。`s.units`列はGROUP BYで指定する必要はありません。これは、製品ごとの売上計算の集約式（`sum(...)`）の中だけで使われるためです。この問い合わせは、各製品に対して製品の全販売に関する合計行が返されます。

`products`テーブルがそのように、つまり`product_id`がプライマリキーであるように設定されている場合、`name`と`price`列は製品ID（`product_id`）に函数依存しており、このため製品IDグループそれぞれに対してどの`name`と`price`の値を返すかに関するあいまいさがありませんので、上の例では`product_id`でグループ化することで十分です。

厳密なSQLでは、GROUP BYは、元となるテーブルの列によってのみグループ化できますが、PostgreSQLでは、選択リストの列によるグループ化もできるように拡張されています。単純な列名の代わりに、評価式でグループ化することもできます。

GROUP BYを使ってグループ化されたテーブルで特定のグループのみ必要な場合、結果から不要なグループを除くのに、WHERE句のようにHAVING句を使うことができます。構文は以下の通りです。

```
SELECT select_list FROM ... [WHERE ...] GROUP BY ... HAVING boolean_expression
```

HAVING句内の式は、グループ化された式とグループ化されていない式（この場合は集約関数が必要になります）の両方を参照することができます。

例を示します。

```
=> SELECT x, sum(y) FROM test1 GROUP BY x HAVING sum(y) > 3;
 x | sum
---+-----
 a |    4
 b |    5
(2 rows)

=> SELECT x, sum(y) FROM test1 GROUP BY x HAVING x < 'c';
 x | sum
---+-----
 a |    4
 b |    5
(2 rows)
```

次に、より現実的な例を示します。

```
SELECT product_id, p.name, (sum(s.units) * (p.price - p.cost)) AS profit
FROM products p LEFT JOIN sales s USING (product_id)
WHERE s.date > CURRENT_DATE - INTERVAL '4 weeks'
GROUP BY product_id, p.name, p.price, p.cost
HAVING sum(p.price * s.units) > 5000;
```

上の例で、WHERE句は、グループ化されていない列によって行を選択している（この式では最近の4週間の売上のみが真になります）のに対し、HAVING句は出力を売上高が5000を超えるグループに制限しています。集約式が、問い合わせ内で常に同じである必要がないことに注意してください。

ある問い合わせが集約関数を含んでいるがGROUP BY句がない場合でも、グループ化は依然として行われます。結果は単一グループ行（またはHAVINGで単一行が削除されれば、行が多分全くなくなる）となります。HAVING句を含んでいれば、集約関数呼び出しやGROUP BY句がまったく存在しなくても同じことが言えます。

7.2.4. ウィンドウ関数処理

問い合わせがウィンドウ関数（[項3.5](#)、[項9.21](#)と[項4.2.8](#)を参照）を含んでいれば、これらの関数はグループ化、集約およびHAVING条件検索が行われた後に評価されます。つまり、問い合わせが何らかの集約、GROUP BYまたはHAVINGを使用していれば、ウィンドウ関数により見える行はFROM/WHEREでの本来のテーブル行ではなく、グループ行となります。

複数のウィンドウ関数を使用した場合、そのウィンドウ定義にある構文的に同等であるPARTITION BYおよびORDER BY句を持つすべてのウィンドウ関数は、データ全体に渡って単一の実行手順により評価されることが保証されています。したがって、ORDER BYが一意的に順序付けを決定しなくても同一の並び替え順序付けを見ることができます。しかし、異なるPARTITION BYまたはORDER BY仕様を持つ関数の評価については保証されません。（このような場合、並び替え手順がウィンドウ関数評価の諸手順間で一般的に必要となり、ORDER BYが等価と判断する行の順序付けを保存するような並び替えは保証されません。）

現時点では、ウィンドウ関数は常に事前に並び替えられたデータを必要とするので、問い合わせ出力はウィンドウ関数のPARTITION BY/ORDER BY句のどれか1つに従って順序付けされます。とはいえ、これに依存することは薦められません。確実に結果が特定の方法で並び替えられるようにしたいのであれば、明示的な最上階層のORDER BYを使用します。

7.3. 選択リスト

前節で示したように、`SELECT` コマンド中のテーブル式は、テーブルの結合やビュー、行の抽出、グループ化などにより中間の仮想テーブルを作ります。このテーブルは最終的に選択リストに渡されます。選択リストは、中間のテーブルのどの列を実際に出力するかを決めます。

7.3.1. 選択リスト項目

テーブル式が生成するすべての列を出力する*が最も簡単な選択リストです。そうでなければ、選択リストは、カンマで区切られた ([項4.2](#)で定義された) 評価式のリストです。例えば、以下のような列名のリストであっても構いません。

```
SELECT a, b, c FROM ...
```

`a`、`b`、`c` という列名は、`FROM` 句で参照されるテーブルの実際の列名か、あるいは [項7.2.1.2](#) で説明したような列名に対する別名です。グループ化されていなければ、選択リストで使用可能な名前空間は `WHERE` 句と同じです。グループ化されている場合は、`HAVING` 句と同じとなります。

もし、2つ以上のテーブルで同じ名前の列がある場合は、次のように、テーブル名を必ず指定しなければいけません。

```
SELECT tbl1.a, tbl2.a, tbl1.b FROM ...
```

複数のテーブルを使用する場合、特定のテーブルのすべての列を求める方法も便利かもしれません。

```
SELECT tbl1.*, tbl2.a FROM ...
```

([項7.2.2](#)も参照してください。)

任意の評価式が選択リストで使われる場合、返されるテーブルは、概念的には新たに仮想的な列を追加したものとなります。評価式は、それぞれの結果行で、その列参照を置換した行の値としていったん評価されます。しかし、選択リストの式は `FROM` 句で指定されたテーブル式内の列を参照するものである必要はありません。例えば、定数算術式であっても構いません。

7.3.2. 列ラベル

選択リスト中の項目は、`ORDER BY` 句の中での参照、もしくはクライアントアプリケーションによる表示に使うように、それに続く処理のために名前を割り当てることができます。例を示します。

```
SELECT a AS value, b + c AS sum FROM ...
```

もし `AS` を使った列名の指定がなかった場合、システムはデフォルトの列名を割り当てます。単純な列参照では参照された列名となります。関数呼び出しでは関数名となります。複雑な表現についてはシステムが汎用の名前を生成します。

`AS` キーワードは省略することができますが、新規列名が PostgreSQL キーワード ([付録C](#)を参照) のいかなるものとも一致しない場合のみです。あるキーワードと予測外の一致を防ぐために、列名を二重引用符で囲むことができます。例えば、`VALUE` はキーワードですのうまく動作しません。

```
SELECT a value, b + c AS sum FROM ...
```

しかしこれは動きます。

```
SELECT a "value", b + c AS sum FROM ...
```

ありえそうな将来のキーワードの追加に対応するため、`AS` を記述するか、出力列名を二重引用符で囲むかのいずれかを推奨します。

注意: ここでの出力列の名前は、`FROM` 句 ([項7.2.1.2](#)を参照) で示したものと異なります。同じ列の名前を2度変更することができますが、渡されるのは選択リストの中で割り当てられたものです。

7.3.3. DISTINCT

選択リストが処理された後、結果テーブルの重複行を削除の対象にすることもできます。これを指定するためには、`SELECT` の直後に `DISTINCT` キーワードを記述します。

```
SELECT DISTINCT select_list ...
```

(`DISTINCT` の代わりに `ALL` キーワードを指定すると、すべての行がデフォルトで保持されます。)

少なくとも1つの列の値が異なる場合、それら2行は別個とみなされます。 NULL値は、この比較において等しいとみなされません。

また、任意の式を使用して、どの行が別であるかを決定することもできます。

```
SELECT DISTINCT ON (expression [, expression ...]) select_list ...
```

expression は、すべての行で評価される任意の評価式です。 すべての式が等しくなる行の集合は、重複しているとみなされ、集合の最初の行だけが出力されます。 `DISTINCT` フィルタに掛けられる行の順序の一意性を保証できるよう問い合わせが列を並び替えしない限り、出力される集合の"最初の行"は予想不可能であることに注意してください。（`DISTINCT ON` 処理は、`ORDER BY` による並び替えの後に行われます。）

`DISTINCT ON` 句は標準SQLではありません。 さらに、結果が不定となる可能性があるため、好ましくないスタイルとみなされることもあります。 `GROUP BY` と `FROM` 中の副問い合わせをうまく使うことにより、この構文を使わずに済みますが、`DISTINCT ON` 句はしばしば非常に便利な代案となります。

[前のページ](#)
テーブル式

[ホーム](#)
[上に戻る](#)

[次のページ](#)
問い合わせの結合

7.4. 問い合わせの結合

2つの問い合わせの結果は、和、積、差の集合演算を使って結合することができます。構文は以下の通りです。

```
query1 UNION [ALL] query2
query1 INTERSECT [ALL] query2
query1 EXCEPT [ALL] query2
```

`query1`と`query2`は、これまで説明した機能をすべて使用することができる問い合わせです。集合演算は入れ子にしたり、繋げたりすることができます。例えば、以下の文を見てみましょう。

```
query1 UNION query2 UNION query3
```

上記の文は以下のように実行されます。

```
(query1 UNION query2) UNION query3
```

`UNION`は、`query2`の結果を`query1`の結果に付加します（しかし、実際に行が返された順序である保証はありません）。さらに、`UNION ALL`を指定しないと、`DISTINCT`と同様に、結果から重複している行を削除します。

`INTERSECT`は、`query1`の結果と`query2`の結果の両方に含まれているすべての行を返します。`INTERSECT ALL`を使用しないと、重複している行は削除されます。

`EXCEPT`は、`query1`の結果には含まれているけれども、`query2`の結果には含まれていないすべての行を返します。（これが2つの問い合わせの差であると言われることがあります。）この場合も、`EXCEPT ALL`を使用しないと、重複している行は削除されます。

2つの問い合わせの和、積、差を算出するために、その2つの問い合わせは"和演算互換"でなければいけません。つまり、その問い合わせが同じ数の列を返し、対応する列は互換性のあるデータ型（[項10.5](#)を参照）でなければなりません。

7.5. 行の並び替え

ある問い合わせが1つの出力テーブルを生成した後（選択リストの処理が完了した後）、並べ替えることができます。並べ替えが選ばれなかった場合、行は無規則な順序で返されます。そのような場合、実際の順序は、スキャンや結合計画の種類や、ディスク上に格納されている順序に依存します。しかし、当てにはなりません。特定の順序で出力することは、明白に並べ替える処理を選択することでのみ保証されます。

ORDER BY 句は並べ替えの順番を指定します。

```
SELECT select_list
FROM table_expression
ORDER BY sort_expression1 [ASC | DESC] [NULLS { FIRST | LAST }]
[, sort_expression2 [ASC | DESC] [NULLS { FIRST | LAST }] ...]
```

並び替え式は問い合わせの選択リスト内で有効な任意の式を取ることができます。以下に例を示します。

```
SELECT a, b FROM table1 ORDER BY a + b, c;
```

複数の式が指定された場合、前の値と同じ値を持つ行を並び替えする際に後の値が使用されます。列指定の後に省略可能なASCもしくはDESCを付与することで、並び替えの方向を昇順、降順にするかを設定することができます。ASC順がデフォルトです。昇順では、小さな値を先に出力します。ここでの"小さい"とは、<演算子によって決定されます。同様に降順では>演算子で決定されます。[1]

NULLS FIRST および NULLS LAST オプションを使用して、その並び替え順においてNULL値を非NULL値の前にするか後にするかを決定することができます。デフォルトでは、NULL値はあたかもすべての非NULL値よりも大きいとみなして並び替えます。と言うことは、NULLS FIRST はDESC順序付けのデフォルトで、そうでなければNULLS LAST です。

この順序づけオプションは、並び替えされる列を個別に指定したとみなされることに注意してください。例えば、ORDER BY x, y DESC は、ORDER BY x DESC, y DESC と同じではなく、ORDER BY x ASC, y DESC を意味します。

sort_expression は以下のように列ラベルもしくは出力列の番号で指定することができます。

```
SELECT a + b AS sum, c FROM table1 ORDER BY sum;
SELECT a, max(b) FROM table1 GROUP BY a ORDER BY 1;
```

両方とも最初の出力列で並び替えされます。出力列名は単体でなければなりません。つまり式としては使用できないことに注意してください。例えば以下は間違いです。

```
SELECT a + b AS sum, c FROM table1 ORDER BY sum + c;          -- 間違い
```

これは多義性を減らすための制限です。ORDER BY 項目が単純な名前であっても、出力列名とテーブル式による列と同じ名前となる場合、多義性はまだ存在します。この場合、出力列名が使用されます。AS を使用して他のテーブル列の名前と同じ名前に出力列を変名した場合にのみ混乱が発生します。

ORDER BY を、UNION、INTERSECT、EXCEPT 組み合わせの結果に適用することができます。しかしこの場合、出力列の名前または番号でのみ並び替えすることができ、式では並び替えすることができません。

注意

[1]

実際、PostgreSQL は、ASC と DESC の並び替え順を決定するために、式のデータ型用のデフォルトの B-tree 演算子クラスを使用します。慣習的に、データ型は < と > 演算子をこの並び替え順になるように設定されます。しかし、ユーザ定義データ型の設計者は異なるものを選択することができます。

7.6. LIMITとOFFSET

LIMITおよびOFFSETを使うことで、問い合わせの実行で生成された行の一部だけを取り出すことができます。

```
SELECT select_list
      FROM table_expression
      [ ORDER BY ... ]
      [ LIMIT { number | ALL } ] [ OFFSET number ]
```

限度数を指定すると、指定した行数より多くの行が返されることはありません（しかし、問い合わせの結果によって、指定より少ないことはあります）。LIMIT ALLは、LIMIT句を省略した場合と同じです。

OFFSETは、返す行の開始位置を飛ばす行数を指定します。OFFSET 0は、OFFSET句を省略した場合と同じで、LIMIT NULLはLIMIT句を省略した場合と同じです。OFFSETおよびLIMITの両者が指定された場合、OFFSET分の行を飛ばしてから、返されるLIMIT行を数え始めます。

LIMITを使用する時は、結果の行を一意的順序に制約するORDER BY句を使用することが重要です。ORDER BYを使わなければ、予測できない問い合わせの行の部分集合を得ることになるでしょう。10番目から20番目の行を問い合わせることもあるでしょうが、10番目から20番目というのは、どのような順序を用いるのでしょうか？ORDER BYを指定しなければ、並び順はわかりません。

問い合わせオプティマイザは、問い合わせ計画を生成する時にLIMITを考慮します。したがって、LIMITとOFFSETに指定した値によって、（行の順序が異なる）異なった計画が得られます。このように、1つの問い合わせ結果から異なる部分集合を選び出すために、異なるLIMIT/OFFSETの値を使用すると、ORDER BYで結果の順序を制約しない限りは、矛盾した結果が生じるでしょう。これは不具合ではありません。ORDER BYを使って順序を制約しない限り、SQLは必ずしも特定の順序で問い合わせの結果を渡さないという特性の必然的な結果です。

OFFSET句で飛ばされる行を、実際にはサーバ内で計算しなければなりません。そのため、大きな値のOFFSETは非効率的になることがあります。

7.7. VALUES リスト

VALUES は、実際にディスク上にテーブルを生成し、値を投入することなく、問い合わせの中で使用することができる"定数テーブル"を生成する手法を提供します。構文を以下に示します。

```
VALUES ( expression [, ...] ) [, ...]
```

括弧で括られた式のリストがそれぞれ、そのテーブル内である行を生成します。リストは同一の要素数（つまり、テーブルの列数）を持たなければなりません。また、各リストで対応する項目のデータ型に互換性がなければなりません。最終的に実際各列に割り当てられるデータ型は、UNION 同様の規則に従って決定されます。（[項10.5](#)を参照してください。）

以下に例を示します。

```
VALUES (1, 'one'), (2, 'two'), (3, 'three');
```

これは、2列3行のテーブルを返します。実質以下と同じです。

```
SELECT 1 AS column1, 'one' AS column2
UNION ALL
SELECT 2, 'two'
UNION ALL
SELECT 3, 'three';
```

デフォルトでは、PostgreSQLはVALUES テーブルの各列にcolumn1、column2といった名前をつけます。標準SQLでは列名は規定されていないので、他のデータベースシステムでは他の名前を付与しています。そのため、通常はテーブル別名リストを使用して、デフォルトの名前を上書きする方がよいでしょう。

文法的には、式リストに続くVALUESは、以下と同様に扱われます。

```
SELECT select_list FROM table_expression
```

そして、SELECTが記述できるところであれば、記述することができます。例えば、UNIONの一部として使用することもできますし、sort_specification (ORDER BY、LIMIT、OFFSET)に付けることもできます。VALUESはINSERT コマンドの元データとしてもっとも頻繁に使用されます。次に使用頻度が高いのは副問い合わせとしての使用です。

詳しくは[VALUES](#)を参照してください。

7.8. WITH 問い合わせ（共通テーブル式）

WITH は、より大規模な問い合わせで使用される補助文を記述する方法を提供します。これらの文は共通テーブル式または CTE とよく呼ばれるものであり、1 つの問い合わせのためだけに存在する一時テーブルを定義するものと考えられます。WITH 句内の補助文はそれぞれ SELECT、INSERT、UPDATE または DELETE を取ることができます。そして WITH 句自身は、これも SELECT、INSERT、UPDATE または DELETE を取ることができる主文に付与されます。

7.8.1. WITH 内の SELECT

WITH 内の SELECT の基本的な価値は、複雑な問い合わせをより単純な部品に分解することです。以下に例を示します。

```
WITH regional_sales AS (
    SELECT region, SUM(amount) AS total_sales
    FROM orders
    GROUP BY region
), top_regions AS (
    SELECT region
    FROM regional_sales
    WHERE total_sales > (SELECT SUM(total_sales)/10 FROM regional_sales)
)
SELECT region,
       product,
       SUM(quantity) AS product_units,
       SUM(amount) AS product_sales
FROM orders
WHERE region IN (SELECT region FROM top_regions)
GROUP BY region, product;
```

これは地域（region）販売トップのみの製品ごとの売上高を表示します。WITH 句は、regional_sales、top_regions という名前の 2 つの補助文を定義します。ここで、regional_sales の出力は top_regions 内で使用され、top_regions は SELECT 主問い合わせで使用されます。この例は WITH なしでも記述できますが、二階層の入れ子の副 SELECT を必要とします。この方法に従うほうが多少扱いやすいです。

オプションの RECURSIVE 修飾子は、WITH を、単に構文上の利便性の高めるだけでなく標準的な SQL では不可能な機能を実現させます。RECURSIVE を使用すれば、WITH 問い合わせが行った自己の結果を参照できるようになります。1 から 100 までの数を合計する非常に単純な問い合わせは以下のようなものです。

```
WITH RECURSIVE t(n) AS (
    VALUES (1)
    UNION ALL
    SELECT n+1 FROM t WHERE n < 100
)
SELECT sum(n) FROM t;
```

再帰的 WITH 問い合わせの汎用形式は常に、非再帰的表現（*non-recursive term*）、そして UNION（または UNION ALL）、そして再帰的表現（*recursive term*）です。再帰的表現だけが、その問い合わせ自身の出力への参照を含むことができます。このような問い合わせは以下のように実行されます。

再帰的問い合わせの評価

- 1 11 非再帰的表現を評価します。UNION（しかし UNION ALL ではありません）では、重複行を廃棄します。その再帰的問い合わせの結果に残っている全ての行を盛り込み、同時にそれらを暫定的作業テーブルに置きます。
- 1 11 作業テーブルが空でないのであればこれらの手順を繰り返します。
 - 1 11 再帰自己参照に対する作業テーブルの実行中の内容を置換し、再帰的表現を評価します。UNION（しかし UNION ALL ではない）に対し、重複行と前の結果行と重複する行を破棄します。その再帰的問い合わせの結果に残っているすべての行を盛り込み、同時にそれらを暫定的中間テーブルに置きます。
 - 1 11 中間テーブルの内容で作業テーブルの内容を差し替え、中間テーブルを空にします。

注意: 厳密には、この手順は反復であって再帰ではありませんが、RECURSIVE は SQL 標準化委員会で選ばれた用語です。

上記の例で、作業テーブルはそれぞれの手順での単なる単一行で、引き続き作業で 1 から 100 間での値を獲得します。100 番目の作業で、WHERE 句による出力が無くなり、問い合わせが終了します。

再帰的問い合わせは階層的、またはツリー構造データに対処するため一般的に使用されます。実用的な例は、直接使用する部品

を表すテーブル1つのみが与えられ、そこから製品すべての直接・間接部品を見つける問い合わせです。

```
WITH RECURSIVE included_parts(sub_part, part, quantity) AS (  
    SELECT sub_part, part, quantity FROM parts WHERE part = 'our_product'  
    UNION ALL  
    SELECT p.sub_part, p.part, p.quantity  
    FROM included_parts pr, parts p  
    WHERE p.part = pr.sub_part  
)  
SELECT sub_part, SUM(quantity) as total_quantity  
FROM included_parts  
GROUP BY sub_part
```

再帰的問い合わせを扱う場合、問い合わせの再帰部分が時としてタプルを返さない、または、問い合わせが永久にループするといったことのないように注意することが重要です。 たまには、UNION ALLの替わりにUNIONを使用し、重複する前回の出力行を廃棄することで、これを実現できます。 しかし、ある周期でしばしば完全に重複している出力行を含みません。 同じ場所が既に到達されたかどうかを確認するために、1つだけ、または数フィールドを検査する必要があるかもしれません。 このような状態を取り扱う標準手法は、既に巡回された値の配列を計算することです。 例えば、linkフィールドを使ってテーブルgraphを検索する以下の問い合わせを考えて見ます。

```
WITH RECURSIVE search_graph(id, link, data, depth) AS (  
    SELECT g.id, g.link, g.data, 1  
    FROM graph g  
    UNION ALL  
    SELECT g.id, g.link, g.data, sg.depth + 1  
    FROM graph g, search_graph sg  
    WHERE g.id = sg.link  
)  
SELECT * FROM search_graph;
```

この問い合わせはlink関係が循環を含んでいればループします。 "depth"出力が必要ですので、UNION ALLをUNIONに変えるだけでは、ループを取り除くことができません。 その代わり、linkの特定の経路をたどっている間、同じ列に到達したかどうかを認識する必要があります。 このループしやすい問い合わせに、pathとcycleの2列を加えます。

```
WITH RECURSIVE search_graph(id, link, data, depth, path, cycle) AS (  
    SELECT g.id, g.link, g.data, 1,  
        ARRAY[g.id],  
        false  
    FROM graph g  
    UNION ALL  
    SELECT g.id, g.link, g.data, sg.depth + 1,  
        path || g.id,  
        g.id = ANY(path)  
    FROM graph g, search_graph sg  
    WHERE g.id = sg.link AND NOT cycle  
)  
SELECT * FROM search_graph;
```

巡回防止は別として、配列値は"path"がどんな特定行に到達したかの表示としての公正さからしばしば利用価値があります。

循環を認識するために検査するために必要なフィールドが複数存在する一般的な状況では、行の配列を使用します。 例えば、f1とf2 のフィールドを比較する必要があるときは次のようにします。

```
WITH RECURSIVE search_graph(id, link, data, depth, path, cycle) AS (  
    SELECT g.id, g.link, g.data, 1,  
        ARRAY[ROW(g.f1, g.f2)],  
        false  
    FROM graph g  
    UNION ALL  
    SELECT g.id, g.link, g.data, sg.depth + 1,  
        path || ROW(g.f1, g.f2),  
        ROW(g.f1, g.f2) = ANY(path)  
    FROM graph g, search_graph sg  
    WHERE g.id = sg.link AND NOT cycle  
)  
SELECT * FROM search_graph;
```

ティップ: 循環を認識するために検査するために必要なフィールドが1つだけである一般的な場合では、ROW() 構文を削除します。 これで、複合型配列ではなく単純配列が得られ、効率も上がります。

ティップ: 再帰的問い合わせ評価アルゴリズムは、横型検索順でのその出力を作成します。 このようにして作られた"path"列を外側問い合わせでORDER BYし、縦型検索順の結果の表示が可能です。

ループするかどうか確信が持てない問い合わせをテストする有益な秘訣として、親問い合わせにLIMITを配置します。 例えば、以下の問い合わせはLIMITがないと永久にループします。

```
WITH RECURSIVE t(n) AS (  
    SELECT 1  
    UNION ALL
```

```

        SELECT n+1 FROM t
    )
    SELECT n FROM t LIMIT 100;

```

これが動作するのは、PostgreSQLの実装が、実際に親問い合わせで取り出されるのと同じ数のWITH問い合わせの行のみを評価するからです。この秘訣を実稼動環境で使用することは勧められません。他のシステムでは異なった動作をする可能性があるからです。同時に、もし外部問い合わせを再帰的問い合わせの結果を並び替えしたり、またはそれらを他のテーブルと結合するような書き方をした場合、動作しません。このような場合、外部問い合わせは通常、WITH問い合わせの出力をとにかくすべて取り込もうとするからです。

有用なWITH問い合わせの特性は、親問い合わせ、もしくは兄弟WITH問い合わせによりたとえ1回以上参照されるとしても、親問い合わせ実行でたった1回だけ評価されることです。したがって、複数の場所で必要な高価な計算は、冗長作業を防止するためWITH問い合わせの中に配置することができます。他にありうるアプリケーションとしては、望まれない副作用のある関数の多重評価を避けることです。しかし、反対の見方をすれば、オブティマイザの能力は、親問い合わせからの制約を通常の副問い合わせではなく、WITH問い合わせに押し下げるには、劣っています。WITH問い合わせは一般的に、親問い合わせが後で破棄するであろう行を抑制せずに、書かれた通りに評価されます。（しかし、上で述べたように、問い合わせの参照が限定された数の行のみを要求する場合、評価は早期に停止します。）

上の例ではSELECTを使用するWITHのみを示しています。しかし、同じ方法でINSERT、UPDATE、またはDELETEを付与することができます。それぞれの場合において、これは主コマンド内で参照可能な一時テーブルを実質的に提供します。

7.8.2. WITH内のデータ変更文

WITH内でデータ変更文（INSERT、UPDATE、DELETE）を使用することができます。これにより同じ問い合わせ内で複数の異なる操作を行うことができます。

```

WITH moved_rows AS (
    DELETE FROM products
    WHERE
        "date" >= '2010-10-01' AND
        "date" < '2010-11-01'
    RETURNING *
)
INSERT INTO products_log
SELECT * FROM moved_rows;

```

この問い合わせは実質、productsからproducts_logに行を移動します。WITH内のDELETEはproductsから指定した行を削除し、そのRETURNING句により削除した内容を返します。その後、主問い合わせはその出力を読み取り、それをproducts_logに挿入します。

上の例の見事なところは、INSERT内の副SELECTではなくWITH句がINSERTに付与されていることです。これは、データ更新文は最上位レベルの文に付与されるWITH句内でのみ許されているため必要です。しかし、通常のWITHの可視性規則が適用されますので、副SELECTからWITH文の出力を参照することができます。

上の例で示したように、WITH内のデータ変更文は通常RETURNING句を持ちます。データ変更文の対象テーブルではなく、RETURNING句の出力が問い合わせの残りの部分で参照することができる一時テーブルを形成するものです。WITH内のデータ変更文がRETURNING句を持たない場合、一時テーブルを形成しませんので、問い合わせの残りの部分で参照することができません。これにもかかわらずこうした文は実行されます。特別有用でもない例を以下に示します。

```

WITH t AS (
    DELETE FROM foo
)
DELETE FROM bar;

```

この例はfooテーブルとbarテーブルからすべての行を削除します。クライアントに報告される影響を受けた行数にはbarから削除された行のみが含まれます。

データ変更文内の再帰的な自己参照は許されません。一部の場合において、再帰的なWITHの出力を参照することで、この制限を回避することができます。以下に例を示します。

```

WITH RECURSIVE included_parts(sub_part, part) AS (
    SELECT sub_part, part FROM parts WHERE part = 'our_product'
    UNION ALL
    SELECT p.sub_part, p.part
    FROM included_parts pr, parts p
    WHERE p.part = pr.sub_part
)
DELETE FROM parts
WHERE part IN (SELECT part FROM included_parts);

```

この問い合わせはある製品の直接的な部品と間接的な部品をすべて削除します。

WITH内のデータ変更文は正確に1回のみ実行され、主問い合わせがその出力をすべて（実際にはいずれか）を呼び出したかどうか

かに関係なく、常に完了します。これが、前節で説明した主問い合わせがその出力を要求した時にSELECTの実行が行われるというWITH内のSELECTについての規則と異なることに注意してください。

WITH内の副文はそれぞれと主問い合わせで同時に実行されます。したがってWITH内のデータ変更文を使用する時、指定したデータ変更文が実際に実行される順序は予測できません。すべての文は同じスナップショット（[第13章](#)参照）を用いて実行されます。このため互いが対象テーブルに行った影響を"見る"ことはできません。これは、行の更新に関する実際の順序が予測できないという影響を軽減し、RETURNINGデータが別のWITH副文と主問い合わせとの間で変更を伝える唯一の手段であることを意味します。この例を以下に示します。

```
WITH t AS (  
    UPDATE products SET price = price * 1.05  
    RETURNING *  
)  
SELECT * FROM products;
```

外側のSELECTはUPDATEの動作前の元々の価格を返します。

```
WITH t AS (  
    UPDATE products SET price = price * 1.05  
    RETURNING *  
)  
SELECT * FROM t;
```

一方こちらでは外側のSELECTは更新されたデータを返します。

単一の文で同じ行を2回更新する試行はサポートされていません。1つの変更のみが行われますが、どちらかを確実に予測することは簡単ではありません（場合によっては不可能です）。これはまた、同じ文内ですでに更新された行を削除する場合でも当てはまります。したがって一般的には単一の文で1つの行を2回変更しようと試みることを避けなければなりません。具体的には主文または同レベルの副文で変更される行と同じ行に影響を与えるWITH副文を記述することは避けてください。こうした文の影響は予測することはできません。

現状、WITH内のデータ変更文の対象として使用されるテーブルはすべて、複数の文に展開される条件付きルール、ALSOルール、INSTEADルールを持つてはなりません。

[前のページ](#)
VALUES リスト

[ホーム](#)
[上に戻る](#)

[次のページ](#)
データ型

第 8 章 データ型

目次

8.1. [数値データ型](#)

- 8.1.1. [整数データ型](#)
- 8.1.2. [任意の精度を持つ数](#)
- 8.1.3. [浮動小数点データ型](#)
- 8.1.4. [連番型](#)

8.2. [通貨型](#)

8.3. [文字型](#)

8.4. [バイナリ列データ型](#)

- 8.4.1. [bytea hex書式](#)
- 8.4.2. [bytea エスケープ書式](#)

8.5. [日付/時刻データ型](#)

- 8.5.1. [日付/時刻の入力](#)
- 8.5.2. [日付/時刻出力](#)
- 8.5.3. [時間帯](#)
- 8.5.4. [時間間隔入力](#)
- 8.5.5. [時間間隔出力](#)

8.6. [論理値データ型](#)

8.7. [列挙型](#)

- 8.7.1. [列挙型の宣言](#)
- 8.7.2. [順序](#)
- 8.7.3. [型の安全性](#)
- 8.7.4. [実装の詳細](#)

8.8. [幾何データ型](#)

- 8.8.1. [座標点](#)
- 8.8.2. [線分](#)
- 8.8.3. [矩形](#)
- 8.8.4. [経路](#)
- 8.8.5. [多角形](#)
- 8.8.6. [円](#)

8.9. [ネットワークアドレス型](#)

- 8.9.1. [inet](#)
- 8.9.2. [cidr](#)
- 8.9.3. [inet と cidr データ型の違い](#)
- 8.9.4. [macaddr](#)

8.10. [ビット列データ型](#)

8.11. [テキスト検索に関する型](#)

- 8.11.1. [tsvector](#)
- 8.11.2. [tsquery](#)

8.12. [UUID型](#)

8.13. [XML型](#)

- 8.13.1. [XML値の作成](#)
- 8.13.2. [符号化方式の取扱い](#)
- 8.13.3. [XML値へのアクセス](#)

8.14. [JSONデータ型](#)

8.15. [配列](#)

- 8.15.1. [配列型の宣言](#)
- 8.15.2. [配列の値の入力](#)
- 8.15.3. [配列へのアクセス](#)
- 8.15.4. [配列の変更](#)
- 8.15.5. [配列内の検索](#)
- 8.15.6. [配列の入出力構文](#)

8.16. [複合型](#)

- 8.16.1. [複合型の宣言](#)
- 8.16.2. [複合型の値の入力](#)
- 8.16.3. [複合型へのアクセス](#)
- 8.16.4. [複合型の変更](#)
- 8.16.5. [複合型の入出力構文](#)

8.17. [範囲型](#)

- 8.17.1. [組み込みの範囲型](#)
- 8.17.2. [例](#)
- 8.17.3. [閉じた境界と開いた境界](#)
- 8.17.4. [無限の\(境界のない\)範囲](#)
- 8.17.5. [範囲の入出力](#)
- 8.17.6. [範囲の生成](#)
- 8.17.7. [離散的な範囲型](#)
- 8.17.8. [新しい範囲型の定義](#)
- 8.17.9. [インデックス](#)
- 8.17.10. [範囲の制約](#)

8.18. [オブジェクト識別子データ型](#)

8.19. [疑似データ型](#)

PostgreSQLにはユーザが使用可能な豊富なデータ型が始めから備わっています。 [CREATE TYPE](#) コマンドでPostgreSQLに対し新しいデータ型を追加できます。

[表8-1](#)に組み込みの汎用データ型をすべて示します。 "別名"欄に列挙された代替名称のほとんどは、歴史的な理由によりPostgreSQLの内部で使用されている名前です。 他にも、内部で使用するデータ型、削除予定のデータ型もありますが、ここにはリストされていません。

表 8-1. データ型

名称	別名	説明
bigint	int8	8バイト符号付き整数
bigserial	serial8	自動増分8バイト整数
bit [(n)]		固定長ビット列
bit varying [(n)]	varbit	可変長ビット列
boolean	bool	論理値 (真/偽)
box		平面上の矩形
bytea		バイナリデータ ("バイトの配列 (byte array) ")
character varying [(n)]	varchar [(n)]	可変長文字列
character [(n)]	char [(n)]	固定長文字列
cidr		IPv4もしくはIPv6ネットワークアドレス
circle		平面上の円
date		暦の日付 (年月日)
double precision	float8	倍精度浮動小数点 (8バイト)
inet		IPv4もしくはIPv6ホストアドレス
integer	int, int4	4バイト符号付き整数
interval [fields] [(p)]		時間間隔
line		平面上の無限直線

lseg		平面上の線分
macaddr		MAC（メディアアクセスコントロール）アドレス
money		貨幣金額
numeric [(<i>p</i> , <i>s</i>)]	decimal [(<i>p</i> , <i>s</i>)]	精度の選択可能な高精度数値
path		平面上の幾何学的経路
point		平面上の幾何学的点
polygon		平面上の閉じた幾何学的経路
real	float4	単精度浮動小数点（4バイト）
smallint	int2	2バイト符号付き整数
smallserial	serial2	自動増分2バイト整数
serial	serial4	自動増分4バイト整数
text		可変長文字列
time [(<i>p</i>)] [without time zone]		時刻（時間帯なし）
time [(<i>p</i>)] with time zone	timetz	時間帯付き時刻
timestamp [(<i>p</i>)] [without time zone]		日付と時刻（時間帯なし）
timestamp [(<i>p</i>)] with time zone	timestampz	時間帯付き日付と時刻
tsquery		テキスト検索問い合わせ
tsvector		テキスト検索文書
txid_snapshot		ユーザレベルのトランザクションIDスナップショット
uuid		汎用一意識別子
xml		XMLデータ
json		JSONデータ

互換性: 次に挙げるデータ型（あるいはその綴り方）はSQLで規定されています。 bigint、bit、bit varying、boolean、char、 character varying、character、varchar、 date、double precision、integer、 interval、numeric、decimal、real、 smallint、time（時間帯付き、なしの両方）、 timestamp（時間帯付き、なしの両方）、xml。

それぞれのデータ型はそのデータ型の入出力関数で決定される外部表現を保有しています。 組み込みデータ型の多くには、はっきりとした外部書式があります。 とは言っても、経路のようなPostgreSQLに特有な型や、あるいは、日付や時刻データ型のように書式を複数選択できる型が多くあります。 一部の入出力関数は転置することができません。 つまり、出力関数による結果は最初の入力と比較した場合精度を失う可能性があります。

8.1. 数値データ型

数値データ型には2、4、8バイト整数と、4、8バイト浮動小数点および精度設定が可能な数があります。 [表8-2](#)に使用可能な型を列挙します。

表 8-2. 数値データ型

型名	格納サイズ	説明	範囲
smallint	2バイト	狭範囲の整数	-32768から+32767
integer	4バイト	典型的に使用する整数	-2147483648から+2147483647
bigint	8バイト	広範囲整数	-9223372036854775808から9223372036854775807
decimal	可変長	ユーザ指定精度、正確	小数点前までは131072桁、小数点以降は16383桁
numeric	可変長	ユーザ指定精度、正確	小数点前までは131072桁、小数点以降は16383桁
real	4バイト	可変精度、不正確	6桁精度
double precision	8バイト	可変精度、不正確	15桁精度
smallserial	2バイト	狭範囲自動増分整数	1から32767
serial	4バイト	自動増分整数	1から2147483647
bigserial	8バイト	広範囲自動増分整数	1から9223372036854775807

数値データ型に対する定数の構文は[項4.1.2](#)で説明しています。数値データ型には対応する算術演算子と関数の一式が揃っています。詳細は[第9章](#)を参照してください。次節でデータ型について詳しく説明します。

8.1.1. 整数データ型

smallint、integer、bigintは各種範囲の整数、つまり小数点以下の端数がない数を保持します。許容範囲から外れた値を保存しようするとエラーになります。

integer型は数値の範囲、格納サイズおよび性能において最も釣合いが取れていますので、一般的に使用されます。smallint型は一般的にディスク容量に制限が付いている場合にのみ使用します。bigint型はintegerの許容範囲では十分ではない場合にのみ使用すべきです。integerデータ型の方がずっと速いからです。

非常にわずかなオペレーティングシステムでbigint型は、8バイト整数をコンパイラがサポートしているかどうかに依存しますので正常に機能するとは限りません。そのようなマシンではbigintはintegerと同じように振舞いますが、領域は8バイトまで必要です。（この状況が当てはまる最近のプラットフォームは把握していません。）

SQLでは整数の型としてinteger（またはint）とsmallint、bigintのみを規定しています。int2、int4およびint8は拡張ですが、いくつか他のSQLデータベースシステムでも使われています。

8.1.2. 任意の精度を持つ数

numeric型は、非常に大きな桁数で数値を格納でき、正確な計算を行えます。通貨金額やその他正確性が求められる数量を保存する時は特に、この型を推奨します。とは言っても、numericの値に対する算術演算の動作は次節で説明する整数型、もしくは浮動小数点データ型に比較し非常に遅くなります。

この後の説明では、次の用語を使用します。numericの位取りとは、小数点の右側の小数点以下の桁数をいいます。numericの精度とは数字全体の有効桁数です。すなわち、小数点をはさんでいる両側の桁数の合計です。そのため、23.5141という数値の精度は6で位取りは4となります。整数の位取りは、ゼロであるとみなすことができます。

numeric列の数値の最大精度と最大位取りはともに設定することができます。列のデータ型をnumericと宣言するには次の構文を使います。

```
NUMERIC (precision, scale)
```

精度は正数、位取りは0もしくは正数でなければなりません。他に

```
NUMERIC (precision)
```

は位取りが0であることを選択します。

NUMERIC

精度または位取りの指定がない場合、実装されている限界の精度まで、いかなる精度あるいは位取りの値も格納できる列が作られます。この類の列はいかなる特定の位取りに対しても入力値を強要しませんが、宣言された位取りを持つnumeric列は入力値にその位取りを強要します。（標準SQLはデフォルトとして位取り0を要求していて、整数に対する厳密性を強制しています。しかし、この方法はあまり役に立たないと思われる。もし移植性を心配するなら、常に精度と位取りを明示的に設定してください。）

注意: 明示的に型宣言で指定される場合に許される最大精度は1000です。精度の指定がないNUMERICは表8-2で説明する制限に従います。

格納される値の位取りが宣言された列の位取りより大きかった場合、システムは指定された小数部の桁まで値を丸めます。そして、小数点の左側の桁数が、宣言された精度から宣言された位取りを差し引いた数を超える場合にエラーとなります。

数値は物理的に先頭や末尾に0を付与されることなく格納されます。したがって、列の宣言された精度と位取りは最大であり、固定的に割り当てられていません。（この意味ではnumericはchar(n)よりもvarchar(n)に似ています。）実際の格納に必要な容量は、10進数4桁のそれぞれのグループに対して2バイトと、3から8バイトのオーバヘッドです。

通常の数値に加え、numeric型は、"非数値"を意味するNaNという特別な値を取ることができます。NaNに対する操作はすべて、別のNaNを生成します。この値をSQLコマンドの定数として記述する場合は、例えばUPDATE table SET x = 'NaN'のように、引用符でくくられなければなりません。入力の際は、NaNという文字列は大文字小文字の区別なく認識されます。

注意: ほとんどの"非数"の実装において、NaNは（NaNを含む）他の数値と等価になるとみなされていません。numeric値をソートできる、また、ツリーを基にしたインデックスでできるように、PostgreSQLはNaN同士は等しく、すべてのNaN以外の値よりも大きな値となるものとして扱います。

decimalとnumeric型は等価です。2つのデータ型はともに標準SQLに従っています。

8.1.3. 浮動小数点データ型

realとdouble precisionは不正確な可変精度の数値データ型です。実際にはこれらのデータ型は、使用しているプロセッサ、オペレーティングシステムおよびコンパイラがサポートしていれば、通常は（それぞれ単精度および倍精度の）バイナリ浮動小数点演算用のIEEE規格754の実装です。

不正確というのは、ある値はそのまま内部形式に変換されずに近似値として保存されるということです。ですから、保存しようとする値と抽出しようとする値の間に多少の差異が認められます。これらのエラーを管理し計算によって補正をどうするかについては、数学とコンピュータ科学の系統すべてに関わることで、以下の点を除き触れません。

- （金銭金額など）正確な記録と計算が必要な時は代わりにnumericを使用してください。
- これらのデータ型で何か重要な件に対し複雑な計算を必要とする時、特に（無限大やアンダーフローのような）境界線におけるある種の振舞いについて信頼を置かなければならないのであれば、実装を注意深く検証しなければなりません。
- 2つの浮動小数点値が等価であるのかどうかの比較は予想通りに行かない時もあります。

ほとんどのプラットフォームではrealは最低6桁の精度を持った少なくとも-1E+37と+1E+37の範囲です。double precisionは通常最低15桁の精度でおよそ-1E+308と+1E+308の範囲です。大き過ぎたり小さ過ぎる値はエラーの原因となります。入力値の精度が高過ぎる場合は丸められることがあります。ゼロに限りなく近い値で、しかもゼロとは区別されているようにみなされない数値はアンダーフローエラーを引き起こします。

通常の数値に加え、浮動小数点型では以下の特殊な値を取ります。

Infinity
-Infinity
NaN

これらはそれぞれ、IEEE 754の特殊な値、"無限大"、"負の無限大"、"非数値"を表します。（浮動小数点計算がIEEE 754に従わないマシンでは、これらの値はおそらく正しく動作しません。）これらの値をSQLコマンドの定数として記述する場合、例えばUPDATE table SET x = 'Infinity'のように引用符でくる必要があります。入力の際、これらの文字列は大文字小文字の区別なく認識されます。

注意: IEEE754では、NaNは（NaNを含む）他のすべての浮動小数点値と比べた時に不等でなければならないと規定しています。浮動小数点値をソートできる、また、ツリーを基にしたインデックスで使えるように、PostgreSQLはNaN同士は等しく、すべてのNaN以外の値よりも大きな値となるものとして扱います。

また、PostgreSQLでは不正確な数値型を規定する標準SQLのfloatとfloat(p)をサポートしています。ここで、pは2進数桁数で最低受け付ける精度を指定します。PostgreSQLはfloat(1)からfloat(24)をrealを選択するものとして受け付

け、`float(25)` から `float(53)` を `double precision` を選択するものとして受け付けます。許容範囲外の p の値はエラーになります。精度指定のない `float` は `double precision` として解釈されます。

注意: 7.4より前のPostgreSQLでは、`float(p)` の精度は10進数桁数として解釈されました。これは、2進数桁数の精度を規定する標準SQLに一致するように変更されたものです。`real` と `double precision` の仮数がそれぞれ24ビットと53ビットであるという前提は、IEEE標準浮動小数点の実装では正しいものです。非IEEEのプラットフォームでは、一部無効になる可能性があります。単純化のためにすべてのプラットフォームで p の範囲は同一です。

8.1.4. 連番型

`smallserial`、`serial` および `bigserial` データ型は正確にはデータ型ではなく、テーブルの列に一意の識別子を作成する簡便な表記法です（他のデータベースでサポートされる `AUTO_INCREMENT` プロパティに似ています）。現在の実装では、

```
CREATE TABLE tablename (  
    colname SERIAL  
);
```

は以下を指定することと同じです。

```
CREATE SEQUENCE tablename_colname_seq;  
CREATE TABLE tablename (  
    colname integer NOT NULL DEFAULT nextval('tablename_colname_seq')  
);  
ALTER SEQUENCE tablename_colname_seq OWNED BY tablename.colname;
```

このように整数列を作成し、その列のデフォルト値が連番ジェネレータから割り当てられるようにしました。また、`NOT NULL` 制約を適用することによって、`NULL` 値が挿入されないようにします。（たいていの場合は、重複する値を間違って挿入しないように、`UNIQUE` 制約または `PRIMARY KEY` 制約も追加することが推奨されますが、これは自動的には行われません。）最後に、シーケンスは列に"より所有"されるものと印が付きます。したがって、テーブルの列が削除された場合にシーケンスは削除されます。

注意: `smallserial`、`serial` および `bigserial` はシーケンスを使って実装されているため、行の削除が行われていなくとも、列に"穴"や連番の抜けが発生するかもしれません。また、テーブルへ正常に挿入されていないにも関わらず、シーケンスの値を"消費してしまう"こともあります。これは、例えば挿入したトランザクションがロールバックされた時に発生することがあります。詳細は[項9.16](#)の `nextval()` を参照してください。

注意: PostgreSQL 7.3より前では、`serial` は `UNIQUE` を意味していました。現在では自動的には行われません。連番の列に一意性制約もしくはプライマリキーを付与したい場合は、他のデータ型同様指定しなければなりません。

`serial` 列にシーケンスの次の値を挿入するには、`serial` 列にそのデフォルト値を割り当てるよう指定してください。これは、`INSERT` 文の列リストからその列を除外する、もしくは `DEFAULT` キーワードを使用することで行うことができます。

`serial` と `serial4` という型の名称は等価です。ともに `integer` 列を作成します。`bigserial` と `serial8` という型の名称も `bigint` 列を作成することを除いて同じ振舞いをします。もしテーブルを使用する期間で 2^{31} 以上の識別子を使用すると予測される場合、`bigserial` を使用しなければいけません。`smallserial` と `serial2` という型の名称もまた、`smallint` 列を作成することを除いて同じ振舞いをします。

`serial` 列用に作成されたシーケンスは、それを所有する列が削除された時に自動的に削除されます。列を削除せずにシーケンスを削除することができますが、これにより強制的に列のデフォルト式が削除されます。

[前のページ](#)
データ型

[ホーム](#)
[上に戻る](#)

[次のページ](#)
通貨型

8.2. 通貨型

`money` 型は貨幣金額を固定精度の小数点で格納します。 [表8-3](#)を参照してください。 小数点精度はデータベースの`lc_monetary`設定で決定されます。この表が示すように範囲は小数点2桁を想定しています。 `'$1,000.00'`などの典型的な通貨書式の他、整数、浮動小数点リテラルなど様々な書式の入力を受け付けます。 出力形式は通常は"典型的な"通貨書式となりますが、ロケールによって異なります。

表 8-3. 通貨型

型名	格納サイズ	説明	範囲
money	8バイト	貨幣金額	-92233720368547758.08 から +92233720368547758.07

このデータ型の出力はロケールにより変動しますので、`lc_monetary`設定が異なるデータベースに`money`データをロードする場合には動作しない可能性があります。 この問題を防ぐためには、ダンプを新しいデータベースにリストアする前に、`lc_monetary`がダンプを行ったデータベースと同じまたは等価であることを確認してください。

`numeric`、`int`そして`bigint`型は`money`型にキャストすることができます。 `real`型や`double precision`型は最初に`numeric` 型にキャストした後に行なう必要があります。以下に例を示します。

```
SELECT '12.34'::float8::numeric::money;
```

しかしこれは推奨されません。浮動小数点数値は丸めの可能性がありますので貨幣を扱うために使用すべきではありません。

`money`型の値は精度を落とすことなく`numeric`にキャストすることができます。 他の型への変換では精度を落とす可能性がありますので、2段階で行う必要があります。

```
SELECT '52093.89'::money::numeric::float8;
```

`money`型の値を別の`money`型の値で除算すると、結果は`double precision`型（通貨ではなく純粋な数値）になります。 除算では通貨の単位は相殺されます。

8.3. 文字型

表 8-4. 文字型

型名	説明
<code>character varying(n)</code> 、 <code>varchar(n)</code>	上限付き可変長
<code>character(n)</code> 、 <code>char(n)</code>	空白で埋められた固定長
<code>text</code>	制限なし可変長

表8-4はPostgreSQLで使用可能な汎用文字型を示したものです。

SQLは2つの主要な文字データ型を定義しています。 `character varying(n)` と `character(n)` です。ここで n は正の整数です。これらのデータ型は2つとも n 文字長（バイト数ではなく）までの文字列を保存できます。超過している文字がすべて空白の場合を除いて、上限を越えた文字列をこの型の列に保存しようとするとエラーになります。すべて空白の場合は長さの限界で切り捨てられます。（この一風変わった例外は標準SQLで要求されています。）もし宣言された上限よりも文字列が短い時は `character` の値は空白で埋められ、`character varying` の値は単にその短い文字列で保存されます。

明示的に値を `character varying(n)` または `character(n)` にキャストした場合、指定長を超えるとエラーなしで n 文字まで切り詰められます。（これもまた標準SQLの仕様です。）

`char(n)` および `varchar(n)` という表記法はそれぞれ `character(n)` と `character varying(n)` の別名です。長さ指定がない `character` は `character(1)` と同じです。`character varying` が長さ指定なしで使われた時は、いかなる長さの文字列でも受け付けます。後者はPostgreSQLの拡張です。

さらにPostgreSQLは、いかなる長さの文字列でも格納できる `text` をサポートします。`text` 型は標準SQLにはありませんが、多くの他のSQLデータベース管理システムも同様にサポートしています。

`character` 型の値は、指定長 n になるまで物理的に空白で埋められ、そのまま格納、表示されます。しかし、追加された空白は意味的に重要ではないものとして扱われます。最後の空白は、2つの `character` 型の値を比べる際には無視されます。また、`character` 値を他の文字列型に変換する際には除去されます。`character varying` 型と `text` 型の値では、`LIKE` や正規表現でのパターンマッチを行なう際、最後の空白は意味的に重要なものですので、注意してください。

短い文字列（126バイトまで）の保存には、実際の文字列に1バイト加えたサイズが必要です。`character` では空白埋め込み分が加わります。より長い文字列では1バイトではなく4バイトのオーバーヘッドになります。長い文字列はシステムにより自動的に圧縮されますので、ディスク上の物理的必要容量サイズはより小さくなるかもしれません。また、非常に長い値はより短い列の値への高速アクセスに干渉しないように、バックグラウンドテーブルに格納されます。いずれの場合にあっても保存できる最長の文字列は約1ギガバイトです。（データ型宣言に使われる n に許される最大値はこれより小さいものです。マルチバイト文字符号化方式においては文字数とバイト数はまったく異なっているため、この値の変更は便利ではありません。特定の上限を設けずに長い文字列を保存したい場合は、任意の上限を設けるよりも長さの指定がない `text` もしくは `character varying` を使用してください。）

ティップ: 空白で埋められる型を使用した場合の保存領域の増加、および、長さ制限付きの列に格納する際に長さを検査するためにいくつか余計なCPUサイクルが加わる点を別にして、これら3つの型の間で性能に関する差異はありません。他の一部のデータベースシステムでは `character(n)` には性能的な優位性がありますが、PostgreSQLではこうした利点はありません。実際には、格納の際に追加のコストがあるため、`character(n)` は3つの中でもっとも低速です。

文字列リテラルの構文については[項4.1.2.1](#)、利用可能な演算子と関数については[第9章](#)を参照してください。データベースの文字セットは、テキストの値を格納する時に使用される文字セットを決定します。文字セットのサポートに関する詳細については[項22.3](#)を参照してください。

例 8-1. 文字データ型を使って

```
CREATE TABLE test1 (a character(4));
INSERT INTO test1 VALUES ('ok');
SELECT a, char_length(a) FROM test1; -- (1)
 a | char_length
----+-----
ok |          2

CREATE TABLE test2 (b varchar(5));
INSERT INTO test2 VALUES ('ok');
INSERT INTO test2 VALUES ('good');
```



```
INSERT INTO test2 VALUES ('too long');
ERROR:  value too long for type character varying(5)
INSERT INTO test2 VALUES ('too long'::varchar(5)); -- 明示的な切り捨て
SELECT b, char_length(b) FROM test2;
  b      | char_length
-----+-----
ok       |          2
good     |          5
too l    |          5
```

(1) char_length関数は項9.4で説明されています。

PostgreSQLには、表8-5に示すように、この他2つの固定長文字型があります。name型は内部のシステムカタログ内の識別子の格納のためにのみ存在するもので、一般ユーザによって使用されることを意図していません。現在長さは64バイト（63バイトの利用可能文字と終止文字）と定義されていますが、CソースコードにあるNAMEDATALEN定数を使って参照される必要があります。この長さはコンパイル時に設定されます（そのため特別な用途に合わせ調整できます）。デフォルトの最大長は今後のリリースで変更される可能性があります。"char"（二重引用符に注意）は、char(1)とは異なり、1バイトの領域しか使用しません。過度に単純化した列挙型としてシステムカタログで内部的に使用されます。

表 8-5. 特別な文字データ型

型名	格納サイズ	説明
"char"	1バイト	単一バイト内部データ型
name	64バイト	オブジェクト名用の内部データ型

8.4. バイナリ列データ型

bytea データ型はバイナリ列の保存を可能にします。表8-6を参照してください。

表 8-6. バイナリ列データ型

型名	格納サイズ	説明
bytea	1または4バイトと実際のバイナリ列の長さ	可変長のバイナリ列

バイナリ列はオクテット（またはバイト）の連続です。バイナリ列は2つの方法で文字列と区別されます。1点目は、バイナリ列はゼロの値のオクテットと他の"表示できない"オクテット（通常32から126の範囲外のオクテット）を保存できるということです。文字列ではゼロというオクテットは使用できません。また、データベースで選択している文字セット符号化方式で無効なオクテット値やオクテット値の並びも使用できません。2点目は、バイナリ列を演算すると実際のバイトが処理されるのに対して、文字列の処理はロケール設定に従うということです。まとめると、バイナリ列はプログラマが"バイト列そのもの"と考えるものを格納するのに適し、文字列はテキストを格納するのに適しています。

bytea 型は入出力用に2つの外部書式をサポートします。PostgreSQLの歴史的な"エスケープ"書式と"hex"です。入力ではこれらの両方とも常に受け入れられます。出力書式はbytea_output設定パラメータに依存し、デフォルトではhexです。（hex書式はPostgreSQL 9.0から導入されたものであることに注意してください。以前のバージョンや一部のツールではこれを理解しません。）

標準SQLは、BLOBまたはBINARY LARGE OBJECTという、異なるバイナリ列型を定義します。入力書式はbyteaと異なりますが、提供される関数および演算子はほぼ同じです。

8.4.1. bytea hex書式

"hex"書式ではバイナリデータをバイト単位で上位の1/2バイトを先頭に2桁の16進数に符号化します。（エスケープ書式と区別するために）文字列全体は\xという並びの後に付けられます。一部の文脈では、エスケープ書式でバックスラッシュを二重にしなければならない一部の場合において、先頭のバックスラッシュを二重にしてエスケープさせる必要があるかもしれません。詳細に付いては後で説明します。16進数の桁は大文字でも小文字でも構いません。桁の組み合わせと次の組み合わせの間に空白文字を入れることができます。（しかし桁の組み合わせの間や先頭の\xの間には入れることはできません。）hex書式は外部のアプリケーションおよびプロトコルの間で広く互換性を持ち、また、エスケープ書式と比べ変換が高速になる傾向があります。このため使用が好まれます。

例

```
SELECT E'\x00DEADBEEF';
```

8.4.2. bytea エスケープ書式

"エスケープ"書式はbytea 型用の伝統的なPostgreSQLの書式です。これは、バイナリ列をASCII文字の並びとして表現しASCII文字として表現できないバイトは特殊なエスケープシーケンスとして表現するという方式を取ります。アプリケーションの見地から文字として表現されたバイトが有意であれば、この表現は簡便です。しかし現実にはバイナリ列と文字列の間の区別があいまいになりますので、通常は混乱します。また特に選択されたエスケープ機構は多少非効率的です。このためこの書式はおそらくほとんどの新しいアプリケーションでは避けるべきでしょう。

エスケープ書式でbytea 値を入力する際に、特定の値のオクテットをエスケープする必要があります。なお、すべてのオクテットの値をエスケープすることができます。一般的にあるオクテットをエスケープするには、それをその3桁の8進番号に変換し、バックスラッシュ(エスケープ文字列構文を使用してリテラルとして値を記述する場合は2つのバックスラッシュ)を前に付けます。他にもバックスラッシュ自体(オクテット値92)を二重のバックスラッシュとして表現することができます。表8-7には、エスケープする必要がある文字と、その適用可能な代替エスケープシーケンスを示しています。

表 8-7. オクテットをエスケープしたbytea リテラル

10進オクテット値	説明	エスケープされた入力表現	例	出力表現
0	ゼロオクテット	E'\000'	SELECT E'\000'::bytea;	\000
39	単一引用符	''もしくはE'\047'	SELECT E'\'::bytea;	'
92	バックスラッシュ	E'\\\'もしくはE'\134'	SELECT E'\\\\'::bytea;	\\

0 to 31 and 127 to 255	"表示できない"オクテット	E'¥¥xxx' (8進数)	SELECT E'¥¥001'::bytea;	¥001

実際には、表示できないオクテットに対するエスケープ要求はロケールの設定に依存して変化します。 インスタンスによっては、エスケープをしないで済むこともあります。 [表8-7](#)の例の各結果は、出力表現が時として1文字以上であっても、長さは正確に1オクテットであることに注意してください。

[表8-7](#)で示したように、複数バックスラッシュが必要な理由は、文字列リテラルとして記述された入力文字列は、PostgreSQLサーバ上で2つの解析段階を通過する必要があることです。 各組み合わせの最初のバックスラッシュは文字列リテラル用パーサでエスケープ文字と解釈（エスケープ文字列構文の使用を前提）され、2番目のバックスラッシュを残して、そこで消費されます。（この段階のエスケープを避けるためにドル引用符による文字列を使用することができます。） 残りのバックスラッシュは、bytea入力関数が3桁のオクテット値の先頭に付く記号、もしくは、他のバックスラッシュをエスケープする記号として認識します。 例えば、E'¥¥001'としてサーバに渡された文字列リテラルは、エスケープ文字列パーサを通過した後¥001のようになります。 ¥001はその後bytea入力関数に送られ、10進数値1の1つのオクテットに変換されます。 単一引用符文字はbyteaでは特別に扱われず、通常の文字列リテラルの規則に従うことに注意してください。（[項4.1.2.1](#)も参照してください。）

また、Byteaオクテットは出力時も時々エスケープされます。 一般的には、各"表示できない"オクテットは、1つのバックスラッシュの後に等価な3桁8進数という形に変換されます。 ほとんどの"表示できる"オクテットは、クライアント文字セットにおける標準表現で表されます。 10進数で92となるオクテット（バックスラッシュ）は出力では二重になります。 詳細は[表8-8](#)に示します。

表 8-8. bytea出力のエスケープされたオクテット

10進オクテット値	説明	エスケープされた出力表現	例	出力結果
92	バックスラッシュ	¥¥	SELECT E'¥¥134'::bytea;	¥¥
0から31および127から255	"表示できない"オクテット	¥xxx (8進数)	SELECT E'¥¥001'::bytea;	¥001
32から126	"表示できる"オクテット	クライアント文字セットにおける表現	SELECT E'¥¥176'::bytea;	~

使用するPostgreSQLのフロントエンドによっては、bytea文字列をエスケープまたはアンエスケープする際に、追加的な作業が必要になることがあります。 例えば、使用するインタフェースが改行文字や復帰文字を自動的に翻訳してしまう場合、これらの文字もエスケープしなければならないかもしれません。

8.5. 日付/時刻データ型

PostgreSQLでは、[表8-9](#)に示されているSQLの日付と時刻データ型のすべてがサポートされています。これらのデータ型で利用できる演算子については[項9.9](#)で説明します。グレゴリオ暦が導入される前の年であっても（[項B.4](#)参照）、日付はグレゴリオ暦にしたがって計算されます。

表 8-9. 日付/時刻データ型

型名	格納サイズ	説明	最遠の過去	最遠の未来	精度
timestamp [(p)] [without time zone]	8 バイト	日付と時刻両方（時間帯なし）	4713 BC	294276 AD	1μ秒、14桁
timestamp [(p)] with time zone	8バイト	日付と時刻両方、時間帯付き	4713 BC	294276 AD	1μ秒、14桁
date	4バイト	日付（時刻なし）	4713 BC	5874897 AD	1日
time [(p)] [without time zone]	8バイト	時刻（日付なし）	00:00:00	24:00:00	1μ秒、14桁
time [(p)] with time zone	12バイト	その日の時刻のみ、時間帯付き	00:00:00+1459	24:00:00-1459	1μ秒、14桁
interval [fields] [(p)]	12バイト	時間間隔	-178000000年	178000000年	1μ秒、14桁

注意: 標準SQLでは、単なるtimestampという記述はtimestamp without time zoneと同じであることを要求します。PostgreSQLはこれに準じます。（7.3より以前では、これをtimestamp with time zoneとして扱いました。）timestamp with time zoneはtimestamp_{tz}と省略することが許容されています。これはPostgreSQL独自の拡張です。

time、timestampおよびintervalは秒フィールドに保有されている小数点以下の桁数を指定するオプションの精度値であるpを受け付けます。デフォルトでは、明示的な精度に対する限界はありません。pの許容範囲はtimestamp型とinterval型の場合は0から6です。

注意: timestampの値が8バイト整数（現在のデフォルト）で格納されていれば、すべての値についてμ秒精度が有効です。timestampの値が倍精度浮動小数点数（将来のサポートが保証されないコンパイル時のオプション）で格納されていると、有効な精度は6より小さいかもしれません。timestampの値は2000-01-01深夜を基準にした経過秒数として格納されます。timestampの値が浮動小数点数として格納されていれば、2000-01-01から数年の範囲でμ秒精度が得られますが、それより離脱すると精度は劣化します。浮動小数点datetimesを使用すると上で示した範囲より広いtimestampの値（4713 BCから5874897 ADまで）が認められます。

同じコンパイル時オプションはtimeおよびintervalの値が浮動小数点数か、8バイト整数のいずれかによって格納されるかを決定します。浮動小数点の場合、大きなintervalでは間隔が増加する際に精度が落ちます。

time型で8バイト整数が使用された場合には、pは0から6までの範囲で使用可能です。また、浮動小数点を使用される場合には0から10までが使用可能です。

intervalデータ型には追加のオプションがあり、以下の1つの語句を使用して格納されるフィールドの集合を制約します。

```
YEAR
MONTH
DAY
HOUR
MINUTE
SECOND
YEAR TO MONTH
DAY TO HOUR
DAY TO MINUTE
DAY TO SECOND
HOUR TO MINUTE
HOUR TO SECOND
MINUTE TO SECOND
```

fieldsおよびpが共に指定されると、精度は秒のみに適用されるので、fieldsはSECONDを含まなければならないことに注意してください。

time with time zoneは標準SQLで定義されていますが、その定義は、その有用性を疑問視することになりかねない特性を示しています。ほとんどの場合、date、time、timestamp without time zone、timestamp with time zoneの組み合わせで、すべてのアプリケーションで要求される日付/時刻機能すべてを提供しているはずです。

abstimeとreltimeは精度の低いデータ型で、内部で使用されます。アプリケーションでこれらの型の使用を避けてください。これら内部型は今後のリリースで削除される可能性があります。

8.5.1. 日付/時刻の入力

日付と時刻の入力は、ISO 8601、SQL互換、伝統的なPOSTGRES、その他を含むほとんどの適正とみなされる書式を受け付けます。日付の入力における日-月-年の順序のようないずれとも解釈されるいくつかの書式については、それらのフィールドを好きな順序に指定できるようになっています。DateStyleパラメータをMDYに設定すれば、月日年という順で解釈され、DMYに設定すれば日月年という順で、YMDに設定

すれば年月日という順で解釈されます。

PostgreSQLは日付/時刻の運用において標準SQLの要求よりも柔軟です。日付/時刻の入力における正確な構文解析規則と、月および週、そして時間帯を含む使用可能なテキストフィールドに関しては[付録B](#)を参照してください。

テキスト文字列のように、日付や時刻リテラルは単一引用符で囲む必要があることを思い出してください。詳細は[項4.1.2.7](#)を参照してください。SQLでは下記の構文が必要です。

```
type [ (p) ] 'value'
```

ここで、*p*は秒フィールドの小数点以下の桁数を与えるオプションの精度の指定です。精度はtime、timestampおよびinterval型に対して設定できます。値の許容範囲は既に説明しています。定数指定において精度指定がない場合は、デフォルトのリテラル値の精度になります。

8.5.1.1. 日付

[表8-10](#)はdate型で入力可能なものの一部を示します。

表 8-10. 日付入力

例	説明
1999-01-08	ISO 8601。すべてのモードで1月8日になります（推奨書式）。
January 8, 1999	すべてのdatestyle入力モードにおいて曖昧さがありません。
1/8/1999	MDYモードでは1月8日、DMYモードでは8月1日。
1/18/1999	MDYモードでは1月18日、他のモードでは拒絶されます。
01/02/03	MDYモードでは2003年1月2日、DMYモードでは2003年2月1日、YMDモードでは2001年2月3日。
1999-Jan-08	すべてのモードで1月8日になります。
Jan-08-1999	すべてのモードで1月8日になります。
08-Jan-1999	すべてのモードで1月8日になります。
99-Jan-08	YMDモードで1月8日、他のモードではエラー。
08-Jan-99	1月8日。ただしYMDモードではエラー。
Jan-08-99	1月8日。ただしYMDモードではエラー。
19990108	ISO 8601。すべてのモードで1月8日になります。
990108	ISO 8601。すべてのモードで1月8日になります。
1999.008	年とその日までの累計
J2451187	ユリウス日
January 8, 99 BC	西暦紀元前99年

8.5.1.2. 時刻

ある一日の時刻を表す型はtime [(p)] without time zoneとtime [(p)] with time zoneです。time単独ではtime without time zoneと同じです。

これらの型への有効な入力は、時刻、その後にオプションで時間帯からなります。（[表8-11](#)と[表8-12](#)を参照してください。）time without time zoneへの入力に時間帯が指定された場合、時間帯は警告なく無視されます。また、America/New Yorkなど夏時間規則を含む時間帯名を使用していない限り、日付を指定することはできませんが、これは無視されます。この場合、標準が夏時間かどちらを適用するかを決定できるように、日付の指定が必要です。適切な時間帯オフセットはtime with time zone型の値に記録されています。

表 8-11. 時刻入力

例	説明
04:05:06.789	ISO 8601
04:05:06	ISO 8601
04:05	ISO 8601
040506	ISO 8601
04:05 AM	04:05と同じ。AMは値に影響を与えない。
04:05 PM	16:05と同じ。時の入力は12以下でなければなりません。
04:05:06.789-8	ISO 8601
04:05:06-08:00	ISO 8601
04:05-08:00	ISO 8601
040506-08	ISO 8601

04:05:06 PST	省略形による時間帯の指定。
2003-04-12 04:05:06 America/New_York	名前による時間帯の指定。

表 8-12. 時間帯入力

例	説明
PST	省略形（米国太平洋標準時間）
America/New_York	完全な時間帯名
PST8PDT	POSIX書式の time zone 仕様
-8:00	ISO 8601。PST用のオフセット
-800	ISO 8601。PST用のオフセット
-8	ISO 8601。PST用のオフセット
zulu	UTC用の軍事用略記
z	zuluの略記

時間帯の指定方法に関する詳細は[項8.5.3](#)を参照してください。

8.5.1.3. タイムスタンプ

タイムスタンプ型への有効な入力は、日付と時刻を連結し、さらにその後にオプションで時間帯、その後にオプションで^{AD}もしくは^{BC}からなります。（他に^{AD}/^{BC}を時間帯の前に付ける方法もありますが、これは推奨される順序ではありません。）したがって、

```
1999-01-08 04:05:06
と
1999-01-08 04:05:06 -8:00
```

は有効な値で、ISO 8601に準拠しています。また、広く使用されている

```
January 8 04:05:06 1999 PST
という書式もサポートされます。
```

標準SQLでは、`timestamp without time zone`のリテラルと`timestamp with time zone`のリテラルを、"+"もしくは "-" 記号と時刻の後の時間帯補正を付けて区別します。そのため、標準に従うと、

```
TIMESTAMP '2004-10-19 10:23:54'
はtimestamp without time zoneに、
TIMESTAMP '2004-10-19 10:23:54+02'
```

は`timestamp with time zone`になります。PostgreSQLは、その型が決まる前に文字列リテラルの内容を検証しません。そのため上のように`timestamp without time zone`を扱います。リテラルが確実に`timestamp with time zone`として扱われるようにするには、例えば、

```
TIMESTAMP WITH TIME ZONE '2004-10-19 10:23:54+02'
```

のように正しい明示的な型を指定してください。`timestamp without time zone`と決定済みのリテラルでは、PostgreSQLは警告なく時間帯情報をすべて無視します。つまり、結果の値は明示された入力値の日付/時刻フィールドから持ち込まれますが、時間帯の調整はなされません。

`timestamp with time zone`について内部に格納されている値は常にUTCです（協定世界時、歴史的にグリニッジ標準時GMTとして知られています）。時間帯が明示的に指定された入力値は、その時間帯に適したオフセットを使用してUTCに変換されます。入力文字列に時間帯が指定されていない場合は、システムの`TimeZone`パラメータに示されている値が時間帯とみなされ、`timezone`時間帯用のオフセットを使用してUTCに変換されます。

`timestamp with time zone`値が出力されると、この値はUTCから現行の`timezone`に変換され、その時間帯のローカル時間として表示されます。他の時間帯での時間を表示するには、`timezone`を変更するか、あるいは`AT TIME ZONE`構文（[項9.9.3](#)を参照）を使用します。

`timestamp without time zone`と`timestamp with time zone`間の変換では、通常`timestamp without time zone`値は`timezone`ローカル時間としてみなされる、または、指定されるものと想定されます。`AT TIME ZONE`を使用する変換では、異なる時間帯を指定できます。

8.5.1.4. 特殊な値

PostgreSQLでは利便性のために、[表8-13](#)に示されているような特別な日付/時刻入力値をサポートしています。`infinity`と`-infinity`の値は、特別にシステム内部で表現され、変更されずに表示されます。他のものは、単に簡略化された表記で、読み込まれる際には通常の日付/時刻値に変換されます。（特に`now`とその関連文字列は読み込まれるとすぐにその時点の値に変換されます。）これらの値はすべて、SQLコマンドで定数として使う場合は、単一引用符でくくらなければなりません。

表 8-13. 特殊な日付/時刻定数

入力文字列	有効な型	説明
epoch	date, timestamp	1970-01-01 00:00:00+00（Unixシステム時間におけるゼロ）
infinity	date, timestamp	他のすべてのタイムスタンプより将来
-infinity	date, timestamp	他のすべてのタイムスタンプより過去
now	date, time, timestamp	現トランザクションの開始時刻
today	date, timestamp	今日の始まり
tomorrow	date, timestamp	明日の始まり
yesterday	date, timestamp	昨日の始まり
allballs	time	00:00:00.00 UTC

SQL互換の関数である、CURRENT_DATE、CURRENT_TIME、CURRENT_TIMESTAMP、LOCALTIME、LOCALTIMESTAMPも、対応するデータ型の日付または時間の値として使用できます。後の4つでは、オプションとして秒以下の精度指定が可能です。（[項9.9.4](#)を参照してください。）これらはSQL関数であり、データ入力文字列として認識されないことに注意してください。

8.5.2. 日付/時刻出力

日付/時刻型の出力書式は、ISO 8601、SQL (Ingres)、伝統的なPOSTGRES（Unix date書式）またはGermanの4つのいずれかに設定されます。デフォルトはISO書式です。（標準SQLではISO 8601書式の使用が定められています。"SQL"という出力書式名は歴史的な事故です。）[表8-14](#)に各出力書式の例を示します。dateとtimeの書式は、言うまでもなく示された例のそれぞれ日付と時刻の部分となります。

表 8-14. 日付/時刻出力形式

様式仕様	説明	例
ISO	ISO 8601/標準SQL	1997-12-17 07:37:16-08
SQL	伝統的な様式	12/17/1997 07:37:16.00 PST
Postgres	特有の様式	Wed Dec 17 07:37:16 1997 PST
German	地域限定様式	17.12.1997 07:37:16.00 PST

注意: ISO 8601の仕様では大文字のTは日付と時間を分割します。PostgreSQLは入力ではこの書式を受け付けますが、上記のように出力ではTではなく空白を使用します。これは読みやすさと他のデータベースシステムと同様のRFC3339に準拠し整合性を保つためです。

SQLとPOSTGRESでは、DMYフィールド順が指定された場合は月の前に日が現れます。指定がなければ日の前に月が現れます。（この設定が入力値の解釈にどう影響を与えるのかについては[項8.5.1](#)を参考にしてください）。[表8-15](#)に例を示します。

表 8-15. 日付の順序の慣習

datestyleの設定	入力の順序	出力例
SQL, DMY	day (日) /month (月) /year (年)	17/12/1997 15:37:16.00 CET
SQL, MDY	month (月) /day (日) /year (年)	12/17/1997 07:37:16.00 PST
Postgres, DMY	day (日) /month (月) /year (年)	Wed 17 Dec 07:37:16 1997 PST

ユーザはSET DATESTYLEコマンド、postgresql.conf構成ファイルのDateStyleパラメータ、そしてサーバクライアントのPGDATESTYLE環境変数を使用して、日付/時刻の様式を選択することができます。

日付/時刻出力のより柔軟な書式設定方法として、書式設定関数to_char（[項9.8](#)を参照）を使用することもできます。

8.5.3. 時間帯

時間帯および時間帯の取り決めは地球の幾何学的要素のみでなく政治的判断に影響されます。世界にまたがる時間帯は1900年代に標準化されたようですが、特に夏時間規則の点で、勝手に変更する傾向が続いています。PostgreSQLは歴史的な時間帯ルールについての情報に、広く使われているzoneinfo時間帯データベースを使用します。将来の時間は、ある与えられた時間帯に対する最新の既知のルールが、将来長きに渡りそのまま遵守が継続されるということを前提としています。

PostgreSQLは汎用的に使用できるように標準SQLへの互換性に対し最大限の努力をしています。しかし、標準SQLには、日付と時刻のデータ型と機能に関する混乱が見受けられます。2つの明らかな問題点を以下に示します。

- date型にはそれに関連する時間帯を持たませんが、time型にはあります。現実の世界において、時間帯のオフセットが夏時間への切り替えにより年間を通じて変化することから、時刻と同様に日付もそれに結び付けられていないと意味がありません。
- デフォルトの時間帯はUTCからの整数定数オフセットとして指定されています。したがってDST（夏時間）への切り替えをまたいで

日付/時刻演算を行う場合、夏時間を適用することは不可能です。

このような問題を解決するためには、時間帯を使用する際に日付と時刻の両方を保持できる日付/時刻データ型を使用することを勧めます。`time with time zone`型の使用はお勧めしません（もっともPostgreSQLでは、旧式のアプリケーションや、標準SQLとの互換性のために、`time with time zone`型の使用をサポートしています）。PostgreSQLは、日付または時刻のみを保持するデータ型のすべては使用中の時間帯であると前提しています。

すべての時間帯付きの日付と時刻はUTCで内部的に保存されます。これらはクライアントに表示される前に[TimeZone](#)設定パラメータで指定された時間帯におけるローカル時間に変換されます。

PostgreSQLでは、3つの形式で時間帯を指定することができます。

- `America/New_York`などの完全な時間帯名称。認識できる時間帯名称は`pg_timezone_names`ビューに列挙されています（[項45.69](#)を参照してください）。PostgreSQLはこの目的のためによく使用されている`zoneinfo`時間帯データを使用します。したがって、多くの他のソフトウェアでも同じ名前が認識されます。
- `PST`などの時間帯省略形。こうした指定は、単に特定のUTCからのオフセットを定義します。一方、完全な時間帯名称では夏時間遷移規則群も組み込まれます。認識可能な省略形は`pg_timezone_abbrevs`ビューに列挙されています（[項45.68](#)を参照してください）。時間帯省略形に対して[TimeZone](#)設定パラメータおよび[log_timezone](#)設定パラメータを設定することはできませんが、日付時刻型の入力値や`AT TIME ZONE`演算子に省略形を使用することができます。
- 時間帯名やその省略形に加え、PostgreSQLは、`STDoffset`や`STDoffsetDST`という形式のPOSIX様式の時間帯指定を受付けます。ここで、`STD`は時間帯省略形、`offset`はUTCから西に数えた時間単位のオフセットです。また、`DST`は省略可能で夏時間時間帯の省略形で、指定オフセットから1時間進むことを意味します。例えば、`EST5EDT`が認識済みの時間帯名でない場合でも、受け付けられ、合衆国東海岸時間と同じものとして動作します。夏時間時間帯名があると、`zoneinfo`時間帯データベースの`posixrules`項目で使用される夏時間変換規則と同じ規則に従って使用されるものと前提されます。標準的なPostgreSQLインストールでは、`posixrules`は`US/Eastern`と同じです。このためPOSIX書式の時間帯指定はUSA夏時間規則に従います。必要に応じて`posixrules`ファイルを置き換えることで、この動作を調整することができます。

一言で言うと、これは省略形と正式名称間の差異です。省略形はUTCから固定したオフセットを表わすのに対して、ほとんどの省略なしの名称はローカルの夏時間規定を意味するので、UTCオフセットには2つの可能性があります。

時間帯省略系の正当性を検査しないため、POSIX書式時間帯機能ではおかしな入力を警告なく受け付けてしまう可能性があることに注意すべきです。例えば、システムの動作はより独特なUTCの省略系を使用し続けた状態で、`SET TIMEZONE TO FOOBAR0`でも動作します。他にも、POSIX時間帯名称では正のオフセットがグリニッジの西側で使用されるという問題には注意しなければなりません。PostgreSQLは、他ではすべてISO-8601規約にしたがい、正の時間帯オフセットはグリニッジの東としています。

すべての場合において、時間帯名は大文字小文字の区別なく認識されます。（これはPostgreSQLのバージョン8.2以前からの変更です。以前はすべてではありませんが、ある文脈では大文字小文字が区別されました。）

省略なし、省略形のどちらもサーバ内に組み込まれるわけではありません。インストールディレクトリの`.../share/timezone/`および`.../share/timezonesets/`の下に保存される構成ファイルから取得されます（[項B.3](#)を参照ください）。

[TimeZone](#)は`postgresql.conf`ファイルや[第18章](#)で説明する他の標準的な方法で設定することができます。以下に、いくつか特別な設定方法を示します。

- `SET TIME ZONE` SQLコマンドはセッションの時間帯を設定します。これはSQL仕様互換の文法により従っている`SET TIMEZONE TO`の別名です。
- `PGTZ`環境変数は、`libpq`クライアントが接続時にサーバに`SET TIME ZONE`コマンドを送信するために用いられます。

8.5.4. 時間間隔入力

`interval`値は以下の冗長な構文を使って記述されます。

```
[@] quantity unit [quantity unit...] [direction]
```

ここで、`quantity`は（符号付き）時間量、`unit`（単位）は`microsecond`、`millisecond`、`second`（秒）、`minute`（分）、`hour`（時）、`day`（日）、`week`（週）、`month`（月）、`year`（年）、`decade`（10年単位）、`century`（100年単位）、`millennium`（1000年単位）あるいはこれらの単位の簡略形または複数形です。`direction`（方向）は`ago`もしくは空です。アットマーク（`@`）はオプションで、付けても付けなくても構いません。異なる単位における時間量は適切に符号を考慮して暗黙的に足されます。`ago`はすべてのフィールドを反転させます。この構文はまた、[IntervalStyle](#)が`postgres_verbose`に設定されている場合に時間間隔の出力でも使用されます。

日、時、分、および秒の時間量は明示的に単位を指定しなくても構いません。例えば、`'1 12:59:10'`は`'1 day 12 hours 59 min 10 sec'`（1日と12時間59分10秒）と解釈されます。また年と月の組み合わせはダッシュを使って指定することができます。例えば、`'200-10'`は`'200 years 10 months'`（200年と10か月）と解釈されます。（実際これらの簡略形は標準SQLで許されたもののみです。そして[IntervalStyle](#)が`sql_standard`に設定されている場合に出力でも使用されます。）

標準の4.4.3.2節の"指定文字付書式"または4.4.3.3節の"代替書式"のどちらかを使用して、時間間隔値はISO 8601時間間隔として書くこともできます。指定文字付の書式は以下のようなものです。

```
P quantity unit [ quantity unit ...] [ T [ quantity unit ...]]
```

文字列は`P`で始まらなければならない、また、一日における時間をもたらし`T`を含めることができます。利用可能な単位の省略形を[表8-16](#)に示

します。単位は省略しても構いません。また任意の順番で指定できます。しかし、1日より小さな単位はTの後に書かなければなりません。特にMの意味はTの前にあるか後にあるかに依存します。

表 8-16. ISO 8601における時間間隔単位の省略形

省略形	意味
Y	年
M	月（日付部分における）
W	週
D	日
H	時間
M	分（時刻部分における）
S	秒

別の書式を示します。

P [years-months-days] [T hours:minutes:seconds]

上の代替書式では、文字列はPから始まらなければなりません。そして、Tは時間間隔の日付部分と時刻部分とを分割します。値はISO 8601日付と同様の数字で指定されます。

fields指定を使って時間間隔定数を記述する場合、または、fields仕様で定義された時間間隔列に文字列を割り当てる場合、マークされていない時間量の解釈はfieldsに依存します。例えばINTERVAL '1' YEARは1年と解釈され、一方でINTERVAL '1'は1秒と解釈されます。同時に、fields仕様によって許可される最下位フィールドの"右側の"フィールド値は警告なしに破棄されます。例えば、INTERVAL '1 day 2:03:04' HOUR TO MINUTEと書くことで、二番目のフィールドは削除されますが、日付フィールドは削除されません。

標準SQLに従うと、時間間隔値のフィールドはすべて同じ符号を持たなければなりません。このため、先頭の負の符号はすべてのフィールドに適用されます。例えば時間間隔リテラル'-1 2:03:04'の負の符号は、日付部分にも時、分、秒部分にも適用されます。PostgreSQLではフィールドに異なる符号を持たせることができます。また伝統的にテキスト形式表現における各フィールドは独立した符号を持つものとして扱われます。このため、この例では時、分、秒部分は正であるとみなされます。IntervalStyleがsql_standardに設定されている場合、先頭の符号はすべてのフィールドに適用されるものとみなされます（ただし他に符号がない場合のみです）。さもなくば、伝統的なPostgreSQLの解釈が使用されます。あいまいさを防ぐために、負のフィールドがある場合には個別に明示的な符号を付けることを勧めます。

内部的には、intervalの値は月数、日数および秒数として格納されています。ある月の日数は月によって変わりますし、夏時間が導入されていると1日が23時間であったり25時間であったりするからです。月数と日数フィールドは整数ですが、秒数フィールドは小数を保管することができます。時間間隔は通常定数列、もしくはtimestampからの引き算で生成されますので、このように保存すればほとんどの場合うまく行きます。justify_daysとjustify_hours関数で通常の範囲を越えた日にちと時刻を調整できます。

冗長な入力書式、および、より簡略な書式の一部のフィールドでは、フィールド値は小数部分を持つことができます。例えば'1.5 week'や'01:02:03.45'です。こうした入力は格納の際適切な月数、日数、秒数に変換されます。これが月または日数が小数になる結果となる場合、小数部分は1月は30日、1日は24時間という変換規則を使用して、より低い順のフィールドに加えられます。例えば'1.5 month'は1月と15日となります。秒だけが出力において小数を示すことになります。

表8-17は有効なinterval入力のいくつかの例を示しています。

表 8-17. 時間間隔入力

例	説明
1-2	標準SQL書式。1年2ヶ月
3 4:05:06	標準SQL書式。3日4時間5分6秒
1 year 2 months 3 days 4 hours 5 minutes 6 seconds	伝統的Postgres書式。1年2月3日4時間5分6秒
P1Y2M3DT4H5M6S	ISO 8601 "指定文字付き書式"。意味は上と同じ
P0001-02-03T04:05:06	ISO 8601 "代替書式"。意味は上と同じ

8.5.5. 時間間隔出力

時間間隔型の出力書式は、SET intervalstyleコマンドを使用して、sql_standard、postgres、postgres_verboseまたはiso_8601の4つのうちの1つを設定できます。デフォルトはpostgres書式です。表8-18はそれぞれの出力形式を示した例です。

sql_standard形式は、時間間隔値が標準制約（構成要素に正負が混在していない年数と月数のみ、または日数と時間のみ）を満足する場合、時間間隔リテラル文字列に対し標準SQLに準拠する出力を作成します。そうでなければ、出力は、正負混在した時間間隔のあいまいさを無くす明示的な符号が付加され、日数-時間数リテラル文字列を伴った標準年数-月数リテラル文字列のようになります。

postgres書式の出力は、DateStyleパラメータがIsoに設定されたとき、8.4以前のリリースと一致します。

postgres_verbose書式の出力は、[DateStyle](#)パラメータがisoに設定されたとき、8.4以前のリリースと一致します。

iso_8601書式の出力はISO 8601 標準の4.4.3.2セクションに記述の"format with designators（指名付き書式）"に一致します。

表 8-18. 時間間隔出力形式の例

形式指定	年-月時間間隔	日-時刻時間間隔	混在した時間間隔
sql_standard	1-2	3 4:05:06	-1-2 +3 -4:05:06
postgres	1 year 2 mons	3 days 04:05:06	-1 year -2 mons +3 days -04:05:06
postgres_verbose	@ 1 year 2 mons	@ 3 days 4 hours 5 mins 6 secs	@ 1 year 2 mons -3 days 4 hours 5 mins 6 secs ago
iso_8601	P1Y2M	P3DT4H5M6S	P-1Y-2M3DT-4H-5M-6S

8.6. 論理値データ型

PostgreSQLでは、標準SQLの`boolean`型が提供されています。 [表8-19](#)を参照してください。 `boolean`型はいくつかの状態を取ることができます。 "真"もしくは"偽"、そして第3の状態はSQLではNULL値で表現される"不明"の状態です。

表 8-19. 論理値データ型

名前	格納サイズ	説明
<code>boolean</code>	1バイト	真または偽の状態

"真"状態に対する有効なリテラル値には次のものがあります。

```
TRUE
't'
'true'
'y'
'yes'
'on'
'1'
```

"偽"状態に対する有効なリテラル値には次のものがあります。

```
FALSE
'f'
'false'
'n'
'no'
'off'
'0'
```

先頭または末尾の空白文字は無視され、大文字小文字の区別は関係ありません。 `TRUE` キーワードと `FALSE` キーワードは（SQL互換の）推奨の使用法です。

[例8-2](#)は、`boolean` 値の出力に `t` および `f` という文字が使用されることを示しています。

例 8-2. `boolean`型を使って

```
CREATE TABLE test1 (a boolean, b text);
INSERT INTO test1 VALUES (TRUE, 'sic est');
INSERT INTO test1 VALUES (FALSE, 'non est');
SELECT * FROM test1;
 a |      b
---+-----
 t | sic est
 f | non est

SELECT * FROM test1 WHERE a;
 a |      b
---+-----
 t | sic est
```

8.7. 列挙型

列挙 (enum) 型は静的、順序付き集合から構成されるデータ型です。これは、多くのプログラミング言語でサポートされているenum型と同じです。列挙型の例として、曜日や個々のデータについての状態値の集合が挙げられます。

8.7.1. 列挙型の宣言

列挙型は[CREATE TYPE](#) コマンドを使用して作成されます。以下に例を示します。

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
```

作成後、他のデータ型とほとんど同じように、列挙型をテーブルや関数定義で使うことができます。

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
CREATE TABLE person (
    name text,
    current_mood mood
);
INSERT INTO person VALUES ('Moe', 'happy');
SELECT * FROM person WHERE current_mood = 'happy';
 name | current_mood
-----+-----
 Moe  | happy
(1 row)
```

8.7.2. 順序

列挙型内の値の順序はその型が作成された時に値を列挙した順番になります。列挙型に対して、すべての比較演算子と関連する集約関数がサポートされます。以下に例を示します。

```
INSERT INTO person VALUES ('Larry', 'sad');
INSERT INTO person VALUES ('Curly', 'ok');
SELECT * FROM person WHERE current_mood > 'sad';
 name | current_mood
-----+-----
 Moe  | happy
 Curly | ok
(2 rows)

SELECT * FROM person WHERE current_mood > 'sad' ORDER BY current_mood;
 name | current_mood
-----+-----
 Curly | ok
 Moe  | happy
(2 rows)

SELECT name
FROM person
WHERE current_mood = (SELECT MIN(current_mood) FROM person);
 name
-----
 Larry
(1 row)
```

8.7.3. 型の安全性

それぞれの列挙型データ型は別個のもので、他の列挙型と比較することはできません。以下の例を参照してください。

```
CREATE TYPE happiness AS ENUM ('happy', 'very happy', 'ecstatic');
CREATE TABLE holidays (
    num_weeks integer,
    happiness happiness
);
INSERT INTO holidays(num_weeks,happiness) VALUES (4, 'happy');
INSERT INTO holidays(num_weeks,happiness) VALUES (6, 'very happy');
INSERT INTO holidays(num_weeks,happiness) VALUES (8, 'ecstatic');
INSERT INTO holidays(num_weeks,happiness) VALUES (2, 'sad');
ERROR:  invalid input value for enum happiness: "sad"
SELECT person.name, holidays.num_weeks FROM person, holidays
WHERE person.current_mood = holidays.happiness;
ERROR:  operator does not exist: mood = happiness
```

もし本当に上のようなことが必要ならば、独自の演算子を作成するか、問い合わせに明示的なキャストを付けることで行うこと

ができます。

```
SELECT person.name, holidays.num_weeks FROM person, holidays
WHERE person.current_mood::text = holidays.happiness::text;
name | num_weeks
-----+-----
Moe  |          4
(1 row)
```

8.7.4. 実装の詳細

列挙型の値はディスク上では4バイトを占めます。 列挙型の値のテキストラベルの長さは、PostgreSQLに組み込まれたNAMEDATALEN設定により制限されます。 標準の構築では、これは最大63バイトを意味します。

列挙型のラベルは大文字小文字を区別します。 このため'happy'は'HAPPY'と同じではありません。 ラベル内の空白文字も意味を持ちます。

列挙型の内部値からテキスト形式のラベルへの変換は、[pg_enum](#)システムカタログ内に保持されます。 このカタログを直接問い合わせることが役に立つ場合があります。

8.8. 幾何データ型

幾何データ型は2次元空間オブジェクトを表現します。表8-20は、PostgreSQLで使用可能な幾何データ型を列挙したものです。一番の基本となる型はpointで、すべての他の型の基礎を形成します。

表 8-20. 幾何データ型

型名	格納サイズ	表現	説明
point	16バイト	平面における座標点	(x,y)
line	32バイト	無限の直線（完全には実装されていません）	((x1,y1),(x2,y2))
lseg	32バイト	有限の線分	((x1,y1),(x2,y2))
box	32バイト	矩形	((x1,y1),(x2,y2))
path	16+16nバイト	閉経路（多角形に類似）	((x1,y1),...)
path	16+16nバイト	開経路	[(x1,y1),...]
polygon	40+16nバイト	多角形（閉経路に類似）	((x1,y1),...)
circle	24バイト	円	<(x,y),r>（中心と半径）

拡大縮小、変形、回転、交点の算出といった様々な幾何学的操作を行う関数と演算子の集合が豊富に揃っています。このことについては項9.11に説明があります。

8.8.1. 座標点

座標点は幾何データ型の基礎となる2次元構成要素です。point型の値は次の構文のいずれかで指定されます。

```
( x , y )
x , y
```

ここでxとyは、それぞれの座標を浮動小数点数数値で表したものです。

座標点は1番目の構文で出力されます。

8.8.2. 線分

線分 (lseg) は座標点の組み合わせで表現されます。lseg型の値は次の構文のいずれかで指定されます。

```
[ ( x1 , y1 ) , ( x2 , y2 ) ]
( ( x1 , y1 ) , ( x2 , y2 ) )
( x1 , y1 ) , ( x2 , y2 )
x1 , y1 , x2 , y2
```

ここで、(x1,y1) と (x2,y2) は線分の終端点です。

線分は1番目の構文で出力されます。

8.8.3. 矩形

矩形は、矩形の対角線の両端の座標点の組み合わせで表されます。box型の値は以下の構文のいずれかで指定されます。

```
( ( x1 , y1 ) , ( x2 , y2 ) )
( x1 , y1 ) , ( x2 , y2 )
x1 , y1 , x2 , y2
```

ここで(x1,y1) と (x2,y2) は矩形の対角線の両端です。

矩形は最初に示した構文で出力されます。

任意の対角頂点を入力として指定することができます。しかし頂点は右上の頂点を最初に、左下の頂点をその後に格納する必要があります。

8.8.4. 経路

経路は接続している座標点のリストで表現されます。 経路は最初の座標点と最後の座標点が接続されていないとみなされる開いている状態か、最初の座標点と最後の座標点が接続されているとみなされる閉じた状態かのいずれかです。

path型の値は次の構文のいずれかで指定されます。

```
[ ( x1 , y1 ) , ... , ( xn , yn ) ]
( ( x1 , y1 ) , ... , ( xn , yn ) )
( x1 , y1 ) , ... , ( xn , yn )
( x1 , y1 , ... , xn , yn )
x1 , y1 , ... , xn , yn
```

ここで、各座標点は、経路を構成する線分の終端点です。 大括弧 ([]) は開経路を、括弧 (()) は閉経路を示します。 3番目から4番目の構文のようにもっとも外側の括弧が省略された場合、閉経路と仮定されます。

経路は最初または 2 番目の適切な構文で出力されます。

8.8.5. 多角形

多角形は座標点（多角形の頂点）のリストで表現されます。 多角形は閉経路ととても良く似ていますが、異なった方式で格納されると同時にそれぞれ独自のサポート関数群を持っています。

polygon型の値は次の構文のいずれかで指定されます。

```
( ( x1 , y1 ) , ... , ( xn , yn ) )
( x1 , y1 ) , ... , ( xn , yn )
( x1 , y1 , ... , xn , yn )
x1 , y1 , ... , xn , yn
```

各座標点は多角形の境界を構成する線分の終端点です。

多角形は最初の構文で出力されます。

8.8.6. 円

円は中心座標点と半径で表現されます。 circle型の値は次の構文のいずれかで指定されます。

```
< ( x , y ) , r >
( ( x , y ) , r )
( x , y ) , r
x , y , r
```

ここで (x,y) は円の中心点、rは円の半径です。

円は最初の構文で出力されます。

8.9. ネットワークアドレス型

PostgreSQLは、[表8-21](#)に示すように、IPv4アドレス、IPv6アドレス、MACアドレスを格納するデータ型を提供します。 ネットワークアドレスを格納するには普通のテキストデータ型の代わりにこれらの型を使うことの方が優れています。 なぜなら、これらのデータ型は入力値のエラー検査と専用の演算子と関数を提供しているからです ([項9.12](#)を参照してください)。

表 8-21. ネットワークアドレスデータ型

名前	格納サイズ	説明
<code>cidr</code>	7もしくは19バイト	IPv4、およびIPv6ネットワーク
<code>inet</code>	7もしくは19バイト	IPv4もしくはIPv6ホスト、およびネットワーク
<code>macaddr</code>	6バイト	MACアドレス

`inet`もしくは`cidr`をソートする時、IPv4アドレスは常にIPv6よりも前にソートされます。 `::10.2.3.4`や`::ffff:10.4.3.2`などIPv6アドレス内に埋め込まれた、もしくは関連付けされたIPv4アドレスも同様です。

8.9.1. inet

`inet`型はIPv4もしくはIPv6ホストアドレスとオプションでそのサブネットを1つのフィールドに保持します。 サブネットはホストアドレスのうち何ビットがネットワークアドレス ("ネットマスク") を表すかを指定することで表現されます。 もしネットマスクが32の場合、IPv4では単一ホストを意味し、サブネットを示しません。 IPv6ではアドレス長は128ビットです。 そのため128ビットが一意なホストアドレスを指定します。 ネットワークのみを使用したい場合は`inet`ではなく`cidr`型を利用してください。

このデータ型に対する入力書式は、IPv4、IPv6両方とも`address/y`です。 `y`がネットマスクのビット数です。 もし`/y`部分がない場合、ネットマスクはIPv4では32、IPv6では128となり、つまり、その値は単一ホストを表現します。 ネットマスクが単一ホストを表す場合、その表示時、`/y`の部分は抑制されます。

8.9.2. cidr

`cidr`データ型はIPv4、IPv6ネットワーク仕様を保持します。 入出力書式はクラス分けのないインターネットドメインルーティング協定に従います。 ネットワークを指定する時の書式は`address/y`で、`address`がIPv4もしくはIPv6アドレスで表したネットワークです。 `y`はネットマスクのビット数です。 もし`y`が省略された場合には、入力時に書き込まれたオクテットすべてが含まれるように大きさが確保されること以外は、従来のクラス付きアドレス番号指定システムに従って計算されます。 指定したネットマスクの右側にビットをセットしたネットワークアドレスを指定するとエラーになります。

[表8-22](#)に例をいくつか示します。

表 8-22. cidrデータ型入力例

cidr入力	cidr出力	abbrev(cidr)
192.168.100.128/25	192.168.100.128/25	192.168.100.128/25
192.168/24	192.168.0.0/24	192.168.0/24
192.168/25	192.168.0.0/25	192.168.0.0/25
192.168.1	192.168.1.0/24	192.168.1/24
192.168	192.168.0.0/24	192.168.0/24
128.1	128.1.0.0/16	128.1/16
128	128.0.0.0/16	128.0/16
128.1.2	128.1.2.0/24	128.1.2/24
10.1.2	10.1.2.0/24	10.1.2/24
10.1	10.1.0.0/16	10.1/16
10	10.0.0.0/8	10/8
10.1.2.3/32	10.1.2.3/32	10.1.2.3/32

2001:4f8:3:ba::/64	2001:4f8:3:ba::/64	2001:4f8:3:ba::/64
2001:4f8:3:ba:2e0:81ff:fe22:d1f1/128	2001:4f8:3:ba:2e0:81ff:fe22:d1f1/128	2001:4f8:3:ba:2e0:81ff:fe22:d1f1
::ffff:1.2.3.0/120	::ffff:1.2.3.0/120	::ffff:1.2.3/120
::ffff:1.2.3.0/128	::ffff:1.2.3.0/128	::ffff:1.2.3.0/128

8.9.3. inet と cidr データ型の違い

inet データ型と cidr データ型との基本的な相違は、inet ではネットマスクの右側に0でないビット値を受け付けますが、cidr では受け付けないことです。

ティップ: もしinetもしくはcidrの値の出力書式が気に入らないのであれば、関数host、textおよびabbrevを試してください。

8.9.4. macaddr

macaddr データ型は例えばイーサネットカードのハードウェアアドレスとして知られるMACアドレスを保持します（MACアドレスは他の目的でも使われますが）。 入力には以下の形式を受け入れます。

```
'08:00:2b:01:02:03'
'08-00-2b-01-02-03'
'08002b:010203'
'08002b-010203'
'0800.2b01.0203'
'08002b010203'
```

これらの例はすべて同一のアドレスを指定します。 aからfまでの桁は大文字小文字どちらでも構いません。 出力は常に最初に示された形式となります。

IEEE標準802-2001では、2番目の書式（ハイフンを使用）をMACアドレスの正規の表現と規定しています。 また、最初の書式（コロンを使用）をビット反転記法、つまり08-00-2b-01-02-03は01:00:4D:08:04:0Cであると規定しています。 この規約は現在ではほぼ無視され、古びたネットワーク（トークンリングなど）のみに関連するものです。 PostgreSQLではビット反転に関する準備をしていません。 また、すべての受付け可能な書式では正規のLSB順を使用します。

残る4つの入力書式はどの標準にも属しません。

[前のページ](#)
幾何データ型

[ホーム](#)
[上に戻る](#)

[次のページ](#)
ビット列データ型

8.10. ビット列データ型

ビット列とは1と0のビットが連続したものです。ビットマスクを格納したり顕在化するために使用されます。SQLのビット型には2つあります。 `bit(n)` と `bit varying(n)` です。ここで n は正の整数です。

`bit` 型のデータは厳密に n で表される長さに一致しなければなりません。この長さより長い短いビット列を格納しようとするとうエラーになります。 `bit varying` 型のデータは最大 n までの可変長です。最大長を越えるビット列は受け付けません。長さ指定のない `bit` データ型は `bit(1)` データ型と同一で、長さ指定のない `bit varying` データ型は無限長を意味します。

注意: ビット列の値を明示的に `bit(n)` にキャストすると、厳密に n ビットになるように、切り捨てられるか右側をゼロ詰めされ、エラーにはなりません。同様に、ビット列の値を明示的に `bit varying(n)` にキャストすると、ビット数が n を超える場合は右側が切り捨てられます。

ビット列定数に関する構文についての情報は[項4.1.2.5](#)を参照してください。ビット論理演算子とビット列操作関数が用意されています。[項9.6](#)を参照してください。

例 8-3. ビット列データ型を使って

```
CREATE TABLE test (a BIT(3), b BIT VARYING(5));
INSERT INTO test VALUES (B'101', B'00');
INSERT INTO test VALUES (B'10', B'101');
ERROR:  bit string length 2 does not match type bit(3)
INSERT INTO test VALUES (B'10'::bit(3), B'101');
SELECT * FROM test;
 a | b
----+----
101 | 00
100 | 101
```

ビット列の値は8ビット毎に1バイト、さらにビット列長に応じた5または8バイトのオーバーヘッドが必要です。（しかし、文字列に関する[項8.3](#)の説明と同様、長い値は圧縮または行外に移動する可能性があります。）

8.11. テキスト検索に関する型

PostgreSQLは、自然言語の文書の集合を通して検索を行い問い合わせに最も合致する文書を見つける機能である全文検索をサポートするために設計された2つのデータ型を提供します。 `tsvector` 型はテキスト検索に最適化された形式で文書を表現します。 `tsquery` 型は同様に問い合わせを表現します。 [第12章](#)ではこの機能を詳しく説明します。 また、[項9.13](#)では、関連する関数や演算子を要約します。

8.11.1. tsvector

`tsvector`の値は重複がない字句単位のソート済みリストです。 また、これらの単語は同じ単語の変種を吸収するために正規化が行われます（詳細は[第12章](#)を参照）。 以下の例に示すようにソートと重複除去は入力の際に自動的になされます。

```
SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector;
          tsvector
-----
'a' 'and' 'ate' 'cat' 'fat' 'mat' 'on' 'rat' 'sat'
```

空白文字または句読点を含む字句単位を表現するには、引用符でくくってください。

```
SELECT $$the lexeme '      ' contains spaces$$::tsvector;
          tsvector
-----
'      ' 'contains' 'lexeme' 'spaces' 'the'
```

（この例と次の例では、リテラル内の二重引用符記号が含まれることによる混乱を防ぐためにドル引用符付け文字列を使用します。） 引用符およびバックスラッシュが埋め込まれている場合は、以下のように二重にしなければなりません。

```
SELECT $$the lexeme 'Joe's' contains a quote$$::tsvector;
          tsvector
-----
'Joe's' 'a' 'contains' 'lexeme' 'quote' 'the'
```

オプションとして、字句要素に整数の位置を付けることもできます。

```
SELECT 'a:1 fat:2 cat:3 sat:4 on:5 a:6 mat:7 and:8 ate:9 a:10 fat:11 rat:12'::tsvector;
          tsvector
-----
'a':1,6,10 'and':8 'ate':9 'cat':3 'fat':2,11 'mat':7 'on':5 'rat':12 'sat':4
```

位置は通常、元の単語の文書中の位置を示します。 位置情報を近接順序に使用することができます。 位置の値は1から16383までで、これより大きな値は警告なく16383に設定されます。 同一字句要素に対する重複する位置項目は破棄されます。

位置を持つ字句単位はさらに重み付きのラベルを付与することができます。 ラベルはA、B、C、Dを取ることができます。 Dはデフォルトですので、以下の結果には現れません。

```
SELECT 'a:1A fat:2B,4C cat:5D'::tsvector;
          tsvector
-----
'a':1A 'cat':5 'fat':2B,4C
```

典型的に重みは、例えば、表題の単語には本文の単語と異なる印をつけるといった、文書構造を反映させるために使用されます。 テキスト検索の順序付け関数は異なる重み印に異なる優先度を割り当てることができます。

`tsvector`型自体は正規化を行わないことを理解することは重要です。 与えられる単語はアプリケーションで適切に正規化されていると仮定しています。 以下に例を示します。

```
select 'The Fat Rats'::tsvector;
          tsvector
-----
'Fat' 'Rats' 'The'
```

ほとんどの英文テキスト検索アプリケーションでは、上の単語は正規化されていないとみなされますが、`tsvector`は気にしません。 検索が適切に行われるように単語を正規化するために、生の文書テキストは通常`to_tsvector`経由で渡されます。

```
SELECT to_tsvector('english', 'The Fat Rats');
          to_tsvector
-----
'fat':2 'rat':3
```

繰り返しますが、詳細は[第12章](#)を参照してください。

8.11.2. tsquery

`tsquery`の値には検索される字句単位が格納され、それらは`&` (論理積)、`|` (論理和)、`!` (否定)論理演算子を遵守することで組み合わせられます。括弧を使用して演算子を強制的にグループ化することができます。

```
SELECT 'fat & rat'::tsquery;
      tsquery
-----
'fat' & 'rat'

SELECT 'fat & (rat | cat)'::tsquery;
      tsquery
-----
'fat' & ( 'rat' | 'cat' )

SELECT 'fat & rat & ! cat'::tsquery;
      tsquery
-----
'fat' & 'rat' & !'cat'
```

括弧がないと、`!` (否定)が最も強く、次に`&` (論理積)、そして`|` (論理和)という強さで結合されます。

省略することもできますが、`tsquery`内の字句単位に1つ以上の重み文字をもったラベルを付けることができます。これにより、こうした一致に関する重みの1つを持つ`tsvector`字句要素のみに一致するように制限させることができます。

```
SELECT 'fat:ab & cat'::tsquery;
      tsquery
-----
'fat':AB & 'cat'
```

同時に、`tsquery`内の字句単位は、前置一致を指定するため`*`でラベルを付けることができます。

```
SELECT 'super:*'::tsquery;
      tsquery
-----
'super':*
```

この問い合わせは、"super"で始まる`tsvector`内のいかなる単語とも一致します。接頭語はテキスト検索の設定により、最初に処理されることに注意してください。これはこの比較が真を返すことを意味します。

```
SELECT to_tsvector( 'postgraduate' ) @@ to_tsquery( 'postgres:*' );
?column?
-----
t
(1 row)
```

`postgres`が`postgr`という語幹になるためです。

```
SELECT to_tsquery('postgres:*');
      to_tsquery
-----
'postgr':*
(1 row)
```

このため`postgraduate`に一致します。

字句単位の引用符規則は前に説明した`tsvector`における字句単位と同じです。また、`tsvector`同様、必要な単語の正規化は`tsquery`型に変換する前に行う必要があります。こうした正規化の実行には`to_tsquery`関数が簡便です。

```
SELECT to_tsquery('Fat:ab & Cats');
      to_tsquery
-----
'fat':AB & 'cat'
```

[前のページ](#)

ビット列データ型

[ホーム](#)

[上に戻る](#)

[次のページ](#)

UUID型

8.12. UUID型

uuidデータ型は、RFC 4122、ISO/IEC 9834-8:2005および関連する標準に従う、汎用一意識別子（UUID）を格納します。（一部のシステムでは、このデータ型をグローバル一意識別子（GUID）と呼んでいます。）この識別子は、同一のアルゴリズムを使用しても既知の世界上の他の誰かが同一識別子が生成される可能性がほとんどないように選択されたアルゴリズムで生成された128ビット量の値です。したがって、分散システムにおいて、これら識別子は、単一データベース内でしか一意にならないシーケンスジェネレータよりも優れた一意性保証を提供します。

UUIDは、小文字の16進数表記桁の並びをいくつかのグループでハイフンで区切って表現されます。具体的には、8桁のグループが1つ、4桁のグループが3つ、次いで、12桁のグループが1つとなり、合計32桁で128ビットを表します。この標準形式のUUIDの例を以下に示します。

```
a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11
```

また、PostgreSQLは入力として、大文字表記の桁、中括弧でくくった標準形式、いくつかまたはすべてのハイフンを省略したもの、4桁のすべてのグループにハイフンを付加した別形式も受け付けます。以下に例を示します。

```
A0EEBC99-9C0B-4EF8-BB6D-6BB9BD380A11  
{a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11}  
a0eebc999c0b4ef8bb6d6bb9bd380a11  
a0ee-bc99-9c0b-4ef8-bb6d-6bb9-bd38-0a11  
{a0eebc99-9c0b4ef8-bb6d6bb9-bd380a11}
```

出力は常に標準形式になります。

PostgreSQLは、UUIDについて格納関数と比較関数を提供します。しかし、すべてのアプリケーションで十分適切な単一アルゴリズムが存在しませんので、データベースの中核にはUUIDを生成する関数はまったく含んでいません。[uuid-oss](#) モジュールは、複数の標準的なアルゴリズムを実装した関数を提供します。この他にも、UUIDをクライアントアプリケーションで生成することも、サーバサイドで他のライブラリを呼び出して生成することもできます。

8.13. XML型

`xml` データ型を使用して、XML データを格納することができます。 `text` 型のフィールドに XML データを格納する方法より、入力された値が整形式かどうかを検査する利点があります。 また、型を安全に操作するサポート関数があります。 [項9.14](#)を参照してください。 このデータ型を使用するためには、インストールが `configure --with-libxml` で構築されていることが必要です。

`xml` 型は、XML 標準で定義された整形式の"文書"およびXML 標準によるXMLDecl? content 生成規則で定義された"コンテンツ"フラグメントを格納することができます。 大雑把に言うと、これは、コンテンツフラグメントが1つ以上の最上位要素や文字ノードを持つことができることを意味します。 `xmlvalue IS DOCUMENT` という式を使用して、特定の`xml`が完全な文書か単なるコンテンツフラグメントか評価することができます。

8.13.1. XML値の作成

文字データから`xml`型の値を生成するためには、`xmlparse`関数を使用してください。

```
XMLPARSE ( { DOCUMENT | CONTENT } value )
```

例：

```
XMLPARSE (DOCUMENT '<?xml version="1.0"?><book><title>Manual</title><chapter>...</chapter></book>')
XMLPARSE (CONTENT 'abc<foo>bar</foo><bar>foo</bar>')
```

標準SQLに従って文字列をXML値に変換するためにはこの方法しかありませんが、次のようなPostgreSQL固有の構文も使用することができます。

```
xml '<foo>bar</foo>'
'<foo>bar</foo> '::xml
```

`xml` 型では文書型定義 (DTD) に対して入力値を検証することは、入力値がDTDを指定していたとしても、行いません。 また同様に、現時点ではXML Schemaなどの他のXMLスキーマ言語に対する検証サポートも組み込まれていません。

`xml` から文字列値を生成するという逆の操作では`xmlserialize`関数を使用してください。

```
XMLSERIALIZE ( { DOCUMENT | CONTENT } value AS type )
```

ここで、`type`は、`character`、`character varying`、`text` (またはこれらの別名) を取ることができます。 繰り返しになります。 が、標準SQLに従って`xml`と文字列型間の変換を行うためにはこの方法しかありません。 PostgreSQLでは単に値をキャストすることが可能です。

`XMLPARSE` や `XMLSERIALIZE` を使わずに文字列値と`xml`との間をキャストした場合、`DOCUMENT` か `CONTENT` かという選択が"XMLオプション"セッション設定パラメータによって決定されます。 このパラメータは標準コマンド

```
SET XML OPTION { DOCUMENT | CONTENT };
```

または、よりPostgreSQLらしい構文

```
SET xmloption TO { DOCUMENT | CONTENT };
```

を使用して設定することができます。 デフォルトは`CONTENT`ですので、すべての書式のXMLデータを扱うことができます。

注意: デフォルトのXMLオプション設定では、文書型宣言を含む文字列を`xml`に直接キャストすることはできません。 XMLコンテンツフラグメントの定義が受け付けられないためです。 この操作を行う必要がある場合、`XMLPARSE`を使用するかXMLオプションを変更してください。

8.13.2. 符号化方式の取扱い

クライアント側、サーバ側、および、これらを経由してやり取りされるXMLデータ内部で複数の文字符号化方式を扱う場合には注意が必要です。 サーバへの問い合わせをテキストモードで渡す場合とクライアントへの問い合わせ結果をテキストモードで渡す場合 (どちらも通常モードです)、PostgreSQLは、クライアントからサーバ、サーバからクライアントでやり取りされるすべての文字データを受信側の文字符号化方式に変換します。 [項22.3](#)を参照してください。 これには上の例のようなXML値の文字列表現も含まれます。 これは通常、埋め込まれた`encoding`宣言は変更されずに、クライアント/サーバ間でやり取りされる間に文字データが他方の符号化方式に変換されてしまうので、XMLデータ内の`encoding`が無効になる可能性があることを意味します。 この動作に対処するため、`xml`型の入力として表現された文字列に含まれている`encoding`宣言は無視され、その内容は常にサーバの現在の符号化方式になっているものと仮定されます。 したがって、正しく処理するためには、XMLデータにおける文

字列をクライアントの現在の符号化方式で送信しなければなりません。サーバに送信する前に文書を現在のクライアントの符号化方式に変換するか、クライアントの符号化方式を適切に調節するかは、クライアントの責任です。出力では`xml`型の値は`encoding`宣言を持ちません。クライアントはすべてのデータが現在のクライアントの符号化方式であることを前提としなければなりません。

バイナリモードを使用して、問い合わせパラメータをサーバに渡す場合、および、問い合わせ結果をクライアントに返す場合、文字セットの変換は行われません。このため状況は異なります。この場合、XMLデータ内の`encoding`宣言が認識され、もし存在しなければ、データがUTF-8であると仮定されます。（XML標準の要求通りです。PostgreSQLはUTF-16をサポートしていないことに注意してください。）出力では、データはクライアントの符号化方式を指定した`encoding`宣言を持ちます。ただし、もしクライアントの符号化方式がUTF-8の場合は`encoding`宣言は省略されます。

言うまでもありませんが、PostgreSQLを使用したXML処理では、XMLデータの符号化方式、クライアントの符号化方式、サーバの符号化方式が同じ場合にエラーが起こりづらく、より効率的です。XMLデータは内部的にUTF-8として処理されますので、サーバの符号化方式が同一のUTF-8である場合、最も効率が上がります。

注意
サーバ符号化方式がUTF-8でない場合、いくつかのXMLに関係した関数は非ASCIIデータに対して全く機能しないことがあります。特に <code>xpath()</code> に対する問題として知られています。

8.13.3. XML値へのアクセス

`xml`データ型は通常と異なり、比較演算子をまったく提供しません。これは、XMLデータに対し、よく定義され、誰にとっても有用な比較アルゴリズムが存在しないためです。この結果、`xml`列を検索値と比べて行を取り出すことはできません。したがって通常XML値には、IDなどの別のキーフィールドを一般的に付属させなければなりません。XML値の比較を行うもうひとつの方法は、文字列に一度変換することです。しかし、文字列比較は有用なXML比較方法といえないことに注意してください。

`xml`データ型用の比較演算子がありませんので、この型の列に直接インデックスを作成することはできません。XMLデータを高速に検索することが望まれるなら、その表現を文字列型にキャストし、それをインデックス付けするか、または、XPath式をインデックス付けするかという対策をとることができます。当然ながら、インデックス付けされた式で検索されるよう実際の問い合わせを調整する必要があります。

PostgreSQLのテキスト検索機能を使用して、XMLデータの全文検索速度をあげることもできます。しかし、PostgreSQL配布物では必要な前処理を未だサポートしていません。

8.14. JSON データ型

`json` 型には [RFC 4627](#) で定義されている JSON (JavaScript Object Notation) データを格納することができます。 `text` 型のデータも格納することができますが、 `json` 型は格納された各々の JSON の値が有効であるかをチェックできる点で優れています。関連する関数もあります。以下のリンクを参照してください。 [項9.15](#)

PostgreSQL はデータベース毎にサーバ符号化方式を 1 つのみ許します。 そのため、サーバ符号化方式が UTF-8 でない限り、厳密に仕様を満たす JSON に対応することができません。 サーバ符号化方式で表現できない文字を直接含めようとするとう失敗します。 反対に、サーバ符号化方式では表現できるが UTF-8 では表現できない文字は許されます。 サーバ符号化方式と関係なく `\uXXXX` エスケープは許され、構文の正当性のみが検査されます。

8.15. 配列

PostgreSQLではテーブルの列を可変長多次元配列として定義できます。あらゆる組み込み型、ユーザ定義基本型、列挙型または複合型の配列も作成可能です。ドメインの配列はサポートされていません。

8.15.1. 配列型の宣言

実際に配列の使い方を説明するために、次のテーブルを作成します。

```
CREATE TABLE sal_emp (
    name          text,
    pay_by_quarter integer[],
    schedule       text[][]
);
```

見ておわかりのように配列データ型は配列要素のデータ型の名前に大括弧 ([]) を付けて指定します。このコマンドは `text` 型文字列 (`name`)、従業員の四半期の給与を保存する `integer` 型の一次元配列 (`pay_by_quarter`)、そして従業員の週間スケジュールを保存する `text` 型の二次元配列 (`schedule`) の列を持つ `sal_emp` という名前のテーブルを作成します。

`CREATE TABLE` 構文で指定する配列の正確な大きさを指定することができます。

```
CREATE TABLE tictactoe (
    squares integer[3][3]
);
```

とは言っても現在の実装では指定された配列の大きさの制限を無視します。つまり、長さの指定がない配列と同じ振舞いをします。

現在の実装では次元数の宣言も強制していません。特定の要素型の配列はすべて大きさあるいは次元数とは無関係に同じ型とみなされます。ですから `CREATE TABLE` で配列の大きさや次元数を宣言することは、単なる説明です。実行時の動作に影響を及ぼしません。

SQLに準拠し、`ARRAY` キーワードを使用したもう1つの構文を一次元配列に使うことができます。 `pay_by_quarter` を次のように定義することもできます。

```
pay_by_quarter integer ARRAY[4],
```

または、もし配列の大きさが指定されない場合は次のようになります。

```
pay_by_quarter integer ARRAY,
```

しかし、前で触れたようにPostgreSQLはどんな場合でも大きさの制限を強要しません。

8.15.2. 配列の値の入力

リテラル定数として配列の値を書き込むには、その要素の値を中括弧で囲み、それぞれの要素の値をカンマで区切ります (C言語を知っているならば、構造体を初期化するための構文のようなものと考えてください)。要素の値を二重引用符でくくることができ、カンマもしくは中括弧がある時は必ずそのように書かなければなりません (詳細は以下に出てきます)。したがって配列定数の一般的書式は次のようになります。

```
'{ val1 delim val2 delim ... }'
```

ここで `delim` はその `pg_type` 項目に記録されている型の区切り文字です。PostgreSQL配布物で提供されている標準データ型の内、セミコロン (;) を使用する `box` 型を除き、すべてはカンマ (,) を使います。それぞれの `val` は配列要素の型の定数か副配列です。配列定数の例を以下に示します。

```
'{{1,2,3},{4,5,6},{7,8,9}}'
```

この定数は整数の3つの副配列を持っている二次元3×3の配列です。

配列定数の要素を `NULL` とするためには、その要素値に `NULL` と記載してください。 (`NULL` を大文字で書いても小文字で書いても構いません。) "`NULL`" という文字列値を指定したければ、二重引用符でくくって記載しなければなりません。

(この種の配列定数は実際 [項4.1.2.7](#) で説明されている一般型定数の特別の場合に過ぎません。この定数は元々文字列として扱われていて配列入力ルーチンに渡されます。明示的な型指定が必要かもしれません。)

では、`INSERT` 文をいくつか紹介します。

```
INSERT INTO sal_emp
VALUES ('Bill',
'{10000, 10000, 10000, 10000}',
'{"meeting", "lunch"}, {"training", "presentation"}');

INSERT INTO sal_emp
VALUES ('Carol',
'{20000, 25000, 25000, 25000}',
'{"breakfast", "consulting"}, {"meeting", "lunch"}');
```

上に記載した2つの挿入文の結果は次のようになります。

```
SELECT * FROM sal_emp;
name | pay_by_quarter | schedule
-----+-----+-----
Bill | {10000,10000,10000,10000} | {{meeting,lunch},{training,presentation}}
Carol | {20000,25000,25000,25000} | {{breakfast,consulting},{meeting,lunch}}
(2 rows)
```

多次元配列では、各次元の範囲を合わせなければなりません。一致しないと以下のようにエラーが発生します。

```
INSERT INTO sal_emp
VALUES ('Bill',
'{10000, 10000, 10000, 10000}',
'{"meeting", "lunch"}, {"meeting"}');
ERROR: multidimensional arrays must have array expressions with matching dimensions
```

ARRAY 生成子構文も使えます。

```
INSERT INTO sal_emp
VALUES ('Bill',
ARRAY[10000, 10000, 10000, 10000],
ARRAY[['meeting', 'lunch'], ['training', 'presentation']]);

INSERT INTO sal_emp
VALUES ('Carol',
ARRAY[20000, 25000, 25000, 25000],
ARRAY[['breakfast', 'consulting'], ['meeting', 'lunch']]);
```

配列要素は通常のSQL定数もしくは演算式であることに注意してください。例えば文字列リテラルは配列リテラルと同様、二重引用符ではなく単一引用符でくくられます。ARRAY 生成子構文は[項4.2.12](#)により詳しい説明があります。

8.15.3. 配列へのアクセス

ではテーブルに対していくつかの問い合わせを行ってみましょう。初めに、配列の単一要素にアクセスする方法を示します。この問い合わせは第2四半期に給与が更新された従業員の名前を抽出します。

```
SELECT name FROM sal_emp WHERE pay_by_quarter[1] <> pay_by_quarter[2];

name
-----
Carol
(1 row)
```

配列の添字番号は大括弧で囲んで記述されます。デフォルトでPostgreSQLは配列に対し「1始まり」の振り番規定を採用しています。つまり要素が n 個ある配列は`array[1]`で始まり、`array[n]`で終わります。

次の問い合わせは全ての従業員の第3四半期の給与を抽出します。

```
SELECT pay_by_quarter[3] FROM sal_emp;

pay_by_quarter
-----
10000
25000
(2 rows)
```

また、配列や副配列の任意の縦方向の部分を切り出すこともできます。一次元以上の配列についてその一部を表現するには、`lower-bound:upper-bound`と記述します。例えばこの問い合わせはBillのその週の初めの2日に最初何が予定されているかを抽出します。

```
SELECT schedule[1:2][1:1] FROM sal_emp WHERE name = 'Bill';

schedule
-----
{{meeting},{training}}
(1 row)
```

任意の次元を部分として、つまりコロンを含めて記述すると、すべての次元が部分として扱われます。単一の番号のみ（コロン

を持たない) を持つ次元はすべて、1から指定番号までと扱われます。例えば、[2]は以下の例のように [1:2] と扱われます。

```
SELECT schedule[1:2][2] FROM sal_emp WHERE name = 'Bill';

          schedule
-----
{{meeting,lunch},{training,presentation}}
(1 row)
```

切り出しのない場合と混乱を避けるため、すべての次元に対し切り出し構文を使用することが最善です。例えば、[2][1:1]ではなく、[1:2][1:1]のようにします。

配列自体がNULLもしくはその添字式がNULLとなる場合、配列添字式はNULLを返します。また、配列の範囲を超える添字の場合もNULLが返されます（この場合はエラーになりません）。例えば、scheduleが現在[1:3][1:2]次元であれば、schedule[3][3]の参照はNULLとなります。同様にして、添字として間違った値を指定して配列を参照した場合もエラーではなく、NULLが返されます。

同様に、部分配列式も配列自体がNULLもしくはその添字式がNULLとなる場合にNULLを返します。しかし、現在の配列範囲を完全に超えた部分配列を選択する場合では、部分配列式はNULLではなく空の（0次元）の配列を返します。（これは切り出しなしの動作に一致せず、歴史的理由で行われるものです。）要求された部分配列が配列の範囲に重なる場合、NULLを返さずに、警告なく重複部分だけに減少させます。

array_dims関数で任意の配列値の現在の次元を取り出せます。

```
SELECT array_dims(schedule) FROM sal_emp WHERE name = 'Carol';

          array_dims
-----
[1:2][1:2]
(1 row)
```

array_dims関数はtext型で結果を返します。人間が結果を見るためには便利ですが、プログラムにとって都合がよくありません。次元はarray_upperとarray_lowerでも抽出することができ、それぞれ特定の配列の次元の上限と下限を返します。

```
SELECT array_upper(schedule, 1) FROM sal_emp WHERE name = 'Carol';

          array_upper
-----
                2
(1 row)
```

array_lengthは指定された配列次元の長さを返します。

```
SELECT array_length(schedule, 1) FROM sal_emp WHERE name = 'Carol';

          array_length
-----
                2
(1 row)
```

8.15.4. 配列の変更

配列の値を全て置き換えることができます。

```
UPDATE sal_emp SET pay_by_quarter = '{25000,25000,27000,27000}'
WHERE name = 'Carol';
```

もしくはARRAY演算構文を用いて次のように書きます。

```
UPDATE sal_emp SET pay_by_quarter = ARRAY[25000,25000,27000,27000]
WHERE name = 'Carol';
```

配列の1つの要素を更新することも可能です。

```
UPDATE sal_emp SET pay_by_quarter[4] = 15000
WHERE name = 'Bill';
```

あるいは一部分の更新も可能です。

```
UPDATE sal_emp SET pay_by_quarter[1:2] = '{27000,27000}'
WHERE name = 'Carol';
```

保存されている配列の値は、存在しない要素に代入することで拡張することができます。過去に存在した位置と新しく代入された位置との間はNULLで埋められます。例えば、現在配列myarrayの要素数が4の場合、myarray[6]を割り当てる更新の後6要素を持つこととなり、myarray[5]はNULLを含みます。現在、こうした方法での拡張は、1次元配列でのみ許されます。多次元配列では行うことができません。

添字指定の代入で1始まり以外の添字がある配列を作れます。例えば添字が-2から7までの値を持つ配列をarray[-2:7]で指定できます。

新規の配列の値は連結演算子||を用いて作成することもできます。

```
SELECT ARRAY[1,2] || ARRAY[3,4];
?column?
-----
{1,2,3,4}
(1 row)

SELECT ARRAY[5,6] || ARRAY[[1,2],[3,4]];
?column?
-----
{{5,6},{1,2},{3,4}}
(1 row)
```

連結演算子を使うと、一次元配列の最初もしくは最後に1つの要素を押し込むことができます。さらには2つのN-次元配列もしくはN-次元配列とN+1-次元配列にも対応しています。

1つの要素が1次元配列の先頭や末尾に押し込まれた時、結果は配列演算項目と同じ下限添字を持つ配列となります。以下に例を示します。

```
SELECT array_dims(1 || '[0:1]={2,3}':int[]);
array_dims
-----
[0:2]
(1 row)

SELECT array_dims(ARRAY[1,2] || 3);
array_dims
-----
[1:3]
(1 row)
```

等しい次元を持った2つの配列が連結された場合、結果は左側演算項目の外側の次元の下限添字を引き継ぎます。結果は右側被演算子のすべての要素に左側被演算子が続いた配列となります。例を挙げます。

```
SELECT array_dims(ARRAY[1,2] || ARRAY[3,4,5]);
array_dims
-----
[1:5]
(1 row)

SELECT array_dims(ARRAY[[1,2],[3,4]] || ARRAY[[5,6],[7,8],[9,0]]);
array_dims
-----
[1:5][1:2]
(1 row)
```

N-次元配列がN+1-次元配列の最初または最後に押し込まれると、結果は上記と似通った要素配列になります。それぞれのN-次元副配列は本質的にN+1-次元配列の外側の次元の要素となります。例を挙げます。

```
SELECT array_dims(ARRAY[1,2] || ARRAY[[3,4],[5,6]]);
array_dims
-----
[1:3][1:2]
(1 row)
```

配列はarray_prepend、array_append、もしくはarray_catを使って構築することもできます。初めの2つは一次元配列にしか対応していませんが、array_catは多次元配列でも使えます。上で説明した連結演算子はそれぞれの関数を直接実行することよりも望ましいことを覚えておいてください。事実、それらの関数は連結演算子の実装のため主として存在します。とは言っても、ユーザ定義の集約関数を作る時そのまま使えます。例を挙げます。

```
SELECT array_prepend(1, ARRAY[2,3]);
array_prepend
-----
{1,2,3}
(1 row)

SELECT array_append(ARRAY[1,2], 3);
array_append
-----
{1,2,3}
(1 row)

SELECT array_cat(ARRAY[1,2], ARRAY[3,4]);
array_cat
-----
{1,2,3,4}
(1 row)
```



```
SELECT array_cat (ARRAY[[1,2],[3,4]], ARRAY[5,6]);
      array_cat
-----
 {{1,2},{3,4},{5,6}}
(1 row)

SELECT array_cat (ARRAY[5,6], ARRAY[[1,2],[3,4]]);
      array_cat
-----
 {{5,6},{1,2},{3,4}}
```

8.15.5. 配列内の検索

配列内のある値を検索するにはそれぞれの値が検証されなければなりません。もし配列の大きさがわかっているならば手作業でも検索できます。例を挙げます。

```
SELECT * FROM sal_emp WHERE pay_by_quarter[1] = 10000 OR
      pay_by_quarter[2] = 10000 OR
      pay_by_quarter[3] = 10000 OR
      pay_by_quarter[4] = 10000;
```

とは言ってもこの方法では大きい配列では大変な作業となりますし、配列の大きさが不明な場合この方法は使えません。代わりになる方法が[項9.23](#)で説明されています。上の問い合わせは以下のように書くことができます。

```
SELECT * FROM sal_emp WHERE 10000 = ANY (pay_by_quarter);
```

さらに配列で行の値が全て10000に等しいものを見つけることもできます。

```
SELECT * FROM sal_emp WHERE 10000 = ALL (pay_by_quarter);
```

代わりとして、`generate_subscripts`関数を使うことができます。以下はその例です。

```
SELECT * FROM
  (SELECT pay_by_quarter,
    generate_subscripts(pay_by_quarter, 1) AS s
   FROM sal_emp) AS foo
 WHERE pay_by_quarter[s] = 10000;
```

この関数は[表9-49](#)に記載されています。

ティップ: 配列は集合ではありません。特定の配列要素に検索をかけることはデータベース設計が誤っている可能性があります。配列の要素とみなされるそれぞれの項目を行に持つ別のテーブルを使うことを検討してください。この方が検索がより簡単になり要素数が大きくなっても規模的拡張性があります。

8.15.6. 配列の入出力構文

配列の値の外部表現は配列の要素の型に対するI/O変換ルールに基づいて解釈された項目と配列の構造を示す装飾項目で構成されています。装飾は配列の値を中括弧（{と}）で囲んだものと次の項目との間を区切り文字で区切ったものです。区切り文字は通常カンマ（,）ですが他の文字でも構いません。配列の要素の型`typdelim`を設定することで決まります。PostgreSQL配布物における標準のデータ型の中でセミコロン（;）を使う`box`型を除いて、すべてはカンマを使います。多次元配列ではそれぞれの次元（行、面、立体など）はそれ自身の階層において中括弧、同じ階層の中括弧でくくられた次の塊との間に区切り文字が書かれていなければなりません。

空の文字列や中括弧や区切り文字、二重引用符、バックスラッシュ、空白、`NULL`という単語が含まれていると、配列出力処理は要素の値を二重引用符でくくります。要素の値に組み込まれている二重引用符とバックスラッシュはバックスラッシュでエスケープされます。数値データ型に対しては二重引用符が出現しないと想定するのが安全ですが、テキストデータ型の場合引用符がある場合とない場合に対処できるようにしておくべきです。

デフォルトでは配列の次元の下限インデックス値は1に設定されています。他の下限値を持つ配列を表現したければ、配列定数を作成する前に明示的に配列添字範囲を指定することで実現できます。修飾項目はそれぞれの配列次元の上限と下限をコロンの(:)で区切って前後を大括弧([])でくくった形式になっています。代入演算子(=)の後に配列次元修飾項目が続きます。例を示します。

```
SELECT f1[1][ -2 ][3] AS e1, f1[1][ -1 ][5] AS e2
   FROM (SELECT '[1:1][ -2: -1 ][3:5]={{{1,2,3},{4,5,6}}}'::int[] AS f1) AS ss;

 e1 | e2
-----+-----
  1 |  6
(1 row)
```

1とは異なる下限を持つ場合にのみ、配列出力関数はその結果に明示的な次元を含めます。

要素に指定された値が`NULL`（またはその亜種）の場合、要素は`NULL`として扱われます。引用符やバックスラッシュがあると、これは無効となり、`"NULL"`という文字列リテラルを入力することができます。また、8.2以前のPostgreSQLとの後方互換性のため、`array_nulls`設定パラメータを`off`にして、`NULL`を`NULL`として認識しないようにすることができます。

前に示したように配列に値を書き込む場合は独立した配列要素を二重引用符でくくります。配列値パーサが配列要素値によって混乱を来さないように必ずこの形式を守ってください。例えば、中括弧、カンマ（もしくはデータ型の区切り文字）、二重引用符、バックスラッシュもしくは前後に付いた空白を含む要素は必ず二重引用符でくくられなければなりません。空文字列や`NULL`という単語自体も同様に引用符でくくられなければなりません。二重引用符もしくはバックスラッシュを引用符付きの配列要素に付け加えたい場合、エスケープ文字列構文を使用し、そしてその直前にバックスラッシュを付けます。別の方法として配列構文とみなされるような全てのデータ文字を、引用符を使用しないでバックスラッシュでエスケープしても構いません。

括弧の右側もしくは左側それぞれの前と後に空白を追加することができます。同様に独立した項目の文字列の前後に空白を付け加えることもできます。これらすべての場合において空白は無視されます。とは言っても二重引用符で囲まれた要素の中の空白、もしくは要素の空白文字以外により両側がくくられているものは無視されません。

注意: SQLコマンドで書かれたものは最初に文字列リテラルとして解釈され、その次に配列として解釈されることを覚えておいてください。と言うことは、バックスラッシュの数が倍になることを意味します。例えばバックスラッシュと二重引用符を含んだ`text`配列値を挿入する場合、次のようになります。

```
INSERT ... VALUES (E'{"¥¥¥¥¥¥","¥¥"}');
```

エスケープ文字列プロセッサは1つの階層のバックスラッシュを取り除きますので、配列値パーサに渡された時は`{"¥¥¥¥¥¥","¥¥"}`のようになります。さらに`text`データ型入力ルーチンに与えられた文字列はそれぞれ`¥`と`"`になります（もし入力ルーチンがバックスラッシュを特別に取り扱うデータ型を操作している場合（例えば`bytea`）、1つのバックスラッシュを配列要素に保存したい時はコマンドの中に8つのバックスラッシュが必要です）。ドル引用符付け（[項4.1.2.4](#)参照）を使用して、バックスラッシュを二重化する必要性をなくすることができます。

ティップ: SQLコマンドの中で配列値を書く時、配列リテラル構文よりも`ARRAY`生成子構文（[項4.2.12](#)を参照）の方が往々にして扱いやすい場合があります。

[前のページ](#)
JSONデータ型

[ホーム](#)
[上に戻る](#)

[次のページ](#)
複合型

8.16. 複合型

複合型は、行もしくはレコードの構造を表現します。本質的には、これは単なるフィールド名とそのデータ型のリストです。PostgreSQLでは、単純な型において使用される方法と多くは同じ方法で複合型を使用できます。例えば、テーブルの列は複合型の型のものとして宣言することができます。

8.16.1. 複合型の宣言

複合型の宣言の例を以下に2つ示します。

```
CREATE TYPE complex AS (  
    r          double precision,  
    i          double precision  
);  
  
CREATE TYPE inventory_item AS (  
    name        text,  
    supplier_id integer,  
    price       numeric  
);
```

この構文は、フィールド名とその型のみを指定できるという点を除き、`CREATE TABLE`と同等です。現在は、制約（`NOT NULL`など）を含めることはできません。`AS`キーワードが重要であることに注意してください。これがないと、システムは`CREATE TYPE`の意味を異なって解釈し、おかしい構文エラーを引き起こします。

定義済みの型を使用して、以下のようにテーブルや関数を生成することができます。

```
CREATE TABLE on_hand (  
    item      inventory_item,  
    count     integer  
);  
  
INSERT INTO on_hand VALUES (ROW('fuzzy dice', 42, 1.99), 1000);
```

また、関数においては以下のように利用できます。

```
CREATE FUNCTION price_extension(inventory_item, integer) RETURNS numeric  
AS 'SELECT $1.price * $2' LANGUAGE SQL;  
  
SELECT price_extension(item, 10) FROM on_hand;
```

テーブルを生成する時には、テーブルの行型を表現するために、テーブル名と同じ名前の複合型も自動的に生成されます。例えば、以下のように

```
CREATE TABLE inventory_item (  
    name        text,  
    supplier_id integer REFERENCES suppliers,  
    price       numeric CHECK (price > 0)  
);
```

テーブルを作成すると、上述のものと同じ`inventory_item`という複合型が副次的に作成され、同様に使用することができるようになります。しかし、現在の実装には、次のような重要な制限があることに注意してください。複合型には制約が関連付けられませんので、テーブル定義に含まれる制約は、テーブルの外部に作成される複合型には適用されません（部分的な回避方法は、複合型のメンバとしてドメイン型を使用することです）。

8.16.2. 複合型の値の入力

複合型をリテラル定数として記述するには、フィールド値をカンマで区切り、それらを括弧で括ります。フィールド値を二重引用符で括ることができ、また、値にカンマや括弧を含む場合は二重引用符で括らなければなりません（より詳細については後で説明します）。したがって、複合型の一般的な書式は以下ようになります。

```
'( val1 , val2 , ... )'
```

以下に例を示します。

```
'("fuzzy dice",42,1.99)'
```

これは、上述の`inventory_item`型の値として有効なものです。フィールドをNULLにするには、リスト中の該当位置を空にします。例えば、以下の定数は3番目のフィールドにNULLを指定しています。

```
'("fuzzy dice",42,)'
```

NULLではなく空文字列にしたいのであれば、以下のように引用符を二重に記述します。

```
'("",42,)'
```

これにより、最初のフィールドは非NULLの空文字列に、3番目のフィールドはNULLになります。

(実際には、こうした定数は[項4.1.2.7](#)で説明した、一般的な型の定数の特殊な場合に過ぎません。定数はまず、文字列として扱われ、複合型の入力変換処理に渡されます。明示的な型指定が必要になることもあります。)

また、ROW式構文も、複合値を生成する際に使用することができます。複数の階層に渡る引用符について考慮する必要がないため、おそらくほとんどの場合、これは文字列リテラル構文よりも簡単に使用できます。上記において、既にこの方法を使用しています。

```
ROW('fuzzy dice', 42, 1.99)
ROW('', 42, NULL)
```

式の中に2つ以上のフィールドがある場合には、ROWキーワードは実際には省略することができます。ですので、以下のように簡略化することができます。

```
('fuzzy dice', 42, 1.99)
('', 42, NULL)
```

ROW構文については[項4.2.13](#)でより詳細に説明します。

8.16.3. 複合型へのアクセス

複合型の列のフィールドにアクセスするには、テーブル名からフィールドを選択する場合とほぼ同様に、ドットとフィールド名を記述します。実際、テーブル名からの選択とかなり似ていますので、パーサを混乱させないように括弧を使用しなければならないことがしばしばあります。例えば、`on_hand`というテーブルの例からサブフィールドを選択しようとした場合、以下のように書くかもしれません。

```
SELECT item.name FROM on_hand WHERE item.price > 9.99;
```

これは、SQLの構文規則に従って`item`が`on_hand`の列名ではなくテーブル名として解釈されるため、動作しません。以下のように記述しなければなりません。

```
SELECT (item).name FROM on_hand WHERE (item).price > 9.99;
```

また、テーブル名も使用しなければならない場合（例えば複数テーブルに対する問い合わせ）、以下のようになります。

```
SELECT (on_hand.item).name FROM on_hand WHERE (on_hand.item).price > 9.99;
```

これで、括弧で括られたオブジェクトは正しく`item`列への参照として解釈され、サブフィールドはそこから選択できるようになります。

似たような構文上の問題は、複合型からフィールドを選択する時、常に発生します。例えば、複合型の値を返す関数の結果から1つだけフィールドを選択する場合、以下のように記述しなければなりません。

```
SELECT (my_func(...)).field FROM ...
```

追加の括弧がないと、これは構文エラーを生成します。

8.16.4. 複合型の変更

複合型の列への挿入と更新についての適切な構文の例をいくつか示します。まず、列全体を挿入、更新する例です。

```
INSERT INTO mytab (complex_col) VALUES((1.1,2.2));

UPDATE mytab SET complex_col = ROW(1.1,2.2) WHERE ...;
```

最初の例ではROWを省略し、2番目の例ではROWを使用しています。どちらの方法でも行うことができます。

以下のようにして、複合型の列の個々のサブフィールドを更新することができます。

```
UPDATE mytab SET complex_col.r = (complex_col).r + 1 WHERE ...;
```

ここで、SET直後の列名の周りに括弧を記述する必要がないこと（実際には記述できないこと）、しかし、等号の右で同じ列を参照する場合には括弧が必要なことに注意してください。

また、INSERTの対象としてサブフィールドを指定することもできます。

```
INSERT INTO mytab (complex_col.r, complex_col.i) VALUES(1.1, 2.2);
```

列のサブフィールド全ての値を与えていなければ、残りのサブフィールドはNULL値になります。

8.16.5. 複合型の入出力構文

複合型の外部テキスト表現は、個々のフィールド用のI/O変換規則に従って解釈される項目群と、複合構造を意味する修飾から構成されます。この修飾は、値全体を括る括弧（`()`）と隣接した項目間のカンマ（`,`）で構成されます。括弧の外側の空白文字は無視されますが、括弧の内部ではフィールド値の一部とみなされます。ただし、空白に意味があるかないかについては、そのフィールドのデータ型用の入力変換規則に従います。例えば、

```
' ( 42) '
```

括弧内の空白文字は、そのフィールド型が整数の場合は無視されますが、テキストの場合は無視されません。

前述の通り、複合型の値を記述する時には、個々のフィールド値を二重引用符で括ることができます。もし、フィールド値が複合型値用のパーサを混乱させる場合には、これは必須です。具体的には、括弧、カンマ、二重引用符、バックスラッシュを含むフィールドの場合、二重引用符で括る必要があります。引用符で括った複合型のフィールド値内に二重引用符やバックスラッシュが存在する場合、その前にバックスラッシュを付けてください（また、引用符で括った複合型のフィールド値内に二重の引用符の組み合わせがあると、これは二重引用符を表す文字として解釈されます。これは、SQLリテラル文字列内の単一引用符の規則と同じです）。そのままでは複合型に対する構文として解釈されてしまう、全てのデータ文字を保護する他の方法として、引用符付けをせずにバックスラッシュによるエスケープを使用することができます。

完全な空フィールド値（カンマや括弧の間にまったく文字がないもの）はNULLを表します。NULLではなく空文字列を値として記述するには `""` と記述してください。

複合型の出力処理では、もしフィールド値が空文字列の場合や括弧、カンマ、二重引用符、バックスラッシュ、空白文字を含む場合には、そのフィールド値を二重引用符で括って出力します（空白文字に対するこの処理は重要ではありませんが、可読性を高めます）。フィールド値内に埋め込まれた二重引用符やバックスラッシュは二重化されます。

注意: SQLコマンド内部に記述したものは、まず文字列リテラルとして、その後、複合型として解釈されることを覚えておってください。これは必要なバックスラッシュの数を倍にします（エスケープ文字列構文が使用されることを仮定しています）。例えば、複合型の値の中に二重引用符とバックスラッシュを持つ`text`フィールドに挿入するには、以下のように書かなければなりません。

```
INSERT ... VALUES (E'("¥¥"¥¥¥¥")');
```

文字列リテラルプロセッサが第1レベルのバックスラッシュを取り除くため、複合型値のパーサに渡されるものは `("¥¥¥¥")` のようになります。そして、`text` データ型の入力関数に渡される文字列は `¥¥` になります（もし、例えば `bytea` といった、その入力関数もバックスラッシュを特別に扱うデータ型を扱っている場合、1つのバックスラッシュを複合型のフィールドに格納するためにコマンド内に8個ものバックスラッシュが必要になります）。ドル引用符付け（[項4.1.2.4](#)を参照）を使用して、このバックスラッシュの二重化を防ぐことができます。

ティップ: SQLコマンド内に複合型の値を書く時、通常、`ROW`生成構文の方が複合型のリテラル構文より作業が簡単です。`ROW`による記述では、複合型のメンバ以外の記述方法と同じ方法で個々のフィールド値を記述することができます。

[前のページ](#)
配列

[ホーム](#)
[上に戻る](#)

[次のページ](#)
範囲型

8.17. 範囲型

範囲型は、ある要素型(その範囲の派生元型と呼ばれます)の値の範囲を表わすデータ型です。例えば、`timestamp`の範囲は、会議室が予約されている時間の範囲を表すのに使うことができます。この場合、データ型は`tsrange("timestamp range"の短縮)`で、`timestamp`が派生元型となります。派生元型には完全な順序がなければなりません。これは、要素の値が範囲の前、中間、後のどこにあるのか明確に定義されている必要があるからです。

範囲型は、一つの範囲内の多くの要素の値を表現できる、また、範囲の重なりなどの概念が明確に表現できる、などの理由で便利です。スケジューリングのために時刻と日付の範囲を使うのがもっとも簡単な例ですが、価格の範囲、機器による測定値の範囲などといったものにも利用できます。

8.17.1. 組み込みの範囲型

PostgreSQLには、以下の組み込みの範囲型があります。

- `int4range` — `integer`の範囲
- `int8range` — `bigint`の範囲
- `numrange` — `numeric`の範囲
- `tsrange` — `timestamp without time zone`の範囲
- `tstzrange` — `timestamp with time zone`の範囲
- `daterange` — `date`の範囲

この他にも、独自の範囲型を定義することができます。詳しくは[CREATE TYPE](#)を参照してください。

8.17.2. 例

```
CREATE TABLE reservation (room int, during tsrange);
INSERT INTO reservation VALUES
    (1108, '[2010-01-01 14:30, 2010-01-01 15:30)');

-- 含有
SELECT int4range(10, 20) @> 3;

-- 重なり
SELECT numrange(11.1, 22.2) && numrange(20.0, 30.0);

-- 上限の取得
SELECT upper(int8range(15, 25));

-- 共通部分の計算
SELECT int4range(10, 20) * int4range(15, 25);

-- 範囲は空か
SELECT isempty(numrange(1, 5));
```

範囲型についての演算子と関数の完全なリストについては、[表9-43](#)と[表9-44](#)を参照してください。

8.17.3. 閉じた境界と開いた境界

空でない範囲には必ず2つの境界、つまり下限値と上限値があります。これらの値の間にある値はすべてその範囲に含まれます。閉じた境界とは、その境界値自体が範囲に含まれることを意味し、開いた境界とは、その境界値が範囲に含まれないことを意味します。

範囲を文字列の形式で表すとき、閉じた下限値は"`[`"で、開いた下限値は"`(`"で表します。同様に、閉じた上限値は"`]`"で、開いた上限値は"`)`"で表します。(詳しくは [項8.17.5](#)を参照してください。)

関数`lower_inc`および`upper_inc`はそれぞれ、範囲の下限値と上限値が閉じているかどうかを検査します。

8.17.4. 無限の(境界のない)範囲

範囲の下限値は省略することができ、このとき、上限値より小さいすべての値はその範囲に含まれることになります。同じように、範囲の上限値も省略することができ、このときは、下限値より大きいすべての値がその範囲に含まれることになります。下限値と上限値が両方とも省略されたときは、その要素型のすべての値がその範囲に含まれるとみなされます。

これは、それぞれ、下限値が"マイナス無限大"である、あるいは、上限値が"プラス無限大"である、と見なすことと同等です。ただし、これらの無限大の値は範囲型の値にはならない、ということ、そして範囲の一部になることはできない、ということに注意してください。(つまり、閉じた無限大の境界などといったものではありません — そのように書いてもそれは自動的に開いた境界に変換されます。)

いくつかの要素型には"無限大"の概念がありますが、範囲型の機能に関する限り、これは一つの値として扱われます。例えばtimestampの範囲で[`today,`]の意味は[`today,`)と同じです。しかし、[`today,` infinity]の意味は[`today,` infinity)と少し違います — 後者はtimestampの特別な値infinityを含みません。

関数`lower_inf`および`upper_inf`はそれぞれ範囲の下限値と上限値が無限大かどうかを検査します。

8.17.5. 範囲の入出力

範囲値の入力は、以下の形式の一つに従わなければなりません。

```
( lower-bound , upper-bound )
( lower-bound , upper-bound ]
[ lower-bound , upper-bound )
[ lower-bound , upper-bound ]
empty
```

前にも述べたとおり、丸括弧と大括弧は下限値と上限値が開いているか閉じているかを表します。最後の形式が`empty`であることに注意してください。これは空の範囲(範囲に含まれる値が1つもない)を表します。

`lower-bound`は、その派生元型の有効な入力値となる文字列か、あるいは省略して下限値がないことを指定するか of のいずれかです。同様に、`upper-bound`は、その派生元型の有効な入力値となる文字列か、あるいは省略して上限値がないことを指定するか of のいずれかです。

境界値は"(二重引用符)で括弧することができます。これは特に境界値が丸括弧、大括弧、カンマ、二重引用符、あるいはバックスラッシュを含んでいる場合に必要となります。そうしなければ、これらの文字は範囲の構文の一部とみなされてしまうからです。二重引用符あるいはバックスラッシュを引用符で括られた境界値の中に入れるには、その直前にバックスラッシュを入れてください。(また、SQLの文字列リテラルと同じように、二重引用符で括られた境界値の中で二重引用符を2つ続けることで1つの二重引用符を表すこともできます。)あるいは、引用符で括る代わりに、範囲の構文の一部とみなされるすべての文字をバックスラッシュでエスケープする、ということもできます。なお、境界値として空文字列を指定するには""と書いてください。何も書かないと、境界値が無限大であることになってしまいます。

境界値の前後に空白文字を入れることができますが、括弧内にある空白文字はすべて下限値あるいは上限値の一部とみなされます。(このことは、要素型によっては重要かもしれませんが、重要でないかもしれません。)

注意: これらの規則は、複合型のリテラルにフィールド値を記述する時と非常によく似ています。詳細な解説は[項8.16.5](#)を参照してください。

例:

```
-- 3を含み、7を含まない。その間の数はすべて含まれる
SELECT '[3,7) '::int4range;

-- 3も7も含まないが、その間の数はすべて含まれる
SELECT '(3,7) '::int4range;

-- 1つの値、4だけを含む
SELECT '[4,4] '::int4range;

-- 含まれる点は何もない('empty'に正規化される)
SELECT '[4,4) '::int4range;
```

8.17.6. 範囲の生成

範囲型には、その範囲型と同じ名前のコンストラクタ関数があります。コンストラクタ関数を使うと、境界値の指定で余計な引用を使わずに済むので、リテラルの定数で範囲を記述するよりも便利なことが多いでしょう。コンストラクタ関数は2つ、または3つの引数をとります。引数が2つの形式では、(閉じた下限値, 開いた上限値)という標準的な形式の範囲を生成します。引数が3つの形式では、3番目の引数で指定した形式の境界の範囲を生成します。3番目の引数は、以下の文字列のいずれかでなければなりません。"`()`"、"`()`"、"`[]`"、または"`[]`"。例えば、

```
-- 完全な形式では、下限値、上限値、そして境界が開いているか閉じているかを
-- 示す文字列の引数を指定する
```



```
SELECT numrange(1.0, 14.0, '[]');

-- 3番目の引数が省略されると、'[]'を指定したのと同じになる
SELECT numrange(1.0, 14.0);

-- ここでは'[]'を指定しているが、int8rangeは離散的な範囲型(下記参照)なので
-- 正規化された形式に変換されて表示される
SELECT int8range(1, 14, '[]');

-- 境界値にNULLを指定すると、範囲の上限、あるいは下限がないことになる
SELECT numrange(NULL, 2.2);
```

8.17.7. 離散的な範囲型

離散的な範囲とは、`integer`や`date`のように明確に定義された"ステップ"のある要素型の範囲のことです。このような型において、2つの要素の間に有効な値が1つもないとき、その2つの要素は隣接している、と言います。これは連続的な範囲と対照的です。連続的な範囲では、任意の2つの値について、それらの間に別の値を見つけることが、いつでも(あるいは、ほとんどいつでも)可能です。例えば、`numeric`型や`timestamp`型の範囲は連続的です。(`timestamp`の精度は限界があるので、理論的には離散的として取り扱うことも可能ですが、ステップの大きさについて関心がないのが普通ですから、連続的であると考えの方が良いでしょう。)

離散的な範囲型に関するもう1つの考え方は、各要素の値について、"次"あるいは"前"の値というもののが明確に考えられるか、ということです。これを知っていれば、範囲の境界の閉じた表現、あるいは開いた表現について、その値の次、あるいは前の値を使って、表現を変換することができます。例えば、整数の範囲型`[4, 8]`と`(3, 9)`は同じ値の集合を意味しますが、これが`numeric`の範囲型であったならそうではありません。

離散的な範囲型はその要素型で使いたいステップのサイズを認識する正規化関数を持つべきです。正規化関数は同等な値の範囲型を、同一の表現に、特に、閉じた境界、開いた境界について一定の形式に変換します。正規化関数が指定されない場合、異なる形式の範囲は必ず等しくないものとして扱われます。これは例え、それらが現実的に同じ値の集合であったとしても、等しくないとされます。

組み込みの範囲型である`int4range`、`int8range`、および`daterange`はいずれも閉じた下限値と開いた上限値、つまり`[]`の正規化形式を使います。しかし、ユーザ定義の範囲型はこれとは別の方式を使うことができます。

8.17.8. 新しい範囲型の定義

独自の範囲型を定義することもできます。もっともありそうな理由は、組み込みの範囲型では、その派生元型についての範囲型が提供されていない、ということでしょう。例えば、`float8`を派生元型とする新しい範囲型を定義するには次のようにします。

```
CREATE TYPE floatrange AS RANGE (
    subtype = float8,
    subtype_diff = float8mi
);

SELECT '[1.234, 5.678]':floatrange;
```

`float8`には意味のある"ステップ"がないので、この例では正規化関数を定義していません。

派生元型が、連続的ではなく離散的な値を持つと考えられる場合は、`CREATE TYPE`コマンドで`canonical`(正規化)関数を指定する必要があります。正規化関数は、範囲の値を入力として受け取り、それと同等な範囲の値を返さなければなりません。戻り値は、入力値とは異なる境界値と形式になっているかもしれません。同じ値の集合を表す範囲、例えば、整数の範囲である`[1, 7]`と`[1, 8)`の正規化出力は、同一である必要があります。異なる形式の同等な値が、いつでも同じ形式の同じ値に変換されるのであれば、正規化出力の形式は何であってかまいません。正規化関数は、閉じた境界、開いた境界の形式を調整するだけではありません。派生元型が格納できるよりも大きなサイズのステップを使いたい場合は境界値を丸めることもあります。例えば`timestamp`の範囲型をステップのサイズを1時間として定義することができます。このとき、正規化関数は1時間の倍数になっていない境界値を丸める必要があります。あるいは、その代わりにエラーを投げることもできます。

独自の範囲型を定義すると、派生元型とは異なるB-tree演算子クラスや照合順を指定でき、どの値が指定の範囲に入るかを決定するソート順を変更することもできます。

また、GiSTインデックスと一緒に使われる範囲型は、派生元型の差分、つまり`subtype_diff`関数を定義すべきです。(GiSTインデックスは`subtype_diff`がなくても機能しますが、差分関数が提供されている時に比べると、あまり効果的でないことが多いでしょう。)派生元型の差分関数は、2つの派生元型の入力値をとり、その差分(つまり、 x 引く y)を`float8`型の値として返します。上の例では、通常の`float8`のマイナス演算子が呼び出す関数を使うことができますが、それ以外の派生元型では何らかの型変換が必要となるでしょう。差分をいかにして数字で表現するかについて、創造的な発想も必要になるかもしれません。可能な限りにおいて、`subtype_diff`関数は、選択した演算子クラスと照合順が示唆するソート順と矛盾しないようにすべき、つまり、ソート順で、1番目の引数が2番目の引数より上に来る場合は、必ず差分関数の結果は正になるべきです。

範囲型の作成について、より詳細な情報は[CREATE TYPE](#)を参照してください。

8.17.9. インデックス

範囲型の列にGiSTインデックスを作成することができます。例えば、

```
CREATE INDEX reservation_idx ON reservation USING gist (during);
```

GiSTインデックスがあると、以下の範囲演算子を含む検索を高速に実行できます。 =、&&、<@、@>、<<、>>、-|-、&<、および> (より詳細な情報は[表9-43](#)を参照してください)。

さらに、範囲型の列にB-treeおよびハッシュインデックスを作ることもできます。これらのインデックスについては、基本的に、等値演算のみが有効な範囲の演算です。範囲の値についてB-treeのソート順が、<および>演算子について定義されていますが、現実にはこの順序はあまり意味がなく、有効ではありません。範囲型のB-treeとハッシュのサポートは実際にインデックスを作ることよりも、むしろ、検索時に内部的にソートやハッシュをできるようにするのが主な目的です。

8.17.10. 範囲の制約

UNIQUEはスカラー値には自然な制約ですが、範囲型には通常は適当ではありません。代わりに排他(exclude)制約を使うことの方が適切なことが多いです([CREATE TABLE ... CONSTRAINT ... EXCLUDE](#)を参照してください)。排他制約により、範囲型について"重なりがない"などといった制約を指定することができます。例えば、

```
ALTER TABLE reservation ADD EXCLUDE USING gist (during WITH &&);
```

この制約は、テーブル上で重なりのある値が同時に存在することを防ぎます。

```
INSERT INTO reservation VALUES
    (1108, '[2010-01-01 11:30, 2010-01-01 13:00)');
INSERT 0 1

INSERT INTO reservation VALUES
    (1108, '[2010-01-01 14:45, 2010-01-01 15:45)');
ERROR:  conflicting key value violates exclusion constraint "reservation_during_excl"
DETAIL:  Key (during)=([ 2010-01-01 14:45:00, 2010-01-01 15:45:00 ]) conflicts
with existing key (during)=([ 2010-01-01 14:30:00, 2010-01-01 15:30:00 ]).
```

[btree_gist](#)の拡張を使って通常のスカラーのデータ型について排他制約を定義することができます。これをさらに範囲の排他と組み合わせることで大きな柔軟性を得ることができます。例えば、[btree_gist](#)をインストールした時、次の制約は範囲の重なりについて、会議室の部屋番号も同じ時にのみ拒絶します。

```
CREATE TABLE room_reservation (
    room text,
    during tsrange,
    EXCLUDE USING gist (room WITH =, during WITH &&)
);

INSERT INTO room_reservation VALUES
    ('123A', '[2010-01-01 14:00, 2010-01-01 15:00)');
INSERT 0 1

INSERT INTO room_reservation VALUES
    ('123A', '[2010-01-01 14:30, 2010-01-01 15:30)');
ERROR:  conflicting key value violates exclusion constraint "room_reservation_room_during_excl"
DETAIL:  Key (room, during)=(123A, [ 2010-01-01 14:30:00, 2010-01-01 15:30:00 ]) conflicts with
existing key (room, during)=(123A, [ 2010-01-01 14:00:00, 2010-01-01 15:00:00 ]).
```

```
INSERT INTO room_reservation VALUES
    ('123B', '[2010-01-01 14:30, 2010-01-01 15:30)');
INSERT 0 1
```

[前のページ](#)
複合型

[ホーム](#)
[上に戻る](#)

[次のページ](#)
オブジェクト識別子データ型

8.18. オブジェクト識別子データ型

オブジェクト識別子（OID）はPostgreSQLの内部で様々なシステムテーブルのプライマリキーとして使用されます。また、テーブル作成時にWITH OIDSが指定されているか、[default_with_oids](#)設定変数が有効な場合を除き、ユーザ作成のテーブルにはOIDは追加されません。oidデータ型はオブジェクト識別子を表します。oidには別名型もいくつかあります。regproc、regprocedure、regoper、regoperator、regclass、regtype、regconfigおよびregdictionaryです。[表8-23](#)に概要を示します。

oidデータ型は現在、符号なし4バイト整数として実装されています。これは、大規模なデータベース全体にわたって一意とするには十分ではありません。大規模な個別のテーブル内においても十分ではないかもしれません。そのため、ユーザ作成テーブルのOID列をプライマリキーとして使用するのはお勧めできません。OIDの使用は、システムテーブルへの参照の場合にのみ推奨されます。

oidデータ型自体は、比較以外の演算はほとんど行いません。しかし、整数としてキャストすることもでき、その場合標準の整数演算子を使用して操作することができます。（これを行うと、符号付きと符号なしの間で混乱が起きかねないことに注意してください。）

OIDの別名データ型は、専用の入出力ルーチン以外には演算を行いません。これらのルーチンでは、oid型が使用するような未加工の数値ではなく、システムオブジェクト用のシンボル名を受け入れたり表示したりできます。別名データ型により、オブジェクトのOID値の検索が簡単になります。例えば、mytableテーブルに関連したpg_attribute行を確認するには、以下のよう記述することができます。

```
SELECT * FROM pg_attribute WHERE attrelid = 'mytable'::regclass;
```

次のように記述する必要はありません。

```
SELECT * FROM pg_attribute
WHERE attrelid = (SELECT oid FROM pg_class WHERE relname = 'mytable');
```

後者もそう悪くは見えますが、これは過度に単純化されています。異なるスキーマにmytableテーブルが複数ある場合には、正しいOIDを選択するために、より複雑なSELECTが必要となります。regclass入力変換ではスキーマパスの設定に従ってテーブル検索を扱いますので、自動的に"正しい検索"を行います。同様に、テーブルのOIDをregclassにキャストすることは、数値のOIDのシンボル表示に便利です。

表 8-23. オブジェクト識別子データ型

型名	参照	説明	値の例
oid	すべて	数値オブジェクト識別子	564182
regproc	pg_proc	関数名	sum
regprocedure	pg_proc	引数の型を持つ関数	sum(int4)
regoper	pg_operator	演算子名	+
regoperator	pg_operator	引数の型を持つ演算子	*(integer,integer) or -(NONE,integer)
regclass	pg_class	リレーション名	pg_type
regtype	pg_type	データ型名	integer
regconfig	pg_ts_config	テキスト検索設定	english
regdictionary	pg_ts_dict	テキスト検索辞書	simple

すべてのOID別名型はスキーマ修飾名を受け入れ、出力時にスキーマ修飾名を表示します。ただし、現在の検索パスでオブジェクトが見つけれなければ、修飾せずに出力します。regprocとregoper別名型は、一意な（オーバーロードしていない）名前のみを入力として受け入れるため、これらの使用には限度があります。ほとんどの場合、regprocedureまたはregoperatorを使用するのが適切です。regoperatorの場合、単項演算子は未使用のオペランドをNONEと記述することによって指定されます。

OID別名型のさらなる属性は依存性の作成です。これらの型の1つの定数が格納された式内に存在する場合（列のデフォルト式やビューなど）、参照されるオブジェクトへの依存性を生成します。例えば、列がnextval('my_seq'::regclass)というデフォルト式を持つ場合、PostgreSQLはデフォルト式がmy_seqシーケンスに依存することを理解します。システムは先にこのデフォルト式が削除されない限り、このシーケンスを削除させません。

システムが使用するもう1つの識別子の型はxid、すなわちトランザクション（略してxact）識別子です。これはxminシステム列およびxmaxシステム列のデータ型です。トランザクション識別子は32ビット長です。

システムが使用する3つ目の識別子はcid、すなわちコマンド識別子です。これはcminシステム列およびcmaxシステム列のデータ型です。 コマンド識別子も32ビット長です。

システムが使用する最後の識別子はtid、すなわちタプル識別子（行識別子）です。 これはctidシステム列のデータ型です。 タプルIDはテーブル内の行の物理的位置を識別するための組（ブロック番号、ブロック内のタプルインデックス）です。

（システム列の詳細は[項5.4](#)で説明します。）

[前のページ](#)
範囲型

[ホーム](#)
[上に戻る](#)

[次のページ](#)
疑似データ型

8.19. 疑似データ型

PostgreSQL 型システムには、疑似データ型と総称される特殊用途のエントリが多数含まれます。疑似データ型は列データ型としては使用できませんが、関数の引数や結果データ型を宣言するために使用できます。これらの使用可能な疑似データ型は、ある関数の振舞いが、特定のSQLデータ型の値を単に取得したり返したりする操作に対応していない場合に便利です。表8-24に既存の疑似データ型を列挙します。

表 8-24. 疑似データ型

型名	説明
any	関数がどのような入力データ型でも受け入れることを示します。
anyelement	関数がどのような入力データ型でも受け入れることを示します。(参照 項35.2.5)
anyarray	関数がどのような配列データ型でも受け入れることを示します (項35.2.5 を参照してください)。
anynonarray	関数がどのような非配列データ型でも受け入れることを示します (項35.2.5 を参照してください)。
anyenum	関数が何らかの列挙データ型を受け付けることを示します (項35.2.5 および 項8.7 を参照してください)。
anyrange	ファンクションが範囲データ型を受け付けることを示します(項35.2.5 と 項8.17 を参照してください)。
cstring	関数がヌル終端のC文字列を受け入れる、もしくは返すことを示します。
internal	関数がサーバ内部用データ型を受け入れる、もしくは返すことを示します。
language_handler	手続き言語呼び出しハンドラはlanguage_handlerを返すものとして宣言されます。
fdw_handler	外部データラッパは fdw_handlerを返すものとして宣言されます。
record	未指定の行型を返す関数を指定します。
trigger	トリガ関数はtriggerを返すものとして宣言されます。
void	関数が値を返さないことを示します。
opaque	過去に上記の目的すべてを果たしていた古いデータ型の名前です。

C言語で作成された関数（それが組み込みか動的にロードされるかに関係なく）は、これらの疑似データ型のどれでも受け入れたり返したりするように宣言することができます。引数型として疑似データ型が使用されても関数が安全に機能するように、関数の作成時に気を付ける必要があります。

手続き型言語で作成された関数では、実装する言語によって許可された疑似データ型のみを使用できます。現在、すべての手続き型言語では疑似データ型を引数型として使用することが原則として禁止されており、結果型としてのvoidとrecord（および関数がトリガとして使用される場合のtrigger）のみが許可されています。ま

た、anyelement、anyarray、anynonarray、anyenumおよびanyrange型を使用する多様性関数をサポートするものもあります。

internal疑似データ型は、データベースシステムによって内部的にのみ呼び出される関数を宣言する場合に使用され、SQL問い合わせでの直接呼び出しには使用できません。関数に少なくとも1つのinternal型の引数があると、これをSQLから呼び出すことはできません。この制限の影響からデータ型の安全性を保持するためには、次のコーディング規則に従うことが重要です。

internal引数が少なくとも1つある場合を除き、internalを返すと宣言される関数を作成すべきではありません。

第 9 章関数と演算子

目次

- 9.1. [論理演算子](#)
- 9.2. [比較演算子](#)
- 9.3. [算術関数と演算子](#)
- 9.4. [文字列関数と演算子](#)
- 9.5. [バイナリ文字列関数と演算子](#)
- 9.6. [ビット文字列関数と演算子](#)
- 9.7. [パターンマッチ](#)
 - 9.7.1. [LIKE](#)
 - 9.7.2. [SIMILAR TO](#)正規表現
 - 9.7.3. [POSIX](#)正規表現
- 9.8. [データ型書式設定関数](#)
- 9.9. [日付/時刻関数と演算子](#)
 - 9.9.1. [EXTRACT, date_part](#)
 - 9.9.2. [date_trunc](#)
 - 9.9.3. [AT TIME ZONE](#)
 - 9.9.4. [Current Date/Time](#)
 - 9.9.5. [遅延実行](#)
- 9.10. [列挙型サポート関数](#)
- 9.11. [幾何関数と演算子](#)
- 9.12. [ネットワークアドレス関数と演算子](#)
- 9.13. [テキスト検索関数と演算子](#)
- 9.14. [XML関数](#)
 - 9.14.1. [XML内容の生成](#)
 - 9.14.2. [XML述語](#)
 - 9.14.3. [XMLの処理](#)
 - 9.14.4. [XMLにテーブルをマップ](#)
- 9.15. [JSON関数](#)
- 9.16. [シーケンス操作関数](#)
- 9.17. [条件式](#)
 - 9.17.1. [CASE](#)
 - 9.17.2. [COALESCE](#)
 - 9.17.3. [NULLIF](#)
 - 9.17.4. [GREATEST](#)および[LEAST](#)
- 9.18. [配列関数と演算子](#)
- 9.19. [範囲関数と演算子](#)
- 9.20. [集約関数](#)
- 9.21. [ウィンドウ関数](#)
- 9.22. [副問い合わせ式](#)
 - 9.22.1. [EXISTS](#)
 - 9.22.2. [IN](#)
 - 9.22.3. [NOT IN](#)
 - 9.22.4. [ANY/SOME](#)
 - 9.22.5. [ALL](#)
 - 9.22.6. [行に関する比較](#)
- 9.23. [行と配列の比較](#)
 - 9.23.1. [IN](#)
 - 9.23.2. [NOT IN](#)
 - 9.23.3. [ANY/SOME \(array\)](#)
 - 9.23.4. [ALL \(array\)](#)

9.23.5. [行に関する比較](#)

9.24. [集合を返す関数](#)

9.25. [システム情報関数](#)

9.26. [システム管理関数](#)

9.26.1. [構成設定関数](#)

9.26.2. [サーバシグナル送信関数](#)

9.26.3. [バックアップ制御関数](#)

9.26.4. [リカバリ制御関数](#)

9.26.5. [スナップショット同期関数](#)

9.26.6. [データベースオブジェクト管理関数](#)

9.26.7. [汎用ファイルアクセス関数](#)

9.26.8. [勧告的ロック用関数](#)

9.27. [トリガ関数](#)

PostgreSQLは組み込みデータ型に対して数多くの関数と演算子を用意しています。また、[パートV](#)で解説しているように、ユーザは独自の関数と演算子を定義することもできます。psqlの`\df` コマンドと`\do` コマンドはそれぞれ全ての使用可能な関数と演算子をリストするのに使用されます。

もし移植性が気になるのであれば、本章で説明する大多数の関数と演算子は、ほとんどの基本的算術演算子、比較演算子およびいくつかの明示的に印を付けた関数を除いて、標準SQLで規定されていない点に注意してください。この拡張機能のいくつかは、他のSQLデータベース管理システムにも備わっており、多くの場合この機能には各種実装間で互換性と整合性があります。同時に、本節は完全なものではありません。追加の関数はマニュアルの関連のある節に出てきます。

[前のページ](#)

疑似データ型

[ホーム](#)

[上に戻る](#)

[次のページ](#)

論理演算子

9.1. 論理演算子

通常の論理演算子が使用できます。

AND
OR
NOT

SQLはtrue、false、そして"不明"を意味するNULL値の3値のロジックシステムを使用します。以下の真理値表を参照してください。

<i>a</i>	<i>b</i>	<i>a</i> AND <i>b</i>	<i>a</i> OR <i>b</i>
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	NULL	NULL	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	NULL	FALSE	NULL
NULL	NULL	NULL	NULL

<i>a</i>	NOT <i>a</i>
TRUE	FALSE
FALSE	TRUE
NULL	NULL

AND演算子とOR演算子は交換可能です。つまり、結果に影響を与えることなく左右のオペランドを交換することができます。しかし、副式の評価順についてのより詳細については[項4.2.14](#)を参照してください。

9.2. 比較演算子

表9-1に示す、通常の比較演算子は使用可能です。

表 9-1. 比較演算子

演算子	説明
<	小なり
>	大なり
<=	等しいかそれ以下
>=	等しいかそれ以上
=	等しい
<> または !=	等しくない

注意: !=演算子は構文解析で<>に変換されます。!=演算子と<>演算子に異なる処理を行わせる実装はできません。

比較演算子は関連性のある全てのデータ型で使用できます。全ての比較演算子は二項演算子で、boolean データ型を返します。1 < 2 < 3 のような式は（ブール値と3を比較する<演算子がないので）無効です。

比較演算子に加えて特殊なBETWEEN構文が使えます。

```
a BETWEEN x AND y
```

は

```
a >= x AND a <= y
```

BETWEENは範囲内に含まれるとして終点値を扱うことに注意してください。NOT BETWEENはその反対の比較をします（指定した値は包括しません）。

```
a NOT BETWEEN x AND y
```

は

```
a < x OR a > y
```

と同一です。BETWEEN SYMMETRICは、ANDの左側の引数が右側の引数より小さいか、もしくは等しいという必要性が無い点を除きBETWEENと同じです。この条件を満たしていない場合、2つの引数は自動的に置き換えられますので、常に空ではない範囲となります。

値がNULLかNULLでないかを検証するには次の構文を使います。

```
expression IS NULL
expression IS NOT NULL
```

あるいは、これと同等の、非標準の構文も使えます。

```
expression ISNULL
expression NOTNULL
```

NULLとNULLとは"等しい"関係にはありませんので、expression = NULLと記述してはいけません（NULL値は不明の値を表しているため、不明な値同士が同じかどうかは識別できません）。これは標準SQLに従った動作です。

ティップ: アプリケーションによっては、expression = NULLが、expressionがNULL値と評価されるのであれば真を返すことを期待することがあります。こうしたアプリケーションは標準SQLに従うように改修することを強く推奨します。しかし、それができなければtransform_null_equalsを使用することで対応することができます。これを有効にした場合、PostgreSQLはx = NULL句をx IS NULLに変換します。

注意: expressionが行値の場合、行式自体がNULLまたは、行のフィールドすべてがNULLの場合にIS NULLは真となります。一方IS NOT NULLは、行式自体が非NULLまたは、行のフィールドすべてが非NULLの場合に真となります。この動作により、IS NULLおよびIS NOT NULLは行値評価式に対し常に反対の結果を返しません。つまりNULLと非NULLの値の両方を含む行値式はどちらの試験でも偽を返します。この定義は標準SQLに従ったもので、8.2より前のバージョンのPostgreSQLにおける一貫性のない動作から変更されました。

入力のどちらかが真または偽ではなくNULLの場合、通常の比較演算子は（"不明"を意味する）nullを生成します。例えば7 = NULLはnullになります。7 <> NULLも同様です。この動作が適切でない場合は、IS [NOT] DISTINCT FROM構文を使用してください。

```
expression IS DISTINCT FROM expression
expression IS NOT DISTINCT FROM expression
```

非NULLの入力では、IS DISTINCT FROMは<>演算子と同じです。しかし、入力がどちらもNULLの場合、これは偽を返し、片方の入力のみがNULLの場合は真を返します。同様に、IS NOT DISTINCT FROMは非NULL入力では=と同じですが、両方の入力がNULLであれば真を、片方のみがNULLの場合は偽を返します。このように、これらの構文はNULLを"不明な値"ではなく、通常の値かのように動作します。

論理値も次の構文で検証できます。

```
expression IS TRUE
expression IS NOT TRUE
expression IS FALSE
expression IS NOT FALSE
expression IS UNKNOWN
expression IS NOT UNKNOWN
```

これらは、常に真か偽を返し、演算項目がNULLであってもNULL値を返すことはありません。NULL値が入力されると、"不明"という論理値として扱われます。IS UNKNOWNとIS NOT UNKNOWNが、入力式が論理値型でなければならないという点を除き、それぞれ実質的にIS NULLとIS NOT NULLと同じであることに注意してください。

9.3. 算術関数と演算子

PostgreSQLの数多くの型に対する算術演算子が用意されています。標準算術表現法に従わない型（例えば、日付/時刻データ型）については、後続する節で実際の動作を説明します。

表9-2は使用可能な算術演算子を示しています。

表 9-2. 算術演算子

演算子名	説明	例	結果
+	和	2 + 3	5
-	差	2 - 3	-1
*	積	2 * 3	6
/	商（整数の割り算では余りを切り捨て）	4 / 2	2
%	剰余（余り）	5 % 4	1
^	累乗	2.0 ^ 3.0	8
/	平方根	/ 25.0	5
/	立方根	/ 27.0	3
!	階乗	5 !	120
!!	階乗（前置演算子）	!! 5	120
@	絶対値	@ -5.0	5
&	バイナリのAND	91 & 15	11
	バイナリのOR	32 3	35
#	バイナリのXOR	17 # 5	20
~	バイナリのNOT	~1	-2
<<	バイナリの左シフト	1 << 4	16
>>	バイナリの右シフト	8 >> 2	2

バイナリ演算子は内部データ型のみに使用できます。一方他の演算子は全ての数値データ型に使用できます。また、バイナリ演算子は表9-10に示す、bit、bit varying ビット文字列型に対しても使用することができます。

表9-3に使用可能な算術関数を示します。この表内のdpは、double precisionを意味します。これら関数の多くは、異なる引数型を持つ複数の形で提供されています。特別な場合を除き、ある任意形式の関数はその引数と同じデータ型を返します。double precisionデータに対する関数のほとんどはホストシステムのCライブラリの上層に実装されています。このため、境界近くの場合の精度と振舞いはホストシステムに依存して変わります。

表 9-3. 算術関数

関数	戻り値型	説明	例	結果
abs(x)	(入力と同じ)	絶対値	abs(-17.4)	17.4
cbrt(dp)	dp	立方根	cbrt(27.0)	3
ceil(dp or numeric)	(入力型と同一)	引数より小さくない最小の整数	ceil(-42.8)	-42
ceiling(dp or numeric)	(入力型と同一)	引数より小さくない最小の整数（ceilの別名）	ceiling(-95.3)	-95
degrees(dp)	dp	ラジアンに対応する度	degrees(0.5)	28.6478897565412
div(y numeric, x numeric)	numeric	y/xの整数商	div(9,4)	2
exp(dp or numeric)	(入力型と同一)	指数	exp(1.0)	2.71828182845905

<code>floor(dp or numeric)</code>	(入力型 と同一)	引数より大きくない最大の整数	<code>floor(-42.8)</code>	-43
<code>ln(dp or numeric)</code>	(入力型 と同一)	自然対数	<code>ln(2.0)</code>	0.693147180559945
<code>log(dp or numeric)</code>	(入力型 と同一)	10を底とした対数 (常用対数)	<code>log(100.0)</code>	2
<code>log(b numeric, x numeric)</code>	numeric	bを底とした対数	<code>log(2.0, 64.0)</code>	6.0000000000
<code>mod(y, x)</code>	(引数の 型と同一)	y/xの剰余	<code>mod(9, 4)</code>	1
<code>pi()</code>	dp	"円周率 (π) "定数	<code>pi()</code>	3.14159265358979
<code>power(a dp, b dp)</code>	dp	aのb乗	<code>power(9.0, 3.0)</code>	729
<code>power(a numeric, b numeric)</code>	numeric	aのb乗	<code>power(9.0, 3.0)</code>	729
<code>radians(dp)</code>	dp	度に対応するラジアン	<code>radians(45.0)</code>	0.785398163397448
<code>random()</code>	dp	0.0 <= x < 1.0の範囲の乱数値	<code>random()</code>	
<code>round(dp or numeric)</code>	(入力型 と同一)	四捨五入	<code>round(42.4)</code>	42
<code>round(v numeric, s int)</code>	numeric	小数点sの位まで四捨五入して切捨て	<code>round(42.4382, 2)</code>	42.44
<code>setseed(dp)</code>	void	今後の <code>random()</code> 呼び出しで使用されるシード(種)の設定 (-1.0から1.0までの境界を含む値)	<code>setseed(0.54823)</code>	
<code>sign(dp or numeric)</code>	(入力型 と同一)	引数の符号 (-1、0、+1)	<code>sign(-8.4)</code>	-1
<code>sqrt(dp or numeric)</code>	(入力型 と同一)	平方根	<code>sqrt(2.0)</code>	1.4142135623731
<code>trunc(dp or numeric)</code>	(入力型 と同一)	切り捨て	<code>trunc(42.8)</code>	42
<code>trunc(v numeric, s int)</code>	numeric	sの桁で切り捨て	<code>trunc(42.4382, 2)</code>	42.43
<code>width_bucket(op numeric, b1 numeric, b2 numeric, count int)</code>	int	operandが割り当てられる、b1からb2までのバケット数countの等深度ヒストグラムのバケットを返す	<code>width_bucket(5.35, 0.024, 10.06, 5)</code>	3
<code>width_bucket(op dp, b1 dp, b2 dp, count int)</code>	int	operandが割り当てられる、b1からb2までのバケット数countの等深度ヒストグラムのバケットを返す	<code>width_bucket(5.35, 0.024, 10.06, 5)</code>	3

最後に、使用可能な三角関数を表9-4に示します。全ての三角関数はdouble precisionデータ型の引数と戻り値を取ります。三角関数の引数はラジアンで表されます。逆関数の戻り値はラジアンで表されます。上記の`radians()`と`degrees()`といった単位変換関数を参照して下さい。

表 9-4. 三角関数

関数	説明
<code>acos(X)</code>	逆余弦関数
<code>asin(X)</code>	逆正弦関数
<code>atan(X)</code>	逆正接関数
<code>atan2(y, x)</code>	y/xの逆正接関数
<code>cos(X)</code>	余弦関数
<code>cot(X)</code>	余接関数
<code>sin(X)</code>	正弦関数
<code>tan(X)</code>	正接関数

9.4. 文字列関数と演算子

本節では文字列の値の調査や操作のための関数と演算子について説明します。ここでの文字列とは`character`データ型、`character varying`データ型、および`text`データ型の値を含みます。補足説明のない限り、下記に挙げている全ての関数はこれら全てのデータ型に対して使用できますが、`character`データ型を使用した場合、自動的に空白文字がパッドされるという潜在的作用がありますので注意してください。バイナリ列データ型にたいしてもともとから存在するいくつかの関数もあります。

SQLでは引数の区切りにカンマではなくキーワードを使用する文字列関数をいくつか定義しています。詳細については表9-5を参照してください。またPostgreSQLは、これらの関数に対して通常の関数呼び出し構文を使用するバージョンを提供します（表9-6を参照してください）。

注意: PostgreSQL 8.3以前において、これらの関数はいくつかの非文字列データ型の値を警告なしに受け付けたのは、それらデータ型を暗黙的に`text`型に型変換していたことによります。この強制的な変換は、頻繁に予期しない動作の原因となったので削除されました。しかし、文字列連結演算子（`||`）は表9-5で示されるように、少なくともひとつの入力が文字列型であれば、依然として非文字列入力を受け付けます。その他の場合、以前と同じ動作を期待するのなら、`text`への明示的な変換を行ってください。

表 9-5. SQL文字列関数と演算子

関数	戻り 値型	説明	例	結果
<code>string string</code>	<code>text</code>	文字列結合	<code>'Post' 'greSQL'</code>	PostgreSQL
文字列 非文字列 または、非文字列 文字列	<code>text</code>	ひとつの非文字列のある入力の文字列結合	<code>'Value: ' 42</code>	Value: 42
<code>bit_length(string)</code>	<code>int</code>	文字列中のビット数	<code>bit_length('jose')</code>	32
<code>char_length(string)</code> または <code>character_length(string)</code>	<code>int</code>	文字列中の文字数	<code>char_length('jose')</code>	4
<code>lower(string)</code>	<code>text</code>	文字列を小文字に変換	<code>lower('TOM')</code>	tom
<code>octet_length(string)</code>	<code>int</code>	文字列中のバイト数	<code>octet_length('jose')</code>	4
<code>overlay(string placing string from int [for int])</code>	<code>text</code>	部分文字列の置換	<code>overlay('Txxxxas' placing 'hom' from 2 for 4)</code>	Thomas
<code>position(substring in string)</code>	<code>int</code>	指定した部分文字列の場所	<code>position('om' in 'Thomas')</code>	3
<code>substring(string [from int] [for int])</code>	<code>text</code>	部分文字列の取り出し	<code>substring('Thomas' from 2 for 3)</code>	hom
<code>substring(string from pattern)</code>	<code>text</code>	POSIX正規表現に一致する副文字列を取り出します。パターンマッチに関してより詳細は、項9.7を参照してください。	<code>substring('Thomas' from '...\$')</code>	mas
<code>substring(string from pattern for escape)</code>	<code>text</code>	SQL正規表現に一致する部分文字列を取り出します。パターンマッチに関してより詳細は、項9.7を参照してください。	<code>substring('Thomas' from '%#o_a#"_" for '#')</code>	oma
<code>trim([leading trailing both] [characters] from string)</code>	<code>text</code>	<code>characters</code> （デフォルトでは空白）で指定された文字のみを含む最も長い文字列を、 <code>string</code> の先頭、末尾、そしてその両方から削除します。	<code>trim(both 'x' from 'xTomxx')</code>	Tom
<code>upper(string)</code>	<code>text</code>	文字列を大文字に変換	<code>upper('tom')</code>	TOM

この他、表9-6に列挙する文字列操作関数が使えます。そのいくつかは、表9-5で一覧した標準SQLの文字列関数を実装するため、内部的に使用されます。

表 9-6. その他の文字列関数

関数	戻り値 型	説明	例	結果
<code>ascii(string)</code>	<code>int</code>	引数の最初の文字のASCIIコード。UTF8に対しては文字のUnicodeコードポイントを返します。その他のマルチバイト符号化方式の場合、引数はASCII文字でなくてはなりません。	<code>ascii('x')</code>	120
<code>btrim(string text [, characters text])</code>	<code>text</code>	<code>characters</code> （デフォルトは空白文字）で指定された文字のみを有する最長の文字列を、 <code>string</code> の先頭と末尾から削除します。	<code>btrim('xyxtrimyxx', 'xy')</code>	trim
		与えられたコードの文字。UTF8では引数		

<code>chr(int)</code>	text	はUnicodeコードポイントとして扱われます。その他のマルチバイト符号化方式で引数はASCII文字である必要があります。 NULL (0)文字はテキストデータ型がそのようなバイトを格納することができないので許可されません。	<code>chr(65)</code>	A
<code>concat(str "any" [, str "any" [, ...]])</code>	text	すべての引数の結合。ただしNULLは無視される。	<code>concat('abcde', 2, NULL, 22)</code>	abcde222
<code>concat_ws(sep text, str "any" [, str "any" [, ...]])</code>	text	第一引数を区切り文字とし、残りのすべての引数を結合する。最初の引数は区切り文字として使われる。NULLは無視される。	<code>concat_ws(',', 'abcde', 2, NULL, 22)</code>	abcde,2,22
<code>convert(string bytea, src_encoding name, dest_encoding name)</code>	bytea	文字列を <code>dest_encoding</code> に変換します。元の符号化方式は <code>src_encoding</code> で指定されます。 <code>string</code> はこの符号化方式で有効でなければなりません。変換はCREATE CONVERSIONで定義されます。有効な変換については 表9-7 を参照してください。	<code>convert('text_in_utf8', 'UTF8', 'LATIN1')</code>	Latin-1 (ISO 8859-1) 符号化方式のtext_in_utf8
<code>convert_from(string bytea, src_encoding name)</code>	text	データベースの符号化方式に文字列を変換します。元の符号化方式は <code>src_encoding</code> で指定されます。 <code>string</code> はこの符号化方式で有効なものでなければなりません。	<code>convert_from('text_in_utf8', 'UTF8')</code>	現在のデータベース符号化方式のtext_in_utf8
<code>convert_to(string text, dest_encoding name)</code>	bytea	文字列を <code>dest_encoding</code> に変換します。	<code>convert_to('some text', 'UTF8')</code>	UTF8符号化方式のsome text
<code>decode(string text, format text)</code>	bytea	<code>string</code> からバイナリデータを復号します。オプションの <code>format</code> は <code>encode()</code> と同じです。	<code>decode('MTIzAAE=', 'base64')</code>	¥x3132330001
<code>encode(data bytea, format text)</code>	text	バイナリデータをテキスト表現形式に符号化します。サポートされている形式は、base64、hex、escapeです。escapeは0バイトと最上位ビットがセットされているバイトを8進数のシーケンス(¥nnn)に変換し、バックスラッシュを二重化します。	<code>encode(E'123¥¥000¥¥001', 'base64')</code>	MTIzAAE=
<code>format(formatstr text [, str "any" [, ...]])</code>	text	文字列の書式を整えます。この関数はC言語関数の <code>sprintf</code> と似ています。ただし、次の変換指定子のみ受け付けます。 %sは該当する引数を文字列へ変換します。 %iは引数をSQL識別子へ変換します。 %lはSQLリテラルへ変換します。 %%は%リテラルを出力します。変換では、変換指定子の前に <code>n\$</code> を記述することで、明示的なパラメータの位置を参照することができます <code>n</code> は引数の位置です。 例39-1 も参照してください。	<code>format('Hello %s, %1\$s', 'World')</code>	Hello World, World
<code>initcap(string)</code>	text	それぞれの単語の第一文字を大文字にし、残りは小文字のまま残します。ここで単語とは、英数字以外の文字で区切られた、英数字からなる文字の並びのことです。	<code>initcap('hi THOMAS')</code>	Hi Thomas
<code>left(str text, n int)</code>	text	文字列の先頭から <code>n</code> 文字を返します。 <code>n</code> が負数の場合、 <code> n </code> の数だけ文字列の末尾から切り取った文字列を返します。	<code>left('abcde', 2)</code>	ab
<code>length(string)</code>	int	<code>string</code> 内の文字数	<code>length('jose')</code>	4
<code>length(stringbytea, encoding name)</code>	int	与えられた <code>encoding</code> での <code>string</code> 内の文字数。 <code>string</code> はこの符号化方式で有効でなければなりません。	<code>length('jose', 'UTF8')</code>	4
<code>lpad(string text, length int [, fill text])</code>	text	文字 <code>fill</code> (デフォルトは空白文字) を文字列の前に追加して、 <code>string</code> を <code>length</code> の長さにします。 <code>string</code> が既に <code>length</code> の長さを超えている場合は (右側が) 切り捨てられます。	<code>lpad('hi', 5, 'xy')</code>	xyxhi
<code>ltrim(string text [, characters text])</code>	text	<code>string</code> の最初から、 <code>characters</code> (デフォルトはスペース) で指定された文字だけ	<code>ltrim('zzzytrim', 'xyz')</code>	trim

		を有する最長の文字列を削除します。		
md5(string)	text	stringのMD5ハッシュを計算し、16進数で結果を返します。	md5('abc')	900150983cd24fb0d6963f7d28e17f72
pg_client_encoding()	name	現在のクライアントの符号化方式の名前。	pg_client_encoding()	SQL_ASCII
quote_ident(string text)	text	与えられた文字列を、SQL問い合わせ文字列で識別子として使用できるように、適切な引用符を付けて返します。引用符は、必要な場合（例えば、文字列に識別子として使用できない文字が含まれる場合や、大文字変換される場合）にのみ追加されます。埋め込まれた引用符は、適切に二重化されます。 例39-1 も参照してください。	quote_ident('Foo bar')	"Foo bar"
quote_literal(string text)	text	与えられた文字列を、SQL問い合わせ文字列で文字リテラルとして使用できるように、適切な引用符を付けて返します。埋め込まれた単一引用符およびバックスラッシュは、適切に二重化されます。quote_literalはNULL入力に対してNULLを返すことに注意してください。引数がNULLとなる可能性がある場合、よりquote_nullableの方がしばしば適しています。 例39-1 も参照してください。	quote_literal(E'O¥'Reilly')	'O''Reilly'
quote_literal(value anyelement)	text	与えられた値をテキストに強制し、そしてリテラルとして引用符付けします。組み込まれた一重引用符と逆スラッシュは適切に二重化されます。	quote_literal(42.5)	'42.5'
quote_nullable(string text)	text	与えられた文字列を、SQL問い合わせ文字列で文字リテラルとして使用できるように、適切な引用符を付けて返します。また、引数がNULLの場合、NULLを返します。埋め込まれた単一引用符およびバックスラッシュは適切に二重化されます。 例39-1 も参照してください。	quote_nullable(NULL)	NULL
quote_nullable(value anyelement)	text	与えられた値をテキストに強制し、そしてリテラルとして引用符付けします。そうでないと、引数がNULLの場合はNULLを返します。埋め込まれた単一引用符とバックスラッシュは適切に二重化されます。	quote_nullable(42.5)	'42.5'
regexp_matches(string text, pattern text [, flags text])	setof text[]	stringに対してPOSIX正規表現でマッチし、捕捉されたすべての部分文字列を返します。より詳細は 項9.7.3 を参照してください。	regexp_matches('foobarbequebaz', '(bar)(beque)')	{bar,beque}
regexp_replace(string text, pattern text, replacement text [, flags text])	text	POSIX正規表現に一致する部分文字列を置換します。より詳細は 項9.7.3 を参照してください。	regexp_replace('Thomas', '[mN]ä.', 'M')	ThM
regexp_split_to_array(string text, pattern text [, flags text])	text[]	POSIX正規表現を区切り文字に使用してstringを分割します。詳しくは 項9.7.3 を参照ください。	regexp_split_to_array('hello world', 'E'¥\$+')	{hello,world}
regexp_split_to_table(string text, pattern text [, flags text])	setof text	POSIX正規表現を区切り文字に使用してstringを分割します。詳しくは 項9.7.3 を参照ください。	regexp_split_to_table('hello world', 'E'¥\$+')	hello world (2 rows)
repeat(string text, number int)	text	指定されたnumberの数だけstringを繰り返し返します。	repeat('Pg', 4)	PgPgPgPg
replace(string text, from text, to text)	text	stringに出現する全てのfrom部分文字列をto部分文字列に置換します。	replace('abcdefabcdef', 'cd', 'XX')	abXXefabXXef
reverse(str)	text	逆順にした文字列を返します。	reverse('abcde')	edcba
right(str text, n int)	text	文字列の末尾からn文字を返します。nが負数の場合は、 n の数だけ文字列の末尾から切り取った文字列を返します。	right('abcde', 2)	de
		文字fill（デフォルトはスペース）を文		

<code>rpadd(string text, length int [, fill text])</code>	text	字列に追加して、stringをlengthの長さ にします。stringが既にlengthの長さを 超えている場合は切り捨てられます。	<code>rpadd('hi', 5, 'xy')</code>	hixyx
<code>rtrim(string text [, characters text])</code>	text	stringの末尾から、characters（デフォ ルトはスペース）で指定された文字のみ を有する最長の文字列を削除します。	<code>rtrim('trimxxxx', 'x')</code>	trim
<code>split_part(string text, delimiter text, field int)</code>	text	stringをdelimiterで分割し、その結果か ら（1から始まるように数える）指定した フィールドを返します。	<code>split_part('abc~@~def~@~ghi', '~@~', 2)</code>	def
<code>strpos(string, substring)</code>	int	指定した部分文字列の位置を特定します （position(substring in string) と同 じですが、引数の順序が逆になっている ことに注意してください）。	<code>strpos('high', 'ig')</code>	2
<code>substr(string, from [, count])</code>	text	指定した部分文字列を取り出します （substring(string from from for count)と同じです）。	<code>substr('alphabet', 3, 2)</code>	ph
<code>to_ascii(string text [, encoding text])</code>	text	stringを他の符号化方式からASCIIに変 換しま す。LATIN1、LATIN2、LATIN9、WIN1250符 号化方式からの変換のみをサポートしま す。）	<code>to_ascii('Karel')</code>	Karel
<code>to_hex(number int or bigint)</code>	text	numberを、同等の16進数表現に変換しま す。	<code>to_hex(2147483647)</code>	7fffffff
<code>translate(string text, from text, to text)</code>	text	from集合で指定された文字と一致す るstringにある全ての文字は、それに対 応するtoで指定された文字に置き換えら れます。もしfromがtoより長い場 合、fromで指定される余分な文字に一致 するものは削除されます。	<code>translate('12345', '143', 'ax')</code>	a2x5

項9.20内のstring_agg集約関数も参照してください。

表 9-7. 組み込みの変換

変換名 [a]	変換前の符号化方式	変換後の符号化方式
<code>ascii_to_mic</code>	SQL_ASCII	MULE_INTERNAL
<code>ascii_to_utf8</code>	SQL_ASCII	UTF8
<code>big5_to_euc_tw</code>	BIG5	EUC_TW
<code>big5_to_mic</code>	BIG5	MULE_INTERNAL
<code>big5_to_utf8</code>	BIG5	UTF8
<code>euc_cn_to_mic</code>	EUC_CN	MULE_INTERNAL
<code>euc_cn_to_utf8</code>	EUC_CN	UTF8
<code>euc_jp_to_mic</code>	EUC_JP	MULE_INTERNAL
<code>euc_jp_to_sjis</code>	EUC_JP	SJIS
<code>euc_jp_to_utf8</code>	EUC_JP	UTF8
<code>euc_kr_to_mic</code>	EUC_KR	MULE_INTERNAL
<code>euc_kr_to_utf8</code>	EUC_KR	UTF8
<code>euc_tw_to_big5</code>	EUC_TW	BIG5
<code>euc_tw_to_mic</code>	EUC_TW	MULE_INTERNAL
<code>euc_tw_to_utf8</code>	EUC_TW	UTF8
<code>gb18030_to_utf8</code>	GB18030	UTF8
<code>gbk_to_utf8</code>	GBK	UTF8
<code>iso_8859_10_to_utf8</code>	LATIN6	UTF8
<code>iso_8859_13_to_utf8</code>	LATIN7	UTF8
<code>iso_8859_14_to_utf8</code>	LATIN8	UTF8
<code>iso_8859_15_to_utf8</code>	LATIN9	UTF8
<code>iso_8859_16_to_utf8</code>	LATIN10	UTF8
<code>iso_8859_1_to_mic</code>	LATIN1	MULE_INTERNAL
<code>iso_8859_1_to_utf8</code>	LATIN1	UTF8
<code>iso_8859_2_to_mic</code>	LATIN2	MULE_INTERNAL
<code>iso_8859_2_to_utf8</code>	LATIN2	UTF8
<code>iso_8859_2_to_windows_1250</code>	LATIN2	WIN1250
<code>iso_8859_3_to_mic</code>	LATIN3	MULE_INTERNAL
<code>iso_8859_3_to_utf8</code>	LATIN3	UTF8
<code>iso_8859_4_to_mic</code>	LATIN4	MULE_INTERNAL
<code>iso_8859_4_to_utf8</code>	LATIN4	UTF8

iso_8859_5_to_koi8_r	ISO_8859_5	KOI8R
iso_8859_5_to_mic	ISO_8859_5	MULE_INTERNAL
iso_8859_5_to_utf8	ISO_8859_5	UTF8
iso_8859_5_to_windows_1251	ISO_8859_5	WIN1251
iso_8859_5_to_windows_866	ISO_8859_5	WIN866
iso_8859_6_to_utf8	ISO_8859_6	UTF8
iso_8859_7_to_utf8	ISO_8859_7	UTF8
iso_8859_8_to_utf8	ISO_8859_8	UTF8
iso_8859_9_to_utf8	LATIN5	UTF8
johab_to_utf8	JOHAB	UTF8
koi8_r_to_iso_8859_5	KOI8R	ISO_8859_5
koi8_r_to_mic	KOI8R	MULE_INTERNAL
koi8_r_to_utf8	KOI8R	UTF8
koi8_r_to_windows_1251	KOI8R	WIN1251
koi8_r_to_windows_866	KOI8R	WIN866
koi8_u_to_utf8	KOI8U	UTF8
mic_to_ascii	MULE_INTERNAL	SQL_ASCII
mic_to_big5	MULE_INTERNAL	BIG5
mic_to_euc_cn	MULE_INTERNAL	EUC_CN
mic_to_euc_jp	MULE_INTERNAL	EUC_JP
mic_to_euc_kr	MULE_INTERNAL	EUC_KR
mic_to_euc_tw	MULE_INTERNAL	EUC_TW
mic_to_iso_8859_1	MULE_INTERNAL	LATIN1
mic_to_iso_8859_2	MULE_INTERNAL	LATIN2
mic_to_iso_8859_3	MULE_INTERNAL	LATIN3
mic_to_iso_8859_4	MULE_INTERNAL	LATIN4
mic_to_iso_8859_5	MULE_INTERNAL	ISO_8859_5
mic_to_koi8_r	MULE_INTERNAL	KOI8R
mic_to_sjis	MULE_INTERNAL	SJIS
mic_to_windows_1250	MULE_INTERNAL	WIN1250
mic_to_windows_1251	MULE_INTERNAL	WIN1251
mic_to_windows_866	MULE_INTERNAL	WIN866
sjis_to_euc_jp	SJIS	EUC_JP
sjis_to_mic	SJIS	MULE_INTERNAL
sjis_to_utf8	SJIS	UTF8
tcvn_to_utf8	WIN1258	UTF8
uhc_to_utf8	UHC	UTF8
utf8_to_ascii	UTF8	SQL_ASCII
utf8_to_big5	UTF8	BIG5
utf8_to_euc_cn	UTF8	EUC_CN
utf8_to_euc_jp	UTF8	EUC_JP
utf8_to_euc_kr	UTF8	EUC_KR
utf8_to_euc_tw	UTF8	EUC_TW
utf8_to_gb18030	UTF8	GB18030
utf8_to_gbk	UTF8	GBK
utf8_to_iso_8859_1	UTF8	LATIN1
utf8_to_iso_8859_10	UTF8	LATIN6
utf8_to_iso_8859_13	UTF8	LATIN7
utf8_to_iso_8859_14	UTF8	LATIN8
utf8_to_iso_8859_15	UTF8	LATIN9
utf8_to_iso_8859_16	UTF8	LATIN10
utf8_to_iso_8859_2	UTF8	LATIN2
utf8_to_iso_8859_3	UTF8	LATIN3
utf8_to_iso_8859_4	UTF8	LATIN4
utf8_to_iso_8859_5	UTF8	ISO_8859_5
utf8_to_iso_8859_6	UTF8	ISO_8859_6
utf8_to_iso_8859_7	UTF8	ISO_8859_7
utf8_to_iso_8859_8	UTF8	ISO_8859_8
utf8_to_iso_8859_9	UTF8	LATIN5
utf8_to_johab	UTF8	JOHAB
utf8_to_koi8_r	UTF8	KOI8R
utf8_to_koi8_u	UTF8	KOI8U
utf8_to_sjis	UTF8	SJIS
utf8_to_tcvn	UTF8	WIN1258
utf8_to_uhc	UTF8	UHC
utf8_to_windows_1250	UTF8	WIN1250
utf8_to_windows_1251	UTF8	WIN1251
utf8_to_windows_1252	UTF8	WIN1252
utf8_to_windows_1253	UTF8	WIN1253
utf8_to_windows_1254	UTF8	WIN1254

utf8_to_windows_1255	UTF8	WIN1255
utf8_to_windows_1256	UTF8	WIN1256
utf8_to_windows_1257	UTF8	WIN1257
utf8_to_windows_866	UTF8	WIN866
utf8_to_windows_874	UTF8	WIN874
windows_1250_to_iso_8859_2	WIN1250	LATIN2
windows_1250_to_mic	WIN1250	MULE_INTERNAL
windows_1250_to_utf8	WIN1250	UTF8
windows_1251_to_iso_8859_5	WIN1251	ISO_8859_5
windows_1251_to_koi8_r	WIN1251	KOI8R
windows_1251_to_mic	WIN1251	MULE_INTERNAL
windows_1251_to_utf8	WIN1251	UTF8
windows_1251_to_windows_866	WIN1251	WIN866
windows_1252_to_utf8	WIN1252	UTF8
windows_1256_to_utf8	WIN1256	UTF8
windows_866_to_iso_8859_5	WIN866	ISO_8859_5
windows_866_to_koi8_r	WIN866	KOI8R
windows_866_to_mic	WIN866	MULE_INTERNAL
windows_866_to_utf8	WIN866	UTF8
windows_866_to_windows_1251	WIN866	WIN
windows_874_to_utf8	WIN874	UTF8
euc_jis_2004_to_utf8	EUC_JIS_2004	UTF8
ut8_to_euc_jis_2004	UTF8	EUC_JIS_2004
shift_jis_2004_to_utf8	SHIFT_JIS_2004	UTF8
ut8_to_shift_jis_2004	UTF8	SHIFT_JIS_2004
euc_jis_2004_to_shift_jis_2004	EUC_JIS_2004	SHIFT_JIS_2004
shift_jis_2004_to_euc_jis_2004	SHIFT_JIS_2004	EUC_JIS_2004

注意:
a. 変換名は、標準命名体系に従っています。変換元符号化方式の公式名の内、英数字以外の文字をアンダースコアで置き換え、その後に_to_を付け、そして変換先符号化方式名を同様に処理したものを続けたものです。したがって、これらの名称は慣習的なコード名とは異なる可能性があります。

[前のページ](#)

算術関数と演算子

[ホーム](#)

[上に戻る](#)

[次のページ](#)

バイナリ文字列関数と演算子

9.5. バイナリ文字列関数と演算子

本節ではbytea型の値を調べたり操作するための関数と演算子について説明します。

SQLでは、引数の区切りにカンマではなくキーワードを使う文字列関数を、いくつか定義しています。詳細は[表9-8](#)を参照してください。またPostgreSQLは、これらの関数に対して通常関数呼び出し構文を使用するバージョンを提供します（[表9-6](#)を参照してください）。

注意: このページで示されるサンプル結果は、サーバパラメータである[bytea_output](#)がescapeに設定されていることを想定しています。（この設定はPostgreSQLの伝統的な形式です）

表 9-8. SQLバイナリ文字列関数と演算子

関数	戻り 値型	説明	例	結果
string string	bytea	文字列結合	E'¥¥¥Post'::bytea E'¥¥047gres¥¥000'::bytea	¥¥Post'gres¥¥000
octet_length(string)	int	バイナリ文字列中のバイト数	octet_length(E'jo¥¥000se'::bytea)	5
overlay(string placing string from int [for int])	bytea	部分文字列の置換	overlay(E'Th¥¥000omas'::bytea placing E'¥¥002¥¥003'::bytea from 2 for 3)	T¥¥002¥¥003mas
position(substring in string)	int	指定した部分文字列の位置	position(E'¥¥000om'::bytea in E'Th¥¥000omas'::bytea)	3
substring(string [from int] [for int])	bytea	部分文字列の抽出	substring(E'Th¥¥000omas'::bytea from 2 for 3)	h¥¥000o
trim([both] bytes from string)	bytea	文字列stringの先頭から末尾bytesのバイトのみを含む最長の文字列を削除します。	trim(E'¥¥000'::bytea from E'¥¥000Tom¥¥000'::bytea)	Tom

この他、[表9-9](#)に列挙するバイナリ列操作関数が使えます。そのいくつかは、[表9-8](#)で説明した標準SQLの文字列関数を実装するために内部的に使用されています。

表 9-9. その他のバイナリ文字列関数

関数	戻り 値型	説明	例	結果
btrim(string bytea, bytes bytea)	bytea	bytes で指定されたバイトのみを有する最長の文字列をstringの先頭と末尾から削除します。	btrim(E'¥¥000trim¥¥000'::bytea, E'¥¥000'::bytea)	trim
decode(string text, format text)	bytea	stringで表現されているテキストデータをバイナリデータに復号化します。formatのオプションはencodeと同じです。	decode(E'123¥¥000456', 'escape')	123¥¥000456
encode(string bytea, format text)	text	バイナリデータをテキスト表現形式に符号化します。サポートされている形式は、base64、hex、escapeです。escapeは0バイトと最上位ビットがセットされているバイトを8進数のシーケンス(¥nnn)に変換し、バックスラッシュを二重化します。	encode(E'123¥¥000456'::bytea, 'escape')	123¥¥000456
get_bit(string, offset)	int	文字列から1ビット抽出します。	get_bit(E'Th¥¥000omas'::bytea, 45)	1
get_byte(string, offset)	int	文字列から1バイト抽出します。	get_byte(E'Th¥¥000omas'::bytea, 4)	109
length(string)	int	バイナリ文字列の長さ	length(E'jo¥¥000se'::bytea)	5
md5(string)	text	stringのMD5のハッシュを計算し、結果を16進数で返します。	md5(E'Th¥¥000omas'::bytea)	8ab2d3c9689aaf18b4958c334c82d8b1
set_bit(string, offset, newvalue)	bytea	文字列内のビットを設定します。	set_bit(E'Th¥¥000omas'::bytea, 45, 0)	Th¥¥000omAs
set_byte(string, offset, newvalue)	bytea	文字列内の1バイトを設定します。	set_byte(E'Th¥¥000omas'::bytea, 4, 64)	Th¥¥000o@as

`get_byte`と`set_byte`はバイナリ列の先頭バイトを0バイトとして数えます。`get_bit`と`set_bit`は各バイト内を右からビットを数えます。例えばビット0は先頭バイトの最下位ビットとなり、ビット15は第二バイトの最上位ビットとなります。

[項9.20](#)における`string_agg`集約関数も参照してください。

[前のページ](#)
文字列関数と演算子

[ホーム](#)
[上に戻る](#)

[次のページ](#)
ビット文字列関数と演算子

9.6. ビット文字列関数と演算子

本節では、ビット文字列の検査と操作を行う関数と演算子について説明します。ビット文字列とは、`bit`型と`bit varying`型の値のことです。通常の比較演算子に加え、[表9-10](#)に示す演算子も使用可能です。`&`、`|`、`#`のビット文字列オペランドは同一長でなければなりません。ビットシフト処理の際、例に示す通り元の長さは保持されます。

表 9-10. ビット文字列演算子

演算子	説明	例	結果
<code> </code>	結合	<code>B'10001' B'011'</code>	<code>10001011</code>
<code>&</code>	ビットのAND	<code>B'10001' & B'01101'</code>	<code>00001</code>
<code> </code>	ビットのOR	<code>B'10001' B'01101'</code>	<code>11101</code>
<code>#</code>	ビットのXOR	<code>B'10001' # B'01101'</code>	<code>11100</code>
<code>~</code>	ビットのNOT	<code>~ B'10001'</code>	<code>01110</code>
<code><<</code>	ビットの左シフト	<code>B'10001' << 3</code>	<code>01000</code>
<code>>></code>	ビットの右シフト	<code>B'10001' >> 2</code>	<code>00100</code>

次の標準SQL関数は文字列同様にビット文字列で動作します。 `length`, `bit_length`, `octet_length`, `position`, `substring`, `overlay`.

次の関数はバイナリ同様にビット文字列で動作します。 `get_bit`, `set_bit`. ビット文字列に処理を行った場合、これらの関数は文字列の最初(一番左)のビットを0として数えます。

さらに、`bit`型から整数値にキャストすることも整数から`bit`型にキャストすることも可能です。以下に例を示します。（訳注：原文は`integral value`となっていますが`integer value`として訳出）

```
44::bit(10)           0000101100
44::bit(3)            100
cast(-44 as bit(12))  111111010100
'1110'::bit(4)::integer 14
```

単に`"bit"`にキャストすることは`bit(1)`にキャストすることを意味することに注意してください。つまり、単に整数の最下位ビットのみがわたされることになります。

注意: PostgreSQL 8.0より前まででは、整数を`bit(n)`にキャストすると、整数の最上位ビットから`n`ビットまでがコピーされました。現在は最下位ビットから`n`ビット分複写されます。また、整数からビット数が元の整数より多いビット文字列にキャストする際、符号ビットでその左部分を拡張します。

9.7. パターンマッチ

PostgreSQLには、パターンマッチを行うに際して3つの異なった手法があります。伝統的なSQLのLIKE演算子、これより新しいSIMILAR TO演算子（SQL:1999で追加されました）、およびPOSIX様式の正規表現です。基本の"この文字列はこのパターンに一致するか?"を別としても、一致した部分文字列を取り出したり置換したり、そして一致部分で文字列を分割する関数が用意されています。

ティップ: 上記の手法では検索できないようなパターンマッチが必要な場合は、PerlもしくはTclでユーザ定義関数を作成することを検討してください。

9.7.1. LIKE

```
string LIKE pattern [ESCAPE escape-character]
string NOT LIKE pattern [ESCAPE escape-character]
```

LIKE式は供給されたpatternにstringが一致すれば真を返します（想像される通り、NOT LIKE式はLIKE式が真を返す場合には偽を返し、その逆もまた同じです。同等の式としてNOT (string LIKE pattern)とも表現できます）。

patternがパーセント記号もしくはアンダースコアを含んでいない場合patternは自身の文字列そのものです。この場合LIKE式は等号演算子のように振舞います。patternの中にあるアンダースコア () は任意の一文字との一致を意味し、パーセント記号 () は0文字以上の並びとの一致を意味します。

例:

```
'abc' LIKE 'abc'      true
'abc' LIKE 'a%'       true
'abc' LIKE ' _b '     true
'abc' LIKE 'c'        false
```

LIKEによるパターン一致は常に文字列全体に対して行われます。従って、文字列内の任意位置における並びと一致させたい場合には、パーセント記号を先頭と末尾に付ける必要があります。

他の文字の一致に使用するのではなく、アンダースコアやパーセント記号そのものを一致させたい場合には、patternの中のそれぞれのアンダースコアとパーセント記号の前にエスケープ文字を付けなければなりません。デフォルトのエスケープ文字はバックスラッシュですが、ESCAPE句で他の文字を指定することができます。エスケープ文字そのものを一致させるにはエスケープ文字を2つ書きます。

注意: もしstandard_conforming_stringsパラメータをoffにしていた場合、リテラル文字列定数に記述するバックスラッシュは2倍必要となります。詳細は[項4.1.2.1](#)を参照してください。

同時にESCAPE ' 'と記述することでエスケープ文字を選択しないことも可能です。これにより、事実上エスケープ機構が働かなくなります。つまり、パターン内のアンダースコアおよびパーセント記号の特別な意味を解除することはできなくなります。

現在のロケールに従って大文字小文字を区別しない一致を行うのであれば、LIKEの代わりにILIKEキーワードを使うことができます。これは標準SQLではなく、PostgreSQLの拡張です。

~~演算子はLIKE式と等価で、~~*はILIKEに対応します。またNOT LIKEおよびNOT ILIKEを表す!~~および!~~*演算子があります。これら全ての演算子はPostgreSQL固有のものです。

9.7.2. SIMILAR TO正規表現

```
string SIMILAR TO pattern [ESCAPE escape-character]
string NOT SIMILAR TO pattern [ESCAPE escape-character]
```

SIMILAR TO演算子は、そのパターンが与えられた文字列に一致するかどうかにより、真もしくは偽を返します。これは、標準SQLの正規表現定義を使用してパターンを解釈するという点以外は、LIKEに類似しています。SQLの正規表現は、LIKE表記と一般的な正規表現の表記とを混ぜ合わせたようなものになっています。

LIKEと同様、SIMILAR TO演算子は、そのパターンが文字列全体に一致した場合のみ処理を行います。これは、パターンが文字列の一部分であっても一致する、一般的な正規表現の動作とは異なっています。また、LIKEと同様、SIMILAR TOでは、%および.を、それぞれ任意の文字列および任意の単一文字を意味するワイルドカード文字として使用します（これらは、POSIX正規表現での.*および.に相当します）。

LIKEから取り入れた上記の機能に加え、SIMILAR TOでは、以下のようにPOSIX正規表現から取り入れたパターンマッチメタ文字もサポートしています。

- `|`は、二者択一（2つの選択肢のうちいずれか）を意味します。
- `*`は、直前の項目の0回以上複数の繰り返しを意味します。
- `+`は、直前の項目の1回以上複数の繰り返しを意味します。
- `?`は、直前の項目の0回もしくは1回の繰り返しを意味します。
- `{m}`は、直前の項目の正確な m 回の繰り返しを意味します。
- `{m,}`は、直前の項目の m 回以上の繰り返しを意味します。
- `{m,n}`は、直前の項目の m 回以上かつ n 回以下の繰り返しを意味します。
- 丸括弧 `()` は、項目を1つの論理項目にグループ化することができます。
- 大括弧式 `[...]` は、POSIX正規表現と同様に文字クラスを指定します。

`SIMILAR TO` ではピリオド `.` はメタ文字ではないことに注意してください。

`LIKE`と同様、バックスラッシュは全てのメタ文字の特殊な意味を無効にします。また、異なるエスケープ文字を`ESCAPE`で指定することが可能です。

例：

```
'abc' SIMILAR TO 'abc'      true
'abc' SIMILAR TO 'a'        false
'abc' SIMILAR TO '%(b|d)%'  true
'abc' SIMILAR TO '(b|c)%'   false
```

3つのパラメータを持つ`substring`関数、`substring(string from pattern for escape-character)`を使用して、SQL正規表現パターンに一致する部分文字列を取り出すことができます。`SIMILAR TO`と同様、指定したパターンがデータ文字列全体に一致する必要があります。一致しない場合、関数は終了し、`NULL`を返します。一致した場合に返されるべきパターンの一部を示すために、エスケープ文字の後に二重引用符（`"`）を繋げたものを2つパターンに含める必要があります。これらの印で括られたパターンの一部に一致するテキストが返されます。

例：`#`を使用して返される文字列を区切ります。

```
substring('foobar' from '%"o b#"' for '#')      oob
substring('foobar' from '%"o_ b#"' for '#')      NULL
```

9.7.3. POSIX正規表現

[表9-11](#)に、POSIX正規表現を使ったパターン一致に使用可能な演算子を列挙します。

表 9-11. 正規表現マッチ演算子

演算子	説明	例
<code>~</code>	正規表現に一致、大文字小文字の区別あり	<code>'thomas' ~ '*.thomas.*'</code>
<code>~*</code>	正規表現に一致、大文字小文字の区別なし	<code>'thomas' ~* '.*Thomas.*'</code>
<code>!~</code>	正規表現に一致しない、大文字小文字の区別あり	<code>'thomas' !~ '.*Thomas.*'</code>
<code>!~*</code>	正規表現に一致しない、大文字小文字の区別なし	<code>'thomas' !~* '.*vadim.*'</code>

POSIX正規表現は、パターンマッチという意味合いでは、`LIKE`および`SIMILAR TO`演算子よりもさらに強力です。`egrep`、`sed`、あるいは`awk`のような多くのUnixツールはここで解説しているのと類似したパターンマッチ言語を使用しています。

正規表現とは文字列の集合（正規集合）の簡略された定義である文字が連なっているものです。ある文字列が正規表現で記述された正規集合の要素になっていれば、その文字列は正規表現に一致していると呼ばれます。`LIKE`と同様、正規表現言語で特殊文字とされているもの以外、パターン文字は文字列と完全に一致されます。とは言っても、正規表現は`LIKE`関数が使用するのとは異なる特殊文字を使用します。`LIKE`関数のパターンと違って正規表現は、明示的に正規表現が文字列の最初または最後からと位置指定されていない限り文字列内のどの位置でも一致を行います。

例：

```
'abc' ~ 'abc'      true
'abc' ~ '^a'        true
'abc' ~ '(b|d)'     true
```

```
'abc' ~ '^(b|c)' false
```

POSIXパターン言語について以下により詳しく説明します。

2つのパラメータを持つ`substring`関数、`substring(string from pattern)`を使用して、POSIX正規表現パターンに一致する部分文字列を取り出すことができます。この関数は、一致するものがない場合にはNULLを返し、ある場合はパターンにマッチしたテキストの一部を返します。しかし、任意の丸括弧を持つパターンの場合、最初の丸括弧内部分正規表現（左丸括弧が最初に来るもの）に一致するテキストの一部が返されます。この例外を起こさずにパターン中に丸括弧を使用したいのであれば、常に正規表現全体を丸括弧で囲むことができます。パターン内の抽出対象の部分文字列より前に丸括弧が必要な場合、後述の捕捉されない丸括弧を参照してください。

例：

```
substring('foobar' from 'o.b')      oob
substring('foobar' from 'o(.)b')    o
```

`regexp_replace`関数は、POSIX正規表現パターンに一致する部分文字列を新規テキストと置換します。構文は、`regexp_replace (source, pattern, replacement [, flags ])`です。`pattern`にマッチしない場合は、`source`文字列がそのまま返されます。一致すると、マッチ部分文字列を`replacement`文字列で置換した`source`文字列が返されます。`replacement`は`%n`を含むことができます。ここで`n`は、`n`番目に丸括弧で括られたパターンの部分表現に一致する元の部分文字列を示す、1から9までの数です。`%&`が含まれた場合、パターン全体と一致する部分文字列が挿入されることを示します。置換テキスト内にリテラルバックスラッシュを挿入する必要がある時は`%%`と記述します。`flags`パラメータは、関数の動作を変更するゼロもしくはそれ以上の1文字フラグを含むオプションのテキスト文字列です。フラグ`i`は大文字小文字を区別しない一致を指定する一方、フラグ`g`は、最初に一致したもののみではなく、それぞれ一致した部分文字列の置換を指定します。その他の有効なフラグは[表9-19](#)に記述されています。

例：

```
regexp_replace('foobarbaz', 'b..', 'X')
                                fooXbaz
regexp_replace('foobarbaz', 'b..', 'X', 'g')
                                fooXX
regexp_replace('foobarbaz', 'b(..)', 'E'X%Y1Y', 'g')
                                fooXarYXazY
```

`regexp_matches`関数はPOSIX正規表現パターンマッチの結果捕捉された全ての部分文字列のテキスト配列を返します。`regexp_matches(string, pattern [, flags ])`の構文になります。この関数は何も行を返さない、1行を返す、複数行を返すといったことができます。下記の`g`フラグを参照して下さい。もし`pattern`に対して一致しない場合、関数は行を返しません。もし、パターンが丸括弧に括られた部分文字列を含まない場合、結果はパターン全体に一致する部分文字列を含む単一要素のテキスト配列となります。もし、パターンが丸括弧に括られた部分文字列を含む場合、関数はパターンの`n`番目に丸括弧で括られた部分文字列に一致する、`n`番目の要素が部分文字列であるテキスト配列を返します（"捕捉されない"丸括弧は数えません）。`flags`パラメータは、関数の動作を変更するゼロもしくは複数の単一文字フラグを含むオプションのテキスト文字列です。フラグ`g`は関数に、最初のマッチだけでなく文字列の中で全てのマッチを検出させ、それら一致の行を返させます。その他有効なフラグは[表9-19](#)に記載されています。

例：

```
SELECT regexp_matches('foobarbequebaz', '(bar) (beque)');
  regexp_matches
-----
{bar,beque}
(1 row)

SELECT regexp_matches('foobarbequebazilbarfbonk', '(b[^b]+) (b[^b]+)', 'g');
  regexp_matches
-----
{bar,beque}
{bazil,barf}
(2 rows)

SELECT regexp_matches('foobarbequebaz', 'barbeque');
  regexp_matches
-----
{barbeque}
(1 row)
```

副問い合わせを使用することで、`regexp_matches()`が常に1行しか返さないように強制することが可能です。これは、`SELECT`対象のリストに対し、マッチするものが無い場合であっても全ての行を返したい場合に特に有効です。

```
SELECT col1, (SELECT regexp_matches(col2, '(bar) (beque)')) FROM tab;
```

`regexp_split_to_table`関数はPOSIX正規表現パターンを区切り文字として使用し、文字列を分割します。`regexp_split_to_table(string, pattern [, flags ])`の構文になります。`pattern`に一致しない場合、関数は`string`を返します。少なくともひとつの一致があれば、それぞれの一致に対して関数は最後のマッチの終わり（あるいは文字列の始め）から最初のマッチまでのテキストを返します。もはやマッチしなくなると最後のマッチの終わりから文字列の最後までテキストを返

します。 `flags` パラメータは、関数の動作を変更するゼロもしくは複数の単一文字フラグを含むオプションのテキスト文字列です。 `regexp_split_to_table` は [表9-19](#) で記載されているフラグをサポートします。

`regexp_split_to_array` 関数は、 `regexp_split_to_array` がその結果を `text` 配列で返すことを除いて、 `regexp_split_to_table` と同じ動作をします。 `regexp_split_to_array(string, pattern [, flags ])` の構文になります。

例：

```
SELECT foo FROM regexp_split_to_table('the quick brown fox jumped over the lazy dog', E'\\Y\\S+') AS foo;
foo
-----
the
quick
brown
fox
jumped
over
the
lazy
dog
(9 rows)

SELECT regexp_split_to_array('the quick brown fox jumped over the lazy dog', E'\\Y\\S+');
      regexp_split_to_array
-----
{the,quick,brown,fox,jumped,over,the,lazy,dog}
(1 row)

SELECT foo FROM regexp_split_to_table('the quick brown fox', E'\\Y\\S*') AS foo;
foo
-----
t
h
e
q
u
i
c
k
b
r
o
w
n
f
o
x
(16 rows)
```

最後の例が明らかにしているように、 `regexp` 分割関数は文字列の最初あるいは終わり、もしくは前のマッチの直後に発生する長さを持たないマッチを無視します。 `regexp_matches` で実装された `regexp` マッチの厳格な定義にこれは相容れませんが、実務上は最も使い勝手の良い動作です。 Perl のような他のソフトウェアシステムも似たような定義を使用します。

9.7.3.1. 正規表現の詳細

PostgreSQL の正規表現は Henry Spencer により書かれたソフトウェアパッケージを使用して実装されています。以下に説明する正規表現の多くの部分は同氏のマニュアルから一字一句複製したものです。

POSIX 1003.2 の定義によると、正規表現 (RE) には2つの形式があるとされます。拡張 RE もしくは ERE (大まかにいって `egrep` に代表されるもの)、および基本 RE もしくは BRE (大まかにいって `ed` に代表されるもの) です。PostgreSQL は両方の形式をサポートし、さらに、POSIX 標準にはないけれども Perl や Tcl などのプログラミング言語で利用できることから広く使用されるようになった、いくつかの拡張もサポートしています。本書では、非 POSIX 拡張を使用した RE を高度な RE もしくは ARE と呼びます。ARE は ERE の正確な上位セットですが、BRE とは複数の記法上の非互換な点があります (さらに非常に多くの制限が課されています)。まず、ARE と ERE 形式について説明し、そして、ARE にのみ適用される機能の注意を、さらに BRE との違いについて説明します。

注意: PostgreSQL は常に、まず正規表現は ARE 規則に従うと推測します。しかし、RE パターンの前に、 [項9.7.3.4](#) に記載されているような埋め込みオプションを追加することにより、より限られた ERE、あるいは BRE 規則を選択することができます。これは、POSIX 1003.2 の規則を正確に期待しているアプリケーションとの互換性に関して有用です。

正規表現は `|` で区切られた、1 つまたは複数のブランチとして定義されます。ブランチのいずれか1つに一致すれば一致したことになります。

ブランチはゼロ個以上の量化アトムもしくは制約の連結です。最初のものにマッチに、次に第 2 番目のものにマッチを、というふうに一致します。なお、空のブランチは空文字列に一致します。

量化アトムとは、単一の量指定子が後ろに付くアトムのことです。量指定子がないと、アトムに一致するものが一致したことになります。量指定子がある場合、アトムとの一致が何回あるかで一致したことになります。アトムは、[表9-12](#)に示したもののいずれかを取ることができます。[表9-13](#)に設定可能な量指定子とその意味を示します。

制約は空文字に、特定の条件に合う場合のみに一致します。アトムを使用できるところには制約を使用することができます。ただしその後に量指定子を付けることはできません。単純な制約を[表9-14](#)に示します。後で他のいくつかの制約を説明します。

表 9-12. 正規表現のアトム

アトム	説明
(<i>re</i>)	(ここで <i>re</i> は任意の正規表現で、) <i>re</i> とのマッチに適合するものです。 マッチは可能である報告用と意味づけられます。
(?: <i>re</i>)	上と同じ。ただし、一致は報告用と意味づけられません。 ("捕捉されない"括弧の集合) (AREのみ)
.	任意の1文字に一致します。
[<i>chars</i>]	ブラケット式。 <i>chars</i> のいずれか1つに一致します (詳細は 項9.7.3.2 を参照してください) 。
¥ <i>k</i>	(ここで <i>k</i> は英数字以外です。) 普通の文字として指定した文字に一致します。例えば、¥¥はバックスラッシュ文字です。
¥ <i>c</i>	ここで <i>c</i> は英数字です (おそらく他の文字が後に続きます) 。 エスケープです。 項9.7.3.3 を参照してください (AREのみ、EREとBREではこれは <i>c</i> に一致します) 。
{	直後に数字以外がある場合、左中括弧{に一致します。 直後に数字が続く場合、 <i>bound</i> (後述) の始まりです。
x	ここで <i>x</i> は他に意味を持たない1文字です。 <i>x</i> に一致します。

REはバックスラッシュ¥を終端とすることはできません。

注意: もし[standard conforming strings](#)パラメータをoffにしていた場合、リテラル文字列定数に記述するバックスラッシュは2倍必要となります。詳細は[項4.1.2.1](#)を参照してください。

表 9-13. 正規表現量指定子

量指定子	マッチ
*	アトムの0個以上複数の並びに一致
+	アトムの1個以上複数の並びに一致
?	アトムの0個または1個の並びに一致
{ <i>m</i> }	アトムの正確に <i>m</i> 個の並びに一致
{ <i>m</i> , }	アトムの <i>m</i> 個以上の並びに一致
{ <i>m</i> , <i>n</i> }	アトムの <i>m</i> 個以上 <i>n</i> 以下の並びに一致。 <i>m</i> は <i>n</i> を超えることはできません。
*?	*の最短マッチを行うバージョン
+?	+の最短マッチを行うバージョン
??	?の最短マッチを行うバージョン
{ <i>m</i> }?	{ <i>m</i> }の最短マッチを行うバージョン
{ <i>m</i> , }?	{ <i>m</i> , }の最短マッチを行うバージョン
{ <i>m</i> , <i>n</i> }?	{ <i>m</i> , <i>n</i> }の最短マッチを行うバージョン

{...}を使用する形式はバウンドとして知られています。バウンド内の*m*と*n*という数は符号なし10進整数であり、0以上255以下の値を取ることができます。

最短マッチを行う量指定子 (AREのみで使用可能) は、対応する通常の (欲張りの) ものと同じものに一致しますが、最大のマッチではなく最小のマッチを取ります。 詳細は[項9.7.3.5](#)を参照してください。

注意: 量指定子の直後に量指定子が続けることはできません。例えば**は無効です。 量指定子から式や副式を始めることはできず、また、^や|の直後に付けることもできません。

表 9-14. 正規表現制約

制約	説明
<code>^</code>	文字列の先頭に一致
<code>\$</code>	文字列の末尾に一致
<code>(?=re)</code>	先行肯定検索は、 <code>re</code> に一致する部分文字列から始まる任意の場所に一致します（AREのみ）。
<code>(?!re)</code>	先行否定検索は、 <code>re</code> に一致しない部分文字列から始まる任意の場所に一致します（AREのみ）。

先行検索制約には後方参照（[項9.7.3.3](#)を参照）を含めることはできません。また、その中の括弧は全て取り込むものではないとみなされます。

9.7.3.2. ブラケット式

ブラケット式とは、`[]`内の文字のリストです。通常これはそのリスト内の任意の1文字に一致します（しかし、以降を参照してください）。リストが`^`から始まる場合、そのリストの残리にはない任意の1文字に一致します。リスト内の2文字が`-`で区切られていた場合、これは2つ（を含む）の間にある文字範囲全体を表す省略形となります。例えば、ASCIIにおける`[0-9]`は全ての数字に一致します。例えば`a-c-e`といった、終端を共有する2つの範囲は不正です。範囲は並びの照合順に非常に依存しています。ですので、移植予定のプログラムではこれに依存してはなりません。

このリストに`]`そのものを含めるには、それを先頭文字（もしそれが使用されれば`^`の後）にしてください。`-`そのものを含めるには、それを先頭もしくは末尾の文字とするか、範囲の2番目の終端としてください。`-`を範囲の最初の終端で使用するには、`[.と.]`でそれを囲み、照合要素（後述）にしてください。これら文字と、`[`（次段落を参照）のなんらかの組み合わせ、およびエスケープ（AREのみ）を例外として、他の全ての特殊文字はブラケット式内では特殊な意味を持ちません。特に、`¥`はEREとBRE規則に従う場合は特別でなくなります。しかし、AREでは（エスケープの始まりとして）特別な意味を持ちます。

ブラケット式内に、照合要素（文字、単一文字であるかのように照合する複数文字の並び、もしくはそれぞれの照合並びの名前）が`[.と.]`の間にあると、その照合要素の文字の並びを意味します。この並びはブラケット式のリストの一要素として取り扱われます。このことにより、ブラケット式は要素を照合する複数文字を含むブラケット式を1文字以上に一致させることができます。例えば、照合並びが`ch`照合要素を含む場合、正規表現`[[:ch:]]*c`は`chchcc`という文字の最初の5文字に一致します。

注意: 今のところ、PostgreSQLは複数文字照合要素をサポートしません。この情報は将来の振舞いの可能性を説明したものです。

ブラケット式内の`[=:と=]`の間に照合要素は同値クラスです。全ての照合要素の文字の並びが自身を含むものと等価であることを示します（他に等価な照合要素がある場合、`[.と.]`で囲まれたかのように扱われます）。例えば、`[[:o=]]`、`[[:^=]]`および`[[:^]]`が全て同意語であれば、`o`と`^`は同値クラスのメンバです。同値クラスは範囲の終端にはなりません。

ブラケット式内では、`[.:と:]`の間にある文字クラスの名称は、そのクラスに属する全ての文字のリストを意味します。標準文字クラス名は、`alnum`、`alpha`、`blank`、`cntrl`、`digit`、`graph`、`lower`、`print`、`punct`、`space`、`upper`、`xdigit`です。これらはctypeで定義された文字クラスを意味します。ロケールは別のものを提供可能です。文字クラスは範囲の終端では使用することができません。

ブラケット式には2つの特殊な場合があります。`[[:<:]]`と`[[:>:]]`というブラケット式は、先頭と終端の単語がそれぞれ空文字であることに一致する制約です。単語は、単語文字が前後に付かない単語文字の並びとして定義されます。単語文字とは1つの`alnum`文字です（ctypeで定義されています）。これは、POSIX 1003.2との互換性がありますが、そこでは定義されていない式です。ですので、他システムへ移植予定のソフトウェアでの使用には注意が必要です。通常後述の制約エスケープの方がよく使われます。これはもはや標準ではありませんが、入力しやすいものです。

9.7.3.3. 正規表現エスケープ

エスケープとは、`¥`から始まり英数字がその後に続く特殊な並びです。エスケープには、文字エントリ、クラス省略、制約エスケープ、後方参照といった様々な変種があります。`¥`の後に英数字が続くけれども、有効なエスケープを構成しない並びはAREでは不正です。EREにはエスケープはありません。ブラケット式の外側では、`¥`の後に英数字が続く並びは単に普通の文字としてその文字を意味します。ブラケット式の内側では、`¥`は普通の文字です（この文字はEREとARE間の非互換性の1つです）。

文字エントリエスケープは非印字文字やRE内でその他の不便な文字の指定を簡略化するために存在します。これらを[表9-15](#)に示します

クラス省略エスケープは、あるよく使用される文字クラスの省略形を提供します。これらを[表9-16](#)に示します。

制約エスケープは、指定した条件に合う場合に空文字に一致する制約をエスケープとして表したものです。これらを[表9-17](#)に示します。

後方参照（`¥n`）は、直前に括弧で囲まれた副式によって一致された、`n`番目の同一文字列に一致します（[表9-18](#)を参照してください）。例えば、`([bc])¥1`は`bb`もしくは`cc`に一致しますが、`bc`や`cb`には一致しません。REでは副式全体は後方参照の前になけ

ればなりません。副式は開括弧の順番で番号付けされます。取り込まない括弧は副式を定義しません。

注意: エスケープの先頭の¥をSQL文字定数としてパターンに入力する時には二重にしなければならないことを忘れないでください。以下に例を示します。

```
'123' ~ E'^¥¥d{3}' true
```

表 9-15. 正規表現文字エントリエスケープ

エスケープ	説明
¥a	C言語と同じ警報（ベル）文字
¥b	C言語と同じバックスペース
¥B	バックスラッシュの必要な二重化回数を減らすためのバックスラッシュ（¥）の同義語
¥cX	（ここでxは任意の文字で）その下位5ビットがxと同一、その他のビットが0となる文字
¥e	照合順名がESCとなる文字、それに失敗したら、033という8進数値を持つ文字。
¥f	C言語と同じ改ページ
¥n	C言語と同じ改行
¥r	C言語と同じ復帰
¥t	C言語と同じ水平タブ
¥uwxxyz	（ここでwxyzは正確に4桁の16進数で）使用マシンのバイト順序付けで表した、U+wxxyz というUTF16（16ビット、Unicode）文字
¥Ustuvwxyz	（ここでstuvwxyzは正確に8桁の16進数です。）32ビットにUnicodeが拡張されるという仮説のために確保
¥v	C言語と同じ垂直タブ
¥xhhh	（ここでhhhは任意の16進数の並びで）その文字の16進数値が0xhhhとなる文字（使用される16進数の桁数にかかわらず単一の文字）
¥0	その値が0（NULLバイト）となる文字
¥xy	（ここでxyは正確に2桁の8進数で、後方参照ではない）その値が0xyとなる文字
¥xyz	（ここでxyzは正確に3桁の8進数で、後方参照ではない）その値が0xyzとなる文字

16進数の桁とは0-9、a-f、A-Fです。8進数の桁とは0-7です。

この文字エントリエスケープは常に普通の文字と解釈されます。例えば、¥135はASCIIの]となり、¥135はブラケット式の終端にはなりません。

表 9-16. 正規表現クラス省略エスケープ

エスケープ	説明
¥d	[[[:digit:]]
¥s	[[[:space:]]
¥w	[[[:alnum:]]_]（アンダースコアが含まれることに注意）
¥D	[^[:digit:]]
¥S	[^[:space:]]
¥W	[^[:alnum:]]_]（アンダースコアが含まれることに注意）

ブラケット式内では、¥d、¥s、および¥wはその外側の大括弧を失い、¥D、¥Sおよび¥Wは不正です（ですから、例えば[a-c¥d]は[a-c[:digit:]]と同じになります。また、[a-c¥D]は[a-c^[:digit:]]と同じになり、不正です）。

表 9-17. 正規表現制約エスケープ

エスケープ	説明
¥A	文字列の先頭にのみ一致します（^との違いについては 項9.7.3.5 を参照してください）。
¥m	単語の先頭にのみ一致します。
¥M	単語の末尾にのみ一致します。
¥Y	単語の先頭もしくは末尾にのみ一致します。

¥Y	単語の先頭もしくは末尾以外の場所にのみ一致します。
¥Z	文字列の末尾にのみ一致します（\$との違いについては 項9.7.3.5 を参照してください）。

単語は前述の`[[[:<:]]`と`[[[:>:]]`の規定通りに定義されます。ブラケット式内では制約エスケープは不正です。

表 9-18. 正規表現後方参照

エスケープ	説明
¥m	(ここでmは非ゼロの数です。) 副式のm番目への後方参照
¥mnn	(ここでmは非ゼロの数です。nnでさらに桁を指定します。mnn10進数値は取り込み括弧の数よりも多くてはなりません。) 副式のmnn番目への後方参照

注意: 8進数の文字エントリエスケープと後方参照の間には曖昧性があります。上でヒントとして示したようにこれは以下の発見的手法で解決されます。先頭の0は常に8進数エスケープを示します。その後に数字が続かない単一の非ゼロ数字は常に後方参照として解釈されます。ゼロから始まらない複数数字の並びは、適切な副式の後にあれば（つまり、その番号が後方参照用の範囲内にあれば）後方参照として解釈されます。さもなくば、8進数として解釈されます。

9.7.3.4. 正規表現メタ構文

上述の主構文の他に、特殊な形式や雑多な構文的な機能が使用可能です。

REは、2つの特殊な決定子前置詞のどちらかから始まります。REが`***:`から始まるものであれば、REの残りはAREと解釈されます。(PostgreSQLはREをAREとして推測するため、通常は影響を受けません。ただし、正規表現関数に対して`flags`パラメータを指定されたEREやBREモードでは影響を受けます。) REが`***=`から始まるものであれば、REの残りは、全ての文字を普通の文字とみなしたリテラル文字列と解釈されます。

AREは埋め込みオプションから始められます。(?`xyz`)という並びで残りのREに影響するオプションを指定します(ここで`xyz`は1つ以上の英字です)。このオプションは、事前に決定されたオプションを上書きします。— 特に、正規表現演算子、もしくは正規表現関数に与えられた`flags`パラメータにより示される大文字小文字の区別を上書きします。使用可能なオプション文字を[表9-19](#)に示します。これらの同じオプション文字が、正規表現関数の`flags`パラメータで使用されることに注意して下さい。

表 9-19. ARE埋め込みオプション文字

オプション	説明
b	残りのREはBRE
c	大文字小文字を区別する一致（演算子で規定される大文字小文字の区別よりこの指定が優先されます）。
e	残りのREはERE
i	大文字小文字を区別しない一致（ 項9.7.3.5 を参照）（演算子で規定される大文字小文字の区別よりこの指定が優先されます）。
m	nの歴史的な同義語
n	改行を区別する一致（ 項9.7.3.5 を参照）
p	部分的な改行を区別する一致（ 項9.7.3.5 を参照）
q	残りのREはリテラル（"引用符付けされた"）文字列、全て普通の文字
s	改行を区別しないマッチ（デフォルト）
t	厳しめの構文（デフォルト、後述）
w	部分的な改行区別の逆（"ワイアード"）マッチ（ 項9.7.3.5 を参照）
x	拡張構文（後述）

埋め込みオプションはその並びの終端)で有効になります。AREの先頭（もし`***:`決定子があればその後）でのみ利用可能です。

全ての文字が意味を持つ、通常の（厳しめの）RE構文に加え、`x`埋め込みオプションを指定することで利用できる拡張構文があります。拡張構文では、RE内の空白文字は無視され、`#`とその後の改行（もしくはREの終端）の間の全ての文字も同様です。これにより、段落付けや複雑なREのコメント付けが可能になります。基本規則に対して3つの例外があります。

- 直前に`¥`が付いた空白文字もしくは`#`は保持されます。
- ブラケット式内の空白文字もしくは`#`は保持されます。
- `(?:`などの複数文字シンボルでは、空白文字とコメントは不正です。

ここでの空白文字とは、空白、タブ、改行、スペース文字クラスに属する文字です。

最後に、AREのブラケット式の外側では、`(?#ttt)`という並びはコメントになります（ここで`ttt`は）を含まない任意のテキストです）。繰り返しになりますが、これは`(?:`などの複数文字シンボルの文字間では使用できません。こうしたコメントは実用性というより歴史的所産です。そのため、この使用は勧めません。代わりに拡張構文を使用してください。

初めに`***=`決定子が指定され、ユーザの入力がREではなくリテラルとして扱われる場合、これらのメタ構文拡張は使用できません。

9.7.3.5. 正規表現マッチ規則

REが文字列の中の1つ以上の部分文字列と一致する場合において、REは最初にマッチが始まった部分文字列と一致します。その位置からまた1つ以上の部分文字列とマッチした際は、正規表現は最短マッチを行わない（欲張り型）か最短マッチを行う（非欲張り型）かによって、最長一致もしくは最短一致の文字列のどちらかに一致します

REが最長マッチかどうかは以下の規則によって決まります。

- ほとんどのアトムおよび全ての式は欲張り属性を持ちません（これらは変動する量のテキストにまったく一致しないからです）。
- REを括弧で括ることは欲張りかどうかを変更しません。
- `{m}`もしくは`{m}?`といった固定繰り返し数の量指定子を持つ量指定付きアトムは、アトム自身と同一の欲張りさを持ちます（まったく持たない可能性もあります）。
- 他の通常の量指定子（`{m,n}`、*m*と*n*が等しい場合も含みます）を持つ量指定付きアトムは欲張り型です（最長マッチを使用します）。
- 他の非欲張り型量指定子（`{m,n}?`、*m*と*n*が等しい場合も含みます）を持つ量指定付きアトムは非欲張り型です（最短マッチを使用します）。
- 最上位レベルの`|`演算子を持たないREであるブランチは、最初の欲張り属性を持つ量指定付きアトムと同一の欲張り属性を持ちます。
- `|`演算子で接続された2つ以上のブランチからなるREは常に欲張り型です。

上の規則は、個々の量指定付きアトムだけではなく、量指定付きアトムを複数含むブランチやRE全体の欲張り属性に関連します。つまり、ブランチやRE全体が全体として最長または最短の部分文字列に一致するという方法でマッチ処理が行われます。全体のマッチの長さが決まると、特定の部分式に一致する部分とその部分式の欲張り属性によって決まります。この時、RE内でより前にある部分式が後にある部分式よりも高い優先度を持ちます。

この意味の例を示します。

```
SELECT SUBSTRING('XY1234Z', 'Y*([0-9]{1,3})');
Result: 123
SELECT SUBSTRING('XY1234Z', 'Y*?([0-9]{1,3})');
Result: 1
```

最初の例では、`Y*`が欲張り型であるため、REは全体として欲張り型です。マッチは`Y`の位置から始まり、そこから可能な限り最長の文字列に一致します。つまり`Y123`となります。出力は括弧で括られた部分、つまり`123`となります。2番目の例では、`Y*?`が非欲張り型のため、REは全体として非欲張り型です。マッチは`Y`の位置から始まり、そこから可能な限り最短の文字列に一致します。つまり`Y1`となります。部分式`[0-9]{1,3}`は欲張り型ですが、決定された一致する全体の長さを変更することはできません。したがって、強制的に`1`に一致することになります。

まとめると、REが欲張り型部分式と非欲張り型部分式の両方を持つ場合、全体のマッチ長はRE全体に割り当てられる属性に応じて、最長マッチ長か最短マッチ長のどちらかになります。部分式に割り当てられた属性は、部分式の中でどれだけの量をその部分式の中で"消費"できるかのみに影響します。

`{1,1}`および`{1,1}?`量指定子を副式もしくはRE全体に使用して、それぞれ、欲張りか欲張りでないかを強制することが可能です。

マッチの長さは照合要素ではなく文字列で測られます。空文字列はまったく一致する要素がない文字列よりも長いと考えられます。例えば、`bb*`は`abbbc`の真中の3文字と一致し、`(week|wee)(night|knights)`は`weeknights`の全ての10文字と一致し、`abcl`に対

して `(.*)` が一致されると、括弧内の部分正規表現は3つの文字全てに一致し、`bc` に対して `(a*)` が一致されると、全体のREと括弧内の正規表現は空文字列に一致します。

もし大文字小文字を区別しない一致が指定されると、アルファベット文字の大文字小文字の区別がまったくなくなったのと同じ効果を与えます。ブラケット式の外側にアルファベットの大文字小文字が混ざった通常の文字が出てきた場合、例えば、`x` が `[xX]` となるように大文字小文字ともにブラケット式に実質的に転換されます。ブラケット式の中に現れた時は、（例えば）`[x]` が `[xX]` となり、また `[^x]` が `[^xX]` となるように、全ての大文字小文字それぞれの対がブラケット式に追加されます。

改行を区別するマッチが指定されると、`.` と `^` を使用するブラケット式は（REが明示的に調整されていたとしてもマッチが改行をまたがらないようにするために）改行文字に一致しなくなります。また、`^` と `$` はそれぞれ改行直後と直前の空文字列に一致し、さらに、それぞれ文字列の先頭と末尾に一致します。しかし、AREエスケープの `\A` と `\Z` は、継続して、文字列の先頭と末尾のみに一致します。

部分的に改行を区別するマッチが指定されると、`.` とブラケット式は改行を区別する一致を行うようになりますが、`^` と `$` は変更されません。

部分的に改行を区別する逆マッチが指定されると、`^` と `$` は改行を区別する一致を行うようになりますが、`.` とブラケット式は変更されません。これはあまり有用ではありません。対称性のために提供されています。

9.7.3.6. 制限と互換性

本実装ではREの長さに関する制限はありません。しかし、移植性を高めたいプログラムでは、256バイトを超えるREを使用すべきではありません。POSIX互換の実装ではそうしたREでは混乱する可能性があります。

AREの機能のうち、POSIX EREと実質的な非互換性があるのは、`\` がブラケット式の内側で特殊な意味を失わないという点のみです。他の全てのARE機能は、POSIX EREでは不正、未定義、未指定な効果となる構文を使用しています。決定子の `***` 構文などはBREおよびEREのPOSIX構文にはありません。

多くのARE式はPerlから拝借したものです。しかし、いくつかは整理され、Perlの拡張のいくつかは存在しません。注意すべき非互換性には、`\b`、`\B`、改行の取り扱いに関する特殊な措置の欠落、改行を区別する一致に影響する点について補足したブラケット式の追加、括弧と先行検索制約内の後方参照についての制限、最長/最短（最初に一致するではなく）マッチのセマンティックがあります。

PostgreSQLリリース7.4より前で認知された、AREとERE構文間で大きな非互換が2つあります。

- AREでは、`\` の後に英数字が続くものはエスケープもしくはエラーとなります。以前のリリースでは、これは単に、英数字を記述する他の方法でした。これは、大きな問題にはならないはずですが。以前のリリースではこうした並びを記述する理由がないからです。
- AREでは、`\` は `[]` 内でも特別な文字です。したがって、ブラケット式では `\` を `\\` と記述しなければなりません。

9.7.3.7. 基本正規表現

BREはEREといくつかの面において異なります。BREにおいては、`|`、`+`、`?` は普通の文字であり、それらの機能と等価なものはありません。バウンドの区切りは `\{` と `\}` であり、`{` と `}` 自身は普通の文字です。副式を入れ子にするための括弧は `\(` と `\)` であり、`(` と `)` 自身は普通の文字です。`^` は、REの先頭にある場合や括弧内の副式の先頭の場合を除き、普通の文字です。`$` は、REの末尾にある場合や括弧内の副式の末尾の場合を除き、普通の文字です。また、`*` はREの先頭にある場合や括弧内の副式の先頭にある場合には普通の文字になります（その前に `^` が付いている可能性もあります）。最後に、1桁の後方参照を使用することができ、また、BREにおいては、`\<` と `\>` はそれぞれ `[:<:]` と `[[:>:]]` と同義です。

[前のページ](#)

ビット文字列関数と演算子

[ホーム](#)
[上に戻る](#)

[次のページ](#)

データ型書式設定関数

9.8. データ型書式設定関数

PostgreSQLの書式設定関数は多彩なデータ型（日付/時刻データ型、整数データ型、浮動小数点数データ型、数値データ型）を整形された文字列に変換したり、整形された文字列を特定のデータ型に変換する強力なツールの一式を提供しています。[表9-20](#)にこれらを列挙しています。これら関数は共通の呼び出し規約を踏襲しています。最初の引数は整形される値で2番目の引数は入力書式または出力書式を定義するテンプレートです。

単一引数の`to_timestamp`関数も利用可能です。これは`double precision`型の引数を取り、Unix時刻（1970-01-01 00:00:00+00からの秒数）から`timestamp with time zone`に変換します。（`Integer`型のUnix時刻は暗黙的に`double precision`にキャストされます。）

表 9-20. 書式設定関数

関数	戻り値	説明	例
<code>to_char(timestamp, text)</code>	<code>text</code>	タイムスタンプを文字列に変換	<code>to_char(current_timestamp, 'HH12:MI:SS')</code>
<code>to_char(interval, text)</code>	<code>text</code>	時間間隔を文字列に変換	<code>to_char(interval '15h 2m 12s', 'HH24:MI:SS')</code>
<code>to_char(int, text)</code>	<code>text</code>	整数を文字列に変換	<code>to_char(125, '999')</code>
<code>to_char(double precision, text)</code>	<code>text</code>	実数、倍精度数を文字列に変換	<code>to_char(125.8::real, '999D9')</code>
<code>to_char(numeric, text)</code>	<code>text</code>	数値を文字列に変換	<code>to_char(-125.8, '999D99S')</code>
<code>to_date(text, text)</code>	<code>date</code>	文字列を日付に変換	<code>to_date('05 Dec 2000', 'DD Mon YYYY')</code>
<code>to_number(text, text)</code>	<code>numeric</code>	文字列を数値に変換します	<code>to_number('12,454.8-', '99G999D9S')</code>
<code>to_timestamp(text, text)</code>	<code>timestamp with time zone</code>	文字列をタイムスタンプに変換	<code>to_timestamp('05 Dec 2000', 'DD Mon YYYY')</code>
<code>to_timestamp(double precision)</code>	<code>timestamp with time zone</code>	Unix時刻をタイムスタンプに変換	<code>to_timestamp(1284352323)</code>

（`to_char`用）出力テンプレート文字列には、値に基づいて認識され、適切に整形されたデータで置き換えられるパターンがあります。テンプレートパターンではない全てのテキストは単にそのままコピーされます。同様に、（その他の関数に対し）入力テンプレート文字列では、テンプレートパターンは入力されたデータ文字列で供給される値を特定します。

[表9-21](#)に、日付/時刻型の値の書式に使用可能なテンプレートパターンを示します。

表 9-21. 日付/時刻型の書式テンプレートパターン

パターン	説明
HH	時 (01～12)
HH12	時 (01～12)
HH24	時 (00～23)
MI	分 (00～59)
SS	秒 (00～59)
MS	ミリ秒 (000～999)
US	マイクロ秒 (000000～999999)
SSSS	深夜0時からの秒数 (0～86399)
AM、am、 PM、またはpm	午前/午後の指定（ピリオドなし）
A.M.、a.m.、 PM、またはp.m.	午前/午後の指定（ピリオド付き）
Y,YYY	句読点（コンマ）付き年（4桁以上）
YYYY	年（4桁以上）
YYY	年の下3桁

YY	年の下2桁
Y	年の下1桁
YYYY	ISO年（4以上の桁）
YYY	ISO年の下3桁
IY	ISO年の下2桁
I	ISO年の下1桁
BC、bc、 AD、またはad	紀元前後の指定（ピリオドなし）
B.C.、b.c.、 A.D.、またはa.d.	紀元前後の指定（ピリオド付き）
MONTH	大文字での完全な月名（9文字になるように空白でパッド）
Month	大文字で書き始める完全な月名（9文字になるように空白でパッド）
month	小文字での完全な月名（9文字になるように空白でパッド）
MON	大文字での短縮形の月名（英語では3文字、現地語化された場合は可変長）
Mon	大文字で書き始める短縮形の月名（英語では3文字。現地語化された場合は可変長）
mon	小文字での短縮形の月名（英語では3文字。現地語化された場合は可変長）
MM	月番号（01～12）
DAY	大文字での完全な曜日名（9文字になるように空白でパッド）
Day	大文字で書き始める完全な曜日名（9文字になるように空白でパッド）
day	小文字での完全な曜日名（9文字になるように空白でパッド）
DY	短縮形の大文字での短縮形の曜日名（英語では3文字。現地語化された場合は可変長）
Dy	大文字で書き始める短縮形の曜日名（英語では3文字。現地語化された場合は可変長）
dy	小文字での短縮形の曜日名（英語では3文字。現地語化された場合は可変長）
DDD	通年の日にち番号（001～366）
IDDD	ISO通年の日にち番号（001-371：通年 第1日は最初のISO週の月曜日）
DD	ひと月通算の日にち番号（01～31）
D	1週通算の曜日番号、日曜日（1）から土曜日（7）まで
ID	ISO1週通算の曜日番号、月曜日（1）から日曜日（7）まで
W	月中の週番号（1～5）（その月の初日がある週が第1週）
WW	年間を通じた週番号（1～53）（元旦のある週が第1週）
IW	年間を通じたISO週番号（01 - 53：新年の最初の木曜日がある週が第1週）
CC	世紀（2桁。21世紀は2001-01-01から開始）
J	ユリウス日（紀元前4714年11月24日午前零時からの通算経過日）
Q	クォータ（四半期：to_dateとto_timestampにより無視されます）
RM	大文字ローマ数字による月（I～XII：I=1月）
rm	小文字ローマ数字による月（i～xii：i=1月）
TZ	大文字による時間帯名
tz	小文字による時間帯名

修飾子はどのようなテンプレートパターンに対しても、その振舞いを変更するために適用することができます。例えば、FMMonthはFM修飾子の付いたMonthパターンです。[表9-22](#)に、日付/時刻書式の修飾子パターンを示します。

表 9-22. 日付/時刻書式用のテンプレートパターン修飾子

修飾子	説明	例
FM接頭辞	字詰めモード（空白のパディング、および末尾の0を無効）	FMMonth
TH接尾辞	大文字の序数を追加	DDTH、例えば12TH
th接尾辞	小文字の序数を追加	DDth、例えば12th
FX接頭辞	固定書式のグローバルオプション（使用上の注意事項を参照）	FX Month DD Day
TM接頭辞	翻訳モード（ lc_time ）に基づき、現地語化された日付、月名を表示します。	TMMonth

日付/時刻型書式の使用上の注意事項は次のとおりです。

- FMはパターンの出力を固定長にするため、先頭にはゼロ、末尾には空白を追加してしまう機能を無効にします。PostgreSQLでは、FMはその次に記述されたものだけを変更します。一方Oracleでは、FMはそれに続く全ての記述に対して影響し、FM修飾詞の切り替えに応じて繰り返されます。
 - TMには末尾の空白は含まれません。
 - FXオプションがテンプレートが使用されている場合を除き、to_timestampとto_dateは入力文字列の複数の空白スペースを無視します。例えば、to_timestampにはたった1つのスペースがあることになっているので、to_timestamp('2000 JUN', 'YYYY MON')は動作しますが、to_timestamp('2000 JUN', 'FXYYYY MON')はエラーを返します。FXはテンプレートの第1項目として指定される必要があります。
 - to_charテンプレートでは、通常のテキストが許され、そのまま出力されます。部分文字列を二重引用符で括ることで、部分文字列にパターン用のキーワードがあったとしても、強制的にリテラルテキストとして解釈させることができます。例えば、'Hello Year "YYYY"'ではYYYYは年データに置換されてしまいますが、Year内のYは置換されません。to_date、to_number、to_timestampでは、二重引用符で括られた文字の数だけ入力された文字をスキップします。例えば"xx"が指定された場合は2文字がスキップされます。
 - 出力に二重引用符を付けたい場合、E'¥¥"YYYY Month¥¥"'のようにその前にバックスラッシュを付けなければなりません。
 - もし、YYYの様に4桁未満の年書式が指定され、かつ与えられる年が4桁未満だった場合、年は2020年に最も近くなるよう調整されます。例えば、95の場合は1995年になります。
 - 文字列をtimestamp型もしくはdate型にするYYYY変換は、4桁以上の年数値を処理するとき制限が加えられます。このような場合、数字以外の文字またはYYYYの後にテンプレートを使わなければなりません。 そうしないと年は常に4桁と解釈されます。例えば(20000年として)、to_date('200001131', 'YYYYMMDD')は4桁の年と解釈されるので、to_date('20000-1131', 'YYYY-MMDD')またはto_date('20000Nov31', 'YYYYMonDD')のように数字でない区切り符号の使用をお勧めします。
 - 文字列からtimestampもしくはdateへの変換において、YYY、YYYY、もしくはY,YYYフィールドが存在するとCC(世紀)フィールドは無視されます。CCがYYもしくはYと共に使用されると、年は(CC-1)*100+YYのように計算されます。
 - ISO週番号と週の曜日番号(グレゴリオ暦の日付とは異なって)はto_timestampとto_dateの2つの方法のうちのひとつで指定できます。
 - 年、通年の週番号、週の曜日番号。例えば、to_date('2006-42-4', 'IYYY-IW-ID')は、日付2006-10-19を返します。曜日番号を省略した場合、1(月曜日)と想定されます。
 - 年と通年の日付番号。例えば、to_date('2006-291', 'IYYY-IDDD')のは、同様2006-10-19を返します。
- ISO週番号とグレゴリアン暦日のフィールドを混在して使用して日付を構築する試みは無意味なことで、エラーの原因になります。ISO年の文脈によると、"月"、あるいは"月中日付番号"は意味を持ちません。グレゴリオ暦の年の文脈では、ISO週番号は意味を持ちません。グレゴリアンとISOの日付の仕様の混在を避けなければなりません。
- 文字列型からtimestamp型への変換に際し、ミリ秒_{MS}またはマイクロ秒_{US}の値は小数点の位置の後の秒の桁として使用されます。例えば、to_timestamp('12:3', 'SS:MS')は3ミリ秒ではなく300ミリ秒です。なぜなら変換においてこれは12 + 0.3と計算されるからです。ということは、SS:MS書式に対して入力値である12:3、12:30、および12:300は同じミリ秒数を指定します。3ミリ秒数が必要な場合には12:003のようにしなければなりません。この時、変換において12 + 0.003 = 12.003秒と計算します。
- もう少し複雑な例を挙げます。 to_timestamp('15:12:02.020.001230', 'HH:MI:SS.MS.US') は15時間12分と2秒+20ミリ秒+1230マイクロ秒 = 2.021230秒です。
- to_char(..., 'ID')の曜日番号付けはextract(isodow from ...)関数に一致しますが、to_char(..., 'D')の曜日番号付けはextract(dow from ...)の曜日番号付けに一致しません。
 - to_char(interval)関数は、HHとHH12により、例えば0時と36時を12とするような、12時間表示の出力をします。一方HH24はインターバル間で23を超えることが可能な24時間表示の出力をします。

表9-23に、数値型の値の書式設定に使用可能なテンプレートパターンを示します。

表 9-23. 数値書式用のテンプレートパターン

パターン	説明
9	指定された桁数での値
0	前にゼロが付いた値
. (ピリオド)	小数点
, (コンマ)	千単位で区切る符号
PR	負の値の角括弧表示
S	(ロケール使用の) 符号付き値
L	(ロケール使用の) 通貨記号
D	(ロケール使用の) 小数点
G	(ロケール使用の) グループ区切り文字
MI	(数値 < 0であれば) 指定位置にマイナス記号
PL	(数値 > 0であれば) 指定位置にプラス記号
SG	指定された位置にプラス/マイナス記号
RN	(1~3999の入力値による) ローマ数字
THまたはth	序数接尾辞
V	<i>n</i> 桁シフト (注意事項を参照)
EEEE	科学技術表記法用の指数

数値型書式の使用上の注意事項は次のとおりです。

- SG、PL、またはMIで整形された符号は、数値と関連付けられません。例えば、`to_char(-12, 'MI9999')`は' -12'となる一方、`to_char(-12, 'S9999')`は' -12'となります。Oracleの実装では9の前のMIが置かれてはならず、9の後にMIが置かれることを要求しています。
- 9は9が並んでいる数と同じ桁数の値を出力します。桁が使用可能でない場合、スペースを出力します。
- THはゼロ未満の値と小数は変換しません
- PL、SG、およびTHはPostgreSQLの拡張です。
- vは入力値を実質的に10^n乗します。ここでnはvに続く桁数です。to_char関数は小数点とvとの混在をサポートしません (例えば、99.9v99 は許可されません)。
- EEEE(科学技術表記)は、他の書式パターンや桁と小数点のパターンを除く修飾子との組み合わせで使うことはできず、また必ず書式文字列の最後に位置しなければなりません。(例えば、9.99EEEEは正しい表記となります。)

ある修飾子をその動作を変えるために、任意のテンプレートに適用することができます。例えば、FM9999はFM修飾子が付いた9999パターンです。表9-24に、数値の書式用の修飾子パターンを示します。

表 9-24. 数値の書式用テンプレートパターン修飾子

修飾子	説明	例
FM添え字	字詰めモード (空白パディングの無効、および0の除去)	FM9999
TH添え字	大文字による序数添え字	999TH
th添え字	小文字による序数添え字	999th

表9-25に、to_char関数を使用した例をいくつか示します。

表 9-25. to_charの例

Expression	Result
<code>to_char(current_timestamp, 'Day, DD HH12:MI:SS')</code>	'Tuesday , 06 05:39:18'
<code>to_char(current_timestamp, 'FMDay, FMDD HH12:MI:SS')</code>	'Tuesday, 6 05:39:18'
<code>to_char(-0.1, '99.99')</code>	' -.10 '
<code>to_char(-0.1, 'FM9.99')</code>	'-.1 '
<code>to_char(0.1, '0.9')</code>	' 0.1 '
<code>to_char(12, '9990999.9')</code>	' 0012.0 '

to_char(12, 'FM9990999.9')	'0012.'
to_char(485, '999')	' 485'
to_char(-485, '999')	'-485'
to_char(485, '9 9 9')	' 4 8 5'
to_char(1485, '9,999')	' 1,485'
to_char(1485, '9G999')	' 1 485'
to_char(148.5, '999.999')	' 148.500'
to_char(148.5, 'FM999.999')	'148.5'
to_char(148.5, 'FM999.990')	'148.500'
to_char(148.5, '999D999')	' 148,500'
to_char(3148.5, '9G999D999')	' 3 148,500'
to_char(-485, '999S')	'485-'
to_char(-485, '999MI')	'485-'
to_char(485, '999MI')	'485 '
to_char(485, 'FM999MI')	'485'
to_char(485, 'PL999')	'+485'
to_char(485, 'SG999')	'+485'
to_char(-485, 'SG999')	'-485'
to_char(-485, '9SG99')	'4-85'
to_char(-485, '999PR')	'<485>'
to_char(485, 'L999')	'DM 485'
to_char(485, 'RN')	' CDLXXXV'
to_char(485, 'FMRN')	'CDLXXXV'
to_char(5.2, 'FMRN')	'v'
to_char(482, '999th')	' 482nd'
to_char(485, '"Good number:"999')	'Good number: 485'
to_char(485.8, '"Pre:"999" Post:" .999')	'Pre: 485 Post: .800'
to_char(12, '99V999')	' 12000'
to_char(12.4, '99V999')	' 12400'
to_char(12.45, '99V9')	' 125'
to_char(0.0004859, '9.99EEEE')	' 4.86e-04'

9.9. 日付/時刻関数と演算子

[表9-27](#)は、日付/時刻型の値の処理で使用可能な関数を示しています。詳細は、以下の副節で説明します。[表9-26](#)は、(+、*等の) 基本的な算術演算子の振舞いを説明しています。書式設定関数については[項9.8](#)を参照してください。[項8.5](#)を参照して、日付/時刻データ型についての背景となっている情報に精通していなければなりません。

後述のtimeもしくはtimestamp型の入力を受け取る関数および演算子は全て、実際には2つの種類があります。1つはtime with time zone型またはtimestamp with time zone型を取るもので、もう1つはtime without time zone型もしくはtimestamp without time zone型を取るものです。省略のため、これらの種類の違いは個別に示していません。また、+と*演算子は交代演算子を持ちます (例えばdate + integerとinteger + date)。こうした組み合わせは片方のみ示します。

表 9-26. 日付/時刻演算子

演算子名	例	結果
+	date '2001-09-28' + integer '7'	date '2001-10-05'
+	date '2001-09-28' + interval '1 hour'	timestamp '2001-09-28 01:00:00'
+	date '2001-09-28' + time '03:00'	timestamp '2001-09-28 03:00:00'
+	interval '1 day' + interval '1 hour'	interval '1 day 01:00:00'
+	timestamp '2001-09-28 01:00' + interval '23 hours'	timestamp '2001-09-29 00:00:00'
+	time '01:00' + interval '3 hours'	time '04:00:00'
-	- interval '23 hours'	interval '-23:00:00'
-	date '2001-10-01' - date '2001-09-28'	integer '3' (日付)
-	date '2001-10-01' - integer '7'	date '2001-09-24'
-	date '2001-09-28' - interval '1 hour'	timestamp '2001-09-27 23:00:00'
-	time '05:00' - time '03:00'	interval '02:00:00'
-	time '05:00' - interval '2 hours'	time '03:00:00'
-	timestamp '2001-09-28 23:00' - interval '23 hours'	timestamp '2001-09-28 00:00:00'
-	interval '1 day' - interval '1 hour'	interval '1 day -01:00:00'
-	timestamp '2001-09-29 03:00' - timestamp '2001-09-27 12:00'	interval '1 day 15:00:00'
*	900 * interval '1 second'	interval '00:15:00'
*	21 * interval '1 day'	interval '21 days'
*	double precision '3.5' * interval '1 hour'	interval '03:30:00'
/	interval '1 hour' / double precision '1.5'	interval '00:40:00'

表 9-27. 日付/時刻関数

関数名	戻り値型	説明	例	結果
age(timestamp, timestamp)	interval	引数間の減算。年と月を使用した"シンボルによる"結果を生成	age(timestamp '2001-04-10', timestamp '1957-06-13')	43 years 9 mons 27 days
age(timestamp)	interval	current_date (午前零時) から減算	age(timestamp '1957-06-13')	43 years 8 mons 3 days
clock_timestamp()	timestamp with time zone	現在の日付と時刻です。(文実行時に変わります。) 項9.9.4 を参照。		
current_date	date	現在の日付。 項9.9.4 を参照		
current_time	time with time zone	現在の時刻。 項9.9.4 を参照。		
current_timestamp	timestamp with time zone	現在の日付と時刻 (現在のトランザクションの開始日付時刻)。 項9.9.4 を参照。		
date_part(text, timestamp)	double precision	部分フィールドの取得 (extractと同じ)。 項9.9.1 を参照。	date_part('hour', timestamp '2001-02-16 20:38:40')	20
date_part(text, interval)	double precision	部分フィールドの取得 (extractと同じ)。 項9.9.1 を参照。	date_part('month', interval '2 years 3 months')	3
date_trunc(text, timestamp)	timestamp	指定された精度で切り捨て。 項9.9.2 も参照。	date_trunc('hour', timestamp '2001-02-16 20:38:40')	2001-02-16 20:00:00
extract(field from timestamp)	double precision	部分フィールドの取得。 項9.9.1 を参照。	extract(hour from timestamp '2001-02-16 20:38:40')	20

<code>extract (field from interval)</code>	<code>double precision</code>	部分フィールドの取得。 項9.9.1 を参照。	<code>extract(month from interval '2 years 3 months')</code>	3
<code>isfinite(date)</code>	<code>boolean</code>	日付が有限 (+/-無限と等しくない) かどうかの検査	<code>isfinite(date '2001-02-16')</code>	true
<code>isfinite(timestamp)</code>	<code>boolean</code>	タイムスタンプが有限 (+/-無限でない) かどうかの検査	<code>isfinite(timestamp '2001-02-16 21:28:30')</code>	true
<code>isfinite(interval)</code>	<code>boolean</code>	時間間隔が有限かどうかの検査	<code>isfinite(interval '4 hours')</code>	true
<code>justify_days(interval)</code>	<code>interval</code>	30日周期が1月を表すように時間間隔を調整	<code>justify_days(interval '35 days')</code>	1 mon 5 days
<code>justify_hours(interval)</code>	<code>interval</code>	24時間を1日とする時間間隔の調整	<code>justify_hours(interval '27 hours')</code>	1 day 03:00:00
<code>justify_interval(interval)</code>	<code>interval</code>	<code>justify_days</code> および <code>justify_hours</code> を使用し、さらに符号による調整を行っての時間間隔の調整	<code>justify_interval(interval '1 mon -1 hour')</code>	29 days 23:00:00
<code>localtime</code>	<code>time</code>	現在の時刻。 項9.9.4 を参照。		
<code>localtimestamp</code>	<code>timestamp</code>	現在の日付と時刻（現在のトランザクションの開始）。 項9.9.4 を参照。		
<code>now()</code>	<code>timestamp with time zone</code>	現在の日付と時刻（現在のトランザクションの開始）。 項9.9.4 を参照。		
<code>statement_timestamp()</code>	<code>timestamp with time zone</code>	現在の日付と時刻（現在の文の開始）。 項9.9.4 を参照。		
<code>timeofday()</code>	<code>text</code>	現在の日付と時刻。（ <code>clock_timestamp</code> と似ていますが、 <code>text</code> 型文字列として返す。） 項9.9.4 を参照。		
<code>transaction_timestamp()</code>	<code>timestamp with time zone</code>	現在の日付と時刻（現在のトランザクションの開始）。 項9.9.4 を参照。		

これらの関数に加え、OVERLAPS SQL演算子がサポートされています。

```
(start1, end1) OVERLAPS (start2, end2)
(start1, length1) OVERLAPS (start2, length2)
```

この式は、2つの時間間隔が重なる（その終端で定義されます）時に真を返します。重ならない場合は偽を返します。終端は日付、時刻、タイムスタンプ、もしくは、日付/時刻/タイムスタンプに続く時間間隔で指定されます。値の組み合わせが与えられた際、開始と終了のいずれを先に記述しても構いません。OVERLAPSは与えられた値の内、若いものを開始として扱います。各時間間隔は、`start <= time < end`といった半開区間として見なされます。ただし、`start`と`end`が同じ値の場合には単一の時間点となります。これは、例えば終了点のみが同じである2つの時間間隔は、必ずしも重ならないことを意味します。

```
SELECT (DATE '2001-02-16', DATE '2001-12-21') OVERLAPS
       (DATE '2001-10-30', DATE '2002-10-30');
Result: true
SELECT (DATE '2001-02-16', INTERVAL '100 days') OVERLAPS
       (DATE '2001-10-30', DATE '2002-10-30');
Result: false
SELECT (DATE '2001-10-29', DATE '2001-10-30') OVERLAPS
       (DATE '2001-10-30', DATE '2001-10-31');
Result: false
SELECT (DATE '2001-10-30', DATE '2001-10-30') OVERLAPS
       (DATE '2001-10-30', DATE '2001-10-31');
Result: true
```

`timestamp with time zone`の値に`interval`の値を加える時（または`timestamp with time zone`の値から`interval`の値を差し引く時）日にちの部分は、日にちの数で示された `timestamp with time zone`の日付を先に進めます（もしくは後に戻します）。夏時間への移行に跨っての変更に関しては（セッションの時間帯がDSTを認識するようになっていれば）、`interval '1 day'`が`interval '24 hours'`に等しい必要はありません。例えば、セッションの時間帯がCST7CDTに設定されている時に、`timestamp with time zone '2005-04-02 12:00-07' + interval '1 day'` は、`timestamp with time zone '2005-04-03 12:00-06'`をもたらします。一方同じ初期`timestamp with time zone`に`interval '24 hours'`を加えると、`timestamp with time zone '2005-04-03 13:00-06'`という結果になります。その理由はCST7CDT時間帯で2005-04-03 02:00に夏時間への変更があるからです。

異なる月では日数が異なりますので`age`で返される`months`にはあいまいさがあります。PostgreSQLのやり方は月にまたがる2つの日付の計算において、日付の早いほうの月を使用します。例えば、`age('2004-06-01', '2004-04-30')`は4月を使用して1 mon 1 dayを得ます。一方5月を使用すると、5月は31日あり、4月は30日のため、結果は1 mon 2 daysです。

9.9.1. EXTRACT, date_part

`EXTRACT(field FROM source)`

`extract`関数は、日付/時刻の値から年や時などの部分フィールドを抽出します。*source*はtimestamp型、time型、またはinterval型の評価式でなければなりません（date型の式はtimestamp型にキャストされますので、同様に使用可能です）。*field*はsourceの値からどのフィールドを抽出するかを選択する識別子もしくは文字列です。`extract`関数はdouble precision型の値を返します。以下に有効なフィールド名を示します。

century

世紀

```
SELECT EXTRACT(CENTURY FROM TIMESTAMP '2000-12-16 12:21:13');
Result: 20
SELECT EXTRACT(CENTURY FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 21
```

当時の人々にはそのような意識はありませんでしたが、最初の世紀は0001-01-01 00:00:00 ADから始まります。この定義は全てのグレゴリアン暦を使用する国で適用されています。0という値の世紀はありません。-1世紀の次は1世紀です。この定義に納得できなければ、苦情をバチカンローマ聖パウロ大聖堂のローマ法王に伝えてください。

PostgreSQLリリース8.0以前では、世紀の番号付けの慣習に従っていませんでした。単に年を100で除算したものを返していました。

day

timestamp値については、(月内の) 日付フィールド（1～31）。interval値については日付。

```
SELECT EXTRACT(DAY FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 16

SELECT EXTRACT(DAY FROM INTERVAL '40 days 1 minute');
Result: 40
```

decade

年フィールドを10で割ったもの

```
SELECT EXTRACT(DECADE FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 200
```

dow

日曜日（0）から土曜日（6）までの曜日

```
SELECT EXTRACT(DOW FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 5
```

`extract`関数の曜日指定番号は`to_char(..., 'D')`関数のそれとは異なる点に注意してください。

doy

年内での通算日数（1～365/366）

```
SELECT EXTRACT(DOY FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 47
```

epoch

timestamp with time zone型の値において、1970-01-01 00:00:00 UTCからの秒数（負の数の場合もあり）。dateとtimestamp型の値において、ローカルタイムの1970-01-01 00:00:00からの秒数。interval型の値ではその時間間隔における秒の合計。

```
SELECT EXTRACT(EPOCH FROM TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40.12-08');
Result: 982384720.12

SELECT EXTRACT(EPOCH FROM INTERVAL '5 days 3 hours');
Result: 442800
```

以下に、この経過秒数をタイムスタンプ値に変換する方法を示します。

```
SELECT TIMESTAMP WITH TIME ZONE 'epoch' + 982384720.12 * INTERVAL '1 second';
```

(`to_timestamp`関数は上記の変換をカプセル化します。)

hour

時のフィールド（0～23）

```
SELECT EXTRACT(HOUR FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 20
```

isodow

月曜日 (1) から日曜日 (7) までの曜日

```
SELECT EXTRACT(ISODOW FROM TIMESTAMP '2001-02-18 20:38:40');
Result: 7
```

日曜日を除きdowと同一。ISO 8601曜日番号付けに一致。

isoyear

日付が同期するISO 8601年 (intervalには対応しない)。

```
SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-01');
Result: 2005
SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-02');
Result: 2006
```

全てのISO年は1月4日を含む週の月曜日から開始されます。従って、1月上旬、または12月下旬でISO年がグレゴリオ年と異なる可能性があります。より詳細はweekフィールドを参照してください。

このフィールドは8.3以前のPostgreSQLリリースでは有効ではありません。

microseconds

端数部分も含み、1,000,000を乗じられた秒フィールド。全ての秒を含むことに注意。

```
SELECT EXTRACT(MICROSECONDS FROM TIME '17:12:28.5');
Result: 28500000
```

millennium

ミレニアム (1千年期間)

```
SELECT EXTRACT(MILLENNIUM FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 3
```

1900年代の年は第2ミレニアムです。第3ミレニアムは2001年1月1日から始まりました。

PostgreSQLリリース8.0以前では、ミレニアムの番号付けの慣習に従っていませんでした。単に年フィールドを1000で割った値を返していました。

milliseconds

端数部分も含み、1000を乗じられた秒フィールド。全ての秒を含むことに注意してください。

```
SELECT EXTRACT(MILLISECONDS FROM TIME '17:12:28.5');
Result: 28500
```

minute

分フィールド (0~59)

```
SELECT EXTRACT(MINUTE FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 38
```

month

timestamp型の値に対しては年内の月番号 (1~12)。interval型の値に対しては月番号で、12の剰余 (0~11)。

```
SELECT EXTRACT(MONTH FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 2
```

```
SELECT EXTRACT(MONTH FROM INTERVAL '2 years 3 months');
Result: 3
```

```
SELECT EXTRACT(MONTH FROM INTERVAL '2 years 13 months');
Result: 1
```

quarter

その日付が含まれる年の四半期 (1~4)。

```
SELECT EXTRACT(QUARTER FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 1
```

second

端数を含んだ秒フィールド (0~59、[\[1\]](#))。

```
SELECT EXTRACT(SECOND FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 40

SELECT EXTRACT(SECOND FROM TIME '17:12:28.5');
Result: 28.5
```

timezone

秒単位のUTCからの時間帯オフセット。正の値はUTCより東の時間帯に対応し、負の値はUTCより西の時間帯に対応。(技術的には、PostgreSQLはうるう秒を制御しないためUT1を使用。)

timezone_hour

時間帯オフセットの時の成分。

timezone_minute

時間帯オフセットの分の成分。

week

その日の年間通算での週を計算します。(ISO 8601の)定義では週は月曜日から始まり、その年の1月4日を含む週をその年の第1週としています。つまり、年の最初の木曜日がある週がその年の第1週となります。

ISOの定義では、1月の早い日にちは前年の第52週もしくは第53週となることがあり、12月の遅い日にちには次年の最初の週の一部となることがあります。例えば、2005-01-01は2004年の第53週であり、2006-01-01は2005年の第52週の一部です、一方2012-12-31は2013年の第1週の一部となります。整合性のある結果を得るため、isoyearフィールドとweekを併用することを推奨します。

```
SELECT EXTRACT(WEEK FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 7
```

year

年フィールド。AD零年が存在しないことは忘れないでください。このためADの年からBCの年を減する時には注意が必要です。

```
SELECT EXTRACT(YEAR FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 2001
```

extract関数は元々演算処理の目的でした。日付/時刻の値を表示する目的での書式については[項9.8](#)を参照してください。

date_part関数は伝統的なIngres上で設計されたもので、標準SQLのextract関数と等価です。

```
date_part('field', source)
```

ここでfieldパラメータが名前ではなく文字列値である必要があることに注意してください。date_partで有効なフィールド名はextractと同じです。

```
SELECT date_part('day', TIMESTAMP '2001-02-16 20:38:40');
Result: 16

SELECT date_part('hour', INTERVAL '4 hours 3 minutes');
Result: 4
```

9.9.2. date_trunc

date_trunc関数は概念的に数値に対するtrunc関数と類似しています。

```
date_trunc('field', source)
```

sourceは、データ型timestampもしくはintervalの評価式です(データ型dateとtimeはそれぞれ自動的にtimestampもしくはintervalにキャストされます)。fieldは、入力値の値をどの精度で切り捨てるかを選択します。戻り値の値は、選択されたものの以下をゼロに設定(日と月の場合は1に設定)した、全てのフィールドを持つtimestampもしくはinterval型です。

fieldの有効値には次のものがあります。

```
microseconds
milliseconds
second
minute
hour
day
week
month
quarter
```

year
decade
century
millennium

例：

```
SELECT date_trunc('hour', TIMESTAMP '2001-02-16 20:38:40');
Result: 2001-02-16 20:00:00

SELECT date_trunc('year', TIMESTAMP '2001-02-16 20:38:40');
Result: 2001-01-01 00:00:00
```

9.9.3. AT TIME ZONE

AT TIME ZONE構文を使用することにより、タイムスタンプを異なる時間帯に変換することができます。表9-28にその種類を示します。

表 9-28. AT TIME ZONE Variants

式	戻り値	説明
timestamp without time zone AT TIME ZONE zone	timestamp with time zone	与えられた時間帯なしタイムスタンプを指定された時間帯にあるとして取り扱います。
timestamp with time zone AT TIME ZONE zone	timestamp without time zone	与えられた時間帯付きタイムスタンプを新規の時間帯に、時間帯の指定なく変換します。
time with time zone AT TIME ZONE zone	time with time zone	与えられた時刻with time zoneを新しい時間帯に変換します。

これらの式では、設定する時間帯zoneは、（'PST'のような）テキスト文字列、または（INTERVAL '-08:00'のような）時間間隔で指定することができます。 テキストの場合、項8.5.3に示した方法で時間帯名称を指定することができます。

以下に例を示します（ローカル時間帯をPST8PDTと想定しています）。

```
SELECT TIMESTAMP '2001-02-16 20:38:40' AT TIME ZONE 'MST';
Result: 2001-02-16 19:38:40-08

SELECT TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40-05' AT TIME ZONE 'MST';
Result: 2001-02-16 18:38:40
```

最初の例は、時間帯のないタイプスタンプを使用し、それをMST時間（UTC-7）として解釈し、UTCタイムスタンプを生成します。それから、UTCタイムスタンプが、表示用にPST（UTC-7）に置き換えられます。2番目の例は、EST（UTC-5）で指定されたタイムスタンプを使用し、MST（UTC-7）でのローカル時間に変換しています。

関数timezone(zone, timestamp)は、SQL準拠の構文timestamp AT TIME ZONE zoneと等価です。

9.9.4. Current Date/Time

PostgreSQLは、現在の日付時刻に関した値を返す多くの関数を提供します。 これらの標準SQL関数はすべて、現在のトランザクションの開始時刻に基づいた値を返します。

```
CURRENT_DATE
CURRENT_TIME
CURRENT_TIMESTAMP
CURRENT_TIME(precision)
CURRENT_TIMESTAMP(precision)
LOCALTIME
LOCALTIMESTAMP
LOCALTIME(precision)
LOCALTIMESTAMP(precision)
```

CURRENT_TIMEおよびCURRENT_TIMESTAMP関数では、時間帯を伴う値を扱います。一方、LOCALTIMEおよびLOCALTIMESTAMP関数では、時間帯を伴わない値を扱います。

CURRENT_TIME、CURRENT_TIMESTAMP、LOCALTIME、およびLOCALTIMESTAMP関数では、精度のパラメータをオプションで取ることができ、それに合わせて秒フィールドの端数桁を丸める結果をもたらします。精度のパラメータがない場合、結果は使用可能な最大精度で出力されます。

例：

```
SELECT CURRENT_TIME;
```



```

Result: 14:39:53.662522-05

SELECT CURRENT_DATE;
Result: 2001-12-23

SELECT CURRENT_TIMESTAMP;
Result: 2001-12-23 14:39:53.662522-05

SELECT CURRENT_TIMESTAMP(2);
Result: 2001-12-23 14:39:53.66-05

SELECT LOCALTIMESTAMP;
Result: 2001-12-23 14:39:53.662522

```

これらの関数は、現在のトランザクションの開始時刻を返します。この値は、トランザクションが実行されている間は変化しません。これは、次の機能を検討した結果です。単一トランザクションで、"current"時間を一貫性を持った表現を行うことができるようにすることを目的とし、このため、同一トランザクションで何回変更を行っても同一のタイムスタンプを生成します。

注意: 他のデータベースシステムでは、これらの値をより頻繁に増加させることがあります。

PostgreSQLはまた、関数を呼び出した時の実際の現在時刻や現在の文の開始時刻を返す関数も提供します。非標準SQLの関数の全一覧を以下に示します。

```

transaction_timestamp()
statement_timestamp()
clock_timestamp()
timeofday()
now()

```

`transaction_timestamp()` は `CURRENT_TIMESTAMP` と等価ですが、明確に何を返すかを反映する名前になっています。`statement_timestamp()` は現在の文の実行開始時刻を返すものです（より具体的にいうと、直前のコマンドメッセージをクライアントから受け取った時刻です）。`statement_timestamp()` および `transaction_timestamp()` はトランザクションの最初のコマンドでは同じ値を返しますが、その後に引き続くコマンドでは異なる可能性があります。`clock_timestamp()` は実際の現在時刻を返しますので、その値は単一のSQLコマンドであっても異なります。`timeofday()` はPostgreSQLの歴史的な関数です。`clock_timestamp()` 同様、実際の現在時刻を返しますが、`timestamp with time zone` 型の値ではなく、整形されたtext文字列を返します。`now()` は `transaction_timestamp()` と同じもので、伝統的なPostgreSQL関数です。

すべての日付/時刻型はまた、現在の日付と時刻（繰り返しますがトランザクションの開始時刻として解釈されます）を表す特殊なリテラル値 `now` を受け付けます。したがって、下記の3つの実行結果は全て同じものとなります。

```

SELECT CURRENT_TIMESTAMP;
SELECT now();
SELECT TIMESTAMP 'now'; -- incorrect for use with DEFAULT

```

ティップ: テーブルを作成する時、`DEFAULT` 句を指定するのに3番目の形式を使おうとは思わないでしょう。定数が解析された時、システムが `now` を `timestamp` に変換するので、デフォルト値としてテーブルが作成された時刻が使われます。最初の2つの形式は関数呼び出しのためデフォルト値が使用されるまで評価されません。ですから、これらの関数は列の挿入時間をデフォルトとする、望ましい振舞いをします。

9.9.5. 遅延実行

以下の関数は、サーバプロセスの実行を遅延させるために使用可能です。

```
pg_sleep(seconds)
```

`pg_sleep` は、`seconds` 秒経過するまで、現在のセッションのプロセスを休止させます。`seconds` は `double precision` 型の値です。そのため、小数単位で休止秒数を指定することができます。以下に例を示します。

```
SELECT pg_sleep(1.5);
```

注意: 休止時間の有効な分解能はプラットフォームに依存します。0.01秒が一般的な値です。遅延は少なくとも指定した通り行われます。サーバの負荷が要因となり、より長くなる可能性があります。

警告

`pg_sleep` を呼び出す時、セッションが必要以上のロックを保持していないことを確実にしてください。さもないと、他のセッションが休止中のプロセスを待機しなければならないかもしれません。そのためシステム全体の速度が低下することになるかもしれません。

注意

[\[1\]](#) オペレーティングシステムでうるう秒が実装されている場合は60まで。

[前のページ](#)

データ型書式設定関数

[ホーム](#)
[上に戻る](#)

[次のページ](#)

列挙型サポート関数

9.10. 列挙型サポート関数

列挙型（[項8.7](#)で解説）に対し、特に列挙型の値をハードコーディングせず簡潔なプログラミングを可能にするいくつかの関数があります。それらの関数は[表9-29](#)で一覧されています。例は以下のようにして列挙型が作成されていることを想定しています。

```
CREATE TYPE rainbow AS ENUM ('red', 'orange', 'yellow', 'green', 'blue', 'purple');
```

表 9-29. 列挙型サポート関数

関数	説明	例	例の結果
enum_first(anyenum)	最初を入力列挙型の値を返す	enum_first(null::rainbow)	red
enum_last(anyenum)	最後を入力列挙型の値を返す	enum_last(null::rainbow)	purple
enum_range(anyenum)	ある順序付けられた配列にもとづいて入力列挙型の全ての値を返す	enum_range(null::rainbow)	{red,orange,yellow,green,blue,purple}
enum_range(anyenum, anyenum)	与えられた2つの列挙型値の範囲を、ある順序付けられた配列として返す。値は同一の列挙型に拠らなければならない。始めのパラメータがNULLの場合、結果は列挙型の最初の値から始まる。2番目のパラメータがNULLの場合、結果は列挙型の最後の値で終端する。	enum_range('orange'::rainbow, 'green'::rainbow)	{orange,yellow,green}
		enum_range(NULL, 'green'::rainbow)	{red,orange,yellow,green}
		enum_range('orange'::rainbow, NULL)	{orange,yellow,green,blue,purple}

enum_rangeの2引数形式を除いて、関数に渡された特定の値はそれら関数が無視することに注意してください。関数は宣言されたデータ型のみ配慮します。NULLまたはその型の特定値のみが渡され、同一の結果をもたらします。例が示唆しているように、組み込まれている型名に対してではなく、テーブル列もしくは関数引数にこれらの関数を適用することがより一般的です。

9.11. 幾何関数と演算子

point、box、lseg、line、path、polygon、およびcircle幾何データ型には、PostgreSQLが元々サポートしている関数と演算子が豊富に揃っています（[表9-30](#)、[表9-31](#)、および[表9-32](#)を参照してください）。

注意
"同じを示す"~=演算子はpoint、box、polygon、およびcircle型に対し通常の等価概念を示すことに注意してください。これらのいくつかの型は=演算子を持ちますが、=は面積の等しさのみを比較します。その他のスカラー比較演算子（<=など）は同様にこれらの型の面積を比較します。

表 9-30. 幾何データ演算子

演算子	説明	例
+	平行移動	box '((0,0),(1,1))' + point '(2.0,0)'
-	平行移動	box '((0,0),(1,1))' - point '(2.0,0)'
*	拡大縮小/回転	box '((0,0),(1,1))' * point '(2.0,0)'
/	拡大縮小/回転	box '((0,0),(2,2))' / point '(2.0,0)'
#	交差点、交差矩形	'((1,-1),(-1,1))' # '((1,1),(-1,-1))'
#	経路もしくは多角形の点の数	# '((1,0),(0,1),(-1,0))'
@-@	長さもしくは円周	@-@ path '((0,0),(1,0))'
@@	中心	@@ circle '((0,0),10)'
##	第1演算項目と第2演算項目との近接点	point '(0,0)' ## lseg '((2,0),(0,2))'
<->	距離	circle '((0,0),1)' <-> circle '((5,0),1)'
&&	重なっているかどうか？(一つでも共通する点があれば真となります)	box '((0,0),(1,1))' && box '((0,0),(2,2))'
<<	厳密に左側にあるか？	circle '((0,0),1)' << circle '((5,0),1)'
>>	厳密に右側にあるか？	circle '((5,0),1)' >> circle '((0,0),1)'
&<	右側にはみ出さないか？	box '((0,0),(1,1))' &< box '((0,0),(2,2))'
&>	左側にはみ出さないか？	box '((0,0),(3,3))' &> box '((0,0),(2,2))'
<<	厳密に下側にあるか？	box '((0,0),(3,3))' << box '((3,4),(5,5))'
>>	厳密に上側にあるか？	box '((3,4),(5,5))' >> box '((0,0),(3,3))'
&<	上側に出ているか？	box '((0,0),(1,1))' &< box '((0,0),(2,2))'
&>	下側に出ているか？	box '((0,0),(3,3))' &> box '((0,0),(2,2))'
<^	(接触を許可し) 下側にあるか？	circle '((0,0),1)' <^ circle '((0,5),1)'
>^	(接触を許可し) 上側にあるか？	circle '((0,5),1)' >^ circle '((0,0),1)'
?#	交わるか？	lseg '((-1,0),(1,0))' ?# box '((-2,-2),(2,2))'
?-	水平か？	?- lseg '((-1,0),(1,0))'
?-	水平方向に揃っているか？	point '(1,0)' ?- point '(0,0)'
?	垂直か？	? lseg '((-1,0),(1,0))'
?	垂直方向に揃っているか？	point '(0,1)' ? point '(0,0)'
?-	直角をなしているか？	lseg '((0,0),(0,1))' ?- lseg '((0,0),(1,0))'
?	平行か？	lseg '((-1,0),(1,0))' ? lseg '((-1,2),(1,2))'
@>	包含しているか？	circle '((0,0),2)' @> point '(1,1)'
<@	包含もしくは境界上を含むか？	point '(1,1)' <@ circle '((0,0),2)'
~=	同等か？	polygon '((0,0),(1,1))' ~= polygon '((1,1),(0,0))'

注意: PostgreSQL 8.2より前では、含有演算子@>および@<はそれぞれ~および~という名前でした。これらの名前はまだ利用できますが、削除予定で予告なしでなくなるかもしれません。

表 9-31. 幾何データ型関数

関数	戻り値	説明	例
area(<i>object</i>)	double precision	面積	area(box '((0,0),(1,1))')
center(<i>object</i>)	point	中心	center(box '((0,0),(1,2))')
diameter(circle)	double precision	円の直径	diameter(circle '((0,0),2.0)')
height(box)	double precision	矩形の高さ	height(box '((0,0),(1,1))')
isclosed(path)	boolean	閉経路か？	isclosed(path '((0,0),(1,1),(2,0))')
isopen(path)	boolean	開経路か？	isopen(path '[(0,0),(1,1),(2,0)]')
length(<i>object</i>)	double precision	長さ	length(path '((-1,0),(1,0))')
npoints(path)	int	点の数	npoints(path '[(0,0),(1,1),(2,0)]')
npoints(polygon)	int	点の数	npoints(polygon '((1,1),(0,0))')
pclose(path)	path	閉経路に変換	pclose(path '[(0,0),(1,1),(2,0)]')
popen(path)	path	開経路に変換	popen(path '((0,0),(1,1),(2,0))')
radius(circle)	double precision	円の半径	radius(circle '((0,0),2.0)')
width(box)	double precision	矩形の幅	width(box '((0,0),(1,1))')

表 9-32. 幾何型変換関数

関数	戻り値	説明	例
box(circle)	box	円から矩形	box(circle '((0,0),2.0)')
box(point, point)	box	座標点を矩形に変換	box(point '(0,0)', point '(1,1)')
box(polygon)	box	多角形を矩形に変換	box(polygon '((0,0),(1,1),(2,0))')
circle(box)	circle	矩形を円に変換	circle(box '((0,0),(1,1))')
circle(point, double precision)	circle	中心と半径から円に変換	circle(point '(0,0)', 2.0)
circle(polygon)	circle	多角形を円に変換	circle(polygon '((0,0),(1,1),(2,0))')
lseg(box)	lseg	矩形の対角線を線分に変換	lseg(box '((-1,0),(1,0))')
lseg(point, point)	lseg	座標点を線分に変換	lseg(point '(-1,0)', point '(1,0)')
path(polygon)	path	多角形を経路に変換	path(polygon '((0,0),(1,1),(2,0))')
point(double precision, double precision)	point	座標点の構築	point(23.4, -44.5)
point(box)	point	矩形の中心	point(box '((-1,0),(1,0))')
point(circle)	point	円の中心	point(circle '((0,0),2.0)')
point(lseg)	point	線分の中心	point(lseg '((-1,0),(1,0))')
point(polygon)	point	多角形の中心	point(polygon '((0,0),(1,1),(2,0))')
polygon(box)	polygon	矩形から4頂点の多角形	polygon(box '((0,0),(1,1))')
polygon(circle)	polygon	円から12角形	polygon(circle '((0,0),2.0)')
polygon(<i>npts</i> , circle)	polygon	円から <i>npts</i> 角形	polygon(12, circle '((0,0),2.0)')
polygon(path)	polygon	経路から多角形	polygon(path '((0,0),(1,1),(2,0))')

あたかもpointは添字0、1を有する配列であるかのように、pointの2つの構成要素にアクセスすることができます。例えば、t.pがpoint列の場合、SELECT p[0] FROM tという式でX座標点を抽出できます。また、UPDATE t SET p[1] = ...でY座標点を変更できます。同様に、boxまたはlsegでも、2つのpointからなる配列のように扱えます。

area関数は、box、circle、path型に対して動作します。pathデータ型に対するarea関数は、そのpathが交差しない場合にのみ動作します。例えば、'((0,0),(0,1),(2,1),(2,2),(1,2),(1,0),(0,0))':PATHというpathは動作しません。しかし、見た目は同じですが、'((0,0),(0,1),(1,1),(1,2),(2,2),(2,1),(1,1),(1,0),(0,0))':PATHというpathでは動作します。pathの交差する、しないという概念がよくわからなければ、上の2つのpathを並べてグラフ用紙に書いてみてください。

9.12. ネットワークアドレス関数と演算子

[表9-33](#)に、inet型およびcidr型で使用可能な演算子を示します。演算子 <<, <=, >>, および >>= はサブネットに含まれているかどうかを調べるものです。ネットワークの部分のみを考慮（ネットワークアドレスのホスト部分を無視）して2つのネットワークアドレスが等しいか、もしくは一方のネットワークが他方のサブネットになっているかを特定します。

表 9-33. cidrとinet演算子

演算子	説明	例
<	未満	inet '192.168.1.5' < inet '192.168.1.6'
<=	未満もしくは等しい（以下）	inet '192.168.1.5' <= inet '192.168.1.5'
=	等しい	inet '192.168.1.5' = inet '192.168.1.5'
>=	より大きいもしくは等しい（以上）	inet '192.168.1.5' >= inet '192.168.1.5'
>	より大きい	inet '192.168.1.5' > inet '192.168.1.4'
<>	等しくない	inet '192.168.1.5' <> inet '192.168.1.4'
<<	内包される	inet '192.168.1.5' << inet '192.168.1/24'
<=<	内包されるか等しい	inet '192.168.1/24' <=< inet '192.168.1/24'
>>	内包する	inet '192.168.1/24' >> inet '192.168.1.5'
>=>	内包するか等しい	inet '192.168.1/24' >=> inet '192.168.1/24'
~	ビット否定	~ inet '192.168.1.6'
&	ビット積	inet '192.168.1.6' & inet '0.0.0.255'
	ビット和	inet '192.168.1.6' inet '0.0.0.255'
+	加算	inet '192.168.1.6' + 25
-	減算	inet '192.168.1.43' - 36
-	減算	inet '192.168.1.43' - inet '192.168.1.19'

[表9-34](#)に、inet型およびcidr型で使用可能な関数を示します。関数abbrev、host()、およびtext()、は主として、代替の整形表示を提供する目的のものです。

表 9-34. cidrおよびinet型関数

関数	戻り値	説明	例	結果
abbrev(inet)	text	表示用テキスト省略形	abbrev(inet '10.1.0.0/16')	10.1.0.0/16
abbrev(cidr)	text	表示用テキスト省略形	abbrev(cidr '10.1.0.0/16')	10.1/16
broadcast(inet)	inet	ネットワークブロードキャストアドレス	broadcast('192.168.1.5/24')	192.168.1.255/24
family(inet)	int	アドレスファミリの取り出し。IPv4では4、IPv6では6。	family(':::1')	6
host(inet)	text	IPアドレスをテキストで抽出	host('192.168.1.5/24')	192.168.1.5
hostmask(inet)	inet	ネットワーク用のホストマスクの構築	hostmask('192.168.23.20/30')	0.0.0.3
masklen(inet)	int	ネットマスク長の抽出	masklen('192.168.1.5/24')	24
netmask(inet)	inet	ネットワーク用ネットマスクの構築	netmask('192.168.1.5/24')	255.255.255.0
network(inet)	cidr	ネットワークアドレスからネットワーク部を抽出	network('192.168.1.5/24')	192.168.1.0/24
set_masklen(inet, int)	inet	inet値のネットマスク長を設定	set_masklen('192.168.1.5/24', 16)	192.168.1.5/16
set_masklen(cidr, int)	cidr	cidr値のネットマスク長を設定	set_masklen('192.168.1.0/24'::cidr, 16)	192.168.0.0/16
text(inet)	text	IPアドレスとネットマスク長をテキスト	text(inet '192.168.1.5')	192.168.1.5/32

`cidr`値はすべて、暗黙的または明示的に`inet`型にキャスト可能です。したがって、上で示した`inet`に対する操作を行う関数は`cidr`値でも動作します。（`inet`と`cidr`で別々に関数が存在しますが、それは、この2つで動作に違いがあるためです。）また、`inet`値から`cidr`へのキャストも許されます。これが行われると、ネットマスクの右側のビットは、`cidr`として有効な値とするために0となります。さらに、通常のキャスト構文、例えば、`inet(expression)`や`colname::cidr`を使用して、テキスト値を`inet`や`cidr`にキャストすることもできます。

[表9-35](#)に、`macaddr`型で使用可能な関数を示します。関数`trunc(macaddr)`は終わりの3バイトがゼロに設定されたMACアドレスを返します。これは、残った接頭部を製造者に対応させることに使えます。

表 9-35. `macaddr`型関数

関数	戻り値	説明	例	結果
<code>trunc(macaddr)</code>	<code>macaddr</code>	終わりの3バイトをゼロに設定	<code>trunc(macaddr '12:34:56:78:90:ab')</code>	<code>12:34:56:00:00:00</code>

`macaddr`型は（`>`、`<=`等の）辞書編集順に並べる標準関係演算子とNOT、AND、ORのビット演算子（`~`、`&`、`|`）もサポートしています。

9.13. テキスト検索関数と演算子

[表9-36](#)、[表9-37](#)および [表9-38](#)は全文検索用に提供されている関数と演算子を要約しています。PostgreSQLのテキスト検索機能の詳細は[第12章](#)を参照してください。

表 9-36. テキスト検索演算子

演算子	説明	例	結果
@@	tsvectorがtsqueryに一致するか？	to_tsvector('fat cats ate rats') @@ to_tsquery('cat & rat')	t
@@@	@@に対する廃止予定の同義語	to_tsvector('fat cats ate rats') @@@ to_tsquery('cat & rat')	t
	tsvector (複数) を連結	'a:1 b:2'::tsvector 'c:1 d:2 b:3'::tsvector	'a':1 'b':2,5 'c':3 'd':4
&&	tsquery (複数) をともに積	'fat rat'::tsquery && 'cat'::tsquery	('fat' 'rat') & 'cat'
	tsquery (複数) をともに和	'fat rat'::tsquery 'cat'::tsquery	('fat' 'rat') 'cat'
!!	tsqueryを否定	!! 'cat'::tsquery	!'cat'
@>	tsqueryは他を包含するか？	'cat'::tsquery @> 'cat & rat'::tsquery	f
<@	tsqueryは包含されるか？	'cat'::tsquery <@ 'cat & rat'::tsquery	t

注意: tsquery 格納演算子は結合された演算子を無視し、2つの問い合わせでリストされた語彙素のみを対象とします。

表に示された演算子に加え、通常のB-tree比較演算子(=、<など)が、型tsvectorおよびtsqueryに対して定義されます。これらはテキスト検索に対してそれほど有用ではありませんが、例えばこれらの型の列に一意インデックスを作成することを許可します。

表 9-37. テキスト検索関数

関数	戻り値型	説明	例	結果
get_current_ts_config()	regconfig	デフォルトの全文検索設定の取得	get_current_ts_config()	english
length(tsvector)	integer	tsvectorにある語彙素の数	length('fat:2,4 cat:3 rat:5A'::tsvector)	3
numnode(tsquery)	integer	tsqueryにある語彙素の数と演算子の数	numnode('(fat & rat) cat'::tsquery)	5
plainto_tsquery([config regconfig ,] query text)	tsquery	句読点を無視して、tsqueryを作成	plainto_tsquery('english', 'The Fat Rats')	'fat' & 'rat'
querytree(query tsquery)	text	tsqueryのインデックス付け可能部分の取得	querytree('foo & ! bar'::tsquery)	'foo'
setweight(tsvector, "char")	tsvector	tsvectorのそれぞれの要素に重みを割り当て	setweight('fat:2,4 cat:3 rat:5B'::tsvector, 'A')	'cat':3A 'fat':2A,4A 'rat':5A
strip(tsvector)	tsvector	tsvectorから位置と重みを削除	strip('fat:2,4 cat:3 rat:5A'::tsvector)	'cat' 'fat' 'rat'
to_tsquery([config regconfig ,] query text)	tsquery	単語 (複数) を正規化しtsqueryに変換	to_tsquery('english', 'The & Fat & RaTs')	'fat' & 'rat'
to_tsvector([config regconfig ,] document text)	tsvector	ドキュメントテキストをtsvectorに	to_tsvector('english', 'The Fat RaTs')	'fat':2 'rat':3

		縮小		
<code>ts_headline([config regconfig,] document text, query tsquery [, options text])</code>	text	問い合わせによるマッチを表示	<code>ts_headline('x y z', 'z'::tsquery)</code>	<code>x y</code> <code>z</code>
<code>ts_rank([weights float4[],] vector tsvector, query tsquery [, normalization integer])</code>	float4	問い合わせのためのドキュメント順位付け	<code>ts_rank(textsearch, query)</code>	0.818
<code>ts_rank_cd([weights float4[],] vector tsvector, query tsquery [, normalization integer])</code>	float4	cover densityを用いた問い合わせのためのドキュメント順位付け	<code>ts_rank_cd('{0.1, 0.2, 0.4, 1.0}', textsearch, query)</code>	2.01317
<code>ts_rewrite(query tsquery, target tsquery, substitute tsquery)</code>	tsquery	問い合わせ内で対象を代替と置換	<code>ts_rewrite('a & b'::tsquery, 'a'::tsquery, 'foo bar'::tsquery)</code>	<code>'b' & (</code> <code>'foo' </code> <code>'bar')</code>
<code>ts_rewrite(query tsquery, select text)</code>	tsquery	SELECT から対象と代替を使用して置換	<code>SELECT ts_rewrite('a & b'::tsquery, 'SELECT t,s FROM aliases')</code>	<code>'b' & (</code> <code>'foo' </code> <code>'bar')</code>
<code>tsvector_update_trigger()</code>	trigger	tsvector列自動更新のため関数をトリガ	<code>CREATE TRIGGER ... tsvector_update_trigger(tsvcol, 'pg_catalog.swedish', title, body)</code>	
<code>tsvector_update_trigger_column()</code>	trigger	tsvector列自動更新のため関数をトリガ	<code>CREATE TRIGGER ... tsvector_update_trigger_column(tsvcol, configcol, title, body)</code>	

注意: オプションのregconfig引数を受け付ける全てのテキスト検索関数は、その引数が省略された場合default_text_search_configで指定された構成を使用します。

表9-38の関数は、日常のテキスト検索操作では通常使用されないなので、別途リストされます。

表 9-38. テキスト検索デバッグ関数

関数	戻り値型	説明	例	結果
<code>ts_debug([config regconfig,] document text, OUT alias text, OUT description text, OUT token text, OUT dictionaries regdictionary[], OUT dictionary regdictionary, OUT lexemes text[])</code>	setof record	構成を検査	<code>ts_debug('english', 'The Brightest supernovaes')</code>	<code>(asciiword,"Word, all ASCII",The,{english_stem},english_stem,{})</code> ...
<code>ts_lexize(dict regdictionary, token text)</code>	text[]	辞書を検査	<code>ts_lexize('english_stem', 'stars')</code>	<code>{star}</code>
<code>ts_parse(parser_name text, document text, OUT tokid integer, OUT token text)</code>	setof record	パーサを検査	<code>ts_parse('default', 'foo - bar')</code>	<code>(1,foo) ...</code>
<code>ts_parse(parser_oid oid, document text, OUT tokid integer, OUT token text)</code>	setof record	パーサを検査	<code>ts_parse(3722, 'foo - bar')</code>	<code>(1,foo) ...</code>
<code>ts_token_type(parser_name text, OUT tokid integer, OUT alias text, OUT description text)</code>	setof record	パーサで定義されたトークンの型を入手	<code>ts_token_type('default')</code>	<code>(1,asciiword,"Word, all ASCII") ...</code>
<code>ts_token_type(parser_oid oid, OUT tokid integer, OUT alias text, OUT description text)</code>	setof record	パーサで定義されたトークンの型を入手	<code>ts_token_type(3722)</code>	<code>(1,asciiword,"Word, all ASCII") ...</code>
<code>ts_stat(sqlquery text, [weights text,] OUT word text, OUT ndoc integer, OUT nentry integer)</code>	setof record	tsvector列の統計情報を入手	<code>ts_stat('SELECT vector from apod')</code>	<code>(foo,10,15) ...</code>

9.14. XML関数

この節で説明される関数および擬似関数式は、`xml`型の値に対して機能します。`xml`型についての情報は[項8.13](#)を点検してください。`xml`型のやりとりを変換する`xmlparse`および`xmlserialize`擬似関数式はここでは繰り返しません。これらの多くの関数を使用するには、インストレーションの際`configure --with-libxml`付きでビルドされている必要があります。

9.14.1. XML内容の生成

SQLデータからXML内容を生成するために関数と擬似関数式の一式が提供されています。そのようなものとして、クライアントアプリケーションが問い合わせ結果を処理のためXML文書に書式化するのにこれらは特に適しています。

9.14.1.1. `xmlcomment`

`xmlcomment (text)`

関数`xmlcomment`は特定のテキストを内容としてXMLコメントを含んだXML値を作成します。テキストには結果として構築されるXMLコメントが有効になるよう`--`または`-`で終結する内容を含んではなりません。引数がNULLならば結果もNULLになります。

例：

```
SELECT xmlcomment('hello');

 xmlcomment
-----
<!--hello-->
```

9.14.1.2. `xmlconcat`

`xmlconcat (xml [, ...])`

関数`xmlconcat`は、XMLの内容断片を含む単一値を作成するために個別のXML値のリストを結合します。NULL値は削除され、NULL以外の引数が存在しないときのみ結果はNULLになります。

例：

```
SELECT xmlconcat('<abc/>', '<bar>foo</bar>');

 xmlconcat
-----
<abc/><bar>foo</bar>
```

XML宣言が提示されている場合は次のように組み合わせることができます。全ての引数の値が同一のXMLversion宣言を持っていれば、そのversionが結果に使用されます。さもなければversionは使用されません。もし全ての引数の値がstandalone宣言値である`"yes"`であれば、その値が結果に使用されます。もし全ての引数の値がstandalone宣言値であり、その中で1つでも`"no"`がある場合、それが結果に使用されます。さもなければ、結果はstandalone宣言を持ちません。もし結果がstandalone宣言を必要としていても、standalone宣言がない場合には、XMLはversion宣言を含むXML宣言を要求するため、version 1.0のversion宣言が使用されます。encoding宣言は無視され、全ての場合で削除されます。

例：

```
SELECT xmlconcat('<?xml version="1.1"?><foo/>', '<?xml version="1.1" standalone="no"?><bar/>');

 xmlconcat
-----
<?xml version="1.1"?><foo/><bar/>
```

9.14.1.3. `xmlelement`

`xmlelement (name name [, xmlattributes(value [AS attname] [, ...])] [, content, ...])`

`xmlelement`式は与えられた名前、属性、および内容を持つXML要素を生成します。

例：

```
SELECT xmlelement(name foo);

 xmlelement
```

```

-----
<foo/>

SELECT xmlelement(name foo, xmlattributes('xyz' as bar));

      xmlelement
-----
<foo bar="xyz"/>

SELECT xmlelement(name foo, xmlattributes(current_date as bar), 'cont', 'ent');

      xmlelement
-----
<foo bar="2007-01-26">content</foo>

```

有効でないXML名の要素と属性名は、シーケンス `_xHHHH` により障害となる文字を置換することでエスケープされます。ここで、`HHHH`は16進数によるその文字のUnicode文字コード番号です。例をあげます。

```

SELECT xmlelement(name "foo$bar", xmlattributes('xyz' as "a&b"));

      xmlelement
-----
<foo_x0024_bar a_x0026_b="xyz"/>

```

属性値が列参照の場合、明示的な属性名を指定する必要はありません。この場合、デフォルトで列名が属性名として使用されます。いずれの場合でも、属性は明示的な名前と与えられなければなりません。従って、以下の例は有効です。

```

CREATE TABLE test (a xml, b xml);
SELECT xmlelement(name test, xmlattributes(a, b)) FROM test;

```

しかし、以下の例は有効ではありません。

```

SELECT xmlelement(name test, xmlattributes('constant'), a, b) FROM test;
SELECT xmlelement(name test, xmlattributes(func(a, b))) FROM test;

```

もし要素内容が指定されればそのデータ型に従って書式化されます。もし内容そのものが`xml`型であれば、複合XML文書が構築されます。例をあげます。

```

SELECT xmlelement(name foo, xmlattributes('xyz' as bar),
                  xmlelement(name abc),
                  xmlcomment('test'),
                  xmlelement(name xyz));

      xmlelement
-----
<foo bar="xyz"><abc/><!--test--><xyz/></foo>

```

そのほかの型の内容は有効なXML文字データにフォーマットされます。これは特に文字`<`、`>`、および`&`がエンティティに変換されることを意味します。バイナリデータ（データ型は`bytea`）は、構成パラメータ`xmlbinary`の設定にしたがって、base64もしくは16進符号化方式で表現されます。個々のデータ型に対する特定の動作は、XMLスキーマ仕様でのSQLおよびPostgreSQLデータ型に調整するため展開されると期待されます。この時点でより詳細な記述が出現します。

9.14.1.4. xmlforest

```
xmlforest (content [AS name] [, ...])
```

`xmlforest` 式は与えられた名前と内容を使用し、要素のXMLフォレスト（シーケンス）を生成します。

例：

```

SELECT xmlforest('abc' AS foo, 123 AS bar);

      xmlforest
-----
<foo>abc</foo><bar>123</bar>

```

```

SELECT xmlforest(table_name, column_name)
FROM information_schema.columns
WHERE table_schema = 'pg_catalog';

```

```

      xmlforest
-----
<table_name>pg_authid</table_name><column_name>rolname</column_name>
<table_name>pg_authid</table_name><column_name>rolsuper</column_name>
...

```

第2の例に見られるように、内容の値が列参照の場合要素名は省略可能です。この時は、列名がデフォルトで使用されます。そうでない時は、名前が指定されなければなりません。

有効なXML名ではない要素名は上の`xmlelement`で説明した通りエスケープされます。同様に、既に`xml`型であるものを除き、内容データは有効なXML内容になるようにエスケープされます。

XMLフォレストは1より多い要素からなる場合、有効なXML文書ではないことに注意してください。したがって、`xmlelement`に`xmlforest`式をラップすることが有用ことがあります。

9.14.1.5. `xmlpi`

```
xmlpi(name target [, content])
```

`xmlpi` 式はXML処理命令を作成します。内容が存在すれば、その内容は`<?>`文字シーケンスを含んではなりません。

例：

```
SELECT xmlpi(name php, 'echo "hello world";');

          xmlpi
-----
<?php echo "hello world";?>
```

9.14.1.6. `xmlroot`

```
xmlroot(xml, version text | no value [, standalone yes|no|no value])
```

`xmlroot` 式はXML値のルートノードの属性を変更します。versionが指定されていると、ルートノードのversion宣言での値を変更し、standalone設定が指定されていると、ルートノードのstandalone宣言での値を変更します。

```
SELECT xmlroot(xmlparse(document '<?xml version="1.1"?><content>abc</content>'),
               version '1.0', standalone yes);

          xmlroot
-----
<?xml version="1.0" standalone="yes"?>
<content>abc</content>
```

9.14.1.7. `xmlagg`

```
xmlagg(xml)
```

ここで説明している関数とは異なり、`xmlagg`関数は集約関数です。単一行の要素ではなく、複数行にまたがった連結となりますが、`xmlconcat`が行うように、入力値を集約関数呼び出しに連結します。集約関数についての追加情報は[項9.20](#)を参照してください。

例：

```
CREATE TABLE test (y int, x xml);
INSERT INTO test VALUES (1, '<foo>abc</foo>');
INSERT INTO test VALUES (2, '<bar/>');
SELECT xmlagg(x) FROM test;

          xmlagg
-----
<foo>abc</foo><bar/>
```

連結の順序を決定するため、[項4.2.7](#)に記述されているようにORDER BY句を集計呼び出しに追加すると良いでしょう。以下は例です。

```
SELECT xmlagg(x ORDER BY y DESC) FROM test;

          xmlagg
-----
<bar/><foo>abc</foo>
```

下記は以前のバージョンで推奨されていた、非標準的な方法例です。特定のケースでは有用かもしれませんが。

```
SELECT xmlagg(x) FROM (SELECT * FROM test ORDER BY y DESC) AS tab;

          xmlagg
-----
<bar/><foo>abc</foo>
```

9.14.2. XML述語

この節で記述されている式は、`xml`値の属性をチェックします。

9.14.2.1. IS DOCUMENT

```
xml IS DOCUMENT
```

式 `IS DOCUMENT` は引数XML値が適切なXML文書であれば真を返し、そうでなければ（つまり、内容の断片）偽を返すか、もしくは引数がNULLであればNULLを返します。文書と内容の断片の差異については[項8.13](#)を参照してください。

9.14.2.2. XMLEXISTS

```
XMLEXISTS(text PASSING [BY REF] xml [BY REF])
```

関数 `xmlexists` はXPath式の第一引数が何かしらのノードであれば真を返し、そうでなければ偽を返します。（もしいずれの引数もNULLであった場合はNULLを返します。）

例:

```
SELECT xmlexists('//town[text() = ''Toronto'']' PASSING BY REF
'<towns><town>Toronto</town><town>Ottawa</town></towns>');

 xmlexists
-----
 t
(1 row)
```

`BY REF` 句は、PostgreSQLには何の影響も与えませんが、別製品とのSQL互換性や順応性のため、付与することができます。SQL標準では最初に`BY REF`を必要としており、2つ目の引数以降はオプションです。加えてSQL標準では`xmlexists`はXQueryを第一引数として取る構成としていますが、PostgreSQLでは現在XQueryのサブセットにあたるXPathのみサポートしていることに注意してください。

9.14.2.3. xml_is_well_formed

```
xml_is_well_formed(text)
xml_is_well_formed_document(text)
xml_is_well_formed_content(text)
```

これらの関数は`text`文字列が整形式かどうかをチェックし、論理値で結果を返します。`xml_is_well_formed_document`は文書が整形式かをチェックし、一方`xml_is_well_formed_content`は内容が整形式かをチェックします。

`xml_is_well_formed`は、[xmloption](#)パラメータ値が`DOCUMENT`に設定されていれば前者を、`CONTENT`が設定されていれば後者のチェックを実施します。これは、`xml_is_well_formed`は単純な`xml`型へのキャストが成功するかの判断に有用であり、その他の2つの関数はXMLPARSEの対応による変換が成功するかの判断に有用であることを意味します。

例:

```
SET xmloption TO DOCUMENT;
SELECT xml_is_well_formed('<>');
 xml_is_well_formed
-----
 f
(1 row)

SELECT xml_is_well_formed('<abc/>');
 xml_is_well_formed
-----
 t
(1 row)

SET xmloption TO CONTENT;
SELECT xml_is_well_formed('abc');
 xml_is_well_formed
-----
 t
(1 row)

SELECT xml_is_well_formed_document('<pg:foo xmlns:pg="http://postgresql.org/stuff">bar</pg:foo>');
 xml_is_well_formed_document
-----
 t
(1 row)

SELECT xml_is_well_formed_document('<pg:foo xmlns:pg="http://postgresql.org/stuff">bar</my:foo>');
 xml_is_well_formed_document
-----
 f
(1 row)
```

最後の例は、名前空間が正しく一致しているかのチェックも含むことを示しています。

9.14.3. XMLの処理

データ型`xml`の値を処理するため、PostgreSQLは関数`xpath`と`xpath_exists`を提供しており、それはXPath 1.0式を評価します。

```
xpath(xpath, xml [, nsarray])
```

関数`xpath`は、XML値`xml`に対し、XPath式`xpath`(ひとつの`text`値)を評価します。そして、XPath式で作成されたノードセットに対応するXML値の配列を返します。もし、XPath式がノードセットではなくスカラー値を返す場合、単一要素の配列が返されます。

2番目の引数は整形済XML文書でなければなりません。特に、単一のルートノード要素を持たなければなりません。

オプションな関数の3番目の引数は名前空間マッピング配列です。この配列は、第2軸が2に等しい長さをもつ2次元`text`配列です（つまり、それは配列の配列で、それぞれは正確に2つの要素からなります）。それぞれの配列のエントリの最初の要素は名前空間の名前（別名）で、2番目は名前空間のURIです。この配列内で提供される別名がXML文書自身で 사용되는ものと同じであることは必要ではありません（言い換えると、XML文書内および`xpath`関数の両方の文脈の中で、別名はローカルです）。

例：

```
SELECT xpath('/my:a/text()', '<my:a xmlns:my="http://example.com">test</my:a>',
            ARRAY[ARRAY['my', 'http://example.com']]);

 xpath
-----
 {test}
(1 row)
```

デフォルト(匿名)名前空間を取り扱うためには、以下のようなことを実施してください。

```
SELECT xpath('/mydefns:b/text()', '<a xmlns="http://example.com"><b>test</b></a>',
            ARRAY[ARRAY['mydefns', 'http://example.com']]);

 xpath
-----
 {test}
(1 row)

xpath_exists(xpath, xml [, nsarray])
```

関数`xpath_exists`は、`xpath`関数の特別な形式です。この関数は、XPathを満足する個別のXML値を返す代わりに、問い合わせがそれを満足するかどうかを論理値で返します。この関数は、名前空間にマッピングされた引数をもサポートする点を除き、標準の`XMLEXISTS`述語と同じです。

例：

```
SELECT xpath_exists('/my:a/text()', '<my:a xmlns:my="http://example.com">test</my:a>',
                    ARRAY[ARRAY['my', 'http://example.com']]);

 xpath_exists
-----
 t
(1 row)
```

9.14.4. XMLにテーブルをマップ

以下の関数はリレーショナルテーブルの内容をXML値にマップします。これらはXMLエクスポート機能と考えることができます。

```
table_to_xml(tbl regclass, nulls boolean, tableforest boolean, targetns text)
query_to_xml(query text, nulls boolean, tableforest boolean, targetns text)
cursor_to_xml(cursor refcursor, count int, nulls boolean,
              tableforest boolean, targetns text)
```

それぞれの関数の戻り値型は`xml`です。

`table_to_xml`は、パラメータ`tbl`として渡された名前付きのテーブルの内容をマップします。`regclass`型はオプションのスキーマ修飾と二重引用符を含む、通常の表記法を使用しテーブルを特定する文字列を受け付けます。`query_to_xml`は、パラメータ`query`としてテキストが渡された問い合わせを実行し、結果セットをマップします。`cursor_to_xml`は、パラメータ`cursor`で指定されたカーソルから提示された行数を取得します。この変異形は、それぞれの関数により結果値がメモリーに構築されるため、巨大なテーブルをマップする必要がある場合推奨されます。

`tableforest`が偽であれば、結果のXML文書は以下のようになります。

```
<tablename>
<row>
```

```

      <columnname1>data</columnname1>
      <columnname2>data</columnname2>
    </row>

    <row>
      ...
    </row>

    ...
  </tablename>

```

`tableforest` が真であれば、結果は以下のようなXML文書の断片です。

```

<tablename>
  <columnname1>data</columnname1>
  <columnname2>data</columnname2>
</tablename>

<tablename>
  ...
</tablename>

...

```

テーブル名が1つも有効でなければ、つまり、問い合わせ、またはカーソルをマップする時、文字列 `table` が最初の書式で使われ、`row` が2番目の書式で使われます。

これらの書式を選択するのはユーザ次第です。最初の書式は適切なXML文書で、多くのアプリケーションにおいて重要です。第2の書式は、後に結果値が1つの文書に再び組み立てられる場合、`cursor_to_xml` 関数内でより有用になる傾向があります。上記で説明したXML内容を作成する関数、特に `xmlelement` は結果を好みにかえるために使用することができます。

データの値は上記関数 `xmlelement` で説明したのと同じ方法でマップされます。

パラメータ `nulls` は出力にNULL値が含まれる必要があるかを決定します。もし真であれば列内のNULL値は以下のように表現されます。

```

<columnname xsi:nil="true"/>

```

ここで `xsi` はXMLスキーマインスタンスに対するXML名前空間接頭辞です。適切な名前空間宣言が結果値に追加されます。もし偽の場合、NULL値を含む列は単に出力から削除されます。

パラメータ `targetns` は結果の希望するXML名前空間を指定します。特定の名前空間が必要なければ、空文字列を渡す必要があります。

以下の関数は、対応する上記関数により行われたマッピングを記述するXMLスキーマ文書を返します。

```

table_to_xmlschema(tbl regclass, nulls boolean, tableforest boolean, targetns text)
query_to_xmlschema(query text, nulls boolean, tableforest boolean, targetns text)
cursor_to_xmlschema(cursor refcursor, nulls boolean, tableforest boolean, targetns text)

```

一致するXMLデータマッピングとXMLスキーマ文書を取得するため、同じパラメータが渡されることが不可欠です。

以下の関数は、XMLデータマッピングとそれに対応するXMLスキーマがお互いにリンクされた、1つの文書（またはフォレスト）を作成します。これらは自己完結した、自己記述的な結果を希望する場合に便利です。

```

table_to_xml_and_xmlschema(tbl regclass, nulls boolean, tableforest boolean, targetns text)
query_to_xml_and_xmlschema(query text, nulls boolean, tableforest boolean, targetns text)

```

さらに、以下の関数がスキーマ全体、または現在のデータベース全体の類似マッピングを作成するため利用できます。

```

schema_to_xml(schema name, nulls boolean, tableforest boolean, targetns text)
schema_to_xmlschema(schema name, nulls boolean, tableforest boolean, targetns text)
schema_to_xml_and_xmlschema(schema name, nulls boolean, tableforest boolean, targetns text)

database_to_xml(nulls boolean, tableforest boolean, targetns text)
database_to_xmlschema(nulls boolean, tableforest boolean, targetns text)
database_to_xml_and_xmlschema(nulls boolean, tableforest boolean, targetns text)

```

これらはメモリー内に作成される必要がある、多くのデータを生成する潜在的可能性があることに注意してください。巨大なスキーマ、またはデータベースの内容マッピングを要求する際は、カーソル経由の可能性があっても、その代わりにテーブルを別々にマップすることを検討することは無駄ではありません。

スキーマ内容マッピングの結果は以下のようになります。

```

<schemaname>

table1-mapping

```

```

table2-mapping
...
</schemaname>

```

ここで、テーブルマッピング書式は上で説明したとおり `tableforest` パラメータに依存します。

データベース内容マッピング書式は以下のようになります。

```

<dbname>

<schemaname>
...
</schemaname>

<schema2name>
...
</schema2name>

...

</dbname>

```

ここで、スキーママッピングは上記のとおりです。

これらの関数で作成された出力を使用する 1 つの例として、[図9-1](#)は、テーブルデータの表形式への翻訳を含む `table_to_xml_and_xmlschema` から HTML 文書への出力の変換をおこなう XSLT スタイルシートを示します。同じようにして、これらの関数の結果は他の XML 基準書式に変換されます。

図 9-1. SQL/XML 出力を HTML に変換する XSLT スタイルシート

```

<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://www.w3.org/1999/xhtml"
>

  <xsl:output method="xml"
    doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
    doctype-public="-//W3C/DTD XHTML 1.0 Strict//EN"
    indent="yes"/>

  <xsl:template match="/*">
    <xsl:variable name="schema" select="//xsd:schema"/>
    <xsl:variable name="tabletypename"
      select="$schema/xsd:element[@name=name(current())]/@type"/>
    <xsl:variable name="rowtypename"

select="$schema/xsd:complexType[@name=$tabletypename]/xsd:sequence/xsd:element[@name='row']/@type"/>

    <html>
      <head>
        <title><xsl:value-of select="name(current())"/></title>
      </head>
      <body>
        <table>
          <tr>
            <xsl:for-each select="$schema/xsd:complexType[@name=$rowtypename]/xsd:sequence/xsd:element/@name">
              <th><xsl:value-of select="."/></th>
            </xsl:for-each>
          </tr>

          <xsl:for-each select="row">
            <tr>
              <xsl:for-each select="*">
                <td><xsl:value-of select="."/></td>
              </xsl:for-each>
            </tr>
          </xsl:for-each>
        </table>
      </body>
    </html>
  </xsl:template>

</xsl:stylesheet>

```

9.15. JSON関数

[表9-39](#)に、JSON(詳細は[項8.14](#))データを作成するために利用可能な関数を示します。

表 9-39. JSONサポート関数

関数	説明	例	結果
<code>array_to_json(anyarray [, pretty_bool])</code>	配列をJSON型として返します。PostgreSQLの多次元配列はJSON配列の配列となります。もし <code>pretty_bool</code> が真の場合、次元を1つの要素とし、その間へラインフィードが加えられます。	<code>array_to_json('{{1,5},{99,100}}'::int[])</code>	<code>[[1,5],[99,100]]</code>
<code>row_to_json(record [, pretty_bool])</code>	行をJSON型として消します。もし <code>pretty_bool</code> が真の場合、レベルを1つの要素とし、その間へラインフィードが加えられます。	<code>row_to_json(row(1,'foo'))</code>	<code>{"f1":1,"f2":"foo"}</code>

9.16. シーケンス操作関数

本節ではPostgreSQLのシーケンスオブジェクトに対し演算を行う関数について説明します。シーケンスオブジェクト（シーケンスジェネレータとも単にシーケンスとも呼ばれます）とは[CREATE SEQUENCE](#)で作成される1行の特別なテーブルです。シーケンスオブジェクトは通常テーブルの行に一意的識別子を生成するために使用されます。[表9-40](#)に列挙されているシーケンス関数は、シーケンスオブジェクトから連続したシーケンス値を取得するための、簡易でマルチユーザに対応した関数です。

表 9-40. シーケンス関数

関数	戻り値	説明
<code>currval (regclass)</code>	<code>bigint</code>	指定されたシーケンスに対し <code>nextval</code> で得られた最新の値を返す
<code>lastval ()</code>	<code>bigint</code>	どんなシーケンスに対しても <code>nextval</code> により最も最近取得された値を返す
<code>nextval (regclass)</code>	<code>bigint</code>	シーケンスを進め、新しい値を返す
<code>setval (regclass, bigint)</code>	<code>bigint</code>	シーケンスの現在値を設定する
<code>setval (regclass, bigint, boolean)</code>	<code>bigint</code>	シーケンスの現在値と <code>is_called</code> フラグを設定する

シーケンス関数により操作されるシーケンスは`regclass`引数で指定され、そしてそれは`pg_class`システムカタログ内のシーケンスのOIDに過ぎません。手作業でOIDを検索してはいけませんが、`regclass`データ型の入力変換器が代わってその作業を行ってくれます。リテラル関数のように見えるようにするため、単一引用符で括られたシーケンス名を記述するだけです。通常のSQLの名称での操作との互換のため、文字列はシーケンス名が二重引用符で括られている以外、小文字に変換されます。よって、以下ようになります。

```
nextval('foo')           operates on sequence foo
nextval('FOO')           operates on sequence foo
nextval('"Foo"')         operates on sequence Foo
```

必要であれば、以下のようにシーケンス名をスキーマで修飾することができます。

```
nextval('myschema.foo')   operates on myschema.foo
nextval('"myschema".foo') same as above
nextval('foo')            searches search path for foo
```

`regclass`に関してのより詳細な情報は[項8.18](#)を参照してください。

注意: PostgreSQL 8.1以前においては、シーケンス関数の引数は`regclass`型ではなく、`text`型で、そして上記のテキスト文字列からOID値への変換はそれぞれの呼び出し実行時に起こりました。後方互換性のため、この仕組みはまだ存在しますが、内部的には関数が実行される前に`text`から`regclass`への暗黙的強制型変換として現在処理されています。

ありのままのリテラル文字列としてシーケンス関数の引数を記述する時は、`regclass`データ型の定数になります。これは単にOIDなので、後で名前付けが再び行われたとか、スキーマの再割り振りとかに係わず、最初に特定されたシーケンスを引き継ぎます。この"初期束縛"の動作は、通常列のデフォルトとビュー参照するシーケンスにとって魅力があります。しかし、たまには実行時にシーケンス参照が解決されるような"動的束縛"が望まれます。動的束縛の動作を得るには、定数を`regclass`ではなく`text`定数としてその定数を保存させます。

```
nextval('foo'::text)      foo is looked up at runtime
```

動的束縛はPostgreSQLのリリース8.1以前でサポートされた動作であったので、旧来のアプリケーションのセマンティクスを保ちたい場合このようにする必要があるかもしれません。

もちろん、シーケンス関数の引数は定数とともに、評価式であることも可能です。もしテキスト式の場合は暗黙的強制型変換が実行時検索に用いられます。

使用できるシーケンス関数には以下のものがあります。

`nextval`

シーケンスオブジェクトをその次の値に進め、その値を返します。これは自動的に処理されます。複数のセッションが同時に`nextval`を実行したとしても、それぞれのセッションは個別のシーケンス値を間違いなく受け取ります。

デフォルトのパラメータによってシーケンスオブジェクトが作成されているなら、引き続き`nextval`呼び出しは1から始まる連続的な値を返します。その他の動作を[CREATE SEQUENCE](#)コマンドの特別なパラメータを使用してさせることが可能です。より詳しくはコマンドのリファレンスガイドを参照ください。

重要項目: 同一のシーケンスから数値を取得する同時実行トランザクション同士のブロックを防止するため、`nextval` 演算は決してロールバックされません。 言うことは、たとえ`nextval` を実行したトランザクションが後にアボートしたとしても、値が一度取り出されたらそれは使用されたものと考えます。 つまり、アボートされたトランザクションは、割り当てられた値のシーケンス内に未使用の"欠損"を残す可能性があります。

`currval`

現在のセッションにおいて、そのシーケンスから`nextval`によって取得された直近の値を返します（セッション内でそのシーケンスに対し`nextval`が呼ばれていない場合には、エラーが報告されます）。これはローカルのセッション値を返すことから、現在のセッションが実行してから別のセッションが`nextval`を実行してもしくても、予想に違わない回答をもたらします。

`lastval`

現在のセッションの`nextval`で戻される最新の値を返します。この関数は、現在のセッションの中で`nextval`によって使用される最後のシーケンスの値をフェッチする引数としてのシーケンス名を取ることを除いて、`currval`と同等です。現在のセッションで`nextval`が未だ呼ばれていなければエラーになります。

`setval`

シーケンスオブジェクトの計数値をリセットします。2つのパラメータを所有する形式では、シーケンスの`last_value`フィールドを指定された値に設定し、`is_called`フィールドを`true`（真）に設定します。この意味は、次の`nextval`が値を返す前にシーケンスを進めるということです。`currval`で報告された値も指定された値に設定されます。3パラメータ形式の場合、`is_called`を`true`（真）もしくは`false`（偽）に設定することができます。`true`（真）は2パラメータ形式と同じ効果があります。`false`（偽）に設定された場合、次の`nextval`が指定された正確な値を返し、シーケンスの進行は引き続き`nextval`から始まります。さらにこの場合（これは8.3以前の動作からの変更）、`currval`で報告された値は変更されません。例えば、次の例です。

```
SELECT setval('foo', 42);           Next nextval will return 43
SELECT setval('foo', 42, true);      Same as above
SELECT setval('foo', 42, false);     Next nextval will return 42
```

`setval`によって返される結果は単にその第2番目の引数の値です。

重要項目: シーケンスは非トランザクショナルに扱われるため、`setval`による変更は、そのトランザクションがロールバックされたとしても 元に戻りません。

9.17. 条件式

本節ではPostgreSQLで使用可能なSQL準拠の条件式について説明します。

ティップ: ここで説明する条件式より発展した機能を求める場合は、より表現の豊富なプログラム言語でストアドプロシージャを記述することで解決されます。

9.17.1. CASE

SQLのCASE式は他のプログラミング言語のif/else構文に類似した汎用条件式です。

```
CASE WHEN condition THEN result
      [WHEN ...]
      [ELSE result]
END
```

CASE 句は式が有効な位置であればどこでも使用可能です。それぞれの`condition`とはboolean型の結果を返す式です。もしconditionの結果が真であれば、CASE式の値は、conditionに続く`result`となります。そして、CASE式の残りは処理されません。もしconditionの結果が偽であれば後に続く全てのWHEN句が同じようにして調べられます。WHENのconditionの1つも真でない場合、CASE式の値はELSE句の`result`になります。ELSE句が省略され、どのconditionも真でない場合、結果はNULLです。

以下に例を示します。

```
SELECT * FROM test;

 a
---
 1
 2
 3

SELECT a,
       CASE WHEN a=1 THEN 'one'
            WHEN a=2 THEN 'two'
            ELSE 'other'
       END
FROM test;

 a | case
---+-----
 1 | one
 2 | two
 3 | other
```

全ての`result`式のデータ型は単一の出力型に互換性がなければなりません。詳細は[項10.5](#)を参照してください。

以下のように、上記の一般的な形式と異なるCASE式の"単純な"形式が存在します。

```
CASE expression
     WHEN value THEN result
     [WHEN ...]
     [ELSE result]
END
```

最初の`expression`は計算され、そしてそれに等しいものが見つかるまでWHEN句のそれぞれの`value`式と比較されます。等しいものが見つからない場合、ELSE句の`result`（もしくはNULL値）が返されます。これはC言語のswitch文に似ています。

上の例は簡略形CASE構文を使って次のように書くことができます。

```
SELECT a,
       CASE a WHEN 1 THEN 'one'
            WHEN 2 THEN 'two'
            ELSE 'other'
       END
FROM test;

 a | case
---+-----
 1 | one
 2 | two
 3 | other
```

CASE式は、結果を決定するために不必要などんな副式をも評価しません。例えば、以下は0除算エラーを防ぐための方法です。


```
SELECT ... WHERE CASE WHEN x <> 0 THEN y/x > 1.5 ELSE false END;
```

9.17.2. COALESCE

```
COALESCE(value [, ...])
```

COALESCE関数は、NULLでない自身の最初の引数を返します。全ての引数がNULLの場合にのみNULLが返されます。データを表示目的で取り出す際、NULL値をデフォルト値で置き換えるためによく使用されています。以下に例を示します。

```
SELECT COALESCE(description, short_description, '(none)') ...
```

これは`description`がNULLでなければをそれ返します。そうでない場合(NULLの場合は)、`short_description`がNULLでなければそれを返します。そうでもない場合は`(none)`が帰ります。

CASE式同様、COALESCEは結果を決定するために必要な引数のみを評価します。つまり、非NULL引数が見つければ、その右側にある引数は評価されません。このSQL標準関数はNVLとIFNULLと類似の機能を提供し、他のいくつかのデータベースシステムで使用されています。

9.17.3. NULLIF

```
NULLIF(value1, value2)
```

NULLIF関数は、*value1*が*value2*と等しい場合、NULL値を返します。 その他の場合は*value1*を返します。 これを使って、上記のCOALESCEの例の逆演算を実行できます

```
SELECT NULLIF(value, '(none)') ...
```

この例では、*value1*が`(none)`ならばNULLが返ります。 さもなくば*value1*を返します

9.17.4. GREATESTおよびLEAST

```
GREATEST(value [, ...])
```

```
LEAST(value [, ...])
```

GREATESTと**LEAST**関数は式の任意の数のリストから最大値もしくは最小値を選択します。評価される全ての式は、結果として得られるデータの型と共通の型に変換できなくてはなりません（詳細は[項10.5](#)を参照してください）。リストの中のNULL値は無視されます。全ての式がNULLと評価された場合に限り結果はNULLになります。

GREATESTおよび**LEAST**はSQL標準に載っていませんが、共通した拡張です。他のいくつかのデータベースでは、全てがNULLの場合に限定せず、いずれかの引数がNULLである場合にNULLを返すようにしているものもあります。

[前のページ](#)

シーケンス操作関数

[ホーム](#)

[上に戻る](#)

[次のページ](#)

配列関数と演算子

9.18. 配列関数と演算子

[表9-41](#)に、配列型で使用可能な演算子を示します。

表 9-41. 配列演算子

演算子	説明	例	結果
=	等しい	ARRAY[1.1,2.1,3.1)::int[] = ARRAY[1,2,3]	t
<>	等しくない	ARRAY[1,2,3] <> ARRAY[1,2,4]	t
<	未満	ARRAY[1,2,3] < ARRAY[1,2,4]	t
>	より大きい	ARRAY[1,4,3] > ARRAY[1,2,4]	t
<=	以下	ARRAY[1,2,3] <= ARRAY[1,2,3]	t
>=	以上	ARRAY[1,4,3] >= ARRAY[1,4,3]	t
@>	包含する	ARRAY[1,4,3] @> ARRAY[3,1]	t
<@	・・・により包含される	ARRAY[2,7] <@ ARRAY[1,7,4,2,6]	t
&&	重複する(共通要素を持つ)	ARRAY[1,4,3] && ARRAY[2,1]	t
	配列と配列を連結	ARRAY[1,2,3] ARRAY[4,5,6]	{1,2,3,4,5,6}
	配列と配列を連結	ARRAY[1,2,3] ARRAY[[4,5,6],[7,8,9]]	{{1,2,3},{4,5,6},{7,8,9}}
	要素と配列を連結	3 ARRAY[4,5,6]	{3,4,5,6}
	配列と要素を連結	ARRAY[4,5,6] 7	{4,5,6,7}

配列比較では、配列要素ごとに、要素のデータ型のデフォルトのB-tree比較関数を使用して、その内容が比較されます。多次元配列では、行番号を優先して取り出します（最後の添え字が最も最初になります）。2つの配列で内容が同じで要素数が異なる場合、どの次元で最初に違いがあったかによってソート順が変わります。（これは8.2より前のPostgreSQLでは異なります。古いバージョンでは、回数や添え字範囲が異なっていたとしても、内容が同じであれば、2つの配列は同じものとしていました。）

配列演算子の振舞いの詳細は[項8.15](#)を参照してください。

[表9-42](#)に配列型で使用可能な関数を示します。これらの関数の情報と例については[項8.15](#)を参照してください。

表 9-42. 配列関数

関数	戻り値型	説明	例	結果
array_append(anyarray, anyelement)	anyarray	配列の末尾に要素を追加	array_append(ARRAY[1,2], 3)	{1,2,3}
array_cat(anyarray, anyarray)	anyarray	2つの配列を連結	array_cat(ARRAY[1,2,3], ARRAY[4,5])	{1,2,3,4,5}
array_ndims(anyarray)	int	その配列の次数を返す	array_ndims(ARRAY[[1,2,3],[4,5,6]])	2
array_dims(anyarray)	text	配列の次数をテキスト表現で返す	array_dims(ARRAY[[1,2,3],[4,5,6]])	[1:2][1:3]
array_fill(anyelement, int[], [, int[]])	anyarray	提供された値と次数で初期化された配列を返す。1以外の下限を持たせることもできます	array_fill(7, ARRAY[3], ARRAY[2])	[2:4]={7,7,7}
array_length(anyarray, int)	int	入力された配列次元の長さを返す	array_length(array[1,2,3], 1)	3
array_lower(anyarray, int)	int	配列次元の下限を返す	array_lower('[0:2]={1,2,3}':int[], 1)	0
array_prepend(anyelement, anyarray)	anyarray	配列の先頭に要素を追加	array_prepend(1, ARRAY[2,3])	{1,2,3}
array_to_string(anyarray, text [, text])	text	配列の要素を提供された区切り文字かオプションなNULL文字を使用して連結	array_to_string(ARRAY[1, 2, 3, NULL, _5], ', ', '*')	1,2,3,*,5
		入力された配列の次元の上		

<code>array_upper (anyarray, int)</code>	int	限を返す	<code>array_upper (ARRAY[1,8,3,7], 1)</code>	4
<code>string_to_array (text, text [, text])</code>	text[]	提供された区切り文字、およびオプションなNULL文字を使用して、文字列を配列の要素に分割	<code>string_to_array ('xx~^~yy~^~zz', '~^~','- 'y')</code>	{xx,NULL,zz}
<code>unnest (anyarray)</code>	setof anyelement	配列を行集合に展開	<code>unnest (ARRAY[1,2])</code>	1 2 (2 rows)

`string_to_array`では、もし区切り文字がNULLの場合、入力された文字列の各々の文字が分割され要素となった配列を返します。区切り文字が空白文字の場合、入力された文字列全体が一つの要素となる配列を返します。そうでなければ、入力された文字列が区切り文字で分割されます。

`string_to_array`では、NULL文字パラメータが省略、もしくはNULLの指定がされた場合、入力された部分文字列がNULLに変換されることはありません。`array_to_string`では、NULL文字パラメータが省略、もしくはNULLの指定がされた場合、すべてのNULL文字の処理がスキップされて出力文字列に現れることはありません。

注意: `string_to_array`は、PostgreSQL9.1から、前のバージョンとは2つの異なる振る舞いするようになりました。1つ目は、入力した文字列長が0の場合、NULLを返すのではなく空の(要素が0の)配列を返すようになりました。2つ目は区切り文字がNULLの場合、以前はNULLを返していましたが9.1からは入力文字列を個別の文字列で分割するようになりました。

配列を伴った集約関数の使用法は、[項9.20](#)も参照してください。

9.19. 範囲関数と演算子

範囲型の概要に関しては[項8.17](#)を参照してください。

範囲型に対して利用可能な演算子を[表9-43](#)に示します。

表 9-43. 範囲演算子

演算子	説明	例	結果
=	等しい	<code>int4range(1,5) = '[1,4]':::int4range</code>	t
<>	等しくない	<code>numrange(1.1,2.2) <> numrange(1.1,2.3)</code>	t
<	未満	<code>int4range(1,10) < int4range(2,3)</code>	t
>	より大きい	<code>int4range(1,10) > int4range(1,5)</code>	t
<=	以下	<code>numrange(1.1,2.2) <= numrange(1.1,2.2)</code>	t
>=	以上	<code>numrange(1.1,2.2) >= numrange(1.1,2.0)</code>	t
@>	範囲を包含する	<code>int4range(2,4) @> int4range(2,3)</code>	t
@>	要素を包含する	<code>'[2011-01-01,2011-03-01]':::tsrange @> '2011-01-10':::timestamp</code>	t
<@	・・・により範囲が包含される	<code>int4range(2,4) <@ int4range(1,7)</code>	t
<@	・・・により要素が包含される	<code>42 <@ int4range(1,7)</code>	f
&&	重複する(共通点を持つ)	<code>int8range(3,7) && int8range(4,12)</code>	t
<<	厳密に左に位置する	<code>int8range(1,10) << int8range(100,110)</code>	t
>>	厳密に右に位置する	<code>int8range(50,60) >> int8range(20,30)</code>	t
&<	右側を越えない	<code>int8range(1,20) &< int8range(18,20)</code>	t
&>	左側を越えない	<code>int8range(7,20) &> int8range(5,10)</code>	t
- -	隣接	<code>numrange(1.1,2.2) - - numrange(2.2,3.3)</code>	t
+	結合範囲	<code>numrange(5,15) + numrange(10,20)</code>	[5,20)
*	交差範囲	<code>int8range(5,15) * int8range(10,20)</code>	[10,15)
-	差分範囲	<code>int8range(5,15) - int8range(10,20)</code>	[5,10)

単純な比較演算子である<、>、<=そして>=は、最初に下限の境界を比較し、それらが等しい場合のみ上限の境界を比較する。これらの比較演算子は、範囲型のデータに関しては大抵便利なものではありませんが、範囲型へBtreeインデックスを付与するのに必要なため提供されています。

左に位置する/右に位置する/隣接の演算子は、空の範囲が呼ばれた際は常に偽を返します。つまり、空の範囲に対しては、他のどの範囲の前後のどちらであるかは考慮されないということです。

結合範囲と差分範囲演算子は、もし得られる結果が互いに離れた2つの範囲を含む必要がある場合、このような範囲を表現することができないため失敗するでしょう。

範囲型に対して利用可能な関数を[表9-44](#)に示します。

表 9-44. 範囲関数

関数	戻り値型	説明	例	結果
<code>lower(anyrange)</code>	範囲の要素の型	範囲の下限	<code>lower(numrange(1.1,2.2))</code>	1.1
<code>upper(anyrange)</code>	範囲の要素の型	範囲の上限	<code>upper(numrange(1.1,2.2))</code>	2.2
<code>isempty(anyrange)</code>	boolean	空の範囲か？	<code>isempty(numrange(1.1,2.2))</code>	false
<code>lower_inc(anyrange)</code>	boolean	下限は内包されているか？	<code>lower_inc(numrange(1.1,2.2))</code>	true
<code>upper_inc(anyrange)</code>	boolean	上限は内包されているか？	<code>upper_inc(numrange(1.1,2.2))</code>	false
<code>lower_inf(anyrange)</code>	boolean	下限は無限大か？	<code>lower_inf('(',')':::daterange)</code>	true

upper_inf(anyrange)	boolean	上限は無限大か？	upper_inf('(',')'::daterange)	true

lower と upper 関数は、もし範囲が空か対象の境界が無限大だった場合、NULLを返します。
lower_inc、upper_inc、lower_inf、そしてupper_inf関数は、空の範囲に対しては全て偽を返します。

[前のページ](#)
配列関数と演算子

[ホーム](#)
[上に戻る](#)

[次のページ](#)
集約関数

9.20. 集約関数

集約関数は複数の入力値から単一の結果を計算します。 [表9-45](#)および[表9-46](#)に組み込み集約関数を示します。 集約関数の特殊な構文に関する考察は[項4.2.7](#)で説明されています。また、初歩的な情報については[項2.7](#)を参照して下さい。

表 9-45. 汎用集約関数

関数	引数のデータ型	戻り値型	説明
<code>array_agg(expression)</code>	any	引数型の配列	配列に連結されたNULLを含む入力値
<code>avg(expression)</code>	smallint, int, bigint, real, double precision, numeric, or interval	整数型の引数であれば全てnumeric、浮動小数点の引数であればdouble precision、それ以外は引数のデータ型と同じ	全ての入力値の平均値（算術平均）
<code>bit_and(expression)</code>	smallint, int, bigint, or bit	引数のデータ型と同じ	全ての非NULLの入力値のビット積、非NULLの入力値がなければNULL
<code>bit_or(expression)</code>	smallint、int、bigint、または bit	引数のデータ型と同じ	全ての非NULLの入力値のビット和、非NULLの入力値がなければNULL
<code>bool_and(expression)</code>	bool	bool	全ての入力が真ならば真、そうでなければ偽
<code>bool_or(expression)</code>	bool	bool	少なくとも1つの入力値が真ならば真。そうでなければ偽
<code>count(*)</code>		bigint	入力行の数
<code>count(expression)</code>	全て	bigint	<code>expression</code> が非NULL値を持つ入力行の個数
<code>every(expression)</code>	bool	bool	<code>bool_and</code> と等価
<code>max(expression)</code>	全ての配列、数値、文字列、または日付時刻型	引数の型と同じ	全ての入力値にわたり <code>expression</code> の最大値
<code>min(expression)</code>	全ての配列、数値、文字列、または日付時刻型	引数の型と同じ	全ての入力値にわたり <code>expression</code> の最小値
<code>string_agg(expression, delimiter)</code>	(text, text)または(bytea, bytea)	引数と同じ型	入力された値が指定したデリミタで区切られた一つの文字列に連結されます。
<code>sum(expression)</code>	smallint、int、bigint、real、double precision、numeric、またはinterval	smallintまたはint型の引数であればbigint、bigint型の引数であればnumeric、浮動小数点の引数であればdouble precision、それ以外は引数のデータ型と同じ	全ての入力値に渡り <code>expression</code> の和
<code>xmlagg(expression)</code>	xml	xml	XML値の連結（ 項9.14.1.7 も参

上記の関数は、count関数を除き、1行も選択されなかった場合NULL値を返すことに注意してください。特に、行の選択がないsum関数は、予想されるであろうゼロではなくNULLを返し、そしてarray_aggは、入力行が存在しない場合に、空配列ではなくNULLを返します。必要であれば、NULLをゼロまたは空配列と交換する目的でcoalesce関数を使うことができます。

注意: bool_and、bool_or論理集約関数は標準SQLの集約関数every、anyまたはsomeに対応します。anyとsomeについてですが、標準の構文には曖昧さがあるようです。

```
SELECT b1 = ANY((SELECT b2 FROM t2 ...)) FROM t1 ...;
```

ここで、副問い合わせが論理値での1行を返す場合、ANYは副問い合わせを導入するもの、もしくは集約関数であるもののいずれかとみなすことができます。従って、これらの集約関数に標準の名前を付けることはできません。

注意: 他のSQLデータベース管理システムでの作業に親しんだユーザは、count集約関数がテーブル全体に適用される場合の性能に失望するかも知れません。

```
SELECT count(*) FROM sometable;
```

のような問い合わせはテーブルサイズに比例した労力が必要です。PostgreSQLはテーブル全体か、そのテーブルの全ての行を含んだインデックス全体のスキャンを必要とします。

集約関数array_aggとstring_agg、およびxmlagg、そして類似のユーザ定義の集約関数は、入力値の順序に依存した意味のある別の結果値を生成します。この並び順はデフォルトでは指定されませんが、[項4.2.7](#)に記述されているように、集計呼び出し中にORDER BY句を書くことで制御可能となります。別の方法として、並び替えられた副問い合わせから入力値を供給することでも上手くいきます。例をあげます。

```
SELECT xmlagg(x) FROM (SELECT x FROM test ORDER BY y DESC) AS tab;
```

しかしこの構文はSQL標準では許されておらず、他のデータベースシステムに移植性はありません。

統計解析処理によく使用される集約関数を[表9-46](#)に示します。（これらは、より一般的に使用される集約関数との混乱を防ぐために別出ししました。）説明の部分における N は、すべての入力式が非NULLの入力行の個数を表します。すべての場合に、例えば N が0の時など計算が無意味である場合にはNULLが返されます。

表 9-46. 統計処理用の集約関数

関数	引数の型	戻り値の型	説明
corr(Y , X)	double precision	double precision	相関係数
covar_pop(Y , X)	double precision	double precision	母共分散
covar_samp(Y , X)	double precision	double precision	標本共分散
regr_avgx(Y , X)	double precision	double precision	独立変数の平均値 ($\text{sum}(X)/N$)
regr_avgy(Y , X)	double precision	double precision	依存変数の平均値 ($\text{sum}(Y)/N$)
regr_count(Y , X)	double precision	bigint	両式が非NULLとなる入力行の個数
regr_intercept(Y , X)	double precision	double precision	(x , y)の組み合わせで決まる、線型方程式に対する最小二乗法の Y 切片
regr_r2(Y , X)	double precision	double precision	相関係数自乗値
regr_slope(Y , X)	double precision	double precision	x , y)の組み合わせで決まる、最小自乗法に合う線型方程式の傾き
regr_sxx(Y , X)	double precision	double precision	$\text{sum}(X^2) - \text{sum}(X)^2/N$ (依存変数の"二乗和")
regr_sxy(Y , X)	double precision	double precision	$\text{sum}(X*Y) - \text{sum}(X) * \text{sum}(Y)/N$ (依存変数×独立変数の"和")
regr_syy(Y , X)	double precision	double precision	$\text{sum}(Y^2) - \text{sum}(Y)^2/N$ (独立変数の"自乗和")
stddev(<i>expression</i>)	smallint、int、bigint、real、double precision、またはnumeric	浮動小数点型の引数ではdouble precision。それ以外ではnumeric	stddev_sampの歴史的な別名
	smallint、int、	浮動小数点型の引数ではdouble	

<code>stddev_pop(<i>expression</i>)</code>	bigint、real、double precision、またはnumeric	precision。それ以外ではnumeric	入力値に対する母標準偏差
<code>stddev_samp(<i>expression</i>)</code>	smallint、int、 bigint、real、double precision、またはnumeric	浮動小数点型の引数ではdouble precision。それ以外ではnumeric	入力値に対する標本標準偏差
<code>variance(<i>expression</i>)</code>	smallint、int、 bigint、real、double precision、またはnumeric	浮動小数点型の引数ではdouble precision。それ以外ではnumeric	<code>var_samp</code> の歴史的な別名
<code>var_pop(<i>expression</i>)</code>	smallint、int、 bigint、real、double precision、またはnumeric	浮動小数点型の引数ではdouble precision。それ以外ではnumeric	入力値に対する母分散（母標準偏差の自乗）
<code>var_samp(<i>expression</i>)</code>	smallint、int、 bigint、real、double precision、またはnumeric	浮動小数点型の引数ではdouble precision。それ以外ではnumeric	入力値に対する標本分散（標本標準偏差の二乗）

9.21. ウィンドウ関数

ウィンドウ関数は現在の問い合わせ行に関連した行集合に渡っての計算処理機能を提供します。この機能の手引きは[項3.5](#)を参照してください。

組み込みウィンドウ関数は[表9-47](#)に一覧されています。これらの関数は必ずウィンドウ関数構文で呼び出されなければなりません。つまり、`OVER`句が必要です。

これら関数に加え、どんな組み込み、またはユーザ定義の集約関数もウィンドウ関数として使用できます（組み込み集約関数一覧は[項9.20](#)を参照）。集約関数は、呼び出しの後`OVER`句が続いた場合のみウィンドウ関数として動作します。それ以外、通常の集約関数として動作します。

表 9-47. 汎用ウィンドウ関数

関数	戻り値	説明
<code>row_number()</code>	<code>bigint</code>	1から数えたパーティション内の現在行の数
<code>rank()</code>	<code>bigint</code>	ギャップを含んだ現在行の階層で、その最初の（対となる）ピアの <code>row_number</code> と同一
<code>dense_rank()</code>	<code>bigint</code>	ギャップを含まない現在行の階層で、この関数は（対となる）ピアグループ数を計算する
<code>percent_rank()</code>	<code>double precision</code>	現在行の相対順位。 $(rank - 1) / (\text{総行数} - 1)$
<code>cume_dist()</code>	<code>double precision</code>	現在行の相対順位。 (先行する行または現在の行を持つピアの番号) / (total rows)
<code>ntile(num_buckets integer)</code>	<code>integer</code>	できるだけ等価にパーティションで割り算した、1から引数値までの整数
<code>lag(value any [, offset integer [, default any]])</code>	<code>value</code> と同じ型	パーティション内の現在行以前の <code>offset</code> 行である行で評価された <code>value</code> を返します。該当する行がない場合、その代わりとして <code>default</code> を返します。 <code>offset</code> と <code>default</code> は共に現在行について評価されます。省略された場合、 <code>offset</code> は1となり、 <code>default</code> はNULLになります。
<code>lead(value any [, offset integer [, default any]])</code>	<code>value</code> と同じ型	パーティション内の現在行以降の <code>offset</code> 行である行で評価された <code>value</code> を返します。該当する行がない場合、その代わりとして <code>default</code> を返します。 <code>offset</code> と <code>default</code> は共に現在行について評価されます。省略された場合、 <code>offset</code> は1となり、 <code>default</code> はNULLになります。
<code>first_value(value any)</code>	<code>value</code> と同じ型	ウィンドウフレームの最初の行である行で評価された <code>value</code> を返す
<code>last_value(value any)</code>	<code>value</code> と同じ型	ウィンドウフレームの最後の行である行で評価された <code>value</code> を返す
<code>nth_value(value any, nth integer)</code>	<code>value</code> と同じ型	ウィンドウフレームの（1から数えて） <code>nth</code> 番目の行である行で評価された <code>value</code> を返す。行が存在しない場合はNULLを返す

[表9-47](#)に列挙された関数はすべて、対応するウィンドウ定義の`ORDER BY`句で指定されるソート順に依存します。`ORDER BY`順で重複する行はピアと呼ばれます。4つの順位付け関数は、任意の2つのピア間で同じ答えになるように定義されています。

`first_value`、`last_value`、`nth_value`関数は"ウィンドウフレーム"内の行のみを考慮することに注意してください。デフォルトで、ウィンドウフレームにはパーティションの先頭から現在の行の最終ピアまでの行が含まれます。これは`last_value`、または時々`nth_value`では有用ではない結果を得ることになりがちです。`OVER`句に適切なフレーム指定(`RANGE`もしくは`ROWS`)を加えることで、フレームを再定義することができます。フレーム指定についての詳細は[項4.2.8](#)を参照してください。

集約関数をウィンドウ関数として使用する場合、現在の行のウィンドウフレーム内の行に渡って集約処理を行います。`ORDER BY`を付けた集約、および、デフォルトのウィンドウフレーム定義では、"中間和"のような動作を行います。これが望まれる場合もあれば、望まれない場合もあります。パーティション全体に渡る集約処理を行うためには、`ORDER BY`を省略するか`ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING`を使用してください。他のフレーム指定を使用することで様々な結果を得ることができます。

注意: SQL標準は、`lead`、`lag`、`first_value`、`last_value`、および`nth_value`に対し`RESPECT NULLS`、または`IGNORE NULLS`オプションを定義します。これはPostgreSQLに実装されていません。動作は常に標準のデフォルトと同一です。つまり、`RESPECT NULLS`です。同様に、標準の`nth_value`に対する`FROM FIRST`、または`FROM LAST`オプションは実装されていません。デフォルトの`FROM FIRST`動作のみに対応しています。（`ORDER BY`順序付けを逆に行うことで、`FROM LAST`の結果を得ることができます。）

9.22. 副問い合わせ式

本節ではPostgreSQLで使えるSQL準拠の副問い合わせについて説明します。本節で記載した全ての式は結果として論理値（真/偽）を返します。

9.22.1. EXISTS

`EXISTS (subquery)`

EXISTSの引数は、任意のSELECT文または副問い合わせです。副問い合わせはそれが何らかの行を返すか否かの決定のために評価されます。もし1つでも行を返すのであれば、EXISTSの結果は"true（真）"となり、副問い合わせが行を返さない場合、EXISTSの結果は"false（偽）"となります。

副問い合わせは、取り囲んでいる問い合わせから変数を参照することができ、その値は副問い合わせの評価時には定数として扱われます。

この副問い合わせは通常、最後まで実行されず、少なくとも1つの行が返されたかどうかを判定し得るに足りる時点まで実行されます。（シーケンス関数を呼び出すような）副作用のある副問い合わせを記述することは配慮不足です。副作用が生じるかどうかは予想できません。

結果は何らかの行が返されるのかのみに依存し、それらの行の内容には依存しないことから、副問い合わせの出力リストは通常重要ではありません。よく使われるコーディング規約は、全てのEXISTSテストをEXISTS(SELECT 1 WHERE ...)といった形式で記述することです。とは言っても、INTERSECTを使う副問い合わせのようにこの規則には例外があります。

以下の簡単な例はcol2上の内部結合に似ていますが、しかしたとえtab2の行といくつか一致したとしてもtab1のそれぞれの行に対して最大限1つの出力行を生成します。

```
SELECT col1
FROM tab1
WHERE EXISTS (SELECT 1 FROM tab2 WHERE col2 = tab1.col2);
```

9.22.2. IN

`expression IN (subquery)`

右辺は括弧で括られた副問い合わせで、正確に1列を返すものでなければなりません。左辺式は評価され、副問い合わせの結果行と比較されます。副問い合わせの行のどれかと等しい場合、INの結果は"true（真）"です。（副問い合わせが行を返さない場合を含め）等しい行が見つからない場合、結果は"false（偽）"です。

左辺の式がNULLを生じる場合、または右辺の値に等しいものがなくて少なくとも1つの右辺の行がNULLを持つ場合、IN構文の結果は偽ではなくNULLとなることに注意してください。これは、NULL値の論理的な組み合わせに対するSQLの標準規則に従うものです。

EXISTSと同様、副問い合わせが完全に評価されると前提してはなりません。

`row_constructor IN (subquery)`

INのこの形式の左辺は、[項4.2.13](#)で説明する、行のコンストラクタです。右辺は括弧で括られた副問い合わせで、左辺の行にある式の数と正確に同じ数の列を返さなければなりません。左辺の式は副問い合わせの結果のそれぞれの行に対し、行に関して評価、比較が行われます。副問い合わせの行に等しいものが見つかった場合、INの結果は"true（真）"となります。（副問い合わせが行を返さない場合を含め）等しい行が見つからない場合、結果は"false（偽）"です。

行にあるNULL値はいつもSQLの論理式の標準規則で結合されます。2つの行は対応する全ての構成要素が非NULLかつ等しい場合に等しいとみなされます。1つでも対応する構成要素が非NULLかつ等しくないものがあれば、2つの行は等しくないとみなされます。それ以外その行の比較結果は不明（NULL）です。行毎の結果すべてが不等もしくはNULLの場合、少なくとも1つのNULLがあると、INの結果はNULLとなります。

9.22.3. NOT IN

`expression NOT IN (subquery)`

右辺は括弧で括られた副問い合わせで、正確に1つの列を返さなければなりません。左辺の式は副問い合わせ結果の行それぞれに対して評価、比較されます。（副問い合わせが行を返さない場合を含む）等しくない副問い合わせの行だけがあると、NOT INの結果は"true（真）"です。等しい行が1つでもあれば、結果は"false（偽）"です。

左辺の式でNULLが生じる場合、または右辺の値に等しいものがなく、少なくとも1つの右辺の式がNULLを生み出す場合、`NOT IN`構文の結果は真ではなくNULLとなることに注意してください。これは、NULL値の論理的な組み合わせに対するSQLの標準規則に従うものです。

`EXISTS`と同様、副問い合わせが完全に評価されると前提してはなりません。

```
row_constructor NOT IN (subquery)
```

`NOT IN`のこの形式の左辺は、[項4.2.13](#)で説明する行コンストラクタです。右辺は括弧で括られた副問い合わせで、左辺の行にある式の数と正確に同じ数の列を返さなければなりません。左辺の式は副問い合わせの結果のそれぞれの行に対し、評価、比較が行われます。（副問い合わせが行を返さない場合を含め）副問い合わせの行に不等のもののみが見つかった場合、`NOT IN`の結果は"true（真）"となります。等しい行が1つでも見つかった場合、結果は"false（偽）"です。

行にあるNULL値はいつもSQLの論理式の標準規則で結合されます。2つの行は対応する全ての構成要素が非NULLかつ等しい場合に等しいとみなされます。1つでも構成要素が非NULLかつ等しくない場合、2つの行は等しくないとみなされます。それ以外その行の比較結果は不明（NULL）です。行毎の結果すべてが不等もしくはNULLの場合、少なくとも1つのNULLがあると、`NOT IN`の結果はNULLとなります。

9.22.4. ANY / SOME

```
expression operator ANY (subquery)
expression operator SOME (subquery)
```

右辺は括弧で括られた副問い合わせで、正確に1つの列を返さなければなりません。左辺の式は副問い合わせの結果行それぞれに対して、指定された`operator`を使用して評価、比較されます。なお、`operator`は結果として論理値を生成する必要があります。真の結果が1つでもあると、`ANY`の結果は"true（真）"です。（副問い合わせが行を返さない場合を含む）真の結果がないと、結果は"false（偽）"です。

`SOME`は`ANY`の同義語です。`IN`は`= ANY`と等価です。

成功がなく、右辺の行が演算子の結果として1つでもNULLを生成した場合、`ANY`構文の結果は偽ではなくNULLになることに注意してください。これは、NULL値の論理的な組み合わせに対するSQLの標準規則に従うものです。

`EXISTS`と同様、副問い合わせが完全に評価されると前提してはなりません。

```
row_constructor operator ANY (subquery)
row_constructor operator SOME (subquery)
```

`ANY`のこの形式の左辺は、[項4.2.13](#)で説明されている行コンストラクタです。右辺は括弧で括られた副問い合わせで、左辺の行にある式の数と正確に同じ数の列を返さなければなりません。左辺の式は副問い合わせの結果のそれぞれの行に対し、与えられた`operator`を使用して行に関する評価、比較が行われます。比較の結果、副問い合わせのいかなる行に対して真となる場合、`ANY`の結果は"true（真）"です。比較の結果、副問い合わせの全ての行に対して偽となる場合（副問い合わせが行を返さないという場合も含めて）、結果は"false（偽）"です。比較の結果、いかなる行でも真を返さず、かつ、少なくとも1つの行がNULLを返す場合、結果はNULLになります。

行比較の意味についての詳細は[項9.23.5](#)を参照して下さい。

9.22.5. ALL

```
expression operator ALL (subquery)
```

右辺は括弧で括られた副問い合わせで、正確に1つの列を返さなければなりません。左辺の式は副問い合わせの結果行それぞれに対して、指定された`operator`を使用して評価、比較されます。なお、`operator`は結果として論理値を生成する必要があります。（副問い合わせが行を返さない場合を含む）全ての行が真になる場合、`ALL`の結果は"true（真）"です。1つでも偽の結果があると、結果は"false（偽）"です。比較がすべての行で偽を返さず、かつ、少なくとも1つの行でNULLを返した場合、結果はNULLとなります。

`NOT IN`は`<> ALL`と等価です。

`EXISTS`と同様、副問い合わせが完全に評価されると前提してはなりません。

```
row_constructor operator ALL (subquery)
```

`ALL`のこの形式の左辺は、[項4.2.13](#)で説明する行コンストラクタです。右辺は括弧で括られた副問い合わせで、左辺の行にある式の数と正確に同じ数の列を返さなければなりません。左辺の式は副問い合わせの結果のそれぞれの行に対し、与えられた`operator`を使用して行に関する評価、比較が行われます。比較した結果、すべての副問い合わせ行に対して真を返す場合（副問い合わせが行を返さないという場合も含めて）、`ALL`の結果は"true（真）"となります。比較した結果、いずれかの副問い合わせ行で偽を返す場合、この結果は"false（偽）"となります。比較結果がすべての副問い合わせ行に対して偽を返さず、少なくとも

も 1 行でNULLを返す場合、結果はNULLとなります。

行に関する比較の意味については[項9.23.5](#)を参照してください。

9.22.6. 行に関する比較

row_constructor operator (subquery)

左辺は、[項4.2.13](#)で説明されている行コンストラクタです。右辺は括弧で括られた副問い合わせで、左辺の行とまったく同じ数の列を返さなければなりません。さらに、副問い合わせは複数行を返すことはできません（行をまったく返さない場合、結果はNULLとみなされます）。左辺は副問い合わせの結果の単一行に対し行全体で評価、比較が行われます。

行に関する比較の意味についての詳細は[項9.23.5](#)を参照してください。

9.23. 行と配列の比較

本節では、値のグループ間で複数の比較を行う、さまざまな特殊化したコンストラクトについて説明します。この形式は構文的には、前節の副問い合わせ形式と関係しています。しかし、副問い合わせを含みません。配列副式を含むこの形式はPostgreSQLの拡張です。この他はSQL準拠です。本節で記載した全ての式形式は結果として論理値（真/偽）を返します。

9.23.1. IN

```
expression IN (value [, ...])
```

右辺は括弧で括られたスカラ式のリストです。左辺の式の結果が右辺の式のいずれかと等しい場合、結果は"true（真）"になります。これは以下の省略形です。

```
expression = value1
OR
expression = value2
OR
...
```

左辺の式がNULLを生じる場合、または右側の値に等しいものがなくて少なくとも1つの右辺の行がNULLを持つ場合、`IN`構文の結果は偽ではなくNULLとなることに注意してください。これは、NULL値の論理的な組み合わせに対するSQLの標準規則に従うものです。

9.23.2. NOT IN

```
expression NOT IN (value [, ...])
```

右辺は括弧で括られたスカラ式のリストです。左辺の式の結果が右辺の式の全てと等しくない場合、結果は"真"です。これは以下の省略形です。

```
expression <> value1
AND
expression <> value2
AND
...
```

左辺の式でNULLが生じる場合、または右側の値に等しいものがなく、少なくとも1つの右辺の式がNULLを生み出す場合、予想通り`NOT IN`構文の結果は真ではなくNULLとなることに注意してください。これは、NULL値の論理的な組み合わせに対するSQLの標準規則に従うものです。

ティップ: 全ての場合において、`x NOT IN y`は`NOT (x IN y)`と等価です。しかし、`IN`を使用するよりも`NOT IN`を使用する方が初心者がNULL値による間違いをしやすくなります。可能な限り条件を肯定的に表現することが最善です

9.23.3. ANY/SOME (array)

```
expression operator ANY (array expression)
expression operator SOME (array expression)
```

右辺は括弧で括られた式で、配列値を返さなければなりません。左辺の式は配列要素それぞれに対して、指定された`operator`を使用して評価、比較されます。なお、`operator`は結果として論理値を生成する必要があります。真の結果が1つでもあると、`ANY`の結果は"true（真）"です。（配列の要素数がゼロである場合を含み）真の結果がないと、結果は"false（偽）"です。

配列式がNULL配列を生成する場合、`ANY`の結果はNULLになります。左辺式がNULLとなる場合、`ANY`の結果は通常NULLになります。（あまり厳密でない比較演算子では異なる結果になるかもしれません。）また、右辺の配列にNULL要素が含まれ、かつ、比較した結果真でない値になった場合、`ANY`の結果は偽ではなくNULLになります。（繰り返しになりますが、厳密な演算子の場合です。）これは、NULLに対する、SQLの論理値組み合わせに関する標準規則に従うものです。

`SOME`は`ANY`の同義語です。

9.23.4. ALL (array)

```
expression operator ALL (array expression)
```

右辺は括弧で括られた式で、配列値を返さなければなりません。左辺の式は配列の要素それぞれに対して、指定され

た`operator`を使用して評価、比較されます。なお、`operator`は結果として論理値を生成する必要があります。（配列の要素数がゼロである場合を含み）全ての比較が真になる場合、`ALL`の結果は"true（真）"です。1つでも偽の結果があると、結果は"false（偽）"です。

配列式が`NULL`配列を生成する場合、`ALL`の結果は`NULL`になります。左辺式が`NULL`となる場合、`ALL`の結果は通常`NULL`になります。（厳密でない比較演算子では異なる結果になるかもしれません。）また、右辺の配列に`NULL`要素が含まれ、かつ、比較した結果偽でない値になった場合、`ALL`の結果は真ではなく`NULL`になります。（繰り返しになりますが、厳密な演算子の場合です。）これは、`NULL`に対する、SQLの論理値組み合わせに関する標準規則に従うものです。

9.23.5. 行に関する比較

```
row_constructor operator row_constructor
```

両辺とも[項4.2.13](#)で説明する行コンストラクタです。この2つの行値は同じフィールド数でなければなりません。両辺はそれぞれ評価され、行として比較されます。行の比較は、`operator`が`=`、`<>`、`<`、`<=`、`>`、`>=`の場合、または、これらのうちの1つと同じ意味を持つ場合に認められます。（より具体的には、演算子がB-Tree演算子クラスのメンバ、または、B-Tree演算子クラスの`=`メンバの否定子であれば、行比較演算子となることができます。）

`=`と`<>`の場合、他と動作が多少異なります。2つの行は対応する全ての構成要素が非`NULL`かつ等しい場合に等しいとみなされます。1つでも構成要素が非`NULL`かつ等しくない場合、2つの行は等しくないとみなされます。それ以外その行の比較結果は不明（`NULL`）です。

`<`、`<=`、`>`、`>=`の場合、行の要素は左から右に比較されます。そして、不等または`NULL`の組み合わせが見つかったところで停止します。要素の組み合わせのどちらかが`NULL`であった場合、行比較の結果は不明（`NULL`）です。さもなくば、要素の組み合わせの比較により結果が決まります。例えば、`ROW(1,2,NULL) < ROW(1,3,0)` は、3番目の要素の組み合わせまで進まないため、`NULL`ではなく真を返します。

注意: PostgreSQL 8.2以前では、`<`、`<=`、`>`、`>=`の場合SQL仕様に従っていませんでした。`ROW(a,b) < ROW(c,d)`などの比較は正しくは`a < c OR (a = c AND b < d)`ですが、`a < c AND b < d`として実装されていました。

```
row_constructor IS DISTINCT FROM row_constructor
```

このコンストラクトは`<>`行比較と類似していますが、`NULL`入力に対して`NULL`を生成しない点が異なります。その代わりに、全ての`NULL`値は非`NULL`値と等しくない（`DISTINCT FROM`）ものとみなされ、また、2つの`NULL`は等しい（`NOT DISTINCT`）ものとみなされます。したがって、結果は真か偽のいずれかで、`NULL`にはなりません。

```
row_constructor IS NOT DISTINCT FROM row_constructor
```

このコンストラクトは`=`行比較と類似していますが、`NULL`入力に対して`NULL`を生成しません。代わりに、`NULL`値を、すべての非`NULL`の値に対して不等（`DISTINCT FROM`）とみなし、2つの`NULL`を等しいもの（`NOT DISTINCT`）とみなします。したがって、結果は常に真か偽となり、`NULL`になることはありません

注意: SQL仕様は、結果が`NULL`値、または`NULL`と非`NULL`の比較に基づくのであれば、行の観点からの比較を必要とします。PostgreSQLは、2つの行構成子の出力の比較を行う時、または副問い合わせの出力に対し（[項9.22](#)にあるように）行構成子の比較を行う時のみこれを実施します。2つの複合型の値が比較されるほかの状況では、2つの`NULL`フィールドの値は等しいと考えられ、`NULL`は非`NULL`より大きいとみなされます。複合型に対して、これは一貫した並び替えとインデックス付け動作担保のため必要です。

[前のページ](#)
副問い合わせ式

[ホーム](#)
[上に戻る](#)

[次のページ](#)
集合を返す関数

9.24. 集合を返す関数

本節では、場合により複数行を返す関数について説明します。現在、このクラスの関数のみが、[表9-48](#)、および[表9-49](#)で説明する、連続生成関数です。

表 9-48. Series Generating Functions

関数	引数型	戻り型	説明
generate_series(start, stop)	intまたはbigint	setof intまたはsetof bigint (引数の型と同一)	startからstopまで、刻み1で連続する値を生成します。
generate_series(start, stop, step)	intまたはbigint	setof intまたはsetof bigint (引数の型と同一)	startからstopまで、刻みstepで連続する値を生成します。
generate_series(start, stop, step interval)	timestamp または timestamp with time zone	setof timestamp または setof timestamp with time zone (引数型と同じ)	startからstopまで、刻みstepで連続する値を生成します。

stepが正の場合、startがstopよりも大きいと0行が返ります。反対に、stepが負の場合は、startがstopよりも小さいと0行が返ります。stepが0の時はエラーになります。また、NULLが入力された場合も0行が返ります。以下にいくつか例を示します。

```
SELECT * FROM generate_series(2,4);
generate_series
-----
                2
                3
                4
(3 rows)

SELECT * FROM generate_series(5,1,-2);
generate_series
-----
                5
                3
                1
(3 rows)

SELECT * FROM generate_series(4,3);
generate_series
-----
(0 rows)

-- この例は日付に整数を足し込む演算子に依存します。
SELECT current_date + s.a AS dates FROM generate_series(0,14,7) AS s(a);
dates
-----
2004-02-05
2004-02-12
2004-02-19
(3 rows)

SELECT * FROM generate_series('2008-03-01 00:00'::timestamp,
                              '2008-03-04 12:00', '10 hours');
generate_series
-----
2008-03-01 00:00:00
2008-03-01 10:00:00
2008-03-01 20:00:00
2008-03-02 06:00:00
2008-03-02 16:00:00
2008-03-03 02:00:00
2008-03-03 12:00:00
2008-03-03 22:00:00
2008-03-04 08:00:00
(9 rows)
```

表 9-49. 添え字生成関数

関数	戻り値型	説明

generate_subscripts(array anyarray, dim int)	setof int	指定した配列の添え字を構成するシリーズを生成します。
generate_subscripts(array anyarray, dim int, reverse boolean)	setof int	指定した配列の添え字を構成する連番を生成します。 reverseが真の場合、連番は逆順に返されます。

generate_subscriptsは、指定した配列の指定した次数で有効な添え字からなる集合を生成するために便利な関数です。 要求された次数を持たない配列またはNULL配列では行は返されません。 （しかしこれはNULL配列要素では有効な添え字です。） いくつかの例を以下に示します。

```
-- 基本的用法
SELECT generate_subscripts('{NULL,1,NULL,2}'::int[], 1) AS s;
s
---
1
2
3
4
(4 rows)
-- ある配列では、添え字とその添え字が示す値は
-- 副問い合わせが必要です。
SELECT * FROM arrays;
a
-----
{-1,-2}
{100,200,300}
(2 rows)

SELECT a AS array, s AS subscript, a[s] AS value
FROM (SELECT generate_subscripts(a, 1) AS s, a FROM arrays) foo;
array | subscript | value
-----+-----+-----
{-1,-2} | 1 | -1
{-1,-2} | 2 | -2
{100,200,300} | 1 | 100
{100,200,300} | 2 | 200
{100,200,300} | 3 | 300
(5 rows)

-- 2次元配列の入れ子を解きます。
CREATE OR REPLACE FUNCTION unnest2(anyarray)
RETURNS SETOF anyelement AS $$
select $1[i][j]
from generate_subscripts($1,1) g1(i),
generate_subscripts($1,2) g2(j);
$$ LANGUAGE sql IMMUTABLE;
CREATE FUNCTION
postgres=# SELECT * FROM unnest2(ARRAY[[1,2],[3,4]]);
unnest2
-----
1
2
3
4
(4 rows)
```

9.25. システム情報関数

[表9-50](#)に、セッションおよびシステムの情報を抽出する関数を示します。

本節でリストされている関数のほかに、同様にシステム情報を提供する統計システムに関連した数多くの関数があります。[項27.2.2](#)にさらに情報があります。

表 9-50. セッション情報関数

名前	戻り型	説明
<code>current_catalog</code>	name	(SQL標準で"カタログ"と呼ばれる) 現在のデータベースの名前
<code>current_database()</code>	name	現在のデータベースの名前
<code>current_query()</code>	text	クライアントから提示された、現在実行中問い合わせのテキスト (複数の文を含むことあり)
<code>current_schema()</code>	name	現在のスキーマの名前
<code>current_schemas(boolean)</code>	name[]	オプションで暗黙的なスキーマも含む、検索パス内のスキーマの名前
<code>current_user</code>	name	現在の処理実施コンテキストのユーザ名
<code>inet_client_addr()</code>	inet	リモート接続のアドレス
<code>inet_client_port()</code>	int	リモート接続のポート
<code>inet_server_addr()</code>	inet	ローカル接続のアドレス
<code>inet_server_port()</code>	int	ローカル接続のポート
<code>pg_backend_pid()</code>	int	現在のセッションに結びついたサーバプロセスのプロセスID
<code>pg_conf_load_time()</code>	timestamp with time zone	設定を読み込んだ時刻
<code>pg_is_other_temp_schema(oid)</code>	boolean	スキーマが他のセッションの一時スキーマかどうか?
<code>pg_listening_channels()</code>	setof text	今のセッションにおいて現在LISTEN中のチャンネル名
<code>pg_my_temp_schema()</code>	oid	セッションの一時スキーマのOID。もしなければ 0
<code>pg_postmaster_start_time()</code>	timestamp with time zone	サーバの起動時刻
<code>pg_trigger_depth()</code>	int	PostgreSQLのトリガの現在の入れ子の深さ (直接もしくは間接的に、トリガ内部から呼ばれていなければ0を返す)
<code>session_user</code>	name	セッションのユーザ名
<code>user</code>	name	<code>current_user</code> と等価
<code>version()</code>	text	PostgreSQLバージョン情報

注意: `current_catalog`、`current_schema`、`current_user`、`session_user` および `user` はSQLにおいて特殊な構文上の地位を持っており、最後に括弧を付けずに呼び出さなければなりません。(PostgreSQLでは`current_schema`の場合括弧を使用することができますが、他は使えません。)

`session_user` は、通常、現在のデータベース接続を初期化したユーザです。しかし、スーパーユーザはこの設定を[SET SESSION AUTHORIZATION](#)を使用して変更することができます。`current_user` は、権限の検査に適用されるユーザ識別子です。通常はセッションユーザと同じですが、[SET ROLE](#)を使用して変更可能です。`SECURITY DEFINER`属性を持つ関数の実行中にも変更します。Unix用語で言うと、セッションユーザは"実ユーザ"で、現在のユーザは"実効ユーザ"です。

`current_schema` 関数は、検索パスの最初にあるスキーマ名 (もしくは、検索パスが空の場合はNULL値) を返します。これは、対象スキーマを指定せずに作成された、任意のテーブルまたは他の名前付きオブジェクトに使用されるスキーマです。また、`current_schemas(boolean)` は、検索パスに存在する全てのスキーマ名の配列を返します。booleanオプションにより、`pg_catalog`のような暗黙的に含まれているシステムスキーマを、結果の検索パスに含めるかどうかを指定します。

注意: 検索パスは、実行時の設定で変更することができます。使用するコマンドは以下の通りです。

```
SET search_path TO schema [, schema, ...]
```

`pg_listening_channels` は、現在のセッションでLISTEN中のチャンネル名のセットを返します。詳細は[LISTEN](#)を参照して下さい。

`inet_client_addr`は現在のクライアントのIPアドレスを返し、`inet_client_port`はそのポート番号を返します。`inet_server_addr`は、現在の接続を受け付けたサーバのIPアドレスを返し、`inet_server_port`はそのポート番号を返します。これら全ての関数は接続がUnix-domain socket経由の場合NULLを返します。

`pg_my_temp_schema`は、現在のセッションの一時スキーマのOIDを返します。もし（どんな一時テーブルも作成していないので）存在しなければゼロを返します。`pg_is_other_temp_schema`は、指定したOIDが他のセッションの一時スキーマのOIDであれば、真を返します。（例えば、他のセッションの一時テーブルをカタログ表示から除外したい場合などで有用です。）

`pg_postmaster_start_time`はサーバが起動した時のtimestamp with time zoneを返します。

`pg_conf_load_time`はサーバの設定ファイルが最後に読み込まれた時のtimestamp with time zoneを返します。（現在のセッションがその時点で稼働していた場合、これはセッション自体が設定ファイルを読み込んだ時刻となります。このためこの読み取った結果は多少他のセッションと異なります。この他の場合、postmasterプロセスが設定ファイルを再読み込みした時刻となります。）

`version`関数はPostgreSQLサーバのバージョンを記述した文字列を返します。

[表9-51](#)に列挙した関数を使用して、ユーザはオブジェクトのアクセス権限をプログラムから問い合わせることができます。権限についての詳細は、[項5.6](#)を参照してください。

表 9-51. アクセス権限照会関数

名前	戻り型	説明
<code>has_any_column_privilege(user, table, privilege)</code>	boolean	ユーザがテーブルのどの列に対しても権限を所有しているか
<code>has_any_column_privilege(table, privilege)</code>	boolean	現在のユーザがテーブルのどの列に対しても権限を所有しているか
<code>has_column_privilege(user, table, column, privilege)</code>	boolean	ユーザに列に対する権限があるか
<code>has_column_privilege(table, column, privilege)</code>	boolean	現在のユーザに列に対する権限があるか
<code>has_database_privilege(user, database, privilege)</code>	boolean	ユーザにデータベースに対する権限があるのか
<code>has_database_privilege(database, privilege)</code>	boolean	現在のユーザにデータベースに対する権限があるのか
<code>has_foreign_data_wrapper_privilege(user, fdw, privilege)</code>	boolean	ユーザに外部データラップに対する権限があるのか
<code>has_foreign_data_wrapper_privilege(fdw, privilege)</code>	boolean	現在のユーザに外部データラップに対する権限があるのか
<code>has_function_privilege(user, function, privilege)</code>	boolean	ユーザに関数に対する権限があるのか
<code>has_function_privilege(function, privilege)</code>	boolean	現在のユーザに関数に対する権限があるのか
<code>has_language_privilege(user, language, privilege)</code>	boolean	ユーザに言語に対する権限があるのか
<code>has_language_privilege(language, privilege)</code>	boolean	現在のユーザに言語に対する権限があるのか
<code>has_schema_privilege(user, schema, privilege)</code>	boolean	ユーザにスキーマに対する権限があるのか
<code>has_schema_privilege(schema, privilege)</code>	boolean	現在のユーザにスキーマに対する権限があるのか
<code>has_sequence_privilege(user, sequence, privilege)</code>	boolean	ユーザにシーケンスに対する権限があるのか
<code>has_sequence_privilege(sequence, privilege)</code>	boolean	現在のユーザにシーケンスに対する権限があるのか
<code>has_server_privilege(user, server, privilege)</code>	boolean	ユーザに外部サーバに対する権限があるのか
<code>has_server_privilege(server, privilege)</code>	boolean	現在のユーザに外部サーバに対する権限があるのか
<code>has_table_privilege(user, table, privilege)</code>	boolean	ユーザにテーブルに対する権限があるのか
<code>has_table_privilege(table, privilege)</code>	boolean	現在のユーザにテーブルに対する権限があるのか
<code>has_tablespace_privilege(user, tablespace, privilege)</code>	boolean	ユーザにテーブル空間に対する権限があるのか
<code>has_tablespace_privilege(tablespace, privilege)</code>	boolean	現在のユーザにテーブル空間に対する権限があるのか
<code>pg_has_role(user, role, privilege)</code>	boolean	ユーザにロールに対する権限があるのか
<code>pg_has_role(role, privilege)</code>	boolean	現在のユーザにロールに対する権限があるのか

`has_table_privilege`はあるユーザが特定の 방법으로テーブルにアクセス可能かどうかを検査します。ユーザは名前、OID(`pg_authid.oid`)、もしくは擬似的なPUBLICロールを意味する`public`で指定できます。省略された場合は`current_user`が使われます。テーブルは名前もしくはOIDで指定可能です。（従って、実際には`has_table_privilege`の6つの変形があり、引数の数と型で区別されます。）名前を指定する場合、必要であればスキーマ権限で指定できます。目的とするアクセス権限の型はテキスト文字列で指定され、それは、`SELECT`、`INSERT`、`UPDATE`、`DELETE`、`TRUNCATE`、`REFERENCES`、また

はTRIGGERの内の1つで評価されます。オプションとして、権限がGRANTオプションにより付与されているかどうかの権限の検査するため、WITH GRANT OPTIONを付加することができます。更に、複数の権限型はコンマで分割してたりスト可能です。この場合、結果はリストされた権限の1つでも存在すれば真となります。（権限文字列が有効でない場合、追加の空白を権限名の中を除き、間に挿入できます。）例を示します。

```
SELECT has_table_privilege('myschema.mytable', 'select');
SELECT has_table_privilege('joe', 'mytable', 'INSERT, SELECT WITH GRANT OPTION');
```

has_sequence_privilegeはあるユーザが特定の方法でシーケンスにアクセス可能かどうかを検査します。可能な引数はhas_table_privilegeに対するものと類似しています。USAGE、SELECTまたはUPDATEの内のいずれかにより目的とする権限型が評価されます。

has_any_column_privilegeはあるユーザが特定の方法でテーブルのどんな列にもアクセス可能かどうかを検査します。可能な引数はhas_table_privilegeに対するものと類似していますが、SELECT、INSERT、UPDATE、またはREFERENCESのいくつかの組み合わせで目的とする権限型が評価される点を除きます。テーブル階層におけるこれらの権限を所有することは、テーブルのそれぞれの列に対し暗黙的にそれら権限を与えることに注意してください。従って、同一の引数に対してhas_table_privilegeが真を返す場合、has_any_column_privilegeは常に真を返します。しかし、少なくとも一つの列に対し権限の列階層許可があれば、has_any_column_privilegeも成功します。

has_column_privilegeはあるユーザが特定の方法でテーブルのある列にもアクセス可能かどうかを検査します。可能な引数は、列が追加的に名前または属性番号のいずれかで指定される点を加えれば、has_table_privilegeと類似しています。目的と知るアクセス権限型は、SELECT、INSERT、UPDATE、またはREFERENCESのいくつかの組み合わせで評価されなければなりません。テーブル階層におけるこれらの権限を所有することは、テーブルのそれぞれの列に対し暗黙的にそれら権限を与えることに注意してください。

has_database_privilege関数は、あるユーザが特定の方法でデータベースにアクセス可能かどうかを検査します。可能な引数は、has_table_privilegeに類似しています。対象とするアクセス権限の種類は、CREATE、CONNECT、TEMPORARY、またはTEMP（TEMPORARYと同じ）のいくつかの組み合わせで評価する必要があります。

has_function_privilege関数は、あるユーザが特定の方法で関数にアクセス可能かどうかを検査します。引数として取り得るものはhas_table_privilegeと同じです。関数をOIDではなくテキスト文字列で指定した場合、regprocedure データ型（[項8.18](#)を参照）と同じ入力が可能です。対象とするアクセス権限の種類は、現在EXECUTEで評価する必要があります。以下に例を示します。

```
SELECT has_function_privilege('joeuser', 'myfunc(int, text)', 'execute');
```

has_foreign_data_wrapper_privilege関数は、あるユーザが特定の方法で外部データラッパーにアクセス可能かどうかを検査します。引数として取り得るものはhas_table_privilegeと同じです。対象とするアクセス権限の種類はUSAGEで評価する必要があります。

has_language_privilege関数は、あるユーザが特定の方法で手続き言語にアクセス可能かどうかを検査します。可能な引数は、has_table_privilegeに類似しています。対象とするアクセス権限の種類は、現在USAGEで評価する必要があります

has_schema_privilege関数は、あるユーザが特定の方法でスキーマにアクセス可能かどうかを検査します。可能な引数は、has_table_privilegeに類似しています。対象とするアクセス権限の種類は、CREATEもしくはUSAGEの何らかの組み合わせで評価する必要があります。

has_server_privilegeはユーザが特定の方法で外部サーバにアクセスできるかどうかを検査します。可能な引数はhas_table_privilegeに類似しています。対象とするアクセス権限はUSAGEで評価する必要があります。

has_tablespace_privilege関数は、あるユーザが特定の方法でテーブル空間にアクセス可能かどうかを検査します。可能な引数は、has_table_privilegeに類似しています。対象とするアクセス権限の種類は、CREATEで評価する必要があります。

pg_has_roleは、ユーザが特定の方法でロールにアクセスできるかどうかを検査します。取り得る引数については、publicがユーザ名として使用不可であることを除き、has_table_privilegeと類似しています。対象とするアクセス権限の種類は、MEMBER、USAGEの組み合わせで評価する必要があります。MEMBERは、ロールの直接または間接メンバ資格（つまり、SET ROLEを行う権限）を表します。一方、USAGEは、そのロール権限がSET ROLEをしなくてもすぐに利用できることを表します。

[表9-52](#)に、特定のオブジェクトが、現行スキーマの検索パスにおいて可視かどうかを判別する関数を示します。例えば、あるテーブルを含むスキーマが検索パス内に存在し、検索パス内の前方に同じ名前のテーブルがない場合、そのテーブルは可視であると言えます。つまり、これは、テーブルが明示的なスキーマ修飾なしで名前によって参照可能であるということです。全ての可視テーブルの名前を列挙するには以下のようにします。

```
SELECT relname FROM pg_class WHERE pg_table_is_visible(oid);
```

表 9-52. スキーマ可視性照会関数

名前	戻り型	説明

pg_collation_is_visible(collation_oid)	boolean	照合順序が検索パスにおいて可視かどうか
pg_conversion_is_visible(conversion_oid)	boolean	変換が検索パスにおいて可視かどうか
pg_function_is_visible(function_oid)	boolean	関数が検索パスにおいて可視であるか
pg_opclass_is_visible(opclass_oid)	boolean	演算子クラスが検索パスにおいて可視であるか
pg_operator_is_visible(operator_oid)	boolean	演算子が検索パスにおいて可視であるか
pg_opfamily_is_visible(opclass_oid)	boolean	演算子族が検索パスにおいて可視であるか
pg_table_is_visible(table_oid)	boolean	テーブルが検索パスにおいて可視かどうか
pg_ts_config_is_visible(config_oid)	boolean	テキスト検索設定が検索パスにおいて可視かどうか
pg_ts_dict_is_visible(dict_oid)	boolean	テキスト検索辞書が検索パスにおいて可視かどうか
pg_ts_parser_is_visible(parser_oid)	boolean	テキスト検索パーサが検索パスにおいて可視かどうか
pg_ts_template_is_visible(template_oid)	boolean	テキスト検索テンプレートが検索パスにおいて可視かどうか
pg_type_is_visible(type_oid)	boolean	型（またはドメイン）が検索パスにおいて可視かどうか

それぞれの関数はデータベースオブジェクトの1つの型に対して可視性の検査を行います。pg_table_is_visibleがビュー、インデックス、シーケンスに対しても使用できること、pg_type_is_visibleがドメインに対しても使用できることに注意してください。関数および演算子では、パスの前方に同じ名前かつ同じ引数のデータ型を持つオブジェクトが存在しなければ、検索パス内のオブジェクトは可視です。演算子クラスでは、名前と関連するインデックスアクセスメソッドが考慮されます。

これらの関数は全て、検査するオブジェクトを識別するために、オブジェクトのOIDを必要とします。オブジェクトを名前でテストする場合、OID別名型 (regclass、regtype、regprocedure、regoperator、regconfig、またはregdictionary) を使用すると便利です。例えば、以下のようになります。

```
SELECT pg_type_is_visible('myschema.widget '::regtype);
```

ただし、このようなやり方でスキーマ修飾されていない型名をテストしても、あまり意味がないことに注意してください。名前が認識されれば、それは必ず可視ということになります。

表9-53に、システムカタログから情報を抽出する関数を列挙します。

表 9-53. システムカタログ情報関数

名前	戻り型	説明
format_type(type_oid, typemod)	text	データ型のSQL名の取得
pg_describe_object(catalog_id, object_id, object_sub_id)	text	データベースオブジェクトの説明の取得
pg_get_constraintdef(constraint_oid)	text	制約の定義を取得
pg_get_constraintdef(constraint_oid, pretty_bool)	text	制約の定義を取得
pg_get_expr(pg_node_tree, relation_oid)	text	式中のいかなるVarも、2番目のパラメータによって指定されるリレーションを参照すると仮定して、式の内部形式を逆コンパイル
pg_get_expr(pg_node_tree, relation_oid, pretty_bool)	text	式中のいかなるVarも、2番目のパラメータによって指定されるリレーションを参照すると仮定して、式の内部形式を逆コンパイル
pg_get_functiondef(func_oid)	text	関数定義の取得
pg_get_function_arguments(func_oid)	text	関数定義（デフォルト値付き）の引数リストを取得
pg_get_function_identity_arguments(func_oid)	text	関数（デフォルト値の付いていない）特定の引数リストを取得
pg_get_function_result(func_oid)	text	関数に対するRETURNS句の取得
pg_get_indexdef(index_oid)	text	インデックスに対するCREATE INDEX コマンドの取得
pg_get_indexdef(index_oid, column_no, pretty_bool)	text	インデックスに対するCREATE INDEX コマンドの取得、またはcolumn_noが非ゼロの場合は、1つのインデックス列のみの定義の取得
pg_get_keywords()	setof record	SQLキーワードとそれらの種類のリストを取得
pg_get_ruledef(rule_oid)	text	ルールに対するCREATE RULE コマンドを取得
pg_get_ruledef(rule_oid, pretty_bool)	text	ルールに対するCREATE RULE コマンドを取得
pg_get_serial_sequence(table_name, column_name)	text	serial、smallserial、またはbigserial列が使用するシーケンス名の取得

pg_get_triggerdef(trigger_oid)	text	トリガに対するCREATE [CONSTRAINT] TRIGGER コマンドの取得
pg_get_triggerdef(trigger_oid, pretty_bool)	text	トリガに対するCREATE [CONSTRAINT] TRIGGER コマンドの取得
pg_get_userbyid(role_oid)	name	与えられたOIDでロール名を取得
pg_get_viewdef(view_name)	text	ビューの元になるSELECT コマンドを取得（廃止予定）
pg_get_viewdef(view_name, pretty_bool)	text	ビューの元になるSELECT コマンドを取得。pretty_boolが真の場合フィールド行は80文字で折り返します。（廃止予定）
pg_get_viewdef(view_oid)	text	ビューの元になるSELECT コマンドを取得
pg_get_viewdef(view_oid, pretty_bool)	text	ビューの元になるSELECT コマンドを取得。pretty_boolが真の場合フィールド行は80文字で折り返します。
pg_get_viewdef(view_oid, wrap_column_int)	text	ビューの元になるSELECT コマンドを取得。折り返し文字数(wrap_column_int)が指定された場合は、その行で折り返します。
pg_options_to_table(reloptions)	setof record	格納オプションの名前/値のペアのセットを取得
pg_tablespace_databases(tablespace_oid)	setof oid	テーブル空間内にオブジェクトを所有するデータベースOID集合を取得
pg_tablespace_location(tablespace_oid)	text	指定したテーブル空間が実際に配置されているファイルシステム上のパスを取得
pg_typeof(any)	regtype	指定した値のデータ型を取得
collation for (any)	text	引数の照合順序を取得

関数format_typeは、自身の型OIDと場合により型修飾子によって特定されるデータ型のSQL名を返します。特定の修飾子が既知でなければ型修飾子にNULLを渡します。

関数pg_get_keywordsは、サーバが認識したSQLキーワードを記述するレコード一式を返します。word列にはキーワードが含まれます。catcode列は種類コードを含みます。それらは、予約されていないを表わすU、列名のC、関数名型のT、または予約語であることのR、のいずれかです。catdesc列は種類を記述する現地語化された可能性のある文字列を含みます。

pg_get_constraintdef、pg_get_indexdef、pg_get_ruledef、およびpg_get_triggerdefそれぞれは、対応する制約、インデックス、ルール、もしくはトリガに対するコマンド作成を再構築します。（これはコマンドのオリジナルテキストではなく、逆コンパイルされた再構築であることに注意してください。）pg_get_exprは、列のデフォルト値のような個々の式の内部形式を逆コンパイルします。システムカタログの内容を試験するときに便利です。もしも式がVarを含んでいるとすれば、2番目のパラメータとして参照する関連（リレーション）のOIDを指定します。もしVarがないと期待されればゼロで充分です。pg_get_viewdefはビューを定義するSELECT問い合わせを再構築します。これらの関数のほとんどは2つの異形があり、そのうちの1つはオプションとして結果を"pretty print"（訳注："見やすく表示"）します。pretty printされた書式はより読みやすい半面、デフォルトの書式の方が今後のバージョンのPostgreSQLでも同様に解釈することができそうです。ダンプを目的とする場合は、見やすく表示する出力を使用しないでください。見やすい表示用のパラメータにfalseを渡すことで、このパラメータをまったく持たない種類の結果と同一の結果を生成します

pg_get_functiondefは、ある関数のCREATE OR REPLACE FUNCTION文全てを返します。pg_get_function_argumentsは、CREATE FUNCTIONの中に出現しなければならない形式で、引数リストを返します。pg_get_function_resultは同様、その関数の適切なRETURNS句を返します。pg_get_function_identity_argumentsは、インスタンスに対し、ALTER FUNCTIONの中に出現しなければならない形式で、ある関数を特定するのに必要な引数リストを返します。この形式はデフォルト値を省略します。

pg_get_serial_sequenceは、列に関連したシーケンスの名前を返します。もし、列に関連したシーケンスが存在しなければ、NULLを返します。最初の入力パラメータはテーブル名です。スキーマを付けることもできます。2番目のパラメータは列名です。最初のパラメータは普通スキーマとテーブルですので、二重引用符付の識別子としては解釈されません。つまり、デフォルトで小文字に変換されます。一方2番目のパラメータは単なる列名であり、二重引用符付として解釈され、その大文字小文字は保持されます。この関数は、シーケンス関数に渡すことができるよう適切な書式で値を返します（[項9.16](#)を参照）。この関連付けはALTER SEQUENCE OWNED BYにより変更、削除することができます。（この関数はおそらくpg_get_owned_sequenceから呼び出されるはずです。その現在の名前は、serialやbigserial列で実際に入力されたものが反映されます。）

pg_get_userbyidはそのOIDで与えられたロールの名前を抽出します。

pg_options_to_tableはpg_class.reloptionsかpg_attribute.attoptionsへ格納オプションが設定されている場合に そのペア(option_name/option_value)を返します。

pg_tablespace_databasesによりテーブル空間の使用状況を確認することができます。これは、そのテーブル空間内に格納されたオブジェクトを持つデータベースのOIDの集合を返します。この関数が何らかの行を返した場合、このテーブル空間は空ではなく、そのため、削除することができません。テーブル空間内に格納された特定のオブジェクトを表示するために、pg_tablespace_databasesで識別されたデータベースに接続し、そのpg_classカタログに問い合わせを行う必要があります。

pg_describe_objectはカタログOID、オブジェクトOID、もしくは(おそらく0の)サブオブジェクトOIDで指定されたデータベースオブジェクトの説明を返します。これはpg_dependカタログに格納されたオブジェクトの識別判断の際に有用です。

pg_typeofは、OIDに渡される値のデータ型のOIDを返します。これはトラブル解決作業、または動的にSQL問い合わせを生成するのに便利です。この関数は、OID型の別名であるregtypeを返すものとして宣言されます(項8.18を参照)。つまり、比較目的でのOIDと同一ですが、型名として表示されます。以下に例をあげます。

```
SELECT pg_typeof(33);

 pg_typeof
-----
 integer
(1 row)

SELECT typlen FROM pg_type WHERE oid = pg_typeof(33);
 typlen
-----
      4
(1 row)
```

collation for式は、それに渡された値の照合順序を返します。例を示します。

```
SELECT collation for (description) FROM pg_description LIMIT 1;
 pg_collation_for
-----
 "default"
(1 row)

SELECT collation for ('foo' COLLATE "de_DE");
 pg_collation_for
-----
 "de_DE"
(1 row)
```

値はクォートで囲い、スキーマを明示的に指定します。もし引数の式から何も照合順序が得られなかった場合、NULLが返されます。もし引数が照合順序をサポートしないデータ型だった場合、エラーとなります。

表9-54に示される関数は、COMMENTコマンドによって以前に保存されたコメントを抽出します。指定されたパラメータに対するコメントが存在しない場合、NULL値が返されます。

表 9-54. コメント情報関数

名前	戻り型	説明
col_description(table_oid, column_number)	text	テーブル列のコメントを取得
obj_description(object_oid, catalog_name)	text	データベースオブジェクトのコメントを取得
obj_description(object_oid)	text	データベースオブジェクトのコメントを取得 (廃止予定)
shobj_description(object_oid, catalog_name)	text	共有データベースオブジェクトのコメントを取得

col_description() 関数は、テーブルのOIDとその列番号で指定されるテーブルの列のコメントを返します。obj_description() 関数は、テーブル列自体がOIDを所有していませんので、テーブル列に対して使用することはできません。

obj_description() 関数の2つのパラメータを取る形式はそのOIDと保有しているシステムカタログの名前で指定されたデータベースオブジェクトのコメントを返します。例えば、obj_description(123456, 'pg_class') はOID 123456を持つテーブルのコメントを抽出します。obj_description() 関数の単一パラメータ形式はオブジェクトのOIDのみ必要とします。異なったシステムカタログに渡ってOIDが一意である保証はないことから廃止予定です。従って、間違ったコメントが返ることがあります。

shobj_descriptionは、共有オブジェクトのコメント取得のために使用されることを除いて、obj_descriptionと同じように使用されます。一部のシステムカタログは、各クラスタ内のデータベース全体に対して大域的で、これらのコメントも同様に大域的に格納されます。

表9-55で示される関数はサーバトランザクション情報をエクスポートできる形式で提供します。これら関数の主な使用目的は2つのスナップショット間でどちらのトランザクションがコミットされたのかを特定するためです。

表 9-55. トランザクションIDとスナップショット

名前	戻り型	説明
txid_current()	bigint	現在のトランザクションIDの取得

txid_current_snapshot()	txid_snapshot	現在のスナップショットの取得
txid_snapshot_xip(txid_snapshot)	setof bigint	スナップショットにある進行中のトランザクションIDの取得
txid_snapshot_xmax(txid_snapshot)	bigint	スナップショットのxmaxの取得
txid_snapshot_xmin(txid_snapshot)	bigint	スナップショットのxminの取得
txid_visible_in_snapshot(bigint, txid_snapshot)	boolean	スナップショットにあるトランザクションIDは可視か？（サブトランザクションIDと一緒に使用しないこと）

内部トランザクションID型（xid）は32ビット幅あるので40億トランザクション毎ラップします。とは言っても、これらの関数は"epoch"カウンタにより64ビット形式にエクスポートするため、インストレーションの生涯にわたってラップしません。これらの関数で使用するデータ型、txid_snapshotはある時間特有時のトランザクションIDの可視化に関する情報を格納します。構成要素は[表9-56](#)に記載されています。

表 9-56. スナップショット構成要素

名前	説明
xmin	いまだ使用中の最先として出現したトランザクションID（txid）。全ての最も早期のトランザクションはコミットされ、可視化となるか、またはロールバックされ消滅するのいずれか。
xmax	最初のまだ割り当てられていないtxid。これと等しいかより大きい全てのtxidはスナップショットの時点で未開始。従って不可視。
xip_list	スナップショット時の使用中のtxid。リストはxminとxmax間にのみ使用中のtxidを含む。xmaxより高位の使用中のtxidが存在することもある。xmin <= txid < xmaxであり、このリストにないtxidはスナップショット時に既に完了している。従い、コミット状態で可視か消滅かのいずれか。リストには副トランザクションのtxidは含まれない。

txid_snapshotのテキスト表現はxmin:xmax:xip_list。例えば、10:20:10,14,15はxmin=10, xmax=20, xip_list=10, 14, 15を指す。

9.26. システム管理関数

本節で説明する関数は、PostgreSQL インストレーションの制御と監視を行うために使用されます。

9.26.1. 構成設定関数

表9-57は、実行時構成パラメータの問い合わせや変更に使用できる関数を示しています。

表 9-57. 構成設定関数

名前	戻り型	説明
current_setting(setting_name)	text	現在の設定値の取得
set_config(setting_name, new_value, is_local)	text	パラメータを設定し、新規値を返す

関数current_settingは、設定setting_nameの現在の値を返します。この関数は、SQLのSHOWコマンドと同じです。以下に例を示します。

```
SELECT current_setting('datestyle');

current_setting
-----
ISO, MDY
(1 row)
```

set_config関数は、パラメータsetting_nameをnew_valueに設定します。ただし、is_localがtrueの場合、新規値は現在のトランザクションにのみ適用されます。新規値を現在のセッションに適用する場合は、代わりにfalseを使用してください。この関数は、SQLのSETコマンドと同じです。以下に例を示します。

```
SELECT set_config('log_statement_stats', 'off', false);

set_config
-----
off
(1 row)
```

9.26.2. サーバシグナル送信関数

表9-58に示す関数は、制御用シグナルを他のサーバプロセスに送信します。これらの関数の使用は、著名な例外を除き、大抵の場合スーパーユーザのみに制限されています。

表 9-58. サーバシグナル送信関数

名前	戻り型	説明
pg_cancel_backend(pid int)	boolean	バックエンドの現在の問い合わせを取り消す。もし関数を読むユーザが取り消す対象のバックエンドと正確に同じロールを保持している場合は、関数を実行することができます。その他全てのケースでは、スーパーユーザでなければいけません。
pg_reload_conf()	boolean	サーバプロセスに構成ファイルの再読み込みをさせる
pg_rotate_logfile()	boolean	サーバログファイルを循環させる
pg_terminate_backend(pid int)	boolean	バックエンドを終結する。もし関数を読むユーザが取り消す対象のバックエンドと正確に同じロールを保持している場合は、関数を実行することができます。その他全てのケースでは、スーパーユーザでなければいけません。

これらのそれぞれの関数は成功の場合true（真）を返し、そうでない場合はfalse（偽）を返します。

pg_cancel_backendとpg_terminate_backendは（それぞれ、SIGINTまたはSIGTERM）シグナルをプロセス識別子で特定されたバックエンドプロセスに送ります。使用中のバックエンドのプロセス識別子はpg_stat_activityビューのpid列から、もしくは、（Unixではps、WindowsではTask Managerにより）サーバ上のpostgresプロセスをリストすることで見つけられます。実行中のバックエンドのロールはpg_stat_activityのusername列から確認することができます。

pg_reload_confはSIGHUPシグナルをサーバに送り、その結果全てのサーバプロセスが構成ファイルを再読み込みすることにな

ります。

pg_rotate_logfileはログファイルマネージャに即座に新規出力ファイルに切替えるよう信号を送ります。これは組み込みログ取得が起動している場合のみ有効です。起動していない場合はログファイルマネージャの子プロセスが存在しない理由からです。

9.26.3. バックアップ制御関数

表9-59に示す関数はオンラインバックアップの作成を支援するものです。これらの関数は、リカバリ中には実行できません。(pg_xlog_location_diffは除く)

表 9-59. バックアップ制御関数

名前	戻り型	説明
pg_create_restore_point(name text)	text	リストア実行用に名前付けされたポイントを作成(スーパーユーザのみ実施可能です)
pg_current_xlog_insert_location()	text	現在のトランザクションログの挿入位置の取得
pg_current_xlog_location()	text	現在のトランザクションログの書き込み位置を取得
pg_start_backup(label text [, fast boolean])	text	オンラインバックアップの実行準備(スーパーユーザかレプリケーション用のロールでのみ実施可能です)
pg_stop_backup()	text	オンラインバックアップの実行の終了(スーパーユーザかレプリケーション用のロールでのみ実施可能です)
pg_switch_xlog()	text	新しいトランザクションログファイルへの強制移行(スーパーユーザのみ実施可能です)
pg_xlogfile_name(location text)	text	トランザクションログの位置を表す文字列をファイル名に変換
pg_xlogfile_name_offset(location text)	text, integer	トランザクションログの位置を表す文字列を、ファイル名とファイル内の10進のバイトオフセットに変換
pg_xlog_location_diff(location text, location text)	numeric	2つのトランザクションログの位置差分を算出

pg_start_backupは、ユーザが任意に定義したバックアップラベルを受け付けます（通常、格納に使用するバックアップダンプファイルにちなんだ名前が付けられます）。この関数は、データベースクラスタのデータディレクトリにバックアップラベルファイル（backup_label）を書き出し、チェックポイントを実行し、バックアップを始めるトランザクションログの位置をテキスト形式で返します。ユーザはこの結果値を無視することができますが、便利なこともあるので提供されています。

```
postgres=# select pg_start_backup('label_goes_here');
pg_start_backup
-----
0/D4445B8
(1 row)
```

オプションのboolean型パラメータがあります。真であれば、すべからく早くpg_start_backupの実行を指定します。いかなる現時点で実行中の問い合わせも速度を落とし、I/O操作で急増の原因の即時チェックポイントを強要します。

pg_stop_backupは、pg_start_backupで作成されたラベルファイルを削除し、トランザクションログ格納領域にバックアップ履歴ファイルを作成します。履歴ファイルにはpg_start_backupで付与されたラベル、バックアップのトランザクションログの位置の開始位置、終了位置、バックアップ開始時刻、終了時刻が含まれます。戻り値は、バックアップの終了トランザクションログの位置です（これも同様に無視可能です）。終了位置を記録した後、現在のトランザクションログの挿入位置は自動的に、次のトランザクションログに進みます。ですので、終了トランザクションログファイルをすぐにアーカイブし、バックアップを完了させることができます。

pg_switch_xlogは、次のトランザクションログファイルに移動し、現在のファイルをアーカイブできるようにします。（アーカイブを続けて使用することを前提とします。）戻り値は、完了した現在のトランザクションログファイル内の終了トランザクションログの位置に1を加えたものです。前回のトランザクションログファイルの切り替えからトランザクションログに変化がなければ、pg_switch_xlogは現在使用中のトランザクションログファイルの開始位置を返します。

pg_create_restore_pointはリカバリターゲットとして使用可能な名前付けされたトランザクションログレコードを生成し、それに該当するログ位置を返します。与えられた名前は、どこまでリカバリをするかを明示的に指定するrecovery_target_nameパラメータに使用することができます。リカバリ処理はリカバリターゲットに指定した名前と一致した最初の時点で終了するため、同じ名前で複数のリストアポイントを作成することは避けてください。

pg_current_xlog_locationは、上記の関数で使用する同一の書式で現在のトランザクションログの書き込み位置を表示します。同様にpg_current_xlog_insert_locationは、現在のトランザクションログの挿入位置を表示します。挿入位置は"論理的"な任意の時点のトランザクションログの終了位置です。一方、書き込み位置は、サーバの内部バッファから書き出された実際

の終了位置です。書き込み位置はサーバ外部から検証可能なものの終端です。通常は、部分的に完了したトランザクションログファイルのアーカイブ処理を行いたい場合に必要とされるものです。挿入位置はサーバをデバッグする際に主に使用されます。これらはどちらも読み取りのみの操作であり、スーパーユーザ権限を必要としません。

pg_xlogfile_name_offsetを使用して、上記いずれの関数の結果からも、対応するトランザクションログファイルとバイトオフセットを取り出すことができます。以下に例を示します。

```
postgres=# SELECT * FROM pg_xlogfile_name_offset(pg_stop_backup());
      file_name      | file_offset
-----+-----
00000001000000000000000D |      4039624
(1 row)
```

同様に、pg_xlogfile_nameは、トランザクションログファイル名のみを取り出します。指定したトランザクションログの位置が正確にトランザクションログファイルの境界であった場合、これらの両関数は前のトランザクションログファイルの名前を返します。通常これは、トランザクションログファイルのアーカイブ動作では好まれる動作です。前のファイルが現在のアーカイブで必要とする最後のファイルであるからです。

pg_xlog_location_diffは、2つのトランザクションログの位置の差分をバイト数で算出します。この関数はpg_stat_replicationや表9-59に示される関数と併用することで、レプリケーションの遅延の確認に使用できます。

これらの関数の正しい使用方法については、項24.3を参照してください。

9.26.4. リカバリ制御関数

表9-60に示される関数は、スタンバイサーバの現在のステータス情報を提供します。これらの関数はリカバリ中、および通常稼働時に実行することができるでしょう。

表 9-60. リカバリ情報関数

名前	戻り型	説明
pg_is_in_recovery()	bool	まだリカバリ実施中であれば真を返します。
pg_last_xlog_receive_location()	text	ストリーミングレプリケーションにより受信されディスクに書き込みされた、トランザクションログの最後の位置を取得します。ストリーミングレプリケーションが実施されている場合は、この値が単調に増加していくでしょう。リカバリが完了した場合、受信されディスクに書き込まれた最後のWALレコードの位置の値がそのまま残ります。ストリーミングレプリケーションが無効、もしくは開始されていない場合、この関数はNULLを返します。
pg_last_xlog_replay_location()	text	リカバリ中に再生された最後のトランザクションログの位置を取得します。リカバリが実施されている場合は、この値が単調に増加していくでしょう。リカバリが完了した場合は、リカバリ時に適用された最後のWALレコードの値がそのまま残ります。もしサーバがリカバリ無しで普通に起動された場合、この関数はNULLを返します。
pg_last_xact_replay_timestamp()	timestamp with time zone	リカバリ中に再生された最後のトランザクションのタイムスタンプを取得します。このタイムスタンプは、プライマリにて該当するトランザクションがコミット、もしくはアボートされた際のトランザクションログが生成された時間です。もしリカバリ中に何のトランザクションも再生されていない場合、この関数はNULLを返します。もしリカバリが完了した場合、この関数はリカバリ中に再生した最後のトランザクションの時間を静的に示し続けます。サーバがリカバリ処理無しに開始された場合、この関数はNULLを返します。

表9-61に示す関数は、リカバリの進行を制御する関数です。これらの関数はリカバリ中のみ実施することが可能です。

表 9-61. リカバリ制御関数

名前	戻り型	説明
pg_is_xlog_replay_paused()	bool	リカバリが停止中であれば真を返す
pg_xlog_replay_pause()	void	即座にリカバリを停止する
pg_xlog_replay_resume()	void	もしリカバリ停止中であれば再開する

リカバリ停止中は、それ以降のデータベースへの変更は適用されません。ホットスタンバイ側であれば、停止以降に発行されたSQLは、同じ一貫性を持ったデータベースのスナップショットを参照することができます。そしてリカバリが再開されるま

で、以降のSQLとプライマリへのSQLの衝突は発生しません。

もしストリーミングレプリケーションが無効の場合、停止状態はいつまでも問題なく継続するでしょう。一方、ストリーミングレプリケーションによりWALレコードの受信が継続されていた場合、停止時間、WALの生成速度、ディスクの残存容量によりますが、ディスク溢れが発生する可能性があります。

9.26.5. スナップショット同期関数

PostgreSQLはデータベースのセッションに対して、それらのスナップショットを同期させることが可能です。snapshotは、そのスナップショットを使用しているトランザクションがどのデータを可視とできるかを決定します。同期スナップショットは、2つかそれ以上のセッションにおいて、全く同じデータベース内容を見たい場合に必要となります。もし、2つのセッションが独立したそれぞれのトランザクションを開始しただけであれば、ある第3のトランザクションのコミットが、2つのトランザクションのSTART TRANSACTIONの狭間で実行される可能性があり、そのため一方のトランザクションではそのコミット結果が見え、他方では見えないという影響を受けてしまうでしょう。

このような問題を解決するため、PostgreSQLではトランザクションが使用しているスナップショットをエクスポートできるようになっています。エクスポートしたトランザクションが開かれ続けている限り、他のトランザクションがそれをインポートすることができます。そしてこれにより最初のトランザクションと正確に同じとなるデータベースの可視性を保証されます。ただし、これらの(スナップショットを共有している)トランザクションによって発生したデータベースへの変更は、コミットされていないトランザクションによる変更と同様に、(スナップショットを共有している)他のトランザクションには見えないままです。そのため、既存データに対しては同期されますが、それら自身による変更については通常の振る舞いをします。

スナップショットは、表9-62に示すpg_export_snapshot関数を用いてエクスポートされ、SET TRANSACTIONコマンドを用いてインポートされます。

表 9-62. スナップショット同期関数

名前	戻り値型	説明
pg_export_snapshot()	text	現在のスナップショットを保存し、その識別子を返す

pg_export_snapshot関数は現在のスナップショットを保存し、そのスナップショットを識別するtext文字列を返します。この文字列はスナップショットをインポートしたい(データベース外の)クライアントに渡されなければなりません。エクスポートしたトランザクションが終わるまでの間のみ、そのスナップショットをインポートすることができます。必要ならばトランザクションを複数回エクスポートすることができます。REPEATABLE READや上位の隔離レベルでは、トランザクションはその有効期間の間同じスナップショットを使用しますので、これはREAD COMMITTEDトランザクションでのみ有用であることに注意してください。一旦スナップショットをエクスポートしたトランザクションでは、PREPARE TRANSACTIONによるPREPARE TRANSACTIONを使用することができなくなります。

エクスポートしたトランザクションの使用方法の詳細についてはSET TRANSACTIONを参照してください。

9.26.6. データベースオブジェクト管理関数

表9-63で示された関数はデータベースオブジェクトのディスク領域を計算します。

表 9-63. データベースオブジェクト容量関数

名前	戻り型	説明
pg_column_size(any)	int	特定の値を格納するのに使用される（場合により圧縮された）バイト数
pg_database_size(oid)	bigint	指定されたOIDのデータベースで使用されるディスク容量
pg_database_size(name)	bigint	指定された名前のデータベースで使用されるディスク容量
pg_indexes_size(regclass)	bigint	指定されテーブルに付与されたインデックスで使用される総ディスク容量。
pg_relation_size(relation regclass, fork text)	bigint	指定されたテーブルまたはインデックスの指定されたフォーク ('main'、'fsm'または'vm') で使用されるディスク容量
pg_relation_size(relation regclass)	bigint	pg_relation_size(..., 'main') の省略表現
pg_size_pretty(bigint)	text	64ビット整数で表現されたバイト単位のサイズを可読性が高いサイズ単位の書式に変換
pg_size_pretty(numeric)	text	numeric値で表現されたバイト単位のサイズを可読性が高いサイズ単位の書式に変換

<code>pg_table_size(regclass)</code>	<code>bigint</code>	指定されたテーブルで使用される容量の内、すべてのインデックスを除外した(しかしTOAST、空き領域マップ、可視性マップを含む)ディスク総容量。
<code>pg_tablespace_size(oid)</code>	<code>bigint</code>	指定されたOIDを持つテーブル空間で使用されるディスク容量
<code>pg_tablespace_size(name)</code>	<code>bigint</code>	指定された名前を持つテーブル空間で使用されるディスク容量
<code>pg_total_relation_size(regclass)</code>	<code>bigint</code>	指定されたテーブルで使用される、すべてのインデックスとTOASTデータを含むディスク総容量

`pg_column_size`はどんな個別のデータ値を格納するのにも使用される領域を示します。

`pg_total_relation_size`は、テーブルまたはTOASTテーブルのOIDまたは名前を受け付け、指定されたテーブルと関連する全てのインデックスで使用される総ディスク容量を返します。この関数は`pg_table_size + pg_indexes_size`の結果と等しいです。

`pg_table_size`は、テーブルのOIDまたは名前を受け付け、インデックスを除いたテーブルのみで使用されるディスク容量を返します。(TOAST領域、空き領域マップ、可視性マップを含みます。)

`pg_indexes_size`は、テーブルのOIDまたは名前を受け付け、指定されたテーブル付与されている全てのインデックスで使用されるディスク容量を返します。

`pg_database_size`と`pg_tablespace_size`はデータベースまたはテーブル空間の名前またはOIDを受け付け、そこで使用される総容量を返します。

`pg_relation_size`はOIDもしくはテーブル名、インデックスもしくはtoastテーブルを受け付け、ディスク容量をバイト単位で返します。'main'を指定するか、第二番目の引数を除外するとその関係の主データフォークの容量を返します。'fsm'を指定すると、関係(リレーション)に関連した空き領域マップ(項56.3を参照)を返します。'vm'を指定すると、関係に関連した可視性マップ(項56.4を参照)の容量を返します。この関数は一つのフォークのみのサイズを示すことに注意して下さい。大方の目的では、`pg_total_relation_size`や`pg_table_size`などの関数を使う方が便利です。

`pg_size_pretty`は、適切にkB、MB、GB、もしくはTB単位を使用して目で見て判るようにその他の関数の1つの結果を整形するのに使用可能です。

上記の関数において、テーブルやインデックスを`regclass`引数として受け取って処理するものがありますが、この引数は単に`pg_class`システムカタログにあるテーブルやインデックスのOIDです。ただし、`regclass`データ型が自動で入力変換を行うため、ユーザが手動で該当するOIDを調べる必要はありません。リテラル定数のようにシングルクォートで囲んだテーブル名を記述するだけです。通常のSQL名に対する処理互換のため、テーブル名をダブルクォートで囲わない限り、テーブル名として入力された文字列は小文字に変換されます。

上記の関数に対し、既存オブジェクトに該当するOIDがないものが渡された場合はNULLが返されます。

表9-64 に示される関数は、データベースオブジェクトに関連する特定のディスクファイルを確認する際の手助けとなります。

表 9-64. データベースオブジェクト位置関数

名前	戻り型	説明
<code>pg_relation_filepath(relation regclass)</code>	<code>oid</code>	指定されたリレーションのファイルノード番号
<code>pg_relation_filepath(relation regclass)</code>	<code>text</code>	指定されたリレーションのファイルパス

`pg_relation_filepath`は、テーブル、インデックス、シーケンス、もしくはTOASTテーブルのOIDまたは名前を受け付け、現在それに充てられている"ファイルノード"を返します。ファイルノードは、リレーションに使用しているファイル名の基礎部分です。(詳しくは項56.1を参照して下さい。)ほとんどのテーブルについては、結果が`pg_class.relfilenode`と同じになります。ただし、いくつかのシステムカタログでは`relfilenode`が0になるため、これらのシステムカタログの正しいファイルノードを取得するには、この関数を使用しなければいけません。この関数は、ビューの様にストレージに格納されないリレーションが指定された場合はNULLを返します。

`pg_relation_filepath`は`pg_relation_filepath`と似ていますが、こちらはリレーションのファイルパス名(データベースクラスターのディレクトリであるPGDATAからの相対パス)を返します。

9.26.7. 汎用ファイルアクセス関数

表9-65 で示されている関数はサーバをホスティングしているマシン上のファイルに対し、生来のアクセスを提供します。データベースクラスターディレクトリと`log_directory`に存在するファイルのみがアクセス可能です。クラスターディレクトリ内のファイルに対して相対パスを、そしてログファイルに対しては`log_directory`構成設定に一致するパスを使用してください。

表 9-65. 汎用ファイルアクセス関数

名前	戻り型	説明
pg_ls_dir(dirname text)	setof text	ディレクトリ内容のリスト
pg_read_file(filename text [, offset bigint, length bigint])	text	テキストファイルの内容を返す
pg_read_binary_file(filename text [, offset bigint, length bigint])	bytea	ファイルの内容を返す
pg_stat_file(filename text)	record	ファイル情報を返す

pg_ls_dirは、特別なエントリである"."、および ".."を除いた、指定されたディレクトリの全ての名前を返します。

pg_read_fileは与えられたoffsetから始まり、最大lengthバイト（最初にファイルの終りに到達すればこれより少なくなります）テキストファイルの一部を返します。offsetが負の場合にはファイルの終りから数えた位置から読み出します。もしoffsetとlengthが省略された場合、ファイル全体が返されます。ファイルからのバイトの読み込みは、そのサーバ符号化方式に対する文字列と同様の解釈をされます。そのため、読み込んだバイト列がその符号化方式において有効でない場合にはエラーが投げられます。

pg_read_binary_fileは、結果がbytea値となり、従って符号化の検査がされないことを除き、pg_read_fileと似ています。convert_from関数と組み合わせることで、この指定した符号化方式でファイルの読み込みを行うことができます。

```
SELECT convert_from(pg_read_binary_file('file_in_utf8.txt'), 'UTF8');
```

pg_stat_fileはファイル容量、最終アクセス時刻、最終更新時刻、最後にファイルステータスを変更した時刻（これはUnixプラットフォームのみ）、ファイル作成時刻（Windowsのみ）およびもしディレクトリであればそれを示すbooleanを返します。典型的な使用法を示します。

```
SELECT * FROM pg_stat_file('filename');
SELECT (pg_stat_file('filename')).modification;
```

9.26.8. 勧告的ロック用関数

表9-66に示す関数は勧告的ロックを管理します。これらの関数の適切な使用方法についての詳細は、項13.3.4を参照してください。

表 9-66. 勧告的ロック用関数

名前	戻り型	説明
pg_advisory_lock(key bigint)	void	セッションレベルの排他勧告的ロックを獲得
pg_advisory_lock(key1 int, key2 int)	void	セッションレベルの排他勧告的ロックを獲得
pg_advisory_lock_shared(key bigint)	void	セッションレベルの共有勧告的ロックを獲得
pg_advisory_lock_shared(key1 int, key2 int)	void	セッションレベルの共有勧告的ロックを獲得
pg_advisory_unlock(key bigint)	boolean	セッションレベルの排他勧告的ロックを解放
pg_advisory_unlock(key1 int, key2 int)	boolean	セッションレベルの排他勧告的ロックを解放
pg_advisory_unlock_all()	void	現在のセッションで保持している全てのセッションレベルの勧告的ロックを解放
pg_advisory_unlock_shared(key bigint)	boolean	セッションレベルの共有勧告的ロックを解放
pg_advisory_unlock_shared(key1 int, key2 int)	boolean	セッションレベルの共有勧告的ロックの解放
pg_advisory_xact_lock(key bigint)	void	トランザクションレベルの排他勧告的ロックの獲得
pg_advisory_xact_lock(key1 int, key2 int)	void	トランザクションレベルの排他勧告的ロックの獲得
pg_advisory_xact_lock_shared(key bigint)	void	トランザクションレベルの共有勧告的ロックの獲得
pg_advisory_xact_lock_shared(key1 int, key2 int)	void	トランザクションレベルの共有勧告的ロックの獲得
pg_try_advisory_lock(key bigint)	boolean	可能ならばセッションレベルの排他勧告的ロックを獲得
pg_try_advisory_lock(key1 int, key2 int)	boolean	可能ならばセッションレベルの排他勧告的ロックを獲得
pg_try_advisory_lock_shared(key bigint)	boolean	可能ならばセッションレベルの共有勧告的ロックを獲得
pg_try_advisory_lock_shared(key1 int, key2 int)	boolean	可能ならばセッションレベルの共有勧告的ロックを獲得
pg_try_advisory_xact_lock(key bigint)	boolean	可能ならばトランザクションレベルの排他勧告的ロックの獲得
pg_try_advisory_xact_lock(key1 int, key2 int)		

int)	boolean	可能ならばトランザクションレベルの排他勧告的ロックの獲得
pg_try_advisory_xact_lock_shared(key bigint)	boolean	可能ならばトランザクションレベルの共有勧告的ロックの獲得
pg_try_advisory_xact_lock_shared(key1 int, key2 int)	boolean	可能ならばトランザクションレベルの共有勧告的ロックの獲得

pg_advisory_lockは、アプリケーションが定義したリソースをロックします。キーは単一の64ビットキー値、または、2つの32ビットキー（この2つのキー空間は重複しないことに注意）によって識別されます。もし、別のセッションが同一リソースに対するロックを保持している場合、関数はリソースが利用可能になるまで待機します。ロックは排他ロックです。複数のロック要求が待ち状態になります。ですので、同一リソースが3回ロックされた場合、他のセッションが使用できるように解放するためにはロック解除を3回行わなければなりません。

pg_advisory_lock_sharedの動作はpg_advisory_lockと同じですが、他のセッションの共有ロックと共有できるロックである点が異なります。排他ロック要求のみ締め出されます。

pg_try_advisory_lockはpg_advisory_lockと同様ですが、この関数の場合、ロックが利用可能になるまで待機しません。ロックを即座に取得しtrueを返すか、ロックを即座に獲得できなかった場合にfalseを返すかのいずれかです。

pg_try_advisory_lock_sharedの動作は pg_try_advisory_lockと同じですが、排他ロックではなく共有ロックの獲得を試みません。

pg_advisory_unlockは、事前に獲得したセッションレベルの勧告的排他ロックを解放します。ロックの解放に成功した場合、trueを返します。ロックを保持していない場合、falseを返し、さらに、SQL警告がサーバから発生します。

pg_advisory_unlock_sharedの動作はpg_advisory_unlockと同じですが、セッションレベルの勧告的共有ロックを解放する点が異なります。

pg_advisory_unlock_allは、現在のセッションで保持するセッションレベルの勧告的ロックをすべて解放します。（この関数は、クライアントとの接続がぶざまに切れた場合でも、セッション終了時に暗黙的に呼び出されます。）

pg_advisory_xact_lockの動作はpg_advisory_lockと同じですが、現在のトランザクションの終了時に自動的にロックが解放され、明示的なロックの解放はできません。

pg_advisory_xact_lock_sharedの動作はpg_advisory_lock_sharedと同じですが、現在のトランザクションの終了時に自動的にロックが解放され、明示的なロックの解放はできません。

pg_try_advisory_xact_lockの動作はpg_try_advisory_lockと同じですが、ロックが獲得できた場合は現在のトランザクションの終了時に自動的にロックが解放され、明示的なロックの解放はできません。

pg_try_advisory_xact_lock_sharedの動作はpg_try_advisory_lock_sharedと同じですが、ロックが獲得できた場合は現在のトランザクションの終了時に自動的にロックが解放され、明示的なロックの解放はできません。

9.27. トリガ関数

現在、PostgreSQLは、1つの組み込みトリガ関数`suppress_redundant_updates_trigger`を提供しており、通常の動作、つまりデータが変更されようがされまいがに関わらず常に更新をする場合とは正反対に、行のデータを実際に変更しない更新が起こらないようにします。（この通常の動作は、検査を必要としないため更新をより迅速に行い、場合によっては便利です。）

理想的には、通常実際レコード内のデータを変更しない更新の実行を避けるべきです。冗長な更新により、特に変更対象の多くのインデックスが存在する場合、無視できない不要な時間にかかるコストが発生することがあります。また、最後にはバキュームしなければならない不要行が場所を取るようになります。しかし、こうした状況をクライアント側で判定することは常に簡単ではありません。また、可能であったとしても、それを検知するための式の記述はエラーを招きがちです。他の方法として、`suppress_redundant_updates_trigger`を使用することがあります。これはデータを変更しない更新を飛ばします。しかしこの関数は注意して使用しなければなりません。このトリガはレコードごとに小さな、しかし僅かではない時間がかかります。このため、更新が影響するレコードのほとんどが実際には変更されない場合、このトリガは確実に更新の実行を低速にします。

`suppress_redundant_updates_trigger`関数は以下のようにテーブルに追加できます。

```
CREATE TRIGGER z_min_update
BEFORE UPDATE ON tablename
FOR EACH ROW EXECUTE PROCEDURE suppress_redundant_updates_trigger();
```

ほとんどの場合、それぞれの行に対しこのトリガを最後に起動させる必要が生じます。トリガは名前順に起動されることを判っているとして、テーブル上に存在する可能性のある他のトリガの名前の後続くようトリガ名を選択できます。

トリガ作成についてより詳細は[CREATE TRIGGER](#)を参照ください。

第 10 章 型変換

目次

- 10.1. [概要](#)
- 10.2. [演算子](#)
- 10.3. [関数](#)
- 10.4. [値の格納](#)
- 10.5. [UNION、CASE および関連する構文](#)

意図的かどうかにかかわらず、SQLの問い合わせでは1つの式の中に異なる型を混ぜ合わせた式を持つことができます。PostgreSQLは、異なる型が混在する式の評価に関して幅広い能力を持っています。

多くの場合、ユーザは型変換機構の詳細を理解する必要はありません。しかし、PostgreSQLによって暗黙的に行われる変換は問い合わせの結果に影響を及ぼします。必要に応じて、明示的な型変換を用いて結果を目的とするものに合わせることができます。

本章では、PostgreSQLの型変換機構とその規定について紹介します。特定のデータ型、使用できる関数と演算子についての情報については、[第8章](#)と[第9章](#)の関連する節を参照してください。

10.1. 概要

SQLは強く型付けされた言語です。つまり、各データ項目は、その動作と許される使用方法を決定するデータ型を所有しています。PostgreSQLには、他のSQLの実装よりもより一般的で柔軟性のある、拡張可能な型システムがあります。このために、PostgreSQLでのほとんどの型変換の動作は、特定の目的について勝手に作り上げられることなく一般的な規則で管理されています。これにより、ユーザ定義型についても型の混在する式を使用できます。

PostgreSQLのスキナ/パーサは字句要素を、整数、非整数値、文字列、識別子、キーワードという5個の基礎カテゴリに分解します。ほとんどの非数値型定数は、まず文字列にクラス分けされます。SQL言語定義では、文字列で型の名前を指定することを許していて、パーサが正しい手順に沿って処理を始められるようにPostgreSQLも採用しています。例えば、以下のような問い合わせを考えてみましょう。

```
SELECT text 'Origin' AS "label", point '(0,0)' AS "value";
```

```
label | value
-----+-----
Origin | (0,0)
(1 row)
```

この問い合わせは、`text` と `point` という2つの型を指定したリテラル定数を持ちます。文字列リテラルに型が指定されていない場合、後述するように、後の段階で解決されるようにとりあえず場所を確保するための型である `unknown` が割り当てられます。

PostgreSQLのパーサには、個別の型変換規則が必要な4つの基礎的なSQL構成要素があります。

関数呼び出し

PostgreSQL型システムの大部分は、高度な関数群によって構築されています。関数は複数の引数を取ることができます。PostgreSQLでは関数のオーバーロードが可能ですので、関数名だけでは呼び出すべき関数を一意に識別できません。パーサは、提供される引数のデータ型に基づいて、正しい関数を選択しなければなりません。

演算子

PostgreSQLでは、(引数が2つの)二項演算子と同様に、(引数が1つの)前置、後置単項演算子を持つ式が使用できます。関数と同様、演算子もオーバーロード可能ですので、正しい演算子を選択する時に同じ問題が存在します。

値の格納

SQLの `INSERT` と `UPDATE` 文は式の結果をテーブルの中に格納します。文内の式は対象となる列の型に一致する、または、変換できるものである必要があります。

UNION、CASE、および関連する構文

UNIONを構成する `SELECT` 文からの選択結果は全て、ある1つの列集合として現れなければいけませんので、各 `SELECT` 句の結果型は統一された集合に一致し変換できる必要があります。同様に、`CASE` 構文が全体として既知の出力型を持つようになるために、`CASE` 構文の結果式は共通の型に変換される必要があります。これは `ARRAY` 構文でも `GREATEST` 関数、`LEAST` 関数でも同じです。

システムカタログには、データ型間の変換(もしくはキャスト)が有効かどうかや、その変換の実行方法に関する情報を格納します。ユーザは `CREATE CAST` コマンドを使用してキャストを追加することができます(これは通常新しいデータ型を定義する時にまとめて行われます。組み込み型間のキャスト集合は注意深く作成されており、また、変更しないことが最善です)。

暗黙のキャストを持つデータ型間の処理において、パーサが適切なキャスト処理のより良い決定を行えるよう追加の自律機構を備えています。データ型は、`boolean`、`numeric`、`string`、`bitstring`、`datetime`、`timespan`、`geometric`、`network`、及びユーザ定義を含むいくつかの型カテゴリに分けられます。(表45-51の一覧を参照してください。独自の型カテゴリを作成可能なことに注意して下さい。)各カテゴリは、候補となる型の選択があった場合に、好ましいとされる1つ以上の好ましい型を持ちます。好ましい型の選択と利用可能な暗黙のキャストに注意すれば、曖昧な式(複数の解析結果候補を持つもの)が有効な方法で解決されることを保証することが可能です。

全ての型変換規則は次のようないくつかの基本的な考え方に基づいて設計されています。

- 暗黙的な変換は、意外な、あるいは予想できない結果を決して生成させてはなりません。
- 暗黙的な型変換を必要としない問い合わせの場合、パーサやエクゼキュータに余計なオーバーヘッドがあるべきではありません。つまり、問い合わせ文がきちんとまとめられ、型が既に一致するものになっていれば、パーサ内で余計な時間を費やさず、また、問い合わせに不要な暗黙的な型変換関数が使用されないように、問い合わせは処理されるべきです。

さらに、もし問い合わせが通常ならば関数を使った暗黙的な変換を要求していたものであり、そして、ユーザが正しい引数型を持つ関数を明示的に定義した場合、パーサはこの新しい関数を使い、古い関数を使用した暗黙的な変換を行わないようにしなければなりません。

[前のページ](#)[型変換](#)[ホーム](#)[上に戻る](#)[次のページ](#)[演算子](#)

10.2. 演算子

演算式に参照される特定の演算子は、以下の手順を用いて決定されます。 関連する演算子の優先順位によりどの下位式をどの演算子の入力と見なすかが決定されますので、この手順はこの優先順位により間接的な影響を受けることに注意して下さい。 詳細は[項4.1.6](#)を参照してください。

演算子における型の解決

- 1 11 `pg_operator` システムカタログから、調査の対象とする演算子を選択します。 スキーマ修飾がされていない演算子名が使用される場合（通常の場合）、現行の検索パスで可視になっていて、同一の名前と引数の数を持つ演算子が調査対象であるとみなされます（[項5.7.3](#)を参照してください）。 修飾された演算子名が与えられている場合、指定されたスキーマの演算子のみが調査対象とみなされます。
- 1 11 検索パスで引数のデータ型が同じである複数の演算子を検出した場合、そのパスで最初に検出された演算子のみを調査対象とみなします。 引数のデータ型が異なる演算子は、検索パス内の位置に関係なく、同じように調べられません。
- 1 11 正確に入力引数型を受け付ける演算子があるかどうか検査します。 該当する演算子があれば（調査される演算子の集合内で正確に一致するものは1つしかあり得ません）、それを使用します。
 - 1 11 二項演算子の1つの引数が`unknown`型であった場合、この検査のもう片方の引数と同一の型であると仮定します。 2つの`unknown`入力、もしくは`unknown`入力を伴う単項演算子が呼び出された場合、この段階では対を見つけることはありません。
- 1 11 最もよく合うものを検索します。
 - 1 11 演算子の候補のうち、入力値のデータ型が一致せず、また、（暗黙的な変換を使用して）一致するように変換できないものを破棄します。 `unknown` リテラルは、上記の目的で何にでも変換可能とみなされます。 1つの候補しか残らない場合、それを使います。 それ以外の場合は次の段階に進みます。
 - 1 11 全ての候補を検索し、入力型に最も正確に合うものを残します（この時、ドメインはその基本型と同一であるとみなします）。 正確に合うものが何もない場合は全ての候補を残します。 1つの候補しか残らない場合、それを使います。 それ以外の場合は次の段階に進みます。
 - 1 11 全ての候補を検索し、型変換が必要とされる所で（入力データ型カテゴリの）好ましい型を受け付けるものを残します。 好ましい型を受け付けるものが何もない場合は全ての候補を残します。 1つの候補しか残らない場合、それを使います。 それ以外の場合は次の段階に進みます。
 - 1 11 入力引数で`unknown`のものがあつた場合、それらの残った候補に引数位置で受け入れられる型カテゴリを検査します。 各位置において、候補が`string`カテゴリを受け付ける場合は、そのカテゴリを選択します（`unknown` 型のリテラルは文字列のようなものですので、この文字列への重み付けは適切です）。 そうでなければ、もし残った全ての候補が同じ型カテゴリを受け入れる場合はそのカテゴリを選択します。 そうでもない場合は、さらに手掛かりがなければ正しい選択が演繹されることができませんので、失敗となります。 ここで、選択された型カテゴリを受け付けられない演算子候補は破棄されます。 さらに、それらカテゴリ内の好ましい型を受け付ける候補が1つでもある場合、その引数の好ましい型ではないものを受け付ける候補は破棄されます。 これらの検査をどれも通らなかつたら全ての候補を残します。 1つの候補しか残らない場合、それを使います。 それ以外の場合は次の段階に進みます。
 - 1 11 もし`unknown`と既知の型の引数の両方があり、そして全ての既知の型の引数が同じ型を持っていた場合、`unknown`引数も同じ型であると仮定し、どの候補が`unknown`引数の位置にある型を受け付けることができるかを検査します。 正確に1つの候補がこの検査を通過した場合、それを使います。 それ以外は失敗します。

以下に例を示します。

例 10-1. 階乗演算子の型解決

階乗演算子として、`bigint`を引数とするものが標準カタログ内に1つのみ定義されています（!を後に付けます）。 スキャナは、以下の問い合わせ式の引数にまず`integer`型を割り当てます。

```
SELECT 40 ! AS "40 factorial";

-----
40 factorial
815915283247897734345611269596115894272000000000
(1 row)
```


パーサはオペランドを型変換し、問い合わせは以下と等価になります。

```
SELECT CAST(40 AS bigint) ! AS "40 factorial";
```

例 10-2. 文字列連結演算子の型解決

文字列類似構文は、文字列の作業の他、複雑な拡張型の作業にも使用されます。型の指定がない文字列は、類似演算子候補に一致します。

例えば、以下は指定がない引数が1つあります。

```
SELECT text 'abc' || 'def' AS "text and unknown";

text and unknown
-----
abcdef
(1 row)
```

この場合、パーサは両引数で`text`を取る演算子があるかどうかを検索します。この演算子は存在しますので、第二引数は`text`型として解釈されるものと仮定されます。

以下は型の指定がない2つの値の連結です。

```
SELECT 'abc' || 'def' AS "unspecified";

unspecified
-----
abcdef
(1 row)
```

この場合、問い合わせ内に型が指定されていないので、どの型を使用すべきかについての初期の指針がありません。ですから、パーサは全ての演算子候補を検索し、文字列カテゴリとビット列カテゴリ入力を受け付ける候補を見つけます。使用できる場合は文字列カテゴリが優先されますので、文字列カテゴリが選択され、それからその好ましい型である`text`が、不明のリテラルを解決する型として使用されます。

例 10-3. 絶対値と否定演算子の型解決

PostgreSQLの演算子カタログには、前置演算子`@`用に複数の項目があります。これは全て各種数値データ型に対する絶対値計算を実装するものです。その1つは、数値カテゴリの好ましい型である`float8`型用の項目です。したがって、PostgreSQLは、`unknown`の入力があった場合にこれを使用します。

```
SELECT @ '-4.5' AS "abs";

abs
-----
4.5
(1 row)
```

ここでシステムは、選択した演算子を適用する前に、`unknown`型のリテラルを`float8`へ暗黙的に型変換します。以下のよう`float8`が使用され、他の型が使用されていないことを検証することができます。

```
SELECT @ '-4.5e500' AS "abs";

ERROR:  "-4.5e500" is out of range for type double precision
```

一方、前置演算子`~`（ビット否定）は、整数データ型のみで定義され、`float8`用は定義されていません。ですから、`~`における上と同様の場合は、以下のような結果になります。

```
SELECT ~ '20' AS "negation";

ERROR:  operator is not unique: ~ "unknown"
HINT:  Could not choose a best candidate operator. You might need to add explicit type casts.
```

これは、システムが、複数の`~`演算子候補のうちどれが好ましいかを決定することができなかったため発生します。明示的なキャストを使用することで補助することができます。

```
SELECT ~ CAST('20' AS int8) AS "negation";

negation
-----
-21
(1 row)
```

例 10-4. 配列含有演算子の型解決

一方は既知でありもう一方は未知である入力に伴った演算子の解決のもう一つの例です。

```
SELECT array[1,2] <@ '{1,2,3}' as "is subset";

 is subset 
-----
 t
(1 row)
```

PostgreSQLの演算子カタログは、<@中置演算子のためのいくつかのエントリを持っていますが、 数値型配列を左側に受け付けることができるのは配列含有(`anyarray <@ anyarray`)と範囲含有(`anyelement <@ anyrange`)の2つのみです。 これらの多様な擬似データ型[項8.19](#)を参照は好ましいと見なされないため、このような方法ではパーサは曖昧さを解決することができません。 しかし、最後の解決規則では、未知の型のリテラルを別の入力と同じ型であると仮定するために数値配列とみなします。 今のところ2つのうち一つの演算子だけがマッチできるため、配列含有が選択されます。(範囲含有が選択された場合、演算子の右側にある文字列は正しい範囲型のリテラルではないため、エラーとなるでしょう。)

10.3. 関数

関数呼び出しによって参照される特定の関数は、以下の手順に従って解決されます。

関数における型の解決

- 1 11 `pg_proc` システムカタログから、調査の対象とする関数を選択します。スキーマ修飾がされていない関数名が使用される場合、現行の検索パスで可視になっていて、同一の名前と引数の数を持つ関数が調査対象であるとみなされます（[項5.7.3](#)を参照してください）。修飾された関数名が与えられている場合、指定されたスキーマの関数のみが調査対象とみなされます。
- 1 11 検索パスで、引数のデータ型が同じである複数の関数を検出した場合、そのパスで最初に検出された関数のみを調査対象とみなします。引数のデータ型が異なる関数は、検索パス内の位置に関係なく、同じように調べられます。
- 1 11 もし関数が `VARIADIC` 型の配列パラメータを伴って定義されており、そして `VARIADIC` キーワードを用いずに呼ばれた場合は、呼び出しに適合するよう、一つかそれ以上の要素の型に配列のパラメータを置き換えた形で扱われます。このような拡張後は、その関数は実際の引数の型を持つので、他の非可変長の引数を持つ関数と同一になるかもしれません。この場合、検索パスで先に見つかった関数が使われます。また、同じスキーマに2つの関数が見つかった場合は非可変長の関数が優先されます。
- 1 11 パラメータにデフォルト値を持つ関数は、デフォルト指定可能なパラメータ位置の0以上の省略されたどのような呼び出しに対しても適合すると見なされます。もし呼び出し時にこのような関数の一つ以上適合した場合、検索パスで先に見つかったものが使用されます。もし、デフォルト指定のない位置に同じパラメータ型を持つ関数（もしそれらが異なるデフォルト指定のあるパラメータのセットを持っていればあり得ます）が同じスキーマに2つ以上あった時は、どの関数を使うべきを決定できず、適合する関数が見つからなかったという "ambiguous function call" エラーが結果として返るでしょう。
- 1 11 正確に入力引数型を受け付ける関数があるかどうか検査します。該当する関数があれば（調査される関数の集合内で正確に一致するものは1つしかあり得ません）、それを使用します（`unknown` を含む場合は、この段階で一致するものは決して見つかりません）。
- 1 11 正確に一致するものが存在しなかった場合、その関数呼び出しが特別な型変換要求であるかどうかを確認します。これは、関数呼び出しがただ1つの引数を取り、関数名が何らかのデータ型の（内部的な）名前と同一である場合に発生します。さらに、その関数の引数は、`unknown` 型のリテラルか指定されたデータ型へのバイナリ変換可能な型か、型の入出力関数を適用することで指定された型に変換可能な型（つまり、変換が標準文字列型との間の変換である）であるかのいずれかでなければなりません。これらの条件に合う場合、関数呼び出しは `CAST` 仕様の形式と同様に扱われます。[1]
- 1 11 最適なものを検索します。
 - 1 11 関数の候補のうち、入力値のデータ型が一致せず、また、（暗黙的な変換を使用して）一致するように変換できないものを破棄します。`unknown` リテラルは、上記の目的で何にでも変換可能とみなされます。1つの候補しか残らない場合、それを使います。それ以外の場合は次の段階に進みます。
 - 1 11 全ての候補を検索し、入力型に最も正確に合うものを残します（この時、ドメインはその基本型と同一であるとみなします）。正確に合うものが何もない場合は全ての候補を残します。1つの候補しか残らない場合、それを使います。それ以外の場合は次の段階に進みます。
 - 1 11 全ての候補を検索し、型変換が必要とされるところで（入力データ型カテゴリの）好ましい型を受け付けるものを残します。好ましい型を受け付けるものが何もない場合は全ての候補を残します。1つの候補しか残らない場合、それを使います。それ以外の場合は次の段階に進みます。
 - 1 11 入力引数で `unknown` のものがあつた場合、それらの残った候補に引数位置で受け入れられる型カテゴリを検査します。各位置で候補が `string` カテゴリを受け付ける場合は、そのカテゴリを選択します（`unknown` 型のリテラルは文字列のようなものですので、この文字列への重み付けは適切です）。そうでなければ、もし残った全ての候補が同じ型カテゴリを受け入れる場合はそのカテゴリを選択します。そうでなければ、さらに手掛かりがなければ正しい選択が演繹されることができませんので、失敗となります。ここで、選択された型カテゴリを受け付けられない演算子候補は破棄されます。さらに、このカテゴリ内の好ましい型を受け付ける候補が1つでもある場合、その引数の好ましい型ではないものを受け付ける候補は破棄されます。これらの検査をどれも通らなかったら全ての候補を残します。1つの候補しか残らない場合、それを使います。それ以外の場合は次の段階に進みます。
 - 1 11 もし `unknown` と既知の型の引数の両方があり、そして全ての既知の型の引数が同じ型を持っていた場合、`unknown` 引数も同じ型であると仮定し、どの候補が `unknown` 引数の位置にある型を受け付けることができるかを検査します。正確に1つの候補がこの検査を通過した場合、それを使います。それ以外は失敗します。

この"最善一致"規則は演算子と関数の型解決で同一であることに注意してください。 以下に例を示します。

例 10-5. 丸め関数引数の型解決

2つの引数を取る`round`関数は1つしかありません 第1引数として`numeric`型、第2引数として`integer`型を取ります。 ですから、以下の問い合わせは自動的に、`integer`型の第1引数を`numeric`に変換します。

```
SELECT round(4, 4);

 round
-----
 4.0000
(1 row)
```

問い合わせはパーサによって実質以下のように変形されます。

```
SELECT round(CAST (4 AS numeric), 4);
```

小数点を持つ数値定数はまず`numeric`に割り当てられますので、以下の問い合わせでは型変換が不要です。そのためかなり効率的になる可能性があります。

```
SELECT round(4.0, 4);
```

例 10-6. 部分文字列関数の型解決

`substr`関数は複数存在します。 その1つは`text`と`integer`型を取ります。 型の指定がない文字列定数で呼び出した場合、システムは好ましいカテゴリ`string` (すなわち`text`型) の引数を受け付ける候補関数を選択します。

```
SELECT substr('1234', 3);

 substr
-----
      34
(1 row)
```

文字列が`varchar`型と宣言された場合、これはテーブルから取り出した場合が考えられますが、パーサはそれを`text`になるように変換しようと試みます。

```
SELECT substr(varchar '1234', 3);

 substr
-----
      34
(1 row)
```

これは以下になるようにパーサによって変換されます。

```
SELECT substr(CAST (varchar '1234' AS text), 3);
```

注意: パーサは`pg_cast`カタログから`text`と`varchar`がバイナリ互換、つまり、何らかの物理的な変換を行うことなく片方を受け付ける関数にもう片方を渡すことができることを学習します。 したがって、この場合実際に挿入される型変換呼び出しはありません。

また、`integer`型の引数でこの関数が呼び出された場合、パーサはそれを`text`に変換しようと試みます。

```
SELECT substr(1234, 3);
ERROR:  function substr(integer, integer) does not exist
HINT:   No function matches the given name and argument types. You might need to add explicit type casts.
```

`integer`は`text`への暗黙的なキャストを持たないため、これは失敗します。 成功させるには、以下のように明示的なキャストを行います。

```
SELECT substr(CAST (1234 AS text), 3);

 substr
-----
      34
(1 row)
```

注意

[1]

この処理の理由は、実際にはキャスト関数が存在しない状況において、関数形態のキャスト仕様をサポートすることで、キャスト関数が存在する場合、慣習的に出力型に困って名付けられます。 ですので、特殊な状況を持つ必要はありません。

ません。 詳細な解説については[CREATE CAST](#)を参照してください。

[前のページ](#)
演算子

[ホーム](#)
[上に戻る](#)

[次のページ](#)
値の格納

10.4. 値の格納

以下の手順に従って、テーブルに挿入される値は対象とする列のデータ型に変換されます。

値の格納における型変換

- 1 11 対象に正確に一致するかどうかをチェックします。
- 1 11 なければ、式を対象の型に変換してみます。もし2つの型の間のキャストが登録されている場合、これは成功します。もし式がunknown型リテラルの場合、リテラル文字列の内容は対象の型の入力変換ルーチンに与えられます。
- 1 11 対象の型に対してサイズ調整キャストがあるかどうかをチェックします。サイズ調整キャストは、ある型からその同じ型へのキャストです。pg_cast カタログに1つ見つかった場合は、格納先の列に収納する前に式に適用します。こうしたキャストを実装する関数は、常にinteger型のパラメータを1つ余計に取ります。このパラメータは、格納先の列のatttypmod値を受け付けます (atttypmodの解釈方法はデータ型によって異なりますが、典型的にはその宣言された長さです)。また、キャストが明示的か暗黙的かを示す、第三のbooleanパラメータを取ることもできます。サイズ検査や切り詰めなど、長さに依存したセマンティックスの適用について、キャスト関数が責任を持ちます。

例 10-7. character 格納における型変換

varchar(20) として宣言された対象の列への以下の文では、対象の大きさが正確に調整されることを示します。

```
CREATE TABLE vv (v character(20));
INSERT INTO vv SELECT 'abc' || 'def';
SELECT v, octet_length(v) FROM vv;
```

v	octet_length
-----+-----	
abcdef	20
(1 row)	

ここで実際に起こったのは、デフォルトで||演算子がtextの連結として解決できるように、2つのunknownリテラルがtextに解決されたということです。そして演算子のtext型の結果は対象の列の型に合うようにbpchar("空白が埋められる文字", character データ型の内部名)に変換されます (しかし、textからbpcharへバイナリ変換可能ですので、この型変換のために実際の関数呼び出しは挿入されません)。最後に、bpchar(bpchar, integer, boolean) サイズ調整関数がシステムカタログの中から見つかり、演算子の結果と格納する列の長さを適用します。この型特有の関数は必要とされる長さを検査し、空白の埋め込みを行います。

10.5. UNION、CASE および関連する構文

UNION SQL 構文は、似ていない可能性がある型を 1 つの検索結果になるように適合させなければなりません。解決アルゴリズムは 1 つの union 問い合わせの出力列ごとに適用されます。INTERSECT 構文と EXCEPT 構文は、UNION と同じ方法で、似ていない可能性がある型の解決を行います。CASE 構文と ARRAY 構文、VALUES、GREATEST、LEAST もまた、同一のアルゴリズムを使用して、その要素式を適合させ、結果のデータ型を選択します。

UNION、CASE および関連する構文の型解決

- 1 11 もし全ての入力値が同一型だった場合、unknown にはならず、その型として解決されます。そうでない場合は、リスト内のドメイン型を背後にある基本型で置換します。
- 1 11 もし全ての入力値が unknown 型だった場合、text 型（文字列カテゴリの好ましい型）として解決されます。そうでない場合は unknown 入力は無視します。
- 1 11 もし unknown ではない入力値が全て同じ型カテゴリでなければ失敗します。
- 1 11 最初の unknown ではない入力データ型を選択します。もし存在すれば、このデータ型はそのカテゴリの好ましい型です。
- 1 11 さもなくば、すべての先行する非 unknown の入力値を暗黙的に変換させることができる、最後の unknown ではない入力型を選択します。（少なくともリストの先頭の型はこの条件を満たす必要がありますので、常にこのような型は存在します。）
- 1 11 全ての入力値を選択された型に変換します。指定された入力から選択された型への変換が存在しない場合は失敗します。

以下に例を示します。

例 10-8. Union における指定された型の型解決

```
SELECT text 'a' AS "text" UNION SELECT 'b';

 text
-----
 a
 b
(2 rows)
```

ここで、unknown 型のリテラル 'b' は text へと解決されます。

例 10-9. 簡単な Union における型解決

```
SELECT 1.2 AS "numeric" UNION SELECT 1;

 numeric
-----
      1
     1.2
(2 rows)
```

numeric 型のリテラル 1.2 と integer 型の値 1 は、暗黙的に numeric にキャスト可能です。したがって、この型が使用されます。

例 10-10. 転置された UNION における型解決

```
SELECT 1 AS "real" UNION SELECT CAST('2.2' AS REAL);

 real
-----
      1
     2.2
(2 rows)
```

ここで、real 型を暗黙的に integer 型にキャストすることはできませんが、integer 型を暗黙的に real 型にキャストすることはできるため、UNION の結果データ型は real 型として解決されます。

第 11 章 インデックス

目次

- 11.1. [序文](#)
- 11.2. [インデックスの種類](#)
- 11.3. [複数列インデックス](#)
- 11.4. [インデックスと ORDER BY](#)
- 11.5. [複数のインデックスの組み合わせ](#)
- 11.6. [一意インデックス](#)
- 11.7. [式に対するインデックス](#)
- 11.8. [部分インデックス](#)
- 11.9. [演算子クラスと演算子族](#)
- 11.10. [インデックスと照合順序](#)
- 11.11. [インデックス使用状況の検証](#)

インデックスは、データベースの性能を向上させるための一般的な方法です。 データベースサーバでインデックスを使用すると、インデックスを使用しない場合に比べてかなり速く、特定の行を検出し抽出することができます。 しかし、インデックスを使用すると、データベースシステム全体にオーバーヘッドを追加することにもなるため、注意して使用する必要があります。

11.1. 序文

次のようなテーブルを考えてみましょう。

```
CREATE TABLE test1 (  
    id integer,  
    content varchar  
);
```

アプリケーションはこの形式の多くの問い合わせを発行します。

```
SELECT content FROM test1 WHERE id = constant;
```

高度な準備を行っていないければ、システムで一致する項目を全て検出するためには通常、`test1` テーブル全体を1行ごとにスキャンする必要があります。`test1` に数多くの行があり、その問い合わせで返されるのが数行（おそらく0行か1行）しかない場合、これは明らかに効率が悪い方法と言えます。システムがインデックスを `id` 列上で維持するように指示されていれば、一致する行を検出するのにより効率の良い方法を使うことができます。例えば、検索ツリーを数層分検索するだけで済む可能性もあります。

ほとんどのノンフィクション小説で、同じような手法が使われています。読者が頻繁に調べる用語および概念は、その本の最後にアルファベット順に索引としてまとめられています。その本に興味を持った読者は、索引（インデックス）を調べ、比較的速く簡単に該当するページを開くことができるため、見たい場所を探すために本全部を読む必要はありません。読者がよく調べそうな項目を予想するのが著者の仕事であるように、どのインデックスが実用的であるかを予測するのはデータベースプログラマの仕事です。

上述のように `id` 列にインデックスを作成する場合は、以下のようなコマンドが使用できます。

```
CREATE INDEX test1_id_index ON test1 (id);
```

`test1_id_index` というインデックス名には、何を選んでも構いませんが、そのインデックスを何のために作成したかを後で思い出せるような名前を選ぶべきです。

インデックスを削除するには、`DROP INDEX` コマンドを使用します。テーブルのインデックスは、いつでも追加および削除できます。

いったんインデックスを作成すれば、それ以上の処理は必要はありません。システムは、テーブルが変更される時インデックスを更新し、シーケンシャルスキャンよりもインデックススキャンを行うことがより効率的と判断した場合、問い合わせでインデックスを使用します。しかし、問い合わせプランナで情報に基づいた判断をするためには、定期的に `ANALYZE` コマンドを実行し、統計情報を更新する必要があるかもしれません。インデックスが使われているかどうか、およびプランナがインデックスを使わないと判断した状況および理由を調べる方法については、[第14章](#)を参照してください。

インデックスは、`UPDATE` や `DELETE` コマンドの検索条件でも使用できます。インデックスは結合問い合わせでより多く使用されます。したがって、結合条件で記述されている列にインデックスを定義すれば、結合を伴った問い合わせにかかる時間もかなり短縮できます。

大規模テーブルに対するインデックス作成が長期にわたる可能性があります。デフォルトでPostgreSQLはインデックス作成と並行してテーブルを読み取る（`SELECT` 文）ことができますが、書き込み（`INSERT`、`UPDATE`、`DELETE`）はインデックス作成が終わるまでブロックされます。これは多くの運用環境では受け入れられません。インデックス作成中でも並行して書き込みできるようにすることができますが、いくつか注意しなければならないことがあります。[インデックスの同時作成](#)の情報を参照してください。

インデックスが作成された後、システムでは、テーブルとインデックスとの間で常に同期を取っておく必要があります。これにより、データ操作の処理にオーバーヘッドが加わります。したがって、めったに使用されないインデックスや、まったく使用されなくなったインデックスは、削除しておいた方が良いでしょう。

11.2. インデックスの種類

PostgreSQLでは、B-tree、Hash、GiST、SP-GiST、GINといった複数の種類のインデックスを使用可能です。インデックスの各種類は、異なる種類の問い合わせに最も適した、異なるアルゴリズムを使用します。デフォルトでCREATE INDEX コマンドは、B-treeインデックスを作成し、それはほとんどの一般的状況に適合します。

B-treeインデックスは、ある順番でソート可能なデータに対する等価性や範囲を問い合わせることを扱うことができます。具体的には、PostgreSQLの問い合わせプランナは、インデックスの付いた列を次の演算子を使用して比較する場合に、B-treeインデックスの使用を検討します。

<
<=
=
>=
>

また、BETWEENやINなどのこれらの演算子の組み合わせと等価な式をB-treeインデックス検索で実装することができます。インデックスの付いた列に対するIS NULLやIS NOT NULLでもB-treeインデックスを使用することができます。

オプティマイザは、パターンマッチ演算子LIKE、~を含む問い合わせでも、そのパターンが定数であり、先頭文字列を指定している場合B-treeインデックスを使用することができます。例えば、col LIKE 'foo%'またはcol ~ '^foo'です。col LIKE '%bar'では使用されません。しかし、データベースがCロケールを使用していない場合、パターンマッチ問い合わせのインデックス付けをサポートする特別な演算子クラスでインデックスを作成しなければなりません。後述の項11.9を参照してください。なお、ILIKEと~*でもB-treeインデックスを使用することができますが、パターンが英字以外の文字、つまり、大文字小文字の違いの影響がない文字で始まる場合のみです。

B-treeインデックスをソートされた順序でデータを受けとるために使用することもできます。これは常に単純なスキャンとソート処理より高速になるものではありませんが、よく役に立つことがあります。

ハッシュインデックスは単純な等価性比較のみを扱うことができます。問い合わせプランナでは、インデックスの付いた列を=演算子を使用して比較する場合に、ハッシュインデックスの使用を検討します。ハッシュインデックスを作成するには、以下のようなコマンドを使用してください。

```
CREATE INDEX name ON table USING hash (column);
```

注意

現在ハッシュインデックス操作はWALに記録されません。そのため、データベースクラッシュの後ハッシュインデックスをREINDEXで再構築しなければならない可能性があります。また、最初のバックアップを取得して以降、ストリーミングレプリケーションやファイルベースのレプリケーションでは変更が反映されないため、その後ハッシュインデックスを使うクエリは誤った結果を返します。これらの理由により、ハッシュインデックスの使用は現在お勧めできません。

GiSTインデックスは単一種類のインデックスではなく、多くの異なるインデックス戦略を実装することができる基盤です。したがって、具体的なGiSTインデックスで使用できる演算子はインデックス戦略（演算子クラス）によって異なります。例えば、PostgreSQLの標準配布物には、複数の二次元幾何データ型用のGiST演算子クラスが含まれています。これらの演算子を使用してインデックス付けされた問い合わせをサポートします。

<<
&<
&>
>>
<<|
&<|
|&>
|>>
@>
<@
~=
&&

（これらの演算子の意味については項9.11を参照してください。）他の多くのGiST演算子クラスがcontrib群や別のプロジェクトとして利用可能です。詳細は第53章を参照してください。

GiSTインデックスは以下のような"最近傍"検索を最適化する機能も持ちます。

```
SELECT * FROM places ORDER BY location <-> point '(101,456)' LIMIT 10;
```

これは指定された対象地点に最も近い10箇所を見つけ出します。繰り返しますが、これができるかどうかは使用される特定の演算子クラスに依存します。

SP-GiSTインデックスは、GiSTインデックスと同様に様々な種類の検索を支援する基盤を提供します。SP-GiST索引は広域な異なる不均衡でディスクベースのデータ構造、つまり、四分木、kd木、接尾辞木のような実装を認めます。例えば、PostgreSQL標準配布物には、以下の演算子を使用する問い合わせに対するインデックスをサポートする2次元の点用のSP-GiST用の演算子クラスが含まれています。

```
<<
>>
~=
<@
<^
>^
```

(演算子の意味は以下を参照してください[項9.11](#)) 詳細は [第54章](#)を参照してください。

GINは転置インデックスであり、例えば配列など複数のキーを持つ値を扱うことができます。GiSTやSP-GiST同様、GINも多くの異なるユーザ定義のインデックス戦略を持つことができ、GINが使用できる具体的な演算子はインデックス戦略によって変化します。例えば、PostgreSQL標準配布物には、1次元配列用のGIN演算子クラスが含まれています。これは、以下の演算子を使用する問い合わせに対するインデックスをサポートします。

```
<@
@>
=
&&
```

(これらの演算子の意味については[項9.18](#)を参照してください。) 他の多くのGIN演算子クラスはcontrib群または別のプロジェクトで利用可能です。 詳細は[第55章](#)を参照してください。

11.3. 複数列インデックス

インデックスは、テーブルの1つ以上の列に定義することができます。例えば、以下のようなテーブルがあるとします。

```
CREATE TABLE test2 (  
    major int,  
    minor int,  
    name varchar  
);
```

(例えば、`/dev`ディレクトリの内容をデータベースに保持していて) 頻繁に下記のような問い合わせを発行するとします。

```
SELECT name FROM test2 WHERE major = constant AND minor = constant;
```

このような場合、`major`および`minor`という2つの列に1つのインデックスを定義する方が適切かもしれません。

```
CREATE INDEX test2_mm_idx ON test2 (major, minor);
```

現在、B-tree、GiST、およびGINインデックス型でのみ、複数列インデックスをサポートしています。最高32列まで指定可能です。(この上限は、PostgreSQLを構築する際に変更可能です。 `pg_config_manual.h` ファイルを参照してください。)

複数列に対するB-treeインデックスをインデックス対象列の任意の部分集合を含む問い合わせ条件で使用することができます。しかし、先頭側の(左側)列に制約がある場合に、このインデックスはもっとも効率的になります。正確な規則は、先頭側の列への等価制約、および、等価制約を持たない先頭列への不等号制約がスキャン対象のインデックス範囲を制限するために使用されます。これらの列の右側の列に対する制約は、このインデックス内から検査されます。ですので、テーブルアクセスを適切に抑えますが、スキャンされるインデックスの範囲を減らしません。例えば、`(a, b, c)`に対するインデックスがあり、`WHERE a = 5 AND b >= 42 AND c < 77`という問い合わせ条件があったとすると、`a = 5`かつ`b = 42`を持つ項目を先頭に、`a = 5`となる最後の項目までのインデックスをスキャンしなければなりません。`c >= 77`を持つインデックス項目は飛ばされますが、スキャンを行わなければなりません。このインデックスは原理上、`a`に対する制約を持たない、`b`および`c`、あるいは、`b`または`c`を制約に持つ問い合わせで使用することができます。しかし、インデックス全体がスキャンされますので、ほとんどの場合、プランナはインデックスの使用よりもシーケンシャルテーブルスキャンを選択します。

複数列GiSTインデックスは、インデックス対象列の任意の部分集合を含む問い合わせ条件で使用することができます。他の列に対する条件は、インデックスで返される項目を制限します。しかし、先頭列に対する条件が、インデックスのスキャン量を決定するもっとも重要なものです。先頭列の個別値がわずかな場合、他の列が多くの個別値を持っていたとしても、相対的にGiSTインデックスは非効率的になります。

複数列GINインデックスは、インデックス対象列の任意の部分集合を含む問い合わせ条件で使用することができます。B-TreeやGiSTと異なり、インデックス検索の効果はどのインデックス列が問い合わせ条件で使用されているかに関係なく同じです。

当然ながら、インデックス種類に対して適切な演算子を各列に使用しなければなりません。他の演算子を含む句は考慮されません。

複数列インデックスは慎重に使用する必要があります。多くの場合、単一列のインデックスで十分であり、また、その方がディスク領域と時間を節約できます。テーブルの使用方法が極端に様式化されていない限り、3つ以上の列を使用しているインデックスは、ほぼ間違いなく不適切であると言えます。異なるインデックス構成の利点に関するこの他の説明について[項11.5](#)も参照してください。

11.4. インデックスとORDER BY

単に問い合わせによって返される行を見つけ出すだけではなく、インデックスは、その行を指定した順番で取り出すことができます。これにより、別途ソート処理を行うことなく、問い合わせのORDER BY指定に従うことが可能です。PostgreSQLが現在サポートするインデックスの種類の中で、B-Treeのみがソート出力を行うことができます。他の種類のインデックスでは指定なし、または、実装固有の順序で一致した行を返します。

プランナは、指定に一致し利用可能なインデックスでスキャンする、または、テーブルを物理的な順番でスキャンし明示的なソートを行うことにより、ORDER BY指定を満足させるかを考慮します。テーブルの大部分のスキャンが必要な問い合わせでは、後に発生するシーケンシャルなアクセスパターンのために要求されるディスクI/Oが少ないため、インデックスを使用するよりも、明示的なソートの方が高速です。数行を取り出す必要がある場合のみ、インデックスの方が有用になります。ORDER BYとLIMIT *n*が組み合わされた場合が、重要かつ特別です。先頭の*n*行を識別するために、明示的なソートを全データに対して行う必要があります。しかし、もしORDER BYに合うインデックスが存在すれば、残りの部分をスキャンすることなく、先頭の*n*行の取り出しを直接行うことができます。

デフォルトでは、B-Treeインデックスは項目を昇順で格納し、NULLを最後に格納します。これは、*x*列に対するインデックスの前方方向のスキャンでORDER BY *x*（より冗長に言えばORDER BY *x* ASC NULLS LAST）を満たす出力を生成することを意味します。また、インデックスを後方方向にスキャンすることもでき、この場合、ORDER BY *x* DESC（より冗長に言えばORDER BY *x* DESC NULLS FIRST。NULLS FIRSTがORDER BY DESCのデフォルトだからです。）を満たす出力を生成します。

インデックスを作成する時に、以下のようにASC、DESC、NULLS FIRST、NULLS LASTオプションを組み合わせて指定することにより、B-Treeインデックスの順序を調整することができます。

```
CREATE INDEX test2_info_nulls_low ON test2 (info NULLS FIRST);
CREATE INDEX test3_desc_index ON test3 (id DESC NULLS LAST);
```

昇順かつNULL先頭という順で格納されたインデックスは、スキャンされる方向に依存してORDER BY *x* ASC NULLS FIRSTまたはORDER BY *x* DESC NULLS LASTを満たすことができます。

4つの全方向を提供する理由が何か、後方方向へのスキャンの可能性があることを考慮した2方向で、すべての種類のORDER BYを網羅できるのではないかと疑問を持つかもしれません。単一系列に対するインデックスでは、このオプションは実際冗長ですが、複数列に対するインデックスでは有用になります。(*x*, *y*) という2つの列に対するインデックスを仮定します。これを前方方向にスキャンすればORDER BY *x*, *y*を満たし、後方方向にスキャンすればORDER BY *x* DESC, *y* DESCを満たします。しかし、ORDER BY *x* ASC, *y* DESCをよく使用しなければならないアプリケーションが存在する可能性があります。簡素なインデックスからこの順序を取り出す方法がありません。しかし、インデックスが(*x* ASC, *y* DESC) または(*x* DESC, *y* ASC) として定義されていれば、取り出すことができます。

明確なことですが、デフォルト以外のソート順を持つインデックスはかなり特殊な機能です。しかし、特定の問い合わせにおいては恐ろしいほどの速度を向上させることがあります。こうしたインデックスを維持する価値があるかどうかは、特殊なソート順を要求する問い合わせを使用する頻度に依存します。

11.5. 複数のインデックスの組み合わせ

単一のインデックススキャンは、インデックスの列をその演算子クラスの演算子で使用する問い合わせ句と、それをAND結合したもののみ使用されます。例えば、(a, b)というインデックスとWHERE a = 5 AND b = 6という問い合わせでは、インデックスが使用されます。しかし、WHERE a = 5 OR b = 6のような問い合わせではインデックスは直接使用されません。

幸いにも、PostgreSQLは、単一のインデックススキャンでは実装できない場合を扱うために、複数のインデックス（同じインデックスの複数回使用を含む）を組み合わせる機能を持ちます。システムは複数のインデックススキャンを跨がる、条件のANDまたはORを形成します。例えば、WHERE x = 42 OR x = 47 OR x = 53 OR x = 99という問い合わせは、問い合わせ句の1つを使用してx上のインデックスをスキャンする4つのスキャンに分割することができます。その後、これらのスキャンの結果はOR演算でまとめられ、結果を生成します。他の例としてxとyに別個のインデックスがある場合を考えます。WHERE x = 5 AND y = 6のような問い合わせに対して取り得る実装は、適切な問い合わせ句で各インデックスを使用し、インデックスの結果をANDでまとめ、結果行を識別することです。

複数のインデックスを組み合わせるために、システムは必要なインデックスそれぞれをスキャンし、インデックス条件に一致するものと報告されたテーブル行の位置を与えるためにメモリ上にビットマップを準備します。その後、このビットマップは問い合わせで必要とされたように、ANDまたはOR演算されます。最後に、実際のテーブル行がアクセスされ、返されます。テーブル行は物理的な順番でアクセスされます。ビットマップにこの順番で格納されているからです。これは、元のインデックスの順序が失われていることを意味します。そのため、もし問い合わせがORDER BY句を持つ場合、この他のソート手続きが必要となります。この理由、および、追加のインデックススキャンそれぞれのために余計な時間が加わることから、プランナは追加のインデックスが同様に使用できる場合であっても、単純なインデックススキャンを選択することがあります。

もっとも単純なアプリケーション以外のほとんどすべてのアプリケーションでは、インデックスの有用な組み合わせはいろいろあります。このため、データベース開発者は妥協点を探してどのようなインデックスを提供するかを決定しなければなりません。複数列インデックスが最善な場合がありますし、別々のインデックスを作成し、インデックスの組み合わせ機能に依存する方が優れている場合もあります。例えば、作業にx列のみを含む場合とy列のみを含む場合、両方の列を含む場合が混在する問い合わせが含まれる場合、xとyに対し、別個に2つのインデックスを作成し、両方の列を使用する問い合わせを処理する時にインデックスの組み合わせに依存することを選ぶことができます。また、(x, y)に対する複数列インデックスを作成することもできます。両方の列を含む問い合わせでは、通常このインデックスの方がインデックスの組み合わせよりも効率的です。しかし、[項11.3](#)で説明した通り、yのみを含む問い合わせではほとんど意味がありません。従って、このインデックスのみとすることはできません。複数列インデックスとyに対する別のインデックスの組み合わせがかなりよく役に立ちます。xのみを含む問い合わせでは、複数列インデックスを使用することができます。しかし、これはより大きくなりますので、xのみインデックスよりも低速になります。最後の別方法は、3つのインデックスすべてを作成することです。しかしこれはおそらく、テーブルの検索頻度が更新頻度よりもかなり高く、3種類の問い合わせすべてが良く使用される場合のみ合理的です。問い合わせの中の1つの頻度が他よりも少なければ、おそらく良く使用される種類にもっとも合うように2つだけインデックスを作成した方がよいでしょう。

11.6. 一意インデックス

インデックスは、列値の一意性や、複数列を組み合わせた値の一意性を強制するためにも使用できます。

```
CREATE UNIQUE INDEX name ON table (column [, ...]);
```

現在、一意インデックスとして宣言できるのはB-treeインデックスのみです。

一意インデックスが宣言された場合、同じインデックス値を有する複数のテーブル行は許されなくなります。 NULL 値は同じ値とはみなされません。 複数列の一意インデックスは、複数の行でインデックス列の全てが同一の場合のみ拒絶されます。

PostgreSQLでは、テーブルに一意性制約またはプライマリキーが定義されると、自動的に一意インデックスを作成します。 このインデックスが、プライマリキーや一意性制約（適切ならば複数列のインデックスで）となる列に対して作成され、この制約を強制する機構となります。

注意: テーブルに一意性制約を追加するためによく使用される方法は、`ALTER TABLE ... ADD CONSTRAINT` です。一意性制約を課すためのインデックスの用法は、直接アクセスしてはならない実装の細部とみなされることがあります。しかし、手作業で一意列に対しインデックスを作成する必要がないことには注意してください。これは、単に自動作成されるインデックスを二重にするだけです。

11.7. 式に対するインデックス

インデックス列は、基礎をなすテーブルにある列である必要はなく、そのテーブルの1つ以上の列から計算される関数やスカラ式とすることもできます。この機能は、ある演算結果に基づいた高速テーブルアクセスを行う時に有用です。

例えば、大文字小文字を区別せずに比較するための一般的な方法である、`lower` 関数での使用例を以下に示します。

```
SELECT * FROM test1 WHERE lower(coll) = 'value';
```

`lower(column)` 関数の結果にインデックスが定義されていれば、この問い合わせでインデックスを使用することができます。

```
CREATE INDEX test1_lower_coll_idx ON test1 (lower(coll));
```

このインデックスを `UNIQUE` と宣言したとすると、`coll` の値が同一となる行だけでなく、`coll` の大文字小文字だけが違う行の生成を防ぐことになります。したがって、式に対するインデックスを使用して、単なる一意性制約では定義できないような制約を強制することができます。

別の例として、以下のような問い合わせが頻繁に行われる場合を考えます。

```
SELECT * FROM people WHERE (first_name || ' ' || last_name) = 'John Smith';
```

この場合、以下のようなインデックスを作成する価値があるでしょう。

```
CREATE INDEX people_names ON people ((first_name || ' ' || last_name));
```

2 番目の例に示すように `CREATE INDEX` コマンドの構文は通常、インデックス式を括弧で括る必要があります。最初の例のように、式が単なる関数呼び出しの場合には括弧を省略することができます。

派生した式が、行が挿入、更新される度に実行されなければなりませんので、インデックス式は相対的に見て維持が高価です。しかし、インデックス式はインデックスを使用する検索の間で再計算されません。上の両方の例では、システムは問い合わせを単なる `WHERE indexedcolumn = 'constant'` と理解しますので、この検索速度は他の単純なインデックス問い合わせと同じです。したがって、式に対するインデックスは取り出し速度が挿入、更新速度より重要な場合にのみ有用です。

11.8. 部分インデックス

部分インデックスとは、テーブルの部分集合に構築されるインデックスです。部分集合は、（部分インデックスの述語と呼ばれる）条件式で定義されます。部分インデックスには、その述語を満たすテーブル行のみに対するエントリが含まれます。部分インデックスは特別な機能です。しかし、これらが有用となる状況が複数あります。

部分インデックスを利用する主な目的は、頻出値に対してインデックスを作成しないようにすることです。（テーブル全体の行のうち、数パーセント以上を占める）頻出値を検索する問い合わせでは、いかなる場合でもインデックスを使用しないため、インデックスにそれらの行を持ち続けることは全く意味がありません。これによりインデックスのサイズが小さくなりますので、インデックスを使用する問い合わせが速くなります。また、インデックスを更新する必要のないケースも生じるため、テーブルを更新する作業の多くも速くなります。例11-1にこの概念に基づいた用例を示します。

例 11-1. 頻出値を除外するための部分インデックスの作成

ウェブサーバのアクセスログをデータベースに格納しているとします。多くのアクセスは、社内のIPアドレスの範囲内から発信されています。しかし、範囲外のアドレス（例えば、社員がダイアルアップ接続している場所）からの発信もあります。主に範囲外からのアクセスをIPアドレスで検索する場合、社内のサブネットに該当するIPアドレスの範囲にインデックスを作成する必要はないでしょう。

以下のようなテーブルがあると想定します。

```
CREATE TABLE access_log (  
    url varchar,  
    client_ip inet,  
    ...  
);
```

この例に適する部分インデックスを作成するには、以下のようなコマンドを使用します。

```
CREATE INDEX access_log_client_ip_ix ON access_log (client_ip)  
WHERE NOT (client_ip > inet '192.168.100.0' AND  
           client_ip < inet '192.168.100.255');
```

このインデックスを使用できる問い合わせの典型的な例は、以下のようなものです。

```
SELECT *  
FROM access_log  
WHERE url = '/index.html' AND client_ip = inet '212.78.10.32';
```

また、以下のような問い合わせの場合、このインデックスは使用できません。

```
SELECT *  
FROM access_log  
WHERE client_ip = inet '192.168.100.23';
```

このような部分インデックスを使用するには、あらかじめ頻出値が何であるかを知っている必要があることに注意してください。値の分布が変わらない場合に、このような部分インデックスが最善です。新しいデータの分布を調整するためにときたまインデックスを再作成することも可能ですが、これはメンテナンス作業を増やしてしまいます。

部分インデックスを使用する有効な他の方法としては、一般的な問い合わせに必要な値をインデックスから取り除くことです。例11-2を参照してください。この方法の利点は上で示したものと同じです。ただ、この方法を使用すると、インデックススキャンが適している場合でも、“必要のない”値へのインデックスを介したアクセスが防止されてしまいます。以上のことから明白なように、このようなケースで部分インデックスを作成する際は、細心の注意を払い、十分な検証を行う必要があります。

例 11-2. 必要のない値を除外するための部分インデックスの作成

支払済み注文書および未支払注文書からなる、1つのテーブルがあるとします。そして、未支払注文書の方がテーブル全体に対する割合が小さく、かつその部分へのアクセス数が最も多かったとします。このような場合、未支払の行のみにインデックスを作成することにより、性能を向上させることができます。インデックスの作成には、以下のようなコマンドを使用します。

```
CREATE INDEX orders_unbilled_index ON orders (order_nr)  
WHERE billed is not true;
```

このインデックスを使用する問い合わせの例としては、次のものが考えられます。

```
SELECT * FROM orders WHERE billed is not true AND order_nr < 10000;
```

しかし、このインデックスは、`order_nr`をまったく使用しない問い合わせでも使用することができます。以下は、その例です。

```
SELECT * FROM orders WHERE billed is not true AND amount > 5000.00;
```

この問い合わせでは、システムがインデックス全体を検索する必要があるため、`amount`列に部分インデックスを作成した場合ほど効率は良くありません。しかし、未支払注文書データが比較的少ない場合は、この部分インデックスを未支払注文書を検出するためだけに使用した方が効率が良い可能性があります。

以下の問い合わせでは、このインデックスを使用できないことに注意してください。

```
SELECT * FROM orders WHERE order_nr = 3501;
```

注文番号3501は支払済みかもしれませんし、未支払かもしれないからです。

[例11-2](#)でもわかるように、インデックスが付けられた列名と、述語で使用されている列名は、一致している必要はありません。PostgreSQLでは、インデックス付けされるテーブルの列だけが含まれているのなら、任意の述語で部分インデックスを使用できます。しかし、この述語は、インデックスを使用させたい問い合わせの条件と一致する必要があることに留意してください。正確に言うと、部分インデックスを問い合わせで使用できるのは、インデックスの述語が問い合わせのWHERE条件に数学的に当てはまるとシステムが判断できる場合のみです。PostgreSQLには、異なった形式で記述された述語が数学的に同等のものであると判断できるような、洗練された定理証明機能はありません。（そのような汎用的な定理証明機能の作成は、非常に困難であるだけでなく、おそらく実際の利用にはあまりにも実行速度が遅過ぎるでしょう。）システムでは、例えば" $x < 1$ "は" $x < 2$ "を意味するというような、単純な比較演算子の意味は認識可能です。しかし、それ以外の場合は、述語条件は問い合わせのWHERE条件と完全に一致している必要があります。一致していない場合は、インデックスは使用可能と認識されません。一致するかどうかは、実行時ではなく、問い合わせ計画作成時に判定されます。したがって、パラメータ付きの問い合わせでは部分インデックスは動作しません。たとえば、" $x < ?$ "と指定されたパラメータを持つ、プリペアド問い合わせでは、どのようなパラメータ値であっても" $x < 2$ "を表しません。

部分インデックスの考えられる3つ目の用法では、問い合わせでインデックスをまったく使用しません。この考え方は、テーブルの部分集合に一意インデックスを作成するというものです。[例11-3](#)を参照してください。これにより、インデックスの述語を満たさない行を制約することなく、その述語を満たす行での一意性を強制します。

例 11-3. 一意な部分インデックスの作成

テストの結果が格納されているテーブルがあるとします。与えられた件名 (subject) および対象 (target) の組み合わせに対して、"成功"のエントリが確実に1つしかないようにします。"失敗"のエントリは、複数あっても構いません。以下に、これを実行する一例を示します。

```
CREATE TABLE tests (
    subject text,
    target text,
    success boolean,
    ...
);

CREATE UNIQUE INDEX tests_success_constraint ON tests (subject, target)
WHERE success;
```

これは、成功するテストがほとんどなく、失敗するテストが多い場合に特に有効な手法です。

最後に、部分インデックスは、システムの問い合わせ計画の選択を変更するためにも使用できます。特殊なデータ分布を持つデータ集合では、システムが実際には使用すべきでないインデックスを使用してしまうことがあります。このような場合、特定の問い合わせでは使用することができないインデックスを設定することができます。通常、PostgreSQLはインデックスの使用について適切な選択を行います（例えば、頻出値の検索にはインデックスを使用しませんので、前述の例はインデックスのサイズを実際に小さくするだけのもので、インデックスの使用を制限するためには必要はありません）。まったく不適切な計画を選択するようであれば、バグとして報告してください。

部分インデックスを作成するには、少なくとも問い合わせプランナと同等の知識を持っていること、特に、インデックスが有益となる状況を理解している必要があることに留意してください。このような知識を得るためには、PostgreSQLでインデックスがどのように機能するかを理解し、経験を積むことが必要です。ほとんどの場合、通常のインデックスと比べて、部分インデックスを使用する利点は微細です。

部分インデックスの詳細については、[The case for partial indexes](#)、[Partial indexing in POSTGRES: research project](#)、および[Generalized Partial Indexes](#)（キャッシュされたバージョン）を参照してください。

11.9. 演算子クラスと演算子族

インデックス定義では、インデックスの各列に演算子クラスを指定することができます。

```
CREATE INDEX name ON table (column opclass [sort options] [, ...]);
```

演算子クラスにより、その列のインデックスで使用される演算子が特定されます。例えば、`int4`型に対するB-treeインデックスには、`int4_ops`クラスを使用します。この演算子クラスには、`int4`型の値用の比較関数が含まれています。実際には、通常、列のデータ型のデフォルト演算子クラスで十分です。演算子クラスを持つ主な理由は、複数の有意義なインデックスの振舞いを持つ、いくつかのデータ型に対してです。例えば、複素数データ型を、絶対値と実数部のどちらかでソートする必要があるとします。この処理は、そのデータ型の2つの演算子クラスを定義した上で、インデックスを作成する際に適切なクラスを選択することで、実行可能です。演算子クラスは基本的なソート順を決定します。（これは`COLLATE ASC/DESC`、`NULLS FIRST/NULLS LAST`、またはこの両方というソートオプションを付けることで変更される可能性があります。）

以下のように、デフォルトの演算子クラスとは別に、組み込み演算子クラスがいくつかあります。

- `text_pattern_ops`、`varchar_pattern_ops`、`bpchar_pattern_ops`、`name_pattern_ops` 演算子クラスは、それぞれ、`text`、`varchar`、`char`型上のB-treeインデックスをサポートします。デフォルトの演算子クラスとの違いは、ローケル特有の照合規則に従わずに、文字同士を厳密に比較する点です。これらの演算子クラスを、標準"C"ローケルを使用しないデータベースにおける、パターンマッチ式（`LIKE`やPOSIX正規表現）を含む問い合わせでの使用に適したものにします。例えば、以下のように`varchar`のインデックスを作成できます。

```
CREATE INDEX test_index ON test_table (col varchar_pattern_ops);
```

また、インデックスを使うために通常の`<`、`<=`、`>`、または`>=`比較を含む問い合わせを希望するのであれば、デフォルトの演算子クラスでインデックスを作成しなければならないことに注意してください。こうした問い合わせでは`xxx_pattern_ops`演算子クラスを使用することができません。（しかし、通常の等価比較はこれらの演算子クラスを使用することができます。）異なる演算子クラスで使用する列と同じ列に対して複数のインデックスを作成することができます。Cローケルを使用する場合は、`xxx_pattern_ops`演算子クラスは必要ありません。Cローケルでのパターンマッチ問い合わせでは、デフォルト演算子クラスを使用したインデックスが使用できるためです。

以下の問い合わせは、定義済みの演算子クラスを全て返します。

```
SELECT am.amname AS index_method,
       opc.opcname AS opclass_name
FROM pg_am am, pg_opclass opc
WHERE opc.opcmethod = am.oid
ORDER BY index_method, opclass_name;
```

実際のところ演算子クラスは、演算子族と呼ばれる大きな構造の一部でしかありません。複数のデータ型が似たような動作を行う場合、データ型を跨る演算子を定義し、インデックスで使用可能とすることが有用な場合がよくあります。このためには、各型に対する演算子クラスが同一の演算子族にまとめられている必要があります。データ型を跨る演算子は演算子族の要素です。演算子族内の1つの演算子クラスに結びついているわけではありません。

以下の問い合わせは、定義済みの演算子族と各演算子族に含まれる演算子をすべて表示します。

```
SELECT am.amname AS index_method,
       opf.opfname AS opfamily_name,
       amop.amopopr::regoperator AS opfamily_operator
FROM pg_am am, pg_opfamily opf, pg_amop amop
WHERE opf.opfmethod = am.oid AND
      amop.amopfamily = opf.oid
ORDER BY index_method, opfamily_name, opfamily_operator;
```

11.10. インデックスと照合順序

インデックスはインデックス列当たり 1 つの照合順序のみをサポートすることができます。複数の照合順序を考慮しなければならない場合、複数のインデックスが必要になるかもしれません。

以下の文を考えてみます。

```
CREATE TABLE test1c (  
    id integer,  
    content varchar COLLATE "x"  
);  
  
CREATE INDEX test1c_content_index ON test1c (content);
```

このインデックスは自動的に背後にある列の照合順序を使用することになり、

```
SELECT * FROM test1c WHERE content > constant;
```

という形式の問い合わせでは、この比較はデフォルトで列の照合順序を使用しますので、このインデックスを使用することになります。しかし、このインデックスは何らかの他の照合順序を含む問い合わせを高速化することはできません。このため

```
SELECT * FROM test1c WHERE content > constant COLLATE "y";
```

という形式の問い合わせも考慮しなければならない場合は、以下のように"y"照合順序をサポートする追加のインデックスを作成することになります。

```
CREATE INDEX test1c_content_y_index ON test1c (content COLLATE "y");
```


11.11. インデックス使用状況の検証

PostgreSQLでは、インデックスのメンテナンスまたはチューニングは必要ありませんが、どのインデックスが実際の問い合わせで使われているかを確認することは、やはり重要です。個々のコマンドでのインデックスの使用状況は、[EXPLAIN](#)コマンドで検証できます。この目的のための用例を[項14.1](#)に示します。また、[項27.2](#)に示す通り、稼働中のサーバにおけるインデックス使用状況の全体的な統計情報を取り出すこともできます。

どのインデックスを作成すべきかを判断するための一般的な手順を定めることは困難です。これまでの節では、例として典型的なケースをいくつか記述してきました。十分な検証がしばしば必要です。本節の残りで、検証のためのヒントをいくつか説明しておきます。

- まず、必ず[ANALYZE](#)コマンドを実行してください。このコマンドにより、テーブル内の値の分布に関する統計情報を収集します。この情報は、問い合わせにより返される行数を推測する際に必要となります。推測された行数は、利用できる各問い合わせ計画に実際のコストを割り当てるために、プランナで必要となります。実際の統計情報が少しでも欠如している場合、何らかのデフォルト値が仮定されます。このデフォルト値は、ほぼ間違いなく不正確です。したがって、[ANALYZE](#)コマンドを実行せずに、アプリケーションのインデックス使用状況を検証しても、あまり意味がありません。より詳細な情報は[項23.1.3](#)と[項23.1.6](#)を参照してください。
- 検証には、実際に使用するデータを使ってください。テストデータを使ってインデックスを作成した場合、テストデータに必要なインデックスはわかりますが、それ以上はわかりません。

非常に小さなテストデータを使用することも、結果に特に致命的な影響を与えます。100,000行から1,000行を選択する場合は、インデックスが使用される可能性があります。100行から1行を選択する場合はインデックスはまず使用されません。なぜなら、100行はおそらく1つのディスクページに収まるため、1ページを逐次読み取るよりも高速な計画は存在しないからです。

また、アプリケーションがまだ実動していない場合、テストデータを作成しなければならないことがよくありますが、その際にも注意が必要です。非常に類似した値や、完全にランダムな値、またはソートされた順序で値が挿入されている場合は、その統計情報は、実際のデータの分布とかけ離れたものになってしまいます。

- インデックスが使用されていない場合、テストのためにインデックスを強制的に使用するようにすると便利です。各種の計画型を無効にするかを設定できる実行時パラメータがあります ([項18.7.1](#)を参照してください)。例えば、最も基本的な計画であるシーケンシャルスキャン (`enable_seqscan`) および入れ子状ループ結合 (`enable_nestloop`) を無効に設定すると、システムは別の計画を使用するように強制されます。そのような設定を行っても、システムがシーケンシャルスキャンや入れ子状ループ結合を選択する場合は、インデックスを使用しない理由としておそらくもっと根本的な理由があるということになります。例えば、問い合わせの条件がインデックスに一致しない、などが考えられます。(どのような問い合わせで、どのようなインデックスを使用できるかは、前節で説明済みです。)
- 強制的にインデックスを使うように設定し、インデックスを使用している場合は、次の2つの状況が考えられます。システムの判断が正しく、インデックスの使用が実際には適切ではないという状況と、問い合わせ計画のコスト推定が実情を反映していない状況です。したがって、インデックスを使った問い合わせの実行時間と、使わない場合の実行時間を計測する必要があります。この場合、`EXPLAIN ANALYZE` コマンドが便利です。
- コスト推定が間違っていると判明した場合、やはり2つの状況が考えられます。総コストは、各計画ノードの行単位のコストに、計画ノードの推定選択度を掛けることで算出されます。計画ノードのコスト推定は、実行時パラメータによって設定することができます ([項18.7.2](#)を参照してください)。推定選択度が不正確であるのは、統計情報が不十分であるのが原因です。統計情報収集用のパラメータを調節することによって、この状況を改善することができます ([ALTER TABLE](#)を参照してください)。

コストを適切に調節できない場合は、明示的にインデックスを使用する必要が考えられます。あるいは、PostgreSQL開発者に問題の調査を依頼することになるかもしれません。

第 12 章 全文検索

目次

12.1. [導入](#)

- 12.1.1. [文書とは何か?](#)
- 12.1.2. [基本的なテキスト照合](#)
- 12.1.3. [設定](#)

12.2. [テーブルとインデックス](#)

- 12.2.1. [テーブルを検索する](#)
- 12.2.2. [インデックスの作成](#)

12.3. [テキスト検索の制御](#)

- 12.3.1. [文書のパース](#)
- 12.3.2. [問合わせのパース](#)
- 12.3.3. [検索結果のランキング](#)
- 12.3.4. [結果の強調](#)

12.4. [追加機能](#)

- 12.4.1. [文書の操作](#)
- 12.4.2. [問合わせを操作する](#)
- 12.4.3. [自動更新のためのトリガ](#)
- 12.4.4. [文書の統計情報の収集](#)

12.5. [パーサ](#)

12.6. [辞書](#)

- 12.6.1. [ストップワード](#)
- 12.6.2. [simple辞書](#)
- 12.6.3. [同義語辞書](#)
- 12.6.4. [類語辞書](#)
- 12.6.5. [Ispell辞書](#)
- 12.6.6. [Snowball辞書](#)

12.7. [設定例](#)

12.8. [テキスト検索のテストとデバッグ](#)

- 12.8.1. [設定のテスト](#)
- 12.8.2. [パーサのテスト](#)
- 12.8.3. [辞書のテスト](#)

12.9. [GiSTおよびGINインデックス種類](#)

12.10. [psqlサポート](#)

12.11. [制限事項](#)

12.12. [8.3以前のテキスト検索からの移行](#)

12.1. 導入

全文検索(または単にテキスト検索)は、問い合わせを満たす自然言語の文書を識別し、更には問い合わせとの関連性の順に並び替えることができます。もっとも一般的な検索は、与えられた検索語を含む文書を探し、問い合わせとの類似性の順に返す、というものです。問い合わせと類似性の記法は非常に柔軟で、特定の用途に適合できます。もっとも単純な検索では、問い合わせは単語の集合として、類似性は文書中の問い合わせ対象の単語の頻度として扱います。

テキスト検索演算子は、データベースシステムに長年存在していました。PostgreSQLは、テキストデータ型用に、`~`、`~*`、`LIKE`、`ILIKE`の各演算子を持っています。しかし、近代的な情報システムに必要な以下の本質的な特徴を欠いています。

- 英語にさえ、言語学的なサポートがありません。派生語、たとえば`satisfies`に対して`satisfy`を容易に扱えないため、正規表現は十分ではありません。`satisfy`を探すときは、たぶんあなたは`satisfies`も探したいでしょうが、それらを含む文書は探せないかもしれません。ORを使えば複数の派生語を検索することができますが、退屈で間違いやすいです(ある種の単語は数千の派生語を持つことがあります)。
- 検索結果を順序付け(順位付け)することができません。その結果、数千の合致する文書が見つかったような場合に非効率的です。
- インデックスをサポートしないので毎回検索時にすべての文書进行处理しなければならず、遅いです。

全文検索のインデックス付けでは、文書を前もって処理しておき、後で素早く検索するために、インデックスを保存しておくことができます。前処理には以下があります。

文書からトークンを解析します。トークンを色々なクラス、たとえば数、単語、複合単語、電子メールアドレスに分けて識別することが有効です。そうすれば、扱いを変えることができます。原則として、トークンのクラスは、特定の用途に依存します。しかし、ほとんどの目的には、あらかじめ定義されたクラスの集合を使うのが適当です。PostgreSQLは、パーサを使ってこの処理段階を実行します。標準搭載のパーサが提供されますが、特別な用途にはカスタム仕様のパーサを作ることもできます。

トークンを語彙素(*lexemes*)に変換します。語彙素はトークンと同じ文字列ですが、違う形態の同じ単語が同じになるように正規化されています。たとえば、正規化においてはほぼ常に大文字を小文字に変換し、接尾を取り除くことが多いです(英語の`s`または`es`)。これにより、可能性のあるすべての変種を地道に入力すること無く、同じ単語の変化形を検索できます。また、このステップでは、あまりにありふれていて、検索の役に立たないストップワードを取り除くことが多いです。(つまり、トークンは文書テキストの未加工の断片そのものであり、語彙素はインデックス付けや検索に有用と思われる単語です。)PostgreSQLは、辞書を使ってこのステップを実行します。いろいろな標準辞書が提供されています。特定の用途向けにカスタム辞書を作ることもできます。

検索に最適化された前処理済の文書を保存します。たとえば、個々の文書は、正規化された語彙素の整列済の配列として表現されます。語彙素とともに、適合性ランキング用に、位置情報を格納しておくことがしばしば望まれます。そうすることにより、問い合わせの語を"高密度"に含んでいる文書を、まばらに含む文書よりも高くランクづけすることができます。

辞書を使ってトークンの正規化を細かく制御できます。適当な辞書を用意すれば次のようなことができます。

- インデックスしたくないストップワードの定義
- Ispellを使って、同義語を単一の単語に関連づける
- 類語辞書(thesaurus)を使って、成句を単一の単語に関連づける
- Ispell辞書を使って、単語の変種を正規の単語に関連づける
- Snowball語幹規則を使って、単語の変種を正規の単語に関連づける

前処理した文書を格納するために、データ型`tsvector`が提供されています。また、処理済問い合わせを表現するために`tsquery`型も提供されています(項8.11)。これらのデータ型のために、多数の関数と演算子が利用できますが(項9.13)、もっとも重要なのは、項12.1.2で紹介している`@@`演算子です。全文検索はインデックス(項12.9)を使って高速化できます。

12.1.1. 文書とは何か?

文書は全文検索システムにおける検索の単位です。たとえば、雑誌記事やメールのメッセージです。テキスト検索エンジンは、文書をパースし、語彙素(キーワード)とそれが含まれる親文書の関連を格納できなければなりません。後で、この関連を使って問い合わせ語を含む文書を検索するのに使います。

PostgreSQLでの検索においては、ドキュメントはデータベースのテーブルの行内のテキストフィールドか、あるいはそのようなフィールドの組み合わせ(結合)でもよいです。そうしたフィールドはおそらく複数のテーブルに格納されていたり、動的に獲得されるものであったりします。言い換えると、文書はインデックス付けのために複数の異なる部分から構成されても良く、それらが全体としてはひとまとまりに格納されていなくても良いのです。例を示します。

```
SELECT title || ' ' || author || ' ' || abstract || ' ' || body AS document
FROM messages
WHERE mid = 12;

SELECT m.title || ' ' || m.author || ' ' || m.abstract || ' ' || d.body AS document
FROM messages m, docs d
WHERE mid = did AND mid = 12;
```

注意: 実際には、これらの例の問い合わせでは、`coalesce`を使って、一部NULLが含まれているためにドキュメント全体がNULLになってしまうのを防ぐべきです。

別な方法としては、ファイルシステム上に文書を単純なテキストファイルとして格納することです。この場合、データベースは、フルテキストインデックスを格納し、検索を実行するために使うことができます。ファイルシステムから文書を取り出すためには、何かのユニークな識別子を使います。しかし、データベースの外にあるファイルを取り出すには、スーパーユーザの許可か、特殊な関数のサポートが必要です。そういうわけでたいいの場合はPostgreSQLの中にすべてのデータを保持するのよりも不便です。また、すべてのデータをデータベースに保持することにより、文書のインデックス付けと表示の際に文書のメタデータにアクセスすることが容易になります。

テキスト検索という目的のため、各々の文書は前処理されて`tsvector`形式に変換しておかなければなりません。検索と順位付けはすべて`tsvector`表現の文書上で行われます。検索とランキングは文書の`tsvector`表現上で実行されます — オリジナル文書は、ユーザに表示のため選択された場合にのみ取り出される必要があります。というわけで、ここでは`tsvector`を文書と見なすことがよくあります。といっても、`tsvector`は完全な文書の縮小表現でしかありません。

12.1.2. 基本的なテキスト照合

PostgreSQLにおける全文検索は、`tsvector`(文書)が、`tsquery`(問い合わせ)に一致したら真を返す照合演算子`@@`に基づいています。どちらのデータ型を先に書いても構いません。

```
SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector @@ 'cat & rat'::tsquery;
?column?
-----
t

SELECT 'fat & cow'::tsquery @@ 'a fat cat sat on a mat and ate a fat rat'::tsvector;
?column?
-----
f
```

上記の例でわかるように、`tsquery`は、`tsvector`と違って、単なるテキストではありません。`tsquery`は正規化済の語彙素である検索表現を含み、AND, OR, NOT演算子を使って複数の表現を組み合わせても構いません。(詳細はこれを見てください [項8.11](#)。)たとえば、テキスト中の単語を正規化することにより、ユーザが入力したテキストを適切な`tsquery`に変換する`to_tsquery`と`plainto_tsquery`という関数があります。同様に、文書文字列をパースして正規化するために`to_tsvector`が利用できます。というわけで、実際にはテキスト検索照合はこんな感じになります。

```
SELECT to_tsvector('fat cats ate fat rats') @@ to_tsquery('fat & rat');
?column?
-----
t
```

この照合は、もしつぎのように書くとうまくいかないことに注意してください。

```
SELECT 'fat cats ate fat rats'::tsvector @@ to_tsquery('fat & rat');
?column?
-----
f
```

というのも、単語`rats`に対して正規化が行われないからです。`tsvector`の要素は、すでに正規化されている語彙素であることになっているので、`rats`は`rat`に一致しません。

また、`@@`演算子は、`text`を入力として受付けるので、簡単に使うときには、明示的にテキスト文字列を`tsvector`または`tsquery`に変換することを省略できます。応用として以下のものがあります。

```
tsvector @@ tsquery
tsquery  @@ tsvector
text     @@ tsquery
text     @@ text
```

12.1.3. 設定

今までののはすべて単純なテキスト検索の例でした。すでに述べたように、全文検索機能を使えば、もっと色々なことができます。インデックス付けの際に特定の単語をスキップ(ストップワード)、同義語(synonym)処理、賢いパース処理、すなわち、単に空白区切りに基づくパース処理以上のものです。この機能はテキスト検索設定で制御します。PostgreSQLには、多くの言語用の設定があらかじめ組み込まれていますが、ユーザ設定を容易に作ることもできます。(psqlの`\xdf`コマンドで、利用できる設定を表示できます)。

インストールの際には、適当な設定が選ばれ、[default_text_search_config](#)が`postgresql.conf`中にセットされます。クラスタ全体で同じ設定を使用する場合は`postgresql.conf`の設定値を利用できます。クラスタの設定とは異なるが、あるデータベースの中で同じ設定を使う場合には、`ALTER DATABASE ... SET`を利用します。さもないと、セッション単位で`default_text_search_config`を設定できます。

設定に依存するテキスト検索関数は、オプションで`regconfig`引数を持っており、使用する設定を明示的に指定できます。`default_text_search_config`は、この引数が省略されたときだけ使用されます。

カスタムテキスト検索設定を作り易くするため、設定はより単純なデータベースオブジェクトから作られます。PostgreSQLのテキスト検索機能は、4つの設定関連のデータベースオブジェクトを提供しています。

- テキスト検索パーサは、文書をトークンに分解し、トークンを分類します(たとえば、単語とか数のように)。
- テキスト検索辞書はトークンを正規化された形式に変換し、ストップワードを排除します。
- テキスト検索テンプレートは、現在の辞書が利用する関数を提供します(辞書は、単にテンプレートと、その引数の集合を指定するだけです)。
- テキスト検索設定は、パーサと使用する辞書の集合を選択し、パーサが生成したトークンを正規化します。

テキスト検索パーサとテンプレートは、低レベルのC関数で作ります。したがって、新しく開発するためにはCのプログラミング能力と、データベースにインストールするためのスーパーユーザ権限が必要になります。(PostgreSQLの配布物の`contrib/`には、追加パーサとテンプレートの例があります)。辞書と設定は、単に配下のパーサとテンプレートのパラメータを設定し、両者を結び付けるだけなので、新しい辞書と設定を作るために特別な権限は必要ありません。この章の後でカスタム辞書と設定を作る例が登場します。

[前のページ](#)
全文検索

[ホーム](#)
[上に戻る](#)

[次のページ](#)
テーブルとインデックス

12.2. テーブルとインデックス

前の節の例では、単純な文字列定数を使った全文検索照合を説明しました。この節では、テーブルのデータを検索する方法、そしてインデックスを使う方法を示します。

12.2.1. テーブルを検索する

インデックスがなくても全文検索をすることは可能です。body フィールド中のfriend という単語を含む行のtitle を印刷する単純な問い合わせは次のようになります。

```
SELECT title
FROM pgweb
WHERE to_tsvector('english', body) @@ to_tsquery('english', 'friend');
```

同時に、これは、friends、friendlyのように、関連する単語を見つけ出します。これらはすべて同じ正規化された語彙素に帰結するからです。

上の問い合わせはenglish設定を使って文字列をパースして正規化することを指定しています。別の方法としては、設定パラメータを省略することができます。

```
SELECT title
FROM pgweb
WHERE to_tsvector(body) @@ to_tsquery('friend');
```

この問い合わせは[default_text_search_config](#)で設定された設定を使用します。

もっと複雑な例として、createとtableをtitleまたはbodyに含む文書のうち新しい順に10個選ぶというものを示します。

```
SELECT title
FROM pgweb
WHERE to_tsvector(title || ' ' || body) @@ to_tsquery('create & table')
ORDER BY last_mod_date DESC
LIMIT 10;
```

細かいことですが、この例では、ふたつのうちのフィールドにNULLを含む行を探すために必要なcoalesce関数の呼び出しを省略しています。

これらの問い合わせはインデックスなしでも動きますが、たまに実行する一時的な問い合わせ用を除くと、たいていの用途には、たぶん不定期のその場その場の検索を除いて遅すぎます。実用上は、インデックスを作成することが必要なのが普通です。

12.2.2. インデックスの作成

テキスト検索を高速化するために、GINインデックス([項12.9](#))を作ることができます。

```
CREATE INDEX pgweb_idx ON pgweb USING gin(to_tsvector('english', body));
```

2引数バージョンのto_tsvectorを使っていることに注意してください。設定名を指定するテキスト検索関数だけが、式インデックス([項11.7](#))で使えます。これは、インデックス内容が、[default_text_search_config](#)の影響を受けないためです。もし影響を受けるとすると、異なるテキスト検索設定で作られたtsvectorを持つエントリの間でインデックス内容が首尾一貫しなくなるからです。そして、どのエントリがどのようにして作られたのか、推測する方法はないでしょう。そのようなインデックスを正しくダンプ、リストアするのは不可能でしょう。

上記のインデックスでは、2引数バージョンのto_tsvectorが使われているので、同じ設定名の2引数バージョンのto_tsvectorを使う問い合わせ参照だけがそのインデックスを使います。すなわち、WHERE to_tsvector('english', body) @@ 'a & b'はインデックスが使えますが、WHERE to_tsvector(body) @@ 'a & b'は使えません。これにより、インデックスエントリを作ったときの設定と、同じ設定のときだけインデックスが使われることが保証されます。

他の列によって設定名が指定されたより複雑な式インデックスを作ることができます。すなわち、

```
CREATE INDEX pgweb_idx ON pgweb USING gin(to_tsvector(config_name, body));
```

ここで、config_nameはpgwebテーブルの列です。これによって、各々のインデックスエントリで使用された設定を記録しつつ、同じインデックスの中で異なる設定を混在させることができます。これは、例えば文書の集まりが異なる言語の文書を含む場合に有用です。繰り返しになりますが、インデックスを使うよう考慮されている問い合わせは、合致するように書かれなければなりません。すなわち、WHERE to_tsvector(config_name, body) @@ 'a & b'.

インデックスには、列を連結することさえできます。

```
CREATE INDEX pgweb_idx ON pgweb USING gin(to_tsvector('english', title || ' ' || body));
```

別の方法として、`to_tsvector`の出力を保持する別の`tsvector`列を作る方法があります。この例では、`title`と`body`を連結、`coalesce`を使って、一つのフィールドが`NULL`であっても他のフィールドがインデックス付けされることを保証しています。

```
ALTER TABLE pgweb ADD COLUMN textsearchable_index_col tsvector;  
UPDATE pgweb SET textsearchable_index_col =  
    to_tsvector('english', coalesce(title, '') || ' ' || coalesce(body, ''));
```

そして、GINインデックスを作って検索速度を上げます。

```
CREATE INDEX textsearch_idx ON pgweb USING gin(textsearchable_index_col);
```

これで、高速全文検索を実行する準備ができました。

```
SELECT title  
FROM pgweb  
WHERE textsearchable_index_col @@ to_tsquery('create & table')  
ORDER BY last_mod_date DESC  
LIMIT 10;
```

`tsvector`形式を保存するために別の列を使う場合、`title`あるいは`body`が変更されたら`tsvector`列を最新の状態にいつでも維持するためにトリガを作る必要があります。 [項12.4.3](#)にその方法が説明されています。

別列方式が式インデックスに勝る点の一つは、インデックスを使うために問い合わせの中でテキスト検索設定を明示的に指定する必要がないことです。上の例で示したように、問い合わせは`default_text_search_config`に依存できます。もう一つの利点は、インデックスの合致を検証するために`to_tsvector`を再実行する必要がないのでより高速だということです。(この点はGINインデックスを使うときよりも、GiSTインデックスを使う場合に重要です。 [項12.9](#)参照。)しかしながら、式インデックス方式はセットアップがより容易で、`tsvector`表現を明示的に保存する必要がないので、ディスクスペースの消費が少ないです。

[前のページ](#)
導入

[ホーム](#)
[上に戻る](#)

[次のページ](#)
テキスト検索の制御

12.3. テキスト検索の制御

全文検索を実装するためには、文書から `tsvector` を、そしてユーザの問い合わせから `tsquery` を作成する関数が存在しなければなりません。また、結果を意味のある順で返す必要があります。そこで、問い合わせとの関連性で文書を比較する関数も必要になってきます。結果を体裁良く表示できることも重要です。PostgreSQLはこれらすべての機能を提供しています。

12.3.1. 文書のパース

PostgreSQLは、文書を `tsvector` データ型に変換する `to_tsvector` 関数を提供しています。

```
to_tsvector([ config regconfig, ] document text) returns tsvector
```

`to_tsvector` は、テキスト文書をパースしてトークンにし、トークンを語彙素に変換、文書中の位置とともに語彙素をリストとして持つ `tsvector` を返します。文書は、指定したものか、あるいはデフォルトのテキスト検索設定にしたがって処理されます。単純な例を示します。

```
SELECT to_tsvector('english', 'a fat cat sat on a mat - it ate a fat rats');
           to_tsvector
-----
'ate':9 'cat':3 'fat':2,11 'mat':7 'rat':12 'sat':4
```

上に示す例では、結果の `tsvector` で、`a`、`on`、`it` という単語が含まれないこと、`rats` という単語が `rat` になっていること、句読点記号 `-` が無視されていることがわかります。

`to_tsvector` 関数は、文書をトークンに分解して、そのトークンに型を割り当てるパーサを内部的に呼び出しています。それぞれのトークンに対して辞書(項12.6)のリストが検索されます。ここで、辞書のリストはトークンの型によって異なります。最初の辞書は、トークンを認識し、トークンを表現する一つ以上の正規化された語彙素を出力します。例えば、ある辞書は `rats` は `rat` の複数形であることを認識しているので、`rats` は `rat` になります。ある単語はストップワード(項12.6.1)として認識されます。これは、意味のある検索中にあまりも多く出現するため、無視されるものです。先の例では、`a`、`on`、および `it` がそれです。もしリスト中の辞書のどれもがトークンを認識しなければ、そのトークンは無視されます。先の例では、句読点の `-` がそうです。なぜなら、実際にはそのトークン型(`Space symbols`)に対して辞書が割り当てられておらず、空白トークンは決してインデックス付けされないことを意味します。パーサ、辞書、そしてどのトークンがインデックス付けされるかという選択は、テキスト検索設定(項12.7)によって決められます。同じデータベース中に多くの異なった設定を持つことができ、多くの言語用に定義済の設定が用意されています。先の例では、英語用として、デフォルトの `english` 設定を使っています。

関数 `setweight` を使って `tsvector` のエントリに与えられた重みのラベルを与えることができます。ここで重みは、`A`、`B`、`C`、`D` のどれかの文字です。重みの典型的な使い方は、文書の各部分がどこから来たのかをマークすることです。たとえば、タイトルから来たのか、本文から来たのかなど。後でこの情報は検索結果のランキングに利用できます。

`to_tsvector(NULL)` は `NULL` を返すので、`NULL` になる可能性のある列に対しては `coalesce` を使うことをお勧めします。構造化された文書から `tsvector` を作るための推奨できる方法を示します。

```
UPDATE tt SET ti =
    setweight(to_tsvector(coalesce(title,'')), 'A') ||
    setweight(to_tsvector(coalesce(keyword,'')), 'B') ||
    setweight(to_tsvector(coalesce(abstract,'')), 'C') ||
    setweight(to_tsvector(coalesce(body,'')), 'D');
```

ここでは、完成した `tsvector` の語彙素に対して、ラベル付けのために `setweight` を使っています。そして、`tsvector` の連結演算子 `||` を使って、ラベルづけされた `tsvector` の値をマージします。(詳細は項12.4.1を参照してください。)

12.3.2. 問い合わせのパース

PostgreSQLは、問い合わせを `tsquery` に変換する `to_tsquery` 関数と `plainto_tsquery` 関数を提供しています。`to_tsquery` は、`plainto_tsquery` よりも多くの機能を提供していますが、入力のチェックはより厳格です。

```
to_tsquery([ config regconfig, ] querytext text) returns tsquery
```

`to_tsquery` は、`querytext` から `tsquery` としての値を生成します。`querytext` は、論理演算子 `&` (`AND`)、`|` (`OR`)、`!` (`NOT`) で区切られる単一のトークンから構成されなければなりません。これらの演算子は括弧でグループ化できます。言い換えると、`to_tsquery` の入力は、項8.11で述べられている一般規則にしたがっていなければなりません。違いは、基本的な `tsquery` の入力はトークンの表面的な値を受け取るのに対し、`to_tsquery` は指定した、あるいはデフォルトの設定を使ってトークンを語彙素へと正規化し、設定にしたがって、ストップワードであるようなトークンを破棄します。例を示します。

```
SELECT to_tsquery('english', 'The & Fat & Rats');
           to_tsquery
```

```
-----  
'fat' & 'rat'
```

基本的な`tsquery`の入力では、各々の語彙素に重みを付加することにより、同じ重みを持つ`tsvector`の語彙素のみに照合することができます。例を示します。

```
SELECT to_tsquery('english', 'Fat | Rats:AB');  
       to_tsquery  
-----  
'fat' | 'rat':AB
```

また、明示的な前方一致検索のため、`*`を語彙素に与えることもできます。

```
SELECT to_tsquery('supern:*A & star:A*B');  
       to_tsquery  
-----  
'supern':*A & 'star':*AB
```

このような語彙素は、与えられた文字列で始まる`tsvector`中のどんな単語にも照合するでしょう。`to_tsquery`は、単一引用符で囲まれた語句を受け付けることもできます。これは主に、設定の中にそういった語句を持つ同義語辞書を含んでいるときに有用です。以下の例では、ある同義語の中に`supernovae stars : sn`という規則が含まれています。

```
SELECT to_tsquery('''supernovae stars'' & !crab');  
       to_tsquery  
-----  
'sn' & !'crab'
```

引用符がない場合は、`to_tsquery`は、ANDあるいはOR演算子で区切られていないトークンに対して構文エラーを引き起こします。

```
plainto_tsquery([ config regconfig, ] querytext text) returns tsquery
```

`plainto_tsquery`は整形されていないテキスト`querytext`を、`tsquery`に変換します。テキストはパースされ、`to_tsvector`としてできる限り正規化されます。そして、`&` (AND) 論理演算子が存続した単語の間に挿入されます。

例：

```
SELECT plainto_tsquery('english', 'The Fat Rats');  
       plainto_tsquery  
-----  
'fat' & 'rat'
```

`plainto_tsquery`は、入力中の論理演算子も、重さラベルも、前方一致ラベルも認識できないことに注意してください。

```
SELECT plainto_tsquery('english', 'The Fat & Rats:C');  
       plainto_tsquery  
-----  
'fat' & 'rat' & 'c'
```

ここでは、入力中のすべての句読点がスペース記号に変換された結果、破棄されています。

12.3.3. 検索結果のランキング

ランキングはある問合わせに対して、どの程度文書が関連しているかを計測しようとするものです。合致している文書が多数あるとき、もっとも関連している文書が最初に表示されるようにするためです。PostgreSQLは、2つの定義済ランキング関数を提供しています。それらは、辞書情報、近接度情報、構造的情報を加味します。すなわち、問合わせの用語がどの位の頻度で文書に出現するか、文書中でどの程度それらの用語が近接しているか、どの用語が含まれる文書部位が度の程度重要なのかを考慮します。しかし、関連度という概念は曖昧で、用途に強く依存します。異なる用途は、ランキングのために追加の情報を必要とするかも知れません。たとえば、文書の更新時刻など。組み込みのランキング関数は例に過ぎません。利用者の目的に応じて、自分用のランキング関数を作ったり、その結果を追加の情報と組み合わせることができます。

今のところ、二種類のランキング関数が利用可能です。

```
ts_rank([ weights float4[], ] vector tsvector,  
        query tsquery [, normalization integer ]) returns float4
```

それらの語彙素にマッチした頻度に基づくベクターのランク。

```
ts_rank_cd([ weights float4[], ] vector tsvector,  
            query tsquery [, normalization integer ]) returns float4
```

この関数は、"Information Processing and Management"ジャーナルに掲載された Clarke, Cormack, Tudhopeの"Relevance Ranking for One to Three Term Queries"で述べられている方法で、与えられた文書ベクターと問合わせの表面密度(*cover density*)を計算します。

この関数は、入力中に位置情報を必要とします。ですから、"剥き出しの" 値では動きません — この場合は常に0を返します。

これらの関数では、単語がどの程度ラベル付けに依存するかを、単語ごとに指定する機能が`weights`オプションパラメータによって提供されています。重み配列で、それぞれのカテゴリの単語がどの程度重み付けするかを指定します。その順は以下のようになっています。

```
{D-weight, C-weight, B-weight, A-weight}

weightsを与えない場合は、次のデフォルト値が使われます。

{0.1, 0.2, 0.4, 1.0}
```

重みの典型的な使い方は、文書のタイトルやアブストラクトのような特定の場所にある単語をマーク付けするような使い方で、そうすることにより、文書の本体に比べてそこにある単語がより重要なのか、そうでないのか、扱いを変えることができます。

文書が長ければ、それだけ問合わせ用語を含む確率が高くなるため、文書のサイズを考慮にいれることは理にかなっています。たとえば、5つの検索語を含む100語の文書は、たぶん5つの検索語を含む1000語の文書よりも関連性が高いでしょう。ランキング関数には、どちらも整数型の正規化オプションがあります。これは、文書の長さがランクに影響を与えるのかどうか、与えらとすればどの程度か、ということを指定します。この整数オプションは、いくつかの挙動を制御するので、ビットマスクになっています。複数の挙動を|で指定できます(例: 2|4)。

- 0(デフォルト):文書の長さを無視します
- 1:ランクを(1 + log(文書の長さ))で割ります
- 2:ランクを文書の長さで割ります
- 4:ランクをエクステントの間の調和平均距離で割ります(これは`ts_rank_cd`で実装されています)
- 8: ランクを文書中の一意の単語の数で割ります
- 16: ランクをlog(文書中の一意の単語の数)+1 で割ります
- 32 ランクをランク自身+1 で割ります

2以上のフラグビットが指定された場合には、変換は上記に列挙された順に行われます。

これは重要なことですが、ランキング関数はグローバル情報を一切使わないので、時には必要になる1%から100%までの均一な正規化はできません。正規化オプション32(`rank/(rank+1)`)を適用することにより、すべてのランクを0から1に分布させることができます。しかし、もちろんこれは表面的な変更に過ぎません。検索結果のならば順に影響を与えるものではありません。

マッチする順位の高い10位までを選ぶ例を示します。

```
SELECT title, ts_rank_cd(textsearch, query) AS rank
FROM apod, to_tsquery('neutrino|(dark & matter)') query
WHERE query @@ textsearch
ORDER BY rank DESC
LIMIT 10;
```

title	rank
Neutrinos in the Sun	3.1
The Sudbury Neutrino Detector	2.4
A MACHO View of Galactic Dark Matter	2.01317
Hot Gas and Dark Matter	1.91171
The Virgo Cluster: Hot Plasma and Dark Matter	1.90953
Rafting for Solar Neutrinos	1.9
NGC 4650A: Strange Galaxy and Dark Matter	1.85774
Hot Gas and Dark Matter	1.6123
Ice Fishing for Cosmic Neutrinos	1.6
Weak Lensing Distorts the Universe	0.818218

同じ例を正規化ランキングを使ったものを示します。

```
SELECT title, ts_rank_cd(textsearch, query, 32 /* rank/(rank+1) */ ) AS rank
FROM apod, to_tsquery('neutrino|(dark & matter)') query
WHERE query @@ textsearch
ORDER BY rank DESC
LIMIT 10;
```

title	rank
Neutrinos in the Sun	0.756097569485493
The Sudbury Neutrino Detector	0.705882361190954
A MACHO View of Galactic Dark Matter	0.668123210574724
Hot Gas and Dark Matter	0.65655958650282

294/2500

The Virgo Cluster: Hot Plasma and Dark Matter	0.656301290640973
Rafting for Solar Neutrinos	0.655172410958162
NGC 4650A: Strange Galaxy and Dark Matter	0.650072921219637
Hot Gas and Dark Matter	0.617195790024749
Ice Fishing for Cosmic Neutrinos	0.615384618911517
Weak Lensing Distorts the Universe	0.450010798361481

ランキングは、I/Oバウンドであるが故に遅い処理である、文書の`tsvector`へのアクセスが必要なので、高価な処理であるかもしれない。不幸なことに、実際の間合わせでは往々にして大量の検索結果が生じるため、これはほとんど不可避であると言えます。

12.3.4. 結果の強調

検索結果を表示する際には、文書の該当部分を表示し、どの程度間合わせと関連しているかを示すのが望ましいです。PostgreSQLはこの機能を実装した`ts_headline`関数を提供しています。

```
ts_headline([ config regconfig, ] document text, query tsquery [, options text ]) returns text
```

`ts_headline`は、間合わせと一緒に文書を受け取り、間合わせが注目した文書中の語句を抜粋して返します。文書をパースするのに使われる設定を`config`で指定することができます。`config`が省略された場合は、`default_text_search_config`設定が使われます。

`options`文字列を指定する場合は、一つ以上の`option=value`のペアをカンマで区切ったものでなければなりません。

- `StartSel`, `StopSel`: 文書中に表れる間合わせ単語を区切るこの文字列は、他の抜粋される単語と区別されます。これらの文字列が空白やカンマを含んでいる場合は、二重引用符で囲う必要があります。
- `MaxWords`, `MinWords`: この数字を使って見出しの最大の長さで最小の長さを指定します。
- `ShortWord`: この長さか、それ以下の長さの単語は、見出しの最初と最後から削除されます。デフォルト値の3は、常用される英語の冠詞を取り除きます。
- `HighlightAll`: 論理値フラグ; `true`なら文書全体が見出しの様にハイライトされ、前の3つのパラメータは無視されます。
- `MaxFragments`: 表示するテキスト引用句、もしくは断片の最大数です。デフォルト値の0は断片化を起こさない見出しの生成の選択となります。0より大きい場合は断片化を基本とした見出しの生成の選択となります。この方法は、可能な限り多くの検索単語でテキスト断片を探し出し、検索単語周辺のそれらのテキスト断片を広げます。結果として、検索単語が各断片の中央部分近くに位置し、両端に単語を持つこととなります。各断片は最大で`MaxWords`と同数の単語を持ち、`ShortWord`より少ないサイズの単語を断片の両端に持ちません。もし全ての検索単語を文書から見つけられなかった場合は、文書中の最初の`MinWords`分の単語数から成る一つの断片が表示されるでしょう。
- `FragmentDelimiter`: 複数の断片が表示される時、その断片はこの文字で区切られます。

指定されなかったオプションの値は以下のデフォルトになります。

```
StartSel=<b>, StopSel=</b>,
MaxWords=35, MinWords=15, ShortWord=3, HighlightAll=FALSE,
MaxFragments=0, FragmentDelimiter=" ... "
```

例を示します。

```
SELECT ts_headline('english',
  'The most common type of search
  is to find all documents containing given query terms
  and return them in order of their similarity to the
  query.',
  to_tsquery('query & similarity'));
          ts_headline
-----
containing given <b>query</b> terms
and return them in order of their <b>similarity</b> to the
<b>query</b>.

SELECT ts_headline('english',
  'The most common type of search
  is to find all documents containing given query terms
  and return them in order of their similarity to the
  query.',
  to_tsquery('query & similarity'),
  'StartSel = <, StopSel = >');
          ts_headline
-----
containing given <query> terms
and return them in order of their <similarity> to the
<query>.
```

ts_headlineは、tsvectorの要約ではなく、元の文書を使います。ですので遅い可能性があり、注意深く使用する必要があります。よくある間違いは、たった10個の文書を表示しようとしているのに、すべての合致した文書にts_headlineを適用することです。SQLの副問い合わせがこのときに役に立ちます。例を示します。

```
SELECT id, ts_headline(body, q), rank
FROM (SELECT id, body, q, ts_rank_cd(ti, q) AS rank
      FROM apod, to_tsquery('stars') q
      WHERE ti @@ q
      ORDER BY rank DESC
      LIMIT 10) AS foo;
```

12.4. 追加機能

この節では、全文検索に関連する便利な追加の関数と演算子を説明します。

12.4.1. 文書の操作

[項12.3.1](#)に、もとのテキスト形式の文書がどのようにして`tsvector`に変換されるのか書いてあります。また、PostgreSQLでは`tsvector`形式に変換済の文書进行操作する関数と演算子が提供されています。

```
tsvector || tsvector
```

`tsvector`の結合演算子で、2つのベクターの語彙素と位置情報を合成し、`tsvector`を返します。右辺のベクターの位置は左辺のベクターの一番大きな位置情報のオフセットになります。その結果、この関数の結果は、元の文書を結合したものに`to_tsvector`を適用したものとほぼ同じになります(まったく同じと言うわけではありません。左辺の引数の最後の位置にあるストップワードは取り除かれるのに対し、テキストの結合が行われた場合は、その影響が右辺の引数にある語彙素位置に影響を与えるからです)。

`to_tsvector`を適用する前のテキストを結合するよりも、ベクターを結合することの利点の一つは、文書の異なる部分をパースするために、異なる設定を使うことができることです。なお、`setweight`関数は与えられたベクターのすべての語彙素を同じ方法でマーク付けするため、もしも文書に異なる部分に別の重み付けを行いたいなら、結合する前に文書をパースして`setweight`を適用することが必要です。

```
setweight(vector tsvector, weight "char") returns tsvector
```

`setweight`は、A, B, C, Dのいずれかの与えられた`weight`を入力したベクター中の位置にラベル付けし、そのコピーを返します(Dは新しいベクターのデフォルトで、出力する際には表示されません)。これらのラベルはベクターが結合される際に保存されるので、ランキング関数によって文書中の異なる部分の語が別々に重み付けすることができます。

なお、重み付けラベルは語彙素ではなく位置に与えられることに注意してください。入力したベクターから位置が削除されていると、`setweight`は何もしません。

```
length(vector tsvector) returns integer
```

ベクター中に格納されている語彙素の数を返します。

```
strip(vector tsvector) returns tsvector
```

位置、重みの情報がないことを除けば入力したベクターと同じ語彙素のリストを持つベクターを返します。返却されたベクターは、情報を削除されていないベクターに比べてランキングに関しては、ずっと有用性が低くなりますが、通常非常に小さくなります。

12.4.2. 問い合わせを操作する

[項12.3.2](#)は、元のテキストがいかにして`tsquery`値に変換されるかを解説しています。またPostgreSQLは、`tsquery`形式に変換済の問い合わせを操作するために使用できる関数と演算子を提供しています。

```
tsquery && tsquery
```

2つの問い合わせをANDで結合したものを返します。

```
tsquery || tsquery
```

2つの問い合わせをORで結合したものを返します。

```
!! tsquery
```

与えられた問い合わせの否定を返します。

```
numnode(query tsquery) returns integer
```

`tsquery`中のノード(語彙素と演算子)の数を返します。この関数は、問い合わせが意味のあるものか(返却値 > 0)、ストップワードだけを含んでいるか(返却値 0)を判断するのに役に立ちます。例を示します。

```
SELECT numnode(plainto_tsquery('the any'));
NOTICE:  query contains only stopword(s) or doesn't contain lexeme(s), ignored
 numnode
-----
```



```

0
SELECT numnode('foo & bar'::tsquery);
numnode
-----
3

querytree(query tsquery) returns text

```

インデックス検索の際に使用できる`tsquery`の部分を返します。この関数は、たとえばストップワードのみ、あるいは否定語だけのように、インデックス検索できない問い合わせを検出するのに役立ちます。例を示します。

```

SELECT querytree(to_tsquery('!defined'));
querytree
-----

```

12.4.2.1. 問い合わせの書き換え

`ts_rewrite` ファミリー関数は、与えられた`tsquery`から目的の副問い合わせ部分を探し、それを代わりの副問い合わせに置き換えます。本質的には、この操作は、部分文字列置き換えの`tsquery`版です。置き換え候補と置き換え内容の組は、問い合わせ書き換えルールであると考えられます。そのような書き換えルールの集合は、強力な検索ツールとなり得ます。たとえば、同義語(たとえば`new york`, `big apple`, `nyc`, `gotham`)を使って問い合わせをより広範囲にしたり、逆によりホットな話題にユーザを導くために問い合わせを狭い範囲に絞ったりすることができます。この機能と、同義語辞書(項12.6.4)の間には、機能的な重複があります。しかし、再インデックス付けすることなしに、その場で書き換えルールを変更できるのに対し、同義語辞書の更新が有効になるためには、再インデックス付けを行わなければなりません。

```
ts_rewrite (query tsquery, target tsquery, substitute tsquery) returns tsquery
```

この形式の `ts_rewrite` は、単純に単一の書き換えルールを適用します。`query`中に表れる`target`は、`substitute`ですべて置き換えられます。例を示します。

```

SELECT ts_rewrite('a & b'::tsquery, 'a'::tsquery, 'c'::tsquery);
ts_rewrite
-----
'b' & 'c'

```

```
ts_rewrite (query tsquery, select text) returns tsquery
```

この形式の`ts_rewrite`は、開始問い合わせと、テキスト文字列で与えられるSQLの`SELECT`コマンドを受け取ります。`SELECT`は、`tsquery`型の2つの列を出力しなければなりません。現在の問い合わせは、`SELECT`のそれぞれの結果行中の最初の列の結果(ターゲット)が、2番目の列の結果(置き換え値)に、置き換えられます。例を示します。

```

CREATE TABLE aliases (t tsquery PRIMARY KEY, s tsquery);
INSERT INTO aliases VALUES('a', 'c');

SELECT ts_rewrite('a & b'::tsquery, 'SELECT t,s FROM aliases');
ts_rewrite
-----
'b' & 'c'

```

なお、複数の書き換えルールを適用する際は、適用する順番が重要です。ですから、実際には並び替えのキーを適用する`ORDER BY`を問い合わせに入れておくのがよいでしょう。

天文学上の実際的な例を考えてみます。テーブル駆動の書き換えルールを使って、`supernovae`を展開します。

```

CREATE TABLE aliases (t tsquery primary key, s tsquery);
INSERT INTO aliases VALUES(to_tsquery('supernovae'), to_tsquery('supernovae|sn'));

SELECT ts_rewrite(to_tsquery('supernovae & crab'), 'SELECT * FROM aliases');
ts_rewrite
-----
'crab' & ( 'supernova' | 'sn' )

```

テーブルを更新するだけで、書き換えルールを変更することができます。

```

UPDATE aliases
SET s = to_tsquery('supernovae|sn & !nebulae')
WHERE t = to_tsquery('supernovae');

SELECT ts_rewrite(to_tsquery('supernovae & crab'), 'SELECT * FROM aliases');
ts_rewrite
-----
'crab' & ( 'supernova' | 'sn' & !'nebula' )

```

書き換えルールが多くなると、書き換えが遅くなる可能性があります。なぜなら、書き換えの対象になるものを求めて、すべてのルールをチェックするからです。明らかに使われないルールを取り除くために、`tsquery`の含有演算子を使うことができます。以下の例では、元の問い合わせにマッチするルールだけを選ぶことができます。

```
SELECT ts_rewrite('a & b'::tsquery,
                  'SELECT t,s FROM aliases WHERE ''a & b'::tsquery @> t');
ts_rewrite
-----
'b' & 'c'
```

12.4.3. 自動更新のためのトリガ

`tsvector`形式の文書を格納するために別の列を使う場合、文書の内容を格納した列が変更されたときに`tsvector`を格納した列を更新するトリガを作っておく必要があります。この目的のために、2つの組み込み関数を利用できます。自分で関数を書くこともできます。

```
tsvector_update_trigger(tsvector_column_name, config_name, text_column_name [, ... ])
tsvector_update_trigger_column(tsvector_column_name, config_column_name, text_column_name [, ... ])
```

これらのトリガ関数は、1つ以上のテキスト列から、`CREATE TRIGGER`コマンドで指定されたパラメータの制御により、`tsvector`列を自動的に計算します。使い方の例を示します。

```
CREATE TABLE messages (
    title      text,
    body       text,
    tsv        tsvector
);

CREATE TRIGGER tsvectorupdate BEFORE INSERT OR UPDATE
ON messages FOR EACH ROW EXECUTE PROCEDURE
tsvector_update_trigger(tsv, 'pg_catalog.english', title, body);

INSERT INTO messages VALUES('title here', 'the body text is here');

SELECT * FROM messages;
   title   |          body          |          tsv
-----+-----+-----
title here | the body text is here | 'bodi':4 'text':5 'titl':1

SELECT title, body FROM messages WHERE tsv @@ to_tsquery('title & body');
   title   |          body
-----+-----
title here | the body text is here
```

このトリガを作っておくことにより、`title` または `body` への変更は、アプリケーションで考慮しなくても自動的に`tsv`に反映されます。

トリガの最初の引数は更新対象の`tsvector`の列名でなければなりません。2番目の引数は、変換を実行する際に使用されるテキスト検索の設定です。`tsvector_update_trigger`では、設定の名前は単に2番目のトリガ引数で与えられます。上で示すように、スキーマ修飾されていなければなりません。`search_path`の変更がトリガの振る舞いに影響を与えないためです。`tsvector_update_trigger_column`では、2番目のトリガ引数は別のテーブル列の列名です。この列の型は`regconfig`でなければなりません。この方法により、設定を行単位で変えることができます。残りの引数はテキスト型(`text`, `varchar`, `char`のいずれか)の列の名前です。与えられた順に、文書中に取り込まれます。NULL値はスキップされます(ただし、それ以外の列はインデックス付けされます)。

これらの組み込みトリガの制限事項として、すべての列を同じようにしか扱えないというものがあります。それぞれの列を違うように扱うには — たとえば本文とタイトルの重みを変えとか —、カスタムトリガを書く必要があります。トリガ言語としてPL/pgSQLを使った例を示します。

```
CREATE FUNCTION messages_trigger() RETURNS trigger AS $$
begin
    new.tsv :=
        setweight(to_tsvector('pg_catalog.english', coalesce(new.title, '')), 'A') ||
        setweight(to_tsvector('pg_catalog.english', coalesce(new.body, '')), 'D');
    return new;
end
$$ LANGUAGE plpgsql;

CREATE TRIGGER tsvectorupdate BEFORE INSERT OR UPDATE
ON messages FOR EACH ROW EXECUTE PROCEDURE messages_trigger();
```

`tsvector`値をトリガ内で作るときには、設定名を明示的に与えることが重要であることを銘記しておいてください。そうすれば、`default_text_search_config`が変更されても列の内容は影響を受けません。これを怠ると、ダンプしてリロードすると検索結果が変わってしまうような問題が起きる可能性があります。

12.4.4. 文書の統計情報の収集

`ts_stat`関数は、設定をチェックしたり、ストップワードの候補を探すのに役立ちます。

```
ts_stat(sqlquery text, [ weights text, ]
      OUT word text, OUT ndoc integer,
      OUT nentry integer) returns setof record
```

*sqlquery*は単一の *tsvector* 列を返すSQL問い合わせのテキスト値です。 *ts_stat* は問い合わせを実行し、 *tsvector* データに含まれる語彙素(単語)各々の統計情報を返します。返却される列は以下のものです。

- *word* text — 語彙素の値
- *ndoc* integer — 単語が含まれる文書(*tsvector*)の数
- *nentry* integer — 含まれる単語の数

weights が与えられていたら、その重みを持つものだけがカウントされます。

たとえば、文書中もっとも頻繁に現れる単語の上位10位を探すには以下のようにします。

```
SELECT * FROM ts_stat('SELECT vector FROM apod')
ORDER BY nentry DESC, ndoc DESC, word
LIMIT 10;
```

同じ例で、重みがAかBの単語だけをカウントするには、以下のようにします。

```
SELECT * FROM ts_stat('SELECT vector FROM apod', 'ab')
ORDER BY nentry DESC, ndoc DESC, word
LIMIT 10;
```

[前のページ](#)
テキスト検索の制御

[ホーム](#)
[上に戻る](#)

[次のページ](#)
パーサ

12.5. パーサ

テキスト検索パーサは、もとの文書テキストを分割してトークンに変換し、それぞれのトークンの型を識別子する役割を持っています。ここで、可能な型の集合は、パーサ自身が定義します。パーサは文書をまったく変更しないことに注意してください。それは、単に可能な単語の境界を識別するだけです。このような制限があるため、カスタム辞書を作るのに比べ、用途限定のカスタムパーサを作る必要性は少ないです。今のところ、PostgreSQLはたった一つの組み込みパーサを提供しています。これは広い範囲の用途に対して有用であると考えられています。

組み込みのパーサはpg_catalog.defaultというものです。表12-1に示す23のトークンを理解します。

表 12-1. Default Parser's Token Types

別名	説明	例
asciiword	単語、すべてのASCII文字	elephant
word	単語、すべての文字	mañana
numword	単語、文字、数字	beta1
asciihword	ハイフンでつながれた単語、すべてのASCII	up-to-date
hword	ハイフンでつながれた単語、すべての文字	lógico-matemática
numhword	ハイフンでつながれた単語、すべての文字、数字	postgresql-beta1
hword_asciipart	ハイフンでつながれた単語の一部、すべての ASCII	postgresql-beta1のpostgresql
hword_part	ハイフンでつながれた単語の一部、すべての文字	lógico-matemáticaのlógicoまたはmatemática
hword_numpart	ハイフンでつながれた単語の文字+数字の部分	postgresql-beta1のbeta1
email	電子メールアドレス	foo@example.com
protocol	プロトコルヘッダー	http://
url	URL	example.com/stuff/index.html
host	ホスト名	example.com
url_path	URL中のパス名	URL中の/stuff/index.html
file	ファイルまたはパス名	URL中でない/usr/local/foo.txt
sfloat	科学技術表記	-1.234e56
float	10進表記	-1.234
int	符号付き整数	-1234
uint	符号なし整数	1234
version	バージョン番号	8.3.0
tag	XMLタグ	
entity	XMLエンティティ	&
blank	空白記号	(他のものに解釈できない空白または句読点)

注意: パーサにとっての"文字"は、データベースのロケールの設定、特にlc_ctypeによって決まります。基本的なASCIIのみを含む単語は、別のトークン型として報告されます。ときには、それらを他と区別することが有用だからです。ヨーロッパのたいていの言語では、word と asciiwordは、同じように扱われます。

emailはRFC5322で定義されたすべての有効なメールアドレス文字をサポートしません。メールアドレスのユーザ名としてサポートされる英数字以外の文字はピリオド、ダッシュ、アンダースコアのみです。

パーサがテキストの同じ部分から重複したトークンを生成することはあり得ます。たとえば、ハイフン付の単語は、単語全体と、各部分の両方を報告します。例を示します。

```
SELECT alias, description, token FROM ts_debug('foo-bar-beta1');
alias      | description                                     | token
-----+-----+-----
numhword   | Hyphenated word, letters and digits             | foo-bar-beta1
hword_asciipart | Hyphenated word part, all ASCII                 | foo
blank      | Space symbols                                   | -
hword_asciipart | Hyphenated word part, all ASCII                 | bar
```

blank | Space symbols | -
hword_numpart | Hyphenated word part, letters and digits | beta1

この挙動は好ましいものです。単語全体と、各々の部分の両方に対して検索ができるからです。初歩的な別の例を示します。

```
SELECT alias, description, token FROM ts_debug('http://example.com/stuff/index.html');
  alias      | description      | token
-----+-----+-----
protocol | Protocol head | http://
url      | URL             | example.com/stuff/index.html
host     | Host            | example.com
url_path | URL path        | /stuff/index.html
```

12.6. 辞書

辞書は、検索の対象とならない単語(ストップワード)を削除するために使われます。また、同じ単語から派生した異なる形態の単語が照合するようにするために、単語を正規化するためにも使われます。検索の品質を向上するという面以外にも、正規化とストップワードの削除は、`tsvector`表現の文書のサイズを小さくし、結果として性能を向上させます。正規化は常に言語学的な意味を持つとは限らず、通常は用途の意味論に依存します。

正規化の例を示します。

- 言語学的 - Ispell辞書は入力された単語を正規化された形式に変換しようとします。語幹辞書は単語の終了部を削除します。
- 以下のようなURLが同一のURLに一致するように正規化することができます。
 - `http://www.pgsql.ru/db/mw/index.html`
 - `http://www.pgsql.ru/db/mw/`
 - `http://www.pgsql.ru/db/./db/mw/index.html`
- 色の名前は、16進値に変換できます。例: `red, green, blue, magenta -> FF0000, 00FF00, 0000FF, FF00FF`
- 数をインデックス付けする際には、可能な範囲を縮小するために、端数を削除することができます。たとえば、もし正規化後に小数点未満2桁を保持するならば、`3.1415926`, `3.14`は同じことになります。

辞書は、トークンを入力し、以下を返すプログラムです。

- 入力が辞書に登録されていれば語彙素の配列(一つのトークンが一つ以上の語彙素を生成する可能性があることに注意してください)
- 辞書が入力を認識しないが、ストップワードであることは認識する場合は空の配列
- 元々のトークンを新規のトークンに置き換え、それに続く辞書にその新規トークン渡す場合は、`TSL_FILTER`フラグセットを伴う単一の語彙素(このような置き換え機能をもつ辞書はフィルタリング辞書と呼ばれます)
- 辞書が入力トークンを認識しない場合は`NULL`

PostgreSQLは、多くの言語に定義済みの辞書を提供しています。また、カスタムパラメータを使った新しい辞書を作るためのテンプレートも利用できます。定義済みの辞書のテンプレートについては、以下で述べています。今あるテンプレートが適当でないのなら、新しいものを作ることもできます。例は、PostgreSQLの配布物の`contrib/`をご覧ください。

テキスト検索設定は、パーサと、パーサの出力トークンを処理する辞書の集合を結び付けます。パーサが返却する各々のトークン型に対して、設定で辞書のリストを指定します。パーサがあるトークン型を見つけると、ある辞書が単語を認識するまでリスト中の辞書が順番に調べられます。ストップワードであるか、あるいはどの辞書もトークンを認識しない場合はそれは捨てられ、インデックス付けや検索の対象となりません。通常、非`NULL`を返す最初の辞書の出力が結果を決めることになり、他の残りの辞書は参照されません。しかし、フィルタリング辞書は与えられたワードを変更し、それを続く辞書へ渡すことができます。

辞書をリストする一般的な方法は、まずもっとも範囲の狭い、特定用途向の辞書を配置し、次にもっと一般的な辞書を置き、最後にSnowball語幹処理や`simple`辞書のような、すべてを認識する非常に一般的な辞書を置くことです。たとえば、天文学向の検索では(`astro_en`設定)では、`asciiword` (ASCII単語)型を天文学用語の同義語辞書、一般的な英語辞書、そしてSnowball英語語幹辞書に結び付けることができます。

```
ALTER TEXT SEARCH CONFIGURATION astro_en
ADD MAPPING FOR asciiword WITH astrosyn, english_ispell, english_stem;
```

フィルタリング辞書は、リスト中の好きな場所へ配置できます。(役に立たなくなるリストの最後を除きます。)フィルタリング辞書は、後続の辞書の処理を単純化するために、一部の文字の正規化を行うのに有用です。例えば、フィルタリング辞書は`unaccent`モジュールで実施される様な、アクセント記号が付与された文字からアクセント記号を取り除くのに使用することができます。

12.6.1. ストップワード

ストップワードは、ほとんどすべての文書に現れるような非常に一般的で、ほかのものと同じようには扱う価値のない単語です。ですから、全文検索の際には無視して構いません。たとえば、すべての英語のテキストは`a`や`the`のような単語を含んでお

り、インデックスの中にそれらを入れても役に立ちません。しかし、ストップワードは`tsvector`中の位置に影響を与えるので、結局ランキングにも影響があります。

```
SELECT to_tsvector('english','in the list of stop words');
       to_tsvector
-----
'list':3 'stop':5 'word':6
```

位置1, 2, 4は、ストップワードのために失われています。ストップワード有無により、文書のために計算されたランクは非常に影響を受けます。

```
SELECT ts_rank_cd (to_tsvector('english','in the list of stop words'), to_tsquery('list & stop'));
       ts_rank_cd
-----
          0.05

SELECT ts_rank_cd (to_tsvector('english','list stop words'), to_tsquery('list & stop'));
       ts_rank_cd
-----
          0.1
```

ストップワードをどのように扱うかは、特定の辞書に任されています。例えば、`ispell`辞書はまず単語を正規化し、そして、ストップワードのリストを検索します。一方、`snowball`語幹抽出はまずストップワードのリストを検査します。動作が異なる理由は、ノイズが紛れ込む可能性を減らすことです。

12.6.2. simple辞書

`simple`辞書テンプレートは、入力トークンを小文字に変換し、ストップワードのファイルに対してチェックすることによって動作します。もしファイルの中にあれば、空の配列が返却され、そのトークンは捨てられます。そうでないときは、小文字形式の単語が正規化された語彙素として返却されます。別の方法としては、ストップワードではないものは、認識できないものとすることもできます。そうすることにより、それらをリスト中の次の辞書に渡すことができます。

`simple`テンプレートを使った辞書定義の例を示します。

```
CREATE TEXT SEARCH DICTIONARY public.simple_dict (
    TEMPLATE = pg_catalog.simple,
    STOPWORDS = english
);
```

ここで、`english`は、ストップワードファイルのベースネームです。ファイルのフルネームは、`$SHAREDIR/tsearch_data/english.stop`です。`$SHAREDIR`は、PostgreSQLインストール先の共有データディレクトリです。これは、よく`/usr/local/share/postgresql`に置いてあります。(良くわからない場合は`pg_config --sharedir`を使ってください)。ファイル形式は、単に1行ごとに単語を書くだけです。空行と、後方の空白は無視されます。大文字は小文字に変換されます。このファイルの内容に関する処理はこれだけです。

これで辞書のテストができます。

```
SELECT ts_lexize('public.simple_dict','Yes');
       ts_lexize
-----
{yes}

SELECT ts_lexize('public.simple_dict','The');
       ts_lexize
-----
{}
```

また、ストップワードファイルの中に見つからないときに、小文字に変換した単語を返す代りに、`NULL`を返すことを選ぶこともできます。この挙動は、辞書の`Accept`パラメータを`false`に設定することで選択されます。さらに例を続けます。

```
ALTER TEXT SEARCH DICTIONARY public.simple_dict ( Accept = false );

SELECT ts_lexize('public.simple_dict','Yes');
       ts_lexize
-----

SELECT ts_lexize('public.simple_dict','The');
       ts_lexize
-----
{}
```

デフォルト設定の`Accept = true`では、`simple`辞書は、辞書リストの最後に置かなければ意味がありません。なぜなら、後続の辞書にトークンを渡すことがないからです。逆に`Accept = false`は、後続の辞書が少なくとも一つはあるときに意味があります。

注意

ほとんどの辞書の形式は、ストップワードファイルのように設定ファイルに依存します。これらのファイルは必ずUTF-8エンコーディングにしてください。サーバのエンコーディングがUTF-8でない場合は、サーバに読み込まれる際に実際のデータベースエンコーディングに変換されます。

注意

通常、辞書はデータベースセッションの中で最初に使われる際に、一度だけ読み込まれます。辞書を変更し、現在使われているセッションの中で新しい内容が読み込まれるようにしたい場合は、その辞書に対してALTER TEXT SEARCH DICTIONARYを発行してください。これは実際にはどんなパラメータ値をも変更しない"ダミー"の更新でよいです。

12.6.3. 同義語辞書

この辞書テンプレートは、単語を同義語に置き換える辞書を作るために使われます。語句はサポートされていません(そのためには類語テンプレート([項12.6.4](#))を使ってください)。同義語辞書は、言語学的な問題、たとえば、英語語幹辞書が"Paris"という単語を"pari"に縮小してしまうのを防ぎます。Paris pariという行を同義語辞書に登録し、english_stem辞書の前に置くようにするだけでよいのです。下記はその例です。

```
SELECT * FROM ts_debug('english', 'Paris');
  alias | description | token | dictionaries | dictionary | lexemes
-----+-----+-----+-----+-----+-----
asciiword | Word, all ASCII | Paris | {english_stem} | english_stem | {pari}

CREATE TEXT SEARCH DICTIONARY my_synonym (
  TEMPLATE = synonym,
  SYNONYMS = my_synonyms
);

ALTER TEXT SEARCH CONFIGURATION english
  ALTER MAPPING FOR asciiword
  WITH my_synonym, english_stem;

SELECT * FROM ts_debug('english', 'Paris');
  alias | description | token | dictionaries | dictionary | lexemes
-----+-----+-----+-----+-----+-----
asciiword | Word, all ASCII | Paris | {my_synonym,english_stem} | my_synonym | {paris}
```

synonym テンプレートに必要なパラメータはSYNONYMSだけで、その設定ファイルのベースネームです — 上の例ではmy_synonymsです。ファイルのフルネームは、\$SHAREDIR/tsearch_data/my_synonyms.syn となります(ここで\$SHAREDIRは、PostgreSQLをインストールした際の、共有データディレクトリです)。ファイルの形式は、置き換え対象の1単語につき1行で、単語には空白で区切られた同義語が後に続きます。空行、後方の空白は無視されます。

synonym テンプレートはまた、CaseSensitiveというオプションパラメータを持っており、デフォルトはfalseです。CaseSensitiveがfalseの時は、同義語辞書内の単語は入力トークンと同様に小文字に変換されます。trueの時は、単語とトークンは小文字に変換されずそのまま比較されます。

アスタリスク(*)は設定ファイル中の同義語の最後に付与することができます。これは同義語を接頭語とすることを意味します。アスタリスクは、エントリがto_tsvector()で使用される場合には無視されますが、to_tsquery()で使用される場合、結果は前方一致を伴った問い合わせになるでしょう。(詳しくは[項12.3.2](#)を見てください。)例えば、\$SHAREDIR/tsearch_data/synonym_sample.synに以下の様なエントリをもっていたとします。

```
postgres      postgresql
postgresql    postgresql
postgre       postgresql
google        googl
indices       index*
```

この場合、次のような結果を得ることになります。

```
mydb=# CREATE TEXT SEARCH DICTIONARY syn (template=synonym, synonyms='synonym_sample');
mydb=# SELECT ts_lexize('syn','indices');
 ts_lexize
-----
 {index}
(1 row)

mydb=# CREATE TEXT SEARCH CONFIGURATION tst (copy=simple);
mydb=# ALTER TEXT SEARCH CONFIGURATION tst ALTER MAPPING FOR asciiword WITH syn;
mydb=# SELECT to_tsvector('tst','indices');
 to_tsvector
-----
 {index}
```

```
-----
'index':1
(1 row)

mydb=# SELECT to_tsquery('tst','indices');
      to_tsquery
-----
'index':*
(1 row)

mydb=# SELECT 'indexes are very useful'::tsvector;
      tsvector
-----
'are' 'indexes' 'useful' 'very'
(1 row)

mydb=# SELECT 'indexes are very useful'::tsvector @@ to_tsquery('tst','indices');
      ?column?
-----
t
(1 row)
```

12.6.4. 類語辞書

類語辞書(TZと略されることがあります)は、単語と語句の関係情報を集めたものです。つまり、広義用語(BT)、狭義用語(NT)、優先用語、非優先用語、関連用語などです。

基本的には、類語辞書は、非優先用語を優先用語に置き換え、オプションで元の用語もインデックス付けのため保存します。PostgreSQLの現在の類語辞書の実装は、同義語辞書を拡張し、語句のサポートを追加したものです。類語辞書は、以下のようなフォーマットの設定ファイルを必要とします。

```
# this is a comment
sample word(s) : indexed word(s)
more sample word(s) : more indexed word(s)
...
```

ここで、コロン(:)は、語句とその置き換え対象の区切りです。

類語辞書は、副辞書(辞書設定で指定します)を、一致する語句をチェックする前に入力テキストを正規化するために使います。副辞書はただ一つだけ選べます。副辞書が単語を認識できない場合はエラーが報告されます。その場合は、その単語の利用を止めるか、副辞書にそのことを教えなければなりません。アスタリスク(*)をインデックス付けされた単語の先頭に置くことにより、副辞書の適用をスキップできます。しかしながら、すべてのサンプルの単語は、副辞書に認識されなければなりません。

複数の類語が照合するときは、類語辞書はもっとも長いものを選びます。そして、語句は、最後の定義を使って分解されます。

特定のストップワードを副辞書に認識するように指定することはできません。その代り、ストップワードが出現する位置を?でマークします。たとえば、aとtheが副辞書によればストップワードだったとします。

```
? one ? two : ssws
```

は、a one the twoとthe one a twoに照合します。そして、両方ともsswsに置き換えられます。

類語辞書は語句を認識することができるので、状態を記憶してパーサと連携を保たなければなりません。類語辞書は、この機能を使って次の単語を引き続き処理するのか、単語の蓄積を止めるのかを決定します。類語辞書の設定は注意深く行わなければなりません。たとえば、類語辞書がasciwordトークンだけを扱うようになっている場合、one 7のような類語辞書の定義は、トークン型 uintが類語辞書にアサインされていないので動きません。

注意
類語辞書はインデックス付けの際に利用されるので、類語辞書を設定変更すると、再インデックス付けが必要になります。他のほとんどの辞書では、ストップワードを追加あるいは削除するような小さな変更は、インデックス付けを必要としません。

12.6.4.1. 類語設定

新しい類語辞書を定義するには、thesaurusテンプレートを使います。例を示します。

```
CREATE TEXT SEARCH DICTIONARY thesaurus_simple (
    TEMPLATE = thesaurus,
    DictFile = mythesaurus,
    Dictionary = pg_catalog.english_stem
);
```

ここで、

- `thesaurus_simple`は新しい辞書の名前です。
- `mythesaurus`は、類語設定ファイルのベースネームです。(フルパスは、`$SHAREDIR/tsearch_data/mythesaurus.ths`となります。ここで、`$SHAREDIR`はインストール時の共有データディレクトリです。)
- 類語正規化で使用する`pg_catalog.english_stem`は副辞書です(ここでは、Snowball英語語幹辞書)。副辞書にはそれ用の設定(たとえばストップワード)があることに注意してください。ここではそれは表示していません。

これで、類語辞書`thesaurus_simple`を、設定中の希望のトークンにバインドすることができるようになります。例を示します。

```
ALTER TEXT SEARCH CONFIGURATION russian
  ALTER MAPPING FOR asciiword, asciihword, hword_asciipart
  WITH thesaurus_simple;
```

12.6.4.2. 類語の例

天文学の単語の組合わせを含む単純な天文学用の`thesaurus_astro`類語を考えます。

```
supernovae stars : sn
crab nebulae : crab
```

以下で辞書を作り、トークン型を天文学類語辞書と英語の語幹辞書に結び付けます。

```
CREATE TEXT SEARCH DICTIONARY thesaurus_astro (
  TEMPLATE = thesaurus,
  DictFile = thesaurus_astro,
  Dictionary = english_stem
);

ALTER TEXT SEARCH CONFIGURATION russian
  ALTER MAPPING FOR asciiword, asciihword, hword_asciipart
  WITH thesaurus_astro, english_stem;
```

さあ、これでどのように動くか試せます。`ts_lexize`は類語をテストする目的にはあまり有用ではありません。なぜなら、それは入力を単一のトークンとして扱うからです。その代りに、`plainto_tsquery`と`to_tsvector`を使って入力文字列を複数のトークンに分解します。

```
SELECT plainto_tsquery('supernova star');
plainto_tsquery
-----
'sn'

SELECT to_tsvector('supernova star');
to_tsvector
-----
'sn':1
```

原則として、引数を引用符で囲めば`to_tsquery`が使えます。

```
SELECT to_tsquery(''supernova star'');
to_tsquery
-----
'sn'
```

`english_stem`語幹辞書を同義語辞書の定義時に指定したので、`supernova star`が`thesaurus_astro`中の`supernovae stars`に照合していることに注意してください。語幹処理が`e`と`s`を削除しています。

置き換え後の語句とオリジナルの語句の両方をインデックス付けするには、定義の右項にオリジナルを追加するだけで良いです。

```
supernovae stars : sn supernovae stars

SELECT plainto_tsquery('supernova star');
plainto_tsquery
-----
'sn' & 'supernova' & 'star'
```

12.6.5. Ispell辞書

Ispell辞書テンプレートは、形態論辞書を提供します。これによって、言語学的に多様な単語の形態を同じ語彙素に変換することができます。たとえば、英語Ispell辞書は、検索語`bank`の語形変化と活用変化、たとえば`banking, banked, banks, banks', and bank's`に 照合します。

PostgreSQLの標準配布には、Ispellの設定ファイルは含まれていません。多くの言語用の辞書がIspellで入手できます。また、

より新しい辞書のフォーマットもサポートされています — [MySpell](#) (OO < 2.0.1) と [Hunspell](#) (OO >= 2.0.2)。多数の辞書のリストが [OpenOffice Wiki](#) で入手できます。

IsPELL辞書を作るには、組み込みの`ispell`テンプレートを使い、いくつかパラメータを指定します。

```
CREATE TEXT SEARCH DICTIONARY english_ispell (  
    TEMPLATE = ispell,  
    DictFile = english,  
    AffFile = english,  
    StopWords = english  
);
```

ここで、`DictFile`、`AffFile`、および`StopWords`は、辞書のベースネーム、接辞ファイル、ストップワードファイルを指定します。ストップワードファイルは、上で説明した`simple`辞書と同じ形式です。ほかのファイルの形式はここでは説明されませんが、上にあげたウェブサイトの説明があります。

IsPELL辞書は通常限られた数の単語を認識します。ですので、なんでも認識できるSnowball辞書のような、より適用範囲の広い辞書による後処理が必要です。

IsPELL辞書を使って複合語を分割することができます。これはPostgreSQLがサポートする優れた機能です。接辞ファイルは、複合語形式の候補になる辞書中の単語に印を付ける`compoundwords controlled`文を使う特別なフラグを指定しなければならないことに注意してください。

```
compoundwords controlled z
```

ノルウェー語の例をいくつか示します。

```
SELECT ts_lexize('norwegian_ispell', 'overbuljongterningpakkemesterassistent');  
{over,buljong,terning,pakk,mester,assistent}  
SELECT ts_lexize('norwegian_ispell', 'sjokoladefabrikk');  
{sjokoladefabrikk,sjokolade,fabrikk}
```

注意: MySpellは複合語をサポートしていません。Hunspellは複合語の高度なサポートを提供しています。いまのところ、PostgreSQLはHunspellの基本的な複合語操作しかサポートしていません。

12.6.6. Snowball辞書

Snowball辞書テンプレートは、有名な「英語用のポーターの語幹アルゴリズム」を発明したMartin Porterのプロジェクトに基づいています。Snowballは今では多くの言語用の語幹アルゴリズムを提供しています(詳細は[Snowballのサイト](#)を参照してください)。各々のアルゴリズムにより、その言語において単語の共通部分を取りだし、基本部もしくは語幹の綴りに縮退させることができます。Snowball辞書には、どの語幹処理を使うかを識別する言語パラメータが必須で、加えて、オプションで無視すべき単語のリストを保持するストップワードファイルを指定することもできます。(PostgreSQLの標準的なストップワードファイルもまたSnowball projectから提供されています)。たとえば、以下と同じ組み込みの定義があります。

```
CREATE TEXT SEARCH DICTIONARY english_stem (  
    TEMPLATE = snowball,  
    Language = english,  
    StopWords = english  
);
```

ストップワードファイルの形式はすでに説明されているものと同じです。

Snowball辞書は、単純化できるかどうかに関係なく、すべての単語を認識するので、辞書リストの最後に置く必要があります。他の辞書の前に置くのは意味がありません。Snowball辞書は決してトークンを次の辞書に渡さないからです。

[前のページ](#)
パーサ

[ホーム](#)
[上に戻る](#)

[次のページ](#)
設定例

12.7. 設定例

テキスト検索設定は、文書を `tsvector` に変換する必要なすべてのオプションを指定します。すなわち、テキストをトークンに分解するパーサ、そしてトークンを語彙素に変換する辞書です。 `to_tsvector` または `to_tsquery` を呼び出すたびに、処理を進めるためにテキスト検索設定が必要になります。設定パラメータの [default_text_search_config](#) は、デフォルトの設定を指定します。これは、明示的な設定が省略されたときにテキスト検索関数が使用します。 `postgresql.conf` に設定するか、個々のセッションで `SET` コマンドを使って設定できます。

既定のテキスト検索設定がいくつか利用できます。また、カスタム設定を作るのも容易です。テキスト検索オブジェクトを管理する機能を実現するために、SQL コマンドが一通り用意されています。テキスト検索オブジェクトに関する情報を表示する `psql` コマンドもいくつか用意されています ([項12.10](#))。

例として、組み込みの `english` 設定のコピーを用いて、新しい `pg` 設定を作ります。

```
CREATE TEXT SEARCH CONFIGURATION public.pg ( COPY = pg_catalog.english );
```

PostgreSQL 固有の同義語リストを使い、それを `$$SHAREDIR/tsearch_data/pg_dict.syn` に格納します。ファイルの内容は以下のようになります。

```
postgres    pg
psql        pg
postgresql  pg
```

同義語辞書を次のように定義します。

```
CREATE TEXT SEARCH DICTIONARY pg_dict (
    TEMPLATE = synonym,
    SYNONYMS = pg_dict
);
```

次に、IsPELL 辞書の `english_ispell` を登録します。これにはそれ自身の設定があります。

```
CREATE TEXT SEARCH DICTIONARY english_ispell (
    TEMPLATE = ispell,
    DictFile = english,
    AffFile = english,
    StopWords = english
);
```

ここで、 `pg` 設定に単語用のマッピングを設定します。

```
ALTER TEXT SEARCH CONFIGURATION pg
    ALTER MAPPING FOR asciiword, asciihword, hword_ascii,
                        word, hword, hword_part
    WITH pg_dict, english_ispell, english_stem;
```

組み込み設定が扱っているいくつかのトークンに関しては、インデックス付けと検索に扱わないことにします。

```
ALTER TEXT SEARCH CONFIGURATION pg
    DROP MAPPING FOR email, url, url_path, sfloat, float;
```

これでここまで作った設定を試すことができます。

```
SELECT * FROM ts_debug('public.pg', '
PostgreSQL, the highly scalable, SQL compliant, open source object-relational
database management system, is now undergoing beta testing of the next
version of our software.
');
```

次に、セッションの中で新しい設定を使うようにします。この設定は、 `public` スキーマの中に作られています。

```
=> \dF
      List of text search configurations
 Schema | Name | Description
-----+-----+-----
 public | pg   |

SET default_text_search_config = 'public.pg';
SET

SHOW default_text_search_config;
 default_text_search_config
-----
 public.pg
```


12.8. テキスト検索のテストとデバッグ

カスタムテキスト検索設定の挙動は複雑になりがちで、結果として混乱を招くことになります。この章では、テキスト検索オブジェクトのテストの際に役に立つ関数を説明します。設定が完全かどうか、パーサと辞書を別々にテストすることなどが可能です。

12.8.1. 設定のテスト

`ts_debug`関数により、テキスト検索設定の容易なテストができます。

```
ts_debug([ config regconfig, ] document text,
OUT alias text,
OUT description text,
OUT token text,
OUT dictionaries regdictionary[],
OUT dictionary regdictionary,
OUT lexemes text[])
returns setof record
```

`ts_debug`は、パーサが生成し、設定された辞書が処理した`document`のすべてのトークンの情報を表示します。その際、`config`で指定した設定が使われます。引数が省略されると`default_text_search_config`が使われます。

`ts_debug`は、パーサが認識したテキスト中のトークンを1行につき一つ返します。返却される列は以下です。

- `alias text` — トークン型の短縮名
- `description text` — トークン型の説明
- `token text` — トークンテキスト
- `dictionaries regdictionary[]` — 設定によってこのトークン型用に選択された辞書
- `dictionary regdictionary` — トークンを認識した辞書。もし認識した辞書がなければ `NULL`
- `lexemes text[]` — トークンを認識した辞書が生成した語彙素。もしどの辞書も認識しなければ`NULL`。(`{}`)が返った場合は、ストップワードとして認識されたことを示す

簡単な例を示します。

```
SELECT * FROM ts_debug('english','a fat cat sat on a mat - it ate a fat rats');
 alias | description | token | dictionaries | dictionary | lexemes
-----+-----+-----+-----+-----+-----
asciiword | Word, all ASCII | a | {english_stem} | english_stem | {}
blank | Space symbols | | {} | | |
asciiword | Word, all ASCII | fat | {english_stem} | english_stem | {fat}
blank | Space symbols | | {} | | |
asciiword | Word, all ASCII | cat | {english_stem} | english_stem | {cat}
blank | Space symbols | | {} | | |
asciiword | Word, all ASCII | sat | {english_stem} | english_stem | {sat}
blank | Space symbols | | {} | | |
asciiword | Word, all ASCII | on | {english_stem} | english_stem | {}
blank | Space symbols | | {} | | |
asciiword | Word, all ASCII | a | {english_stem} | english_stem | {}
blank | Space symbols | | {} | | |
asciiword | Word, all ASCII | mat | {english_stem} | english_stem | {mat}
blank | Space symbols | | {} | | |
blank | Space symbols | - | {} | | |
asciiword | Word, all ASCII | it | {english_stem} | english_stem | {}
blank | Space symbols | | {} | | |
asciiword | Word, all ASCII | ate | {english_stem} | english_stem | {ate}
blank | Space symbols | | {} | | |
asciiword | Word, all ASCII | a | {english_stem} | english_stem | {}
blank | Space symbols | | {} | | |
asciiword | Word, all ASCII | fat | {english_stem} | english_stem | {fat}
blank | Space symbols | | {} | | |
asciiword | Word, all ASCII | rats | {english_stem} | english_stem | {rat}
```

もう少し高度なデモをお見せするために、まず英語用の`public.english`設定と、Ispell辞書を作ります。

```
CREATE TEXT SEARCH CONFIGURATION public.english ( COPY = pg_catalog.english );

CREATE TEXT SEARCH DICTIONARY english_ispell (
    TEMPLATE = ispell,
```

```

DictFile = english,
AffFile = english,
StopWords = english
);

ALTER TEXT SEARCH CONFIGURATION public.english
ALTER MAPPING FOR asciiword WITH english_ispell, english_stem;

SELECT * FROM ts_debug('public.english','The Brightest supernovaes');

```

alias	description	token	dictionaries	dictionary	lexemes
asciiword	Word, all ASCII	The	{english_ispell,english_stem}	english_ispell	{}
blank	Space symbols		{}		
asciiword	Word, all ASCII	Brightest	{english_ispell,english_stem}	english_ispell	{bright}
blank	Space symbols		{}		
asciiword	Word, all ASCII	supernovaes	{english_ispell,english_stem}	english_stem	{supernova}

この例では、単語Brightestは、ASCII word (エイリアスはasciiword)として認識されています。このトークン型のための辞書リストはenglish_ispell とenglish_stemです。この単語はenglish_ispellに認識され、名詞brightへと縮退されています。単語supernovaesはenglish_ispell辞書には認識されず、次の辞書に渡され、幸い認識されました(実際には、english_stemはSnowball辞書で、何でも認識します。それで、この辞書は辞書リストの最後に置かれているわけです)。

単語Theは、english_ispell辞書によってストップワード(項12.6.1)として認識されており、インデックス付けされません。空白も捨てられます。この設定では空白に関する辞書が提供されていないからです。

明示的に見たい列を指定することにより、出力の幅を減らすことができます。

```

SELECT alias, token, dictionary, lexemes
FROM ts_debug('public.english','The Brightest supernovaes');

```

alias	token	dictionary	lexemes
asciiword	The	english_ispell	{}
blank			
asciiword	Brightest	english_ispell	{bright}
blank			
asciiword	supernovaes	english_stem	{supernova}

12.8.2. パーサのテスト

次にあげた関数により、テキスト検索パーサを直接テストすることができます。

```

ts_parse(parser_name text, document text,
OUT tokid integer, OUT token text) returns setof record
ts_parse(parser_oid oid, document text,
OUT tokid integer, OUT token text) returns setof record

```

ts_parseは与えられたdocumentをパースし、パーサが生成したトークンを1行に1個もつ一連のレコードを返します。それぞれのレコードには、割り当てられたトークン型を示すtokidと、テキストのトークンであるtokenが含まれます。例を示します。

```

SELECT * FROM ts_parse('default', '123 - a number');

```

tokid	token
22	123
12	
12	-
1	a
12	
1	number

```

ts_token_type(parser_name text, OUT tokid integer,
OUT alias text, OUT description text) returns setof record
ts_token_type(parser_oid oid, OUT tokid integer,
OUT alias text, OUT description text) returns setof record

```

ts_token_typeは、指定したパーサが認識できるトークン型を記述したテーブルを返します。各々のトークン型に対し、パーサがトークン型をラベル付けするのに使用する整数tokid、設定コマンド中のトークンの名前であるalias、簡単な説明であるdescriptionが含まれます。例を示します。

```

SELECT * FROM ts_token_type('default');

```

tokid	alias	description
1	asciiword	Word, all ASCII
2	word	Word, all letters
3	numword	Word, letters and digits
4	email	Email address
5	url	URL
6	host	Host
7	sfloat	Scientific notation
8	version	Version number

9		hword_numpart		Hyphenated word part, letters and digits
10		hword_part		Hyphenated word part, all letters
11		hword_asciipart		Hyphenated word part, all ASCII
12		blank		Space symbols
13		tag		XML tag
14		protocol		Protocol head
15		numhword		Hyphenated word, letters and digits
16		asciihword		Hyphenated word, all ASCII
17		hword		Hyphenated word, all letters
18		url_path		URL path
19		file		File or path name
20		float		Decimal notation
21		int		Signed integer
22		uint		Unsigned integer
23		entity		XML entity

12.8.3. 辞書のテスト

`ts_lexize`関数は辞書のテストを支援します。

```
ts_lexize(dict regdictionary, token text) returns text[]
```

`ts_lexize`は、入力`token`が辞書に認識されれば語彙素の配列を返します。辞書に認識され、それがストップワードである場合には空の配列を返します。認識されなければ`NULL`を返します。

例:

```
SELECT ts_lexize('english_stem', 'stars');
ts_lexize
-----
{star}

SELECT ts_lexize('english_stem', 'a');
ts_lexize
-----
{}
```

注意: `ts_lexize`関数には、テキストではなく単一のトークンを与えます。これを間違えると次のようになります。

```
SELECT ts_lexize('thesaurus_astro','supernovae stars') is null;
?column?
-----
t
```

類語辞書`thesaurus_astro`は語句`supernovae stars`を認識しますが、`ts_lexize`はしません。なぜなら、入力をテキストではなく、単一のトークンとして扱うからです。類語辞書をテストするには、`plainto_tsquery`または`to_tsvector`を使ってください。例を示します。

```
SELECT plainto_tsquery('supernovae stars');
plainto_tsquery
-----
'sn'
```

[前のページ](#)
設定例

[ホーム](#)
[上に戻る](#)

[次のページ](#)
GiSTおよびGINインデックス種類

12.9. GiSTおよびGINインデックス種類

全文検索を高速化するために、2種類のインデックスが使えます。全文検索のためにインデックスが必須だと言うわけではないことを覚えておかなければなりません、日常的に検索される列には、インデックスを使った方が良いでしょう。

```
CREATE INDEX name ON table USING gist(column);
```

GiST (Generalized Search Tree)インデックスを作ります。columnは `tsvector` または `tsquery` 型です。

```
CREATE INDEX name ON table USING gin(column);
```

GIN (Generalized Inverted Index)インデックスを作ります。columnは `tsvector` 型でなければなりません。

この2つのインデックス形式には、性能上の大きな違いがあります。ですので、それらの特長を理解しておくことが重要です。

GiSTインデックスは、非可逆です。つまり、インデックスは間違った結果を返すかも知れないので、間違った結果を排除するために、テーブルの行をチェックすることが必要です。PostgreSQLはこの処理が必要とされた時に自動的に行います。GiSTインデックスが非可逆なのは、インデックス中の各文書が固定長の署名によって表現されているからです。署名は、各々の単語をハッシュして単一なビットにして、これらのビットをnビットの文書署名にORし、nビットの列中のビットにすることで実現されています。2つの単語が同じビット位置を生成すると、間違った一致が起こります。問い合わせ対象のすべての単語が照合すると(それが正しいか間違っているかは別として)、その照合が正しいものかどうかテーブルの行を取得して調べなければなりません。

非可逆性は、間違った照合によるテーブルからの不必要なデータ取得のため、性能を劣化させます。テーブルへのランダムアクセスは遅いので、GiSTインデックスの有用性は制限されています。誤った照合がどの位あるかという可能性はいくつか要因によりますが、とりわけユニークな単語の数に依存します。ですから、辞書を使ってユニークな単語の数を減らすことをお勧めします。

GINインデックスは標準の問い合わせに対しては非可逆ではありませんが、その性能はユニークな単語の数の対数に依存します。(しかしながら、GINインデックスは `tsvector` 値の中の単語(語彙素)のみを保持しており、重み付けラベルは持っていません。したがって、重み付けを伴う問い合わせではテーブルの行を再チェックしなければなりません。)

GiST、GINのどちらのインデックス形式を選ぶにあたっては、以下の性能上の違いを考慮してください。

- GINインデックスの検索はGiSTの約3倍高速です
- GINインデックスの構築はGiSTの約3倍時間がかかります
- GINインデックスに対する更新はGiSTよりも非常に低速ですが、もし高速更新サポートを無効にしている場合は10倍の低速になります(詳細は[項55.3.1](#)を見てください)
- GINインデックスは、GiSTインデックスより2から3倍大きいです

大雑把に言うと、GINインデックスは検索が高速なので、静的なデータにもっとも向いています。動的なデータには、GiSTインデックスの更新が高速です。とりわけ、GiSTインデックスは、動的なデータに非常に向いており、ユニークな単語(語彙素)が100,000未満ならば高速です。一方GINインデックスは100,000以上の語彙素をよりうまく扱うことができますが、更新が遅いです。

GINインデックスの構築時間は[maintenance_work_mem](#)を増やすことによってしばしば改善することができることに注意してください。一方GiSTインデックスの構築時間にはあまりそのパラメータは効きません。

大きなデータをパーティショニングし、GiST、GINインデックスを使うことによってオンラインの更新を伴いながら、非常に高速な検索を実現することができます。パーティショニングは、継承を使ってデータベースレベルで実現できます。あるいは、文書を複数のサーバに分散させ、[dblink](#) モジュールを使って検索結果を集約できます。これは、ランキング関数がローカルな情報しか使わないために可能になります。

12.10. psqlサポート

psqlでテキスト検索設定オブジェクトに関する情報は、コマンドの集まりを使って取得できます。

```
¥dF{d,p,t}[+] [PATTERN]
```

オプションの+はより詳細な情報を出力します。

オプションパラメータのPATTERNはテキスト検索オブジェクトの名前にすることができます。オプションとしてスキーマ修飾することができます。PATTERNが省略されると、すべての可視的なオブジェクトが表示されます。PATTERNは正規表現を与えることができ、さらにスキーマとオブジェクト名に対して別々のパターンを与えることができます。次の例はこれを説明するものです。

```
=> ¥dF *fulltext*
      List of text search configurations
 Schema | Name          | Description
-----+-----+-----
 public | fulltext_cfg |

=> ¥dF *.fulltext*
      List of text search configurations
 Schema | Name          | Description
-----+-----+-----
 fulltext | fulltext_cfg |
 public  | fulltext_cfg |
```

以下のコマンドが利用できます。

```
¥dF[+] [PATTERN]
```

テキスト検索設定を表示します(+追加で詳細表示)。

```
=> ¥dF russian
      List of text search configurations
 Schema | Name          | Description
-----+-----+-----
 pg_catalog | russian | configuration for russian language

=> ¥dF+ russian
Text search configuration "pg_catalog.russian"
Parser: "pg_catalog.default"
      Token          | Dictionaries
-----+-----+-----
 asciihword          | english_stem
 asciiword            | english_stem
 email               | simple
 file                 | simple
 float               | simple
 host                 | simple
 hword                | russian_stem
 hword_asciipart     | english_stem
 hword_numpart       | simple
 hword_part           | russian_stem
 int                  | simple
 numhword             | simple
 numword              | simple
 sfloat               | simple
 uint                 | simple
 url                  | simple
 url_path             | simple
 version              | simple
 word                 | russian_stem
```

```
¥dFd[+] [PATTERN]
```

テキスト検索辞書を表示します(+追加で詳細表示)。

```
=> ¥dFd
      List of text search dictionaries
 Schema | Name          | Description
-----+-----+-----
 pg_catalog | danish_stem | snowball stemmer for danish language
 pg_catalog | dutch_stem  | snowball stemmer for dutch language
 pg_catalog | english_stem | snowball stemmer for english language
 pg_catalog | finnish_stem | snowball stemmer for finnish language
 pg_catalog | french_stem  | snowball stemmer for french language
 pg_catalog | german_stem  | snowball stemmer for german language
 pg_catalog | hungarian_stem | snowball stemmer for hungarian language
 pg_catalog | italian_stem | snowball stemmer for italian language
```



```

pg_catalog | norwegian_stem | snowball stemmer for norwegian language
pg_catalog | portuguese_stem | snowball stemmer for portuguese language
pg_catalog | romanian_stem | snowball stemmer for romanian language
pg_catalog | russian_stem | snowball stemmer for russian language
pg_catalog | simple | simple dictionary: just lower case and check for stopword
pg_catalog | spanish_stem | snowball stemmer for spanish language
pg_catalog | swedish_stem | snowball stemmer for swedish language
pg_catalog | turkish_stem | snowball stemmer for turkish language

```

¥dFp[+] [PATTERN]

テキスト検索パーサを表示します(+追加で詳細表示)。

```

=> ¥dFp
      List of text search parsers
      Schema | Name | Description
      -----+-----+-----
pg_catalog | default | default word parser
=> ¥dFp+
      Text search parser "pg_catalog.default"
      Method | Function | Description
      -----+-----+-----
Start parse | prsd_start |
Get next token | prsd_nexttoken |
End parse | prsd_end |
Get headline | prsd_headline |
Get token types | prsd_lextype |

      Token types for parser "pg_catalog.default"
      Token name | Description
      -----+-----
asciihword | Hyphenated word, all ASCII
asciipart | Word, all ASCII
blank | Space symbols
email | Email address
entity | XML entity
file | File or path name
float | Decimal notation
host | Host
hword | Hyphenated word, all letters
hword_asciipart | Hyphenated word part, all ASCII
hword_numpart | Hyphenated word part, letters and digits
hword_part | Hyphenated word part, all letters
int | Signed integer
numhword | Hyphenated word, letters and digits
numword | Word, letters and digits
protocol | Protocol head
sfloat | Scientific notation
tag | XML tag
uint | Unsigned integer
url | URL
url_path | URL path
version | Version number
word | Word, all letters
(23 rows)

```

¥dFt[+] [PATTERN]

テキスト検索テンプレートを表示します(+追加で詳細表示)。

```

=> ¥dFt
      List of text search templates
      Schema | Name | Description
      -----+-----+-----
pg_catalog | ispell | ispell dictionary
pg_catalog | simple | simple dictionary: just lower case and check for stopword
pg_catalog | snowball | snowball stemmer
pg_catalog | synonym | synonym dictionary: replace word by its synonym
pg_catalog | thesaurus | thesaurus dictionary: phrase by phrase substitution

```

[前のページ](#)

GiSTおよびGINインデックス種類

[ホーム](#)

[上に戻る](#)

[次のページ](#)

制限事項

12.11. 制限事項

PostgreSQL のテキスト検索機能の制限事項は以下です。

- 各々の語彙素の長さは 2K バイト未満でなければなりません
- `tsvector` の長さ (語彙素 + 位置) は 1M バイト未満でなければなりません
- 語彙素の数は 2^{64} 未満でなければなりません
- `tsvector` の位置量は、0 より大きくかつ 16,383 以下でなければなりません
- 語彙素の位置情報は 256 以下でなければなりません
- `tsquery` 中のノードの数 (語彙素 + 演算子) は 32,768 未満でなければなりません

比較対象として述べておくと、PostgreSQL 8.1 のドキュメントは 10,441 のユニークな単語を含み、全部の単語数は 335,420 で、最頻出の単語 "postgresql" は 655 の文書中に 6,127 回出現しました。

別の例です — PostgreSQL メーリングリストのアーカイブは 910,989 のユニークな単語を含み、461,020 のメッセージ中に 57,491,343 の語彙素がありました。

12.12. 8.3以前のテキスト検索からの移行

テキスト検索のために [tsearch2](#) モジュールを使っていたアプリケーションは、組み込みの機能に対して多少の調整が必要になります。

- いくつかの関数は名前が変わり、引数に変更され、すべて `pg_catalog` スキーマに入りました。以前はこれらの関数は `public` か、あるいはほかのシステム用でないスキーマに格納されていました。新しいバージョンの `tsearch2` が、この種の互換性問題を解決するためのレイヤーとして提供されています。
- 古い `tsearch2` 関数と、その他のオブジェクトは、8.3以前のデータベースのダンプをロードする際に、無効にしておかなければなりません。ほとんどのものはどのみちロードできませんが、2, 3ロードされ、悪さをするものがあります。この問題に対処する簡単な方法は、ダンプファイルをリストアする前に、新しい `tsearch2` モジュールをロードしておくことです。そうすれば、古いオブジェクトがロードされるのを防ぐことができます。
- テキスト検索設定をセットアップする方法がまったく変わりました。設定テーブルに手動で行を挿入する代わりに、この章の最初の方で説明した特別なSQLコマンドを使って設定するようになりました。既存の設定を8.3用に自動的に変換する方法は提供されていません。自分でやってください。
- ほとんどの形式の辞書はデータベース外の設定ファイルに依存しています。これらは8.3以前の使い方と概ね互換性がありますが、一部以下のような違いがあります。
 - 設定ファイルは、単一のディレクトリ (`$SHAREDIR/tsearch_data`) に格納しておかなければなりません。そして、以前いろいろな種類の辞書形式について述べたように、その種類に応じて特定の拡張子を持たなければなりません。この制限はあらかじめセキュリティ上の問題を考慮して追加されました。
 - 設定ファイルは、データベースエンコーディングが何であろうと、UTF-8エンコーディングで符号化しておかなければなりません。
 - 同義語辞書設定ファイルでは、ストップワードは `?` でマークしておかなければなりません。

第 13章 同時実行制御

目次

- 13.1. [序文](#)
- 13.2. [トランザクションの分離](#)
 - 13.2.1. [リードコミティド分離レベル](#)
 - 13.2.2. [リピータブルリード分離レベル](#)
 - 13.2.3. [シリアライザブル分離レベル](#)
- 13.3. [明示的ロック](#)
 - 13.3.1. [テーブルレベルロック](#)
 - 13.3.2. [行レベルロック](#)
 - 13.3.3. [デッドロック](#)
 - 13.3.4. [勧告的ロック](#)
- 13.4. [アプリケーションレベルでのデータの一貫性チェック](#)
 - 13.4.1. [シリアライザブルトランザクションを用いた一貫性の強化](#)
 - 13.4.2. [明示的なブロッキングロックを用いた一貫性の強化](#)
- 13.5. [ロックとインデックス](#)

本章では同時に2つ以上のセッションが同じデータにアクセスしようとした場合、PostgreSQLデータベースシステムがどう振舞うかについて説明します。このような状況でデータの整合性を確実に保つ一方、全てのセッションに対して効果的なアクセスを許可するようにすることが目的です。データベースアプリケーションを開発する方は、本章で扱われている内容を熟知していなければなりません。

13.1. 序文

PostgreSQLは、データへの同時アクセスを管理するために高度な開発者向けツール群を提供します。内部的に、データー貫性は多版方式（多版型同時実行制御MVCC）を使用して管理されています。つまり、データベースへの問い合わせ実行の際、各トランザクションは処理の基礎となっているデータの現在の状態にかかわらず、遡ったある時点におけるスナップショット（データベースバージョン）を参照する、というものです。これは、同時に並行している（別の）トランザクションが同じ行を更新することによって引き起こす整合性を欠いたデータの参照からトランザクションを保護し、それぞれのデータベースセッションに対してトランザクションの分離を提供します。MVCCは、マルチユーザ環境で理想的な性能を得るために、伝統的なデータベースシステムで行われるようなロック手法を避けることで、ロックの競合を最小化します。

ロックではなく同時実行制御のMVCCモデルを使用する主な利点は、MVCCでは問い合わせ（読み込み）ロックの獲得と、書き込みロックの獲得が競合しないことです。したがって、読み込みは書き込みを絶対にブロックしませんし、書き込みも読み込みをブロックすることがありません。革新的なシリアライズブルスナップショット分離（SSI）レベルの使用を通した最も厳密なトランザクションの分離レベルを提供する場合にもPostgreSQLはこれの保証を維持します。

全般的に完全なトランザクションの分離を必要とせず、明示的に競合する点を管理することを望むアプリケーションのために、PostgreSQLではテーブルレベルおよび行レベルのロック機能も使用可能です。とはいえ、MVCCを適切に使用すると通常ロックよりも性能が向上します。さらに、アプリケーションが定義した勧告的ロックは単一トランザクションに拘束されないロックの獲得機構を提供します。

13.2. トランザクションの分離

SQLの標準規格では、トランザクションの分離について4つのレベルを定義しています。標準規格で定義されているもののうち最も厳密なものはシリアライザブルです。1セットのシリアライザブルなトランザクションを同時実行した場合には、ある順番でひとつずつそれらを実行した場合と同じ結果となることが保証されるものです。本文で詳しく述べます。他の3レベルは、同時実行しているトランザクション間の相互作用に起因する、各レベルでは発生してはならない現象面に基づき定義されます。標準規格のシリアライザブルの定義では、このレベルではこれらの現象が起こりえないと述べています。(これは驚くことではありません。トランザクションの効果がひとつずつ実行された場合と一貫性を持たなければならないとしたら、相互作用によって発生した現象はどうやっても見つけ出すことはできないでしょう。)

各種レベルにおける禁止される現象を以下に示します。

ダーティリード

同時に実行されている他のトランザクションが書き込んで未だコミットしていないデータを読み込んでしまう。

反復不能読み取り

トランザクションが、以前読み込んだデータを再度読み込み、そのデータが別のトランザクションによって更新されたことを見出す(別のトランザクションが読み込み間隔の間にコミットした)。

ファントムリード

トランザクションが、複数行のある集合を返す検索条件で問い合わせを再実行した時、別のトランザクションがコミットしてしまったために、同じ検索条件で問い合わせを実行しても異なる結果を得てしまう。

4つの分離レベルとその動作を[表13-1](#)に示します。

表 13-1. 標準SQLトランザクション分離レベル

分離レベル	ダーティリード	反復不能読み取り	ファントムリード
リードアンコミットド	可能性あり	可能性あり	可能性あり
リードコミットド	安全	可能性あり	可能性あり
リピータブルリード	安全	安全	可能性あり
シリアライザブル	安全	安全	安全

PostgreSQLでは、4つの標準トランザクション分離レベルを全て要求することができます。しかし、内部的には、リードコミットド、リピータブルリードおよびシリアライザブルに対応する3つの独立した分離レベルのみがあります。リードアンコミットドレベルを選択した時、実際にはリードコミットドになり、リピータブルリードのPostgreSQLでの実装ではファントムリードの可能性はありません。このように実際の分離レベルは選択したレベルより厳密になることがあります。これは標準SQLでも許されています。この4つの分離レベルについては、発生してはならない事象のみが定義され、発生しなければならない事象は定義されていません。PostgreSQLが3つの分離レベルのみを提供している理由は、多版型同時実行制御という仕組みに標準の分離レベルを関連付ける現実的な方法がこれしかなかったことです。利用可能な分離レベルについては後で詳細に説明します。

トランザクションのトランザクション分離レベルを設定するには[SET TRANSACTION](#)コマンドを使用してください。

13.2.1. リードコミットド分離レベル

PostgreSQLではリードコミットドがデフォルトの分離レベルです。トランザクションがこの分離レベルを使用すると、`SELECT`問い合わせ(`FOR UPDATE/SHARE`句を伴わない)はその問い合わせが実行される直前までにコミットされたデータのみを参照し、まだコミットされていないデータや、その問い合わせの実行中に別の同時実行トランザクションがコミットした更新は参照しません。結果として、`SELECT`問い合わせはその問い合わせが実行を開始した時点のデータベースのスナップショットを参照することになります。しかし`SELECT`文は、自分自身のトランザクション内で実行され更新された結果はたとえまだコミットされていなくても参照します。単一のトランザクション内であっても、`SELECT`文を2回連続して発行した場合、最初の`SELECT`文を処理している最中に他のトランザクションが更新をコミットすると、最初とその次に発行した`SELECT`問い合わせは異なるデータを参照してしまうことにも注意してください。

`UPDATE`、`DELETE`、`SELECT FOR UPDATE`、および`SELECT FOR SHARE`コマンドは対象行を検索する際に`SELECT`コマンドと同じように振舞います。これらのコマンドは、問い合わせが開始された時点で既にコミットされた対象行のみを検出します。しかし、そ

の対象行は、検出されるまでに、同時実行中の他のトランザクションによって、既に更新（もしくは削除あるいはロック）されてしまっているかもしれません。このような場合更新されるべき処理は、最初の更新トランザクションが（それがまだ進行中の場合）コミットもしくはロールバックするのを待ちます。最初の更新処理がロールバックされるとその結果は無視されて、2番目の更新処理で元々検出した行の更新を続行することができます。最初の更新処理がコミットされると、2番目の更新処理では、最初の更新処理により行が削除された場合はその行を無視します。行が削除されなかった時の更新処理は、最初のコミットで更新された行に適用されます。コマンドの検索条件（WHERE 句）は、更新された行がまだその検索条件に一致するかどうかの確認のため再評価されます。検索条件と一致している場合、2番目の更新処理は、更新された行を使用して処理を開始します。SELECT FOR UPDATEおよびSELECT FOR SHAREの場合、ロックされクライアントに返されるのは、更新されるバージョンの行であることを意味します。

このような仕組みにより、更新コマンドが、互いに矛盾したスナップショットを参照する可能性があります。それは更新を試みている同じ行に対する同時実行更新の結果を参照しますが、データベース中の他の行に対する同時実行の問い合わせの結果は参照できません。このような動作をするために複合検索条件を含む問い合わせにリードコミティッドモードを使用することは適切ではありません。しかし、より単純な検索条件の場合、このモードの使用が適しています。例えば、銀行の残高を更新する以下のようなトランザクションを考えてみます。

```
BEGIN;
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 12345;
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 7534;
COMMIT;
```

同時に実行される2つのトランザクションが、口座番号12345の残高を変更しようとした場合、口座12345の行の更新に伴って2番目のトランザクションを開始することは明らかに望まれるところです。各コマンドが事前に決定していた行にのみ処理を行うため、更新されたバージョンの行は問題となる不整合を引き起こしません。

より複雑な用法によりリードコミティッドモードでは好ましくない結果を生成します。例えば、別のコマンドによりその制約条件から追加・削除の両方が行われようとしているデータに作用するDELETEコマンドを考えます。例を挙げると、websiteは2行のテーブルで、そこに9と10の値を持つwebsite.hitsがあります。

```
BEGIN;
UPDATE website SET hits = hits + 1;
-- 別のセッションから
DELETE FROM website WHERE hits = 10;
を執行します
COMMIT;
```

UPDATE前後にwebsite.hits = 10行が存在したとしてもDELETEは効果を生みません。なぜこうなるのかと言うと、事前更新された行の値9は読み飛ばされ、UPDATEが完了し、DELETEがロックを取得した時点では、新規行の値はもはや10ではなく11となり、判定基準にもはや一致しません。

リードコミティッドモードは、その時点までにコミットされた全てのトランザクションを含む新規スナップショットを伴うそれぞれのコマンドで開始するので、同一のトランザクション内でそれに続くコマンドは、いかなる場合でもコミットされた同時実行トランザクションの結果を参照します。上記問題の要点は単一のコマンドがデータベースの厳密に一貫性のある見え方を目撃するか否かです。

リードコミティッドモードで提供されている部分的なトランザクション分離は、多くのアプリケーションでは適切です。またこのモードは高速で、使い方も簡単ですが、全ての場合に対して充分ではありません。複雑な問い合わせや更新を行うアプリケーションは、リードコミティッドモードが提供する以上のより厳正なデータベースの厳密に一貫性のある見え方を必要とします。

13.2.2. リピータブルリード分離レベル

リピータブルリード分離レベルは、トランザクションが開始される前までにコミットされたデータのみを参照します。コミットされていないデータや、そのトランザクションの実行中に別のトランザクションでコミットされた変更を参照しません。（しかし、まだコミットされていないとしても、その問い合わせはそのトランザクション自身内で行われた過去の更新を参照します。）これはSQLの標準規格で求められるものよりもより強く保証するもので、[表13-1](#)で述べている現象をすべて防ぎます。上で述べたように、これは標準規格によって特に許容されており、それぞれの分離レベルが提供しなくてはならない最小の保護が示されています。

このレベルでは、リピータブルリードトランザクションがトランザクション開始時点のスナップショットを見ます。トランザクション内の現在の問い合わせ開始時ではないという点で、リードコミティッドとは異なります。従って、単一トランザクション内の連続するSELECT文は、常に同じデータを参照していることになります。つまり、その自身のトランザクションの後にコミットされた他のトランザクションによる変更を参照しません。

このレベルを使ったアプリケーションでは、直列化の失敗によるトランザクションの再実行に備えておく必要があります。

UPDATE、DELETE、SELECT FOR UPDATE、およびSELECT FOR SHAREコマンドでは、SELECTと同じように対象行を検索します。これらのコマンドでは、トランザクションが開始された時点で既にコミットされている対象行のみを検出します。しかし、その対象行は、検出されるまでに、同時実行中の他のトランザクションによって、既に更新（もしくは削除あるいはロック）されている可能性があります。このような場合、リピータブルリードトランザクションは、最初の更新トランザクションが（それらがまだ

進行中の場合) コミットもしくはロールバックするのを待ちます。最初の更新処理がロールバックされると、その結果は無視され、リピータブルリードトランザクションでは元々検出した行の更新を続行することができます。しかし、最初の更新処理がコミット(かつ、単にロックされるだけでなく、実際に行が更新または削除)されると、リピータブルリードトランザクションでは、以下のようなメッセージを出力してロールバックを行います。

```
ERROR:  could not serialize access due to concurrent update
```

これは、リピータブルリードトランザクションでは、トランザクションが開始された後に別のトランザクションによって更新されたデータは変更またはロックすることができないためです。

アプリケーションがこのエラーメッセージを受け取った場合、現在のトランザクションを中断して、トランザクション全体を始めからやり直されなければなりません。2回目では、トランザクションはコミットされた変更含めてデータベースの最初の状態とみなすので、新しいバージョンの行を新しいトランザクションにおける更新の始点としても、論理的矛盾は起こりません。

更新トランザクションのみ再実行する必要があるかもしれません。読み込み専用トランザクションでは直列化の衝突は決して起こりません。

リピータブルリードモードでは、全てのトランザクションがデータベースの一貫した不変のビューの状態を参照することが保証されます。しかし、このビューは常にいくつかの同じレベルの同時実行トランザクションの並列(一度に一つずつの)実行で一貫性を持つ必要はありません。例えば、このレベルの読み取りのみのトランザクションは、バッチが完了したことを示すために更新された制御レコードを参照することができますが、制御レコードのより以前のバージョンを読み取るため、論理的にそのバッチの一部となる詳細なレコードの1つを参照することはできません。この分離レベルで実行するトランザクションによりビジネスルールを強制しようとすることは、競合するトランザクションをブロックするために注意深く明示的なロックを持たないと、正確に動作しないことが多くあります。

注意: PostgreSQL version 9.1より前まででは、シリアライズブル分離レベルの要求はここで説明した通りの動作をそのまま提供していました。以前のシリアライズブルの動作を維持するためには、リピータブルリードを要求しなければならなくなりました。

13.2.3. シリアライズブル分離レベル

シリアライズブル分離レベルは、最も厳しいトランザクションの分離性を提供します。このレベルではトランザクションが同時にではなく、次から次へと、あたかも順に実行されているように逐次的なトランザクションの実行をエミュレートします。しかし、このレベルを使ったアプリケーションでは、リピータブルリードレベルと同様に、直列化の失敗によるトランザクションの再実行に備えておく必要があります。実際、この分離レベルは、(ある時点で)逐次実行可能なすべてのトランザクションにおいて、シリアライズブルトランザクションの同時実行の組が一貫性のないような振る舞いをしていないか監視することを除き、リピータブルリードと全く同じ動きをします。この監視では、リピータブルリードが示すものを越えてブロックすることはありませんが、監視によりいくらかのオーバーヘッドがあり、直列化異常を引き起こすような状態の検知は、直列化の失敗を引き起こすでしょう。

例えば、以下の初期データを持つ`mytab`というテーブルを考えてみます。

class	value
1	10
1	20
2	100
2	200

ここでシリアライズブルトランザクションAが以下を計算し、

```
SELECT SUM(value) FROM mytab WHERE class = 1;
```

そして、`value`にその結果(30)を、`class = 2`の行として新たに挿入したとします。同時にシリアライズブルトランザクションBが以下を計算し、

```
SELECT SUM(value) FROM mytab WHERE class = 2;
```

その結果300を得、そして、この結果を`class = 1`の新たな行として挿入したとします。その後、両方のトランザクションがコミットを試みます。もし一方の処理がリピータブルリード分離レベルで実行していれば、両方のコミットが許されるでしょう。しかし、結果と一致する順序で実行されたものがないため、シリアライズブルトランザクションを使用した場合は、ひとつのトランザクションがコミットを許され、他方はこのメッセージとともにロールバックされることになります。

```
ERROR:  could not serialize access due to read/write dependencies among transactions
```

この理由は、もしAがBよりも前に実行が完了していた場合、Bの総和は300ではなく330と計算されるためです。同様に別の順序で実行されたとしてもAで計算される総和は異なる結果になります。

真の直列性を保証するためにPostgreSQLでは、最初に走らせた、同時実行トランザクションより前に読み取った結果に対して、書き込みがいつ影響を及ぼしたかを断定可能にするロックを保持することを意味する、述部ロックを使います。

PostgreSQLでは、これらのロックはブロッキングを引き起こさないため、デッドロックの要因とならないものです。それらは、同時実行中のシリアライズトランザクションが、直列化異常につながる組み合わせであることを識別しフラグを立てることに使用されます。それとは対照的に、データの一貫性を保証したいリードコミットドあるいはリピータブルリードトランザクションでは、(そのテーブルを使用しようとしている他のユーザをブロックすることができた)テーブル全体のロックを必要とするかもしれません。あるいは、他のトランザクションをブロックするだけでなくディスク・アクセスを引き起こす `SELECT FOR UPDATE` あるいは `SELECT FOR SHARE` を使用するかもしれません。

PostgreSQLの述部ロックは、他のほとんどのデータベースシステムと同様、トランザクションによって実際にアクセスされたデータを元にしています。これらは、`SIReadLock`のmodeを持つ `pg_locks` システムビューで見ることができます。問い合わせの実行期間中に獲得した特殊なロックは、問い合わせが使用した計画に依存するでしょう。また、ロックを追跡するために使用されるメモリの消費を防ぐために、多数のよりきめの細かいロック(例えばタプル・ロック)はトランザクションの間に、より少数のより粗粒子状のロック(例えばページ・ロック)へ組み合わせられるかもしれません。直列化異常につながるような競合が継続して生じないことを検知すると、`READ ONLY` トランザクションが完了する前に`SIRead`ロックを解除するかもしれません。実際、`READ ONLY` トランザクションは、よく開始時点でその事実を確認し、どんな述部ロックもとらないこともあります。 `SERIALIZABLE READ ONLY DEFERRABLE` トランザクションを明示的に要求した場合には、この事実を確認できるまでブロックします。(これは、シリアライズトランザクションはブロックするけれども、リピータブルリードトランザクションはブロックしない唯一のケースです。) 他方で、`SIRead`ロックは、しばしば読み取りと書き込みが重なっているトランザクションが完了するまで、過去のトランザクションのコミットに保持される必要があります。

シリアライズトランザクションの一貫した使用は開発を単純化することができます。同時実行したシリアライズトランザクションのどんなセットも同じ効果があるという保証は、あたかも、それらが一度に実行されたものだったかのように、単一のトランザクションとして証明できることを意味します。これは、単独で実行された場合に正しく動作するよう書かれている場合、シリアライズトランザクションが混在していても正しく動作する確証を持つことができます。この技術を使用する環境で、直列化の失敗('40001'のSQLSTATEの値で常に返る)を扱う場合、一般的な手段を持っていることは重要です。なぜなら、トランザクションが読み取り/書き込みの依存性にどれだけ影響する可能性があるかということと、直列化異常を防ぐためにロールバックさせる必要があるかといこうとを、正確に予測することは非常に困難だからです。読み取り/書き込みの依存性を監視したり、直列化異常で終了したトランザクションを再起動することはコストがかかります。しかしながら、このコストと、明示的なロックあるいは `SELECT FOR UPDATE` や `SELECT FOR SHARE` を使用したブロッキングとで比較検討すると、シリアライズトランザクションはいくつかの環境において最良な実行を選択することになります。

同時実行制御のためにシリアライズトランザクションを使用する場合、最適に実行するためには、以下の問題を考慮すべきです。

- できる限り `READ ONLY` として宣言してください。
- もし必要ならばコネクションプールを使用して、活動中の接続数を制御してください。これは常に重要な実行時の考慮点ですが、シリアライズトランザクションを使用した多忙なシステムにおいては、特に重要になる可能性があります。
- 完全性を目的とした必要以上の単一トランザクションを置かないようにしてください。
- 必要以上に長く"トランザクション内で待機状態"で接続したまま放置しておかないようにしてください。
- シリアライズトランザクションにより自動的に提供される保護により、不必要な、明示的なロック、`SELECT FOR UPDATE` および `SELECT FOR SHARE` を取り除いてください。
- システムが、単一の関係レベルでの述部ロックに、多数のページレベルでの述部ロックを組み合わせることを強いられる場合、述部ロックのテーブルはメモリが不足するため、直列化失敗の割合が増加する恐れがあります。これは、`max_pred_locks_per_transaction`を増やすことにより回避することができます。
- シーケンシャルスキャンは常にリレーションレベルでの述部ロックを必要とするでしょう。これは直列化失敗の増加した割合に起因する場合があります。`random_page_cost`を縮小および(または)`cpu_tuple_cost`を増加することによりインデックススキャンの使用を促進することは有用かもしれません。問い合わせ実行時間の全体的な変化に不利となる、トランザクションのロールバックや再起動を減少させるように、必ず検討してください。

警告

シリアライズトランザクション分離レベルのサポートは、まだホットスタンバイレプリケーションは対象に(項25.5で述べます)加えられていません。ホットスタンバイモードで現在サポートされた最も厳しい分離レベルはリピータブルリードです。マスタのシリアライズトランザクション中にデータベースへ行なっているすべての永続的な書き込みの間、スタンバイはすべて最終的に一貫した状態に達するだろうということを保証するでしょう。スタンバイ上で実行されたリピータブルリードトランザクションは、時々マスタのシリアライズトランザクションの任意の連続する実行と一致しない過渡状態を見ることができます。

13.3. 明示的ロック

PostgreSQLは、テーブル内のデータに対する同時アクセスを制御するために様々な種類のロックモードを備えています。これらのモードは、MVCCでは必要な動作を得られない場合、アプリケーション制御のロックに使用することができます。また、ほとんどのPostgreSQLコマンドでは、参照されるテーブルがそのコマンドの実行中に別の方法で削除もしくは変更されていないことを確実にするために、適切なモードのロックを自動的に獲得します。（例えば、`TRUNCATE` コマンドは、同じテーブルに対する他の操作とは同時に実行することは危険です。そのため、そのテーブルへの排他ロックを強制的に獲得します。）

現在のデータベースサーバで重要なロックの一覧を確認するには、[pg_locks](#) システムビューを使用してください。ロック管理サブシステムの状況監視についての詳細は[第27章](#)を参照してください。

13.3.1. テーブルレベルロック

以下のリストに、PostgreSQLで自動的に使用される、使用可能なロックモードとその文脈を示します。また、[LOCK](#) コマンドを使用して、こうしたロックを明示的に獲得することもできます。これらのロックモードは、たとえその名前に"row (行)"という言葉が付いていても、全てテーブルレベルのロックであることに注意してください。ロックモードの名前は歴史的なものです。これらの名前は、各ロックモードの代表的な使用方法をある程度表しています。しかし、意味的には全て同じです。ロックモード間における唯一の実質的な差異は、どのモードがどのモードと競合するかというロックモードの組み合わせです([表13-2](#)を参照してください)。2つのトランザクションで、競合するモードのロックを同時に同一テーブル上に保持することはできません（しかし、トランザクションは自分自身とは決して競合しません。例えば、`ACCESS EXCLUSIVE` ロックを獲得し、その後同じテーブルに`ACCESS SHARE` ロックを獲得できる可能性があります）。競合しないロックモードは、多くのトランザクションで同時に保持することが可能です。特に、ロックモードには、自己競合するもの（例えば、`ACCESS EXCLUSIVE` は同時に複数のトランザクションで保持することは不可能）と、自己競合しないもの（例えば、`ACCESS SHARE` は複数のトランザクションで保持可能）があることに注意してください。

テーブルレベルロックモード

`ACCESS SHARE`

`ACCESS EXCLUSIVE` ロックモードとのみ競合します。

`SELECT` コマンドにより、参照されるテーブルに対してこのモードのロックが獲得されます。通常、テーブルの読み取りのみで変更を行わない問い合わせであれば全て、このロックモードを獲得します。

`ROW SHARE`

`EXCLUSIVE` および `ACCESS EXCLUSIVE` ロックモードと競合します。

`SELECT FOR UPDATE` および `SELECT FOR SHARE` コマンドは、（参照はされているが、`FOR UPDATE/FOR SHARE` として選択はされていない他のテーブルに対する`ACCESS SHARE` ロックに加えて）対象となるテーブル上にこのモードのロックを獲得します。

`ROW EXCLUSIVE`

`SHARE`、`SHARE ROW EXCLUSIVE`、`EXCLUSIVE`、および `ACCESS EXCLUSIVE` ロックモードと競合します。

`UPDATE`、`DELETE`、および `INSERT` コマンドは、（参照される他の全てのテーブルに対する`ACCESS SHARE` ロックに加えて）対象となるテーブル上にこのモードのロックを獲得します。通常、このロックモードは、テーブルのデータを変更する問い合わせにより獲得されます。

`SHARE UPDATE EXCLUSIVE`

`SHARE UPDATE EXCLUSIVE`、`SHARE`、`SHARE ROW EXCLUSIVE`、`EXCLUSIVE`、および `ACCESS EXCLUSIVE` ロックモードと競合します。このモードにより、同時実行されるスキーマの変更および`VACUUM` コマンドの実行から、テーブルを保護します。

（`FULL`なしの）`VACUUM` コマンド、`ANALYZE` コマンド、`CREATE INDEX CONCURRENTLY`、および、`ALTER TABLE` のいくつかの形式によって獲得されます。

`SHARE`

`ROW EXCLUSIVE`、`SHARE UPDATE EXCLUSIVE`、`SHARE ROW EXCLUSIVE`、`EXCLUSIVE`、および `ACCESS EXCLUSIVE` ロックモードと競合します。このモードは、同時実行されるデータ変更からテーブルを保護します。

（`CONCURRENTLY`なしの）`CREATE INDEX` によって獲得されます。

`SHARE ROW EXCLUSIVE`

ROW EXCLUSIVE、SHARE UPDATE EXCLUSIVE、SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE、およびACCESS EXCLUSIVEロックモードと競合します。このモードは、1つのセッションだけが一度にそれを保持することができるよう、自己排他的に同時のデータ変更からテーブルを保護します。

このロックモードを自動的に獲得するPostgreSQLコマンドはありません。

EXCLUSIVE

ROW SHARE、ROW EXCLUSIVE、SHARE UPDATE EXCLUSIVE、SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE、およびACCESS EXCLUSIVEロックモードと競合します。このモードは、同時実行されるACCESS SHAREのみを許可します。つまり、このロックモードを保持するトランザクションと並行して実行できる処理は、テーブルの読み取りだけです。

ユーザテーブルに対してこのロックモードを自動的に獲得するPostgreSQLコマンドはありません。

ACCESS EXCLUSIVE

全てのモードのロック（ACCESS SHARE、ROW SHARE、ROW EXCLUSIVE、SHARE UPDATE EXCLUSIVE、SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE、およびACCESS EXCLUSIVE）と競合します。このモードにより、その保持者以外にテーブルにアクセスするトランザクションがないことが保証されます。

ALTER TABLE、DROP TABLE、TRUNCATE、REINDEX、CLUSTER、VACUUM FULLコマンドによって獲得されます。また、明示的にモードを指定しなければ、これがLOCK TABLE文を使用する際のデフォルトのロックモードです。

ティップ: ACCESS EXCLUSIVEロックのみが、SELECT（FOR UPDATE/SHAREなし）文をブロックします。

通常ロックは獲得した後、トランザクションの終わりまで保持されます。しかし、ロックがセーブポイントの確立後に獲得された場合、セーブポイントがロールバックされると、ロックは即座に解放されます。これは、ROLLBACKがセーブポイント以降に行われたすべてのコマンドの効果を取消すという原則と整合性が取れています。PL/pgSQL例外ブロック内で獲得されたロックに対しても同様です。PL/pgSQL例外ブロック内で獲得されたロックに対しても道央です。そのブロックからエラーで抜けた後、獲得されたロックは解放されます。

表 13-2. ロックモードの競合

要求するロックモード	現在のロックモード							
	ACCESS SHARE	ROW SHARE	ROW EXCLUSIVE	SHARE UPDATE EXCLUSIVE	SHARE	SHARE ROW EXCLUSIVE	EXCLUSIVE	ACCESS EXCLUSIVE
ACCESS SHARE								X
ROW SHARE							X	X
ROW EXCLUSIVE					X	X	X	X
SHARE UPDATE EXCLUSIVE				X	X	X	X	X
SHARE			X	X		X	X	X
SHARE ROW EXCLUSIVE			X	X	X	X	X	X
EXCLUSIVE		X	X	X	X	X	X	X
ACCESS EXCLUSIVE	X	X	X	X	X	X	X	X

13.3.2. 行レベルロック

テーブルレベルロックに加えて、排他ロックまたは共有ロックを行うことができる、行レベルロックがあります。特定の行に対する行レベルの排他ロックは、行が更新または削除される時に自動的に獲得されます。トランザクションがコミットまたはロールバックされるまで、単にテーブルレベルロックのように、このロックは保持されます。行レベルロックは、データの問い合わせには影響を与えません。行レベルロックは、同じ行に対する書き込みだけををブロックします。

実際に行を変更せずに行に対して行レベルロックを獲得するには、該当する行をSELECT FOR UPDATEで選択してください。いったん行レベルロックが獲得されると、競合を心配しないで、そのトランザクション内では何回でも行の変更が可能であるということを覚えておいてください。

ある行に対する行レベルの共有ロックを獲得するには、`SELECT FOR SHARE`を使用してその行を選択してください。共有ロックは、他のトランザクションによる同じ共有ロックの獲得を阻害しません。しかし、他のトランザクションが共有ロックを保持している行に対して、更新、削除、排他ロックの獲得を行うことができるトランザクションはありません。これらを試すと、共有ロックが解放されるまでブロックされます。

PostgreSQLでは、メモリ上に変更された行の情報を記憶しないため、同時にロックできる行数の上限はありません。しかし、行をロックする際に、ディスクに書き込む作業が発生するかもしれません。例えば`SELECT FOR UPDATE`は、選択された行をロックしたものと印を付けるために変更を行いますので、ディスクにその結果を書き込むことになります。

テーブルと行ロックに加え、ページレベルの共有/排他ロックがあり、これらは共有バッファプールにあるテーブルページへの読み書きのアクセスを管理するために使用されます。これらのロックは、行が取得された後や更新された後に即座に解除されます。アプリケーション開発者は通常ページレベルロックを考慮する必要はありませんが、ロックについて全てを説明したかったためここで取り上げました。

13.3.3. デッドロック

明示的なロックの使用は、デッドロックの原因となる可能性があります。デッドロックとは、2つ（もしくはそれ以上）のトランザクションにおいて、それぞれが、他方のトランザクションが必要とするロックを所持してしまうことです。例えば、トランザクション1がテーブルAに排他ロックを獲得していて、次にテーブルBに排他ロックを獲得しようとする際に、トランザクション2が既にテーブルBに排他ロックを獲得済みであって、今からテーブルAに排他ロックを獲得しようと試みる場合、どちらのトランザクションも処理を進められません。PostgreSQLでは、自動的にデッドロック状況を検知し、関係するトランザクションの一方をアボートすることにより、この状況を解決し、もう一方のトランザクションの処理を完了させます（どちらのトランザクションをアボートするかを正確に予測するのは難しく、これに依存すべきではありません）。

デッドロックは行レベルロックの結果として発生する可能性があります（したがって、明示的なロック処理を使用していなくても発生する可能性があります）。2つの同時実行トランザクションがあるテーブルを変更する状況を考えてみます。1つ目のトランザクションは以下を実行します。

```
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 11111;
```

これは、指定した口座番号の行に対し行レベルロックを獲得します。次に2番目のトランザクションが以下を実行します。

```
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 22222;  
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 11111;
```

1つ目の`UPDATE`文は指定された行に対する行レベルロックの獲得に成功し、この行の更新に成功します。しかし、2つ目の`UPDATE`文は、更新対象の行がロックされていることを検知し、ロックを獲得したトランザクションが完了するまで待機します。トランザクション2は、ここで、続きを実行する前にトランザクション1が完了するのを待機しています。さて、トランザクション1がここで以下を実行します。

```
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 22222;
```

トランザクション1は指定した行の行レベルロックを獲得しようとはしますが、これは不可能です。トランザクション2がそのロックを既に獲得しているからです。そのため、トランザクション2が完了するのを待機することになります。こうして、トランザクション1はトランザクション2でブロックされ、トランザクション2はトランザクション1でブロックされる、つまり、デッドロック状態です。PostgreSQLはデッドロック状態を検知し、片方のトランザクションを中断させます。

デッドロックを防ぐ最も良い方法は、データベースを使用する全てのアプリケーションが、整合性のある順序で複数のオブジェクトに対するロックを獲得することです。前に示したデッドロックの例で、もし両方のトランザクションで同じ順序で行を更新していたらデッドロックは起こりません。また、トランザクション内のオブジェクトに対して獲得した最初のロックが、そのオブジェクトが必要とする最も制限的なモードであることを確実に保証すべきです。このことが事前に検証できない場合、デッドロックによりアボートするトランザクションを再試行すれば、デッドロックをデータベースを稼働させながらも処理することができます。

デッドロック状況が検出されなければ、テーブルレベルロックもしくは行レベルロックを要求するトランザクションは、競合するロックが解放されるまで、無期限に待機します。したがって、アプリケーションで長時間（例えば、ユーザの入力待ち）トランザクションを開いたまま保持しておくのは、推奨されません。

13.3.4. 勧告的ロック

PostgreSQLは、アプリケーション独自の意味を持つロックを生成する手法を提供します。これは、その使用に関してシステムによる制限がないこと、つまり、正しい使用に関してはアプリケーションが責任を持つことから勧告的ロックと呼ばれます。勧告的ロックは、MVCC方式に合わせづらいロック戦略で有用に使用することができます。例えば、勧告的ロックのよくある利用として、よく"フラットファイル"データ管理システムと呼ばれる、悲観的なロック戦略を模擬することです。この用途のためにテーブル内にフラグを格納することもできますが、勧告的ロックの方が高速で、テーブルの膨張を防ぐことができます。また、セッション終了時にサーバによる自動整理を行うこともできるようになります。

PostgreSQLには、セッションレベルとトランザクションレベルという2つの勧告的ロックの獲得方法があります。セッションレベルで獲得すると、勧告的ロックは明示的に解放されるか、セッションが終了するまで保持されます。標準のロック要求と異なり、セッションレベル勧告的ロックはトランザクションという意味には従いません。ロックがトランザクション期間中に獲得され、そのトランザクションを後でロールバックしたとしても、ロールバック後も保持されます。そして、呼び出し元のトランザクションが後で失敗したとしてもロック解除は有効です。所有するプロセスの中で、同一のセッションレベルのロックを複数回獲得することもできます。この場合、個々のロック要求に対して、ロックを実際に解放する前に対応するロック解除要求がなければなりません。一方トランザクションレベルのロックはより通常のロックに似たように動作します。それらは、処理の終わりに自動的に解放されますので、明示的なロック解放操作はありません。短期間の勧告的ロックを利用する場合は、セッションレベルの動作よりもこの動作の方が便利なが多くあります。同じ勧告的ロック識別子に対するセッションレベルのロックとトランザクションレベルのロック要求は、想像通り互いをブロックします。セッションがすでに指定された勧告的ロックを保持している場合、他のセッションがそのロックを待機していたとしても、追加の要求は常に成功します。これは保持されているロックと新しい要求がセッションレベルかトランザクションレベルかどうかに関わらず、この文は当てはまります。

PostgreSQLにおけるすべてのロックと同様に、現時点ですべてのセッションで保持されている勧告的ロックの全一覧は[pg_locks](#)システムビューにあります。

Both advisory locks and regular locks are stored in a shared memory pool whose size is defined by the configuration variables [max_locks_per_transaction](#) and [max_connections](#). Care must be taken not to exhaust this memory or the server will be unable to grant any locks at all. This imposes an upper limit on the number of advisory locks grantable by the server, typically in the tens to hundreds of thousands depending on how the server is configured. --> 勧告的ロックと通常のロックは共有メモリプールに割り当てられ、その容量は[max_locks_per_transaction](#)と[max_connections](#)設定変数により決定されます。このメモリを浪費しないように注意が必要です。さもないと、サーバはロック獲得をまったく許可することができなくなります。これは、サーバで許可できる勧告的ロック数に上限があることを意味します。サーバの設定によりますが、通常、1万から10万程度になります。

特に明示的な順序付けとLIMIT句を持つ問い合わせでは、この勧告ロックモードを使用する幾つの場合において、SQL式が評価される順序を考慮し獲得されたロックを制御することに気を配らなければなりません。以下に例を示します。

```
SELECT pg_advisory_lock(id) FROM foo WHERE id = 12345; -- 問題なし
SELECT pg_advisory_lock(id) FROM foo WHERE id > 12345 LIMIT 100; -- 危険！
SELECT pg_advisory_lock(q.id) FROM
(
  SELECT id FROM foo WHERE id > 12345 LIMIT 100
) q; -- 問題なし
```

上の例では、ロック獲得関数が実行される前にLIMIT が適用されることを保障できないため、2番目の形式は危険です。これにより、アプリケーションが想定していないなんからのロックが生成される可能性があります。そのため、（セッションが終了するまで）解放に失敗することになります。アプリケーションから見ると、こうしたロックはただの飾りですが、pg_locksからは参照され続けます。

勧告的ロックを扱うための関数については、[項9.26.8](#)で説明します。

[前のページ](#)

トランザクションの分離

[ホーム](#)

[上に戻る](#)

[次のページ](#)

アプリケーションレベルでのデータの一貫性チェック

13.4. アプリケーションレベルでのデータの一貫性チェック

データの参照範囲は各ステートメントで変化するので、リードコミティドトランザクションを使用して、データ保全性に関するビジネスルールを強化するのは非常に難しいことです。また、書き込み競合が生じる場合、単一のステートメントでさえステートメントのスナップショットに限定されないかもしれません。

リピータブルリードトランザクションは実行全体にわたってデータの安定した参照範囲を持ちますが、MVCCスナップショットをデータ完全性チェックに使用することによる、読み取り/書き込み競合として知られるものを含む、微妙な問題があります。1つのトランザクションがデータを書き、同時に実行するトランザクションが、同じデータ(書き込みの前に、あるいはその書き込みの後にも)を読むことを試みる場合、それは別のトランザクションの働きを見ることができません。その後、読み手は、どれが最初にスタートしたか、あるいは、どれが最初にコミットしたかにかかわらず最初に実行したように見えます。そのままいけば問題はありませんが、読み手がさらにデータを書けば、どれが同時に実行したトランザクションがそれを読んだ場合、上で述べたトランザクションのどちらかの前に走ったように見えるトランザクションとなってしまいます。最後に実行したように見えるトランザクションが実際には最初にコミットしていた場合、トランザクションの実行順のグラフには循環が容易に出現します。そのような循環が出現する時、完全性のチェックはなにかしらの支援がなければ正しく動作しません。

[項13.2.3](#)により述べたように、シリアライズابلトランザクションは、危険なパターンの読み取り/書き込み競合のための非ブロッキング監視を加えたリピータブルリードトランザクションです。明白に実行順が循環を引き起こすパターンが検知された場合、含まれていたトランザクションのうちの1つは循環を断ち切るためにロールバックされます。

13.4.1. シリアライズابلトランザクションを用いた一貫性の強化

シリアライズابلトランザクション分離レベルが、データの一貫性を必要とするすべての書き込みおよびすべての読み取りに使用される場合、一貫性を確実にするために必要なことは他にはありません。一貫性を保証するためにシリアライズابلトランザクションを使用するよう書かれている他の環境からのソフトウェアは、PostgreSQLでこの点に関して"正しく動く"べきです。

この技術を使用した場合、アプリケーションソフトウェアが直列化失敗でロールバックしたトランザクションを自動的に再試行するようなフレームワークを備えている場合、アプリケーションプログラマにとって不必要な負担を生み出さないようにするでしょう。default_transaction_isolationをserializableにセットすることはよい考えかもしれません。他のトランザクション分離レベルは使用されないことを保証する処置を講ずる、そうでなければ、不注意に完全位チェックを失わないよう、トリガーでトランザクション分離レベルのチェックをすることも賢明でしょう。

実行に関する提言は[項13.2.3](#)を参照してください。

警告

シリアライズابلトランザクションを使用する整合性保護レベルは、まだホットスタンバイモード([項25.5](#))には拡張されていません。そのために、ホットスタンバイを使用する場合は、マスタにおけるリピータブルリードと明示的なロック処理の利用が望まれるかもしれません。

13.4.2. 明示的なブロッキングロックを用いた一貫性の強化

非シリアライズアブルの書き込みが可能な場合、ある行の現時点の有効性を確実なものとし、同時更新を避けるためには、SELECT FOR UPDATE文やSELECT FOR SHARE文、適切なLOCK TABLE文を使用する必要があります (SELECT FOR UPDATE文およびSELECT FOR SHARE文は返ってきた行のみを同時に起こる更新からロックし、LOCK TABLEはテーブル全体をロックします)。これはPostgreSQLに他の環境からアプリケーションを移植する時に考慮されなければなりません

他の環境から切り替えた場合のさらなる注意点としては、同時実行トランザクションが選択された行を更新しないか削除しないということをSELECT FOR UPDATEが保証しないという事実です。PostgreSQLでそれをするためには、値を変更する必要がなくても、実際に行を更新しなければなりません。SELECT FOR UPDATEは、他のトランザクションが同じロックを獲得すること、または、ロックされた行に影響するUPDATEまたはDELETEを実行することを一時的にブロックします。しかしトランザクションがコミットするかロールバックして一度このロックを獲得すると、ロックが獲得されている間に、行の実際のUPDATEが行なわれなかった場合、ブロックされたトランザクションは、競合した操作を続けることになります。

MVCCにおいては全体的な有効性チェックに特別な考慮を払わなければなりません。例えば銀行のアプリケーションで、1つのテーブルにある全ての貸方の合計が、別のテーブルにある借方の合計と同じであることを、二つのテーブルが常に更新されているときに、チェックする必要があるとします。2つの連続するSELECT SUM(...)コマンドの結果を比べると、2番目の問い合わせは、おそらく最初の問い合わせによってカウントされなかったトランザクションの結果を含んでいるため、リードコミティドモードでは信頼のおける処理を実行できないことがわかります。1つのシリアライズابلトランザクションで2つの合計を出力

すると、シリアライズブルトランザクションが開始される前にコミットされたトランザクション結果のみの正確な状況を得ることができます。しかし、その結果がもたらされた時点でもなお妥当であるかどうかは、実際には疑わしいかもしれません。整合性チェックを行う前にシリアライズブルトランザクション自身の変更を行った場合、そのチェックの有効性はさらに疑わしくなります。これにより、トランザクション開始後に行われる変更の全てだけでなく、何か別のものが含まれるためです。このような場合、注意深い人であれば、現状を確実に把握するためにチェックに必要な全てのテーブルをロックするでしょう。

SHARE モード（もしくはそれ以上）のロックにより、現在のトランザクションでの変更を除き、ロックされたテーブルにコミットされていない変更が存在しないことが保証されます。

同時に、明示的なロック処理を使用して、同時に変更が実行されるのを防ごうとする場合、リードコミティドモードの使用または、リピータブルリードモードを使用する場合は、問い合わせを実行する前にロックを獲得するよう留意してください。もしくは、シリアライズブルモードを使用する場合は、問い合わせを実行する前にロックを獲得するよう留意してください。リピータブルリードトランザクションにおいて獲得されたロックは、テーブルに変更をかける他のトランザクションが現在実行されていないことを保証します。しかし、トランザクションが参照しているスナップショットが、ロックの獲得より前に取得されたものであれば、そのスナップショットは現時点においてコミットされている変更より前のテーブルのものである可能性があります。リピータブルリードトランザクションのスナップショットは、実際にはその最初の問い合わせもしくはデータ変更コマンド（SELECT、INSERT、UPDATE、またはDELETE）が開始された時点で取得されます。したがって、スナップショットを取得する前に、明示的にロックを獲得することが可能です。

[前のページ](#)

明示的ロック

[ホーム](#)

[上に戻る](#)

[次のページ](#)

ロックとインデックス

13.5. ロックとインデックス

PostgreSQLは、テーブルデータへのノンブロック読み込み/書き込みアクセスを備えています。しかし現在、この機能はPostgreSQLで実装されている全てのインデックスアクセスメソッドに対して実装されているわけではありません。 各種のインデックスでは下記のように扱われます。

B-treeインデックス、GiSTおよびSP-GiSTインデックス

読み込み/書き込みアクセスに短期の共有/排他モードのページレベルロックを使います。 ロックは、インデックス行が挿入または取り出されるとただちに解放されます。 これらのインデックス種類は、デッドロック状態になることなく、最も高い同時実行性を提供します。

ハッシュインデックス

読み込み/書き込みアクセスに共有/排他モードのハッシュバケットレベルロックを使います。 ロックは、バケット全体が処理された後に解放されます。 バケットレベルロックは、インデックスレベルのロックよりも同時実行性に優れていますが、1つのインデックス操作よりも長くロックが保持されますので、デッドロックに陥りやすくなります。

GINインデックス

読み込み/書き込みアクセスに短期の共有/排他モードのページレベルロックを使います。 ロックは、インデックス行が挿入または取り出されるとただちに解放されます。 しかし、GINによりインデックス付けされた値の挿入は、通常1行当たり複数のインデックスキーの挿入をもたらすことに注意してください。 そのため、GINは単一の値を挿入する時に更に多くの作業を行います。

現時点では、B-treeインデックスは同時実行アプリケーションにおいて最善の性能を提供します。 これはまた、ハッシュインデックスよりも多くの機能を持つため、スカラデータのインデックスが必要な同時実行アプリケーションで推奨するインデックス型です。 非スカラデータを扱う場合、B-treeを使用することができないことは明確です。 この場合は代わりにGiST、SP-GiSTもしくはGINインデックスを使用すべきです。

第 14 章 性能に関するヒント

目次

14.1. [EXPLAIN の利用](#)

14.1.1. [EXPLAIN の基本](#)

14.1.2. [EXPLAIN ANALYZE](#)

14.1.3. [警告](#)

14.2. [プランナで使用される統計情報](#)

14.3. [明示的な JOIN 句でプランナを制御する](#)

14.4. [データベースへのデータ投入](#)

14.4.1. [自動コミットをオフにする](#)

14.4.2. [COPY の使用](#)

14.4.3. [インデックスを削除する](#)

14.4.4. [外部キー制約の削除](#)

14.4.5. [maintenance_work_mem を増やす](#)

14.4.6. [checkpoint_segments を増やす](#)

14.4.7. [WAL アーカイブ処理とストリーミングレプリケーションの無効化](#)

14.4.8. [最後に ANALYZE を実行](#)

14.4.9. [pg_dump に関するいくつかの注意](#)

14.5. [永続性がない設定](#)

問い合わせの性能は多くの要因に影響されます。ユーザが制御できるものもありますが、背後にあるシステム設計に起因する根本的な要因もあります。本章では PostgreSQL の性能を理解し、チューニングするためのヒントを提供します。

14.1. EXPLAINの利用

PostgreSQLは受理した問い合わせから問い合わせ計画を作り出します。問い合わせの構造と含まれるデータの性質に適した正しい問い合わせ計画を選択することが、良い性能を得るために非常に重要になります。ですので、システムには優れた計画の選択を試みる複雑なプランナが存在します。[EXPLAIN](#) コマンドを使えば、任意の問い合わせに対してプランナがどのような問い合わせ計画を作ったのかわかります。問い合わせ計画を読みこなすためには相当な経験が必要な腕が必要です。本節ではその基本を提供しようと考えます。

本節の例は、9.2の開発版ソースを用いて `VACUUM ANALYZE` を実行した後でリグレーション試験データベースから取り出したものです。実際にこの例を試すと、似たような結果になるはずですが、おそらく推定コストや行数は多少異なることになるでしょう。`ANALYZE` による統計情報は厳密なものではなくランダムなサンプリングを行った結果であり、また、コストは本質的にプラットフォームに何かしら依存するためです。

例では、簡潔で人が読みやすい `EXPLAIN` のデフォルトの "text" 出力書式を使用します。今後の解析で `EXPLAIN` の出力をプログラムに渡すことを考えているのであれば、代わりに機械読み取りが容易な出力書式 (XML、JSON、YAML) のいずれかを使用すべきです。

14.1.1. EXPLAINの基本

問い合わせ計画は計画ノードのツリー構造です。ツリー構造の最下層ノードはスキャンノードで、テーブルから行そのものを返します。シーケンシャルスキャン、インデックススキャン、ビットマップインデックススキャンといったテーブルアクセス方法の違いに応じ、スキャンノードの種類に違いがあります。また、`VALUES` 句や `FROM` 内の集合を返す関数など独自のスキャンノード種類を持つ、テーブル行を元にしないものがあります。問い合わせが結合、集約、ソートなど、行そのものに対する操作を必要としている場合、スキャンノードの上位に更に、これらの操作を行うためのノードが追加されます。これらの操作の実現方法にも通常複数の方法がありますので、異なった種類のノードがここに出現することもあり得ます。`EXPLAIN` には計画ツリー内の各ノードにつき1行の出力があり、基本ノード種類とプランナが生成したその計画ノードの実行に要するコスト推定値を示します。さらに、ノードの追加属性を表示するためにノードの要約行からインデント付けされた行が出力される可能性があります。最初の1行目 (もっとも上位ノード) には、計画全体の実行コスト推定値が含まれます。プランナはこの値が最小になるように動作します。

どのような出力となるのかを示すためだけに、ここで簡単な例を示します。

```
EXPLAIN SELECT * FROM tenk1;

               QUERY PLAN
-----
Seq Scan on tenk1  (cost=0.00..458.00 rows=10000 width=244)
```

この問い合わせには `WHERE` 句がありませんので、テーブル行をすべてスキャンしなければなりません。このためプランナは単純なシーケンシャルスキャン計画を使用することを選びました。(左から右に) 括弧で囲まれた数値には以下のものがあります。

- 初期処理の推定コスト。出力段階が開始できるようになる前に消費される時間、例えば、`SORT` ノードで実行されるソート処理の時間です。
- 全体推定コスト。これは計画ノードが実行完了である、つまりすべての利用可能な行を受け取ることを前提として示されます。実際には、ノードの親ノードはすべての利用可能な行を読む前に停止する可能性があります (以下の `LIMIT` の例を参照)。
- この計画ノードが出力する行の推定数。ここでも、ノードが実行を完了することを前提としています。
- この計画ノードが出力する行の (バイト単位での) 推定平均幅。

コストはプランナのコストパラメータ ([項18.7.2](#) 参照) によって決まる任意の単位で測定されます。取り出すディスクページ単位でコストを測定することが、伝統的な方式です。つまり、[seq_page_cost](#) を慣習的に 1.0 に設定し、他のコストパラメータを相対的に設定します。本節の例では、デフォルトのコストパラメータで実行しています。

上位ノードのコストには、すべての子ノードのコストもその中に含まれていることを理解することは重要です。このコストはプランナが関与するコストのみ反映する点もまた重要です。とりわけ、結果の行をクライアントに転送するコストは、実際の処理時間の重要な要因となる可能性があるにもかかわらず、考慮されません。プランナは、計画をいかに変更しようと、どうすることもできないため、これを無視します。(正しい計画はどんなものであれ、すべて同じ行を結果として出力すると信じています。)

`rows` の値は、計画ノードによって処理あるいはスキャンされた行数を表していず、ノードによって発行された行数を表すので、多少扱いにくくなっています。該当ノードに適用されるすべての `WHERE` 句条件によるフィルタ処理の結果、スキャンされる行よ

り少ない行数になることがよくあります。理想的には、最上位の行数の推定値は、実際に問い合わせによって返され、更新され、あるいは削除された概算の行数となります。

例に戻ります。

```
EXPLAIN SELECT * FROM tenk1;

               QUERY PLAN
-----
Seq Scan on tenk1  (cost=0.00..458.00 rows=10000 width=244)
```

この数は非常に素直に導かれます。

```
SELECT relpages, reltuples FROM pg_class WHERE relname = 'tenk1';
```

を実行すると、tenk1には358のディスクページと10000の行があることがわかります。推定コストは（ディスクページ読み取り × [seq_page_cost](#)） + （スキャンした行 × [cpu_tuple_cost](#)）と計算されます。デフォルトでは、seq_page_costは1.0、cpu_tuple_costは0.01です。ですから、推定コストは(358 * 1.0) + (10000 * 0.01) = 458となります。

では、WHERE条件を加えて、問い合わせを変更してみます。

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 7000;

               QUERY PLAN
-----
Seq Scan on tenk1  (cost=0.00..483.00 rows=7001 width=244)
  Filter: (unique1 < 7000)
```

EXPLAINの出力が適用されるWHERE句が、Scan計画ノードに付随する"フィルタ"条件として表示していることに注意してください。これは、この計画ノードがスキャンした各行に対してその条件を検査することを意味し、その条件を通過したもののみが出力されます。WHERE句があるため、推定出力行数が小さくなっています。しかし、依然として10000行すべてをスキャンする必要があるため、コストは小さくなっていません。実際には、WHERE条件を検査するためにCPU時間が余計にかかることを反映して、ほんの少し（正確には10000 × [cpu_operator_cost](#)）ですがコストが上昇しています。

この問い合わせが選択する実際の行数は7000です。しかし、rowsの推定行数は概算値に過ぎません。この実験を2回実行した場合、おそらく多少異なる推定値を得るでしょう。もっと言うと、これはANALYZEコマンドを行う度に変わることがあり得ます。なぜなら、ANALYZEで生成される統計情報は、テーブルのランダムな標本から取り出されるからです。

では、条件をより強く制限してみます。

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100;

               QUERY PLAN
-----
Bitmap Heap Scan on tenk1  (cost=5.03..229.17 rows=101 width=244)
  Recheck Cond: (unique1 < 100)
-> Bitmap Index Scan on tenk1 unique1  (cost=0.00..5.01 rows=101 width=0)
    Index Cond: (unique1 < 100)
```

ここでは、プランナは2段階の計画を使用することを決定しました。子の計画ノードは、インデックスを使用して、インデックス条件に合う行の場所を検索します。そして、上位計画ノードが実際にテーブル自体からこれらの行を取り出します。行を別々に取り出すことは、シーケンシャルな読み取りに比べ非常に高価です。しかし、テーブルのすべてのページを読み取る必要はありませんので、シーケンシャルスキャンより低価になります。（2段階の計画を使用する理由は、別々に行を取り出すコストを最小にするために、上位の計画ノードがインデックスにより識別された行の位置を読み取る前に物理的な順序でソートすることです。ノードで記載されている"bitmap"は、ソートを行う機構の名前です。）

ここでWHERE句に別の条件を付与してみましょう。

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND stringu1 = 'xxx';

               QUERY PLAN
-----
Bitmap Heap Scan on tenk1  (cost=5.01..229.40 rows=1 width=244)
  Recheck Cond: (unique1 < 100)
  Filter: (stringu1 = 'xxx'::name)
-> Bitmap Index Scan on tenk1 unique1  (cost=0.00..5.01 rows=101 width=0)
    Index Cond: (unique1 < 100)
```

追加されたstringu1 = 'xxx'条件は出力行数推定値を減らしますが、同じ行集合にアクセスしなければなりませんので、コストは減りません。このインデックスがunique1列に対してのみ存在するため、stringu1句をインデックス条件として適用できないことに注意してください。代わりに、インデックスによって取り出される行に対するフィルタとして適用されます。これにより、追加の検査分を反映するため、コストは実際には少し上がります。

場合によってはプランナは"単純な"インデックススキャン計画を選択します。

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 = 42;
```

QUERY PLAN

```
-----
Index Scan using tenk1_unique1 on tenk1  (cost=0.00..8.27 rows=1 width=244)
  Index Cond: (unique1 = 42)
```

この種の計画では、テーブル行はインデックス順で取り出されます。 このため読み取りがより高価になりますが、これが行の位置のソートにかかる追加コストが高価になることはほとんどありません。 単一の行のみを取り出す問い合わせでは、この計画種類がよく現れます。 また、`ORDER BY`を満たすために必要となる余分な必要なソート処理がないため、インデックスの順序に一致する`ORDER BY`条件を持つ問い合わせでよく使用されます。

`WHERE`句で参照される複数の列に対するインデックスが存在する場合、プランナはインデックスをANDやORで組み合わせて使用することを選択する可能性があります。

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000;
```

QUERY PLAN

```
-----
Bitmap Heap Scan on tenk1  (cost=25.01..60.14 rows=10 width=244)
  Recheck Cond: ((unique1 < 100) AND (unique2 > 9000))
  -> BitmapAnd (cost=25.01..25.01 rows=10 width=0)
        -> Bitmap Index Scan on tenk1_unique1 (cost=0.00..5.01 rows=101 width=0)
              Index Cond: (unique1 < 100)
        -> Bitmap Index Scan on tenk1_unique2 (cost=0.00..19.74 rows=999 width=0)
              Index Cond: (unique2 > 9000)
```

しかし、これは両方のインデックスを参照する必要があります。 そのため、インデックスを1つ使用し、他の条件についてはフィルタとして扱う方法と比べて常に勝るとは限りません。 含まれる範囲を変更すると、それに伴い計画も変わることが分かります。

以下に`LIMIT`の影響を示す例を示します。

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000 LIMIT 2;
```

QUERY PLAN

```
-----
Limit  (cost=0.00..14.25 rows=2 width=244)
  -> Index Scan using tenk1_unique2 on tenk1  (cost=0.00..71.23 rows=10 width=244)
        Index Cond: (unique2 > 9000)
        Filter: (unique1 < 100)
```

これは上と同じ問い合わせですが、すべての行を取り出す必要がないため`LIMIT`を付けています。 プランナはどうすべきかについて考えを変えました。 インデックススキャンノードの総コストと総行数があたかも実行完了したかのように表示されていることに注意してください。 しかし`Limit`ノードが、これらの内5行のみを取り出した後で停止することが想定されています。 そのため総コストは1/5程度のみとなり、これが問い合わせの実際の推定コストとなります。 この計画は、以前の計画に`Limit`ノードを追加することより好まれます。 `Limit`はビットマップスキャンの起動コストを払うことを避けることができないためです。 このため総コストはこの方法に25単位ほど増加します。

今まで説明に使ってきたフィールドを使って2つのテーブルを結合してみましょう。

```
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t1.unique2 = t2.unique2;
```

QUERY PLAN

```
-----
Nested Loop  (cost=4.33..118.25 rows=10 width=488)
  -> Bitmap Heap Scan on tenk1 t1  (cost=4.33..39.44 rows=10 width=244)
        Recheck Cond: (unique1 < 10)
        -> Bitmap Index Scan on tenk1_unique1 (cost=0.00..4.33 rows=10 width=0)
              Index Cond: (unique1 < 10)
  -> Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.00..7.87 rows=1 width=244)
        Index Cond: (unique2 = t1.unique2)
```

この計画では、入力または子として2つのテーブルスキャンを持つ入れ子状ループ結合ノードがあります。 計画のツリー構造を反映して、ノード要約行はインデント付けされます。 結合の先頭、"外部"、子は以前に説明したものと似たビットマップスキャンです。 そのコストと行数は、該当ノードに`unique1 < 10 WHERE`句が適用されるため、`SELECT ... WHERE unique1 < 10`で得られたものと同じです。 この段階では`t1.unique2 = t2.unique2`句は関係しておらず、外部スキャンにおける出力行数に影響していません。 入れ子状ループ結合ノードは、外部の子から得られた行毎に、その2番目または"内部の"子を一回実行します。 現在の外部の行からの列の値は内部スキャンに組み込まれます。 ここで、外部行からの`t1.unique2`の値が利用できますので、上述の単純な`SELECT ... WHERE t2.unique2 = constant`の場合に示したものと似た計画とコストが得られます。（実際、推定コストは、`t2`に対するインデックススキャンが繰り返される間に発生することが想定されるキャッシュの結果、上で示した値よりわずかに低くなります。） ループノードのコストは、外部スキャンのコストと、各々の外部の行に対して内部スキャンが繰り返されることによるコスト（ここでは $10 * 7.87$ ）を加え、さらに結合処理を行うための少々のCPU時間を加えたものになります。

この例では、結合の出力行数は2つのスキャンの出力行数の積に等しくなっていますが、いつもそうなるわけではありません。

2つのテーブルに関係するWHERE句は、入力スキャン時ではなく、結合を行う際に適用されるからです。例として、もう1つ条件を加えてみます。

```
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t1.unique2 = t2.unique2 AND t1.hundred < t2.hundred;
```

QUERY PLAN

```
Nested Loop (cost=4.33..118.28 rows=3 width=488)
  Join Filter: (t1.hundred < t2.hundred)
  -> Bitmap Heap Scan on tenk1 t1 (cost=4.33..39.44 rows=10 width=244)
      Recheck Cond: (unique1 < 10)
      -> Bitmap Index Scan on tenk1_unique1 (cost=0.00..4.33 rows=10 width=0)
          Index Cond: (unique1 < 10)
  -> Index Scan using tenk2_unique2 on tenk2 t2 (cost=0.00..7.87 rows=1 width=244)
      Index Cond: (unique2 = t1.unique2)
```

t1.hundred < t2.hundred追加条件はtenk2_unique2インデックスの中では試験されません。このため結合ノードで適用されます。これは結合ノードの推定出力行数を減らしはしますが、入力スキャンには影響しません。

外部結合を扱う時、"結合フィルタ"および通常の"フィルタ"の両方が付随する結合計画ノードが現れる可能性があります。結合フィルタ条件は外部結合のON句を元にしますので、結合フィルタ条件に合わない行がNULLで展開された行として発行され続けます。しかし通常のフィルタ条件が外部結合規則の後に適用され、条件に合わない行は削除されます。内部結合では、これらのフィルタ種類の間に意味的な違いはありません。

問い合わせの選択性を少し変更すると、非常に異なる結合計画が得られるかもしれません。

```
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;
```

QUERY PLAN

```
Hash Join (cost=230.43..713.94 rows=101 width=488)
  Hash Cond: (t2.unique2 = t1.unique2)
  -> Seq Scan on tenk2 t2 (cost=0.00..445.00 rows=10000 width=244)
  -> Hash (cost=229.17..229.17 rows=101 width=244)
      -> Bitmap Heap Scan on tenk1 t1 (cost=5.03..229.17 rows=101 width=244)
          Recheck Cond: (unique1 < 100)
          -> Bitmap Index Scan on tenk1_unique1 (cost=0.00..5.01 rows=101 width=0)
              Index Cond: (unique1 < 100)
```

ここでプランナはハッシュ結合の使用を選択しました。片方のテーブルの行がメモリ内のハッシュテーブルに格納され、もう片方のテーブルがスキャンされた後、各行に対して一致するかどうかハッシュテーブルを探索します。繰り返しますが、インデント付けにより計画の構造が表されます。tenk1に対するビットマップスキャンはハッシュノードへの入力です。外部の子計画から行を読み取り、各行に対してハッシュテーブルを検索します。

他にも、以下に示すようなマージ結合という結合があり得ます。

```
EXPLAIN SELECT *
FROM tenk1 t1, onek t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;
```

QUERY PLAN

```
Merge Join (cost=197.83..267.93 rows=10 width=488)
  Merge Cond: (t1.unique2 = t2.unique2)
  -> Index Scan using tenk1_unique2 on tenk1 t1 (cost=0.00..656.25 rows=101 width=244)
      Filter: (unique1 < 100)
  -> Sort (cost=197.83..200.33 rows=1000 width=244)
      Sort Key: t2.unique2
      -> Seq Scan on onek t2 (cost=0.00..148.00 rows=1000 width=244)
```

マージ結合は、結合キーでソートされる入力データを必要とします。この計画では、正確な順序で行をアクセスするためにtenk1データがインデックススキャンを用いてソートされます。しかし、このテーブルの中でより多くの行がアクセスされるため、onekではシーケンシャルスキャンとソートが好まれています。（多くの行をソートする場合、インデックススキャンでは非シーケンシャルなディスクアクセスが必要となるため、シーケンシャルスキャンとソートの方がインデックススキャンより優れています。）

[項18.7.1](#)に記載したenable/disableフラグを使用して、プランナが最も良いと考えている戦略を強制的に無視させる方法により、異なった計画を観察することができます。（非常に原始的なツールですが、利用価値があります。[項14.3](#)も参照してください。）例えば、前の例にてonekテーブルを扱う最善の方法がシーケンシャルスキャンとソートであると納得できなければ、以下を試みることができます。

```
SET enable_sort = off;
```

```
EXPLAIN SELECT *
```

```
FROM tenk1 t1, onek t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;
```

QUERY PLAN

```
-----
Merge Join  (cost=0.00..292.36 rows=10 width=488)
  Merge Cond: (t1.unique2 = t2.unique2)
    -> Index Scan using tenk1_unique2 on tenk1 t1  (cost=0.00..656.25 rows=101 width=244)
        Filter: (unique1 < 100)
    -> Index Scan using onek_unique2 on onek t2  (cost=0.00..224.76 rows=1000 width=244)
```

これは、プランナが、シーケンシャルスキャンとソートよりインデックススキャンによるonekのソート処理がおよそ12%程高価であるとみなしたことを示します。 当然ながら、次の疑問はこれが正しいかどうかでしょう。 後で説明するEXPLAIN ANALYZEを使用することで調査することができます。

14.1.2. EXPLAIN ANALYZE

EXPLAINのANALYZEオプションを使用して、プランナが推定するコストの精度を点検することができます。 このオプションを付けるとEXPLAIN実際にその問い合わせを実行し、計画ノードごとに実際の行数と要した実際の実行時間を、普通のEXPLAINが示すものと同じ推定値と一緒に表示します。 例えば、以下のような結果を得ることができます。

```
EXPLAIN ANALYZE SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t1.unique2 = t2.unique2;
```

QUERY PLAN

```
-----
Nested Loop  (cost=4.33..118.25 rows=10 width=488) (actual time=0.370..1.126 rows=10 loops=1)
  -> Bitmap Heap Scan on tenk1 t1  (cost=4.33..39.44 rows=10 width=244) (actual time=0.254..0.380 rows=10 loops=1)
      Recheck Cond: (unique1 < 10)
      -> Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.33 rows=10 width=0) (actual time=0.164..0.164 rows=10 loops=1)
          Index Cond: (unique1 < 10)
      -> Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.00..7.87 rows=1 width=244) (actual time=0.041..0.048 rows=1 loops=10)
          Index Cond: (unique2 = t1.unique2)
  Total runtime: 2.414 ms
```

"actual time"値は実時間をミリ秒単位で表されていること、cost推定値は何らかの単位で表されていることに注意してください。 ですからそのまま比較することはできません。 注目すべきもっとも重要な点は通常、推定行数が実際の値と合理的に近いかどうかです。 この例では、推定はすべて正確ですが、現実的にはあまりありません。

問い合わせ計画の中には、何回も副計画ノードを実行できるものがあります。 例えば、上述の入れ子状ループの計画では、内部インデックススキャンは外部の行ごとに一度行われます。 このような場合、loops値はそのノードを実行する総回数を報告し、表示される実際の時間と行数は1実行当たりの平均です。 これで値を表示された推定コストと比較できるようになります。 loops値をかけることで、そのノードで実際に費やされた総時間を得ることができます。 上の例では、tenk2に対するインデックススキャンの実行のために合計0.480ミリ秒要しています。

場合によっては、EXPLAIN ANALYZEは計画ノードの実行時間と行数以上の実行統計情報をさらに表示します。 例えば、ソートとハッシュノードでは以下のような追加情報を提供します。

```
EXPLAIN ANALYZE SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2 ORDER BY t1.fivethous;
```

QUERY PLAN

```
-----
Sort  (cost=717.30..717.56 rows=101 width=488) (actual time=104.950..105.327 rows=100 loops=1)
  Sort Key: t1.fivethous
  Sort Method: quicksort  Memory: 68kB
  -> Hash Join  (cost=230.43..713.94 rows=101 width=488) (actual time=3.680..102.396 rows=100 loops=1)
      Hash Cond: (t2.unique2 = t1.unique2)
      -> Seq Scan on tenk2 t2  (cost=0.00..445.00 rows=10000 width=244) (actual time=0.046..46.219 rows=10000 loops=1)
      -> Hash  (cost=229.17..229.17 rows=101 width=244) (actual time=3.184..3.184 rows=100 loops=1)
          Buckets: 1024  Batches: 1  Memory Usage: 27kB
          -> Bitmap Heap Scan on tenk1 t1  (cost=5.03..229.17 rows=101 width=244) (actual time=0.612..1.959 rows=100 loops=1)
              Recheck Cond: (unique1 < 100)
              -> Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.01 rows=101 width=0) (actual time=0.390..0.390 rows=100 loops=1)
                  Index Cond: (unique1 < 100)
  Total runtime: 107.392 ms
```

ソートノードは使用されるソート方式（具体的にはソートがメモリ内かディスク上か）および必要なメモリまたはディスクの容量を表示します。 ハッシュノードでは、ハッシュバケット数とバッチ数、ハッシュテーブルで使用されるメモリのピーク容量が

表示されます。（バッチ数が1を超える場合、同時にディスクの使用容量も含まれますが、表示はされません。）

他の種類の追加情報はフィルタ条件によって除外される行数があります。

```
EXPLAIN ANALYZE SELECT * FROM tenk1 WHERE ten < 7;

                                QUERY PLAN
-----
--
Seq Scan on tenk1  (cost=0.00..483.00 rows=7000 width=244) (actual time=0.111..59.249 rows=7000 loops=1)
  Filter: (ten < 7)
  Rows Removed by Filter: 3000
  Total runtime: 85.340 ms
```

特に結合ノードで適用されるフィルタ条件ではこれらの数が有用です。 "Rows Removed"行は、少なくともスキャンされた1行、結合ノードにおける結合組み合わせの可能性がフィルタ条件によって拒絶された時にのみ現れます。

"非可逆"インデックススキャンはフィルタ条件に似た状況です。 例えば、特定の点を含有する多角形の検索を考えてみます。

```
EXPLAIN ANALYZE SELECT * FROM polygon_tbl WHERE f1 @> polygon '(0.5,2.0)';

                                QUERY PLAN
-----
Seq Scan on polygon_tbl  (cost=0.00..1.05 rows=1 width=32) (actual time=0.251..0.251 rows=0 loops=1)
  Filter: (f1 @> '((0.5,2))'::polygon)
  Rows Removed by Filter: 4
  Total runtime: 0.517 ms
```

プランナは（ほぼ正確に）、インデックススキャンを考慮するには例のテーブルが小さ過ぎるとみなします。 このため、フィルタ条件によってすべての行が拒絶される、普通のシーケンシャルスキャンとなります。 しかしインデックススキャンの使用を強制するのであれば、以下のようになります。

```
SET enable_seqscan TO off;

EXPLAIN ANALYZE SELECT * FROM polygon_tbl WHERE f1 @> polygon '(0.5,2.0)';

                                QUERY PLAN
-----
Index Scan using gpolygonind on polygon_tbl  (cost=0.00..8.27 rows=1 width=32) (actual time=0.293..0.293 rows=0 loops=1)
  Index Cond: (f1 @> '((0.5,2))'::polygon)
  Rows Removed by Index Recheck: 1
  Total runtime: 1.054 ms
```

ここで、インデックスが1つの候補行を返し、それがインデックス条件の再検査により拒絶されることが分かります。 多角形の含有試験ではGiSTインデックスが"非可逆"であるため、これは発生します。 実際には対象と重なる多角形を持つ行を返し、そしてこれらの行が正確に含有関係であることを試験しなければなりません。

EXPLAINには、より多くの実行時統計情報を取り出すために、ANALYZEに付与できるBUFFERSオプションがあります。

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000;

                                QUERY PLAN
-----
Bitmap Heap Scan on tenk1  (cost=25.07..60.23 rows=10 width=244) (actual time=3.069..3.213 rows=10 loops=1)
  Recheck Cond: ((unique1 < 100) AND (unique2 > 9000))
  Buffers: shared hit=16
-> BitmapAnd (cost=25.07..25.07 rows=10 width=0) (actual time=2.967..2.967 rows=0 loops=1)
   Buffers: shared hit=7
   -> Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.02 rows=102 width=0) (actual time=0.732..0.732 rows=200 loops=1)
    Index Cond: (unique1 < 100)
    Buffers: shared hit=2
   -> Bitmap Index Scan on tenk1_unique2  (cost=0.00..19.80 rows=1007 width=0) (actual time=2.015..2.015 rows=1009 loops=1)
    Index Cond: (unique2 > 9000)
    Buffers: shared hit=5
  Total runtime: 3.917 ms
```

BUFFERSにより提供される数は、問い合わせのどの部分がもっとも大きいI/Oであるかを識別する役に立ちます。

EXPLAIN ANALYZEが実際に問い合わせを実行しますので。 EXPLAINのデータを出力することを優先して問い合わせの出力が破棄されたとしても、何らかの副作用が通常とおり発生することに注意してください。 テーブルを変更すること無くデータ変更問い合わせの解析を行いたければ、以下の例のように、実行後コマンドをロールバックしてください。

```
BEGIN;

EXPLAIN ANALYZE UPDATE tenk1 SET hundred = hundred + 1 WHERE unique1 < 100;
```

```
-----
Update on tenk1 (cost=5.03..229.42 rows=101 width=250) (actual time=81.055..81.055 rows=0 loops=1)
-> Bitmap Heap Scan on tenk1 (cost=5.03..229.42 rows=101 width=250) (actual time=0.766..3.396 rows=100
loops=1)
    Recheck Cond: (unique1 < 100)
    -> Bitmap Index Scan on tenk1_unique1 (cost=0.00..5.01 rows=101 width=0) (actual
time=0.461..0.461 rows=100 loops=1)
        Index Cond: (unique1 < 100)
Total runtime: 81.922 ms

ROLLBACK;
```

この例で分かるように、問い合わせがINSERT、UPDATE、DELETEである場合、テーブル変更を行うための実作業は最上位のInsert、Update、Delete計画ノードで行われます。このノード以下にある計画ノードは、古い行の検索、新しいデータの計算、あるいはその両方を行います。このため、前に述べたものと同じ種類のビットマップテーブルスキャンがあり、その出力が更新される行を格納するUpdateノードに渡されることが分かります。データ変更ノードが実行時間の多くを費やす可能性があります（現在これが一番多くの時間を費やしています）が、プランナは現在その作業を考慮してコスト推定に何も加えません。これは、行われる作業がすべての正確な問い合わせ計画の作業と同一であるためであり、このため計画の決定に影響を与えません。

EXPLAIN ANALYZEで表示されるTotal runtime（総実行時間）にはエクゼキュータの起動、停止時間、発行される何らかのトリガの実行時間も含まれますが、解析や書き換え、計画作成の時間は含まれません。BEFOREトリガがあればその実行時間は関連するInsert、Update、Deleteノード用の時間に含まれます。しかし、AFTERトリガは計画全体が完了した後に発行されますので、AFTERトリガの実行時間は計上されません。また、各トリガ（BEFORE、AFTERのいずれか）で費やされる総時間は別々に表示されます。しかし、遅延制約トリガはトランザクションが終わるまで実行されませんので、EXPLAIN ANALYZEでは表示されないことに注意してください。

14.1.3. 警告

EXPLAIN ANALYZEにより測定される実行時間が同じ問い合わせを普通に実行する場合と大きくそれる可能性がある、2つの重大な点があります。1つ目は、出力行がクライアントに配信されませんので、ネットワーク転送コストとI/O変換に関するコストが含まれないことです。2つ目は、EXPLAIN ANALYZEによって加わる測定オーバーヘッドが大きくなることが、特にgettimeofday()オペレーティングシステムコールが低速なマシンであり得ることです。[pg_test_timing](#)を用いて、使用中のシステムの時間測定にかかるオーバーヘッドを測ることができます。

EXPLAINの結果を試験を行ったものと大きく異なる状況の推定に使ってはいけません。例えば、小さなテーブルの結果は、巨大なテーブルに適用できるとは仮定できません。プランナの推定コストは線形ではなく、そのため、テーブルの大小によって異なる計画を選択する可能性があります。極端な例ですが、テーブルが1ディスクページしか占めない場合、インデックスが使用できる、できないに関係なく、ほとんど常にシーケンシャルスキャン計画を得ることになります。プランナは、どのような場合でもテーブルを処理するために1ディスクページ読み取りを行うので、インデックスを参照するための追加的ページ読み取りを行う価値がないことを知っています。（上述のpolygon_tbl1の例でこれが起こることを示しています。）

実際の値と推定値がうまく合わないが本当は間違っただけの場合があります。こうした状況の1つは、LIMITや同様な効果により計画ノードの実行が短時間で終わる時に起こります。例えば、以前に使用したLIMIT問い合わせでは

```
EXPLAIN ANALYZE SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000 LIMIT 2;
```

QUERY PLAN

```
-----
Limit (cost=0.00..14.25 rows=2 width=244) (actual time=1.652..2.293 rows=2 loops=1)
-> Index Scan using tenk1_unique2 on tenk1 (cost=0.00..71.23 rows=10 width=244) (actual
time=1.631..2.259 rows=2 loops=1)
    Index Cond: (unique2 > 9000)
    Filter: (unique1 < 100)
    Rows Removed by Filter: 287
Total runtime: 2.857 ms
```

インデックススキャンノードの推定コストと行数が実行完了したかのように表示されます。しかし現実では、Limitノードが2行を取り出した後に行の要求を停止します。このため実際の行数は2行のみであり、実行時間は提示された推定コストより小さくなります。これは推定間違いではなく、単なる推定値と本当の値を表示する方法における矛盾です。

またマージ結合には、注意しないと混乱を招く測定上の乱れがあります。マージ結合は他の入力を使い尽くされ、ある入力の次のキー値が他の入力の最後のキー値より大きい場合、その入力の読み取りを停止します。このような場合、これ以上一致することはあり得ず、最初の入力の残りをスキャンする必要があります。この結果、子のすべては読み取られず、LIMITの説明のようになります。また、外部（最初）の子が重複するキー値を持つ行を含む場合、内部（2番目）の子はバックアップされ、そのキー値が一致する行部分を再度スキャンされます。EXPLAIN ANALYZEはこうした繰り返される同じ内部行の排出を実際の追加される行と同様に計上します。外部で多くの重複がある場合、内部の子計画ノードで繰り返される実際の行数は、内部リレーションにおける実際の行数より非常に多くなることがあり得ます。

実装上の制限のため、BitmapAndおよびBitmapOrノードは常に実際の行数をゼロと報告します。

[前のページ](#)
性能に関するヒント

[ホーム](#)
[上に戻る](#)

[次のページ](#)
プランナで使用される統計情報

14.2. プランナで使用される統計情報

前節で説明した通り、問い合わせプランナは、より良い問い合わせ計画を選択するために問い合わせによって取り出される行数の推定値を必要としています。本節では、システムがこの推定に使用する統計情報について簡単に説明します。

統計情報の1つの構成要素は、各テーブルとインデックスの項目の総数と、各テーブルとインデックスが占めるディスクブロック数です。この情報は`pg_class`の`reltuples`と`relpages`列に保持されます。以下のような問い合わせによりこれを参照することができます。

```
SELECT relname, relkind, reltuples, relpages
FROM pg_class
WHERE relname LIKE 'tenk1%';
```

relname	relkind	reltuples	relpages
tenk1	r	10000	358
tenk1_hundred	i	10000	30
tenk1_thous_tenthous	i	10000	30
tenk1_unique1	i	10000	30
tenk1_unique2	i	10000	30

(5 rows)

ここで、`tenk1`とそのインデックスには10000行が存在し、そして、（驚くには値しません）インデックスはテーブルよりもかなり小さなものであることがわかります。

効率を上げるため、`reltuples`と`relpages`は処理の度には更新されず、したがって通常は多少古い値のみ所有しています。これらは`VACUUM`、`ANALYZE`、`CREATE INDEX`などの一部のDDLコマンドによって更新されます。テーブル全体をスキャンしない`VACUUM`、`ANALYZE`操作（一般的な状況です）は、スキャンされたテーブルの部分に基づいて`reltuples`値を漸次更新し、概算値を生成します。いずれの場合でもプランナは、現在の物理的なテーブルサイズに合わせるために`pg_class`から検索した値を調整して、より高精度な近似値を持ちます。

ほとんどの問い合わせは、検証される行を制限する`WHERE`句によって、テーブル内の行の一部のみを取り出します。したがって、プランナは`WHERE`句の選択性、つまり`WHERE`句の各条件にどれだけの行が一致するかを推定する必要があります。この処理に使用される情報は`pg_statistic`システムカタログ内に格納されます。`pg_statistic`内の項目は、`ANALYZE`と`VACUUM ANALYZE`コマンドによって更新され、また1から更新がかかったとしても常に概算値になります。

統計情報を手作業で確認する場合、`pg_statistic`を直接参照するのではなく、`pg_stats`ビューを参照する方が良いでしょう。`pg_stats`はより読みやすくなるように設計されています。さらに、`pg_stats`は誰でも読み取ることができますが、`pg_statistic`はスーパーユーザのみ読み取ることができます。（これは、非特権ユーザが統計情報から他人のテーブルの内容に関わる事項を読み取ることを防止します。`pg_stats`ビューは現在のユーザが読み取ることができるテーブルに関する行のみを表示するよう制限されています。）例えば、以下を行うことができます。

```
SELECT attname, inherited, n_distinct,
       array_to_string(most_common_vals, E'\n') as most_common_vals
FROM pg_stats
WHERE tablename = 'road';
```

attname	inherited	n_distinct	most_common_vals
name	f	-0.363388	I- 580 Ramp+ I- 880 Ramp+ Sp Railroad + I- 580 + I- 680 Ramp
name	t	-0.284859	I- 880 Ramp+ I- 580 Ramp+ I- 680 Ramp+ I- 580 + State Hwy 13 Ramp

(2 rows)

同じ列に対して2行が表示されていることに注意してください。1つは`road`テーブルが始まる継承階層(`inherited=t`)全体に相当し、もう1つは`road`テーブル自身(`inherited=f`)のみを含むものです。

`ANALYZE`により`pg_statistic`に格納される情報量、具体的には、それぞれの列に対する`most_common_vals`内と`histogram_bounds`配列のエントリの最大数は、`ALTER TABLE SET STATISTICS`コマンドによって列ごとに、`default_statistics_target`設定パラメータを設定することによってグローバルに設定することができます。現在のデフォルトの上限は100エントリです。この上限を上げることで、特に、少し変わったデータ分布を持つ列でより正確なプランナの推定が行われますが、`pg_statistic`により多くの容量が必要になり、多少推定計算にかかる時間が多くなります。反対に上限を下げることは、単純なデータ分布の列に対して順当です。

プランナによる統計情報の使用に関する詳細については[第58章](#)を参照してください。

14.3. 明示的なJOIN句でプランナを制御する

明示的なJOIN構文を使って問い合わせプランナをある程度制御できます。 どうしてこういうことが問題になるのか、まずその背景を見る必要があります。

単純な問い合わせ、例えば

```
SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
```

では、プランナは自由に与えられたテーブルを任意の順で結合することができます。 例えば、WHERE条件のa.id = b.idを使ってまずAとBを結合し、他のWHERE条件を使ってその結合テーブルにCを結合するといった計画を立てることができます。あるいは、BとCを結合し、その結果にAを結合することもできます。あるいは、AとCを結合し、その結果にBを結合することもできるでしょう。しかし、それでは効率が良くありません。なぜなら、結合の最適化を行うために適用できる条件がWHERE句にないので、AとCの全直積が作られるからです。（PostgreSQLのエクゼキュータでは、結合はすべて2つのテーブルの間で行われるため、このようにして1つひとつ結果を作っていかなければなりません。）重要なのは、これらの違った結合の方法は意味的には同じ結果なのですが、実行コストは大きく異なる可能性があるということです。ですから、プランナは最も効率の良い計画を探すために可能な計画をすべて検査します。

結合の対象がせいぜい2、3個のテーブルなら心配するほど結合の種類は多くありません。しかし、テーブル数が増えると可能な結合の数は指数関数的に増えていきます。10程度以上にテーブルが増えると、すべての可能性をしらみつぶしに探索することはもはや実用的ではなくなります。6や7個のテーブルでさえも、計画を作成する時間が無視できなくなります。テーブルの数が多過ぎる時は、PostgreSQLのプランナはしらみつぶしの探索から、限られた可能性だけを探索する遺伝的確率的な探索へと切り替わります。（切り替えの閾値は[geqo_threshold](#)実行時パラメータで設定されます。）遺伝的探索は短い時間で探索を行います、必ずしも最適な計画を見つけるとは限りません。

外部結合が含まれるような問い合わせでは、通常の（内部）結合よりプランナの選択の余地が小さくなります。例えば、次のような問い合わせを考えます。

```
SELECT * FROM a LEFT JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);
```

この問い合わせの検索条件は前述の例と表面的には似ているように思えますが、BとCの結合結果の行に適合しないAの各行が出力されなければならないため、意味的には異なります。したがって、ここではプランナには結合順に関して選択の余地がありません。まずBとCを結合し、その結果にAを結合しなければならないのです。そういうわけで、この問い合わせでは計画を立てるのに要する時間は前の例よりも短くなります。その他の場合、プランナが安全な結合順を複数決定できる可能性があります。例えば、以下を考えてみます。

```
SELECT * FROM a LEFT JOIN b ON (a.bid = b.id) LEFT JOIN c ON (a.cid = c.id);
```

この場合、Aを先にBと結合してもCと結合しても有効です。現時点では、FULL JOINのみが完全に結合順を制限します。LEFT JOINやRIGHT JOINを含む、ほとんどの実環境では、何らかの拡張に再調整することができます。

明示的な内部結合構文（INNER JOIN、CROSS JOIN、装飾のないJOIN）は、意味的にはFROM内の入力リレーションの列挙と同じです。したがって、結合順を制約しません。

ほとんどの種類のJOINは完全に結合順を制約しませんが、PostgreSQL問い合わせプランナに、すべてのJOIN句に対してとりえず結合順を制限させることができます。例えば、以下の3つの問い合わせは論理的には同一です。

```
SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
SELECT * FROM a CROSS JOIN b CROSS JOIN c WHERE a.id = b.id AND b.ref = c.id;
SELECT * FROM a JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);
```

しかし、プランナにJOINの順番を守るように伝えた場合、2番目と3番目の問い合わせは最初のものよりも短い時間で計画を立てることができます。この効果はたった3つのテーブルでは気にするほどのものではありませんが、多くのテーブルを結合する際には最後の頼みの綱になるかもしれません。

プランナを強制的に明示的なJOINに潜在する結合順に従わせるには、[join_collapse_limit](#)実行時パラメータを1に設定してください。（以下で他の取り得る値について説明します。）

検索時間を節約するために、結合順を完全に束縛する必要はありません。なぜなら、単純なFROMリストの項目内にJOIN演算子を使っても構わないからです。例えば、次の例です。

```
SELECT * FROM a CROSS JOIN b, c, d, e WHERE ...;
```

join_collapse_limit = 1とした場合、プランナは強制的に他のテーブルと結合する前にAとBを結合しますが、それ以外については特に拘束はありません。この例では、結合順の候補は5の階乗分の1に減ります。

こうした方法でプランナの検索に制約を加えることは、計画作成時間の短縮とプランナに対する優れた問い合わせ計画への方向

付けの両方のために有用な技法です。 プランナが劣った結合順をデフォルトで選択するのであれば、JOIN構文経由でより良い順番を選択するように強制することができます。 ただし、より良い順番を理解しているという前提があります。 これには実験することを勧めます。

計画作成時間に影響する密接に関連した問題として、副問い合わせをその親問い合わせに折り畳むことがあります。 例えば、以下を考えてみます。

```
SELECT *
FROM x, y,
      (SELECT * FROM a, b, c WHERE something) AS ss
WHERE somethingelse;
```

こうした状況は、結合を含むビューを使用する際に現れます。 そのビューのSELECTルールはビューを参照するところに挿入され、上のような問い合わせを生成します。 通常、プランナは副問い合わせを親問い合わせに折り畳み、以下を生成します。

```
SELECT * FROM x, y, a, b, c WHERE something AND somethingelse;
```

これは通常、副問い合わせの計画を別途作成するより優れた計画を作成します。（例えば、外部のWHERE条件はXをAに結合するようになり、まずAの多くの行が取り除かれます。 これにより、副問い合わせの完全な論理的出力が不要になります。） しかし、同時に計画作成時間が増加します。 この場合、2つの3通りの結合問題から5通りの結合問題になります。 候補数は指数関数的に増加するため、これは大きな違いになります。 プランナは大規模な結合検索問題で行き詰まらないように、もしfrom_collapse_limit個のFROM項目が親問い合わせで発生してしまう場合は副問い合わせの折り畳みを抑制します。 この実行時パラメータの値を上下に調整することで計画作成時間と計画の質をトレードオフすることができます。

両者はほとんど同じことを行うため、[from_collapse_limit](#)と[join_collapse_limit](#)は似たような名前になっています。 片方は副問い合わせの"平坦化"をプランナがいつ行うかを制御し、もう片方は明示的な結合の平坦化をいつ行うかを制御します。 通常、join_collapse_limitをfrom_collapse_limitと同じ値に設定する（明示的な結合と副問い合わせの動作を同じにする）か、join_collapse_limitを1に設定する（明示的な結合で結合順を制御したい場合）かのどちらかを行います。 しかし、計画作成時間と実行時間の間のトレードオフを十分に行う場合は、これらを別の値に設定しても構いません。

[前のページ](#)

プランナで使用される統計情報

[ホーム](#)

[上に戻る](#)

[次のページ](#)

データベースへのデータ投入

14.4. データベースへのデータ投入

データベースにデータを初期投入するために、大量のテーブル挿入操作を行う必要がままあります。本節では、この作業を効率良く行うためのちょっとした提言を示します。

14.4.1. 自動コミットをオフにする

複数回のINSERTを実行するのであれば、自動コミットを無効にして最後に1回だけコミットしてください。（普通のSQLでは、これはBEGINを開始時に、COMMITを最後に発行することを意味します。クライアント用ライブラリの中にはこれを背後で実行するものもあります。その場合は、要望通りにライブラリが行っているかどうかを確認しなければなりません。）各挿入操作で個別にコミットすることを許すと、PostgreSQLは行を追加する度に多くの作業をしなければなりません。1つのトランザクションですべての挿入を行うことによるもう1つの利点は、1つの行の挿入に失敗した場合、その時点までに挿入されたすべての行がロールバックされることです。その結果、一部のみがロードされたデータの対処に困ることはありません。

14.4.2. COPYの使用

単一コマンドですべての行をロードするために一連のINSERTコマンドではなく、COPYを使用してください。COPYコマンドは行を大量にロードすることに最適化されています。このコマンドはINSERTに比べ柔軟性に欠けていますが、大量のデータロードにおけるオーバーヘッドを大きく低減します。COPYコマンドでテーブルにデータを投入する場合、コマンドは1つなので、自動コミットを無効にする必要はありません。

COPYを使用できない場合、準備されたINSERT文をPREPAREを使用して作成し、必要回数だけEXECUTEを実行する方が良いでしょう。これにより、繰り返し行われるINSERTの解析と計画作成分のオーバーヘッドを省くことになります。この機能のための方法はインタフェースによって異なります。このインタフェースの文書の"準備された文"を参照してください。

COPYを使用した大量の行のロードは、ほとんどすべての場合において、INSERTを使用するロードよりも高速です。たとえ複数の挿入を単一トランザクションにまとめたとしても、またその際にPREPAREを使用したとしても、これは当てはまります。

COPYは、前もって行われるCREATE TABLEまたはTRUNCATEコマンドと同一トランザクションで行った場合に、最速です。この場合、エラーが起きた場合に新しくロードされるデータを含むファイルがとにかく削除されますので、WALを書き出す必要がありません。しかし、wal_levelがminimalに設定されている場合のみにこの方法は当てはまります。この他の場合には、すべてのコマンドをWALに書き出さなければならないためです。

14.4.3. インデックスを削除する

新規に作成したテーブルをロードする時、最速の方法は、テーブルを作成し、COPYを使用した一括ロードを行い、そのテーブルに必要なインデックスを作成することです。既存のデータに対するインデックスを作成する方が、各行がロードされる度に段階的に更新するよりも高速です。

既存のテーブルに大量のデータを追加しているのであれば、インデックスを削除し、テーブルをロード、その後にインデックスを再作成する方がよいかもしれません。もちろん、他のユーザから見ると、インデックスが存在しない間データベースの性能は悪化します。また、一意性インデックスを削除する前には熟考しなければなりません。一意性制約によるエラーチェックがその期間行われないからです。

14.4.4. 外部キー制約の削除

インデックスの場合と同様、外部キー制約は一行一行検査するよりも効率的に、"まとめて"検査することができます。従って、外部キー制約を削除し、データをロード、そして、制約を再作成する方法は有用となる場合があります。繰り返しますが、データロードの速度と、制約が存在しない間のエラー検査がないという点とのトレードオフがあります。

外部キー制約をすでに持つテーブルにデータをロードする時、新しい行はそれぞれ(行の外部キー制約を検査するトリガを発行しますので)サーバの待機中トリガイベントのリスト内に項目を要求します。数百万の行をロードすると、トリガイベントのキューが利用可能なメモリをオーバーフローさせてしまい、耐えられないほどのスワッピングが発生してしまう、最悪はそのコマンドが完全に失敗してしまう可能性があります。したがって単に好ましいだけでなく、大量のデータをロードする時には外部キーを削除し再度適用することが必要かもしれません。一時的な制約削除が受け入れられない場合に他に取れる手段は、ロード操作をより小さなトランザクションに分割することだけかもしれません。

14.4.5. maintenance_work_memを増やす

大規模なデータをロードする時[maintenance_work_mem](#)設定変数を一時的に増やすことで性能を向上させることができます。これは、CREATE INDEX コマンドとALTER TABLE ADD FOREIGN KEYの速度向上に役立ちます。COPY 自体には大して役立ちませんので、この助言は、上述の技法の片方または両方を使用している時にのみ有効です。

14.4.6. checkpoint_segmentsを増やす

大規模なデータをロードする時[checkpoint_segments](#)設定変数を一時的に増やすことで高速化することができます。大量のデータをPostgreSQLにロードすることで、通常のチェックポイントの頻度（[checkpoint_timeout](#)設定変数により指定されます）よりも頻繁にチェックポイントが発生するためです。チェックポイントが発生すると、すべてのダーティページ（ディスクに未書き込みの変更済みメモリページ）はディスクに吐き出されなければなりません。大量のデータロードの際に一時的に[checkpoint_segments](#)を増加させることで、必要なチェックポイント数を減らすことができます。

14.4.7. WALアーカイブ処理とストリーミングレプリケーションの無効化

大量のデータをWALアーカイブ処理またはストリーミングレプリケーションを使用するインストレーションにロードする時、増加する大量のWALデータを処理するより、ロードが完了した後に新しくベースバックアップを取る方が高速です。ロード中のWALログの増加を防ぐためには、[wal_level](#)をminimalに、[archive_mode](#)をoffに、[max_wal_senders](#)をゼロに設定することにより、アーカイブ処理とストリーミングレプリケーションを無効にしてください。しかし、これらの変数を変更するにはサーバの再起動が必要な点に注意してください。

こうすると、WALデータを処理する保管処理またはWAL送信処理にかかる時間がかからないことの他に、実際のところ、特定のコマンドをより高速にします。[wal_level](#)がminimalの場合、これらのコマンドではWALへの書き出しは全く予定されないためです。（これらは最後にfsyncを実行することで、WALへの書き込みより安価にクラッシュした場合の安全性を保証することができます。）これは以下のコマンドで当てはまります。

- CREATE TABLE AS SELECT
- CREATE INDEX（またはALTER TABLE ADD PRIMARY KEYなどの亜種）
- ALTER TABLE SET TABLESPACE
- CLUSTER
- 同一トランザクションで前もって対象テーブルが作成された、あるいは、消去された場合のCOPY FROM

14.4.8. 最後にANALYZEを実行

テーブル内のデータ分布を大きく変更した時は毎回、[ANALYZE](#)を実行することを強く勧めます。これは、テーブルに大量のデータをまとめてロードする場合も含まれます。[ANALYZE](#)（またはVACUUM ANALYZE）を実行することで、確実にプランナがテーブルに関する最新の統計情報を持つことができます。統計情報が存在しない、または古い場合、プランナは、そのテーブルに対する問い合わせの性能を損なわせる、お粗末な問い合わせ計画を選択する可能性があります。自動バキュームデーモンが有効な場合、[ANALYZE](#)が自動的に実行されます。詳細は[項23.1.3](#)および[項23.1.6](#)を参照してください。

14.4.9. pg_dumpに関するいくつかの注意

pg_dumpで生成されるダンプスクリプトは自動的に上のガイドラインのいくつかを適用します（すべてではありません）。pg_dumpダンプをできる限り高速にリロードするには、手作業で更に数作業が必要です。（これらは作成時に適用するものではなく、ダンプを復元する時に適用するものです。psqlを使用してテキスト形式のダンプをロードする時とpg_dumpのアーカイブファイルからpg_restoreを使用してロードする時にも同じことが適用できます。）

デフォルトでは、pg_dumpはCOPYを使用します。スキーマとデータのダンプ全体を生成する場合、インデックスと外部キー制約を作成する前にデータをロードすることに注意してください。ですので、この場合、ガイドラインのいくつかは自動的に行われます。残された作業は以下のとおりです。

- [maintenance_work_mem](#)および[checkpoint_segments](#)を適切な（つまり通常よりも大きな）値に設定します。
- WALアーカイブ処理またはストリーミングレプリケーションを使用する場合は、リストア時にこれを無効にすることを検討してください。このためにはダンプをロードする前に[archive_mode](#)をoffに、[wal_level](#)をminimalに、[max_wal_senders](#)をゼロに設定してください。その後それらを正しい値に戻し、新規にベースバックアップを取ってください。
- ダンプ全体を単一トランザクションとしてリストアすべきかどうか検討してください。このためにはpsqlまたはpg_restoreに-iまたは--single-transactionコマンドラインオプションを指定してください。このモードを使用する場合、たとえ小さなエラーであっても、エラーがあればリストア全体がロールバックされます。データ同士の関連性がど

の程度あるかに依存しますが、手作業での整理の際には好まれるかと思います。さもなくばあまり勧めません。単一トランザクションで実行し、WALアーカイブを無効にしている場合、COPY コマンドは最も高速に行われます。

- データベースサーバで複数のCPUが利用できるのであれば、pg_restoreの--jobs オプションの利用を検討してください。これによりデータのロードとインデックスの作成を同時に行うことができます。
- この後でANALYZEを実行してください。

データのためのダンプもCOPY コマンドを使用しますが、インデックスの削除と再作成を行いません。また、通常は外部キー制約を変更しません。 [1] したがって、データのためのダンプをロードする時、上の技法を使用したければ自らインデックスと外部キーを削除、再作成しなければなりません。データをロードする時にcheckpoint_segmentsを増やすことも有用です。しかし、maintenance_work_memを増やすことは考えないでください。これは、後でインデックスと外部キーを手作業で再作成する時に行う方がよいでしょう。また、実行した後でANALYZEを行うことを忘れないでください。詳細は項23.1.3および項23.1.6を参照してください。

注意

[1]

--disable-triggers オプションを使用して、外部キーを無効にさせることができます。しかし、これは外部キー制約を遅らせるのではなく、除去することに注意してください。そのため、これを使用すると不正なデータを挿入することができてしまいます。

[前のページ](#)

明示的なJOIN句でプランナを制御する

[ホーム](#)
[上に戻る](#)

[次のページ](#)

永続性がない設定

14.5. 永続性がない設定

永続性とは、サーバがクラッシュしたり電源が落ちたりしたとしても、コミットされたトランザクションが記録されていることを保証するデータベースの機能です。しかし、永続性はデータベースに多くのオーバーヘッドを与えます。このためこの保証を必要としないサイトでは、PostgreSQLをかなり高速に実行するように設定することができます。以下に、こうした状況で性能を向上させるために行うことができる設定変更を示します。後述の注意を除き、データベースソフトウェアがクラッシュした場合でも、永続性は保証されています。突然のオペレーティングシステムの停止だけが、この設定を使用した時のデータ損失、破損の危険性を引き起こします。

- データベースクラスタのデータディレクトリをメモリ上のファイルシステム（つまりRAMディスク）に設定します。これはすべてのデータベースによるディスクI/Oを取り除きますが、データ量が利用可能なメモリ（およびスワップも使われるかもしれません）量までに制限されます。
- [fsync](#)を無効にします。データをディスクに吐き出す必要がありません。
- [full_page_writes](#)を無効にします。部分的なページ書き出しから保護する必要がありません。
- [checkpoint_segments](#)および[checkpoint_timeout](#)を増加させます。これによりチェックポイントの頻度が減少しますが、`/pg_xlog`で必要とする容量が増加します。
- [synchronous_commit](#)を無効にします。コミット毎にディスクにWALを書き出す必要がありません。この設定は、データベース単体のクラッシュした場合、トランザクション損失（データ破損ではありません）の危険性があります。

III. サーバの管理

ここでは、PostgreSQLデータベース管理者にとって関心のある事項を扱います。これには、ソフトウェアのインストール、サーバの設定や構成、ユーザやデータベースの管理、および保守作業が含まれます。PostgreSQLサーバを個人的に使用している場合もそうですが、特に業務で使用している場合は、ここで扱う事項に精通する必要があります。

ここに記載された情報は、大体において新規ユーザが読み進めるべき順番に並べられています。ただ、章ごとに内容が独立していますので、必要に応じて個々の章を読むこともできます。ここでの情報は、項目単位で限定して説明されています。特定のコマンドについて完全な説明を知りたい場合は、[パートVI](#)を参照してください。

最初の数章は、これからサーバを構築する新規ユーザにも読めるように、前提知識がなくても理解できるようになっています。残りの部分では調整や管理について記されていますが、その資料は読者がPostgreSQLデータベースシステムの一般的な使用について理解していることを前提としています。詳細な情報については[パートI](#)や[パートII](#)を参照することをお勧めします。

目次

15章 [ソースコードからインストール](#)

- 15.1. [簡易版](#)
- 15.2. [必要条件](#)
- 15.3. [ソースの入手](#)
- 15.4. [インストール手順](#)
- 15.5. [インストール後の設定作業](#)
- 15.6. [サポートされるプラットフォーム](#)
- 15.7. [プラットフォーム特有の覚書](#)

16章 [Windowsにおけるソースコードからのインストール](#)

- 16.1. [Visual C++またはMicrosoft Windows SDKを使用した構築](#)
- 16.2. [Visual C++またはBorland C++を使用したlibpqの構築](#)

17章 [サーバの準備と運用](#)

- 17.1. [PostgreSQLユーザアカウント](#)
- 17.2. [データベースクラスタの作成](#)
- 17.3. [データベースサーバの起動](#)
- 17.4. [カーネルリソースの管理](#)
- 17.5. [サーバのシャットダウン](#)
- 17.6. [PostgreSQLクラスタのアップグレード処理](#)
- 17.7. [サーバのなりすまし防止](#)
- 17.8. [暗号化オプション](#)
- 17.9. [SSLによる安全なTCP/IP接続](#)
- 17.10. [SSHトンネルを使った安全なTCP/IP接続](#)
- 17.11. [WindowsにおけるEvent Logの登録](#)

18章 [サーバの設定](#)

- 18.1. [パラメータの設定](#)
- 18.2. [ファイルの場所](#)
- 18.3. [接続と認証](#)
- 18.4. [資源の消費](#)
- 18.5. [ログ先行書き込み \(WAL\)](#)
- 18.6. [レプリケーション](#)
- 18.7. [問い合わせ計画](#)
- 18.8. [エラー報告とログ取得](#)
- 18.9. [実行時統計情報](#)
- 18.10. [自動Vacuum作業](#)
- 18.11. [クライアント接続デフォルト](#)
- 18.12. [ロック管理](#)
- 18.13. [バージョンとプラットフォーム互換性](#)
- 18.14. [エラー処理](#)
- 18.15. [設定済みのオプション](#)
- 18.16. [独自のオプション](#)
- 18.17. [開発者向けのオプション](#)

18.18. [短いオプション](#)

19章 [クライアント認証](#)

- 19.1. [pg_hba.conf](#) ファイル
- 19.2. [ユーザ名マップ](#)
- 19.3. [認証方式](#)
- 19.4. [認証における問題点](#)

20章 [データベースロール](#)

- 20.1. [データベースロール](#)
- 20.2. [ロールの属性](#)
- 20.3. [ロールのメンバ資格](#)
- 20.4. [関数とトリガのセキュリティ](#)

21章 [データベース管理](#)

- 21.1. [概要](#)
- 21.2. [データベースの作成](#)
- 21.3. [テンプレートデータベース](#)
- 21.4. [データベースの設定](#)
- 21.5. [データベースの削除](#)
- 21.6. [テーブル空間](#)

22章 [多言語対応](#)

- 22.1. [ロケールのサポート](#)
- 22.2. [照合サポート](#)
- 22.3. [文字セットサポート](#)

23章 [定常的なデータベース保守作業](#)

- 23.1. [定常的なバキューム作業](#)
- 23.2. [定常的なインデックスの再作成](#)
- 23.3. [ログファイルの保守](#)

24章 [バックアップとリストア](#)

- 24.1. [SQLによるダンプ](#)
- 24.2. [ファイルシステムレベルのバックアップ](#)
- 24.3. [継続的アーカイブとポイントインタイムリカバリ \(PITR\)](#)

25章 [高可用性、負荷分散およびレプリケーション](#)

- 25.1. [様々な解法の比較](#)
- 25.2. [ログ SHIPPING スタンバイサーバ](#)
- 25.3. [フェールオーバー](#)
- 25.4. [その他のログ SHIPPING の方法](#)
- 25.5. [ホットスタンバイ](#)

26章 [リカバリの設定](#)

- 26.1. [アーカイブリカバリの設定](#)
- 26.2. [リカバリ対象の設定](#)
- 26.3. [スタンバイサーバの設定](#)

27章 [データベース活動状況の監視](#)

- 27.1. [標準的なUnixツール](#)
- 27.2. [統計情報コレクタ](#)
- 27.3. [ロックの表示](#)
- 27.4. [動的追跡](#)

28章 [ディスク使用量の監視](#)

- 28.1. [ディスク使用量の決定](#)
- 28.2. [ディスク容量不足による問題](#)

29章[信頼性とログ先行書き込み](#)

- 29.1. [信頼性](#)
- 29.2. [ログ先行書き込み\(WAL\)](#)
- 29.3. [非同期コミット](#)
- 29.4. [WALの設定](#)
- 29.5. [WALの内部](#)

30章[リグレッションテスト](#)

- 30.1. [テストの実行](#)
- 30.2. [テストの評価](#)
- 30.3. [各種の比較用ファイル](#)
- 30.4. [テストが網羅する範囲の検証](#)

[前のページ](#)

永続性がない設定

[ホーム](#)

[次のページ](#)

ソースコードからインストール

第 15 章 ソースコードからインストール

目次

- 15.1. [簡易版](#)
- 15.2. [必要条件](#)
- 15.3. [ソースの入手](#)
- 15.4. [インストール手順](#)
- 15.5. [インストール後の設定作業](#)
 - 15.5.1. [共有ライブラリ](#)
 - 15.5.2. [環境変数](#)
- 15.6. [サポートされるプラットフォーム](#)
- 15.7. [プラットフォーム特有の覚書](#)
 - 15.7.1. [AIX](#)
 - 15.7.2. [Cygwin](#)
 - 15.7.3. [HP-UX](#)
 - 15.7.4. [IRIX](#)
 - 15.7.5. [MinGW/ネイティブWindows](#)
 - 15.7.6. [SCO OpenServer と SCO UnixWare](#)
 - 15.7.7. [Solaris](#)

この章では、PostgreSQLのソースコード配布物を使用したインストール方法について説明します（RPMやDebianパッケージなどパッケージ済みの配布物をインストールしている場合は、この章を無視し、代わりにパッケージの手順を読んでもください）。

15.1. 簡易版

```
./configure
gmake
su
gmake install
adduser postgres
mkdir /usr/local/pgsql/data
chown postgres /usr/local/pgsql/data
su - postgres
/usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
/usr/local/pgsql/bin/postgres -D /usr/local/pgsql/data >logfile 2>&1 &
/usr/local/pgsql/bin/createdb test
/usr/local/pgsql/bin/psql test
```

本章の残りで詳細を説明します。

15.2. 必要条件

通常、最近のUnix互換プラットフォームならばPostgreSQLを動作させることができるはずです。リリース時に明示的なテストを受けたプラットフォームの一覧を[項15.6](#)に示します。配布物のdocサブディレクトリには、プラットフォーム固有のFAQ文書がいくつかありますので、問題が発生した場合は参照してください。

PostgreSQLを構築するには、以下のソフトウェアパッケージが必要です。

- GNU makeのバージョン3.80以上が必要です。他のmakeや古いGNU makeでは動作しません。GNU makeはよくgmakeという名前でインストールされます。本書では常にその名前を使用します（システムによって、GNU makeがデフォルトのツールになっているものもあります。）GNU makeの試験を行うためには以下を実行してください。

```
gmake --version
```
- ISO/ANSI Cコンパイラ（最低限C89-準拠）が必要です。GCC の最近のバージョンをお勧めしますが、PostgreSQLは異なるベンダの、様々なコンパイラを使用して構築できることで知られています。
- 配布物を展開するために、tarおよびgzipかbzip2のどちらかが必要です。
- GNU Readlineライブラリは、デフォルトで使用されます。これによりpsql（PostgreSQLコマンドラインSQLインタプリタ）は入力したコマンドの記憶、さらに、カーソルキーを使用した過去のコマンドの再実行や編集ができるようになります。これは非常に役に立ちますので、強く推奨します。使用したくない場合は、configureに--without-readlineオプションを指定する必要があります。その代わりとして、BSDライセンスのlibeditライブラリを使用することもできます。このライブラリは元々NetBSDで開発されていました。libeditライブラリはGNUのReadlineと互換性があり、libreadlineを認識できなかった場合やconfigureのオプションに--with-libedit-preferredが使用された場合に使用されます。パッケージベースのLinuxディストリビューションを使用し、そのディストリビューションの中でreadlineとreadline-develパッケージが別個に存在していた場合、両方とも必要ですので注意してください。
- zlib圧縮ライブラリはデフォルトで使用されます。これを使用しなくなければ、configureに対し--without-zlibを指定しなければなりません。このオプションを使用すると、pg_dumpおよびpg_restore内の圧縮アーカイブサポートが無効になります。

以下のパッケージはオプションです。これらはデフォルトの設定では必要ありませんが、下記のように特定の構築オプションを有効とする場合に必要となります。

- サーバプログラム言語であるPL/Perlを構築するには、libperlライブラリとヘッダファイルを含む完全なPerlのインストールが必要です。PL/Perlは共有ライブラリですので、ほとんどのプラットフォームにおいてlibperlライブラリも共有ライブラリでなければなりません。これは最近のバージョンのPerlではデフォルトのようですが、以前のバージョンではデフォルトではありませんでした。とにかく、これはPerlをサイトにインストールした人により決定されます。もし、PL/Perlを意図的に使用するつもりであるのなら、Perlのインストールがusemultiplicityオプションを有効にして実施されているかを確認すべきです。(perl -vにより有効かどうかを確認できます。)

必要な共有ライブラリが存在しない場合、PostgreSQL構築中に表示される次のようなメッセージにより、そのことが指摘されます。

```
*** Cannot build PL/Perl because libperl is not a shared library.
*** You might have to rebuild your Perl installation. Refer to
*** the documentation for details.
```

（画面上の指示に従わない場合は、単にPL/Perlライブラリオブジェクトplperl.soまたは類似のものがインストールされないだけです。）このメッセージが表示された場合、PL/Perlを構築できるようにするために、Perlを手動で再構築し、インストールする必要があります。Perlの構築処理で、共有ライブラリとするように指定してください。

- PL/Pythonサーバプログラム言語を構築するには、ヘッダファイルとdistutilsモジュールを含むPythonのインストールが必要です。Pythonバージョン2.3が最低でも必要です。バージョン3.1以降であればPython 3もサポートされます。しかしPython 3を使用する場合は[項42.1](#)を参照してください。

PL/Pythonは共有ライブラリになりますので、ほとんどのプラットフォームでは、libpythonもまた共有ライブラリである必要があります。これは、Pythonのインストールではデフォルトではありません。PostgreSQLを構築し、インストールした後に、（他の拡張子かもしれませんが）pypython.soという名前のファイルがあれば、全てが成功しています。さもなくば、以下のような警告が表示されたはずです。

```
*** Cannot build PL/Python because libpython is not a shared library.
*** You might have to rebuild your Python installation. Refer to
```

*** the documentation for details.

これは、この共有ライブラリを作成するためにはPythonインストレーション（の一部）を再構築する必要があることを意味しています。

問題があった場合は、Python 2.3以降のconfigureを`--enable-shared`を付けて実行してください。オペレーティングシステムの中には、共有ライブラリの構築が必要ないものもあります。しかし、その場合はPostgreSQL構築システムにそのことを確認・指定する必要があります。詳しくはsrc/pl/plpythonディレクトリのMakefileを参照してください。

- PL/Tcl手続き言語の構築には、もちろんTclのインストレーションが必要です。8.4より前のリリースのTclを使用している場合は、マルチスレッドのサポートなしで構築していることを確認してください。
- 各国語サポート（NLS）、つまり、英語以外の言語によるプログラムメッセージの表示機能を有効にするには、Gettext APIの実装が必要です。オペレーティングシステムの中には（例えば、Linux、NetBSD、Solarisなど）、組み込み済みのものがあります。他のシステムでは、追加パッケージを<http://www.gnu.org/software/gettext/>からダウンロードすることができます。GNU Cライブラリのgettextの実装を使用する場合、さらにいくつかのユーティリティプログラムのためにGNU Gettextパッケージが必要となります。他の実装の場合には必要ありません。
- Kerberos、OpenSSL、OpenLDAP、PAM、または、この全ても、そのサービスを使用した認証や暗号化をサポートする場合には必要となります。

リリースされたソースパッケージではなくGitツリーからの構築の場合、またはサーバ開発を行いたい場合には、以下のパッケージも必要となります。

- GNUのFlexおよびBisonは、Gitチェックアウトから構築する場合や、実際のスキャナとパーサの定義ファイルを変更した場合に必要となります。それらが必要な場合は、Flex 2.5.31以降とBison 1.875以降を使うようにしてください。他のlexとyaccプログラムは使用できません。
- Gitからチェックアウトしたものから構築する場合や、構築時にPerlスクリプトを使用して作成されるファイルの入力となるファイルを変更する場合にはPerl5.8以降が必要です。Windows上で構築する場合、いずれにしてもPerlは必要です。

GNUパッケージの入手が必要な場合、近くのGNUミラーサイトから探してください（ミラーサイトの一覧は<http://www.gnu.org/order/ftp.html>にあります）。または、<ftp://ftp.gnu.org/gnu/>から探してください。

また、十分なディスク領域があることも確認してください。コンパイル中、ソースツリーのために100メガバイト、インストールディレクトリに20メガバイトほど必要となります。空のデータベースクラスタだけでも約35メガバイト必要です。データベースは、同じデータのフラットテキストファイルと比べて5倍ほどの領域が必要になります。リグレーションテストを実行する場合は、一時的に最大で150メガバイトの領域がさらに必要になります。ディスクの空き容量を確認するためにはdfコマンドを使います。

[前のページ](#)
簡易版

[ホーム](#)
[上に戻る](#)

[次のページ](#)
ソースの入手

15.3. ソースの入手

PostgreSQL 9.2.4のソースは、<ftp://ftp.postgresql.org/pub/v9.2.4/postgresql-9.2.4.tar.gz>からアノニマスFTPで入手できます。その他のダウンロード方法をWebサイト<http://www.postgresql.org/download/>に記載しています。 ファイルを入手したら、展開します。

```
gunzip postgresql-9.2.4.tar.gz
tar xf postgresql-9.2.4.tar
```

これにより、カレントディレクトリ以下に`postgresql-9.2.4`というディレクトリが作成され、PostgreSQLのソースが展開されます。この後のインストール手順を行うために、このディレクトリに移動してください。

またバージョン管理システムのリポジトリから直接ソースを入手することができます。[付録I](#)を参照してください。

15.4. インストール手順

1 11 構成

インストール手順の最初のステップは、システムに合わせてソースツリーを設定し、使用するオプションを選択することです。configureスクリプトを実行することでこれを行います。デフォルトのインストールを行う場合は、単に以下を入力してください。

```
./configure

mkdir build_dir
cd build_dir
/path/to/source/tree/configure [options go here]
gmake
```

デフォルトの構成では、サーバ、ユーティリティの他に、Cコンパイラだけを必要とするクライアントアプリケーションやインタフェースを構築します。デフォルトでは、全てのファイルは/usr/local/pgsql以下にインストールされます。

configureに以下のコマンドラインオプションを1つ以上指定することで、構築処理やインストール処理を変更することができます。

`--prefix=PREFIX`

/usr/local/pgsqlではなく、PREFIXディレクトリ以下に全てのファイルをインストールします。ファイルは実際には様々なサブディレクトリにインストールされ、PREFIXディレクトリの直下にインストールされるファイルはありません。

特別な必要性があるのであれば、以下のオプションを使用して個々のサブディレクトリを変更することもできます。しかし、これらをそのまま使用した場合、インストレーションは位置再変更可能になります。つまり、インストールの後にディレクトリを移動することができます（manとdocの場所はこの影響を受けません。）

インストールの位置再変更のために、configureの--disable-rpathを使用しようとするかもしれません。その場合は、オペレーティングシステムにその共有ライブラリの場所を通知する必要があるでしょう。

`--exec-prefix=EXEC-PREFIX`

アーキテクチャ依存のファイルをPREFIXの設定とは別の接頭辞EXEC-PREFIX以下にインストールすることができます。ホスト間でアーキテクチャ非依存のファイルを共有する場合に便利です。省略した場合、EXEC-PREFIXはPREFIXと同じに設定され、アーキテクチャに依存するファイルも非依存なファイルも同じツリー以下にインストールされます。ほとんどの場合、これが望まれています。

`--bindir=DIRECTORY`

実行可能プログラム用のディレクトリを指定します。デフォルトではEXEC-PREFIX/binであり、通常/usr/local/pgsql/binとなります。

`--sysconfdir=DIRECTORY`

各種設定ファイル用のディレクトリを設定します。デフォルトではPREFIX/etcです。

`--libdir=DIRECTORY`

ライブラリや動的ロード可能モジュールをインストールする場所を設定します。デフォルトはEXEC-PREFIX/libです。

`--includedir=DIRECTORY`

CおよびC++のヘッダファイルをインストールするディレクトリを設定します。デフォルトはPREFIX/includeです。

`--datarootdir=DIRECTORY`

いろいろな種類の読み取り専用データファイル用のルートディレクトリを設定します。これは後述のオプションの一部についてのデフォルトを設定するだけです。デフォルトはPREFIX/shareです。

`--datadir=DIRECTORY`

インストールプログラムが使用する読み取り専用のディレクトリを設定します。デフォルトはDATAROOTDIRです。これはインストールするデータベースファイルがどこに設置されるかとは関係ないことを覚えておってください。

--localedir=*DIRECTORY*

特にメッセージ翻訳カタログファイルのロケールデータをインストールするディレクトリを設定します。デフォルトは*DATAROOTDIR/locale*です。

--mandir=*DIRECTORY*

PostgreSQL付属のマニュアルページがこのディレクトリ以下の、対応する*manX*サブディレクトリにインストールされます。デフォルトは*DATAROOTDIR/man*です。

--docdir=*DIRECTORY*

"man"ページを除いた、ドキュメント一式ファイルをインストールするルートディレクトリを設定します。これは以下のオプションのデフォルトのみを設定します。このオプションのデフォルト値は*DATAROOTDIR/doc/postgresql*です。

--htmldir=*DIRECTORY*

PostgreSQLのHTML形式化文書一式はこのディレクトリの下にインストールされます。デフォルトは*DATAROOTDIR*です。

注意: (*/usr/local/include*といった) 共用のインストール場所に、システムの他の名前空間に影響を与えことなくPostgreSQLをインストールすることができるような配慮がなされています。まず、完全に展開したディレクトリ名に"postgres"か"pgsql"という文字列が含まれていない場合、"/postgresql"という文字列が自動的に*datadir*、*sysconfdir*、*docdir*に追加されます。例えば、接頭辞として*/usr/local*を使用する場合、文書は*/usr/local/doc/postgresql*にインストールされますが、接頭辞が*/opt/postgres*の場合は*/opt/postgres/doc*にインストールされます。クライアントインタフェース用の外部向けCヘッダファイルは*includedir*にインストールされ、名前空間の問題はありません。内部向けヘッダファイルやサーバ用ヘッダファイルは、*includedir*以下の非公開ディレクトリにインストールされます。各インタフェース用のヘッダファイルにアクセスする方法についての情報は、そのインタフェースの文書を参照してください。最後に、適切であれば、動的ロード可能モジュール用に*libdir*以下にも非公開用のサブディレクトリが作成されます。

--with-includes=*DIRECTORIES*

*DIRECTORIES*には、コンパイラがヘッダファイルを検索するディレクトリのリストをコロンで区切って指定します。(GNU Readlineなどの) オプションのパッケージが非標準的な場所にインストールされている場合、このオプションと、おそらく対応する--with-librariesオプションを使用する必要があります。

例: --with-includes=/opt/gnu/include:/usr/sup/include

--with-libraries=*DIRECTORIES*

*DIRECTORIES*には、ライブラリを検索するディレクトリのリストをコロンで区切って指定します。パッケージが非標準的な場所にインストールされている場合は、おそらくこのオプション (と対応する--with-includesオプション) を使用する必要があります。

例: --with-libraries=/opt/gnu/lib:/usr/sup/lib

--enable-nls[=*LANGUAGES*]

各国語サポート (NLS)、つまり、英語以外の言語によるプログラムメッセージの表示機能を有効にします。*LANGUAGES*はオプションであり、サポートさせたい言語コードを空白で区切ったリストを指定します。例えば、--enable-nls='de fr'などとし (指定したリストと実際に用意された翻訳との論理積が自動的に計算されます)。リストに何も指定しなかった場合、利用可能な翻訳すべてがインストールされます。

このオプションを使用するためには、gettext APIの実装が必要です。上記を参照してください。

--with-pgport=*NUMBER*

サーバとクライアントのデフォルトのポート番号を*NUMBER*に設定します。デフォルトは5432です。このポートは後でいつでも変更することができますが、ここで指定した場合、サーバとクライアントはコンパイル時に同じデフォルト値を持つようになります。これは非常に便利です。通常、デフォルト以外の値を選択すべき唯一の理由は、同じマシンで複数のPostgreSQLを稼働させることです。

--with-perl

PL/Perlサーバサイド言語を構築します。

--with-python

PL/Pythonサーバサイド言語を構築します。

--with-tcl

PL/Tclサーバサイド言語を構築します。

--with-tclconfig=DIRECTORY

Tclは、Tclへのインタフェースモジュールを構築するために必要な設定情報を含むtclConfig.shファイルをインストールします このファイルは通常、自動的に一般的に知られている場所にありますが、もしTclの別のバージョンを使いたい場合は、検索対象のディレクトリを指定することができます。

--with-gssapi

GSSAPI認証のサポートを構築します。多くのシステムでは、GSSAPIシステム（通常Kerberosインストールの一部）はデフォルトの検索場所（例えば/usr/includeや/usr/lib）にインストールされていません。そのため、--with-includesと--with-librariesオプションをさらに追加して使わなければいけません。configureは、処理を進める前にGSSAPIが正しくインストールされていることを確認するために、必要とされるヘッダファイルとライブラリを検査します。

--with-krb5

Kerberos 5 認証のサポートを構築します。多くのシステムでは、Kerberosシステムはデフォルトの検索場所（例えば/usr/includeや/usr/lib）にインストールされていません。そのため、--with-includesと--with-librariesオプションをさらに追加して使わなければいけません。configureは、処理を進める前にKerberosが正しくインストールされていることを確認するために、必要とされるヘッダファイルとライブラリを検査します。

--with-krb-srvnam=NAME

Kerberos のサービスプリンシパルのデフォルトの名前です（GSSAPIでも使用されます）。デフォルトでは"postgres"です。これを変える理由はWindows環境がない限り、特にありません。Windows環境がある場合は大文字のPOSTGRESに設定する必要があります。

--with-openssl

SSL（暗号化）接続のサポートを有効にして構築します。これには、OpenSSLパッケージがインストールされていなければなりません。configureは、処理を進める前にOpenSSLのインストールを確認するために、必要なヘッダファイルとライブラリを検査します。

--with-pam

PAM（プラグブル認証モジュール）のサポートを有効にして構築します。

--with-ldap

認証および接続パラメータ検索用のLDAPサポートを有効にして構築します。（[項31.17](#)および[項19.3.8](#)を参照してください。） Unixでは、OpenLDAPパッケージがインストールされていることが要求されます。WindowsではデフォルトのWinLDAPライブラリが使用されます。configureは、処理を進める前にOpenLDAPのインストールが十分されているかどうかを確認するために、必要なヘッダファイルとライブラリを検査します。

--without-readline

Readlineライブラリ（およびlibedit）の使用を防止します。これによりpsqlでのコマンドライン編集および履歴が無効となるため、推奨されません。

--with-libedit-preferred

GPLライセンスのReadlineではなくBSDライセンスのlibeditライブラリを優先して使用します。このオプションは両方のライブラリがインストールされている場合にのみ重要です。この場合デフォルトでReadlineが使用されます。

--with-bonjour

Bonjourサポートを有効にして構築します。これにはオペレーティングシステムがBonjourをサポートしていることが必要です。Mac OS Xでは推奨します。

--with-oss-p-uuid

ビルドにOSSP UUIDライブラリを使用します。具体的には、UUIDを生成する関数を提供する [uuid-oss](#) モジュールを構築します。

--with-libxml

libxmlを使用して構築します（SQL/XMLサポートが有効になります）。この機能のためにはlibxmlバージョン2.6.23以降が必要です。

libxmlがインストールするxml2-configプログラムを使用して、必要なコンパイラオプション、リンクオプションを検出することができます。PostgreSQLは、見つけれればこのプログラムを使用します。通常以外の場所にインストールしたlibxmlインストールを指定するためには、環境変数XML2_CONFIGがそのインストール用のxml2-configプログラムを指し示すように設定してください。または、--with-includesおよび--with-librariesを使用してください。

--with-libxslt

[xml2](#)モジュールを構築する場合はlibxsltを使用してください。xml2はXMLのXSL変換を行うために、このライブラリに依存します。

--disable-integer-datetimes

日付時刻および時間間隔の格納に64ビット整数格納方式を無効にし、その代わり浮動小数点で格納します。PostgreSQLリリース8.4以前では日付時刻の浮動小数点格納方式がデフォルトでしたが、timestamp値がとる値の全ての範囲でマイクロ秒精度をサポートしないため現在非推奨となりました。しかし、整数ベースの日付時刻格納には64-ビット整数型が必要です。従って、このオプションはこの型が使えない場合での使用、または以前のPostgreSQLバージョン用に書かれたアプリケーションとの互換性を保たなければならない場合に使用します。詳細は [項8.5](#)を参照してください。

--disable-float4-byval

float4値の"値"渡しを無効にし、"参照"渡しで渡すようにします。このオプションは性能に関するコストがかかりますが、C言語で開発された古いユーザ定義の関数との互換性を保持する場合や"version 0"呼び出し規約を使用する場合に必要になります。長期的に見てより良い解決方法はこうした関数を更新して、"version 1"呼び出し規約を使用するようにすることです。

--disable-float8-byval

float8値の"値"渡しを無効にし、"参照"渡しで渡すようにします。このオプションは性能に関するコストがかかりますが、C言語で開発された古いユーザ定義の関数との互換性を保持する場合や"version 0"呼び出し規約を使用する場合に必要になります。長期的に見てより良い解決方法はこうした関数を更新して、"version 1"呼び出し規約を使用するようにすることです。このオプションはfloat8のみだけに影響するのではなく、int8やタイムスタンプなど一部の関連した型についても影響することに注意してください。32ビットプラットフォームでは、--disable-float8-byvalがデフォルトで--enable-float8-byvalを選択することはできません。

--with-segsize=SEGSIZE

セグメントサイズをギガバイト単位で指定します。大規模なテーブルはこのセグメントサイズと同じサイズの複数のオペレーティングシステムのファイルに分割されます。これにより多くのプラットフォームで存在するファイルサイズ上限に関する問題を防ぎます。デフォルトのセグメントサイズは1ギガバイトで、サポートされるすべてのプラットフォームで安全です。使用するオペレーティングシステムが"ラージファイル"をサポートしていれば（最近ほとんどサポートしています）、より大きなセグメントサイズを使用することができます。非常に大規模なテーブルで作業する時のファイル記述子の消費数を減らすために、これが役に立つことができます。しかし、プラットフォーム、または使用予定のファイルシステムでサポートされる値以上に大きな値を指定しないように注意してください。tarなどの使用したい、その他のツールにも使用できるファイルサイズに制限があることがあります。絶対に必要ではありませんが、この値を2のべき乗にすることを勧めます。この値の変更にはinitdbが必要であることに注意してください。

--with-blocksize=BLOCKSIZE

キロバイト単位でブロック容量を設定します。これはテーブル内でのストレージとI/O単位です。8キロバイトのデフォルトはほとんどの場合適切ですが、特別な場合は他の値が役立ちます。値は1から32（キロバイト）の範囲の2のべき乗でなければなりません。この値の変更はinitdbを必要とすることを覚えておいてください。

--with-wal-segsize=SEGSIZE

メガバイト単位でWALセグメント容量を設定します。これはWALログ内のそれぞれ個別のファイルの容量です。この容量を調整することで、WALログ配送の粒度を制御するのに役立ちます。デフォルト容量は16メガバイトです。1から64（メガバイト）の範囲の2のべき乗でなければなりません。この値の変更はinitdbを必要とすることを覚えておいてください。

--with-wal-blocksize=BLOCKSIZE

キロバイト単位でWALブロック容量を設定します。これはこれはWALログ内でのストレージとI/O単位です。8キロバイトのデフォルトはほとんどの場合適切ですが、特別な場合は大きめの値が役立ちます。値は1から64（キロバイト）の範囲の2のべき乗でなければなりません。この値の変更はinitdbを必要とすることを覚えておいてください。

--disable-spinlocks

PostgreSQLがそのプラットフォーム用のCPUスピンのロックをサポートしない場合でも、構築に成功するようにします。スピンのロックのサポートの欠落により、性能は悪化します。したがって、このオプションは、構築が失敗し、その原因が使用するプラットフォームでスピンのロックサポートが欠落している場合にのみ使用してください。使用するプラットフォームにおけるPostgreSQLの構築にこのオプションが必要とされた場合は、PostgreSQL開発者にその問題を報告してください。

`--disable-thread-safety`

クライアントライブラリのスレッドセーフを無効にします。これにより、libpqやECPGプログラム内の同時実行スレッドは、安全にその固有の接続ハンドルを制御できなくなります。

`--with-system-tzdata=`*DIRECTORY*

PostgreSQLは、日付時刻に関する操作に必要な、独自の時間帯データベースを持ちます。実際のところ、この時間帯データベースはFreeBSD、Linux、Solarisなどの多くのオペレーティングシステムで提供される"zoneinfo"時間帯データベースと互換性があります。このため、これを再びインストールすることは冗長です。このオプションが使用されると、*DIRECTORY*にあるシステムが提供する時間帯データベースがPostgreSQLソース配布物に含まれるものの代わりに使用されます。*DIRECTORY*は絶対パスで指定しなければなりません。

`/usr/share/zoneinfo`がオペレーティングシステムの一部でよく使われます。インストール処理が時間帯データが一致しない、またはエラーがあることを検知しないことに注意してください。このオプションを使用する場合、指定した時間帯データがPostgreSQLで正しく動作するかどうかを検証するためにリグレーションテストを実行することが推奨されています。

このオプションの主な目的は、対象オペレーティングシステムを熟知しているパッケージ配布者向けです。このオプションを使用する大きな利点は、多くの局所的な夏時間規則の変更があってもPostgreSQLパッケージを更新する必要がないことです。他の利点として、時間帯データベースファイルをインストール時に構築する必要がありませんので、PostgreSQLのクロスコンパイルをより簡単に行うことができます。

`--without-zlib`

Zlibライブラリの使用を抑制します。これは、`pg_dump`と`pg_restore`における圧縮アーカイブのサポートを無効にします。このオプションは、このライブラリが利用できないごく少数のシステム向けだけのものです。

`--enable-debug`

全てのプログラムとライブラリをデバッグシンボル付きでコンパイルします。これは、問題を解析するためにデバッガ内のプログラムを実行できることを意味します。これはインストールする実行形式ファイルのサイズをかなり大きくし、また、GCC以外のコンパイラでは、通常はコンパイラによる最適化が行われなくなりますので、低速になります。しかし、デバッグシンボルが利用できるということは、発生した問題に対応する時に非常に便利です。現在のところ、GCCを使用している場合にのみ、稼働用のインストレーションにこのオプションを使用することを推奨します。しかし、開発作業時やベータ版を実行する時は、常にこれを有効にすべきです。

`--enable-coverage`

GCCを使用している場合、すべてのプログラムとライブラリはコードカバレッジ試験機構付きでコンパイルされます。実行すると、これらは構築用ディレクトリ内にコードカバレッジメトリックを持ったファイルを生成します。詳細は[項30.4](#)を参照してください。このオプションはGCC専用であり、また、開発作業中に使用するためのものです。

`--enable-profiling`

GCCを使用する場合、すべてのプログラムとライブラリがプロファイリング可能状態でコンパイルされます。バックエンドの終了時、プロファイリングに使用する`gmon.out`ファイルを含むサブディレクトリが作成されます。このオプションはGCCを使用する場合のみ使用でき、開発作業を行う時に使用します。

`--enable-cassert`

サーバにおける、多くの"あり得ない"状態をテストするアサーションチェックを有効にします。これは、プログラムの開発のためには測り知れない価値がありますが、このテストによりサーバはかなり低速になります。また、このテストを有効にすると、サーバの安定性を向上させるとは限りません！アサーションチェックは、重要度によって分類されていませんので、比較的危害がないようなバグでも、アサーション失敗をトリガとした、サーバの再起動が行われてしまいます。稼働用にこのオプションを使用することは推奨されませんが、開発作業時やベータ版を実行する場合は、これを有効にすべきです。

`--enable-depend`

自動依存関係追跡を有効にします。このオプションを使用すると、ヘッダファイルが変更された場合に、影響を受ける全てのオブジェクトファイルが再構築されるように、`makefile`が設定されます。これは開発作業時には有用ですが、単に一度コンパイルしインストールするだけであれば、これは無駄なオーバーヘッドです。現在のところ、GCCでのみ、このオプションは動作します。

--enable-dtrace

動的追跡ツールDTraceのサポートを有効にしてPostgreSQLをコンパイルします。より詳細な情報は[項27.4](#)を参照してください。

dtraceプログラムを指し示すためにDTRACE環境変数を設定することができます。dtraceは通常、検索パス内に存在しない可能性がある/usr/sbin以下にインストールされていますので、この設定はよく必要になります。

さらにdtraceプログラム用のコマンドラインオプションをDTRACEFLAGS環境変数で指定することができます。Solarisで64ビットバイナリでDTraceをサポートするには、DTRACEFLAGS="-64"をconfigureに指定してください。例えばGCCコンパイラを使用する場合は以下のようにします。

```
./configure CC='gcc -m64' --enable-dtrace DTRACEFLAGS='-64' ...
```

Sunのコンパイラを使用する場合は以下のようにします。

```
./configure CC='/opt/SUNWspro/bin/cc -xtarget=native64' --enable-dtrace DTRACEFLAGS='-64' ...
```

configureが選ぶものと違うCコンパイラを使いたいという場合には、CC環境変数をその使用したいプログラムに設定することができます。デフォルトでは、configureは利用できるのであればgccを、利用できなければプラットフォームのデフォルト（通常cc）を選択します。同様に、デフォルトのコンパイラフラグは必要に応じてCFLAGS変数で上書きすることもできます。

次のようにして、configureコマンドラインに環境変数を指定することができます。

```
./configure CC=/opt/bin/gcc CFLAGS='-O2 -pipe'
```

以下は、この方式で設定可能な重要な環境変数の一覧です。

BISON

Bisonプログラム。

CC

Cコンパイラ。

CFLAGS

Cコンパイラに渡すオプション。

CPP

Cプリプロセッサ。

CPPFLAGS

Cプリプロセッサに渡すオプション。

DTRACE

dtraceプログラムの場所。

DTRACEFLAGS

dtraceプログラムに渡すオプション。

FLEX

Flexプログラム。

LDFLAGS

実行ファイルや共有ライブラリにリンクする場合に使用するオプション。

LDFLAGS_EX

実行ファイルのリンク時のみに追加されるオプション。

LDFLAGS_SL

共有ライブラリのリンク時のみに追加されるオプション。

MSGFMT

多言語サポート（NLS）用のmsgfmtプログラム。

Perlインタプリタのフルパス。これは、PL/Perl構築に関する依存性を決定するために使用されます。

Pythonインタプリタのフルパス。これは、PL/Python構築に関する依存性を決定するために使用されます。またここでPython 2または3を指定するかどうかで(あるいは暗黙的に選択されます)、どちらのPL/Python言語が利用可能になるかも決まります。[項42.1](#)を参照してください。

Tclインタプリタのフルパスです。これは、PL/Tcl構築に関する依存性を決定するために使用され、Tclスクリプト内を置き換えます。

libxmlインストレーションの場所を特定するために使用するxml2-configプログラムです。

注意: サーバ内部のコード開発を行う場合、`--enable-cassert`(多数の実行時エラーチェックを有効にする)オプションと`--enable-debug`(デバッグツールの利便性を向上させる)オプションの使用を推奨します。

GCCを使う場合、少なくとも`-O1`レベルの最適化で構築することがベストです。なぜなら、何の最適化もしない(`-O0`)と、重要なコンパイル警告(初期化されていない変数の使用など)が無効になるからです。しかし、最適化を行うことでソースコードとコンパイルされたコードのステップは1対1とはなくなるため、デバッグは複雑になるかもしれません。最適化されたコードのデバッグに悩まされてしまう場合は、関心のある特定のファイルに対して`-O0`で再コンパイルしてください。これを実行するための簡単な方法は、`gmake PROFILE=-O0 file.o`のように、make経由でオプションを渡すことです。

1 11 構築

構築作業を開始するには、以下を入力してください。

```
gmake
```

(GNU makeを使用することを忘れないでください。) ハードウェアに依存しますが、構築作業には数分かかります。最後に以下のような行が表示されるはずです。

```
All of PostgreSQL is successfully made. Ready to install.
```

もし、ドキュメント(HTMLやman)や追加モジュール(`contrib`)を含め、構築可能なものを全てを構築したい場合、次の様に実施します。

```
gmake world
```

最終行は次のように表示されるはずです。

```
PostgreSQL, contrib and HTML documentation successfully made. Ready to install.
```

1 11 リグレーションテスト

インストールを行う前に、新しく構築したサーバをテストしたい場合、この時点でリグレーションテストを実行することができます。リグレーションテストとは、使用するマシンにおいてPostgreSQLが、開発者の想定通りに動作することを検証するためのテストのまとめです。次のように入力します。

```
gmake check
```

(これは `root` では動作しません。非特権ユーザとして実行してください。) [第30章](#)にはテスト結果の表示に関する詳しい情報があります。同じコマンドを入力することで、後にいつでもテストを繰り返すことができます。

1 11 ファイルのインストール

注意: もし既存のシステムのアップグレードをする場合、DBクラスタのアップグレードの解説が記載されている [項17.6](#)を参照してください。

PostgreSQLをインストールするには、以下を入力してください。

```
gmake install
```

これは、ファイルを[ステップ1](#)で指定されたディレクトリにインストールします。その領域に書き込むための権限を持っていることを確認してください。通常はこのステップはrootで行う必要があります。代わりに対象とするディレクトリを前もって作成し、適切に権限を調整することも可能です。

ドキュメント(HTMLやman)をインストールするには、以下を入力して下さい。

```
gmake install-docs
```

先にすべてを(worldを付けて)構築していた場合には、代わりに以下を実行してください。

```
gmake install-world
```

これによりドキュメントもインストールされます。

`gmake install`の代わりに`gmake install-strip`を使用することで、インストール時に実行可能ファイルやライブラリをストリップ(strip)することができます。これにより、多少の容量を節約できます。デバッグをサポートするように構築している場合でも、ストリップするとデバッグのサポートは実質、除去されてしまいます。したがって、これはデバッグが必要なくなった場合にのみ実行すべきです。`install-strip`は容量を節約するために適切な作業を行おうとしますが、実行可能ファイルから全ての不必要なバイトを完全にストリップすることはできません。可能な限りのディスク容量を全て節約したい場合は、手動で作業を行う必要があります。

この標準的なインストール方法では、クライアントアプリケーションの開発に必要なヘッダファイルと、Cで独自の関数やデータ型を作成するといったサーバ側のプログラムの開発用のヘッダファイルが用意されます (PostgreSQL 8.0より前までは、後で別途`gmake install-all-headers`コマンドが必要でした。しかし、この手順は標準のインストールに含まれるようになりました)。

クライアント側のみのインストール：. クライアントアプリケーションとインタフェースライブラリのみをインストールしたい場合、下記のコマンドを使います。

```
gmake -C src/bin install
gmake -C src/include install
gmake -C src/interfaces install
gmake -C doc install
```

`src/bin`にはサーバ用の数個のバイナリがあります。これらは小さなものです。

アンインストール：. インストールを取り消すには、`gmake uninstall` コマンドを使います。しかし、作成済みのディレクトリは削除されません。

クリーニング：. インストールが終わったら、`gmake clean` コマンドを使ってソースツリーから構築用のファイルを削除し、ディスク領域を解放することができます。`configure` プログラムが作るファイルは保持されますので、後で`gmake` コマンドですべてを再構築できます。ソースツリーを配布された時の状態に戻りたい場合は、`gmake distclean` コマンドを使います。同じソースツリー内で複数のプラットフォーム向けに構築する場合、これを実行して、それぞれのプラットフォームに対し再構成しなければなりません。(または、未変更のソースツリーを維持するために、各プラットフォームで別々の構築用ツリーを使用してください。)

構築作業を行った後で`configure`用オプションが間違っていることに気付いた場合や、`configure`の調査結果に何らかの変更を加えた場合(例えば、ソフトウェアのアップグレードなど)、再設定と再構築の前に`gmake distclean`を行うことをお勧めします。さもないと、設定選択肢の変更は、必要なところ全てには反映されない可能性があります。

[前のページ](#)
ソースの入手

[ホーム](#)
[上に戻る](#)

[次のページ](#)
インストール後の設定作業

15.5. インストール後の設定作業

15.5.1. 共有ライブラリ

共有ライブラリを持つ何らかのシステムの中には、新しくインストールされた共有ライブラリを探す場所をシステムに通知する必要があるものがあります。これが必要ではないシステムはFreeBSD、HP-UX、IRIX、Linux、NetBSD、OpenBSD、Tru64 UNIX（以前はDigital UNIX）、およびSolarisです。

共有ライブラリの検索パスを設定する方法は、プラットフォームによって異なります。しかし、最もよく使用される方法はLD_LIBRARY_PATHといった環境変数を以下のように設定することです。 Bourne シェル系（sh、ksh、bash、zsh）では、

```
LD_LIBRARY_PATH=/usr/local/pgsql/lib
export LD_LIBRARY_PATH
```

cshまたはtcshでは、以下のように設定します。

```
setenv LD_LIBRARY_PATH /usr/local/pgsql/lib
```

/usr/local/pgsql/libは**ステップ1**で--libdirに設定したものに置き換えてください。 /etc/profileや~/.bash_profileといったシェルの起動ファイルにこれらのコマンドを追加してください。この方法に関する警告についての優れた情報がhttp://xahlee.org/UnixResource_dir/_/ldpath.htmlにあります。

システムによっては、構築作業の前にLD_RUN_PATH環境変数を設定した方が良い場合があります。

Cygwinでは、ライブラリディレクトリをPATHに追加するか、.dll ファイルをbin/ディレクトリに移動します。

もし確信が持てない場合は、システムのマニュアルページ（おそらくld.soかrld）を参照してください。もし後に下記のようなメッセージが出たら、このステップが必要だったということです。

```
psql: error in loading shared libraries
libpq.so.2.1: cannot open shared object file: No such file or directory
```

この場合は処置を行ってください。

もしLinuxを使用していて、root権限があれば、

```
/sbin/ldconfig /usr/local/pgsql/lib
```

（または同等のディレクトリ）をインストール後に実行して、実行時リンクが共有ライブラリを素早く検索できるようにできます。より詳細についてはldconfigのマニュアルページを参照してください。FreeBSD、NetBSDおよびOpenBSDの場合のコマンドは以下の通りです。

```
/sbin/ldconfig -m /usr/local/pgsql/lib
```

同様なコマンドを持つ他のシステムは知られていません。

15.5.2. 環境変数

もし/usr/local/pgsqlか、もしくはデフォルトでプログラムが検索されない場所にインストールした場合、/usr/local/pgsql/bin（もしくは**ステップ1**で--bindirに設定した場所）をPATHに追加する必要があります。厳密に言えば、これは必要ではありません。しかし、これによってPostgreSQLの使用がずっと便利になります。

これを行うためには、以下を~/.bash_profile（もしくは、もし全てのユーザに反映したい場合は/etc/profile）のようなシェルの起動ファイルに追加してください。

```
PATH=/usr/local/pgsql/bin:$PATH
export PATH
```

cshやtcshを使用している場合は、以下のコマンドを使用してください。

```
set path = ( /usr/local/pgsql/bin $path )
```

デフォルトで検索される場所にインストールした場合を除き、システムがmanドキュメントを検索できるようにするためには、以下の行をシェルの起動ファイルに追加する必要があります。

```
MANPATH=/usr/local/pgsql/man:$MANPATH
export MANPATH
```

環境変数PGHOSTとPGPORTは、クライアントアプリケーションにデータベースサーバのホストとポートを指定し、コンパイル時に決定されたデフォルト値を無効にします。 クライアントアプリケーションをリモートで実行する場合、データベースを使用する予定の全てのユーザがPGHOSTを設定していると便利です。 しかしこれは必須ではありません。 この設定は、ほとんどのクライアントプログラムのコマンドラインオプションでも設定することができます。

[前のページ](#)

インストール手順

[ホーム](#)
[上に戻る](#)

[次のページ](#)

サポートされるプラットフォーム

15.6. サポートされるプラットフォーム

プラットフォーム（CPUアーキテクチャとオペレーティングシステムの組合せ）は、そのプラットフォーム上で動作する仕組みがコード内に存在し、かつ、そのプラットフォーム上で構築およびリグレーション試験に合格することが最近検証できた場合に、PostgreSQL開発者コミュニティによってサポートされたものとみなされます。現在、プラットフォームの互換性に関するほとんどの試験は[PostgreSQLビルドファーム](#)中の試験マシンによって自動的に行われます。ビルドファームに存在しないが、コードが動作するあるいは動作させることができたプラットフォームにおけるPostgreSQLの使用に興味のあるかたは、継続した互換性を確実にするために、ビルドファームのメンバマシンとして設定することを強く勧めます。

一般的に、PostgreSQLは、次のCPUアーキテクチャで動作できることを期待することができます。 x86, x86_64, IA64, PowerPC, PowerPC 64, S/390, S/390x, Sparc, Sparc 64, Alpha, ARM, MIPS, MIPSEL, M68K, PA-RISC。 M32R, NS32K, VAXをサポートするコードは存在しますが、これらのアーキテクチャで試験が行われたという報告は最近ありません。 `--disable-spinlocks` を付けることで、未サポートの種類のCPUでも構築することがよくできますが、性能は低下します。

PostgreSQLは次のオペレーティングシステムで動作することを期待することができます。 Linux (最近のディストリビューションすべて), Windows (Win2000 SP4以降), FreeBSD, OpenBSD, NetBSD, Mac OS X, AIX, HP/UX, IRIX, Solaris, Tru64 Unix, UnixWare。 他のUnixに似たシステムでも動作するかもしれませんが、最近試験されていません。 ほとんどの場合、指定されたオペレーティングシステムでサポートされるCPUアーキテクチャはすべて動作するでしょう。 特に古めのシステムを使用している場合、以下の[項15.7](#)を参照し、使用するオペレーティングシステム固有の情報がないか確認してください。

最近のビルドファームの結果でサポートしているものとされているプラットフォームでインストールに問題があった場合は、pgsql-bugs@postgresql.orgに報告してください。 新しいプラットフォームへのPostgreSQLの移植に興味があるのならば、pgsql-hackers@postgresql.orgがその議論に適しています。

15.7. プラットフォーム特有の覚書

本節はPostgreSQLのインストールと設定に関する追加のプラットフォーム固有の問題について説明します。インストール手順、特に[項15.2](#)を注意して読んでください。またリグレーション試験結果の解釈については[第30章](#)を確認してください。

ここで触れられていないプラットフォームは、インストールに関してプラットフォーム特有の問題がありません。

15.7.1. AIX

AIX上でPostgreSQLは動作しますが、適切にインストールをやり遂げるのは挑戦でもあります。AIXバージョン4.3.3から6.1はサポートされていると考えられます。GCCまたは在来のIBMコンパイラ`xlc`が使用できます。一般的に、最新のAIXとPostgreSQLバージョンを使用することが手助けになります。どのバージョンのAIXが動くと知られているかについての最新情報をビルドファームで見てください。

サポートされるAIXバージョンに対する最低限推奨される修正レベルを示します。

AIX 4.3.3

保守レベル 11 + ML11 バンドル以降

AIX 5.1

保守レベル 9 + ML9 バンドル以降

AIX 5.2

技術レベル 10 サービスパック 3

AIX 5.3

技術レベル 7

AIX 6.1

基本レベル

使用している修正レベルをチェックするには、AIX 4.3.3 から AIX 5.2 ML 7 までは`oslevel -r`、またはそれ以降のバージョンでは`oslevel -s`を使用します。

もし、`/usr/local`にインストール済のreadlineやライブラリがある場合は、次のconfigureフラグを加えて下さい。 `--with-includes=/usr/local/include --with-libraries=/usr/local/lib`

15.7.1.1. GCCの問題

AIX 5.3でGCCを使ってPostgreSQLがコンパイルし実行するためには幾つかの問題点があります。

特にパッケージ版を使用している場合、3.3.2以降のバージョンのGCCの使用が望まれるかもしれませんが、4.0.1ではうまくいきません。より早期のバージョンによる諸問題は、GCC自体の問題というよりもIBMがGCCをパッケージした方法に関連があるように見えます。従い、GCCを自身でコンパイルしている場合は、早期のバージョンのGCCを使用しても成功するかもしれません。

15.7.1.2. Unixドメインソケット破損

AIX 5.3には`sockaddr_storage`が十分大きく定義されていないという問題があります。バージョン5.3でIBMはUnixドメインソケットのアドレス構造体である`sockaddr_un`のサイズを増やしましたが、対応する`sockaddr_storage`のサイズを増やしませんでした。この結果、PostgreSQLでUnixドメインソケットを使用しようとすると、`libpq`でデータ構造体がオーバーフローするようになります。Unixドメインソケットを除くTCP/IP接続の動作は正常です。これはリグレーションテストの動作を妨害します。

この問題はIBMに報告済みでバグ報告PMR29657として記録されています。メンテナンスレベル5300-03以降に更新していれば、この修正が含まれています。すぐに解決させたいければ、`/usr/include/sys/socket.h`内で`_SS_MAXSIZE`を1025に変更してください。どちらの場合でも、ヘッダファイルを修正した後でPostgreSQLを再コンパイルしてください。

15.7.1.3. インターネットアドレス問題

PostgreSQLはlisten_addresses、pg_hba.conf、その他の中のIPアドレスを構文解釈するのにシステムのgetaddrinfo関数に頼ります。旧バージョンのAIXにはこの関数にさまざまなバグがあります。これら設定に関係した問題に遭遇した場合、上記に示されている適切な修正レベルへ更新することが問題解決になります。

あるユーザの報告です。

PostgreSQLバージョン8.1をAIX 5.3で実装している時、統計情報コレクタが"不思議なことに"正常に起動しないという問題が周期的に起こりました。これは、IPv6の実装における想定外の動作という結果で出現します。そのため、AIX5.3上ではPostgreSQLとIPv6が一緒にはうまく動作しないように見えました。

以下のいずれかの対応でこの問題が"解決できます"。

- localhostに対するIPv6アドレスを削除します。

```
(as root)
# ifconfig lo0 inet6 ::1/0 delete
```

- ネットサービスからIPv6を削除します。AIXの/etc/netsvc.confファイルは大まかに言ってSolaris/Linuxの/etc/nsswitch.confと同等です。AIXのデフォルトは、以下のようになっています。

```
hosts=local,bind
```

これを次で置き換えます。

```
hosts=local4,bind4
```

そうすると、IPv6アドレスの検索が解除されます。

警告
これは、実際には未熟なIPv6サポートに関連する問題の回避策です。AIX5.3のリリースから、IPv6サポートが目に見えて改善されています。この回避策はAIX5.3上で動きはしますが、この問題に対しての良い解決方法ではありません。AIX6.1ではIPv6サポートがさらに改善されたため、この対策は不必要になっただけでなく、問題を引き起こすことが報告されています。

15.7.1.4. メモリ管理

AIXはメモリ管理手法の観点から見ると多少独特です。 ギガバイト単位のRAMが空いているサーバがあっても、アプリケーションを実行している時にメモリ不足やアドレス空間エラーが発生することがあります。 こうした例の1つが、見慣れないエラーによるcreatelangの失敗です。 例えば、PostgreSQLインストールの所有者として実行してみます。

```
-bash-3.00$ createlang plperl template1
createlang: language installation failed: ERROR:  could not load library "/opt/dbs/pgsql748/lib/plperl.so": A
memory address is not in the address space for the process.
```

PostgreSQLインストールの処理グループ内の所有者以外として実行してみます。

```
-bash-3.00$ createlang plperl template1
createlang: language installation failed: ERROR:  could not load library "/opt/dbs/pgsql748/lib/plperl.so":
Bad address
```

他の実例は、PostgreSQLサーバログ中のメモリ不足エラーで、256 MB以上もしくはその近辺で全てのメモリ割り当てが失敗します。

これら問題のすべての総合原因は、サーバプロセスで使用されるデフォルトのビット割当とメモリモデルです。 デフォルトでは、AIXで構築されたすべてのバイナリは32ビットです。 これは使用中のハードウェアの種類やカーネルに依存しません。 これらの32ビットプロセスは、数個のモデルの1つを使用して256メガバイトのセグメントで割りつけられた4ギガバイトメモリに制限されます。 デフォルトでは、スタックで1つのセグメントとして共有されるものとしてヒープ内の256メガバイト未満の領域が許されます。

上記、createlang例の場合において、PostgreSQLインストールにおけるバイナリのumaskとパーミッションをチェックしてください。 例に関与したバイナリは32-ビットであり、755ではなく750モードでインストールされました。 このような形式で設定されたパーミッションのため、所有者もしくはグループ所有のメンバーのみライブラリを読み込みます。それは誰もが読み取り可能ではないため、ローダは、そうでない場合に配置される共有ライブラリセグメントにではなく、オブジェクトをプロセスのヒープに配置します。

これに対しての"理想的な"解決策はPostgreSQLの64-ビットビルドを使うことですが、32-ビットプロセッサのシステムでは64-ビットバイナリをビルドできませんが実行できないので、常に実務的ではありません。

32-ビットバイナリを要求する場合、PostgreSQLサーバを起動する前に`LDR_CNTRL=MAXDATA=0x00000000`に設定します。ここで、 $1 \leq n \leq 8$ です。そして異なる値と`postgresql.conf`設定で満足に稼動する構成を見つけ出します。この`LDR_CNTRL`使用するとAIXに対して、サーバがヒープにかかわらず、256 MBセグメントに割り当てられた`MAXDATA`バイトセットを持つようにさせたい意図を表明します。稼動する構成を見つけたとき、意図したヒープ容量をデフォルトで使用するよう`ldedit`を使用してバイナリを変更することができます。同じ効果を得るため、PostgreSQLは`configure LDFLAGS="-Wl,-bmaxdata:0x00000000"`を渡して再構築することもできます。

64-ビット構築に対し、`OBJECT_MODE`を64に設定し、`configure`に`CC="gcc -maix64"`と`LDFLAGS="-Wl,-bbigtoc"`を渡します。
(`xlc`に対するオプションは異なるかもしれません。) `OBJECT_MODE`の`export`を省略すると、構築はリンカエラーで失敗することがあります。`OBJECT_MODE`が設定された場合、`ar`、`as`、および`ld`のようなAIXの構築ユーティリティにどの種類のオブジェクトがデフォルトで対応されるのかを伝えます。

デフォルトで、ページングスペースのオーバーコミットが起こることがあります。これが起こることを経験したことはありませんが、AIXはメモリを使い切って、オーバーコミットがアクセスされたときにプロセスをkillします。システムが別のプロセスに対する十分なメモリがないことを判断したためにフォークが失敗するという、これとよく似たことは経験したことがあります。多くの他のAIX部分のように、ページングスペース割り当て方式とメモリ不足によるプロセス停止は、これが問題となるのであれば、システム全体またはプロセス全体を基準として設定可能です。

参照とリソース

"[Large Program Support](#)", AIX Documentation: General Programming Concepts: Writing and Debugging Programs.

"[Program Address Space Overview](#)", AIX Documentation: General Programming Concepts: Writing and Debugging Programs.

"[Performance Overview of the Virtual Memory Manager \(VMM\)](#)", AIX Documentation: Performance Management Guide.

"[Page Space Allocation](#)", AIX Documentation: Performance Management Guide.

"[Paging-space thresholds tuning](#)", AIX Documentation: Performance Management Guide.

[Developing and Porting C and C++ Applications on AIX](#), IBM Redbook.

15.7.2. Cygwin

Windowsに対するLinux的環境である、Cygwinを使ってPostgreSQLを構築することが可能です。しかし、この手法はWindowsネイティブビルド([第16章](#)を参照)には及ばないので、もはや推奨されません。

ソースから構築する場合、以下のCygwin特有の差異に注意し、通常のインストール手順に従って進めます（つまり、`./configure`; `make`; など）。

- Windowsユーティリティの前に使用するCygwinのbinディレクトリのパスを設定します。コンパイルにおける問題を回避する助けになります。
- GNU make コマンドは`gmake`ではなく、`make`と呼ばれます。
- `adduser` コマンドはサポートされていません。Windows NT、2000、またはXP上の適切なユーザ管理アプリケーションを使用してください。そうでなければ、この手順を飛ばします。
- `su` コマンドはサポートされていません。Windows NT、2000、またはXP上で`su`をシミュレートするため、`ssh`を使用します。そうでなければ、この手順を飛ばします。
- OpenSSLはサポートされていません。
- 共有メモリサポートのために`cygserver`を開始します。行うためには、コマンド`/usr/sbin/cygserver &`を入力します。このプログラムはPostgreSQLサーバを起動するとき、または(`initdb`で)データベースクラスタを初期化するときにはいつでも必要です。システム資源が欠けていることによるPostgreSQLの失敗を避けるため、デフォルトの`cygserver`設定は(例えば`SEMMNS`を増加させることで)変更される必要があります。
- いくつかのシステムでは、Cロケール以外を使っている場合に構築が失敗するかもしれません。これに対処するために

は、構築前に`export LANG=C.utf8`を実施してロケールをCに設定し、PostgreSQLのインストール後に以前の設定に戻してください。

- 並行リグレーションテスト (`make check`) は、接続拒絶エラーやハングアップを引き起こす`listen()`バックログキューのオーバーフローにより、誤ったリグレーションテストの失敗を生成する可能性があります。`make 変数MAX_CONNECTIONS`を使用して、最大接続数を制限できます。つまり次のようにします。

```
make MAX_CONNECTIONS=5 check
```

(いくつかのシステムでは、同時接続を10まで広げられます。)

Windows NTサービスとして`cygserver`とPostgreSQLサーバをインストールすることができます。これを実現する方法は、CygwinのPostgreSQLバイナリパッケージに含まれる`README`文書を参照してください。それは`/usr/share/doc/Cygwin`ディレクトリにインストールされます。

15.7.3. HP-UX

PostgreSQL 7.3以上は、適切なシステムパッチレベルと構築ツールが与えられているとして、HP-UX 10.Xまたは11.Xが稼動するSeries 700/800 PA-RISCコンピュータで動作します。少なくとも1つの開発者が定期的にHP-UX10.20で試験を行っており、HP-UX11.00と11.11上へのインストールが成功していることの報告を受けています。

PostgreSQLのソース配布物は別として、GNU make (HPのmakeは通りません) と、GCCもしくはHPのANSI Cコンパイラの全部いづれかが必要です。配布物tarボールではなくGitソースから構築を行う時は、Flex (GNU lex) とBison (GNU yacc) も必要です。同時に、正確に最新のHPパッチが当たっていることの確認を推奨します。HP-UX 11.11上で64ビット構築を行うのであれば最小限、PHSS_30966 (11.11)またはその後継パッチが必要です。さもないと、`initdb`がハングアップします。以下を参考にしてください。

PHSS_30966 s700_800 ld(1) and linker tools cumulative patch

一般原則として、HPのCコンパイラを使用する場合、`libc`と`ld/dld`パッチとコンパイラパッチが最新版でなければなりません。それらの最新パッチの無償コピーについては、<http://itrc.hp.com>および <ftp://us-ffs.external.hp.com/>のHPサポートサイトを見てください。

PA-RISC 2.0マシン上でGCCを使用して64-ビットバイナリを構築したい場合、GCC 64-ビット版を使用しなければなりません。HP-UX PA-RISCとItanium用のバイナリは<http://www.hp.com/go/gcc>から入手できます。同時に`binutils`も入手しインストールすることを忘れないでください。

PA-RISC 2.0装置上での構築で、PA-RISC 1.1装置で稼動するコンパイル済みのバイナリが必要な場合、`CFLAGS`で`+DAPortable`を指定する必要があります。

HP-UX Itaniumマシン上で構築するのであれば、依存するパッチまたはその後継のパッチが当たった最新のHP ANSI Cコンパイラが必要です。以下です。

PHSS_30848 s700_800 HP C Compiler (A.05.57)

PHSS_30849 s700_800 u2comp/be/plugin library Patch

HPのCコンパイラとGCCの2つが存在するとき、`configure`実行時に明示的に使用するコンパイラを指定したい場合、HPのCコンパイラでは、

```
./configure CC=cc
```

GCCの場合は、

```
./configure CC=gcc
```

のようにします。この設定を省略すると、選択できる場合には`configure`は`gcc`を選びます。

デフォルトのインストール対象配置場所は`/usr/local/pgsql`で、`/opt`のような場所に変更したい場合もあります。そのような時は、`configure`で`--prefix`スイッチを使用します。

リグレーションテストにおいて、幾何試験で下位ビット桁の差異があるかもしれません。これはどのコンパイラを使用しているか、使用している数学ライブラリのバージョンは何かに依存し、変化します。その他のエラーは原因を疑ってください。

15.7.4. IRIX

PostgreSQLは無事に、MIPSProコンパイラのバージョン 7.30、7.3.1.2m、7.3、および7.4.4mが有効なIRIX 6.5.5m、6.5.12、6.5.13、および6.5.26が稼動するMIPS r8000、r10000 (ip25とip27の双方)、およびr12000 (ip35) プロセッサでの実行が報告されています。

MIPSProの完全なANSI C コンパイラが必要です。GCCで構築しようとするとき問題があります。それは、ある種類の構造を返す関数の使用に関連した、知られているGCCのバグ（バージョン3.0時点で修正されていない）です。バグは`inet_ntoa`、`inet_lnaof`、`inet_netof`、`inet_makeaddr`、および`semctl`のような関数に影響します。libgccでのそれらの関数にコードのリンクを強制して修正ができそうですが、未だ検証されていません。

MIPSProコンパイラのバージョン7.4.1mは不正なコードを生成することが知られています。症状はデータベースを開始する際、"invalid primary checkpoint record（無効な主チェックポイントレコード）"となることです。バージョン7.4.4mは大丈夫ですが、その中間にあるバージョンは不明確です。

次のようなコンパイル問題があるかもしれません。

```
cc-1020 cc: ERROR File = pqcomm.c, Line = 427
The identifier "TCP_NODELAY" is undefined.

if (setsockopt(port->sock, IPPROTO_TCP, TCP_NODELAY,
```

幾つかのバージョンはTCP定義を`sys/xti.h`に含みますので、`src/backend/libpq/pqcomm.c`と`src/interfaces/libpq/fe-connect.c`に`#include <sys/xti.h>`を加える必要があります。これに遭遇した場合、報告してもらえると適切な修正を開発できます。

リグレーションテストの中で、これは使用しているFPUに依存し、幾何試験で下位ビット桁の差異があるかもしれません。その他のエラーは原因を疑ってください。

15.7.5. MinGW/ ネイティブWindows

Windows用PostgreSQLは、Microsoftオペレーティングシステム用のUnixに似た構築環境であるMinGW、またはMicrosoftのVisual C++コンパイラ一式を使って構築できます。MinGW版の構築は本章で記述されている通常の構築システムを使用します。Visual C++構築は、[第16章](#)で記述するようにまったく異なった動作をします。それは完全にネイティブな構築で、MinGWのような追加ソフトウェアを使用しません。既製のインストーラがPostgreSQLのメインウェブサイトから入手できます。

ネイティブに移植されたWindows版ではWindows 2000以降の32ビットまたは64ビット版が必要です。これより前のオペレーティングシステムには十分な構造基盤がありません（ただし、Cygwinはそれら上で使える可能性があります）。Unixに似た構築ツールであるMinGWと、`configure`のようなシェルスクリプトを実行するために必要なUnixツール群であるMSYSは、<http://www.mingw.org/>からダウンロード可能です。作成されたバイナリの実行にはいづれも必要ありません。バイナリの作成のためのみ必要です。

MinGWを使って64ビット版バイナリをビルドするためには、<http://mingw-w64.sourceforge.net/>から64ビット用のツールを入手してインストールし、`PATH`にある`bin`ディレクトリへそれらを入れ、そして`--host=x86_64-w64-mingw`オプション付きで`configure`を実施します。

MSYSコンソールはバッファリングに問題があるので、すべてをインストールした後に`CMD.EXE`下で`psql`を実行することを推奨します。

15.7.5.1. Windows上でのクラッシュダンプの収集

もしWindows上のPostgreSQLがクラッシュした場合、Unixにおけるコアダンプと似た、クラッシュの原因を追跡するために使用できるminidumpsを生成することができます。このダンプはWindows Debugger ToolsやVisual Studioを使うことで解析できます。Windowsにてダンプを生成できるように、`crashdumps`という名前のサブディレクトリをデータベースクラスタディレクトリの中に作成します。ダンプは、クラッシュ時の現在時間と原因となったプロセスの識別子を元にした一意な名前としてこのディレクトリの中に生成されます。

15.7.6. SCO OpenServer と SCO UnixWare

PostgreSQLはSCO UnixWare 7 と SCO OpenServer 5上で構築できます。OpenServerについて、OpenServer Development Kit または the Universal Development Kitのいずれかが使用できます。しかし、以下に記載するような微調整が必要です。

15.7.6.1. Skunkware

SCO Skunkware CDのコピーがどこにあるかを知る必要があります。Skunkware CDはUnixWare 7と現在バージョンのOpenServer 5に含まれています。Skunkwareには、インターネットから入手可能な数多くのよく知られたプログラムをすぐインストールできるように準備したバージョンが含まれます。例えば、gzip、gunzip、GNU Make、FlexおよびBisonはすべて含まれています。UnixWare 7.1では、このCDは現在"Open License Software Supplement"のように名称付けられています。このCDを持っていない場合、その中にあるソフトウェアは<http://www.sco.com/skunkware/>から入手できます。

SkunkwareはUnixWare と OpenServer用に異なったバージョンがあります。以下に注意書きされている点を除き、オペレーティングシステムに対して適切なバージョンをインストールしたことを確認してください。

UnixWare 7.1.3以降では、UDK CDの中にGCCコンパイラとGNU Makeが含まれます。

15.7.6.2. GNU Make

GNU Makeプログラムを使用する必要があり、それはSkunkware CDの中にあります。デフォルトでは`/usr/local/bin/make`としてインストールします。SCO `make` プログラムとの混乱を避けるため、GNU `make`を`gmake`と名称変更した方が便利かもしれません。

UnixWare 7.1.3およびそれ以上で、GNU MakeプログラムはUDK CDのOSTK部分で、`/usr/gnu/bin/gmake`にあります。

15.7.6.3. Readline

ReadlineライブラリはSkunkware CDにあります。しかし、UnixWare 7.1 Skunkware CDにはありません。もしUnixWare 7.0.0 または 7.0.1 Skunkware CDを所有していれば、そこからインストールできます。そうでない場合、<http://www.sco.com/skunkware/>を試してください。

デフォルトでReadlineは`/usr/local/lib` と `/usr/local/include`にインストールされます。しかし、PostgreSQLの`configure` プログラムは援助無しにそこにあることを見つけません。Readlineをインストールしたなら、`configure`で以下のオプションを使ってください。

```
./configure --with-libraries=/usr/local/lib --with-includes=/usr/local/include
```

15.7.6.4. OpenServer上でUDKを使用

もしOpenServer上で新規のUniversal Development Kit (UDK) コンパイラを使用しているのであれば、以下のようにUDKライブラリの場所を指定する必要があります。

```
./configure --with-libraries=/udk/usr/lib --with-includes=/udk/usr/include
```

Putting these together with the Readline options from above:

```
./configure --with-libraries="/udk/usr/lib /usr/local/lib" --with-includes="/udk/usr/include  
/usr/local/include"
```

15.7.6.5. PostgreSQLマニュアルページを読む

デフォルトで、PostgreSQLマニュアルページは`/usr/local/pgsql/man`にインストールされます。デフォルトで、UnixWareはマニュアルページとしてその場所を見ません。マニュアルページを読めるようにするには、以下の例のように、`/etc/default/man`の`MANPATH`変数を変更します。

```
MANPATH=/usr/lib/scohelp/%L/man:/usr/dt/man:/usr/man:/usr/share/man:scohelp:/usr/local/man:/usr/local/pgsql/man
```

OpenServerについてマニュアルページが使用できるようにするため幾つかの追加研究が注ぎ込まれる必要があります。その理由はマニュアルシステムは他のプラットフォームとは多少異なるからです。現在、PostgreSQLはそれら全てをインストールしません。

15.7.6.6. 7.1.1b機能追加でのC99問題

7.1.1b機能追加を含む、OpenUNIX 8.0.0 (UnixWare 7.1.2) と共にリリースされたものより早期のコンパイラでは、`CFLAGS`または`CC`環境変数で`-xb`を指定する必要があります。この兆候は、インライン関数を参照する`tuplesort.c`をコンパイルするエラーです。明らかに、これは7.1.2(8.0.0)とそれ以降のコンパイラで変更されています。

15.7.6.7. UnixWare上のスレッド

スレッド処理のためにはすべての`libpq`を使用するプログラムにて`-Kpthread`を使用する必要があります。`libpq`は`pthread_*`呼び出しを使用しますが、`-Kpthread/-Kthread`フラグを伴わないと有効ではありません。

15.7.7. Solaris

PostgreSQLはSolaris上でとても良くサポートされています。オペレーティングシステムが更新されればされる程、問題点の遭遇は少なくなります。以下に詳細を示します。

15.7.7.1. 必要なツール

GCCもしくはSunのコンパイラ式により構築できます。より良いコード最適化のため、SunのコンパイラはSPARC構造上においては、強く推奨されます。GCC 2.95.1を使用する場合に問題があったと報告を受けています。GCC 2.95.3以降を勧めます。Sunのコンパイラを使用するのであれば、`/usr/ucb/cc`を選択せず、`/opt/SUNWsprow/bin/cc`を使用するように注意してください。

<http://developers.sun.com/sunstudio/downloads/>からSun Studioをダウンロードできます。数多くのGNUツールがSolaris 10に統合、もしくはSolaris companion CDの中にあります。Solarisのより古いバージョンに対するパッケージを好むのであれば、それらのツールは<http://www.sunfreeware.com>、もしくは<http://www.blastwave.org>にあります。ソースの方が良いという方は<http://www.gnu.org/order/ftp.html>を参照してください。

15.7.7.2. OpenSSLでの問題点

OpenSSLサポート付きでPostgreSQLを構築するとき、以下に示したファイルでコンパイルエラーを受け取ることがあります。

- `src/backend/libpq/crypt.c`
- `src/backend/libpq/password.c`
- `src/interfaces/libpq/fe-auth.c`
- `src/interfaces/libpq/fe-connect.c`

これは標準`/usr/include/crypt.h`ヘッダファイルとOpenSSLから供給されるヘッダファイル間での名前空間衝突によるものです。

OpenSSLインストレーションを0.9.6aに更新することでこの問題は解決されます。Solaris 9とそれ以降のバージョンは、より新しいOpenSSLバージョンを持っています。

15.7.7.3. 失敗したテストプログラムについてconfigureが出すエラー

もしconfigureが失敗したテストプログラムについてエラーを出す場合、おそらく実行時のリンクがlibz、libreadline、またはlibsslのような非標準のライブラリを見つけ出せないことによりです。それを正しい場所に指し示すため、configureコマンドラインでLDFLAGS環境変数を以下のように設定します。

```
configure ... LDFLAGS="-R /usr/sfw/lib:/opt/sfw/lib:/usr/local/lib"
```

より詳細な情報はldマニュアルページを参照ください。

15.7.7.4. 時折クラッシュする64-ビット構築

Solaris 7以前の64bit版のlibcには不具合のあるvsnprintfルーチンがあり、それはPostgreSQLのエラーコアダンプに繋がります。既知の最も単純な回避策は、ライブラリのコピーではなく、自身のvsnprintfバージョンを使うように仕向けることです。これを行うためには、configureを実行した後、configureで生成されたファイルを編集します。つまり、`src/Makefile.global`の以下の行

```
LIBOBJJS =
```

を次のように変更します。

```
LIBOBJJS = snprintf.o
```

(この変数に既に他のファイルが列挙されているかもしれませんが。その順序は関係ありません。) その後、通常通りに構築してください。

15.7.7.5. 最適性能を得るためのコンパイル

SPARCアーキテクチャにおけるコンパイルでは、Sun Studioを強く推奨します。特筆するような速さのバイナリを生成するため、`-xO5`最適化フラグを使用してみてください。浮動小数点演算と、(`-fast`のような) `errno`演算を修正するようなフラグはすべて使ってはいけません。それらのフラグは、例えばdate/time演算において、PostgreSQLの標準ではない動作をさせることがあります。

SPARCで64ビットバイナリを使用する理由がないのであれば、32ビット版を選択してください。64ビット操作はより遅く、64ビットバイナリは32ビット版より遅いのです。一方で、AMD64 CPUファミリ上の32ビットコードはネイティブではありません。これが、このCPUファミリで32ビットコードが非常に低速な理由です

性能に関してPostgreSQLとSolarisを調整する秘訣がいくつ

かhttp://www.sun.com/servers/coolthreads/tnb/applications_postgresql.jspにあります。この論文はT2000プラット

フォームに主として焦点が当てられていますが、Solarisを使用するその他のハードウェア上でも同様に有用なアドバイスになるでしょう。

15.7.7.6. PostgreSQLをトレースするためのDTrace使用

そのとおりです。DTraceを使うことができます。より詳細な情報は [項27.4](#)を参照してください。より多くの情報がhttp://blogs.sun.com/robertlor/entry/user_level_dtrace_probes_in文書にあります。

以下のようなエラーメッセージでpostgres実行形式のリンクが中断することを体験した場合、そのDTraceインストールが静的関数におけるプローブを扱うには古すぎると言うことです。Solaris 10u4もしくはそれより新しいものが必要です。

```
Undefined          first referenced
  symbol            in file
AbortTransaction   utils/probes.o
CommitTransaction  utils/probes.o
ld: fatal: Symbol referencing errors. No output written to postgres
collect2: ld returned 1 exit status
gmake: *** [postgres] Error 1
```


第 16章Windowsにおけるソースコードからのインストール

目次

16.1. [Visual C++またはMicrosoft Windows SDKを使用した構築](#)

- 16.1.1. [必要条件](#)
- 16.1.2. [64ビット版のWindowsにおける特別な考慮事項](#)
- 16.1.3. [構築](#)
- 16.1.4. [整理およびインストール](#)
- 16.1.5. [リグレーションテストの実行](#)
- 16.1.6. [文書の構築](#)

16.2. [Visual C++またはBorland C++を使用したlibpqの構築](#)

ほとんどのユーザには、PostgreSQLウェブサイトからワンクリックインストーラパッケージとして入手可能なWindows用のバイナリ配布物をダウンロードすることを推奨します。ソースからの構築はPostgreSQLそのもの、もしくはその拡張の開発者のみを対象としています。

WindowsでPostgreSQLを構築する方法は複数存在します。Microsoftのツールを使用した最も単純な構築方法では、Visual Studio Express 2012 for Windows Desktopをインストールし、それに含まれるコンパイラをインストールします。また完全なMicrosoft Visual C++ 2005, 2008 or 2010を使用しても構築することができます。コンパイラの他にWindows SDKのインストールが必要となる場合があります。

また、MinGWで提供されるGNUコンパイラツール、または、古めのWindowsではCygwinを使用してPostgreSQLを構築することができます。

最後に、クライアントアクセスライブラリ (libpq) は、Visual C++ 7.1またはBorland C++を使用して構築された静的リンクのアプリケーションとの互換性のため、Visual C++ 7.1またはBorland C++を用いて構築することができます。

MinGWまたはCygwinを使用した構築では、通常の構築システムを使用します。 [第15章](#)、[項15.7.5](#)および[項15.7.2](#)にある固有の注記を参照してください。これらの環境で64ビットバイナリを生成するためにはMinGW-w64のツールを使用します。また、これらのツールは32ビットと64ビットWindows上で、LinuxやDarwinなどの他のホストを対象としたクロスコンパイルをする際にも使用されます。Cygwinは商用サーバでの稼働は推奨されません。これは、Windows 98などのネイティブな構築ができない古めのバージョンのWindowsでの使用に限定してください。MinGWは、それを使用して他のモジュールを構築したい場合にのみ推奨します。公式のバイナリはVisual Studioを使用して構築しています。

psqlのネイティブな構築はコマンドライン編集をサポートしていません。Cygwinによる構築はコマンドライン編集をサポートしているので、Windows上でインタラクティブなpsqlの振る舞いが必要になる場合はこちらを使うべきでしょう。

16.1. Visual C++またはMicrosoft Windows SDKを使用した構築

Microsoftが提供するVisual C++コンパイラスイートを使用してPostgreSQLを構築することができます。これらのコンパイラはVisual Studio、Visual Studio Express、いくつかのバージョンのMicrosoft Windows SDKにあります。すでにVisual Studio環境が設定されているのであれば、Microsoftから無料でダウンロードできるWindows SDK 7.1やその他のVisual Studio Express 2012 for Windows Desktopのコンパイラを使用することが最も簡単な方法です。

PostgreSQLはVisual Studio 2005からVisual Studio 2012(Expressエディション含む)までに付属のコンパイラをサポートします。単体のWindows SDKの6.0から7.1までについても同様です。64ビットPostgreSQLの構築はMicrosoft Windows SDKのバージョン6.0aから7.1、もしくはVisual Studio 2008かそれ以上のバージョンでのみサポートしています。

Visual C++を使用して構築するためのツールがsrc/tools/msvcディレクトリに存在します。構築する際に、MinGWやCygwinに付属するツールがシステムPATHに存在しないことを確認してください。また、PATH上で必要なVisual C++ツールがすべて利用可能になっていることを確認してください。Visual Studioでは、Visual Studio コマンドプロンプトを起動してください。64ビット版を構築したい場合には、それぞれ64ビット版のコマンドを使用する必要があります。Microsoft Windows SDKでは、SDKのスタートメニューにあるCMD shellを起動してください。最近のSDKでは、setenvを使うことで対象のCPUアーキテクチャを変更することができます。すべてのコマンドはsrc/tools/msvcディレクトリから実行しなければなりません。

構築する前に、config.pl ファイルを編集して、変更したい設定オプションや使用する何らかのサードパーティ製のライブラリのパスを反映させる必要があるかもしれません。まずconfig_default.pl ファイルを読み取り、解析することから始まり、そしてconfig.pl内の何らかの変更が適用されて、すべての設定が決定されます。例えば、Pythonインストールの場所を指定する場合、以下をconfig.plに記載します。

```
$config->{python} = 'c:\python26';
```

config_default.pl内の指定と異なるパラメータのみを指定する必要があります。

何か他に環境変数を設定する必要があるあれば、buildenv.pl という名前のファイルを作成し、そこに必要なコマンドを記載してください。たとえば、PATHにbison用のパスを追加したいのであれば、以下を含むファイルを作成してください。

```
$ENV{PATH}=$ENV{PATH} . 'c:\some\where\bison\bin';
```

16.1.1. 必要条件

PostgreSQLを構築するには以下の追加製品が必要です。config.pl ファイルを使用してライブラリを利用できるディレクトリを指定してください。

Microsoft Windows SDK

もしあなたの構築環境がサポートされているMicrosoft Windows SDKのバージョンを搭載していないのであれば、入手可能な最新版(現在はバージョン7.1)までアップグレードすることを推奨します。

<http://www.microsoft.com/downloads/>からダウンロードできます。

SDKのWindows Headers and Librariesを常にインクルードしなければなりません。Visual C++ Compilersに含まれるWindows SDKをインストールしている場合、構築のためにVisual Studioは必要ありません。バージョン8.0aでは、Windows SDKは完全なコマンドライン構築環境を提供していないことに注意してください。

ActiveState Perl

ActiveState Perlが構築生成スクリプトを実行するために必要です。MinGWまたはCygwinのPerlでは動作しません。また、PATH内に含まれていなければなりません。<http://www.activestate.com>からバイナリをダウンロードできます (注意: バージョン5.8以降が必要です。フリー版の標準配布で十分です)。

使用できるようになることが目的であれば以下の追加製品は必要ありませんが、完全なパッケージを構築する場合には必要です。config.plを使用してライブラリが利用できるディレクトリを指定してください。

ActiveState TCL

PL/TCLを構築する時に必要です (注意: バージョン8.4が必要です。フリー版の標準配布で十分です)。

BisonおよびFlex

Gitから構築する場合はBisonおよびFlexが必要です。リリースファイルから構築する場合は不要です。Bison 1.875ま

たはバージョン2.2以降のみで動作することに注意してください。 またFlex 2.5.31以降が必要です。 Bisonは<http://gnuwin32.sourceforge.net>からダウンロードすることができます。 Flexは<http://www.postgresql.org/ftp/misc/winflex/>からダウンロードすることができます。 もしPostgreSQLのGitリポジトリへのアクセスにmsysGitを使用している場合、 あなたのGitのバイナリディレクトリには、既に最近のバージョンのBisonやFlexがあるでしょう。

注意: GnuWin32からのBisonディストリビューションでは、C:\Program Files\GnuWin32の様に名前に空白を持つディレクトリにインストールされると正常に機能しないというバグがあります。 代わりにC:\GnuWin32へインストールすることを検討してください。

Diff

リグレーションテストを実行するにはdiffが必要です。 <http://gnuwin32.sourceforge.net>からダウンロードできます。

Gettext

NLSサポート付きで構築する場合はgettextが必要です。 <http://gnuwin32.sourceforge.net>からダウンロードできます。 バイナリ、依存物、開発用ファイルすべてが必要であることに注意してください。

MIT Kerberos

Kerberos認証をサポートする場合に必要です。 MIT Kerberosは<http://web.mit.edu/Kerberos/dist/index.html>からダウンロードできます。

libxml2およびlibxslt

XMLサポートのために必要です。 バイナリは<http://zlatkovic.com/pub/libxml>から、ソースは<http://xmlsoft.org>からダウンロードできます。 libxml2はiconvを必要することに注意してください。 同じ場所からダウンロードできます。

openssl

SSLサポートのために必要です。 バイナリは<http://www.slproweb.com/products/Win32OpenSSL.html>から、ソースは<http://www.openssl.org>からダウンロードできます。

ossp-uuid

UUID-OSSPサポート (contribのみ) が必要です。 ソースは<http://www.ossfp.org/pkg/lib/uuid/>にあります。

Python

PL/Pythonを構築する場合に必要です。 バイナリは<http://www.python.org>からダウンロードできます。

zlib

pg_dumpおよびpg_restoreにおける圧縮をサポートするために必要です。 バイナリは<http://www.zlib.net>からダウンロードできます。

16.1.2. 64ビット版のWindowsにおける特別な考慮事項

64ビット版Windowsにおいてx64アーキテクチャのみでPostgreSQLを構築することができます。 Itaniumプロセッサをサポートしていません。

同じ構築用ツリーで32ビット版と64ビット版を混在させることはサポートされません。 構築システムは32ビット環境で動作しているか64ビット環境で動作しているかを自動的に検出し、それにしたがってPostgreSQLを構築します。 このため構築作業を始める前に正しいコマンドプロンプトを実行させることが重要です。

pythonやopensslなどのサーバサイドのサードパーティ製ライブラリを使用するためには、ライブラリも64ビット版である必要があります。 64ビット版のサーバで32ビット版のライブラリをロードすることはサポートされていません。 PostgreSQLがサポートするサードパーティ製のライブラリで32ビット版しか利用できないものが複数あります。 こうした場合、64ビット版のPostgreSQLで使用することはできません。

16.1.3. 構築

リリース条件 (デフォルト) でPostgreSQLをすべて構築するためには、以下のコマンドを実行してください。

```
build
```

デバッグ条件でPostgreSQLをすべて構築するためには、以下のコマンドを実行してください。

```
build DEBUG
```

単一のプロジェクトのみを構築するためには、たとえばpsqlであれば、以下のコマンドを実行してください。

```
build psql
build DEBUG psql
```

デバッグのためにデフォルトの構築条件を変更するためには、以下をbuildenv.plファイルに記載してください。

```
$ENV{CONFIG}="Debug";
```

また、Visual Studio GUI内から構築することも可能です。この場合はコマンドプロンプトから以下を実行しなければなりません。

```
perl mkvcbuild.pl
```

その後に、生成されたpgsql.sln（ソースツリーのトップディレクトリに存在します。）をVisual Studioで開いてください。

16.1.4. 整理およびインストール

ほとんどの場合、Visual Studioの持つ自動依存関係追跡により変更されたファイルが扱われます。しかし、大規模な変更が行われた場合、インストレーションを整理する必要があるかもしれません。このためには、clean.batコマンドを実行してください。これにより、生成されたファイルがすべて自動的に消去されます。また、distパラメータを使用して実行することも可能です。この場合、make distcleanのように振舞い、flex/bisonの出力ファイルも削除します。

デフォルトで、すべてのファイルはdebugまたはreleaseディレクトリ以下のサブディレクトリに書き出されます。これらのファイルを標準レイアウトでインストールし、データベースの初期化や使用に必要なファイルを生成するためには、以下のコマンドを実行してください。

```
install c:%destination%directory
```

16.1.5. リグレッションテストの実行

リグレッションテストを実行するためには、まず必要なすべての部品の構築が完了していることを確認してください。また、システムのすべての部品で必要とするDLL（手続き言語用のPerlのDLLやPythonのDLLなど）がシステムパスに含まれていることを確認してください。もし含まれていなければ、buildenv.plファイルを介して設定してください。試験を実行するためには、以下のコマンドのいずれかをsrc%tools%msvcディレクトリから実行してください。

```
vc regress check
vc regress installcheck
vc regress plcheck
vc regress contribcheck
```

使用するスケジュール（デフォルトはparallelです）を変更するためには、コマンドラインに以下のように追加してください。

```
vc regress check serial
```

リグレッションテストの詳細については[第30章](#)を参照してください。

16.1.6. 文書の構築

PostgreSQLの文書をHTML書式で構築するためには、いくつかツールやファイルが必要です。これらのファイルをすべて格納するためのトップディレクトリを作成し、以下のリストに記載したサブディレクトリに格納してください。

OpenJade 1.3.1-2

http://sourceforge.net/projects/openjade/files/openjade/1.3.1/openjade-1_3_1-2-bin.zip/downloadからダウンロードし、openjade-1.3.1サブディレクトリで伸長してください。

DocBook DTD 4.2

<http://www.oasis-open.org/docbook/sgml/4.2/docbook-4.2.zip>からダウンロードし、docbookサブディレクトリで伸長してください。

DocBook DSSSL 1.79

<http://sourceforge.net/projects/docbook/files/docbook-dsssl/1.79/docbook-dsssl-1.79.zip/download>からダウンロードし、docbook-dsssl-1.79サブディレクトリで伸長してください。

<http://www.oasis-open.org/cover/ISOEnts.zip>からダウンロードし、docbookサブディレクトリで伸長してください。

builddoc.pl ファイルを編集し、このトップディレクトリの場所を示す変数を追加してください。 例を示します。

```
$ENV{DOCRROOT}='c:¥docbook';
```

文書を構築するためには、builddoc.bat コマンドを実行してください。 これは実際にはインデックスを作成するために、2回構築を行うことに注意してください。 生成されたHTMLファイルはdoc¥src¥sgml に格納されます。

前のページ Windowsにおけるソースコードからのインストール	ホーム 上に戻る	次のページ Visual C++またはBorland C++を使用したlibpqの構築

16.2. Visual C++またはBorland C++を使用したlibpqの構築

異なるデバッグ用フラグやリリース用フラグを付けたバージョンを必要とする場合や静的リンクでアプリケーションとリンクする必要がある場合に限って、libpqを構築する際にVisual C++ 7.1-9.0またはBorland C++を使用することが推奨されます。普通に使用するのであれば、MinGW、Visual Studio、Windows SDKによる方法を推奨します。

Visual Studio 7.1以降を使用してlibpqクライアントライブラリを構築するためには、srcディレクトリに移動し、以下のコマンドを入力してください。

```
nmake /f win32.mak
```

Visual Studio 6.0以降を使用して64ビット版のlibpqクライアントライブラリを構築するためには、srcディレクトリに移動し、以下のコマンドを入力してください。

```
nmake /f win32.mak CPU=AMD64
```

サポートされる変数についての詳細についてはwin32.mak ファイルを参照してください。

Borland C++以降を使用してlibpqクライアントライブラリを構築するためには、srcディレクトリに移動し、以下のコマンドを入力してください。

```
make -N -DCFG=Release /f bcc32.mak
```

16.2.1. 生成されるファイル

以下のファイルが構築されます。

```
interfaces¥libpq¥Release¥libpq.dll
```

動的リンク可能なフロントエンドライブラリ

```
interfaces¥libpq¥Release¥libpqdll.lib
```

ユーザのプログラムをlibpq.dllにリンクする際に使用するインポートライブラリ

```
interfaces¥libpq¥Release¥libpq.lib
```

静的なフロントエンドライブラリ

通常何らかのクライアント用のファイルをインストールする必要はありません。アプリケーションの実行ファイルと同じディレクトリにlibpq.dll ファイルを格納しなければなりません。絶対に必要な理由がない限り、libpq.dllをWindowsのSystemやSystem32ディレクトリにインストールしないでください。このファイルがセットアッププログラムを使ってインストールされるならば、ライブラリの新しいバージョンが上書きされないように、ファイルに含まれるVERSIONINFOリソースを使ったバージョン検査をしなければいけません。

もしこのマシンでlibpqを使った開発を行う場合は、src¥includeとsrc¥interfaces¥libpqというソースツリーのサブディレクトリをコンパイラ設定のインクルードパスに追加する必要があります。

ライブラリを使うためには、プロジェクトにlibpqdll.lib ファイルを追加しなくてはなりません。（Visual C++ではプロジェクトを右クリックし、追加を選択してください。）

第 17 章 サーバの準備と運用

目次

- 17.1. [PostgreSQL ユーザアカウント](#)
- 17.2. [データベースクラスタの作成](#)
- 17.3. [データベースサーバの起動](#)
 - 17.3.1. [サーバ起動の失敗](#)
 - 17.3.2. [クライアント接続の問題](#)
- 17.4. [カーネルリソースの管理](#)
 - 17.4.1. [共有メモリとセマフォ](#)
 - 17.4.2. [リソースの制限](#)
 - 17.4.3. [Linuxのメモリオーバーコミット](#)
- 17.5. [サーバのシャットダウン](#)
- 17.6. [PostgreSQL クラスタのアップグレード処理](#)
 - 17.6.1. [pg_dump を介したデータのアップグレード](#)
 - 17.6.2. [ダンプを使用しないアップグレード方法](#)
- 17.7. [サーバのなりすまし防止](#)
- 17.8. [暗号化オプション](#)
- 17.9. [SSL による安全な TCP/IP 接続](#)
 - 17.9.1. [クライアント証明書の使用](#)
 - 17.9.2. [サーバにおける SSL 関連ファイルの利用](#)
 - 17.9.3. [自己署名証明書の作成](#)
- 17.10. [SSH トンネルを使った安全な TCP/IP 接続](#)
- 17.11. [Windows における Event Log の登録](#)

本章では、データベースサーバの設定と実行方法、そしてオペレーティングシステムとの相互作用について説明します。

[前のページ](#)

Visual C++ または Borland C++ を使用
した libpq の構築

[ホーム](#)
[上に戻る](#)[次のページ](#)
PostgreSQL ユーザアカウント

17.1. PostgreSQL ユーザアカウント

外部へアクセスできるサーバデーモンと同じように、PostgreSQL を独立したユーザアカウントで実行することをお勧めします。このユーザアカウントは、サーバによって管理されるデータのみを所有するようにすべきです。また、他のデーモンとアカウントを共有すべきではありません。（例えば、"nobody" ユーザの使用はお勧めできません。）このユーザによって所有される実行プログラムをインストールすることも好ましくありません。システムが攻撃を受けた場合などに、自分自身のバイナリを変更されてしまう可能性があるからです。

システムに Unix のユーザアカウントを追加するためには、`useradd` コマンドか `adduser` コマンドを使用してください。`postgres` というユーザ名がよく使われ、本書全体でも使用していますが、好みの名前を使用しても構いません。

17.2. データベースクラスタの作成

まず第一に、ディスク上にデータベース格納領域を初期化する必要があります。この格納領域をデータベースクラスタと呼びます（SQLではカタログクラスタという用語が使用されます）。データベースクラスタはデータベースの集合で、稼働しているデータベースサーバのただ1つのインスタンスを通して管理されます。初期化が終わると、データベースクラスタには `postgres` という名前のデータベースが含まれています。このデータベースは、ユーティリティやユーザ、サードパーティ製アプリケーションのデフォルトデータベースとして用意されています。データベースサーバ自身はこの `postgres` データベースの存在を必要としていません。初期化中に他にも `template1` というデータベースが各クラスタ内に作成されます。その名前から推測できるように、これはその後に作成されるデータベースのテンプレートとして使われます。したがって、実際の作業に使用すべきではありません。（クラスタ内における新しいデータベースの作成については[第21章](#)を参照してください。）

ファイルシステムの観点から見ると、データベースクラスタというのは、すべてのデータが格納される1つのディレクトリということになります。これはデータディレクトリもしくはデータ領域と呼ばれます。どこにデータを格納するかは完全にユーザの自由です。特にデフォルトの領域はありませんが、一般的によく使われるのは `/usr/local/pgsql/data` か `/var/lib/pgsql/data` です。データベースクラスタを初期化するためには、PostgreSQLと一緒にインストールされる `initdb` コマンドを使用してください。データベースクラスタのファイルシステム上の場所は `-D` オプションで示します。例えば次のようにします。

```
$ initdb -D /usr/local/pgsql/data
```

このコマンドは、前節で説明したPostgreSQLユーザアカウントでログインして実行する必要があります。

ティップ: `-D` オプションを使う代わりに `PGDATA` 環境変数を設定することもできます。

他にも以下のように `pg_ctl` プログラム経由で `initdb` を実行することができます。

```
$ pg_ctl -D /usr/local/pgsql/data initdb
```

`pg_ctl` がデータベースサーバインスタンスの管理に使用する単一のコマンドになりますので、サーバの起動や停止に `pg_ctl` を使用している場合([項17.3](#)参照)はこちらの方がより直感的かもしれません。

もし指定したディレクトリが存在しない場合は、`initdb` はその新しいディレクトリを作成しようとします。しかし（このアドバイスに従って非特権アカウントを作成した場合など）作成権限がない場合があります。その場合は（`root` として）手動でディレクトリ自体を作成し、その所有者をPostgreSQLユーザに変更します。下記がその例となります。

```
root# mkdir /usr/local/pgsql/data
root# chown postgres /usr/local/pgsql/data
root# su postgres
postgres$ initdb -D /usr/local/pgsql/data
```

データディレクトリが既に初期化されているように見える場合は、`initdb` は実行を拒否します。

データディレクトリにはデータベースの中のすべてのデータが保持されるため、権限を持たない人からのアクセスを確実に制限することが不可欠です。ですから、`initdb` はPostgreSQLユーザ以外にはアクセス権を与えません。

しかし、ディレクトリの内容は安全ですが、デフォルトのクライアント認証の設定では、すべてのローカルユーザはデータベースに接続でき、データベーススーパーユーザになることさえ可能です。他のローカルユーザを信用しない場合、`initdb` の `-w`、`--pwprompt`、`--pwfile` オプションのいずれか1つを使用して、データベーススーパーユーザにパスワードを付与することを推奨します。また、デフォルトの `trust` 認証モードを使用しないように、`-A md5` もしくは `-A password` を指定してください。もしくは、`initdb` の後、初回のサーバの起動の前に、生成済みの `pg_hba.conf` ファイルを変更してください。（他の穏当な方法として、`peer` 認証やファイルシステムの権限を使用して、接続を制限することもできます。詳細については[第19章](#)を参照してください。）

`initdb` はまた、データベースクラスタのデフォルトのロケールを初期化します。通常は、環境のロケール設定を初期化されたデータベースにそのまま適用します。データベースに異なるロケールを指定することも可能です。詳細については[項22.1](#)を参照してください。特定のデータベースクラスタ内で使用されるデフォルトのソート順は `initdb` で設定されます。異なるソート順を使用する新しいデータベースを作成することもできますが、`initdb` が作成するテンプレートデータベースで使用される順は削除して再作成しない限り変更することができません。また、`C` や `POSIX` 以外のロケールを使用する場合には性能上の影響もあります。ですので初回にこれを正しく選択することが重要です。

また `initdb` は、データベースクラスタのデフォルトの文字セット符号化方式も設定します。通常これは、ロケールの設定と合うものが選ばれなければなりません。詳細は[項22.3](#)を参照してください。

17.2.1. ネットワークファイルシステム

多くのインストレーションはネットワークファイルシステム上にデータベースクラスタを作成します。直接NFSを介して行うこともありますし、NFSを内部的に使うネットワークアタッチトストレージ（NAS）デバイスを使用することもあります。PostgreSQLはNFSファイルシステムに特別なことを何も行いません。つまりNFSはローカル接続のドライブ（DAS、ダイレクトアタッチトストレージ）とまったく同様に振る舞うという前提です。クライアントとサーバのNFS実装が標準セマンティックスを持たないならば、信頼性に関する問題が発生します（http://www.time-travellers.org/shane/papers/NFS_considered_harmful.html参照）。特にNFSサーバへの遅延（非同期）書き込みが信頼性に関する問題を引き起こします。可能ならばNFSファイルシステムを（キャッシュ無しで）同期マウントし、これを防止してください。更に、NFSをソフトマウントすることは薦められません。（ストレージエリアネットワーク（SAN）はNFSよりも低レベルな通信プロトコルを使用します。）

[前のページ](#)

PostgreSQLユーザアカウント

[ホーム](#)

[上に戻る](#)

[次のページ](#)

データベースサーバの起動

17.3. データベースサーバの起動

データベースにアクセスするためには、まずデータベースサーバを起動しなくてはなりません。データベースサーバプログラムは`postgres`という名前です。`postgres`プログラムは自分が使用するデータがどこにあるのかを知っている必要があります。これは`-D`オプションで指定されます。したがって、サーバを起動する一番簡単な方法は、以下のようなコマンドとなります。

```
$ postgres -D /usr/local/pgsql/data
```

上記のコマンドはサーバをフォアグラウンドで実行させます。これは、PostgreSQLユーザアカウントにログインして実行されなくてはなりません。`-D`オプションが指定されていない場合、サーバは`PGDATA`環境変数で指定されたデータディレクトリを使用しようと試みます。どちらの変数も指定されていないと失敗します。

通常はバックグラウンドで`postgres`を起動することをお勧めします。そのためには以下のように通常のUnixシェルの構文を使います。

```
$ postgres -D /usr/local/pgsql/data >logfile 2>&1 &
```

この例のように、サーバの標準出力と標準エラー出力をどこかに保管しておくことが重要です。これは追跡記録的な目的と問題の原因究明に役立ちます。（ログファイルの取り扱いについての全体的な説明については[項23.3](#)を参照してください。）

`postgres`プログラムには、この他にも多くのコマンドラインオプションを指定することができます。詳細は[postgres](#)マニュアルページと後述の[第18章](#)を参照してください。

こうしたシェル構文は長くなりがちです。そのため、`pg_ctl`ラッププログラムが提供されていて、いくつかのタスクを単純化しています。以下に例を示します。

```
pg_ctl start -l logfile
```

これは、サーバをバックグラウンドで起動し、出力を指定されたログファイルに書き出します。`-D`オプションは、ここでも`postgres`の場合と同じ意味を持ちます。`pg_ctl`によってサーバを停止させることもできます。

通常、コンピュータが起動された時にデータベースサーバも一緒に起動したい場合が多いと思われます。自動起動スクリプトはオペレーティングシステム固有のものです。いくつかはPostgreSQLの`/contrib/start-scripts`ディレクトリに同梱されています。このインストールにはおそらくroot権限が必要となります。

起動時にデーモンを開始する方法はシステムによって異なります。多くのシステムには`/etc/rc.local`ファイルや`/etc/rc.d/rc.local`ファイルがあります。他のシステムでは`init.d`や`rc.d`ディレクトリが使用されます。何を実行するにしても、サーバはPostgreSQLユーザアカウントで起動させなければなりません。`root`であってはいけませんし、他のユーザでもいけません。したがって、`su -c '...' postgres`を使用してコマンドを実行しなければならないでしょう。以下に例を示します。

```
su postgres -c 'pg_ctl start -D /usr/local/pgsql/data -l serverlog'
```

さらにいくつかのオペレーティングシステム固有の提案を挙げます。（ここでは一般的な値で説明していますので、各項目において適切なインストールディレクトリとユーザ名に置き換えて読んでください。）

- FreeBSDでは、PostgreSQLで配布されるソースの中にある`contrib/start-scripts/freebsd`ファイルを参照してください。
- OpenBSDでは、以下の数行を`/etc/rc.local`ファイルに追加してください。

```
if [ -x /usr/local/pgsql/bin/pg_ctl -a -x /usr/local/pgsql/bin/postgres ]; then
    su -l postgres -c '/usr/local/pgsql/bin/pg_ctl start -s -l /var/postgresql/log -D
    /usr/local/pgsql/data'
    echo -n ' postgresql'
fi
```

- Linuxシステムでは、

```
/usr/local/pgsql/bin/pg_ctl start -l logfile -D /usr/local/pgsql/data
```

を`/etc/rc.d/rc.local`や`/etc/rc.local`に追加してください。または、PostgreSQLで配布されるソースの中にある`contrib/start-scripts/linux`ファイルを参照してください。

- NetBSDでは、FreeBSDかLinuxの好きな方の起動スクリプトを使用してください。
- Solarisでは、`/etc/init.d/postgresql`というファイルを作成し、そこに以下の1行を記述してください。

```
su - postgres -c "/usr/local/pgsql/bin/pg_ctl start -l logfile -D /usr/local/pgsql/data"
```

そして、`/etc/rc3.d`以下に`S99postgresql`としてそのファイルに対するシンボリックリンクを作成してください。

サーバが実行している間は、そのPIDはデータディレクトリの中の`postmaster.pid`ファイルに記述されています。これは同じデータディレクトリで複数のサーバインスタンスが実行されるのを防止し、また、サーバの停止にも使うことができます。

17.3.1. サーバ起動の失敗

サーバの起動が失敗する理由として代表的なものがいくつかあります。サーバのログファイルを点検するか、（標準出力や標準エラーをリダイレクトせずに）手で起動して、どのようなエラーメッセージが出ているか確認してください。以下に、よく発生するエラーメッセージのいくつかをより詳細に説明します。

```
LOG:  could not bind IPv4 socket: Address already in use
HINT:  Is another postmaster already running on port 5432? If not, wait a few seconds and retry.
FATAL:  could not create TCP/IP listen socket
```

これはたいていの場合メッセージが示す通りの意味です。既にサーバが動いているポートで別の`postmaster`を起動しようとしたことを示しています。しかし、カーネルエラーメッセージが`Address already in use`やそれに類似したものではない場合は、別の問題の可能性もあります。例えば、予約済みのポート番号でサーバを起動しようすると下記のようなメッセージが出るかもしれません。

```
$ postgres -p 666
LOG:  could not bind IPv4 socket: Permission denied
HINT:  Is another postmaster already running on port 666? If not, wait a few seconds and retry.
FATAL:  could not create TCP/IP listen socket
```

次のようなメッセージが表示された場合、

```
FATAL:  could not create shared memory segment: Invalid argument
DETAIL:  Failed system call was shmget(key=5440001, size=4011376640, 03600).
```

これは、おそらくカーネルによる共有メモリのサイズの上限がPostgreSQLが作ろうとしている作業領域（この例では4011376640バイト）よりも小さい可能性があります。または、System V方式の共有メモリサポートがカーネルにまったく設定されていない可能性もあります。一時的な策として、（[shared buffers](#)を使用して）サーバを通常よりも少ないバッファ数で起動することもできます。しかし最終的には、カーネルを再設定し、使用可能な共有メモリサイズを増やした方が良いでしょう。このメッセージは、同じマシン上で複数のサーバを起動しようとした時に、要求された領域の合計がカーネルの上限を超えた場合にも表示されます。

下記のようなエラーはディスクの空き容量がなくなったということを示しているわけではありません。

```
FATAL:  could not create semaphores: No space left on device
DETAIL:  Failed system call was semget(5440126, 17, 03600).
```

これはカーネルのSystem Vセマフォの上限が、PostgreSQLが作成しようとしている数よりも小さいということを示しています。上記のように、許可される接続の数を減らして（[max_connections](#)を使用して）サーバを起動させることで問題は回避できるかもしれませんが、最終的にはカーネルの設定を変えてセマフォの上限を増やした方が良いでしょう。

"illegal system call"というエラーが表示された場合は、使用しているカーネルでは共有メモリやセマフォがまったくサポートされていない可能性があります。その場合、これらの機能を使えるようにカーネルを設定し直すことが唯一の選択肢となります。

System V IPC設備の設定についての詳細は[項17.4.1](#)を参照してください。

17.3.2. クライアント接続の問題

クライアント側で起こり得るエラー状態はきわめて多様で、アプリケーションに依存します。その中のいくつかはサーバが起動された方法と直接関係するかもしれません。以下で説明する以外の状態については各々のクライアントアプリケーションの資料を参照してください。

```
psql: could not connect to server: Connection refused
        Is the server running on host "server.joe.com" and accepting
        TCP/IP connections on port 5432?
```

これは一般的な"接続するサーバが見つけれませんでした"という失敗です。TCP/IP通信を試みた時に上記のように表示されます。よくある間違いはサーバにTCP/IPを許可する設定を忘れていることです。

代わりに、ローカルのサーバにUnixソケット通信を試みると下記のような表示が出ます。

```
psql: could not connect to server: No such file or directory
        Is the server running locally and accepting
        connections on Unix domain socket "/tmp/.s.PGSQL.5432"?
```


最後の行は、クライアントが正しいところに接続しようとしていることを実証するのに役立ちます。もしそこに動いているサーバがない場合、典型的なカーネルエラーメッセージは、表示されているように`Connection refused`もしくは`No such file or directory`となります。（この場合の`Connection refused`はサーバが接続要求を受け付けた後に拒否したわけではないということを理解しておくことが大切です。もしそうだった場合は[項19.4](#)で示されるような別のメッセージが表示されます。）`Connection timed out`のような他のメッセージは、例えばネットワーク接続の欠如のようなもっと根本的な問題を表しています。

[前のページ](#)

データベースクラスタの作成

[ホーム](#)

[上に戻る](#)

[次のページ](#)

カーネルリソースの管理

17.4. カーネルリソースの管理

大規模なPostgreSQLインストールでは、すぐに各種オペレーティングシステムのリソース制限を超えてしまうことがあります。（システムによっては、実際に"大規模"なインストールでなくても、出荷時のデフォルトでは低過ぎるものもあります。）この種の問題が発生したら、これらを読んでください。

17.4.1. 共有メモリとセマフォ

共有メモリとセマフォはひとまとめに"System V IPC"と呼ばれます（メッセージキューも一緒ですが、これはPostgreSQLとは関係ありません）。ほとんどすべての最近のオペレーティングシステムはこれらの機能を提供していますが、デフォルトではこれらの多くで有効になっていませんし、また、特に搭載されるRAMやデータベースアプリケーションの需要が大きくなっているにも関わらず、十分なサイズがありません。（Windows版では、PostgreSQLは独自の代替的な実装でこれらの機能を提供しています。このため本節のほとんどは無視することができます）。

これらの機能の完全な欠落は、サーバ起動時のIllegal system callエラーによって判明します。その場合はカーネルを設定し直すしかありません。PostgreSQLはこれらの機能なしでは動きません。しかし最近のオペレーティングシステムではこうした状況はまれなものです。

PostgreSQLが様々なIPCのハードリミットの1つを超えると、サーバは起動を拒否し、問題および何をすべきかを説明するエラーメッセージを残します。（[項17.3.1](#) も参照してください。）関係するカーネルパラメータは別々のシステム上でも統一して名付けられています。[表17-1](#)で概略がわかります。しかしこれらを設定するための方法は異なります。以下に、いくつかのプラットフォームへの提案を挙げます。

表 17-1. System V IPCパラメータ

名前	説明	適切な値
SHMMAX	共有メモリセグメントの最大サイズ (バイト)	最小でも数メガバイト（本文を参照してください）
SHMMIN	共有メモリセグメントの最小サイズ (バイト)	1
SHMALL	使用可能な共有メモリの総量 (バイトまたはページ)	バイト指定の場合SHMMAXと同じです。 ページ指定の場合は <code>ceil (SHMMAX / PAGE_SIZE)</code> です。
SHMSEG	プロセスごとの共有メモリセグメントの最大数	必要なのは1セグメントだけですが、デフォルトではもっと多くなっています
SHMMNI	システム全体の共有メモリセグメントの最大数	SHMSEGと同様 + 他のアプリケーション用の空間
SEMMNI	セマフォ識別子の最大数（つまりセット）	最低 <code>ceil ((max_connections + autovacuum_max_workers + 4) / 16)</code>
SEMMNS	システム全体のセマフォの最大数	<code>ceil ((max_connections + autovacuum_max_workers + 4) / 16) * 17 + 他</code> のアプリケーション用の空間
SEMMSL	セットごとのセマフォの最大数	最低17
SEMMAP	セマフォマップの中の項目の数	本文を参照
SEVMX	セマフォの最大値	最低1000（デフォルトはしばしば32767ですが、必要がなければ変更しないでください）

共有メモリに関する一番重要なパラメータは、共有メモリセグメントの最大サイズのバイト数SHMMAXです。もしshmgetから"Invalid argument"のようなエラーメッセージを受けた場合、おそらくこの上限を超えています。必要な共有メモリセグメントのサイズは、[表17-2](#)に示す各種のPostgreSQL設定パラメータによって変わります。（エラー時に出力されるメッセージにはすべて、割り当て要求に失敗した正確なサイズが記載されています。）エラーをなくするための一時的な策として、これらの設定を低くすることもできます。SHMMAXを2メガバイトとしてPostgreSQLを稼働させることができますが、受容できる性能を確保するためにはかなりより多くのサイズが必要です。10メガバイト単位から100メガバイトから数ギガバイトの設定を推奨します。

また、システムの中には、システムにおける共有メモリの総量（SHMALL）に対する制限があるものがあります。この値を確実に、PostgreSQLと共有メモリセグメントを使用する他のアプリケーションの合計よりも十分に大きくしてください。多くのシステムでSHMALLはバイト単位ではなくページ単位であることに注意してください。

問題が少ないのは共有メモリセグメントの最小サイズ（`SHMMIN`）で、PostgreSQLでは最大でもおよそ500キロバイトのはずです（通常では1です）。システム全体のセグメントの最大数（`SHMMNI`）もしくはプロセスごとのセグメントの最大数（`SHMSEG`）に関して、使用しているシステムで0に設定されていない限り、問題が起きることはほぼありません。

PostgreSQLは、許可した接続（[max_connections](#)）および許可したワーカプロセス（[autovacuum_max_workers](#)）ごとに1つのセマフォを使用し、16個のセマフォを一集合として扱います。この集合それぞれは17個目のセマフォを持ち、そのセマフォは他のアプリケーションに使われているセマフォセットとの衝突を検出するための"マジックナンバー"を持っています。システム内のセマフォの最大数は`SEMMNS`によって設定され、その結果としてその値は少なくとも`max_connections + autovacuum_max_workers`と同じ、ただし、許可された接続とワーカ16個ごとに余分な1個を加えた値以上はなければいけません（[表17-1](#)の公式を参照してください）。`SEMMNI`パラメータはシステム上に同時に存在できるセマフォ集合の数の上限を決定します。ですからこのパラメータは少なくとも`ceil((max_connections + autovacuum_max_workers + 4) / 16)`以上はなくてははいけません。一時的な失敗の回避策としては許可される接続の数を下げることができますが、"No space left on device"という紛らわしい言葉が`semget()`関数から表示されます。

場合によっては`SEMMAP`を少なくとも`SEMMNS`と同程度に増やすことが必要になる場合があるかもしれません。このパラメータはセマフォリソースマップのサイズを定義し、その中では有効なセマフォのそれぞれの隣接したブロックの項目が必要です。セマフォ集合が解放されると、解放されたブロックに隣接する既に存在する項目に追加されるか、もしくは新しいマップの項目の下に登録されます。もしマップが一杯だった場合、解放されたセマフォは（再起動するまで）失われます。セマフォ空間の断片化により時間が経つごとに、有効なセマフォがあるべき量よりも少なくなる可能性があります。

1つの集合の中にいくつのセマフォがあるかを決める`SEMMSL`はPostgreSQLでは少なくとも17はなくてははいけません。

`SEMMNU`と`SEMUME`のような、その他の様々な"semaphore undo"に関する設定はPostgreSQLには影響を与えません。

AIX

少なくともバージョン5.1では、すべてのメモリが共有メモリとして使用できるように設定されているようにみえますので、`SHMMAX`などのパラメータに対して特別な設定は必要ありません。これはDB/2などの他のデータベースでも使用される、一般的な設定方法です。

しかし、`/etc/security/limits`内の大域的な`ulimit`情報は変更しなければならないかもしれません。デフォルトのファイルサイズ（`fsize`）とファイル数（`nofiles`）用のハードリミットは低過ぎるかもしれないためです。

FreeBSD

デフォルトの設定は、小規模なインストレーションでのみ適しています（例えば、デフォルトの`SHMMAX`は32メガバイトです）。`sysctl`または`loader`インタフェースを使用して変更を行うことができます。以下では`sysctl`を使用してパラメータを変更しています。

```
$ sysctl -w kern.ipc.shmall=32768
$ sysctl -w kern.ipc.shmmax=134217728
$ sysctl -w kern.ipc.semmap=256
```

これらの設定をリブートしても永続化するには、`/etc/sysctl.conf`を変更します。

残りのセマフォ設定は`sysctl`では読み取りのみとみなされていますが、起動前に`loader`プロンプトを使用して変更することができます。

```
(loader) set kern.ipc.semnni=256
(loader) set kern.ipc.semns=512
(loader) set kern.ipc.semnu=256
```

同様に、これらの設定をリブートしても永続化させるには`/boot/loader.conf`に保存します。

また、共有メモリをRAM上に残し、スワップへのページアウトを行わせないようにさせたいかもしれません。これは`sysctl`の`kern.ipc.shm_use_phys`設定を使用して実現できます。

`sysctl`の`security.jail.sysvipc_allowed`を有効にしてFreeBSD jailを実行している場合、異なるjailで実行するpostmasterを異なるオペレーティングシステムユーザで実行しなければなりません。これは、非特権ユーザが別のjailの共有メモリやセマフォに干渉することを防止できるため、セキュリティが向上します。また、これによりPostgreSQLのIPCを整理するコードを適切に動作させることができます。（FreeBSD 6.0以降では、IPC整理コードは他のjailにおけるプロセスを適切に検出せず、異なるjailで同一ポートでpostmasterを実行させることができます。）

FreeBSDバージョン4.0以前では、（後述の）OpenBSDと同様に動作します。

NetBSD

NetBSD以降では、以下の例のようにIPCパラメータを`sysctl`を用いて調整することができます。

```
$ sysctl -w kern.ipc.shmmax=16777216
```

この設定をリブートしても永続化させるためには/etc/sysctl.confを編集してください。

また、共有メモリをRAM上に固定して、スワップのためのページアウトを避けるために、カーネルを設定したいと考えるかもしれません。

5.0より前のバージョンのNetBSDでは、（後述の）OpenBSDのように動作します。ただし、パラメータはoptionではなくoptionsキーワードを付けて設定しなければなりません。

OpenBSD

SYSVSHMオプションとSYSVSEMオプションはカーネルのコンパイル時に有効にする必要があります（デフォルトでは有効になっています）。共有メモリの最大サイズはSHMMAXPGSオプション（ページ数）で決定されます。以下に様々なパラメータの設定方法の例を示します。

```
option      SYSVSHM
option      SHMMAXPGS=4096
option      SHMSEG=256

option      SYSVSEM
option      SEMMNI=256
option      SEMMNS=512
option      SEMMNU=256
option      SEMMAP=256
```

また、共有メモリをRAMの中にロックするようにカーネルを設定することで、スワップにページアウトしないようにもできます。sysctlを使用してkern.ipc.shm_use_physを設定することができます。

HP-UX

デフォルトの設定は通常のインストールではほぼ十分です。HP-UX 10ではSEMMNSの出荷時のデフォルトは128ですが、これは大規模なデータベースサイトには低過ぎるかもしれません。

IPCパラメータはシステム管理マネージャ（SAM）からKernel Configuration->Configurable Parametersの下で、設定することができます。終わったらCreate A New Kernelを選択してください。

Linux

デフォルトの最大セグメントサイズは32メガバイトで、非常に小規模なPostgreSQLインストレーションのみに適しています。デフォルトの最大総サイズは2097152ページです。"大規模ページ"を持つ通常ではないカーネル設定を除き、1ページはほとんど常に4096バイトです。（検証にはgetconf PAGE_SIZEを使用してください。）これによりデフォルトの上限は8ギガバイトになります。たいていは十分ですが、常に十分というわけではありません。

共有メモリサイズの設定はsysctlインタフェースを使用して変更可能です。例えば16ギガバイトまで許すには以下のようになります。

```
$ sysctl -w kernel.shmmax=17179869184
$ sysctl -w kernel.shmall=4194304
```

更にこれらの設定をリブート時に保持できるように/etc/sysctl.confに保存することができます。こうすることを大いに勧めます。

古めのディストリビューションではsysctlプログラムが存在しない可能性があります。この場合、/procファイルシステムに対する操作で同等の変更を行うことができます。

```
$ echo 17179869184 >/proc/sys/kernel/shmmax
$ echo 4194304 >/proc/sys/kernel/shmall
```

他のデフォルトはかなり豊富なサイズですので、通常は変更する必要はありません。

Mac OS X

OS Xにおける共有メモリの推奨設定方法は、以下のような変数代入文からなる/etc/sysctl.confという名称のファイルを作成することです。

```
kern.sysv.shmmax=4194304
kern.sysv.shmmin=1
kern.sysv.shmmni=32
kern.sysv.shmseg=8
kern.sysv.shmall=1024
```

一部のバージョンのOS Xでは/etc/sysctl.conf内に共有メモリパラメータ5つすべてを設定しなければならないという点に注意してください。さもなければ値が無視されます。

最近のリリースのOS Xは、SHMMAXを4096の倍数以外に設定しようとすると無視しますので、注意してください。

このプラットフォームではSHMALLは4キロバイトページ単位です。

古めのバージョンのOS Xでは、共有メモリパラメータの変更を有効にするためにリブートが必要になります。10.5からSHMMNI以外の変更は、sysctlを用いることにより、その場で行うことができます。しかしいずれにせよ/etc/sysctl.conf経由で望む値に設定することが最善です。リブートを行っても値が保持されるからです。

/etc/sysctl.confはOS X 10.3.9以降でのみ使用されます。もしこれより前の10.3.xリリースを使用しているのであれば、/etc/rc ファイルを編集し、以下のコマンドで値を変更しなければなりません。

```
sysctl -w kern.sysv.shmmax
sysctl -w kern.sysv.shmmin
sysctl -w kern.sysv.shmmni
sysctl -w kern.sysv.shmseg
sysctl -w kern.sysv.shmall
```

通常/etc/rcはOS Xのアップデートで上書きされることに注意してください。ですので、アップデートの度に編集し直す必要があるものと考えなければなりません。

OS X 10.2以前では、代わりに/System/Library/StartupItems/SystemTuning/SystemTuning ファイル内にあるこれらのコマンドを編集してください。

SCO OpenServer

デフォルトの設定では、セグメント当たり512キロバイトの共有メモリが許されています。この設定を増加させるには、まず、/etc/conf/cf.dディレクトリに移動します。SHMMAXの現在値を表示させるには、以下を実行します。

```
./configure -y SHMMAX
```

SHMMAXに新しい値を設定するには以下を実行します。

```
./configure SHMMAX=value
```

ここで、valueが希望する新しい値（バイト単位）です。そして、以下のようにカーネルを再構築し、リブートします。

```
./link_unix
```

Solaris 2.6から2.9 (Solaris 6からSolaris 9)

共有メモリセグメントのデフォルトの最大サイズはPostgreSQLには低過ぎる設定になっています。必要な設定は/etc/systemで変えることができ、例えば以下ようになります。

```
set shmsys:shminfo_shmmax=0x2000000
set shmsys:shminfo_shmmin=1
set shmsys:shminfo_shmmni=256
set shmsys:shminfo_shmseg=256

set semsys:seminfo_semmap=256
set semsys:seminfo_semmni=512
set semsys:seminfo_semmns=512
set semsys:seminfo_semmsl=32
```

変更を反映させるには再起動する必要があります。古めのバージョンのSolarisにおける共有メモリの情報は<http://sunsite.uakom.sk/sunworldonline/swol-09-1997/swol-09-insidesolaris.html>を参照してください。

Solaris 2.10 (Solaris 10)

OpenSolaris

Solaris 10とOpenSolarisでは、デフォルトの共有メモリとセマフォ設定は大抵のPostgreSQLアプリケーションで十分あります。SolarisのデフォルトのSHMMAXはシステムのRAMの1/4になりました。共有メモリの設定をより大きく設定するためにこの値を増やしたいのであれば、postgresユーザに関するプロジェクト設定を使用しなければなりません。例えば以下をroot権限で実行してください。

```
projadd -c "PostgreSQL DB User" -K "project.max-shm-memory=(privileged,8GB,deny)" -U postgres -G postgres user.postgres
```

このコマンドはuser.postgresプロジェクトを追加し、postgresユーザの共有メモリの最大サイズを8GBまで上げます。この影響は次にこのユーザがログインした時、またはPostgreSQLを再起動した時（再読み込み時ではありません）に有効になります。上ではPostgreSQLはpostgresグループに属するpostgresユーザにより実行されていることを前提としています。サーバの再起動は不要です。

多くの接続を受け付けるデータベースサーバにおいて推奨するカーネル設定にはこの他に以下があります。


```
project.max-shm-ids=(priv,32768,deny)
project.max-sem-ids=(priv,4096,deny)
project.max-msg-ids=(priv,4096,deny)
```

さらに、ゾーン内でPostgreSQLを実行している場合、ゾーンのリソース使用上限も上げる必要があるかもしれません。
projectsとprctlについてはSolaris 10 System Administrator's Guideの第2章 プロジェクトとタスクを参照してください。

UnixWare

UnixWare 7では、共有メモリセグメントの最大サイズはデフォルト設定で512キロバイトしかありません。現在のSHMMAX値を表示するためには下記を実行してください。

```
/etc/conf/bin/ldtune -g SHMMAX
```

これは現在値、デフォルト値、最小値、および最大値を、バイト単位で表示します。SHMMAXの新しい値を設定するためには、以下を実行します。

```
/etc/conf/bin/ldtune SHMMAX value
```

ここでvalue は、希望する新しい値（バイト）です。SHMMAXの設定が終わったらカーネルを再構築し、リブートします。

```
/etc/conf/bin/ldbuild -B
```

表 17-2. PostgreSQLの共有メモリ使用量

使用法	必要な共有メモリのおおよそのバイト数（8.3時点）
接続	$(1800 + 270 * \text{max\_locks\_per\_transaction}) * \text{max\_connections}$
自動バキュームワーカー	$(1800 + 270 * \text{max\_locks\_per\_transaction}) * \text{autovacuum\_max\_workers}$
プリペアドトランザクション	$(770 + 270 * \text{max\_locks\_per\_transaction}) * \text{max\_prepared\_transactions}$
共有ディスクバッファ	$(\text{block\_size} + 208) * \text{shared\_buffers}$
WALバッファ	$(\text{wal\_block\_size} + 8) * \text{wal\_buffers}$
固定の必要な空き容量	770 kB

17.4.2. リソースの制限

UnixライクなオペレーティングシステムではPostgreSQLサーバの操作と関係する可能性のある様々な種類のリソース制限があります。特に重要なのは、ユーザごとのプロセス数の制限、プロセスごとのオープンファイルの数、プロセスごとの利用可能なメモリの量です。これらのそれぞれが"ハード"と"ソフト"の2つの制限を持っています。ソフト制限が実際に有効な制限ですが、ユーザによってハード制限まで変えることが可能です。ハード制限はrootユーザによってのみ変えることができます。setrlimitシステムコールがこれらのパラメータの設定を行います。シェルの組み込みコマンドulimit（Bourne シェル）もしくはlimit（csh）は、コマンドラインからリソース制限を制御するために使われます。BSD派生システム上では/etc/login.confファイルが、ログイン時に設定される様々なリソース制限を制御します。詳細はオペレーティングシステムの文書を参照してください。関連するパラメータはmaxproc、openfiles、datasizeです。以下に例を示します。

```
default:¥
...
:datasize-cur=256M:¥
:maxproc-cur=256:¥
:openfiles-cur=256:¥
...
```

（-curはソフト制限です。ハード制限を設定するためには-maxを付けてください。）

カーネルはいくつかのリソースに対して、システム全体の制限も持つことができます。

- Linuxでは、/proc/sys/fs/file-maxが、カーネルがサポートするオープンファイル数の最大を決定します。この数を変えるためには、そのファイルに別の数を書き込むか、あるいは/etc/sysctl.confに代入式を追加します。プロセスごとのファイルの最大制限はカーネルがコンパイルされた時に固定されます。詳しい情報については/usr/src/linux/Documentation/proc.txtを参照してください。

PostgreSQLサーバは接続ごとに1つのプロセスを使うので、少なくとも許可された接続の数だけのプロセスに残りのシステムで必要な分を追加したものがようになります。通常はこれは問題ではありませんが、1つのマシン上でいくつかのサーバを起動している場合は厳しい状況になるかもしれません。

オープンファイルの制限の出荷時のデフォルトは、しばしば大多数のユーザはマシン上でシステムリソースの不正使用をしないという前提に立った"社会的に友好的な"値を設定してしまいます。もし1つのマシン上で複数のサーバを起動する場合はそれが

必要でしょうが、専用サーバではこの制限を上げたいかもしれません。

反対に、個々のプロセスが多数のファイルをオープンすることを許可するシステムもあります。そのようなプロセスが数個以上あれば、システム全体の制限は簡単に超えてしまいます。この発生を検知し、システム全体の制限の変更を望まない場合は、PostgreSQLの[max_files_per_process](#)設定パラメータを設定し、オープンファイルの消費を制限することができます。

17.4.3. Linuxのメモリオーバーコミット

Linux 2.4以降では、デフォルトの仮想メモリの動作はPostgreSQLには最適ではありません。カーネルがメモリオーバーコミットを実装する方法のため、カーネルは、他のプロセスのメモリ要求がシステムの仮想メモリを枯渇させた場合、PostgreSQL postmaster（マスターサーバプロセス）を終了させる可能性があります。

これが発生した場合、以下のようなカーネルメッセージが現れます（こうしたメッセージを検索する場所についてはシステム文書と設定を参照してください）。

```
Out of Memory: Killed process 12345 (postgres).
```

これは、postgresプロセスがメモリ不足のために終了してしまったことを示します。起動中のデータベース接続は正常に動作しますが、新しい接続は受け付けられません。復旧するには、PostgreSQLを再起動しなければなりません。

この問題を防止する1つの方法として、PostgreSQLを他のプロセスがそのマシンのメモリを枯渇させないことが確実なマシンで起動するというものがあります。物理メモリとスワップ領域が消費尽くされた時のみにメモリ不足（OOM）エラーが発生するため、メモリが不足する場合、オペレーティングシステムのスワップ領域を増やすことが問題解決の役に立ちます。

Linux 2.6以降では、メモリを"オーバーコミット"させないようにカーネルの動作を変更することができます。この設定は完全に[OOMエラー](#)の発生を防ぐことはできませんが、その発生頻度をかなり軽減しますので、システム動作の堅牢性をより高めます。これは、以下のようにsysctlを使用して厳密なオーバーコミットモードを選択すること、もしくは、/etc/sysctl.confに同等の項目を記述することで実施されます。

```
sysctl -w vm.overcommit_memory=2
```

また、関連するvm.overcommit_ratio設定を変更した方が良いでしょう。詳細はDocumentation/vm/overcommit-accountingカーネル文書を参照してください。

vm.overcommit_memoryの変更と関係なく使用できるその他の方法は、プロセス固有のoom_score_adj値をpostmasterプロセス向けに-1000に設定することです。これによりOOMエラーの対象とならないことが保証されます。このための最も簡単な方法は以下をpostmasterの起動スクリプト内でpostmasterを実行する直前に実行することです。

```
echo -1000 > /proc/self/oom_score_adj
```

この作業をrootで実行しなければならないことに注意して下さい。さもないと効果がありません。このためrootが所有する起動スクリプトがこれを行うためには最も簡単な場所です。こうした場合、PostgreSQLの構築時に -

DLINUX_OOM_SCORE_ADJ=0をCPPFLAGSに追加したいと考えerでしょう。これによりpostmasterの子プロセスは通常のゼロというoom_score_adjで実行されるようになり、必要に応じてOOMエラーの対象となります。

古いLinuxカーネルは/proc/self/oom_score_adjを提供していませんが、同様の機能を持つ/proc/self/oom_adjがあるでしょう。これは、無効にするための設定値が-1000ではなく-17であるという点を除いては同じように動作します。相当するPostgreSQLのビルドフラグは-DLINUX_OOM_ADJ=0です。

注意: Linux 2.4カーネルのベンダの中には、2.6のオーバーコミットsysctl版を持つものがあることが報告されています。しかし、関係するコードを持たない2.4カーネルでvm.overcommit_memoryを2に設定することはより状況を悪化させます。2.4のインストールではこれを試す前に、実際のカーネルソースコードを調査し、その中でサポートしているかどうかを検証することをお勧めします（mm/mmap.cファイル内のvm_enough_memory関数を参照してください）。overcommit-accounting文書ファイルの存在は、この機能が存在するかどうかを証明するものではありません。疑わしい場合は、使用中のカーネルベンダのカーネル専門家に相談してください。

[前のページ](#)

データベースサーバの起動

[ホーム](#)

[上に戻る](#)

[次のページ](#)

サーバのシャットダウン

17.5. サーバのシャットダウン

データベースサーバをシャットダウンする方法は複数あります。 マスタ `postgres` プロセスに異なるシグナルを送ることで、シャットダウンの方法を制御します。

SIGTERM

これはスマートシャットダウンモードです。 SIGTERMを受け取った後で、サーバは新しい接続を禁止しますが、既に存在するセッションは通常通り動作させます。 すべてのセッションが通常に終了するまではシャットダウンしません。サーバがオンラインバックアップモードである場合、オンラインバックアップモードが活動しなくなるまでさらに待ちます。 バックアップモードが活動している間、新規接続は依然として許可されますが、スーパーユーザに対してだけです（この例外はスーパーユーザがオンラインバックアップモード停止のための接続を許可します）。 スマートシャットダウンが要求された時にサーバがリカバリ状態である場合、すべての通常のセッションが終了した後のみでリカバリとストリーミングレプリケーションが停止します。

SIGINT

これは高速シャットダウンモードです。 サーバは新しい接続を禁止しすべての存在するサーバプロセスにSIGTERMを送り、この結果サーバプロセスは現在のトランザクションをアボートし、即座に終了します。 そして サーバはすべてのサーバプロセスの終了を待って、最後にシャットダウンします。 サーバがオンラインバックアップモードである場合、バックアップモードは終了しますので、そのバックアップは使用することができなくなります。

SIGQUIT

これは即時シャットダウンです。 マスタ `postgres` プロセスは、すべての子プロセスに SIGQUITを送り、自らを正しくシャットダウンせず、即座に終了します。 子プロセスは同様にSIGQUITを受け取ると即座に終了します。 これは次の起動時に（WALログを再実行することで）リカバリをすることになります。 これは緊急の時にのみ使うことを勧めます。

`pg_ctl` プログラムは、サーバをシャットダウンさせるシグナルを送信するための便利なインタフェースを提供します。 他にも、Windows以外のシステムでは`kill`を使用して直接シグナルを送信することもできます。 `postgres` プロセスのPIDは、`ps` プログラム、もしくはデータディレクトリの中の`postmaster.pid`ファイルを使用して見つけることができます。 例えば、高速シャットダウンをするためには下記のようになります。

```
$ kill -INT `head -1 /usr/local/pgsql/data/postmaster.pid`
```

重要項目: サーバをシャットダウンするためにSIGKILLを使わない方が良いでしょう。 これによってサーバが共有メモリとセマフォを解放できなくなり、新しくサーバを開始する前に、手動でその作業をしなければならない可能性があります。 さらに、SIGKILLは、子プロセスにシグナルを中継することなく `postgres` を停止させます。 このため、個々の子プロセスを停止させるために、同じ作業を手作業で行わなければならないようになります。

他のセッションを継続させながら個別のセッションを停止するには`pg_terminate_backend()`（[表9-58](#)を参照）を使用するか、そのセッションに関連する子プロセスにSIGTERMシグナルを送ります。

17.6. PostgreSQL クラスタのアップグレード処理

本節ではPostgreSQLリリースからより新しいリリースにデータベースデータをアップグレードする方法を説明します。

PostgreSQLメジャーバージョンはバージョン番号の最初の2つの数字の組み合わせ、例えば8.4で表されます。PostgreSQLのマイナーバージョンはバージョンを表す数字の3つの組み合わせで表されます。例えば8.4.2は8.4リリースにおける2番目のマイナーリリースです。マイナーリリースでは内部格納書式が変わることは決してありませんので、同じメジャーバージョンにおける前後のマイナーリリースとの間で常に互換性があります。例えば8.4.2は8.4、8.4.1、8.4.6との間で互換性があります。互換性があるバージョンとの間で更新するためには、サーバを停止させ、実行ファイルを置き換え、サーバを再起動させるだけです。データディレクトリはまったく変更されません。マイナーリリースのアップグレードは簡単です。

PostgreSQLのメジャーリリースでは、内部データ格納書式は変更されがちです。したがって、アップグレードは複雑になります。新しいメジャーバージョンにデータを移行する伝統的な方法はデータベースをダンプしてリロードすることです。以下で説明しますが、他の方法もあります。

新しいメジャーバージョンは通常、ユーザにも影響する非互換性がいくつか導入されます。このためアプリケーションのプログラム変更が必要になる可能性があります。ユーザに影響する変更はすべてリリースノート ([付録E](#)) に列挙されています。「移行」という名前の節に特に注意してください。複数のメジャーバージョンをまたいでアップグレードする場合は、関連するバージョンそれぞれのリリースノートを確認してください。

用心深いユーザは、完全に切り替える前に新しいバージョンにおける自身のクライアントアプリケーションを試験したいと考えるでしょう。このため古いバージョンと新しいバージョンを並行してインストールさせるというのは、よく良い考えとなります。PostgreSQLメジャーアップグレードを試験する時、以下に示す変更があり得る分野を検討してください。

管理

各メジャーリリースにおいて、管理者が利用できるサーバの監視、制御機能はよく変更、向上されます。

SQL

通常、これには新しいSQLコマンド機能が含まれます。リリースノートに特に記載がない限りその動作には変更はありません。

ライブラリAPI

繰り返しになりますが、リリースノートに記載がない場合のみですが、通常libpqのようなライブラリには新しい機能が追加されるだけです。

システムカタログ

システムカタログの変更は通常データベース管理用ツールにのみ影響します。

サーバC言語API

ここには、Cプログラム言語で作成されたバックエンド関数APIにおける変更が含まれます。こうした変更は、サーバ内部深くにあるバックエンド関数を参照するコードに影響します。

17.6.1. pg_dumpを介したデータのアップグレード

あるメジャーバージョンのPostgreSQLのデータをダンプし、他のバージョンにリロードするためには、pg_dumpを使用しなければなりません。ファイルシステムレベルのバックアップ方法は動作しません。(あるデータディレクトリ間違っただバージョンのサーバを起動しようとして、大きな損害が起こることがないように、適所に互換性がないバージョンのPostgreSQLのデータディレクトリが使用されないようにするための検査があります。)

より新しいバージョンのPostgreSQLのpg_dumpとpg_dumpallを使用することを勧めます。これらのプログラムになされた可能性がある改良点という利点があるからです。現在のリリースのダンププログラムは7.0以降のバージョンのサーバからデータを読み取ることができます。

以下の手順では、既存のインストレーションが/usr/local/pgsql以下にあり、そのデータ領域が/usr/local/pgsql/dataにあることを前提としています。使用しているパスに適切に置き換えてください。

- 1 11 バックアップを作成する場合、使用しているデータベースが確実に更新されないようにしてください。これはバックアップの一貫性には影響しませんが、当然ながら変更されたデータがバックアップに含まれません。必要に応じ

/usr/local/pgsql/data/pg_hba.conf

て、（またはこれと等価なファイル）における権限を変更して、バックアップを行うユーザ以外からのアクセスを禁止してください。アクセス制御に関する情報は[第19章](#)を参照してください。

データベースインストレーションをバックアップするためには以下を入力してください。

```
pg_dumpall > outputfile
```

（外部キーとしてOIDを使用している場合など）OIDを保持する必要がある場合は、pg_dumpallを実行する時に-oオプションを使用してください。

バックアップを作成するために、現在起動中のバージョンのpg_dumpallコマンドを使用することができます。しかし最善の結果を得るためには、PostgreSQL 9.2.4のpg_dumpallコマンドを試してください。このバージョンでは、過去のバージョンに対して、不具合の修正や改良が含まれているからです。新しいバージョンをまだインストールしていないので、この勧告は奇異に思えるかもしれませんが、古いバージョンと並行して新しいバージョンをインストールすることを計画しているのであれば、これに従うことを推奨します。この場合、インストールを普通に完了させてからデータを移行することができます。これは同時に停止時間を短縮します。

1 11 古いサーバを停止します。

```
pg_ctl stop
```

起動時にPostgreSQLを実行させるようにしているシステムではおそらく、同じことを達成する起動ファイルがあります。例えばRed Hat Linuxシステムでは、以下が動作することが分かります。

```
/etc/rc.d/init.d/postgresql stop
```

サーバの起動と停止については[第17章](#)を参照してください。

1 11 バックアップからリストアする場合、古いインストレーションディレクトリを変名または削除してください。問題があった場合に戻さなければならない場合に備え、削除するよりディレクトリの名前を変更する方を勧めます。このディレクトリが多くのディスク容量を占めている可能性があることに注意してください。ディレクトリの名前を変更するためには、以下のようなコマンドを使用してください。

```
mv /usr/local/pgsql /usr/local/pgsql.old
```

（相対パスが維持されるように確実にディレクトリ単位で移動してください。）

1 11 概要を [項15.4](#) で示すように、新しいバージョンのPostgreSQLをインストールしてください。

1 11 必要に応じて新しいデータベースクラスタを作成してください。（アップグレードの場合はすでに存在している）特別なデータベースユーザアカウントでログインして、このコマンドを実行しなければならないことに注意してください。

```
/usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
```

1 11 以前のpg_hba.confとpostgresql.confに加えた何らかの変更を戻してください。

1 11 繰り返しになりますが、特別なデータベースユーザアカウントを使用して、データベースサーバを起動してください。

```
/usr/local/pgsql/bin/postgres -D /usr/local/pgsql/data
```

1 11 最後に、新しいpsqlを用いて、バックアップからデータをリストアしてください。

```
/usr/local/pgsql/bin/psql -d postgres -f outputfile
```

新しいサーバを異なるディレクトリにインストールし、古いサーバと新しいサーバを別のポートで並行して実行させることで、停止時間を最小にすることができます。この場合、データを移行するために以下のようなコマンドを使用することができます。

```
pg_dumpall -p 5432 | psql -d postgres -p 5433
```

17.6.2. ダンプを使用しないアップグレード方法

[pg_upgrade](#) モジュールにより、PostgreSQLのあるバージョンから次のバージョンにインストレーションをその場で移行することができます。アップグレードを瞬時に行うことができます。

更新対象のバージョンのPostgreSQLをスタンバイサーバとして作成して、Slonyなどの特定のレプリケーション方式を使用することもできます。Slonyが異なるメジャーバージョンのPostgreSQLとの間でレプリケーションすることができるため、これが実現できます。スタンバイは同じコンピュータで作成することも異なるコンピュータで作成することもできます。（古いバージョンのPostgreSQLで実行している）マスタサーバと同期した後、マスタを切り替え、スタンバイをマスタにし、古いデータベースインスタンスを停止することができます。このようなスイッチオーバーの結果、数秒の停止時間でアップグレードされます。

17.7. サーバのなりすまし防止

サーバが稼動中、悪意のあるユーザが通常のデータベースサーバに取って代わることはできません。しかし、サーバが停止している時、ローカルユーザに対し、独自のサーバを起動させることで正常なサーバになりすますことは可能です。なりすましたサーバで、クライアントから送信されたパスワードを読み取ることも問い合わせを読み取ることも可能です。しかし、PGDATAディレクトリの安全性はディレクトリの権限により維持されていますので、データを返すことはできません。誰もがデータベースサーバを起動させることができるため、なりすましは可能です。特殊な設定がなされていないければ、クライアントは無効なサーバであることを識別できません。

local接続に対してなりすましを防ぐ、もっとも簡単な方法は、信頼できるローカルユーザのみに書き込み権限を付与したUnixドメインソケットディレクトリ ([unix socket directory](#)) を使用することです。これにより、悪意のあるユーザがそのディレクトリに独自のソケットを作成することを防ぐことができます。一部のアプリケーションがソケットファイルのために /tmp を参照し、なりすましに対して脆弱であるかもしれないと気にするならば、オペレーティングシステムの起動時に、再割り当てされたソケットファイルを指し示す /tmp/.s.PGSQL.5432 というシンボリックリンクを作成してください。また、このシンボリックリンクが削除されることを防ぐために、/tmp を整理するスクリプトを変更する必要があるかもしれません。

TCP接続のなりすましを防ぐための最善の方法は、SSL証明書を使用し、クライアントがサーバの証明書を確実に検査することです。これを行うためには、サーバはhostssl接続 ([項19.1](#)) のみを受け付け、SSLキーと証明書ファイル ([項17.9](#)) を持たせる必要があります。TCPクライアントはsslmode=verify-caもしくはverify-fullを使用して接続し、また、適切なルート証明書ファイルをインストールしなければなりません ([項31.1](#))。

17.8. 暗号化オプション

PostgreSQLは、複数レベルの暗号化を備え、データベースサーバ自身、管理者の注意不足、安全ではないネットワークを原因とした漏洩からデータを柔軟に保護することができます。また、診療記録や財務業務など高セキュリティが求められるデータを格納する場合に暗号化が必要な場合もあります。

パスワード格納に関する暗号化

デフォルトでは、データベースユーザのパスワードはMD5ハッシュとして格納されます。ですので、ユーザに割り当てられているパスワードは管理者でも分かりません。MD5暗号化がクライアント認証に使用されている場合、平文のパスワードはサーバ上に瞬間的にも存在することはありません。ネットワーク上に流れる前にクライアントがMD5暗号化を行うからです。

特定の列に関する暗号化

`pgcrypto`モジュールにより、あるフィールドを暗号化して保存することができます。データの一部が極秘データであるような場合に有用です。クライアントが提供した復号化用のキーで、サーバ上のデータを復号化し、クライアントに返されます。

復号化されたデータと復号化用のキーは、復号処理中およびクライアントサーバ間の通信中サーバ上に存在します。このため、データベースサーバへのすべてのアクセス権限を持つユーザ（例えばシステム管理者）によって、データとキーが盗まれる瞬間があります。

データパーティションに関する暗号化

Linuxでは"loopback device"を使用してファイルシステムの最上位で暗号化を行うことができます。これにより、ファイルシステムパーティション全体をディスク上で暗号化することができます。これはオペレーティングシステムにより復号化されます。これと同等の機能は、FreeBSDではGEOMベースのディスク暗号化、`gbde`と呼ばれます。また、Windowsを含む他のオペレーティングシステムの多くでもこうした機能をサポートします。

この機構により、ディスク装置やコンピュータ全体が盗まれた場合でも、ディスクから平文のデータが読み取られることを防止できます。ファイルシステムがマウントされている時は、この機構による保護は働きません。マウント時にはオペレーティングシステムが復号化したデータを提供するからです。しかし、ファイルシステムをマウントするためになんらかの方法で暗号化キーをオペレーティングシステムに渡さなければなりません。そのディスク装置をマウントするホストのどこかに暗号化キーを格納している場合もあります。

ネットワーク越しのパスワード暗号化

MD5認証方式は、サーバにパスワードを送信する前に、パスワードを二重に暗号化します。まず、ユーザ名を元にパスワードをMD5で暗号化します。そして、データベース接続が確立した時に、ランダムなソルトを使用して更に暗号化します。この二重に暗号化された値が、ネットワーク越しにサーバへ渡されます。二重の暗号化により、パスワードの盗聴を防ぐだけでなく、将来確立する別の接続で同じ暗号化値が使用されなくなります。

ネットワーク越しのデータ暗号化

SSL接続により、ネットワーク越しに送信されるデータ（パスワード、問い合わせ、結果のデータ）はすべて暗号化されます。`pg_hba.conf`ファイルを使用して、管理者はどのホストは暗号化しない接続を使用し（`host`）、どのホストがSSLで暗号化された接続を必要とするか（`hostssl`）を指定することができます。また、SSL経由のサーバとの接続のみを使用するようにクライアントで指定することもできます。StunnelやSSHを使用して暗号化転送を行うこともできます。

SSLホスト認証

クライアントとサーバの両方で証明書を互いに提供することができます。これには両方で追加の設定を行わなければなりませんが、これにより単なるパスワードの使用よりも強力な身元の検証を行うことができます。クライアントから送信されたパスワードを盗聴する偽装サーバからコンピュータを保護します。また、クライアントとサーバとの間にあるコンピュータがサーバになりすまし、クライアントとサーバ間で流れるデータを読み取り中継する、"中間者"攻撃から保護することもできます。

クライアントサイドの暗号化

サーバマシンのシステム管理者を信頼できない場合、クライアント側でデータを暗号化する必要があります。この場合、平文のデータはデータベースサーバ上に存在しません。データはサーバに送信される前にクライアント上で暗号化されます。また、使用する前にデータベースからの結果をクライアントで復号化しなければなりません。

17.9. SSLによる安全なTCP/IP接続

PostgreSQLは標準でSSL接続をサポートし、クライアント/サーバの通信がさらに安全になるよう暗号化します。そのためにはOpenSSLがクライアントとサーバシステムの両方にインストールされ、構築時にPostgreSQLにおけるそのサポートが有効になっている必要があります（[第15章](#)を参照してください）。

SSLサポートを有効にしてコンパイルされた場合、PostgreSQLサーバは、`postgresql.conf`において`ssl`パラメータを`on`にすることで、SSLサポートを有効にして起動することができます。サーバは同じTCPポートで通常の接続とSSL接続の両方を待ち受け、クライアントとの接続にSSLを使用するかどうかを調停します。デフォルトでは、これはクライアント側の選択肢です。一部またはすべての接続でSSLの使用を必要とさせるためのサーバ側の設定方法に関しては[項19.1](#)を参照してください。

PostgreSQLは、システム全体用のOpenSSL設定ファイルを読み取ります。デフォルトでは、このファイルの名前は`openssl.cnf`であり、`openssl version -d`で報告されるディレクトリに存在します。このデフォルトは、環境変数`OPENSSL_CONF`を希望する設定ファイルの名前に設定することにより変更可能です。

OpenSSLは、強度が異なる、多くの暗号化および認証用のアルゴリズムをサポートします。暗号の一覧はOpenSSL設定ファイル内で指定することができますが、`postgresql.conf`内の`ssl_ciphers`を変更することで、データベースサーバで使用される暗号を指定することができます。

注意: `NULL-SHA`または`NULL-MD5`暗号を使用して暗号化のオーバーヘッドなしで認証を行うことが可能です。しかし、中間者はクライアントサーバ間の通信を読み取り中継することができます。また、暗号化のオーバーヘッドは認証のオーバーヘッドと比べると最小です。こうした理由によりNULL暗号は推奨しません。

SSLモードで起動するには、サーバ証明書と秘密鍵を含むファイルが存在していなければなりません。デフォルトでは、これらのファイルは`server.crt`および`server.key`という名前で、それぞれがサーバのデータディレクトリに存在していることが想定されていますが、設定パラメータの`ssl_cert_file`と`ssl_key_file`によって他の名前、他の場所を指定することもできます。Unixシステムでは、`server.key`の権限は所有者以外からのアクセスを許可してはなりません。これは`chmod 0600 server.key`というコマンドで実現できます。秘密キーがパスフレーズで保護されている場合、サーバはパスフレーズの入力を促し、入力されるまでは起動しません。

サーバ証明書がクライアントで直接信頼している認証局ではなく、"中間"認証局により署名されている場合があります。こうした証明書を使用するために、`server.crt`ファイルに署名した認証局の証明書と、クライアントで直接信頼している"ルート"認証局までその親となる認証局の証明書を追加してください。`server.crt`に複数の証明書が含まれる場合はすべて、ルート証明書を含めなければなりません。

17.9.1. クライアント証明書の使用

クライアントに信頼できる証明書を要求するためには、信頼する認証局（CA）の証明書をデータディレクトリ内の`root.crt`ファイルに記述し、`postgresql.conf`の`ssl_ca_file`パラメータで`root.crt`を指定し、`pg_hba.conf`内の適切な`hostssl`行（複数もあり）に`clientcert`パラメータを1に設定します。その後、SSL接続起動時にクライアントからの証明書が要求されます。（クライアントにある証明書の設定方法については[項31.18](#)を参照してください。）サーバは、クライアントの証明書が信頼する認証局のいずれかにより署名されていることを検証します。`ssl_crl_file`パラメータが設定されている場合、証明失効リスト（CRL）項目も検査されます。（SSL証明書の使用方法を示す図についてはhttp://h71000.www7.hp.com/DOC/83final/BA554_90007/ch04s02.htmlを参照してください。）

`pg_hba.conf`の中の`clientcert`オプションはすべての認証方式に対して有効ですが、`hostssl`として指定された行のみに対してです。`clientcert`が指定されていない、または0と設定されている場合でも、認証局のリストが設定されていれば、サーバはその認証局に対してクライアント証明書の検証を行います、クライアント証明書の存在そのものを要求しません。

`root.crt`は、クライアント証明書の署名に対して信頼できるとみなしている最上位のCAを列挙していることに注意してください。原理的には、サーバの証明書を署名したCAを列挙する必要はありませんが、ほとんどの場合、そのCAはクライアント証明書でも信頼されています。

クライアント証明書を設定している場合、接続の安全性を提供するとともに証明書でユーザ認証を制御できるように`cert`認証方式を使用したいと考えるかもしれません。詳細については[項19.3.10](#)を参照してください。

17.9.2. サーバにおけるSSL関連ファイルの利用

[表17-3](#)にて、サーバにおけるSSLの設定に関連するファイルをまとめます。（表示されているファイル名はデフォルトまたは一般的な名前です。異なる名前を個別に設定することもできます。）

表 17-3. SSLサーバファイルの使用方法

ファイル	内容	影響
ssl_cert_file (\$PGDATA/server.crt)	サーバ証明書	サーバの身元を示すためにクライアントに送信します
ssl_key_file (\$PGDATA/server.key)	サーバの秘密キー	サーバ証明書が所有者によって送られたことを証明します。証明書所有者が信頼できることを意味しません。
ssl_ca_file (\$PGDATA/root.crt)	信頼できる認証局	信頼する認証局により署名されたクライアント証明書が検査します。
ssl_crl_file (\$PGDATA/root.crl)	認証局により失効された証明書	クライアント証明書はこの一覧にあってははいけません。

server.key、server.crt、root.crt、root.crlファイル（またはその他の設定されたファイル名）は、サーバ起動時にのみ検査されます。 ですので、これらのファイルの変更を有効にするためにはサーバを再起動する必要があります。

17.9.3. 自己署名証明書の作成

サーバ用の自己署名証明書を簡単に作るためには下記のOpenSSLコマンドを使ってください。

```
openssl req -new -text -out server.req
```

opensslから出される質問に答えてください。 この時、"Common Name"には確実にローカルホスト名を入力してください。 チャレンジパスワードは空白でも構いません。 このプログラムはパスフレーズで保護されたキーを生成しますが、4文字以下のパスフレーズは認められません。 パスフレーズを削除するためには（サーバの自動起動を行いたいのであれば）、下記のコマンドを実行してください。

```
openssl rsa -in privkey.pem -out server.key
rm privkey.pem
```

既存のキーのロックを外すために、古いパスフレーズを入力します。 そして、下記を実行してください。

```
openssl req -x509 -in server.req -text -key server.key -out server.crt
```

このように、証明書を自己署名の証明書にして、キーと証明書とをサーバが検索する場所にコピーします。 サーバは、もしこのファイルがこれよりもゆるい権限の場合拒否するので、最後に以下を行います。

```
chmod og-rwx server.key
```

サーバの秘密キーおよび証明書を作成するための詳しい方法についてはOpenSSLの文書を参照してください。

自己署名証明書を試験のために使用することはできますが、クライアントがサーバの身元を検証できるように、運用時は（グローバルなCAの1つまたはローカルな認証局のいずれかの）認証局（CA）により署名された証明書を使用すべきです。 もしすべてのクライアントが組織においてローカルであれば、ローカルCAの使用が推奨されます。

17.10. SSHトンネルを使った安全なTCP/IP接続

クライアントとPostgreSQLサーバ間のネットワーク接続を暗号化するためにSSHを使うことができます。正しく行えば、SSL機能がクライアントになくても、これで十分に安全なネットワーク接続を行うことができます。

まずSSHサーバがPostgreSQLサーバと同じマシン上で正しく起動していて、`ssh`を使ってログインできるユーザが存在することを確認してください。そうすると、次に示すコマンドでクライアントマシンから安全なトンネルを確立することができます。

```
ssh -L 63333:localhost:5432 joe@foo.com
```

`-L`引数の1番目の数字（63333）はトンネルのローカル側のポート番号で、未使用のポートを選択することが可能です。（IANAは49152から65535までのポートを私的使用のため予約しています。）2番目の数字（5432）は、トンネルのリモート側のサーバが使用しているポート番号です。2つのポート番号間の名前もしくはアドレスは、ログインしようとしているホストから見られる接続しようとしているデータベースサーバのホストで、例の中の`foo.com`です。このトンネルを使ってデータベースサーバに接続するためには、ローカルマシンのポート63333に接続します。

```
psql -h localhost -p 63333 postgres
```

データベースサーバにとっては、ユーザが実際にホスト`foo.com`上のユーザ`joe`であるかのように見え、このユーザとホストによる接続向けに設定された認証手続きが使用されます。サーバはこの接続がSSLで暗号化されているとみなさないことに注意してください。実際、SSHサーバとPostgreSQLサーバとの間は暗号化されません。この2つのサーバが同一マシンにある限り、セキュリティ上の危険性が増すことはありません。

トンネルの確立が成功するためには、`ssh`を使用して端末セッションを作成したのと同様に、`joe@foo.com`ユーザが`ssh`を通して接続することが許可されていなければいけません。

以下に示すようにポートフォワードを設定することができます。

```
ssh -L 63333:foo.com:5432 joe@foo.com
```

しかしそうすると、データベースサーバはその`foo.com`インタフェースから接続が来たように判断し、デフォルトの設定である`listen_addresses = 'localhost'`では開かれません。通常これは好ましいことではありません。

どこかのログインホスト経由でデータベースサーバに"跳躍"しなければならない場合、以下のようにすることが可能です。

```
ssh -L 63333:db.foo.com:5432 joe@shell.foo.com
```

`shell.foo.com`から`db.foo.com`へのこのような接続はSSHトンネルで暗号化されません。SSHはいろいろな方法でネットワークが制約されているとき、かなりの数の設定可能性を提供しています。詳細はSSHの文書を参照してください。

ティップ: ここで説明してきたものと似た概念の手続きを使用して、安全なトンネルを提供可能なアプリケーションが他にもいくつか存在します。

17.11. WindowsにおけるEvent Logの登録

Windowsの OSのevent logライブラリに登録するには、以下のコマンドを発行します:

```
regsvr32 pgsql_library_directory/pgevent.dll
```

このコマンドは、PostgreSQLというデフォルトのイベントソース名で、イベントビューアが使用するレジストリエントリを作成します。

異なるイベントソース名 ([event_source](#)参照) を指定するには、/nおよび/iオプションを使ってください:

```
regsvr32 /n /i:event_source_name pgsql_library_directory/pgevent.dll
```

OSからevent logライブラリを削除するには、以下のコマンドを発行します:

```
regsvr32 /u [/i:event_source_name] pgsql_library_directory/pgevent.dll
```

注意: データベースサーバにおけるイベントロギングを有効にするには、eventlogを含むようにpostgresql.confの[log_destination](#)を変更してください。

第 18 章 サーバの設定

目次

- 18.1. [パラメータの設定](#)
 - 18.1.1. [パラメータ名とその値](#)
 - 18.1.2. [設定ファイルによるパラメータの設定](#)
 - 18.1.3. [パラメータ設定を行う他の手段](#)
 - 18.1.4. [パラメータ設定の検証](#)
- 18.2. [ファイルの場所](#)
- 18.3. [接続と認証](#)
 - 18.3.1. [接続設定](#)
 - 18.3.2. [セキュリティと認証](#)
- 18.4. [資源の消費](#)
 - 18.4.1. [メモリ](#)
 - 18.4.2. [ディスク](#)
 - 18.4.3. [カーネル資源使用](#)
 - 18.4.4. [コストに基づく Vacuum 遅延](#)
 - 18.4.5. [バックグラウンドドライタ](#)
 - 18.4.6. [非同期動作](#)
- 18.5. [ログ先行書き込み \(WAL\)](#)
 - 18.5.1. [諸設定](#)
 - 18.5.2. [チェックポイント](#)
 - 18.5.3. [アーカイビング](#)
- 18.6. [レプリケーション](#)
 - 18.6.1. [送出サーバ群](#)
 - 18.6.2. [マスターサーバ](#)
 - 18.6.3. [スタンバイサーバ](#)
- 18.7. [問い合わせ計画](#)
 - 18.7.1. [プランナメソッド設定](#)
 - 18.7.2. [プランナコスト定数](#)
 - 18.7.3. [遺伝的問い合わせ最適化](#)
 - 18.7.4. [その他のプランナオプション](#)
- 18.8. [エラー報告とログ取得](#)
 - 18.8.1. [ログの出力先](#)
 - 18.8.2. [いつログを取得するか](#)
 - 18.8.3. [何をログに](#)
 - 18.8.4. [CSV書式のログ出力の利用](#)
- 18.9. [実行時統計情報](#)
 - 18.9.1. [問い合わせおよびインデックスに関する統計情報コレクタ](#)
 - 18.9.2. [統計情報の監視](#)
- 18.10. [自動 Vacuum 作業](#)
- 18.11. [クライアント接続デフォルト](#)
 - 18.11.1. [文の動作](#)
 - 18.11.2. [ロケールと書式設定](#)
 - 18.11.3. [その他のデフォルト](#)
- 18.12. [ロック管理](#)

18.13. [バージョンとプラットフォーム互換性](#)

18.13.1. [以前のPostgreSQLバージョン](#)

18.13.2. [プラットフォームとクライアント互換性](#)

18.14. [エラー処理](#)

18.15. [設定済みのオプション](#)

18.16. [独自のオプション](#)

18.17. [開発者向けのオプション](#)

18.18. [短いオプション](#)

データベースシステムの動作に影響を与える数多くのパラメータがあります。この章の最初の節で、どのように設定パラメータを定めるかについて説明します。引き続き節で、それぞれのパラメータの詳細を説明します。

[前のページ](#)

WindowsにおけるEvent Logの登録

[ホーム](#)

[上に戻る](#)

[次のページ](#)

パラメータの設定

18.1. パラメータの設定

18.1.1. パラメータ名とその値

全てのパラメータの名前は大文字と小文字を区別しません。それぞれのパラメータは、論理値、整数、浮動小数点、文字列、またはenum（列挙型）の5つの型のいずれかの値を取ります。論理値は、on、off、true、false、yes、no、1、0、または、これらとすぐに推測できるどんな接頭文字でも記述することができます（全て大文字小文字の区別はありません）。

一部の設定では、メモリや時刻に関する値を指定します。それぞれは暗黙的な単位を持ちます。キロバイト、ブロック（通常8キロバイト）、ミリ秒、秒、分などです。デフォルトの単位はpg_settings.unitを参照することで調べることができます。便宜上、別の単位を明示的に指定することができます。メモリに関する単位では、kB（キロバイト）、MB（メガバイト）、GB（ギガバイト）が有効です。時刻に関する単位ではms（ミリ秒）、s（秒）、min（分）、h（時間）、d（日数）が有効です。メモリ単位の乗数は1000ではなく1024であることに注意してください。

"enum（列挙）"型のパラメータは文字列パラメータと同様の方法で指定されますが、限定された値の集合に制限されます。有効な値はpg_settings.enumvalsにあります。列挙型パラメータ値には大文字小文字の区別がありません。

18.1.2. 設定ファイルによるパラメータの設定

これらのパラメータを設定する1つの方法は、postgresql.confファイルを編集することで、これは通常 data ディレクトリに格納されています。（デフォルトのコピーはデータベースクラスタディレクトリが初期化されるときそこにインストールされます。）このファイルがどういったもののかの例を示します。

```
# This is a comment
log_connections = yes
log_destination = 'syslog'
search_path = '$user', public
shared_buffers = 128MB
```

1つの行毎に1つのパラメータが指定されます。名前と値の間の等号はオプションです。空白（white space）は特に意味を持たず、何も無い行は無視されます。何処にあってもハッシュ記号（#）はその行の後の表記が注釈であることを意味します。単純でない識別子、または数値でないパラメータ値は単一引用符で括られなければなりません。パラメータ値の中に単一引用符を埋め込む場合、二重引用符（推奨）もしくは逆引用符で括ります。

パラメータ設定の他、postgresql.confにはinclude 指示子があります。この指示子は、設定ファイルの当該位置に指定したファイルが挿入されているかのように、そのファイルを読み込み処理することを指定します。この機能により構成ファイルは物理的に個別の部分に分割できます。include指示子は以下のような単純なものです。

```
include 'filename'
```

ファイル名が絶対パスでなければ、参照元の設定ファイルがあるディレクトリからの相対パスとみなされます。includeは入れ子にすることができます。

同時に include_if_exists 指示子があります。これは参照されたファイルが存在しないか、読み取れない場合を除いて include 指示子と同じ振る舞いを行います。通常の include はこの状態をエラーと見做しますが、include_if_exists はメッセージをログするだけで参照している設定ファイルの処理を継続します。

設定ファイルはメインサーバプロセスが（pg_ctl reloadにより、一番簡単に送られる）SIGHUP信号を受け取る時は常に再読み込みされます。メインサーバプロセスは同時にこの信号を、現存のセッションが同様に新しい値を入手できるように、全ての現在実行しているサーバプロセスに伝播します。他の手段として、直接単一のサーバプロセスを送出することも可能です。いくつかのパラメータはサーバの起動時のみ設定されます。設定ファイル中のそれらのエントリのいかなる変更も、サーバが再起動されるまで無視されます。設定ファイル内で無効なパラメータが設定された場合はこのように（ログには残りますが）SIGHUP 処理中に無視されます。

18.1.3. パラメータ設定を行う他の手段

これら設定パラメータを設定する第二の方法は、以下のようにpostgresコマンドに対してコマンドラインオプションとして与えます。

```
postgres -c log_connections=yes -c log_destination='syslog'
```

コマンドラインオプションはpostgresql.confにある、どんな矛盾した設定をも上書きします。これが意味することは、postgresql.confを編集するだけでは、値をすぐに変更できず、従ってコマンドラインによる方法は便利とは言っても、後に

柔軟性に犠牲を払う羽目に陥ることを覚えておいてください。

場合によっては、1つの特定セッションのみにコマンドラインオプションを与えることが便利です。環境変数 `PGOPTIONS`はこの目的のためにクライアント側で使用できます。

```
env PGOPTIONS='-c geqo=off' psql
```

(これは、単にpsqlだけではなく、全てのlibpqに基づいたクライアントアプリケーションに対して有効です。) これは、サーバが起動された時に修正されたパラメータ、または、`postgresql.conf`で指定されなければならないパラメータに対しては効果がないことを覚えておいてください。

更に、ユーザもしくはデータベースにパラメータ設定のひとそろいを割り当てることができます。セッションが開始された時はいつでも、ユーザとデータベースに関連したデフォルトの設定が読み込まれます。[ALTER USER](#)および[ALTER DATABASE](#)コマンドは、それぞれこれらの設定を設定するために用いられます。データベース毎の設定は、`postgres`コマンドライン、または設定ファイルから受け取った何によっても上書きされます。そして順次ユーザ毎の設定が上書きします。これらは共にセッション毎の設定で上書きされます。

個別のSQLセッションのいくつかのパラメータは[SET](#)コマンドで変更可能です。例えば：

```
SET ENABLE_SEQSCAN TO OFF;
```

`SET`が使える場合には、そのパラメータに対して設定された値が上書きされます。いくつかのパラメータは `SET` では変更できません。例えば、そのパラメータがPostgreSQLサーバ全体を再起動しなければ変更できない動作を制御している場合です。さらに、いくつかのパラメータは`SET`または`ALTER`による修正にスーパーユーザ権限を必要とします。

18.1.4. パラメータ設定の検証

[SHOW](#)コマンドで全てのパラメータの現在値を調査することができます。

仮想テーブル`pg_settings`もセッション実行時パラメータの表示と更新を可能にします。異なった変数型と、いつそれらが変更可能かについての詳細と説明は [項45.64](#) を参照してください。 `pg_settings`は`SHOW`と`SET`と等価です。しかし、他のテーブルとの結合や、目的とする条件を指定した検索ができるので、より使い勝手に優れています。更にそれぞれのパラメータについて`SHOW`よりも多くの情報が入っています。

[前のページ](#)
サーバの設定

[ホーム](#)
[上に戻る](#)

[次のページ](#)
ファイルの場所

18.2. ファイルの場所

すでに説明した`postgresql.conf`ファイルに加え、PostgreSQLは、（[第19章](#)で使用方法が説明されている）クライアント認証の管理を行うために、他の2つの手作業で編集される設定ファイルを使用します。全ての3つの設定ファイルは、デフォルトではデータベースクラスタの`data`ディレクトリに格納されます。本節で説明するパラメータにより、設定ファイルを何処にでも置くことが可能です。（そのようにすると管理がしやすくなります。具体的には、設定ファイルを分けて保存することで、設定ファイルの適切なバックアップを確実に行うことが容易になります。）

`data_directory (string)`

データ格納に使用するディレクトリを指定します。このパラメータはサーバ起動時のみ設定可能です

`config_file (string)`

メインサーバ設定ファイルを指定します（通例`postgresql.conf`と呼ばれます）。このパラメータは`postgres`コマンドライン上でのみ設定可能です。

`hba_file (string)`

ホストベース認証（HBA）用のファイルを指定します（通例`pg_hba.conf`と呼ばれます）。このパラメータはサーバ起動時のみ設定可能です。

`ident_file (string)`

[項19.2](#)ユーザ名マッピングの設定ファイルを指定します（通例`pg_ident.conf`と呼ばれます）。このパラメータはサーバ起動時のみ設定可能です。

`external_pid_file (string)`

サーバ管理プログラムの使用のためにサーバが作成しなくてはならない、追加のプロセス識別子（PID）ファイルの名前を指定します。このパラメータはサーバ起動時のみ設定可能です。

デフォルトのインストールでは、上記のいかなるパラメータも明示的に設定されません。その代わりに、`data` ディレクトリは`-D`コマンドラインオプション、または`PGDATA`環境変数で指定され、設定ファイル全てはその`data` ディレクトリ内に格納されます。

`data`ディレクトリ以外の場所に設定ファイルを格納したいのであれば、`postgres`の`-D`コマンドラインオプション、または`PGDATA`環境変数で設定ファイルの場所を指し示し、そして`data`ディレクトリが実際どこに存在するのを示すため、`postgresql.conf`の（もしくはコマンドライン上で）`data_directory`パラメータを設定しなければなりません。`data_directory`は、設定ファイルの場所ではなく、`data` ディレクトリの位置に関して、`-D`および`PGDATA`を上書きすることに注意してください。

必要に応じて、パラメータ`config_file`、`hba_file`、そして/もしくは`ident_file`を使用し、設定ファイルの名前と場所を個別に指定することができます。`config_file`は`postgres`コマンドラインによってのみ指定されますが、その他は主設定ファイル内で設定できます。全ての3つのパラメータと`data_directory`が明示的に設定されていれば、`-D`または`PGDATA`を指定する必要はありません。

これらのパラメータのいずれかを設定する場合、相対パスは、`postgres`が起動されるディレクトリから見た相対パスとして解釈されます。

18.3. 接続と認証

18.3.1. 接続設定

`listen_addresses (string)`

クライアントアプリケーションからの接続をサーバが監視する TCP/IP アドレスを指定します。この値は、ホスト名をコンマで区切ったリスト、そして/もしくは、数値によるIPアドレスです。*という特別なエントリは利用可能な全てのIPインタフェースに対応します。エントリ0.0.0.0は全てのIPv4アドレスの監視を、そしてエントリ::は全てのIPv6アドレスの監視を許容します。リストが空の場合、サーバはいかなるIPインタフェースも全く監視しないで、Unixドメインソケットのみを使用して接続が行われます。デフォルトの値はlocalhostで、ローカルなTCP/IP "loopback"接続のみ許可します。クライアント認証 ([第19章](#))は誰がサーバにアクセス可能かをきめ細かく制御するのに対し、`listen_addresses`はどのインタフェースが接続を試みるかを制御します。これにより、安全でないネットワークインタフェース上において繰り返して行われる悪意のある接続要求の防止に役立ちます。このパラメータはサーバ起動時のみ設定可能です。

`port (integer)`

サーバが監視するTCPポートで、デフォルトは 5432です。サーバが監視する全てのIPアドレスに対し、同じポート番号が使用されることを覚えておってください。このパラメータはサーバ起動時のみ設定可能です。

`max_connections (integer)`

データベースサーバに同時接続する最大数を決定します。デフォルトは典型的に100接続ですが、カーネルの設定が (initdbの過程で) それをサポートしていない場合、もっと少なくなることがあります。このパラメータはサーバ起動時のみに設定可能です。

このパラメータを増加させると、使用しているオペレーティングシステムのデフォルト構成が許容するSystem V共有メモリ、またはセマフォの限界を越えた要求を PostgreSQLが要求する原因となることがあります。必要であれば、このパラメータの調整の仕方について[項17.4.1](#)を参照ください。

スタンバイサーバを運用している場合、このパラメータはマスターサーバでの設定と同じ、もしくはより高い値に設定しなければなりません。そうしないと問い合わせがスタンバイサーバ内で受け入れられません。

`superuser_reserved_connections (integer)`

PostgreSQLのスーパーユーザによる接続のために予約されている接続"開口部 (スロット)"の数を決定します。最大、[max_connections](#)の数までの接続を同時に有効にすることができます。何時の時点にあっても、有効な接続数は、少なくとも`max_connections`から`superuser_reserved_connections`を差し引いた数であって、新規接続はスーパーユーザのみが許可され、新たなレプリケーション接続は受け入れられません。

デフォルトの値は3です。この値は `max_connections` での値より小さくなくてはなりません。このパラメータはサーバ起動時のみ設定可能です。

`unix_socket_directory (string)`

サーバがクライアントアプリケーションからの接続要求を監視するUnixドメインソケットのディレクトリを指定します。デフォルトは通常/tmpですが、構築時に変更できます。このパラメータはサーバ起動時のみ設定可能です。

ソケットファイル、これはnnnnがポート番号である、`.s.PGSQL.nnnn`と名前が付いているもの、それ自身に加え、`.s.PGSQL.nnnn.lock`と名前が付いている通常のファイルが`unix_socket_directory`ディレクトリの中に作成されることがあります。いずれのファイルも手作業で削除してはいけません。

Unixドメインソケットを持たないWindowsではこのパラメータは無関係です。

`unix_socket_group (string)`

Unixドメインソケットを所有するグループを設定します (ソケットを所有するユーザは常にサーバを起動するユーザです)。`unix_socket_permissions`パラメータとの組合せで、Unixドメインソケット接続の追加的アクセス管理機構として使うことができます。デフォルトでは空文字列で、どのユーザがサーバユーザのデフォルトグループかを示します。このパラメータはサーバ起動時のみ設定可能です。

Unixドメインソケットを持たないWindowsではこのパラメータは無関係です。

`unix_socket_permissions (integer)`

Unix ドメインソケットのアクセス許可を設定します。 Unix ドメインソケットは通常のUnix ファイルシステム許可設定の一式を使用します。 パラメータ値は、`chmod` および `umask` システムコールが受け付ける数値形式での指定を想定しています。（通常使われる8進数形式を使用するのであれば、0（ゼロ）で始まらなければなりません。）

デフォルトでの許可は、誰でも接続できる`0777`になっています。 変更するならば`0770`（ユーザとグループのみです。 `UNIX_SOCKET_GROUP` も参照してください）や`0700`（ユーザのみ）が適切です。（実際、Unix ドメインソケットでは書き込み権限だけが問題です。そのため、読み込み許可や実行許可を設定または解除する意味はありません。）

のアクセス制御機構は [第19章](#)で記述されたものとは別個のものです。

このパラメータはサーバ起動時のみ設定可能です。

Unix ドメインソケットを持たないWindowsではこのパラメータは無関係です。

`bonjour (boolean)`

Bonjourによりサーバの存在を公表することを可能にします。デフォルトは`off`です。このパラメータはサーバ起動時のみ設定可能です。

`bonjour_name (string)`

Bonjourサービス名を指定します。 このパラメータが空文字列''（デフォルトです）に設定されていると、コンピュータ名が使用されます。 サーバがBonjourサポート付でコンパイルでされていない場合は無視されます。 このオプションはサーバ起動時のみに設定可能です。

`tcp_keepalives_idle (integer)`

別の待機接続にkeepaliveを送信するまでの秒単位の時間を指定します。0の場合はシステムのデフォルト値を使用します。このパラメータは`TCP_KEEPIDL`または`TCP_KEEPA`シンボルをサポートするシステムとWindowsのみサポートされます。 その他のシステムではゼロでなければなりません。 Unix ドメインソケット経由で接続されたセッションでは、このパラメータは無視され、常にゼロとして読み取られます。

注意: Windowsでは値0はこのパラメータを2時間に設定します。なぜなら、Windowsはシステムデフォルト値を読む手段を提供していないからです。

`tcp_keepalives_interval (integer)`

別の待機接続にkeepaliveを送信するまでの秒単位の時間を指定します。 0の場合はシステムのデフォルト値を使用します。 このパラメータは`TCP_KEEPI`シンボルをサポートするシステムとWindowsのみサポートされます。 その他のシステムではゼロでなければなりません。 Unix ドメインソケット経由で接続されたセッションでは、このパラメータは無視され、常にゼロとして読み取られます。

注意: Windowsでは値0はこのパラメータを1秒に設定します。なぜなら、Windowsはシステムデフォルト値を読む手段を提供していないからです。

`tcp_keepalives_count (integer)`

別の待機接続にkeepaliveパケットを送信する数を指定します。0の場合はシステムのデフォルト値を使用します。 このパラメータは`TCP_KEEPC`シンボルをサポートするシステムのみサポートされます。その他のシステムではゼロでなければなりません。 Unix ドメインソケット経由で接続されたセッションでは、このパラメータは無視され、常にゼロとして読み取られます。

注意: このパラメータはWindowsではサポートされておらず、ゼロでなければなりません。

18.3.2. セキュリティと認証

`authentication_timeout (integer)`

秒単位による、クライアント認証を完了するまでの最大時間です。 もし、この時間内に自称クライアントが認証プロトコルを完了しない場合、サーバは接続を閉じます。 これはハングしたクライアントが接続を永久に占有することを防ぎます。 デフォルトは1分（1m）です。 このパラメータは`postgresql.conf` ファイル、またはサーバのコマンドラインでのみ設定可能です。

`ssl (boolean)`

SSLによる接続を有効にします。これを使用する前に[項17.9](#)をお読みください。デフォルトは`off`です。このパラメータ

はサーバ起動時のみ設定可能です。SSL通信はTCP/IP接続の場合のみ有効です。

`ssl_ca_file (string)`

SSLサーバ認証局 (CA) の名称が入っているファイル名を設定します。デフォルトは空です。つまりCAファイルは読み込まれず、クライアントのサーバ検証は行われません。(以前のPostgreSQLリリースでは、このファイルは `root.crt` としてハードコードされていました。) 相対パスはそのデータディレクトリと相対関係にあります。このパラメータはサーバ起動時のみ設定可能です。

`ssl_cert_file (string)`

SSLサーバ証明書が入っているファイル名を設定します。デフォルトは `server.crt` です。相対パスはそのデータディレクトリと相対関係にあります。このパラメータはサーバ起動時のみ設定可能です。

`ssl_crl_file (string)`

SSLサーバ証明書失効リスト (CRL) が入っているファイル名を設定します。デフォルトは空です。つまりCRLファイルは読み込まれません。(以前のPostgreSQLリリースでは、このファイルは `root.crt` としてハードコードされていました。) 相対パスはそのデータディレクトリと相対関係にあります。このパラメータはサーバ起動時のみ設定可能です。

`ssl_key_file (string)`

SSLサーバの秘密鍵が入っているファイル名を設定します。デフォルトは `server.key` です。相対パスはそのデータディレクトリと相対関係にあります。このパラメータはサーバ起動時のみ設定可能です。

`ssl_renegotiation_limit (integer)`

そのセッションキーの再調停が行われる前に、SSL暗号化接続上で、どれだけの量のデータのが流れても良いかを指定します。再調停は攻撃者が大量のトラフィックを送出した場合、暗号解析を行う機会を減少させます。しかし、同時に多大な性能上の不利益をもたらします。送出されたのと受け取られたトラフィックの合計はその限度の検証に使用されます。もしパラメータが0に設定されると再調停は無効になります。デフォルトは512MBです。

注意: SSLプロトコルにおける脆弱性によりSSL再調停を使用した場合、2009年11月以前のSSLライブラリは安全ではありません。この脆弱性への応急的な対処として、いくつかのベンダーは再調停を行う機能がないSSLライブラリを出荷しました。もしこれらのライブラリがクライアントまたはサーバで使用されているのであれば、SSL再調停は無効にすべきです。

`ssl_ciphers (string)`

安全な接続で利用できるSSL暗号のリストを指定します。サポートされる暗号のリストについてはopensslのマニュアルページを参照してください。

`password_encryption (boolean)`

ENCRYPTEDもしくはUNENCRYPTEDの指定なしで、[CREATE USER](#) もしくは[ALTER USER](#)にてパスワードが指定されている場合、このパラメータはパスワードを暗号化するか否かを決定します。デフォルトはon (パスワードを暗号化) です。

`krb_server_keyfile (string)`

Kerberosサーバの鍵ファイルの場所を設定します。詳細は[項19.3.5](#)または[項19.3.3](#)を参照してください。このパラメータは`postgresql.conf` ファイル、またはサーバのコマンドラインでのみ設定可能です。

`krb_srvname (string)`

Kerberosのサービス名を設定します。詳細は[項19.3.5](#)を参照してください。このパラメータは`postgresql.conf` ファイルか、サーバコマンドラインでのみ設定可能です。

`krb_caseins_users (boolean)`

KerberosおよびGSSAPIのユーザ名が大文字小文字を区別すべきかの設定をします。デフォルトはoff (大文字小文字を区別する) です。このパラメータは`postgresql.conf` ファイル、またはサーバのコマンドラインでのみ設定可能です。

`db_user_namespace (boolean)`

このパラメータはデータベース毎のユーザ名を可能にします。デフォルトではオフです。このパラメータは`postgresql.conf` ファイル内、またはサーバのコマンドラインのみで設定可能です。

これがオンの場合、`username@dbname` の様にしてユーザを作成しなければなりません。`username` が接続中のクライアントより渡された時、`@`およびデータベース名がユーザ名に付加され、そのデータベース特有のユーザ名をサーバが見に行き

ます。SQL環境下で@を含む名前のユーザを作成する場合、そのユーザ名は引用符で括られなければならないことに注意してください。

このパラメータを有効にしても通常の広域ユーザを作成することができます。クライアントにユーザ名を指定する時に、たとえばjoe@のように単に@を付け加えてください。@はサーバがユーザ名を検索する以前に取り去られます。

db_user_namespaceはクライアントとサーバのユーザ名の表示を区別することができます。認証検査は常にサーバのユーザ名で行われるので、認証方式はクライアントのではなくサーバのユーザ名で構成されなければなりません。md5では、クライアントおよびサーバの両方でユーザ名をソルトとして使用しますので、md5をdb_user_namespaceと一緒に使用することはできません。

注意: この機能は完全な解決策が見つかるまでの暫定的な手段です。見つかった時点でこのオプションは削除されます。

[前のページ](#)
ファイルの場所

[ホーム](#)
[上に戻る](#)

[次のページ](#)
資源の消費

18.4. 資源の消費

18.4.1. メモリ

`shared_buffers (integer)`

データベースサーバが使用する共有メモリバッファのために使用するメモリ量を設定します。デフォルトは一般的に32メガバイト(32MB)です。しかし、稼働中のカーネルの設定が (initdbの過程で決定されます) この値をサポートしていない場合、より少なくなることがあります。この設定は最低限128キロバイトなければなりません。(デフォルト値以外のBLCKSZでこの最小値は変化します。) しかし、良い性能を引き出すためには、最小値よりかなり高い値の設定が通例が必要です。このパラメータはサーバ起動時にのみ設定可能です。

1GBまたはそれより多いRAMを載せた専用データベースサーバを使用している場合、`shared_buffers`に対する妥当な初期値はシステムメモリの25%です。`shared_buffers`に対し大きな値が有効であってもなんらかの作業負荷は存在します。しかし、PostgreSQLは同時にオペレーティングシステムキャッシュを信頼しますので、`shared_buffers`にRAMの40%以上を割り当てても、より少ない量より動きがより良くなると言う見込みはありません。`shared_buffers`をより大きく設定するは、通常対応するcheckpoint_segmentsを増やす必要があります。より長い期間にわたっての新規、または変更された多量のデータを書き出すプロセスを展開するためです。

1GB以下のRAMのシステムでは、オペレーティングシステムに十分な余裕を残せるため、より少ない比率のRAMが適切です。同時に、Windowsでは`shared_buffers`に対し大きな値は有効でありません。設定を比較的少なく保ち、その代わりオペレーティングシステムのキャッシュを使用するとより良い結果が見つかります。Windowsシステムでの`shared_buffers`の範囲は一般的に64MBから512MBです。

このパラメータを増加させると、PostgreSQLは使用しているオペレーティングシステムのデフォルト構成が許容するSystem V共有メモリの限界を越えた要求を行う要因となることがあります。必要であれば、どの様にこのパラメータを調整するかについて[項17.4.1](#)を参照ください。

`temp_buffers (integer)`

それぞれのデータベースセッションが使用する一時バッファの最大数を設定します。一時テーブルにアクセスする時のみ使用されるセッション局所バッファが存在します。デフォルトは8メガバイト (8MB) です。設定はそれぞれのセッション内で変更できますが、そのセッション内の一時テーブルが最初に使用するまでになります。引き続いて値の変更を試みても、そのセッションでは効果がありません。

多くの一時バッファを実際に必要としないセッションで大きな値を設定するコストとは、`temp_buffers`の増分毎に、バッファ記述子分、バイトで言うと64バイトです。しかし、バッファが実際に使用されると、それに対して追加の8192バイト (または、通常BLCKSZバイト) が消費されます。

`max_prepared_transactions (integer)`

"プリペアド"状態におけるトランザクションの最大数を同時に設定します ([PREPARE TRANSACTION](#)を参照してください)。このパラメータをゼロ (これがデフォルトです) に設定すると、プリペアドトランザクション機能を無効にします。このパラメータはサーバ起動時にのみ設定可能です。

プリペアドトランザクションの使用を意図しないのであれば、このパラメータはプリペアドトランザクションが偶然に作成されないようゼロに設定されなければなりません。プリペアドトランザクションを使用する場合、全てのセッションがプリペアドトランザクションを保留できるように、`max_prepared_transactions`を最低[max_connections](#)と同じ大きさに設定しても構いません。

このパラメータを増加させると、使用しているオペレーティングシステムのデフォルト構成が許容するSystem V共有メモリの限界を越えた要求を PostgreSQLが行う原因となることがあります。必要であれば、どの様にしてこのパラメータを調整するかについて[項17.4.1](#)を参照ください。

スタンバイサーバを運用している場合、このパラメータはマスターサーバ上の設定よりも同等かもしくはより高水準に設定しなければなりません。そうしないと問い合わせがスタンバイサーバ内で受け入れられません。

`work_mem (integer)`

一時ディスクファイルに書き込む前に、内部並べ替えとハッシュテーブル操作が使用するメモリ容量を指定します。デフォルト値は1メガバイト (1MB) です。複雑な問い合わせの場合、いくつかの並び替えもしくはハッシュ操作が並行して実行されることに注意してください。それぞれの操作による一時メモリへの書き込み開始の前に、この値が指定するのと同じメモリ容量の使用をそれらの操作に許容します。さらに、いくつかの実行中のセッションはこれらの動作を同時に行

います。したがって、使用されるメモリの合計は、`work_mem`の数倍になります。値を選択する時には、この事実に留意することが必要です。並び替え操作は`ORDER BY`、`DISTINCT`、およびマージ結合に対して使われます。ハッシュテーブルはハッシュ結合、ハッシュに基づいた集約、および`IN`副問い合わせのハッシュに基づいた処理で使用されます。

`maintenance_work_mem (integer)`

`VACUUM`、`CREATE INDEX`、および`ALTER TABLE ADD FOREIGN KEY`の様な保守操作で使われるメモリの最大容量を指定します。デフォルト値は16メガバイト（16MB）です。1つのデータベースセッションでは、一度に1つしか上記操作はできませんし、通常インストレーションでこうした操作が同時に非常に多く発生することはありませんので、これを`work_mem`よりもかなり多めの値にしても安全です。大きい値を設定することで`vacuum`処理と、ダンプしたデータベースのリストア性能が向上します。

オートバキュームを稼働させると、最大`autovacuum_max_workers`倍ほどこのメモリーが配分されるので、デフォルトの値をあまり高く設定しないよう注意することを覚えておいてください。

`max_stack_depth (integer)`

サーバの実行スタックの最大安全深度を指定します。このパラメータの理想的な設定はカーネルにより強要される実際のスタック容量の（`ulimit -s`もしくは局所での同等の値で設定された）限界から、1メガバイト程度の安全余地を差し引いたものです。安全余地は、サーバが各ルーチンでスタック深度を検査をせず、式評価などの主要な潜在的に再帰的なルーチンの場合のみ必要となるものです。デフォルト設定は2メガバイト（2MB）で、かなり控え目で、クラッシュの危険はなさそうです。しかし、複雑な関数の実行を許容するには小さ過ぎるかも知れません。スーパーユーザのみがこの設定を変更することができます。

`max_stack_depth`を実際のカーネルの制限よりも高い値に設定した場合、暴走した再帰関数により、個々のバックエンドプロセスがクラッシュするかもしれません。PostgreSQLがカーネルの制限を決定することができるプラットフォームでは、この変数を危険な値に設定させません。しかし、すべてのプラットフォームがこの情報を提供できるわけではありません。このため、値を選ぶ時には注意が必要です。

18.4.2. ディスク

`temp_file_limit (integer)`

あるセッションが暫定ファイルとして使用できるディスクの最大容量を設定します。例えば、ソートやハッシュの一時ファイルであったり、カーソルを保持する格納ファイルです。この値を超えようとするトランザクションは無視されます。値はキロバイト単位で指定され、（デフォルトである）`-1`の場合は制限がありません。この設定はスーパーユーザのみ可能です。

この設定により、ある与えられた PostgreSQL セッションにより使用される暫定ファイルの容量がいかなる場合にも制約されます。問い合わせの実行に裏舞台で使われる暫定ファイルとは対抗するように、明示的暫定テーブルに使用されるディスク容量はこの制限に不利に作用することはありません。

18.4.3. カーネル資源使用

`max_files_per_process (integer)`

それぞれのサーバ子プロセスが同時にオープンできるファイル数の最大値をセットします。デフォルトは1000ファイルです。もしもカーネルがプロセス毎の安全制限を強要している場合、この設定を気にかける必要はありません。しかし、いくつかのプラットフォーム（特にほとんどのBSDシステム）では、もし多くのプロセス全てがそれだけ多くのファイルを開くことを試みたとした場合、実際にサポートできるファイル数より多くのファイルを開くことを許しています。もしも"Too many open files"エラーが発生した場合、この設定を削減してみてください。このパラメータはサーバ起動時にのみ設定可能です。

`shared_preload_libraries (string)`

この変数はサーバが稼働する時点で事前に読み込まなければならない1つ以上の共有ライブラリを指定します。たとえば、`'$libdir/mylib'`は`mylib.so`（一部のプラットフォームでは`mylib.sl`）をインストレーションの共有ライブラリディレクトリから事前読み込みします。二重引用符でくくられてる場合を除いて、すべてのライブラリ名は小文字に変換されます。もし1つ以上のライブラリの読み込みが必要であれば、それらの名前をコンマで区切ってください。このパラメータはサーバ起動時にのみ設定可能です。

PostgreSQLの手続き言語ライブラリはこのようにして、典型的には構文`'$libdir/plXXX'`を使用し、事前読み込みされます。ここで、`XXX`は`pgsql`、`perl`、`tcl`、もしくは`python`です。

共有ライブラリを事前読み込みすることで、ライブラリが最初に使用される時、ライブラリの立上り時間を省略できます。とは言うても、それぞれの新規サーバプロセスを開始させる時間は、そのプロセスがライブラリを使用しないとして

も、多少増加することがあります。ですから、このパラメータはほとんどのセッションで使用されそうなライブラリにのみ限定することをお勧めします。

注意: Windowsでは、サーバ起動時にライブラリを事前読み込みしても、新しいサーバプロセスの起動に要する時間は減りません。各サーバプロセスは事前に読み込まれたライブラリをすべて、再度読み込みます。しかし、`shared_preload_libraries`はWindowsホストでも有用です。共有ライブラリの中には、`postmaster`起動時にのみ特定の操作を行わなければならないものがあるためです。（例えば、共有ライブラリは、`postmaster`の起動が終わった後に実行することができない、軽量ロックや共有メモリの予約を行う必要があるかもしれません。）

指定したライブラリが存在しないと、サーバの起動に失敗します。

PostgreSQLがサポートするライブラリはすべて、互換性を保証するために検査される"マジックブロック"を持ちます。このため、この方法でPostgreSQL以外のライブラリが読み込まれることはありません。

18.4.4. コストに基づく **Vacuum** 遅延

VACUUM および **ANALYZE** コマンドの実行中、実行される各種I/O操作の予測コストを追跡し続ける内部カウンタをシステムが保守します。累積されたコストが (`vacuum_cost_limit` で指定された) 限度に達すると、操作を実行しているプロセスは `vacuum_cost_delay` で指定されたちょっとの間スリープします。その後、カウンタをリセットし、実行を継続します。

この機能の目的は、同時実行的データベース活動上のこれらコマンドのI/Oに対する影響を管理者から軽減させます。**VACUUM** および **ANALYZE** の様な保守用コマンドが即座に終了することが重要ではない事態が数多くあります。しかし、他のデータベースの操作を行うに当たって、これらのコマンドがシステムの能力に多大な障害を与えないことは通常とても重要です。コストに基づいたvacuum遅延はこれを実現するための方法を管理者に提供します。

手動で**VACUUM**コマンドを実行することができるように、デフォルトでこの機能は無効になっています。有効にするには、`vacuum_cost_delay`変数をゼロでない値に設定します。

`vacuum_cost_delay (integer)`

コストの限度を越えた場合、プロセスがスリープするミリ秒単位の時間の長さです。デフォルトの値は0で、コストに基づいたvacuum遅延機能を無効にします。正の整数はコストに基づいたvacuumを有効にします。多くのシステムで、スリープ遅延の有効な分解能は10ミリ秒です。`vacuum_cost_delay`の値の設定を10の倍数としない場合、次に大きい10の倍数に設定した結果と同一になるかもしれないことを覚えておってください。

コストに基づいたバキューム処理を使用する場合、`vacuum_cost_delay`の適切な値は通常かなり小さくなり、たいていは10または20ミリ秒になります。バキュームによるリソース消費の調整は、他のバキュームのコストパラメータを変更して行うことが最善です。

`vacuum_cost_page_hit (integer)`

共有バッファキャッシュの中のバッファにvacuumを掛ける予測コストです。バッファプールのロック、共有ハッシュテーブルの検索、およびページ内容走査のコストを示します。デフォルトの値は1です。

`vacuum_cost_page_miss (integer)`

ディスクから読み込まなければならないバッファにvacuumを掛ける予測コストです。これが示すものは、バッファプールロックの試み、共有ハッシュテーブルの参照、ディスクから目的ブロックの読み込み、そしてその内容走査です。デフォルトの値は10です。

`vacuum_cost_page_dirty (integer)`

vacuumが、先だって掃除したブロックを変更する時に果たされた予測コストです。ダーティブロックを再度ディスクに吐き出すのに必要な余分なI/Oを表します。デフォルトの値は20です。

`vacuum_cost_limit (integer)`

vacuumを掛けるプロセスをスリープさせることになる累計されたコストです。デフォルトの値は200です。

注意: 重要なロックを保有し可能なかぎり早急に完了しなければならないある種の操作があります。コストに基づいたvacuum遅延はこの様な操作では起こりません。したがって、コストの累計が指定された限度をかなり高く越える可能性があります。このような場合無駄な長い遅延を防止するため、実際の遅延は $\text{vacuum\_cost\_delay} * 4$ を上限として、以下のように計算されます。
 $\text{vacuum\_cost\_delay} * \text{accumulated\_balance} / \text{vacuum\_cost\_limit}$

18.4.5. バックグラウンドライタ

バックグラウンドライタと呼ばれる個別のサーバプロセスがあり、その機能は（新規または更新された）"ダーティ"な共有バッファの書き込みを行うことです。それは共有バッファを書き込むため、ユーザの問い合わせを処理するサーバプロセスは書き込みが発生するまでごくたまに、あるいは決して待ち状態になりません。しかし、バックグラウンドライタはI/O読み込みに対し正味の全体的増加の原因になります。その理由は、繰り返しダーティ化されるページはチェックポイント間隔で一度だけ書き出される可能性がある一方、バックグラウンドライタは同じ間隔内でダーティ化されるごとにそれを複数回書き出します。本節で説明したこのパラメータはサイト独自の必要に応じて動作を調整することに使用できます。

`bgwriter_delay (integer)`

バックグラウンドライタの動作周期間の遅延を指定します。それぞれの周期でライタは、（以下のパラメータで管理される）一部のダーティバッファの書き込みを行います。そして`bgwriter_delay`ミリ秒スリープした後、これを繰り返します。しかし、バッファプールにダーティバッファが存在しない場合、`bgwriter_delay`に係わらずより長くスリープします。デフォルトの値は200ミリ秒（200ms）です。多くのシステムで、スリープ遅延の実精度は10ミリ秒です。`bgwriter_delay`の値の設定を10の倍数としない場合、次に大きい10の倍数に設定した結果と同一になるかもしれないことを覚えておってください。このパラメータは`postgresql.conf`ファイル、または、サーバのコマンドラインで設定可能です。

`bgwriter_lru_maxpages (integer)`

それぞれの周期で、この数以上のバッファはバックグラウンドライタにより書き込まれません。ゼロに設定することでバックグラウンド書き込みは無効になります。（分離し、そして専用の補助プロセスにより管理されるチェックポイントは影響を受けません。）デフォルト値は100バッファです。このパラメータは`postgresql.conf`ファイル内、または、サーバのコマンドラインでのみで設定可能です。

`bgwriter_lru_multiplier (floating point)`

各周期で書き出されるダーティバッファ数は、最近の周期でサーバプロセスが必要とした新しいバッファ数を基にします。次の周期で必要となるバッファ数を推定するために、最近必要とされた平均が`bgwriter_lru_multiplier`と掛け合わせられます。ダーティバッファの書き出しは、同数の整理済み、再利用可能なバッファが利用できるようになるまで行われます。（しかし1周期に`bgwriter_lru_maxpages`を越えるバッファ数を書き出しません。）したがって、1.0と設定することは、必要と予想されるバッファ数の書き込みについて"必要なときに必要なだけ"というポリシーを表します。より大きな値は突発的な要求に対する多少の緩衝材を提供します。より小さな値はサーバプロセスでなされる書き込みを意図的に残します。デフォルトは2.0です。このパラメータは`postgresql.conf`ファイル、または、サーバのコマンドラインでのみで設定可能です。

`bgwriter_lru_maxpages`および`bgwriter_lru_multiplier`の値がより少ないと、バックグラウンドライタで引き起こされる追加のI/O負荷を軽減しますが、サーバプロセスが自分自身で行わなければならない書き込みが増加することになり、会話型問い合わせを遅らせることになります。

18.4.6. 非同期動作

`effective_io_concurrency (integer)`

PostgreSQLが同時処理を予定している同時実行ディスクI/O作業の数を設定します。この値を大きくすると、あらゆる個別のPostgreSQLセッションが平行して開始を試みるI/O作業数が増加します。許容範囲は1から1000まで、または非同期I/Oリクエストの発行を無効にするゼロです。現在、この設定はビットマップヒープスキャンのみに有効です。

この設定の開始点として優れた値は、データベースに使用されるRAID 0ストライプ、RAID 1ミラーを構成する個々のドライブ数です。RAID 5ではパリティ用のドライブを数に含めてはなりません。）しかし、同時実行セッションで発行される複数の問い合わせでデータベースが頻繁にビジーとなる場合、ディスクアレイのビジー率を抑えるために、より小さな値で十分であるかもしれません。ディスクビジー率を抑えるために必要な値より大きな値を設定しても、余計なCPUオーバーヘッドを発生させるだけです。

メモリベースのストレージやバス帯域幅で制限されたRAIDアレイなどの、より斬新なシステムにおける正しい値は利用できるI/Oパスの数となるかもしれません。最善の値を見つけ出すには、何らかの実験が必要かもしれません。

非同期I/Oは実質的に`posix_fadvise`関数に依存します。これは一部のオペレーティングシステムには存在しません。この関数が存在しない場合、この値をゼロ以外に設定するとエラーとなります。一部のオペレーティングシステム（例えばSolaris）では存在するけれども、実際何も行わないものもあります。

18.5. ログ先行書き込み (WAL)

WALとチェックポイントの調整についての詳細は[項29.4](#)も参照してください。

18.5.1. 諸設定

wal_level (enum)

wal_level はどれだけの情報がWALに書かれるかを決定します。デフォルト値はminimalで、クラッシュまたは即時停止から回復するのに必要な情報のみ書き出します。archiveはWALアーカイビングに必要なロギングを追加し、hot_standbyは更にスタンバイサーバ上の読み取り専用問い合わせの情報を追加します。このパラメータはサーバ起動時のみ設定可能です。

minimal水準では、何らかの巨大な操作でのWALロギングは安全を期して省略されます。そうすることで、一連の操作をより高速にさせます ([項14.4.7](#)を参照してください)。この最適化が適用される操作には以下のものがあげられます。

```
CREATE TABLE AS
CREATE INDEX
CLUSTER
```

同一トランザクション内で作成されたか、もしくは切り詰められたテーブルに対するCOPY

しかしminimal WALはベースバックアップとWALログからデータを再構築するための十分な情報を持ち合わせていません。したがい、WALアーカイビング ([archive mode](#)) とストリーミングレプリケーションを有効にするにはarchiveまたはhot_standby水準を使用しなければなりません。

hot_standby水準においては、archiveと同じ情報がログ取得されるのに加え、WALから実行中のトランザクション状態を再構築するのに必要な情報が得られます。スタンバイサーバ上で読み取り専用問い合わせを有効にするには、hot_standbyがプライマリサーバで設定され、hot_standbyがスタンバイサーバで有効になっていなければなりません。hot_standby水準とarchive水準を使用した場合にちょっとした計測可能な性能上の差異がありますので、実際に運用して問題が見つかった場合はご意見を聞かせてください。

fsync (boolean)

このパラメータがオンの場合、PostgreSQLサーバはfsync() システム・コールを発行したり、もしくはこれに相当する方法で ([wal_sync_method](#)を参照) 更新が物理的にディスクに書き込まれたかの確認を得ようと試みます。これは、オペレーティングシステムもしくはハードウェアがクラッシュした後、確実にデータベースクラスタを一貫した状態に復旧させることを保証します。

fsyncを停止することはしばしば性能上の利益になるとは言っても、予期せぬシステム停止やクラッシュの際に回復不可能なデータ破壊になることがあります。従って外部データから全てのデータベースを簡単に再構築できる場合のみfsyncを停止してください。

fsyncを停止しても安全な状況の例としては、以下があげられます。データベースが削除され、そして再構築されたデータの一群の処理のため、または頻繁に再構築され、かつフェイルオーバーには使用されない読み取り専用のデータベースクローンに対し、バックアップファイルから新規データベースクラスタの初回読みを行う場合です。高性能なハードウェアであるからと言って、fsyncを停止することは正当性を主張する十分な理由とはなりません。

多くの場合、致命的ではないトランザクションにおいてsynchronous_commitを無効にすることにより、データ破壊の付随的危険性を伴うことなく、fsyncを無効にした場合に潜在する性能上のメリットの多くを得ることができます。

fsync はpostgresql.conf ファイル、または、サーバのコマンドラインでのみ設定可能です。このパラメータを無効にする場合、full_page_writesも同時に無効にすることを検討してください。

synchronous_commit (enum)

トランザクションのコミットがクライアントに"success"の表示を返す前に、WALレコードがディスク上に書き込まれるまで待つかどうかの指定をします。有効な値はon、remote_write、local、およびoffです。デフォルトかつ安全な設定はonです。offの場合、クライアントに成功を報告する時点とトランザクションが本当にサーバクラッシュに対して安全になるまでの間に遅延が発生します。(最大の遅延は、wal_writer_delayの3倍です。) fsyncと異なり、このパラメータをoffに設定することによって、データベースの一貫性が損なわれる可能性はありません。オペレーティングシステムやデータベースのクラッシュにより最近コミットされたということになっているトランザクションの一部が失われる可能性があります、これらのトランザクションが正常にアボートされた時とデータベースの状態は変わりません。ですので、synchronous_commitを無効にすることは、トランザクションの信頼性が確実であることよりも性能が重要である場

合に有効な方法です。詳細は[項29.3](#)を参照してください。

[synchronous_standby_names](#)が設定されていると、このパラメータもやはり、トランザクションのWALレコードが、スタンバイサーバに複製されるまでトランザクションコミットを待機するか否かを制御します。on に設定された場合、現在の同期スタンバイがトランザクションのコミットレコードを受け取り、記憶装置に既書き込まれたことを確認するまでコミットは待機は待機されます。このことにより、プライマリおよびスタンバイがそれぞれのデータベース記憶装置の故障を被った場合を除いて、トランザクションが失われることはありません。remote_write に設定された場合、現在の同期スタンバイがトランザクションのコミットレコードを受け取り、スタンバイのオペレーティングシステムに書き出されたことを確認するまでコミットは待機は待機されます。しかし、データがスタンバイの記憶装置に安定して書き込まれたか否かは必須ではありません。この設定は PostgreSQL のスタンバイインスタンスがクラッシュしたとしても、データ保護を保証するのに充分です。しかし、スタンバイがオペレーティングシステムのレベルでクラッシュした場合はこの限りではありません。

同期レプリケーションが使用されている場合、通常、ディスクに対してのローカルな書き込みとWALレコードのレプリケーションのいずれかを待つか、トランザクションに非同期でコミットさせるかどうかの選択を行うよう実用的になっています。しかし、トランザクションに対し特別の値であるlocalが使用でき、同期レプリケーションではなく、ディスクへのローカルフラッシュの待機を要請することが可能です。もし synchronous_standby_names が設定されていなければ、on、remote_write および local の設定は全て同一の同期レベルを提供します。トランザクションのコミットはローカルディスクへの書き込みのみ待機します。

このパラメータはいつでも変更可能です。この設定により任意の1つのトランザクションのコミット時の動作が決まります。したがって、一部のトランザクションのコミットを同期的に、その他を非同期的にすることが可能で、かつ、有用です。例えば、デフォルトが同期コミットの場合に単一の複数文トランザクションを非同期にコミットさせるためには、トランザクション内でSET LOCAL synchronous_commit TO OFFを発行します。

wal_sync_method (enum)

WALの更新をディスクへ強制するのに使用される方法です。fsyncがオフの場合この設定は役に立ちません。と言うのはWALファイルの更新が全く強制されないからです。取り得る値は以下のものです。

- open_datasync (open() オプションO_DSYNCでWALファイルの書き込み)
- fdatasync (コミット毎にfdatasync() を呼び出し)
- fsync (コミット毎にfsync() を呼び出し)
- fsync_writethrough (いかなるディスク書き込みキャッシュもライトスルーさせるため、コミット毎にfsync() を呼び出し)
- open_sync (open() オプションO_SYNCでWALファイルの書き込み)

もし可能ならopen_*オプションもO_DIRECTを使用します。全てのプラットフォームでこれら全ての選択肢が使えるわけではありません。デフォルトは、上のリストのプラットフォームでサポートされるものの最初に列挙されているものです。ただしLinuxではfdatasyncがデフォルトです。デフォルトは必ずしも理想的なものではありません。この設定、あるいはクラッシュに適応した構成か、アーカイブの最適性能を導くために使用しているシステム構成の形態を変更することが必要かもしれません。これらの側面は[項29.1](#)で解説されます。このパラメータはpostgresql.conf ファイル、または、サーバのコマンドラインでのみ設定可能です。

full_page_writes (boolean)

このパラメータが有効の場合、PostgreSQLサーバは、チェックポイントの後にそのページが最初に変更された過程で、ディスクページの全ての内容をWALに書き込みます。オペレーティングシステムがクラッシュした時に進行中のページ書き込みは途中までしか終わっていない可能性があるため、これが必要です。この場合、ディスク上のページは古いデータと新しいデータが混在する状態になります。通常WAL内に保存される行レベルの変更データは、クラッシュ後のリカバリ時にこうしたページを完全に復旧させるには不十分です。完全なページイメージが、ページを正確に復旧できることを保証します。しかし、WALに書き込まなければならないデータ量が増加する代償と引きかえになります。(WAL再生は常にチェックポイントから始まるため、チェックポイント後のそれぞれのページの最初の変更時にこれを行うことで差し支えありません。従って、完全ページ書き出しのコストを低減する方法の1つは、チェックポイント間隔パラメータを増やすことです。)

このパラメータを無効にすると、通常の操作速度が上がりますが、システム障害後に、回復不能なデータ破損、あるいは警告なしのデータ損壊をもたらすかもしれません。このリスクは小さいながらfsyncを無効にした場合と似ています。そしてそのfsyncに対して推奨されている同一の状況に基づく限りにおいて停止されなければなりません。

このパラメータを無効にしてもポイントインタイムリカバリ (PITR) 用のWALアーカイブの使用に影響ありません ([項24.3](#)を参照してください)。

このパラメータはpostgresql.conf ファイル内、または、サーバのコマンドラインでのみ設定可能です。デフォルト

はonです。

wal_buffers (integer)

未だディスクに書き込まれていないWALデータに対して使用される共有メモリ容量です。デフォルトの設定である-1は、[shared_buffers](#)の1/32（約3%）の容量に等しい大きさを選択します。しかし、64kB未満ではなく、かつ典型的に16MBであるWALセグメントの大きさを越えることはありません。もし、自動設定による選択が大きすぎたり、小さすぎる場合この値は手作業で設定可能です。しかし、32kB未満のどんな正の値であっても、32kBとして取り扱われます。このパラメータはサーバ起動時のみ設定可能です。

WALバッファの内容はトランザクションのコミット毎にディスクに書き込まれます。したがって、極端に大きな値は有意な効果を期待できません。しかし、この値を数メガバイトに設定することにより、多くのクライアントが同時にコミットするトラフィック量の多いサーバでは書き込み性能が向上します。デフォルト設定の-1で選択される自動チューニングによると、ほとんどの場合妥当な結果が得られます。

このパラメータを増加させると、使用しているオペレーティングシステムのデフォルト構成が許容するSystem V共有メモリの限界を越えた要求をPostgreSQLが行う原因となることがあります。必要であれば、どの様にしてこのパラメータを調整するかについて[項17.4.1](#)を参照ください。

wal_writer_delay (integer)

WALライタの活動周期の間隔を指定します。ライタのこの各周期でWALをディスクに吐き出します。そしてwal_writer_delayミリ秒待機し、それを繰り返します。デフォルト値は200ミリ秒（200ms）です。多くのシステムでは、待機間隔の実質的な分解能は10ミリ秒です。10の倍数以外の値をwal_writer_delayに設定しても、その次に大きい10の倍数を設定した場合と同じ結果となります。このパラメータはpostgresql.confファイル内またはサーバのコマンドラインでのみ設定可能です。

commit_delay (integer)

トランザクションに対するコミットデータがディスクにフラッシュされた時、その時点において用意万端のいかなる追加的コミットも同じくフラッシュされます。マイクロ秒単位で設定されるcommit_delayは、トランザクションがWALバッファをディスクにフラッシュを試みる前までの時間遅延を追加します。システム負荷が十分に高く、指定期間内にコミット準備ができる他のトランザクションが存在しない場合、非ゼロの遅延時間で複数のトランザクションを一度のフラッシュ操作で行えるようになります。しかし、もし他のトランザクションがコミットできる状態にならなければ、この遅延は無駄になります。と言うことは、サーバプロセスがそのコミットレコードを書き込んだ瞬間に他のトランザクションが進行している commit_siblingsにだけ少なくとも実行されます。デフォルトの commit_delay はゼロ（遅延無し）です。この設定にかかわらず、全ての保留中のコミットデータはそれぞれのフラッシュの際に書き込まれるので、このパラメータを大きくして遅延を追加しても実際に性能が向上することは稀にしか起こりません。

commit_siblings (integer)

commit_delay遅延を実行する前に必要とされる同時に開いているトランザクションの最小数です。より大きい値は、遅延周期の間に、少なくとも1つの他のトランザクションがコミットの準備を整わせることを確実にします。デフォルトは5トランザクションです。

18.5.2. チェックポイント

checkpoint_segments (integer)

自動WALチェックポイント間の最大ログファイル数です。（それぞれのセグメントは通常16メガバイト）デフォルトは3セグメントです。このパラメータを増やすと、クラッシュリカバリで必要となる時間が増加します。このパラメータはpostgresql.confファイル、または、サーバのコマンドラインでのみ設定可能です。

checkpoint_timeout (integer)

自動的WALチェックポイント間の最大間隔を秒単位で指定します。デフォルトは5分（5min）です。このパラメータを増やすと、クラッシュリカバリで必要となる時間が増加します。このパラメータはpostgresql.confファイル、または、サーバのコマンドラインでのみ設定可能です。

checkpoint_completion_target (floating point)

チェックポイントの完了目標をチェックポイント間の総時間の割合として指定します。デフォルトは0.5です。このパラメータはpostgresql.confファイル、または、サーバのコマンドラインでのみ設定可能です。

checkpoint_warning (integer)

チェックポイントセグメントファイルが溢れることが原因で起きるチェックポイントが、ここで指定した秒数よりも頻繁

に発生する場合、サーバログにメッセージを書き出します（これは、`checkpoint_segments`を増やす必要があることを示唆しています）。デフォルトは30秒（30s）です。零の場合は警告を出しません。このパラメータは`postgresql.conf`ファイル、または、サーバのコマンドラインでのみ設定可能です。

18.5.3. アーカイピング

`archive_mode` (boolean)

`archive_mode`が有効な時、[archive_command](#)を設定して、完了したWALセグメントをアーカイブ格納領域に送信されます。アーカイブモードを抜けることなく`archive_command`を変更できるように、`archive_mode`と`archive_command`は分離されました。このパラメータはサーバ起動時のみ設定可能です。`wal_level`が`minimal`に設定されている場合、`archive_mode`は有効になりません。

`archive_command` (string)

完了したWALファイルセグメントのアーカイブを実行するシェルコマンドです。文字列内のいかなる`%p`は、格納されるファイルの絶対パスで置き換えられ、そして、`%f`はファイル名のみ置換します。（このパス名はサーバの作業用ディレクトリ、つまり、クラスタのデータディレクトリからの相対パスです。）コマンド内で実際の`%`文字を埋め込むには`%%`を使用します。コマンドが成功した場合に限って終了ステータスゼロを返すことが重要です。より詳しくは[項24.3.1](#)を参照ください。

このパラメータは`postgresql.conf`ファイル、または、サーバのコマンドラインでのみ設定可能です。サーバ起動時に`archive_mode`が有効でなければ、これは無視されます。`archive_command`が空文字列（デフォルト）、かつ、`archive_mode`が有効な場合、WALアーカイブ処理は一時的に無効になりますが、コマンドが後で提供されることを見越して、サーバはWALセグメントの蓄積を続けます。例えば、`/bin/true`（Windowsでは`REM`）のように、コマンドに対し`archive_command`を真を返すだけで何もしないように設定すると効果的にアーカイブ処理を無効にしますが、アーカイブからの復帰に必要なWALファイルの連鎖を同時に断ち切ります。従って、特別な場合のみ使用するようにしなければなりません。

`archive_timeout` (integer)

[archive_command](#)は完了したWALセグメントに対してのみ呼び出されます。従って、サーバのWAL転送量が少ししかない（処理を行わないながの期間がある）場合、トランザクションの完了とアーカイブ格納領域への安全な記録との間に長期にわたる遅延があることとなります。古い未アーカイブのデータをどうするかについて制限を付けるために、`archive_timeout`を設定して、強制的にサーバを新しいWALセグメントに定期的に切り替えるようにすることができます。このパラメータが0より大きければ、サーバは前回のセグメントファイル切り替えから指定秒数経過した場合、および単一のチェックポイントを含む何らかのデータベース操作が行われた場合、新しいセグメントファイルに切り替えます。（`checkpoint_timeout`を大きくすると待機状態のシステム上のなくともいいチェックポイントを削減します。）強制切り替えにより早期に閉ざされたアーカイブ済みファイルは完全に完了したファイルと同じ大きさを持つことに注意してください。そのため、非常に小さな`archive_timeout`を使用することは思慮を欠いています。格納領域を膨張させてしまいます。通常1分程度の`archive_timeout`設定が妥当です。もしそれより高速にデータをマスターサーバにコピーをしたいのであれば、アーカイブするよりストリーミングレプリケーションの選択を検討すべきです。このパラメータは`postgresql.conf`ファイル、または、サーバのコマンドラインでのみ設定可能です。

[前のページ](#)
資源の消費

[ホーム](#)
[上に戻る](#)

[次のページ](#)
レプリケーション

18.6. レプリケーション

これらの設定は組み込みのストリーミングレプリケーション機能の動作を制御します（[項25.2.5](#)を参照ください）。サーバ群のサーバはマスターかスタンバイのいずれかです。マスターはデータを送出する一方、複数のスタンバイは複製されたデータを常に受け取ります。カスケードレプリケーション（[項25.2.6](#)を参照）が使用されている場合、スタンバイサーバ群は受け取り手でもあり、送先でもあります。パラメータは主として送先とスタンバイサーバ用ですが、いくつかのパラメータはマスターサーバのみに効力を発します。必要とあればクラスターに渡って問題なく設定を変化させることができます。

18.6.1. 送先サーバ群

これらのパラメータはレプリケーションデータを1つ、またはそれ以上複数のスタンバイサーバに送るいかなるサーバ上で設定することができます。マスターは常に送先サーバであるため、パラメータは常にマスター上に設定されなければなりません。これらのパラメータの役割と意味はスタンバイが後にマスターに昇格しても変わりません。

`max_wal_senders (integer)`

複数のスタンバイサーバ、またはストリーミングを基盤とする予備（バックアップ）クライアントからの同時接続を受け取る接続最大値を設定します（つまり、同時に稼働するWAL送信プロセスの最大値です）。デフォルトはゼロです。その意味するところはレプリケーションは無効です。WAL送先プロセスは最大接続数を数えているため、[max_connections](#)を越えたパラメータの設定できません。このパラメータはサーバ起動時のみ設定可能です。スタンバイサーバからの接続を許可するには、`wal_level`が`archive`、または`hot_standby`に設定されなければなりません。

`wal_keep_segments (integer)`

ストリーミングレプリケーションにおいて、スタンバイサーバが過去のファイルセグメントを取得する必要がある場合に備え、`pg_xlog`ディレクトリに保持しておくファイルセグメント数の最小値を指定します。それぞれのセグメントは通常16メガバイトです。もし送先サーバに接続しているスタンバイサーバが`wal_keep_segments`セグメントを越えて遅延した場合、送先サーバはスタンバイサーバが今後とも必要とするWALセグメントを削除する可能性があります。この場合、レプリケーション接続は終了させられます。結果として下流に対する接続が同時に結果として終了されることがあります。（しかし、WALアーカイブが使用されていれば、スタンバイサーバはアーカイブからセグメントを取り出し、復旧することができます。）

`pg_xlog`に保持され続けるセグメントの最小値のみを設定します。システムはWALアーカイブのため、またはチェックポイントからの復旧のため、より多くのセグメント保持が必要となることがあります。もし`wal_keep_segments`が（デフォルトの）ゼロの場合、システムは待機目的でのいかなる追加セグメントも保持しません。従って、スタンバイサーバが使用できる古いWALセグメントの多くは、前回のチェックポイントの場所とWALアーカイブの状態を捕捉する機能になります。このパラメータは、`postgresql.conf`ファイル、もしくはサーバコマンドラインでのみ設定可能です。

`replication_timeout (integer)`

指定されたミリ秒単位の値より長く非活動のレプリケーション接続を停止します。スタンバイサーバのクラッシュ、またはネットワークの停止を送先サーバが検出することにこれが役立ちます。値ゼロはタイムアウト機能を無効にします。このパラメータは、`postgresql.conf`ファイル、もしくはサーバコマンドラインでのみ設定可能です。デフォルトの値は60秒です。

時期尚早に接続が停止されることを防ぐため、スタンバイサーバで[wal_receiver_status_interval](#)が必ず有効になっていなければなりません。また、その値は`replication_timeout`の値未満でなければなりません。

18.6.2. マスターサーバ

これらのパラメータはレプリケーションデータを1つ、またはそれ以上複数のスタンバイサーバに送るマスター/プライマリサーバ上で設定することができます。これらパラメータに加え、[wal_level](#)はマスターサーバ上で適切に設定される必要があり、任意的にWALアーカイブがサーバとしても有効になります（[項18.5.3](#)を参照してください）。スタンバイサーバがマスターサーバになるかもしれない状況に備え、それらのパラメータをスタンバイサーバで設定したいと考えたとしても、スタンバイサーバ上でのパラメータの値は意味をなしません。

`synchronous_standby_names (string)`

[項25.2.7](#)で説明されているように、同期レプリケーションをサポート可能なコンマで区切られたリストでスタンバイサーバの名前を指定します。いつの時点においても、最低限一つの活動している同期スタンバイサーバが存在します。コミットを待機しているトランザクションは、このスタンバイサーバがそのデータの受信を確認してから処理の継続が許可されます。同期スタンバイサーバはこのリストで一番目に名前が挙げられており、現時点で接続され、そしてデータ

をリアルタイムでストリーミングしているものです ([pg_stat_replication](#) ビューにおいてストリーミング状態として示されています)。このリストの後の方に記載されているその他のスタンバイサーバは潜在的に同期スタンバイサーバになることを示しています。もし現在の同期スタンバイサーバが理由にかかわらず切断された場合、次に順位の高いスタンバイサーバがすぐに取って代わります。二つ以上のスタンバイサーバ名を指定することでかなりの高可容性が得られます。

この目的のスタンバイサーバの名前は、スタンバイサーバのwalreceiverのprimary_conninfoで設定されるのと同じく、スタンバイサーバのapplication_name設定となります。一意性を強要する仕組みにはなっていません。walreceiverのデフォルトのアプリケーション名を含めて、特別の記載である*は全てのapplication_nameにマッチします。

ここに同期スタンバイ名が指定されていない場合、同期レプリケーションは有効とはならず、トランザクションコミットはレプリケーションを待機しません。これがデフォルトの設定です。同期レプリケーションが有効であっても、[synchronous_commit](#)パラメータをlocal または off に設定することにより、個別のトランザクションをレプリケーションに対して待機しないように設定できます。

このパラメータは、postgresql.conf ファイル、もしくはサーバコマンドラインでのみ設定可能です。

vacuum_defer_cleanup_age (integer)

VACUUM および HOTの更新が不要行バージョンの清掃と差があるかを定めるトランザクションの数を指定します。デフォルトはゼロトランザクションです。この意味は、不要行バージョンは速やかに削除され、即いかなる開いているトランザクションから不可視となります。項25.5に記載されているように、ホットスタンバイサーバをサポートしている場合、非ゼロ値に設定したい場合があります。早期に掃除された行のため、衝突を回避するためスタンバイ上での問い合わせが完了するのにより時間を割くことができます。しかし、値はプライマリサーバ上で発生している書き込みトランザクションの観点から計測されるため、スタンバイの問い合わせにたいして猶予時間がどのくらい有効となるかは予測できません。このパラメータは、postgresql.conf ファイル、もしくはサーバコマンドラインでのみ設定可能です。

このパラメータの使用の代わりにスタンバイサーバ上に hot_standby_feedback の設定を考慮する必要もあります。

18.6.3. スタンバイサーバ

これらの設定はレプリケーションデータを受け取るスタンバイサーバの動作を管理します。マスターサーバ上のこれらの値は無意味です。

hot_standby (boolean)

項25.5に記載されている通り、リカバリの最中に接続し、そして問い合わせを実行できるか否かを設定します。デフォルト値はoffです。このパラメータはサーバ起動時のみ設定可能です。これは、アーカイブリカバリ期間、又は待機モードにある場合にのみ効果をもたらします。

max_standby_archive_delay (integer)

ホットスタンバイが稼動している場合、このパラメータは項25.5.2に記載されているように、まさに適用されようとしているWALエントリと衝突するスタンバイサーバの問い合わせをキャンセルするにはどれだけ待機しなければならないかを設定します。max_standby_streaming_delayはWALデータをWALアーカイブから読み込んでいる時に適用されます(従って最新ではありません)。デフォルトは30秒です。特に指定が無ければ単位はミリ秒です。値-1は衝突する問い合わせが完了するまでスタンバイサーバが待ち続けられるようにします。このパラメータは、postgresql.conf ファイル、もしくはサーバコマンドラインでのみ設定可能です。

max_standby_archive_delayはキャンセル前に問い合わせが実行できる最大の時間の長さと同じでないことに注意してください。むしろ、任意の1つのWALセグメントのデータに適用される最大許可時間です。従って、ある問い合わせがWALセグメント内で時間的に初期の段階で大幅な遅延となった場合、その後の衝突する問い合わせの猶予時間はましてさらに短くなります。

max_standby_streaming_delay (integer)

ホットスタンバイが稼動している場合、このパラメータは項25.5.2に記載されているように、まさに適用されようとしているWALエントリと衝突するスタンバイサーバの問い合わせをキャンセルするにはどれだけ待機しなければならないかを設定します。max_standby_streaming_delayはWALデータをストリーミングレプリケーションから受け取っている時に適用されます。デフォルトは30秒です。特に指定が無ければ単位はミリ秒です。値-1は衝突する問い合わせが完了するまでスタンバイサーバが待ち続けられるようにします。このパラメータは、postgresql.conf ファイル、もしくはサーバコマンドラインでのみ設定可能です。

max_standby_streaming_delayはキャンセル前に問い合わせが実行できる最大の時間の長さと同じでないことに注意してください。むしろ、プライマリサーバから一度受け取られたWALデータを適用するための最大許可時間です。従って、ある問い合わせがWALセグメントにおいてそれまでに大幅な遅延となった場合、それに続いて衝突する問い合わせは、スタンバイサーバがふたたび遅れを取り戻すまで、猶予時間はさらに短くなります。

wal_receiver_status_interval (integer)

プライマリー、または上位サーバに対してレプリケーションの進捗情報を送信するため、スタンバイ上のWAL受信プロセスの最小頻度を指定します。ここで、[pg_stat_replication](#)ビューにより確認することが可能です。スタンバイサーバは既書き込まれた最終のログ位置を報告し、その最終位置がディスクにフラッシュされ、その最終位置が適用されます。このパラメータの値がそれぞれの報告間における秒単位の最大の時間間隔です。書き込み、またはフラッシュ位置が変更される毎に更新が行われます。あるいは、少なくともこのパラメータで設定された頻度で行われます。このパラメータをゼロに設定すると、ステータスの更新を完全に無効化します。このパラメータは、`postgresql.conf`ファイル、もしくはサーバコマンドラインでのみ設定可能です。デフォルトの値は10秒です。

送出サーバ上で[replication_timeout](#)が有効になっている場合、`wal_receiver_status_interval`も有効になっていなければなりません。そして、その値は`replication_timeout`の値未満である必要があります。

hot_standby_feedback (boolean)

ホットスタンバイがスタンバイサーバ上で現在処理を行っている問い合わせについて、プライマリーまたは上位サーバにフィードバックを送るか否かを指定します。このパラメータはレコードの後片付けに起因する問い合わせの取り消しを排除するために使用することができます。しかし、いくつかの作業負荷に対してはプライマリーサーバ上でのデータベース肥大の原因となります。フィードバックメッセージは`wal_receiver_status_interval`毎に、一回以上送信されません。デフォルトの値は`off`です。このパラメータは、`postgresql.conf`ファイル、もしくはサーバコマンドラインでのみ設定可能です。

カスケードレプリケーションが使用されている場合、フィードバックは最終的にプライマリーに到達するまで上位サーバに転送されます。スタンバイは上位に転送以外、受け取ったフィードバックを他に使用しません。

[前のページ](#)

ログ先行書き込み (WAL)

[ホーム](#)
[上に戻る](#)

[次のページ](#)
問い合わせ計画

18.7. 問い合わせ計画

18.7.1. プランナメソッド設定

これらの設定パラメータは、問い合わせオプティマイザが選択する問い合わせ計画に影響する大雑把な手法を提供します。もしも、ある問い合わせに対してオプティマイザが選択したデフォルト計画が最適でない場合、暫定的な解決策は、これらの設定パラメータの1つを使用し、オプティマイザに異なる計画を選択するように仕向けることです。品質を改良する方策は以下を含みます。プランナコスト定数の調節 ([項18.7.2](#)を参照)、[default_statistics_target](#)設定パラメータの値を手作業で大きくし、そしてALTER TABLE SET STATISTICSを使用して、特定の列に対して収集された統計情報を増やします。

`enable_bitmapscan (boolean)`

問い合わせプランナがビットマップスキャン計画型を選択することを有効もしくは無効にします。デフォルトはonです。

`enable_hashagg (boolean)`

問い合わせプランナがハッシュ集約計画型を選択することを有効もしくは無効にします。デフォルトはonです。

`enable_hashjoin (boolean)`

問い合わせプランナがハッシュ結合計画型を選択することを有効もしくは無効にします。デフォルトはonです。

`enable_indexscan (boolean)`

問い合わせプランナがインデックス走査計画型を選択することを有効もしくは無効にします。デフォルトはonです。

`enable_indexonlyscan (boolean)`

問い合わせプランナがインデックス走査計画型のみを選択することを有効もしくは無効にします。デフォルトはonです。

`enable_material (boolean)`

問い合わせプランナの実体化の使用を有効、または無効にします。全体にわたって具体化を差し止めることはできませんが、この値をoffにすることにより、正確性が要求される場合を除いて、具体化ノードをプランナが挿入することを防止します。デフォルトはonです。

`enable_mergejoin (boolean)`

問い合わせプランナがマージ結合計画型を選択することを有効もしくは無効にします。デフォルトはonです。

`enable_nestloop (boolean)`

問い合わせプランナが入れ子になったループ結合計画を選択することを有効もしくは無効にします。入れ子ループ結合を完全に禁止することは不可能ですが、この変数をオフにすると、もし他の方法が利用できるのであれば、プランナはその使用を行わないようになります。デフォルトはonです。

`enable_seqscan (boolean)`

問い合わせプランナがシーケンシャル走査計画を選択することを有効もしくは無効にします。シーケンシャル走査を完全に禁止することは不可能ですが、この変数をオフにすると、もし他の方法が利用できるのであれば、プランナはその使用を行わないようになります。デフォルトはonです。

`enable_sort (boolean)`

問い合わせプランナが明示的並び替え手順を選択することを有効もしくは無効にします。明示的並び替えを完全に禁止することは不可能ですが、この変数をオフにすると、もし他の方法が利用できるのであれば、プランナはその使用を行わないようになります。デフォルトはonです。

`enable_tidscan (boolean)`

問い合わせプランナがTID走査計画型を選択することを有効もしくは無効にします。デフォルトはonです。

18.7.2. プランナコスト定数

本節で扱うコスト変数は、任意の尺度で測られます。これらは相対的な値でしかありません。そのため、同じ因子で尺度を変えても、プランナの選択は結果として変わりません。デフォルトではこれらのコスト変数はシーケンシャルなページ取り込みに基づいています。つまり、`seq_page_cost`を慣習的に1.0とし、他のコスト変数をそれを参考にして設定されていました。しかし望むなら、特定のマシンにおけるミリ秒単位の実行時間など、異なる尺度を使用することができます。

注意: 残念ながら、コスト変数に対する理想的な値を決定する、上手く定義された方法がありません。特定のインストレーションが受け取る問い合わせ全体を混在させたものの平均を最善のものとして扱われています。数回の実験のみを根拠にこの値を変更することは危険であるといえます。

`seq_page_cost (floating point)`

シーケンシャルな一連の取り出しの一部となる、ディスクページ取り出しに関する、プランナの推定コストを設定します。デフォルトは1.0です。この値は同じ名前の特定期間パラメータを設定することで、その中にあるテーブルとインデックスに対し上書きされます ([ALTER TABLESPACE](#)を参照してください)。

`random_page_cost (floating point)`

非シーケンシャル的に取り出されるディスクページのコストに対するプランナの推測を設定します。デフォルトは4です。この値は同じ名前の特定期間パラメータを設定することで、その中にあるテーブルとインデックスを上書きします ([ALTER TABLESPACE](#)を参照してください)。

この値を`seq_page_cost`に相対的に減少させると、システムはインデックススキャンを好んで使用するようになります。増加させると、インデックススキャンが相対的に高価になります。両方の値を増減させることで、CPUコストに対するディスクI/Oの重要性を変更させることができます。これについては、後述のパラメータで説明します。

機械的ディスク記憶装置に対するランダムアクセスは通常4回の逐次アクセスよりかなり高価です。しかし、より低いデフォルト (4.0) が使用されます。というのはインデックスのついた読み取りのようなディスクに対するランダムアクセスのほとんどはキャッシュにあると想定されるからです。デフォルト値は逐次よりも40倍遅いとモデル化したランダムアクセスで考えられる一方、90%のランダム読み込みがキャッシュと期待されるからです。

作業負荷に対し、90%のキャッシュ率は誤った仮定と信ずるのであれば、ランダム記憶装置読み込みのコストをより良く反映するため `random_page_cost` を増加させることができます。対応的に、データが完全にキャッシュされていると思われるのであれば、`random_page_cost` を引き下げることが適切です。例えば、データベースがサーバメモリより小さい場合などです。例えば半導体ドライブのような、逐次アクセスに相対して記憶装置が低いランダム読み込みコストの場合、`random_page_cost` に対しより低い値のモデル化もあるかもしれません。

ティップ: システムは`random_page_cost`を`seq_page_cost`よりも小さな値に設定しようとはしますが、これには物理的な意味はありません。しかし、データベースが完全にRAMにキャッシュされる場合、同じ値に設定することは意味を持ちます。この場合、順序通りではないページアクセスに対するペナルティが存在しないからです。また、多くがキャッシュされるデータベースでは、CPUパラメータに対して両値を小さく設定すべきです。RAM内に存在するページの取り出しコストは通常よりもかなり小さくなるためです。

`cpu_tuple_cost (floating point)`

問い合わせ間にそれぞれの行の処理に対するプランナの推測を設定します。デフォルトは0.01です。

`cpu_index_tuple_cost (floating point)`

インデックス走査間にそれぞれのインデックス行の処理に対するプランナの推測を設定します。デフォルトは0.005です。

`cpu_operator_cost (floating point)`

問い合わせ時に実行される各演算子や関数の処理コストに対するプランナの推測を設定します。デフォルトは0.0025です。

`effective_cache_size (integer)`

単一の問い合わせで利用できるディスクキャッシュの実効容量に関するプランナの条件を設定します。これは、インデックスを使用するコスト推定値の要素となります。より高い値にすれば、よりインデックススキャンが使用されるようになり、より小さく設定すれば、シーケンシャルスキャンがより使用されるようになります。このパラメータを設定する時には、PostgreSQLの共有バッファとPostgreSQLデータファイルに使用されるカーネルのディスクキャッシュの量の両方を考慮しなければなりません。また、利用可能な領域を共有しますので、異なるテーブルに対して同時に実行される問い合わせの総定数も考慮してください。このパラメータは、PostgreSQLで割り当てられる共有メモリの大きさには影響を与えません。また、カーネルのディスクキャッシュを予約したりもしません。これは推定目的のみで使用されます。同時に、システムは問い合わせの間のディスクキャッシュ内のデータの残滓を想定していません。デフォルトは128メガバイト (128MB) です。

18.7.3. 遺伝的問い合わせオプティマイザ

遺伝的問い合わせオプティマイザ (GEQO) はヒューリスティック (発見的) 検索法を用いた問い合わせ計画を行なう演算法手法です。通常のしらみつぶしの検索演算法で見いだされる計画よりも時として劣った計画を作成するという代償を払いますが、この手法は (多くのリレーションを結合するような) 複雑な問い合わせに対し計画時間を軽減します。より詳細は[第51章](#)を参照してください。

`geqo (boolean)`

遺伝的問い合わせ最適化を有効もしくはは無効にします。デフォルトは有効です。運用時には無効にしないことが通常最善です。 `geqo_threshold` 変数は、GEQOを制御するためよりきめ細かな方法を提供します。

`geqo_threshold (integer)`

少なくともこれだけの数の `FROM` 項目数で問い合わせを計画するのに遺伝的問い合わせ最適化を使用します。 (`FULL OUTER JOIN` の生成子は、1つの `FROM` 項目として計算することに注意してください。) デフォルトは12です。もっと単純な問い合わせでは、通常の、そしてしらみつぶしの検索プランナを使用するのが最善ですが、多くのテーブルを持つ問い合わせでは、しらみつぶしの検索は非常に時間がかかり、しばしば事前の計画を実行する代償より長くなります。従って、問い合わせの大きさに対する限界値はGEQOの使用を管理するのに便利な方法です。

`geqo_effort (integer)`

GEQOにおける計画時間と問い合わせ計画の品質間のトレードオフを制御します。この変数は1から10までの範囲の整数でなければなりません。デフォルトの値は5です。値を大きくすると、問い合わせ計画作成により多くの時間を費すことになりませんが、より効率的な問い合わせ計画が選択される可能性が増加します。

実際 `geqo_effort` は直接何も行いません。それはGEQOの動作に影響を与える他の変数に対し、デフォルトの値を計算するためにのみ使用されます (以下で説明します)。もしよければ、代わりに手作業で他のパラメータを設定できます。

`geqo_pool_size (integer)`

GEQOで使用されるプール容量を管理します。それは遺伝的個体群内の個体数です。最低でも2つはなければならず、よく100から1000までの値が使用されます。もし (デフォルトの設定である) 零に設定されると、 `geqo_effort` および問い合わせの中のテーブル数に基づいて、適切な値が選択されます。

`geqo_generations (integer)`

GEQOで使用される世代の数を管理します。それはアルゴリズムの反復数です。最低でも1はなければならず、よくプールサイズと同じ範囲の値が使用されます。これを0に設定 (デフォルトの設定) すると、適切な値が `geqo_effort` に基づいて選択されます。

`geqo_selection_bias (floating point)`

GEQOで使用される淘汰の偏りを管理します。淘汰の偏りは個体群内の (遺伝的な) 自然淘汰です。値は1.50から2.00で、2.00がデフォルトです。

`geqo_seed (floating point)`

結合順序検索空間にわたって、GEQOが無作為のパスを選択するために使用される乱数発生器の初期値を制御済ます。値は0 (デフォルト) から1までの範囲です。値を変動させると探査される結合パスの集合を変化させ、それが見つかっているより良いか、より悪い最善のパスとなる可能性があります。

18.7.4. その他のプランナオプション

`default_statistics_target (integer)`

`ALTER TABLE SET STATISTICS` で列特定の目的セットの無いテーブル列に対し、デフォルトの統計対象を設定します。より大きい値は `ANALYZE` に必要な時間を増加させますが、プランナの予測の品質を向上させます。デフォルトは100です。PostgreSQLの問い合わせプランナによる統計情報の使用法に関するより詳細な情報は、[項14.2](#)を参照してください。

`constraint_exclusion (enum)`

問い合わせを最適化するため、テーブル制約に対しての問い合わせプランナの使用を制御します。 `constraint_exclusion` に許容される値は、 `on` (全てのテーブルに対し制約を検査する)、 `off` (決して制約を検査しない)、および `partition` (継承された子テーブルおよび `UNION ALL` 副問い合わせのみ制約を検査する) です。 `partition` がデフォルトです。継承テーブルと分割されたテーブルの性能向上のためしばしば使用されます。

このパラメータが特定のテーブルに対して許される時、プランナはそのテーブルのCHECK制約で問い合わせ条件を比較し、制約と矛盾する条件のテーブルの走査を省きます。例えば以下ようになります。

```
CREATE TABLE parent(key integer, ...);
CREATE TABLE child1000(check (key between 1000 and 1999)) INHERITS(parent);
CREATE TABLE child2000(check (key between 2000 and 2999)) INHERITS(parent);
...
SELECT * FROM parent WHERE key = 2400;
```

制約排除が有効であると、このSELECTは全くchild1000を走査せず、性能を向上させます。

現在constraint_exclusionはデフォルトで、テーブルパーティショニングを実装するためによく使用される場合のみで有効です。すべてのテーブルを有効にすることは、計画作成において単純な問い合わせでは無視できない程の余計なオーバーヘッドをもたらします。パーティショニングされたテーブルがない場合、完全に無効にする方が良いでしょう。

制約排除とテーブル分割についてのより進んだ情報は[項5.9.4](#)を参照ください。

cursor_tuple_fraction (floating point)

検索されるカーソル行の割合のプランナの見積もりを設定します。デフォルトは0.1です。この設定をより小さくすると、プランナはカーソルに対し"起動を高速にする"計画を使用するようになりがちになります。この場合先頭の数行の取り出しは高速になりますが、行全体を取り出す場合に時間がかかるようになる可能性があります。この値をより大きくすると、推定時間全体がより強調されるようになります。最大の設定である1.0の場合、カーソルは通常の問い合わせとまったく同様に計画されます。つまり、推定時間全体のみが考慮され、先頭の行の取り出しにかかる時間は考慮されなくなります。

from_collapse_limit (integer)

プランナは、FROMリストがこの数の項目より少ない結果の場合、副問い合わせを上位の問い合わせに併合します。より小さい値は計画時間を縮小させますが、劣った問い合わせ計画をもたらす可能性があります。デフォルトは8です。詳細は[項14.3](#)を参照してください。

この値をgeqo_thresholdか、それ以上に設定するとGEQOプランナ使用の誘引となり、最適ではない計画をもたらします。 [項18.7.3](#)を参照してください。

join_collapse_limit (integer)

最終的にリストがこの項目数以下になる時、プランナは、明示的なJOIN構文（FULL JOINを除く）をFROM項目のリストに直します。この値を小さくすれば計画作成時間は減少しますが、劣った問い合わせ計画が作成される可能性があります。

デフォルトでは、この値はfrom_collapse_limitと同じ値に設定されており、殆どの場合に適切です。これを1に設定すると明示的なJOINの再順序付けは行われなくなります。したがって、問い合わせで指定された明示的結合順序は、関係（リレーション）が結合される実際の順序となります。問い合わせプランナは常に最適な結合順序を選択するとは限らないので、上級ユーザなら暫定的にこの変数を1に設定し、明示的に希望とする結合順序を指定してもよいでしょう。詳細は[項14.3](#)を参照してください。

この値をgeqo_thresholdか、それ以上に設定するとGEQOプランナ使用の誘引となり、非決定的計画をもたらします。 [項18.7.3](#)を参照してください。

[前のページ](#)
レプリケーション

[ホーム](#)
[上に戻る](#)

[次のページ](#)
エラー報告とログ取得

18.8. エラー報告とログ取得

18.8.1. ログの出力先

`log_destination (string)`

PostgreSQLは、stderr、csvlogおよびsyslogを含めて、サーバメッセージのログ取得に対し数種類の方法を提供します。Windowsでは、eventlogも同時に提供します。このパラメータを設定するには、コンマ区切りでお好みのログ出力先を記載します。デフォルトでは、ログはstderrのみに出力されます。このパラメータは`postgresql.conf`ファイル、または、サーバのコマンドラインでのみ設定されます。

csvlogが`log_destination`に含まれる場合、ログ項目はプログラムへの読み込みが簡便な"カンマ区切り値"書式 (CSV) で出力されます。詳細は[項18.8.4](#)を参照してください。CSV書式のログ出力を生成するためには[logging_collector](#)を有効にする必要があります。

注意: `log_destination`でsyslogオプションを使用できるようにするために、ほとんどのUnixシステムではシステムのsyslogデーモンの設定を変更しなければならないでしょう。PostgreSQLではログを`LOCAL0`から`LOCAL7`までのsyslogファシリティで記録することができます ([syslog_facility](#)を参照してください)。しかし、ほとんどのプラットフォームのデフォルトのsyslog設定ではこれらのメッセージはすべて破棄されます。うまく動作させるためにsyslogデーモンの設定に以下のようなものを追加しなければならないでしょう。

```
local0.*      /var/log/postgresql
```

Windowsで`log_destination`に対しeventlogオプションを使用する場合、Windows Event Viewer がイベントログメッセージを手際良く表示できるよう、オペレーティングシステムでイベントソースとそのライブラリを登録しなければなりません。詳細は[項17.11](#)を参照ください。

`logging_collector (boolean)`

このパラメータは[logging_collector](#)を有効にします。その形成要素はstderrに送られたログメッセージを補足し、ログファイルにリダイレクトするバックグラウンドプロセスです。この手法はsyslogへのログよりもしばしば有用です。メッセージの一部の種類がsyslogでは出力されない可能性があるためです。(一般的な例として、ダイナミックリンクのエラーメッセージがあり、その他の例として[archive_command](#)のようなスクリプトにより生成されたエラーメッセージが挙げられます)。このパラメータはサーバ起動時のみ設定可能です。

注意: ログ収集機構を使用せずにstderrのログを取ることは可能です。ログメッセージはサーバのstderrが指し示すいかなる場所にも向かうだけです。しかし、その方法はログファイルを巡回させる都合のよい方法を提供しないので、ログ容量が小さい場合のみに適しています。同時に、ログ収集機構を使用しないいくつかのプラットフォームにおいては、ログ出力が失われたり、文字化けします。なぜなら、同一のログファイルに同時に書き込みを行うマルチプロセッサはそれぞれの出力を上書きできるからです。

注意: ログ取得機構はメッセージを決して失わないために設計されています。これは、極端に高い負荷の場合、サーバプロセスはコレクタが遅れをとった場合、追加のログメッセージを送信しようと試みる時に阻止される可能性があります。それとは対照的にsyslogは、もし書き込みができなかったときメッセージの廃棄方を選びます。これらの場合にはいくつかのログメッセージを失うことになりますが、残ったシステムを阻止しません。

`log_directory (string)`

`logging_collector`を有効と設定した場合、このパラメータはログファイルが作成されるディレクトリを確定します。ディレクトリは、絶対パス、もしくはデータベースクラスタのディレクトリに対する相対パスで指定することができます。このパラメータは`postgresql.conf`ファイル、またはサーバコマンドラインからのみ設定可能です。

`log_filename (string)`

`logging_collector`が有効な場合、このパラメータは作成されたログファイルのファイル名を設定します。値はstrftimeパターンとして扱われるため、`%`エスケープを使用して、時刻によって変動するファイル名を指定することができます。(時間帯に依存した`%`エスケープが存在する場合、[log_timezone](#)で指定された時間帯で計算が行われます。) サポートされている`%`-エスケープは[strftime](#)仕様によく似ています。システムのstrftimeは直接使用されないため、プラットフォーム固有の(非標準)の拡張は動作しません。

エスケープすることなくファイル名を指定する場合、ディスク全体を使い切ってしまうことを防止するためにログロー

テーションを行うユーティリティを使用することを計画しなければなりません。 8.4より前のリリースのPostgreSQLでは、%エスケープがなければ、新しいログファイルの生成時のエポック時刻を付与しますが、これはもはや当てはまりません。

CSV書式の出力がlog_destinationで有効な場合、タイムスタンプ付きのログファイル名に.csvを付与し、最終的なCSV書式出力用のファイル名が作成されます。(log_filenameが.logで終わる場合は後置詞が置き換えられます。) 上の例の場合、ファイル名はserver_log.1093827753.csvとなります。

このパラメータはpostgresql.confか、サーバコマンドラインのみ設定可能です。

log_file_mode (integer)

Unixシステムにおいては、logging_collectorが有効になっている場合、このパラメータはログファイルのパーミッションを設定します。(Microsoft Windowsではこのパラメータは無視されます。) パラメータの値はchmod およびumaskシステムコールで許可されるフォーマットで指定される数値モードであると期待されます。(慣例的な8進数フォーマットを使用する場合、番号は0(ゼロ)で始まらなければなりません。

デフォルトのパーミッションは0600で、意味するところはサーバの所有者のみログファイルの読み書きが可能です。そのほか一般的に実用的な設定は0640で、所有者のグループはファイルを読み込めます。しかし、これらの設定を活用するにはlog_directoryがクラスタデータディレクトリの外部のどこかにあるファイルを格納できるように変更する必要があります。

このパラメータはpostgresql.confファイル、またはサーバコマンドラインからのみ設定可能です。

log_rotation_age (integer)

logging_collectorが有効な場合、このパラメータは個々のログファイルの最大寿命を決定します。ここで指定した時間(分単位)経過すると、新しいログファイルが生成されます。ゼロに設定することで、時間に基づいた新しいログファイルの生成は無効になります。このパラメータは、postgresql.confファイル、または、サーバのコマンドラインでのみで設定されます。

log_rotation_size (integer)

logging_collectorが有効な場合、このパラメータは個々のログファイルの最大要領を決定します。ここで指定したキロバイト分ログファイルに出力された後、新しいログファイルが生成されます。ゼロに設定することで、サイズに基づいた新しいログファイルの生成は無効になります。このパラメータはpostgresql.confファイル、または、サーバのコマンドラインでのみで設定されます。

log_truncate_on_rotation (boolean)

logging_collectorが有効な場合、このパラメータにより、PostgreSQLは既存の同名のファイルに追加するのではなく、そのファイルを切り詰める(上書きする)ようになります。しかし、切り詰めは時間を基にしたローテーションのために新規にファイルが開かれた時にのみ発生し、サーバ起動時やサイズを基にしたローテーションでは発生しません。偽の場合、全ての場合において既存のファイルは追記されます。例えば、この設定をpostgresql-%H.logのようなlog_filenameと組み合わせて使用すると、24個の時別のログファイルが生成され、それらは周期的に上書きされることになります。このパラメータはpostgresql.confファイル、または、サーバのコマンドラインで設定されます。

例: 7日間のログを保存し、毎日のログを server_log.Mon、server_log.Tue、等とし、そして自動的に前週のログを今週のログで上書きするには以下のように設定します。log_filename を server_log.%aとし、log_truncate_on_rotation を onにし、そして log_rotation_age を 1440に設定します。

例: 24時間のログを保持、1時間おきに1つのログファイルを作成、ただし、ログファイルのサイズが1ギガバイトを超えそうな場合即座に切り替えさせるには、log_filename を server_log.%H%Mにし、log_truncate_on_rotation を onにし、log_rotation_age を 60にし、そして log_rotation_size を 1000000に設定します。log_filenameに%Mを含めると、元の時間毎のファイル名と異なる名前を選択する可能性がある、サイズを元にしたローテーションを行うことができます。

syslog_facility (enum)

syslogへのログ取得が有効な場合、このパラメータはsyslogの"facility"が使われるように確定します。

LOCAL0、LOCAL1、LOCAL2、LOCAL3、LOCAL4、LOCAL5、LOCAL6、LOCAL7の中から選んでください。デフォルトはLOCAL0です。使用しているシステムのsyslogデーモンの文書を同時に参照してください。このパラメータはpostgresql.confファイル、または、サーバのコマンドラインでのみで設定されます。

syslog_ident (string)

syslogにログ取得が有効な場合、このパラメータはsyslogログ内のPostgreSQLメッセージを特定するのに使用するプログラム名を確定します。デフォルトはpostgresです。このパラメータは、postgresql.confファイル、または、サーバの

コマンドラインでのみで設定されます。

event_source (string)

event logへのログ取得が有効になっていると、このパラメータはログ中のPostgreSQLメッセージを特定するのに使用されるプログラム名を決定します。デフォルトはPostgreSQLです。このパラメータは、postgresql.conf ファイル、または、サーバのコマンドラインでのみで設定されます。

18.8.2. いつログを取得するか

client_min_messages (enum)

どのメッセージ階層をクライアントに送るかを管理します。有効な値は、DEBUG5、DEBUG4、DEBUG3、DEBUG2、DEBUG1、LOG、NOTICE、WARNING、ERROR、FATAL、およびPANICです。それぞれの階層はそれに続く全ての階層を含みます。階層が後の方になるにつれ、より少ないメッセージが送られます。デフォルトはNOTICEです。ここでのLOGの優先順位がlog_min_messagesの場合と異なることに注意してください。

log_min_messages (enum)

どのメッセージ階層をサーバログに書き込むかを管理します。有効な値は、DEBUG5、DEBUG4、DEBUG3、DEBUG2、DEBUG1、INFO、NOTICE、WARNING、ERROR、LOG、FATAL、およびPANICです。それぞれの階層はその下の全ての階層を含みます。階層を低くする程、より少ないメッセージがログに送られます。デフォルトはWARNINGです。ここでのLOGの優先順位がclient_min_messagesの場合と異なることに注意してください。スーパーユーザのみこの設定を変更できます。

log_min_error_statement (enum)

エラー条件の原因となったどのSQL文をサーバログに記録するかを制御します。設定したレベル以上のメッセージについては現在のSQL文がログに記録されます。有効な値は、DEBUG5、DEBUG4、DEBUG3、DEBUG2、DEBUG1、INFO、NOTICE、WARNING、ERROR、LOG、FATAL、PANICです。デフォルトはERRORです。エラー、ログメッセージ、致命的エラー、パニックを引き起こした文がログに記録されることを意味します。失敗した文の記録を実質的に無効にするには、このパラメータをPANICに設定してください。スーパーユーザのみがこのオプションを変更することができます。

log_min_duration_statement (integer)

文の実行に少なくとも指定したミリ秒数かかった場合、それぞれの文の実行に要した時間をログに記録します。0に設定すれば、すべての文の実行時間が出力されます。-1（デフォルト）は、文実行時間の記録を無効にします。例えば、250msと設定した場合、250msもしくはそれ以上長くかかった全てのSQL文がログとして残ります。このパラメータを有効にすることにより、アプリケーションで最適化されていない問い合わせを追跡するのが便利になります。スーパーユーザのみこの設定を変更できます。

拡張問い合わせプロトコルを使用するクライアントでは、Parse、Bind、Executeそれぞれの段階で要した時間が独立して記録されます。

注意: このオプションとlog_statementと一緒に使用する時、log_statementによってログされるテキスト文は、実行時間のログには重複されません。syslogを使用していなければ、プロセスIDとセッションIDを使用して、文メッセージと後の実行時間メッセージを関連付けできるように、log_line_prefixを使用してPIDまたはセッションIDをログに記録することを勧めます。

表18-1で、PostgreSQLで使用するメッセージ深刻度レベルを説明します。ログ出力がsyslogまたはWindowsのeventlogに送られる場合、この深刻度レベルは表で示すように変換されます。

表 18-1. メッセージ深刻度レベル

深刻度	使用方法	syslog	eventlog
DEBUG1..DEBUG5	開発者が使用する連続的かつより詳細な情報を提供します。	DEBUG	INFORMATION
INFO	VACUUM VERBOSEの出力などの、ユーザによって暗黙的に要求された情報を提供します。	INFO	INFORMATION
NOTICE	長い識別子の切り詰めに関する注意など、ユーザの補助になる情報を提供します。	NOTICE	INFORMATION
WARNING	トランザクションブロック外でのCOMMITの様な、ユーザへの警告を提供します。	NOTICE	WARNING
ERROR	現在のコマンドを中断させる原因となったエラーを報告します。	WARNING	ERROR
LOG	チェックポイントの活動の様な、管理者に関心のある情報を報告します。	INFO	INFORMATION
FATAL	現在のセッションを中断させる原因となったエラーを報告します。	ERR	ERROR

PANIC	全てのデータベースセッションを中断させる原因となったエラーを報告します。	CRIT	ERROR

18.8.3. 何をログに

application_name (string)

application_nameはNAMEDATALEN（標準構築では64）文字以下の任意の文字列を指定できます。通常はサーバへの接続時にアプリケーションによって設定されます。この名前は pg_stat_activityビューに表示され、CSVログに含まれます。またlog_line_prefixパラメータにより通常のログ項目に含めることができます。application_nameには表示可能なASCII文字のみ使用することができ、それ以外の文字は疑問符(?)に置換されます。

debug_print_parse (boolean)
debug_print_rewritten (boolean)
debug_print_plan (boolean)

これらのパラメータは生成される各種デバッグ出力を有効にします。設定すると実行された問い合わせそれぞれに対し、最終的な解析ツリー、問い合わせリライタの出力、実行計画を出力します。これらのメッセージはLOGメッセージレベルで出力されますので、デフォルトではサーバログに出力され、クライアントには渡されません。
[client_min_messages](#)、[log_min_messages](#)またはその両方を調整することで変更することができます。デフォルトではこれらのパラメータは無効です。

debug_pretty_print (boolean)

設定された場合、debug_print_parse、debug_print_rewritten、または debug_print_planで生成されたメッセージを字下げします。設定されない場合の"コンパクト"形式よりもより見やすく、しかしより長いものとなります。デフォルトは有効です。

log_checkpoints (boolean)

チェックポイントおよびリスタートポイントをサーバログに記録するようにします。書き出されたバッファ数や書き出しに要した時間など、いくつかの統計情報がこのログメッセージに含まれます。このパラメータはpostgresql.confファイルまたはサーバのコマンドラインでのみ設定可能です。デフォルトはoffです。

log_connections (boolean)

これにより、クライアント認証の成功終了などのサーバへの接続試行がログに残ります。このパラメータはセッションが開始された後は変更できません。デフォルトはoffです。

注意: psqlなどクライアントプログラムの中には、パスワードを要求されると決定まで2回接続を試みるので、二重の"connection received"メッセージは必ずしも問題を示すものではありません。

log_disconnections (boolean)

これはlog_connectionsと同様の内容をサーバログに行として出力します。しかし、セッションが完了した時点であって、かつセッションが持続していたことが条件です。デフォルトではoffです このパラメータはセッションが開始された後は変更できません。

log_duration (boolean)

すべての完了した文について、その経過時間をログするようにします。デフォルトはoffです。スーパーユーザのみがこの設定を変更することができます。

拡張問い合わせプロトコルを使用するクライアントでは、Parse、Bind、Executeそれぞれの段階で要した時間が独立して記録されます。

注意: このオプションと[log_min_duration_statement](#)を0に設定する方法との違いは、log_min_duration_statementを超えた場合、テキスト版の問い合わせが強制的に出力されるのに対して、このオプションでは出力されないという点です。したがって、log_durationがon、かつ、log_min_duration_statementが正の値を持つ場合、すべての経過時間がログに記録されますが、閾値を超えた文のみがテキスト版の問い合わせが含まれるようになります。この動作は、高負荷なインストレーションで統計情報を収集する際に有用です。

log_error_verbosity (enum)

ログ取得されるそれぞれのメッセージに対し、サーバログに書き込まれる詳細の量を制御します。有効な値は、TERSE、DEFAULT、およびVERBOSEで、それぞれは表示されるメッセージにより多くのフィールドを追加します。TERSEはDETAIL、HINT、QUERY、およびCONTEXTエラー情報を除外します。VERBOSE出力は、SQLSTATEエラーコード ([付](#)

[録A](#)も参照)、および、ソースコードファイル名、関数名、そしてエラーを生成した行番号を含みます。 スーパユーザのみこの設定を変更できます。

log_hostname (boolean)

デフォルトでは、接続ログメッセージは接続元ホストのIPアドレスのみを表示します。 このパラメータを有効にすると、ホスト名もログに残るようになります。 ホスト名解決方法の設定に依存しますが、これが無視できないほどの性能劣化を起こす可能性があることに注意してください。 このパラメータは`postgresql.conf`ファイル内またはサーバのコマンドラインでのみ設定可能です。

log_line_prefix (string)

これは、各ログ行の先頭に出力する`printf`の書式文字列です。 `%`から始まる"エスケープシーケンス"は、後述の通りのステータス情報で置き換えられます。 この他のエスケープは無視されます。 他の文字はそのままログ行に出力されます。 エスケープの中には、セッションプロセスによってのみ認識可能なものがあり、これらはメインサーバプロセスなどのバックグラウンドプロセスでは無視されます。 このパラメータは、`postgresql.conf`ファイル、または、サーバのコマンドラインでのみ設定することができます。 デフォルトは空文字列です。

エスケープ	効果	セッションのみ
<code>%a</code>	アプリケーション名	☐
<code>%u</code>	ユーザ名	☐
<code>%d</code>	データベース名	☐
<code>%r</code>	遠隔ホスト名、またはIPアドレス、およびポート番号	☐
<code>%h</code>	遠隔ホスト名、またはIPアドレス	☐
<code>%p</code>	プロセス識別子	☒
<code>%t</code>	ミリ秒無しのタイムスタンプ	☒
<code>%m</code>	ミリ秒付きタイムスタンプ	☒
<code>%i</code>	コマンドタグ。セッションの現在のコマンド種類	☐
<code>%e</code>	SQLSTATE エラーコード	☒
<code>%c</code>	セッションID。下記参照	☒
<code>%l</code>	各セッションまたは各プロセスのログ行の番号。1から始まります。	☒
<code>%s</code>	プロセスの開始タイムスタンプ	☒
<code>%v</code>	仮想トランザクションID (backendID/localXID)	☒
<code>%x</code>	トランザクションID (未割り当ての場合は0)	☒
<code>%q</code>	何も出力しません。 非セッションプロセスではこのエスケープ以降の出力を停止します。 セッションプロセスでは無視されます。	☒
<code>%%</code>	<code>%</code> 文字そのもの	☒

`%c`エスケープは、2つの4バイトの16進数（先頭のゼロは省略）をドットで区切った構成の、準一意なセッション識別子を表示します。 この数値はプロセスの起動時間とそのプロセスIDです。 したがって、`%c`を使用して、これらの項目を出力するための文字数を省略することができます。 例として、`pg_stat_activity`からセッション識別子を生成するには以下の問い合わせを行ないます。

```
SELECT to_hex(EXTRACT(EPOCH FROM backend_start)::integer) || '.' ||
       to_hex(pid)
FROM pg_stat_activity;
```

ティップ: `log_line_prefix`に空白文字以外の値を設定する場合、通常、ログ行の残りとの区切りを明確にするために、その最後の文字を空白文字にすべきです。 句読点用の文字も使用できます。

ティップ: Syslogは独自にタイムスタンプとプロセスID情報を生成します。 ですのでおそらく、Syslogにログを保管する場合は、こうしたエスケープを含めるとは考えないでしょう。

log_lock_waits (boolean)

セッションがロックの獲得までの間に[deadlock timeout](#)より長く待機する場合にログメッセージを生成するかどうかを制御します。 これは、ロックが待たされ性能がでていないのかどうか決める時に有用です。 デフォルトは`off`です。

log_statement (enum)

どのSQL文をログに記録するかを制御します。有効な値は、none (off)、ddl、mod、およびall (全てのメッセージ) です。ddlは、CREATE、ALTER、およびDROP文といった、データ定義文を全てログに記録します。modは、全てのddl文に加え、INSERT、UPDATE、DELETE、TRUNCATE、およびCOPY FROMといった、データ変更文をログに記録します。PREPAREとEXPLAIN ANALYZEコマンドも、そこに含まれるコマンドが適切な種類であればログが録られます。拡張問い合わせプロトコルを使用するクライアントでは、Executeメッセージを受け取った時にBindパラメータの値が (すべての単一引用符が二重にされた状態で) 含まれていた場合、ログに記録されます。

デフォルトはnoneです。スーパーユーザのみこの設定を変更できます。

注意: ログメッセージの発行は、基本解析により文の種類が決まった後に行われますので、log_statement = allという設定を行ったとしても、単純な構文エラーを持つ文は記録されません。拡張問い合わせプロトコルの場合も同様に、この設定ではExecute段階以前 (つまり、解析や計画作成期間) に失敗した文は記録されません。こうした文のログを記録するには、log_min_error_statementをERROR (以下) に設定してください。

log_temp_files (integer)

一時ファイル名と容量を制御します。一時ファイルはソート処理やハッシュ処理、一時的な問い合わせの結果のために作成されます。ログの項目はすべての一時ファイルそれぞれについて削除されたときに生成されます。ゼロという値ではすべての一時ファイル情報のログが残ります。正数ではキロバイト単位の指定値以上の容量のファイルのみがログに残ります。デフォルトの設定は-1で、このログ処理を無効にします。スーパーユーザのみがこの設定を変更できます。

log_timezone (string)

サーバログに書き出す際に使用される時間帯を設定します。TimeZoneと異なり、すべてのセッションで一貫性を持ってタイムスタンプが報告されるようにこの値はクラスタ全体に適用されます。組み込まれているデフォルトはGMTですが、postgresql.confにより通常は上書きされます。initdbによりこれらと関連した設定をシステム環境にインストールされます。詳細は項8.5.3を参照してください。このパラメータはpostgresql.conf内またはサーバのコマンドラインでのみ設定することができます。

18.8.4. CSV書式のログ出力の利用

log_destination リストにcsvlogを含めることは、ログファイルをデータベーステーブルにインポートする簡便な方法を提供します。このオプションはカンマ区切り値書式 ((CSV) で以下の列を含むログ行を生成します。ミリ秒単位のtimestamp、ユーザ名、データベース名、プロセス識別子、クライアントホスト:ポート番号、セッション識別子、セッション前行番号、コマンドタグ、セッション開始時間、仮想トランザクション識別子、通常トランザクション識別子、エラーの深刻度、SQL状態コード、エラーメッセージ、詳細エラーメッセージ、ヒント、エラーとなった内部的な問い合わせ (もしあれば)、内部問い合わせにおけるエラー位置の文字数、エラーの文脈、PostgreSQLソースコード上のエラー発生場所 (log_error_verbosityがverboseに設定されているならば) アプリケーション名 以下にcsvlog出力を格納するためのテーブル定義のサンプルを示します。

```
CREATE TABLE postgres_log
(
    log_time timestamp(3) with time zone,
    user_name text,
    database_name text,
    process_id integer,
    connection_from text,
    session_id text,
    session_line_num bigint,
    command_tag text,
    session_start_time timestamp with time zone,
    virtual_transaction_id text,
    transaction_id bigint,
    error_severity text,
    sql_state_code text,
    message text,
    detail text,
    hint text,
    internal_query text,
    internal_query_pos integer,
    context text,
    query text,
    query_pos integer,
    location text,
    application_name text,
    PRIMARY KEY (session_id, session_line_num)
);
```

このテーブルにインポートするためには、COPY FROMコマンドを使用してください。

```
COPY postgres_log FROM '/full/path/to/logfile.csv' WITH csv;
```


CSVログファイルをインポートする作業を単純にするためにいくつか必要な作業があります。

- 1 11 一貫性があり、予測可能なログファイル命名機構を提供するために、`log_filename`および`log_rotation_age`を設定してください。これによりどのようなファイル名になると、個々のログファイルが完了しインポートする準備が整ったかが推測できるようになります。
- 1 11 ログファイル名の予測が困難になりますので、`log_rotation_size`を0にして容量を基にしたログの回転を無効にしてください。
- 1 11 同じファイルに古いログデータと新しいログデータが混在しないようにするために、`log_truncate_on_rotate`をonに設定してください。
- 1 11 上のテーブル定義にはプライマリキーの指定が含まれています。これにより、同じ情報が2回インポートされる事故を防止するために有用です。`COPY`コマンドは、一度にインポートするすべてのデータをコミットしますので、何か1つでもエラーがあればインポート全体が失敗します。ログファイルの一部をインポートし、そのファイルが完了した後に再度インポートしようとした場合、プライマリキー違反によりインポートが失敗します。インポートする前に、ログファイルの完了を待ち、閉じるまで待機してください。この手順は、`COPY`が失敗する原因となる、完全に書き込まれなかった欠落した行をインポートするという事故も防止します。

[前のページ](#)
問い合わせ計画

[ホーム](#)
[上に戻る](#)

[次のページ](#)
実行時統計情報

18.9. 実行時統計情報

18.9.1. 問い合わせおよびインデックスに関する統計情報コレクタ

これらのパラメータは、サーバ全体の統計情報収集機能を制御します。統計情報収集が有効ならば、生成されるデータは `pg_stat` と `pg_statio` 系のシステムビュー経由でアクセス可能です。詳細は [第27章](#) を参照してください。

`track_activities` (boolean)

各セッションで実行中のコマンドに関する情報とそのコマンドの実行開始時刻の収集を有効にします。このパラメータはデフォルトで有効です。有効な場合であっても、すべてのユーザがこの情報を見ることができず、スーパーユーザと報告されたセッションの所有者のみから可視である点に注意してください。このためセキュリティ上の危険性はありません。スーパーユーザのみがこの設定を変更することができます。

`track_activity_query_size` (integer)

`pg_stat_activity.query` フィールドに対し、それぞれの活動中のセッションで現在実行されているコマンドを追跡記録するため予約されるバイト数を指定します。デフォルトの値は1024です。このパラメータはサーバ起動時のみ設定可能です。

`track_counts` (boolean)

データベースの活動についての統計情報の収集を有効にします。収集される情報を自動バキュームデーモンが必要とするため、このオプションはデフォルトで有効です。スーパーユーザのみがこの設定を変更することができます。

`track_io_timing` (boolean)

データベースによるI/O呼び出しの時間的調節を可能にします。このパラメータはデフォルトで無効になっています。その理由は、現時点の時刻をオペレーティングシステムに繰り返し問い合わせるので、プラットフォームによっては深刻な負荷の原因になります。自身の使用している負荷のタイミングを計測するため [pg_test_timing](#) ツールが使用できます。I/O タイミング情報は、`BUFFERS` オプションが設定されている時 [EXPLAIN](#) の出力として、また [pg_stat_statements](#) により表示されます。スーパーユーザのみこの設定を変更できます。

`track_functions` (enum)

関数の呼び出し数と費やされた時間の追跡を有効にします。手続き言語関数のみを追跡するためには `p1` と指定してください。SQL関数、C言語関数も追跡するためには `all` と指定してください。デフォルトは、統計情報追跡機能を無効にする `none` です。スーパーユーザのみがこの設定を変更できます。

注意: 呼び出す問い合わせ内に"インライン化"できる位単純なSQL言語関数は、この設定と関係なく、追跡されません。

`update_process_title` (boolean)

サーバが新しいSQLコマンドを受け取る時に毎回、プロセスタイトルを更新できるようにします。プロセスタイトルは通常、`ps` コマンドやWindowsではProcess Explorerを使用して表示させることができます。スーパーユーザのみがこの設定を変更することができます。

`stats_temp_directory` (string)

統計情報データを一時的に格納するディレクトリを設定します。これをデータディレクトリからの相対パスとすることも絶対パスとすることもできます。デフォルトは `pg_stat_tmp` です。これをRAMベースのファイルシステムを指し示すようにすることで物理I/O要求が減り、性能を向上させることができます。このパラメータは、`postgresql.conf` ファイルまたはサーバのコマンドラインのみで設定可能です。

18.9.2. 統計情報の監視

`log_statement_stats` (boolean)

`log_parser_stats` (boolean)

`log_planner_stats` (boolean)

`log_executor_stats` (boolean)

各問い合わせに対し、対応するモジュールの性能に関する統計情報をサーバログに出力します。これは、Unixの`getrusage()`オペレーティングシステム機能に類似した、雑なプロファイリング手段です。`log_statement_stats`は文に関する統計情報全体を、この他はモジュール毎の統計情報を報告します。`log_statement_stats`とモジュール毎のオプションを一緒に有効にすることはできません。デフォルトでこれらのオプションはすべて無効です。スーパーユーザのみがこの設定を変更することができます。

[前のページ](#)

エラー報告とログ取得

[ホーム](#)
[上に戻る](#)

[次のページ](#)
自動Vacuum作業

18.10. 自動Vacuum作業

以下に示す設定は自動バキューム機能の動作を制御します。詳細は[項23.1.6](#)を参照してください。

`autovacuum (boolean)`

サーバがautovacuumランチャデーモンを実行すべきかどうかを管理します。デフォルトでは有効です。しかしautovacuumを作動させるためには[track_counts](#)も有効でなければなりません。このパラメータは`postgresql.conf` ファイル、または、サーバのコマンドラインでのみで設定されます。

このパラメータが無効であってとしてもシステムは、トランザクションIDの周回を防止する必要があるれば、autovacuumプロセスを起動することに注意してください。詳細は[項23.1.5](#)を参照してください。

`log_autovacuum_min_duration (integer)`

少なくとも指定ミリ秒実行した場合、autovacuumで実行される各活動がログに残るようになります。これをゼロに設定すると、すべてのautovacuumの活動がログに残ります。マイナス1（デフォルト）はautovacuum活動のログを無効にします。例えば、これを250msに設定すると、250ms以上かかって実行されたautovacuumや解析はすべてログに残ります。さらに、-1以外の値にこのパラメータが設定された場合、競合するロックの存在によりオートバキューム動作が省略されるとメッセージはログに記録されます。このパラメータを有効にすることは、autovacuum活動の追跡に役に立ちます。このパラメータは`postgresql.conf` ファイル、または、サーバのコマンドラインでのみで設定されます。

`autovacuum_max_workers (integer)`

同時に実行することができるautovacuumプロセス（autovacuumランチャ以外）の最大数を指定します。デフォルトは3です。サーバ起動時のみで設定可能です。

`autovacuum_naptime (integer)`

あるデータベースについて実行されるautovacuumデーモンの最小遅延を指定します。それぞれの周期で、デーモンはそのデータベースを試験し、そしてそのデータベース内のテーブルで必要性が認められると、VACUUMおよびANALYZE コマンドを発行します。遅延は秒単位で計測され、デフォルトは1分（1min）です。このパラメータは`postgresql.conf` ファイル、または、サーバのコマンドラインでのみで設定されます。

`autovacuum_vacuum_threshold (integer)`

いかなる1つのテーブル内にVACUUMをトリガする必要がある、更新もしくは削除されたタプルの最小数を指定します。デフォルトは50タプルです。このパラメータは`postgresql.conf` ファイル、または、サーバのコマンドラインでのみで設定されます。この設定は`pg_autovacuum`内の項目により、それぞれのテーブルに対して上書きすることができます。

`autovacuum_analyze_threshold (integer)`

いかなる1つのテーブル内にANALYZEをトリガする必要がある、挿入、更新、もしくは削除されたタプルの最小数を指定します。デフォルトは50タプルです。このパラメータは`postgresql.conf` ファイル、または、サーバのコマンドラインでのみで設定されます。この設定はにより、それぞれのテーブルに対して上書きすることができます。

`autovacuum_vacuum_scale_factor (floating point)`

VACUUMをトリガするか否かを決定する場合、`autovacuum_vacuum_threshold`に追加するテーブル容量の断片を指定します。デフォルトは0.2（テーブルサイズの20%）です。このパラメータは`postgresql.conf` ファイル、または、サーバのコマンドラインでのみで設定されます。この設定は格納パラメータの変更により、それぞれのテーブルに対して上書きされます。

`autovacuum_analyze_scale_factor (floating point)`

ANALYZEをトリガするか否かを決定する場合、`autovacuum_vacuum_threshold`に追加するテーブル容量の小部分を指定します。デフォルトは0.1（テーブルサイズの10%）です。このパラメータは`postgresql.conf` ファイル、または、サーバのコマンドラインでのみで設定されます。この設定は格納パラメータの変更により、それぞれのテーブルに対して上書きされます

`autovacuum_freeze_max_age (integer)`

トランザクションID周回を防ぐためにVACUUM操作が強制される前までにテーブルの`pg_class.relfrozensxid` フィールドが到達できる最大（トランザクションにおける）年代を指定します。自動バキュームが無効であった時でも、システムは周

回を防ぐために自動バキューム子プロセスを起動することに注意してください。

`vacuum`は同時に`pg_clog`サブディレクトリから古いファイルの削除を許可します。これはなぜフォルトが2億トランザクションをやや下回る理由です。このパラメータはサーバ起動時にのみ設定可能です。しかし、この設定は格納パラメータの変更により、それぞれのテーブルで減らすことができます。詳細は[項23.1.5](#)を参照してください。

`autovacuum_vacuum_cost_delay (integer)`

自動`VACUUM`操作に使用されるコスト遅延値を指定します。-1に指定されると、一定の [vacuum_cost_delay](#)の値が使用されます。デフォルト値は20ミリ秒です。このパラメータは`postgresql.conf`ファイル内、または、サーバのコマンドラインのみで設定可能です。この設定は格納パラメータの変更により、それぞれのテーブルに対して上書きすることができます。

`autovacuum_vacuum_cost_limit (integer)`

自動`VACUUM`操作に使用されるコスト限界値を指定します。（デフォルトの）-1が指定されると、一定の [vacuum_cost_limit](#)の値が使用されます。この値は、実行中の自動バキュームワーカが複数存在する場合ワーカすべてに比例分配されることに注意してください。したがって各ワーカの制限を足し合わせてもこの変数による制限を超えることはありません。このパラメータは`postgresql.conf`ファイル、または、サーバのコマンドラインのみで設定可能です。この設定は格納パラメータの変更により、それぞれのテーブルに対して上書きすることができます。

[前のページ](#)

実行時統計情報

[ホーム](#)

[上に戻る](#)

[次のページ](#)

クライアント接続デフォルト

18.11. クライアント接続デフォルト

18.11.1. 文の動作

`search_path (string)`

この変数は、オブジェクト（テーブル、データ型、関数など）がスキーマを指定されていない単純な名前で参照されている場合に、スキーマを検索する順番を指定します。異なるスキーマに同じ名前のオブジェクトがある場合、検索パスで最初に見つかったものが使用されます。検索パス内のどのスキーマにも存在しないオブジェクトを参照するには、修飾名（ドット付き）でそのオブジェクトが含まれるスキーマを指定する必要があります。

`search_path`の値は、スキーマの名前をカンマで区切った一覧でなければなりません。存在していないスキーマ、またはユーザが`USAGE`権限を所有していないスキーマは警告なしに無視されます。

If one of the list items is the special name `$user`, then the schema having the name returned by `SESSION_USER` is substituted, if there is such a schema and the user has `USAGE` permission for it. (If not, `$user` is ignored.) --> もしそのようなスキーマが存在し、ユーザがそれにたいして`USAGE`権限を所有している場合、一覧内の項目の1つが特別な名前である`$user`の場合、`SESSION_USER`と同じ名前を持つスキーマがあれば、そのスキーマが置換されます。（このような名前空間がない場合は`$user`は無視されます。）

システムカタログのスキーマである`pg_catalog`は、パスでの指定の有無にかかわらず、常に検索されます。パスで指定されている場合は、指定された順序で検索されます。`pg_catalog`がパスに含まれていない場合、パスに含まれる項目を検索する前に検索が行われます

同様に、現在のセッションの一時テーブルスキーマ`pg_temp_nnn`も、存在すれば常に検索されます。これは`pg_temp`という別名を使用してパスに明示的に列挙させることができます。パスに列挙されていない場合、最初に（`pg_catalog`よりも前であっても）検索されます。しかし、一時スキーマはリレーション（テーブル、ビュー、シーケンスなど）とデータ型名に対してのみ検索されます。関数や演算子名に対してはまったく検索されません。

対象となる特定のスキーマを指定せずにオブジェクトが作成された場合、それらのオブジェクトは`search_path`で名前を付けられた最初に有効となっているスキーマに配置されます。検索パスが空の場合、エラーが報告されます。

このパラメータのデフォルト値は`"$user", public`です。この設定はデータベースの共有（どのユーザも非公開のスキーマを持たず、全員が`public`を共有）、ユーザごとの非公開のスキーマ、およびこれらの組み合わせがサポートします。デフォルトの検索パスの設定を全体的またはユーザごとに変更することで、その他の効果を得ることもできます。

SQL関数の`current_schemas`によって、検索パスの現在の有効な値を調べることができます（[項9.25](#)を参照してください）。これは、`search_path`の値を調べるのとは異なります。`current_schemas`は、`search_path`に現れる項目がどのように解決されたかを表すからです。

スキーマ操作に付いてより多くの情報は、[項5.7](#)を参照してください。

`default_tablespace (string)`

この変数は、`CREATE`コマンドで明示的にテーブル空間を指定していない場合にオブジェクトの作成先となるデフォルトのテーブル空間を指定します。

値はテーブル空間名、もしくは現在のデータベースのデフォルトのテーブル空間を使用することを意味する空文字列です。この値が既存のテーブル空間名と一致しない場合、PostgreSQLは自動的に現在のデータベースのデフォルトのテーブル空間を使用します。デフォルト以外のテーブル空間が指定された場合、ユーザはそのテーブル空間で`CREATE`権限を持たなければなりません。さもなければ作成に失敗します。

この変数は一時テーブル向けには使用されません。一時テーブル向けには代わりに[temp tablespaces](#)が考慮されます。

同時に、この変数はデータベース作成時には使用されません。

テーブル空間に付いてより詳細な情報は[項21.6](#)を参照してください。

`temp_tablespaces (string)`

この変数は、`CREATE`コマンドで明示的にテーブル空間が指定されない場合に、生成する一時オブジェクト（一時テーブルと一時テーブル上のインデックス）を格納するテーブル空間（複数可）を指定します。大規模データ集合のソートなどを目的とした一時ファイルもまた、このテーブル空間（複数可）に作成されます。

この値はテーブル空間名のリストです。 リストに複数の名前が存在する場合、一時オブジェクトが作成される度にPostgreSQLは無作為にリストから要素を選択します。 トランザクションの内側は例外で、連続して作成される一時オブジェクトはそのリストで連続するテーブル空間に格納されます。 リスト内の選択された要素が空文字列だった場合、PostgreSQLは自動的に現在のデータベースのデフォルトのテーブル空間を代わりに使用します。

`temp_tablespaces`を対話式に設定する場合、存在しないテーブル空間を指定するとエラーになります。 ユーザが`CREATE`権限を持たないテーブル空間を指定した場合も同様です。 しかし事前に設定された値を使用する場合、存在しないテーブル空間は無視されます。 ユーザが`CREATE`権限を持たないテーブル空間も同様です。 具体的には、この規則は`postgresql.conf`内で設定した値を使用する場合に適用されます。

デフォルト値は空文字列です。 この結果、すべての一時オブジェクトは現在のデータベースのデフォルトのテーブル空間内に作成されます。

[default_tablespace](#)も参照してください

`check_function_bodies (boolean)`

このパラメータは通常オンです。 `off`に設定すると、[CREATE FUNCTION](#)の間で関数本体文字列の妥当性検証を無効にします。 妥当性検証を無効にすることは、時たまダンプから関数定義をリストアする時に参照を転送するような問題を回避する場合には便利です。

`default_transaction_isolation (enum)`

SQLトランザクションはそれぞれ、`"read uncommitted"`、`"read committed"`、`"repeatable read"`、または`"serializable"`のいずれかの隔離レベルを持ちます。 このパラメータは各新規トランザクションのデフォルトの隔離レベルを制御します。 デフォルトは`"read committed"`です。

より詳細は [第13章](#) および [SET TRANSACTION](#) を調べてください。

`default_transaction_read_only (boolean)`

読み取り専用のSQLトランザクションでは、非一時的テーブルを変更することができません。 このパラメータは、各新規トランザクションのデフォルトの読み取りのみ状況を制御します。 デフォルト`off`（読み書き）です。

より詳細な情報は[SET TRANSACTION](#)を調べてください。

`default_transaction_deferrable (boolean)`

シリアライズ可能隔離レベルで運用されている場合、繰り延べ読み取り専用SQLトランザクションは、その処理の許可の前に遅延されることがあります。 しかし、ひとたび処理が開始されるとシリアライズ可能可能性を保障するために必要ないかなるオーバーヘッドも発生させません。 従って、シリアル化（直列化）のコードは、このオプションを長期間にわたる読み取り専用トランザクションに対して適切な処置と位置づけ、同時実行の更新の観点から中断を強制する理由はありません。

このパラメータはそれぞれの新規トランザクションのデフォルトでの繰り延べ状態を制御します。 現時点では、読み取り専用トランザクション、またはシリアライズ可能より低位の隔離レベルの運用に対して効果はありません。

より詳細は[SET TRANSACTION](#)を参照してください。

`session_replication_role (enum)`

現在のセッションでのレプリケーションに関連したトリガおよびルールの発行を制御します。 この変数を設定するにはスーパーユーザ権限が必要で、かつ、これまでにキャッシュされた問い合わせ計画が破棄されることになります。 取り得る値は、`origin`（デフォルト）、`replica`、`local`です。 詳細については[ALTER TABLE](#)を参照してください。

`statement_timeout (integer)`

指定されたミリ秒数以上、コマンドがクライアントからサーバに届いてから経過したどんなコマンドの実行も停止します。 もし、`log_min_error_statement`が`ERROR`もしくはそれ以下に設定されると、タイムアウトする文は同時にログに書き込まれます。 値がゼロ（デフォルト）の場合、これを無効にします。

すべてのセッションに影響しますので、`postgresql.conf`内で`statement_timeout`を設定することは推奨されません。

`vacuum_freeze_table_age (integer)`

テーブルの`pg_class.relfrozenxid`フィールドがこの設定で指定した時期に達すると、`VACUUM`はテーブル全体の走査を行います。 デフォルトは1.5億トランザクションです。 ユーザはこの値をゼロから10億までの任意の値に設定することができますが、`VACUUM`は警告することなく、周回問題対策の`autovacuum`がテーブルに対して起動する前に定期的な手動`VACUUM`が実行する機会を持つように、[autovacuum freeze max age](#)の95%に実効値を制限します。 [項23.1.5](#)を参

照してください。

`vacuum_freeze_min_age (integer)`

VACUUMがテーブルスキャン時にトランザクションIDをFrozenXIDに置き換えるかどうかを決定する際に使用する、カットオフ（トランザクション）年代を指定します。デフォルトは5千万トランザクションです。ユーザはこの値を0から10億までの間で任意の値に設定することができますが、VACUUMは警告なく[autovacuum_freeze_max_age](#)の半分までの値に値を制限します。このため、強制的なautovacuumの間隔が不合理に短くなることはありません。詳細は[項23.1.5](#)を参照してください。

`bytea_output (enum)`

bytea型の値の出力形式を設定します。有効な値はhex（デフォルト）、およびescape（PostgreSQLの伝統的な書式）です。より詳細は[項8.4](#)を参照してください。bytea型は常にこの設定に係わず、入力時に双方の書式を受け付けます。

`xmlbinary (enum)`

バイナリデータをXMLに符号化する方法を設定します。例えばこれは、xmlelementやxmlforest関数でbytea値をXMLに変換する際に適用されます。取り得る値はbase64とhexです。どちらもXMLスキーマ標準で定義されています。デフォルトはbase64です。XMLに関連した関数については[項9.14](#)を参照してください。

実のところこの選択はほとんど趣味の問題で、クライアントアプリケーションで起こり得る制限のみに制約されます。どちらの方法もすべての値をサポートしますが、hex符号化方式はbase64符号化方式より少し大きくなります。

`xmloption (enum)`

XMLと文字列値との変換時にDOCUMENTとするかCONTENTとするかを設定します。この説明については[項8.13](#)を参照してください。有効な値はDOCUMENTとCONTENTです。デフォルトはCONTENTです。

標準SQLに従うと、このオプションを設定するコマンドは以下のようになります。

```
SET XML OPTION { DOCUMENT | CONTENT };
```

この構文はPostgreSQLでも使用可能です。

18.11.2. ロケールと書式設定

`DateStyle (string)`

日付時刻値の表示書式を設定し、曖昧な日付入力の解釈規則を設定します。歴史的な理由により、この変数には2つの依存した要素が含まれています。出力書式指定（ISO、Postgres、SQL、German）と年/月/日の順序の入出力指定（DMY、MDY、YMD）です。これらは分けて設定することもまとめて設定することもできます。EuroおよびEuropeanキーワードはDMYの同義語であり、US、NonEuro、NonEuropeanはMDYの同義語です。詳細は[項8.5](#)を参照してください。組み込みのデフォルトはISO、MDYですが、initdbにより、選択されたlc_timeロケールの動作に対応した設定で設定ファイルが初期化されます。

`IntervalStyle (enum)`

間隔の値の表示形式を設定します。sql_standard値は、SQL標準間隔リテラルに一致する出力を生成します。（デフォルトの）値postgresは、[DateStyle](#)パラメータがISOに設定されている場合、リリース8.4以前のPostgreSQLに一致する出力を生成します。値postgres_verboseは、DateStyleパラメータが非ISO出力に設定されている場合、リリース8.4以前のPostgreSQLに一致する出力を生成します。値iso_8601は、ISO 8601の4.4.3.2節で定義されている時間間隔"format with designators"に一致する出力を生成します。

またIntervalStyleパラメータはあいまいに入力された時間間隔の解釈に影響を与えます。詳細については[項8.5.4](#)を参照してください。

`TimeZone (string)`

表示用およびタイムスタンプ解釈用の時間帯を設定します。組み込まれているデフォルトはGMTですが、通常はpostgres.confにより上書きされます。initdbによりこれらと関連した設定をシステム環境にインストールされます。詳細は[項8.5.3](#)を参照してください。

`timezone_abbreviations (string)`

サーバで日付時刻の入力として受け付け可能となる時間帯省略形の集合を設定します。デフォルトは'Default'です。これはほぼ全世界で通じる集合です。また、Australia、India、その他特定のインストラクションで定義可能な集合が存在します。詳細は[付録B](#)を参照してください。

`extra_float_digits (integer)`

このパラメータは、`float4`、`float8`、幾何データ型などの浮動小数点値の表示桁数を調整します。パラメータ値が標準的な桁数（`FLT_DIG`もしくは`DBL_DIG`どちらか適切な方）に追加されます。この値は、部分有効数を含めるために3まで設定することができます。これは基本的に、正確にリストアする必要がある浮動小数点データをダンプするために有用です。もしくは、不要な桁を抑制するために負の値を設定することもできます。

`client_encoding (string)`

クライアント側符号化方式（文字セット）を設定します。デフォルトはデータベース符号化方式を使用します。PostgreSQLサーバでサポートされている文字セットは[項22.3.1](#)に記載されています。

`lc_messages (string)`

メッセージが表示される言語を設定します。使用可能な値はシステムに依存します。詳細については[項22.1](#)を参照してください。この変数が空に設定された場合（これがデフォルトです）、値はシステムに依存する方法でサーバの実行環境から継承されます。

システムによっては、このロケールのカテゴリが存在しません。この変数を設定することはできませんが、実効性はありません。また、指定の言語に翻訳されたメッセージが存在しないこともあります。その場合は、引き続き英語のメッセージが表示されます。

サーバログやクライアントに送信されるメッセージに影響するため、および、全ての不適切な値がサーバログの信頼性を損ねる可能性があるため、スーパーユーザのみがこの設定を変更することができます。

`lc_monetary (string)`

通貨書式で使用するロケールを設定します。例えば、`to_char()`系の関数で使います。使用可能な値はシステムに依存します。詳細については[項22.1](#)を参照してください。この変数が空に設定された場合（これがデフォルトです）、値はシステムに依存する方法でサーバの実行環境から継承されます。

`lc_numeric (string)`

数字の書式で使用するロケールを設定します。例えば、`to_char`系の関数で使います。使用可能な値はシステムに依存します。詳細については[項22.1](#)を参照してください。この変数が空に設定された場合（これがデフォルトです）、値はシステムに依存する方法でサーバの実行環境から継承されます。

`lc_time (string)`

例えば`to_char`系関数における、日付と時間の書式で使用するロケールを設定します。使用可能な値はシステムに依存します。詳細については[項22.1](#)を参照してください。この変数が空に設定された場合（これがデフォルトです）、値はシステムに依存する方法でサーバの実行環境から継承されます。

`default_text_search_config (string)`

明示的な設定指定引数を持たないテキスト検索関数の亜種で使われる、テキスト検索設定を選択します。詳細は[第12章](#)を参照してください。組み込みのデフォルトは`pg_catalog.simple`ですが、`initdb`は、ロケールに合う設定を認識することができれば、選択された`lc_ctype`ロケールに対応した設定で設定ファイルを初期化します。

18.11.3. その他のデフォルト

`dynamic_library_path (string)`

オープンする必要がある動的ロード可能なモジュールについて、その`CREATE FUNCTION`や`LOAD`コマンドで指定されたファイル名にディレクトリ要素がなく（つまり、名前にスラッシュが含まれずに）指定された場合、システムは必要なファイルをこのパスから検索します。

`dynamic_library_path`の値は、絶対パスのディレクトリ名をコロン（Windowsの場合はセミコロン）を区切った一覧です。この一覧の要素が特別な`$libdir`という値から始まる場合、コンパイルされたPostgreSQLパッケージのライブラリディレクトリで`$libdir`は置換されます。ここには、PostgreSQLの標準配布物により提供されるモジュールがインストールされます（このディレクトリ名を表示するには、`pg_config --pkglibdir`を使用してください）。例を以下に示します。

```
dynamic_library_path = '/usr/local/lib/postgresql:/home/my_project/lib:$libdir'
```

Windows環境の場合は以下です。

```
dynamic_library_path = 'C:\tools\postgresql;H:\my_project\lib;$libdir'
```

このパラメータのデフォルト値は'`$libdir`'です。この値が空に設定された場合、自動的なパス検索は無効になります。

このパラメータはスーパーユーザによって実行時に変更することができますが、この方法での設定は、そのクライアント接続が終わるまでしか有効になりません。ですので、この方法は開発目的でのみ使用すべきです。推奨方法はこのパラメータを`postgresql.conf`設定ファイル内で設定することです。

`gin_fuzzy_search_limit` (integer)

GINインデックス走査により返されるセットのソフトな上限サイズです。詳細は[項55.4](#)を参照してください。

`local_preload_libraries` (string)

この変数は、接続時に事前読み込みされる、1つまたは複数の共有ライブラリを指定します。複数のライブラリが読み込まれる場合、コンマで名前を区切ってください。二重引用符で括られていない限り、全てのライブラリ名は小文字に変換されます。このパラメータは、特定のセッションが開始した後で変更することはできません。

これはスーパーユーザのみのオプションではありませんので、読み込み可能なライブラリはインストレーションの共有ライブラリディレクトリのサブディレクトリ`plugins`内にあるものに制限されています。（確実に"安全"なライブラリのみをここにインストールすることはデータベース管理者の責任です。）`local_preload_libraries`内の項目で、たとえば`$libdir/plugins/mylib`のようにこのディレクトリを明示的に指定することも、単にライブラリ名を指定することも可能です。`mylib` は`$libdir/plugins/mylib`と同じ効果です。

[shared_preload_libraries](#)とは異なり、最初使用する時にライブラリを読み込む操作とセッション開始時にライブラリを読み込む操作との間に性能的な違いはありません。しかし、この昨日の目的は、特定のセッションに明示的な`LOAD`コマンドを行わずに、デバッグ用または性能測定用のライブラリを読み込ませることを可能にすることです。たとえば、このパラメータを`ALTER ROLE SET`を使用して設定すれば、あるユーザ名で行われるすべてのセッションでデバッグが可能になります。

指定したライブラリが存在しない場合、接続に失敗します。

PostgreSQLがサポートするライブラリはすべて、互換性を保証するために検査される"マジックブロック"を持ちます。このため、この方法でPostgreSQL以外のライブラリが読み込まれることはありません。

[前のページ](#)
自動Vacuum作業

[ホーム](#)
[上に戻る](#)

[次のページ](#)
ロック管理

18.12. ロック管理

`deadlock_timeout (integer)`

これは、デッドロック状態があるかどうかを調べる前にロックを待つ時間をミリ秒で計算したものです。デッドロックの検査は比較的高価なので、サーバはロックを待つ度にこれを実行するわけではありません。楽天的ですがデッドロックは実用レベルのアプリケーションでは頻繁に発生しないと仮定し、デッドロックの検査の前にしばらくはロック待ちをします。この値を増やすことにより必要のないデッドロックの検査が無駄にされる時間は減りますが、本当にデッドロックがあった場合の報告が遅れます。デフォルトは1秒 (1s) で、おそらく実用の際にはこれ以上は必要でしょう。負荷の大きいサーバではもっと必要かもしれません。理想としてはこの設定は通常のトランザクションにかかる時間を超えているべきです。そうすればロック待ちトランザクションがデッドロックの検査をする前にロックが解除される可能性が改善されます。スーパーユーザのみこの設定を変更できます。

[log_lock_waits](#)が設定された場合、このパラメータはロック待機に関するログメッセージを出力する前の待機時間を決定します。ロック遅延の調査を行う場合は、通常の`deadlock_timeout`よりも短い値を設定することを勧めます。

`max_locks_per_transaction (integer)`

共有ロックテーブルは、`max_locks_per_transaction * (max_connections + max_prepared_transactions)` オブジェクト (例えばテーブル) 上のロック追跡します。したがって、ある時点でこの数以上の個々のオブジェクトをロックすることはできません。このパラメータは各トランザクションで割り当てられるオブジェクトロックの平均値を制御します。個々のトランザクションでは、このロックテーブルにすべてのトランザクションのロックが収まる限りオブジェクトのロックを獲得できます。これは、ロックできる行数ではありません。この値には制限がありません。デフォルトの64は、経験的に十分であると証明されていますが、単一のトランザクションで数多くの異なるテーブルをいじるクライアントがいる場合、この値を大きくする必要があるかも知れません。このパラメータはサーバ起動時のみ設定されます。

このパラメータを増加すると、使用しているオペレーティングシステムのデフォルト構成が許容するSystem V共有メモリの限界を越えた要求をPostgreSQLが行う原因となることがあります。必要であれば、どの様にしてこのパラメータを調整するかについて[項17.4.1](#)を参照ください。

スタンバイサーバを稼動するとき、このパラメータをマスターサーバと同じか、より高い値に設定しなければなりません。そうしないと、問い合わせはスタンバイサーバでは許可されません。

`max_pred_locks_per_transaction (integer)`

共有記述ロックテーブル (shared predicate lock table) は、`max_pred_locks_per_transaction * (max_connections + max_prepared_transactions)` オブジェクト (例えば諸テーブル) のロックを追跡します。従って、この数以上の明確なオブジェクトは同時にロックされません。このパラメータはそれぞれのトランザクションに対して割り当てられたオブジェクトのロックの平均数を管理します。個別のトランザクションはロックテーブル内の全てのトランザクションのロックが適合する限り、より多くのオブジェクトをロックできます。これはロック可能な行数ではありません。その値は無制限です。デフォルトは64で、テストでは一般的に充分ですが、単一のシリアライズブルトランザクションで数多くの異なるテーブルに触れるクライアントが存在する場合、この値を大きくする必要があることがあります。このパラメータはサーバ起動時のみ設定可能です。

このパラメータを大きくするとオペレーティングシステムのデフォルトの設定が許容している以上のSystem V共有メモリをPostgreSQLが要求することがあります。これらパラメータをどのように調整するかについて、必要であれば[項17.4.1](#)を参照してください。

18.13. バージョンとプラットフォーム互換性

18.13.1. 以前のPostgreSQLバージョン

`array_nulls (boolean)`

これは、配列入力パーサが引用符のない`NULL`を`NULL`配列要素として認識するかどうかを制御します。デフォルトでは、これは`on`で、`NULL`値を持つ配列値を入力することができます。しかし、8.2より前のバージョンのPostgreSQLでは、配列内の`NULL`値をサポートしておらず、`NULL`を`"NULL"`という値の文字列を持つ通常の配列要素として扱っていました。古い動作を必要とするアプリケーションの後方互換性のため、この変数を`off`にすることができます。

この変数が`off`であっても、`NULL`値を含む配列値を作成することができることに注意してください。

`backslash_quote (enum)`

文字列リテラルの中で引用符が`¥`で表現されるかどうかを管理します。引用符の表現としてSQL準拠の方式では二重化(`''`)ですが、PostgreSQLは歴史的に`¥`も受け付けます。とは言っても、いくつかのクライアント文字集合符号化方式において、最終バイトが数値的にASCIIの`¥`に等しいマルチバイト文字があり、`¥`を使用するとセキュリティ上問題を引き起こす可能性があります。クライアント側のコードが事実上エスケープを正しく扱わない場合、SQLインジェクション攻撃が可能になります。この危険性の回避は、サーバが逆スラッシュでエスケープされた引用符を含む問い合わせを拒絶するようにします。許可される`backslash_quote`の値は、`on` (常に`¥`を許可)、`off` (常に拒否)、および`safe_encoding` (クライアント符号化方式がASCIIの`¥`を許可しないときのみ、マルチバイト文字内で許可)。`safe_encoding`がデフォルトの設定。

標準に従った文字列リテラルでは、`¥`は単に`¥`を意味するものです。このパラメータのみが、エスケープ文字列構文(`E'...'`)を含む標準に従わないリテラルの取り扱いに影響します。

`default_with_oids (boolean)`

もし`WITH OIDS`もしくは`WITHOUT OIDS`のいずれも指定されていない場合、`CREATE TABLE`および`CREATE TABLE AS`がオブジェクト識別子 (OID) 列を新規作成のテーブルに含めるか否かを管理します。同時に`SELECT INTO`で作成されたテーブルにOIDが含まれるか否かも決定します。このパラメータはデフォルトで`off`です。PostgreSQL 8.0およびそれ以前ではデフォルトで`on`でした。

ユーザテーブルでのOID使用は良くないと考えらるため、ほとんどのインストレーションはこの変数を無効にしたままになっています。特定のテーブルに対してOIDを必要とするアプリケーションは、テーブル作成にあたって`WITH OIDS`を指定しなければなりません。この変数は、この動作を行わない古いアプリケーションとの互換性を有効にします。

`escape_string_warning (boolean)`

有効の場合、通常の文字列リテラル (`'...'` 構文) にバックスラッシュ (`¥`) があり、`standard_conforming_strings`が無効な場合、警告が発せられます。デフォルトは`on`です。

通常文字列のデフォルトの振る舞いは、SQL標準ではバックスラッシュを通常文字として取り扱うため、バックスラッシュをエスケープとして使用したいアプリケーションは、エスケープ文字列構文(`E'...'`)を使用するように変更すべきです。この変数は変更すべきコードを突き止めるのに役立つよう、有効にすることができます。

`lo_compat_privileges (boolean)`

9.0以前のPostgreSQLリリースでは、ラージオブジェクトはアクセス権限が無く、そしてその結果、全てのユーザが読み込み、書き込みが可能でした。この変数を`on`にすると、以前のリリースとの互換性のため、新規の権限チェックが無効になります。デフォルトは`off`です。

この変数を設定してもラージオブジェクトに関連した全ての安全性チェックを無効にしません。PostgreSQL 9.0.で変更されたデフォルトの動きに対してのみです。例えば、`lo_import()` と `lo_export()` はこの設定とは別にスーパーユーザの権限を必要とします。

`quote_all_identifiers (boolean)`

データベースがSQLを生成する時、たとえ (現在) キーワードになっていなくても、全ての識別子を引用符で囲むことを強制します。これは `EXPLAIN` の出力に影響を与えるのみならず、`pg_get_viewdef` のような関数の結果にも影響します。[pg_dump](#) および [pg_dumpall](#) の `--quote-all-identifiers` オプションも参照してください。

`sql_inheritance` (boolean)

この設定は、修飾されていないテーブル参照が継承された子テーブルを含むものとみなすかどうかを制御します。デフォルトは`on`であり、子テーブルが含まれることを意味します（つまりデフォルトでは`*`が仮定されます）。`off`に変更すると、子テーブルは含まれません（つまり`ONLY`接頭辞が仮定されます）。標準SQLでは、子テーブルが含まれることを要求します。このため`off`という設定は標準互換ではありません。しかしPostgreSQL 7.1以前のリリースとの互換性のため提供されています。[項5.8](#)を参照してください。

`sql_inheritance`を無効にすることは廃止予定です。この動作は標準SQLと比べてエラーを招きやすいことが分かっているためです。通常本書で説明する継承に関する動作説明では、`on`であることを前提としています。

`standard_conforming_strings` (boolean)

標準SQLで規定されたように、通常の文字列リテラル (`'...'`) がバックスラッシュをそのまま取り扱うか否かを制御します。PostgreSQL 9.1の初期においてデフォルトは`on`です（それ以前のリリースではデフォルトとして`off`でした）。どのように文字列リテラルが処理されるかを定めるこのパラメータを、アプリケーションで検査することができます。このパラメータの存在は、エスケープ文字列構文 (`E'...'`) がサポートされているかどうかを示すものとも考えられます。エスケープ文字列構文 ([項4.1.2.2](#))は、アプリケーションでバックスラッシュをエスケープ文字として扱いたい場合に使用すべきです。

`synchronize_seqscans` (boolean)

これにより、同時実行スキャンがほぼ同じ時間に同じブロックを読み取り、I/Oへの負荷を分散できるように、互いに同期して、大規模テーブルをシーケンシャルスキャンすることができます。これが有効な場合、スキャンはテーブルの途中から始まり、進行中のスキャンの活動と同期するように、行全体を覆うように終端を"巻き上げる"可能性があります。これにより、`ORDER BY`句を持たない問い合わせが返す行の順序は予想できない程変わってしまいます。このパラメータを`off`にすることで、シーケンシャルスキャンが常にテーブルの先頭から始まるという、8.3より前の動作を保証します。デフォルトは`on`です。

18.13.2. プラットホームとクライアント互換性

`transform_null_equals` (boolean)

有効の場合、`expr = NULL`（もしくは`NULL = expr`）形式の式は`expr IS NULL`として取り扱われ、それは、もし`expr`がNULL値と評価すれば真を返し、そうでなければ偽を返します。`expr = NULL`の正しいSQL仕様準拠の動作は常にNULL（判らない）を返すことです。従って、このパラメータのデフォルトは`off`になっています。

しかし、Microsoft Accessのフィルタ形式はNULL値を検査するために`expr = NULL`を使用する問い合わせを生成しますので、そのインタフェースを使用してデータベースにアクセスする場合は、このオプションを有効にする方が良いでしょう。`expr = NULL`という形の式は（SQL標準解釈を使用した結果）常にNULL値を返しますので、通常のアプリケーションでは意味がほとんどなく、減多に使用されません。ですので、このオプションは実際は害はありません。しかし、慣れていないユーザはしばしばNULL値に関する式の意味に戸惑いますので、デフォルトでこのオプションは`off`です。

このオプションは`= NULL`という形式にのみ影響することに注意してください。他の比較演算子や等価演算子を呼び出す他の（`IN`のような）式と計算する上で等価となる式には影響を与えません。したがって、このオプションは間違ったプログラミングの汎用的な問題解決を行いません。

関連する情報は[項9.2](#)を参照してください。

[前のページ](#)
ロック管理

[ホーム](#)
[上に戻る](#)

[次のページ](#)
エラー処理

18.14. エラー処理

`exit_on_error` (boolean)

真の場合、全てのエラーは現在のセッションを中止させます。 デフォルトでこれは偽に設定されますので、 FATALエラーのみがセッションを中止させます。

`restart_after_crash` (boolean)

真の場合、これがデフォルトですが、PostgreSQLはバックエンドのクラッシュの後、自動的に再初期化を行います。 この値を真のままにしておくことは通常データベースの可用性を最大化する最適の方法です。 しかし、 PostgreSQLがクラスタウェアにより起動された時のような状況により、クラスタウェアが制御を獲得し、適切とみなされるいかなる振る舞いをも行えるように再起動を無効にすることも有益です。

18.15. 設定済みのオプション

以下の"パラメータ"は読み取り専用で、PostgreSQLのコンパイル時、もしくはインストール時に決定されます。そのため、これらは`postgresql.conf`のサンプルから除かれています。このオプションは、特定のアプリケーション、特に管理用フロントエンドによって注目される可能性があるPostgreSQLの様々な部分の振舞いを報告します。

`block_size (integer)`

ディスクブロックの容量を報告します。サーバ構築の際に`BLCKSZ`の値で決定されます。デフォルトの値は8192バイトです。([shared buffers](#) の様な) いくつかの構成変数の意味は`block_size`によって影響されます。これに関する情報は [項18.4](#) を参照してください。

`integer_datetimes (boolean)`

PostgreSQLが64ビット整数による日付と時刻のサポート付きで構築されたかどうかを報告します。PostgreSQLを構築するときに、`--disable-integer-datetimes`の構成で設定することで無効になります。デフォルトの値は`on`です。

`lc_collate (string)`

テキストどおりのデータが並び替えられたロケールを報告します。詳細は [項22.1](#) を参照してください。この値はデータベースが作成されたときに決定されます。

`lc_ctype (string)`

文字分類を決定するロケールを報告します。詳細は [項22.1](#) を参照してください。この値はデータベースが作成されたときに決定されます。通常、これは`lc_collate`と同一ですが、特殊なアプリケーションでは異なって設定されることがあります。

`max_function_args (integer)`

関数の引数の最大数を報告します。サーバを構築する時、`FUNC_MAX_ARGS`の値で決定されます。デフォルトの値は100引数です。

`max_identifier_length (integer)`

最長の識別子の長さを報告します。サーバ構築時の`NAMEDATALEN`の値より一つ少なく設定されます。デフォルトの`NAMEDATALEN`の値は64ですので、デフォルトの`max_identifier_length`は63バイトで、マルチバイト符号化方式を使用している場合、63文字以下になることがあります。

`max_index_keys (integer)`

インデックスキーの最大値を報告します。サーバ構築の際に`INDEX_MAX_KEYS`の値で決定されます。デフォルトの値は32キーです。

`segment_size (integer)`

あるファイルセグメントの中に格納できるブロック数(ページ数)を報告します。サーバ構築時に`RELSEG_SIZE`の値で決定されます。バイト単位の一セグメントファイルの最大容量は、`block_size` 倍の `segment_size` と等しくなります。デフォルトでは1GBです。

`server_encoding (string)`

データベース符号化方式(文字セット)を報告します。データベースが作成された時に決定されます。通常クライアントは [client_encoding](#) の値にのみ注意する必要があります。

`server_version (string)`

サーバのバージョン番号を報告します。サーバ構築の際の`PG_VERSION`の値によって決定されます。

`server_version_num (integer)`

サーバのバージョン番号を整数として返します。この値は、サーバ構築時の`PG_VERSION_NUM`の値により決まります。

`wal_block_size (integer)`

WALディスクブロックの容量を報告します。 サーバ構築時にXLOG_BLCKSZの値で決定されます。デフォルトの値は8192バイトです。

wal_segment_size (integer)

WALセグメントファイルのブロック数（ページ数）を報告します。 バイト単位のWALセグメント総容量はwal_block_size倍のwal_segment_sizeに等しくなります。 デフォルトでは16MBです。より詳細は[項29.4](#)を参照してください。

[前のページ](#)
エラー処理

[ホーム](#)
[上に戻る](#)

[次のページ](#)
独自のオプション

18.16. 独自のオプション

この機能、追加モジュール（手続き言語など）によって追加される PostgreSQL が識別できないパラメータを使えるように設計されたものです。これにより拡張モジュールは標準の方法で構成されます。

カスタムオプションには 2 つに分かれた名称があります。拡張名につづいてドット、そして特定のパラメータ名です。SQL の修飾名に良く似ています。例として `plpgsql.variable_conflict` が挙げられます。

カスタムオプションは読み込まれていない関連性のある拡張モジュールのプロセスに設定される必要がある場合があります。で、PostgreSQL はどんな 2 つの部分のパラメータ名による設定を受け付けます。これらの変数は代替物として取り扱われ、それらを定義したモジュールが読み込まれるまで機能しません。拡張モジュールが読み込まれた時、その変数定義が追加され、それら定義に基づいた代替値が変換され、そしてその拡張名の確認されない代替物に対して警告が発せられます。

18.17. 開発者向けのオプション

以下のパラメータは、PostgreSQLのソースコードに対する作業用のものです。中には深刻な損傷を負ったデータベースの復旧に役立つものもあります。実運用のデータベースでこれらを設定する理由はないはずです。したがって、これらはサンプルの `postgresql.conf` からは除外されています。これらのパラメータの多くは、それを動作させるために特殊なソースコンパイルを必要としていることに注意してください。

`allow_system_table_mods` (boolean)

システムテーブルの構造変更を許可します。これは `initdb` で使用されます。このパラメータはサーバ起動時にのみ設定可能です。

`debug_assertions` (boolean)

各種のアサーション検査を有効にします。これはデバッグ用の道具です。もしおかしな問題とかクラッシュを経験すれば、プログラミングの間違いが顕在化するので、これを有効にする必要があるかもしれません。このパラメータを使用するには、マクロ `USE_ASSERT_CHECKING` が、PostgreSQLの構築時に (`configure` オプションの `--enable-cassert` で) 定義されなければなりません。もし、PostgreSQLがアサーション付で構築されていれば、`debug_assertions` のデフォルトは `on` であることに注意してください。

`ignore_system_indexes` (boolean)

システムテーブルの読み込み時にシステムインデックスを無視します（しかしテーブルが更新された時はインデックスを更新します）。障害があるシステムインデックスを復旧する時、これは有用です。セッションが始まった後に、このパラメータを変更することはできません。

`post_auth_delay` (integer)

非ゼロの場合、サーバプロセスが始まり認証手続きが終わった後に、指定した秒数の遅延が発生します。これは、デバッグを使用してサーバプロセスに接続する機会を開発者に提供することを目的としています。セッションが始まった後に、このパラメータを変更することはできません。

`pre_auth_delay` (integer)

非ゼロの場合、ここで指定した秒数分の遅延が新しくサーバプロセスがforkした後、認証手続きに入る前に発生します。これは、認証における誤動作を追跡するために、デバッグを使用してサーバプロセスに接続する機会を開発者に提供することを目的としたものです。このパラメータは `postgresql.conf` ファイル内、または、サーバのコマンドラインでのみ設定可能です。

`trace_notify` (boolean)

`LISTEN` と `NOTIFY` コマンドのための大量なデバッグ出力を生成します。この出力をクライアントもしくはサーバログに送信するためには、それぞれ、[client_min_messages](#) もしくは [log_min_messages](#) は `DEBUG1` 以下でなければなりません。

`trace_recovery_messages` (enum)

復旧関連のデバッグ出力のログ取得を有効にします。さもないとログは取られません。このパラメータはユーザに対し、[log_min_messages](#) の通常設定を上書きすることを許可します。しかし、特定のメッセージに対してのみです。これはホットスタンバイのデバッグを意図したものです。有効な値は、`DEBUG5`、`DEBUG4`、`DEBUG3`、`DEBUG2`、`DEBUG1`、および `LOG` です。デフォルトの `LOG` は、ログ取得の決定に全く影響しません。その他の値は、あたかも `LOG` 優先度を所有しているごとく、それ、またはより高い優先度でログ取得される復旧関連デバッグメッセージの要因となります。

`log_min_messages` の通常設定に対し、これは無条件にそれらをサーバログに送り込みます。このパラメータは `postgresql.conf` ファイル内、または、サーバコマンドラインでのみ設定可能です。

`trace_sort` (boolean)

もしも有効であれば、並び替え操作の間のリソース使用についての情報を放出します。このパラメータは PostgreSQL がコンパイルされた時、`TRACE_SORT` マクロが定義されている場合にのみ有効です。（とは言っても、現在 `TRACE_SORT` はデフォルトで定義されています。）

`trace_locks` (boolean)

有効な場合、ロックの使用状況に関する情報を出力します。出力される情報には、ロック操作の種類、ロックの種類、ロックまたはロック解除されているオブジェクトの一意な識別子が含まれます。また、このオブジェクトに既に与えられ

ているロック種類やこのオブジェクトで待機しているロック種類を表すビットマスクも含まれます。ロック種類それぞれについて、与えられているロック数、待機中のロック数がその総数と共に出力されます。ログファイル出力例を以下に示します。

```
LOG:  LockAcquire: new: lock(0xb7acd844) id(24688,24696,0,0,0,1)
      grantMask(0) req(0,0,0,0,0,0)=0 grant(0,0,0,0,0,0)=0
      wait(0) type(AccessShareLock)
LOG:  GrantLock: lock(0xb7acd844) id(24688,24696,0,0,0,1)
      grantMask(2) req(1,0,0,0,0,0)=1 grant(1,0,0,0,0,0)=1
      wait(0) type(AccessShareLock)
LOG:  UnGrantLock: updated: lock(0xb7acd844) id(24688,24696,0,0,0,1)
      grantMask(0) req(0,0,0,0,0,0)=0 grant(0,0,0,0,0,0)=0
      wait(0) type(AccessShareLock)
LOG:  CleanUpLock: deleting: lock(0xb7acd844) id(24688,24696,0,0,0,1)
      grantMask(0) req(0,0,0,0,0,0)=0 grant(0,0,0,0,0,0)=0
      wait(0) type(INVALID)
```

ダンプされる構造の詳細は、src/include/storage/lock.h にあります。

このパラメータはPostgreSQLがコンパイル時にLOCK_DEBUG マクロが定義された場合のみ有効です。

trace_lwlocks (boolean)

有効な場合、軽量ロックの使用状況に関する情報を出力します。軽量ロックは主に、共有メモリ上のデータ構造へのアクセスに関する排他制御機能を提供することを意図したものです。

このパラメータはPostgreSQLがコンパイル時にLOCK_DEBUG マクロが定義された場合のみ有効です。

trace_userlocks (boolean)

有効な場合、ユーザロックの使用状況に関する情報を出力します。出力はtrace_locksと同じですが、助言ロックに関するもののみを出力します。

このパラメータはPostgreSQLがコンパイル時にLOCK_DEBUG マクロが定義された場合のみ有効です。

trace_lock_oidmin (integer)

設定すると、このOID未満のテーブルに関するロックの追跡を行いません。（システムテーブルに関する出力を抑えるために使用します。）

このパラメータはPostgreSQLがコンパイル時にLOCK_DEBUG マクロが定義された場合のみ有効です。

trace_lock_table (integer)

このテーブル（OID）に対し無条件でロックを追跡します。

このパラメータはPostgreSQLがコンパイル時にLOCK_DEBUG マクロが定義された場合のみ有効です。

debug_deadlocks (boolean)

設定すると、デッドロックタイムアウトが発生した時全ての進行中のロックについての情報がダンプされます。

このパラメータはPostgreSQLがコンパイル時にLOCK_DEBUG マクロが定義された場合のみ有効です。

log_btree_build_stats (boolean)

設定すると、各種B-Tree操作に関するシステムリソース（メモリとCPU）の使用についての統計情報をログに出力します。

このパラメータはPostgreSQLがコンパイル時にBTREE_BUILD_STATS マクロが定義された場合のみ有効です。

wal_debug (boolean)

もしonであれば、WALに関連したデバッグ出力が有効になります。このパラメータはWAL_DEBUG マクロが PostgreSQLのコンパイルの時に定義された場合にのみ有効です。

zero_damaged_pages (boolean)

ページヘッダの障害がわかると、通常PostgreSQLはエラーの報告を行い、現在のトランザクションを中断させます。

zero_damaged_pagesをonに設定することにより、システムは代わりに警告を報告し、障害のあるメモリ内のページをゼロで埋め、処理を継続します。この動作により、障害のあったページ上にある全ての行のデータが破壊されます。しかし、これによりエラーを確実に無視し、正常なページに存在するテーブル内の行を取り出すことができます。ハードウェア

アまたはソフトウェアのエラーによって破損が発生した場合のデータの復旧時に有用です。 障害のあるページからのテーブルのデータの復旧をあきらめた場合を除き、通常はこれをonにしてはいけません。 ゼロで埋められたページはディスクに書き込みを強要されないため、このパラメータを再び無効にする以前にテーブル、またはインデックスを再作成することを勧めます。 デフォルトはoffであり、スーパーユーザのみ変更可能です。

[前のページ](#)
独自のオプション

[ホーム](#)
[上に戻る](#)

[次のページ](#)
短いオプション

18.18. 短いオプション

簡便性のために、一文字のコマンドラインオプションスイッチも、幾つかのパラメータのために用意されています。それらは[表18-2](#)に解説されています。一部のオプションは歴史的な理由のために存在します。また、この一文字オプションが存在することが、このオプションを多く使用することを支持することを示しているわけではありません。

表 18-2. 短いオプションキー

短いオプション	同義
-A X	debug_assertions = X
-B X	shared_buffers = X
-d X	log_min_messages = DEBUGX
-e	datestyle = euro
-fb, -fh, -fi, -fm, -fn, -fo, -fs, -ft	enable_bitmapscan = off, enable_hashjoin = off, enable_indexscan = off, enable_mergejoin = off, enable_nestloop = off, enable_indexonlyscan = off, enable_seqscan = off, enable_tidscan = off
-F	fsync = off
-h X	listen_addresses = X
-i	listen_addresses = '*'
-k X	unix_socket_directory = X
-l	ssl = on
-N X	max_connections = X
-O	allow_system_table_mods = on
-p X	port = X
-P	ignore_system_indexes = on
-s	log_statement_stats = on
-S X	work_mem = X
-tpa, -tpl, -te	log_parser_stats = on, log_planner_stats = on, log_executor_stats = on
-W X	post_auth_delay = X

第 19章クライアント認証

目次

19.1. [pg_hba.confファイル](#)

19.2. [ユーザ名マップ](#)

19.3. [認証方式](#)

19.3.1. [Trust認証](#)

19.3.2. [パスワード認証](#)

19.3.3. [GSSAPI認証](#)

19.3.4. [SSPI認証](#)

19.3.5. [Kerberos認証](#)

19.3.6. [Ident認証](#)

19.3.7. [Peer認証](#)

19.3.8. [LDAP認証](#)

19.3.9. [RADIUS Authentication](#)

19.3.10. [証明書認証](#)

19.3.11. [PAM認証](#)

19.4. [認証における問題点](#)

クライアントアプリケーションがデータベースサーバに接続する時、Unixコンピュータに特定のユーザとしてログインする時と同じように、どのPostgreSQLユーザ名で接続するかを指定します。SQL 環境の中では存在するユーザ名でデータベースオブジェクトへのアクセス権限が決まります。詳しい情報は[第20章](#)を参照してください。ですから、どのデータベースユーザがデータベースに接続できるかを制限することが基本となります。

注意: [第20章](#)で説明されていますが、実はPostgreSQLは"roles"によって権限の管理を行っています。この章では一貫してデータベースユーザという呼び方を"ログイン権限を持つロール"として使用します。

認証はデータベースサーバがクライアントの身元を識別し、その延長としてクライアントアプリケーション（もしくはクライアントアプリケーションを実行するユーザ）が要求されたデータベースユーザ名で接続することができるかどうかを決定する手順です。

PostgreSQLには異なったクライアント認証方法を複数提供します。特定のクライアント接続の認証に使用する方法は、（クライアントの）ホストアドレス、データベース、およびユーザに従って選択できます。

PostgreSQLデータベースユーザ名は稼働しているサーバのオペレーティングシステムのユーザ名とは論理的に分かれています。もし特定のサーバの全てのユーザがサーバマシン上にもアカウントを持っている場合、そのオペレーティングシステムのユーザ名に一致するデータベースユーザ名を割り当てることは理にかなっています。しかし、リモート接続を受け付けるサーバは、ローカルなオペレーティングシステムのアカウントを持たないデータベースユーザを多く持っている場合もあります。そのような時にはデータベースユーザ名とOSのユーザ名との間の関連性は必要ありません。

[前のページ](#)

短いオプション

[ホーム](#)

[上に戻る](#)

[次のページ](#)

pg_hba.conf ファイル

19.1. pg_hba.conf ファイル

クライアント認証はデータベースクラスタのデータディレクトリ内の、伝統的にpg_hba.confという名前の設定ファイルで管理されています（HBAとは、host-based authentication: ホストベース認証の略です）。デフォルトのpg_hba.confファイルは、データディレクトリがinitdbで初期化される時にインストールされます。しかし、この認証設定ファイルを他の場所に設置することができます。[hba_file](#)設定パラメータを参照してください。

pg_hba.confファイルの一般的な書式は、1行につき1つのレコードというレコードの集合です。空行はコメント用の#文字以降の文字と同じく無視されます。レコードは行をまたいで続けることはできません。レコードはスペースもしくはタブ、もしくはその両方で区切られた、複数のフィールドで構成されています。フィールドには、フィールド値が引用符付きの場合空白文字を含むことができます。データベース、ユーザもしくはアドレス(例: all or replication)フィールド内のキーワードの一つを引用するとその特定の文字が失われ、データベース名、ユーザ名もしくはホスト名が名称と一致するようになります。

それぞれのレコードは接続形式、（接続形式に対して意味を持つのであれば）クライアントのIPアドレス範囲、データベースの名前、ユーザ名およびこれらのパラメータに一致する接続で使用する認証方法を指定します。接続形式、クライアントアドレス、要求されたデータベース、およびユーザ名に一致する最初のレコードが認証処理に使用されます。"失敗時の継続"や、あるいは"バックアップ"はありません。これは、もしあるレコードが選択されて認証に失敗した場合、後続のレコードは考慮されないということです。どのレコードも一致しない時はアクセスが拒否されます。

レコードは7つの書式のうちの1つの形式を取ります。

```
local      database user auth-method [auth-options]
host       database user address auth-method [auth-options]
hostssl    database user address auth-method [auth-options]
hostnossl  database user address auth-method [auth-options]
host       database user IP-address IP-mask auth-method [auth-options]
hostssl    database user IP-address IP-mask auth-method [auth-options]
hostnossl  database user IP-address IP-mask auth-method [auth-options]
```

フィールドの意味は以下のようにになっています。

local

このレコードはUnixドメインソケットを使用する接続に対応します。この種類のレコードを使用しないと、Unixドメインソケット経由の接続は拒否されます。

host

このレコードは、TCP/IPを使用した接続に対応します。hostレコードは、SSLもしくは非SSL接続のいずれかに対応します。

注意: サーバのデフォルトの動作は、ローカルループバックアドレスであるlocalhostのみTCP/IP接続を監視しています。よってサーバにおいてlisten_addressesパラメータが適切な値に設定された状態で起動されていない限り、リモートのTCP/IP接続はできません。

hostssl

このレコードは、接続がSSLで暗号化されている場合にのみTCP/IPネットワークを使用する接続に対応します。

このオプションを使用するためには、サーバはSSLサポートができるように構築されていなければいけません。また、SSLはsslパラメータを設定することによりサーバの起動時に有効になっていなくてはなりません（詳細は[項17.9](#)を参照してください）。

hostnossl

このレコードは、hostsslと反対の動作で、SSLを使用していないTCP/IPの接続のみに対応します。

database

このレコードで対応するデータベース名を指定します。allという値は、全てのデータベースと対応することを指定します。sameuserという値は、要求されたデータベースが要求ユーザと同じ名前を持つ場合にレコードが対応することを指定します。sameroleという値は、要求ユーザが要求されたデータベースと同じ名前のロールのメンバでなければならないことを指定します。（以前はsamegroupと書いていましたが、sameroleと記述してください）スーパーユーザは、もし明示的または直接的もしくは間接的にsameroleのメンバではない限りsameroleのメンバとはみなされません。また、スーパーユーザであるからといってsameroleのメンバとはみなされません。replicationの値は、もしレプリケーション接続が要求された場合（レプリケーション接続は特定のデータベースを指定しないことに注意して下さい）にレコードが一致することを指定します。それ以外の場合には、特定のPostgreSQLデータベースの名前になります。データベースの名前はカンマで区切ることで複数指定できます。データベース名を含む別のファイルを、そのファイル名の前に@を付けることで

指定できます。

`user`

このレコードで対応するデータベースユーザを指定します。 `all` という値は、全てのユーザが対応することを指定します。 それ以外の場合には特定のデータベースユーザの名前もしくは+で始まるグループ名のどちらかになります。

(PostgreSQLではユーザとグループの明確な区別がないことを思い出してください。 +のマークは、"このロールの直接的もしくは間接的なメンバのどちらかに一致していること"を意味しています。 一方、+のマークのない名前は特定のロールにのみ一致します) このため、スーパーユーザは、もし明示的または直接的もしくは間接的にロールのメンバである場合にのみ、ロールのメンバとみなされます。 スーパーユーザであるからといってロールのメンバとはみなされません。 重複したユーザ名は、ユーザ名をカンマで区切ることによって使用できるようになります。 ユーザ名を含む別のファイルを、そのファイル名の前に@を付けることで指定できます。

`address`

このレコードに対応しているクライアントマシンのアドレス。 このフィールドはホスト名、IPアドレスの範囲、もしくは下記の特別なキーワードの1つを含んでいます。

このIPアドレスは、CIDRマスクの長さを含んだ標準のドット区切り10進表記で指定されます。 CIDRマスク長とは、クライアントIPアドレスが一致しなければならない、高位のビット数を表すものです。 指定するIPアドレスのこれより右側のビットには、0を指定しなければなりません。 IPアドレスと/、およびCIDRマスク長の間には空白を入れてはいけません。

典型的なIPアドレス範囲の例は、単一のホストでは172.20.143.89/32、小規模ネットワークでは172.20.143.0/24、大規模ネットワークでは10.6.0.0/16のようなものです。 0.0.0.0/0は全てのIPv4アドレスを意味します。 また、::/0は全てのIPv6アドレスを意味しています。 単一ホストを指定するには、IPv4では32というCIDRマスクを、IPv6では128を使用してください。 ネットワークアドレスでは末尾の0を省略できません。

IPv4書式で与えられたIPアドレスは、対応するアドレスを持つIPv6接続に対しても対応します。 例えば、127.0.0.1は::ffff:127.0.0.1 IPv6アドレスに対応します。 IPv6書式で与えられた項目は、たとえそのアドレスがIPv6内のIPv4の範囲内であったとしてもIPv6接続のみに対応します。 IPv6書式の項目は、システムのCライブラリがIPv6アドレスをサポートしていない場合拒絶されることに注意してください。

どのIPアドレスにも一致するように`all`と書くこともできますし、サーバ自身のどのIPアドレスにも一致するように`samehost`と書くこともできます。 もしくは、サーバが直セル接続されているサブネット内のどのアドレスにも一致するように`samenet`と書くことができます。

もし、ホスト名 (IPアドレスではない場合の全て、もしくは潜在的なホスト名として処理される特別なキーワード) が指定されている場合は、その名前は、クライアントのIPアドレスの逆引き名前解決の結果と比較されます (例えば、もしDNSが使用されている場合は逆引きDNS検索により解決されます)。 ホスト名の比較は、大文字小文字が区別されます。 もし一致するものがあった場合は、解決された、どのアドレスもクライアントのIPアドレスと等しいか否かをチェックするために (例えば、前方一致のDNS検索のような) ホスト名の前方一致の名前解決が実行されます。 もし前方、後方の両方向で一致した場合は、エントリは一致するものとみなされます。 (`pg_hba.conf`内で使用されているホスト名は、クライアントのIPアドレスのアドレス-名前解決が返すホスト名の1つでなければいけません。 もしそうでなければこの行は一致しません。 1つのIPアドレスを複数のホスト名に関連付けるホスト名データベースもありますが、IPアドレスの解決を要求された場合にオペレーティングシステムは1つのホスト名のみを返します。)

ドット (.) で始まるホスト名の特定は実際のホスト名のサフィックスに一致します。 よって、`.example.com`は、`foo.example.com`に一致します (`example.com`だけでは一致しません)。

ホスト名が`pg_hba.conf`内で指定されている場合、名前解決が適度に早いことを確かめてください。 `nscd`のようなローカル名前解決のキャッシュを設定することによる恩恵です。 また、クライアントのIPアドレスの代わりにホスト名がログで見られるように、`log_hostname`の設定パラメータを有効化することもできます。

時折、ユーザは、なぜホスト名が2つの名前解決やIPアドレスの逆引きが必要になる、というような一見複雑に見える方法で扱われるのか不思議に思うことがあり、その結果、幾つかの望ましくないホスト名を設定したり指定することをしません。 第一の理由として、効率です。 接続要求は今のクライアントのアドレスの2つのリゾルバの検索が必要になります。 もしリゾルバにおいて、そのアドレスに問題があった場合、クライアントのみの問題となります。 前方検索のみを行うような実装を仮に行っていると、全ての接続要求において`pg_hba.conf`内に記載された全てのホスト名を解決しなくてはいけなくなります。 これは、それ自身で処理が遅くなります。 また、リゾルバにおいて1つのホスト名に問題があった場合、全員の問題となってしまいます。

さらに、逆引き検索はサフィックス一致の機能を実装するために必要です。 というのも実際のクライアントのホスト名はホスト名がパターンに対して一致するために、知られる必要があるためです。

このふるまいは、Apache HTTPサーバやTCPラッパーのような他のよくあるホスト名ベースのアクセス制御の実装と

一致していることに注意して下さい。

これらのフィールドは`host`および`hostssl`、`hostnossl`レコードにのみ適用されます。

`IP-address`
`IP-mask`

このフィールドは`CIDR-address`表記の代替として使用可能です。 マスク長を指定する代わりに、実際のマスクを分離した列で指定します。例えば`255.0.0.0`はIPv4のCIDRマスク長8を意味し、`255.255.255.255`はCIDRマスク長32を意味しています。

このフィールドは`host`および`hostssl`、`hostnossl`レコードにのみ適用されます。

`auth-method`

接続がこのレコードに一致する場合に使用する認証方式を指定します。 使用できる選択肢は以下にまとめていますが、詳しくは[項19.3](#)を参照してください。

`trust`

接続を無条件で許可します。この方式は、PostgreSQLデータベースサーバに接続できる全てのユーザが、任意のPostgreSQLユーザとしてパスワードや他の認証なしでログインすることを許可します。詳細は[項19.3.1](#)を参照してください。

`reject`

接続を無条件に拒否します。特定のホストをあるグループから"除外"するために便利です。例えば、1行の`reject`は特定のホストが接続することを拒否します。一方、後ろの行では特定のネットワーク内の残りのホストが接続することを許可します。

`md5`

クライアントに対して認証時にMD5暗号化パスワードを要求します。詳細は[項19.3.2](#)を参照してください。

`password`

クライアントに対して認証時に平文のパスワードを要求します。パスワードはネットワークを通じて普通のテキスト形式で送られますので、信頼されていないネットワークでは使用しないでください。詳細は[項19.3.2](#)を参照してください。

`gss`

ユーザの認証にGSSAPIを使用します。これはTCP/IP接続を使用するときのみ使用可能です。詳細は[項19.3.3](#)を参照してください。

`sspi`

ユーザの認証にSSPIを使用します。これはWindowsを使用するときのみ使用可能です。詳細は[項19.3.4](#)を参照してください。

`krb5`

ユーザ認証にKerberos V5を使用します。TCP/IP接続の時のみ有効です。詳細は[項19.3.5](#)を参照してください。

`ident`

クライアントのオペレーティングシステムにおけるユーザ名をクライアント上のidentサーバに尋ねてユーザ名が要求されたデータベースユーザ名と一致するか検査します。Ident認証は、TCP/IP接続でのみ使用可能です。ローカル接続が指定されている場合は、peer認証が代わりに使用されます。詳細は[項19.3.6](#)を参照してください。

`peer`

クライアントのオペレーティングシステムにおけるユーザ名をオペレーティングシステムから取得し、ユーザ名が要求されたデータベースユーザ名と一致するか検査します。これはローカル接続の時のみ使用可能です。詳細は[項19.3.7](#)を参照してください。

`ldap`

LDAPサーバを使用して認証します。詳細は[項19.3.8](#)を参照してください。

`radius`

RADIUSサーバを使用して認証します。詳細は[項19.3.9](#)を参照してください。

cert

SSLクライアント証明を使用して認証します。 詳細は[項19.3.10](#)を参照してください。

pam

オペレーティングシステムによって提供されるPAM（Pluggable Authentication Modules）サービスを使用した認証です。 詳細は[項19.3.11](#)を参照してください。

auth-options

`auth-method`フィールドの後ろに、認証方式のオプションを指定する、`name=value`の形式のフィールドが存在する可能性があります。 どのオプションがどの認証方式に使用できるのか、についての詳細は以下で説明します。

@式により含められるファイルは、空白文字あるいはカンマのどちらかで区切られた名前の列挙として読み込まれます。 コメントは、`pg_hba.conf`と同様に#から始まります。 また、@式を入れ子にすることもできます。 @の後のファイル名が絶対パスでない限り、参照元ファイルが存在するディレクトリから見た相対パスであるとみなされます。

`pg_hba.conf`レコードは接続が試みられる度に順番に検査されますので、レコードの順序はとても大切です。 典型的には、始めの方のレコードには厳しい接続照合パラメータと緩い認証方式があるのに対し、終わりの方のレコードにはより緩い照合パラメータとより厳しい認証方式があります。 例えば、ローカルTCP接続では`trust`認証方式、リモートTCP接続に対してはパスワードを要求したいとします。 この場合、広範囲にわたって許可されるクライアントのIPアドレスに対するパスワード認証を指定するレコードの前に127.0.0.1からの接続に対する`trust`認証指定のレコードが置かれなければなりません。

`pg_hba.conf`ファイルは起動時と、主サーバプロセスがSIGHUP信号を受け取った時に読み込まれます。 稼働中のシステムでファイルを編集した場合は、（`pg_ctl reload`あるいは`kill -HUP`を使用して）postmasterにファイルをもう一度読み込むように信号を出さなければなりません。

ティップ: 特定のデータベースに接続するためには、ユーザは`pg_hba.conf`による検査を通過しなければならない他、そのデータベースに対する`CONNECT`権限を持たなければなりません。 どのユーザがどのデータベースに接続できるかを制限したければ、通常、`pg_hba.conf`項目に規則を追加するよりも、`CONNECT`権限の付与・削除を行う方が簡単です。

`pg_hba.conf`ファイルの例をいくつか[例19-1](#)に示します。 各種認証方式の詳細についてはその後で説明します。

例 19-1. `pg_hba.conf`の項目の例

```
# ローカルシステム上の全てのユーザが、任意のデータベースに
# 任意のデータベースユーザ名でUnixドメインソケットを使用して接続することを許可
# （ローカル接続ではデフォルト）。
#
# TYPE      DATABASE      USER      ADDRESS      METHOD
local      all            all
trust

# ローカルループバックのTCP/IP接続であることを除いて上記に同じ。
#
# TYPE      DATABASE      USER      ADDRESS      METHOD
host       all            all        127.0.0.1/32  trust

# IPv6でも上記に同じ
#
# TYPE      DATABASE      USER      ADDRESS      METHOD
host       all            all        ::1/128      trust

# ホスト名使用すること以外は上記に同じ（一般的にIPv4とIPv6の両方をカバーします）
#
# TYPE      DATABASE      USER      ADDRESS      METHOD
host       all            all        localhost    trust

# ネットマスク列を分けて記載していることを除いて上記に同じ
#
# TYPE      DATABASE      USER      IP-ADDRESS   IP-MASK      METHOD
host       all            all        127.0.0.1    255.255.255.255  trust

# IPアドレス192.168.93.xを持つ全てのホストの全てのユーザが、
# identがその接続について報告するのと同じユーザ名（典型的にはオペレーティングシステムのユーザ名）で
# データベース「postgres」へ接続することを許可。
#
# TYPE      DATABASE      USER      ADDRESS      METHOD
host       postgres    all        192.168.93.0/24  ident

# ユーザのパスワードが正しく入力された場合、
# ホスト192.168.12.10からのどのようなユーザでもデータベース「postgres」へ接続することを許可。
#
# TYPE      DATABASE      USER      ADDRESS      METHOD
host       postgres    all        192.168.12.10/32  md5
```

```

# ユーザのパスワードが正しく指定された場合は、
# example.comドメイン内のホストからの、どのユーザからのデータベース接続も許可する。
#
# TYPE      DATABASE      USER      ADDRESS      METHOD
host       all              all       .example.com  md5

# 先行する「host」行がなければ、これら2行によって192.168.54.1からの接続
# の試みを全て拒否（この項目が最初に照合されるため）。
# ただし、インターネット上の他の全ての場所からのKerberos 5接続は許可。
# ゼロマスクは、ホストIPアドレスのビットが考慮されずに
# どのホストでも照合できることになる。
#
# TYPE      DATABASE      USER      ADDRESS      METHOD
host       all              all       192.168.54.1/32  reject
host       all              all       0.0.0.0/0      krb5

# 192.168.x.xホストからのユーザが、ident 検査に通る場合、
# どのデータベースにでも接続を許可。もし、例えば、identが「bryanh」と認定し
# 「bryanh」がPostgreSQLのユーザ「quest1」として
# 接続要求を出す場合、「bryanh」は「quest1」として接続が許可されるという
# マップ「omicron」に対する記載事項がpg_ident.confにあれば接続を許可。
#
# TYPE      DATABASE      USER      ADDRESS      METHOD
host       all              all       192.168.0.0/16  ident map=omicron

# ローカル接続に対して、以下のたった3行しか記載がない場合、ローカルユーザは
# 自分のデータベース（データベースユーザ名と同じ名前のデータベース）にのみ接続許可。
# ただし管理者とロール「support」のメンバは全てのデータベースに接続可能。
# $PGDATA/adminsファイルは管理者のリストを含む。
# 全ての場合にパスワードが必要。
#
# TYPE      DATABASE      USER      ADDRESS      METHOD
local      sameuser    all              md5
local      all          @admins      md5
local      all          +support     md5

# 上記の最後の2行は1つの行にまとめることが可能。
local      all          @admins,+support md5

# データベースの列にはリストやファイル名も使用できる。
local      db1,db2,@demodbs all          md5

```

19.2. ユーザ名マップ

IdentやGSSAPIといった外部の認証システムを使用する場合は、接続を開始したオペレーティングシステムのユーザ名がデータベースユーザ名と同じであるとは限りません。ユーザ名マップを使用するには、pg_hba.conf内でmap=map-nameオプションを指定してください。このオプションは、外部ユーザ名を受け取るすべての認証方式をサポートしています。異なる接続に対して、異なるマップが必要となる可能性があります。そのため、それぞれの接続に対して使用されるマップを指定するために、使用するマップの名称は pg_hba.conf内のmap-nameパラメータで指定されます。

ユーザ名マップはidentマップファイルに定義されています。デフォルトではファイル名はpg_ident.confで クラスタのデータディレクトリに保存されています。（他の場所にも保存できますが、詳細は[ident file](#)設定パラメータを参照してください。）identマップファイルは一般的な形式の行を含んでいます。

```
map-name system-user name database-user name
```

コメントと空白はpg_hba.confと同様に扱われます。map-nameは pg_hba.conf内で参照される任意の名称です。他の2つのフィールドは、どのオペレーティングシステムユーザが、どのデータベースユーザに接続することを許可されているかを指定しています。同じmap-nameは、1つのマップ内でユーザをマップするために繰り返し使用されます。

どれだけのデータベースユーザがオペレーティングシステムのユーザに対して一致しているか、またその逆に対しても制限はありません。よってマップ内のエントリは、それらが等しいというよりもむしろ "このオペレーティングシステムのユーザはこのデータベースユーザとして接続する" という意味になります。もし外部の認証システムから得られたユーザ名と接続要求を行ったデータベースユーザ名が対となるエントリがマップ内にある場合は、接続は許可されます。

もしsystem-user nameフィールドがスラッシュ (/) で始まっている場合は、このフィールドの残りは正規表現として扱われます。（PostgreSQLの正規表現構文の詳細については[項9.7.3.1](#)を参照してください。）正規表現は単一検索や括弧を使用した表現、database-user nameフィールドで¥1（バックスラッシュ-1）で参照されるような表現を含みます。これにより、1行で複数のユーザ名のマップが可能となり、簡単な構文で特に使いやすくなります。例を以下に示します。

```
mymap    /^(.*)@mydomain¥.com$      ¥1
mymap    /^(.*)@otherdomain¥.com$   guest
```

上記のエントリでは、@mydomain.comで終わるシステムユーザ名のドメイン部分を削除して、@otherdomain.comで終わるシステムユーザ名がguestとしてログインすることを許可します。

ティップ: デフォルトでは正規表現は、文字列の一部を一致させることに注意してください。上記の例で示したように、システムユーザ名全体を強制的に一致させるために^や\$を使用すると有効です。

pg_ident.confファイルは起動時と、メインサーバのプロセスが SIGHUPを受信したときに読み込まれます。起動しているシステムで編集した場合は、ファイルを再読み込みするために (pg_ctl reloadまたはkill -HUPを使用して) postmasterにシグナルを送信する必要があります。

pg_ident.confファイルは、pg_hba.confファイルと結合して使用されます。[例19-2](#)に[例19-1](#)の例があります。この例では、192.168のネットワーク内のマシンにログインしている、オペレーティングシステムのユーザ名でbryanh、ann、robert以外の誰もが、アクセスを許可されていません。UnixユーザrobertはPostgreSQLユーザであるbobとして接続しようとした時のみアクセス可能で、robertや他の名前ではアクセスできません。annはannとして接続した時のみ許可され、bryanhはbryanh自身もしくはguest1としてアクセスが可能となります。

例 19-2. pg_ident.confファイルの例

# MAPNAME	SYSTEM-user name	PG-user name
omicron	bryanh	bryanh
omicron	ann	ann
# bobyはこれらのマシン内でrobertというユーザ名を持っています。		
omicron	robert	bob
# bryanhはguest1としても接続可能です。		
omicron	bryanh	guest1

19.3. 認証方式

以下の小節では、認証方式について詳細に説明します。

19.3.1. Trust認証

trust 認証が指定されるとPostgreSQLは、サーバに接続できる全ての人に対して（データベーススーパーユーザさえも）その人が指定する任意のデータベースユーザ名としてのアクセス権限が付与されていると想定します。当然ながらdatabaseとuser列にある制限は適用されます。この方式はサーバに接続する際に適切なオペレーティングシステムレベルの保護が掛けられている場合にのみ使用すべきです。

trust 認証はユーザが1人のみのワークステーション上でローカル接続を行う場合は適切であると同時に非常に便利です。複数ユーザが存在するマシン上では一般的に適切ではありません。とは言っても、ファイルシステムの許可属性を使ってサーバのUnixドメインソケットファイルへのアクセスを制限すればtrust 認証を複数ユーザのマシン上で使用することも可能です。その方法は、[項18.3](#)に記載されているようにunix_socket_permissions（およびunix_socket_groupパラメータの可能性もあります）パラメータを設定します。もしくは、unix_socket_directory設定パラメータでソケットファイルをそれに相応しく制限されているディレクトリにします。

Unixソケット接続を行うただ1つの方法は、ファイルシステムの許可を設定することです。ローカルのTCP/IP接続は、ファイルシステムにより制限はされていません。よってローカルでファイルシステムの許可を使用したい場合はpg_hba.confから host ... 127.0.0.1 ... の行を削除するか、trust 認証とは異なる方法に変更する必要があります。

TCP/IP接続におけるtrust 認証は、trust を指定するpg_hba.confの行によってサーバに接続を許可される全てのマシン上の全てのユーザを信用（trust）できる場合にのみ相応しいものです。ローカルホスト（127.0.0.1）以外からのTCP/IP接続にtrust 認証を用いる理由はほとんど見当たりません。

19.3.2. パスワード認証

パスワードをベースにした認証方式には md5およびpasswordがあります。これらの方式はパスワードが接続間でどのように送信されるか（すなわち、それぞれMD5-hashed、平文）を別にすれば似たような動作をします。

もしパスワード“盗聴”攻撃に最大の関心があればmd5を使うのがよいでしょう。可能ならば、平文のpasswordの使用は常に避けなければなりません。しかしmd5はdb_user_namespace機能といっしょに使用することはできません。接続がSSL暗号化により保護されているのであれば、passwordを安全に使用することができます。（ただしSSLの使用に依存するという点では、SSL証明書認証を使用した方が良いかもしれません。）

PostgreSQLデータベースパスワードはオペレーティングシステムのユーザパスワードとも別のものです。各データベースユーザのパスワードはpg_authidシステムカタログテーブルの中に格納されます。CREATE USER foo WITH PASSWORD 'secret';のように、パスワードはSQLコマンドCREATE USERとALTER ROLEを使って管理できます。もしユーザに対してパスワードが設定されない場合、格納されるパスワードはNULLとなり、そのユーザのパスワード認証は常に失敗します。

19.3.3. GSSAPI認証

GSSAPIは、RFC 2743で定義されている安全な認証のための業界標準のプロトコルです。PostgreSQLは、RFC 1964によりKerberos認証と共にGSSAPIをサポートします。GSSAPIは、GSSAPIをサポートしているシステムに対して自動認証（シングルサインオン）を提供します。認証自体は安全ですが、データベース接続を通じて送信されるデータは、SSLが使用されていない場合は平文となります。

GSSAPIがKerberosを使用しているとき、GSSAPIは、servicename/hostname@realmの書式において標準の原則を使用します。原則についての情報や、どのようにして要求された鍵を設定するのかについては[項19.3.5](#)を参照してください。

GSSAPIサポートは、PostgreSQLが構築されたときに使用可能となります。詳細は、[第15章](#)を参照してください。

次の設定オプションはGSSAPIのためにサポートされています。

include_realm

1に設定されている場合は、認証されたユーザプリンシパルからのrealm名が、ユーザ名マッピング（[項19.2](#)）で渡されるシステムユーザ名の中に含まれています。これは複数realmからのユーザを扱う場合に便利です。

map

システムとデータベースの間のマッピングを許可します。詳細は[項19.2](#)を参照してください。Kerberosプリンシパルに対しては`user name/hostbased@EXAMPLE.COM`、もし`include_realm`が無効になっている場合はマッピングに対して使用されるユーザ名は`user name/hostbased`、`include_realm`が有効になっている場合は`user name/hostbased@EXAMPLE.COM`です。

`krb_realm`

`realm`をユーザプリンシパル名に一致するように設定します。もしこのパラメータが設定されている場合は `realm`のユーザのみが受け付けられます。もしこれが設定されていない場合は、どのような`realm`のユーザも接続可能で、ユーザ名マッピングが設定されていれば、どれでも影響を受けます。

19.3.4. SSPI認証

SSPIは、シングルサインオンで安全な認証を行うためのWindowsの技術です。PostgreSQLは、`negotiate`モードにおいてSSPIを使用します。これは、可能な場合はKerberosを使用し、他の場合については自動的にNTLMを使用することを意味しています。SSPI認証は、サーバ、クライアントが共にWindows上もしくはGSSAPIが利用可能な場合はWindowsではないプラットフォームで稼動しているときにのみ動作します。

Kerberos認証を使用しているとき、SSPIは、GSSAPIと同じように動作します。詳細は[項19.3.3](#)を参照してください。

次の設定オプションはSSPIのためにサポートされています。

`include_realm`

1に設定されている場合は、認証されたユーザプリンシパルからの`realm`名が、ユーザ名マッピング ([項19.2](#)) で渡されるシステムユーザ名の中に含まれています。これは複数`realm`からのユーザを扱う場合に便利です。

`map`

システムとデータベースユーザ名間のマッピングを許可します。詳細は[項19.2](#)を参照してください。

`krb_realm`

`realm`をユーザプリンシパル名に一致するように設定します。もしこのパラメータが設定されている場合は `realm`のユーザのみが受け付けられます。もしこれが設定されていない場合は、どのような`realm`のユーザも接続可能で、ユーザ名マッピングが設定されていれば、どれでも影響を受けます。

19.3.5. Kerberos認証

注意: ネイティブのKerberos認証は、推奨されず互換性のためのみに使用されるべきです。新規に更新されたインストールは業界標準のGSSAPI認証を使用することを推奨しています。(詳細は[項19.3.3](#)を参照してください)

Kerberosは業界標準の安全な認証システムで、公開ネットワークにまたがった分散処理に適しています。Kerberosシステムの説明はこの文書の範囲外で、概ねとても（強力ですが）複雑なものと言えます。[KerberosFAQ](#)あるいは[MIT Kerberosページ](#)が探索を開始するにはお勧めです。Kerberosのソースコード配布物はいくつか存在します。Kerberosは安全な認証を提供しますが、ネットワーク越しに渡される問い合わせやデータの暗号化を行いません。このためにはSSLを使用してください。

PostgreSQLはKerberosのバージョン5をサポートしています。Kerberos認証のサポートはPostgreSQLがビルドされた時点で開始されます。詳細は[第15章](#)を参照してください。

PostgreSQLは通常の Kerberos サービスのように動作します。サービスのプリンシパル名は`servicename/hostname@realm`です。

`servicename`は[krb_srvname](#)設定パラメータを使用したサーバ側によって設定されます。さらに、クライアント側では`krbsrvname`接続パラメータを使用します。([項31.1](#)も参照してください。) インストール時のデフォルト`postgres` (ビルド時には`./configure --with-krb-srvnam=whatever`を使用しています) は、変更が可能です。多くの環境では、このパラメータは変更する必要はないでしょう。しかしながら、同一ホスト内で複数のPostgreSQLをサポートするような場合には、必要となります。いくつかのKerberosの実装では、異なるサービス名が必要になります。Microsoftアクティブディレクトリではサービス名は (`POSTGRES`) のように大文字にする必要があります。

`hostname`はサーバマシンの完全修飾されたホスト名です。サービスプリンシパルの`realm`はサーバマシンが提起した`realm`です[訳注: プリンシパルとは大雑把に2つのものを指します。1つはサービスを受けるクライアントで、もう1つはサービスを提供するサーバアプリケーションです。どちらも、認証に関してはKerberosのKDCから見るとクライアントになります]。

クライアントのプリンシパルは第一の構成要素としてPostgreSQLのデータベースユーザ名を所有しなければなりません。例えば、`pguser name/otherstuff@realm`です。もう1つの方法として、プリンシパル名の第一の構成要素からデータベースユーザ名にマップするユーザ名マッピングを使用することもできます。デフォルトではクライアントの`realm`はPostgreSQLで検査されません。相互の`realm`認証が使用可能になっていて`realm`を確認する必要がある場合は、`krb_realm`パラメータを使用するか

`include_realm`を使用可能にしてrealmを検査するためにユーザ名を使用してください。

サーバ鍵ファイルがPostgreSQLサーバアカウントによって読み込み可能（そしてできれば読み込み専用）であることを確認してください（[項17.1](#)を参照してください）。鍵ファイルの保存場所は`krb_server_keyfile`設定パラメータで指定されます。デフォルトは、`/usr/local/pgsql/etc/krb5.keytab`（もしくはビルド時に`sysconfdir`で指定されたディレクトリ）です。

keytabファイルはKerberosのソフトウェアによって作成されます。詳細はKerberosのドキュメントを参照してください。MIT互換のKerberos5実装の例を以下に示します。

```
kadmin% ank -randkey postgres/server.my.domain.org
kadmin% ktadd -k krb5.keytab postgres/server.my.domain.org
```

データベースに接続しようとしている時要求されるデータベースユーザ名に一致するプリンシパルのチケットを所有しているか確認してください。例えば、データベースユーザ名`fred`に対し、`fred@EXAMPLE.COM`のプリンシパルは接続できるでしょう。`fred/users.example.com@EXAMPLE.COM`のプリンシパルも許可するためには[項19.2](#)内に記述されているユーザ名マップを使用して下さい。

Apache Web サーバ上で[mod_auth_kerb](#)と`mod_perl`を使うのであれば`mod_perl`スクリプトで`AuthType KerberosV5SaveCredentials`が使用可能です。この方法により追加のパスワードを必要とせず、Webを通して安全なデータベースアクセスができます。

次の設定オプションはKerberosのためにサポートされています。

map

システムとデータベースユーザ名間のマッピングを許可します。詳細は[項19.2](#)を参照してください。

include_realm

1に設定されている場合は、認証されたユーザプリンシパルからのrealm名が、ユーザ名マッピング（[項19.2](#)）で渡されるシステムユーザ名の中に含まれています。これは複数realmからのユーザを扱う場合に便利です。

krb_realm

realmをユーザプリンシパル名に一致するように設定します。もしこのパラメータが設定されている場合は realmのユーザのみが受け付けられます。もしこれが設定されていない場合は、どのようなrealmのユーザも接続可能で、ユーザ名マッピングが設定されていれば、どれでも影響を受けます。

krb_server_hostname

サービスプリンシパルのホスト名部分を設定します。これは`krb_srvname`と組み合わせて使用されますが、完結したサービスプリンシパルを生成するために使用されます。つまり以下ようになります。

`krb_srvname/krb_server_hostname@REALM` 設定されていない場合は、デフォルトではサーバのホスト名になっています。

19.3.6. Ident認証

ident認証方式は、クライアントのオペレーティングシステムのユーザ名をidentサーバから入手し、それを（オプションのユーザ名マップとともに）許可されているデータベースのユーザ名として使用します。これはTCP/IP接続のみサポートされます。

注意: identが（TCP/IPではない）ローカル接続で指定されている場合、ピア認証（[項19.3.7](#)を参照してください）が代わりに使用されます。

次の設定オプションはidentのためにサポートされています。

map

システムとデータベースユーザ名間のマッピングを許可します。詳細は[項19.2](#)を参照してください。

"身元特定 (Identification) プロトコル"についてはRFC 1413で説明されています。事実上全てのUnix系のオペレーティングシステムの配布には、デフォルトでTCPポート113を監視するidentサーバが付属しています。identサーバの基本的な機能は"どのユーザがポートxからの接続を開始し、自分のポートyへの接続を初期化したのか?"というような質問に答えることです。PostgreSQLは物理的な接続が確立された時にxとyの両方を認識するので、接続するクライアントのホスト上のidentサーバに応答指令信号を送ることができ、理論的には与えられたどの接続にもオペレーティングシステムユーザを決定できます。

この手続きの欠点は、クライアントの正直さに頼るところが大きいということです。もしクライアントマシンが信用されない、もしくは危険に晒されている場合、攻撃者はポート113上でほぼどんなプログラムでも実行することができ、どのユーザ名でも好きに選んで返すことができます。したがってこの認証方式は、各々のクライアントマシンが厳格な管理下にあり、データベースとシステム管理者が密接に連絡を取り合って動作している、外界から閉ざされたネットワークにのみ適していると言えます。言い換えると、identサーバが稼働しているマシンを信用しなければなりません。次の警告に注意してください。

身元特定プロトコルは、認証、あるいはアクセス管理プロトコルには意図されていません。

--RFC 1413

いくつかの身元特定サーバは、ユーザ名を（マシンの管理者のみが知っているキーで）暗号化して返すような非標準のオプションを持っています。このオプションは、身元特定サーバとPostgreSQLとを一緒に使用する場合には、使用してはいけません。理由はPostgreSQLは、返された文字列を復号化して本当のユーザを決定するための手段を持っていないためです。

19.3.7. Peer認証

peer認証方式はカーネルからクライアント上のオペレーティングシステムのユーザ名を取得し、それをデータベースユーザ名（オプションのユーザ名マップとともに）として使用することにより動作します。この方法はローカル接続でのみ使用可能です。

次の設定オプションはpeerのためにサポートされています。

map

システムとデータベースのユーザ名のマッピングを許可します。詳細は[項19.2](#)を参照してください。

Peer認証はオペレーティングシステムが、`getpeereid()` 関数、`SO_PEERCRECRED`のソケットパラメータ、もしくは同じような仕組みを提供しているときにのみ使用可能です。現状では、Linux、Mac OS Xを含むBSD系、そしてSolarisに含まれています。

19.3.8. LDAP認証

この認証方式はpasswordと似ていますが、パスワード確認にLDAPを使用する点が異なります。LDAPはユーザの名前とパスワードの組み合わせの検証のみに使用されます。そのため、LDAPを使用して認証を行うようにする前に、ユーザはデータベースに存在しなければなりません。

LDAP認証は2つのモードで動作します。1つ目のモードではサーバは`prefix user name suffix`として区別された名前にバインドします。一般的に、`prefix`パラメータはActive Directory環境での`cn=`や`DOMAIN\`を特定するために使用されます。`suffix`は、Active Directory環境ではない場合でのDNの残りの部分を特定するために使用されます。

2つ目のモードでは、サーバは最初に`ldapbinddn`と`ldapbindpasswd`で指定された、固定されたユーザ名とパスワードを使用してLDAPディレクトリにバインドします。それからデータベースにログインしようとしているユーザを検索します。もしユーザとパスワードが指定されていなかった場合は、ディレクトリに対して匿名でバインドします。検索は`ldapbasedn`のサブツリーまで行われ、`ldapsearchattribute`で指定された属性に正確に一致するかどうかまで行われます。もし属性が指定されていない場合は、`uid`属性が使用されます。この検索において、一度ユーザが見つかったらサーバは切断して、クライアントで指定されたパスワードを使用してこのユーザとして再度ディレクトリにバインドします。これはログインが正しいかどうかを確認するためです。この方法は、ユーザオブジェクトがディレクトリに配置されている場合に、かなりの柔軟性があります。しかし、LDAPサーバへの2つの分離した接続が作成されます。

次の設定オプションはLDAPのためにサポートされています。

ldapserver

接続するLDAPサーバの名称もしくはIPアドレスの名称。空白で区切ることで複数のサーバを指定できます。

ldapport

LDAPサーバに接続するためのポート番号。もしポートが指定されていない場合はLDAPライブラリ内のデフォルトポート設定が使用されます。

ldaptls

1に設定すると、PostgreSQLとLDAPサーバ間の接続にTLSによる暗号化を使用します。これはLDAPサーバへのトラフィックのみを暗号化することに注意してください。— クライアントへの接続はSSLを使用しない限り暗号化されません。

ldapprefix

単純なバインド認証を行う場合のDNを生成する際にユーザ名の前に追加する文字列

ldapsuffix

単純なバインド認証を行う場合のDNを生成する際にユーザ名の前に追加する文字列

ldapbasedn

検索とバインドの認証を行う場合のユーザ名がログインするための検索を始めるためのDN

ldapbinddn

検索とバインドの認証を行う場合のディレクトリと検索をバインドするためのユーザのDN

ldapbindpasswd

検索とバインドの認証を行う場合のディレクトリと検索をバインドするためのユーザのパスワード

ldapsearchattribute

検索とバインドの認証を行う場合の検索時のユーザ名に対して一致させる属性

注意: LDAPは、DNの異なる部分を分けるのにしばしばカンマと空白を使用します。そのため LDAPオプションを設定する際に2重引用符パラメータ値を使用する必要があります。 例を以下に示します。

```
ldapserver=ldap.example.net ldapprefix="cn=" ldapsuffix=", dc=example, dc=net"
```

19.3.9. RADIUS Authentication

この認証方法は、RADIUSをパスワード検証として使用するという点を除いてpasswordと似た動作をします。 RADIUSはユーザ名/パスワードの組のみを検証するために使用されます。 よってユーザはRADIUSが認証に使用される以前にデータベースにすでに存在していなければいけません。

RADIUS認証を使用する場合に、設定されたRADIUSサーバにアクセスリクエストメッセージが送信されます。 このリクエストはAuthenticate Onlyの形式になり、ユーザ名、(暗号化された)パスワード、NAS識別子を含んでいます。 リクエストはサーバと共有している秘密を用いて暗号化されます。 RADIUSサーバは、このサーバに対してAccess AcceptもしくはAccess Rejectを返します。 RADIUSアカウントのサポートはありません。

RADIUSのために次の設定オプションがサポートされています。

radiusserver

接続するRADIUSサーバの名称もしくはIPアドレス。このパラメータは必要です。

radiussecret

RADIUSサーバとのやり取りに使用される共有の秘密。これはPostgreSQLとRADIUSサーバにおいて 厳密に同じ値にする必要があります。少なくとも16文字以上の文字列が推奨されます。このパラメータは必要です。

注意: PostgreSQLがOpenSSLをサポートする場合に暗号化ベクター使用するときのみ 強力になります。他の場合にはRADIUSサーバへの伝送は難読化されているだけで、安全ではなく 必要な場合は外部のセキュリティ方法を適用すべきです。

radiusport

接続するRADIUSサーバのポート番号。もしポート番号が指定されていない場合は、デフォルトポートである1812が使用されます。

radiusidentifier

RADIUSリクエスト内でNAS Identifierとして使用されている文字列。 ユーザがどのデータベースユーザに対して認証しようとしているか、RADIUSサーバにおいてポリシーを一致させるために何が使用されるか、 を識別するために、このパラメータは2番目のパラメータとして使用されます。 もし識別子が指定されていない場合は、デフォルトのpostgresqlが使用されます。

19.3.10. 証明書認証

この認証方法は、認証のためにSSLクライアント証明書を使用します。 よってこの方法は、SSL接続を使用します。 この認証方法を使用する際は、サーバはクライアントが有効な証明書を提供することを要求します。 パスワードのプロンプトはクライアントに送信されません。 証明書のcn (Common Name) 属性は、要求されたデータベースユーザ名と比較されます。 もしそれらが一致した場合はログインが許可されます。ユーザ名マッピングは、cnがデータベースユーザ名と異なるものであることを許可するために使用されます。

次の設定オプションはSSL証明書認証のためにサポートされています。

map

システムとデータベースユーザ名間のマッピングを許可します。 詳細は[項19.2](#)を参照してください。

19.3.11. PAM認証

この認証方式は認証機構としてPAM（Pluggable Authentication Modules）を使用することを除いてpasswordのように動作します。デフォルトのPAMサービス名はpostgresqlです。PAMはユーザ名/パスワードの組を承認するためだけに使用されま
す。PAMについての詳細は[Linux-PAMページ](#)と[Solaris PAMページ](#)を読んでください。

次の設定オプションはPAMのためにサポートされています。

pamservice

PAMサービス名。

注意: PAMが/etc/shadowを読み取るように設定されている場合は、PostgreSQLがルートユーザで起動されてい
ないため、認証は失敗するでしょう。しかしPAMがLDAPや他の認証方法を使用するように設定されている場合は、
これは問題ではありません。

[前のページ](#)
ユーザ名マップ

[ホーム](#)
[上に戻る](#)

[次のページ](#)
認証における問題点

19.4. 認証における問題点

本来の認証失敗とそれに関連した問題は、一般的に以下のようなエラーメッセージを通して明示されます。

```
FATAL: no pg_hba.conf entry for host "123.123.123.123", user "andym", database "testdb"
```

たいがい、サーバとの接触に成功はしたものの、サーバが通信を拒否した場合です。メッセージが指摘するようにサーバは接続要求を拒否しました。なぜならpg_hba.conf設定ファイルに一致する項目を見つけることができなかったからです。

```
FATAL: password authentication failed for user "andym"
```

この種のメッセージは、サーバと接触し、サーバも通信することを許可したが、pg_hba.confファイルの中で指定された認証方式に合格していないことを表します。入力したパスワードを確認するか、もしエラーがKerberos、ident認証型のいずれかを指摘している場合はKerberosあるいはidentソフトウェアを確認してください。

```
FATAL: user "andym" does not exist
```

与えられたデータベースユーザ名は見つかりませんでした。

```
FATAL: database "testdb" does not exist
```

接続しようとしているデータベースは存在しません。データベース名を指定しなければ、それが望む望まざるにかかわらず、データベースユーザ名がデフォルトとなることに注意してください。

ティップ: クライアントに報告される以上により多くの情報がサーバログに残ります。失敗した原因についてよくわからなければサーバのログを見てください。

第 20 章 データベースロール

目次

- 20.1. [データベースロール](#)
- 20.2. [ロールの属性](#)
- 20.3. [ロールのメンバ資格](#)
- 20.4. [関数とトリガのセキュリティ](#)

PostgreSQLは、ロールという概念を使用してデータベースへの接続承認を管理します。ロールは、その設定方法に応じて、データベースユーザ、またはデータベースユーザのグループとみなすことができます。ロールはデータベースオブジェクト（例えばテーブル）を所有することができます。またロールは、どのオブジェクトに誰がアクセスできるかを制御するために、それらのオブジェクトに対しての権限を他のロールに割り当てることができます。更に、ロールのメンバ資格を他のロールに与えることもできます。そのため、メンバとなったロールは別のロールに割り当てられた権限を使用することができます。

ロールの概念には、"ユーザ"という概念と"グループ"という概念が含まれます。PostgreSQLバージョン8.1より前まででは、ユーザとグループは異なる種類の実体として扱われていました。しかし、現在ではロールしか存在しません。すべてのロールは、ユーザとして、グループとして、またはその両方として動作することができます。

本章では、ロールの作成と管理の方法について説明します。様々なデータベースオブジェクト上の権限の効果について、さらに詳細な情報は[項5.6](#)に記載されています。

20.1. データベースロール

データベースロールは概念的に、オペレーティングシステムユーザとは完全に分離されています。 実行する上でユーザ名を一致させておくくと便利ですが、必須ではありません。 データベースロール名はデータベースクラスインストール全体で共通です（個別のデータベースごとではありません）。 ユーザを作成するためには[CREATE ROLE](#) SQLコマンドを使います。

```
CREATE ROLE name;
```

name はSQL識別子の規則に従います。 特殊な文字を持たない無装飾のものか、二重引用符に囲まれたもののどちらかです。（現実的には、通常他のオプション、例えばLOGINなどをこのコマンドに付与することになるでしょう。 詳細は後で説明します。） 既存のユーザを削除するためには類似のコマンド[DROP ROLE](#)を使用してください。

```
DROP ROLE name;
```

利便性のために、これらのSQLコマンドのラッパである、シェルのコマンドラインから呼び出し可能な[createuser](#)プログラムと[dropuser](#)プログラムが提供されています。

```
createuser name
dropuser name
```

既存のロール群を求めるためには、以下のようにpg_rolesシステムカタログを確認してください。

```
SELECT rolname FROM pg_roles;
```

また、[psql](#)プログラムの`\du`メタコマンドも既存のロールを列挙する際に役に立ちます。

データベースシステム自身を起動するために、初期化されたばかりのシステムは常に定義済みのロールを1つ持ちます。 このロールは必ず"スーパーユーザ"であり、デフォルトでは（`initdb`実行時に変更しない限り）そのデータベースクラスを初期化したオペレーティングシステムユーザと同じ名前となります。 習慣的にこのロールはpostgresと名付けられます。 ロールを追加する場合はまずこの初期ロールで接続しなければいけません。

すべてのデータベースサーバへの接続は、特定のロールの名前を使用して確立し、そのロールによりその接続で発行されるコマンドの初期のアクセス権限が決まります。 特定のデータベース接続に使うロールは、アプリケーション固有の方式で接続要求を開始するクライアントによって指示されます。 例えば、`psql` プログラムでは、`-U` コマンドラインオプションを使って接続するロールを指示します。 多くのアプリケーション（`createuser` および `psql` を含む）では、オペレーティングシステムの現在のユーザ名をデフォルトと想定します。 したがって、ロールとオペレーティングシステムのユーザの組み合わせ間で名前を一致させておくくと便利です。

[第19章](#)で説明されているように、あるクライアント接続で与えられたデータベースロールの集合は、クライアント認証設定で決定された内容で接続できます。（したがって、ユーザのログイン名が本名と一致していなくても構わないのと同様に、クライアントはオペレーティングシステムのユーザ名と同じロール名で接続しなくても構いません）。 接続したクライアントに付与される権限の内容はロールIDによって決定されるため、マルチユーザ環境を設定する際には権限を注意深く設定することが重要です。

20.2. ロールの属性

データベースロールは、権限を定義し、クライアント認証システムと相互作用する数多くの属性を持つことができます。

ログイン権限

`LOGIN`属性を持つロールのみがデータベース接続の初期ロール名として使用できます。`LOGIN` 属性を持つロールは"データベースユーザ"と同じであるとみなすことができます。 ログイン権限を持つロールの作成方法は、以下のいずれかです。

```
CREATE ROLE name LOGIN;  
CREATE USER name;
```

(`CREATE USER`はデフォルトで `LOGIN` を持ち、`CREATE ROLE`は持たないという点を除き、`CREATE USER`は`CREATE ROLE`と同じです。)

スーパーユーザ状態

ログイン権限を除き、データベーススーパーユーザに対する権限検査は全て行われません。 これは危険な権限ですので、安易に使用してはいけません。 作業のほとんどを非スーパーユーザのロールで行うことが最善です。 新しいデータベーススーパーユーザを作成するには、`CREATE ROLE name SUPERUSER`を使用してください。 これはスーパーユーザのロールで実行しなければなりません。

データベース作成

(全ての権限検査が行われないスーパーユーザを除き) ロールに明示的にデータベースを作成するための権限を指定しておかねばいけません。 そのようなロールを作るためには`CREATE ROLE name CREATEDB`を使用してください。

ロールの作成

あるロールがロールを作成するには、明示的な権限が付与されていなければなりません。(スーパーユーザは、すべての権限検査を迂回しますので、例外です。) こうしたロールを作成するには、`CREATE ROLE name CREATEROLE`を使用してください。`CREATEROLE` 権限を持つロールは他のロールを変更したり削除したりすることもできます。 また、他のロールのメンバ資格を付与したり取り上げたりすることもできます。 しかし、スーパーユーザロールの変更、削除、メンバ資格の変更を行うにはスーパーユーザ状態が必要です。`CREATEROLE`だけでは不足しています。

レプリケーションの新規接続

あるロールがストリーミングレプリケーションの新規接続を実施するには、明示的な権限が付与されていなければなりません。(スーパーユーザは、すべての権限検査を迂回しますので、例外です。) ストリーミングレプリケーションを行うロールは、常に`LOGIN`権限が持っていることが必要です。 こうしたロールを作成するには、`CREATE ROLE name REPLICATION LOGIN`を使用してください。

パスワード

パスワードは、クライアント認証方法においてデータベースに接続する際にユーザにパスワードを要求する場合にのみ重要になります。`password`と`md5`認証方式でパスワードが使用されます。 データベースパスワードはオペレーティングシステムのパスワードとは異なります。 ロール作成時に`CREATE ROLE name PASSWORD 'string'`のようにパスワードを指定します。

ロール属性は、ロールを作成した後も`ALTER ROLE`コマンドで変更できます。 詳細は[CREATE ROLE](#)と[ALTER ROLE](#)のマニュアルページを参照してください。

ティップ: スーパーユーザ以外に`CREATEDB`権限と`CREATEROLE`権限を持つロールを作成することを勧めます。 そして、このロールを使用して、データベースとロールを管理するためのすべての処理を行ってください。 この方法によって、実際には不要な処理をスーパーユーザとして行う危険性を避けることができます。

ロールは、[第18章](#)で説明されている実行時の設定の多くをロールごとのデフォルトに設定することもできます。 例えば何らかの理由で、自分が接続する時は常にインデックススキャンを無効にしたい場合(注:お勧めしません)、次のようにします。

```
ALTER ROLE myname SET enable_indexscan TO off;
```

このようにして設定を保存します(ただし、すぐに反映はされません)。 以降のこのロールによる接続においては、セッションの開始の直前に`SET enable_indexscan TO off;`が呼び出されたのと同様になります。 これはデフォルトとする設定をするだけなので、設定をセッション途中いつでも変更できます。 ロール固有のデフォルト設定を削除するには、`ALTER ROLE rolename RESET varname`を使用してください。 呼び出されることがありませんので、`LOGIN`権限を持たないロールにロール固有のデフォ

ルトを持たせることに意味がないことに注意してください。

[前のページ](#)
データベースロール

[ホーム](#)
[上に戻る](#)

[次のページ](#)
ロールのメンバ資格

20.3. ロールのメンバ資格

権限の管理を簡単にするために、ユーザをグループにまとめることはしばしば便利です。グループ全体に対して権限を与えることも、取り消すこともできます。PostgreSQLでは、グループを表すロールを作成することで行われます。そして、そのグループロールに個々のユーザロールのメンバ資格を与えます。

グループロールを設定するには、まずロールを作成します。

```
CREATE ROLE name;
```

通常、グループとして使用されるロールにはLOGIN属性を持たせません。しかし、そうしたければ持たせることもできます。

グループロールをいったん作成すれば、[GRANT](#)および[REVOKE](#)コマンドを使用してメンバの追加と削除を行うことができます。

```
GRANT group_role TO role1, ... ;  
REVOKE group_role FROM role1, ... ;
```

他のグループロールへのメンバ資格を与えることもできます。（グループロールと非グループロールとの間には実際には区別がないからです。）データベースはグループのメンバ資格がループし、循環するような設定はさせません。また、ロール内のメンバ資格をPUBLICに付与することはできません。

グループロールのメンバは、2つの方法でロールの権限を使用することができます。1つ目は、グループ内のすべてのメンバは明示的に、一時的にそのグループロールに"なる"ために[SET ROLE](#)を行うことができます。この状態では、データベースセッションは元々のログインロールの権限ではなくグループの権限でアクセスされます。そして、作成されたデータベースオブジェクトの所有者はログインロールではなくグループロールであるとみなされます。2つ目は、[INHERIT](#)属性を持つメンバロールは、それらがメンバとなるロールの権限を自動的に使用します。これには、ロールによって継承されるいかなる権限も含んでいます。例えば、以下の状態を想定します。

```
CREATE ROLE joe LOGIN INHERIT;  
CREATE ROLE admin NOINHERIT;  
CREATE ROLE wheel NOINHERIT;  
GRANT admin TO joe;  
GRANT wheel TO admin;
```

joeロールで接続するとすぐに、joeはadmin権限を"継承"しますので、そのデータベースセッションではjoeに直接与えられた権限に加えて、adminに与えられた権限を使用することができます。しかし、wheelに与えられた権限は利用できません。joeは間接的にwheelのメンバですが、admin経由のメンバ資格はNOINHERIT属性を持っているためです。

```
SET ROLE admin;
```

を行った後、セッションはadminに与えられた権限のみを使用できるようになります。joeに与えられた権限は使用できなくなります。

```
SET ROLE wheel;
```

を行った後、セッションはwheelに与えられた権限のみを使用できるようになり、joeやadminに与えられた権限は使用できなくなります。元の状態の権限に戻すには、以下のいずれかを行います。

```
SET ROLE joe;  
SET ROLE NONE;  
RESET ROLE;
```

注意: SET ROLE コマンドによりいつでも、元のログインロールが直接あるいは間接的にメンバ資格を持つすべてのロールを選ぶことができます。従って、上の例において、wheelになる前にadminになることは必要ありません。

注意: 標準SQLでは、ユーザとロールとの間に明確な違いがあり、ユーザはロールのように自動的に権限を継承することができません。PostgreSQLでこの振舞いを実現させるには、SQLロールとして使用するロールにはINHERIT属性を付与し、SQLユーザとして使用するロールにはNOINHERIT属性を付与します。しかし、8.1リリースより前との互換性を保持するために、PostgreSQLはデフォルトで、すべてのロールにINHERIT属性を付与します。以前は、ユーザは常にメンバとして属するグループに付与された権限を常に使用できました。

LOGIN、SUPERUSER、CREATEDB、およびCREATEROLEロール属性は、特別な権限とみなすことができますが、データベースオブジェクトに対する通常の権限のように継承されません。こうした属性の1つを使用できるようにするためには、その属性を特定のロールに設定するように実際にSET ROLEを行う必要があります。上の例を続けると、adminロールにCREATEDB権限とCREATEROLE権限を付与することを選ぶことができます。こうすると、joeロールとして接続するセッションでは、すぐさまこれらの権限を持ちません。SET ROLE adminを行った後で、この権限を持ちます。

グループロールを削除するには、[DROP ROLE](#)を使用してください。

DROP ROLE *name* ;

グループロール内のメンバ資格も自動的に取り上げられます。（しかし、メンバロールには何も影響ありません。）しかし、先にグループロールが所有するオブジェクトをすべて削除しておくか所有者を変更しておく必要があり、また、グループに付与された権限もすべて取り上げられることに注意してください。

[前のページ](#)

ロールの属性

[ホーム](#)
[上に戻る](#)

[次のページ](#)

関数とトリガのセキュリティ

20.4. 関数とトリガのセキュリティ

関数とトリガによって、ユーザは他のユーザが意識しないで実行できるようなコードを、バックエンドサーバに挿入することができます。したがって、両機能によってユーザは比較的簡単に他のユーザにとって"トロイの木馬"となるものを実行することができます。唯一の現実的保護は、誰が関数やトリガを定義できるかを厳格に管理することです。

関数は、データベースサーバデーモンのオペレーティングシステム権限で、バックエンドサーバプロセスの中で実行されます。プログラミング言語で関数に未検査のメモリアクセスを許可している場合、サーバの内部データ構造を変更することが可能です。したがって、その他の多数のことの中でも、そのような関数はどのようなシステムアクセスコントロールも回避することができます。このようなアクセスを許可する関数言語は"信頼されない"ものとみなされ、PostgreSQLはこれらの言語による関数の作成をスーパーユーザのみに限定して許可しています。

第 21 章 データベース管理

目次

- 21.1. [概要](#)
- 21.2. [データベースの作成](#)
- 21.3. [テンプレートデータベース](#)
- 21.4. [データベースの設定](#)
- 21.5. [データベースの削除](#)
- 21.6. [テーブル空間](#)

稼動しているPostgreSQLサーバのすべてのインスタンスは、1つ以上のデータベースを管理します。したがって、データベースはSQLオブジェクト ("データベースオブジェクト") を組織化する場合に最上位の階層レベルとなります。本章では、データベースの特性、作成方法、管理方法、および削除方法について説明します。

21.1. 概要

データベースとは、SQLオブジェクト ("データベースオブジェクト") の集合に名前を付けたものです。一般的には、すべてのデータベースオブジェクト (テーブル、関数など) は、1つのデータベースのみに属します (しかし、例えば、`pg_database` といったシステムカタログのごく一部には、クラスタ全体に属するものもあり、そのクラスタ内のすべてのデータベースからアクセスすることができます)。より正確には、データベースはスキーマの集合であって、スキーマにテーブル、関数などが含まれます。したがって階層の全体像は、サーバ、データベース、スキーマ、テーブル (や関数などの何らかのオブジェクト) となります。

データベースサーバに接続する時、クライアントはその接続要求の際に接続するデータベース名を指定しなければなりません。1つの接続で複数のデータベースにアクセスすることはできません。しかし、アプリケーションには同じデータベースや他のデータベースへの接続に接続数の制限はありません。複数のデータベースは物理的に分離されていて、アクセス制御は接続レベルで管理されています。したがって、分離して、ほとんどの場面で互いに見えないようにする必要のある複数のプロジェクトやユーザを単一のPostgreSQLサーバインスタンスに収容する場合、これらを別々のデータベースに含めることが推奨されます。もし、複数のプロジェクトやユーザが相互に関連していて互いのリソースを使用できる必要がある場合、これらは同じデータベースに含めるべきですが、スキーマを別にすることは可能です。スキーマは純粋に論理的な構造で、誰が何にアクセスできるかは権限システムによって管理されます。スキーマの管理についての詳細は[項5.7](#)に記載されています。

データベースは`CREATE DATABASE`コマンド ([項21.2](#)を参照) で作成され、`DROP DATABASE`コマンド ([項21.5](#)を参照) で破棄されます。既存のデータベース群を求めるには、以下の例のように`pg_database`システムカタログを確認してください。

```
SELECT datname FROM pg_database;
```

また、`psql`プログラムの`\l`メタコマンドや`-l`コマンドラインオプションも既存のデータベースを列挙する際に役に立ちます。

注意: 標準SQLでは、データベースを"カタログ" (catalog) と呼ぶこともありますが、実際のところ違いはありません。

21.2. データベースの作成

データベースを作成する場合、PostgreSQLサーバが起動している必要があります ([項17.3](#)を参照してください)。

[CREATE DATABASE](#) SQLコマンドでデータベースを作成することができます。

```
CREATE DATABASE name;
```

ここで、*name*はSQL識別子の通常の規則に従います。現在のロールが自動的に新しいデータベースの所有者になります。作成後、データベースを削除する権限はこの所有者にあります (この作業では、そのデータベースに属している、所有者のものではないオブジェクトでも、すべて削除されます)。

データベースの作成は制限された作業です。権限の付与に関する詳細は[項20.2](#)を参照してください。

[CREATE DATABASE](#) コマンドを実行するためには、データベースサーバに接続している必要があります。そうすると、あるサイトの最初のデータベースはどのようにして作成するのかという疑問が出てきます。最初のデータベースは`initdb`コマンドでデータ格納領域が初期化される ([項17.2](#)を参照してください。) 時、必ず作成されます。このデータベースは`postgres`と呼ばれます。したがって、最初の"通常の"データベースを作成するには`postgres`に接続してください。

また、`template1`という2つ目のデータベースもデータベースクラスタの初期化時に作成されます。クラスタ内に新しいデータベースが作成されたら、実際に`template1`が複製されます。つまり`template1`に変更を与えると、その後に作成されるデータベースすべてにその変更が反映されることを意味します。このため新しく作成するデータベースすべてに反映させたい場合でない限り`template1`内にオブジェクトを作成することは避けてください。詳細については[項21.3](#)を参照してください。

利便性のために、シェルから`createdb`を実行して、新しいデータベースを作成することができます。

```
createdb dbname
```

`createdb`は魔法ではありません。これは`postgres`データベースに接続し、先に解説した通りに[CREATE DATABASE](#) コマンドを実行します。 [createdb](#)のマニュアルページに実行方法の詳細が説明されています。 引数のない`createdb`は現在のユーザ名のデータベースを作成しますので、注意してください。

注意: 特定のデータベースに誰が接続できるかを制限する方法については[第19章](#)に記載されています。

他のユーザのためにデータベースを作成し、そのユーザ自身が設定、管理できるように新しいデータベースの所有者にさせたい場合も考えられます。 そのためには、次のコマンドのいずれかを使用します。 SQL環境からであれば前者を、さもなくば後者をシェルから使用してください。

```
CREATE DATABASE dbname OWNER rolename;
```

```
createdb -O rolename dbname
```

他のユーザのために (つまり、自身がメンバではないロールのために) データベースを作成することができるのはスーパーユーザだけです。

21.3. テンプレートデータベース

実際のCREATE DATABASEの動作は、既存のデータベースをコピーすることです。デフォルトでは、template1という名前の標準のシステムデータベースをコピーします。したがって、このデータベースは新しく作成するデータベースの元になる"テンプレート"となります。template1にオブジェクトを追加した場合、追加したオブジェクトはその後に作成されるユーザデータベースにコピーされます。この振舞いによって、データベース標準オブジェクト群にサイト独自の変更を行うことができます。例えば、PL/Perl手続き言語をtemplate1にインストールした場合、データベースを作成する時に追加作業を行うことなく、自動的にこの言語をユーザデータベースで使うことができます。

template0という名前の二次的な標準システムデータベースがあります。このデータベースにはtemplate1の初期内容と同じデータが含まれています。つまり、使用しているバージョンのPostgreSQLで定義済みの標準オブジェクトのみから構成されています。template0をデータベースクラスタを初期化した後に変更してはいけません。CREATE DATABASEをtemplate1ではなくtemplate0をコピーするように実行することで、template1に追加されたサイト独自のものを含まない、"汚れがない"ユーザデータベースを作成することができます。これは特に、pg_dump ダンプからリストアする時に便利です。このダンプスクリプトは、後でtemplate1に追加される可能性のあるオブジェクトと衝突しないように、ダンプしたデータベースの内容を正しく再作成するために、汚れのないデータベースにリストアされなければなりません。

template1の代わりにtemplate0をコピーするその他の一般的な理由は、template0をコピーするときに新規の符号化方式とロケールを設定することです。一方、template1のコピーはそれが行ったと同一の設定を使用しなければなりません。これはtemplate0が理解しない符号化方式特有の、またはロケール特有のデータを含んでいる可能性があることに依ります。

template0をコピーしてデータベースを作成するには、

```
CREATE DATABASE dbname TEMPLATE template0;
```

をSQL環境から実行するか、または

```
createdb -T template0 dbname
```

をシェルから実行します。

さらにテンプレートデータベースを作成することができます。また、実際のところCREATE DATABASEのテンプレートとして名前を指定することで、クラスタ内の任意のデータベースをコピーすることができます。しかし、この機能は、(まだ)汎用目的の"COPY DATABASE"能力を意図したものではないことは理解しておいてください。コピー操作の間、他のセッションから元のデータベースに接続することができないという点が大きな制限です。CREATE DATABASEは、その起動時に他の接続が存在する場合失敗します。コピー操作中は元のデータベースへの新しい接続を許しません。

datistemplate列とdataallowconn列という、データベースそれぞれに有用なフラグがpg_databaseに存在します。datistemplateは、そのデータベースがCREATE DATABASEのテンプレートとして使用されることを目的としているものであることを意味するために設定することができます。このフラグが設定された場合、CREATEDB権限を持つすべてのユーザはそのデータベースを複製することができます。設定されていない場合は、スーパーユーザとそのデータベース所有者のみがそれを複製することができます。dataallowconnが偽の場合、そのデータベースへの新規接続はできません(しかし、このフラグを偽にするだけでは既存のセッションは閉ざされません)。template0データベースは、変更を防ぐために、通常dataallowconn = falseとされています。template0とtemplate1の両方は、常にdatistemplate = trueとされていなければなりません。

注意: template1とtemplate0には、template1という名前がCREATE DATABASEのデフォルトのソースデータベースの名前であること以上の特別な地位はありません。例えば、template1を削除し、template0から再作成しても何も問題ありません。この操作は、不注意にごみをtemplate1に追加してしまった場合にお勧めします。(template1を削除するには、pg_database.datistemplate = falseとしなければなりません。)

データベースクラスタが初期化される時、postgresも作成されます。このデータベースは、ユーザとアプリケーションのデフォルトの接続先を意図したものです。これはtemplate1の単純なコピーで、必要に応じて削除したり再作成したりすることができます。

21.4. データベースの設定

[第18章](#)で説明したように、PostgreSQLサーバには多数の実行時の設定変数が存在します。これらの設定の多くに対して、データベース固有のデフォルト値を設定できます。

例えば、何らかの理由で特定のデータベースについてGEQOオプティマイザを無効にしたい場合、通常はすべてのデータベースでこれを無効にするか、またはすべての接続しているクライアントで間違いなく`SET geqo TO off;`が実行されていることを確認する必要があります。特定のデータベースでこの設定をデフォルトにするには、次のコマンドを実行します。

```
ALTER DATABASE mydb SET geqo TO off;
```

これにより設定が保存されます（ただし、すぐに反映はされません）。以降のこのデータベースに対する接続においては、セッションの開始の直前に`SET geqo TO off;`が呼び出されたのと同様になります。これはデフォルトでしかありませんので、ユーザはセッションの途中であってもこの設定を変更することができます。このような設定を解除するには、`ALTER DATABASE dbname RESET varname`を使用します。

21.5. データベースの削除

データベースの削除には、以下の[DROP DATABASE](#)コマンドを使用します。

```
DROP DATABASE name ;
```

データベースの所有者とスーパーユーザのみがデータベースを削除することができます。データベースの削除はそのデータベースに含まれるすべてのオブジェクトを削除します。削除されたデータベースは復元できません。

削除しようとしているデータベースに接続している時には[DROP DATABASE](#)を実行することはできません。しかし、`template1`などのその他のデータベースに接続すれば削除することができます。また、そのクラスタの最後のユーザデータベースを削除する時には、`template1`データベースに接続するしかありません。

利便性のため、データベースを削除する[dropdb](#)シェルプログラムもあります。

```
dropdb dbname
```

(`createdb`と異なり、デフォルトで現在のユーザ名のデータベースを削除するようにはなっていません。)

21.6. テーブル空間

PostgreSQLのテーブル空間により、データベース管理者はデータベースオブジェクトを表すファイルを格納できるファイルシステム上の場所を定義することができます。 テーブル空間を一度作成すると、データベースオブジェクトを作成する時に名前により参照することができます。

テーブル空間を使用することで、管理者はPostgreSQLインストレーションのディスクレイアウトを制御することができます。これは、少なくとも2つの点で有用です。 1つは、クラスタを初期化したパーティションもしくはボリュームの容量が不足し、拡張を行うことができない場合、システムを再構成するまで、別のパーティションにテーブル空間を作成して、このテーブル空間を使用することができます。

もう1つは、テーブル空間により、管理者はデータベースオブジェクトの使用パターンに基づいてデータ格納場所を調整することができます。例えば、非常によく使用されるインデックスを、例えば、高価なソリッドステートディスクなどの高速・高可用性ディスクに格納することができます。同時に、めったに使用されない保存用のデータや性能が求められていない保存用のデータを格納するテーブルを低価格・低速ディスクシステムに格納することもできます。

テーブル空間を定義するには[CREATE TABLESPACE](#)コマンドを使用してください。 以下に例を示します。

```
CREATE TABLESPACE fastspace LOCATION '/mnt/sdal/postgresql/data';
```

この場所は、既存かつ空のディレクトリでなければならず、また、PostgreSQLオペレーティングシステムユーザが所有していなければなりません。 その後に、テーブル空間内に作成されるオブジェクトはすべてこのディレクトリ以下のファイルに格納されます。

注意: 通常、論理ファイルシステム内の個々のファイルの場所を制御することができませんので、1つの論理ファイルシステムに複数のテーブル空間を作成することは意味がありません。 しかし、PostgreSQLにはこうした制限はありません。 実際、システムのファイルシステムの境を直接意識しません。 単に使用することを宣言したディレクトリにファイルを格納します。

テーブル空間自体の生成はデータベーススーパーユーザが行う必要があります。 しかし、その後に、データベース一般ユーザにそのテーブル空間を使用させることができます。 これを行うには、ユーザにテーブル空間に対する[CREATE](#)権限を与えてください。

テーブル、インデックス、およびデータベース全体は特定のテーブル空間に割り当て可能です。 これを行うには、指定テーブル空間に[CREATE](#)権限を持つユーザが関連するコマンドにテーブル空間をパラメータとして渡す必要があります。 例えば、以下はspace1テーブル空間にあるテーブルを作成しています。

```
CREATE TABLE foo(i int) TABLESPACE space1;
```

他の方法として、以下のように[default_tablespace](#)パラメータの使用があります。

```
SET default_tablespace = space1;  
CREATE TABLE foo(i int);
```

`default_tablespace`が空文字以外の何かに設定された場合、この値が、明示的なテーブル空間の指定がない[CREATE TABLE](#)コマンドや[CREATE INDEX](#)コマンドの暗黙的な `TABLESPACE` 句として適用されます。

[temp tablespaces](#)というパラメータも存在します。 これは、一時テーブルとそのインデックス、大規模データ集合のソートなどを目的に使用される一時ファイルの場所を決定するものです。 これは、テーブル空間名を1つだけ指定するものではなく、テーブル空間名のリストを取ることができます。 このため、一時的なオブジェクトに関連する負荷を、複数のテーブル空間にわたって分散することができます。 一時的なオブジェクトを作成する度に、このリストから無作為に要素が選択されます。

データベースに関連付けられたテーブル空間は、そのデータベースのシステムカタログを格納するために使用されます。 さらに、`TABLESPACE` 句を付けずに、または、(適切な) `default_tablespace`や`temp tablespaces`で指定された選択肢がなければ、データベース内に作成されたテーブルやインデックスのデフォルトのテーブル空間としても使用されます。 テーブル空間の指定なしで作成されたデータベースは、コピー元のテンプレートデータベースのテーブル空間と同じものを使用します。

データベースクラスタが初期化される時に2つのテーブル空間が自動的に作成されます。 `pg_global` テーブル空間は共有システムカタログ用に使用されます。 `pg_default` テーブル空間は`template1`と`template0`データベースのデフォルトテーブル空間です (したがって、[CREATE DATABASE](#)の`TABLESPACE` 句で変更されない限り、このテーブル空間が同様に他のデータベースに対するデフォルトのテーブル空間になります)。

テーブル空間は、一度作成すると、要求しているユーザが十分な権限を持っていればすべてのデータベースから使用することができます。 これは、テーブル空間を使用するすべてのデータベースのすべてのオブジェクトが削除されるまで、そのテーブル空間を削除することができないことを意味します。

空のテーブル空間を削除するには、[DROP TABLESPACE](#)コマンドを使用してください。

既存のテーブル空間群を求めるには、以下の例のようにpg_tablespaceを確認してください。

```
SELECT spcname FROM pg_tablespace;
```

また、[psql](#)プログラムの`\db`メタコマンドも既存のテーブル空間を列挙する際に役に立ちます。

テーブル空間の実装を単純化するために、PostgreSQLはシンボリックリンクを使用します。これは、テーブル空間はシンボリックリンクをサポートするシステムでのみ使用することができるということを意味します。

\$PGDATA/pg_tblspcディレクトリには、そのクラスタで定義された非組み込みテーブル空間1つひとつを指し示すシンボリックリンクがあります。推奨はしませんが、こうしたリンクを手作業で再定義してテーブル空間のレイアウトを調整することができます。2点警告します。これをサーバが実行している時に行わないでください。また、サーバを再起動した後に、新しい場所でpg_tablespaceカタログを更新してください。（これを行わないと、pg_dumpが古いテーブル空間の場所を示したままになってしまいます。）

[前のページ](#)
データベースの削除

[ホーム](#)
[上に戻る](#)

[次のページ](#)
多言語対応

第 22 章 多言語対応

目次

22.1. [ロケールのサポート](#)

- 22.1.1. [概要](#)
- 22.1.2. [動作](#)
- 22.1.3. [問題点](#)

22.2. [照合サポート](#)

- 22.2.1. [概念](#)
- 22.2.2. [照合の管理](#)

22.3. [文字セットサポート](#)

- 22.3.1. [サポートされる文字セット](#)
- 22.3.2. [文字セットの設定](#)
- 22.3.3. [サーバ・クライアント間の自動文字セット変換](#)
- 22.3.4. [推奨文書](#)

本章では、管理者の立場から見た、利用可能な多言語対応機能について説明します。 PostgreSQL では、2つの手法で多言語対応をサポートします。

- ロケール固有の照合順序、数字の書式、翻訳されたメッセージなどを提供するためオペレーティングシステムのロケールの機能を使います。 これは[項22.1](#)と[項22.2](#)内で解説されています。
- 全ての種類の言語によるテキストの格納のサポート、およびクライアントサーバ間の文字セット翻訳の提供を行うため、多くの文字セットを提供します。 これは[項22.3](#)内で解説されています。

22.1. ロケールのサポート

ロケールのサポートはアルファベット、並び換え、数字の書式など文化的嗜好を配慮したあるアプリケーションを対象にします。PostgreSQLは、サーバのオペレーティングシステムが提供する、標準ISO CとPOSIXのロケール機能を使用します。これ以上の情報についてはお使いのシステムのドキュメントを参照ください。

22.1.1. 概要

ロケールのサポートは、`initdb`を使用してデータベースクラスタを作成する時自動的に初期化されます。`initdb`は、デフォルトでその実行環境のロケール設定に従ってデータベースクラスタを初期化します。そのため、システムがデータベースクラスタで使いたいロケールを使用するように既に設定してある場合は何も行う必要はありません。違うロケールを使いたい場合（またはシステムのロケール設定が不明な場合）は、`initdb`の`--locale`オプションで希望のロケールを指定することができます。以下に例を示します。

```
initdb --locale=sv_SE
```

Unixシステム用のこの例の設定はロケールをスウェーデン（`sv`）で使用されているスウェーデン語（`sv`）に合わせています。他にも`en_US`（米国英語）や`fr_CA`（カナダのフランス語）などの設定もできます。ロケールに複数の文字セットが使用可能であれば、`language_territory.codeset`のように記述することができます。例えば、`fr_BE.UTF-8`はベルギー（`BE`）で使われているフランス語（`fr`）でUTF-8の文字セットを表します。

お使いのシステムでどのロケールがどういう名前で使えるかはオペレーティングシステムのベンダがどのようなものを提供しているかと、何がインストールされているかに依存します。ほとんどのUnixシステムでは、`locale -a`というコマンドで利用可能なロケールの一覧を入手することができます。Windowsは、`German_Germany`や`Swedish_Sweden.1252`のようなもっと冗長なロケール名を使用しますが、原理は同じです。

英語の照合規則でスペイン語のメッセージを使用する時など、時として複数のロケールの規則を併用すると便利です。これをサポートするために、ロケールには以下のような多言語対応規則の特定の箇所だけを管理する一連のサブカテゴリがあります。

LC_COLLATE	文字列の並び換え順
LC_CTYPE	文字の分類（文字とはどんなもの？大文字小文字を区別しない？）
LC_MESSAGES	メッセージの言語
LC_MONETARY	通貨書式
LC_NUMERIC	数字の書式
LC_TIME	日付と時刻の書式

これらのカテゴリの名前は、特定のカテゴリについてのロケールの選択を上書きするための`initdb`オプションの名前としてそのまま使用できます。例えば、ロケールをカナダのフランス語に設定しながら通貨書式については米国の規則を使用するには、`initdb --locale=fr_CA --lc-monetary=en_US`とします。

システムがロケールをサポートしていないように動作させたい場合は、特別なロケールの`C`、もしくは同等な`POSIX`を使用してください。

一部のロケールカテゴリでは、その値がデータベース生成時に固定されていなければならないものがあります。他のデータベースで他の設定を使用することができますが、一度データベースが生成されると、そのデータベースでは変更することができません。`LC_COLLATE`と`LC_CTYPE`がこれらのカテゴリにあてはまります。これらはインデックスのソート順に影響を及ぼすため、固定されていなければなりません。さもないと、テキスト型の列上のインデックスは破壊されるでしょう。（しかし[項22.2](#)内で述べられているように、照合を使用することで、この制限を緩和することができます）`initdb`が実行された時に、これらのカテゴリのデフォルト値は決定され、`CREATE DATABASE`コマンドで他を指定しない限り、新しいデータベースが作成されるときにこの値が使用されます。

その他のロケールカテゴリは、いつでも、ロケールカテゴリと同じ名前の実行時パラメータを設定することで、希望値に変更することができます（詳細は[第18章](#)を参照してください）。`initdb`で選択された値は、実際のところ、サーバの起動時にデフォルトとして動作するように`postgresql.conf`設定ファイルに書き込まれるだけです。この代入文を`postgresql.conf`で削除すると、サーバは実行環境の設定をそのまま使用します。

サーバのロケールの動作はどのクライアントの環境にも依存せず、サーバが参照できる環境変数で決まります。ですからサーバを稼働させる前に正しいロケール設定を行うように注意してください。結果としてサーバとクライアントで異なるロケールが設定されていると、メッセージはそれらがどこから生じたかによって、異なる言語で表示されます。

注意: 実行環境のロケールをそのまま使用することは、ほとんどのオペレーティングシステムでは次のような意味を持ちます。指定されたロケールカテゴリ（例えば照合）について、設定するものが見つかるまで、以下の環境変数がこの順番で調べられます。LC_ALL、LC_COLLATE（またはそれぞれのカテゴリに対応する変数）、LANG。これらのいずれの環境変数も設定されない場合に、ロケールはデフォルトでCに設定されます。

メッセージの言語を設定する目的で、メッセージ多言語化ライブラリの中には全てのロケール設定を上書きする環境変数LANGUAGEを検索するものがあります。お使いのシステムでの挙動が不明ならばより詳細な情報を得るためお使いのオペレーティングシステムの文書、特にgettextの文書を参照してください。

ユーザの選択した言語にメッセージを翻訳できるようにするためにはNLSを構築時に有効にする（`configure --enable-nls`）必要があります。他のロケールサポートはすべて自動的に構築されます。

22.1.2. 動作

ロケールの設定は以下のSQL機能に影響を与えます。

- 文字列データに対するORDER BYまたは標準の比較演算子を使用した問い合わせにおける並び替え順
- upper、lower、initcap関数
- (LIKE、SIMILAR TOやPOSIX形式の正規表現といった)パターンマッチング演算子では ロケールは大文字、小文字を区別せず正規表現の文字クラスによる文字の区別に影響を及ぼします。
- 一群のto_char関数
- LIKE節が付いたインデックスを使用する性能

CやPOSIX以外で、PostgreSQLでロケールを使用する際の欠点は実行速度です。ロケールは文字の扱いを遅くし、さらにLIKEで通常のインデックスが使用されなくなります。この理由から、本当に必要な時のみロケールを使用してください。

C以外のロケールにおいて、PostgreSQLがLIKE句を持つインデックスを使用できるようにする回避方法として、いくつかのカスタム演算子クラスがあります。これらを用いると、文字と文字を厳密に比較するようなインデックスや、ロケールの比較規則を無視するようなインデックスを作成できます。詳細は[項11.9](#)を参照してください。もうひとつの方法は、[項22.2](#)内で解説されているようなC照合を使用してインデックスを作成することです。

22.1.3. 問題点

上記の説明に従ってロケールのサポートが正常に動作しない場合、オペレーティングシステムのロケールサポートが正確に設定されているか確認してください。指定されたロケールがインストールされているかどうか確認するために、オペレーティングシステムが提供していれば、`locale -a`コマンドを使用することができます。

PostgreSQLが想定しているロケールを実際に使用しているかどうかを確認してください。LC_COLLATEとLC_CTYPEの設定はデータベース作成時に決定され、新しいデータベースを作成する方法以外に変更することはできません。

LC_MESSAGESやLC_MONETARYなど他のロケール設定はサーバ起動時の環境変数によって初めに決定されますが、その場で変更することができます。SHOWコマンドを使用して、使用中のロケール設定を確認することができます。

ソース配布物のsrc/test/localeディレクトリには、PostgreSQLのロケールサポート用の試験一式があります。

エラーメッセージ内のテキストを解析してサーバ側のエラーを扱っているクライアントアプリケーションでは、サーバのメッセージが異なる言語で記載されると、明らかに問題になります。こうしたアプリケーションの作者には、エラーコードスキーマで代替させることを推奨します。

メッセージ翻訳のカタログのメンテナンスにはPostgreSQLに選択した言語を話させてみたいという数多くのボランティアのたゆみのない努力を必要としています。もしあなたの言語が現在使えなかったり完全に翻訳されていない場合、助力をよろしく願います。もし助力頂けるのであれば、[第48章](#)を参照するか開発グループのメーリングリストに投稿してください。

[前のページ](#)
多言語対応

[ホーム](#)
[上に戻る](#)

[次のページ](#)
照合サポート

22.2. 照合サポート

照合機能は、ソート順番と列ごともしくは操作ごとのデータの文字区別の振る舞いを指定することを許可します。これは、作成後のデータベースのLC_COLLATEとLC_CTYPEの設定が変更できない制限を緩和します。

22.2.1. 概念

概念的に照合可能なデータ型のそれぞれの式は、照合を保持しています (ビルトインされている照合可能なデータ型はtext、varchar、charです。ユーザ定義の基本型は照合可能とマーキングできます。もちろん照合可能なデータ型上のドメインは照合可能となります)。もし、式が列参照である場合は、式の照合は列の定義された照合となります。もし、式が定数である場合は、照合は定数のデータ型のデフォルトの照合となります。より複雑な式の照合は、下記に示すように、その入力からの照合から引き出されます。

式の照合は、"default"照合となります。これはデータベースに対して定義されたロケール設定を意味しています。式の照合は非決定となることもあります。そのような場合に、照合が必要となるような順序操作や他の操作は失敗するでしょう。

データベースシステムが並び替えや文字区別を行う場合、データベースは入力の照合を使用します。これは、たとえばORDER BY句や<演算子や関数を使用する際に発生します。ORDER BY句に適用する照合は、単純にソートキーの照合です。関数や演算子の呼び出しに対して適用される照合は、以下に述べるように引数により決まります。比較演算子に加えて、照合はlower、upper、initcapといった小文字と大文字を変換する関数やパターンマッチングの演算子、to_char関連の関数で考慮されています。

関数や演算子の呼び出しに対して、引数の照合検査により得られた照合は実行時に特定の操作を行うために使用されます。もし関数や演算子の呼び出しの結果が照合可能なデータ型であった場合、照合は関数もしくは演算子式の定義済みの照合として解析時にも試用されます。このとき照合の知識が必要となるような囲み式があります。

式の照合の導出は暗黙でも明示的にでも可能です。この区別は、複数の異なる照合が式中に現れるときに照合がどのように組み合わせられるか、に影響を与えます。明示的な照合の導出は、COLLATE句が使用されたときに発生します。他の全ての照合は暗黙となります。例えば関数呼び出しの中では、次の規則が用いられます。

- 1 11 入力式に明示的な照合の導出がある場合、入力式の中の明示的に導出された全ての照合は同一でなくてはなりません。そうでない場合はエラーが発生します。もし明示的に導出された照合がある場合は、それは照合の組み合わせの結果となります。
- 1 11 そうでない場合は、全ての入力式は同一の暗黙の照合の導出またはデフォルトの照合を持たなくてはなりません。もしデフォルトではない照合がある場合は、それは照合の組み合わせの結果となります。もしそうでない場合は、結果はデフォルトの照合となります。
- 1 11 入力式内でデフォルトではない暗黙の照合が衝突している場合、決定不能な照合であるとみなされます。これは、もし呼び出された特定の関数が適用すべき照合を知っておく必要がないかぎりエラーの条件ではありません。もし知っておく必要がある場合は、実行時にエラーとなります。

例えば、このテーブル定義を考えてみます。

```
CREATE TABLE test1 (
    a text COLLATE "de_DE",
    b text COLLATE "es_ES",
    ...
);
```

このとき

```
SELECT a < 'foo' FROM test1;
```

<の比較はde_DEの規則により実行されます。というのも式は暗黙的に導出されたデフォルトの照合と組み合わせます。しかし、

```
SELECT a < ('foo' COLLATE "fr_FR") FROM test1;
```

このとき比較は、明示的な照合の導出は暗黙の照合をオーバライドするためfr_FR規則が用いられます。さらに、次の例では

```
SELECT a < b FROM test1;
```

パーサはどの照合を適用するか決定できません。というのもaとb列は暗黙の衝突する照合を持つためです。<演算子がどちらの照合を使用するか知る必要があるため、これはエラーとなります。

```
SELECT a < b COLLATE "de_DE" FROM test1;
```


もしくは同じく

```
SELECT a COLLATE "de_DE" < b FROM test1;
```

一方で、以下のように構造的に似たケースとして

```
SELECT a || b FROM test1;
```

これはエラーとなりません。というのも|| 演算子は、照合には関係がないためです。 この結果は照合とは関係なく同じになります。

もし関数や演算子が照合可能なデータ型の結果を出力する場合は、関数に割り当てられた照合、もしくは演算子の組み合わせられた入力式は、関数もしくは演算子の結果に対しても 適用されると考えられます。よって、以下の例では

```
SELECT * FROM test1 ORDER BY a || 'foo';
```

順序はde_DE規則に基づき実行されますが、以下のクエリでは

```
SELECT * FROM test1 ORDER BY a || b;
```

エラーとなります。というのも|| 演算子が照合を知る必要がなかったとしても ORDER BY句が照合を知る必要があるためです。以前と同様、この衝突は明示的に照合を指定することにより解決できます。

```
SELECT * FROM test1 ORDER BY a || b COLLATE "fr_FR";
```

22.2.2. 照合の管理

照合は、SQL名称とオペレーティングシステムのロケールをマッピングするSQLスキーマオブジェクトです。 特に照合はLC_COLLATEとLC_CTYPEの組み合わせにマッピングします。(名称から推測されるように、照合の主な目的はソート順番を制御するLC_COLLATEを設定することです。 しかし実際にはLC_CTYPEの設定をLC_COLLATEと異なるようにする必要はほとんどありません。そのため、式ごとにLC_CTYPEを設定するような別の機構を作成するより、これらの設定を収集する方が、より便利です。) また、照合は文字エンコーディングと結びついています([項22.3](#)を参照下さい)。 同一の照合名称は異なるエンコーディングに対して存在しています。

すべてのプラットフォーム上でdefault、CそしてPOSIXという名称の照合が 利用できます。追加の照合はオペレーティングシステムにより利用可能となります。 default照合は、データベース作成時にLC_COLLATE値とLC_CTYPE値を 選択します。CとPOSIX照合は共に"traditional C"の動作を指定します。 これはASCII文字の"A"から"Z"を文字として扱い、文字コードのバイト値により 厳密にソートします。

もし、オペレーティングシステムが単一のプログラム内(newlocaleや関連する関数)で複数のロケールを使用することをサポートしている場合、 データベースクラスタが初期化されるとinitdbは、オペレーティングシステム上で見つけた全てのロケールに基づく照合を システムカタログのpg_collationに書き込みます。 例えば、オペレーティングシステムがde_DE.utf8の名称のロケールを提供した場合、 initdbは、de_DE.utf8に設定されたLC_COLLATEとLC_CTYPEの両方を持つUTF8エンコーディングのde_DE.utf8という名称の照合を作成します。 同時に照合の名称から.utf8を削除した照合も作成します。これは手間を省き、名称がエンコーディングに依存しないようになります。 いうまでもなく、照合名称の初期値はプラットフォーム依存となることに気をつけてください。

LC_COLLATEとLC_CTYPEに対して異なる値が必要な照合の場合のために、 新規の照合がCREATE COLLATIONコマンドにより作成されます。 このコマンドは、既存の照合から新規の照合を作成するためにも使用されます。これはアプリケーションにおいてオペレーティングシステムに依存しない照合名を使用でき、便利です。

特定のデータベース内でデータベースのエンコーディングを使用している照合のみが興味の対象となります。 pg_collation内の他のエントリは無視されます。 よってde_DEといったようなエンコーディング名が省かれた照合は、包括的に一意でなくてもデータベース内で一意であるとみなされます。 データベースのエンコーディングを変更するときに、変更すべきものを1つ減らすことになるため、エンコーディング名が省かれた照合を使用するようにしてください。 default、CそしてPOSIX照合は、データベースのエンコーディングに関係なく使用可能です。

PostgreSQLは、異なる照合オブジェクトは、それらが同じプロパティを持っていたも 互換性がないものとみなします。例えば、

```
SELECT a COLLATE "C" < b COLLATE "POSIX" FROM test1;
```

は、CとPOSIX照合が同じ動作であってもエラーとなります。 よってエンコーディング名が省かれた照合を混ぜて使用することは推奨されません。

22.3. 文字セットサポート

PostgreSQLの文字セット（エンコーディングとも呼ばれます）サポートにより、ISO 8859シリーズなどのシングルバイト文字やEUC（拡張Unixコード）、UTF-8、Mule内部コードなどのマルチバイト文字を含む、各種文字セットでテキストを保存することができます。全ての文字セットはクライアントにより透過的に使用することができますが、いくつかは、サーバ内での（つまりサーバサイドエンコーディングとして）使用はサポートされていません。デフォルトの文字セットは、`initdb`を使用したPostgreSQLデータベースクラスタの初期化時に決定されます。これは、データベースを作成する時に上書きすることができるので、異なる文字セットを使用した複数のデータベースを持つことができます。

しかし重要な制限は、それぞれのデータベースの文字セットがサーバの`LC_CTYPE`（文字分類）および`LC_COLLATE`（文字列並び替え順序）ロケール設定と互換性がなくてはなりません。Cもしくは`POSIX`ロケール設定の場合、どのような文字セットも許可されています。しかし、他のロケール設定の場合、正しく動作する文字セットはひとつだけとなります。（しかしWindowsではUTF-8符号化方式をどのロケールでも使用できます。）

22.3.1. サポートされる文字セット

PostgreSQLで利用できる文字セットを[表22-1](#)に示します。

表 22-1. PostgreSQL文字セット

名前	説明	言語	サーバ?	バイト数/文字	別名
BIG5	Big Five	繁体字	いいえ	1-2	WIN950、Windows950
EUC_CN	Extended UNIX Code-CN	簡体字	はい	1-3	
EUC_JP	Extended UNIX Code-JP	日本語	はい	1-3	
EUC_JIS_2004	Extended UNIX Code-JP, JIS X 0213	日本語	はい	1-3	
EUC_KR	Extended UNIX Code-KR	韓国語	はい	1-3	
EUC_TW	Extended UNIX Code-TW	繁体字、台湾語	はい	1-3	
GB18030	National Standard	中国語	いいえ	1-2	
GBK	Extended National Standard	簡体字	いいえ	1-2	WIN936、Windows936
ISO_8859_5	ISO 8859-5、ECMA 113	ラテン/キリル	はい	1	
ISO_8859_6	ISO 8859-6、ECMA 114	ラテン/アラビア語	はい	1	
ISO_8859_7	ISO 8859-7、ECMA 118	ラテン/ギリシャ語	はい	1	
ISO_8859_8	ISO 8859-8、ECMA 121	ラテン/ヘブライ語	はい	1	
JOHAB	JOHAB	韓国語（ハングル）	いいえ	1-3	
KOI8R	KOI8-R	キリル文字（ロシア）	はい	1	KOI8
KOI8U	KOI8-U	キリル文字（ウクライナ）	はい	1	
LATIN1	ISO 8859-1、ECMA 94	西ヨーロッパ	はい	1	ISO88591

LATIN2	ISO 8859-2、ECMA 94	中央ヨーロッパ	はい	1	ISO88592
LATIN3	ISO 8859-3、ECMA 94	南ヨーロッパ	はい	1	ISO88593
LATIN4	ISO 8859-4、ECMA 94	北ヨーロッパ	はい	1	ISO88594
LATIN5	ISO 8859-9、ECMA 128	トルコ	はい	1	ISO88599
LATIN6	ISO 8859-10、ECMA 144	北欧	はい	1	ISO885910
LATIN7	ISO 8859-13	バルト語派	はい	1	ISO885913
LATIN8	ISO 8859-14	ケルト	はい	1	ISO885914
LATIN9	ISO 8859-15	LATIN1でヨーロッパと訛りを含む	はい	1	ISO885915
LATIN10	ISO 8859-16、ASRO SR 14111	ルーマニア	はい	1	ISO885916
MULE_INTERNAL	Mule内部コード	多言語Emacs	はい	1-4	
SJIS	Shift JIS	日本語	いいえ	1-2	Mskanji、ShiftJIS、WIN932、Windows932
SHIFT_JIS_2004	Shift JIS, JIS X 0213	日本語	いいえ	1-2	
SQL_ASCII	未指定（テキストを参照）	何でも	はい	1	
UHC	統合ハングルコード	韓国語	いいえ	1-2	WIN949、Windows949
UTF8	Unicode、8ビット	すべて	はい	1-4	Unicode
WIN866	Windows CP866	キリル文字	はい	1	ALT
WIN874	Windows CP874	タイ語	はい	1	
WIN1250	Windows CP1250	中央ヨーロッパ	はい	1	
WIN1251	Windows CP1251	キリル文字	はい	1	WIN
WIN1252	Windows CP1252	西ヨーロッパ	はい	1	
WIN1253	Windows CP1253	ギリシャ	はい	1	
WIN1254	Windows CP1254	トルコ	はい	1	
WIN1255	Windows CP1255	ヘブライ	はい	1	
WIN1256	Windows CP1256	アラビア語	はい	1	
WIN1257	Windows CP1257	バルト語派	はい	1	
WIN1258	Windows CP1258	ベトナム語	はい	1	ABC、TCVN、TCVN5712、VSCII

全てのクライアントのAPIが上の一覧表に示した文字セットをサポートしているわけではありません。例えばPostgreSQL JDBCドライバはMULE_INTERNAL、LATIN6、LATIN8、そしてLATIN10をサポートしません。

SQL_ASCIIの設定は、他の設定とかなり異なります。サーバのキャラクタセットがSQL_ASCIIのとき、サーバは0から127のバイト値をASCIIに変換します。一方、128から255までは変換されません。設定がSQL_ASCIIの場合は、符号化は実行されません。よって、この設定は特定の符号化を使用している場合には、その符号化を無視するようになってしまいます。多くの場合、ASCIIではない環境で作業する場合はSQL_ASCIIの設定を使用するのは、賢いことではありません。なぜならPostgreSQLはASCIIではない文字を変換したり検査したりすることは出来ないからです。

22.3.2. 文字セットの設定

initdbでPostgreSQLクラスタのデフォルト文字セット(エンコーディング)を定義します。以下に例を示します。

```
initdb -E EUC_JP
```

これはデフォルトの文字セットをEUC_JP（日本語拡張Unixコード）に設定します。より長いオプションの文字列がお好みなら-Eの代わりに--encodingと書くこともできます。-Eオプションも--encodingオプションも与えられない場合、initdbは、指定もしくはデフォルトのロケールに基づいて適当な符号化方式を決定しようとします。

データベース作成時に選択したロケールと互換性を持つ符号化方式を提供することで、デフォルト以外の符号化方式を指定することができます。

```
createdb -E EUC_KR -T template0 --lc-collate=ko_KR.euckr --lc-ctype=ko_KR.euckr korean
```

これはEUC_KR文字セットとko_KRロケールを使用するkoreanという名前のデータベースを作成します。 SQLコマンドで同じことを行うには次のようにします。

```
CREATE DATABASE korean WITH ENCODING 'EUC_KR' LC_COLLATE='ko_KR.euckr' LC_CTYPE='ko_KR.euckr'
TEMPLATE=template0;
```

上のコマンドにてtemplate0データベースのコピーが指定されていることに注目してください。 他のデータベースからコピーする場合、データが破損する結果となる可能性がありますので、符号化方式とロケール設定を元のデータベースの設定から変更することはできません。 詳細については[項21.3](#)を参照してください。

データベースの符号化方式はpg_databaseシステムカタログに格納されます。 psqlの-lオプションか\lコマンドで符号化方式を確認することができます。

```
$ psql -l
```

List of databases					
Name	Owner	Encoding	Collation	Ctype	Access Privileges
cloaledb	hlinnaka	SQL_ASCII	C	C	
englishdb	hlinnaka	UTF8	en_GB.UTF8	en_GB.UTF8	
japanese	hlinnaka	UTF8	ja_JP.UTF8	ja_JP.UTF8	
korean	hlinnaka	EUC_KR	ko_KR.euckr	ko_KR.euckr	
postgres	hlinnaka	UTF8	fi_FI.UTF8	fi_FI.UTF8	
template0	hlinnaka	UTF8	fi_FI.UTF8	fi_FI.UTF8	{=c/hlinnaka,hlinnaka=CTc/hlinnaka}
template1	hlinnaka	UTF8	fi_FI.UTF8	fi_FI.UTF8	{=c/hlinnaka,hlinnaka=CTc/hlinnaka}

(7 rows)

重要項目: 最近のオペレーティングシステムでは、PostgreSQLは、LC_CTYPEの設定によりどの文字セットが指定されているか決定できます。 そして、一致するデータベース符号化方式のみを強制的に使用します。 古いオペレーティングシステムでは、自分で選択したロケールが想定している符号化方式を確実に使用することは各自の責任になります。ここでの間違いは、ソート処理などのロケールに依存する操作が、奇妙な動作するといったことにつながります。

PostgreSQLは、LC_CTYPEがCもしくはPOSIXでもない場合にも、スーパーユーザがSQL_ASCIIエンコーディングでデータベースを作成することを許可します。 上記のように、SQL_ASCIIは、データベースに保存されているデータが特定のエンコーディングを持つことを強制しません。さらに、この選択はロケールに依存したおかしい動作を引き起こすリスクを高めます。この設定の組み合わせを使用することは、お勧めできませんし、いつの日か完全に禁止されるかもしれません。

22.3.3. サーバ・クライアント間の自動文字セット変換

PostgreSQLは、ある文字セットの組み合わせに対してサーバとクライアントの間で自動的に文字セットを変換する機能を提供しています。 変換情報はpg_conversionシステムカタログに格納されています。 PostgreSQLには、[表22-2](#)で示されているように、事前に定義された変換が付属します。 新しい変換を作成するにはSQLコマンドのCREATE CONVERSIONを使用します。

表 22-2. クライアント・サーバ文字セット変換

サーバ文字 セット	利用可能なクライアント文字セット
BIG5	サーバの符号化方式としてはサポートしていません。
EUC_CN	EUC_CN、 MULE_INTERNAL、 UTF8
EUC_JP	EUC_JP、 MULE_INTERNAL、 SJIS、 UTF8
EUC_KR	EUC_KR、 MULE_INTERNAL、 UTF8
EUC_TW	EUC_TW、 BIG5、 MULE_INTERNAL、 UTF8
GB18030	サーバの符号化方式としてはサポートしていません。
GBK	サーバの符号化方式としてはサポートしていません。
ISO_8859_5	ISO_8859_5、 KOI8、 MULE_INTERNAL、 UTF8、 WIN866、 WIN1251
ISO_8859_6	ISO_8859_6、 UTF8
ISO_8859_7	ISO_8859_7、 UTF8
ISO_8859_8	ISO_8859_8、 UTF8
JOHAB	JOHAB、 UTF8
KOI8R	KOI8R、 ISO_8859_5、 MULE_INTERNAL、 UTF8、 WIN866、 WIN1251

KOI8U	<i>KOI8U</i> 、 UTF8
LATIN1	<i>LATIN1</i> 、 MULE_INTERNAL、 UTF8
LATIN2	<i>LATIN2</i> 、 MULE_INTERNAL、 UTF8、 WIN1250
LATIN3	<i>LATIN3</i> 、 MULE_INTERNAL、 UTF8
LATIN4	<i>LATIN4</i> 、 MULE_INTERNAL、 UTF8
LATIN5	<i>LATIN5</i> 、 UTF8
LATIN6	<i>LATIN6</i> 、 UTF8
LATIN7	<i>LATIN7</i> 、 UTF8
LATIN8	<i>LATIN8</i> 、 UTF8
LATIN9	<i>LATIN9</i> 、 UTF8
LATIN10	<i>LATIN10</i> 、 UTF8
MULE_INTERNAL	<i>MULE_INTERNAL</i> 、 BIG5、 EUC_CN、 EUC_JP、 EUC_KR、 EUC_TW、 ISO_8859_5、 KOI8R、 LATIN1 to LATIN4、 SJIS、 WIN866、 WIN1250、 WIN1251
SJIS	サーバの符号化方式としてはサポートしていません。
SQL_ASCII	どれでも （変換されません）
UHC	サーバの符号化方式としてはサポートしていません。
UTF8	すべてサポートされています。
WIN866	<i>WIN866</i> 、 ISO_8859_5、 KOI8R、 MULE_INTERNAL、 UTF8、 WIN1251
WIN874	<i>WIN874</i> 、 UTF8
WIN1250	<i>WIN1250</i> 、 LATIN2、 MULE_INTERNAL、 UTF8
WIN1251	<i>WIN1251</i> 、 ISO_8859_5、 KOI8R、 MULE_INTERNAL、 UTF8、 WIN866
WIN1252	<i>WIN1252</i> 、 UTF8
WIN1253	<i>WIN1253</i> 、 UTF8
WIN1254	<i>WIN1254</i> 、 UTF8
WIN1255	<i>WIN1255</i> 、 UTF8
WIN1256	<i>WIN1256</i> 、 UTF8
WIN1257	<i>WIN1257</i> 、 UTF8
WIN1258	<i>WIN1258</i> 、 UTF8

自動文字セット変換を有効にするためには、クライアントでどのような文字セット（符号化方式）を使用させたいかをPostgreSQLに伝えなければなりません。 これを行うにはいくつかの方法があります。

- psqlで¥encodingコマンドを使います。 ¥encodingは実行中であってもクライアントの符号化方式を変更させることができます。 例えば符号化方式をSJISに変えたい場合は次のように入力します。

```
¥encoding SJIS
```

- libpq ([項31.10](#))はクライアントの符号化方式を制御する関数を保持しています。
- SET client_encoding TOを使います。 次のSQLコマンドでクライアントの符号化方式を設定できます。

```
SET CLIENT_ENCODING TO 'value';
```

標準SQLの構文SET NAMESを同じ目的で使うこともできます。

```
SET NAMES 'value';
```

現在のクライアントの符号化方式を問い合わせるには次のようにします。

```
SHOW client_encoding;
```

デフォルトの符号化方式に戻すのには次のようにします。

```
RESET client_encoding;
```

- PGCLIENTENCODINGを使います。 クライアントの環境でPGCLIENTENCODING環境変数が定義されていると、サーバと接続が確立した時点で自動的にクライアントの符号化方式が選択されます （上で説明したその他のどんな方法でもその後書き換えできます）。

- [client_encoding](#)変数を使います。 `client_encoding`変数が設定されていると、サーバとの接続が確立した時点で自動的にクライアントの符号化方式が選択されます（上で説明したその他のどんな方法でもその後書き換えできます）。

`EUC_JP`をサーバに、そして`LATIN1`をクライアントに選んだ場合のように、特定の文字の変換ができない時、日本語文字は`LATIN1`に入っていないという旨の日本語が返され、エラーが報告されます。

クライアント側のキャラクタセットが`SQL_ASCII`に定義されている場合は、符号化変換はサーバ側のキャラクタセットに関係無く無効化されます。サーバ側と同じように、`SQL_ASCII`を使用することは、すべてASCIIのデータを扱っている場合を除き、賢い方法ではありません。

22.3.4. 推奨文書

ここに記したものは様々な符号化方式システムを学習するのに良い資料です。

<http://www.i18ngurus.com/docs/984813247.html>

文字セット、エンコーディングについてのドキュメントの集まりです。

CJKV日中韓越情報処理: 中国語、日本語、韓国語 & ベトナム語処理

`EUC_JP`、`EUC_CN`、`EUC_KR`、`EUC_TW`の詳しい説明があります。

<http://www.unicode.org/>

Unicode協会のWebサイトです。

RFC 3629

ここでUTF-8（8ビットUCS/Unicode変換書式）が定義されています。

[前のページ](#)
照合サポート

[ホーム](#)
[上に戻る](#)

[次のページ](#)
定常的なデータベース保守作業

第 23章 定常的なデータベース保守作業

目次

23.1. [定常的なバキューム作業](#)

- 23.1.1. [バキューム作業の基本](#)
- 23.1.2. [ディスク容量の復旧](#)
- 23.1.3. [プランナ用の統計情報の更新](#)
- 23.1.4. [可視性マップの更新](#)
- 23.1.5. [トランザクションIDの周回エラーの防止](#)
- 23.1.6. [自動バキュームデーモン](#)

23.2. [定常的なインデックスの再作成](#)

23.3. [ログファイルの保守](#)

他のデータベースソフトウェア同様、PostgreSQLも、最適な性能を得るために定常的に実施しなければならない作業があります。ここで説明する作業は必要なものであり、その性質上繰り返し行うべきものです。しかし、cronスクリプトなどの標準ツールや、Windowsのタスクスケジューラを使用して簡単に自動化することができます。適切なスクリプトを設定し、その実行がうまく行くかどうかを点検することは、データベース管理者の責任です。

明らかに必要な保守作業の1つに、定期的なデータのバックアップコピーの作成があります。最近のバックアップがなければ、（ディスクの破損、火災、重要なテーブルの間違った削除などの）破滅の後、復旧することができません。PostgreSQLで可能なバックアップとリカバリ機構については、[第24章](#)にて詳細に説明します。

他の保守作業の主なカテゴリには、定期的なデータベースの"バキューム"があります。この作業については[項23.1](#)で説明します。問い合わせプランナで使用される統計情報の更新も密接に関連しますが、こちらに関しては[項23.1.3](#)で説明します。

その他、定期的に行わなければならない作業にログファイルの管理があります。これについては[項23.3](#)で説明します。

[check_postgres](#)が、データベースの健全性を監視し、異常な状態を報告するために用意されています。[check_postgres](#)はNagiosおよびMRTGに組み込まれたものですが、独立して実行させることができます。

PostgreSQLは他のデータベース管理システムに比べ、保守作業は少ないと言えます。それでもなお、これらの作業に適切に注意することは、システムに対する快適かつ充実した経験を確実に得るのに効果があります。

23.1. 定期的なバキューム作業

PostgreSQLデータベースはバキューム処理として知られている定期的な保守を必要とします。多くのインストレーションでは、[項23.1.6](#)で説明されている自動バキュームデーモンでのバキューム処理を行わせることで充分です。それぞれの状況に合った最善の結果を得るため、そこで説明する自動バキューム用パラメータの調整が必要かもしれません。データベース管理者によっては、cronもしくはタスクスケジューラスクリプトに従って典型的に実行される、手作業管理のVACUUMコマンドによりデーモンの活動を補足したり、置き換えたりすることを意図するかもしれません。手作業管理のバキューム処理を適切に設定するためには、以下のいくつかの小節で説明する問題点を理解することが必須です。自動バキューム処理に信頼をおいている管理者にとっても、この資料に目を通すことはそれらの理解と自動バキューム処理の調整に役に立つことでしょう。

23.1.1. バキューム作業の基本

PostgreSQLのVACUUMコマンドは以下の理由により定期的にそれぞれのテーブルを処理しなければなりません。

- 1 11 更新、あるいは削除された行によって占められたディスク領域の復旧または再利用。
- 1 11 PostgreSQL問い合わせプランナによって使用されるデータ統計情報の更新。
- 1 11 インデックスオンリースキャンを高速にする、可視性マップの更新。
- 1 11 トランザクションIDの周回による非常に古いデータの損失を防止。

以降の小節で説明するように、これらの理由の1つ1つはVACUUM操作の実行について、その頻度の変動や対象領域の変動に影響します。

VACUUMには、標準VACUUMとVACUUM FULLという2つの種類があります。VACUUM FULLはより多くのディスク容量を回収することができますが、実行にとても時間がかかります。また、VACUUMの標準形式は実運用のデータベースに対する操作と同時に実行させることができます。(SELECT、INSERT、UPDATE、DELETEなどのコマンドは通常通りに動作し続けます。しかし、バキューム処理中はALTER TABLE ADD COLUMNなどのコマンドを使用してテーブル定義を変更することはできません。) VACUUM FULLはそれが作用する全てのテーブルに対し排他ロックを必要とするので、それらテーブルのその他の用途と並行して行うことはできません。一般的に、管理者は標準VACUUMの使用に努め、VACUUM FULLの使用を避けるべきです。

VACUUMは、かなりの量のI/Oトラフィックを発生させます。このため、他の実行中のセッションの性能を劣化させる可能性があります。バックグラウンドで実行されるバキューム処理による性能への影響を軽減させることを調整できるような設定パラメータがあります。[項18.4.4](#)を参照してください。

23.1.2. ディスク容量の復旧

PostgreSQLでは、行のUPDATEもしくはDELETEは古い行を即座に削除しません。この方法は、多版同時性制御(MVCC。[第13章](#)を参照してください)の恩恵を受けるために必要なものです。あるバージョンの行は他のトランザクションから参照される可能性がある場合は削除されてはなりません。しかし最終的には、更新された前の行や削除された行を参照するトランザクションはなくなります。必要なディスク容量が無制限に増加しないように、これらが占める領域は、新しい行で再利用できるように回収されなければなりません。これはVACUUMを実行することで行われます。

標準形式のVACUUMは、テーブルとインデックス内の不要な行を削除し、その領域を将来の再利用が可能であるものとして記録します。しかし、その領域をオペレーティングシステムに返却することはありません。例外として、テーブルの末尾に完全に空のページが存在し、かつそのテーブルの排他ロックが容易に獲得できるような特殊な場合には、その領域を返却します。対照的にVACUUM FULLは、不要な領域のない全く新しいバージョンのテーブルファイルを書き出すことで、積極的にテーブルを圧縮します。テーブルの容量を最小化しますが、長い時間がかかる可能性があります。また操作が終わるまで、テーブルの新しいコピー用に余計なディスク領域を必要とします。

定期的なバキューム作業の通例の目安はVACUUM FULLの必要性を避けるに充分な頻度で標準VACUUMを行うことです。自動バキュームデーモンはこうにして作動を試みます。そして実際VACUUM FULLを行いません。この手法において、その発想はテーブルを最小サイズに保つのではなく、ディスク領域使用の安定状態を保持することです。それぞれのテーブルは、その最小サイズにバキューム作業とバキューム作業の間で使用されることになる容量を加えたのに等しい空間を占有します。VACUUM FULLは、テーブルをその最小サイズまで縮小し、ディスク空間をオペレーティングシステムに返却するために使用することができますが、もし将来そのテーブルが再び肥大化するのであれば、大した意味がありません。従って、程よい頻度の標準VACUUMを実行するほうが、不定期のVACUUM FULLを実行するより大量の更新テーブルを保守するにはより良い取り組みとなります。

例えば負荷が少ない夜間に全ての作業を行うように、一部の管理者は自身で計画したバキューム作業の方を選びます。固定したスケジュールに従ってバキューム作業を行うことについての問題は、もし更新作業によりテーブルが予期せぬ急増に遭遇した場合、空き領域を回収するためにVACUUM FULLが本当に必要となるとところまで肥大化することです。自動バキュームデーモンを使

用することにより、この問題は緩和されます。なぜなら、このデーモンは更新作業に反応して動的にバキューム作業を計画するからです。完全に作業量を予測することができない限り、デーモンを完全に無効化するのは勧められません。取り得る妥協案の1つは、いつになく激しい更新作業にのみ反応するよう、デーモンのパラメータを設定することです。これにより、抑制可能な範囲を維持しつつ、負荷が標準的な場合に計画化されたVACUUMがまとめて作業を行うことを想定することができます。

自動バキュームを使用しない場合の典型的な方式は、データベース全体のVACUUMを1日1回使用頻度が低い時間帯にスケジュールすることです。必要に応じて、更新頻度の激しいテーブルのバキューム処理をより頻繁に行うよう追加してください。（非常に高い頻度でデータの更新を行うインストレーションの中では、分間隔位という頻度で高負荷なテーブルのVACUUMを行うこともあります。）1つのクラスタで複数のデータベースがある場合、それぞれをバキュームすることを忘れないでください。[vacuumdb](#)プログラムが役に立つかもしれません。

ティップ: 大規模な更新や削除作業の結果としてテーブルが不要な行バージョンを大量に含む場合、通常のVACUUMでは満足のゆくものではないかもしれません。もしそのようなテーブルを所有し、それが占有する余分なディスク空間の回収が必要であれば、VACUUM FULL、またはその代わりに[CLUSTER](#)やテーブルを書き換える[ALTER TABLE](#)構文の1つを使用しなければなりません。これらのコマンドはテーブル全体を新しいコピーに書き換え、それに対する新規インデックスを作成します。これらの選択肢はすべて排他ロックを必要とします。新しいものが完成するまで、テーブルの旧コピーとインデックスは解放されませんので、元のテーブルと同程度の容量の余計なディスク領域も一時的に使用することに注意してください。

ティップ: テーブルの内容が定期的に完全に削除される場合、DELETEの後にVACUUMを使用するよりも、[TRUNCATE](#)を使用する方が良いでしょう。TRUNCATEはテーブルの全ての内容を即座に削除します。また、その後に不要となったディスク容量を回収するためにVACUUMやVACUUM FULLを行う必要がありません。不利な点は厳格なMVCC動作が違反となることです。

23.1.3. プランナ用の統計情報の更新

PostgreSQL問い合わせプランナは、優れた問い合わせ計画を作成するのに、テーブルの内容に関する統計情報に依存しています。この統計情報は[ANALYZE](#)によって収集されます。このコマンドはそのものを呼び出す以外にも、VACUUMのオプション処理としても呼び出すことができます。合理的な精度の統計情報を持つことは重要です。さもないと非効率的な計画を選択してしまい、データベースの性能を悪化させてしまいます。

自動バキュームデーモンが有効になっている場合は、テーブルの内容が大きく変更されたときはいつでも自動的にANALYZEコマンドを発行します。しかし、特にテーブルの更新作業が"興味のある"列の統計情報に影響を与えないことが判っている時、手作業により計画されたANALYZE操作を当てにする方が好ましいと管理者は思うかもしれません。デーモンは、挿入または更新された行数の関数としてANALYZEを厳密に計画します。しかし、意味のある統計情報の変更につながるかどうかは判りません。

領域復旧のためのバキューム処理と同様、頻繁な統計情報の更新は、滅多に更新されないテーブルよりも更新の激しいテーブルにとってより有益です。しかし、頻繁に更新されるテーブルであっても、データの統計的な分布が大きく変更されなければ、統計情報を更新する必要はありません。単純な鉄則は、テーブル内の列の最小値、最大値にどのくらいの変化があったかを考えることです。例えば、行の更新時刻を保持するtimestamp列の場合、最大値は行が追加、更新されるにつれて、単純に増加します。こういった列は、おそらく、例えば、あるWebサイト上のアクセスされたページのURLを保持する列よりも頻繁に統計情報を更新する必要があるでしょう。このURL列の更新頻度も高いものかもしれませんが、その値の統計的な分布の変更は相対的に見ておそらく低いものです。

特定のテーブルに対してANALYZEを実行することができます。また、テーブルの特定の列のみに対してさえも実行することができます。ですので、アプリケーションの要求に応じて、他よりも頻繁に一部の統計情報を更新できるような柔軟性があります。しかし、実際には、操作が高速であるため、単にデータベース全体を解析することが最善です。ANALYZEは、すべての行を読むのではなく、テーブルから統計的にランダムな行を抽出して使用します。

ティップ: 列単位でのANALYZE実行頻度の調整は非常に実用的とは言えるものではありませんが、ANALYZEで集計される統計情報の詳細レベルの調整を列単位で行うことは価値がある場合があります。WHERE句でよく使用され、データ分布の規則性がほとんどない列は、他の列よりもより細かいデータの度数分布が必要になるでしょう。[ALTER TABLE SET STATISTICS](#)を参照するか、[default_statistics_target](#)設定パラメータでデータベース全体のデフォルトを変更してください。

またデフォルトで、関数の選択性に関して利用可能な制限付きの情報がります。しかし、関数呼び出しを使用する式インデックスを作成する場合、有用な統計情報が関数に関して収集されます。これにより式インデックスを使用する問い合わせ計画を大きく改良することができます。

ティップ: 自動バキュームデーモンは、有益になる頻度を決定する手段がありませんので、外部テーブルに対してANALYZEコマンドを発行しません。問い合わせが適切な計画作成のために外部テーブルの統計情報が必要であれば、適当なスケジュールでこれらのテーブルに対して手作業で管理するANALYZEコマンドを実行することを勧めます。

23.1.4. 可視性マップの更新

バキュームは、どのページにすべての有効トランザクション（およびページが再度更新されるまでの将来のトランザクション）で可視であることが分かっているタプルのみが含まれるかを追跡するために、各テーブルの[可視性マップ](#)の保守を行います。2つの目的があります。1つ目はバキューム自身が、整理するものがありませんので、こうしたページを次回飛ばすことができます。

2つ目は、PostgreSQLが、背後にあるテーブルを参照することなく、インデックスのみを使用して一部の問い合わせに応えることができるようになります。PostgreSQLのインデックスにはタプルの可視性に関する情報を持ちませんので、通常のインデックススキャンは合致したインデックス項目のヒープタプルを取り込み、現在のトランザクションから可視であるべきかどうか検査します。一方でインデックスオンリースキャンはまず可視性マップを検査します。そのページのタプルがすべて可視であることが分かれば、ヒープの取り出しを省くことができます。可視性マップがディスクアクセスを防ぐことができるデータ群が大規模である場合、特に顕著です。可視性マップがヒープより非常に小さいため、ヒープが非常に大きい場合であっても簡単にキャッシュすることができます。

23.1.5. トランザクションIDの周回エラーの防止

PostgreSQLのMVCCトランザクションのセマンティックは、トランザクションID (XID) 番号の比較が可能であることに依存しています。現在のトランザクションのXIDよりも新しい挿入時のXIDを持ったバージョンの行は、"未来のもの"であり、現在のトランザクションから可視であってはなりません。しかし、トランザクションIDのサイズには制限（32ビット）があり、長時間（40億トランザクション）稼働しているクラスタはトランザクションの周回を経験します。XIDのカウンタが一周して0に戻り、そして、突然に、過去になされたトランザクションが将来のものに見えるように、つまり、その出力が不可視になります。端的に言うと、破滅的なデータの損失です。（実際はデータは保持されていますが、それを入手することができなければ、慰めにならないでしょう。）これを防ぐためには、すべてのデータベースにあるすべてのテーブルを少なくとも20億トランザクションごとにバキュームする必要があります。

定期的なバキューム処理によりこの問題が解決する理由は、PostgreSQLが特別なXID、FrozenXIDを確保することです。このXIDは通常のXIDの比較規則には従わず、常に全ての通常のXIDよりも古いものとみなされます。通常のXID（2以上の値）は $\text{modulo-}2^{31}$ という数式を使用して比較されます。これは、全ての通常のXIDでは、20億の"より古い"XIDと20億の"より新しい"XIDが存在することを意味します。言い換えると、通常のXID空間は終わることなく循環されているということです。そのため、ある特定のXIDであるバージョンの行を作成すると、そのバージョンの行は、以降の20億トランザクションからはどの通常のXIDについて比較しているのかには関係なく、"過去のもの"と認識されます。そのバージョンの行が20億トランザクション以上後にも存在していた場合、それは突然に未来のものとして認識されます。これを防ぐために、20億トランザクションより古いとみなされるより、少し前に古いバージョンの行のXIDをFrozenXIDに再割り当てする必要があります。この特殊なXIDに割り当てられた後は、周回問題に関係なく、全ての通常のトランザクションから"過去のもの"として認識され、また、そのバージョンの行はどれだけ古いものであろうと、削除されるまで有効状態となります。この古いXIDの再割り当てはVACUUMで扱われます。

`vacuum_freeze_min_age`は、XID値がFrozenXIDで置換される前に、旧XID値がどのくらい経過しているのかを制御します。より大きな`vacuum_freeze_min_age`によりトランザクション情報が長期間保持されます。一方で、この値を小さくすることで、テーブルを次にバキュームする必要が起るまで継続できるトランザクション数を増加させることができます。

VACUUMは通常もはや使用されない行バージョンを持っていないページを読み飛ばしますが、それらのページは旧XID値の行バージョンを未だ所有している可能性があります。全ての旧XIDがFrozenXIDにより置き換えられたかどうかの確認を得るためには、全てのテーブルの走査が必要です。[vacuum_freeze_table_age](#)はVACUUMがいつこれを行うかを制御します。つまり、テーブル全体にわたる清掃は`vacuum_freeze_table_age`から`vacuum_freeze_min_age`を差し引いたトランザクションに対しテーブルが完全に走査されていない場合に強制されます。0に設定することでVACUUMは可視性マップを実質無視し、常に全てのページを走査するようになります。

テーブルをバキュームすることなく処理できる最大の時間は、20億トランザクションからVACUUMが前回テーブル全体を走査した時点の`vacuum_freeze_min_age`の値を差し引いたものです。この時間よりも長期間バキュームを行わないと、データ損失が発生します。これを確実に防止するために、自動バキュームが[autovacuum_freeze_max_age](#)設定パラメータで指定された時代より古いXIDを持つ可能性がある任意のテーブルに対して呼び出されます。（これは自動バキュームが無効であっても起こります。）

これは、あるテーブルがバキュームされていなかったとしても、自動バキュームがおおよそ`autovacuum_freeze_max_age - vacuum_freeze_min_age`トランザクション毎に呼び出されることを意味します。領域確保のために定期的なバキューム処理を行うテーブルでは、これは重要ではありません。しかし、（挿入のみで更新や削除が行われないテーブルを含む）静的なテーブルでは、領域確保のためのバキューム処理を行う必要がなくなりますので、非常に長期間静的なテーブルでは、強制的な自動バキューム間の間隔を最大まで延ばすことができます。記載するまでもありませんが、`autovacuum_freeze_max_age`を増やすことでも`vacuum_freeze_min_age`を減らすことでも、これを行うことができます。

`vacuum_freeze_table_age`に対する有効な最大値は $0.95 * \text{autovacuum\_freeze\_max\_age}$ です。これより値が高いと値は最大値までに制限されます。`autovacuum_freeze_max_age`より高い値は、周回防止用の自動バキュームがその時点でいずれにせよ誘発され、0.95という乗算係数がそれが起る前に手動によるVACUUM実行の余地を残すため、意味を持ちません。経験則に従うと、定期的に計画されたVACUUMもしくは通常の削除・更新作業により誘発された自動バキュームがその期間で実行されるように十分

な間隔を残しておくように、`vacuum_freeze_table_age` はより多少低い値に設定されるべきです。これを余りにも近い値に設定すると、たとえ領域を回収するために最近テーブルがバキュームされたとしても、周回防止用の自動バキュームに帰着します。一方より低い値はより頻繁なテーブル全体の走査を引き起こします。

`autovacuum_freeze_max_age`（およびそれに付随する`vacuum_freeze_table_age`）を増やす唯一の欠点は、データベースクラスタのサブディレクトリ`pg_clog`がより大きな容量となることです。`autovacuum_freeze_max_age`の範囲まですべてのトランザクションのコミット状況を格納しなければならないためです。コミット状況は1トランザクション当たり2ビット使用しますので、もし`autovacuum_freeze_max_age`がその最大許容値である20億に設定している場合、`pg_clog`はおよそ0.5ギガバイトまで膨らむものと考えられます。これがデータベースサイズ全体に対してとるに足らないものであれば、`autovacuum_freeze_max_age`を最大許容値に設定することを勧めます。さもなければ、`pg_clog`の容量として許容できる値に応じてそれらを設定してください。（デフォルトは2億トランザクションです。換算すると`pg_clog`はおよそ50メガバイトの容量となります。）

`vacuum_freeze_min_age`を減らすことにも1つ欠点があります。これにより`VACUUM`が大して役に立たなくなるかもしれません。テーブル行がすぐに変更される場合（新しいXIDを獲得することになります）、テーブル行のXIDをFrozenXIDに変更することは時間の無駄です。そのため、この設定は、行の変更が起こらなくなるまで凍結されない程度に大きくすべきです。この設定を減らすことによる他の欠点として、行を挿入または変更したトランザクションに関する情報がすぐに失われることです。この情報は、特に、データベース障害の後、何がうまくいかなかったのかを解析する場合、便利になることがあります。この2つの理由により、この設定を減らすことは完全に静的なテーブルでない限りお勧めしません。

データベース内のもっとも古いXIDの年代を追跡するために、`VACUUM`はシステムテーブル`pg_class`と`pg_database`にXID統計情報を保持します。特に、テーブルに対応する`pg_class`行の`relfrozenxid`列には、テーブルに対する最後のテーブル全体の`VACUUM`で使用された凍結切捨てXIDが含まれます。この切り捨てXIDよりも古い、全ての通常のXIDはそのテーブルのFrozenXIDによって置換されていることが保証されています。同様に、データベースに対応する`pg_database`行の`datfrozenxid`列は、データベース内で現れる通常のXIDの下限值です。これは、そのデータベース内のテーブル当たりの`relfrozenxid`値の最小値です。この情報を検査する簡便な方法は、以下の問い合わせを実行することです。

```
SELECT relname, age(relfrozenxid) FROM pg_class WHERE relkind = 'r';
SELECT datname, age(datfrozenxid) FROM pg_database;
```

`age`列は切り捨てXIDから現在のトランザクションXIDまでのトランザクション数を測ります。

`VACUUM`は通常最後のバキュームから変更されたページのみ走査しますが、`relfrozenxid`はテーブル全体が走査されたときのみ繰り上がります。テーブル全体は、`relfrozenxid`が`vacuum_freeze_table_age`トランザクション年齢より大きい時、`VACUUM`のFREEZEオプションが使用された時、もしくは使用されない行バージョンを削除するため全てのページをバキュームしなければならなくなった時、走査されます。`VACUUM`がテーブル全体を走査した直後、`age(relfrozenxid)`は、使用された`vacuum_freeze_min_age`設定より若干大きくなるはずですが（`VACUUM`を起動してから始まったトランザクションの数分大きくなります）。もしテーブル全体を走査する`VACUUM`が`autovacuum_freeze_max_age`に達するまでにテーブルに対して発行されない場合、そのテーブルに対して自動バキュームが早急に強制されます。

何らかの理由により自動バキュームがテーブルの古いXIDの整理に失敗した場合、システムはデータベースの最古のXIDが周回ポイントから1000万トランザクションに達した場合と似たような警告メッセージを発行し始めます。

```
WARNING: database "mydb" must be vacuumed within 177009986 transactions
HINT: To avoid a database shutdown, execute a database-wide VACUUM in "mydb".
```

（ヒントで示唆されたように手動`VACUUM`はこの問題を解決します。しかし、`VACUUM`はスーパーユーザで実行されなければなりません。さもないとシステムカタログの処理に失敗し、このためデータベースの`datfrozenxid`を桁上げすることができません。）こうした警告も無視し続け、周回するまでのトランザクションが100万より少なくなると、システムは停止し、新しいトランザクションの起動を拒絶します。

```
ERROR: database is not accepting commands to avoid wraparound data loss in database "mydb"
HINT: Stop the postmaster and use a standalone backend to VACUUM in "mydb".
```

この100万トランザクションという安全マージンは、管理者が必要な`VACUUM`コマンドを手作業で実行することで、データを失うことなくリカバリすることができるようにするために存在します。しかし、システムがこの安全のための停止モードになると、コマンドを実行しませんので、実行するためには、サーバを停止し、シングルユーザモードのバックエンドを使用して`VACUUM`を行う他ありません。この停止モードはシングルユーザモードのバックエンドでは強制されません。シングルユーザモードのバックエンドの使用に関する詳細は[postgres](#) マニュアルページを参照してください。

23.1.6. 自動バキュームデーモン

PostgreSQLには、省略可能ですが強く推奨される自動バキュームという機能があります。これは`VACUUM`と`ANALYZE`コマンドの実行を自動化することを目的としたものです。有効にすると、自動バキュームは大量のタプルの挿入、更新、削除があったテーブルを検査します。この検査は統計情報収集機能を使用します。したがって、[track_counts](#)が`true`に設定されていないと、自動バキュームを使用することができません。デフォルトの設定では、自動バキュームは有効で、関連するパラメータも適切に設定されています。

実際のところ"自動バキュームデーモン"は複数のプロセスから構成されます。自動バキュームランチャという永続的デーモンプロセスが存在し、自動バキュームワーカプロセスがすべてのデータベースを処理します。ランチャは、1つのワーカを各データベースに対し[autovacuum_naptime](#)秒ごとに開始するよう試みることで、時間に対して作業を分散化します。（したがってインストレーションに*N*個のデータベースがある場合、新規ワーカが[autovacuum_naptime](#)/*N*秒毎に起動されます。）同時に最大[autovacuum_max_workers](#)個のプロセスが実行可能です。処理対象のデータベースが[autovacuum_max_workers](#)より多くある場合、次のデータベースは最初のワーカが終了するとすぐに処理されます。それぞれのワーカプロセスはデータベース内の各テーブルを検査し、必要に応じてVACUUMまたはANALYZEコマンドを発行します。

短期間にいくつかの大規模なテーブルがすべてバキューム対象として適切な状態になったとすると、すべての自動バキュームワーカはこうしたテーブルに対するバキューム処理に長い期間占領される可能性があります。これにより、ワーカが利用できるようになるまで、他のテーブルやデータベースに対するバキュームが行われなくなります。また、単一データベースに対するワーカ数には制限はありませんが、ワーカはすでに他のワーカによって実行された作業を繰り返さないように試みます。ワーカの実行数は[max_connections](#)制限にも[superuser_reserved_connections](#)制限にも計上されないことに注意してください。

テーブルの[relfrozenxid](#)値が[autovacuum_freeze_max_age](#)トランザクション年齢よりも古い場合、そのテーブルは常にバキュームされます（これは[freeze_max_age](#)がストレージパラメータにより変更されたテーブルに対しても適用されます。以下を参照）。さもないと、直前のVACUUMの後に不要となったタプル数が"バキューム閾値"を超えると、テーブルはバキュームされます。このバキューム閾値は以下のように定義されます。

バキューム閾値 = バキューム基礎閾値 + バキューム規模係数 * タプル数

ここで、バキューム基礎閾値は[autovacuum_vacuum_threshold](#)、バキューム規模係数は[autovacuum_vacuum_scale_factor](#)、タプル数は[pg_class.reltuples](#)です。不要となったタプル数は、統計情報コレクタから取り出されます。これは、各UPDATE、DELETE操作で更新される、ほぼ正確な数です。（負荷が高いと一部の情報が失われる可能性があることから、これはほぼ正確な数でしかありません。）テーブルの[relfrozenxid](#)値が[vacuum_freeze_table_age](#)トランザクション年齢より大きい場合、テーブル全体は古いタプルを凍結するため走査され、[relfrozenxid](#)を繰り上げます。そうでなければ最後のバキュームの後に変更されたページのみ走査されます。

解析でも似たような条件が使用されます。以下で定義される閾値が、前回のANALYZEの後に挿入、更新、削除されたタプル数と比較されます。

解析閾値 = 解析基礎閾値 + 解析規模係数 * タプル数

一時テーブルには自動バキュームでアクセスすることはできません。したがってセッションのSQLコマンドを用いて適切なバキュームおよび解析操作を行わなければなりません。

デフォルトの閾値と規模係数は、[postgresql.conf](#)から取られますが、テーブル毎に上書きすることができます。より詳細な情報は[格納パラメータ](#)を参照してください。ストレージパラメータで設定が変更されるとその値が使用されます。そうでなければ、全体設定が使われます。全体設定についての詳細な情報は[項18.10](#)を参照してください。

基礎閾値と規模係数の他に、ストレージパラメータを介して各テーブルに対して設定することができる自動バキューム用のパラメータが更に6つあります。最初の[autovacuum_enabled](#)パラメータは、[false](#)に設定することで自動バキュームデーモンにこのテーブル全体を対象から外すようにします。この場合自動バキュームは、トランザクションIDの周回を防ぐためにバキュームする必要がある時のみ、このテーブルを対象とします。別の2つのパラメータ、[autovacuum_vacuum_cost_delay](#)と[autovacuum_vacuum_cost_limit](#)はコストベースのバキューム遅延機能([項18.4.4](#)参照)に対し、テーブル特定の値を設定するために使用されます。[autovacuum_freeze_min_age](#)、[autovacuum_freeze_max_age](#)および[autovacuum_freeze_table_age](#)は、それぞれ[vacuum_freeze_min_age](#)、[autovacuum_freeze_max_age](#)および[vacuum_freeze_table_age](#)に対する値を設定するために使用されます。

複数のワーカプロセスの実行中、コスト制限は実行中のワーカ全体に"振り分け"られます。このため、ワーカの実稼働数に関らず、システムに与える総影響は変わりありません。

[前のページ](#)

定常的なデータベース保守作業

[ホーム](#)

[上に戻る](#)

[次のページ](#)

定常的なインデックスの再作成

23.2. 定常的なインデックスの再作成

定期的に [REINDEX](#) コマンドを使用してインデックスを再構築することが価値がある状況があります。

完全に空になった B-tree インデックスページは再利用のために回収されます。しかしまだ非効率的な領域使用の可能性があります。ページからわずかを残しほとんどすべてのインデックスキーが削除されたとしても、ページは割り当てられたまま残ります。各範囲において、わずかを残しほとんどすべてのキーが削除されるようなパターンで使用されると、領域が無駄に使用されることが分かります。こうした使用状況では、定期的なインデックス再構築を推奨します。

B-tree 以外のインデックスが膨張する可能性はまだよく調査されていません。B-tree 以外の任意の種類のインデックスを使用する際には、インデックスの物理容量を定期的に監視することを勧めます。

また、B-tree インデックスでは、新規に構築したインデックスの方が何度も更新されたインデックスよりもアクセスが多少高速です。新しく構築されたインデックスでは論理的に近接するページが通常物理的にも近接するからです。（これは B-tree 以外のインデックスではあてはまりません。）アクセス速度を向上させるためだけに周期的にインデックスを再構築することは価値があるかもしれません。

[前のページ](#)

定常的なバキューム作業

[ホーム](#)[上に戻る](#)[次のページ](#)

ログファイルの保守

23.3. ログファイルの保守

データベースサーバのログ出力を `/dev/null` に渡して単に破棄するのではなく、どこかに保存しておくことを推奨します。問題の原因を究明する時にログ出力は貴重です。しかし、ログ出力は（特により高いデバッグレベルの時に）巨大になりがちです。で、際限なく保存したくはないでしょう。新しいログファイルを開始させ、適切な期間を経過した古いログファイルを捨てるために、ログファイルを"回転"させる必要があります。

単に `postgres` の `stderr` をファイルに渡している場合、ログ出力を保持できますが、そのログファイルを切り詰めるためにはサーバを停止させ、再度起動させるしか方法がありません。開発環境で PostgreSQL を使用しているのであればこれで構いませんが、実運用サーバでこの振舞いが適切となることはほぼありません。

サーバの `stderr` を何らかのログ回転プログラムに送信する方が良いでしょう。組み込みのログ回転機能があり、`postgresql.conf` の `logging_collector` 設定パラメータを `true` に設定することでこれを使用することができます。このプログラムを制御するパラメータについては [項18.8.1](#) で説明します。また、この方法を使用して、機械読み取りしやすい CSV (カンマ区分値) 書式でログデータを取り込むことができます。

また、既に他のサーバソフトウェアで使用している外部のログ回転プログラムがあるのであれば、それを使用したいと考えるでしょう。例えば、Apache 配布物に含まれる `rotatelogs` を PostgreSQL で使用することができます。これを行うには、単にサーバの `stderr` を目的のプログラムにパイプで渡してください。 `pg_ctl` を使用してサーバを起動している場合は `stderr` は既に `stdout` にリダイレクトされていますので、以下の例のようにコマンドをパイプする必要があるだけです。

```
pg_ctl start | rotatelogs /var/log/pgsql_log 86400
```

この他の実運用レベルのログ出力の管理方法は、`syslog` に送信し、`syslog` にファイルの回転を行わせることです。このためには、`postgresql.conf` の `log_destination` 設定パラメータを `syslog` (`syslog` のみにログを出力) に設定してください。そして、新しいログファイルへの書き込みを始めた時に、`syslog` デーモンに `SIGHUP` シグナルを送信してください。ログ回転を自動化させたい場合は、`logrotate` プログラムを設定することで、`syslog` からのログファイルを扱うことができます。

しかし、多くのシステムでは `syslog` は特に巨大なログメッセージに関してあまり信頼できません。必要なメッセージを切り詰めてしまったり、破棄してしまったりする可能性があります。また、Linux では、`syslog` はメッセージごとにディスクに書き出すため、性能が良くありません。（同期化を無効にするため、`syslog` 設定ファイル内のファイル名の先頭に `-` を使うことができます。）

上述の手法は全て、新しいログファイルを開始する周期を設定することができますが、古い、既に役に立たなくなったログファイルの削除は扱わないことに注意してください。おそらく定期的に古いログファイルを削除するバッチジョブを設定することになるでしょう。他に、回転用プログラムを設定して古いログファイルを周期的に上書きさせるという方法もあります。

[pgFouine](#) という外部プロジェクトは洗練されたログファイルの解析を行います。[check_postgres](#) は、通常ではない多くの状態の検出を行うのと同時にログファイルに重要なメッセージが現れた時に Nagios で警告する機構を提供します。

第 24 章 バックアップとリストア

目次

24.1. [SQLによるダンプ](#)

24.1.1. [ダンプのリストア](#)

24.1.2. [pg_dumpallの使用](#)

24.1.3. [大規模データベースの扱い](#)

24.2. [ファイルシステムレベルのバックアップ](#)

24.3. [継続的アーカイブとポイントインタイムリカバリ \(PITR\)](#)

24.3.1. [WALアーカイブの設定](#)

24.3.2. [ベースバックアップの作成](#)

24.3.3. [低レベルAPIを使用したベースバックアップの作成](#)

24.3.4. [継続的アーカイブによるバックアップを使用した復旧](#)

24.3.5. [タイムライン](#)

24.3.6. [ヒントと例](#)

24.3.7. [警告](#)

貴重なデータを全て保存しているため、PostgreSQLデータベースは定期的にバックアップされなければなりません。バックアップの手順は基本的に簡単ですが、使用されている諸技術といくつかの前提条件を明確に理解しておくことが重要です。

PostgreSQLのデータをバックアップする場合、3つの異なる手法があります。

- [SQLによるダンプ](#)
- [ファイルシステムレベルのバックアップ](#)
- [継続的アーカイブ](#)

それぞれ長所と短所があります。ひとつひとつ順を追って以下の節で説明します。

24.1. SQLによるダンプ

このダンプ方法の背景にある考え方はSQLコマンドでテキストファイルを生成し、そのファイルをサーバが再度読み込みを行った時に、ダンプした時点と同じ状態が再構築されるということです。この目的のため、PostgreSQLは[pg_dump](#)ユーティリティプログラムを提供しています。このコマンドの基本となる使い方は以下の通りです。

```
pg_dump dbname > outfile
```

見てわかる通り、pg_dumpは結果を標準出力に書き出します。これがどのように活用できるかをこれから説明します。

pg_dumpは、（優れた機能を特に発揮する）PostgreSQLの通常のクライアントアプリケーションです。ということは、データベースに接続可能なあらゆるリモートホストからこのバックアップ手順を実行することができます。しかし、pg_dumpは動作に特別な権限を必要とするわけではありませんが、特に、バックアップを行う全てのテーブルに対して読み込み権限を必要とします。ですから、実際の作業はほとんどの場合、データベースのスーパーユーザでバックアップを行わなければなりません。

pg_dumpを行うデータベースサーバを特定するにはコマンドラインの-h *host*オプションと-p *port*オプションを使用します。デフォルトのホストはローカルホスト、またはPGHOST環境変数で指定したものです。同様に、デフォルトのポートはPGPORT環境変数で指定されているか、うまく行かない場合にはコンパイル時の設定がデフォルトとなります（そこはうまくできていて、サーバは通常コンパイル時の設定をデフォルトとします）。

他のPostgreSQLのクライアントアプリケーションのように、pg_dumpはデフォルトでオペレーティングシステムの現在のユーザ名と同じデータベースユーザ名で接続します。これを書き換えるには-Uオプションを付けるかPGUSER環境変数を設定します。pg_dumpの接続は（[第19章](#)で説明されている）通常のクライアント認証方法によることを思い出してください。

後で述べる他のバックアップ手法に対するpg_dumpの重要な利点は、pg_dumpの出力は一般に新しいバージョンのPostgreSQLに再ロードできるということです。一方、ファイルレベルのバックアップと継続的アーカイブは両方とも非常にサーバ、バージョン依存です。pg_dumpは、32ビットから64ビットのサーバに移行するなどの異なるマシンアーキテクチャにデータベースを移す場合に上手くいく唯一の方法でもあります。

pg_dumpで作成されたダンプは、内部的に整合性があります。つまり、ダンプはpg_dumpが開始された際のデータベースのスナップショットを示しています。pg_dumpの操作はデータベースに対する他の作業を妨げません（ALTER TABLEのほとんどの形態であるような排他的ロックが必要な作業は例外です）。

重要項目：（例えば外部キーのように）データベーススキーマがOIDに依存している場合、pg_dumpにOIDも一緒にダンプするよう指定しなければなりません。これを行うには-oコマンドラインオプションを使用します。

24.1.1. ダンプのリストア

pg_dumpで作成されたテキストファイルはpsqlプログラムで読み込まれることを意図しています。以下に、ダンプをリストアする一般的なコマンドを示します。

```
psql dbname < infile
```

ここでinfileはpg_dumpコマンドにより出力されたファイルです。dbnameデータベースはこのコマンドでは作成されません。（例えばcreatedb -T template0 dbname のようにして）psqlを実行する前に自分でtemplate0から作成してください。psqlはpg_dumpと似たような、接続データベースサーバと使用するユーザ名を指定するオプションに対応しています。詳細については、[psql](#)のリファレンスページを参照してください。

SQLダンプのリストアを実行する前に、ダンプされたデータベース内のオブジェクトを所有するユーザやそのオブジェクト上に権限を与えられたユーザも存在しなければなりません。存在していない場合、リストアはそのオブジェクトの元々の所有権や付与された権限を再作成することができません（このようにしたい場合もあるでしょうが、通常そうではありません）。

デフォルトでpsqlスクリプトは、SQLエラーが起きた後も実行を続けます。ON_ERROR_STOP変数を設定してpsqlを実行することで、その動作を変更し、SQLエラーが起きた場合にpsqlが、終了ステータス3で終了するようにしたいと思うかもしれません。

```
psql --set ON_ERROR_STOP=on dbname < infile
```

どちらにしても、部分的にリストアされたデータベースにしかなりません。他に、ダンプ全体を1つのトランザクションとしてリストアするように指定することができます。こうすれば、リストアが完全に終わるか、完全にロールバックされるかのどちらかになります。このモードは、psqlのコマンドラインオプションに-1または--single-transactionを渡すことで指定できます。このモードを使用する場合、数時間かけて実行していたリストアが軽微なエラーでロールバックしてしまうことに注意してください。しかし、部分的にリストアされたダンプから手作業で複雑なデータベースを整理するよりまだましかもしれません。

pg_dumpとpsqlではパイプから読み書きができるので、あるサーバから別のサーバへデータベースを直接ダンプできます。以下に例を示します。

```
pg_dump -h host1 dbname | psql -h host2 dbname
```

重要項目: pg_dumpで作成されるダンプはtemplate0と相対関係にあります。つまりtemplate1を経由して追加されたあらゆる言語、プロシージャなどもpg_dumpによりダンプされます。その結果としてリストアする際に、カスタマイズされたtemplate1を使用している場合は、上記の例のように、template0から空のデータベースを作成する必要があります。

バックアップをリストアした後、問い合わせオプティマイザが有用な統計情報を使用できるように、各データベースに対してANALYZEを実行することを勧めます。より詳しくは、[項23.1.3](#)と[項23.1.6](#)を参照してください。効率的に大規模なデータをPostgreSQLにロードする方法に関するより多くの勧告については、[項14.4](#)を参照してください。

24.1.2. pg_dumpallの使用

pg_dumpは同時に単一のデータベースのみをダンプします。また、ロールやテーブル空間についての情報はダンプしません。（これらはテーブル毎ではなくクラスタ全体のものだからです。）データベースクラスタの全内容の簡便なダンプをサポートするために、[pg_dumpall](#)プログラムが提供されています。pg_dumpallは指定されたクラスタの各データベースのバックアップを行い、そして、ロールやテーブル空間定義などのクラスタ全体にわたるデータを保存します。このコマンドの基本的な使用方法は

```
pg_dumpall > outfile
```

です。ダンプの結果はpsqlでリストアできます。

```
psql -f infile postgres
```

（実際、開始時に任意の既存のデータベース名を指定することができますが、空のクラスタ内にロードする場合は、通常postgresを使用すべきです。）ロールやテーブル空間の情報をリストアしなければならないので、pg_dumpallのダンプをリストアする時には、データベーススーパーユーザのアクセス権限を確実に必要とします。テーブル空間を使用している場合、ダンプ内のテーブル空間のパスが新しいインストレーションで適切であることを確認してください。

pg_dumpallはコマンドを発令することによりロール、テーブル空間、およびデータベースを再作成し、それぞれのデータベースに対してpg_dumpを起動します。このことは、それぞれのデータベースには内部的に矛盾がない一方、異なるデータベースのスナップショットは完全に同期しない可能性があることを示しています。

24.1.3. 大規模データベースの扱い

オペレーティングシステムの中には最大ファイルサイズに制限があるものがあり、大きなpg_dump出力ファイルを作成しているときに問題を引き起こします。幸運なことに、pg_dumpは標準出力に書き出すことができますので、Unix標準のツールを使ってこの潜在的な問題を解決できます。取りうる方法がいくつか存在します。

圧縮ダンプの使用。たとえば、自分が愛用しているgzipのような圧縮プログラムが使えます。

```
pg_dump dbname | gzip > filename.gz
```

元に戻すには次のようにします。

```
gunzip -c filename.gz | psql dbname
```

あるいは次のようにもできます。

```
cat filename.gz | gunzip | psql dbname
```

splitの使用。splitコマンドで結果を使用しているファイルシステムが受け付けられる大きさに分割することができます。例えば1メガバイトずつに分割するには次のようにします。

```
pg_dump dbname | split -b 1m - filename
```

元に戻すには次のようにします。

```
cat filename* | psql dbname
```

pg_dumpのカスタムダンプ書式の使用。もしPostgreSQLがzlib圧縮ライブラリインストール済みのシステム上で構築されたのなら、カスタムダンプ書式では出力ファイルに書き出す時にデータを圧縮します。gzipを使用した時と似通ったダンプサイズとなりますが、テーブルの復元を部分的に行えるという点で優れていると言えます。以下のコマンドは、カスタムダンプ書式でのデータベースのダンプを行います。

```
pg_dump -Fc dbname > filename
```

カスタム書式のダンプはpsql用のスクリプトではありませんので、代わりにpg_restoreでリストアしなければなりません。例えば以下のようにします。

```
pg_restore -d dbname filename
```

詳細は[pg_dump](#)と[pg_restore](#)のリファレンスページを参照してください。

巨大なデータベースに対しては、そのほかの2つの手法のうちの1つと一緒にsplitを組み合わせる必要があるかもしれません。

[前のページ](#)

バックアップとリストア

[ホーム](#)
[上に戻る](#)

[次のページ](#)

ファイルシステムレベルのバックアップ

24.2. ファイルシステムレベルのバックアップ

バックアップ戦略の代替案としてPostgreSQLがデータベース内のデータを保存するために使用しているファイルを直接コピーする方法があります。 [項17.2](#)にこれらのファイルがどこにあるか解説されています。 下記のような通常のファイルシステムのバックアップを行うどんな方法でも問題ありません。

```
tar -cf backup.tar /usr/local/pgsql/data
```

しかしこの方法には2つの制約があり、そのためにあまり実用的ではなく、少なくともpg_dumpより劣ると言わざるを得ません。

- 1 11 有効なバックアップを行うにはデータベースサーバを必ず停止しなければなりません。 全ての接続を無効とするような中途半端な対策では作用しません (`tar`やその類似ツールはある時点におけるファイルシステムの原子的なスナップショットを取らないことと同時に、サーバ内の内部バッファリングの理由によるからです)。 サーバの停止に関しては[項17.5](#)を参照してください。 言うまでもありませんが、データをリストアする前にもサーバを停止させる必要があります。
- 1 11 データベースのファイルシステムレイアウトの詳細を熟知している場合、ある個別のテーブルやデータベースをそれぞれのファイルやディレクトリからバックアップしたり復元したりすることを試みたいと思うことがあったかもしれません。 しかし、それらのファイル内の情報はすべてのトランザクションのコミット状態を保持するコミットログファイルpg_clog/*なしでは使えないため、この方法では正常なバックアップは行えません。 テーブルファイルはこの情報があって初めて意味をなします。 もちろんテーブルとそれに付随するpg_clogデータだけで復元することも、データベースクラスタにある他のテーブルを無効としてしまうのでできません。 ですので、ファイルシステムバックアップは、データベースクラスタ全体の完全なバックアップとリストア処理にのみ動作します。

その他のファイルシステムバックアップ方法として、ファイルシステムが"整合性を維持したスナップショット"機能をサポートしている場合（かつ、正しく実装されていると信用する場合）、データディレクトリのスナップショットを作成する方法があります。 典型的な手順では、データベースを含むボリュームの"凍結スナップショット"を作成し、データディレクトリ全体（上述のように、一部だけではいけません）をスナップショットからバックアップデバイスにコピーし、そして、凍結スナップショットを解放します。 これはデータベースサーバが稼働中であっても動作します。 しかし、こうして作成されたバックアップは、データベースサーバが適切に停止されなかった状態のデータベースファイルを保存します。 そのため、このバックアップデータでデータベースサーバを起動する時、直前のサーバインスタンスがクラッシュしたものとみなされ、WALログが取り直されます。 これは問題ではありません。 単に注意してください（そして、確実にバックアップにWALファイルを含めてください）。 CHECKPOINTコマンドをスナップショット取得前に発行することでリカバリ時間を減らすこともできます。

対象のデータベースが複数のファイルシステムにまたがって分散している場合、全てのボリュームに対して完全に同期した凍結スナップショットを得る方法が存在しない可能性があります。例えば、データファイルとWALログが異なったディスク上にあったり、テーブル空間が異なるファイルシステム上にある場合、スナップショットは同時でなければなりませんので、スナップショットのバックアップを使用できない可能性があります。 こうした状況では、整合性を維持したスナップショット技術を信用する前に使用するファイルシステムの文書を熟読してください。

同時実行のスナップショットができない場合、選択肢の1つとして、全ての機能の停止したスナップショットを確定させるのに十分な時間、データベースサーバをシャットダウンさせることが挙げられます。 他の選択肢は、継続的なベースバックアップの保管 ([項24.3.2](#)) を行うことです。 こうしたバックアップには、バックアップ中のファイルシステムの変更を心配する必要がないためです。 これにはバックアップ処理期間のみに継続的な保管を行う必要があり、継続的なアーカイブリカバリ ([項24.3.4](#)) を使用してリストアを行います。

ファイルシステムをバックアップするその他の選択肢としてrsyncの使用が挙げられます。 これを行うには、先ずデータベースサーバが稼働中にrsyncを実行し、そして次のrsyncを実行するのに十分な余裕をもってデータベースサーバを停止します。 次のrsyncは、比較的転送するデータ量が少なく、サーバが稼働していないため最終結果に矛盾がないことから、最初のrsyncよりもより迅速です。 この方法で最小の稼働停止時間でファイルシステムのバックアップを行う事ができます。

ファイルシステムバックアップは、概してSQLによるダンプより大きくなることに注意してください。（pg_dumpでは、例えばインデックスの内容をダンプする必要はありません。 単にコマンドで再作成します。） しかし、ファイルシステムのバックアップを取るほうがより高速でしょう。

24.3. 継続的アーカイブとポイントインタイムリカバリ (PITR)

PostgreSQLは常に、クラスタのデータディレクトリ以下の`pg_xlog/`ディレクトリ内で先行書き込みログ (WAL) を管理しています。このログはデータベースのデータファイルに行われた全ての変更を記録します。このログは主にクラッシュ時の安全性を目的としています。システムがクラッシュしたとしても、最後のチェックポイント以降に作成されたログ項目を"やり直し"することで、データベースを整合性を維持した状態にリストアすることができます。しかし、この存在するログファイルを使用して、データベースのバックアップ用の第3の戦略が可能になりました。ファイルシステムレベルのバックアップとWALファイルのバックアップを組み合わせるという戦略です。復旧が必要ならば、ファイルシステムバックアップをリストアし、その後にバックアップされたWALファイルを再生することで、システムを最新の状態にできます。管理者にとって、この方法はこれまで説明した方法よりかなり複雑になりますが、以下のような大きな利点が複数あります。

- 開始時点のファイルシステムバックアップは完全な整合状態である必要はありません。そのバックアップ内の内部的な不整合はログのやり直しによって修正されます (これは、クラッシュからの復旧時に行われることと大きな違いはありません)。ですので、ファイルシステムのスナップショット機能を必要としません。単にtarなどのアーカイブツールが必要です。
- 再生の際にWALファイルの並びを数に制限なく連ねて組み合わせられますので、単にWALファイルのアーカイブを続けることで連続したバックアップを達成できます。これは、頻繁に完全なバックアップを行うことが困難な、大規模なデータベースでは特に価値があります。
- WAL項目の再生を最後まで行わなければならないということはありません。やり直しを任意の時点までで停止することができます、それにより、その時点までのデータベースの整合性を持ったスナップショットを得ることができます。このような技術がポイントインタイムリカバリを補助するものであり、元となるベースバックアップの取得時点以降の任意の時点の状態にデータベースをリストアすることが可能になります。
- 連続的に一連のWALファイルを、同一のベースバックアップをロードしている別のマシンに配送することで、"ウォームスタンバイ"システムを保有することができます。つまり、任意の時点でその2番目のマシンを、ほぼ現時点のデータベースの複製を持った状態で有効にすることができます。

注意: `pg_dump`と`pg_dumpall`はファイルシステムレベルのバックアップを生成しませんので、継続的アーカイブ方式の一部として使うことはできません。そのダンプは論理的なものであり、WALのやり直しで使うのに十分な情報を含んでいません。

通常のファイルシステムバックアップ技術の場合と同様、この方法は、一部ではなく、データベースクラスタ全体のリストア処理のみをサポートできます。また、アーカイブ用に大量の格納領域を必要とします。ベースバックアップはかさばる場合があります、また、高負荷なシステムではアーカイブしなければならないWALの流量をメガバイト単位で生成します。しかし、これは、高信頼性が必要な、多くの状況でむしろ好まれるバックアップ手法です。

継続的アーカイブ (多くのデータベースベンダーで"オンラインバックアップ"とも呼ばれます) を使用して復旧を成功させるためには、少なくともバックアップの開始時点まで遡る、連続した一連のアーカイブ済みWALファイルが必要です。ですので、運用するためには、最初のベースバックアップを取得する前にWALファイルをアーカイブする手順を設定し試験しなければなりません。したがって、まずWALファイルのアーカイブ機構について説明します。

24.3.1. WALアーカイブの設定

理論上、PostgreSQLシステムの稼働により、不定長のWAL記録の並びが生成されます。システムは物理的にこの並びを、通常1つ16メガバイト (このセグメントサイズはPostgreSQLの構築時に変更可能です) の、WALセグメントファイルに分割します。このセグメントファイルには、概念的なWALの並び内の位置を反映した、数字の名前が付与されます。WALアーカイブを行わない場合、システムは通常数個のセグメントファイルを生成し、また、不要となったセグメントファイルの名前をより大きなセグメント番号に変更することでそれを"再回収"します。直前のチェックポイントより前の内容を持つセグメントファイルは使用されないと仮定され、再回収されます。

WALデータをアーカイブする場合、完成したセグメントファイルのそれぞれの内容を取り出し、再利用のために回収される前にそのデータをどこかに保存することが必要です。アプリケーションと利用できるハードウェアに依存しますが、数多くの"データをどこかに保存する"方法があります。例えば、NFSでマウントした他のマシンのディレクトリにセグメントファイルをコピーすること、あるいは、テープ装置に書き出すこと (元々のファイル名を識別する手段があることを確認してください)、それらを一度にまとめてCDに焼くこと、そのほか全く異なったなんらかの方法などです。柔軟性をデータベース管理者に提供するために、PostgreSQLは、どのようにアーカイブがなされたかについて一切想定しないようになっています。その代わりにPostgreSQLは、管理者に完全なセグメントファイルをどこか必要な場所にコピーするシェルコマンドを指定させます。このコマンドは単純な`cp`でも構いませんし、また、複雑なシェルスクリプトを呼び出しても構いません。全て管理者に任されています。

WALアーカイブを有効にするには`wal_level`構成パラメータを`archive`（または`hot_standby`）に、`archive_mode`を`on`に設定し、`archive_command` 構成パラメータで使用するシェルコマンドを指定します。実行するには、これらの設定を`postgresql.conf` ファイルに常に置きます。 `archive_command` では、`%p`はアーカイブするファイルのパス名に置換され、`%f`はファイル名部分のみに置換されます。（パス名は、サーバの現在の作業用ディレクトリ、つまり、クラスタのデータディレクトリから見て相対的なものです。） コマンド内に`%`文字自体を埋め込む必要があれば`%%`を使ってください。最も簡単でよく使用されるコマンドは以下のようなものになります。

```
archive_command = 'test ! -f /mnt/server/archivedir/%f && cp %p /mnt/server/archivedir/%f' # Unix
archive_command = 'copy "%p" "C:¥¥server¥¥archivedir¥¥%f"' # Windows
```

これは、アーカイブ可能なWALセグメントを`/mnt/server/archivedir`ディレクトリにコピーします（これは一例です。推奨するものではなく、また、全てのプラットフォームで動作しない可能性があります）。`%p`および`%f`パラメータが置き換えられたあと、実行された実コマンドは以下のようになります。

```
test ! -f /mnt/server/archivedir/000000010000000A9000000065 && cp pg_xlog/000000010000000A9000000065
/mnt/server/archivedir/000000010000000A9000000065
```

類似したコマンドがアーカイブされるそれぞれの新規ファイルに生成されます。

このアーカイブ用コマンドはPostgreSQLサーバを稼働させるユーザと同じ所有権で実行されます。アーカイブされる一連のWALファイルには、実質、データベース内の全てが含まれていますので、アーカイブしたデータをのぞき見から確実に保護しなければならないでしょう。例えば、グループや全員に読み込み権限を付与していないディレクトリにデータをアーカイブしてください。

アーカイブ用コマンドが成功した場合のみにゼロという終了ステータスを返すことが重要です。PostgreSQLは、ゼロという結果に基づいて、そのファイルのアーカイブが成功したことを想定し、そのファイルを削除したり回収するかもしれません。しかし、非ゼロのステータスは、PostgreSQLに対してファイルがアーカイブされなかったことを通知し、成功するまで定期的に再試行させます。

通常アーカイブ用コマンドは既存のアーカイブ済みファイルの上書きを行わないように設計されなければなりません。これは、管理者のミス（例えば2つの異なるサーバの出力を同一のアーカイブ用ディレクトリに送信してしまうなど）といった場合からアーカイブ状況の整合性を保護するための安全策として重要です。

実際に既存のファイルを上書きしないこと、かつ、その場合に非ゼロのステータスを返すことを確認するために使用するアーカイブ用コマンドを試験することを勧めます。上のUnix用のコマンド例では、別途`test`という段階を含めることでこれを確認しています。いくつかのUnixプラットフォームでは`cp`コマンドには`-i` 引数を使うことで煩雑な出力を少なくし使うことができますが、正しい終了コードが返ることを確認せずに使用するべきではありません。（具体的にはGNUの`cp`コマンドは`-i` オプションなどを使い、ターゲットファイルがすでに存在している場合、ゼロのステータスを返します。これは期待していない動作です。

アーカイブ設定を設計する時には、操作者の介入が必要であったり、アーカイブ場所の容量不足の理由でアーカイブ用コマンドが繰り返し失敗した時にどうなるかを考慮してください。例えば、オートチェンジャ機能のないテープに書き出している場合に発生する可能性があります。テープが一杯になった場合、テープを交換するまでアーカイブを行うことができなくなります。こうした状況を相応の早さで解消できるよう、適切に操作者に対しエラーや要求を確実に連絡できるようにしなければなりません。この状況が解消するまで、WALセグメントファイルは`pg_xlog/`ディレクトリ内に格納され続けます。（`pg_xlog/`を含むファイルシステムがいっぱいになると、PostgreSQLはパニック停止します。コミットされたトランザクションは失われませんが、データベースはいくらかの容量を開放するまでオフラインのままです。）

サーバのWALデータの生成に要する平均速度に追いついている限り、アーカイブ用コマンドの処理速度は重要ではありません。アーカイブプロセスが多少遅れたとしても通常の操作は続けられます。アーカイブ処理がかなり遅くされると、災害時に損失するデータの量が増加することになります。また、これは`pg_xlog/`ディレクトリ内に多くのアーカイブ処理待ちのセグメントファイルが格納され、ディスク容量が不足する状況になる可能性があることを意味します。アーカイブ処理が確実に意図通りに動作しているかを監視することを推奨します。

アーカイブ用コマンドを作成する時、アーカイブされるファイル名は最長64文字までで、ASCII文字と数字とドットのどんな組合せを使用しても構いません。元の相対パス（`%p`）を保存する必要はありませんが、ファイル名（`%f`）を保存する必要があります。

WALアーカイブによってPostgreSQLデータベースでなされた変更は全てリストアすることができますが、設定ファイルはSQL操作ではなく手作業で変更されますので、設定ファイル（`postgresql.conf`、`pg_hba.conf`、および`pg_ident.conf`）になされた変更までリストアしないことに注意してください。通常ファイルシステムバックアップ手続きでバックアップされる場所に設定ファイルを保持したい場合があります。設定ファイルの設置場所を変更するには[項18.2](#)を参照してください。

アーカイブコマンドは完全なWALセグメントに対してのみ呼び出されます。このため、サーバが少ししかWAL流量がない（処理を行わないながの期間がある）場合、トランザクションの完了とアーカイブ格納領域への安全な記録との間に長期にわたる遅延があることになります。古い未アーカイブのデータをどうするかについて制限を付けるために、`archive_timeout`を設定して、強制的にサーバを新しいWALセグメントにある程度の間隔で切り替えるようにすることができます。強制切り替えにより早期にアーカイブされたアーカイブ済みファイルは完全に完了したファイルと同じ大きさを持つことに注意してください。そのため、非常に小さな`archive_timeout`を使用することはお勧めしません。格納領域を膨張させてしまいます。通常ならば分単位

の `archive_timeout` 設定が合理的です。

終わったばかりのトランザクションをできるだけ早くアーカイブさせたい場合、`pg_switch_xlog` を使用して手作業でセグメント切り替えを強制することができます。その他のWAL管理に関連した関数を表9-59に列挙します。

`wal_level` が `minimal` の場合、項14.4.7に書かれているように、いくつかのSQLコマンドはWALロギングを回避するため最適化されます。アーカイビングもしくはストリーミングレプリケーションがこれら構文の1つを実行中に作動させられると、アーカイブ復旧のための十分な情報をWALが含まなくなります。（クラッシュ復旧は影響を受けません。）このことにより、`wal_level` はサーバの起動時のみ変更可能です。とは言っても、`archive_command` は構成ファイルを再読み込みすることで変更できます。一時的にアーカイビングを停止したい場合、1つの方法は `archive_command` を空文字列（`''`）に設定することで、このようにすると、動作する `archive_command` が再構築されるまでWALファイルは `pg_xlog/` に蓄積します。

24.3.2. ベースバックアップの作成

ベースバックアップを取得する最も簡単な方法は `pg_basebackup` を実行する方法です。通常のファイルやTAR形式のファイルとしてベースバックアップを取得することができます。もし、`pg_basebackup` より柔軟性が求められる場合は、低レベルなAPIを使ってバックアップを作成することもできます。（詳細は 項24.3.3を参照）

ベースバックアップを取得するための時間を考慮する必要はありません。しかし、普段、`full_page_writes` を無効にして運用している場合、バックアップ取得中は強制的に `full_page_writes` が有効になるため、パフォーマンスが落ちていると感じる可能性があります。

バックアップを使用するためには、ファイルシステムのバックアップ取得中、および、その後に生成されるWALセグメントファイル全てが保存されている必要があります。この目的のために、ベースバックアップの過程で即座にWALアーカイブ領域にバックアップ履歴ファイルが作成されます。このファイルにはファイルシステムのバックアップに最初に必要とされるWALセグメントの名前が付けられます。例えば、最初のWALファイルが `0000000100001234000055CD` である場合、バックアップ履歴ファイルは `0000000100001234000055CD.007C9330.backup` というように名付けられます。（ファイル名の2番目のパートはWALファイルの厳密な位置が記載されます。通常は無視することができます。）一旦、安全にファイルシステムのバックアップとそのバックアップ中に使用されたWALセグメントファイル（バックアップ履歴ファイルから特定できます）を取得すると、それより数値の小さな全てのWALアーカイブセグメントはファイルシステムのリカバリには必要が無く、削除することができます。しかし、データを確実に復旧させるためには数世代のバックアップセットを保持することを考慮すべきです。

バックアップ履歴ファイルは、ほんの小さなテキストファイルです。これには `pg_basebackup` で与えたラベル文字列の他、バックアップの開始、終了時間およびバックアップのWALセグメントが含まれます。このラベルをバックアップを構成するために使うとことで、アーカイブ履歴ファイルはどのバックアップをリストアするべきか間違いなく判断することができます。

最後のベースバックアップ以降のWALアーカイブを保持し続ける必要があるため、通常、ベースバックアップを取得すべき期間は、WALアーカイブを保持するためにどのくらいのストレージを拡張できるかによって決定されます。また、リカバリが必要になった場合に、どのくらいの時間をリカバリに使うと覚悟するのもかも考慮すべきです。システムは全てのWALセグメントを適用する必要があるため、もし、最後のベースバックアップを取得してから長い時間が経過している場合、適用に時間を要する可能性があります。

24.3.3. 低レベルAPIを使用したベースバックアップの作成

低レベルのAPIを使ったベースバックアップを取得するには `pg_basebackup` を使う方法に加えて数ステップが必要ですが、比較的簡単です。これらのステップは順番に実行することが重要で、次のステップに進む前にこれらのステップが成功していることを確認する必要があります。

1 11 WALアーカイブが有効であり、正常に動作することを確認してください。

1 11 データベースにスーパーユーザとして接続し、以下のコマンドを発行してください。

```
SELECT pg_start_backup('label');
```

ここで `label` は、バックアップ操作を一意に識別するために使用する任意の文字列です（推奨方法は、格納先のバックアップダンプファイルの完全パスを使用することです）。`pg_start_backup` は、開始時刻やラベル文字列などのバックアップ情報を持つ `backup_label` という名前のバックアップラベルファイルを、クラスタディレクトリ内に作成します。

このコマンドを実行した時にクラスタ内のどのデータベースに接続したのかは注意する必要はありません。この関数が返す結果は無視できますが、エラーが発生した場合は作業を進める前に対応してください。

デフォルトで、`pg_start_backup` は終了までに長い時間がかかる場合があります。その理由はあるチェックポイントを実行し、そして、デフォルトでは設定したチェックポイント間隔の半分である、そのチェックポイントに必要なI/Oがかなりの時間にわたって広がるためです（設定パラメータについては `checkpoint_completion_target` を参照してください）。通常これは、問い合わせ処理における影響を極小化するので望ましいことです。バックアップをなるべく早く行いたいのであれば、以下を使用します。

```
SELECT pg_start_backup('label', true);
```

これはチェックポイントをできる限り早く行うよう強制します。

- 1 11 (pg_dumpやpg_dumpallではなく)tarやcpioなどの使い慣れた任意のファイルシステムバックアップツールを使用して、バックアップを実行してください。この作業時に、データベースの通常の操作を停止することは不要ですし、望ましい方法でもありません。

- 1 11 再度、スーパーユーザとしてデータベースに接続し、以下のコマンドを発行してください。

```
SELECT pg_stop_backup();
```

これはバックアップモードを終了し、次のWALセグメントへの自動切換えを行います。この切り替えの理由は、バックアップ期間中に書き出された最後のWALファイルがアーカイブできるよう準備することです。

- 1 11 バックアップ間で使用されたWALセグメントファイルがアーカイブされれば完了です。pg_stop_backupの結果で識別されるファイルは、バックアップファイル一式を完結させるのに必要となる最終セグメントです。archive_modeが有効であればpg_stop_backupは最終セグメントがアーカイブされるまで戻りません。すでにarchive_commandを設定していますので、これらのファイルのアーカイブ操作は自動的に発生します。ほとんどの場合、これは瞬時に行われます。しかし、バックアップの完了を確認できるよう、アーカイブシステムを監視し、遅延が無いことの確認をお勧めします。アーカイブコマンドの失敗によりアーカイブ処理が遅れてしまったとしても、アーカイブが成功し、そしてバックアップが完了するまで再試行を繰り返すようになっています。pg_stop_backup実行においての時間期限を設けたい場合、適切なstatement_timeoutの値を設定してください。

ファイルシステムの中には複製している途中でファイルが変更されると警告もしくはエラーを報告するものがあります。稼働しているデータベースのベースバックアップを取っている場合には、この状況は正常でエラーではありません。しかし、この種の警告と本当のエラーとを区別できるか確認が必要です。例えば、rsyncのバージョンによっては"消滅したソースファイル"に対して別の退出コードを返し、そしてこの退出コードをエラーではないと受け付けるドライバスクリプトを記述することができます。同時にGNU tarのバージョンによっては、tarがそれを複製していた途中でファイルが切り詰められると、致命的エラーと識別できないエラーコードを返します。ありがたいことに、GNU tarのバージョン1.16もしくはそれ以降では、バックアップ中にファイルが変更されると1で、それ以外のエラーの時は2でプログラムから抜けます。GNUのtarで1.23以降のバージョンを使用しているのであれば、--warning=no-file-changed --warning=no-file-removedオプションをつけることで関連する警告メッセージを隠すオプションを使用することができます。

バックアップダンプに、データベースクラスタディレクトリ（例えば/usr/local/pgsql/data）以下にある全てのファイルが含まれていることを確認してください。このディレクトリ以下に存在しないテーブル空間を使用している場合、注意して、同様にそれらを含めてください（そして、バックアップダンプがリンクとしてシンボリックリンクをアーカイブしていることを確認してください。さもないとリストアはテーブル空間を壊してしまいます）。

しかし、クラスタのpg_xlog/サブディレクトリにあるファイルをバックアップダンプから省くことができます。このちょっとした調整は、リストア処理中の失敗の危険性を低減できますので、行う価値があります。pg_xlog/がクラスタディレクトリ外のどこかを指し示すシンボリックリンクの場合は調整が簡単です。これは性能上の理由でよく使用される設定です。また、いずれこのバックアップを使うpostmasterではなく、今起動しているpostmasterの情報を記録しているpostmaster.pidとpostmaster.optsも除外できます。（これらのファイルはpg_ctlを誤作動させる可能性があります）

最後のベースバックアップまで遡ることができる全てのアーカイブ済みWALファイルを保持する必要がありますので、ベースバックアップの実行間隔は通常、アーカイブ済みWALファイルを格納する領域をどれだけ拡張したいかによって決定されます。また、復旧処理に費すことができる時間がどの位許されるかについても考慮しなければなりません。復旧が必要になった時に、システムはこれらのセグメントを全てやり直しの必要がありますが、最後のベースバックアップからの経過時間が長ければその分やり直しに時間がかかります。

また、pg_start_backup関数がデータベースクラスタディレクトリ内にbackup_labelという名前のファイルを作成することに注意してください。これは、pg_stop_backupによって削除されます。当然ながら、このファイルはバックアップダンプファイルの一部としてアーカイブされます。バックアップラベルファイルには、pg_start_backupに付与したラベル文字列とpg_start_backupが実行された時刻、最初のWALファイルの名前が含まれます。したがって、当惑した時にバックアップダンプファイルの中身を検索し、そのダンプファイルがどのバックアップセッションに由来したものを確認することができます。

サーバが停止している時にバックアップダンプを作成することも可能です。この場合、わかりきったことですが、pg_start_backupやpg_stop_backupを使用することができません。そのため、どのバックアップダンプが、どのWALファイルと関連し、どこまで戻せばよいかを独自の方法で残さなければなりません。通常は、上述の継続的アーカイブ手順に従う方をお勧めします。

24.3.4. 継続的アーカイブによるバックアップを使用した復旧

さて、最悪の事態が発生し、バックアップから復旧する必要が出てきたものとします。以下にその手順を説明します。

- 1 11 もし稼働しているのであればサーバを停止してください。

- 1 11 もし容量があるのであれば、後で必要になる場合に備えてクラスタデータディレクトリ全体とテーブル空間を全て一時的な場所にコピーしてください。この予防措置は、既存のデータベースを2つ分保持できるだけの空き領域を必要とします。十分な領域がない場合でも、少なくともクラスタのpg_xlogサブディレクトリの内容は保存すべきです。ここには、システムが停止する前にアーカイブされなかったログファイルが含まれているかも知れないからです。
- 1 11 クラスタデータディレクトリ以下、および、使用中のテーブル空間の最上位ディレクトリ以下にある既存の全てのファイルとサブディレクトリを削除してください。
- 1 11 ファイルシステムバックアップからデータベースファイルをリストアします。ファイルが正しい所有権（rootではなくデータベースシステムユーザです！）でリストアされていることを確認してください。テーブル空間を使用している場合は、pg_tblspc/内のシンボリックリンクが正しくリストアされていることを検証する必要があります。
- 1 11 pg_xlog/内にあるファイルをすべて削除してください。これらはファイルシステムバックアップから生成されたものであり、おそらく現在のものより古く使用できないものです。pg_xlog/をまったくアーカイブしていなければ、適切な権限で再作成してください。以前シンボリックリンクとして設定していたのであれば、そのように確実に再構築するように注意してください。
- 1 11 手順2で退避させた未アーカイブのWALセグメントファイルがあるのであれば、pg_xlog/にコピーしてください。（問題が発生し、初めからやり直さなければならない場合に未変更のファイルが残るように、移動させるのではなくコピーすることが最善です。）
- 1 11 復旧コマンドファイルrecovery.conf（[第26章](#)を参照）をクラスタデータディレクトリに作成してください。また、一時的にpg_hba.confを編集し、復旧の成功を確認できるまで一般ユーザが接続できないようにする必要があるかもしれません。
- 1 11 サーバを起動してください。サーバは復旧モードに入り、必要なアーカイブ済みWALファイル群の読み込みを行います。外部的なエラーにより復旧が中断したら、サーバを単に再起動させて、復旧処理を継続してください。復旧処理が完了したら、（誤って後で復旧モードに再度入らないように）postmasterはrecovery.confの名前をrecovery.doneに変更します。その後通常のデータベース操作を開始します。
- 1 11 データベースの内容を検査し、希望する状態まで復旧できていることを確認してください。復旧できなかった場合は手順1に戻ってください。全て問題なければ、ユーザが接続できるようにpg_hba.confを正常状態に戻してください。

ここで重要となるのは、復旧設定ファイルを設定することです。このファイルで、どのように復旧させたいのかやどこまで復旧させたいかを記述します。recovery.conf.sample（通常はインストレーションのshare/ディレクトリに格納されています）を手本として使用することができます。recovery.confで絶対に指定しなければならないことは、アーカイブ済みWALファイルセグメントをどのように戻すかをPostgreSQLに通知するrestore_commandです。archive_command同様、これはシェルコマンド文字列です。ここには、対象のログファイルの名前で置換される%fやログファイルのコピー先を示すパスで置換される%pを含めることができます。（パス名は現在の作業用ディレクトリ、つまり、クラスタのデータディレクトリから見た相対パスです。）コマンド内に%文字自体を埋め込む必要があれば%%と記載してください。最も簡単でよく使われるコマンドは以下のようになります。

```
restore_command = 'cp /mnt/server/archivedir/%f %p'
```

これは事前にアーカイブされたWALセグメントを/mnt/server/archivedirディレクトリからコピーします。当然ながら、もっと複雑なものを使用することができます。例えば、操作者に適切なテープをマウントさせることを要求するようなシェルスクリプトでさえ可能です。

このコマンドが失敗した時に非ゼロの終了ステータスを返すことが重要です。このコマンドは、アーカイブに存在しないファイル要求するかもしれませんが、その場合でも非ゼロを返さなければなりません。これはエラー状態ではありません。必要なファイルの全てがWALセグメントファイルではありません。backup、または.historyが付いているファイルが要求されることを想定しなければなりません。backup、または.historyが付いているファイルに対しての要求を期待することもあるでしょう。同時に、%pパスのファイル名部分は%fと異なることに注意してください。これらが相互に置き換え可能であるとは考えないでください。

アーカイブ場所で見つけれなかったWALセグメントはpg_xlog/から検索されます。これにより、最近の未アーカイブのセグメントを使用することができます。しかし、アーカイブ場所から利用できるセグメントはpg_xlog/内のファイルよりも優先的に使用されます。

通常は利用可能な全てのWALセグメントを使用して復旧処理が行われます。その結果、データベースを現時点まで（もしくは、利用可能なWALセグメントで得られる限り現在に近い時点まで）リストアします。従って、通常の復旧は"file not found"メッセージで終了します。エラーメッセージの正確な文言はrestore_commandの選択によります。また、復旧の開始時点で00000001.historyのようなファイル名のエラーメッセージが出ることがあります。これも単純な復旧作業では不具合を意味するものでなく正常です。論議については[項24.3.5](#)を参照してください。

もし以前のある時点まで復旧させたい場合（例えば、経験不足のデータベース管理者が主トランザクションテーブルを消去した直前）、recovery.confに要求する停止時点を指定するだけです。停止時点は、"recovery target"として既知の停止時点で指定

することも、日付と時刻で指定することも、リストアポイントが完了した特定のトランザクションIDで指定することもできます。本ドキュメントの執筆時点では使用するトランザクションIDの識別を補助するツールがありませんので、ほとんどの場合は日付と時刻による指定のみを使用することになるでしょう。

注意: 停止時点はバックアップの終了時刻、つまり、`pg_stop_backup`の最終時刻より後の時点でなければなりません。バックアップを行っている最中のある時点までベースバックアップを使用して復旧させることはできません（こうした時点まで復旧させるには、その前のベースバックアップまで戻って、そこからロールフォワードしてください）。

復旧時にWALデータの破損がわかると、復旧はその時点で止まり、サーバは起動しません。こうした場合、"復旧対象"に破損時点より前の時点指定することで、復旧処理が正常に完了できるよう、復旧プロセスを初めからやり直すことができます。システムクラッシュなど外的理由により復旧処理が失敗した場合やWALアーカイブがアクセスできなくなった場合、復旧処理を単に再起動させることができます。この場合は失敗した時点とほぼ同じところから再開します。復旧処理の再起動は、次のような通常操作時のチェックポイント処理とほぼ同様に動作します。サーバは定期的にすべての状態をディスクに強制し、再度スキャンする必要がない処理済みのWALデータを示す`pg_control`ファイルを更新します。

24.3.5. タイムライン

過去のある時点までデータベースを復旧できる機能は、SFタイムトラベルやパラレルユニバース（ブラックホールの先の対の宇宙）に類似した、多少の複雑性があります。例えば、データベースの元の履歴で、火曜日の夕方5:15PMに重要なテーブルを削除し、水曜日のお昼まで手違いに気が付かなかったとします。慌てずに、バックアップを取り出して、火曜日の夕方5:14PMの時点にリストアし、データベースを起動させます。データベース世界のこの履歴では、そのテーブルを削除していません。しかし、後になって、これは大した問題ではなかったことが分かり、元の履歴における水曜日に朝の何時かにまで戻したいと考えたと仮定しましょう。データベースは既に起動していますので、元に戻したい時点に至るWALセグメントファイルの一部は上書きされていて、戻すことはできないかもしれません。ですので、このことを避けるために、ポイントインタイムで復旧させた後に生成された一連のWAL記録と元のデータベースの履歴において生成されたWAL記録とを区別する必要があります。

こうした問題を扱うためにPostgreSQLにはタイムラインという概念があります。アーカイブ復旧が完了したときはいつでも、その復旧後に生成されたWAL記録を識別するための新しいタイムラインが生成されます。タイムラインID番号はWALセグメントファイル名の一部です。ですので、新しいタイムラインはこれまでのタイムラインで生成されたWALデータを上書きしません。実際、多くの異なるタイムラインをアーカイブすることができます。不要な機能と考えるかもしれませんが、命綱になることがしばしばあります。どの時点まで復旧すればよいか確実にないといった状況を考えてみてください。その時は、過去の履歴からの分岐点として最善の時点を見つけるために、試行錯誤して何度もポイントインタイムの復旧を行う必要があるでしょう。タイムラインがないと、この手続きはすぐに管理不能な混乱を招いてしまいます。タイムラインを使用して、以前捨てたタイムライン分岐における状態を含む、過去の任意の状態に復旧させることができます。

新しいタイムラインが生成される度に、PostgreSQLは、どのタイムラインがいつどこから分岐したかを示す"タイムライン履歴"ファイルを作成します。この履歴ファイルは、複数のタイムラインを含むアーカイブ場所から復旧する時にシステムが正しいWALセグメントファイルを選択できるようにするために必要です。したがって、履歴ファイルは、WALセグメントファイル同様にWALアーカイブ領域にアーカイブされます。履歴ファイルは（巨大になるセグメントファイルとは異なり）単なる小さなテキストファイルですので、安価かつ適切に無期限で保管できます。必要ならば、履歴ファイルにコメントを追加し、この特定のタイムラインがどのように、なぜ生成されたかについて独自の注釈を付与することができます。特にこうしたコメントは、実験の結果いくつかのタイムラインのもつれがある場合に有用です。

復旧処理のデフォルトは、ベースバックアップが取得された時点のタイムラインと同一のタイムラインに沿った復旧です。別の子タイムラインに沿って復旧させたい（つまり、復旧試行以降に生成されたある状態に戻りたい）場合は`recovery.conf`で対象のタイムラインIDを指定しなければなりません。ベースバックアップより前に分岐したタイムラインに沿って復旧することはできません。

24.3.6. ヒントと例

継続的アーカイブを構成するいくつかのヒントを以下にあげます。

24.3.6.1. スタンドアローンホットバックアップ

スタンドアローンホットバックアップを形成するためPostgreSQLのバックアップ基盤を使用することができます。これらのバックアップはポイントインタイムリカバリに使用することはできないのですが、`pg_dump`によるダンプよりバックアップとリストアが概してより速く行われます。（同時に`pg_dump`のダンプよりより大きくなるので、場合によっては速度による利点が打ち消されるかもしれません。）

ベースバックアップと同様に、スタンドアロンホットバックアップを作成する最も簡単な方法は [pg_basebackup](#) ツールを使用する方法です。実行時に`-x`オプションをつけることでバックアップに必要な全てのトランザクションログを自動的にバックアップに含めることができ、リストアするときには特に特別な作業を行う必要がありません。

より柔軟なバックアップファイルが必要な場合、スタンドアローンホットバックアップと同様に低レベルのプロセスを使うこともできます。低レベルのスタンドアローンホットバックアップを取得するためには、準備として`wal_level`を`archive`もしくは`hot_standby`にセットし、`archive_mode`を`on`にセットし、`switch` ファイルが存在する時のみに実行される`archive_command`をセットします。例：

```
archive_command = 'test ! -f /var/lib/pgsql/backup_in_progress || (test ! -f /var/lib/pgsql/archive/%f && cp %p /var/lib/pgsql/archive/%f)'
```

このコマンドは`/var/lib/pgsql/backup_in_progress` が存在する時のみ実行され、存在しない時は静かに0の退出コードを返します。(PostgreSQLに必要な無いWALファイルを再利用することを許可します)

この準備によって、バックアップは以下のようなスクリプトを使用して取得されます。

```
touch /var/lib/pgsql/backup_in_progress
psql -c "select pg_start_backup('hot_backup');"
tar -cf /var/lib/pgsql/backup.tar /var/lib/pgsql/data/
psql -c "select pg_stop_backup();"
rm /var/lib/pgsql/backup_in_progress
tar -rf /var/lib/pgsql/backup.tar /var/lib/pgsql/archive/
```

完全なWALファイルのアーカイブが行われるように、スイッチファイル、`/var/lib/pgsql/backup_in_progress`が最初に作成されます。バックアップの後、スイッチファイルは削除されます。その後、アーカイブされたWALファイルは、基幹バックアップとすべての必要なWALファイル共に同じtarファイルの一部になるよう、バックアップに追加されます。バックアップスクリプトにエラー処理を加えておくことを思い出してください。

24.3.6.2. 圧縮アーカイブログ

もし、アーカイブのストレージ容量に懸念がある場合、アーカイブファイルを圧縮するためにgzipを使用することもできます。

```
archive_command = 'gzip < %p > /var/lib/pgsql/archive/%f'
```

リカバリ中は `gunzip`を使う必要があります。

```
restore_command = 'gunzip < /mnt/server/archivedir/%f > %p'
```

24.3.6.3. archive_commandスクリプト

`postgresql.conf` の記入事項が以下のように簡素となるため、多くの人が`archive_command`の定義にスクリプトの使用を選択します。

```
archive_command = 'local_backup_script.sh "%p" "%f"'
```

アーカイブ処理手順において単一ではなくそれ以上の数のコマンドを使用したい場合はいつでも、別のスクリプトファイルの使用が推奨されます。そうするとスクリプト内で全ての複雑性が管理されます。スクリプトはbashまたはperlのようなよくあるスクリプト言語で記載できます。

スクリプト内で解決される要件の例として以下があります。

- セキュアなオフサイトデータストレージへのデータのコピー
- 一回に全てではなく3時間毎に転送されるようにWALファイルのバッチ
- その他のバックアップとリカバリーソフトウェアとのインターフェイス
- エラー報告を行う監視ソフトとのインターフェイス

ティップ: `archive_command` スクリプトを使うときは[logging_collector](#)を使えるようにすることが望ましい方法です。そのスクリプトはメッセージをstderrで書き出し、それはデータベースのサーバーログとして書かれます。このためスクリプトでエラーが発生した時に簡単に検知するための細かな設定を行なうことができるからです。

24.3.7. 警告

本ドキュメント作成時点では、継続的アーカイブ技術にいくつかの制限があります。将来のリリースでは修正されるはずですが。

- ハッシュインデックスに対する操作は現在WALログに残りません。したがって、再生してもこれらのインデックスは更新されません。これは、新しい挿入はインデックスにより無視され、更新された行は見かけ上消え、削除された行はポイントを保持し続けるということです。すなわち、ハッシュインデックス付きのテーブルを変更すると、スタンバイサーバでは不正確な問い合わせ結果が得られます。復旧処理が終わった後に手作業でそうしたインデックスそれぞれに対して[REINDEX](#)を行うことが推奨されます。

もしもベースバックアップが行われている時、[CREATE DATABASE](#) コマンドが実行され、ベースバックアップが処理を実行している期間に `CREATE DATABASE` がコピーしている template データベースが変更されると、復旧処理はこれらの変更を作成されたデータベースにも同時に伝播させることは確実です。もちろん、これは望まれる事ではありません。この危険を回避するには、ベースバックアップ期間中にはすべての template データベースを変更しないことが一番です。

- [CREATE TABLESPACE](#) コマンドはリテラルの絶対パス付で WAL にログが記録され、したがって、同じ絶対パスでのテーブル空間作成の時に再生されます。これは、もしログが異なったマシン上で再生される場合には好ましくありません。ログ再生がたとえ同一のマシンであっても、新規のデータディレクトリであれば危険です。なぜなら、再生は元のテーブル空間の内容を上書きし続けるからです。この種の潜在的な振舞いを防ぐためには、テーブル空間を作成もしくは削除後に新規ベースバックアップを行うのが最良の手段です。

また、デフォルトの WAL フォーマットは数多くのディスクページのスナップショットを含んでいるため、かなりかさばるものになってしまっていることに触れておくべきでしょう。これらのページスナップショットは、クラッシュから回復のために設計されています。それというのも、回復処理の際には不完全に書き込まれているディスクページを修復しなければならないことがあるからです。システムのハードウェアやソフトウェアによっては、不完全なディスクページの書き込みが起きてしまう危険性は無視してもよい程微小です。この場合 [full_page_writes](#) パラメータを設定してページスナップショットを無効にすることで、アーカイブされたログの総容量を大幅に縮小できます（実際に設定を行う前に、[第29章](#)の注意事項と警告を読んでください）。ページスナップショットを無効にしても PITR 処理の際にログが使用できなくなることはありません。将来の課題は、`full_page_writes` がたとえオンになっている場合であっても不要なページを取り除き、アーカイブ済み WAL データの圧縮を行うことでしょう。差し当たり管理者は、可能な限りチェックポイント間隔パラメータを大きくすることによって、WAL に含まれるページスナップショットの数を削減することができます。

[前のページ](#)

ファイルシステムレベルのバックアップ

[ホーム](#)

[上に戻る](#)

[次のページ](#)

高可用性、負荷分散およびレプリケーション

第 25章高可用性、負荷分散およびレプリケーション

目次

- 25.1. [様々な解法の比較](#)
- 25.2. [ログ SHIPPING スタンバイサーバ](#)
 - 25.2.1. [計画](#)
 - 25.2.2. [スタンバイサーバの動作](#)
 - 25.2.3. [スタンバイサーバのためのマスタの準備](#)
 - 25.2.4. [スタンバイサーバの設定](#)
 - 25.2.5. [ストリーミングレプリケーション](#)
 - 25.2.6. [カスケードレプリケーション](#)
 - 25.2.7. [同期レプリケーション](#)
- 25.3. [フェールオーバー](#)
- 25.4. [その他のログ SHIPPING の方法](#)
 - 25.4.1. [実装](#)
 - 25.4.2. [レコードベースのログ SHIPPING](#)
- 25.5. [ホットスタンバイ](#)
 - 25.5.1. [ユーザのための概説](#)
 - 25.5.2. [問い合わせコンフリクトの処理](#)
 - 25.5.3. [管理者のための概説](#)
 - 25.5.4. [ホットスタンバイパラメータリファレンス](#)
 - 25.5.5. [警告](#)

データベースサーバは共同して稼働できます。その目的は、最初のサーバが故障したとき次のサーバへ速やかに引き継ぎができること（高可用性）および複数のコンピュータが同一のデータを処理できること（負荷分散）です。データベースサーバがシームレスに共同稼働できれば理想的です。静的なウェブページを提供するウェブサーバは、ウェブからの要求で生ずる負荷を複数のマシンに分散するだけで、簡単に結合できます。実際、読み取り専用のデータベースサーバの結合は、同じようになりやすいです。しかし、大部分のデータベースサーバは、読み書きの混在した要求を受け取り、読み書き両用のサーバの結合はとても困難です。なぜなら、読み取り要求だけの場合、全サーバへのデータの配布は1回で終わります。しかし、書き込み後の読み取り要求に対して一貫性のある結果を返すためには、書き込み要求を受けたサーバだけでなく、他の全サーバにおいてもデータに書き込まなければなりません。

この同時性を持たせるという問題は、共同して稼働するサーバにおいて根本的に困難なものです。すべての使用状況において、単一の解法を用いて同時性の問題の影響を軽減できないため、複数の解法が存在します。各々の解法はこの問題に異なったやり方を適用し、固有の作業負荷に対する影響を最小化します。

幾つかの解法では、1つのサーバだけにデータの更新を許可することにより、同時性を持たせています。データの更新ができるサーバを、読み書きサーバ、マスタサーバまたはプライマリサーバといいます。マスタの変更を追跡するサーバを、スタンバイサーバまたはスレーブサーバといいます。マスタサーバに昇格するまで接続できないスタンバイサーバをウォームスタンバイサーバといいます。接続を受理できて読み取り専用の問い合わせを処理できるスタンバイサーバをホットスタンバイサーバといいます。

いくつかの同期の解法が提供されています。すなわち、データに書き込むトランザクションでは、全サーバがコミットするまでトランザクションはコミットされません。これによって、フェールオーバーにおいてデータの消失がないことが保証されます。また、どのサーバが問い合わせを受理したかに関係なく、全ての負荷分散サーバが一貫した結果を返すことが保証されます。それに対して非同期の解法では、コミット時刻と他サーバへの伝達時刻に時間差がありうるため、バックアップサーバへ交代する時にトランザクションが消失する可能性があります。また、負荷分散サーバにおいては、最新でない結果を応答する可能性があります。サーバ間の非同期の通信は、同期が非常に低速な場合に使用されます。

解法は粒度によって分類することもできます。ある解法ではデータベースサーバ全体だけを範囲として処理しますが、他の解法では各テーブルまたは各データベースを範囲として管理できます。

すべての選択において、作業効率を考えなければなりません。通常、作業効率と機能性は相反する関係にあります。例えば、遅いネットワークの場合、完全同期の解法を使えば作業効率は半分以下となりますが、非同期の解法を使えば作業効率への影響が最小となります。

本節では、フェールオーバーとレプリケーションと負荷分散における種々の解法を説明します。 [glossary](#) も利用できます。

25.1. 様々な解法の比較

共有ディスクを用いたフェールオーバー

データベースのコピーを 1つだけ保有すればよいと、共有ディスクを用いたフェールオーバーは同期によるオーバーヘッドを回避できます。本解法では、複数のサーバが単一のディスクアレイを共有します。主データベースサーバが故障したとき、まるでデータベースの破損から復旧したように、スタンバイサーバが元のデータベースを実装して稼働できます。これはデータの消失がない高速なフェールオーバーを行うことができます。

ハードウェアを共有するという機能は、ネットワーク上の記憶装置では一般的です。ネットワークファイルシステムの利用も可能ですが、そのファイルシステムがPOSIX仕様を満たしているか注意してください。（[項17.2.1](#)を見てください）。本解法には重大な制約があり、共有ディスクアレイが故障または破損したとき、プライマリサーバもスタンバイサーバも機能しなくなります。また、プライマリサーバが稼働している間は、スタンバイサーバが共有記憶装置にアクセスしてはなりません。

ファイルシステム（ブロックデバイス）レプリケーション

ハードウェア共有機能の改善の一つとしてファイルシステムのレプリケーションをあげることができます。それは、あるファイルシステムに対して行われたすべての変更を他のコンピュータに存在するファイルシステムにミラーリングします。制約はただ一つであり、スタンバイサーバがファイルシステムの一貫したコピーを自身の領域に持つようにミラーリングしなければなりません。具体的には、スタンバイサーバへの書き込みがマスターサーバへの書き込みと同じ順序でおこなわれなければなりません。LinuxにおけるDRBDは、ファイルシステムレプリケーションで広く受け入れられている手法です。

トランザクションログ SHIPPING

ウォームスタンバイおよびホットスタンバイサーバは、ログ先行書き込み（WAL）のレコードを解読して最新の状態を保持できます。プライマリサーバが故障したとき、スタンバイサーバがプライマリサーバのほぼすべてのデータを保存して、速やかに新しい主データベースを稼働できます。本解法は同期、非同期で行うことができ、データベース全体だけを範囲として処理できます。

スタンバイサーバは、ファイル単位のログ SHIPPING（[項25.2](#)参照）またはストリーミングレプリケーション（[項25.2.5](#)参照）または両者の併用を使用して実装できます。ホットスタンバイの情報は [項25.5](#) を参照してください。

トリガベースのマスタ・スタンバイレプリケーション

マスタとスタンバイによるレプリケーションでは、データ更新のすべての問い合わせをマスタサーバに送付します。マスタサーバは更新したデータを非同期でスタンバイサーバに送付します。マスタサーバが稼働している間、スタンバイサーバは読み取り問い合わせだけに応答します。スタンバイサーバはデータウェアハウスへの問い合わせに理想的です。

この種類のレプリケーションの一例はSlony-Iであり、テーブル単位の粒度を持ち、複数のスタンバイサーバが稼働できます。（バッチ処理によって）スタンバイサーバのデータを非同期で更新するため、フェールオーバーにおけるデータ消失の可能性あります。

文に基づいたレプリケーションのミドルウェア

文に基づいたレプリケーションのミドルウェアでは、プログラムがすべてのSQL問い合わせを採取して、1つまたはすべてのサーバに送付します。なお、各々のサーバは独立して稼働します。読み書き問い合わせは、すべてのサーバがすべての変更を受け取るように全サーバに送付されなければなりません。しかし、読み取り専用の問い合わせはサーバ全体の読み取り負荷を分散させるために、1つのサーバだけに送付することができます。

問い合わせを修正しないで送付した場合、`random()` 関数による乱数値と `CURRENT_TIMESTAMP` 関数による現在時刻およびシーケンス値が、サーバごとに異なることがあります。その理由は、各サーバが独立して稼働しているため、および、SQL問い合わせの送付では実際に更新した行の識別値を取得できないためです。これが許容できない場合は、ミドルウェアかアプリケーションにおいて1つのサーバにこのような問い合わせを送付し、その結果を書き込み問い合わせで使用しなければなりません。その他の選択肢は従来のマスタとスタンバイによるレプリケーションのオプションを使用するものです。すなわち、データ更新の問い合わせをマスタサーバだけに送付し、ミドルウェアによるレプリケーションを使わずにマスタとスタンバイによるレプリケーションを介してスタンバイサーバに伝達します。トランザクションをコミットするか中断するかについても、全サーバが同一となるよう注意しなければなりません。これには2相コミット（[PREPARE TRANSACTION](#) および [COMMIT PREPARED](#)）を使用することになるでしょう。Pgpool-IIとContinuent Tungstenがこのレプリケーションの一例です。

非同期マルチマスタレプリケーション

ラップトップやリモートマシンのように、通常は接続されていないサーバ間において、データの一貫性を保持することは挑戦的な課題です。非同期マルチマスタレプリケーションの使用により、全サーバの独立した稼働、およびトランザクションの衝突を識別するための定期的な通信を実現します。トランザクションの衝突は、利用者および衝突回避法によって解決できるでしょう。Bucardoはこの種のレプリケーションの一例です。

同期マルチマスタレプリケーション

同期マルチマスタレプリケーションでは全てのサーバが書き込み要求を受理できます。受理したサーバは更新したデータを、トランザクションをコミットする前に、他の全サーバへ配布します。書き込み負荷が重いとき、ロックの掛かり過ぎによる作業効率の低下の原因となります。実際、書き込み効率は単一サーバより悪いことが大半です。読み取り要求はどのサーバにも送付できます。通信による負荷を減らすには、共有ディスクが実装されます。同期マルチマスタレプリケーションは、主に読み取り作業負荷の低減に最適ですが、全てのサーバが書き込み要求を受理できることも大きな利点です。その利点とは、マスタとスタンバイ間で作業負荷を分けなくてよいこと、および更新データが1つのサーバから他のサーバに送付されるため、出力が確定しないrandom()関数などによる問題が起こらないことです。

PostgreSQL では、この種類のレプリケーションを提供しません。しかし、PostgreSQL の 2相コミット ([PREPARE TRANSACTION](#)および[COMMIT PREPARED](#)) を使用すれば、アプリケーションのコードまたはミドルウェアにおいて本解法を実装できます。

商業的な解法

PostgreSQL はオープンソースであり、容易に拡張できます。そのため多数の企業が、PostgreSQLを取り入れて商業的な解法を作成し、フェールオーバーとレプリケーションと負荷分散の機能を独自に実現していますが、ソースコードは非公開です。

表25-1は上述した種々の解法の機能を要約したものです。

表 25-1. 高可用性、負荷分散およびレプリケーションの特徴

特徴	共有ディスクを用いたフェールオーバー	ファイルシステムのレプリケーション	トランザクションログ.shipping	マスタとスタンバイによるトリガに基づいたレプリケーション	文に基づいたレプリケーションのミドルウェア	非同期マルチマスタレプリケーション	同期マルチマスタレプリケーション
最も一般的な実装	NAS	DRBD	ストリーミングレプリケーション	Slony	pgpool-II	Bucardo	
通信方法	共有ディスク	ディスクブロック	WAL	テーブル行	SQL	テーブル行	テーブル行および行ロック
特別なハードウェアが不要		○	○	○	○	○	○
複数のマスタサーバが可能					○	○	○
マスタサーバにオーバヘッドがない	○		○		○		
複数のスレーブサーバを待たない	○		同期が無効の場合	○		○	
マスタの故障によるデータ損失がない	○	○	同期が有効の場合		○		○
スタンバイは読み取り問い合わせだけを受理			ホットスタンバイ使用時	○	○	○	○
テーブルごとの粒度となる				○		○	○
コンフリクトの回避が不要	○	○	○	○			○

上の分類に該当しない解法もあります。

データの分割

データの分割とは、同じテーブルのデータを複数部分に分けることです。各部分に書き込むことができるのは、1つのサーバだけです。例えば、データをロンドンとパリの営業所用に分割でき、サーバをロンドンとパリのどちらにも設置できた状態を考えます。問い合わせにロンドンとパリのデータが混在した場合、アプリケーションは両方のサーバに問い合わせることができます。または、マスタスタンバイレプリケーションを使用して、他の営業所のデータを読み取り専用コピーとして保持できます。

複数サーバによる問い合わせの並列実行

上述した多くの解法は、複数のサーバが複数の問い合わせを処理するものです。処理速度の向上のために、単一の問い合わせに複数のサーバを使用するものではありません。本解法は複数のサーバが単一の問い合わせを共同して実行するものです。その方法は、データをサーバ間で分割し、各サーバが部分的に問い合わせを実行し、各々の結果をプライマリサーバに送付し、プライマリサーバが合体して利用者に返送するものです。Pgpool-IIはこの性能を持っています。また、PL/Proxyツールセットを使用して実装できます。

[前のページ](#)

高可用性、負荷分散およびレプリケーション

[ホーム](#)
[上に戻る](#)

[次のページ](#)
ログ SHIPPING スタンバイサーバ

25.2. ログ SHIPPING スタンバイサーバ

継続的なアーカイブ処理を使用して、プライマリサーバが失敗した場合に操作を引き継ぐ準備がなされた、1つ以上のスタンバイサーバを持つ高可用性(HA)クラスタ構成を作成することができます。この機能はウォームスタンバイまたはログ SHIPPING として広く知られています。

プライマリサーバとスタンバイサーバは、この機能を提供するために共同して稼動しますが、サーバとサーバはゆるく結合しています。プライマリサーバは継続的アーカイブモードで動作し、各スタンバイサーバはプライマリからWALファイルを読み取る、継続的リカバリモードで動作します。この機能を可能にするために、データベースのテーブル変更は不要です。したがって、他のレプリケーションの解法に比べて、管理にかかるオーバーヘッドが減少します。この構成はプライマリサーバの性能への影響も相対的に減少させます。

あるデータベースサーバから他へ直接WALレコードを移動することは通常、ログ SHIPPING と説明されます。PostgreSQLはファイルベースのログ SHIPPING を実装します。つまりWALレコードはある時点で1つのファイル(WALセグメント)として送信されることを意味します。WALファイル(16MB)は隣り合うシステム、同じサイトの別システム、地球の裏側のシステムなど距離に関わらず、簡単かつ安価に送付することができます。この技法に必要な帯域幅はプライマリサーバのトランザクションの頻度に応じて変動します。レコードベースのログ SHIPPING はより粒度を細かくしたもので、ネットワーク接続を介してWALの変更を増分的に流します ([項25.2.5参照](#))。

ログ SHIPPING が非同期であることに注意しなければなりません。つまり、WALレコードはトランザクションがコミットした後には転送されます。結果として、プライマリサーバが災害などの致命的な失敗を受けた場合、送信されていないトランザクションが失われますので、データを損失する空白期間があります。ファイルベースのログ SHIPPING におけるデータ損失の空白期間量を `archive_timeout` パラメータを用いて制限することができます。これは数秒程度まで小さく設定することができます。しかし、低く設定するとファイル転送に必要な帯域幅が増大します。ストリーミングレプリケーション ([項25.2.5参照](#)) により、データを損失する期間を非常に小さくすることができます。

リカバリ処理の性能は十分よく、一度実施されれば、スタンバイサーバが完全な状態から逸脱するのは一時的にしかすぎません。結果としてこれは、高可用性を提供するウォームスタンバイ構成と呼ばれます。保管されたベースバックアップからサーバをリストアし、ロールフォワードを行うことはおそらく長時間かかりますので、これは高可用性のための解法とはいえ、災害からのリカバリのための解法です。スタンバイサーバは読み取り専用の問い合わせに使用することもできます。この場合ホットスタンバイサーバと呼ばれます。詳細については[項25.5](#)を参照してください。

25.2.1. 計画

プライマリサーバとスタンバイサーバを、少なくともデータベースサーバという見地でできる限り同じになるように作成することを通常勧めます。具体的には、テーブル空間に関連するパス名はそのまま渡されますので、テーブル空間機能を使用する場合には、プライマリとスタンバイサーバの両方でテーブル空間用のマウントパスを同じにしておかなければなりません。[CREATE TABLESPACE](#)をプライマリで実行する場合、そのコマンドを実行する前に必要な新しいマウントポイントをプライマリとすべてのスタンバイサーバで作成しなければならないことに注意してください。ハードウェアをまったく同じにする必要はありませんが、経験上アプリケーションとシステムの運用期間に渡って2つの同じシステムを管理する方が、異なる2つのシステムを管理するよりも簡単です。いずれにしてもハードウェアアーキテクチャは必ず同じでなければなりません。例えば32ビットシステムから64ビットシステムへの SHIPPING は動作しません。

マイナーリリースの更新ではディスク書式を変更しないというのがPostgreSQLグローバル開発グループの方針ですので、プライマリサーバとスタンバイサーバとの間でマイナーリリースレベルの違いがあってもうまく動作するはずですが、しかし、この場合、公的なサポートは提供されません。できる限りプライマリサーバとスタンバイサーバとで同じリリースレベルを使用してください。新しいマイナーリリースに更新する場合、もっとも安全な方針はスタンバイサーバを先に更新することです。新しいマイナーリリースは以前のマイナーリリースのWALファイルを読み込むことはできますが、逆はできないかもしれません。

25.2.2. スタンバイサーバの動作

スタンバイモードでは、サーバは継続的にマスタサーバから受け取ったWALを適用します。スタンバイサーバはWALアーカイブ([restore command](#)参照)から、または直接TCP接続(ストリーミングレプリケーション)を介してマスタサーバから、WALを読み取ることができます。またスタンバイサーバはスタンバイクラスタの `pg_xlog` ディレクトリにあるすべてのWALをリストアしようと試みます。これはよくサーバの再起動後、スタンバイが再起動前にマスタから流れ込んだWALを再生する時に発生します。しかしまたファイルを再生する任意の時点で、手作業で `pg_xlog` にコピーすることもできます。

起動時、スタンバイサーバは `restore_command` を呼び出して、アーカイブ場所にある利用可能なすべてのWALをリストアすることから始めます。そこで利用可能なWALの終端に達し、`restore_command` が失敗すると、`pg_xlog` ディレクトリにある利用可能な任意のWALのリストアを試みます。ストリーミングレプリケーションが設定されている場合、これに失敗すると、スタンバイはプライマリサーバへの接続を試み、アーカイブまたは `pg_xlog` 内に存在した最終の有効レコードからWALのストリーミングを

開始します。ストリーミングレプリケーションが未設定時にこれに失敗する場合、または、接続が後で切断される場合、スタンバイは最初に戻り、アーカイブからのファイルのリストアを繰り返し行います。このアーカイブ、pg_xlog、ストリーミングレプリケーションからという再試行の繰り返しはサーバが停止する、あるいはトリガファイルによるフェールオーバーが発行されるまで続きます。

pg_ctl promoteが実行された時またはトリガファイル(trigger_file)が存在する時、スタンバイモードは終了し、サーバは通常の動作に切り替わります。フェールオーバーの前に、アーカイブまたはpg_xlog内の即座に利用可能なWALをすべてリストアします。しかし、マスタへの接続を行おうとはしません。

25.2.3. スタンバイサーバのためのマスタの準備

[項24.3](#)で説明したように、スタンバイからアクセス可能なアーカイブディレクトリに対してプライマリで継続的なアーカイブを設定してください。このアーカイブ場所はマスタが停止した時であってもスタンバイからアクセス可能でなければなりません。つまり、マスタサーバ上ではなく、スタンバイサーバ自身上に存在するか、または他の高信頼性サーバ上に存在しなければなりません。

ストリーミングレプリケーションを使用したい場合、スタンバイサーバ(複数可)からのレプリケーション接続を受け付けるようにプライマリサーバで認証を設定してください。つまり、ロールを作成し適切な項目を提供、あるいは、そのデータベースフィールドとしてreplicationを持つ項目をpg_hba.conf内に設定してください。また、プライマリサーバの設定ファイルにおいてmax_wal_sendersが十分大きな値に設定されていることを確認してください。

[項24.3.2](#)に記述したように、スタンバイサーバの再起動のために、ベースバックアップを取得してください。

25.2.4. スタンバイサーバの設定

スタンバイサーバを設定するためには、プライマリサーバから取得したベースバックアップをリストアしてください([項24.3.4](#)参照)。スタンバイのクラスタデータディレクトリ内にrecovery.confリカバリコマンドファイルを作成し、standby_modeを有効にしてください。WALアーカイブからファイルをコピーする簡単なコマンドをrestore_commandに設定してください。高可用性のために複数のスタンバイサーバを持たせようとしている場合、recovery_target_timelineをlatestに設定し、スタンバイサーバが他のスタンバイにフェールオーバーする時に発生するタイムラインの変更に従うようにします。

注意: ここで説明した組込みのスタンバイモードといっしょにpg_standbyや類似ツールを使用しないでください。restore_commandはファイルが存在しない場合に即座に終了しなければなりません。サーバが必要に応じてそのコマンドを再度実行します。pg_standbyのようなツールを使用するためには[項25.4](#)を参照してください。

ストリーミングレプリケーションを使用したい場合には、ホスト名(またはIPアドレス)とプライマリサーバとの接続に必要な追加情報を含む、libpq接続文字列でprimary_conninfoを記述してください。プライマリで認証用のパスワードが必要な場合はprimary_conninfoにそのパスワードも指定する必要があります。

スタンバイサーバを高可用性を目的に設定しているのであれば、スタンバイサーバはフェールオーバーの後プライマリサーバとして動作しますので、プライマリサーバと同様にWALアーカイブ処理、接続、認証を設定してください。

WALアーカイブを使用している場合、[archive_cleanup_command](#)パラメータを使用してスタンバイサーバで不要となったファイルを削除することで、その容量を最小化することができます。特にpg_archivecleanupユーティリティは、典型的な単一スタンバイ構成([pg_archivecleanup](#)参照)におけるarchive_cleanup_commandと共に使用されるように設計されています。しかし、バックアップを目的にアーカイブを使用している場合には、スタンバイから必要とされなくなったファイルであっても、最新のベースバックアップの時点からリカバリするために必要なファイルを保持しなければならないことに注意してください。

recovery.confの簡単な例を以下に示します。

```
standby_mode = 'on'
primary_conninfo = 'host=192.168.1.50 port=5432 user=foo password=foopass'
restore_command = 'cp /path/to/archive/%f %p'
archive_cleanup_command = 'pg_archivecleanup /path/to/archive %r'
```

スタンバイサーバの台数に制限はありませんが、ストリーミングレプリケーションを使用するなら、プライマリサーバに同時に接続できるようにmax_wal_sendersを十分な数に設定してください。

25.2.5. ストリーミングレプリケーション

ストリーミングレプリケーションによりスタンバイサーバはファイルベースのログ SHIPPING よりもより最近の状態を維持できるようになります。スタンバイは、WALレコードが生成された時にWALファイルがいっぱいになるまで待機せずにWALレコードをスタンバイに流し出すプライマリと接続します。

ストリーミングレプリケーションはデフォルトで非同期で、([項25.2.7](#)参照) この場合、プライマリでトランザクションがコミッ

トされてから、その変更がスタンバイ側で参照可能になるまでの間にわずかな遅延がまだあります。しかし、この遅延はファイルベースのログ SHIPPING よりも非常に小さなもので、ロードを維持できる程度の能力があるスタンバイであれば通常は1秒以下です。ストリーミングレプリケーションでは、データ損失期間を減らすための `archive_timeout` を必要としません。

ファイルベースの継続的アーカイブのないストリーミングレプリケーションを使用している場合、マスタにおいて `wal_keep_segments` の値を、スタンバイが古いWALセグメントに追従する必要がある時にそれらがあまりに早く回収されないことが確実になるように、十分大きく設定しなければなりません。スタンバイ側が大きく遅れてしまった場合、新しいベースバックアップから再初期化する必要があります。スタンバイからアクセスできるようにWALアーカイブを設定する場合は、スタンバイは常に追従するためにアーカイブを使用することができますので、`wal_keep_segments` は必要ありません。

ストリーミングレプリケーションを使用するためには、[項25.2](#)の説明のようにファイルベースのログ SHIPPING を行うスタンバイサーバを設定してください。ファイルベースのログ SHIPPING を行うスタンバイをストリーミングレプリケーションを行うスタンバイに切り替える手順は、`recovery.conf` 内の `primary_conninfo` 設定をプライマリサーバを指し示すように設定することです。スタンバイサーバがプライマリサーバ上の `replication` 疑似データベースに接続できる([項25.2.5.1](#)参照)ように、プライマリで `listen_addresses` と認証オプション(`pg_hba.conf` 参照)を設定してください。

キープアライブソケットオプションをサポートするシステムでは、`tcp_keepalives_idle`、`tcp_keepalives_interval` および `tcp_keepalives_count` を設定することで、プライマリの接続切断の即時検知に有用です。

スタンバイサーバからの同時接続数の最大値を設定してください（詳細は `max_wal_senders` を参照）。

スタンバイが起動し、`primary_conninfo` が正しく設定されると、スタンバイはアーカイブ内で利用可能なWALファイルをすべて再生した後にプライマリと接続します。接続の確立に成功すると、スタンバイでWAL受信プロセスが存在し、プライマリで対応するWAL送信プロセスが存在します。

25.2.5.1. 認証

信頼できるユーザのみがWALストリームを読み取ることができるように、レプリケーション用のアクセス権を設定することは非常に重要です。WALから機密情報を取り出すことは簡単だからです。スタンバイサーバはプライマリに対してプライマリのスーパーユーザが `REPLICATION` 権限を持つアカウントとして認証されなければなりません。レプリケーションのための `REPLICATION` 権限 と `LOGIN` 権限を持つ専用のユーザを作成することをお勧めします。`REPLICATION` 権限は非常に強力な権限なので、`SUPERUSER` のようにプライマリのデータを変更することを許可されていません。

レプリケーション用のクライアント認証は `pg_hba.conf` 内でその `database` フィールドに `replication` を指定したレコードで制御されます。例えば、スタンバイがIPアドレス `192.168.1.100` のホストで稼動し、レプリケーション用のアカウントの名前が `foo` である場合、管理者はプライマリ上の `pg_hba.conf` に以下の行を追加することができます。

```
# 利用者 foo のホスト 192.168.1.100 からプライマリサーバへのレプリケーションスタンバイとしての接続を
# 利用者のパスワードが正しく入力されたならば許可する
#
# TYPE      DATABASE      USER      ADDRESS      METHOD
host       replication      foo       192.168.1.100/32      md5
```

プライマリサーバのホスト名とポート番号、接続する利用者名およびパスワードは、`recovery.conf` ファイルで指定します。パスワードはスタンバイサーバの `~/.pgpass` ファイルでも設定できます（`database` フィールドの `replication` を指定します）。例えば、プライマリサーバが稼動するホストのIPアドレスが `192.168.1.50` でポート番号が `5432` であり、レプリケーションのアカウント名が `foo` であり、パスワードが `foopass` である場合、管理者はスタンバイサーバの `recovery.conf` ファイルに次行を追加できます。

```
# プライマリサーバが 192.168.1.50 のホストの 5432 ポートで稼動し
# 利用者名が foo でパスワードが foopass とする
primary_conninfo = 'host=192.168.1.50 port=5432 user=foo password=foopass'
```

25.2.5.2. 監視

ストリーミングレプリケーションの重要な健全性尺度は、プライマリサーバで生成されたがスタンバイサーバではまだ適用されていないWALレコードの量です。プライマリサーバの現在のWAL書き込み位置とスタンバイサーバの受理したWALの最終位置を比較すれば、この遅延を計算できます。これらの位置は、プライマリサーバでは `pg_current_xlog_location` を、スタンバイサーバでは `pg_last_xlog_receive_location` を使用すれば検索できます（詳細は [表9-59](#) および [表9-60](#) を参照）。スタンバイサーバの最終位置は、`ps` コマンドを使用してWAL受理プロセスの状態としても表示できます（詳細は [項27.1](#) を参照）。

`pg_stat_replication` ビューを介してWAL送信処理プロセスのリストを入手することができます。

`pg_current_xlog_location` と `sent_location` フィールドとの違いが大きい場合、マスタサーバが高負荷状態であることを示している可能性があります。一方でスタンバイサーバ上の `sent_location` と `pg_last_xlog_receive_location` の値の差異は、ネットワーク遅延、またはスタンバイが高負荷状態であることを示す可能性があります。

25.2.6. カスケードレプリケーション

カスケードレプリケーションは、リレーのような振る舞い、つまり、スタンバイサーバから他のスタンバイにレプリケーション接続し、WALレコードを送信することができます。マスターサーバへ直接の接続を減らしたり、サイト相互の帯域オーバーヘッドを最小化するために使用することができます。

カスケードスタンバイとして知られているとおり、スタンバイは受け取り手としても送り手としても振る舞うことができます。よりマスターサーバに近いスタンバイサーバは上流サーバと呼ばれるのに対し、より遠いスタンバイサーバは下流サーバと呼ばれます。カスケードレプリケーションには下流サーバの数に制限は設定されていません。しかし、どのスタンバイサーバも最終的には1つのマスター/プライマリサーバに繋がる1つの上流サーバに接続します。

カスケードスタンバイはマスターから受け取ったWALレコードだけでなく、アーカイブからリストアしたWALアーカイブも送信します。このため、レプリケーション接続が上流サーバで切断しても、ストリーミングレプリケーションは下流サーバへ新しいWALレコードがある限り継続します。

カスケードレプリケーションは現時点では非同期です。同期レプリケーション（参照[項25.2.7](#)）の設定は現時点でカスケードレプリケーションへは影響を与えません。

ホットスタンバイはどの様に配置されていても上流に伝播します。

カスケードスタンバイは昇格時、下流サーバのレプリケーションのための接続を即座に切断します。これはスタンバイサーバ間のタイムラインが変化し、これ以上、レプリケーションを継続できないからです。この影響を受けるスタンバイはストリーミングレプリケーションを再び確立するために、再接続する可能性があります。

カスケードレプリケーションを使うためには、カスケードスタンバイをセットアップ、つまり、レプリケーション接続を許可してください。（[max_wal_senders](#)と[hot_standby](#)および、[クライアント認証](#)を設定してください）また、下流スタンバイにカスケードスタンバイであることを知らせるために[primary_conninfo](#)を設定する必要があります。

25.2.7. 同期レプリケーション

PostgreSQLのストリーミングレプリケーションはデフォルトで非同期です。プライマリサーバがクラッシュした場合、コミットされた一部のトランザクションがスタンバイサーバに複製されず、データ損失を引き起こす可能性があります。データ損失量はフェールオーバー時点のレプリケーション遅延に比例します。

同期レプリケーションは、あるトランザクションでなされた変更はすべて、1つの同期スタンバイサーバに転送されていることを確実にする機能を提供します。これはトランザクションコミットで提供される永続性の標準レベルを拡張します。この保護レベルはコンピュータ科学理論では2-safeレプリケーションと呼ばれます。

同期レプリケーションを要求する時、書き込みトランザクションのコミットはそれぞれ、そのコミットがプライマリサーバおよびスタンバイサーバの両方で、ディスク上のトランザクションログに書き込まれたという確認を受けとるまで待機します。データ損失が起こる可能性は、プライマリサーバとスタンバイサーバが同時にクラッシュしてしまった場合のみです。これは非常に高い永続性を提供することができますが、それはシステム管理者が2つのサーバの設置と管理に関して注意を払っている場合のみです。確認のための待機は、サーバがクラッシュした場合でも変更が失われないということでユーザからの信頼性が大きくなりますが、同時に要求するトランザクションの応答時間も必ず大きくなります。最小待機時間はプライマリとスタンバイの間の往復遅延時間です。

読み取り専用のトランザクションおよびトランザクションのロールバックはスタンバイサーバからの応答を待つ必要はありません。副トランザクションのコミットもスタンバイサーバからの応答を待つことはなく、最上位レベルのコミットのみ待機します。データロード処理やインデックス構築など長時間実行される操作は、最終コミットメッセージまで待機しません。準備およびコミットの両方を含め、二相コミット動作はすべてコミット待機を必要とします。

25.2.7.1. 基本設定

一度、ストリーミングレプリケーションが設定されている場合、同期レプリケーションの設定には必要な追加設定は1つだけ：[synchronous_standby_names](#)を空でない値に設定することです。また[synchronous_commit](#)はonに設定されていなければなりませんが、これはデフォルト値ですので、通常は変更する必要はありません。（[項18.5.1](#) および [項18.6.2](#)を参照してください）この設定によりスタンバイがそのコミットレコードを信頼できるストレージに書き込んだことが確認できるまで、各コミットが待たされるようになります。[synchronous_replication](#)は個々のユーザによって設定することができます。このため、トランザクション単位を基準とした永続性の保証を制御するために、設定ファイルの中で特定のユーザまたはデータベースについて設定することも、アプリケーションによって動的に設定することもできます。

コミットレコードがプライマリ上のディスクに書き出された後、WALレコードがスタンバイに送信されます。スタンバイにて[wal_receiver_status_interval](#)がゼロに設定されていない限り、スタンバイは新しいWALデータのバッチがディスクに書き出す度にメッセージを返します。スタンバイが、プライマリ上の[synchronous_standby_names](#)で指定したものと最初に一致するスタンバイである場合、そのスタンバイからの応答メッセージがコミットレコードの受領を確認するまでの待機を解除するために使用されます。これらのパラメータにより管理者はどのスタンバイサーバが同期スタンバイとすべきかを指定することができます。同期レプリケーションの設定は主にマスタでなされることに注意してください。指名されたスタンバイは直接マスターサーバに接続される必要があります。つまり、マスターサーバは下流サーバがカスケードレプリケーションで使用されているか

について何も知りません。

`synchronous_commit`を`remote_write`に設定することで、スタンバイサーバはコミットされたレコードをスタンバイサーバがメモリ上に受け取ったことを確認するまでコミットを待つようになります。しかし、これはスタンバイサーバのディスクへ書きこまれたことを待つわけではありません。これは、`on`と設定するより、提供される永続性は弱くなります。具体的には、スタンバイサーバは(PostgreSQLではなく)オペレーティングシステムがクラッシュした場合にデータを失う可能性があります。しかし、この設定はトランザクションの応答時間を短くすることができます。データの損失は、プライマリサーバとスタンバイサーバが同時にクラッシュし、かつ、プライマリのデータベースが同時に壊れた場合にのみ発生します。

高速シャットダウンが要求された場合、ユーザは待機を停止します。しかし非同期レプリケーションを使用している時、送信中のWALレコードが現在接続しているスタンバイサーバに転送されるまで、サーバは完全に停止しません。

25.2.7.2. 性能に関する考慮

通常、同期レプリケーションは、アプリケーションが満足できる程度に実行されることを確実にするために、注意深くスタンバイサーバを計画し設置しなければなりません。待機のためにシステムリソースを使用することはありませんが、トランザクションロックは転送が確認されるまで継続して保持されます。結果として同期レプリケーションを注意せずに使用すると、応答時間が増加する、および競合がより高くなるため、データベースアプリケーションの性能は低下します。

PostgreSQLではアプリケーション開発者がレプリケーション経由で必要とする永続性レベルを指定することができます。これをシステム全体に対して指定することができますし、特定のユーザ、接続、個々のトランザクションに対してさえ指定することもできます。

例えばアプリケーションの作業量が、重要な顧客詳細の変更が10%、ユーザ間のチャットメッセージなど、あまり重要ではなく、失ったとしても業務をより簡単に戻すことができるようなデータの変更が90% という構成を考えてみます。

(プライマリ上で) アプリケーションレベルで指定する同期レプリケーションオプションを使用して、作業全体を低速化させることなく、最も重要な変更に対して同期レプリケーションを企てることができます。アプリケーションレベルのオプションは、高い性能が求められるアプリケーションで同期レプリケーションの利点が得られる、重要かつ現実的な手段です。

生成されるWALデータの割合よりネットワーク帯域幅が大きくなければならないことを考慮しなければなりません。

25.2.7.3. 高可用性に関する検討

`synchronous_commit`が`on`もしくは`remote_write`に設定された場合、なされたコミットは同期スタンバイの応答まで待機されます。応答は最後のまたは唯一のスタンバイがクラッシュした場合には決して返されません。

データ損失を防止するための最善の解法は、最後に残る同期スタンバイを失わないことを確実にすることです。

`synchronous_standby_names`を使用して複数の潜在的な同期スタンバイを指名することで実現することができます。最初に指名されたスタンバイは同期スタンバイとして使用されます。この後に列挙されたスタンバイは、最初のスタンバイが失敗した場合に同期スタンバイの役割を引き継ぎます。

スタンバイが最初にプライマリに付与された時、それはまだ適切に同期されていません。これは`catchup`モードと呼ばれます。最初にスタンバイとプライマリ間の遅延がゼロになった時に、実時間`streaming`状態に移ります。追従(`catchup`)期間はスタンバイが作成された直後は長くなるかもしれませんが、スタンバイが停止している場合、追従期間はスタンバイの停止期間にしたがって長くなります。スタンバイは、`streaming`状態に達した後でのみ、同期スタンバイになることができます。

コミットが受領通知を待機している間にプライマリが再起動した場合、プライマリデータベースが復旧した後、待機中のトランザクションは完全にコミットされたものと記録されます。すべてのスタンバイがプライマリのクラッシュ時点で送信中のWALデータのすべてを受信したかどうかを確認する方法はありません。トランザクションの一部は、プライマリではコミットされたものと表示されていたとしても、コミットされていないと表示されるかもしれません。提供できる保証は、WALデータがスタンバイで安全に受信したことが分かるまで、アプリケーションは明示的なトランザクションコミットの成功に関する受領通知を受けとらないことです。

最終スタンバイサーバを本当に失った場合、`synchronous_standby_names`を無効にし、プライマリサーバの設定ファイルを再読み込みしなければなりません。

プライマリが残りのスタンバイサーバと孤立された場合、残りのスタンバイサーバの中から最善の候補にフェールオーバーさせなければなりません。

トランザクションの待機中にスタンバイサーバを再作成する必要がある場合、`pg_start_backup()`および`pg_stop_backup()`を実行するコマンドを`synchronous_commit = off`であるセッション内で確実に実行してください。さもないとこれらの要求はスタンバイに現れるまで永遠に待機します。

25.3. フェールオーバー

プライマリサーバに障害が起こると、スタンバイサーバはフェールオーバー処理を始めなければなりません。

スタンバイサーバが故障した場合、フェールオーバーは不要です。 多少の時間の後に、スタンバイサーバを再起動できれば、再起動可能なリカバリのため、リカバリ処理も即座に再起動することができます。 スタンバイサーバを再起動できなければ、新しい完全なスタンバイサーバのインスタンスを作成しなければなりません。

プライマリサーバに障害が起こりスタンバイサーバが新しいプライマリとなり、その後古いプライマリが再起動した場合、もはやプライマリサーバでなくなっていることを古いプライマリに知らせる機構が必要です。 これはSTONITH (Shoot the Other Node In The Head)と一部ではいわれています。 これは、混乱と最悪はデータ損失をもたらしかねない、両方のシステムが自身をプライマリとして認識してしまう状況を防ぐために必要です。

多くのフェールオーバーシステムではプライマリとスタンバイといった2つのシステムを使用します。 なんらかのハートビート機構でプライマリとスタンバイを接続し、両者の接続性とプライマリの実行能力を継続的に確認します。 また、第3のシステム（証言サーバと呼ばれます）を使用して、不適切なフェールオーバーなどの状況を防ぐこともできます。 しかし、さらに複雑になりますので、十分な注意と厳密な検証の元に設定を行わない限り行う意味がありません。

PostgreSQLは、プライマリサーバの障害を識別し、スタンバイデータベースサーバに通知するために必要なシステムソフトウェアを提供しません。 こうしたツールは多く存在し、IPアドレスの移行といったフェールオーバーを成功させるために必要な機能をオペレーティングシステムにうまく統合させています。

スタンバイサーバへのフェールオーバーが起きた後、運用可能なサーバは1つしかありません。 これは縮退状態と呼ばれます。 以前のスタンバイサーバはプライマリサーバになり、以前のプライマリは停止し、その後も停止し続けるかもしれません。 通常の運用に戻すには、スタンバイサーバを再作成しなければなりません。 以前のプライマリサーバが起動できれば、これを使用して構いませんし、第三のおそらく新規のシステムを使用しても構いません。 完了すれば、プライマリとスタンバイの役割が切り替わったとみなすことができます。 新しいスタンバイサーバを再作成するまでに第三のサーバを使用して新しいプライマリのバックアップを提供することを選択する人もいますが、これがシステム構成と運用手順を複雑にすることは明らかです。

プライマリサーバからスタンバイサーバへの切り替えは高速ですが、フェールオーバークラスタを再度準備するのに多少時間が必要です。 それぞれのシステムを保守のために定期的に停止することができるので、プライマリからスタンバイへの定期的切り替えは有益です。 これは同時に、必要になった時、フェールオーバー機構が実際に機能するかどうかを確認する試験としても役立ちます。 管理手順の文書化を勧めます。

ログシッピングを行うスタンバイサーバのフェールオーバーを発生させるためには、`pg_ctl promote`を実行する、あるいは、`recovery.conf`内の`trigger_file`設定によって指定されるファイル名とパスを持つトリガファイルを作成してください。 フェールオーバーのために`pg_ctl promote`を使用するつもりならば、`trigger_file`は必要ありません。 プライマリから読み取り専用の問い合わせによる負荷を軽減させるためだけに使用し、高可用性を目的としていない、報告処理用サーバを構築する場合は、昇格させる必要はありません。

25.4. この他のログ SHIPPING の方法

これまでの節で説明した組込みのスタンバイモードの他の方法として、アーカイブ場所を順次問い合わせる `restore_command` を使用する方法があります。これはバージョン 8.4 以前では唯一の利用可能な選択肢でした。この設定では、スタンバイ操作で必要とするポーリングを自身で実施しますので、`standby_mode` を無効にします。このリファレンス実装として [pg_standby](#) を参照してください。

このモードでは、サーバは1度に1つのWALファイルを適用することに注意してください。このため問い合わせ用にスタンバイサーバを使用する場合(ホットスタンバイを参照)、マスタにおける動作とそれがスタンバイで可視になるまでの間に、WALファイルをみだすために必要とする時間に相当する、遅延が存在します。`archive_timeout` を使用して遅延を短くすることができます。また、この方法とストリーミングレプリケーションと組み合わせることができないことにも注意してください。

プライマリおよびスタンバイサーバの両方で発生する操作は通常の継続的なアーカイブ処理とリカバリ処理です。2つのデータベースサーバが連携する唯一の点は、両者が共有するWALファイルのアーカイブです。プライマリがアーカイブに書き出し、スタンバイがアーカイブから読み取ります。注意して他のプライマリサーバ由来のWALアーカイブが混在しないことを確実にしなければなりません。さもないと混乱が発生します。スタンバイ操作でのみ必要なものですので、アーカイブは必ずしも巨大になりません。

2つの疎結合サーバを協調させる秘訣は簡単で、スタンバイサーバにて使用される `restore_command` です。これは次のWALファイルを問い合わせ、それをプライマリから利用可能になるまで待機します。この `restore_command` はスタンバイサーバの `recovery.conf` ファイルで指定されます。通常のリカバリ処理はWALアーカイブからファイルを要求し、ファイルが利用できなければ失敗を報告します。スタンバイ処理では、次のWALファイルを入手できないことは異常ではありませんので、スタンバイは利用可能になるまで待機しなければなりません。`.backup` または `.history` で終わるファイルについては、待機する必要はなく、非ゼロの終了コードを返さなければなりません。`restore_command` を待機させるには、次のWALファイルの存在を確認した後ループする独自のスクリプトを作成することで実現できます。また、`restore_command` に割り込み、ループを終了させ、ファイルが存在しないというエラーをスタンバイサーバに返す、フェールオーバを発生させる何らかの方法がなければなりません。これがリカバリ処理を停止しますので、スタンバイサーバは通常のサーバになります。

`restore_command` の擬似コードの一例は以下です。

```
triggered = false;
while (!NextWALFileReady() && !triggered)
{
    sleep(100000L);          /* wait for ~0.1 sec */
    if (CheckForExternalTrigger())
        triggered = true;
}
if (!triggered)
    CopyWALFileForRecovery();
```

待機を行う `restore_command` の実例は [pg_standby](#) モジュール内で提供されています。これは上記のロジックをどのように正確に実装するかについての参照として使用すべきです。また、これを特定の設定または環境をサポートするため必要に応じて拡張することができます。

フェールオーバを通知する手段は計画・設計段階で重要な部分です。考えられる選択肢の1つは `restore_command` です。これは各WALファイルに対して1度実行されるものですが、`restore_command` を実行するプロセスは各ファイルに対して起動・終了します。このようにデーモンやサーバプロセスはありませんので、シグナルやシグナルハンドラを使用することはできません。したがって、`restore_command` はフェールオーバの通知には適していません。特にプライマリサーバ上の既知の `archive_timeout` 設定と関係して使用できるならば、単純なタイムアウト機能を使用することができます。しかし、これはネットワーク障害や高負荷なプライマリサーバによりフェールオーバが始まってしまうため、どちらかというエラーになりやすいものです。実現可能ならば、明示的な通知用ファイルの作成などの通知機構の方が理想的です。

25.4.1. 実装

この代替方式を使用してスタンバイサーバを構築する短めの手順を以下に示します。各段階の詳細については、注記していますので、前の節を参照してください。

- 1 11 できる限り同じようにプライマリシステムとスタンバイシステムを設定してください。同じリリースレベルのPostgreSQLの同一コピーの導入も含まれます。
- 1 11 プライマリサーバで、継続的アーカイブをスタンバイサーバ上のディレクトリ上にWALをアーカイブするように設定してください。プライマリサーバで、[archive_mode](#)、[archive_command](#) および [archive_timeout](#) が適切に設定されていることを確認してください(項24.3.1を参照してください)。
- 1 11 プライマリサーバでベースバックアップを作成(項24.3.2を参照してください)し、スタンバイサーバでこのデータを

ロードしてください。

- 111 スタンバイサーバで、上記の通り待機を行う `restore_command` を指定した `recovery.conf` を使用して、ローカルな WAL アーカイブからリカバリ処理を実行してください ([項24.3.4](#)を参照してください)。

リカバリ処理は WAL アーカイブを読み取り専用として扱います。このため、WAL ファイルがスタンバイシステムにコピーされた後、スタンバイデータベースサーバによる読み取りと同時に WAL ファイルをテープにコピーすることができます。このように、高可用性スタンバイサーバの実行を、災害からのリカバリを目的とした長期的な保管と同時に行うことができます。

試験のためにプライマリサーバとスタンバイサーバを同じシステムで稼働させることができます。これによりサーバ堅牢性が向上することも、高可用性と呼べることもありません。

25.4.2. レコードベースのログ SHIPPING

この代替手法を用いたレコード単位のログ SHIPPING の実装も可能ですが、利用者側の開発が必要です。さらに、完全な WAL ファイルが転送された後のみで、変更がホットスタンバイ問い合わせで可視になります。

外部プログラムから、WAL の現在の終了点のファイル名と正確なバイトオフセットを見つけ出す `pg_xlogfile_name_offset()` 関数 ([項9.26](#)を参照) を呼び出すことができます。そして、WAL ファイルに直接アクセスし、直前の既知の WAL 終了点から現在の終了点までのデータをスタンバイサーバにコピーすることができます。この方法では、データ損失期間はコピー処理プログラムの実行周期となります。非常に短くすることができますし、部分的に使用されたセグメントファイルを強制的にアーカイブするため無駄な帯域もありません。スタンバイサーバの `restore_command` スクリプトが WAL ファイル全体しか扱うことができないことに注意してください。このため、逐次的にコピーしたデータは通常はスタンバイサーバで利用することができません。プライマリサーバが停止した時のみこれを使用します。その場合、プライマリサーバが立ち上がる前に、最後の部分的な WAL ファイルがセカンダリサーバに渡されます。この処理の正しい実装では、データコピープログラムと `restore_command` スクリプトとの関係が必要です。

PostgreSQL バージョン 9.0 から、同じ利点をより少ない設定で実現できるストリーミングレプリケーション ([項25.2.5](#)参照) を使用することができます。

[前のページ](#)
フェールオーバー

[ホーム](#)
[上に戻る](#)

[次のページ](#)
ホットスタンバイ

25.5. ホットスタンバイ

ホットスタンバイという単語は、サーバがアーカイブリカバリを実行している最中にサーバに接続し読み取り専用の問い合わせを実行することができる機能を説明するために使われます。これは、レプリケーションという目的およびバックアップからのリストアの両方で高い精度で好ましい状態にするために有用です。ホットスタンバイという単語はまた、ユーザが問い合わせを実行しながら、または、開いている接続を維持しながら、またはその両方で、サーバをリカバリ状態から通常の動作に移すことができる機能も示すものです。

ホットスタンバイモードにおける問い合わせは、通常の問い合わせに類似していますが、利用上および管理上の差異が多少あり、以下に説明します。

25.5.1. ユーザのための概説

スタンバイサーバで`hot_standby`パラメータが真に設定されている場合、リカバリによりシステムが一貫性を持つようになった後接続を受け付け始めます。こうした接続はすべて読み取り専用に限定されます。一時テーブルであっても書き込むことはできません。

スタンバイ上のデータはプライマリサーバから届くまでに多少の時間がかかります。このため、プライマリとスタンバイの間にはある程度の遅延があります。したがって、同じ問い合わせをほとんど同時にプライマリとスタンバイに対して実行すると、異なる結果が返る可能性があります。スタンバイ上のデータはプライマリに対して最後には一貫性を持つといえます。あるトランザクションのコミットレコードがスタンバイ上で再生されると、そのトランザクションでなされた変更はスタンバイで獲得されるすべての新規スナップショットで可視になります。現在のトランザクション隔離レベルに応じて、スナップショットは各問い合わせの開始時または各トランザクションの開始時に獲得されます。詳細については[項13.2](#)を参照してください。

ホットスタンバイ中に開始されたトランザクションは以下のコマンドを発行することができます。。

- 問い合わせによるアクセス - `SELECT`および`COPY TO`
- カーソルコマンド - `DECLARE`と`FETCH`と`CLOSE`
- パラメータの操作 - `SHOW`と`SET`と`RESET`
- トランザクション管理コマンド
 - `BEGIN`と`END`と`ABORT`と`START TRANSACTION`
 - `SAVEPOINT`と`RELEASE`と`ROLLBACK TO SAVEPOINT`
 - `EXCEPTION`ブロックおよびこの他の内部サブトランザクション
- `LOCK TABLE`。なお、以下のモードが明示された場合に限りです。 `ACCESS SHARE`または`ROW SHARE`または`ROW EXCLUSIVE`
- 計画と資源 - `PREPARE`と`EXECUTE`と`DEALLOCATE`と`DISCARD`
- プラグインと拡張 - `LOAD`

ホットスタンバイ中に開始したトランザクションではトランザクションIDを割り当てられません。また、システムのログ先行書き込みに書き出すことができません。このため、以下の動作はエラーメッセージを生成します。

- データ操作言語（DML）。`INSERT`、`UPDATE`、`DELETE`、`COPY FROM`および`TRUNCATE`。リカバリ中にトリガ内で実行されてしまう場合でも許されていない動作であることに注意してください。現在のホットスタンバイ環境では行うことができないトランザクションIDの割り当てを行うことなく、テーブル行の読み書きを行うことができませんので、この制限は一時テーブルであっても適用されます。
- データ定義言語（DDL）。`CREATE`、`DROP`、`ALTER`および`COMMENT`。この制約は一時テーブルに対しても適用されます。これらの操作の実行がシステムカタログテーブルの更新を必要とするためです。
- `SELECT ... FOR SHARE | UPDATE`。背後のデータファイルを更新することなく行ロックを獲得することはできないためです。
- データ操作言語のコマンドを生成する`SELECT`文のルール
- `ROW EXCLUSIVE MODE`より高いモードを明示的に要求する`LOCK`

- 短いデフォルト構文のLOCK。これはACCESS EXCLUSIVE MODEを要求するためです。
- 読み取り専用でない状態を明示的に設定するトランザクション処理コマンド
 - BEGIN READ WRITEとSTART TRANSACTION READ WRITE
 - SET TRANSACTION READ WRITEとSET SESSION CHARACTERISTICS AS TRANSACTION READ WRITE
 - SET transaction_read_only = off
- 二相コミットコマンド - PREPARE TRANSACTION、COMMIT PREPAREDおよびROLLBACK PREPARED。読み取り専用トランザクションでも、プリペア相（二相コミットの第1相）においてWALの書き込みが必要だからです。
- シーケンス更新の関数 - nextval() とsetval()
- LISTEN、UNLISTENおよびNOTIFY

通常の操作では、"読み取り専用"トランザクションにはシーケンスの更新およびLISTENとUNLISTENとNOTIFYの使用が許可されています。ホットスタンバイセッションの操作では、通常の読み取り専用セッションよりも少し厳しい制約を受けます。将来のリリースではこの制約の一部が緩和されるかもしれません。

ホットスタンバイ中は、transaction_read_onlyパラメータは常に真であり、変更することはできません。しかし、データベースを変更するような試行がない限り、ホットスタンバイ中の接続は他のデータベース接続とほとんど同じように動作します。もし、フェールオーバーまたはスイッチオーバーが発生すると、データベースは通常処理モードに切り替わります。サーバのモードが変わってもセッションは接続を保持します。ホットスタンバイが完了すると、読み書き可能なトランザクションを（ホットスタンバイ中に始まったセッションからであっても）始められるようになります。

ユーザはSHOW transaction_read_onlyを発行することで、そのセッションが読み取り専用かどうかを調べることができます。さらに、ユーザがスタンバイサーバに関する情報にアクセスできる関数群(表9-60)があります。これらによりデータベースの現状認識を行うプログラムを作成することができます。これらを使用して、リカバリの進行状況を監視するために使用したり、データベースを特定の状態にリストアする複雑なプログラムを作成したりすることができます。

25.5.2. 問い合わせコンフリクトの処理

プライマリサーバとスタンバイサーバは、多方面でゆるく結合しています。プライマリサーバの動作はスタンバイサーバに影響します。その結果、負の相互作用またはコンフリクトの可能性があります。最も分かりやすいコンフリクトは性能です。プライマリサーバで巨大なデータがロードされた場合、スタンバイサーバにおいて同様に巨大なWALレコードが生成されるので、スタンバイサーバにおける問い合わせは互いにI/Oなどのシステム資源を奪い合います。

ホットスタンバイで発生する可能性があるコンフリクトの種類には他にもあります。これらのコンフリクトは、問い合わせをキャンセルしなければならない可能性があり、解消させるためにはセッションの接続を閉ざすことになる場合もあるため、致命的なコンフリクトです。ユーザにはこうしたコンフリクトを扱うための複数の方法が提供されます。コンフリクトする状況には以下があります。

- プライマリサーバで獲得されたアクセス排他ロックは、スタンバイの問い合わせにおけるテーブルアクセスとコンフリクトします。明示的なLOCKコマンドおよび各種DDL操作を含みます。
- プライマリでテーブル空間を削除することは、一時作業ファイル用にそのテーブル空間を使用するスタンバイ側の問い合わせとコンフリクトします。
- プライマリでデータベースを削除することは、スタンバイ側でそのデータベースに接続するセッションとコンフリクトします。
- WALからレコードを消去するバキュームアプリケーションは、削除される任意の行を"見る"ことができるスナップショットを持つスタンバイでのトランザクションとコンフリクトします。
- WALからレコードを消去するバキュームアプリケーションは、消去されるデータが可視か否かに関係なく、スタンバイで対象ページにアクセスする問い合わせとコンフリクトします。

プライマリサーバでは、こうした状況は単に待たされるだけです。ユーザはコンフリクトする操作をキャンセルすることを選ぶことができます。しかし、スタンバイ側には選択肢がありません。WALに記録された操作はすでにプライマリで発生したものですので、スタンバイではその適用に失敗してはなりません。さらに、申請されたWALを無制限に待機させることを許すことは、まったく望まない結果になってしまうかもしれません。なぜなら、スタンバイの状態がプライマリの状態とだんだんかけ離れてしまうからです。したがって適用すべきWALレコードとコンフリクトするスタンバイの問い合わせを強制的に取り消す仕組みが用意されています。

この問題の例として、スタンバイサーバで現在問い合わせ対象となっているテーブルをプライマリサーバでDROP TABLEを行う管

理者を考えてみます。スタンバイでDROP TABLEが適用されたら問い合わせを継続できないことは明確です。プライマリ上でこうした状況が発生した場合は、他の問い合わせが終わるまでDROP TABLEは待機されます。しかし、DROP TABLEがプライマリで実行された時、プライマリ側でスタンバイで稼動する問い合わせに関する情報がありませんので、スタンバイ側のこうした問い合わせを待機させることはできません。スタンバイ側で問い合わせが実行している時にWALの変更レコードがスタンバイに届けば、コンフリクトが発生します。スタンバイサーバはWALレコードの申請を遅延させる（およびその後の申請すべても遅延させる）か、DROP TABLEを適用できるようにコンフリクトする問い合わせを取り消すかのいずれかを行わなければなりません。

コンフリクトする問い合わせが短ければ、申請されたWALを多少遅延させることで、問い合わせを完了させることが通常望めます。しかし、申請されたWALが長く遅延することはたいしては望まれません。したがって、取り消し機能は[max_standby_archive_delay](#)と[max_standby_streaming_delay](#)というパラメータを持ちます。これらは申請されたWALに許される遅延を定義するものです。コンフリクトする問い合わせは、何らかの新しく受信したWALデータを適用するための各種遅延設定を超えたら取り消されます。アーカイブからWALデータを読み取る場合（つまりベースバックアップからの初期リカバリや大きく遅延したスタンバイサーバの"追従"）とストリーミングレプリケーションとで異なる遅延値を指定することができるように2つのパラメータが存在します。

主に高可用性のために存在するスタンバイサーバでは、スタンバイ側の問い合わせによって発生する遅延のためにプライマリと大きく遅延が発生することがないように、遅延パラメータを相対的に短く設定することが最善です。しかし、スタンバイサーバが長時間かかる問い合わせを実行するためのものであれば、長い遅延もしくは制限を設けないことが好まれるかもしれません。しかし、長時間かかる問い合わせがWALレコードの適用を遅延させてしまう場合、スタンバイサーバ上の他のセッションがプライマリにおける最近の変更を参照することができなくなることは覚えておいてください。

[max_standby_archive_delay](#)または[max_standby_streaming_delay](#)で指定した遅延を超えると、コンフリクトする問い合わせは取り消されます。通常これは単なる取り消しエラーという結果となりますが、DROP DATABASEを再生する場合では、コンフリクトするセッション全体が終了します。また、コンフリクトが待機中のトランザクションで保持されるロックについてのものであれば、そのコンフリクトするセッションが終了します（この動作は将来変更されるかもしれません）。

取り消された問い合わせは直ちに再試行されます（もちろん新規のトランザクション開始後に）。問い合わせの取り消しは、再生されるWALレコードの性質に依存するので、取り消された問い合わせが再度実行された場合には正常に動作するかもしれません。

遅延パラメータはスタンバイサーバでWALデータを受信してからの経過時間と比べられることに注意してください。したがって、スタンバイ上で任意の問い合わせに許される猶予期間は、この遅延パラメータよりも大きくなることは決してありません。これまでの問い合わせを完了させるために待機した結果、あるいは、大量の更新負荷に追従することができなくなった結果、スタンバイがすでに遅延している場合は相当小さくなります。

スタンバイ側の問い合わせとWAL再生の間でもっともよくあるコンフリクト理由は"早すぎる消去"です。通常PostgreSQLはMVCC規則にしたがって正確なデータの可視性を確実にするために、古い行バージョンを参照するトランザクションが存在しない場合それらを消去することが許されています。しかし、この規則はマスタ上で実行するトランザクションのみに適用させることができます。したがって、スタンバイ上のトランザクションでまだ可視である行バージョンを、マスタ上の消去処理が削除してしまう可能性があります。

熟練したユーザは、行バージョンの消去と行バージョンの凍結の両方ともスタンバイ側の問い合わせとコンフリクトする可能性があることに気づくはずですが、手作業でのVACUUM FREEZEは、更新または削除された行がないテーブルであったとしてもコンフリクトを発生し易いものです。

プライマリサーバにおいて規則的かつ頻繁に更新されるテーブルは、スタンバイサーバにおける問い合わせの取り消しの原因になりやすいことを利用者は理解するべきです。そのような場合、[max_standby_archive_delay](#)または[max_standby_streaming_delay](#)の設定値は[statement_timeout](#)の設定と同様に考えることができます。

取り消しの数が受け入れがたいほど多い場合、他に救済的な処理法があります。1つ目の選択肢は、[hot_standby_feedback](#)パラメータを設定することです。これはVACUUMによる最近不要になった行の削除を防止しますので、消去によるコンフリクトが発生しません。これを行う場合、プライマリで不要になった行の消去が遅延することに注意しなければなりません。望まないテーブルの膨張が発生してしまうかもしれません。しかし、スタンバイ側で行うべき問い合わせをプライマリサーバ上で直接実行することと比べ、こうした消去に関する問題を優先する価値はありません。また、スタンバイに実行負荷を分散できるという利点があります。この場合、遅延したWALファイルにすでに、スタンバイ側で実行させたい問い合わせとコンフリクトする項目が含まれている可能性がありますので、[max_standby_archive_delay](#)は長めにしなければなりません。

他の選択肢は、不要になった行が通常よりも早く消去されないようにプライマリサーバで[vacuum_defer_cleanup_age](#)を増やすことです。これにより、[max_standby_streaming_delay](#)を長くすることなく、スタンバイでキャンセルが起こるようになる前により多くの時間、問い合わせを実行することができます。しかし、[vacuum_defer_cleanup_age](#)はプライマリサーバ上で実行されたトランザクションを単位に測定されますので、この方法では特定の実行期間を保証することは困難です。

問い合わせキャンセルの個数とその原因はスタンバイサーバ上の[pg_stat_database_conflicts](#)システムビューを用いて参照することができます。また[pg_stat_database](#)システムビューには要約された情報が含まれます。

25.5.3. 管理者のための概説

postgresql.confにおいてhot_standbyがonに設定されかつrecovery.confが存在すれば、サーバはホットスタンバイモードで稼働します。しかし、サーバはまず問い合わせが実行できる程度の一貫性を持つ状態を提供するために十分なリカバリを完了させなければなりませんので、ホットスタンバイでの接続が有効になるまでに多少の時間がかかるかもしれません。サーバの準備ができたことを確認するために、アプリケーションで接続試行を繰り返すか、サーバログに以下のメッセージがあるかどうかを確認します。

```
LOG:  entering standby mode
```

```
... 多少時間が経過して ...
```

```
LOG:  consistent recovery state reached
```

```
LOG:  database system is ready to accept read only connections
```

一貫性に関する情報はプライマリでチェックポイント毎に一度記録されます。プライマリでwal_levelがhot_standbyに設定されていない間に書き込まれたWALを読み取っている時、ホットスタンバイを有効にすることはできません。また、一貫性のある状態への到達は、以下の両方が存在する間遅延することがあります。

- サブトランザクション数が64を超える書き込みトランザクション
- 非常に長く実行される書き込みトランザクション

ファイルベースのログ SHIPPING(「ウォームスタンバイ」)を実行しているのであれば、次のWALファイルが届く、長くともプライマリのarchive_timeout設定まで待機しなければなりません。

プライマリサーバにおける設定値が変更した場合、スタンバイサーバにおいて数個のパラメータの再設定が必要です。スタンバイサーバにおける設定値は、プライマリサーバにおける設定値以上でなければなりません。所定値未満の設定の場合、スタンバイは起動を取りやめます。所定値以上の設定により、スタンバイサーバは再起動してリカバリが再び開始されます。このパラメータは以下です。

- max_connections
- max_prepared_transactions
- max_locks_per_transaction

[max_standby_archive_delay](#)および[max_standby_streaming_delay](#)の値が適切であるように管理者が選択することが重要です。最善の選択は業務上の優先順位によって変化します。例えば、サーバが主に高可用性を目的としたサーバとして作業するものであれば、短い遅延を設定したいでしょう。非常に積極的な設定ですが、ゼロにしたいかもしれません。スタンバイサーバが意思決定支援のための問い合わせ用の追加サーバとして作業するものであれば、数時間程度の最大の遅延値の設定、あるいは問い合わせの完了を永遠に待つことを意味する-1という設定でさえ、許容範囲であるかもしれません。

プライマリ側で「ヒントビット」として書き出されたトランザクション状態はWALに記録されません。このためスタンバイ側のデータはスタンバイ側でヒントを再度書き出すことになります。ユーザは大規模なソート用の一時ファイルを書き出し、relcache情報ファイルを再作成します。したがって、ホットスタンバイモードではデータベースのすべてが本当に読み取り専用ではありません。また、ローカルでは読み取り専用のトランザクションであってもdblinkモジュールを使用したリモートデータベースへの書き出しや、その他のPL関数を使用したデータベース外部への操作が可能であることに注意してください。

リカバリモードの間、下記の管理者用コマンドは受理されません。

- データ定義言語、例えばCREATE INDEX
- 権限および所有権 - GRANTとREVOKEとREASSIGN
- 保守コマンド - ANALYZEとVACUUMとCLUSTERとREINDEX

繰り返しますが、これらのコマンドの一部は、プライマリサーバにおける「読み取り専用」モードのトランザクションで実際に許可されていることに注意してください。

その結果、スタンバイ側にのみ存在する追加のインデックスやスタンバイ側にのみ存在する統計情報を作成することはできません。これらの管理者用コマンドが必要な場合、プライマリ側で実行しなければなりません。最終的にこの変更はスタンバイ側に伝播します。

pg_cancel_backend()とpg_terminate_backend()は利用者のバックエンドでは実行できますが、リカバリを実行する起動プロセスでは実行できません。pg_stat_activityは起動プロセスのエントリを表示しないし、リカバリトランザクションが実行中かどうかとも表示しません。その結果、リカバリの間pg_prepared_xactsは常に空となります。準備した確信のもてないトランザクションの状態を解明したい場合、プライマリサーバにおいてpg_prepared_xactsを表示するビューを作成し、ビューに解明のためのコマンドを発行してください。

pg_locksは通常通りバックエンドで保持されるロックを示します。pg_locksはまた、リカバリによって再生されているトランザ

クションで保持されるAccessExclusiveLocksのすべてを所有する、起動プロセスで管理される仮想トランザクションも表示します。 起動プロセスはデータベースの変更を行うためのロックを獲得しません。 このため起動プロセスにおいてAccessExclusiveLocks以外のロックはpg_locksでは表示されません。 これらは存在することを想定されているだけです。

存在を検知する情報が単純なので、Nagiosプラグインは稼動します。 一部の報告値が異なった、混乱を招く結果となりますが、check_postgresの監視スクリプトも動作します。 それでも、プライマリで行われるバキュームはその変更をスタンバイに送信します。

リカバリの間WALの制御コマンドは稼動しません。 例えば、pg_start_backupやpg_switch_xlogなどです。

pg_stat_statementsも含み、動的に読み込み可能なモジュールは稼動します。

デッドロック検出を含むアドバイザリロックは、通常リカバリにおいて稼動します。 アドバイザリロックはWALに決して記録されないので、プライマリサーバでもスタンバイサーバでもWALの再実行においてコンフリクトが起こらないことに注意してください。 プライマリサーバでアドバイザリロックを取得して、スタンバイサーバで同様のアドバイザリロックを掛けることはできません。 アドバイザリロックは取得したサーバだけに関係するものです。

SlonyやLondisteやBucardoのようにトリガに基づいたレプリケーションシステムは、スタンバイサーバで全く稼動しません。 しかし、それによる変更がスタンバイサーバに送られるまでは、プライマリサーバにおいて問題なく稼動します。 WALの再実行はトリガに基づいたものではありません。 したがって、データベースへの付加的な書き込みを必要とするか、トリガの使用に依存するものを、スタンバイサーバを中継して他のシステムへ送ることはできません。

一部のUUIDジェネレータは、データベースに新しい状態を書き出すことに依存していない限り動作可能ですが、新しいOIDを割り当てることはできません。

現時点では、読み取り専用のトランザクションでは一時テーブルの作成は許されません。 このため既存のスクリプトが正しく動作しない場合があります。 この制限は将来のリリースで緩和されるかもしれませんが、これは、標準SQLとの互換性の問題でもあり、技術的な問題でもあります。

テーブル空間が空の場合だけ、DROP TABLESPACEが成功します。 一部のスタンバイ側のユーザはtemp_tablespacesパラメータを介してテーブル空間を活発に使用しているかもしれません。 テーブル空間に一時ファイルが存在する場合、一時ファイルを確実に削除するためすべての問い合わせが取り消されます。 このため、WAL再生を続けながらテーブル空間を削除することができます。

プライマリサーバにおけるDROP DATABASEまたはALTER DATABASE ... SET TABLESPACEの実行により、スタンバイサーバのデータベースに接続するすべてのユーザを強制的に接続を切断させることになるWALエントリを生成します。 これはmax_standby_streaming_delayの設定にかかわらず、直ちに起こります。 ALTER DATABASE ... RENAMEはユーザを切断しないので大部分の場合は気がつきませんが、プログラムがデータベースの名称に依存するときは混乱の原因となることに注意してください。

通常の(リカバリ以外の)モードで、ログイン権限を持つロールが接続している間にそのロールにDROP USERまたはDROP ROLEを発行した場合、接続中のユーザには何も起こらず、接続し続けます。 しかし、そのユーザは再接続できません。 この振舞いはリカバリモードでも適用されます。 このためプライマリ側でDROP USERされたとしても、スタンバイ側のユーザの接続は切断されません。

リカバリの間も統計情報は収集されます。 すべてのスキャン、読み取り、ブロック、インデックスの使用などは、スタンバイサーバにおいて正常に記録されます。 再実行によりプライマリサーバの結果が重複して収集されることはないので、行の挿入によりpg_stat_user_tablesの挿入列の値は増加しません。 リカバリの開始時点で統計情報ファイルが削除されるので、プライマリサーバとスタンバイサーバで統計情報は異なります。 これは将来どうするか検討中であり、バグではありません。

リカバリの間は自動バキュームは稼動しません。 リカバリが終わると正常に起動します。

リカバリの間バックグラウンドライタは稼動して(プライマリサーバにおけるチェックポイントに類似した)リスタートポイントを設定し、通常のブロック消去を行います。 これはスタンバイサーバに保存されるヒントビット情報の更新を含むことができます。 リカバリの間CHECKPOINT コマンドは受理されますが、新規のチェックポイントではなくてリスタートポイントが設定されます。

25.5.4. ホットスタンバイパラメータリファレンス

種々のパラメータが上記[項25.5.2](#)および[項25.5.3](#)で述べられています。

プライマリサーバでは、[wal_level](#)および[vacuum_defer_cleanup_age](#)のパラメータを使用できます。 プライマリサーバに[max_standby_archive_delay](#)および[max_standby_streaming_delay](#)を設定しても無効です。

スタンバイサーバでは[hot_standby](#)と[max_standby_archive_delay](#)と[max_standby_streaming_delay](#)のパラメータを使用できます。 サーバがスタンバイモードの間[vacuum_defer_cleanup_age](#)を設定しても無効です。 しかし、スタンバイサーバがプライマリサーバになった場合、意味を持つようになります。

25.5.5. 警告

ホットスタンバイには幾つかの制限があります。 将来のリリースでは改善されると思われます。

- 現在ハッシュインデックスに対する操作はWALに記録されません。 このため再生してもこれらのインデックスは更新されません。
- スナップショットを取ることができるようになる前に、実行中のトランザクションについての完全な知識が要求されます。(現時点では64を超える)多くのサブトランザクションを使用するトランザクションでは、実行中の最長の書き込みトランザクションが完了するまで、読み取り専用の接続の開始は遅延されます。 この状況が起こると、それを説明するメッセージがサーバログに記録されます。
- スタンバイ問い合わせ用の有効な起動ポイントは、マスタにおけるチェックポイント毎に生成されます。 マスタが停止状態にある時にスタンバイが停止した場合、プライマリが起動し、さらに起動ポイントをWALログに生成するまで再度ホットスタンバイになることができないことがあります。 この状況は、通常考えられる状態では問題ではありません。 一般的に、プライマリが停止し利用できなくなった場合、それはスタンバイに対して新しいプライマリに切り替わることを要求するような深刻な失敗が原因であることが多いはずです。 また、プライマリを意図的に停止させるような状況では、それに伴いスタンバイが新しいプライマリになめらかに切り替わることも普通の手順です。
- リカバリの終了において、準備されたトランザクションが保持するAccessExclusiveLocksには、通常の2倍のロックテーブルへのエントリ数が必要です。 通常AccessExclusiveLocksを取るプリペアドトランザクションを大量に同時実行させる、または、多くのAccessExclusiveLocksを取る大規模なトランザクションを1つ実行させることを考えている場合、max_locks_per_transactionの値を、おそらくプライマリサーバのパラメータ値の倍程度に大きくすることを勧めます。 max_prepared_transactionsの設定が0ならば、これを検討する必要はまったくありません。
- シリアライザブルトランザクション隔離レベルはまだホットスタンバイでは利用できません。(項13.2.3および項13.4.1参照) ホットスタンバイにおいてトランザクションをシリアライザブルトランザクション隔離レベルに設定しようとすると、エラーになります。

[前のページ](#)

その他のログシッピングの方法

[ホーム](#)
[上に戻る](#)

[次のページ](#)
リカバリの設定

第 26 章 リカバリの設定

目次

26.1. [アーカイブリカバリの設定](#)

26.2. [リカバリ対象の設定](#)

26.3. [スタンバイサーバの設定](#)

この章では、`recovery.conf` ファイルにおいて利用できる設定について説明します。これらの設定は、リカバリをしている間だけ有効となります。これらの設定は、その次に実行したいリカバリの際にはリセットされなければなりません。また、これらの設定は一旦リカバリが始まると変更することはできません。

`recovery.conf` における設定は `name = 'value'` の書式を取ります。各行ごとにひとつのパラメータを記述します。ハッシュマーク (#) は、その行の後続がコメントであることを示します。シングルクォート文字を記述するためには、連続する二つのクォート文字 (``) を記述します。

インストールディレクトリの `share/` ディレクトリに、サンプルファイル `share/recovery.conf.sample` が提供されています。

[前のページ](#)[ホットスタンバイ](#)[ホーム](#)[上に戻る](#)[次のページ](#)[アーカイブリカバリの設定](#)

26.1. アーカイブリカバリの設定

`restore_command (string)`

連続したWALファイルのアーカイブを取得するために実行するシェルコマンドを指定します。このパラメータやアーカイブリカバリには必須ですが、ストリーミングレプリケーションの場合には必須ではありません。指定した文字列の中の `%f` はアーカイブから取得するファイル名に置換され、`%p` はコピー先のディレクトリ名に置換されます。（パス名はカレントワーキングディレクトリに対する相対パスとなります。i.e., クラスタのデータディレクトリ）`%r` は、有効な最後のリスタートポイントを含むWALファイルのファイル名に置換されます。これはリスタートを可能にするためのリストア時に必要とされる、保持されているべきもっとも古いファイルとなります。よってこの情報は、現在のリストアからリスタートするために必要となるように、アーカイブを最小化するために利用することができます。`%r` は通常はウォームスタンバイの設定でのみ使用されます（[項25.2](#)を参照）。`%` 文字を埋め込むためには、`%%` と記述してください。

コマンドは、成功した時のみ終了コードのゼロを返却することが重要です。このコマンドは、アーカイブに存在しないファイル名を求めることがあります。その場合には、非ゼロの値を返却しなければなりません。以下に例を示します。

```
restore_command = 'cp /mnt/server/archivedir/%f "%p"'
restore_command = 'copy "C:¥¥server¥¥archivedir¥¥%f" "%p"' # Windows
```

`archive_cleanup_command (string)`

このオプションのパラメータは、すべてのリスタートポイントで実行されるシェルコマンドを指定します。

`archive_cleanup_command` の目的は、スタンバイサーバにとって必要とされない古いアーカイブWALファイルをクリーンアップする仕組みを提供することです。`%r` は、最後の有効なリスタートポイントを含んでいるWALファイルのファイル名に置換されます。これはリスタートを可能にするためのリストア時に必要とされる、保持されているべきもっとも古いファイルとなり、これによって`%r`より前のすべてのファイルは安全に削除されます。この情報は、現在のリストアからリスタートするために必要となるように、アーカイブを最小化するために利用することができます。

[pg_archivecleanup](#)モジュールは、単独のスタンバイ構成で`archive_cleanup_command`によく使用されます。例：

```
archive_cleanup_command = 'pg_archivecleanup /mnt/server/archivedir %r'
```

但し、複数のスタンバイサーバが同一のアーカイブディレクトリからリストアしている場合には、どのサーバからも必要とされなくなるまで削除しないことを担保しなければなりません。通常、`archive_cleanup_command` はウォームスタンバイの設定でのみ使用されます（[項25.2](#)を参照）。`%` 文字を埋め込むためには、`%%` と記述してください。

コマンドが非ゼロの終了ステータスを返した場合、WARNINGログメッセージが出力されます。

`recovery_end_command (string)`

このパラメータは、リカバリの終了時に一度だけ実行されるシェルコマンドを指定します。このパラメータは必須ではありません。`recovery_end_command` の目的は、レプリケーションやリカバリ実行後のクリーンアップのメカニズムを提供することです。`%r` は、[archive_cleanup_command](#) と同様に、有効な最後のリスタートポイントを含むWALファイルのファイル名に置換されます。

コマンドが非ゼロの終了ステータスを返した場合、WARNINGログメッセージが出力されるとともに、それに関わらずデータベースはスタートアップを進行させます。唯一の例外は、コマンドがシグナルによって終了させられた場合で、その場合、データベースはスタートアップを進行させません。

26.2. リカバリ対象の設定

`recovery_target_name` (string)

このパラメータは、リカバリを実施するために`pg_create_restore_point()`により作成され、リストアポイントとして指定されたものを定義します。`recovery_target_name`、[recovery_target_time](#) または [recovery_target_xid](#) のいずれかひとつを指定することができます。デフォルトではWALログの最後までリカバリします。

`recovery_target_time` (timestamp)

このパラメータは、リカバリを実施するタイムスタンプの上限を指定します。

`recovery_target_time`、[recovery_target_name](#) または、[recovery_target_xid](#)のいずれかを指定できます。デフォルトでは、WALログの最後までリカバリを実施します。厳密な停止ポイントは、[recovery_target_inclusive](#) によっても影響を受けます。

`recovery_target_xid` (string)

このパラメータは、リカバリを進行させるトランザクションIDの上限を指定します。トランザクションIDはトランザクションの開始時に順番に割り振られ、トランザクションはそれとは異なる順番で完了し得るということを理解しておいてください。リカバリされるトランザクションは、指定されたものよりも前 (オプションによっては指定されたものも含まれる) にコミットされたものになります。`recovery_target_xid`、[recovery_target_name](#) または、[recovery_target_time](#)のいずれかを指定できます。デフォルトでは、WALログの最後までリカバリを実施します。厳密な停止ポイントは、[recovery_target_inclusive](#) によっても影響を受けます。

`recovery_target_inclusive` (boolean)

指定したリカバリターゲットの後に停止するか (`true`)、その前に停止するか (`false`) を指定します。[recovery_target_time](#) と [recovery_target_xid](#) のどちらが指定されているかに関係なく、双方に適用されます。この設定は、指定したターゲットコミット時間あるいはトランザクションIDのトランザクションが、それぞれリカバリに含まれるかどうかを指定します。デフォルトは `true` です。

`recovery_target_timeline` (string)

リカバリが作成する個別のタイムラインを指定します。デフォルトでは、ベースバックアップが取得された際のものと同じタイムラインに沿ってリカバリされます。これを`latest`に設定すると、アーカイブ時に見つけた最新のタイムラインに回復します。スタンバイサーバで役立つものです。この他にこのパラメータが必要となるのは、ポイントインタイムリカバリの実施後に到達したような状態に戻す場合など、複数回に渡ってリカバリするような複雑なリカバリのシチュエーションのみです。この議論については [項24.3.5](#) を参照してください。

`pause_at_recovery_target` (boolean)

リカバリ対象に到達した場合、リカバリを休止すべきかどうかを指定します。デフォルトは真です。これは、このリカバリ対象がリカバリのための最も望ましいポイントかどうかチェックするデータベースに対して、問い合わせが実行されることを可能にするよう仮定されます。休止された状態は、`pg_xlog_replay_resume()` ([表9-61](#)参照)の使用により再開することができます。その後、それは回復を終了させます。このリカバリ対象はその後希望の止まるポイントでなければ、サーバをシャットダウンし、リカバリを継続するため、後の対象とリカバリを続けるための再起動のために、リカバリ対象の設定を変更します。

この設定は[hot standby](#)が有効になっていない場合、リカバリ対象が設定されていない場合には効果がありません。

26.3. スタンバイサーバの設定

`standby_mode` (boolean)

PostgreSQL サーバをスタンバイとして起動するかどうかを指定します。このパラメータが `on` の場合、サーバはアーカイブWALファイルの最後に到達してもリカバリを終了せず、`restore_command` の実行、および/あるいは、`primary_conninfo` で指定されたプライマリサーバへ接続することによって、新しいWALセグメントの取得を継続しようとします。

`primary_conninfo` (string)

スタンバイサーバがプライマリサーバに接続するための接続文字列を指定します。この文字列は、[項31.1](#) で説明されている `libpq` の `PQconnectdb` 関数が受け入れる書式で記述されます。この文字列に何のオプションも指定されていない場合、これに対応する環境変数 ([項31.14](#) 参照) が確認されます。もし、環境変数も設定されていない場合、デフォルトの値が使われます。

接続文字列では、プライマリサーバのホスト名 (またはアドレス)、スタンバイサーバのデフォルトと異なるのであればポート番号も指定する必要があります。また、プライマリサーバ上で適切な権限を保有するロールのユーザを指定しなければなりません ([項25.2.5.1](#) 参照)。プライマリサーバが要求するのであれば、パスワードも記述される必要があります。パスワードは `primary_conninfo` に記述することもできますし、スタンバイサーバ上の分離されたファイル `~/.pgpass` に記述することもできます (データベース名には `replication` を使います)。`primary_conninfo` 文字列の中には、データベース名を指定しないでください。

`standby_mode` が `off` になっている場合、この設定は何の影響も持ちません。

`trigger_file` (string)

スタンバイサーバにおいて、リカバリの完了を示すトリガファイルを指定します。もしこの値が設定されていない場合、`pg_ctl promote` を使ってスタンバイサーバを昇格させることができます。`standby_mode` が `off` の時には、この設定は何の影響も持ちません。

第 27章 データベース活動状況の監視

目次

27.1. [標準的なUnixツール](#)

27.2. [統計情報コレクタ](#)

27.2.1. [統計情報収集のための設定](#)

27.2.2. [収集した統計情報の表示](#)

27.2.3. [統計情報関数](#)

27.3. [ロックの表示](#)

27.4. [動的追跡](#)

27.4.1. [動的追跡のためのコンパイル](#)

27.4.2. [組み込み済みのプローブ](#)

27.4.3. [プローブの利用](#)

27.4.4. [新規プローブの定義](#)

データベース管理者はよく、"システムは今現在正しく動作しているか"を気にします。本章では監視方法について説明します。

データベース活動状況の監視と性能解析用のツールは多く存在します。本章の大部分はPostgreSQLの統計情報コレクタの説明に費されていますが、`ps`や`top`、`iostat`、`vmstat`などの通常のUnix監視プログラムを無視すべきではありません。また、性能が悪い問い合わせであると認知された問い合わせは、その後、PostgreSQLの[EXPLAIN](#)コマンドを使用して調査を行う必要が発生します。[項14.1](#)では、個々の問い合わせの振舞いを理解するための、`EXPLAIN`やその他の方法について記載しています。

27.1. 標準的なUnixツール

ほとんどのUNIXプラットフォームでは、PostgreSQLは、個々のサーバプロセスが容易に識別できるように、`ps`によって報告されるコマンドタイトル部分を変更します。以下に表示例を示します。

```
$ ps auxww | grep ^postgres
postgres 15551 0.0 0.1 57536 7132 pts/0      S    18:02   0:00 postgres -i
postgres 15554 0.0 0.0 57536 1184 ?          Ss   18:02   0:00 postgres: writer process
postgres 15555 0.0 0.0 57536 916 ?          Ss   18:02   0:00 postgres: checkpointer process
postgres 15556 0.0 0.0 57536 916 ?          Ss   18:02   0:00 postgres: wal writer process
postgres 15557 0.0 0.0 58504 2244 ?         Ss   18:02   0:00 postgres: autovacuum launcher process
postgres 15558 0.0 0.0 17512 1068 ?         Ss   18:02   0:00 postgres: stats collector process
postgres 15582 0.0 0.0 58772 3080 ?         Ss   18:04   0:00 postgres: joe runbug 127.0.0.1 idle
postgres 15606 0.0 0.0 58772 3052 ?         Ss   18:07   0:00 postgres: tgl regression [local] SELECT
waiting
postgres 15610 0.0 0.0 58772 3056 ?         Ss   18:07   0:00 postgres: tgl regression [local] idle in
transaction
```

(`ps`の適切な呼び出し方はプラットフォームによって異なります。同様に、何が詳細に表示されるのかも異なります。この例は最近のLinuxシステムのものです。) この一覧の最初のプロセスはマスタサーバプロセスです。表示されているコマンド引数は、起動時に使用されたものと同じものです。次の5つのプロセスは、マスタプロセスから自動的に起動されるバックグラウンドワーカープロセスです。("自動バキュームランチャ"を無効にできることと同様に、システムを統計情報コレクタが起動しないように設定していた場合は"統計情報コレクタ"はありません)。残るプロセスはそれぞれ、1つのクライアント接続を取り扱うサーバプロセスです。それぞれのプロセスは、次の形式のコマンドライン表示を設定します。

```
postgres: user database host activity
```

ユーザ、データベース、(クライアント)ホストという項目はクライアントの存続期間中変更されることはありませんが、活動状況を示す部分は変わります。活動状況は、`idle` (つまり、クライアントからのコマンド待ち状態)、`idle in transaction` (`BEGIN`ブロックの内側でのクライアントの待ち状態)、または`SELECT`のようなコマンド種類名のいずれかとなります。また、そのサーバプロセスが他のセッションによって保持されたロックを待っている状態の場合は、`waiting`が追加されます。上の例では、プロセス15606はプロセス15610におけるトランザクションの完了とそれに伴うロックの解放を待っていると推測することができます。(他に実行中のセッションがありませんので、プロセス15610がブロックしている側であるはずで、もっと複雑な名合いでは`pg_locks`システムビューを検索し、どのプロセスがブロックしているか決定しなければなりません)

[update_process_title](#)を無効にした場合、活動情報を示す部分は更新されません。新しいプロセスが起動した時に一度、プロセスの表題は設定されます。プラットフォームの中には、これによりコマンドごとのオーバーヘッドをかなり抑えられるものもありますし、まったく意味がないものもあります。

ティップ: Solarisでは特別な取り扱いが必要です。 `/bin/ps`ではなく、`/usr/ucb/ps`を使用しなければなりません。また、`w`フラグを1つではなく2つ使用しなければなりません。さらに、元の`postgres`の呼び出しに関する`ps`のステータス表示は、各サーバプロセスに関するステータス表示よりも短くなければなりません。この3条件を全て満たさないと、各サーバプロセスの`ps`の出力は、元の`postgres`のコマンドラインのものになってしまいます。

27.2. 統計情報コレクタ

PostgreSQLの統計情報コレクタはサーバの活動状況に関する情報を収集し、報告するサブシステムです。現在、コレクタはテーブルとインデックスへのアクセスをディスクブロックおよび個々の行単位で数えることができます。またこれは、各テーブル内の総行数、および、各テーブルでのVACUUMやANALYZEの実施情報を追跡します。また、ユーザ定義関数の呼ばれた回数、それぞれの消費した総時間をカウントします。

また、PostgreSQLは他のサーバプロセスによって現在実行されている問い合わせを正確に報告する機能を持ちます。これはコレクタプロセスから独立している機能です。

27.2.1. 統計情報収集のための設定

統計情報の収集によって問い合わせの実行に少しオーバーヘッドが加わりますので、システムは情報を収集するようにもしないようにも設定することができます。これは通常は`postgresql.conf`内で設定される、設定パラメータによって制御されます（設定パラメータの設定についての詳細は[第18章](#)を参照してください）。

[track_activities](#)パラメータにより、すべてのサーバプロセスで現在実行されているコマンドを監視することができます。

[track_counts](#)パラメータは、テーブルおよびインデックスアクセスに関する統計情報を収集するかどうか制御します。

[track_functions](#)パラメータは、ユーザ定義関数の使用状況を追跡するかどうかを指定できます。

[track_io_timing](#)パラメータは、ブロック読み取りおよび書き込み回数の監視するかどうかを指定できます。

通常、これらの変数は全てのサーバプロセスに適用できるように`postgresql.conf`内で設定されます。しかし、[SET](#)コマンドを使用して、個別のセッションで有効または無効にすることができます（一般ユーザがその活動を管理者に隠すことを防止するために、スーパーユーザのみがSETを使用してこれらのパラメータを変更することができます。）

統計情報コレクタは収集した情報を他のPostgreSQLプロセスに一時ファイルを介して送信します。これらのファイルは[stats_temp_directory](#)で指名されたディレクトリ、デフォルトは`pg_stat_tmp`内に格納されます。性能を向上させるために、`stats_temp_directory`をRAMベースのファイルシステムを指し示すようにして、物理的なI/O要求を減らすことができます。サーバがシャットダウンした際は、統計情報がサーバの再起動を跨がって保持されるように、統計情報データの永続的なコピーが`global`サブディレクトリに格納されます。

27.2.2. 収集した統計情報の表示

統計情報の収集結果を表示するための、多くの定義済みのビューがあり、[表27-1](#)に一覧表示されています。他にも、[項27.2.3](#)で説明する、基礎的な統計情報関数を使用した独自のビューを構築することもできます。

この統計情報を使用して、現在の活動状況を監視する場合、この情報は即座に更新されないことを認識することが重要です。個別のサーバプロセスは、待機状態になる直前に、新しい統計情報に関する数をコレクタに送信します。ですので、実行中の問い合わせやトランザクションは表示上の総和には影響を与えません。また、コレクタ自体もおよそ`PGSTAT_STAT_INTERVAL`（サーバ構築時に変更しない限り500 ms）ミリ秒に一度新しい報告を出力します。ですので、表示上の情報は実際の活動から遅れて表示されます。しかし、[track_activities](#)で収集される現在の問い合わせの情報は常に最新です。

この他の重要なポイントは、いつサーバプロセスが統計情報を表示するように尋ねられるかです。サーバプロセスは、まずコレクタによって発行された最も最近の報告を取り出します。そして、現在のトランザクションが終わるまで、全ての統計情報ビューと関数においてこのスナップショットを使用し続けます。ですから、現在のトランザクションを続けている間、統計情報は一定の情報を示します。同様に、全セッションの現在の問い合わせに関する情報も、そうした情報がトランザクションで最初に要求された時に収集され、そのトランザクションの間同じ情報が表示されます。これはバグではなく、特徴です。なぜなら、これにより、知らない間に値が変更することを考慮することなく、統計情報に対して複数の問い合わせを実行し、その結果を相関することができるからです。しかし、各問い合わせで新しい結果を取り出したい場合は、確実にトランザクションブロックの外側でその問い合わせを行ってください。他にも`pg_stat_clear_snapshot()`を呼び出すこともできます。これは現在のトランザクションの統計情報スナップショットを（もしあれば）破棄します。次に統計情報を使用する場合に新しいスナップショットを取り出すことになります。

トランザクションから

は、`pg_stat_xact_all_tables`、`pg_stat_xact_sys_tables`、`pg_stat_xact_user_tables`、`pg_stat_xact_user_functions`、およびこれらのビューの元となっている関数を通じて、自身の統計情報(まだコレクタに送られていない)も参照することができます。これらの数値はトランザクション中に継続的に更新されていくため上記の様な(静的な情報を示す)振る舞いとはなりません。

表 27-1. 標準統計情報ビュー

ビュー名	説明
pg_stat_activity	サーバ当たり 1 行の形式で、状態や現在の問い合わせ等のプロセスの現在の活動状況に関連した情報を表示します。 詳細については pg_stat_activity を参照してください。
pg_stat_bgwriter	バックグラウンドライタプロセスの活動状況に関する統計情報を 1 行のみで表示します。 詳細については pg_stat_bgwriter を参照してください。
pg_stat_database	データベース当たり 1 行の形で、データベース全体の情報を表示します。 詳細については pg_stat_database を参照してください。
pg_stat_all_tables	現在のデータベースの各テーブルごとに 1 行の形で、特定のテーブルへのアクセスに関する統計情報を示します。 詳細については pg_stat_all_tables を参照してください。
pg_stat_sys_tables	システムテーブルのみが表示される点を除き、pg_stat_all_tablesと同じです。
pg_stat_user_tables	ユーザテーブルのみが表示される点を除き、pg_stat_all_tablesと同じです。
pg_stat_xact_all_tables	pg_stat_all_tablesと似ていますが、現在のトランザクションにて実施された処理結果をカウントします。(数値が見える時点では、これらの数値はpg_stat_all_tablesと関連するビューに含まれていません。) このビューでは、有効行数、無効行数、およびバキュームやアナライズの活動は表示しません。
pg_stat_xact_sys_tables	システムテーブルのみが表示される点を除き、pg_stat_xact_all_tablesと同じです。
pg_stat_xact_user_tables	ユーザテーブルのみが表示される点を除き、pg_stat_xact_all_tablesと同じです。
pg_stat_all_indexes	現在のデータベースのインデックスごとに 1 行の形で、特定のインデックスへのアクセスに関する統計情報を示します。 詳細については pg_stat_all_indexes を参照してください。
pg_stat_sys_indexes	システムテーブルのインデックスのみが表示される点を除き、pg_stat_all_indexesと同じです。
pg_stat_user_indexes	ユーザテーブルのインデックスのみが表示される点を除き、pg_stat_all_indexesと同じです。
pg_statio_all_tables	現在のデータベース内のテーブルごとに 1 行の形で、特定のテーブルに対するI/Oに関する統計情報を示します。 詳細については pg_statio_all_tables を参照してください。
pg_statio_sys_tables	システムテーブルのみが表示される点を除き、pg_statio_all_tablesと同じです。
pg_statio_user_tables	ユーザテーブルのみが表示される点を除き、pg_statio_all_tablesと同じです。
pg_statio_all_indexes	現在のデータベース内のインデックスごとに 1 行の形で、特定のインデックスに対するI/Oに関する統計情報を示します。 詳細については pg_statio_all_indexes を示します。
pg_statio_sys_indexes	システムテーブルのインデックスのみが表示される点を除き、pg_statio_all_indexesと同じです。
pg_statio_user_indexes	ユーザテーブルのインデックスのみが表示される点を除き、pg_statio_all_indexesと同じです。
pg_statio_all_sequences	現在のデータベース内のシーケンスごとに 1 行の形で、特定のシーケンスに対するI/Oに関する統計情報を示します。 詳細については pg_statio_all_sequences を参照してください。
pg_statio_sys_sequences	システムシーケンスのみが表示される点を除き、pg_statio_all_sequencesと同じです (現時点では、システムシーケンスは定義されていませんので、このビューは常に空です)。
pg_statio_user_sequences	ユーザシーケンスのみが表示される点を除き、pg_statio_all_sequencesと同じです。
pg_stat_user_functions	追跡された関数ごとに 1 行の形で、関数の実行に関する統計情報を示します。 詳細については pg_stat_user_functions を参照してください。
pg_stat_xact_user_functions	pg_stat_user_functionsと似ていますが、現在のトランザクション中に呼び出されたものだけをカウントします。(数値が見える時点では、これらの数値はpg_stat_user_functionsに含まれていません。)
pg_stat_replication	WAL送信プロセス毎に 1 行の形式で、送信サーバが接続したスタンバイサーバへのレプリケーションに関する統計情報を表示します。 詳細については pg_stat_replication を参照してください。
pg_stat_database_conflicts	データベース毎に 1 行の形式で、スタンバイサーバにおける復旧との競合のためにキャンセルされた問い合わせについてのデータベース全体の統計情報を表示します。 詳細については pg_stat_database_conflicts を参照してください。

インデックス単位の統計情報は、どのインデックスが使用され、どの程度効果があるのかを評価する際に、特に有用です。

pg_statio_ビューは主に、バッファキャッシュの効率を評価する際に有用です。 実ディスク読み取りの数がバッファヒットの数

よりかなり少ないのであれば、そのキャッシュはカーネル呼び出しを行うことなく、ほとんどの読み取り要求を満足させています。しかし、PostgreSQLバッファキャッシュに存在しないデータはカーネルのI/Oキャッシュにある可能性があり、そのため、物理的な読み取りを行うことなく取り出される可能性があるというPostgreSQLのディスクI/Oの取り扱いのため、これらの統計情報は、完全な論拠を提供しません。PostgreSQLのI/O動作に関するより詳細な情報を入手したいのであれば、PostgreSQL統計情報コレクタとカーネルのI/Oの取り扱いの監視を行うオペレーティングシステムユーティリティを組み合わせることを勧めます。

表 27-2. pg_stat_activityビュー

列	型	説明
datid	oid	バックエンドが接続するデータベースのOIDです。
datname	name	バックエンドが接続するデータベースの名前です。
pid	integer	バックエンドのプロセスIDです。
usesysid	oid	バックエンドにログインしたユーザの識別子です。
username	name	バックエンドに接続したユーザの名前です。
application_name	text	バックエンドに接続したアプリケーションの名前です。
client_addr	inet	バックエンドに接続したクライアントのIPアドレスです。このフィールドがNULLである場合、これはクライアントがサーバマシン上のUnixソケット経由で接続されたか、自動バキュームなど内部処理であることを示します。
client_hostname	text	client_addrの逆引き検索により報告された、接続クライアントのホスト名です。IP接続、かつlog_hostnameが有効である場合にのみこのフィールドは非NULLになります。
client_port	integer	クライアントがバックエンドとの通信に使用するTCPポート、もしUnixソケットを使用する場合は-1です。
backend_start	timestamp with time zone	プロセスが開始、つまりクライアントがサーバに接続した時刻です。
xact_start	timestamp with time zone	プロセスの現在のトランザクションが開始した時刻です。活動中のトランザクションがない場合はNULLです。現在の問い合わせがトランザクションの先頭である場合、この列はquery_start列と同じです。
query_start	timestamp with time zone	現在有効な問い合わせが開始した時刻です。もしstateがactiveでない場合は直前の問い合わせが開始した時刻です。
state_change	timestamp with time zone	stateの最終変更時刻です。
waiting	boolean	バックエンドが現在ロックを待機している場合は真です。
state	text	現在のバックエンドの総体的な状態です。以下のいずれかの値を取ることができます。 - active: バックエンドは問い合わせを実行中です。 - idle: バックエンドは新しいクライアントからのコマンドを待機しています。 - idle in transaction: バックエンドはトランザクションの内部にいますが、現在実行中の問い合わせが現在ありません。 - idle in transaction (aborted): この状態はidle in transactionと似ていますが、トランザクション内のある文がエラーになっている点が異なります。 - fastpath function call: バックエンドは近道関数を実行中です。 - disabled: この状態は、このバックエンドでtrack_activitiesが無効である場合に報告されます。
query	text	バックエンドの最も最近の問い合わせテキストです。stateがactiveの場合、現在実行中の問い合わせを意味します。その他のすべての状態では、実行済みの最後の問い合わせを示します。

pg_stat_activityはサーバプロセス毎に、そのプロセスの現在の活動に関連する情報を表示する 1 行を持ちます。

注意: waitingとstate列は独立しています。バックエンドがactive状態である場合、waitingかもしれないし、そうでないかもしれません。状態がactiveであり、waitingが真である場合、問い合わせは実行中ですが、システム内の何らかのロックによりブロックされていることを意味します。

表 27-3. pg_stat_bgwriterビュー

列	型	説明
checkpoints_timed	bigint	これまでに実行された、スケジュールされたチェックポイントの個数です。
checkpoints_req	bigint	これまでに実行された、要求されたチェックポイントの個数です。
checkpoint_write_time	double precision	チェックポイント処理におけるディスクにファイルを書き出す部分に費やされた、ミリ秒単位の総時間です。
checkpoint_sync_time	double precision	チェックポイント処理におけるディスクにファイルを同期する部分に費やされた、ミリ秒単位の総時間です。
buffers_checkpoint	bigint	チェックポイント期間に書き出されたバッファ数です。
buffers_clean	bigint	バックグラウンドライタにより書き出されたバッファ数です。
maxwritten_clean	bigint	バックグラウンドライタが書き出したバッファ数が多過ぎたために、整理用スキャンを停止した回数です。
buffers_backend	bigint	バックエンドにより直接書き出されたバッファ数です。
buffers_backend_fsync	bigint	バックエンドが独自にfsync呼び出しを実行しなかった回数です。（通常は、バックエンドが独自に書き込んだ場合であっても、バックグラウンドライタがこれらを扱います。）
buffers_alloc	bigint	割当られたバッファ数です。
stats_reset	timestamp with time zone	これらの統計情報がリセットされた最終時刻です。

pg_stat_bgwriterビューは常に、クラスタのグローバルデータに関する1つの行を持ちます。

表 27-4. pg_stat_databaseビュー

列	型	説明
datid	oid	データベースのOIDです。
datname	name	データベースの名前です。
numbackends	integer	現在データベースに接続しているバックエンドの個数です。これは、このビューの中で、現在の状態を反映した値を返す唯一の列です。他の列はすべて、最後にリセットされてから累積された値を返します。
xact_commit	bigint	このデータベースの中でコミットされたトランザクション数です。
xact_rollback	bigint	このデータベースの中でロールバックされたトランザクション数です。
blks_read	bigint	データベース内で読み取られたディスクブロック数です。
blks_hit	bigint	バッファキャッシュに既にあることが分かっているために読み取りが不要だったディスクブロック数です（これにはPostgreSQLのバッファキャッシュにおけるヒットのみが含まれ、オペレーティングシステムのファイルシステムキャッシュは含まれません）。
tup_returned	bigint	データベース内の問い合わせで返された行数です。
tup_fetched	bigint	データベース内の問い合わせで取り出された行数です。
tup_inserted	bigint	データベース内の問い合わせで挿入された行数です。
tup_updated	bigint	データベース内の問い合わせで更新された行数です。
tup_deleted	bigint	データベース内の問い合わせで削除された行数です。
conflicts	bigint	データベース内のリカバリで競合したためキャンセルされた問い合わせ数です。（競合はスタンバイサーバ上でのみ起こります。詳細については pg_stat_database_conflicts を参照してください。）
temp_files	bigint	データベース内の問い合わせによって書き出された一時ファイルの個数です。一時ファイルが作成された理由（ソート処理やハッシュ処理）や log temp files の設定に関わらず、すべての一時ファイルが計上されます。
temp_bytes	bigint	データベース内の問い合わせによって一時ファイルに書き出されたデータ量です。一時ファイルが作成された理由や log temp files の設定に関わらず、すべての一時ファイルが計上されます。
deadlocks	bigint	データベース内で検知されたデッドロック数です。
blk_read_time	double	データベース内でバックエンドによりデータファイルブロックの読み取りに費やされた、ミリ秒単

	precision	位の時間です。
blk_write_time	double precision	データベース内でバックエンドによりデータファイルブロックの書き出しに費やされた、ミリ秒単位の時間です。
stats_reset	timestamp with time zone	統計情報がリセットされた最終時刻です。

pg_stat_databaseには、クラスタ内のデータベース毎にデータベース全体の統計情報を示す 1 行が含まれます。

表 27-5. pg_stat_all_tablesビュー

列	型	説明
relid	oid	テーブルのOIDです。
schemaname	name	テーブルが存在するスキーマの名前です。
relname	name	テーブルの名前です。
seq_scan	bigint	テーブル上で初期化されたシーケンシャルスキャンの個数です。
seq_tup_read	bigint	シーケンシャルスキャンによって取り出された有効行の個数です。
idx_scan	bigint	テーブル上で初期化されたインデックススキャンの個数です。
idx_tup_fetch	bigint	インデックススキャンによって取り出された有効行の個数です。
n_tup_ins	bigint	挿入された行数です。
n_tup_upd	bigint	更新された行数です。
n_tup_del	bigint	削除された行数です。
n_tup_hot_upd	bigint	HOT更新（つまりインデックスの更新を別途必要としない）された行数です。
n_live_tup	bigint	有効業の推定値です。
n_dead_tup	bigint	不要行の推定値です。
last_vacuum	timestamp with time zone	テーブルが手作業でバキュームされた最終時刻です。（VACUUM FULLは含まれません）。
last_autovacuum	timestamp with time zone	自動バキュームデーモンによりテーブルがバキュームされた最終時刻です。
last_analyze	timestamp with time zone	テーブルが手作業で解析された最終時刻です。
last_autoanalyze	timestamp with time zone	自動バキュームデーモンによってテーブルが手作業で解析された最終時刻です。
vacuum_count	bigint	テーブルが手作業でバキュームされた回数です。（VACUUM FULLは含まれません）。
autovacuum_count	bigint	テーブルが自動バキュームデーモンによりバキュームされた回数です。
analyze_count	bigint	テーブルが手作業で解析された回数です。
autoanalyze_count	bigint	テーブルが自動バキュームデーモンによって解析された回数です。

pg_stat_all_tablesビューは現在のデータベース内のテーブル（TOASTテーブルを含む）毎に 1 行の形式で、特定のテーブルへのアクセスに関する統計情報を表示します。 pg_stat_user_tablesおよびpg_stat_sys_tablesビューにも同じ情報が含まれますが、それぞれユーザテーブルとシステムテーブルのみにフィルタされています。

表 27-6. pg_stat_all_indexesビュー

列	型	説明
relid	oid	インデックス対象のテーブルのOIDです。
indexrelid	oid	インデックスのOIDです。
schemaname	name	インデックスが存在するスキーマの名前です。
relname	name	インデックス対象のテーブルの名前です。
indexrelname	name	インデックスの名前です。
idx_scan	bigint	インデックスに対して初期化されたインデックススキャンの個数です。
idx_tup_read	bigint	インデックスに対するスキャンにより返されたインデックス項目の個数です。
idx_tup_fetch	bigint	インデックスを使用する単純なインデックススキャンによって取り出された有効テーブル行数です。

pg_stat_all_indexesビューは、現在のデータベース内のインデックス毎に、特定のインデックスへのアクセスに関する統計情報を示す1行を保持します。 pg_stat_user_indexesとpg_stat_sys_indexesも同じ情報を保持しますが、ユーザ向けのインデックスとシステム向けのインデックスに対する行のみを保持するようにフィルタ処理されています。

単純インデックススキャンまたは"ビットマップ"インデックススキャンのいずれかを介してインデックスを使用することができません。 ビットマップスキャンでは、複数のインデックスの出力をANDやOR規則で組み合わせることができます。 このため、ビットマップスキャンが使用される場合、特定インデックスと個々のヒープ行の取り出しとを関連づけることが困難です。 したがってビットマップスキャンでは、使用したインデックスのpg_stat_all_indexes.idx_tup_read個数を増やし、そのテーブルのpg_stat_all_tables.idx_tup_fetch個数を増やしますが、pg_stat_all_indexes.idx_tup_fetchを変更しません。

注意: idx_tup_readとidx_tup_fetch個数は、ビットマップスキャンがまったく使用されていない場合でも異なります。 idx_tup_readはインデックスから取り出したインデックス項目を計上し、idx_tup_fetchはテーブルから取り出した有効行を計上するからです。 インデックスを用いて不要行やまだコミットされていない行が取り出された場合やインデックスオンリースキャン法によりヒープの取り出しが回避された場合に、後者は減少します。

表 27-7. pg_statio_all_tablesビュー

列	型	説明
relid	oid	テーブルのOIDです。
schemaname	name	テーブルが存在するスキーマの名前です。
relname	name	テーブルの名前です。
heap_blks_read	bigint	テーブルから読み取られたディスクブロック数です。
heap_blks_hit	bigint	テーブル内のバッファヒット数です。
idx_blks_read	bigint	テーブル上のすべてのインデックスから読み取られたディスクブロック数です。
idx_blks_hit	bigint	テーブル上のすべてのインデックス内のバッファヒット数です。
toast_blks_read	bigint	テーブルのTOASTテーブル（もしあれば）から読み取られたディスクブロック数です。
toast_blks_hit	bigint	テーブルのTOASTテーブル（もしあれば）におけるバッファヒット数です。
tidx_blks_read	bigint	テーブルのTOASTテーブルのインデックス（もしあれば）から読み取られたディスクブロック数です。
tidx_blks_hit	bigint	テーブルのTOASTテーブルのインデックス（もしあれば）におけるバッファヒット数です。

pg_statio_all_tablesビューは現在のデータベース内のテーブル（TOASTテーブルを含む）ごとに、特定のテーブルのI/Oに関する統計情報情報を示す1行を保持します。 pg_statio_user_tablesとpg_statio_sys_tablesには同じ情報が保持されますが、ユーザテーブルとシステムテーブルに関する行のみを持つようにフィルタ処理がなされています。

表 27-8. pg_statio_all_indexesビュー

列	型	説明
relid	oid	このインデックスに対応するテーブルのOIDです。
indexrelid	oid	インデックスのOIDです。
schemaname	name	インデックスが存在するスキーマの名前です。
relname	name	このインデックスに対応するテーブルの名前です。
indexrelname	name	インデックスの名前です。
idx_blks_read	bigint	インデックスから読み取られたディスクブロック数です。
idx_blks_hit	bigint	インデックスにおけるバッファヒット数です。

pg_statio_all_indexesビューは、現在のデータベース内のインデックス毎に、特定のインデックスへのI/Oに関する統計情報を持つ1行を保持します。 pg_statio_user_indexesとpg_statio_sys_indexesも同じ情報を保持しますが、それぞれユーザ向けのインデックスとシステム向けのインデックスに対する行のみを保持するようにフィルタ処理されています。

表 27-9. pg_statio_all_sequencesビュー

列	型	説明
relid	oid	シーケンスのOIDです。

schemaname	name	シーケンスが存在するスキーマの名前です。
relname	name	シーケンスの名前です。
blks_read	bigint	シーケンスから読み取られたディスクブロック数です。
blks_hit	bigint	シーケンスにおけるバッファヒット数です。

pg_statio_all_sequencesビューは現在のデータベース内のシーケンスごとに、特定シーケンスにおけるI/Oに関する統計情報を示す 1 行を保持します。

表 27-10. pg_stat_user_functionsビュー

列	型	説明
funcid	oid	関数のOIDです。
schemaname	name	関数が存在するスキーマの名前です。
funcname	name	関数の名前です。
calls	bigint	関数が呼び出された回数です。
total_time	double precision	関数とその関数から呼び出されるその他の関数で費やされた、ミリ秒単位の総時間です。
self_time	double precision	その関数から呼び出されるその他の関数で費やされた時間を含まない、関数自身で費やされた、ミリ秒単位の総時間です。

pg_stat_user_functionsビューは追跡された関数毎に、その関数の実行に関する統計情報を 1 行保持します。
[track_functions](#)パラメータは関数が追跡されるかどうかを正確に制御します。

表 27-11. pg_stat_replicationビュー

列	型	説明
pid	integer	WAL送信プロセスのプロセスIDです。
usesysid	oid	WAL送信プロセスにログインしたユーザのOIDです。
username	name	WAL送信プロセスにログインしたユーザの名前です。
application_name	text	WAL送信処理に接続したアプリケーションの名前です。
client_addr	inet	WAL送信処理に接続したクライアントのIPアドレスです。 このフィールドがNULLの場合、クライアントがサーバマシン上のUnixソケット経由で接続したことを示します。
client_hostname	text	client_addrのDNS逆引き検索により報告された、接続クライアントのホスト名です。 IP接続、かつ log_hostname が有効である場合にのみこのフィールドは非NULLになります。
client_port	integer	クライアントがWAL送信処理との通信に使用するTCPポート、もしUnixソケットを使用する場合は-1です。
backend_start	timestamp with time zone	プロセスが開始、つまりクライアントがWAL送信処理に接続した時刻です。
state	text	現在のWAL送信処理の状態です。
sent_location	text	接続上に送信された最後のトランザクションログ位置です。
write_location	text	スタンバイサーバによってディスクに書き出された最後のトランザクションログ位置です。
flush_location	text	スレーブによってディスクに吐き出された最後のトランザクションログ位置です。
replay_location	text	スタンバイサーバ上のデータベースに再生された最後のトランザクションログ位置です。
sync_priority	integer	このスタンバイサーバが同期スタンバイとして選択される優先度です。
sync_state	text	スタンバイサーバの同期状態です。

pg_stat_replicationビューには、WAL送信プロセス毎に、送信処理に接続したスタンバイサーバへのレプリケーションに関する統計情報を示す 1 行を保持します。 直接接続されたサーバのみが一覧表示されます。 下流のスタンバイサーバに関する情報はありません。

表 27-12. pg_stat_database_conflictsビュー

列	型	説明

datid	oid	データベースのOIDです。
datname	name	データベースの名前です。
confl_tablespace	bigint	データベースにおいて、削除されたテーブル空間のためにキャンセルされた問い合わせの個数です。
confl_lock	bigint	データベースにおいて、ロック時間切れのためにキャンセルされた問い合わせの個数です。
confl_snapshot	bigint	データベースにおいて、古いスナップショットのためにキャンセルされた問い合わせの個数です。
confl_bufferpin	bigint	データベースにおいて、ピンが付いたバッファのためにキャンセルされた問い合わせの個数です。
confl_deadlock	bigint	データベースにおいて、デッドロックのためにキャンセルされた問い合わせの個数です。

`pg_stat_database_conflicts`ビューは、データベース毎にスタンバイサーバでのリカバリと競合するためにキャンセルされた問い合わせに関するデータベース全体の統計情報を示す 1 行を保持します。 マスタサーバでは競合は発生しませんので、スタンバイサーバ上の情報のみが保持されます。

27.2.3. 統計情報関数

統計情報を参照する他の方法は、上述の標準ビューによって使用される基礎的な統計情報アクセス関数と同じ関数を使用した問い合わせを作成することで設定することができます。 こうした関数の名前などに関する詳細については、標準ビューの定義を参照してください。（例えばpsqlでは`\d+ pg_stat_activity`を発行してください。） データベースごとの統計情報についてのアクセス関数は、どのデータベースに対して報告するのかを識別するためにデータベースのOIDを取ります。 テーブルごと、インデックスごとの関数はテーブルの、もしくはインデックスのOIDを取ります。 関数ごとの統計情報の関数は、関数のOIDを取ります。 これらの関数を使用して参照できるテーブルとインデックス、および関数は現在のデータベース内のものだけであることに注意してください）。

その他の統計情報収集に関連した関数を表27-13に示します。

表 27-13. その他の統計情報関数

関数	戻り値の型	説明
<code>pg_backend_pid()</code>	integer	現在のセッションを扱うサーバプロセスのプロセスID。
<code>pg_stat_get_activity(integer)</code>	setof record	PIDが指定された場合、それに該当するバックエンドの情報のレコード、NULLが指定された場合はシステム上のアクティブな各バックエンドの情報がレコードが返されます。 返される情報内容は <code>pg_stat_activity</code> の一部と同じです。
<code>pg_stat_clear_snapshot()</code>	void	現在の統計情報スナップショットを破棄します。
<code>pg_stat_reset()</code>	void	現在のデータベースに関する統計カウンタすべてをゼロにリセットします（スーパーユーザ権限が必要です）。
<code>pg_stat_reset_shared(text)</code>	void	引数に応じて、クラスタ全体の統計情報カウンタの一部をゼロに戻します（スーパーユーザ権限が必要です）。 <code>pg_stat_reset_shared('bgwriter')</code> を呼び出すと、 <code>pg_stat_bgwriter</code> で示される値すべてがゼロになります。
<code>pg_stat_reset_single_table_counters(oid)</code>	void	現在のデータベース内にある、ひとつのテーブルあるいはインデックスの統計情報をゼロにリセットします（スーパーユーザ権限が必要です）。
<code>pg_stat_reset_single_function_counters(oid)</code>	void	現在のデータベース内にある、ひとつの関数の統計情報をゼロにリセットします（スーパーユーザ権限が必要です）。

`pg_stat_activity`ビューの基礎となる`pg_stat_get_activity`関数は、各バックエンドプロセスに関して利用可能な情報をすべて含むレコード集合を返します。 この情報の一部のみを入手することがより簡便である場合があるかもしれません。 このような場合、表27-14に示す、古めのバックエンド単位の統計情報アクセス関数を使用することができます。 これらのアクセス関数は、1 から現在活動中のバックエンドの個数までの値を取る、バックエンドID番号を使用します。
`pg_stat_get_backend_idset`関数は、これらの関数を呼び出すために、活動中のバックエンド毎に 1 行を生成する簡便な方法を提供します。 例えば以下はすべてのバックエンドについてPIDと現在の問い合わせを示します。

```
SELECT pg_stat_get_backend_pid(s.backendid) AS pid,
       pg_stat_get_backend_activity(s.backendid) AS query
FROM (SELECT pg_stat_get_backend_idset() AS backendid) AS s;
```

表 27-14. バックエンド単位の統計情報関数

関数	戻り値の型	説明
pg_stat_get_backend_idset()	setof integer	現在活動中のバックエンドID番号（1から活動中のバックエンドの個数まで）を設定します。
pg_stat_get_backend_activity(integer)	text	バックエンドが最後に行った問い合わせテキストです。
pg_stat_get_backend_activity_start(integer)	timestamp with time zone	最後の問い合わせが開始された時刻です。
pg_stat_get_backend_client_addr(integer)	inet	バックエンドに接続したクライアントのIPアドレスです。
pg_stat_get_backend_client_port(integer)	integer	クライアントが通信に使用しているTCPポート番号です。
pg_stat_get_backend_dbid(integer)	oid	バックエンドが接続するデータベースのOIDです。
pg_stat_get_backend_pid(integer)	integer	バックエンドのプロセスIDです。
pg_stat_get_backend_start(integer)	timestamp with time zone	プロセスが開始された時刻です。
pg_stat_get_backend_userid(integer)	oid	バックエンドにログインしたユーザのOIDです。
pg_stat_get_backend_waiting(integer)	boolean	バックエンドが現在ロックを待機している場合は真です。
pg_stat_get_backend_xact_start(integer)	timestamp with time zone	現在のトランザクションが開始された時刻です。

[前のページ](#)
標準的なUnixツール

[ホーム](#)
[上に戻る](#)

[次のページ](#)
ロックの表示

27.3. ロックの表示

この他に、データベース活動状況の監視に役立つツールとして `pg_locks` システムテーブルがあります。これにより、データベース管理者はロックマネージャ内の未解決のロックに関する情報を参照することができます。例えば、この機能を使用すると以下のことができます。

- 現在未解決のロック、特定データベース内のリレーション上のロック、特定のリレーションのロック、または特定の PostgreSQL セッションが保持するロックを全て表示する。
- 最も許可されにくいロック（データベースクライアント間で競合の原因になる可能性がある）を持つ、現在のデータベースにおけるリレーションを表示する。
- 競合によって変動するデータベースの全トラフィックの範囲に加えて、全体的なデータベースの性能に対するロック競合の影響を判断する。

`pg_locks` ビューの詳細は、[項45.58](#)にあります。PostgreSQLのロックと同時実行性についての詳細は、[第13章](#)を参照してください。

27.4. 動的追跡

PostgreSQLは、データベースサーバの動的追跡をサポートする機能を提供します。これにより、外部ユーティリティをコードの特定のポイントで呼び出すことができ、追跡を行うことができますようになります。

多くの追跡やプローブ用のポイントは、すでにソースコード内部に存在します。これらのプローブはデータベースの開発者や管理者が使うことを意図しています。デフォルトでは、これらのプローブはPostgreSQLにコンパイルされません。ユーザは明示的にconfigureスクリプトでプローブを有効にするように設定する必要があります。

現在、OpenSolaris、Solaris 10、そして Mac OS X Leopardで利用可能なDTraceユーティリティのみがサポートされています。DTraceは将来FreeBSDやその他のオペレーティングシステムで利用できるようになることが予定されています。[SystemTap](#)プロジェクトではDtrace相当の機能をLinux向けに提供しています。他の動的追跡ユーティリティのサポートは、src/include/utils/probes.h内のPG_TRACEマクロ定義を変更することで、理論上は可能です。

27.4.1. 動的追跡のためのコンパイル

デフォルトでは、プローブは有効ではありません。そのため、PostgreSQLでプローブが利用できるようにするためにconfigureスクリプトで明示的に設定しなければなりません。DTraceサポートを含めるには、configureに`--enable-dtrace`を指定します。詳細は[項15.4](#)を参照してください。

27.4.2. 組み込み済みのプローブ

[表27-15](#)で示されるように、多くの標準的なプローブがソースコード内で提供されています。[表27-16](#)はプローブで使用している型を示しています。また、PostgreSQL内の可観測性を強化するためのプローブ追加が可能です。

表 27-15. 組み込み済みのDtraceプローブ

名前	パラメータ	説明
transaction-start	(LocalTransactionId)	新しいトランザクションの開始を捕捉するプローブ。arg0はトランザクションIDです。
transaction-commit	(LocalTransactionId)	トランザクションの正常終了を捕捉するプローブ。arg0はトランザクションIDです。
transaction-abort	(LocalTransactionId)	トランザクションの異常終了を捕捉するプローブ。arg0はトランザクションIDです。
query-start	(const char *)	問い合わせ処理の開始を捕捉するプローブ。arg0は問い合わせ文字列です。
query-done	(const char *)	問い合わせ処理の正常終了を捕捉するプローブ。arg0は問い合わせ文字列です。
query-parse-start	(const char *)	問い合わせのパーズ処理の開始を捕捉するプローブ。arg0は問い合わせ文字列です。
query-parse-done	(const char *)	問い合わせのパーズ処理の正常終了を捕捉するプローブ。arg0は問い合わせ文字列です。
query-rewrite-start	(const char *)	問い合わせの書き換え処理の開始を捕捉するプローブ。arg0は問い合わせ文字列です。
query-rewrite-done	(const char *)	問い合わせの書き換え処理の正常終了を捕捉するプローブ。arg0は問い合わせ文字列です。
query-plan-start	()	問い合わせのプランナ処理の開始を捕捉するプローブ。
query-plan-done	()	問い合わせのプランナ処理の正常終了を捕捉するプローブ。
query-execute-start	()	問い合わせの実行(エクゼキュータ)処理の開始を捕捉するプローブ。
query-execute-done	()	問い合わせの実行(エクゼキュータ)処理の正常終了を捕捉するプローブ。

statement-status	(const char *)	サーバプロセスによるpg_stat_activity.statusの状態の更新を捕捉するプローブ。arg0は新しい状態の文字列です。
checkpoint-start	(int)	チェックポイントの開始を捕捉するプローブです。arg0はチェックポイントの種類の違い(shutdown、immediate、force)を区別するためのビットフラグを持っています。
checkpoint-done	(int, int, int, int, int)	チェックポイントの正常終了を捕捉するプローブ(以下に示すプローブはチェックポイント進行に従い順番に捕捉されます)。arg0は書き込まれたバッファ数、arg1はバッファの総数、arg2、3、4はそれぞれ追加、削除、再利用されたxlogの数です。
clog-checkpoint-start	(bool)	CLOG部分のチェックポイントの開始を捕捉するプローブ。arg0がtrueならば通常のチェックポイントであり、falseならばシャットダウン時のチェックポイントを示します。
clog-checkpoint-done	(bool)	CLOG部分のチェックポイントの正常終了を捕捉するプローブ。arg0はclog-checkpoint-startと同じ意味を持ちます。
subtrans-checkpoint-start	(bool)	サブトランザクション部分のチェックポイントの開始を捕捉するプローブ。arg0がtrueならば通常のチェックポイントであり、falseならばシャットダウン時のチェックポイントを示します。
subtrans-checkpoint-done	(bool)	サブトランザクション部分のチェックポイントの正常終了を捕捉するプローブ。arg0はsubtrans-checkpoint-startと同じ意味を持ちます。
multixact-checkpoint-start	(bool)	マルチトランザクション部分のチェックポイントの開始を捕捉するプローブ。arg0がtrueならば通常のチェックポイントであり、falseならばシャットダウン時のチェックポイントを示します。
multixact-checkpoint-done	(bool)	マルチトランザクション部分のチェックポイントの正常終了を捕捉するプローブ。arg0はmultixact-checkpoint-startと同じ意味を持ちます。
buffer-checkpoint-start	(int)	チェックポイントのバッファ書き込み部分の開始を捕捉するプローブ。arg0はチェックポイントの種類の違い(shutdown、immediate、force)を区別するためのビットフラグを持っています。
buffer-sync-start	(int, int)	チェックポイント中のダーティバッファの書き出し開始を捕捉するプローブ(どのバッファが書き出す必要があるのかを判定した後です)。arg0はバッファの総数で、arg1は現在ダーティであり、書き出す必要のあるバッファ数です。
buffer-sync-written	(int)	チェックポイント中のそれぞれのバッファを書き出し後を捕捉するプローブ。arg0はバッファのIDを示します。
buffer-sync-done	(int, int, int)	全てのダーティバッファの書き出し後を捕捉するプローブ。arg0はバッファの総数です。arg1はチェックポイント処理により実際に書き出されたバッファ数です。arg2は書き出されるであろうと見積もられたバッファ数(buffer-sync-startのarg1相当)です。この違いはチェックポイント中に他のプロセスにより書き出されたバッファ分です。
buffer-checkpoint-sync-start	()	カーネルへのダーティバッファの書き出し処理発行の後、そして同期書き出し要求を開始する前を捕捉するプローブ。
buffer-checkpoint-done	()	バッファからディスクへの同期書き出し処理の終了を捕捉するプローブ。
twophase-checkpoint-start	()	二相コミット部分のチェックポイントの開始を捕捉するプローブ。
twophase-checkpoint-done	()	二相コミット部分のチェックポイントの正常終了を捕捉するプローブ。
buffer-read-start	(ForkNumber, BlockNumber, Oid, Oid, Oid, int, bool)	バッファ読み込みの開始を捕捉するプローブ。arg0はとarg1は読み込みページのフォーク番号とブロック番号です(ただし、リレーションの拡張要求があった場合、arg1は-1になるでしょう)。arg2、arg3、arg4は対象のリレーションを識別するテーブル空間、データベース、そしてリレーションのOIDです。arg5は一時テーブルをローカルバッファに作成していればそのバックエンドのIDであり、無効なバックエンドID(-1)であれば共有バッファを指します。arg6はtrueならばリレーションの拡張要求、falseは通常の読み込みを示します。

buffer-read-done	(ForkNumber, BlockNumber, Oid, Oid, int, bool, bool)	バッファ読み込みの終了を捕捉するプローブ。arg0とarg1は読み込みページのフォーク番号とブロック番号です(もしリレーションの拡張要求があった場合、arg1は新たに追加されたブロックの番号を含みます)。arg2、arg3、arg4は対象のテーブルを識別するテーブル空間、データベース、そしてテーブルのOIDです。arg5は一時テーブルをローカルバッファに作成していればそのバックエンドのIDであり、無効なバックエンドID(-1)であれば共有バッファを指します。arg6はtrueならばリレーションの拡張要求、falseは通常の読み込みを示します。arg7はtrueならばバッファがプール内にある、falseはプール内に無かったことを示します。
buffer-flush-start	(ForkNumber, BlockNumber, Oid, Oid, Oid)	共有バッファへの書き込み要求開始を捕捉するプローブ。arg0とarg1はそのページのフォーク番号とブロック番号です。arg2、arg3、arg4は対象のリレーションを識別するテーブル空間、データベース、そしてテーブルのOIDです。
buffer-flush-done	(ForkNumber, BlockNumber, Oid, Oid, Oid)	書き込み要求の終了を捕捉するプローブ。(これはカーネルへデータを渡したタイミングのみを反映していることに注意してください; 大抵、この時点ではまだ実際にディスクへ書き込まれていません。)引数はbuffer-flush-startと同じです。
buffer-write-dirty-start	(ForkNumber, BlockNumber, Oid, Oid, Oid)	サーバプロセスによるダーティバッファの書き出し開始を捕捉するプローブ。(もしこれが頻発するようでしたら、 shared_buffers が少な過ぎるか、バックグラウンドリタ制御のパラメータの調節が必要なことを意味します。)arg0とarg1はそのページのフォーク番号とブロック番号です。arg2、arg3、arg4は対象のリレーションを識別するテーブル空間、データベース、そしてリレーションのOIDです。
buffer-write-dirty-done	(ForkNumber, BlockNumber, Oid, Oid, Oid)	ダーティバッファの書き出しの終了を捕捉するプローブです。引数はbuffer-write-dirty-startと同じです。
wal-buffer-write-dirty-start	()	WALバッファ領域の不足によるサーバプロセスのダーティなWALバッファを書き出しを捕捉するプローブ。(もしこれが頻発するようでしたら、 wal_buffers が小さすぎることを意味します。)
wal-buffer-write-dirty-done	()	ダーティなWALバッファの書き出し終了を捕捉するプローブ。
xlog-insert	(unsigned char, unsigned char)	WALレコードの挿入を捕捉するプローブ。arg0はレコードのリソースマネージャ(rmid)です。arg1は情報フラグです。
xlog-switch	()	WALセグメントのスイッチ要求を捕捉するプローブです。
smgr-md-read-start	(ForkNumber, BlockNumber, Oid, Oid, Oid, int)	リレーションからのブロック読み込みの開始を捕捉するプローブ。arg0とarg1はそのページのフォーク番号とブロック番号です。arg2、arg3、arg4は対象のリレーションを識別するテーブル空間、データベース、そしてリレーションのOIDです。arg5は一時テーブルをローカルバッファに作成していればそのバックエンドのIDであり、無効なバックエンドID(-1)であれば共有バッファを指します。
smgr-md-read-done	(ForkNumber, BlockNumber, Oid, Oid, Oid, int, int, int)	ブロックの読み込み終了を捕捉するプローブ。arg0とarg1はそのページのフォーク番号とブロック番号です。arg2、arg3、arg4は対象のリレーションを識別するテーブル空間、データベース、そしてリレーションのOIDです。arg5は一時テーブルをローカルバッファに作成していればそのバックエンドのIDであり、無効なバックエンドID(-1)であれば共有バッファを指します。arg6は実際に読み込んだバイト数、arg7はリクエストされた読み込みバイト数です(もし、これらに差異があった場合、何らかの問題があることを示します)。
smgr-md-write-start	(ForkNumber, BlockNumber, Oid, Oid, Oid, int)	リレーションへのブロック書き出しの開始を捕捉するプローブ。arg0とarg1はそのページのフォーク番号とブロック番号です。arg2、arg3、arg4は対象のリレーションを識別するテーブル空間、データベース、そしてリレーションのOIDです。arg5は一時テーブルをローカルバッファに作成していればそのバックエンドのIDであり、無効なバックエンドID(-1)であれば共有バッファを指します。
smgr-md-write-done	(ForkNumber, BlockNumber, Oid, Oid, Oid, int, int, int)	ブロックの書き出し終了を捕捉するプローブ。arg0とarg1はそのページのフォーク番号とブロック番号です。arg2、arg3、arg4は対象のリレーションを識別するテーブル空間、データベース、そしてリレーションのOIDです。arg5は一時テーブルをローカルバッファに作成していればそのバックエンドのIDであり、無効なバックエンドID(-1)であれば共有バッファを指します。arg6は実際に書き出したバイト数、arg7はリクエストされた書き出しバイト数です(もし、これらに差異があった場合、何らかの問題があることを示します)。
sort-start	(int, bool, int, int, bool)	ソート処理の開始を捕捉するプローブ。arg0は対象データがヒープ、インデックス、またはdatumのどれかを示します。arg1はtrueならば一意性を必要としていることを示します。arg2はカラムのキー数です。arg3は許容されている作業メモリ(work_mem)のキロバイト数です。arg4はtrueならばソート結果に対するランダムアクセスが要求されていることを示します。

sort-done	(bool, long)	ソート処理の終了を捕捉するプローブ。arg0はtrueならば外部ソート、falseは内部ソートを示します。arg1は外部ソートで使用されたディスクブロック数、もしくは内部ソートで使用されたメモリのキロバイト数を示します。
lwlock-acquire	(LWLockId, LWLockMode)	LWLock(軽量ロック)の取得を捕捉するプローブ。arg0は軽量ロックのIDを示します。arg1は要求されたロックモード(排他、もしくは共有)を示します。
lwlock-release	(LWLockId)	軽量ロックの開放を捕捉するプローブ(ただし、開放待ちのプロセスはまだこのロックを取得できないことに注意して下さい)。arg0は軽量ロックのIDを示します。
lwlock-wait-start	(LWLockId, LWLockMode)	軽量ロックを即座に取得できなかったため、サーバプロセスがロックを利用できるまでロック待機状態になった際の開始を捕捉するプローブ。arg0は軽量ロックのIDを示します。arg1は要求されたロックモード(排他、もしくは共有)を示します。
lwlock-wait-done	(LWLockId, LWLockMode)	サーバプロセスの軽量ロック待機の開放を捕捉するプローブ(実際にはまだロックは取得していません)。arg0は軽量ロックのIDを示します。arg1は要求されたロックモード(排他、もしくは共有)を示します。
lwlock-condacquire	(LWLockId, LWLockMode)	呼び出し元が待機しないことを指定した際の、軽量ロックの獲得成功を捕捉するプローブ。arg0は軽量ロックのIDを示します。arg1は要求されたロックモード(排他、もしくは共有)を示します。
lwlock-condacquire-fail	(LWLockId, LWLockMode)	呼び出し元が待機しないことを指定した際の、軽量ロックの獲得失敗を捕捉するプローブ。arg0は軽量ロックのIDを示します。arg1は要求されたロックモード(排他、もしくは共有)を示します。
lock-wait-start	(unsigned int, unsigned int, unsigned int, unsigned int, LOCKMODE)	通常のロック(lmgr lock)を即座に取得できなかったため、サーバプロセスがロックを利用できるまでロック待ち状態になった際の開始を捕捉するプローブ。arg0からarg3はロックされたオブジェクトの識別用タグ領域です。arg4はロックされたオブジェクトのタイプを示します。arg5は要求されたロックの種類を示します。
lock-wait-done	(unsigned int, unsigned int, unsigned int, unsigned int, LOCKMODE)	通常のロック(lmgr lock)要求の待機終了を捕捉するプローブ(つまりロックを取得した)。 引数はlock-wait-startと同じです。
deadlock-found	()	デッドロック検知器によるデッドロックの発見を捕捉するプローブ。

表 27-16. プローブパラメータで使われる型の定義

型	定義
LocalTransactionId	unsigned int
LWLockId	int
LWLockMode	int
LOCKMODE	int
BlockNumber	unsigned int
Oid	unsigned int
ForkNumber	int
bool	char

27.4.3. プローブの利用

以下の例では、システムにおけるトランザクション数を解析するDTraceスクリプトを示します。性能試験前後でpg_stat_databaseのスナップショットを取ることで代替可能です。

```
#!/usr/sbin/dtrace -qs

postgresql$1:::transaction-start
{
    @start["Start"] = count();
    self->ts = timestamp;
}
```



```

postgresql$1:::transaction-abort
{
    @abort["Abort"] = count();
}

postgresql$1:::transaction-commit
/self->ts/
{
    @commit["Commit"] = count();
    @time["Total time (ns)"] = sum(timestamp - self->ts);
    self->ts=0;
}

```

実行すると、例のDスクリプトは以下のような出力をします。

```

# ./txn_count.d `pgrep -n postgres` or ./txn_count.d <PID>
^C

Start                                71
Commit                              70
Total time (ns)                     2312105013

```

注意: 基本となる追跡ポイントの互換性はありますが、SystemTapはDTraceと異なる追跡スクリプトの表記を用います。表記に関して特に注意すべき点として、SystemTapでは参照する追跡ポイント名のハイフンの代わりに二重のアンダースコアを用いる必要があります。これは将来的なSystemTapのリリースで修正されることを期待しています。

DTraceスクリプトの作成には注意が必要であり、デバッグが必要であることは忘れないでください。さもないと、収集される追跡情報の意味がなくなるかもしれません。ほとんどの場合、見つかる問題はシステムではなく使用方法の間違いです。動的追跡を使用して見つかった情報に関して議論を行う際には、スクリプトの検査や議論もできるようにスクリプトも含めるようにしてください。

さらに多くのサンプルスクリプトがPgFoundryの[dtrace project](https://pgfoundry.org/dtrace-project/)にあります。

27.4.4. 新規プローブの定義

開発者が望めばコード内に新しくプローブを定義することができます。しかし、これには再コンパイルが必要です。下記は、新規プローブの定義の手順です。

1 11 プローブの名前とプローブの処理を通じて取得可能とするデータを決めます

1 11 src/backend/utils/probes.dにプローブの定義を追加します

1 11 もし、プローブポイントを含むモジュールがpg_trace.hをインクルードしてなければそれをインクルードし、ソースコード中のプローブを行いたい場所にTRACE_POSTGRESQLマクロを挿入します

1 11 再コンパイルを行い、新規プローブが利用できるか確認します

例: これはトランザクションIDを用いて新規トランザクションを追跡するプローブ追加の仕方の例です。

1 11 プローブ名をtransaction-startとし、パラメータとしてLocalTransactionId型を必要とすることを決めます

1 11 src/backend/utils/probes.dにプローブの定義を追加します:

```
probe transaction__start(LocalTransactionId);
```

プローブ名に二重のアンダースコアを使用する場合は注意して下さい。DTraceスクリプトでプローブを用いる場合、二重のアンダースコアをハイフンに置き換える必要があります。そのため、transaction-startがユーザ向けの文書に記載される名前となります。

1 11 コンパイル時に、transaction__startはTRACE_POSTGRESQL_TRANSACTION_STARTと呼ばれるマクロに変換されます(ここではアンダースコアはひとつになります)。このマクロは、pg_trace.hをインクルードすることにより使用可能となります。このマクロをソースコード中の適切な箇所へ追加していきます。この場合、以下のようになります。

```
TRACE_POSTGRESQL_TRANSACTION_START(vxid.localTransactionId);
```

1 11 再コンパイル後に新しいバイナリでサーバを起動し、下記の様なDTraceコマンドの実行により新たに追加したプローブが利用可能なチェックします。下記の様な出力が確認できるはずです:

```

# dtrace -ln transaction-start
ID      PROVIDER      MODULE      FUNCTION NAME
18705   postgresql49878     postgres    StartTransactionCommand transaction-start
18755   postgresql49877     postgres    StartTransactionCommand transaction-start
18805   postgresql49876     postgres    StartTransactionCommand transaction-start

```

```
18855 postgresql49875      postgres      StartTransactionCommand transaction-start
18986 postgresql49873      postgres      StartTransactionCommand transaction-start
```

Cのソースコードに追跡用のマクロを追加する際、いくつかの注意点があります:

- プローブの引数に指定したデータ型がマクロで使用される変数のデータ型と一致するよう注意しなければなりません。でなければ、コンパイル時にエラーとなるでしょう。
- ほとんどのプラットフォームでは、もしPostgreSQLが`--enable-dtrace`付きでビルドされた場合、何の追跡がされなかったとしても、制御がマクロを通過する際にはいつでも追跡用マクロの引数が評価されます。ごく少数のローカルな変数を報告するような場合はそれほど心配はいりません。ただし、追加した高価な関数が引数を呼ぶような場合は注意して下さい。もしそのような関数が必要ある場合、追跡が実際に可能かどうかをチェックするマクロを利用して不要なマクロの実施を避けることを考慮してください:

```
if (TRACE_POSTGRESQL_TRANSACTION_START_ENABLED())
    TRACE_POSTGRESQL_TRANSACTION_START(some_function(...));
```

各追跡マクロは対応する`ENABLED`マクロを持っています。

[前のページ](#)
ロックの表示

[ホーム](#)
[上に戻る](#)

[次のページ](#)
ディスク使用量の監視

第 28 章 ディスク使用量の監視

目次

- 28.1. [ディスク使用量の決定](#)
- 28.2. [ディスク容量不足による問題](#)

本章ではPostgreSQLデータベースシステムのディスク使用量を監視する方法について説明します。

28.1. ディスク使用量の決定

各テーブルには、データの大部分が格納されるプライマリヒープディスクファイルが備わっています。もしテーブルが、可能な限りの値を持つ列を持つ時は、テーブルに関連付けられたTOASTファイルもあります。このファイルは、メインテーブルに収納するには大き過ぎる値をテーブルに格納するために使用されます（[項56.2](#)を参照してください）。TOASTテーブルが存在する場合は、そのテーブルにインデックスが1つあります。基本テーブルに関連付けられたインデックスが存在することもあります。テーブルとインデックスはそれぞれ別のディスクファイルに格納されます。このファイルが1ギガバイトを超える場合は、複数のファイルになります。これらのファイルの命名規約について[項56.1](#)で説明します。

ディスクスペースの監視は、次の3つの方法で行えます。[表9-63](#)にあるSQL関数を使用する方法と `oid2name` モジュールを使用する方法、およびシステムカタログを手動で調べる方法です。SQL関数を使用する方法が、一般的に一番簡単な方法です。本セクションの残りの部分で、システムカタログを調査することによりこの方法を示します。

バキュームされて間もないデータベース、もしくは解析されたデータベース上でpsqlを使用することにより、どのようなテーブルでもディスクの使用量を調べる問い合わせを発行できます。

```
SELECT pg_relation_filepath(oid), relpages FROM pg_class WHERE relname = 'customer';
```

```
pg_relation_filepath | relpages
-----+-----
base/16384/16806    |        60
(1 row)
```

1ページは通常8キロバイトです（`relpages`はVACUUMとANALYZE、さらにCREATE INDEXといったいくつかのDDLによってのみ更新されることに注意してください）。もしテーブルのディスクファイルを直接調べるときは、ファイルのパス名称に注目して下さい。

TOASTテーブルで使用されている容量を示すには、以下のような問い合わせを使用してください。

```
SELECT relname, relpages
FROM pg_class,
     (SELECT reltoastrelid
      FROM pg_class
      WHERE relname = 'customer') AS ss
WHERE oid = ss.reltoastrelid OR
       oid = (SELECT reltoastidxid
              FROM pg_class
              WHERE oid = ss.reltoastrelid)
ORDER BY relname;
```

```
relname          | relpages
-----+-----
pg_toast_16806   |         0
pg_toast_16806_index |         1
```

インデックスサイズについても、以下のように簡単に表示できます。

```
SELECT c2.relname, c2.relpages
FROM pg_class c, pg_class c2, pg_index i
WHERE c.relname = 'customer' AND
      c.oid = i.indrelid AND
      c2.oid = i.indexrelid
ORDER BY c2.relname;
```

```
relname          | relpages
-----+-----
customer_id_index |        26
```

この情報を使用して、以下のように簡単に最大のテーブルとインデックスを見つけ出すことができます。

```
SELECT relname, relpages
FROM pg_class
ORDER BY relpages DESC;
```

```
relname          | relpages
-----+-----
bigtable         |       3290
customer         |       3144
```

28.2. ディスク容量不足による問題

データベース管理者の最も重要なディスク監視作業は、ディスクが容量不足になっていないことを確認することです。容量不足となったディスクにより、データが破壊されることはありません。しかし、データ破壊が起こりやすくなる可能性があります。もし WAL ファイルを持つディスクが一杯になった場合は、データベースサーバはパニックを起こし結果的にシャットダウンします。

他のデータを削除しても、ディスクに空き容量を用意できない場合、テーブル空間を使用することによって、データベースファイルのいくつかを他のファイルシステムに移動させることができます。詳細は [項21.6](#) を参照してください。

ティップ: 一部のファイルシステムは、容量がほぼ一杯になっている場合にパフォーマンスが悪くなります。ですから、ディスクがほぼ一杯になる前に余裕をもって対策を取ってください。

システムでユーザ単位のディスククォータをサポートしている場合、当然ながらデータベースもサーバを実行するユーザに割り当てられたクォータに従います。クォータを超えた場合、ディスク容量が完全になくなった時と同じ悪影響が発生します。

第 29 章信頼性とログ先行書き込み

目次

- 29.1. [信頼性](#)
- 29.2. [ログ先行書き込み\(WAL\)](#)
- 29.3. [非同期コミット](#)
- 29.4. [WALの設定](#)
- 29.5. [WALの内部](#)

本章では、効率的に信頼できる操作を得るためにログ先行書き込みがどのように使用されているかについて説明します。

29.1. 信頼性

信頼性は、すべての深刻なデータベースシステムで重要な特性です。PostgreSQLは信頼できる操作を保証することができます。信頼できる操作とは、コミットされたトランザクションにより記録されたデータはすべて不揮発性の領域に格納され、電源断、オペレーティングシステムの障害、ハードウェアの障害（当然ですが、不揮発性の領域自体の障害は除きます。）があっても安全であるという点です。通常、コンピュータの永続的格納領域（ディスク装置など）へのデータ書き込みの成功がこの条件を満たします。実際、コンピュータに致命的な障害が発生したとしても、もしディスク装置が無事ならば、類似のハードウェアを持つ別のコンピュータに移すことができ、コミットされたトランザクションを元通りに復元できます。

データを周期的にディスクプラッタに書き出すことは簡単な操作に思われるかもしれませんが、そうではありません。ディスク装置は主メモリ、CPU、コンピュータの主メモリとディスクプラッタの間にある各種のキャッシュ層と比べ非常に低速であるからです。まず、オペレーティングシステムのバッファキャッシュが存在します。これは頻繁にアクセス要求があるディスクブロックをキャッシュし、ディスクへの書き込みをまとめます。好運にもすべてのオペレーティングシステムがバッファキャッシュをディスクに強制書き込みさせる方法をアプリケーションに提供しています。PostgreSQLはこの機能を使用します。（これを調整する方法については[wal_sync_method](#)パラメータを参照してください。）

次に、ディスク装置のコントローラキャッシュが存在する可能性があります。特に、RAIDコントローラカードでは、これは一般的です。これらはwrite-throughキャッシュ、つまり、データが届いた時に即座に書き込みがディスク装置に対して行なわれる、場合があります。また、write-backキャッシュ、つまり、多少遅れて書き込みがディスク装置に対して行なわれる、の場合もあります。こうしたキャッシュではディスクコントローラキャッシュが揮発性、つまり、電源障害の際にその内容が失われてしまいますので、信頼性に関する障害が発生する可能性があります。より優れたコントローラカードではバッテリーバックアップ付き装置(BBUs)を持ちます。つまり、システムの電源が落ちている状態でもキャッシュに電源を供給します。後で電源が復旧した後に、データがディスク装置に書き出されます。

最後に、ほとんどのディスク装置がキャッシュを持っています。一部はwrite-throughであり、一部はwrite-backです。ディスクコントローラキャッシュの場合と同様にwrite-backのディスク装置キャッシュの場合にはデータが損失する恐れがあります。一般消費者向けのIDEおよびSATA装置では特に、よくwrite-backキャッシュを使用しています。これは電源障害時にデータが残りません。多くのソリッドステートドライブ(SSD)も同様に揮発性のwrite-backキャッシュを持っています。

これらのキャッシュは、大抵は無効にできます。しかしながらオペレーティングシステムやドライブの種類によってその方法は異なります。

- Linux上で`hdparm -I`を使用することでIDEドライブのキャッシュについて調べることができます。Write cacheの次に*があれば書き込みキャッシュが有効になっています。`hdparm -W 0`により書き込みキャッシュを無効にできます。SCSIドライブであれば`sdparm`を使うことで調査が可能です。`sdparm --get=WCE`によりキャッシュが有効かどうかの確認ができ、`sdparm --clear=WCE`により無効にすることができます。
- FreeBSDでは、IDEドライブに対して`atacontrol`により確認ができ、そして書き込みキャッシュを無効にするには`/boot/loader.conf`の`hw.ata.wc=0`を利用します。SCSIドライブに対しては`camcontrol identify`を確認に使用することができ、`sdparm`を使用できる場合にはそれを用いて書き込みキャッシュの確認と変更が可能です。
- Solarisでは、ディスクの書き込みキャッシュは`format -e`で制御できます。（SolarisのZFSファイルシステムは、独自のディスクキャッシュ書き出しコマンドを発行しているため、ディスクの書き込みキャッシュを有効にしても安全です。）
- Windowsでは、もし`wal_sync_method`が`open_datasync`(デフォルト)の場合、`My Computer¥Open¥disk drive¥Properties¥Hardware¥Properties¥Policies¥Enable write caching on the disk`のチェックを外すことで、書き込みキャッシュを無効にできます。もう一つの方法としては、`wal_sync_method`を`fsync`か`fsync_writethrough`に設定し、書き込みキャッシュを使用しないようにします。
- Mac OS Xでは、`wal_sync_method`を`fsync_writethrough`に設定することで書き込みキャッシュを使用しないようにします。

最近のSATAドライブ(ATAPI-6以降)はドライブキャッシュの書き出しコマンド(`FLUSH CACHE EXT`)を提供しており、一方SCSIドライブでは従来から類似の`SYNCHRONIZE CACHE`コマンドをサポートしていました。これらのコマンドは、直接PostgreSQLに発行されませんが、いくつかのファイルシステム(例えばZFSやext4)では、それらをwrite-backが有効なドライブへデータを書き出すために使います。不幸なことに、このようなwriteバリアを持つファイルシステムは、バッテリーバックアップ付き装置(BBU)のディスクコントローラと組み合わせた際に、好ましくは動作しません。このような処理の流れにおいて、同期コマンドはコントローラキャッシュにあるデータを全てディスクへ強制的に書き込みを行うため、BBUのメリットの大半を失わせています。[pg_test_fsync](#)ユーティリティを使うことで、あなたの環境が影響を受けるかどうかを確認できます。もし影響を受けるようであれば、ファイルシステムのwriteバリアを無効にするか、(オプションがあれば)ディスクコントローラを再設定することで、BBUによる性能上の効果を得ることできるでしょう。もしwriteバリアを無効にした場合は、バッテリーが動作していることを確認しておきましょう。バッテリーの欠陥はデータロスの可能性に繋がります。ファイルシステムやディスクコントローラ的设计者が、いずれはこの動作を修正してくれることが望まれます。

オペレーティングシステムが、ストレージハードウェアに書き込み要求を送信した時、データが不揮発性のストレージ領域に本当に届いたかどうかを確認することはほぼできません。ですので、全てのストレージ構成がデータ整合性を保証することをよく確認しておくことは、管理者の責任です。バッテリーを持たない書き込みキャッシュを持つコントローラを使用しないでください。装置レベルでは、もし装置が停止前にデータが書き出されることを保証できないのであれば、write-backキャッシュを無効にしてください。もしSSDを使っている場合、多くのドライブはデフォルトでキャッシュ書き出しコマンドを無視することに注意して下さい。 [diskchecker.pl](#)を使うことで、I/Oサブシステムの動作の信頼性をテストすることができます。

ディスクプラッタの書き込み操作自体によってもデータ損失が発生することがあります。ディスクプラッタは、通常512バイトのセクタに分割されています。物理的な読み込み操作、書き込み操作はすべて、セクタ全体を処理します。書き込み要求がディスクに達した時、その要求は512バイトの倍数(PostgreSQLでは大抵一度に8192バイトもしくは16セクタ分です)になるでしょう。そして電源断により、任意のタイミングで書き込み処理が失敗することがあります。これは一部の512バイトのセクタに書き込みが行なわれたのに、残りのセクタには書き込みが行なわれていない状況を意味します。こうした問題の対策として、PostgreSQLは、ディスク上の実際のページを変更する前に定期的にページ全体のイメージを永続的なWAL格納領域に書き出します。これにより、PostgreSQLはクラッシュリカバリ時に部分的に書き出されたページをWALから復旧させることができます。もし、部分的なページ書き込みを防止できるファイルシステムソフトウェア（例えばZFS）を使うのであれば、[full_page_writes](#)を無効にしてページイメージ作成を無効にすることができます。バッテリーバックアップ付き（BBU）のディスクコントローラでは、フルページ（8kB）がBBUへ書き込まれることを保証できなければ、部分的なページ書き出しを防止できません。

[前のページ](#)

信頼性とログ先行書き込み

[ホーム](#)

[上に戻る](#)

[次のページ](#)

ログ先行書き込み(WAL)

29.2. ログ先行書き込み(WAL)

ログ先行書き込み (WAL) はデータの一貫性を確実にするための標準的な手法です。詳細にわたって全てを網羅しているとは言えませんが、たいいていのトランザクション処理について書かれた書籍に記載されています。簡単に言うと、WALの基本的な考え方は、(テーブルやインデックスがある) データファイルへの変更は、ログへの記録、つまり、変更内容を記述したログレコードが永続格納領域に書き出された後にのみ書き出されなければならないということです。このような手順に従って処理を行えば、たとえクラッシュが起きてもログを使ってデータベースをリカバリすることができるため、トランザクションのコミットの度にデータページをディスクに吐き出す必要がなくなります。リカバリの時点では、まず、データページに対してまだ行われていない変更分はログレコードを使って再実行されます (これがREDOとして知られているロールフォワードリカバリです)。

ティップ: WALによりデータベースファイルの中身を障害後にリストアするため、信頼性のある格納領域にあるデータファイルやWALファイルに対しては、ジャーナルファイルシステムは必要ありません。実際、特に、もしファイルシステムのデータをディスクにフラッシュさせている場合には、ジャーナリングのオーバーヘッドは性能を劣化させることがあります。幸運なことに、ジャーナリング中のデータのフラッシュをマウントオプションにより無効にできることが多いです。例えばLinuxのext3ファイルシステムでは、`data=writeback`と指定します。ジャーナルファイルシステムは障害後の起動速度を改善します。

WALを使用することでディスクへの書き込み回数が大幅に減少します。と言うのも、トランザクションがコミットされたことを保証するために、そのトランザクションで変更された全てのデータファイルではなく、ログファイルだけをディスクに吐き出す必要があるからです。ログファイルへの書き込みはシーケンシャルに行われるため、データページを吐き出すコストに比べログファイルの同期はずっと低コストになります。これは特に、データ格納領域の様々な部分を変更する小さなトランザクションを多く扱うサーバで顕著に現れます。さらに、サーバが小規模なトランザクションを同時に多く処理する時、ログファイルを一度`fsync`することで、多くのトランザクションをコミットすることができる場合もあります。

また、WALにより、[項24.3](#)で説明するオンラインバックアップとポイントインタイムリカバリをサポートすることができます。WALのデータを保持することにより、そのWALデータが範囲内とする任意の時点に戻すことができます。単純にデータベースの主となる物理バックアップをインストールし、WALログを目的の時点まで単に再生することで実現できます。さらに、物理バックアップはインスタンス化可能なデータベース状態のスナップショットである必要もありません。ある程度の時間を経過して作成されたバックアップであっても、その期間用のWALを再生することにより、内部の不整合を修復します。

29.3. 非同期コミット

非同期コミットとは、トランザクションをより高速に完了することができるオプションです。 もっとも最近のトランザクションがデータベースがクラッシュしてしまった場合に失われるという危険があります。 これは、多くのアプリケーションで受け入れられるトレードオフです。

前説で説明した通り、通常トランザクションのコミットは同期的です。 サーバはトランザクションのWALレコードが永続的格納領域に記録されるまで、クライアントに成功したことを通知することを待機します。 従って、直後にサーバクラッシュといった障害があったとしても、コミットされたと報告されたトランザクションは保持されることをクライアントは保証されます。 しかし、短期のトランザクションでは、この遅延はトランザクションの処理時間の大半を占める要素となります。 非同期コミットモードを選択することは、サーバがWAL記録が実際に作成された通りにディスクに書き込まれるより前に、トランザクションの論理的な完了をもって成功したと通知することを意味します。 これにより、小規模なトランザクションでスループットがかなり向上します。

非同期コミットにはデータ損失の危険があります。 トランザクションの完了をクライアントに通知してからトランザクションが本当に完了する（つまり、サーバクラッシュしても損失がないことが保証される）までの間にわずかな時間が存在します。 したがって、クライアントがトランザクションが記録されているという仮定を元に外部的な動作を行うクライアントがある場合は、非同期コミットを使用すべきではありません。 例えば、銀行では、ATMの現金分配を記録するトランザクションで非同期コミットを使用してはいけません。 しかし、イベント記録など多くのシナリオでは、この種の保証を持って格納する必要はありません。

非同期コミットによりもたらされる危険性は、データの破壊ではなくデータの損失です。 データベースがクラッシュした場合、最後にフラッシュされた記録までWALを再生することで復旧が行われます。 このため、データベースは内部で一貫性を持った状態に復旧されますが、ディスクにフラッシュされていないトランザクションはすべてそこには反映されません。 したがって、影響を受けるのは、最後に行われたいくつかのトランザクションの損失です。 トランザクションはコミットされた順に再生されますので、一貫性が失われることはありません。 例えば、トランザクションBが以前に行われたトランザクションAの結果に依存した変更を行った場合、Bの影響が保存されている限り、Aの影響が失われることは起こり得ません。

ユーザは各トランザクションでコミットモードを選択することができます。 このため、同時実行されるトランザクションを同期的、および非同期の両方でコミットさせることができます。 これにより、性能とトランザクションの信頼性の確実性との間で柔軟な選択を行うことができます。 コミットモードはユーザによる設定が可能なパラメータ `synchronous_commit` で制御されます。 このパラメータは、設定パラメータを設定することができる全ての方法で変更することが可能です。 あるひとつのトランザクションで使用されるモードは、トランザクションのコミットが始まった時の `synchronous_commit` の値に依存します。

例えば `DROP TABLE` などの特定のユーティリティコマンドでは、`synchronous_commit` の設定に関わらず、強制的に同期的コミットが行われます。 これにより、サーバのファイルシステムとデータベースの論理的な状態との間の一貫性が保証されます。

`PREPARE TRANSACTION` などの2層コミットをサポートするコマンドもまた、常に同期的です。

もし非同期コミットとそのトランザクションのWAL記録の書き込みの間の危険期間にデータベースがクラッシュしたとすると、そのトランザクションでなされた変更は失われるでしょう。 バックグラウンドプロセス ("WALライター") が未書き込みのWAL記録を `wal_writer_delay` ミリ秒毎にディスクに吐き出しますので、この危険期間は制限されます。 WALライターは稼働中に一回ページ全体を書き込むように設計されているため、危険期間の実際の最大の長さは `wal_writer_delay` の3倍です。

注意

即時モードのシャットダウンはサーバクラッシュと同じことですので、吐き出されていない非同期コミットが失われることになります。

非同期コミットでは `fsync = off` という設定とは異なる動作になります。 `fsync` はサーバ全体に関する設定であり、すべてのトランザクションの動作を変更します。 これは、PostgreSQLにおける、データベースの別の場所への同期書き込みの試行に関するすべてのロジックを無効にします。 このため、システムクラッシュ (PostgreSQL自体の失敗ではなくハードウェアやオペレーティングシステムのクラッシュ) の結果、予測できないデータベース状態の破壊が起こります。 非同期コミットはデータ破壊の危険性はなく、多くの状況では `fsync` を無効にした場合に得られる性能向上とほぼ同等の性能を提供します。

また `commit_delay` も非同期コミットと類似のように見えますが、これは実のところ同期コミットの一方法です。 (実際、非同期コミット時 `commit_delay` は無視されます。) 同期コミットがWALをディスクに吐き出す直前に、こうしたトランザクションによって実行される一度の吐き出しにより、ほぼ同時期にコミットを行う他のトランザクションの分も処理できるようにすることを目的とした遅延が `commit_delay` により発生します。 `commit_delay` の設定は、多くの同時にコミットを行うトランザクションが存在する時にのみ役に立ち、性能を損なうことなく実際に有用な値に調整することは困難です。

29.4. WALの設定

データベースの性能に影響するようなWALに関連した設定パラメータが複数あります。本節では、その使い方を説明します。サーバ設定パラメータの設定方法についての詳細は[第18章](#)を参照してください。

チェックポイントは、一連のトランザクションにおいて、それ以前に書き出された全ての情報でヒープとインデックスファイルが更新されていることが保証されている時点を示します。チェックポイントでは、全てのダーティデータページがディスクに吐き出され、特別なチェックポイントレコードがログファイルに書き込まれます。(これらのの変更は以前にWALファイルへ書き出されています。)クラッシュした時、クラッシュからの復旧処理は最後のチェックポイントレコードを見つけ、ログの中でどのレコード(これはredoレコードと呼ばれています)から復旧処理がREDOログ操作を開始すべきかを決定します。このチェックポイント以前になされたデータの変更は、すでにディスク上にあることが保証されています。従って、チェックポイント後、redoレコード内のそのチェックポイント以前のログセグメントは不要となり、再利用または削除することができます(WALアーカイブが行われる場合、このログセグメントは削除もしくは再利用される前に保存されなければなりません)。

チェックポイント処理は、全てのダーティデータページをディスクへ書き出すため、大きなI/O負荷を発生させます。このため、チェックポイント処理は利用するI/O帯域を絞り込むようにしており、チェックポイントの開始から次のチェックポイントの開始前までI/Oの発生が続きます。これは、チェックポイント処理中の性能劣化を極力抑える効果があります。

サーバのチェックポイントプロセスは自動的にチェックポイントを頻繁に実行します。[checkpoint_segments](#)ログセグメント数に達するか、または[checkpoint_timeout](#)秒が経過するか、どちらかの条件が満たされるとチェックポイントが作成されます。デフォルトの設定では、それぞれ3セグメントと300秒(5分)となっています。前回のチェックポイント以降書き出すWALがない場合、[checkpoint_timeout](#)が経過したとしても新しいチェックポイントが飛ばされます。WALアーカイブ処理を使用しており、かつ、データ損失の可能性を限定するためにファイルのアーカイブ頻度に対する下限を設定したい場合、このチェックポイントパラメータではなく[archive_timeout](#)パラメータを調節しなければなりません。また、`CHECKPOINT` SQLコマンドで強制的にチェックポイントを作成することもできます。

[checkpoint_segments](#)、[checkpoint_timeout](#)、またはその両者を減少させると、チェックポイントはより頻繁に行われます。これにより、(再処理に要する時間がより少なくなりますので)クラッシュ後の修復は高速になります。しかし、より頻繁に行われるようになる、変更されたデータページの吐き出しにより増大するコストとバランスを考えなければなりません。[full_page_writes](#)が設定されている(デフォルトです)場合、他に考慮しなければならない点があります。データページの一貫性を保証するために、各チェックポイント後の最初に変更されるデータページは、そのページ全体の内容がログに保存されることとなります。このような場合、チェックポイントの間隔を少なくすることは、WALログへの出力を増加させ、間隔を短くする目的の一部を無意味にします。また、確実により多くのディスクI/Oが発生します。

チェックポイントはかなり高価なものです。1番の理由は、この処理は現時点の全てのダーティバッファを書き出す必要があること、2番目の理由は、上記のようにその後余計なWALの書き込みが発生することです。そのため、チェックポイント用のパラメータを高くし、チェックポイントがあまりにも頻発することがないようにすることを勧めます。簡単なチェックポイント用のパラメータの健全性検査として、[checkpoint_warning](#)パラメータを設定することができます。チェックポイントの発生間隔が[checkpoint_warning](#)秒未満の場合、[checkpoint_segments](#)の増加を勧めるメッセージがサーバのログに出力されます。このメッセージが稀に現れたとしても問題にはなりませんが、頻出するようであれば、チェックポイントの制御パラメータを増加させるべきです。[checkpoint_segments](#)を十分高く設定していないと、大規模なCOPY転送などのまとまった操作でこうした警告が多く発生するかもしれません。

ページ書き出しの集中による入出力システムの溢れを防ぐために、チェックポイント期間のダーティバッファの書き出しは一定の期間に分散されます。この期間は[checkpoint_completion_target](#)により制御され、チェックポイント間隔の割合として指定されます。I/Oの割合は、チェックポイントの起動時から[checkpoint_segments](#) WALセグメントが消費された時、あるいは、指定した[checkpoint_timeout](#)秒が経過した時、このどちらかが発生するとすぐに、チェックポイントが完了するように調整されます。デフォルトの0.5という値では、PostgreSQLは、次のチェックポイントが始まるまでのおよそ半分の時間で各チェックポイントが完了するものと想定できることとなります。通常の操作においてほぼ最大のI/Oスループットに近いようなシステムでは、チェックポイントにおけるI/O負荷を減らすために[checkpoint_completion_target](#)を増やすことを勧めます。この欠点は、延長されたチェックポイントがリカバリ時に影響をあたえることです。リカバリ時に使用できるように、より多くのWALセグメントを保持する必要があるためです。[checkpoint_completion_target](#)を最大の1.0に設定することもできますが、より低く抑えること(おそらく最大で0.9)が最善です。チェックポイントには、ダーティバッファを書き出す以外の活動も含まれているからです。1.0という設定は、ある時点でチェックポイントが完了しなくなるという結果に陥ります。これは必要なWALセグメント数が想定以上に変動することになり、性能の劣化が発生することとなります。

WALセグメントファイルは常に少なくとも1つあり、また、通常は $(2 + \text{checkpoint\_completion\_target}) * \text{checkpoint\_segments} + 1$ 、もしくは $\text{checkpoint\_segments} + \text{wal\_keep\_segments} + 1$ より多くはありません。各セグメントファイルは通常16メガバイト(このサイズはサーバのコンパイル時に変更可能)です。このことから、WALで必要とされる領域を推定できます。普通、古いセグメントファイルが不要になった時、それらは再利用(将来順番に使われるセグメントとなるように名前が変更)されます。短時間のログ出力のピークのためにセグメントファイル数が $3 * \text{checkpoint\_segments} + 1$ を超えた場合、システムは、この上限以下になるまで、不要になったセグメントファイルを再利用せずに、削除します。

アーカイブからのリカバリもしくはスタンバイモードにおいて、サーバでは定期的に通常運用でのチェックポイント処理と似たリスタートポイント処理を行います。これは、すでに再生されたWALを再度読み込む必要がないよう、ディスクに現在の状態を強制的に書き込み、`pg_control`ファイルを更新します。また`pg_xlog`中の古いログセグメントを再利用できるようにします。リスタートポイント処理は、少なくとも1回のチェックポイントが再生されており、最後のリスタートポイント処理から`checkpoint_timeout`秒が経過した際に発生します。スタンバイモードでは、最後のリスタートポイント処理時から`checkpoint_segments`分のログセグメントが再生され、かつ少なくとも1回のチェックポイントが再生されていた場合にもリスタートポイント処理が走ります。リスタートポイント処理はチェックポイントが記録された時にしか実施されないの、マスタ側のチェックポイント処理よりも発生頻度は低いでしょう。

よく使われる2つの内部用WAL関数があります。 `LogInsert`と`LogFlush`です。 `LogInsert`は共有メモリ上のWALバッファに新しいレコードを挿入します。 新しいレコードを挿入する余地がない時は、`LogInsert`は、満杯になったWALバッファを書き込み（カーネルキャッシュに移動）しなければいけません。 これは望ましいことではありません。 なぜなら、データベースへの低レベルの変更（例えば行の挿入）の度に`LogInsert`が呼ばれますが、そのような場合には変更を受けたページに対して排他ロックがかかっており、それゆえこの操作は可能な限り高速に実行されなければなりません。 さらに悪いことには、WALバッファへの書き込みの際に、さらに時間がかかる、強制的な新しいログセグメントの生成が必要となるかもしれません。 通常、WALの書き込み、吐き出しは`LogFlush`要求で実施されます。 これはたいていの場合、トランザクションコミットの際に永続的な記憶領域にトランザクションレコードが吐き出されることを保証するために行われます。 ログ出力が大量に行われるシステムでは、`LogInsert`によって必要となる書き込みを防ぐほどには`LogFlush`要求が頻繁に起こらないかもしれません。 そういうシステムでは、[wal_buffers](#)設定パラメータを変更してWALバッファの数を増やしてください。 [full_page_writes](#)が設定され、かつ、システムが高負荷状態である場合、この値を高くすることで、各チェックポイントの直後の応答時間を滑らかにすることができます。

[commit_delay](#)パラメータは、`LogInsert`がコミットレコードをログに書き込んでから`LogFlush`が行われるまでの間にサーバプロセスが何マイクロ秒休止するかを定義します。 この遅延により、他のサーバプロセスがコミットレコードをログに書き込んだ後、それら全てのログレコードを1回のログ同期で吐き出すことができます。 [fsync](#)が無効な場合や、[commit_siblings](#)より少ない数のセッションしか現在アクティブなトランザクションに存在しない場合は、休止しません。 この処理により、すぐにコミットしそうなセッションがない時でも休止してしまうことを防ぐことができます。 たいていのプラットフォームでは、休止の最小単位は10ミリ秒であることに注意してください。 ですから、1マイクロ秒から10,000マイクロ秒までの間での0以外の`commit_delay`の設定は、同じ効果になるでしょう。 これらのパラメータの適切な値はまだ明確ではありません。 実験が必要です。

[wal_sync_method](#)パラメータはPostgreSQLがカーネルに対してWAL更新のディスクへの書き込みを要求する方法を決定します。 `fsync_writethrough`を除き、どういう設定でも信頼性は同じはずです。 `fsync_writethrough`は他のオプションがそうしないときでも、時々ディスクキャッシュの書き出しを強制することができます。 しかしながら、プラットフォームによってどれが一番速いのがまったく違います。 [pg_test_fsync](#)モジュールを使ってオプションの速度テストを行うことができます。 ちなみに、このパラメータは`fsync`が無効になっている場合は役に立ちません。

[wal_debug](#)設定パラメータを有効にすることで、`LogInsert`と`LogFlush`というWAL呼び出しは毎回サーバログにログが残ります（このパラメータをサポートするようにPostgreSQLをコンパイルする必要があります）。 将来このオプションはより一般的な機構に置き換わる可能性があります。

[前のページ](#)
非同期コミット

[ホーム](#)
[上に戻る](#)

[次のページ](#)
WALの内部

29.5. WALの内部

WALは自動的に有効になります。WALログが必要とするディスク容量を確保すること、そして必要ならばチューニングすることを除いては ([項29.4](#)を参照)、管理者は何もする必要はありません。

WALログは、データディレクトリ以下の`pg_xlog`ディレクトリに、通常16メガバイトのサイズを持つセグメントファイルの集合として格納されています(ただし、このサイズはサーバ構築時の`--with-wal-segsize`というconfigureオプションで変更できます)。各セグメントは通常8キロバイトのページに分割されます(このサイズは`--with-wal-blocksize`というconfigureオプションで変更できます)。ログレコード用のヘッダは`access/xlog.h`に記述されています。レコードの内容は、ログの対象となる事象の種類によって異なります。セグメントファイルは名前として000000010000000000000000から始まる、常に増加する数が与えられています。数字は巡回しませんが、利用可能な数字を使い尽くすには非常に長い時間がかかるはずです。

主要なデータベースファイルが置いてあるディスクとは別のディスクにログを置くと利点があります。これは`pg_xlog`ディレクトリを別の場所に(もちろんサーバを終了しておいてから)移動し、主データディレクトリ以下の元々の場所から新しい場所へのシンボリックリンクを張ることによって可能となります。

WALの目的である、確実にデータベースレコードが変更される前にログが書き出されることは、実際にはキャッシュにしかデータがなく、ディスクには格納されていない時に格納に成功したとカーネルに虚偽の報告を行うことによって失われる可能性があります。そのような状況では、電源が落ちた際に、復旧不可能なデータの破壊が起こることがあります。管理者は、PostgreSQLのWALログを保持しているディスク装置がそのような嘘の報告をしないように保証するべきです。([項29.1](#)を参照して下さい。)

チェックポイントが実行され、ログが吐き出された後、チェックポイントの位置は`pg_control`に保存されます。したがって、リカバリの開始の際は、サーバはまず`pg_control`を読み、次にチェックポイントレコードを読みます。そして、チェックポイントレコード内で示されたログの位置から前方をスキャンすることでREDO処理を行います。データページの内容全体は、チェックポイント後の最初のページ変更時にログ内に保存されますので([full page writes](#)パラメータが無効にされていないという前提です)、そのチェックポイント以降に変更された全てのページは一貫した状態に復旧されます。

`pg_control`が壊れた場合に備え、ログセグメントを逆順に読み(すなわち新しいものから古いものへと)、最終チェックポイントを見つける方法を実際には実装した方が良いでしょう。まだこれはできていません。`pg_control`はかなり小さなもの(1ディスクページ未満)ですので、一部のみ書き込みされるという問題は起こりません。またこの書き込みの時点では、`pg_control`自体の読み込みができないことによるデータベースエラーという報告はありません。このため、`pg_control`は理屈では弱点ですが、実質問題になりません。

第 30 章 リグレッションテスト

目次

30.1. [テストの実行](#)

- 30.1.1. [一時的なインストレーションに対するテストの実行](#)
- 30.1.2. [既存のインストレーションに対するテストの実行](#)
- 30.1.3. [ホットスタンバイのテスト](#)
- 30.1.4. [ロケールと符号化方式](#)
- 30.1.5. [追加のテスト](#)

30.2. [テストの評価](#)

- 30.2.1. [エラーメッセージの違い](#)
- 30.2.2. [ロケールの違い](#)
- 30.2.3. [日付と時刻の違い](#)
- 30.2.4. [浮動小数点数の違い](#)
- 30.2.5. [行の順序の違い](#)
- 30.2.6. [スタック長の不足](#)
- 30.2.7. ["乱数" テスト](#)

30.3. [各種の比較用ファイル](#)

30.4. [テストが網羅する範囲の検証](#)

リグレッションテストとは、PostgreSQLのSQL実装についての包括的なテストの集まりです。 リグレッションテストでは、標準SQLの操作に加えてPostgreSQLの拡張SQL機能もテストします。

30.1. テストの実行

リグレッションテストは既にインストールされ稼働中のサーバや、ビルドツリー内の一時的なインストレーションに対して実行することができます。さらに、テストの実行には"並行"と"連続"モードがあります。連続モードでは個々のテストスクリプトを単独で実行し、並行モードでは複数のサーバプロセスを実行し、テストをグループ化して並行的に実行します。並行テストではプロセス間通信とロック機能が正常に作動しているかをテストします。

30.1.1. 一時的なインストレーションに対するテストの実行

構築後、インストール前に並行リグレッションテストを行う場合には、最上位のディレクトリで以下のように入力してください。

```
gmake check
```

(または、`src/test/regress`ディレクトリに移動して、そこで実行してください。) 実行すると、ユーザ定義のトリガ関数のサンプルなど、複数の補助的なファイルがまず作成され、それからテストドライバのスクリプトを実行します。終了したら以下のような表示がされるはずです。

```
=====
All 115 tests passed.
=====
```

これが表示されなければ、テストは失敗したことになります。"失敗"を深刻な問題であると推測する前に、以下の [項30.2](#) を参照してください。

この試験方法では、一時的にサーバを起動させるため、rootユーザでは動作しません (サーバはrootでは起動しません)。rootで構築作業を行ったとしても、すべてを最初からやり直す必要はありません。リグレッションテストのディレクトリを他のユーザから書き込み可能にし、そのユーザでログインしてテストを再実行してください。例えば、以下のようにします。

```
root# chmod -R a+w src/test/regress
root# su - joeuser
joeuser$ cd top-level build directory
joeuser$ gmake check
```

(この方法には"セキュリティリスク"が1つだけあります。それは、別のユーザがこっそりリグレッションテストの結果を変更することが可能ということです。ユーザ権限の付与には常識を働かせてください。)

あるいは、インストール後にテストを実行することも可能です。

古いPostgreSQLのインストレーションが既に存在している場所にPostgreSQLをインストールするように構築した場合、新しいバージョンをインストールする前に`gmake check`を行うと、新しいプログラムがインストール済みの共有ライブラリを使用しようとするために試験が失敗することになります。(典型的な症状は、未定義シンボルに関するエラーメッセージです。)古いインストレーションを上書きする前に試験を行いたいのであれば、`configure --disable-rpath`で構築する必要があります。しかし、このオプションを最終的なインストレーションで使用することは推奨しません。

並行リグレッションテストは、実行したユーザのユーザIDを使用して相当数のプロセスを起動します。現在、最大で20個の並行テストスクリプトが同時に実行されますが、これは合計40個のプロセスが実行されることを意味します。各テストスクリプトに対して、1つのサーバプロセスと1つのpsqlプロセスが存在するためです。ですので、使用するシステムでユーザ当たりのプロセス数に制限を加えている場合は、その上限が少なくとも50程度であることを確認してください。さもないと、並行テストにおいて、ランダムに発生しているように見える失敗が発生するかもしれません。この上限を変更できない場合は、`MAX_CONNECTIONS`パラメータを編集して、並行度を減らすことができます。例えば、以下は同時実行数を10以下で実行します。

```
gmake MAX_CONNECTIONS=10 check
```

30.1.2. 既存のインストレーションに対するテストの実行

インストール ([第15章](#)を参照) 後にテストを実行するには、[第17章](#)で説明したようにデータ領域を初期化し、サーバを起動させ、そして以下を入力してください。

```
gmake installcheck
```

もしくは、並行テストの場合は以下を入力してください。

```
gmake installcheck-parallel
```

テストは、PGHOST環境変数とPGPORT環境変数で指定がない限り、ローカルホストのサーバに接続し、デフォルトのポート番号を使用します。

ソース配布には、オプションとなっている手続き言語と一部のcontribモジュールに対するリグレッションテストも含まれています。現在、これらのテストはインストール済みのサーバに対してのみ使用することができます。構築されインストールされたすべての手続き言語に対してテストを実行するには、構築ツリーのsrc/plディレクトリに移動し、以下を実行してください。

```
gmake installcheck
```

また、これをsrc/plのいずれかのサブディレクトリで実行して、1つの手続き言語のみに対するテストを行うこともできます。テストが提供されているすべてのcontribモジュールに対するテストを実行するには、構築ツリーのcontribディレクトリに移動し、以下を実行してください。

```
gmake installcheck
```

事前にcontribモジュールが構築されインストールされていなければなりません。また、これをcontribのサブディレクトリで実行して、1つのモジュールのみに対してテストを行うことができます。

30.1.3. ホットスタンバイのテスト

ソース配布は、ホットスタンバイの静的な挙動に対するリグレッションテストも含んでいます。これらのテストは、稼働しているプライマリサーバと、ファイルベースのログ転送、またはストリーミングレプリケーションによって新規のWALの変更を受け付けられる稼働中のスタンバイサーバを必要とします。これらのサーバは、自動的に作成されませんし、ここにはセットアップのドキュメント也没有ありません。必要なコマンドや関連する問題について記述されている、既存のドキュメントのさまざまなセクションを参照してください。

最初に、"regression"という名前のデータベースをプライマリに作成します。

```
psql -h primary -c "CREATE DATABASE regression"
```

次に、準備されているスクリプトsrc/test/regress/sql/hs_primary_setup.sqlをregressionデータベース上で実行し、以下のように変更をスタンバイに伝播させます。

```
psql -h primary -f src/test/regress/sql/hs_primary_setup.sql regression
```

ここで、試験用のデフォルト接続がスタンバイサーバの試験環境であることを確認してください。そしてリグレッションテスト用のディレクトリからstandbycheckターゲットを実行してください。

```
cd src/test/regress
gmake standbycheck
```

いくつかの極端な挙動を、スクリプトを用いることでプライマリサーバ上で生成することができます：src/test/regress/sql/hs_primary_extremes.sqlによって、スタンバイのテストのための挙動を実現できます。

今後のリリースでさらに自動化された試験ができるようになるかもしれません。

30.1.4. ロケールと符号化方式

デフォルトでは、一時的なインストレーションに対するテストは、現在の環境で定義されたロケールとinitdbで決定される対応するデータベース符号化方式を使用します。異なるロケールを試験する際は、以下の例のように適切な環境変数を設定することが有効です。

```
gmake check LANG=C
gmake check LC_COLLATE=en_US.utf8 LC_CTYPE=fr_CA.utf8
```

実装上の理由のため、LC_ALLはこの目的には動作しません。その他のロケール関連の環境変数は動作します。

既存のインストレーションに対するテストでは、ロケールは既存のデータベースクラスタによって決まり、テスト実行時に別の値に設定することができません。

また、以下の例のようにENCODING変数を設定することで明示的にデータベース符号化方式を選択することができます。

```
gmake check LANG=C ENCODING=EUC_JP
```

この方法でデータベース符号化方式を設定することは、通常ロケールがCだった場合にのみ意味があります。この他の場合、ロケールから自動的に符号化方式が選択されます。ロケールと一致しない符号化方式を指定してもエラーになるだけです。

符号化方式は一時的なインストレーションに対するテストでも既存のインストレーションに対するテストでも設定することができます。

30.1.5. 追加のテスト

プラットフォームに依存する、または非常に時間がかかる可能性があるという理由で、リグレッションテスト一式にはデフォルトでは動作しないテストがいくつか含まれています。 `EXTRA_TESTS` 変数を設定することでこれらの追加テストやその他のテストを実行することができます。例えば、`numeric_big` テストを以下のように実行します。

```
gmake check EXTRA_TESTS=numeric_big
```

以下のように照合順序テストを実行します。

```
gmake check EXTRA_TESTS=collate.linux.utf8 LANG=en_US.utf8
```

`collate.linux.utf8` テストは、Linux/glibcプラットフォームにおいてUTF-8符号化方式を使用するデータベースで実行した場合のみ動作します。

[前のページ](#)

リグレッションテスト

[ホーム](#)

[上に戻る](#)

[次のページ](#)

テストの評価

30.2. テストの評価

適正にインストールされ、かつ、すべての機能が使用できるようなPostgreSQLインストレーションであっても、浮動小数点の表現やメッセージ内の単語順など、プラットフォーム特有の誤差のために"失敗"することがあります。現在のテストは単純に、(基準となる)参照用システムで生成した出力とのdiffを取ることで結果の検証を行っているため、システムの些細な違いにも反応します。結果が"失敗"となった場合は、予測された結果と実際の結果との差分を必ず評価してください。それらの差異が重大ではないことが判明することもあります。なお、すべてのテストが成功するように、サポートするすべてのプラットフォームに対する正確な参照ファイルの保守に努めています。

実際のリグレッションテストの出力は、`src/test/regress/results`ディレクトリ内のファイルに書き込まれます。テストスクリプトはdiffを使用して、各出力ファイルと`src/test/regress/expected`ディレクトリ内の参照用出力とを比較します。あらゆる差異は調査用に`src/test/regress/regression.diffs`に保存されます。(あるいは、自分でdiffを実行することもできます。)

何らかの理由で、特定のプラットフォームが指定した試験で"失敗"し、その出力の調査により結果の方が有効であると確信できた場合、新しい比較用ファイルを追加し、今後の試験で失敗の報告が発生しないようにすることができます。詳細は[項30.3](#)を参照してください。

30.2.1. エラーメッセージの違い

リグレッションテストのいくつかは、意図的に無効な入力値を使用します。エラーメッセージはPostgreSQLのコードによるもの、または使用しているプラットフォームの関数によるものがあります。後者の場合、プラットフォームによって違いがあるかもしれませんが、似たような内容になるはずです。これらのメッセージの違いによりリグレッションテストは"失敗"する可能性があります、これらは検査で確認できます。

30.2.2. ロケールの違い

Cロケール以外の照合順序のロケールで初期化されたサーバに対してテストを実行する際には、ソート順やその後に発生する失敗に違いが生じる可能性があります。リグレッションテストスイートはこの問題を解決するために、多くのロケールを処理するための代替の結果ファイルを提供するように設定されています。

一時的なインストレーションを使用して、異なるロケールのテストを実行するためには、以下の例のように、`make`コマンドラインに適切なロケール関連の環境変数を渡してください。

```
gmake check LANG=de_DE.utf8
```

(リグレッションテストのドライバは`LC_ALL`を設定しないため、この変数を使ってロケールを選択することはできません。) ロケール無しを使用するためには、すべてのロケール関連の環境変数を設定しない(または、それらを`c`に設定する)か、もしくは以下の特殊な起動を行います。

```
gmake check NO_LOCALE=1
```

既存のインストレーションに対してテストを実行する場合は、ロケール設定は既存のインストレーションによって決まります。変更するためには、`initdb`に適切なオプションを渡して、異なるロケールでデータベースクラスタを初期化してください。

実際に実運用で使用するロケールおよび符号化方式に関連した部分のコードが検証されますので、一般的にはやはり、実運用で使用するロケール設定でリグレッションテストを実行することを推奨します。オペレーティングシステム環境に依存して、結果が失敗する場合もありますが、少なくとも実際のアプリケーションを実行する時に想定されるロケール固有の動作を知ることができます。

30.2.3. 日付と時刻の違い

日付と時刻の結果のほとんどはタイムゾーンの環境に依存します。参照ファイルはタイムゾーン `PST8PDT` (Berkeley, California) 用に生成されているため、このタイムゾーン設定で実行されていないテストは明らかに失敗します。リグレッションテストのドライバは、適切な結果を保証するために、環境変数`PGTZ`に`PST8PDT`を設定します。

30.2.4. 浮動小数点数の違い

いくつかのテストでは、64ビット(`double precision`型)の浮動小数点数値をテーブルの列から取り出して計算を行います。`double precision`列における数学演算関数では、異なった結果が発生する場合があります。これは知られています。`float8`と`geometry`テストは特に、プラットフォーム間、またはコンパイラの最適化の設定による小さな違いが起これやすくなります。これらの違い、通常は小数点以下10桁目以降の相違の、実際の影響度を判断するためには、人間の目で実際に確認する必

要があります。

いくつかのシステムではマイナス0を-0と表示することがあり、その他のシステムでは単に0と表示します。

いくつかのシステムでは、現在のPostgreSQLのコードが想定しているメカニズムと異なるために、`pow()` と `exp()` でエラーを発生する場合があります。

30.2.5. 行の順序の違い

同じ行の出力が、参照ファイルで記述されている順序とは異なっている場合があります。ほとんどの場合、これは厳密に言ってバグではありません。ほとんどのリグレーションテストは、各SELECT文に対してORDER BYを使用するほど規則に厳しくなく、そのため、結果の行の順序はSQLの仕様に従って、明確に決まっていません。実際には、同じソフトウェアで同じデータを用いて同じ問い合わせで参照しているので、すべてのプラットフォームで同じ順序の結果が返されるため、ORDER BYがないことは問題ではないと言えます。しかし、問い合わせによっては、プラットフォーム間の順序の違いが起こる可能性があります。インストール済みのサーバに対して試験を行う場合、C以外のロケール、独自のwork_memや独自のプランナ用のコストパラメータなどデフォルト以外のパラメータ設定により順序の違いが生じる可能性があります。

したがって、順序の違いを見つけた場合、問い合わせにORDER BYが含まれていて順序が影響を及ぼす場合以外は、気にする必要はありません。ただし、将来のリリースにおいて、偽の"失敗"を取り除くために特定の問い合わせにORDER BYを追加していた場合には、ご一報ください。

このような問題を避けるために、なぜ我々がすべてのリグレーションテストに対して明示的に順序を指定しないのか、疑問に思うかもしれません。その理由は、ソートが必要がない場合であってもソートされた結果を生成する問い合わせ計画を実行しようとすることによって、リグレーションテストの意義が増すわけではなく、むしろ減るからです。

30.2.6. スタック長の不足

errorsテストがselect infinite_recurse() コマンドでサーバをクラッシュさせた場合、プラットフォームのプロセススタックサイズがmax_stack_depth パラメータが示す値よりも小さいことを意味します。これは、スタックサイズ制限を高くして（デフォルトのmax_stack_depthでの推奨値は4メガバイト）サーバを実行することで修正することができます。これを行うことができない場合、max_stack_depthの値を少なくすることが代替方法です。

30.2.7. "乱数" テスト

randomテストスクリプトは、無作為な結果を生成することを目的としています。randomリグレーションテストは稀に失敗に終わることがあります。次のように、

```
diff results/random.out expected/random.out
```

と入力すると、ほんの数行だけの差異が生じるはずです。繰り返し失敗しない限り、気に留める必要はありません。

[前のページ](#)
テストの実行

[ホーム](#)
[上に戻る](#)

[次のページ](#)
各種の比較用ファイル

30.3. 各種の比較用ファイル

試験の中には必然的に環境に依存した結果となるものがありますので、"expected"結果ファイルの代替を指定する方法を用意しています。各リグレッションテストは、異なるプラットフォームで出力される可能性がある、複数の比較用ファイルを持つことができます。各試験に対してどの比較用ファイルを使用するかを決定する方法には、独立した2つの機構があります。

1つ目のメカニズムにより、特定のプラットフォームのための比較用ファイルを選ぶことができます。関連付けを行う `src/test/regress/resultmap` というファイルがあり、どの比較用ファイルがどのプラットフォームで使用されるのかを定義します。特定のプラットフォームにおいて試験の"失敗"の誤検知を防ぐためには、まず結果ファイルを選ぶ、あるいは結果ファイルを作成してから、`resultmap` ファイルに1行加えてください。

マッピングファイルの各行の書式は下記の通りです。

```
testname:output:platformpattern=comparisonfilename
```

`testname` は特定のリグレッションテストのモジュール名です。 `output` の値は、どの出力ファイルを検査するのかわを示します。標準のリグレッションテストでは、これは常に `out` です。この値は出力ファイルの拡張子に対応します。 `platformpattern` とは、`expr` Unix ツールスタイル（最初に暗黙的な `^` がある正規表現）のパターンです。これは `config.guess` によって出力されるプラットフォーム名と比較されます。 `comparisonfilename` は置き換える結果比較ファイルの（ディレクトリ部分を除いた）名前です。

以下に例を示します。システムの中には、非常に小さな浮動小数値に対し、アンダーフローエラーを報告せずに0として解釈するものがあります。これにより `horology` リグレッションテストにわずかな違いが発生します。そのため、`float8-small-is-zero.out` という異なる比較ファイルを用意し、そこにこういったシステムでの期待される値を記述します。OpenBSD プラットフォームにおいて偽の"失敗"メッセージ出力を行わせないようにするために、`resultmap` に以下を含めます。

```
float8:out:i.86-.*-openbsd=float8-small-is-zero.out
```

これは、`config.guess` の出力が `i.86-.*-openbsd` に一致するすべてのマシンに対して適用されます。 `resultmap` のその他の行は、他のプラットフォーム向けの適切な比較ファイルを選択します。

2つ目の比較用ファイルの選択の仕組みはかなり自動化されています。これは単純に、提供されている各種比較用ファイルの中から"もっとも一致するもの"を使用します。リグレッション試験のドライバスクリプトは、試験において、標準の比較用ファイル `testname.out` と `testname_digit.out`（ここで `digit` は0-9のいずれかからなる1つの数字です）という名前の別のファイルの両方を考慮します。もしこの中のいずれかのファイルが正確に一致した場合、試験が成功したものとみなします。さもなくば、生成されたdiffの結果がもっとも小さかった結果ファイルを選択して、失敗報告を生成します。（`resultmap` に特定の試験用の項目が含まれていると、`resultmap` 内の名前が元となる `testname` に置き換えられます。）

例えば、`char` の試験では、比較用ファイル `char.out` にはCロケールとPOSIXロケールで想定される結果が含まれています。一方、`char_1.out` ファイルには、他の多くのロケールで現れる結果がソートされて含まれています。

この最善一致の仕組みは、ロケールに依存した結果に対応できるように考え出されました。しかし、この仕組みはプラットフォームの名前だけでは簡単に予測できる試験結果とならないような、任意の状況で使うことができます。この仕組みの制約は、現在の環境でどの種類の比較ファイルが本当に"正しい"のかが試験ドライバでは分からないという点です。単にもっともうまく動いていそうなものを選択しているだけだからです。したがって、すべての文脈で平等に有効とみなすことができるような種類の結果においてのみ利用するのが、もっとも安全です。

30.4. テストが網羅する範囲の検証

PostgreSQLソースコードは、カバレッジテストツールとともにコンパイルすることができるため、リグレッションテスト、あるいはその他のテストスイートによって、コードのどの部分が網羅されているかを評価することができます。これは現在、GCCを使用してコンパイルした時にサポートされ、`gcov`および`lcov`プログラムを必要とします。

典型的な作業の流れは以下ようになります。

```
./configure --enable-coverage ... OTHER OPTIONS ...
gmake
gmake check # or other test suite
gmake coverage-html
```

そして、HTMLブラウザで`coverage/index.html`を参照します。 `gmake` コマンドはサブディレクトリ内においても同様に動作します。

複数回の試験を実行する時、実行回数をリセットするためには以下を実行します。

```
gmake coverage-clean
```

IV. クライアントインタフェース

ここではPostgreSQLに附属するクライアントプログラミングインタフェースについて説明します。各章は独立して読むことができます。クライアントプログラムには、この他にも様々なプログラミングインタフェースがありますが、これらのインタフェースは独自の資料とともに個別に配布されていますのでご注意ください（[付録H](#)に人気があるインタフェースの一部を列挙しています）。読者は、データベースの操作や問い合わせを行うためのSQLコマンド（[パートII](#)を参照）、また、当然ながら、インタフェースが使用するプログラミング言語にも慣れ親しんでいることが必要です。

目次

31章 [libpq - C ライブラリ](#)

- 31.1. [データベース接続制御関数](#)
- 31.2. [接続状態関数](#)
- 31.3. [コマンド実行関数](#)
- 31.4. [非同期コマンドの処理](#)
- 31.5. [1行1行問い合わせ結果を受け取る](#)
- 31.6. [処理中の問い合わせのキャンセル](#)
- 31.7. [近道インタフェース](#)
- 31.8. [非同期通知](#)
- 31.9. [COPY コマンド関連関数](#)
- 31.10. [制御関数](#)
- 31.11. [雑多な関数](#)
- 31.12. [警告処理](#)
- 31.13. [イベントシステム](#)
- 31.14. [環境変数](#)
- 31.15. [パスワードファイル](#)
- 31.16. [接続サービスファイル](#)
- 31.17. [接続パラメータのLDAP検索](#)
- 31.18. [SSLサポート](#)
- 31.19. [スレッド化プログラムの振舞い](#)
- 31.20. [libpqプログラムの構築](#)
- 31.21. [サンプルプログラム](#)

32章 [ラジオブジェクト](#)

- 32.1. [はじめに](#)
- 32.2. [実装機能](#)
- 32.3. [クライアントインタフェース](#)
- 32.4. [サーバ側の関数](#)
- 32.5. [サンプルプログラム](#)

33章 [ECPG - C言語による埋め込みSQL](#)

- 33.1. [概念](#)
- 33.2. [データベース接続の管理](#)
- 33.3. [SQLコマンドの実行](#)
- 33.4. [ホスト変数の使用](#)
- 33.5. [動的SQL](#)
- 33.6. [pgtypes ライブラリ](#)
- 33.7. [記述子領域の使用](#)
- 33.8. [エラー処理](#)
- 33.9. [プリプロセッサ指示子](#)
- 33.10. [埋め込みSQLプログラムの処理](#)
- 33.11. [ライブラリ関数](#)
- 33.12. [ラジオブジェクト](#)
- 33.13. [C++アプリケーション](#)
- 33.14. [埋め込みSQLコマンド](#)
- 33.15. [Informix互換モード](#)
- 33.16. [内部](#)

34章 [情報スキーマ](#)

- 34.1. [スキーマ](#)
- 34.2. [データ型](#)
- 34.3. [information_schema_catalog_name](#)
- 34.4. [administrable_role_authorizations](#)
- 34.5. [applicable_roles](#)
- 34.6. [attributes](#)
- 34.7. [character_sets](#)
- 34.8. [check_constraint_routine_usage](#)
- 34.9. [check_constraints](#)
- 34.10. [collations](#)
- 34.11. [collation_character_set_applicability](#)
- 34.12. [column_domain_usage](#)
- 34.13. [column_options](#)
- 34.14. [column_privileges](#)
- 34.15. [column_udt_usage](#)
- 34.16. [columns](#)
- 34.17. [constraint_column_usage](#)
- 34.18. [constraint_table_usage](#)
- 34.19. [data_type_privileges](#)
- 34.20. [domain_constraints](#)
- 34.21. [domain_udt_usage](#)
- 34.22. [domains](#)
- 34.23. [element_types](#)
- 34.24. [enabled_roles](#)
- 34.25. [foreign_data_wrapper_options](#)
- 34.26. [foreign_data_wrappers](#)
- 34.27. [foreign_server_options](#)
- 34.28. [foreign_servers](#)
- 34.29. [foreign_table_options](#)
- 34.30. [foreign_tables](#)
- 34.31. [key_column_usage](#)
- 34.32. [parameters](#)
- 34.33. [referential_constraints](#)
- 34.34. [role_column_grants](#)
- 34.35. [role_routine_grants](#)
- 34.36. [role_table_grants](#)
- 34.37. [role_udt_grants](#)
- 34.38. [role_usage_grants](#)
- 34.39. [routine_privileges](#)
- 34.40. [routines](#)
- 34.41. [schemata](#)
- 34.42. [sequences](#)
- 34.43. [sql_features](#)
- 34.44. [sql_implementation_info](#)
- 34.45. [sql_languages](#)
- 34.46. [sql_packages](#)
- 34.47. [sql_parts](#)
- 34.48. [sql_sizing](#)
- 34.49. [sql_sizing_profiles](#)
- 34.50. [table_constraints](#)
- 34.51. [table_privileges](#)
- 34.52. [tables](#)
- 34.53. [triggered_update_columns](#)
- 34.54. [triggers](#)
- 34.55. [udt_privileges](#)
- 34.56. [usage_privileges](#)
- 34.57. [user_defined_types](#)
- 34.58. [user_mapping_options](#)
- 34.59. [user_mappings](#)
- 34.60. [view_column_usage](#)
- 34.61. [view_routine_usage](#)
- 34.62. [view_table_usage](#)
- 34.63. [views](#)

第 31章 libpq - C ライブラリ

目次

- 31.1. [データベース接続制御関数](#)
 - 31.1.1. [接続文字列](#)
 - 31.1.2. [パラメータキーワード](#)
- 31.2. [接続状態関数](#)
- 31.3. [コマンド実行関数](#)
 - 31.3.1. [主要な関数](#)
 - 31.3.2. [問い合わせ結果の情報の取り出し](#)
 - 31.3.3. [他の結果情報の取り出し](#)
 - 31.3.4. [SQLコマンドに含めるための文字列のエスケープ処理](#)
- 31.4. [非同期コマンドの処理](#)
- 31.5. [1行1行問い合わせ結果を受け取る](#)
- 31.6. [処理中の問い合わせのキャンセル](#)
- 31.7. [近道インタフェース](#)
- 31.8. [非同期通知](#)
- 31.9. [COPYコマンド関連関数](#)
 - 31.9.1. [COPYデータ送信用関数](#)
 - 31.9.2. [COPYデータ受信用関数](#)
 - 31.9.3. [廃れたCOPY用関数](#)
- 31.10. [制御関数](#)
- 31.11. [雑多な関数](#)
- 31.12. [警告処理](#)
- 31.13. [イベントシステム](#)
 - 31.13.1. [イベントの種類](#)
 - 31.13.2. [イベントコールバックプロシージャ](#)
 - 31.13.3. [イベントサポート関数](#)
 - 31.13.4. [イベント事例](#)
- 31.14. [環境変数](#)
- 31.15. [パスワードファイル](#)
- 31.16. [接続サービスファイル](#)
- 31.17. [接続パラメータのLDAP検索](#)
- 31.18. [SSLサポート](#)
 - 31.18.1. [サーバ証明書のクライアント検証](#)
 - 31.18.2. [クライアント証明書](#)
 - 31.18.3. [異なるモードで提供される保護](#)
 - 31.18.4. [SSLクライアントファイル使用法](#)
 - 31.18.5. [SSLライブラリの初期化](#)
- 31.19. [スレッド化プログラムの振舞い](#)
- 31.20. [libpqプログラムの構築](#)
- 31.21. [サンプルプログラム](#)

libpqは、C言語によるアプリケーションプログラマ用のPostgreSQLインタフェースです。libpqは、クライアントプログラムからPostgreSQLのバックエンドサーバに問い合わせを渡し、その結果を受け取るためのライブラリ関数の集合です。

libpqは、C++、Perl、Python、Tcl、ECPGなどを含む、PostgreSQLの他の各種アプリケーションインタフェースを支えるエンジンでもあります。従って、libpqの動作は、これらのパッケージを使用する人にとって重要なものになります。特に、[項31.14](#)、[項31.15](#)および[項31.18](#)にて、libpqを使用するすべてのアプリケーションのユーザから見える動作を説明します。

本章の最後に、libpqの使い方を示す、いくつかの短いプログラム([項31.21](#))があります。また、ソースコード配布物内のsrc/test/examplesディレクトリに、libpqを利用したアプリケーションプログラム一式の例があります。

libpqを使用してフロントエンドプログラムを作成するには、libpq-fe.hヘッダファイルのインクルードと、libpq ライブラリとのリンクが必要です。

[前のページ](#)
クライアントインタフェース

[ホーム](#)
[上に戻る](#)

[次のページ](#)
データベース接続制御関数

31.1. データベース接続制御関数

PostgreSQLのバックエンドサーバとの接続を作成するには、以下の関数を使用します。アプリケーションプログラムはバックエンドとの接続を一度に複数個開くことができます。（1つの理由として、複数のデータベースへのアクセスが挙げられます。）個々の接続は、`PQconnectdb`、`PQconnectdbParams`または`PQsetdbLogin`関数を呼び出すことで得られる`PGconn`オブジェクトによって表されます。なお、これらの関数は、`PGconn`オブジェクトに割り当てるほんのわずかなメモリの余裕さえもない場合を除き、NULLではなく常にオブジェクトのポインタを返します。また、この接続オブジェクトを通じて問い合わせを送る前に、`PQstatus`関数を呼び出して、データベースとの接続に成功したか戻り値を検査しなければなりません。

警告

Unix上で、libpq接続を開いたプロセスのフォークは、親と子プロセスが同じソケットとオペレーティングシステムの資源を共有するため、予期せぬ結果を招くことがあります。この理由により、新規実行形式を子プロセスが読み込むため`exec`を行うことが安全と言っても、このような使用法は推奨されません。

注意: Windowsでは、単一のデータベース接続が反復して開始と終了を繰り返す場合、性能を向上させる方法があります。内部的には、接続開始と終了に対して、libpqはそれぞれ`WSAStartup()`と`WSACleanup()`を呼び出します。`WSAStartup()`は`WSACleanup()`で値が減少させられた内部Windowsライブラリ参照カウントを増加させます。参照カウントがたった1の場合、`WSACleanup()`呼び出しはすべてのリソースを解放し、すべてのDLLはアンロードされます。これは高価な操作です。これを回避するには、最後のデータベース接続が閉じられる時、リソースが解放されないようにアプリケーションが手動で`WSAStartup()`を呼び出すことができます。

`PQconnectdbParams`

新たにデータベースサーバへの接続を作成します。

```
PGconn *PQconnectdbParams(const char * const *keywords,
                           const char * const *values,
                           int expand_dbname);
```

この関数は、2つのNULL終端の配列から取得したパラメータを使用して、データベースとの接続を新たに1つ確立します。1つ目は文字列配列として定義される`keywords`で、それぞれがキーワードとなります。2つ目は`values`で、各キーワードの値を提供します。後述の`PQsetdbLogin`とは異なり、関数のシグネチャを変更せずにパラメータ集合を拡張できますので、アプリケーションプログラムを新たに作成する際には、この関数(もしくは非ブロックモードでよく似た処理をする`PQconnectStartParams`と`PQconnectPoll`)を使用することをお勧めします。

現在有効なパラメータキーワードを[項31.1.2](#)に示します。

`expand_dbname`が非ゼロの場合、`dbname`キーワードの値を接続文字列として認識させることができます。取り得る書式に関する詳細については[項31.1.1](#)を参照してください。

空の配列を渡してすべてデフォルトパラメータを使用することができます。また渡される配列に1つ以上のパラメータ設定を持たせることもできます。これらの長さは一致しなければなりません。`keywords`配列の最後の非NULL要素で処理は停止します。

パラメータが指定されなかった場合には、対応する環境変数が検査されます([項31.14](#)を参照してください)。環境変数も設定されていない場合は、指定された組み込みのデフォルト値が使用されます。

一般的にキーワードはこれらの配列の先頭からインデックス順で処理されます。この影響はキーワードが繰り返された場合で、最後に処理された値が残ることになります。このため、`dbname`キーワードの記述場所に注意することで、`conninfo`文字列により何が上書きされるか、何が上書きされないかを決定することができます。

`PQconnectdb`

新たにデータベースサーバへの接続を作成します。

```
PGconn *PQconnectdb(const char *conninfo);
```

この関数は`conninfo`文字列から取得されるパラメータを使用して、新しいデータベース接続を開きます。

空の文字列を渡してすべてデフォルトパラメータを使用することができます。また空白文字で区切ることで1つ以上のパラメータ設定を持たせることもできます。さらにURIを含めることができます。詳細については[項31.1.1](#)を参照してください。

`PQsetdbLogin`

新たにデータベースサーバへの接続を作成します。

```
PGconn *PQsetdbLogin(const char *pghost,
                    const char *pgport,
                    const char *pgoptions,
                    const char *pgtty,
                    const char *dbName,
                    const char *login,
                    const char *pwd);
```

これはパラメータ群を固定したPQconnectdbの前身です。設定できないパラメータが常にデフォルト値になる点を除き、同一の機能を持ちます。固定のパラメータに対してNULLもしくは空文字列とすると、それはデフォルトを使用することになります。

dbName内に=記号が含まれる場合、または有効な接続URI接頭辞を持つ場合、PQconnectdbに渡された場合とまったく同じ扱いでconninfo文字列として扱われます。その後残りのパラメータがPQconnectdbParamsの指定のように適用されます。

PQsetdb

新たにデータベースサーバへの接続を作成します。

```
PGconn *PQsetdb(char *pghost,
               char *pgport,
               char *pgoptions,
               char *pgtty,
               char *dbName);
```

これは、loginとpwdにNULLポインタを設定するPQsetdbLoginを呼び出すマクロです。非常に古いプログラムへの後方互換性のために提供されています。

PQconnectStartParams
PQconnectStart
PQconnectPoll

ブロックしない方法で、データベースサーバへの接続を作成します。

```
PGconn *PQconnectStartParams(const char * const *keywords,
                             const char * const *values,
                             int expand_dbname);

PGconn *PQconnectStart(const char *conninfo);

PostgresPollingStatusType PQconnectPoll(PGconn *conn);
```

これら3つの関数を使用して、リモートI/Oの実行時にアプリケーションスレッドの実行がブロックされないように、データベースサーバへの接続を作成します。この手法の利点は、I/Oの終了待ちがPQconnectdbParamsまたはPQconnectdb内部ではなく、アプリケーションプログラムのメインループでできることにあります。これによって、アプリケーションは他の処理と並行してこの処理を管理することができます。

PQconnectStartParamsでは、上でPQconnectdbParamsで説明したように、データベース接続はkeywordsおよびvalues配列から取得され、expand_dbnameによって制御されたパラメータを使用して確立します。

PQconnectStartでは、上でPQconnectdbで説明したように、conninfo文字列から取得されたパラメータを使用してデータベース接続を確立します。

PQconnectStartParams、PQconnectStartとPQconnectPollのどちらも以下の制限に適合する場合ブロックしません。

- hostaddrとhostパラメータは、ホスト名からのIPアドレス検索やホスト名の逆引きが起こらないように適切に使用されなければいけません。詳細に付いては[項31.1.2](#)内のパラメータ説明を参照してください。
- PQtraceを呼び出す場合は、トレースに使用するストリームオブジェクトがブロックされないことが保証されていなくてはなりません。
- プログラム自身が、後に示すように、PQconnectPollを呼び出す前にソケットが適切な状態にあることを保証しなくてはなりません。

注意：PQconnectStartParamsの使用は後述のPQconnectStartと類似しています。

非ブロック接続要求を始めるにはまず、conn=PQconnectStart("connection_info_string")を呼び出します。connがNULLの場合、libpqが新たにPGconn構造体を割り当てられなかったことを表します。そうでない場合は、適切なPGconnへのポインタが返されます（ただし、データベースに正しく接続されていることを表しているわけではありません）。PQconnectStartから値が返ってきた段階で、status=PQstatus(conn)を呼び出します。もし、statusがCONNECTION_BADと等しい場合には、PQconnectStartが失敗しています。

PQconnectStartが成功したら、次は接続シーケンスを進めるために、libpqをポーリングします。データベース接続の背後にあるソケットの記述子を取り出すには、PQsocket(conn)を使用します。以下の繰り返しです。直前のPQconnectPoll(conn)がPGRES_POLLING_READINGの場合、ソケットの読み込み準備が整うまで待機します。(select()やpoll()などのシステム関数で示されます。)そして、再度PQconnectPoll(conn)を呼び出します。反対に直前のPQconnectPoll(conn)がPGRES_POLLING_WRITINGの場合、ソケットの書き込み準備が整うまで待機し、その後、PQconnectPoll(conn)を再度呼び出します。まだPQconnectPollを呼び出していない場合、つまり、PQconnectStartの呼び出し直後では、直前がPGRES_POLLING_WRITINGであった場合と同様の処理を行いません。この繰り返しをPQconnectPoll(conn)が、接続手続きの失敗を示すPGRES_POLLING_FAILED、もしくは、接続確立に成功したことを示すPGRES_POLLING_OKを返すまで継続します。

接続している間は、いつでもPQstatusを呼び出すことで、接続の状態を検査することができます。この関数呼び出しがCONNECTION_BADを返す場合、接続手続きは失敗しており、CONNECTION_OKを返す場合、接続が確立しています。上述のように、このいずれの状態も、PQconnectPollの戻り値から同様に検出できます。これ以外の状態は、非同期の接続手続きの間（のみに）現れることがあります。これらは、接続手続きの現在の段階を示すものであり、例えばユーザへのフィードバックを提供することに使用できます。以下の状態があります。

CONNECTION_STARTED

接続の確立待ち状態です。

CONNECTION_MADE

接続はOKです。送信待ち状態です。

CONNECTION_AWAITING_RESPONSE

サーバからの応答待ち状態です。

CONNECTION_AUTH_OK

認証済みです。バックエンドの起動待ち状態です。

CONNECTION_SSL_STARTUP

SSL暗号化の調停状態です。

CONNECTION_SETENV

環境が提供するパラメータ設定の調停状態です。

これらの定数は（互換性を保つため）なくなることはありませんが、アプリケーションは、これらが特定の順で出現したり、本書に書いてある値のどれかに必ずステータス値が該当するということを決して当てに はいけません。アプリケーションは、以下に示すようにする必要があります。

```
switch(PQstatus(conn))
{
    case CONNECTION_STARTED:
        feedback = "Connecting...";
        break;

    case CONNECTION_MADE:
        feedback = "Connected to server...";
        break;

    .
    .
    .
    default:
        feedback = "Connecting...";
}
```

PQconnectPollを使用する場合、connect_timeout接続パラメータは無視されます。経過時間が長過ぎるかどうかの判定はアプリケーションの責任で行ないます。さもないと、PQconnectStartの後のPQconnectPollの繰り返しがPQconnectdbと同じになります。

PQconnectStartが非NULLポインタを返した場合、処理を終了する際には、構造体や関連するメモリブロックを始末するために、PQfinishを呼び出さなくてはならないことに注意してください。この処理は、接続試行が失敗した場合やその試行を中断する場合にも、必ず実行されなければいけません。

PQconndefaults

デフォルトの接続オプションを返します。

```
PQconninfoOption *PQconndefaults(void);

typedef struct
{
```



```

char    *keyword;    /* このオプションのキーワード */
char    *envvar;     /* 代替となる環境変数の名前 */
char    *compiled;   /* 代替となるコンパイル時に組み込まれたデフォルト値 */
char    *val;        /* オプションの現在値、もしくは、NULL */
char    *label;      /* 接続ダイアログ内の当該フィールドのラベル */
char    *dispchar;   /* 接続ダイアログ内の当該フィールドをどのように表示するかの指示
                        値:
                        ""      入力された値をそのまま表示
                        "*"     値を隠すパスワードフィールド用
                        "D"     デバッグオプション。デフォルトで何も表示しません */
int      dispsize;   /* ダイアログ用のフィールドの大きさ (文字数単位) */
} PQconninfoOption;

```

接続オプションの配列を返します。これは、使用可能なPQconnectdb用オプションのすべてや、その時点でのデフォルト値を決定するために使用することができます。戻り値は、PQconninfoOption構造体の配列へのポインタで、keywordポインタがヌルとなる項目が配列の末尾にきます。メモリが確保できなかった場合にはヌルポインタを返します。現在のデフォルト値(val フィールド)は、環境変数や他のコンテキストに依存します。呼び出し側では、接続オプションの情報は、読み込み専用として取り扱わなければいけません。

オプションの配列を処理した後は、PQconninfoFreeを使用して解放します。この処理をしないと、PQconndefaultsが呼び出されるたびに少しずつメモリリークが発生します。

PQconninfoParse

提供された接続文字列から構文解析された接続オプションを返します。

```
PQconninfoOption *PQconninfoParse(const char *conninfo, char **errmsg);
```

接続文字列の構文解析を行い、配列として結果オプションを返すか、または接続文字列に問題があった場合にNULLを返します。この関数を提供された接続文字列の中のPQconnectdbオプションを取り出すために使用することができます。戻り値はPQconninfoOption構造体の配列を指し示し、それはヌルのkeywordポインタを持つ項目で終結します。

正規なオプションはすべて、結果配列内に現れます。しかし接続文字列内に現れない、何らかのオプション用のPQconninfoOptionはNULLに設定されたvalを持ちます。デフォルトは挿入されません。

errmsgが非NULLであれば、成功した場合*errmsgはNULLに設定され、そうでなければ、問題を説明したmallocされたエラー文字列になります。（*errmsgがNULLに設定され、かつ、この関数がNULLを返すこともあり得ます。これはメモリ不足状態を意味します。）

オプション配列を処理した後、それをPQconninfoFreeに渡して解放してください。これが行われない場合、PQconninfoParseへのそれぞれの呼び出しに対してメモリーリークが起こります。反対に、エラーが起こり、そしてerrmsgが非NULLであれば、PQfreememを使用してエラー文字列を必ず解放してください。

PQfinish

サーバとの接続を閉じます。また、PGconnオブジェクトが占めるメモリも解放します。

```
void PQfinish(PGconn *conn);
```

たとえサーバへの接続試行が失敗しても（PQstatusで調べます）、アプリケーションはPQfinishを呼び出しPGconnオブジェクトが占めるメモリを解放するべきです。そしてPQfinishを呼び出したら、もうPGconnへのポインタを使ってはいけません。

PQreset

サーバへの通信チャンネルをリセットします。

```
void PQreset(PGconn *conn);
```

この関数はサーバへの接続を閉じ、以前使用したパラメータをすべて使用して、同一のサーバへ新しく接続を確立します。これは、作業中の接続が失われた場合のエラーの修復に役立つでしょう。

PQresetStart PQresetPoll

非ブロッキング方式で、サーバへの接続チャンネルをリセットします。

```
int PQresetStart(PGconn *conn);

PostgresPollingStatusType PQresetPoll(PGconn *conn);
```

これらの関数はサーバへの接続を閉じ、それから再度、以前使用したパラメータをすべて使用して、同じサーバと新たな接続を確立しようとします。これらは作業中の接続が失われた場合のエラー修復に役立つでしょう。PQreset（前述）との違いは、この2つの関数が非ブロック方式で動作することです。また、これらの関数はPQconnectStartParams、PQconnectStartおよびPQconnectPollと同じ制限を受けます。

接続のリセットを始めるためには `PQresetStart` を呼び出します。この関数がゼロを返す場合、リセットに失敗しています。戻り値が1ならば、`PQconnectPoll` を使って接続を確立した時とまったく同じに、`PQresetPoll` を使用してリセットのポーリングを行います。

`PQpingParams`

`PQpingParams` はサーバの状態を報告します。この関数は上述の `PQconnectdbParams` と同じ接続パラメータを受け付けます。しかし、サーバの状態を得るために正しいユーザ名、パスワード、データベース名を提供する必要はありません。

```
PGPing PQpingParams(const char * const *keywords,
                    const char * const *values,
                    int expand_dbname);
```

このサーバは以下の値のいずれかを返します。

`PQPING_OK`

サーバは稼動中で、接続を受け付けているようです。

`PQPING_REJECT`

サーバは稼動中ですが、接続を許可しない状態（起動処理中、停止処理中、クラッシュリカバリ中）です。

`PQPING_NO_RESPONSE`

サーバと通信できません。これは、サーバが稼動中ではない、指定した接続パラメータの何か（例えばポート番号の間違い）が間違っている、ネットワーク接続性の問題（例えば接続要求をブロックするファイアウォール）があることを示しているかもしれません。

`PQPING_NO_ATTEMPT`

指定されたパラメータが明らかに間違っている、または、（メモリ不足など）クライアント側の問題があったため、サーバとの通信を試行しませんでした。

`PQping`

`PQping` はサーバの状態を報告します。この関数は上述の `PQconnectdb` と同じ接続パラメータを受け付けます。しかし、サーバの状態を得るために正しいユーザ名、パスワード、データベース名を提供する必要はありません。

```
PGPing PQping(const char *conninfo);
```

戻り値は `PQpingParams` と同じです。

31.1.1.1. 接続文字列

複数の `libpq` 関数は、接続パラメータを得るためにユーザが指定した文字列の解析を行います。この文字列として、普通の `keyword = value` 文字列と [RFC 3986](#) の URI という 2 種類の書式が受け付けられます。

31.1.1.1.1. キーワード/値形式の接続文字列

最初の書式では、各パラメータ設定は `keyword = value` という形式です。等号記号の前後の空白文字は省略可能です。空の値を書く、または空白文字を含む値を書くためには、`keyword = 'a value'` のように単一引用符で値を括ります。値内部の単一引用符とバックスラッシュはバックスラッシュでエスケープしなければなりません。つまり `¥'` と `¥¥` です。

以下に例を示します。

```
host=localhost port=5432 dbname=mydb connect_timeout=10
```

有効なパラメータキーワードを [項31.1.2](#) に示します。

31.1.1.1.2. 接続URI

接続URIの一般的な形式を以下に示します。

```
postgresql://[user[:password]@][netloc][:port][/dbname][?param1=value1&...]
```

URIスキーマ指示子は `postgresql://` または `postgres://` のいずれかを取ることができます。個々のURI部品は省略可能です。以下の例で有効なURI構文の使用例を示します。

```
postgresql://
postgresql://localhost
postgresql://localhost:5433
postgresql://localhost/mydb
```

```
postgresql://user@localhost
postgresql://user:secret@localhost
postgresql://other@localhost/otherdb?connect_timeout=10&application_name=myapp
```

URIの階層部品の要素をパラメータとして与えることができます。以下に例を示します。

```
postgresql:///mydb?host=localhost&port=5433
```

パーセント符号化を使用して、URI部品のいずれかに特殊な意味を持つ記号を含めることができます。

[項31.1.2](#)に示されたキーワードに対応しない接続パラメータは無視され、これに関する警告メッセージが`stderr`に書き出されます。

JDBCの接続URI構文との互換性を高めるために、`ssl=true`パラメータインスタンスは`sslmode=require`に変換されます。

ホスト部分にはホスト名またはIPアドレスを書くことができます。IPv6ホストアドレスを指定するためには角括弧で括弧します。

```
postgresql://[2001:db8::1234]/database
```

ホスト要素は`host`で説明したように解釈されます。具体的には、ホスト部品が空またはスラッシュで始まる場合Unixドメインソケット接続が選択され、さもなければTCP/IP接続で初期化されます。しかしURIの階層部ではスラッシュが予約された文字であることに注意してください。このため、標準以外のUnixドメインソケットディレクトリを指定するためには、URIからホスト指定を省き、パラメータとしてホストを指定するか、URIのホスト要素内のパスをパーセントエスケープするかどちらかを行ってください。

```
postgresql:///dbname?host=/var/lib/postgresql
postgresql://%2Fvar%2Flib%2Fpostgresql/dbname
```

31.1.2. パラメータキーワード

現時点で有効なパラメータのキーワードは以下に示す通りです。

`host`

接続するホスト名を指定します。この引数をスラッシュで始めた場合、TCP/IPによる通信ではなく、Unixドメインの通信を明示することになります。その場合、この値はソケットファイルを格納するディレクトリの名前になります。`host`が指定されなかった場合のデフォルトは、`/tmp`にあるUnixドメインのソケットに接続することです。（または、PostgreSQLの構築時に指定した別のディレクトリにあるソケットです。）Unixドメインソケットを持たないマシンにおけるデフォルトは、`localhost`に接続することです。

`hostaddr`

接続するホストのIPアドレスを指定します。これは、`172.28.40.9`といった標準的なIPv4アドレス書式でなければなりません。使用するマシンでIPv6をサポートする場合は、そのアドレスを使用することもできます。このパラメータに空以外の文字列が指定されると、TCP/IP通信が常に使用されます。

`host`の代わりに`hostaddr`を使用することで、アプリケーションがホスト名の検索を行わずに済みます。特に時間的制約があるアプリケーションでは重要になるでしょう。しかし、Kerberos、GSSAP、SSPI認証方式では、ホスト名が必要になります。`verify-full`SSL証明書検証を行う場合も同様です。以下の規則が使用されます。

- `hostaddr`を使わずに`host`を指定した場合は、ホスト名の検索が発生します。
- `host`を使わずに`hostaddr`を指定した場合、`hostaddr`の値はサーバのネットワークアドレスとなります。認証方式がホスト名を必要とする場合は接続試行が失敗します。
- `host`と`hostaddr`の両方を指定した場合、`hostaddr`がサーバのネットワークアドレスとなります。`host`の値は認証方式で必要とされない限り無視され、必要とされる場合にはホスト名として使用されます。

`host`が`hostaddr`ネットワークアドレスに対応するマシンの名前と一致しない場合は、認証に失敗する可能性があるので注意してください。また、`hostaddr`ではなく`host`が`~/.pgpass` ([項31.15](#)を参照)での接続の識別に使用されます。

ホスト名もホストのアドレスも用いない場合、`libpq`はローカルUnixドメインソケットを使用して接続します。ただし、Unixドメインソケットを持たないマシンでは、`localhost`への接続を試みます。

`port`

サーバホストでの接続用のポート番号、または、Unixドメイン接続の場合は、ソケットファイルの拡張子を指定します。

`dbname`

データベース名を指定します。デフォルトはユーザ名と同じです。特定の文脈では、この値は拡張書式で検査されま

す。 詳細については[項31.1.1](#)を参照してください。

user

データベースへ接続するPostgreSQLユーザ名を指定します。 デフォルトは、そのアプリケーションを実行しているユーザのオペレーティングシステム上の名前と同じです。

password

サーバがパスワードによる認証を必要とした場合に使用されるパスワードを指定します。

connect_timeout

接続用の最大待機時間を秒単位(10進数整数で表した文字列として記述してください)で指定します。 ゼロもしくは未設定は、無限時間の待機を意味します。 2秒未満の待機時間を使用することは勧めません。

client_encoding

接続用の`client_encoding`設定パラメータを設定します。 対応するサーバオプションで受け付けられる値の他に、クライアントにおける現在のロケール (Unixシステムの場合は`LC_CTYPE`環境変数) から正しい符号化方式を決定する`auto`を使用することができます。

options

実行時にサーバに送信するコマンドラインオプションを追加します。 例えば、これを`-c geqo=off`に設定すると、`geqo`パラメータのセッション値は`off`になります。 有効なオプションに関する詳細については[第18章](#)を参照してください。

application_name

[application_name](#)設定パラメータの値を指定します。

fallback_application_name

[application_name](#)設定パラメータの予備値を指定します。 接続パラメータまたは`PGAPPNAME`環境変数により`application_name`の値が指定されない場合に、この値が使用されます。 予備の名前を指定することは、デフォルトのアプリケーション名を設定したいが、ユーザにもそれを上書きできるようにしておきたい、一般的なユーティリティプログラムで有用です。

keepalives

クライアント側におけるTCPキープアライブの使用を制御します。 デフォルト値は1であり、有効であることを意味します。 しかしキープアライブを望まない場合は、無効であることを意味するゼロに設定することができます。 このパラメータはUnixドメインソケット経由の接続では無視されます。

keepalives_idle

TCPがサーバにキープアライブメッセージを送信した後に活動を行わない期間を秒単位で制御します。 ゼロという値ではシステムのデフォルトを使用します。 Unixドメインソケット経由でなされた接続の場合もしくはキープアライブが無効な場合、このパラメータは無視されます。 これは`TCP_KEEPIDL`ソケットオプションまたは`TCP_KEEPA`ソケットオプションが利用できるシステムおよびWindowsでのみサポートされます。 他のシステムでは効果がありません。

keepalives_interval

TCPキープアライブメッセージに対する応答がサーバからない場合に、何秒後に再送を行うかを制御します。 ゼロという値ではシステムのデフォルトを使用します。 Unixドメインソケット経由でなされた接続の場合、またはキープアライブを無効にしている場合、このパラメータは無視されます。 これは`TCP_KEEPI`ソケットオプションが利用できるシステムおよびWindowsでのみサポートされます。 他のシステムでは効果がありません。

keepalives_count

サーバへのクライアント接続が不要になったとみなすまで、何回キープアライブの欠落を認めるかを制御します。 ゼロという値ではシステムのデフォルトを使用します。 Unixドメインソケット経由でなされた接続の場合、またはキープアライブを無効にしている場合、このパラメータは無視されます。 これは`TCP_KEEPC`ソケットオプションが利用できるシステムでのみサポートされます。 他のシステムでは効果がありません。

tty

無視されます(以前は、これはサーバデバッグ出力を送信する場所を指定するものでした)。

sslmode

このオプションは、どのSSLによる安全なTCP/IP接続の優先度でサーバと調停するかを決定します。 6つのモードがあります。

disable

非SSL接続のみ試行

allow

最初に非SSL接続を試行し、失敗したら、SSL接続を試行

prefer (デフォルト)

最初にSSL接続を試行し、失敗したら、非SSL接続を試行

require

SSL接続のみ試行。ルートCAファイルが存在する場合、`verify-ca`が指定された場合と同じ方法で証明書が検証されます。

verify-ca

SSL接続のみ試行し、サーバ証明書が信用された認証局 (CA) から発行されたかを検証

verify-full

SSL接続のみ試行し、サーバ証明書が信用されたCAから発行されたか、およびそのサーバホスト名が証明書内のものと一致するかを検証

これらのオプションがどのように動くのかについては[項31.18](#)を参照してください。

`sslmode`はUnix ドメインソケット通信では無視されます。 SSLサポートなしでPostgreSQLがコンパイルされた場合に、`require`、`verify-ca`、`verify-full`を使用するとエラーになります。 一方、`allow`と`prefer`は使用できますが、実際にlibpqはSSL接続を受け付けません。

`requiressl`

このオプションは`sslmode`設定を支持する観点から廃止予定になっています。

1に設定することで、サーバへのSSL接続が必要になります (これは`sslmode`の`require`と同じです)。サーバがSSL接続を受け付けない場合、libpqは接続を拒絶します。 0(デフォルト)に設定することで、サーバと接続形式の調停を行います。(`sslmode`の`prefer`と同じです。) SSLサポート付きでPostgreSQLをコンパイルした場合にのみ、このオプションが利用できます。

`sslcompression`

1 (デフォルト) に設定することで、SSL接続越えて送信されるデータは圧縮されます (これにはOpenSSLバージョン0.9.8以降が必要です)。 0に設定すると、圧縮が無効になります (これにはOpenSSL1.0.0以降が必要です)。 このパラメータはSSLが確立していない接続や使用されるOpenSSLがサポートしていない場合は無視されます。

圧縮はCPU処理時間を使用しますが、ネットワークが問題である場合はスループットを改良することができます。 CPU性能が制約要因であれば、圧縮を無効にすることで、応答時間やスループットを改良することができます。

`sslcert`

このパラメータは、`~/.postgresql/postgresql.crt`というデフォルトを置き換えるクライアントSSL証明書のファイル名を指定します。 このパラメータはSSL接続が確立していない場合は無視されます。

`sslkey`

このパラメータはクライアント証明書に対して使用される秘密鍵の場所を指定します。 デフォルトの`~/.postgresql/postgresql.key`の代わりに使用されるファイル名、または外部"エンジン" (エンジンとはOpenSSLロード可能なモジュール) から得られるキーを指定することも可能です。 外部エンジンの指定にはコロンで区切ったエンジン名とエンジン特有の鍵識別子を含んでいなければなりません。 SSL接続が確立していない場合このパラメータは無視されます。

`sslrootcert`

このパラメータはSSL認証局(CA)の証明書のファイル名を指定します。 このファイルが存在する場合、サーバ証明書はこれらの認証局の1つで署名されているかどうか検証されます。 デフォルトは`~/.postgresql/root.crt`です。

`sslcr1`

このパラメータはSSL証明書失効リスト (CRL) のファイル名を指定します。 このファイルに列挙された証明書が存在した場合、それはサーバ証明書を承認しようとする時に拒絶されます。 デフォルトは`~/.postgresql/root.crl`です。

`requirepeer`

このパラメータは、例えば`requirepeer=postgres`のようにサーバのオペレーティングシステムのユーザ名を指定します。Unixドメインソケット接続を確立する時に、このパラメータが設定された場合、クライアントは接続開始時にサーバプロセスが指定されたユーザ名で稼働しているか検査し、稼働していない場合は接続をエラーとして中断します。このパラメータは、TCP/IP接続においてSSL証明書で実現するようなサーバ認証を実現するために使用することができます。
(Unixドメインソケットが`/tmp`などの誰にでも書き込むことができる場所にある場合、誰でもそこで接続を監視するサーバを起動できることに注意してください。信頼できるユーザが起動したサーバに接続することを確実に行うために、このパラメータを使用してください。) このオプションは`peer`認証方式が実装されたプラットフォームでのみでサポートされます。 [項19.3.7](#)を参照してください。

`krbsrvname`

Kerberos5またはGSSAPIの認証時に使われるKerberosサービス名です。 成功するためには、これはサーバのKerberos認証設定のサービス名と一致していなければなりません。 ([項19.3.5](#)および[項19.3.3](#)も参照してください。)

`gsslib`

GSSAPI認証で使用するGSSライブラリです。 Windows上のみで使用されます。 `libpq`の認証がデフォルトのSSPIではなく、強制的にGSSAPIライブラリを使用させるには`gssapi`を設定してください。

`service`

追加のパラメータ用に使用されるサービス名です。 `pg_service.conf`内の追加的な接続パラメータを保持するサービス名を指定します。 これによりアプリケーションはサービス名だけを指定でき、接続パラメータを集中的に保守できるようになります。 [項31.16](#)を参照してください。

[前のページ](#)

libpq - C ライブラリ

[ホーム](#)
[上に戻る](#)

[次のページ](#)
接続状態関数

31.2. 接続状態関数

これらの関数を使用して、既存のデータベース接続オブジェクトの状態を調べることができます。

タイプ: libpqアプリケーションのプログラマは注意してPGconnという抽象化を維持してください。PGconnの内容は以下に挙げるアクセス用関数を使って取り出してください。PGconn構造体中のフィールドは将来予告なく変更されることがありますので、libpq-int.hを使用したフィールドの参照は避けてください。

以下の関数は、接続で確立したパラメータの値を返します。これらの値はPGconnの存続期間中で固定されます。

PQdb

接続したデータベース名を返します。

char *PQdb(const PGconn *conn);

PQuser

接続したユーザ名を返します。

char *PQuser(const PGconn *conn);

PQpass

接続したパスワードを返します。

char *PQpass(const PGconn *conn);

PQhost

接続したサーバホスト名を返します。

char *PQhost(const PGconn *conn);

PQport

接続したポートを返します。

char *PQport(const PGconn *conn);

PQtty

接続のデバッグ用TTYを返します。(これは廃れたものです。サーバはもはやTTY設定を参照しません。後方互換性のためにこの関数が残っています。)

char *PQtty(const PGconn *conn);

PQoptions

接続要求時に渡されたコマンドラインオプションを返します。

char *PQoptions(const PGconn *conn);

以下の関数は、PGconnオブジェクトに対して操作を行うことで変更可能な状態データを返します。

PQstatus

接続の状態を返します。

ConnStatusType PQstatus(const PGconn *conn);

この状態は多くの値の中の1つとなるはずですが、しかし非同期接続手順の外部からは、その中でたった2つ、CONNECTION_OKとCONNECTION_BADだけが現れます。データベースへの接続に問題がなければ、CONNECTION_OK状態になります。接続に失敗している場合はCONNECTION_BAD状態となります。通常、OK状態はPQfinishまで維持されますが、通信失敗のために早まってCONNECTION_BADになることもあります。その場合、アプリケーションはPQresetを呼び出して修復を試みることができます。

返される可能性があるその他の状態コードについてはPQconnectStartParams、PQconnectStartおよびPQconnectPollの項目を参照してください。

PQtransactionStatus

サーバの現在のトランザクション内部状態を返します。

PGTransactionStatusType PQtransactionStatus(const PGconn *conn);

この状態は、PQTRANS_IDLE (現在待機中)、PQTRANS_ACTIVE (コマンド実行中)、PQTRANS_INTRANS (有効なトランザクションブロック内で待機中)、PQTRANS_INERROR (無効なトランザクションブロック内で待機中)となり得ます。接続に問題がある場合のみPQTRANS_UNKNOWNが報告されます。サーバへ問い合わせが送信されたが、まだ完了していない場合のみPQTRANS_ACTIVEが報告されます。

注意
autocommitパラメータを無効にしたPostgreSQL 7.3サーバを使用する場合、PQtransactionStatusは間違った結果を返します。サーバサイドでの自動コミットは廃止され、その後のバージョンのサーバでは存在しません。

PQparameterStatus

サーバの現在のパラメータ設定を検索します。

const char *PQparameterStatus(const PGconn *conn, const char *paramName);

あるパラメータ値は、接続開始時にサーバによって自動的に、もしくは、その値が変更された時は常に報告されます。PQparameterStatusはそれらの設定の調査に役立ちます。パラメータの現在値がわかればその値を、わからない場合はNULLを返します。

現在のリリースで報告されるパラメータには、server_version、server_encoding、client_encoding、application_name、is_superuser、session_authorization、DateStyle、IntervalStyle、TimeZone、integer_datetimesおよびstandard_conforming_stringsがあります。(8.0より前ではserver_encoding、TimeZoneおよびinteger_datetimesが、8.1より前ではstandard_conforming_stringsが、そして8.4より前ではIntervalStyleが、9.0より前ではapplication_nameが報告されませんでした。) server_version、server_encodingおよびinteger_datetimesは起動後変更することができないことに注意してください。

プロトコル3.0より前のサーバはパラメータ設定を報告しません。しかし、libpqにはserver_versionとclient_encodingの値を取り出す仕組みがとれあずあります。アプリケーションは、付け焼き刃なコードでこれらの値を決定するのではなく、PQparameterStatusを使用することが求められています。(しかし、3.0より前の接続では、接続開始後にSETによるclient_encodingの変更はPQparameterStatusに反映されないことに注意してください。) server_versionについては、この情報をより比較し易い数値形式で返すPQserverVersionも参照してください。

standard_conforming_stringsの値がないと報告された場合、アプリケーションはoffと推測することができます。つまり、バックスラッシュは文字リテラル中のエスケープ文字として扱います。また、このパラメータが存在すると、エスケープ文字構文(E'...'')が受けられることを意味するものと取られます。

返されるポインタはconstと宣言されていますが、実際にはPGconn構造体に関連付けられた変化する領域を指し示します。このポインタが諸問い合わせに渡って有効なままであると考えるしないでください。

使用されるフロントエンド/バックエンドプロトコルを調査します。

```
int PQprotocolVersion(const PGconn *conn);
```

ある機能がサポートされているかどうかを決定するために、アプリケーションはこの関数を使用することができます。 現在、取り得る値は2(2.0プロトコル)、3(3.0プロトコル)、あるいは0(接続不良)です。 このプロトコルバージョンは接続の開始が完了した後で変更することはできません。 しかし、理論的には接続のリセット時に変更可能です。 PostgreSQL 7.4以降での通信時、通常3.0プロトコルが使用されます。 7.4より前のサーバでは2.0プロトコルのみをサポートします。(1.0プロトコルは廃止され、libpqではサポートされていません。)

バックエンドのバージョンの整数表現を返します。

```
int PQserverVersion(const PGconn *conn);
```

この関数を使用してアプリケーションは接続したデータベースサーバのバージョンを決定することができます。 この数値の形式は、メジャー、マイナー、リビジョン番号を2桁の10進数に変換し、連結させたものです。 例えば、バージョン8.1.5では80105を返し、バージョン8.2では80200を返します。(先頭の0は現れません。) 接続不良の場合は0が返されます。

接続における操作において、最も最近に生成されたエラーメッセージを返します。

```
char *PQerrorMessage(const PGconn *conn);
```

ほとんどすべてのlibpq関数は、失敗時にPQerrorMessage用のメッセージを設定します。 libpqでの決まりとして、空でないPQerrorMessageの結果は複数行に渡ることも可能で、最後に改行が含まれることがある点に注意してください。 呼び出し元はこの結果を直接解放してはいけません。 関連するPGconnハンドルがPQfinishに渡された時にこれは解放されます。 PGconn構造体への操作を跨って、この結果文字列が同一であると想定してはいけません。

サーバとの接続ソケットに対するファイル記述子番号を得ます。 有効な記述子なら値は0以上です。 -1の場合は、サーバとの接続がまだ開いていないことを示します。(これは通常の操作では変更することはできません。 接続設定中やリセット中に変更されます。)

```
int PQsocket(const PGconn *conn);
```

接続を処理するバックエンドのプロセスID(PID)を返します。

```
int PQbackendPID(const PGconn *conn);
```

バックエンドのPIDは、デバッグする場合やNOTIFYメッセージ(これは通知を発行したバックエンドプロセスのPIDを含んでいます)の比較に便利です。 このPIDはデータベースサーバホスト上で実行されているプロセスのものであり、ローカルホスト側のものではありません! 注意してください。

接続認証方式がパスワードを要求し、利用可能なパスワードがない場合真(1)を返します。 さもなくば偽(0)を返します。

```
int PQconnectionNeedsPassword(const PGconn *conn);
```

この関数を、接続試行に失敗した後でユーザにパスワード入力を促すかどうかを決定するために適用することができます。

接続認証方式でパスワードを使用する場合は真(1)、さもなくば偽(0)を返します。

```
int PQconnectionUsedPassword(const PGconn *conn);
```

この関数は、接続の試みが失敗したか成功したかの後に、サーバがパスワードを要求したかどうかを検出するために適用できます。

接続で使用されているSSL構造体を返します。 SSLが使用されていない場合はヌルを返します。

```
SSL *PQgetssl(const PGconn *conn);
```

この構造体は暗号化レベルの検証やサーバ証明書 の検査などに役立ちます。 この構造体についてはOpenSSLの文書を参照してください。

この関数の正しいプロトタイプを得るには、USE_SSLを定義する必要があります。 こうすると、OpenSSLのssl.hも自動的にインクルードされます。

31.3. コマンド実行関数

いったんデータベースサーバへの接続の確立が成功すれば、本節で説明する関数を使ってSQLの問い合わせやコマンドを実行します。

31.3.1. 主要な関数

PQexec

コマンドをサーバに送信し、結果を待機します。

```
PGresult *PQexec(PGconn *conn, const char *command);
```

戻り値はPGresultへのポインタ、場合によってはヌルポインタです。メモリ不足の状態、あるいはサーバへのコマンド送信が不可能といった深刻なエラーの場合を除けば、通常非ヌルのポインタが返ります。PQresultStatus関数を呼び出して、何かエラー（ヌルポインタ値を含むエラー。この場合はPGRES_FATAL_ERRORが返されます）がないか戻り値を検査しなければなりません。こうしたエラーの詳細な情報はPQerrorMessageで得ることができます。

コマンド文字列には(セミコロンで区切った)複数のSQLコマンドを含めることができます。単一のPQexec呼び出しで送信された複数の問い合わせは、単一トランザクションで処理されます。ただし、問い合わせ文字列内に明示的なBEGIN/COMMITがある場合は、複数のトランザクションに分離されます。しかし、返されるPGresult構造体はその文字列内で最後に実行されたコマンドの結果のみが含まれることに注意してください。そのコマンドの1つが失敗したとすると、文字列の処理はそこで中断し、エラー条件が含まれるPGresultが返されます。

PQexecParams

サーバにコマンドを送信し、結果を待ちます。ただし、SQLコマンドテキストとは別にパラメータを渡すことができます。

```
PGresult *PQexecParams(PGconn *conn,
                        const char *command,
                        int nParams,
                        const Oid *paramTypes,
                        const char * const *paramValues,
                        const int *paramLengths,
                        const int *paramFormats,
                        int resultFormat);
```

PQexecParamsとPQexecは似ていますが、前者は次の機能が追加されています。パラメータ値をコマンド文字列とは別に適切に指定することができ、また、問い合わせの結果をテキスト書式としてでもバイナリ書式としてでも要求できます。

PQexecParamsはプロトコル3.0以降でのみサポートされ、プロトコル2.0で使った場合は失敗します。

この関数の引数を以下に示します。

conn

接続オブジェクトです。これを通してコマンドを送信します。

command

実行させるSQLコマンド文字列です。パラメータが使用される場合は、コマンド文字列内で\$1、\$2などのように参照されます。

nParams

提供されるパラメータ数です。これは配列paramTypes[]、paramValues[]、paramLengths[]、paramFormats[]の要素数です。（この配列ポインタは、nParamsが0の場合、NULLとすることができます。）

paramTypes[]

パラメータシンボルに代入されるデータ型をOIDで指定したものです。paramTypesがNULL、または、ある配列要素が0の場合、サーバは、型指定のないリテラル文字列に対して行う推定方法と同じ方法を使用して、パラメータシンボルのデータ型を推定します。

paramValues[]

パラメータの実際の値を指定します。配列内のヌルポインタは対応するパラメータがNULLであることを意味します。さもなくば、このポインタはゼロ終端のテキスト文字列（テキスト書式）、または、サーバで想定している書式によるバイナリデータ（バイナリ書式）を指し示します。

paramLengths[]

バイナリ書式のパラメータの実データ長を指定します。 NULLパラメータおよびテキスト書式のパラメータでは無視されます。 バイナリパラメータが存在しない場合、この配列ポインタはヌルとしてもかまいません。

paramFormats[]

パラメータがテキスト（パラメータに対応する配列要素に0を設定）か、バイナリ（パラメータに対応する配列要素に1を設定）かを指定します。 この配列ポインタがヌルの場合、すべてのパラメータはテキスト文字列であると仮定されます。

バイナリ書式で渡された値は、バックエンドが想定する内部表現の知識を必要とします。 例えば、整数はネットワークバイト順に渡されなければなりません。 numericによる値は、src/backend/utils/adt/numeric.c::numeric_send() およびsrc/backend/utils/adt/numeric.c::numeric_recv() で実装されたようにサーバストレージ書式の知識を必要とします。

resultFormat

結果をテキスト書式で取り出したい場合は0を、バイナリ書式で取り出したい場合は1を指定します。（現時点では、プロトコル内部では実現可能ですが、結果の列ごとに異なる書式を指定して取り出す機構は存在しません。）

PQexecに対するPQexecParamsの主要な利点は、コマンド文字列とパラメータ値を分離することができることです。 これにより、面倒でエラーを招きやすい引用符付けやエスケープ処理を行なう必要がなくなります。

PQexecと異なり、PQexecParamsは、文字列内に最大でも1つのSQLコマンドを入れることができます。（セミコロンを入れることはできますが、空でないコマンドを1つ以上入れることはできません。） これは、プロトコル自体の制限ですが、SQLインジェクション攻撃に対する追加の防御となりますので、多少役に立ちます。

ティップ: OID経由のパラメータ型の指定は、特にプログラムの中で特定のOID値がソースに直接書き込まれることを好まない場合には退屈です。 しかしながら、パラメータの型をサーバ自身で決定できない場合や、望む型と異なる型を選択する場合であっても、これを避けることができます。 SQLコマンドテキストでどのデータ型を送信するかを示すためにパラメータシンボルに明示的なキャストをつけてください。 以下が例です。

```
SELECT * FROM mytable WHERE x = $1::bigint;
```

デフォルトではパラメータ\$1の型はxと同じデータ型に割り当てられますが、これにより強制的にbigintとして扱われます。 この方法または型のOIDを数字で指定する方法で、パラメータの型を強制的に決定することがバイナリ書式においてパラメータ値を送る時に強く推奨されます。 これは、バイナリ書式はテキスト書式より情報が少なく、そのために、サーバが型の不一致という問題を検出する機会が少なくなるためです。

PQprepare

指定パラメータを持つプリペアド文の作成要求を送信し、その完了を待ちます。

```
PGresult *PQprepare(PGconn *conn,
                    const char *stmtName,
                    const char *query,
                    int nParams,
                    const Oid *paramTypes);
```

PQprepareは、後でPQexecPreparedを使用して実行するプリペアド文を作成します。 この機能を使用して、繰り返し使用されるコマンドの解析と計画作成を実行時に毎回行うのではなく、一回のみ行うようにすることができます。

PQprepareはプロトコル3.0以降でのみサポートされ、プロトコル2.0を使用している場合は失敗します。

この関数はquery文字列からstmtNameという名前のプリペアド文を作成します。 queryは単一のSQLコマンドでなければなりません。 stmtNameを""にして、無名の文を作成することができます。 もし、無名の文が既に存在していた場合は自動的に置き換えられます。 その他の場合、文の名前が現在のセッションで既に存在するとエラーになります。 何らかのパラメータが使用される場合、問い合わせ内では\$1、\$2などで参照します。 nParamsはパラメータ数です。 その型については事前にparamTypes[] 配列で指定されています。（nParamsがゼロの場合、この配列ポインタはNULLにすることができます。） paramTypes[] は、OIDによりパラメータシンボルに割り当てるデータ型を指定します。 paramTypesがNULLの場合、もしくは、配列内の特定要素がゼロの場合、サーバはそのパラメータシンボルに対して、型指定の無いリテラル文字列に対する処理と同等の方法でデータ型を割り当てます。 また、問い合わせではnParamsより多くのパラメータシンボルを使用することができます。 これらのシンボルに対するデータ型も同様に推測されます。（どのようなデータ型が推測されるかを検出する手法についてはPQdescribePreparedを参照してください。）

PQexec同様、結果は通常PGresultオブジェクトで、その内容でサーバ側の成功や失敗を示します。 ヌルという結果はメモリ不足や全くコマンドを送信することができなかったことを示します。 こうしたエラーの詳細情報を入手するにはPQerrorMessageを使用してください。

PQexecPreparedで使用するためのプリペアド文は、[PREPARE](#) SQL文を実行することでも作成可能です。 また、プリペアド文を

削除するlibpq関数はありませんが、この目的のために[DEALLOCATE](#) SQL文を使用することができます。

PQexecPrepared

指定パラメータによるプリペアド文の実行要求を送信し、結果を待ちます。

```
PGresult *PQexecPrepared(PGconn *conn,
                          const char *stmtName,
                          int nParams,
                          const char * const *paramValues,
                          const int *paramLengths,
                          const int *paramFormats,
                          int resultFormat);
```

PQexecPreparedとPQexecParamsは似ていますが、前者では実行されるコマンドは、問い合わせ文字列を与えるのではなく、事前にプリペアド文を指名することで指定されます。この機能により、繰り返し使用する予定のコマンドを実行する度にではなく、一度だけ解析、計画作成を行うことができます。この文は現在のセッションで事前に準備されていなければなりません。PQexecPreparedは、プロトコル3.0以降の接続でのみサポートされます。プロトコル2.0で使った場合は失敗します。

パラメータは、問い合わせ文字列ではなく指定されたプリペアド文の名前を与える点を除き、PQexecParamsと同じです。また、paramTypes[] パラメータは存在しません。(プリペアド文のパラメータ型はその作成時点で決定されているため、これは不要です。)

PQdescribePrepared

指定したプリペアド文に関する情報入手要求を送り、入手完了まで待機します。

```
PGresult *PQdescribePrepared(PGconn *conn, const char *stmtName);
```

PQdescribePreparedにより、アプリケーションは事前にプリペアド文に関する情報を入手できます。PQdescribePreparedはプロトコル3.0以降の接続でのみサポートされます。プロトコル2.0で使っていると失敗します。

stmtNameを""またはNULLとすることで、無名の文を参照することができます。これ以外では、存在するプリペアド文の名前でなければなりません。成功すると、PGRES_COMMAND_OKというステータスのPGresultが返されます。PQnparamsおよびPQparamtype関数をこのPGresultに適用して、プリペアド文のパラメータに関する情報を得ることができます。また、PQnfields、PQfname、PQftype関数などを使用して、文の結果列（もしあれば）に関する情報を提供できます。

PQdescribePortal

指定したポータルに関する情報入手要求を送信し、完了まで待機します。

```
PGresult *PQdescribePortal(PGconn *conn, const char *portalName);
```

PQdescribePortalにより、アプリケーションは事前に作成されたポータルの情報を入手することができます。(libpqはポータルへの直接アクセスする方法を提供していませんが、この関数を使用してDECLARE CURSOR SQLコマンドで作成したカーソルの属性を確認することができます。) PQdescribePortalはプロトコル3.0以降の接続でのみサポートされます。プロトコル2.0で使っていると失敗します。

portalNameに""またはNULLを指定して、無名のポータルを参照することができます。これ以外では、既存のポータルの名前でなければなりません。成功すると、PGRES_COMMAND_OKというステータスのPGresultが返されます。PQnfields、PQfname、PQftype関数などをこのPGresultに適用して、ポータルの結果列（もしあれば）に関する情報を得ることができます。

PGresult構造体はサーバから返された結果をカプセル化します。libpqアプリケーションのプログラマは注意してPGresultという抽象化を維持してください。以下のアクセス用関数を使用して、PGresultの内容を取り出してください。将来の変更に影響されますので、PGresult構造体のフィールドを直接参照することは避けてください。

PQresultStatus

コマンドの結果状態を返します。

```
ExecStatusType PQresultStatus(const PGresult *res);
```

PQresultStatusは以下のいずれかの値を返します。

PGRES_EMPTY_QUERY

サーバに送信された文字列が空でした。

PGRES_COMMAND_OK

データを返さないコマンドが正常終了しました。

PGRES_TUPLES_OK

データを返すコマンド(`SELECT`や`SHOW`など)が正常終了しました。

`PGRES_COPY_OUT`

(サーバからの)コピーアウトデータ転送が始まりました。

`PGRES_COPY_IN`

(サーバへの)コピーインデータ転送が始まりました。

`PGRES_BAD_RESPONSE`

サーバが不明な応答を返しました。

`PGRES_NONFATAL_ERROR`

致命的ではない(注意喚起もしくは警告)エラーが発生しました。

`PGRES_FATAL_ERROR`

致命的なエラーが発生しました。

`PGRES_COPY_BOTH`

(サーバからおよびサーバへの)コピーイン/アウトデータ転送が始まりました。現在こればストリーミングレプリケーションのみで使用されます。このためこの状態は通常のアプリケーションでは起こりません。

`PGRES_SINGLE_TUPLE`

`PGresult`には現在のコマンドからの結果タプルが1つ含まれます。この状態は問い合わせで単一行モードが選択された場合(項31.5参照)のみ起こります。

結果状態が`PGRES_TUPLES_OK`または`PGRES_SINGLE_TUPLE`であれば、以下で説明する関数を使って問い合わせが返した行を取り出すことができます。ただし、たまたま`SELECT`コマンドが返す行が0個だったような場合でも`PGRES_TUPLES_OK`となることに注意してください。`PGRES_COMMAND_OK`は、行を全く返さない(`INSERT`、`UPDATE`などの)コマンド用です。

`PGRES_EMPTY_QUERY`という応答はクライアントソフトウェアの不具合を示しているかもしれません。

`PGRES_NONFATAL_ERROR`状態の場合、結果は`PQexec`や他の問い合わせ実行関数によって直接返されません。その代わりに、この種の結果は注意喚起プロセッサ(項31.12参照)に渡されます。

`PQresStatus`

`PQresultStatus`が返す列挙型から状態コードを説明する文字列定数に変換します。呼び出し元はこの結果を解放してはいけません。

```
char *PQresStatus(ExecStatusType status);
```

`PQresultErrorMessage`

コマンドに関するエラーメッセージを返します。エラーが何もない場合は、空の文字列を返します。

```
char *PQresultErrorMessage(const PGresult *res);
```

エラーがあった場合、返される文字列の最後には改行が含まれます。呼び出し元はこの結果を直接解放してはいけません。関連する`PGresult`ハンドルが`PQclear`に渡された時にこれは解放されます。

(接続に対する) `PQerrorMessage`も、`PQexec`または`PQgetResult`呼び出しの直後なら(結果に対する) `PQresultErrorMessage`と同じ文字列を返します。しかし、接続に対するエラーメッセージは続いて操作を行うと変化してしまうのに対し、`PGresult`は自身が破棄されるまでそのエラーメッセージを維持し続けます。この`PQresultErrorMessage`は個々の`PGresult`に結び付けられた状態を確認する時に、そして`PQerrorMessage`は接続における最後の操作の状態を確認する時に使用してください。

`PQresultErrorField`

エラー報告の個々のフィールドを返します。

```
char *PQresultErrorField(const PGresult *res, int fieldcode);
```

`fieldcode`はエラーフィールド識別子です。以下に示すシンボルを参照してください。`PGresult`がエラーではない、もしくは、警告付きの結果である場合や指定したフィールドを含まない場合、`NULL`が返されます。通常フィールド値には改行が含まれません。フィールド値は関連する`PGresult`ハンドルが`PQclear`に渡された時に解放されます。

以下のフィールドコードが使用できます。

`PG_DIAG_SEVERITY`

深刻度。このフィールドの内容は(エラーメッセージの場合)ERROR、FATAL、もしくは、PANIC、(注意喚起メッセージの場合)WARNING、NOTICE、DEBUG、INFO、もしくは、LOGです。これらは、多言語化により翻訳されている可能性があります。常に存在します。

PG_DIAG_SQLSTATE

エラーのSQLSTATEコードです。SQLSTATEコードは発生したエラーの種類を識別します。フロントエンドアプリケーションにより、特定のデータベースエラーに対して所定の操作（エラー処理など）を行うために使用できます。起こり得るSQLSTATEコードの一覧については[付録A](#)を参照してください。このフィールドは多言語化されず、また、常に存在します。

PG_DIAG_MESSAGE_PRIMARY

可読性を高めた主要エラーメッセージです。(通常は1行です。)常に存在します。

PG_DIAG_MESSAGE_DETAIL

詳細です。問題に関するより詳細を表す補助的なエラーメッセージです。複数行に跨る可能性があります。

PG_DIAG_MESSAGE_HINT

ヒントです。問題の対応方法についての補助的な提言です。これは、詳細(detail)とは異なり、問題の事象ではなく、(適切でない可能性があります)アドバイスを提供することを目的としています。複数行に跨る可能性があります。

PG_DIAG_STATEMENT_POSITION

元の問い合わせ文字列のインデックスとなる、エラーが発生したカーソル位置を示す10進整数を持つ文字列です。先頭文字がインデックス1となり、また、バイトではなく、文字数で数えた位置です。

PG_DIAG_INTERNAL_POSITION

この定義はPG_DIAG_STATEMENT_POSITIONフィールドと同じです。しかし、これは、クライアントが発行したコマンドではなく、カーソル位置が内部生成コマンドを参照する場合に使用されます。このフィールドが存在する時は常にPG_DIAG_INTERNAL_QUERYフィールドが存在します。

PG_DIAG_INTERNAL_QUERY

失敗した内部生成コマンドのテキストです。これは、例えば、PL/pgSQL関数で発行されたSQL問い合わせになります。

PG_DIAG_CONTEXT

エラーが発生した文脈を示すものです。今の所、これは活動中の手続き言語関数や内部生成問い合わせの呼び出しスタックの追跡情報が含まれます。この追跡は行単位で1項目であり、その順番は呼び出し順の反対になります。

PG_DIAG_SOURCE_FILE

エラーが報告された場所のソースコードのファイル名です。

PG_DIAG_SOURCE_LINE

エラーが報告された場所のソースコードにおける行番号です。

PG_DIAG_SOURCE_FUNCTION

エラーを報告した、ソースコードにおける関数名です。

表示情報の必要に応じた整形はクライアントの責任です。具体的には、必要に応じて長い行を分割しなければなりません。エラーメッセージフィールド内の改行文字は、改行としてではなく段落として分かれたものとして取扱うべきです。

libpqで内部的に生成されたエラーは、深刻度と主要メッセージを持ちますが、通常は他のフィールドを持ちません。3.0より前のプロトコルのサーバで返されるエラーは、深刻度と主要メッセージ、場合によって詳細メッセージを持ちますが、他のフィールドを持ちません。

エラーフィールドはPGresultからのみ利用でき、PGconnからは利用できません。PQerrorFieldという関数はありません。

PQclear

PGresultに割り当てられた記憶領域を解放します。個々の問い合わせ結果は、必要なくなった時にPQclearで解放すべきです。

```
void PQclear(PGresult *res);
```

`PGresult` オブジェクトは必要な間保持することができます。新しい問い合わせを発行する場合でも、接続を閉じてしまうまでは `PGresult` は消えません。 `PGresult` を解放するには、`PQclear` を呼び出さなくてははいけません。その操作に失敗してしまうと、アプリケーションのメモリリークを引き起こしてしまいます。

31.3.2. 問い合わせ結果の情報の取り出し

これらの関数を使用して、正常終了した問い合わせ結果を示す(つまり、その状態が `PGRES_TUPLES_OK` または `PGRES_SINGLE_TUPLE` となっている) `PGresult` オブジェクトから情報を抽出することができます。また、成功した `Describe` 操作から情報を抽出することもできます。 `Describe` の結果はすべて、実際に問い合わせを実行した時に提供されるものと同じ列情報を持ちますが、行はありません。他の状態値を持つオブジェクトでは、これらの関数は、結果が0行0列であるものと同様に動作します。

`PQntuples`

問い合わせ結果内の行(タプル)数を返します。これは整数を結果として返しますので、32ビットオペレーティングシステムでは、大規模な結果セットの戻り値はオーバーフローする可能性があります。

```
int PQntuples(const PGresult *res);
```

`PQnfields`

問い合わせ結果の各行の列(フィールド)の数を返します。

```
int PQnfields(const PGresult *res);
```

`PQfname`

指定した列番号に対応する列の名前を返します。列番号は0から始まります。呼び出し元はこの結果を直接解放してはいけません。関連する `PGresult` ハンドルが `PQclear` に渡された時にこれは解放されます。

```
char *PQfname(const PGresult *res,
               int column_number);
```

列番号が範囲外であった場合、`NULL` が返ります。

`PQfnumber`

指定した列名に関連する列番号を返します。

```
int PQfnumber(const PGresult *res,
               const char *column_name);
```

指定した名前に一致する列がなければ、`-1` が返ります。

指定した名前はSQLコマンドの識別子同様に扱われます。つまり、二重引用符でくくられていない限り、小文字化されます。例えば、以下のSQLで生成された問い合わせ結果を考えます。

```
SELECT 1 AS FOO, 2 AS "BAR";
```

以下により、結果を取り出すことができます。

<code>PQfname(res, 0)</code>	<code>foo</code>
<code>PQfname(res, 1)</code>	<code>BAR</code>
<code>PQfnumber(res, "FOO")</code>	<code>0</code>
<code>PQfnumber(res, "foo")</code>	<code>0</code>
<code>PQfnumber(res, "BAR")</code>	<code>-1</code>
<code>PQfnumber(res, "¥"BAR¥")</code>	<code>1</code>

`PQftable`

指定した列の抽出元であるテーブルのOIDを返します。列番号は0から始まります。

```
Oid PQftable(const PGresult *res,
              int column_number);
```

列番号が範囲外の場合や指定した列がテーブル列への単純な参照でない場合、3.0より前のプロトコルを使用している場合は、`InvalidOid` が返されます。 `pg_class` システムテーブルに問い合わせ、どのテーブルが参照されているのかを正確に求めることができます。

`libpq` ヘッダファイルをインクルードすると、`Oid` 型と `InvalidOid` 定数が定義されます。これらは両方とも何らかの整数型です。

`PQftablecol`

指定した問い合わせ結果の列を作成した列の(それが属するテーブル内での)列番号を返します。問い合わせ結果の列番号

は0から始まります。 テーブル列には0以外の番号が付けられています。

```
int PQftablecol(const PGresult *res,
                int column_number);
```

列番号が範囲外の場合や指定した列がテーブル列への単純な参照でなかった場合、3.0より前のプロトコルを使用している場合は、ゼロが返されます。

PQfformat

指定した列の書式を示す書式コードを返します。 列番号は0から始まります。

```
int PQfformat(const PGresult *res,
              int column_number);
```

ゼロという書式コードはテキストデータ表現を示し、1という書式コードはバイナリ表現を示します。(他のコードは将来の定義のために予約されています。)

PQftype

指定した列番号に関連したデータ型を返します。 返された整数はその型の内部的なOID番号です。 列番号は0から始まります。

```
Oid PQftype(const PGresult *res,
            int column_number);
```

`pg_type` システムテーブルに問い合わせ、各種データ型の名前や属性を得ることができます。 組み込みデータ型のOIDは、ソースツリー内の`src/include/catalog/pg_type.h` ファイル内で定義されています。

PQfmod

指定した列番号に関連した列の型修飾子を返します。 列番号は0から始まります。

```
int PQfmod(const PGresult *res,
           int column_number);
```

修飾子の値の解釈は型に固有なものです。 通常これらは精度やサイズの制限を示します。 -1という値は"使用できる情報がない"ことを示します。 ほとんどのデータ型は修飾子を使用しません。 この場合は常に-1という値になります。

PQfsize

指定した列番号に関連した列のバイト単位のサイズを返します。 列番号は0から始まります。

```
int PQfsize(const PGresult *res,
            int column_number);
```

`PQfsize` はデータベース行内でその列用に割り当てられる領域を返します。 言い替えると、そのデータ型についてのサーバでの内部表現のサイズです。(従って、実際にはクライアントから見るとあまり役には立ちません。) 負の値は可変長データ型を示します。

PQbinaryTuples

`PGresult` がバイナリデータを持つ場合は1を、テキストデータを持つ場合は0を返します。

```
int PQbinaryTuples(const PGresult *res);
```

この関数は廃れたものです。(COPYを行う接続での使用を除きます。) 単一の `PGresult` で、ある列はテキストデータを持ち、他の列ではバイナリデータを持つことが可能であるためです。 `PQfformat` の利用が推奨されます。 結果のすべての列がバイナリ(書式1)の場合のみ `PQbinaryTuples` は1を返します。

PQgetvalue

`PGresult` の1行における単一フィールドの値を返します。 行番号と列番号は0から始まります。 呼び出し元はこの結果を直接解放してはいけません。 関連する `PGresult` ハンドルが `PQclear` に渡された時に、これは解放されます。

```
char *PQgetvalue(const PGresult *res,
                 int row_number,
                 int column_number);
```

テキスト書式のデータでは、`PQgetvalue` で返される値はフィールド値をヌル終端の文字列表現となります。 バイナリ書式のデータでは、この値はデータ型の `typsend` 関数と `typreceive` 関数で決まるバイナリ表現となります。(実際にはこの場合でも値の終わりにゼロというバイトが付与されます。 しかし、この値の内部には大抵の場合ヌルが埋め込まれていますので、通常このバイトは有用ではありません。)

フィールド値がNULLの場合、空文字列が返されます。 NULL値と空文字列という値とを区別する方法は `PQgetisnull` を参照してください。

PQgetvalueによって返されるポインタはPGresult構造体の一部の格納領域を指し示します。このポインタが指し示すデータを変更すべきではありません。また、PGresult構造体を解放した後も使用し続ける場合は、データを別の格納領域に明示的にコピーしなければなりません。

PQgetisnull

フィールドがNULL値かどうか検査します。行番号と列番号は0から始まります。

```
int PQgetisnull(const PGresult *res,
                int row_number,
                int column_number);
```

この関数は、フィールドがNULLの場合に1を、フィールドが非NULL値を持つ場合は0を返します。（PQgetvalueでは、NULLフィールドはヌルポインタではなく空文字列を返すことに注意してください。）

PQgetlength

実際のフィールド値の長さをバイト単位で返します。行番号と列番号は0から始まります。

```
int PQgetlength(const PGresult *res,
                int row_number,
                int column_number);
```

これは特定のデータ値についての実際のデータ長です。つまり、PQgetvalueによって指し示されるオブジェクトのサイズです。テキストデータ書式ではstrlen()と同一です。バイナリ書式ではこれは重要な情報です。実際のデータ長を取り出すためにPQfsizeを信用してはなりません。

PQnparams

プリペアド文のパラメータ数を返します。

```
int PQnparams(const PGresult *res);
```

この関数はPQdescribePreparedの結果を確認する時にのみ有用です。他の種類の問い合わせではゼロを返します。

PQparamtype

指定された文パラメータのデータ型を返します。パラメータ番号は0から始まります。

```
Oid PQparamtype(const PGresult *res, int param_number);
```

この関数は、PQdescribePreparedの結果を確認する時にのみ有用です。他の種類の問い合わせではゼロを返します。

PQprint

すべての行と列名(省略可能)を指定した出力ストリームに表示します。

```
void PQprint(FILE *fout,          /* 出力ストリーム */
              const PGresult *res,
              const PQprintOpt *po);

typedef struct
{
    pqbool    header;          /* フィールドヘッダ情報と行数の表示出力 */
    pqbool    align;           /* 位置揃えのためのフィールドへの埋め込み */
    pqbool    standard;        /* 古い、無くなりそうな書式 */
    pqbool    html3;           /* HTML表出力 */
    pqbool    expanded;        /* 拡張テーブル */
    pqbool    pager;           /* 必要に応じたページの使用 */
    char      *fieldSep;        /* フィールド区切り文字 */
    char      *tableOpt;        /* HTML表要素の属性 */
    char      *caption;         /* HTML 表の表題 */
    char      **fieldName;      /* フィールド名を置き換えるNULL終端の配列 */
} PQprintOpt;
```

この関数は以前に問い合わせ結果を表示するためにpsqlで使用されていましたが、今ではもう使用されていません。これはすべてのデータがテキスト書式であるという前提で動作することに注意してください。

31.3.3. 他の結果情報の取り出し

これらの関数はPGresultオブジェクトからその他の情報を取り出すために使用されます。

PQcmdStatus

PGresultを生成したSQLコマンドのコマンド状態タグを返します。

```
char *PQcmdStatus(PGresult *res);
```

これは通常単なるコマンド名ですが、処理行数など追加情報が含まれる場合もあります。呼び出し元はこの戻り値を直接解放してはいけません。関連するPGresultハンドルがPQclearに渡された時にこれは解放されます。

PQcmdTuples

SQLコマンドにより影響を受けた行数を返します。

```
char *PQcmdTuples(PGresult *res);
```

この関数はPGresultを生成したSQLコマンドにより影響を受けた行数を持つ文字列を返します。この関数はSELECT、CREATE TABLE AS、INSERT、UPDATE、DELETE、MOVE、FETCH、COPY文の実行、あるいは、INSERT、UPDATE、DELETEを含むプリペアド問い合わせのEXECUTE文の後でのみ使用することができます。PGresultを生成したコマンドが他のコマンドであった場合、PQcmdTuplesは空文字列を返します。呼び出し元はこの戻り値を直接解放してはいけません。関連するPGresultハンドルがPQclearに渡された時にこれは解放されます。

PQoidValue

SQLコマンドが、OIDを持つテーブル内に1行のみを挿入するINSERTだった場合、あるいは、適切なINSERTを持つプリペアド問い合わせのEXECUTEだった場合に、挿入された行のOIDを返します。さもなくばInvalidOidを返します。また、INSERT文の影響を受けたテーブルがOIDを持たなかった場合、この関数はInvalidOidを返します。

```
Oid PQoidValue(const PGresult *res);
```

PQoidStatus

この関数はPQoidValueのため廃止予定になりました。またこれはスレッドセーフではありません。これは挿入された行のOIDを文字列として返します。一方PQoidValueはOID値を返します。

```
char *PQoidStatus(const PGresult *res);
```

31.3.4. SQLコマンドに含めるための文字列のエスケープ処理

PQescapeLiteral

```
char *PQescapeLiteral(PGconn *conn, const char *str, size_t length);
```

PQescapeLiteralは、SQLコマンド内で使用するために文字列をエスケープします。これは、SQLコマンド内のリテラル定数としてデータ値を挿入する時に有用です。特定の文字(引用符やバックスラッシュ)は、SQLパーサによって特殊な解釈がなされないようにエスケープされなければなりません。PQescapeLiteralはこの操作を行います。

PQescapeLiteralはstrパラメータをエスケープしたものをmalloc()で割り当てたメモリ内に返します。その結果が不要になったら、そのメモリをPQfreemem()を使用して解放しなければなりません。ゼロバイト終端は必要なく、lengthに含めて数えてはいけません。(lengthバイトを処理する前にゼロバイト終端が見つかる、PQescapeLiteralはそのゼロで終了します。この動作はstrncpyと似ています。)返される文字列では、PostgreSQL文字列リテラルパーサで適切に処理することができるように、すべての特殊文字は置換されます。ゼロバイト終端も追加されます。PostgreSQLの文字列リテラルでは前後に必要な単一引用符も、その結果文字列には含まれています。

エラー時、PQescapeLiteralはNULLを返し、connオブジェクト内に適切なメッセージを残します。

ティップ: 信用できない入力元から受けとった文字列を扱う場合に適切なエスケープ処理を行なうことは非常に重要です。さもなくば、セキュリティ上の危険性が発生します。"SQLインジェクション"攻撃という弱点となり、好ましくないSQLコマンドがデータベースに流れてしまいます。

データがPQexecParamsまたは同義のルーチン内で別のパラメータとしてデータ値が渡される場合は、エスケープする必要も修正することもないことに注意してください。

PQescapeIdentifier

```
char *PQescapeIdentifier(PGconn *conn, const char *str, size_t length);
```

PQescapeIdentifierは、テーブル、列、関数名などのSQL識別子として使用できるように文字列をエスケープします。これはユーザが提供した識別子に、そのままではSQLパーサで識別子として解釈されない特殊な文字が含まれる可能性がある場合、または、大文字小文字の違いを維持しなければならない状況で識別子に大文字が含まれる可能性がある場合に有用です。

PQescapeIdentifierはstrパラメータをSQL識別子としてエスケープしたものをmalloc()で割り当てたメモリ内に返します。その結果が不要になったら、そのメモリをPQfreemem()を使用して解放しなければなりません。ゼロバイト終端は必要なく、lengthに含めて数えてはいけません。(lengthバイトを処理する前にゼロバイト終端が見つかる、PQescapeIdentifierはそのゼロで終了します。この動作はstrncpyと似ています。)返される文字列では、SQL識別子として適切に処理することができるように、すべての特殊文字は置換されます。ゼロバイト終端も追加されます。その結果文字列の前後には二重引用符が付与されます。

エラー時、PQescapeIdentifierはNULLを返し、connオブジェクト内に適切なメッセージを残します。

ティップ: 文字列リテラルと同様、SQLインジェクション攻撃を防ぐために、信頼できない入力元から受けとる場合にはSQL識別子をエスケープしなければなりません。

PQescapeStringConn

```
size_t PQescapeStringConn(PGconn *conn,
                           char *to, const char *from, size_t length,
                           int *error);
```

PQescapeStringConnは、PQescapeLiteralとほぼ同様に文字列リテラルをエスケープします。PQescapeLiteralとは異なり、呼び出し元が適切な大きさのバッファを提供することに責任を持ちます。さらにPQescapeStringConnはPostgreSQLの文字列リテラルとして囲まれなければならない単一引用符を生成しません。これは、結果をSQLコマンドに挿入するときに付与しなければなりません。fromパラメータはエスケープ対象の文字列の先頭を指すポインタです。lengthパラメータはこの文字列のバイト数を示します。ゼロバイト終端は必要なく、また、lengthではこれを数えてはなりません。(もしlength/バイト処理する前にゼロバイト終端が存在すると、PQescapeStringConnはそのゼロで終了します。この動作はstrncpyと同様です。) toは、最低でもlengthの2倍よりも1バイト多い文字を保持可能なバッファへのポインタにしなければなりません。さもないと、動作は不定になります。toとfrom文字領域が重なる場合の動作も不定です。

errorパラメータがNULLでなければ、*errorには成功の0か、エラーの0以外が設定されます。現時点であり得る唯一のエラー条件は、元文字列に無効なマルチバイト符号が含まれている場合です。出力文字列はエラーであっても生成されますが、サーバが不整合として却下することが想定できます。エラーの際、適切なメッセージはerrorがNULLかどうかにかかわらずconnオブジェクト内に格納されます。

PQescapeStringConnはtoに書き出したバイト数を返します。ただし、文字数にはゼロバイト終端は含まれません。

PQescapeString

PQescapeStringはPQescapeStringConnの推奨されない古いものです。

```
size_t PQescapeString(char *to, const char *from, size_t length);
```

PQescapeStringConnとの唯一の違いは、PQescapeStringはPGconnやerrorパラメータを取らないことです。このため(文字符号化方式のような)接続属性に依存する振舞いを調整できません。その結果間違った結果を返す可能性があります。また、エラー状態を通知する機能がありません。

PQescapeStringは、一度に1つのPostgreSQL接続のみで動作するクライアントプログラムでは安全に利用できます。(この場合知らなければならない"裏側に隠された情報"を知ることができるからです。)他の場合には、セキュリティ要因でありPQescapeStringConnを利用することで避けなければなりません。

PQescapeByteaConn

bytea型としてSQLコマンド内で使用するバイナリデータをエスケープします。PQescapeStringConnと同様、これは、SQLコマンド文字列にデータを直接含める場合にのみに使用されます。

```
unsigned char *PQescapeByteaConn(PGconn *conn,
                                  const unsigned char *from,
                                  size_t from_length,
                                  size_t *to_length);
```

SQL文内のbyteaリテラルの一部として使用する場合、特定のバイト値はエスケープされなければなりません。

PQescapeByteaConnは16進数符号化またはバックスラッシュエスケープ処理を使用してバイトをエスケープします。詳しくは[項8.4](#)を参照してください。

fromパラメータはエスケープ対象の文字列の先頭バイトを指し示すポインタです。from_lengthパラメータは、このバイナリ列内のバイト数を指定します。(ゼロバイト終端は不要、かつ、数えられません。) to_lengthパラメータは結果となるエスケープされた文字列の長さを保持する変数へのポインタです。この結果文字列長は、結果内のゼロバイト終端を含みます。

PQescapeByteaConnは、fromパラメータが示すバイナリ文字列をエスケープしたものをmalloc()で確保したメモリ内に返します。その結果が不要になったら、このメモリをPQfreememを使用して解放しなければなりません。返される文字列では、PostgreSQLリテラル文字列パーサとbytea入力関数によって適切に処理できるように、すべての特殊な文字が置換されています。ゼロバイト終端も追加されます。PostgreSQLのリテラル文字列をくくる単一引用符は結果文字列には含まれません。

エラー時、ヌルポインタを返し適切なエラーメッセージをconnオブジェクトに格納します。現在、唯一あり得るエラーは結果文字列のメモリ不足です。

PQescapeBytea

PQescapeByteaは、PQescapeByteaConnの推奨されない古いものです。

```
unsigned char *PQescapeBytea(const unsigned char *from,
                             size_t from_length,
                             size_t *to_length);
```

PQescapeByteaのPQescapeByteaConnとの唯一の違いは、PGconnパラメータです。このためPQescapeByteaは、一度に1つのPostgreSQL接続を使用するクライアントプログラムのみで安全に利用することができます。（この場合知らなければならない"裏側に隠された情報"を知ることができるからです。）複数のデータベース接続を使用するプログラムでは間違った結果を返す可能性があります。（このような場合はPQescapeByteaConnを使用してください。）

PQunescapeBytea

バイナリデータの文字列表現をバイナリデータに変換します。つまり、PQescapeByteaの逆です。これは、byteaデータをテキスト書式で受けとった場合に必要とされます。しかし、バイナリ書式で受けとった場合は不要です。

```
unsigned char *PQunescapeBytea(const unsigned char *from, size_t *to_length);
```

fromパラメータは、例えば、bytea列にPQgetvalueを行なった場合に返される可能性がある、文字列を指し示すポインタです。PQunescapeByteaは、この文字列表現をバイナリ表現に変換します。malloc()で確保したバッファへのポインタを返します。エラー時はNULLです。また、このバッファのサイズをto_lengthに格納します。不要になったら、この結果をPQfreememを使用して解放しなければなりません。

この変換は、PQescapeByteaの逆ではありません。文字列はPQgetvalueから受け取る場合"エスケープされた"ことを予想しないためです。特にこれは、文字列の引用符付けを意識する必要がなく、そのためPGconnパラメータを持つ必要がないことを意味します。

[前のページ](#)
接続状態関数

[ホーム](#)
[上に戻る](#)

[次のページ](#)
非同期コマンドの処理

31.4. 非同期コマンドの処理

`PQexec`関数は普通の同期処理のアプリケーションにおけるコマンドの送信に適したものです。しかし、一部のユーザにとって重要な問題となり得る、数個の問題があります。

- `PQexec` はコマンドが完了するまで待機します。アプリケーションによっては(例えばユーザインタフェースの調整処理など)他に行うべき作業があります。この場合は応答待ちでブロックさせたくはありません。
- クライアントアプリケーションの実行が結果を待っている間停止されるため、アプリケーションで送信したコマンドをキャンセルさせる指示を行うことは困難です。(シグナルハンドラを使って達成することができますが、他の方法はありません。)
- `PQexec`が返すことができる`PGresult`構造体は1つだけです。もし送信した問い合わせ文字列が複数のSQLコマンドを含んでいる場合、`PQexec`は最後のものだけを除いて、残りすべての`PGresult`を破棄してしまいます。
- `PQexec`は常にコマンドの結果全体を収集し、1つの`PGresult`内に保管します。アプリケーションにおけるエラー処理を簡単にしますが、多くの行になる結果では非現実的になるかもしれません。

アプリケーションにとってこのような制限が望ましくない場合は、代わりに`PQexec`を構成する関数`PQsendQuery`と`PQgetResult`を使用してください。ま

た、`PQsendQueryParams`と`PQsendPrepare`、`PQsendQueryPrepared`、`PQsendDescribePrepared`、`PQsendDescribePortal`もあり、`PQgetResult`を使用して、それぞれ`PQexecParams`と`PQprepare`、`PQexecPrepared`、`PQdescribePrepared`、`PQdescribePortal`と同等の機能を行うことができます。

`PQsendQuery`

結果を待つことなく、サーバにコマンドを発行します。コマンドの登録に成功した場合1が、失敗した場合0が返されます。(後者の場合、`PQerrorMessage`を使用して失敗についてのより多くの情報を取り出してください。)

```
int PQsendQuery(PGconn *conn, const char *command);
```

`PQsendQuery`呼び出しが成功したら、`PQgetResult`を繰り返し呼び出して、実行結果を取得します。`PQgetResult`がヌルポインタを返し、コマンドが完了したことを示すまでは、(同じ接続で)`PQsendQuery`を再度呼び出すことはできません。

`PQsendQueryParams`

結果を待つことなく、サーバにコマンドとパラメータとを分けて発行します。

```
int PQsendQueryParams(PGconn *conn,
                      const char *command,
                      int nParams,
                      const Oid *paramTypes,
                      const char * const *paramValues,
                      const int *paramLengths,
                      const int *paramFormats,
                      int resultFormat);
```

これは、問い合わせのパラメータが問い合わせ文字列と分けて指定できる点を除き、`PQsendQuery`と同じです。この関数のパラメータは`PQexecParams`と同様に扱われます。`PQexecParams`同様、これは2.0プロトコルでは動作しませんし、問い合わせ文字列には1つのコマンドしか指定できません。

`PQsendPrepare`

指定パラメータを持つプリペアド文の作成要求を送信します。その完了を待ちません。

```
int PQsendPrepare(PGconn *conn,
                  const char *stmtName,
                  const char *query,
                  int nParams,
                  const Oid *paramTypes);
```

これは`PQprepare`の非同期版です。要求の登録に成功した場合1が、失敗した場合0が返されます。呼び出しの成功の後、サーバがプリペアド文の生成に成功したかを確認するためには`PQgetResult`を呼び出してください。この関数のパラメータは`PQprepare`と同様に扱われます。`PQprepare`同様、これは2.0プロトコルの接続では動作しません。

`PQsendQueryPrepared`

結果を待つことなく、指定したパラメータでプリペアド文の実行要求を送信します。

```
int PQsendQueryPrepared(PGconn *conn,
                        const char *stmtName,
                        int nParams,
                        const char * const *paramValues,
                        const int *paramLengths,
                        const int *paramFormats,
                        int resultFormat);
```


これはPQsendQueryParamsと似ていますが、実行されるコマンドは問い合わせ文字列ではなく、事前に準備された文の名前で指定されます。この関数のパラメータはPQexecPreparedと同様に扱われます。PQexecPrepared同様、これは2.0プロトコルでは動作しません。

PQsendDescribePrepared

指定したプリペアド文に関する情報入手要求を送ります。入手完了まで待機しません。

```
int PQsendDescribePrepared(PGconn *conn, const char *stmtName);
```

これはPQdescribePreparedの非同期版です。要求の受け付けが可能であれば1が返されます。不可能であれば0が返されます。呼び出しに成功した後、PQgetResultを呼び出して結果を入手してください。この関数のパラメータはPQdescribePreparedと同じように扱われます。PQdescribePrepared同様、2.0プロトコル接続では動作しません。

PQsendDescribePortal

指定したポータルに関する情報入手要求を送信します。完了まで待機しません。

```
int PQsendDescribePortal(PGconn *conn, const char *portalName);
```

これはPQdescribePortalの非同期版です。要求の受け付けが可能であれば1が返されます。不可能であれば0が返されます。呼び出しに成功した後、PQgetResultを呼び出して結果を入手してください。この関数のパラメータはPQdescribePortalと同じように扱われます。PQdescribePortal同様、2.0プロトコル接続では動作しません。

PQgetResult

以前に呼び出し

たPQsendQuery、PQsendQueryParams、PQsendPrepare、PQsendQueryPrepared、PQsendDescribePrepared、PQsendDescribePortalから次の結果を待ち、その結果を返します。コマンドが完了し、これ以上結果がない場合は、ヌルポインタが返されます。

```
PGresult *PQgetResult(PGconn *conn);
```

PQgetResultは、コマンドの完了を示すヌルポインタが返るまで、繰り返し呼び出さなければなりません。(コマンド実行中以外での呼び出しでは、PQgetResultは単にヌルポインタを返します。)PQgetResultの非ヌルの結果はそれぞれ前述と同じPGresultアクセス用関数を使用して処理されなければなりません。各結果オブジェクトに対する処理が終わったら、そのオブジェクトをPQclearを使用して解放することを忘れないでください。コマンドが活動中、かつ、必要な応答データがまだPQconsumeInputで読み込まれていない場合にのみ、PQgetResultがブロックすることに注意してください。

注意: PQresultStatusが致命的なエラーを示した場合であっても、libpqがエラー情報を完全に処理できるようにヌルポインタが返されるまでPQgetResultを呼び出さなければなりません。

PQsendQueryとPQgetResultを使うことでPQexecの問題は1つ解決します。つまり、コマンドが複数のSQLコマンドを含んでいる場合でも、これらのコマンドの結果を個々に得ることができるわけです(これは多重処理を単純な形で実現します。単一のコマンド文字列に含まれる複数の問い合わせの内、後ろのものが処理中でもフロントエンドは先に完了した結果から扱うことができるからです)。

PQsendQueryおよびPQgetResultで得られる、その他のよく望まれる機能は一度に多くの問い合わせ結果1行で受け取ることです。これについては[項31.5](#)で説明します。

By itself, calling PQgetResult will still cause the client to block until the server completes the next SQL command. This can be avoided by proper use of two more functions: --> サーバが次のSQLコマンドの処理に入ると、それが完了するまでやはりPQgetResultの呼び出しがフロントエンドをブロックしてしまいます。さらに以下の2つの関数をうまく使用してこれを防ぐことができます。

PQconsumeInput

サーバからの入力が可能になった場合、それを吸い取ります。

```
int PQconsumeInput(PGconn *conn);
```

PQconsumeInputは通常、"エラーなし"を示す1を返しますが、何らかの障害があると0を返します(この場合は、PQerrorMessageを参考にしてください)。この結果は、何らかの入力データが実際に収集されたかどうかを示しているのではないことに注意してください。PQconsumeInputの呼び出し後、アプリケーションはPQisBusy、または必要があればPQnotifiesを呼び出して状態に変化がないか調べることができます。

PQconsumeInputは、結果や通知を扱うようにまだ準備していないアプリケーションからでも呼び出すことができます。この関数は有効なデータを読み込んでバッファに保存し、結果としてselectによる読み込み準備完了の通知をリセットします。従ってアプリケーションはPQconsumeInputを使うとselect()の検査条件をただちに満たすことができますから、あとはゆっくりと結果を調べてやればいいわけです。

PQisBusy

この関数が1を返したのであれば、問い合わせは処理の最中で、PQgetResultも入力を待ったままブロック状態になってしまいます。0が返ったのであれば、PQgetResultを呼び出してもブロックされないことが保証されます。

```
int PQisBusy(PGconn *conn);
```

`PQisBusy` 自身はサーバからデータを読み込む操作をしません。 ですから、まず最初に `PQconsumeInput` を呼び出す必要があります。 そうしないとビジー状態がいつまでも続きます。

これら3関数を使用するアプリケーションは通常、`select()` もしくは `poll()` を使用するメインループを持ち、対応しなければならないすべての状態を待機しています。 その内の1つの条件は、サーバからの利用可能な入力となるでしょう。 これは、`select()` の見地からは、`PQsocket` で識別されるファイル記述子上で読み込み可能なデータがあることを意味します。 メインループが入力準備完了を検出すると、その入力を読み込むために `PQconsumeInput` を呼び出さなければなりません。 そして、`PQisBusy` を、更に `PQisBusy` が偽(0)を返す場合に `PQgetResult` も呼び出すことができます。 また、`PQnotifies` を呼び出して、NOTIFYメッセージ([項31.8](#)を参照)を検出することもできます。

また、`PQsendQuery/PQgetResult` を使用するクライアントは、サーバで処理中のコマンドに対してキャンセルを試行することができません。 [項31.6](#)を参照してください。 しかし、`PQcancel` の戻り値と関係なく、アプリケーションは `PQgetResult` を使用した通常の結果読み取り手順を続けなければなりません。 キャンセル手続きの成功は単に、そのコマンドを通常よりも早めに終わらせるだけです。

上述の関数を使用して、データベースサーバからの入力待ちのためのブロックを行わずに済みます。 しかしまだ、サーバへの出力送信を待つためにアプリケーションはブロックする可能性があります。 これは比較的あまり発生しませんが、非常に長いSQLコマンドやデータ値が送信される場合に発生することがあります。(しかし、アプリケーションが `COPY IN` 経由でデータを送信する場合よく発生します。) この発生を防ぎ、完全な非ブロックのデータベース操作を行うためには、さらに以下の関数を使用してください。

`PQsetnonblocking`

接続の非ブロック状態を設定します。

```
int PQsetnonblocking(PGconn *conn, int arg);
```

`arg` が1の場合、接続状態を非ブロックに設定します。 `arg` が0の場合はブロックに設定します。 問題がなければ0が、エラー時は-1が返ります。

非ブロック状態では `PQsendQuery`、`PQoutline`、`PQputnbytes` および `PQendcopy` の呼び出しはブロックされませんが、再度呼び出さなければならない場合、エラーが返ります。

`PQexec` は非ブロックモードにはしたがわないことに注意してください。 この関数の呼び出しは、必ずブロック方式で動作します。

`PQisnonblocking`

データベース接続のブロック状態を返します。

```
int PQisnonblocking(const PGconn *conn);
```

接続が非ブロック状態の場合は1が、ブロック状態の場合は0が返ります。

`PQflush`

キューに蓄えられたサーバへの出力データの吐き出しを行います。 成功時(および送信キューが空の場合)は0が返ります。 何らかの原因で失敗した場合は-1が、送信キュー内のデータをすべて送信できなかった場合は1が返ります。(これは接続が非ブロックの場合にのみ発生します。)

```
int PQflush(PGconn *conn);
```

非ブロック接続時にはコマンドやデータを送信した後に、`PQflush` を呼び出してください。 1が返った場合、ソケットの書き込み準備ができるまで待ち、再度呼び出してください。 これを0が返るまで繰り返してください。 `PQflush` が0を返した後は、ソケットの読み込み準備が整うまで待ち、上述のように応答を読み取ってください。

[前のページ](#)[コマンド実行関数](#)[ホーム
上に戻る](#)[次のページ](#)[1行1行問い合わせ結果を受け取る](#)

31.5. 1 行 1 行問い合わせ結果を受け取る

通常、libpqはSQLコマンドの結果全体を収集し、それを1つのPGresultとしてアプリケーションに返します。これは、多くの行数を返すコマンドでは動作しなくなるかもしれません。こうした場合、アプリケーションはPQsendQueryとPQgetResultを単一行モードで使うことができます。このモードでは、結果行は、サーバから受け取ったかのように、アプリケーションに1度に1行返されます。

単一行モードに入るためには、PQsendQuery（または同系列の関数）の呼び出しに成功した直後にPQsetSingleRowModeを呼び出してください。このモード選択は、現在実行中の問い合わせに対してのみ有効です。その後、[項31.4](#)の説明通りに、ヌルを返すようになるまでPQgetResultを繰り返し呼び出してください。問い合わせが何らかの行を返す場合、PGRES_TUPLES_OKではなくPGRES_SINGLE_TUPLE状態コードを持つ以外通常の問い合わせ結果と同じように見える、個々のPGresultオブジェクトを返します。最後の行の後、または問い合わせがゼロ行を返す場合は即座に、PGRES_TUPLES_OK状態のゼロ行のオブジェクトが返されます。これはもう行が届かないことを通知するものです。（しかしヌルが返るまでPQgetResultを呼び出さなければならないことに注意してください。）PGresultオブジェクトのすべては、その問い合わせに対する通常のPGresultと同一の行説明データ（列名、型など）を持ちます。各オブジェクトは通常通りPQclearで解放しなければなりません。

PQsetSingleRowMode

現在実行中の問い合わせについて単一行モードを選択します。

```
int PQsetSingleRowMode(PGconn *conn);
```

この関数はPQsendQueryまたはその系列の関数のいずれかの後即座に、PQconsumeInputやPQgetResultなど接続に対する何らかの他の操作を行う前に呼び出すことができます。正しい時点で呼び出された場合、この関数は現在の問い合わせに対して単一行モードを有効にし、1を返します。この他の場合、モードは変更されず、関数はゼロを返します。いずれの場合でも、現在の問い合わせが完了した後に通常モードに戻ります。

注意

問い合わせを処理している間、サーバはいくつか行を返した後にエラーになり、問い合わせがアボートする可能性があります。通常のlibpqでは、こうした行を破棄しエラーのみを報告します。しかし単一行モードでは、これらの行はすでにアプリケーションに返されています。このためアプリケーションはPGRES_SINGLE_TUPLE状態のPGresultオブジェクトをいくつか見た後にPGRES_FATAL_ERRORオブジェクトを見るかもしれません。適切な振る舞いのトランザクションのために、最終的に問い合わせが失敗した場合、アプリケーションはこれまで処理した行を破棄するまたは取り消すように設計しなければなりません。

31.6. 処理中の問い合わせのキャンセル

本節で説明する関数を使用して、クライアントアプリケーションはサーバで処理中のコマンドをキャンセルする要求を行うことができます。

PQgetCancel

特定のデータベース接続を通して発行されたコマンドをキャンセルするために必要な情報を持つデータ構造を作成します。

```
PGcancel *PQgetCancel(PGconn *conn);
```

PQgetCancelは、与えられたPGconn接続オブジェクトのPGcancelオブジェクトを作成します。与えられたconnがNULLもしくは無効な接続であった場合、NULLが返されます。PGcancelオブジェクトは不透明な構造体であり、アプリケーションから直接アクセスすることができません。これはPQcancelもしくはPQfreeCancelに渡すことしかできません。

PQfreeCancel

PQgetCancelで作成されたデータ構造を解放します。

```
void PQfreeCancel(PGcancel *cancel);
```

PQfreeCancelは事前にPQgetCancelで作成されたデータオブジェクトを解放します。

PQcancel

サーバに現在のコマンドの廃棄処理を要求します。

```
int PQcancel(PGcancel *cancel, char *errbuf, int errbufsize);
```

キャンセル要求の受け入れが成功すれば1を、そうでなければ0を返します。失敗した場合、errbufにそれを説明するエラーメッセージが収納されます。errbufはerrbufsizeサイズの文字配列でなければなりません。(推奨サイズは256バイトです。)

しかし、要求の受け入れが成功したとしても、その要求の効果が出ることは全く保証していません。もしキャンセル操作が有効であれば、現在のコマンドは間もなく中断され、エラーが結果として返ります。キャンセル操作に失敗した場合(例えばバックエンドがすでにコマンド処理を終了していたため)、目に見える結果は何も出てこなくなります。

errbufがシグナルハンドラ内のローカル変数であれば、PQrequestCancelはシグナルハンドラから起動しても問題ありません。PQcancelの実行中、PGcancelは読み取りのみです。従って、PGconnオブジェクトを操作するスレッドと別のスレッドからこの関数を呼び出すこともできます。

PQrequestCancel

PQrequestCancelはPQcancelの廃止予定の変形版です。

```
int PQrequestCancel(PGconn *conn);
```

サーバに現在のコマンドの廃棄処理を要求します。これはPGconnオブジェクトを直接扱い、また、失敗した場合エラーメッセージはPGconnオブジェクト内に収納されます。(PQerrorMessageにより取り出すことができます。)機能的には同一ですが、PGconnのエラーメッセージが上書きされることにより、その接続で現在進行中の操作が壊れてしまうため、この方法は複数スレッドプログラムやシグナルハンドラでは問題が起こります。

31.7. 近道インタフェース

Postgresは、サーバへの簡単な関数呼び出しを送信する近道 (fast-path) インタフェースを用意しています。

ティップ: この関数はどちらかというと廃れたものです。同様の性能やそれ以上の機能を、関数呼び出しを定義したプリペアド文を設定することで達成できるからです。そして、その文をパラメータと結果をバイナリ転送するように実行すれば、近道関数呼び出しを置き換えることになります。

`PQfn`関数は近道インタフェースを使ってサーバ関数の実行を要求します。

```
PGresult *PQfn(PGconn *conn,
               int fnid,
               int *result_buf,
               int *result_len,
               int result_is_int,
               const PQArgBlock *args,
               int nargs);

typedef struct
{
    int len;
    int isint;
    union
    {
        int *ptr;
        int integer;
    } u;
} PQArgBlock;
```

`fnid`引数は実行する関数のOIDです。 `args`と`nargs`は関数に渡すパラメータを定義します。これらは関数宣言における引数リストに一致しなければなりません。パラメータ構造体の`isint`が真の場合、`u.integer`の値はサーバに指定長の整数として送信されます。(これは1、2もしくは4バイトでなければなりません。)この時、適切なバイト順の交換が行なわれます。`isint`が偽の場合は、`*u.ptr`で指定されたバイト数が無処理で送信されます。関数のパラメータデータ型をバイナリ転送で行うために、このデータはサーバで想定する書式である必要があります。`result_buf`は戻り値を格納するバッファです。呼び出し側は戻り値を格納するのに十分な領域を確保しておかなければいけません。(ライブラリ側ではこの検査はしていません!) 結果の実データ長は`result_len`が指す整数で返されます。結果が1、2、4バイト整数だと想定できるなら`result_is_int`を1に、そうでなければ0を設定します。`result_is_int`を1にすれば、必要に応じて値のバイト順を入れ換えるようlibpqに指示することになります。そしてクライアントマシン上で正しい`int`値となるように転送します。`result_is_int`が0の場合は、バックエンドが送ったバイナリ書式のバイト列を何も修正せずに返します。

`PQfn`は常に有効な`PGresult*`を返します。結果を使う前にはまず、結果ステータスを調べておくべきでしょう。結果が必要なくなった時点で、`PQclear`によって、`PGresult`を解放するのは、呼び出し側の責任です。

このインタフェースを使用した場合、NULL引数やNULL結果、セット値の結果を扱うことができないことに注意してください。

31.8. 非同期通知

PostgreSQLは、`LISTEN`と`NOTIFY`コマンドを使用した、非同期通知をサポートします。クライアントセッションは、`LISTEN`コマンドを使用して処理対象とする特定の通知チャンネルを登録します。（通知監視を取り止めるには`UNLISTEN`コマンドを使用します。）任意のセッションでそのチャンネル名による`NOTIFY`コマンドが実行されると、特定チャンネルを監視しているすべてのセッションは非同期に通知を受け取ります。監視者に追加データを通信するために"ペイロード"文字列を渡すことができます。

libpqアプリケーションは、通常のSQLによる問い合わせと同じように`LISTEN`、`UNLISTEN`および`NOTIFY`コマンドを発行することができます。`NOTIFY`メッセージの到着は、続いて`PQnotifies`を呼び出せば検出できます。

`PQnotifies`関数は、サーバから受信した通知メッセージの未処理リストから次の通知を返します。保留中の通知がなくなればヌルポインタを返します。`PQnotifies`が通知を返すと、その通知は処理済みとみなされ、通知リストから取り除かれます。

```
PGnotify *PQnotifies(PGconn *conn);

typedef struct pgNotify
{
    char *relname;           /* 通知チャンネル名 */
    int  be_pid;            /* 通知元サーバプロセスのプロセスID */
    char *extra;            /* 通知ペイロード文字列 */
} PGnotify;
```

`PQnotifies`で返された`PGnotify`オブジェクトの処理が終わったら、`PQfreemem`を使用して確実に解放してください。`PGnotify`ポインタを解放することは重要です。`relname`と`extra`フィールドは別の割り当てを表していません。（これらのフィールド名は歴史的なものです。特にチャンネル名はリレーション名と関係するものである必要はありません。）

[例31-2](#)で非同期通知を使用したサンプルプログラムを示しています。

`PQnotifies()`は実際にサーバのデータを読み出すわけではありません。これは単に、他のlibpq関数が吸収してしまっていた通知メッセージを返すだけです。libpqの以前のリリースでは、通知メッセージを適切な時点で確実に受け取るには、空の問い合わせでも何でも、とにかく一定時間ごとに問い合わせを送り、そして`PQexec()`を実行するたびに`PQnotifies()`を検査するしかありませんでした。今でもこの方法は動作しますが、処理能力の無駄使いをすることになるのでやめておくべきでしょう。

実行すべき問い合わせがない時に通知メッセージを検査するよい方法は、まず`PQconsumeInput()`を呼び出し、それから`PQnotifies()`を検査することです。サーバからのデータの到着を`select()`で待つことができ、不必要な動作でCPUパワーを消費してしまうことはありません。（`select()`で使用するファイル記述子番号の取得については、`PQsocket()`を参照してください。）なお、これは問い合わせに`PQsendQuery`と`PQgetResult`を使った時でも、またはおなじみの`PQexec`を使った時でも動作します。しかし通知がコマンドの処理中に届いていないかどうか、`PQgetResult`あるいは`PQexec`の実行ごとに`PQnotifies()`を調べることを忘れないようにしておくべきです。

31.9. COPY コマンド関連関数

PostgreSQLのCOPYコマンドでは、libpqが使っているネットワーク接続に対して読み込み、あるいは書き込みを選ぶことができます。本節で説明する関数により、アプリケーションはコピーするデータの提供やコピーされるデータの使用が可能になるという利点を持ちます。

全体的な処理として、アプリケーションはまずPQexecもしくは同等な関数経由でCOPY SQLコマンドを発行します。（コマンドでエラーが発生しなければ）この応答は、（指定したコピーの方向に応じて）PGRES_COPY_OUTもしくはPGRES_COPY_INという状態コードを持ったPGresultになります。その後、アプリケーションは本節の関数を使用して、行データを受信、もしくは、送信しなければなりません。データの転送が完了した時、転送に成功したか失敗したかを示す別のPGresultオブジェクトが返されます。その状態は、成功時にはPGRES_COMMAND_OKになり、何らかの問題が起きていた時にはPGRES_FATAL_ERRORになります。この時点で、別のSQLコマンドをPQexec経由で発行することができます。（COPY操作の実行中は、同じ接続を使用して他のSQLコマンドを実行することはできません。）

COPYコマンドが、他にもコマンドを含んだ文字列としてPQexec経由で発行された場合、アプリケーションはCOPY処理を終えた後に、PQgetResult経由で結果の取り出しを続けなければなりません。PQexecコマンド文字列が完了し、その後のコマンドが安全に発行できることが確実になるのは、PQgetResultがNULLを返す時のみです。

本節の関数は、PQexecもしくはPQgetResultからPGRES_COPY_OUTもしくはPGRES_COPY_INという結果状態を得た後のみに実行されなければなりません。

これらの状態値の一つを持つPGresultオブジェクトは、開始したCOPY操作に関する追加データを持ちます。この追加データは、以下の問い合わせ結果を持つ接続で使用される関数を使用して利用することができます。

PQnfields

コピーされる列(フィールド)数を返します。

PQbinaryTuples

0は、コピー全体の書式がテキスト(改行で区切られた行、区切り文字で区切られた列など)であることを示します。
1は、コピー全体の書式がバイナリであることを示します。詳細は[COPY](#)を参照してください。

PQfformat

コピー操作対象の列それぞれに関した書式コード(テキストでは0、バイナリでは1)を返します。コピー全体の書式がテキストの場合は、列単位の書式コードは常にゼロです。しかし、バイナリ書式はテキスト列もバイナリ列もサポートすることができます。（しかし、現在のCOPY実装では、バイナリコピーでのみバイナリ列が発生します。そのため、今の所列単位の書式は常に全体の書式と一致します。）

注意: これらの追加データ値はプロトコル3.0を使用した場合にのみ利用可能です。プロトコル2.0を使用する場合は、これらの関数はすべて0を返します。

31.9.1. COPY データ送信用関数

これらの関数は、COPY FROM STDIN期間にデータを送信するために使用されます。接続がCOPY_IN状態でない時に呼び出された場合、これらは失敗します。

PQputCopyData

COPY_IN状態の間、サーバにデータを送信します。

```
int PQputCopyData(PGconn *conn,
                  const char *buffer,
                  int nbytes);
```

指定したbufferにあるCOPYデータをnbytes長分、サーバに送信します。データが送信された場合、この結果は1になります。送信試行がブロックされたために送信できなかった場合はゼロになります。（これは、接続が非ブロックモードの場合にのみ起こります。）エラーが発生した場合は-1になります。（戻り値が-1の場合、詳細を取り出すためにはPQerrorMessageを使用してください。戻り値がゼロの場合は書き込み準備が整うまで待ち、再実行してください。）

アプリケーションはCOPYデータストリームを使いやすい大きさのバッファに分けて読み込むことができます。送信時の読み込みバッファの境界には意味的な重要性はありません。データストリームの内容は、COPYコマンドで想定しているデータ書式に一致している必要があります。詳細は[COPY](#)を参照してください。

PQputCopyEnd

COPY_IN状態の間に、サーバにデータ終了指示を送信します。

```
int PQputCopyEnd(PGconn *conn,
                 const char *errmsg);
```

errmsgがNULLの場合は、COPY_IN操作の終了に成功しました。errmsgがNULLでない場合は、COPYは強制的に失敗されました。errmsgが指し示す文字列はエラーメッセージとして使用されます。（しかし、このエラーメッセージが正しくサーバから返ったものであるとは仮定すべきではありません。サーバは既に別の原因でCOPYに失敗していた可能性があります。また、この強制的な失敗は3.0より前のプロトコルの接続を使用している場合は動作しません。）

終端データが送信された場合は結果は1になります。送信試行がブロックされたため送信できなかった場合はゼロになります。（これは、接続が非ブロックモードの場合にのみ起こります。）エラーが発生した場合は-1になります。（戻り値が-1の場合、詳細を取り出すためにはPQerrorMessageを使用してください。戻り値がゼロの場合は書き込み準備が整うまで待ち、再実行してください。）

PQputCopyEndの呼び出しに成功した後、PQgetResultを呼び出してCOPYコマンドの最終的な結果状態を取り出してください。通常の方法でこの結果が使用できるようになるまで待機しても構いません。そして、通常の操作に戻ってください。

31.9.2. COPYデータ受信関数

これらの関数はCOPY TO STDOUT時にデータを受信するために使用されます。COPY_OUT状態以外の接続で呼び出すと、失敗します。

PQgetCopyData

COPY_OUT状態時にサーバからデータを受信します。

```
int PQgetCopyData(PGconn *conn,
                  char **buffer,
                  int async);
```

COPY期間中、サーバから別の行データの入手を試みます。常に1度に1つの行データが返されます。部分的な行のみが利用可能な場合は返されません。行データの取得に成功することは、そのデータを保持するためのメモリチャンクの割り当てを意味します。bufferパラメータは非NULLでなければなりません。*bufferは割り当てられたメモリへのポインタに、バッファが返されなかった場合はNULLに設定されます。非NULLの結果バッファは、不要になったらPQfreememを使用して解放しなければなりません。

行の取り込みに成功した時、戻り値は行内のデータのバイト数になります。（これは常に0より大きくなります。）返された文字列は常にヌル終端ですが、おそらくテキストCOPYでのみ有用になるでしょう。ゼロという結果は、COPYが進行中で、行がまだ利用できない状態であることを示します。（asyncが真の場合にのみ発生することがあります。）-1という結果は、COPYが完了したことを示します。-2という結果はエラーが発生したことを示します。（その理由についてはPQerrorMessageを参照してください。）

asyncが真(非0)の場合、PQgetCopyDataは入力待ちのためのブロックを行いません。COPY実行中で完全な行を取り出せない場合PQgetCopyDataは0を返します。（この場合、再試行の前に読み込み準備が整うまで待機してください。

PQconsumeInputを呼び出したかどうかは関係ありません。）asyncが偽(0)の場合、PQgetCopyDataはデータが利用できるようになるまで、もしくは、操作が完了するまでブロックします。

PQgetCopyDataが-1を返した後、PQgetResultを呼び出して、COPYコマンドの最終結果状態を取り出してください。通常の方法で結果が利用できるようになるまで待機しても構いません。そして、通常の操作に戻ってください。

31.9.3. 廃れたCOPY用関数

以下の関数はCOPYを取扱う、古めの手法を行います。これらはまだ動作しますが、エラーの取扱いが貧弱であることやデータの終端を検知する方法が不便であることより使用を奨めません。

PQgetline

改行で終端する文字列（サーバから送信されたもの）を長さlengthのバッファ用文字列に読み込みます。

```
int PQgetline(PGconn *conn,
              char *buffer,
              int length);
```

この関数はバッファにlength-1個までの文字をコピーし、終端の改行を1バイトのゼロに置き換えます。PQgetlineは、入力の終端ではEOFを、行全体が読み込まれれば0を返します。そしてまだ終端の改行が読み込まれていないうちにバッファがいっぱいになってしまった場合は1を返します。

アプリケーションは新しく読み込んだ行が、`¥.` という2文字であるかどうか確認しなければいけません。この2文字は、`COPY` コマンドの結果をサーバが送信し終えたことを示すものです。アプリケーションには、仮に`length-1`文字より長い行を受け取るようなことがあっても、間違いなく`¥.`行を認識するような配慮が必要です（また例えば長いデータの行の終端を、最終行と取り違えないようにもしてください）。

PQgetlineAsync

`COPY` データ行（サーバから送信されたもの）を、ブロッキングなしでバッファに読み込みます。

```
int PQgetlineAsync(PGconn *conn,
                  char *buffer,
                  int bufsize);
```

`PQgetline` と似ていますが、`COPY` のデータを非同期的に、つまりブロッキングなしで読み出さなければならないアプリケーションで使用することができます。`COPY` コマンドを発行し、そして `PGRES_COPY_OUT` 応答を受け取ったら、アプリケーションはデータ終了の合図を受け取るまで `PQconsumeInput` と `PQgetlineAsync` を呼び出します。

`PQgetline` と違い、この関数はデータ終了の検出に対して責任を持ちます。

`PQgetlineAsync` の個々の呼び出しでは、`libpq` の入力バッファ内で完全な行データが利用できる場合にデータを返します。さもなければ、行の残りが届くまでデータは返されません。この関数は、コピーデータの終端を示す符号を認識すると-1を、また何もデータがなければ0を、そしてデータを返す場合はそのバイト数を正の値で返します。もし-1が返されたら、呼び出し側は次に `PQendcopy` を呼び出さなければいけません。それから通常の処理に戻ります。

返されるデータは行データの境界を越えて拡張されることはありません。可能であれば行全体を一度に返します。しかし呼び出し側が準備したバッファが少なすぎ、サーバから送られてくる行を保持しておくことができない場合には、分割された行データを返します。テキストデータでは、これは最後の1バイトが`¥n`かどうかを確認すれば検出できます。（バイナリ`COPY`の場合に同様の検出を行うためには、実際に`COPY`データの書式を解析しなければなりません。）なお、返される文字列はヌル終端ではありません。（ヌル終端を後から付け加えるのであれば、実際に確保するバッファサイズ-1を`bufsize`として渡すようにしてください。）

PQputline

サーバにヌル終端の文字列を送信します。問題なければ0を返します。文字列の送信ができなかった場合は`EOF`を返します。

```
int PQputline(PGconn *conn,
              const char *string);
```

`PQputline` の呼び出しによって送信される`COPY` データストリームは、`PQgetlineAsync` で返される書式と同じ書式を持ちます。ただし、アプリケーションは、`PQputline` 毎に正確に1つのデータ行を送信するように強制されていません。呼び出し毎に行の一部や複数の行を送信しても問題ありません。

注意: PostgreSQLプロトコル3.0より前では、アプリケーションは、サーバに対して`COPY`データの送信を完了したことを通知するために、最終の行として`¥.` という2文字を明示的に送信する必要がありました。これはまだ動作します。しかし、これは廃れたものとして、`¥.` の特殊な意味は将来のリリースで無くなることが予想されます。実際のデータの送信完了後に `PQendcopy` を呼び出すことが重要です。

PQputnbytes

ヌル終端ではない文字列をサーバに送信します。問題なければ0を返します。文字列の送信ができなかった場合は`EOF`を返します。

```
int PQputnbytes(PGconn *conn,
                const char *buffer,
                int nbytes);
```

これはまさに `PQputline` と同様です。ただし、直接送信バイト数を指定するため、ヌル終端である必要がありません。バイナリデータを送信する時はこのプロシージャを使用してください。

PQendcopy

サーバと同期します。

```
int PQendcopy(PGconn *conn);
```

この関数はサーバがコピーを完了するのを待ちます。この関数は、`PQputline` を使ったサーバへの文字列送信が完了した時点、あるいは `PGgetline` を使ったサーバからの文字列受信が完了した時点のいずれでも呼び出さなければなりません。これを発行しないと、サーバはクライアントとの"同期がずれた"状態になってしまいます。この関数から戻った時点で、サーバは次のSQLコマンドを受ける準備が整います。正常に終了した場合、返り値は0です。さもなくば、非ゼロです。（戻り値が非ゼロの場合、`PQerrorMessage` を使用して詳細を取り出してください。）

`PQgetResult` を使う場合、アプリケーションは `PQgetline` を繰り返し呼び出して `PGRES_COPY_OUT` に応答し、終端行を見つけ

たら続いてPQendcopyを呼び出さなければなりません。それから、PQgetResultがヌルポインタを返すまで、PQgetResultのループに戻らなければなりません。同じように PGRES_COPY_INは連続したPQputlineで処理し、それからPQendcopyで締めくくった後にPQgetResultのループに戻ります。このようにすることで、一連のSQLコマンド群に含めたCOPYコマンドを確実に、また正しく実行できるはずです。

比較的古いアプリケーションでは、COPYをPQexecで実行し、PQendcopyの実行でトランザクションは完了する、と想定していることがよくあります。これはコマンド文字列中のSQLがCOPYだけであった時にのみ正しく動作します。

[前のページ](#)
非同期通知

[ホーム](#)
[上に戻る](#)

[次のページ](#)
制御関数

31.10. 制御関数

これらの関数はlibpqの動作の各種詳細を制御します。

PQclientEncoding

クライアント符号化方式を返します。

```
int PQclientEncoding(const PGconn *conn);
```

これがEUC_JPなどのシンボル文字列ではなく符号化方式IDを返すことに注意してください。符号化方式IDを符号化方式名に変換するためには以下を使用してください。

```
char *pg_encoding_to_char(int encoding_id);
```

PQsetClientEncoding

クライアント符号化方式を設定します。

```
int PQsetClientEncoding(PGconn *conn, const char *encoding);
```

connはサーバへの接続、encodingは使用したい符号化方式です。この関数は符号化方式の設定に成功すると、ゼロを返します。さもなければ-1を返します。この接続における現在の符号化方式はPQclientEncodingを使用して決定することができます。

PQsetErrorVerbosity

PQerrorMessageとPQresultErrorMessageで返されるメッセージの冗長度を決定します。

```
typedef enum
{
    PQERRORS_TERSE,
    PQERRORS_DEFAULT,
    PQERRORS_VERBOSE
} PGVerbosity;

PGVerbosity PQsetErrorVerbosity(PGconn *conn, PGVerbosity verbosity);
```

PQsetErrorVerbosityは冗長度モードを設定し、接続における以前の状態を返します。TERSEモードでは、返されるメッセージには深刻度、主テキスト、位置のみが含まれます。これは通常単一行に収まります。デフォルトモードでは、上に加え、詳細、ヒント、文脈フィールドが含まれるメッセージが生成されます。（これは複数行に跨るかもしれません。）VERBOSEモードでは、すべての利用可能なフィールドが含まれます。冗長度の変更は、既に存在するPGresultオブジェクト内から取り出せるメッセージには影響を与えません。その後に作成されたオブジェクトにのみ影響を与えます。

PQtrace

クライアント／サーバ間の通信トレースを有効にし、デバッグ用のファイルストリームに書き出します。

```
void PQtrace(PGconn *conn, FILE *stream);
```

注意: Windowsにおいて、libpqライブラリとアプリケーションを異なるフラグでコンパイルすると、この関数呼び出しでFILEポインタの内部表現の違いによりアプリケーションはクラッシュするでしょう。特に、このライブラリを使用するアプリケーションでは、マルチスレッド/シングルスレッド、リリース/デバッグ、静的リンク/動的リンクに関して、ライブラリと同じフラグを使わなければなりません。

PQuntrace

PQtraceによって起動されたトレース処理を無効にします。

```
void PQuntrace(PGconn *conn);
```

31.11. 雑多な関数

よくあることですが、うまく分類できない関数がいくつか存在します。

PQfreemem

libpqが割り当てたメモリを解放します。

```
void PQfreemem(void *ptr);
```

具体的にはPQescapeByteaConn、PQescapeBytea、PQunescapeByteaおよびPQnotifiesによりlibpqが割り当てたメモリを解放します。Microsoft Windowsにおいてfree()ではなく、この関数を使用することが特に重要です。DLLにおけるメモリ割り当てとアプリケーションにおけるその解放が、DLLとアプリケーションとでマルチスレッド/シングルスレッド、リリース用/デバッグ用、静的/動的フラグが同じ場合でのみ動作するためです。Microsoft Windowsプラットフォーム以外では、この関数は標準ライブラリのfree()関数と同じです。

PQconninfoFree

PQconnndefaultsもしくはPQconninfoParseが割り当てたデータ構造を解放します。

```
void PQconninfoFree(PQconninfoOption *connOptions);
```

単純なPQfreememは、配列が補助文字列への参照を含んでいることから、このためには作業しません。

PQencryptPassword

PostgreSQLパスワードの暗号化された形式を準備します。

```
char * PQencryptPassword(const char *passwd, const char *user);
```

この関数は、ALTER USER joe PASSWORD 'pwd'のようなコマンドを送信したいクライアントアプリケーションで使用されることを意図したものです。こうしたコマンドでは、コマンドログが活動の監視などで晒されてしまうため、元々の平文テキストでパスワードを送信しないことが推奨されています。その代わりに、この関数を使用して送信前にパスワードを暗号化形式に変換してください。引数は平文のパスワードとそのSQL上のユーザ名です。戻り値はmallocで割り当てられた文字列です。メモリ不足の場合にNULLが返されます。呼び出し元は、その文字列にエスケープしなければならない特殊な文字列が含まれていないことを仮定することができます。処理が終わった時にPQfreememを使用して結果を解放してください。

PQmakeEmptyPGresult

与えられたステータスで空のPGresultオブジェクトを構築します。

```
PGresult *PQmakeEmptyPGresult(PGconn *conn, ExecStatusType status);
```

これは空のPGresultオブジェクトを割り当てて、初期化するlibpqの内部関数です。メモリが割り当てられなかった場合、この関数はNULLを返します。一部のアプリケーションで結果オブジェクト（特にエラーステータスを伴ったオブジェクト）それ自身を生成することが便利であることが分かりましたので、外部公開されました。connが非ヌルで、statusがエラーを示唆している場合、特定された接続の現在のエラーメッセージはPGresultにコピーされます。同時に、connが非ヌルの場合、接続で登録された任意のイベントプロシージャはPGresultにコピーされます。（それらはPGEVT_RESULTCREATE呼び出しを受けませんが、PQfireResultCreateEventsを理解します。）libpq自身で返されたPGresultと同様に、最終的にはこのオブジェクトに対してPQclearを呼び出さなければならないことに注意してください。

PQfireResultCreateEvents

PGresultオブジェクトに登録されたそれぞれのイベントプロシージャに対し、PGEVT_RESULTCREATEイベント（[項31.13](#)を参照）を発行します。イベントプロシージャが成功の場合は非ゼロ、失敗の場合はゼロを返します。

```
int PQfireResultCreateEvents(PGconn *conn, PGresult *res);
```

conn引数はイベントプロシージャに渡されますが、直接には使用されません。イベントプロシージャが使用しない場合はNULLです。

このオブジェクトに対し、PGEVT_RESULTCREATEもしくはPGEVT_RESULTCOPYイベントを過去に受け取ったイベントプロシージャは再び発行されません。

この関数がPQmakeEmptyPGresultと分離されている主たる理由は、多くの場合イベントプロシージャを呼び出す前にPGresultを作成し、データを挿入するのが適切であることによります。

PQcopyResult

`PGresult` オブジェクトのコピーを作ります。 コピーは元の結果にいかなる方法でもリンクされず、コピーが不要になった時に `PQclear` を呼び出されなければなりません。 関数が失敗すると `NULL` が返されます。

```
PGresult *PQcopyResult(const PGresult *src, int flags);
```

これは正確なコピーの作成を目的としたものではありません。 返された結果は常に `PGRES_TUPLES_OK` 状態の中に置かれ、元の結果におけるエラーメッセージはまったくコピーされません。 (しかしコマンド状態文字列をコピーします。) `flags` 引数はその他にコピーするものがないかを決定します。 それはいくつかのフラグのビット単位のORです。 `PG_COPYRES_ATTRS` は元の結果の属性 (列定義) のコピーを指定します。 `PG_COPYRES_TUPLES` は元の結果のタプルのコピーを指定します。 (これは属性もコピーされることを意味しています。) `PG_COPYRES_NOTICEHOOKS` は元の結果の警告フックのコピーを指定します。 `PG_COPYRES_EVENTS` は元の結果イベントのコピーを指定します。 (しかし、元の結果に関連したインスタンスデータはまったくコピーされません。)

PQsetResultAttrs

`PGresult` オブジェクトの属性を設定します。

```
int PQsetResultAttrs(PGresult *res, int numAttributes, PGresAttDesc *attDescs);
```

提供された `attDescs` は結果にコピーされます。 もし `attDescs` ポインタが `NULL`、または `numAttributes` が 1 未満の場合、要求は無視され、関数は成功します。 `res` が既に属性を所有している場合、関数は失敗に終わります。 関数が失敗すると、戻り値はゼロです。 関数が成功すると戻り値は非ゼロになります。

PQsetvalue

`PGresult` オブジェクトのタブルフィールド値を設定します。

```
int PQsetvalue(PGresult *res, int tup_num, int field_num, char *value, int len);
```

必要に応じて関数は自動的に結果の内部タブル配列を肥大化させます。 しかし、`tup_num` 引数は `PQntuples` と同じか、もしくは小さくなければなりません。 その意味は、この関数は一回にタブル配列を 1 タブル大きくさせるだけだからです。 とは言っても、存在するいかなるタブルの任意のフィールドも、順序を問わず変更できます。 もし `field_num` に値が既に存在すれば、書き換えられます。 `len` が -1、または `value` が `NULL` であれば、フィールドの値はSQLの `NULL` に設定されます。 `value` は結果のプライベート格納領域にコピーされるため、関数が返った後ではもう必要がなくなります。 関数が失敗すると、戻り値はゼロです。 関数が成功すると戻り値は非ゼロになります。

PQresultAlloc

`PGresult` オブジェクトに補助ストレージを割り当てます。

```
void *PQresultAlloc(PGresult *res, size_t nBytes);
```

`res` が消去された時、この関数で割り付けられたメモリは解放されます。 関数が失敗すると戻り値は `NULL` です。 `malloc` と同じように、どのような種類のデータでも結果は適切に整列されることが保証されています。

PQlibVersion

使用中の `libpq` のバージョンを返します。

```
int PQlibVersion(void);
```

この関数の結果を使用して、実行時に現在読み込まれているバージョンの `libpq` で特定の機能が利用可能かどうかを決定することができます。 例えばこの関数を使用して、`PQconnectdb` でどの接続オプションが利用できるか、PostgreSQL 9.0 で追加された `hex bytea` 出力をサポートするかを確認することができます。

この数値の形式は、メジャー、マイナー、リビジョン番号を2桁の10進数に変換し、連結させたものです。 例えば、バージョン9.1では90100を返し、バージョン9.1.2では90102を返します。 (先頭の0は現れません。)

注意: この関数はPostgreSQLバージョン9.1で追加されました。 このため以前のバージョンにおいて要求される機能を検知するために使用することができません。 この関数へのリンク処理がバージョン9.1とのリンク依存性を作成するためです。

[前のページ](#)
制御関数

[ホーム](#)
[上に戻る](#)

[次のページ](#)
警告処理

31.12. 警告処理

問い合わせ実行関数では、サーバにより生成された通知と警告メッセージは、問い合わせの失敗を意味していないので返されません。その代わり、それらは通知処理関数に渡され、ハンドラから返った後も継続して実行されます。デフォルトの通知処理関数は `stderr` にメッセージを出力しますが、アプリケーションは自身の処理関数を提供することでこの動作を書き換えることができます。

歴史的理由で、通知レシーバと通知プロセッサと呼ばれる2階層の通知処理が存在します。デフォルトの動作は、通知レシーバが通知を書式化し、出力のため通知プロセッサに文字列を渡します。しかし、独自の通知レシーバを提供することを選んだアプリケーションでは、通常、通知プロセッサ層を無視し、すべての作業を単に通知レシーバで行います。

関数 `PQsetNoticeReceiver` は接続オブジェクトに対し現在の通知レシーバを設定もしくは確認します。同様に、`PQsetNoticeProcessor` は現在の通知プロセッサの設定もしくは確認を行います。

```
typedef void (*PQnoticeReceiver) (void *arg, const PGresult *res);

PQnoticeReceiver
PQsetNoticeReceiver(PGconn *conn,
                   PQnoticeReceiver proc,
                   void *arg);

typedef void (*PQnoticeProcessor) (void *arg, const char *message);

PQnoticeProcessor
PQsetNoticeProcessor(PGconn *conn,
                    PQnoticeProcessor proc,
                    void *arg);
```

各関数は、以前の通知レシーバもしくは通知プロセッサ用の関数へのポインタを返し、新しい値を設定します。関数ポインタにヌルを渡した場合、何も変更されず、現在のポインタが返されるだけです。

サーバから注意/警告メッセージを受け取ると、あるいは、libpq内部で注意/警告メッセージが生成されると、通知レシーバ関数が呼び出されます。 `PGRES_NONFATAL_ERROR PGresult` という形でメッセージが渡されます。（これにより、レシーバは `PQresultErrorField` を使用して個々のフィールドを取り出すことや、`PQresultErrorMessage` を使用して事前に整形された完全なメッセージを取り出すことができます。） `PQsetNoticeReceiver` に渡された `void` ポインタと同じものも渡されます。（このポインタを使用して、必要に応じてアプリケーション特有の状態にアクセスすることができます。）

デフォルトの通知レシーバは単に（`PQresultErrorMessage` を使用して）メッセージを取り出し、それを通知プロセッサに渡すだけです。

通知プロセッサは、テキスト形式で与えられた注意/警告メッセージの取扱いに責任を持ちます。メッセージは（最後改行を含む）文字列テキストで渡され、更に、`PQsetNoticeProcessor` に渡したものと同一 `void` ポインタも渡されます。（このポインタを使用して、必要に応じてアプリケーション特有の状態にアクセスすることができます。）

デフォルトの通知プロセッサは以下のような単純なものです。

```
static void
defaultNoticeProcessor(void *arg, const char *message)
{
    fprintf(stderr, "%s", message);
}
```

一旦通知レシーバや通知プロセッサを設定したら、`PGconn` オブジェクトか、それから生成された `PGresult` オブジェクトが存在している間は、その関数が呼び出される可能性があると考えておくべきです。`PGresult` の生成時には、`PGconn` の現在の警告処理用のポインタが、`PQgetvalue` のような関数で使用可能であるように、`PGresult` へコピーされます。

31.13. イベントシステム

libpqのイベントシステムは、PGconnおよびPGresultオブジェクトの作成と削除のような関心を引くlibpqイベントについて登録されたイベントハンドラに通知を行うため設計されています。主たる使用状況は、アプリケーションがそれ自身のデータをPGconnまたはPGresultと提携させ、データが適切な時間に解放されることを保証するものです。

それぞれの登録されたイベントハンドラは、libpqからは曖昧としたvoid *ポインタとしてだけ知られる2つのデータの断片と提携します。イベントハンドラがPGconnで登録された時にアプリケーションが提供する通過地点ポインタがあります。通過地点ポインタはPGconnやそれから生成されたすべての（複数の）PGresultが有効な間決して変わることはありません。したがって使用された場合、長期間生存しているデータを指し示します。さらに、インスタンスデータポインタがあつて、それはすべてのPGconnとPGresultでNULLから開始します。ポインタは、PQinstanceData、PQsetInstanceData、PQresultInstanceDataおよびPQsetResultInstanceData関数を使って操作することができます。通過地点ポインタとは異なり、PGconnのインスタンスデータはそれから作成されたPGresultにより自動的に継承されません。libpqは通過地点とインスタンスデータポインタが（もしあったとしても）何を指し示すのか判らず、決して解放しようとは試みません。それはイベントハンドラの責任です。

31.13.1. イベントの種類

PGEvtId列挙はイベントシステムにより処理されるイベントの種類に名前をつけます。その値はすべてPGEVTで始まる名前を持っています。それぞれのイベントの種類に対し、イベントハンドラに渡されるパラメータを運ぶ関連したイベント情報構造体があります。イベントの種類を以下に示します。

PGEVT_REGISTER

登録イベントはPQregisterEventProcが呼ばれたとき発生します。イベントプロシージャが必要とするかもしれない任意のinstanceDataを初期化するために、これは理想的な時間です。接続毎、イベントハンドラ毎でたった1つの登録イベントが発行されます。イベントプロシージャが失敗すると、登録は中止されます。

```
typedef struct
{
    PGconn *conn;
} PGEvtRegister;
```

PGEVT_REGISTERイベントが受け取られると、evtInfoポインタはPGEvtRegister *にキャストされなければなりません。この構造体はCONNECTION_OK状態ではなくてはならないPGconnを含んでいます。そしてそれは、効果のあるPGconnを取得した直後、PQregisterEventProcを呼び出せば、保証されます。失敗コードを返すとき、PGEVT_CONNDESTROYイベントが送られないので、すべての消去が実行されなければなりません。

PGEVT_CONNRESET

接続初期化イベントはPQresetまたはPQresetPollの完了時点で発行されます。どちらの場合も、初期化が成功したときのみ発行されます。イベントプロシージャが失敗すると、接続初期化全体が失敗します。PGconnはCONNECTION_BAD状態になり、PQresetPollはPGRES_POLLING_FAILEDを返します。

```
typedef struct
{
    PGconn *conn;
} PGEvtConnReset;
```

PGEVT_CONNRESETイベントが受け取られた時、evtInfoポインタはPGEvtConnReset *にキャストされなければなりません。含まれたPGconnは単に初期化されますが、すべてのイベントデータは変更されずに残ります。このイベントはすべての関連したinstanceDataの初期化・再読み込み・再問い合わせに使用されなければなりません。イベントプロシージャがPGEVT_CONNRESET処理に失敗したとしても、接続が閉じられた時PGEVT_CONNDESTROYイベントを依然として受け付けることに注意してください。

PGEVT_CONNDESTROY

接続破棄イベントはPQfinishに対応して発行されます。libpqはこのメモリを管理する機能がありませんので、そのイベントデータを的確に消去するのはイベントプロシージャの責任です。消去の失敗はメモリーリークに通じます。

```
typedef struct
{
    PGconn *conn;
} PGEvtConnDestroy;
```

PGEVT_CONNDESTROYイベントが受け取られた時、evtInfoポインタはPGEvtConnDestroy *にキャストされなければなりません。このイベントはPQfinishが他のすべての消去を行う前に発行されます。イベントプロシージャの戻り値は、PQfinishから失敗を示唆する方法がないので無視されます。同時に、イベントプロシージャの失敗が不要なメモリ消

去処理を中止してはなりません。

PGEVT_RESULTCREATE

結果作成イベントは、PQgetResultを含み、結果を生成する任意の問い合わせ実行関数に対応して発行されます。このイベントは結果が成功裏に作成されたときのみ発行されます。

```
typedef struct
{
    PGconn *conn;
    PGresult *result;
} PGEventResultCreate;
```

PGEVT_RESULTCREATE イベントが受け取られた時、evtInfoポインタはPGEventResultCreate *にキャストされなければなりません。connは結果を生成するために使われた接続です。これは、結果と関連しなければならないすべてのinstanceDataを初期化するために、理想的な場所です。イベントプロシージャが失敗すると、結果は消去され、失敗が伝播します。イベントプロシージャはそれ自身の結果オブジェクトをPQclearしようと試みてはいけません。失敗コードを返す時、PGEVT_RESULTDESTROY イベントは送られないのですべての消去が行われなくてはなりません。

PGEVT_RESULTCOPY

結果コピーイベントはPQcopyResultの応答として発行されます。このイベントはコピーが完了した後にのみ発行されます。元の結果に対するPGEVT_RESULTCREATEもしくはPGEVT_RESULTCOPY イベントを成功裏に処理したイベントプロシージャのみ、PGEVT_RESULTCOPY イベントを受け取ります。

```
typedef struct
{
    const PGresult *src;
    PGresult *dest;
} PGEventResultCopy;
```

PGEVT_RESULTCOPY イベントが受け取られた時、evtInfoポインタはPGEventResultCopy *にキャストされなければなりません。src結果はコピーされるものであり、一方でdest結果はコピー先です。このイベントはinstanceDataのディープコピーを提供するために使用されます。PQcopyResultではこれを行うことができないためです。もしイベントプロシージャが失敗すると、コピー操作全体は失敗になり、dest結果は消去されます。失敗コードを返す時、PGEVT_RESULTDESTROY イベントがコピー先の結果に対し送られないため、すべての消去を行われなければなりません。

PGEVT_RESULTDESTROY

結果破棄イベントはPQclearに対応して発行されます。libpqはこのメモリを管理する機能がありませんので、そのイベントデータを的確に消去するのはイベントプロシージャの責任です。消去の失敗はメモリーリークに通じます。

```
typedef struct
{
    PGresult *result;
} PGEventResultDestroy;
```

PGEVT_RESULTDESTROY が受け取られた時、evtInfoポインタはPGEventResultDestroy *にキャストされなければなりません。このイベントはPQclearがその他の消去を行う以前に起動されなければなりません。イベントプロシージャの戻り値は、PQclearから失敗を示唆する方法がないので無視されます。同時に、イベントプロシージャの失敗が不要なメモリ消去処理を中止してはなりません。

31.13.2. イベントコールバックプロシージャ

PGEvtProc

PGEvtProcはイベントプロシージャへのポインタに対するtypedefです。つまり、libpqからイベントを受け取るユーザーコールバック関数です。イベントプロシージャのシグネチャは以下でなければなりません。

```
int eventproc(PGEvtId evtId, void *evtInfo, void *passThrough)
```

evtIdパラメータはどのevtIdイベントが発生したかを示します。evtInfoポインタは、イベントに対する追加情報を入手するため適切な構造体型にキャストされなければなりません。passThroughパラメータは、イベントプロシージャが登録された時、PQregisterEventProcに提供されるポインタです。関数は成功した場合非ゼロを、失敗した場合ゼロを返さなければなりません。

特定のイベントプロシージャは任意のPGconnにおいても一回だけ登録することができます。これは、プロシージャのアドレスが関連するインスタンスデータを特定する検索キーとして用いられるからです。

注意

Windowsにおいて、関数は2つの異なるアドレスを持つことができます。外部から可視のDLLと内部から可視のDLLです。libpqのイベントプロシージャ関数ではこれらのアドレスのうちの1つだけが使用されることに注意してください。さもないと、混乱が起きます。正常に機能するコードを書く最も単純な規則は、イベントプロシージャがstaticとして宣言されることを確実にすることです。もし、プロシージャのアドレスがそれ自身のファイルの外部から有効とならなければならない場合、アドレスを返すため別の関数を公開します。

31.13.3. イベントサポート関数

PQregisterEventProc

libpqでイベントコールバックプロシージャを登録します。

```
int PQregisterEventProc(PGconn *conn, PGEventProc proc,
                      const char *name, void *passThrough);
```

そのイベントを取得したいそれぞれのPGconnで1回イベントプロシージャは登録されなければなりません。接続で登録することができるイベントプロシージャの数には、メモリ以外の制限はありません。関数は成功した場合非ゼロ、失敗の場合ゼロを返します。

libpqイベントが発行されたときproc引数が呼ばれます。そのメモリアドレスはinstanceDataを検索するのにも使用されます。name引数はエラーメッセージ内でイベントプロシージャを参照するために使用されます。この値はNULLもしくは空文字列であってはなりません。このname文字列はPGconnにコピーされますので、渡されたものは長寿命である必要があります。passThroughポインタはイベントが発生した時はいつでもprocに渡されます。この引数はNULLであっても構いません。

PQsetInstanceData

procプロシージャに対するconn接続のinstanceDataをdataに設定します。成功の場合非ゼロ、失敗の場合ゼロが返ります。（connでprocが正しく登録されていない場合のみ失敗する可能性があります。）

```
int PQsetInstanceData(PGconn *conn, PGEventProc proc, void *data);
```

PQinstanceData

procプロシージャに関連したconn接続のinstanceData、または存在しなければNULLを返します。

```
void *PQinstanceData(const PGconn *conn, PGEventProc proc);
```

PQresultSetInstanceData

procに対する結果のinstanceDataをdataに設定します。成功の場合非ゼロ、失敗の場合ゼロが返ります。（結果でproc正しく登録されていない場合のみ失敗する可能性があります。）

```
int PQresultSetInstanceData(PGresult *res, PGEventProc proc, void *data);
```

PQresultInstanceData

procに関連した結果のinstanceData、または存在しなければNULLを返します。

```
void *PQresultInstanceData(const PGresult *res, PGEventProc proc);
```

31.13.4. イベント事例

以下にlibpq接続と結果に関連したプライベートデータを管理する例の大枠を示します。

```
/* libpqイベントに必要なヘッダ（覚書：libpq-fe.hのインクルード） */
#include <libpq-events.h>
/* instanceData */
typedef struct
{
    int n;
    char *str;
} mydata;

/* PGEventProc */
static int myEventProc(PGEventId evtId, void *evtInfo, void *passThrough);

int
main(void)
{
    mydata *data;
    PGresult *res;
    PGconn *conn = PQconnectdb("dbname = postgres");
```

```

if (PQstatus(conn) != CONNECTION_OK)
{
    fprintf(stderr, "Connection to database failed: %s",
        PQerrorMessage(conn));
    PQfinish(conn);
    return 1;
}
/* イベントを受け取るべきすべての接続で1回呼ばれる
 * myEventProcにPGEVT_REGISTERを送る
 */
if (!PQregisterEventProc(conn, myEventProc, "mydata_proc", NULL))
{
    fprintf(stderr, "Cannot register PGEvtProc¥n");
    PQfinish(conn);
    return 1;
}

/* conn instanceDataが有効 */
data = PQinstanceData(conn, myEventProc);
/* myEventProcにPGEVT_RESULTCREATEを送る */
res = PQexec(conn, "SELECT 1 + 1");
/* 結果 instanceDataが有効 */
data = PQresultInstanceData(res, myEventProc);
/* PG_COPYRES_EVENTSが使われた場合、PGEVT_RESULTCOPYをmyEventProcに送る */
res_copy = PQcopyResult(res, PG_COPYRES_TUPLES | PG_COPYRES_EVENTS);

/* PQcopyResult呼び出しの過程でPG_COPYRES_EVENTSが使用された場合、
 * 結果 instanceDataが有効
 */
data = PQresultInstanceData(res_copy, myEventProc);
/* 双方のclearがPGEVT_RESULTDESTROYをmyEventProcに送る */
PQclear(res);
PQclear(res_copy);

/* PGEVT_CONNDESTROYをmyEventProcに送る */
PQfinish(conn);

return 0;
}

static int
myEventProc(PGEventId evtId, void *evtInfo, void *passThrough)
{
    switch (evtId)
    {
        case PGEVT_REGISTER:
        {
            PGEventRegister *e = (PGEventRegister *)evtInfo;
            mydata *data = get_mydata(e->conn);

            /* アプリ特有のデータを接続に関連付ける */
            PQsetInstanceData(e->conn, myEventProc, data);
            break;
        }

        case PGEVT_CONNRESET:
        {
            PGEventConnReset *e = (PGEventConnReset *)evtInfo;
            mydata *data = PQinstanceData(e->conn, myEventProc);

            if (data)
                memset(data, 0, sizeof(mydata));
            break;
        }

        case PGEVT_CONNDESTROY:
        {
            PGEventConnDestroy *e = (PGEventConnDestroy *)evtInfo;
            mydata *data = PQinstanceData(e->conn, myEventProc);

            /* connが破棄されたのでインスタンスデータを開放 */
            if (data)
                free_mydata(data);
            break;
        }

        case PGEVT_RESULTCREATE:
        {
            PGEventResultCreate *e = (PGEventResultCreate *)evtInfo;
            mydata *conn_data = PQinstanceData(e->conn, myEventProc);
            mydata *res_data = dup_mydata(conn_data);

            /* アプリ特有のデータを結果と (connから複写して) 関連付ける */
            PQsetResultInstanceData(e->result, myEventProc, res_data);
            break;
        }
    }
}

```

```

}

case PGEVT_RESULTCOPY:
{
    PGEventResultCopy *e = (PGEventResultCopy *)evtInfo;
    mydata *src_data = PQresultInstanceData(e->src, myEventProc);
    mydata *dest_data = dup_mydata(src_data);

    /* アプリ特有のデータを結果と（結果から複写して）関連付ける */
    PQsetResultInstanceData(e->dest, myEventProc, dest_data);
    break;
}

case PGEVT_RESULTDESTROY:
{
    PGEventResultDestroy *e = (PGEventResultDestroy *)evtInfo;
    mydata *data = PQresultInstanceData(e->result, myEventProc);

    /* 結果が破棄されたためインスタンスデータを開放 */
    if (data)
        free_mydata(data);
    break;
}

/* 未知のイベント識別子。単にTRUEを返す */
default:
    break;
}

return TRUE; /* イベント処理成功 */
}

```

[前のページ](#)
警告処理

[ホーム](#)
[上に戻る](#)

[次のページ](#)
環境変数

31.14. 環境変数

以下の環境変数を使用して、呼び出し側のプログラムで直接値を指定しなかった場合の接続パラメータのデフォルト値を選ぶことができます。この値は、PQconnectdb、PQsetdbLoginおよびPQsetdbで使用されます。例えば、簡単なクライアントアプリケーションでは、データベース接続情報を直接プログラムに記述しない方が便利です。

- PGHOSTは[host](#)接続パラメータと同様に動作します。
- PGHOSTADDRは[hostaddr](#)接続パラメータと同様に動作します。PGHOSTの代わりに設定して、または、PGHOSTに追加して、DNS検索に要するオーバーヘッドをなくすることができます。
- PGPORTは[port](#)接続パラメータと同様に動作します。
- PGDATABASEは[dbname](#)接続パラメータと同様に動作します。
- PGUSERは[user](#)接続パラメータと同様に動作します。
- PGPASSWORDは[password](#)接続パラメータと同様に動作します。この環境変数は、一部のオペレーティングシステムではroot以外のユーザがpsコマンド経由で環境変数を見ることができるなど、セキュリティ上の理由から現在では推奨されていません。代わりに`~/.pgpass` ([項31.15](#)を参照してください)を使用することを検討してください。
- PGPASSFILEは検索に使用するパスワードファイルの名前を指定します。指定していなければ、デフォルトの`~/.pgpass` ([項31.15](#)を参照してください)となります。
- PGSERVICEは[service](#)接続パラメータと同様に動作します。
- PGSERVICEFILEはユーザごとの接続サービスファイルを指定します。設定されていない場合、デフォルトは`~/.pg_service.conf` ([項31.16](#)参照)となります。
- PGREALMは、PostgreSQLで使用するKerberosのレルムがローカルのものとは異なる場合に、そのレルムを設定します。PGREALMが設定されると、libpqアプリケーションは設定されたレルム用のサーバと認証を行い、また、別のチケットファイルを使って、ローカルのチケットファイルと競合しないようにします。この環境変数は、サーバがKerberosによる認証を選択した場合にのみ使われます。
- PGOPTIONSは[options](#)接続パラメータと同様に動作します。
- PGAPPNAMEは[application_name](#)接続パラメータと同様に動作します。
- PGSSLMODEは[sslmode](#)接続パラメータと同様に動作します。
- PGREQUIRESSLは[requiressl](#)接続パラメータと同様に動作します。
- PGSSLCOMPRESSIONは[sslcompression](#)接続パラメータと同様に動作します。
- PGSSLCERTは[sslcert](#)接続パラメータと同様に動作します。
- PGSSLKEYは[sslkey](#)接続パラメータと同様に動作します。
- PGSSLROOTCERTは[sslrootcert](#)接続パラメータと同様に動作します。
- PGSSLCRLは[sslcrl](#)接続パラメータと同様に動作します。
- PGREQUIREPEERは[requirepeer](#)接続パラメータと同様に動作します。
- PGKRB_SRVNAMEは[krbsrvname](#)接続パラメータと同様に動作します。
- PGSSLIBは[gsslib](#)接続パラメータと同様に動作します。
- PGCONNECT_TIMEOUTは[connect_timeout](#)接続パラメータと同様に動作します。
- PGCLIENTENCODINGは[client_encoding](#)接続パラメータと同様に動作します。

以下の環境変数を使用して、PostgreSQLセッション毎のデフォルト動作を指定することができます。(また、ユーザ毎、もしくは、データベース毎を単位としたデフォルト動作の設定方法については[ALTER ROLE](#)および[ALTER DATABASE](#)コマンドを参照してください。)

- `PGDATESTYLE`はデフォルトの日付/時刻表現形式を設定します。(`SET datestyle TO ...`と等価です。)
- `PGTZ`はデフォルトの時間帯を設定します。(`SET timezone TO ...`と等価です。)
- `PGGEQO`は遺伝的問い合わせオプティマイザのデフォルトのモードを設定します。(`SET geqo TO ...`と等価です。)

これらの環境変数の正確な値については、[SET](#) SQLコマンドを参照してください。

以下の環境変数は、`libpq`の内部動作を決定します。 これらはコンパイル時のデフォルトを上書きします。

- `PGSYSCONFDIR`は`pg_service.conf`ファイルがあるディレクトリを設定します。 また今後のバージョンでは他のシステム全体の設定ファイルとなるかもしれません。
- `PGLOCALEDIR`は国際化メッセージ用の`locale`ファイルがあるディレクトリを設定します。

[前のページ](#)
イベントシステム

[ホーム](#)
[上に戻る](#)

[次のページ](#)
パスワードファイル

31.15. パスワードファイル

ユーザのホームディレクトリの `.pgpass` または `PGPASSFILE` で設定されるファイルは、接続にパスワードが必要な場合(かつ、他に指定されたパスワードが無かった場合)に使用するパスワードを格納するファイルです。Microsoft Windowsでは、このファイルの名前は `%APPDATA%\postgresql\pgpass.conf` (ここで `%APPDATA%` はユーザのプロファイル内のアプリケーションデータディレクトリ)です。

このファイル内の行の書式は次の通りです。

```
hostname:port:database:username:password
```

(このファイルでは、上のような行をコピーし、その先頭に `#` をつけて忘れないようにコメントとして残すことができます。) 先頭の4フィールドはそれぞれリテラル値にすることも、あるいはすべてに一致する `*` を使用することもできます。最初に現在の接続パラメータと一致した行のパスワードフィールドが使用されます。(従って、ワイルドカードを使用する場合は、始めの方により具体的な項目を入力してください。) 項目内に `:` または `\` を含める必要があれば、`\` でこれらの文字をエスケープする必要があります。 `localhost` ホスト名は、ローカルマシンからのTCP接続(ホスト名が `localhost`)とUnixドメインソケット接続(`pghost` が空またはデフォルトのソケットディレクトリ)の両方に一致します。スタンバイサーバでは、`replication` という名称のデータベースは、マスタサーバとの間でなされるストリーミングレプリケーション用の接続に一致します。同一のクラスタ内のすべてのデータベースに対するパスワードは同じものですので、`database` フィールドの有用性は限定的なものです。

Unixシステムにおいて、`.pgpass` の権限はグループ、他者へのアクセスをすべて拒否しなければなりません。これは `chmod 0600 ~/.pgpass` といったコマンドによって行います。権限をこれよりも緩くすると、このファイルは無視されます。Microsoft Windowsにおいては、このファイルが安全なディレクトリに格納されていることを前提としていますので、特別に行われる権限の検査はありません。

31.16. 接続サービスファイル

接続サービスファイルにより、libpq接続パラメータをひとつのサービス名に関連付けることができます。サービス名は、libpq接続によって指定され、関連付けられた設定が利用されます。これは、接続パラメータをlibpqアプリケーションの再コンパイルをせずに修正できるというものです。サービス名はPGSERVICE環境変数を利用することで使用できます。

この接続サービスファイルは、ユーザごとに`~/.pg_service.conf`というサービスファイルとすること、または、PGSERVICEFILE環境変数で指定される場所に行うことができます。また、システム全体についてのファイルとして`etc/pg_service.conf`とすること、PGSYSCONFDIR環境変数で指定されたディレクトリに置くことができます。ユーザ用、システム用のファイルで同名のサービス定義が存在する場合、ユーザ用のものが優先されます。

このファイルは"INIファイル"書式を使用します。セクション名がサービス名となり、パラメータが接続パラメータです。[項31.1](#)のリストを参照してください。以下に例を示します。

```
# comment
[mydb]
host=somehost
port=5433
user=admin
```

例となるファイルが`share/pg_service.conf.sample`にあります。

31.17. 接続パラメータのLDAP検索

libpqがLDAPサポート（configure時の`--with-ldap`オプション）付きでコンパイルされている場合、中央サーバからLDAPを通して`host`や`dbname`などの接続オプションを取り出すことができます。この利点は、データベースの接続パラメータが変わった場合に、すべてのクライアントマシンで接続情報を更新しなくても済む点です。

LDAP接続パラメータ検索は、`pg_service.conf`という接続サービスファイル（[項31.16](#)を参照）を使用します。

`pg_service.conf`内の`ldap://`から始まる行は、LDAP URLとして認識され、LDAP問い合わせが実行されることを示します。その結果は、`keyword = value`という組み合わせのリストでなければなりません。これらが接続用オプションの設定に使用されます。このURLはRFC 1959に従ったもので、以下のような形式でなければなりません。

```
ldap://[hostname[:port]]/search_base?attribute?search_scope?filter
```

ここで、`hostname`のデフォルトは`localhost`、`port`のデフォルトは389です。

`pg_service.conf`の処理はLDAP検索が成功した時に終わります。しかし、もしLDAPサーバへのアクセスができなかった場合は継続します。これはアクセスに失敗した時に、異なるLDAPサーバを指し示す他のLDAP行や以前からの`keyword = value`の組み合わせ、デフォルトの接続オプションを参照する予備機能を提供します。この場合にエラーメッセージを受け取りたい場合は、LDAP URL行の後に文法的に不正な行を記載してください。

LDIFファイルとして作成されたLDAP項目の例を以下に示します。

```
version:1
dn:cn=mydatabase,dc=mycompany,dc=com
changetype:add
objectclass:top
objectclass:groupOfUniqueNames
cn:mydatabase
uniqueMember:host=dbserver.mycompany.com
uniqueMember:port=5439
uniqueMember:dbname=mydb
uniqueMember:user=mydb_user
uniqueMember:sslmode=require
```

これは、以下のようなLDAP URLから得られます。

```
ldap://ldap.mycompany.com/dc=mycompany,dc=com?uniqueMember?one?(cn=mydatabase)
```

また、LDAP検索と通常のサービスファイル項目とを混在させることもできます。`pg_service.conf`の一節について完全な例を以下に示します。

```
# only host and port are stored in LDAP, specify dbname and user explicitly
[customerdb]
dbname=customer
user=appuser
ldap://ldap.acme.com/cn=dbserver,cn=hosts?pgconnectinfo?base?(objectclass=*)
```

31.18. SSLサポート

PostgreSQLは、セキュリティを高めるためにクライアントサーバ間の通信を暗号化するSSL接続の使用を元来サポートしています。サーバ側のSSL機能についての詳細は[項17.9](#)を参照してください。

libpqはシステム全体に対するOpenSSL設定ファイルを読み込みます。デフォルトでは、ファイル名は`openssl.cnf`で、`openssl version -d`で報告されるディレクトリに格納されています。このデフォルトは`OPENSSL_CONF`環境変数に希望する設定ファイル名を設定することで変更することができます。

31.18.1. サーバ証明書のクライアント検証

デフォルトではPostgreSQLはサーバ証明書の検証をまったく行いません。これは、クライアントに知られずにサーバの属性をなりすます（例えば、DNSレコードを変更したり、もしくはサーバのIPアドレスを接続したりして）ことができることを意味します。なりすましを防止するにはSSL証明書検証が使用されなければなりません。

パラメータ`sslmode`が`verify-ca`に設定されている場合、libpqは信頼される認証局(CA)までの証明書連鎖を検査することで、サーバが信用に足るかを検証します。`sslmode`が`verify-full`に設定されていると、libpqは同時にサーバホスト名が証明書のそれと一致するかを検証します。SSL接続はサーバ証明書が検証されない場合失敗します。安全性に慎重を期するほとんどのサーバ環境では`verify-full`を推奨します。

`verify-full`モードでは証明書の`cn`（コモンネーム）属性はホスト名と一致させられます。`cn`属性がアスタリスク（*）で始めると、それはワイルドカードとして取り扱われ、ドット（.）を除くすべての文字と一致します。これは、証明書がサブドメインと一致しないことを意味します。もし接続がホスト名ではなくIPアドレスを使用するのであれば、（いかなるDNS検索もせず）IPアドレスが一致させられます。

サーバ証明書の検証を可能にするには、1つ以上の信頼するCAの証明書を、ユーザのホームディレクトリ内の`~/.postgresql/root.crt`ファイルに置かれなければなりません。（Microsoft Windowsの場合、このファイルの名前は`%APPDATA%\postgresql\root.crt`です。）

`~/.postgresql/root.crl`ファイル（Microsoft Windowsでは`%APPDATA%\postgresql\root.crl`）が存在する場合、証明書失効リスト（CRL）の項目もまた検査されます。

ルート証明書ファイルとCRLの格納場所を接続パラメータ`sslrootcert`と`sslcrl`、もしくは環境変数`PGSSLROOTCERT`と`PGSSLCRL`で変更することができます。

注意: より古いバージョンのPostgreSQLとの後方互換性のために、ルートCAファイルが存在する場合、`sslmode=require`の動作は`verify-ca`の場合と同じになっています。つまり、サーバ証明書がCAに対して検証されます。この動作に依存することは勧めません。また証明書の検証を必要とするアプリケーションは常に`verify-ca`または`verify-full`を使用すべきです。

31.18.2. クライアント証明書

サーバが信頼できるクライアント証明書を要求する場合、libpqはユーザのホームディレクトリにある`~/.postgresql/postgresql.crt`ファイルに格納された証明書を送信します。証明書はサーバで信頼された認証局（CA）のいずれかで署名されなければなりません。対応する`~/.postgresql/postgresql.key`秘密キーファイルも存在しなければなりません。秘密キーファイルは他者やグループからのアクセスを許可してはいけません。`chmod 0600 ~/.postgresql/postgresql.key`コマンドでこれを実現してください。Microsoft Windowsでは、このファイルの名前はそれぞれ`%APPDATA%\postgresql\postgresql.crt`と`%APPDATA%\postgresql\postgresql.key`であり、このディレクトリは安全であると想定されますので、特別な権限検査は行われません。証明書とキーファイルの格納場所は`sslcert`および`sslkey`接続パラメータ、または`PGSSLCERT`および`PGSSLKEY`環境変数で上書きされます。

クライアント証明書が、サーバにより直接信頼された認証局ではなく、"中間"証明局で署名されている場合があります。こうした証明書を使用するためには、署名した認証局の証明書とその親の認証局の証明書をサーバにより信頼される"ルート"認証局まで`postgresql.crt`ファイルに追加してください。`postgresql.crt`に複数の証明書を含める時、すべての場合においてルート証明書を含めなければなりません。

`root.crt`には、サーバ証明書の署名に関して信頼できるとみなされる、最上位のCAが列挙されていることに注意してください。原理的にはクライアント証明書を署名するCAを列挙する必要はありませんが、ほとんどの場合において、そのCAは同時にサーバ証明書に対しても信頼されています。

31.18.3. 異なるモードで提供される保護

sslmodeパラメータ値を変更することで、異なったレベルの保護を提供します。SSLは以下の3種類の攻撃に対する保護を提供することができます。

盗聴

クライアント・サーバ間のネットワークトラフィックを第三者が監視することができれば、（ユーザ名とパスワードを含め）双方の接続情報と通過するデータを読み取ることができます。SSLはこれを防止するために暗号を使用します。

中間者攻撃（MITM）

データがクライアント・サーバ間で渡されている時に、第三者がそのデータを変更できれば、サーバを装うことができ、従ってたとえ暗号化されていてもデータを理解し変更することができます。第三者はそこで、この攻撃を検出不可能にする接続情報とデータを元のサーバに送ることができます。これを行う共通した媒介はDNSポイズニングとアドレス乗っ取りを含み、それに従ってクライアントは意図したサーバではなく異なったサーバに誘導されます。同時に、このことを成し遂げるいくつかの異なった攻撃も存在します。SSLはクライアントに対しサーバを認証することで、この防止に証明書検証を使用します。

なりすまし

第三者が認定されたクライアントを装うことができれば、それはアクセスしてはならないデータに簡単にアクセス可能になります。典型的にこれは心もとないパスワード管理から生じます。SSLは有効な証明書の所持者のみサーバにアクセスできることを確実にすることで、この防止策としてクライアント証明書を使用します。

信頼できるとされる接続では、SSLの使用を接続確立前にクライアントとサーバの双方において設定されなければなりません。サーバのみに構成されると、クライアントはサーバが高度なセキュリティを必要とすることが判る以前に、（例えばパスワードのような）機密事項を扱う情報を結局送ることになります。libpqにおいて、sslmodeパラメータをverify-fullまたはverify-caに設定し、そして対象を検証するためルート証明書をシステムに提供することで、安全な接続を確実に行うことができます。これは暗号化されたweb閲覧に対するhttps URLの使用とよく似ています。

一度サーバが認証されると、クライアントは機密事項を扱うデータを送ることができます。この意味は、これまでクライアントは認証に証明書が使われているかどうかを知る必要がなく、サーバ構成においてのみこのことを指定しても安全だと言うことです。

すべてのSSLオプションでは暗号化の形式と鍵交換といったオーバーヘッドがかかります。このため性能と安全性との間で決定されるべきトレードオフがあります。表31-1は異なるsslmode値が対象を防御するかの危険性と、安全性とオーバーヘッドに対する声明を示したものです。

表 31-1. SSLモードの説明

sslmode	盗聴防止	MITM防止	声明
disable	いいえ	いいえ	セキュリティはどうでもよく、暗号化の負荷を払いたくない
allow	たぶん	いいえ	セキュリティはどうでもよいが、サーバがそれを強く要求するのであれば暗号化のオーバーヘッドを払ってもよい
prefer	たぶん	いいえ	セキュリティはどうでもよいが、サーバがそれをサポートするのであれば暗号化のオーバーヘッドを払ってもよい
require	はい	いいえ	データを暗号化して欲しい。そしてオーバーヘッドも受け入れる。サーバに接続する必要時に常にネットワークが確認してくれることを信用する
verify-ca	はい	CAの規定に依存	データを暗号化して欲しい。そしてオーバーヘッドも受け入れる。信頼するサーバに確実に接続したい
verify-full	はい	はい	データを暗号化して欲しい。そしてオーバーヘッドも受け入れる。信頼するサーバに接続すること、そのサーバが指定したものであることを確実にしたい

verify-caとverify-fullの差異はルートCAの規定に依存します。公的なCAが使用されるとき、verify-caはそのCAで他の誰かが登録したかもしれないサーバへの接続を許可します。この場合、verify-fullが常に使用されなければなりません。独自CAが使用されるとき、または自己署名証明書であったとしてもverify-caは十分な防御策を提供します。

sslmodeのデフォルト値はpreferです。表で示したように、これはセキュリティの視点では意味がなく、可能であれば性能上のオーバーヘッドを保証するだけです。これは後方互換性を提供するのためのみにデフォルトとなっているもので、安全性確保の観点からは推奨されません。

31.18.4. SSLクライアントファイル使用法

表31-2にクライアントにおけるSSL設定に関連するファイルをまとめます。

表 31-2. libpq/ クライアントにおけるSSLファイルの使用方法

ファイル	内容	効果
~/.postgresql/postgresql.crt	クライアント証明書	サーバにより要求されます
~/.postgresql/postgresql.key	クライアントの秘密キー	所有者により送信されるクライアント証明書を証明します。証明書の所有者が信頼できることを意味していません。
~/.postgresql/root.crt	信頼できる認証局	サーバ証明書が信頼できる認証局により署名されたか検査します。
~/.postgresql/root.crl	認証局により失効された証明書	サーバ証明書はこのリストにあってはいけません

31.18.5. SSLライブラリの初期化

使用するアプリケーションがlibsslとlibcryptoの両方またはいずれか一方のライブラリを初期化し、libpqがSSLサポート付きで構築された場合、libsslとlibcryptoの両方またはいずれか一方のライブラリはアプリケーションによって初期化されたことをlibpqに伝えるためPQinitOpenSSLを呼び出さなければなりません。これにより、libpqはこれらのライブラリを初期化しなくなります。SSL APIの詳細はhttp://h71000.www7.hp.com/doc/83final/BA554_90007/ch04.htmlを参照してください。

PQinitOpenSSL

アプリケーションがどのセキュリティライブラリを初期化するか選択することができます。

```
void PQinitOpenSSL(int do_ssl, int do_crypto);
```

do_sslが非ゼロの時、libpqは最初のデータベース接続を開始する以前にOpenSSLライブラリを初期化します。do_cryptoが非ゼロの時、libcryptoライブラリが初期化されます。デフォルトでは（PQinitOpenSSLが呼ばれない場合）、両方のライブラリが初期化されます。SSLサポートがコンパイルされていない場合、この関数は存在しますが何もしません。

使用するアプリケーションがOpenSSLまたはその基礎をなすlibcryptoライブラリのいずれかを使用し、そして初期化するのであれば、最初のデータベース接続開始以前に、適切なパラメータをゼロにしてこの関数を呼び出さなければなりません。同時に、データベース接続開始前に初期化を行ったことの確認をしてください。

PQinitSSL

アプリケーションがどのセキュリティライブラリを初期化するか選択することができます。

```
void PQinitSSL(int do_ssl);
```

この関数はPQinitOpenSSL(do_ssl, do_ssl)と等価です。OpenSSLおよびlibcryptoの両方を初期化する、もしくは両方ともしないアプリケーションにとっては（この関数で）十分です。

PostgreSQL 8.0以降、PQinitSSLは含まれていますが、PQinitOpenSSLはPostgreSQL 8.4で追加されました。従って、旧バージョンのlibpqで動かす必要があるアプリケーションではPQinitSSLの方が好ましいかもしれません。

31.19. スレッド化プログラムの振舞い

デフォルトでlibpqは再入可能、かつ、スレッドセーフです。アプリケーションコードをコンパイルする時にコンパイラの特異的なコマンドラインオプションに追加する必要があるかもしれません。スレッドを有効にしたアプリケーションの構築方法についての情報は、使用するシステムの文書を参照してください。また、`PTHREAD_CFLAGS`と`PTHREAD_LIBS`に関して`src/Makefile.global`も一読してください。以下の関数により、libpqのスレッドセーフ状態を確認することができます。

`PQisthreadsafe`

libpqライブラリのスレッドセーフ状態を返します。

```
int PQisthreadsafe();
```

libpqがスレッドセーフの場合1が、さもなければ0が返ります。

スレッドに関する1つの制限として、異なるスレッドから同時に同一の`PGconn`オブジェクトを操作することはできません。具体的には、異なるスレッドから同一接続オブジェクトを介してコマンドを同時に発行することができません。(コマンドの同時実行が必要な場合、接続を複数使用してください。)

`PGresult`オブジェクトは生成後、読み込み専用であり、そのためスレッド間で自由に渡すことができます。しかし[項31.11](#)や[項31.13](#)で説明する`PGresult`を変更する関数のいずれかを使用している場合、同一の`PGresult`に対する同時操作を防ぐことも、作成者の責任です。

非推奨の関数、`PQrequestCancel`や`PQoidStatus`はスレッドセーフではありませんので、マルチスレッドプログラムでは使用してはなりません。`PQrequestCancel`は`PQcancel`に、`PQoidStatus`は`PQoidValue`に置き換えることができます。

(libpqの内部に加えて)アプリケーション中でKerberosを利用している場合、Kerberos関数はスレッドセーフではありませんのでKerberos呼び出しの前後をロックする必要があるでしょう。libpqとアプリケーション間のロック処理を協調させる方法としてlibpqのソースコードの`PQregisterThreadLock`関数を参照してください。

スレッドアプリケーションで問題が起きた場合、`src/tools/thread`内のプログラムを実行して、プラットフォームがスレッドセーフではない関数を持っていないかどうか確認してください。このプログラムは`configure`より実行されますが、バイナリ配布物の場合は、バイナリ構築時のライブラリとシステムのライブラリが一致しない可能性があります。

31.20. libpqプログラムの構築

libpqを使用するプログラムの構築(つまり、コンパイルとリンク)を行うためには、以下をすべて実施する必要があります。

- libpq-fe.hヘッダファイルをインクルードします。

```
#include <libpq-fe.h>
```

これを忘れると、通常コンパイラから以下のようなエラーメッセージが発生します。

```
foo.c: In function `main':
foo.c:34: `PGconn' undeclared (first use in this function)
foo.c:35: `PGresult' undeclared (first use in this function)
foo.c:54: `CONNECTION_BAD' undeclared (first use in this function)
foo.c:68: `PGRES_COMMAND_OK' undeclared (first use in this function)
foo.c:95: `PGRES_TUPLES_OK' undeclared (first use in this function)
```

- コンパイラに-Idirectoryオプションを付与することで、コンパイラにPostgreSQLヘッダファイルをインストールしたディレクトリを通知します。(デフォルトでこのディレクトリを検索するコンパイラもあります。その場合はこのオプションを省くことができます。)例えば、以下のようなコンパイルコマンドになります。

```
cc -c -I/usr/local/pgsql/include testprog.c
```

Makefileを使用しているのであれば、CPPFLAGS変数にこのオプションを追加してください。

```
CPPFLAGS += -I/usr/local/pgsql/include
```

他のユーザがそのプログラムをコンパイルする可能性がある場合は、上にようにディレクトリの場所を直接書き込むべきではありません。その代わりにpg_configユーティリティを実行して、各システムにおけるヘッダファイルの在処を検索させることができます。

```
$ pg_config --includedir
/usr/local/include
```

正確なオプションを指定できなかった結果、コンパイラは以下のようなエラーメッセージを生成します。

```
testlibpq.c:8:22: libpq-fe.h: No such file or directory
```

- 最終的なプログラムのリンク時、-lpqオプションを指定して、libpqライブラリを組み込んでください。同時に-Ldirectoryオプションを指定して、コンパイラにlibpqライブラリの在処を通知してください。(繰り返しますが、コンパイラはデフォルトでいくつかのディレクトリを検索します。)移植性を最大にするために、-lpqオプションの前に-Lを記述してください。以下に例を示します。

```
cc -o testprog testprog1.o testprog2.o -L/usr/local/pgsql/lib -lpq
```

同様にpg_configを使用してライブラリのあるディレクトリを見つけることもできます。

```
$ pg_config --libdir
/usr/local/pgsql/lib
```

この部分で問題があった場合のエラーメッセージは以下のようなものになります。

```
testlibpq.o: In function `main':
testlibpq.o(.text+0x60): undefined reference to `PQsetdbLogin'
testlibpq.o(.text+0x71): undefined reference to `PQstatus'
testlibpq.o(.text+0xa4): undefined reference to `PQerrorMessage'
```

これは-lpqの付け忘れを示します。

```
/usr/bin/ld: cannot find -lpq
```

これは-Lの付け忘れ、あるいは、ディレクトリ指定の間違いを示します。

31.21. サンプルプログラム

以下を含むサンプルプログラムが、ソースコード配布物内のsrc/test/examplesディレクトリにあります。

例 31-1. libpq サンプルプログラム 1

```

/*
 * testlibpq.c
 *
 *
 *          C言語PostgreSQLフロントエンドライブラリlibpqの試験
 */
#include <stdio.h>
#include <stdlib.h>
#include <libpq-fe.h>

static void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

int
main(int argc, char **argv)
{
    const char *conninfo;
    PGconn     *conn;
    PGresult    *res;
    int         nFields;
    int         i,
               j;

    /*
     * ユーザがコマンドラインでパラメータを提供した場合、conninfo文字列として使用する。
     * 提供されない場合はデフォルトでdbname=postgresを使用する。
     * その他の接続パラメータについては環境変数やデフォルトを使用する。
     */
    if (argc > 1)
        conninfo = argv[1];
    else
        conninfo = "dbname = postgres";

    /* データベースとの接続を確立する */
    conn = PQconnectdb(conninfo);

    /* バックエンドとの接続確立に成功したかを確認する */
    if (PQstatus(conn) != CONNECTION_OK)
    {
        fprintf(stderr, "Connection to database failed: %s",
                PQerrorMessage(conn));
        exit_nicely(conn);
    }

    /*
     * この試験ケースではカーソルを使用する。
     * そのため、トランザクションブロック内で実行する必要がある。
     * すべてを単一の"select * from pg_database"というPQexec()で行うこと
     * も可能だが、例としては簡単過ぎる。
     */

    /* トランザクションブロックを開始する。 */
    res = PQexec(conn, "BEGIN");
    if (PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "BEGIN command failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }

    /*
     * 不要になったら、メモリリークを防ぐためにPGresultをPQclearすべき。
     */
    PQclear(res);

    /*
     * データベースのシステムカタログpg_databaseから行を取り出す。

```

```

    */
    res = PQexec(conn, "DECLARE myportal CURSOR FOR select * from pg_database");
    if (PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "DECLARE CURSOR failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }
    PQclear(res);

    res = PQexec(conn, "FETCH ALL in myportal");
    if (PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        fprintf(stderr, "FETCH ALL failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }

    /* まず属性名を表示する。 */
    nFields = PQnfields(res);
    for (i = 0; i < nFields; i++)
        printf("%-15s", PQfname(res, i));
    printf("\n\n");

    /* そして行を表示する。 */
    for (i = 0; i < PQntuples(res); i++)
    {
        for (j = 0; j < nFields; j++)
            printf("%-15s", PQgetvalue(res, i, j));
        printf("\n");
    }

    PQclear(res);

    /* ポータルを閉じます。ここではエラーチェックは省略した... */
    res = PQexec(conn, "CLOSE myportal");
    PQclear(res);

    /* トランザクションを終了する */
    res = PQexec(conn, "END");
    PQclear(res);

    /* データベースとの接続を閉じ、後始末を行う。 */
    PQfinish(conn);

    return 0;
}

```

例 31-2. libpq サンプルプログラム 2

```

/*
 * testlibpq2.c
 *
 * 非同期通知インタフェースの試験
 *
 * このプログラムを起動し、別ウィンドウからpsqlを使用して以下を実行してください。
 * NOTIFY TBL2;
 * 4回繰り返すとこのプログラムは終了します。
 *
 * もう少し凝りたければ、以下を実施してください。
 * 以下のコマンド(src/test/examples/testlibpq2.sqlで提供)でデータベースを作成します。
 *
 * CREATE TABLE TBL1 (i int4);
 *
 * CREATE TABLE TBL2 (i int4);
 *
 * CREATE RULE r1 AS ON INSERT TO TBL1 DO
 * (INSERT INTO TBL2 VALUES (new.i); NOTIFY TBL2);
 *
 * そして、以下を4回実行してください。
 *
 * INSERT INTO TBL1 VALUES (10);
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/time.h>
#include <libpq-fe.h>

static void
exit_nicely(PGconn *conn)

```

```

{
    PQfinish(conn);
    exit(1);
}

int
main(int argc, char **argv)
{
    const char *conninfo;
    PGconn      *conn;
    PGresult     *res;
    PGnotify     *notify;
    int          nnotifies;

    /*
     * ユーザがコマンドラインでパラメータを提供した場合、conninfo文字列として使用する。
     * 提供されない場合はデフォルトでdbname=postgresを使用する。
     * その他の接続パラメータについては環境変数やデフォルトを使用する。
     */
    if (argc > 1)
        conninfo = argv[1];
    else
        conninfo = "dbname = postgres";

    /* データベースとの接続を確立する。 */
    conn = PQconnectdb(conninfo);

    /* バックエンドとの接続確立に成功したかを確認する */
    if (PQstatus(conn) != CONNECTION_OK)
    {
        fprintf(stderr, "Connection to database failed: %s",
            PQerrorMessage(conn));
        exit_nicely(conn);
    }

    /*
     * LISTENコマンドを発行して、INSERTルールによる通知を有効にする。
     */
    res = PQexec(conn, "LISTEN TBL2");
    if (PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "LISTEN command failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }

    /*
     * 不要になったら、メモリリークを防ぐためにPGresultをPQclearすべき。
     */
    PQclear(res);

    /* 4回通知を受けたら終了する。 */
    nnotifies = 0;
    while (nnotifies < 4)
    {
        /*
         * その接続で何かが起こるまで待機する。ここでは入力待ちのために
         * select(2)を使用する。poll()や類似機能を使用することも可能
         * である。
         */
        int          sock;
        fd_set       input_mask;

        sock = PQsocket(conn);

        if (sock < 0)
            break; /* 発生してはならない。 */

        FD_ZERO(&input_mask);
        FD_SET(sock, &input_mask);

        if (select(sock + 1, &input_mask, NULL, NULL, NULL) < 0)
        {
            fprintf(stderr, "select() failed: %s\n", strerror(errno));
            exit_nicely(conn);
        }

        /* ここで入力を確認する。 */
        PQconsumeInput(conn);
        while ((notify = PQnotifies(conn)) != NULL)
        {
            fprintf(stderr,
                "ASYNC NOTIFY of '%s' received from backend PID %d\n",
                notify->relname, notify->be_pid);
            PQfreemem(notify);
        }
    }
}

```



```

        nnotifies++;
    }
}

fprintf(stderr, "Done.\n");

/* データベースとの接続を閉じ、後始末を行う。 */
PQfinish(conn);

return 0;
}

```

例 31-3. libpq サンプルプログラム 3

```

/*
 * testlibpq3.c
 * 行以外のパラメータとバイナリI/Oの試験。
 */

/* 実行前に、以下のコマンド(src/test/examples/testlibpq3.sqlで提供)を使用して
 * データベースを作成してください。
 */
/* CREATE TABLE test1 (i int4, t text, b bytea);
 *
 * INSERT INTO test1 values (1, 'joe's place', '\000\001\002\003\004');
 * INSERT INTO test1 values (2, 'ho there', '\004\003\002\001\000');
 */
/* 以下の出力が想定されます。
 */
/* tuple 0: got
 * i = (4 bytes) 1
 * t = (11 bytes) 'joe's place'
 * b = (5 bytes) \000\001\002\003\004
 */
/* tuple 0: got
 * i = (4 bytes) 2
 * t = (8 bytes) 'ho there'
 * b = (5 bytes) \004\003\002\001\000
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <libpq-fe.h>

/* ntohl/htonl用 */
#include <netinet/in.h>
#include <arpa/inet.h>

static void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

/*
 * この関数は上のコメントで定義したテーブルからバイナリフォーマットでフェッチした
 * クエリ結果を表示します。
 * main() 関数が2度使うので、結果を分割します。
 */
static void
show_binary_results(PGresult *res)
{
    int i,
        j;
    int i_fnum,
        t_fnum,
        b_fnum;

    /* 結果中の列オーダーの仮定を嫌うので PQfnumber を利用する */
    /* PQfnumber */
    i_fnum = PQfnumber(res, "i");
    t_fnum = PQfnumber(res, "t");
    b_fnum = PQfnumber(res, "b");

    for (i = 0; i < PQntuples(res); i++)
    {
        char *iptr;
        char *tptr;
        char *bptr;
        int blen;
        int ival;
    }
}

```

```

/* 列の値を取得 (NULLを出来る限り無視) */
iptr = PQgetvalue(res, i, i_fnum);
tptr = PQgetvalue(res, i, t_fnum);
bptr = PQgetvalue(res, i, b_fnum);

/*
 * INT4のバイナリ表現はネットワークバイトオーダーによる。
 * よって、ローカルバイトオーダーに合わせた方が良い。
 */
ival = ntohl(*(uint32_t *) iptr);

/*
 * TEXT型のバイナリ表現も同様にテキスト。
 * 更にlibpqはその最後にゼロバイトを付与するので、
 * C言語の文字列として単純に扱うことができる。
 *
 * BYTEA のバイト表現はバイトの集まりである。
 * null 埋め込みを含むのでフィールド長に注意を払わなければいけない。
 */
blen = PQgetlength(res, i, b_fnum);

printf("tuple %d: got\n", i);
printf(" i = (%d bytes) %d\n",
       PQgetlength(res, i, i_fnum), ival);
printf(" t = (%d bytes) '%s'\n",
       PQgetlength(res, i, t_fnum), tptr);
printf(" b = (%d bytes) ", blen);
for (j = 0; j < blen; j++)
    printf("%%03o", bptr[j]);
printf("\n\n");
}

int
main(int argc, char **argv)
{
    const char *conninfo;
    PGconn     *conn;
    PGresult    *res;
    const char *paramValues[1];
    int         paramLengths[1];
    int         paramFormats[1];
    uint32_t    binaryIntVal;

    /*
     * ユーザがコマンドラインでパラメータを提供した場合、conninfo文字列として使用する。
     * 提供されない場合はデフォルトでdbname=postgresを使用する。
     * その他の接続パラメータについては環境変数やデフォルトを使用する。
     */
    if (argc > 1)
        conninfo = argv[1];
    else
        conninfo = "dbname = postgres";

    /* データベースとの接続を確立する */
    conn = PQconnectdb(conninfo);

    /* バックエンドとの接続確立に成功したかを確認する */
    if (PQstatus(conn) != CONNECTION_OK)
    {
        fprintf(stderr, "Connection to database failed: %s",
                PQerrorMessage(conn));
        exit_nicely(conn);
    }

    /*
     * このプログラムのポイントは、行外パラメータを持つPQexecParams() の使用方法、
     * および、データのバイナリ転送を示すことである。
     *
     * この最初の例はパラメータをテキストとして渡す。
     * しかし結果はバイナリフォーマットで受ける。
     * 行外パラメータを使うことで、データがテキストであっても引用符付けや
     * エスケープ処理といった多くの長たらしいゴミをなくすることができる。
     * パラメータ値内部の引用符印に対して特殊な処理を行う必要がないことに注目して
     * ほしい。
     */

    /* 以下が行外パラメータの値である。 */
    paramValues[0] = "joe's place";

    res = PQexecParams(conn,
                       "SELECT * FROM test1 WHERE t = $1",
                       1, /* パラメータは1つ。 */
                       NULL, /* バックエンドにパラメータの型を推測させる。 */
                       paramValues,
                       NULL, /* テキストのため、パラメータ長は不要。 */

```

```

        NULL,          /* デフォルトですべてのパラメータはテキスト。 */
        1);           /* バイナリ結果を要求。 */

if (PGresultStatus(res) != PGRES_TUPLES_OK)
{
    fprintf(stderr, "SELECT failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}

show_binary_results(res);

PQclear(res);

/*
 * 2つ目の例は、バイナリフォームの中で整数値パラメータを渡す。
 * そして再びバイナリフォームで結果を受け取る。
 *
 * バックエンドにパラメータタイプを推測させていると PQexecParams に伝えるが、
 * クエリテキストの中にパラメータシンボルを入れることによって 強制的に決定する。
 * これはバイナリパラメータに送るときに安全で良い大きさである。
 */

/* 整数値 "2" をネットワークバイトオーダーに変換 */
binaryIntVal = htonl((uint32_t) 2);

/* PQexecParams 用にパラメータ配列をセットする */
paramValues[0] = (char *) &binaryIntVal;
paramLengths[0] = sizeof(binaryIntVal);
paramFormats[0] = 1;          /* バイナリ */

res = PQexecParams(conn,
    "SELECT * FROM test1 WHERE i = $1::int4",
    1,          /* パラメータは1つ */
    NULL,      /* バックエンドにパラメータの型を推測させる。 */
    paramValues,
    paramLengths,
    paramFormats,
    1);        /* バイナリ結果を要求。 */

if (PGresultStatus(res) != PGRES_TUPLES_OK)
{
    fprintf(stderr, "SELECT failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}

show_binary_results(res);

PQclear(res);

/* データベースとの接続を閉じ、後始末を行う。 */
PQfinish(conn);

return 0;
}

```

第 32章 ラージオブジェクト

目次

- 32.1. [はじめに](#)
- 32.2. [実装機能](#)
- 32.3. [クライアントインタフェース](#)
 - 32.3.1. [ラージオブジェクトの作成](#)
 - 32.3.2. [ラージオブジェクトのインポート](#)
 - 32.3.3. [ラージオブジェクトのエクスポート](#)
 - 32.3.4. [既存のラージオブジェクトのオープン](#)
 - 32.3.5. [ラージオブジェクトへのデータの書き込み](#)
 - 32.3.6. [ラージオブジェクトからのデータの読み込み](#)
 - 32.3.7. [ラージオブジェクトのシーク](#)
 - 32.3.8. [ラージオブジェクトのシーク位置の入手](#)
 - 32.3.9. [ラージオブジェクトを切り詰める](#)
 - 32.3.10. [ラージオブジェクト記述子を閉じる](#)
 - 32.3.11. [ラージオブジェクトの削除](#)
- 32.4. [サーバ側の関数](#)
- 32.5. [サンプルプログラム](#)

PostgreSQLにはラージオブジェクト機能があります。これは、特殊なラージオブジェクト構造に格納されたユーザデータに対してストリーム様式のアクセスを提供します。全体をまるごと簡単に操作するには巨大過ぎるデータ値を操作する場合、ストリーミングアクセスが有用です。

本章では、PostgreSQLラージオブジェクトデータに関する、実装、プログラミング、問い合わせ言語インタフェースについて説明します。libpq Cライブラリを例として本章で使用していますが、ほとんどのPostgreSQL固有のプログラミングインタフェースは同等の機能を持っています。他のインタフェースでは、巨大な値を汎用的にサポートできるように、ラージオブジェクトインタフェースを内部で使用しているかもしれません。ここでは説明しません。

32.1. はじめに

すべてのラージオブジェクトはpg_largeobjectというひとつのシステムテーブル内に格納されます。PostgreSQLではまた、単一のデータベースページよりも大きな値を自動的にテーブルごとに存在する二次格納領域に格納する"TOAST"という格納システムをサポートします。これによりラージオブジェクトの一部は不要になりました。ラージオブジェクト機能に残る利点の1つは、そのサイズが2ギガバイトまで可能であるという点です。TOASTではフィールドは1ギガバイトまでしか扱えません。また、ラージオブジェクトは、TOASTでの操作よりもかなり性能が良い、read/write APIを使用したランダムな編集操作を行うことができます。

32.2. 実装機能

ラージオブジェクトの実装では、ラージオブジェクトを"チャンク"に分割し、チャンクをデータベース内の行に格納しています。B-treeインデックスは読み書き用のランダムアクセスに際して、正確なチャンク番号の高速検索を保証しています。

PostgreSQL 9.0からラージオブジェクトは所有者およびアクセス権限を持ちます。これは[GRANT](#)および[REVOKE](#)を使用して管理可能です。過去のリリースとの互換性を保つためには[lo_compat_privileges](#)を参照してください。ラージオブジェクトの読み取りには[SELECT](#)権限が必要です。また書き出し、切り詰めのためには[UPDATE](#)権限が必要です。ラージオブジェクトの所有者(またはデータベーススーパーユーザ)のみがラージオブジェクトの削除、コメント付け、所有者の変更が可能です。

32.3. クライアントインタフェース

本節では、PostgreSQLクライアントインタフェースライブラリで提供されるラージオブジェクトへのアクセス手段について説明します。これらの関数を使用したラージオブジェクトの操作は全てSQLトランザクションブロック内で行われなければなりません。PostgreSQLラージオブジェクトインタフェースは、Unixファイルシステムインタフェースに因んで設計されており、`open`、`read`、`write`、`lseek`など同様のインタフェースを有しています。

`libpq`のラージオブジェクトインタフェースを使用するクライアントアプリケーションは、`libpq/libpq-fs.h`ヘッダファイルをインクルードし、`libpq`ライブラリとリンクしなければなりません。

32.3.1. ラージオブジェクトの作成

```
Oid lo_creat(PGconn *conn, int mode);
```

この関数はラージオブジェクトを新規に作成します。戻り値は新規ラージオブジェクトに割り当てられたOIDで、失敗時には`InvalidOid` (0) が返されます。PostgreSQL 8.1では、`mode`は使用されず、無視されます。しかし、以前のリリースとの後方互換性を保持するために、これを`INV_READ`、`INV_WRITE`、`INV_READ | INV_WRITE`に設定することが最善です。（これらの定数シンボルは`libpq/libpq-fs.h`ヘッダファイルで定義されています。）

以下に例を示します。

```
inv_oid = lo_creat(conn, INV_READ|INV_WRITE);

Oid lo_create(PGconn *conn, Oid lobjId);
```

この関数もラージオブジェクトを新規に作成します。割り当てられるOIDを`lobjId`で指定することができます。こうした場合、そのOIDが他のラージオブジェクトですでに使用されていた場合、失敗します。`lobjId`が`InvalidOid` (0) の場合、`lo_create`は未使用のOIDを割り当てます。（これは`lo_creat`と同じ動作です。）戻り値は新規ラージオブジェクトに割り当てられたOIDで、失敗時には`InvalidOid` (0) が返されます。

`lo_create`はPostgreSQL 8.1から導入されました。この関数を古いバージョンで実行させると失敗し、`InvalidOid`が返されます。

例を示します。

```
inv_oid = lo_create(conn, desired_oid);
```

32.3.2. ラージオブジェクトのインポート

オペレーティングシステム上のファイルをラージオブジェクトとしてインポートするには、以下の関数を呼び出します。

```
Oid lo_import(PGconn *conn, const char *filename);
```

`filename`には、ラージオブジェクトとしてインポートするオペレーティングシステム上のファイルのパス名を指定します。戻り値は、新規ラージオブジェクトに割り当てられたOIDです。失敗時は`InvalidOid` (0) が返されます。このファイルがサーバではなく、クライアントインタフェースライブラリから読み取られることに注意してください。ですから、このファイルはクライアントのファイルシステム上に存在し、クライアントアプリケーションから読み取り可能でなければなりません。

```
Oid lo_import_with_oid(PGconn *conn, const char *filename, Oid lobjId);
```

この関数も新規のラージオブジェクトをインポートします。割り当てられるOIDを`lobjId`で指定することができます。こうした場合、そのOIDが他のラージオブジェクトですでに使用されていた場合、失敗します。`lobjId`が`InvalidOid` (0) の場合、`lo_import_with_oid`は未使用のOIDを割り当てます（これは`lo_import`と同じ動作です）。戻り値は新規ラージオブジェクトに割り当てられたOIDで、失敗時には`InvalidOid` (0) が返されます。

`lo_import_with_oid`はPostgreSQL 8.4から導入され、8.1から導入された`lo_create`を内部で使用しています。この関数を8.0以前のバージョンで実行させると失敗し、`InvalidOid`が返されます。

32.3.3. ラージオブジェクトのエクスポート

ラージオブジェクトをオペレーティングシステム上のファイルにエクスポートするには、以下の関数を呼び出します。

```
int lo_export(PGconn *conn, Oid lobjId, const char *filename);
```

`lobjId`引数には、エクスポートさせるラージオブジェクトのOIDを指定し、`filename`引数には、オペレーティングシステム上の

ファイルのパス名を指定します。 このファイルはサーバではなく、クライアントインタフェースライブラリによって書き込まれることに注意してください。 成功時には1、失敗時には-1が返されます。

32.3.4. 既存のラージオブジェクトのオープン

読み取りまたは書き込みのために既存のラージオブジェクトを開く場合は、以下の関数を呼び出します。

```
int lo_open(PGconn *conn, Oid lobjId, int mode);
```

`lobjId` 引数には開きたいラージオブジェクトのOIDを指定します。 `mode` の各ビットは、そのオブジェクトを読み取りのみ (`INV_READ`)、書き込みのみ (`INV_WRITE`)、またはその両方できるように開くのかを制御するものです。 (これらの定数シンボルは `libpq/libpq-fs.h` ヘッダファイルで定義されています。) 作成していないラージオブジェクトを開くことはできません。 `lo_open` は、`lo_read`、`lo_write`、`lo_lseek`、`lo_tell`、`lo_close` で使用する (非負の) ラージオブジェクト記述子を返します。この記述子は現在のトランザクション期間のみで有効です。 失敗時には-1が返されます。

現時点では、サーバは `INV_WRITE` モードと `INV_READ | INV_WRITE` モードとを区別しません。 どちらの場合でも記述子から読み取り可能です。 しかし、これらのモードと `INV_READ` だけのモードとの間には大きな違いがあります。 `INV_READ` モードでは記述子に書き込むことができません。 そして、読み込んだデータは、このトランザクションや他のトランザクションで後で書き込んだかどうかは関係なく、`lo_open` を実行した時に有効だったトランザクションスナップショットの時点のラージオブジェクトの内容を反映したものになります。 `INV_WRITE` を付けて開いた記述子から読み取ると、現在のトランザクションによる書き込みや他のトランザクションがコミットした書き込みすべてを反映したデータが返されます。 これは、通常の `SELECT SQL` コマンドにおける `REPEATABLE READ` トランザクションの動作と `READ COMMITTED` トランザクションの動作の違いに似ています。

以下に例を示します。

```
inv_fd = lo_open(conn, inv_oid, INV_READ|INV_WRITE);
```

32.3.5. ラージオブジェクトへのデータの書き込み

```
int lo_write(PGconn *conn, int fd, const char *buf, size_t len);
```

この関数は、`len` 長のバイトを、`buf` から `fd` ラージオブジェクト記述子に書き込みます。 `fd` 引数は事前に実行した `lo_open` の戻り値でなければいけません。 実際に書き込まれたバイト数が返されます。 エラーが発生した場合は、負の値を返します。

32.3.6. ラージオブジェクトからのデータの読み込み

```
int lo_read(PGconn *conn, int fd, char *buf, size_t len);
```

この関数は、`len` 長のバイトを、`fd` ラージオブジェクト記述子から `buf` に読み込みます。 `fd` 引数は事前に実行した `lo_open` の戻り値でなければいけません。 実際に読み込まれたバイト数が返されます。 エラーが発生した場合は、負の値を返します。

32.3.7. ラージオブジェクトのシーク

ラージオブジェクト記述子に関連付けされている、現在の読み取りまたは書き込みを行う位置を変更するには、以下の関数を呼び出します。

```
int lo_lseek(PGconn *conn, int fd, int offset, int whence);
```

この関数は `fd` で識別されるラージオブジェクト識別子の現在の位置を指すポインタを、`offset` で指定した新しい位置に変更します。 `whence` に指定可能な値は、`SEEK_SET` (オブジェクトの先頭位置からシーク)、`SEEK_CUR` (現在位置からシーク)、`SEEK_END` (オブジェクトの末尾位置からシーク) のいずれかです。 戻り値は新しい位置ポインタで、エラー時に-1が返されます。

32.3.8. ラージオブジェクトのシーク位置の入手

ラージオブジェクト記述子の現在の読み取り、書き込み位置を入手するには、以下の関数を呼び出します。

```
int lo_tell(PGconn *conn, int fd);
```

エラーが発生した場合は負の値が返されます。

32.3.9. ラージオブジェクトを切り詰める

ラージオブジェクトを指定した長さに切り詰めるには、以下を呼び出します。

```
int lo_truncate(PGcon *conn, int fd, size_t len);
```

これはラージオブジェクト記述子`fd`を`len`長に切り詰めます。`fd`引数は前もって`lo_open`が返したものでなければなりません。`len`が現在のラージオブジェクト長より大きければ、ラージオブジェクトはヌルバイト('¥0')で拡張されます。

ファイルのオフセットは変わりません。

成功時`lo_truncate`は0を返します。 エラー時は負の値を返します。

`lo_truncate`はPostgreSQL 8.3で新規に導入されました。 この関数を古いバージョンのサーバに対して実行した場合は失敗し、負の値が返されます。

32.3.10. ラージオブジェクト記述子を閉じる

以下を呼び出すことでラージオブジェクト記述子を閉ざすことができます。

```
int lo_close(PGconn *conn, int fd);
```

ここで、`fd`は`lo_open`の戻り値であるラージオブジェクト記述子です。 成功すると、`lo_close`は0を返します。 失敗すると、負の値を返します。

開いたままのラージオブジェクト記述子は全てトランザクションの終了時に自動的に閉ざされます。

32.3.11. ラージオブジェクトの削除

データベースからラージオブジェクトを削除するには、以下の関数を呼び出します。

```
int lo_unlink(PGconn *conn, Oid lobjId);
```

`lobjId`引数は削除するラージオブジェクトのOIDを指定します。 成功時に1を、失敗時に-1を返します。

[前のページ](#)
実装機能

[ホーム](#)
[上に戻る](#)

[次のページ](#)
サーバ側の関数

32.4. サーバ側の関数

これまで説明したクライアント側の関数それぞれに対応する、SQLから呼び出し可能なサーバ側の関数があります。 実際、ほとんどのクライアント側の関数に対応するサーバ側の関数に対する単なるインタフェースです。 SQLコマンドからの呼び出しが実際に有用な関数は、`lo_creat`、`lo_create`、`lo_unlink`、`lo_import`、`lo_export`です。 これらの使用例を示します。

```
CREATE TABLE image (
    name          text,
    raster        oid
);

SELECT lo_creat(-1);           -- 新しい空のラージオブジェクトのOIDを返します

SELECT lo_create(43213);      -- OID43213でラージオブジェクトの生成を試行します

SELECT lo_unlink(173454);     -- OID 173454のラージオブジェクトを削除します

INSERT INTO image (name, raster)
VALUES ('beautiful image', lo_import('/etc/motd'));

INSERT INTO image (name, raster) -- 上と同じですが使用するOIDを指定します
VALUES ('beautiful image', lo_import('/etc/motd', 68583));

SELECT lo_export(image.raster, '/tmp/motd') FROM image
WHERE name = 'beautiful image';
```

サーバ側の`lo_import`および`lo_export`関数の動作はクライアント側の関数とかなり異なります。 この2つの関数はサーバのファイルシステム上のファイルの読み書きを、データベースを所有するユーザの権限で行います。 したがって、これらの使用はスーパーユーザに限定されています。 対照的に、クライアント側のインポート関数とエクスポート関数はクライアントのファイルシステム上のファイルをクライアントプログラムの権限で読み書きします。 このクライアント側の関数はスーパーユーザ権限を必要としません。

32.5. サンプルプログラム

[例32-1](#)は、libpqを使ったラージオブジェクトインタフェースの使い方を示すサンプルプログラムです。プログラムの一部はコメントアウトされていますが、読者にわかりやすいようにそのまま残してあります。このプログラムは、ソース配布物内のsrc/test/examplesにあります。

例 32-1. Libpqを使用したラージオブジェクトのサンプルプログラム

```

/*-----
 *
 * testlo.c--
 *   libpqによるラージオブジェクトを使用する試験
 *
 * Copyright (c) 1994, Regents of the University of California
 *
 *-----
 */
#include <stdio.h>
#include "libpq-fe.h"
#include "libpq/libpq-fs.h"

#define BUFSIZE          1024

/*
 * importFile
 *   "in_filename"ファイルをラージオブジェクト"lobjOid"としてデータベースにインポートする。
 */
Oid
importFile(PGconn *conn, char *filename)
{
    Oid          lobjId;
    int          lobj_fd;
    char         buf[BUFSIZE];
    int          nbytes,
               tmp;
    int          fd;

    /*
     * 読み込むファイルを開く
     */
    fd = open(filename, O_RDONLY, 0666);
    if (fd < 0)
    {
        /* error */
        fprintf(stderr, "cannot open unix file %s\n", filename);
    }

    /*
     * ラージオブジェクトを作成する
     */
    lobjId = lo_creat(conn, INV_READ | INV_WRITE);
    if (lobjId == 0)
        fprintf(stderr, "cannot create large object\n");

    lobj_fd = lo_open(conn, lobjId, INV_WRITE);

    /*
     * Unixファイルから読み込み、転置ファイルへ書き出す
     */
    while ((nbytes = read(fd, buf, BUFSIZE)) > 0)
    {
        tmp = lo_write(conn, lobj_fd, buf, nbytes);
        if (tmp < nbytes)
            fprintf(stderr, "error while reading large object\n");
    }

    (void) close(fd);
    (void) lo_close(conn, lobj_fd);

    return lobjId;
}

void
pickout(PGconn *conn, Oid lobjId, int start, int len)
{
    int          lobj_fd;
    char         *buf;

```

```

int         nbytes;
int         nread;

lobj_fd = lo_open(conn, lobjId, INV_READ);
if (lobj_fd < 0)
{
    fprintf(stderr, "cannot open large object %d\\n",
               lobjId);
}

lo_lseek(conn, lobj_fd, start, SEEK_SET);
buf = malloc(len + 1);

nread = 0;
while (len - nread > 0)
{
    nbytes = lo_read(conn, lobj_fd, buf, len - nread);
    buf[nbytes] = ' ';
    fprintf(stderr, ">>> %s", buf);
    nread += nbytes;
}
free(buf);
fprintf(stderr, "\\n");
lo_close(conn, lobj_fd);
}

void
overwrite(PGconn *conn, Oid lobjId, int start, int len)
{
    int         lobj_fd;
    char        *buf;
    int         nbytes;
    int         nwritten;
    int         i;

    lobj_fd = lo_open(conn, lobjId, INV_WRITE);
    if (lobj_fd < 0)
    {
        fprintf(stderr, "cannot open large object %d\\n",
                   lobjId);
    }

    lo_lseek(conn, lobj_fd, start, SEEK_SET);
    buf = malloc(len + 1);

    for (i = 0; i < len; i++)
        buf[i] = 'X';
    buf[i] = ' ';

    nwritten = 0;
    while (len - nwritten > 0)
    {
        nbytes = lo_write(conn, lobj_fd, buf + nwritten, len - nwritten);
        nwritten += nbytes;
    }
    free(buf);
    fprintf(stderr, "\\n");
    lo_close(conn, lobj_fd);
}

/*
 * exportFile
 *   ラージオブジェクト"lobjOid"を"out_filename"ファイルにエクスポートする。
 */
void
exportFile(PGconn *conn, Oid lobjId, char *filename)
{
    int         lobj_fd;
    char        buf[BUFSIZE];
    int         nbytes;
    int         tmp;
    int         fd;

    /*
     *   ラージオブジェクトを作成する
     */
    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0)
    {
        fprintf(stderr, "cannot open large object %d\\n",
                   lobjId);
    }

    /*
     *   書き込むファイルを開く

```



```

    */
    fd = open(filename, O_CREAT | O_WRONLY, 0666);
    if (fd < 0)
    {
        /* エラー時 */
        fprintf(stderr, "cannot open unix file %s\n",
            filename);
    }

    /*
     * 転置ファイルから読み込み、Unixファイルへ書き出す。
     */
    while ((nbytes = lo_read(conn, lobj_fd, buf, BUFSIZE)) > 0)
    {
        tmp = write(fd, buf, nbytes);
        if (tmp < nbytes)
        {
            fprintf(stderr, "error while writing %s\n",
                filename);
        }
    }

    (void) lo_close(conn, lobj_fd);
    (void) close(fd);

    return;
}

void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

int
main(int argc, char **argv)
{
    char        *in_filename,
                *out_filename;
    char        *database;
    Oid          lobjOid;
    PGconn      *conn;
    PGresult     *res;

    if (argc != 4)
    {
        fprintf(stderr, "Usage: %s database_name in_filename out_filename\n",
            argv[0]);
        exit(1);
    }

    database = argv[1];
    in_filename = argv[2];
    out_filename = argv[3];

    /*
     * 接続を設定する
     */
    conn = PQsetdb(NULL, NULL, NULL, NULL, database);

    /* バックエンドとの接続が成功したかどうか確認する */
    if (PQstatus(conn) == CONNECTION_BAD)
    {
        fprintf(stderr, "Connection to database '%s' failed.\n", database);
        fprintf(stderr, "%s", PQerrorMessage(conn));
        exit_nicely(conn);
    }

    res = PQexec(conn, "begin");
    PQclear(res);

    printf("importing file %s\n", in_filename);
    /* lobjOid = importFile(conn, in_filename); */
    lobjOid = lo_import(conn, in_filename);
    /*
    printf("as large object %d.\n", lobjOid);

    printf("picking out bytes 1000-2000 of the large object\n");
    pickout(conn, lobjOid, 1000, 1000);

    printf("overwriting bytes 1000-2000 of the large object with X's\n");
    overwrite(conn, lobjOid, 1000, 1000);
    */

    printf("exporting large object to file %s\n", out_filename);
    /* exportFile(conn, lobjOid, out_filename); */

```

```
    lo_export(conn, lobjOid, out_filename);

    res = PQexec(conn, "end");
    PQclear(res);
    PQfinish(conn);
    exit(0);
}
```

[前のページ](#)

サーバ側の関数

[ホーム](#)
[上に戻る](#)

[次のページ](#)

ECPG - C言語による埋め込みSQL

第 33章 ECPG - C言語による埋め込みSQL

目次

- 33.1. [概念](#)
- 33.2. [データベース接続の管理](#)
 - 33.2.1. [データベースサーバへの接続](#)
 - 33.2.2. [接続の選択](#)
 - 33.2.3. [接続を閉じる](#)
- 33.3. [SQLコマンドの実行](#)
 - 33.3.1. [SQL文の実行](#)
 - 33.3.2. [カーソルの使用](#)
 - 33.3.3. [トランザクションの管理](#)
 - 33.3.4. [Prepared Statements](#)
- 33.4. [ホスト変数の使用](#)
 - 33.4.1. [概要](#)
 - 33.4.2. [宣言セクション](#)
 - 33.4.3. [クエリ実行結果の受け取り](#)
 - 33.4.4. [データ型の対応](#)
 - 33.4.5. [非プリミティブSQLデータ型の扱い方](#)
 - 33.4.6. [指示子](#)
- 33.5. [動的SQL](#)
 - 33.5.1. [結果セットを伴わないSQL文の実行](#)
 - 33.5.2. [入力パラメータを伴うSQL文の実行](#)
 - 33.5.3. [結果セットを返却するSQL文の実行](#)
- 33.6. [pgtypes ライブラリ](#)
 - 33.6.1. [numeric 型](#)
 - 33.6.2. [日付型](#)
 - 33.6.3. [timestamp型](#)
 - 33.6.4. [interval型](#)
 - 33.6.5. [decimal型](#)
 - 33.6.6. [pgtypeslibのerrno値](#)
 - 33.6.7. [pgtypeslibの特殊な定数](#)
- 33.7. [記述子領域の使用](#)
 - 33.7.1. [名前付きSQL記述子領域](#)
 - 33.7.2. [SQLDA記述子領域](#)
- 33.8. [エラー処理](#)
 - 33.8.1. [コールバックの設定](#)
 - 33.8.2. [sqlca](#)
 - 33.8.3. [SQLSTATE対SQLCODE](#)
- 33.9. [プリプロセッサ指示子](#)
 - 33.9.1. [ファイルのインクルード](#)
 - 33.9.2. [defineおよびundef指示子](#)
 - 33.9.3. [ifdef、ifndef、else、elif、endif指示子](#)
- 33.10. [埋め込みSQLプログラムの処理](#)
- 33.11. [ライブラリ関数](#)
- 33.12. [ラージオブジェクト](#)
- 33.13. [C++アプリケーション](#)

33.13.1. [ホスト変数のスコープ](#)

33.13.2. [外部のCモジュールを用いたC++アプリケーションの開発](#)

33.14. [埋め込みSQLコマンド](#)

[ALLOCATE DESCRIPTOR](#) -- SQL記述子領域を割り当てます。

[CONNECT](#) -- データベース接続を確立します。

[DEALLOCATE DESCRIPTOR](#) -- SQL記述子領域の割り当てを解除します。

[DECLARE](#) -- カーソルを定義します。

[DESCRIBE](#) -- プリペアド・ステートメントまたは結果セットに関する情報を入手します。

[DISCONNECT](#) -- データベース接続を終了します。

[EXECUTE IMMEDIATE](#) -- SQL文を動的にプリペアし、実行します。

[GET DESCRIPTOR](#) -- SQL記述子領域から情報を入手します。

[OPEN](#) -- 動的カーソルを開きます。

[PREPARE](#) -- 実行のためにSQL文をプリペアします。

[SET AUTOCOMMIT](#) -- 現在のセッションの自動コミット動作を設定します。

[SET CONNECTION](#) -- データベース接続を選択します。

[SET DESCRIPTOR](#) -- SQL記述子領域に情報を設定します。

[TYPE](#) -- 新しいデータ型を定義します。

[VAR](#) -- 変数を定義します。

[WHENEVER](#) -- SQL文により特定の分類の条件が発生する時に行う動作を指定します。

33.15. [Informix互換モード](#)

33.15.1. [追加の型](#)

33.15.2. [追加または存在しない埋め込みSQL文](#)

33.15.3. [Informix互換SQLDA記述子領域](#)

33.15.4. [追加関数](#)

33.15.5. [追加の定数](#)

33.16. [内部](#)

本章では、PostgreSQLの埋め込みSQLパッケージについて説明します。このパッケージはCとC++言語で作成されました。作者はLinus Tolke (<linus@epact.se>) とMichael Meskes (<meskes@postgresql.org>) です。元々これはCで動作するように作成されました。C++でも動作しますが、C++の構文すべてはまだ認識できません。

本書は完全なものではありません。しかし、このインタフェースは標準化されているので、SQLに関するその他の資料から追加情報を入手できます。

[前のページ](#)

サンプルプログラム

[ホーム](#)

[上に戻る](#)

[次のページ](#)

概念

33.1. 概念

埋め込みSQLプログラムは通常のプログラミング言語（ここではC）で記述されたコードで、特別にマークされたセクション内のSQLコマンドとともに使用されます。このプログラムを構築するには、まずソースコード（*.pgc）を埋め込みSQLプリプロセッサに渡します。ソースコードは、プリプロセッサによって通常のCプログラム（*.c）に変換され、その後Cコンパイラによって処理されます。（コンパイルとリンクの詳細については [項33.10](#) を参照してください）変換されたECPGアプリケーションは、libpqライブラリにある関数を埋め込みSQLライブラリ（ecpglib）を介して呼び出し、通常のフロントエンド・バックエンドプロトコルを使ってPostgreSQLサーバと通信します。

CコードからSQLコマンドを扱う場合は、埋め込みSQLの方が他の手法よりも有効です。まず、埋め込みSQLはCプログラムの変数との面倒な双方間の情報移行を処理してくれます。さらに、プログラム内のSQLコードは構築時に正確な構文になっているかどうか検査されます。また、C言語での埋め込みSQLは標準SQLで既に定義されており、他の様々なSQLデータベースシステムでサポートされています。PostgreSQLの実装は可能な限りこの標準に準拠するよう設計されています。また通常の場合、他のSQLデータベース用に作成された埋め込みSQLプログラムを比較的簡単にPostgreSQLへ移植することができます。

先に述べた通り、埋め込みSQLインタフェース用のプログラムは、通常のCプログラムに、データベース関連処理を行うための特別なコードを加えたものです。この特別なコードは、常に、次のような形式になっています。

```
EXEC SQL ...;
```

このSQL文は、構文上でC言語の文の置き換えとなります。SQL文によりますが、グローバルレベル、または関数内で記述することができます。埋め込みSQL文における大文字小文字の区別の有無は、Cコードではなく、通常のSQLコードの規則に従います。

以下の節で、すべての埋め込みSQL文について説明します。

33.2. データベース接続の管理

この節では、データベース接続の開始、終了、および切り替え方について解説します。

33.2.1. データベースサーバへの接続

以下のSQL文を使用して、データベースへ接続します。

```
EXEC SQL CONNECT TO target [AS connection-name] [USER user-name];
```

*target*は以下の方法で指定されます。

- *dbname*[@*hostname*][:*port*]
- tcp:postgresql://*hostname*[:*port*]/[*dbname*][?*options*]
- unix:postgresql://*hostname*[:*port*]/[*dbname*][?*options*]
- 上の形式のいずれかを含むSQL文字列リテラル。
- 上の形式のいずれかを含む文字変数への参照。
- DEFAULT

接続対象をリテラル（つまり、変数を参照しない形）で指定し、その値を引用符でくくらなかった場合、大文字小文字の区別に関して通常のSQLの規則が適用されます。また、この場合、必要に応じて個々のパラメータを二重引用符で別々にくくることができます。実際には、おそらく（単一引用符でくくられた）文字列リテラルもしくは変数の参照を使用した方がエラーをより防止することができます。DEFAULT接続対象は、デフォルトデータベース、デフォルトのユーザ名で接続を初期化します。この場合は、ユーザ名と接続名を分けて指定することができません。

ユーザ名を指定するには、別の方法もあります。

- *username*
- *username/password*
- *username IDENTIFIED BY password*
- *username USING password*

これまで同様、*username*と*password*は、SQL識別子、SQL文字列リテラル、文字型変数への参照を取ることができます。

1つのプログラム内で複数の接続を処理する場合には、*connection-name*を使用します。プログラムで1つしか接続を使わない場合は省略して構いません。最も最近に開かれた接続が現在の接続になり、SQL文を実行しようとする時にデフォルトでこの接続が使用されます（本章の後で説明します）。

以下にCONNECT文について、数例を示します。

```
EXEC SQL CONNECT TO mydb@sql.mydomain.com;

EXEC SQL CONNECT TO unix:postgresql://sql.mydomain.com/mydb AS myconnection USER john;

EXEC SQL BEGIN DECLARE SECTION;
const char *target = "mydb@sql.mydomain.com";
const char *user = "john";
const char *passwd = "secret";
EXEC SQL END DECLARE SECTION;
...
EXEC SQL CONNECT TO :target USER :user USING :passwd;
/* or EXEC SQL CONNECT TO :target USER :user/:passwd; */
```

最後の形式では、文字変数参照として上を参照する変数を使用しています。後の節で、接頭辞にコロンの持つ場合のSQL文内でのC変数の使用方法について説明します。

接続対象の書式は標準SQLでは規定されていないことに注意してください。そのため、移植可能なアプリケーションを開発したいのであれば、上の例の最後の方法を基にして、接続対象文字列をどこかにカプセル化してください。

33.2.2. 接続の選択

前節で示したSQL文は現在の接続、つまり、最も最近に開いた接続上で実行されます。複数の接続を管理する必要があるアプリケーションでは、これを処理する2つの方法があります。

1つ目の選択肢は、各SQL文で明示的に接続を選択することです。以下に例を示します。

```
EXEC SQL AT connection-name SELECT ...;
```

アプリケーションが複数の接続を不特定な順番で使用する必要がある場合、この選択肢は特に適しています。

アプリケーションの実行に複数スレッドを使用する場合、スレッド間で接続を同時に共有できません。接続へのアクセスを（ミューテックスを使用して）明示的に制御するか、または各スレッド用の接続を使用するかを行わなければなりません。各スレッドが個別の接続を使用する場合、AT句を使用してそのスレッドが使用する接続を指定する必要があります。

2番目の選択肢は、現在の接続を切り替えるSQL文を実行することです。以下のSQL文です。

```
EXEC SQL SET CONNECTION connection-name;
```

多くのSQL文を同一接続に対して使用する場合、この選択肢は特に便利です。これはスレッドを考慮していません。

以下に複数のデータベースコネクションを管理しているプログラムの例を示します。

```
#include <stdio.h>

EXEC SQL BEGIN DECLARE SECTION;
    char dbname[1024];
EXEC SQL END DECLARE SECTION;

int
main()
{
    EXEC SQL CONNECT TO testdb1 AS con1 USER testuser;
    EXEC SQL CONNECT TO testdb2 AS con2 USER testuser;
    EXEC SQL CONNECT TO testdb3 AS con3 USER testuser;

    /* This query would be executed in the last opened database "testdb3". */
    EXEC SQL SELECT current_database() INTO :dbname;
    printf("current=%s (should be testdb3)\n", dbname);

    /* Using "AT" to run a query in "testdb2" */
    EXEC SQL AT con2 SELECT current_database() INTO :dbname;
    printf("current=%s (should be testdb2)\n", dbname);

    /* Switch the current connection to "testdb1". */
    EXEC SQL SET CONNECTION con1;

    EXEC SQL SELECT current_database() INTO :dbname;
    printf("current=%s (should be testdb1)\n", dbname);

    EXEC SQL DISCONNECT ALL;
    return 0;
}
```

この例は、次のような出力を生成します。

```
current=testdb3 (should be testdb3)
current=testdb2 (should be testdb2)
current=testdb1 (should be testdb1)
```

33.2.3. 接続を閉じる

接続を閉じるには以下のSQL文を使用します。

```
EXEC SQL DISCONNECT [connection];
```

connection は以下の方法で指定されます。

- *connection-name*
- DEFAULT
- CURRENT
- ALL

接続名の指定がなければ、現在の接続が閉じられます。

アプリケーションでは、過去に開いたすべての接続を明示的に閉じることを推奨します。

[前のページ](#)
概念

[ホーム](#)
[上に戻る](#)

[次のページ](#)
SQLコマンドの実行

33.3. SQL コマンドの実行

すべての SQL コマンドは、埋め込み SQL アプリケーション内で実行できます。以下に例をいくつか示します。

33.3.1. SQL 文の実行

テーブルの作成:

```
EXEC SQL CREATE TABLE foo (number integer, ascii char(16));
EXEC SQL CREATE UNIQUE INDEX num1 ON foo(number);
EXEC SQL COMMIT;
```

行の挿入:

```
EXEC SQL INSERT INTO foo (number, ascii) VALUES (9999, 'doodad');
EXEC SQL COMMIT;
```

行の削除:

```
EXEC SQL DELETE FROM foo WHERE number = 9999;
EXEC SQL COMMIT;
```

行の更新:

```
EXEC SQL UPDATE foo
SET ascii = 'foobar'
WHERE number = 9999;
EXEC SQL COMMIT;
```

単一の行を返す `SELECT` 文は、同様に `EXEC SQL` を用いて直接実行することができます。複数の行を扱うためには、アプリケーションはカーソルを使わなければなりません; [項33.3.2](#) を参照してください。(特殊なケースでは、アプリケーションは複数の行をホスト変数の配列に一度に読み込むことができます; [項33.4.4.3.1](#) を参照してください)

単一行の検索:

```
EXEC SQL SELECT foo INTO :FooBar FROM table1 WHERE ascii = 'doodad';
```

同様に、設定パラメータは `SHOW` コマンドによって取得することができます:

```
EXEC SQL SHOW search_path INTO :var;
```

`:something` という形のトークンはホスト変数です。つまり、C プログラム内の変数を参照するものです。これについては [項33.4](#) で説明します。

33.3.2. カーソルの使用

複数行の結果セットを受け取るためには、アプリケーションはカーソルを定義し、必要に応じてレコードを一行ずつ取り込む必要があります。カーソルを使った処理は、カーソルの宣言、カーソルのオープン、カーソルからの `FETCH`、カーソルのクローズという流れになります。

カーソルを用いた `SELECT`:

```
EXEC SQL DECLARE foo_bar CURSOR FOR
SELECT number, ascii FROM foo
ORDER BY ascii;
EXEC SQL OPEN foo_bar;
EXEC SQL FETCH foo_bar INTO :FooBar, DooDad;
...
EXEC SQL CLOSE foo_bar;
EXEC SQL COMMIT;
```

カーソルの宣言の詳細については [DECLARE](#) を、`FETCH` コマンドの詳細については [FETCH](#) を参照してください。

注意: ECPG の `DECLARE` コマンド自身は、PostgreSQL バックエンドに送られる SQL 文を実行しません。 `OPEN` コマンドが実行された段階で、バックエンド内部で (`DECLARE` コマンドで宣言された) カーソルが開かれます。

33.3.3. トランザクションの管理

デフォルトモードでは、SQL文はEXEC SQL COMMITが発行されることによってのみコミットされます。 埋め込みSQLインタフェースでも、ecpg コマンド（以下を参照）の-tコマンドラインオプション、あるいは EXEC SQL SET AUTOCOMMIT TO ON 文によって（libpqの振舞いに似た）トランザクションの自動コミットをサポートしています。 自動コミットモードでは、問い合わせが明示的なトランザクションブロックの内部にある場合を除き、すべての問い合わせが自動的にコミットされます。 自動コミットモードは、EXEC SQL SET AUTOCOMMIT TO OFFを使用して明示的に無効にすることができます。

以下のトランザクション管理コマンドを使用することができます：

```
EXEC SQL COMMIT
```

実行中のトランザクションのコミット。

```
EXEC SQL ROLLBACK
```

実行中のトランザクションのロールバック。

```
EXEC SQL SET AUTOCOMMIT TO ON
```

自動コミットモードの有効化。

```
SET AUTOCOMMIT TO OFF
```

自動コミットモードの無効化。デフォルト状態。

33.3.4. Prepared Statements

SQL文に渡す値がコンパイル時に決まらない場合、または同じSQL文を何度も実行する場合、プリペARED・ステートメントが便利です。

SQL文は PREPARE コマンドを使ってプリペアします。 まだ決まっていない値については、プレースホルダ "?" を使います：

```
EXEC SQL PREPARE stmt1 FROM "SELECT oid, datname FROM pg_database WHERE oid = ?";
```

SQL文が一行のみの結果を返却する場合には、アプリケーションはSQL文を PREPARE した後、USING を用いてプレースホルダに実際の値を与えて EXECUTE を実行することができます。

```
EXEC SQL EXECUTE stmt1 INTO :dboid, :dbname USING 1;
```

SQL文が複数の行を返却する場合には、アプリケーションはプリペARED・ステートメントの宣言に対応したカーソルを利用することができます。 入力パラメータを設定するために、カーソルは USING とともに開かれなければなりません：

```
EXEC SQL PREPARE stmt1 FROM "SELECT oid, datname FROM pg_database WHERE oid > ?";
EXEC SQL DECLARE foo_bar CURSOR FOR stmt1;
```

```
/* when end of result set reached, break out of while loop */
EXEC SQL WHENEVER NOT FOUND DO BREAK;
```

```
EXEC SQL OPEN foo_bar USING 100;
...
while (1)
{
    EXEC SQL FETCH NEXT FROM foo_bar INTO :dboid, :dbname;
    ...
}
EXEC SQL CLOSE foo_bar;
```

プリペARED・ステートメントをこれ以上必要としなくなったら、解放処理をしなければなりません：

```
EXEC SQL DEALLOCATE PREPARE name;
```

PREPARE についての詳細は [PREPARE](#) を参照してください。 また、プレースホルダと入力パラメータの利用についての詳細は [項33.5](#) を参照してください。

[前のページ](#)

データベース接続の管理

[ホーム](#)

[上に戻る](#)

[次のページ](#)

ホスト変数の使用

33.4. ホスト変数の使用

[項33.3](#)では、埋め込みSQLプログラムでどのようにSQL文を実行するのかについて説明しました。このSQL文の中には固定値しか使用しないものや、ユーザが指定する値をSQL文の中に挿入する手段を提供しないもの、問い合わせが返す値をプログラムで処理する手段を提供しないものがありました。この種のSQL文は実際のアプリケーションでは役に立ちません。本節では、ホスト変数という単純な機構を使用した、Cプログラムと埋め込みSQL文との間でデータをやり取りする方法を詳細に説明します。埋め込みSQLプログラムでは、SQL文をホスト言語となるCプログラムコードにおけるゲスト *guests* とみなします。したがって、Cプログラムの変数はホスト変数と呼ばれます。

PostgreSQLバックエンドとECPGアプリケーションの間で値をやり取りするその他の方法は、[項33.7](#) で説明されているSQLデスク립タを使う方法です。

33.4.1. 概要

埋め込みSQLにおけるCプログラムとSQL文との間でのデータのやり取りは特に単純です。値に適切な引用符を付与するといった、様々な複雑な処理を伴う、プログラムにデータを文中に貼り付けさせるという方法はなく、単にSQL文の中に、先頭にコロンを付けたC変数名を書くだけです。以下に例を示します。

```
EXEC SQL INSERT INTO sometable VALUES (:v1, 'foo', :v2);
```

このSQL文は、`v1`と`v2`という2つのC変数を参照し、また、通常のSQL文字列リテラルも使用しています。これは、使用できるデータの種類の1つだけという制限がないことを表しています。

SQL文内にCの変数を挿入するこの様式は、SQL文で値式が想定されている所であればどこでも動作します。

33.4.2. 宣言セクション

例えば問い合わせ内のパラメータとして、プログラムからデータベースヘデータを渡す、もしくは、データベースからプログラムヘデータを渡すためには、このようなデータを含むように意図されたC変数を、埋め込みSQLプリプロセッサが管理できるように、特殊な印のついたセクションで宣言する必要があります。

このセクションは以下で始まります。

```
EXEC SQL BEGIN DECLARE SECTION;
```

そして、以下で終わります。

```
EXEC SQL END DECLARE SECTION;
```

この行の間は、以下のような通常のC変数宣言でなければなりません。

```
int    x = 4;
char   foo[16], bar[16];
```

見てわかるとおり、省略可能ですが、変数に初期値を代入することができます。変数のスコープはプログラム内の宣言セクションの場所により決まります。また、以下のような暗黙的に宣言セクションを生成する構文を使って変数を宣言することもできます。

```
EXEC SQL int i = 4;
```

プログラム内に複数の宣言セクションを持たせることができます。

また、宣言は普通のC変数としてそのまま出力ファイルに出力されます。ですので、これらを再度宣言する必要はありません。通常、SQLコマンドで使用する予定がない変数はこの特別なセクションの外側で宣言されます。

構造体やunionの定義もまた、`DECLARE`セクションの内側で表す必要があります。さもないと、プリプロセッサはその定義が不明であるために、これらの型を扱うことができません。

33.4.3. クエリ実行結果の受け取り

ここまでで、プログラムで生成したデータをSQLコマンドに渡すことができるようになりました。しかし、どのように問い合わせの結果を取り出すのでしょうか？この目的のために、埋め込みSQLでは、通常の`SELECT`と`FETCH`を派生した、特殊なコマンドを提供しています。これらのコマンドは特別な`INTO`句を持ち、ここで返された値をどのホスト変数に格納すればよいかを指定します。`SELECT`は単一行を返却するクエリに使用され、`FETCH`は複数の業を返却するクエリにおいてカーソルとともに使用さ

れます。

以下にサンプルを示します。

```
/*
 * assume this table:
 * CREATE TABLE test1 (a int, b varchar(50));
 */

EXEC SQL BEGIN DECLARE SECTION;
int v1;
VARCHAR v2;
EXEC SQL END DECLARE SECTION;

...

EXEC SQL SELECT a, b INTO :v1, :v2 FROM test;
```

INTO 句が選択リストとFROM 句の間に現れます。 選択リスト内の要素数とINTO直後のリスト（目的リストとも呼ばれます）の要素数は等しくなければなりません。

以下にFETCH コマンドの使用例を示します。

```
EXEC SQL BEGIN DECLARE SECTION;
int v1;
VARCHAR v2;
EXEC SQL END DECLARE SECTION;

...

EXEC SQL DECLARE foo CURSOR FOR SELECT a, b FROM test;

...

do
{
    ...
    EXEC SQL FETCH NEXT FROM foo INTO :v1, :v2;
    ...
} while (...);
```

ここでは、INTO 句が通常のすべての句の後ろに現れています。

33.4.4. データ型の対応

ECPGアプリケーションがPostgreSQLバックエンドとCアプリケーションの間で値をやり取りする際、例えばサーバからクエリの結果を受け取る、または入力パラメータとともにSQL文を実行する場合、それらの値はPostgreSQLのデータ型とホスト言語の変数の型（具体的にはC言語のデータ型）の間で変換される必要があります。 ECPGの重要な点のひとつは、ほとんどの場合においてECPGがこれらを自動的に扱うということです。

この点において、2つのデータ型があります: いくつかの単純なPostgreSQLのデータ型、integer や text などは、アプリケーションから直接読んだり書いたりすることができます。 その他のPostgreSQLのデータ型、timestamp や numeric などは、特別なライブラリ関数によってしかアクセスすることができません; [項33.4.4.2](#) を参照してください。

[表33-1](#) には、PostgreSQLのどのデータ型がC言語のデータ型に対応するかを示されています。 与えられたPostgreSQLのデータ型へ値を書き込みまたは読み込みしたい場合には、対応するC言語のデータ型の変数を宣言セクションにおいて宣言しなければなりません。

表 33-1. PostgreSQLデータ型とC言語変数型の対応

PostgreSQL data type	Host variable type
smallint	short
integer	int
bigint	long long int
decimal	decimal [a]
numeric	numeric [a]
real	float
double precision	double
smallserial	short
serial	int
bigserial	long long int

oid	unsigned int
character(<i>n</i>), varchar(<i>n</i>), text	char[<i>n</i> +1], VARCHAR[<i>n</i> +1] [b]
name	char[NAMEDATALEN]
timestamp	timestamp [a]
interval	interval [a]
date	date [a]
boolean	bool [c]
注意: a. この型は特別なライブラリ関数を通してのみアクセスできます; 項33.4.4.2 を参照。 b. <code>ecpglib.h</code> で宣言。 c. ネイティブでなければ <code>ecpglib.h</code> で宣言。	

33.4.4.1. 文字列の処理

`varchar` and `text`のような文字列のデータ型を扱うため、ホスト変数を宣言するための2つの方法があります。

ひとつは `char` の配列 `char[]` を使うことで、C言語において文字列データを扱うもっとも一般的な方法です。

```
EXEC SQL BEGIN DECLARE SECTION;
char str[50];
EXEC SQL END DECLARE SECTION;
```

文字列の長さについて、自分自身で気を付けておく必要があります。上記のホスト変数を49文字以上の文字列を返すクエリのターゲット変数として使った場合、バッファオーバーフローが発生します。

その他の方法は、ECPGによって提供される特殊なデータ型 `VARCHAR` を使う方法です。 `VARCHAR` の配列の定義は、すべての変数が名前の付いた `struct` に変換されます。 以下のような宣言は:

```
VARCHAR var[180];
```

次のように変換されます:

```
struct varchar_var { int len; char arr[180]; } var;
```

メンバー変数 `arr` は終端のゼロの1バイトを含む文字列を保持します。 よって、文字列を `VARCHAR` ホスト変数に格納する場合には、ホスト変数はゼロ終端を含んだ長さで宣言されなければなりません。 メンバー変数 `len` は `arr` に格納された文字列のゼロ終端を含まない長さを保持します。 ホスト変数をクエリの入力として使用する際、`strlen(arr)` と `len` が違った場合には短いものが使用されます。

2つ以上の `VARCHAR` ホスト変数は、一行で定義することはできません。 以下のコードは `ecpg` プリプロセッサを混乱させるでしょう:

```
VARCHAR v1[128], v2[128]; /* WRONG */
```

2つの変数は以下のように別々の行で定義されなければなりません:

```
VARCHAR v1[128];
VARCHAR v2[128];
```

`VARCHAR` は大文字でも小文字でも記述することができますが、混在して記述することはできません。

`char` と `VARCHAR` ホスト変数は、他のSQLのデータ型の値を文字列表現として保持することもできます。

33.4.4.2. 特殊なデータ型へのアクセス

ECPGには、PostgreSQLサーバからのいくつかの特殊なデータ型とやりとりするための特殊なデータ型があります。 特に、`numeric`, `decimal`, `date`, `timestamp`, `interval` 型へのサポートを実装しています。 これらのデータ型は複雑な内部構造を持つため、ホスト変数のプリミティブ型 (`int`, `long long int`, または `char[]`) に対応させることはできません。 アプリケーションは特別な型としてホスト変数を宣言し、`pgtypes`ライブラリ内の関数を使ってアクセスすることで、これらの型を扱います。 [項33.6](#) で詳細を解説される`pgtypes`ライブラリは、例えばタイムスタンプにインターバルを加算する際にクエリをSQLサーバに送らずに済ますような、これらの型を扱うための基本的な関数を含んでいます。

移行のサブセクションは、これらの特殊なデータ型を説明します。 `pgtypes`ライブラリ関数についての詳細は [項33.6](#) を参照してください。

33.4.4.2.1. timestamp, date

以下は、`timestamp` 変数をECPGホストアプリケーションで扱う典型的なパターンです。

最初に、プログラムは `timestamp` 型のためのヘッダファイルをインクルードする必要があります：

```
#include <pgtypes_timestamp.h>
```

次に、宣言セクションで `timestamp` 型のホスト変数を宣言します：

```
EXEC SQL BEGIN DECLARE SECTION;
timestamp ts;
EXEC SQL END DECLARE SECTION;
```

そして、ホスト変数へ値を読み込んだら、`pgtypes`ライブラリ関数を使って処理をします。以降の例では、`timestamp` の値は `PGTYPEStimestamp_to_asc()` 関数によって `text` (ASCII) 形式に変換されます：

```
EXEC SQL SELECT now()::timestamp INTO :ts;

printf("ts = %s\n", PGTYPEStimestamp_to_asc(ts));
```

This example will show some result like following:

```
ts = 2010-06-27 18:03:56.949343
```

また、`DATE`型も同じ方法で扱うことができます。プログラムは `pgtypes_date.h` をインクルードし、ホスト変数を `date` 型として宣言し、`PGTYPESdate_to_asc()` 関数によって `DATE` の値を `text` 形式に変換します。`pgtypes`ライブラリ関数についての詳細は、[項33.6](#) を参照してください。

33.4.4.2.2. interval

`interval` 型の扱いは `timestamp` や `date` 型と似ています。但し、`interval` 型の値のために明示的にメモリを確保する必要があります。言い換えると、この変数のためのメモリ領域はスタックではなくヒープ上に確保されます。

以下にプログラム例を示します：

```
#include <stdio.h>
#include <stdlib.h>
#include <pgtypes_interval.h>

int
main(void)
{
    EXEC SQL BEGIN DECLARE SECTION;
        interval *in;
    EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO testdb;

    in = PGYPESinterval_new();
    EXEC SQL SELECT '1 min'::interval INTO :in;
    printf("interval = %s\n", PGYPESinterval_to_asc(in));
    PGYPESinterval_free(in);

    EXEC SQL COMMIT;
    EXEC SQL DISCONNECT ALL;
    return 0;
}
```

33.4.4.2.3. numeric, decimal

`numeric` と `decimal` 型の扱いは `interval` 型と似ています：ポインタ宣言を必要とし、ヒープメモリを確保する必要があります。また、`pgtypes`ライブラリ関数を使って変数にアクセスします。`pgtypes`ライブラリ関数の詳細については、[項33.6](#) を参照してください。

`decimal` 型に対する専用の関数は提供されていません。アプリケーションは処理を行うために `pgtypes`ライブラリ関数を使って `numeric` 変数に変換する必要があります。

以下に `numeric` および `decimal` 型の変数の処理の例を示します。

```
#include <stdio.h>
#include <stdlib.h>
#include <pgtypes_numeric.h>

EXEC SQL WHENEVER SQLERROR STOP;

int
main(void)
{
    EXEC SQL BEGIN DECLARE SECTION;
```

```

    numeric *num;
    numeric *num2;
    decimal *dec;
EXEC SQL END DECLARE SECTION;

EXEC SQL CONNECT TO testdb;

num = PGTYPEsnumeric_new();
dec = PGYPESdecimal_new();

EXEC SQL SELECT 12.345::numeric(4,2), 23.456::decimal(4,2) INTO :num, :dec;

printf("numeric = %s\n", PGTYPEsnumeric_to_asc(num, 0));
printf("numeric = %s\n", PGTYPEsnumeric_to_asc(num, 1));
printf("numeric = %s\n", PGTYPEsnumeric_to_asc(num, 2));

/* Convert decimal to numeric to show a decimal value. */
num2 = PGTYPEsnumeric_new();
PGTYPEsnumeric_from_decimal(dec, num2);

printf("decimal = %s\n", PGTYPEsnumeric_to_asc(num2, 0));
printf("decimal = %s\n", PGTYPEsnumeric_to_asc(num2, 1));
printf("decimal = %s\n", PGTYPEsnumeric_to_asc(num2, 2));

PGTYPEsnumeric_free(num2);
PGYPESdecimal_free(dec);
PGTYPEsnumeric_free(num);

EXEC SQL COMMIT;
EXEC SQL DISCONNECT ALL;
return 0;
}

```

33.4.4.3. 非プリミティブ型のホスト変数

ホスト変数として、配列、typedef、構造体およびポインタも使うことができます。

33.4.4.3.1. 配列

ホスト変数としての配列の使い方には二通りの利用方法があります。一つ目の使い方は、[項33.4.4.1](#) で説明されたように `char[]` または `VARCHAR[]` の何らかのテキスト文字列を保持するための方法です。二つ目の使い方は、カーソルを用いずに複数行を返却するクエリ結果を受け取るために使う方法です。配列を使わない場合、複数行からなるクエリの実行結果を処理するには、カーソルと `FETCH` コマンドを使用する必要があります。しかし、配列のホスト変数を使うと、複数行を一括して受け取ることができます。配列の長さはすべての行を受け入れられるように定義されなければなりません。でなければバッファオーバーフローが発生するでしょう。

以下の例は `pg_database` システムテーブルをスキャンし、利用可能なデータベースのすべてのOIDとデータベース名を表示します：

```

int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    int dbid[8];
    char dbname[8][16];
    int i;
EXEC SQL END DECLARE SECTION;

memset(dbname, 0, sizeof(char) * 16 * 8);
memset(dbid, 0, sizeof(int) * 8);

EXEC SQL CONNECT TO testdb;

/* Retrieve multiple rows into arrays at once. */
EXEC SQL SELECT oid,datname INTO :dbid, :dbname FROM pg_database;

for (i = 0; i < 8; i++)
    printf("oid=%d, dbname=%s\n", dbid[i], dbname[i]);

EXEC SQL COMMIT;
EXEC SQL DISCONNECT ALL;
return 0;
}

```

この例は、以下の結果を表示します。（実際の値はローカルな環境に依存します）

```

oid=1, dbname=template1
oid=11510, dbname=template0
oid=11511, dbname=postgres
oid=313780, dbname=testdb
oid=0, dbname=

```

```
oid=0, dbname=  
oid=0, dbname=
```

33.4.4.3.2. 構造体

メンバー変数の名前がクエリ結果のカラム名に合致する構造体は、複数のカラムを一括して受け取るために利用することができます。構造体は複数のカラムの値を単一のホスト変数で扱うことを可能にします。

以下の例は、pg_database システムテーブルおよび pg_database_size() 関数を使って、利用可能なデータベースのOID、名前、サイズを取得します。この例では、メンバー変数の名前が SELECT 結果の各カラムに合致する構造体 dbinfo_t が、複数のホスト変数に格納することなく FETCH 文の一行の結果を受け取るために使用されています。

```
EXEC SQL BEGIN DECLARE SECTION;  
typedef struct  
{  
    int oid;  
    char datname[65];  
    long long int size;  
} dbinfo_t;  
  
dbinfo_t dbval;  
EXEC SQL END DECLARE SECTION;  
  
memset(&dbval, 0, sizeof(dbinfo_t));  
  
EXEC SQL DECLARE curl CURSOR FOR SELECT oid, datname, pg_database_size(oid) AS size FROM pg_database;  
EXEC SQL OPEN curl;  
  
/* when end of result set reached, break out of while loop */  
EXEC SQL WHENEVER NOT FOUND DO BREAK;  
  
while (1)  
{  
    /* Fetch multiple columns into one structure. */  
    EXEC SQL FETCH FROM curl INTO :dbval;  
  
    /* Print members of the structure. */  
    printf("oid=%d, datname=%s, size=%lld¥n", dbval.oid, dbval.datname, dbval.size);  
}  
  
EXEC SQL CLOSE curl;
```

この例は、次の結果を示します（実際の値はローカルな環境に依存します）

```
oid=1, datname=template1, size=4324580  
oid=11510, datname=template0, size=4243460  
oid=11511, datname=postgres, size=4324580  
oid=313780, datname=testdb, size=8183012
```

構造体のホスト変数は、多数のカラムを構造体のフィールドとして"吸収"します。追加のカラムは他のホスト変数に割り当てることができます。例えば、上記のプログラムは構造体に含まれない size 変数を使って以下のように書き換えることができます。

```
EXEC SQL BEGIN DECLARE SECTION;  
typedef struct  
{  
    int oid;  
    char datname[65];  
} dbinfo_t;  
  
dbinfo_t dbval;  
long long int size;  
EXEC SQL END DECLARE SECTION;  
  
memset(&dbval, 0, sizeof(dbinfo_t));  
  
EXEC SQL DECLARE curl CURSOR FOR SELECT oid, datname, pg_database_size(oid) AS size FROM pg_database;  
EXEC SQL OPEN curl;  
  
/* when end of result set reached, break out of while loop */  
EXEC SQL WHENEVER NOT FOUND DO BREAK;  
  
while (1)  
{  
    /* Fetch multiple columns into one structure. */  
    EXEC SQL FETCH FROM curl INTO :dbval, :size;  
  
    /* Print members of the structure. */  
    printf("oid=%d, datname=%s, size=%lld¥n", dbval.oid, dbval.datname, size);  
}  
  
EXEC SQL CLOSE curl;
```

33.4.4.3.3. Typedefs

新しい型と既存の型を対応付けるためには `typedef` キーワードを使ってください。

```
EXEC SQL BEGIN DECLARE SECTION;
    typedef char mychartype[40];
    typedef long serial_t;
EXEC SQL END DECLARE SECTION;
```

また、同様に以下を使うこともできます：

```
EXEC SQL TYPE serial_t IS long;
```

この宣言は、宣言セクションの一部である必要はありません。

33.4.4.3.4. ポインタ

ほとんどの一般的な型のポインタを宣言することができます。但し、自動メモリ確保を使わずにクエリのターゲット変数として使うことはできません。自動メモリ確保については [項33.7](#) を参照してください。

```
EXEC SQL BEGIN DECLARE SECTION;
    int *intp;
    char **charp;
EXEC SQL END DECLARE SECTION;
```

33.4.5. 非プリミティブSQLデータ型の扱い方

本節では、非スカラー型およびユーザ定義のSQLデータ型をECPGアプリケーションで扱う方法を示します。この内容は、前の説で説明した非プリミティブ型のホスト変数の扱い方とは別のものです。

33.4.5.1. 配列

SQLの配列は、ECPGにおいては直接的にはサポートされていません。SQL配列をC言語の配列のホスト変数に簡単に対応させることはできません。定められていない振る舞いを引き起こすでしょう。但し、いくつかの回避策があります。

もし、クエリが配列の要素に対して個別にアクセスした場合、ECPGにおける配列の利用を避けることができます。その際、要素に対応させることができる型のホスト変数を利用しなければなりません。例えば、カラムの型が `integer` の配列の場合、`int` 型のホスト変数を使用することができます。同様に、要素の型が `varchar` または `text` の場合、`char[]` ないし `VARCHAR[]` 型のホスト変数を使用することができます。

以下に例を示します。次のようなテーブルを仮定します：

```
CREATE TABLE t3 (
    ii integer[]
);

testdb=> SELECT * FROM t3;
      ii
-----
{1,2,3,4,5}
(1 row)
```

以下のプログラム例は、配列の4番目の要素を取得し、それを `int` 型のホスト変数に保存します：

```
EXEC SQL BEGIN DECLARE SECTION;
int ii;
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE curl CURSOR FOR SELECT ii[4] FROM t3;
EXEC SQL OPEN curl;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    EXEC SQL FETCH FROM curl INTO :ii ;
    printf("ii=%d\n", ii);
}

EXEC SQL CLOSE curl;
```

この例は以下のような結果を示します：

```
ii=4
```

複数の配列の要素を、配列型のホスト変数の複数の要素にマッピングするためには、配列型のカラムの各要素とホスト変数配列

の各要素は、以下の例のように別々に管理されなければなりません:

```
EXEC SQL BEGIN DECLARE SECTION;
int ii_a[8];
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE curl CURSOR FOR SELECT ii[1], ii[2], ii[3], ii[4] FROM t3;
EXEC SQL OPEN curl;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    EXEC SQL FETCH FROM curl INTO :ii_a[0], :ii_a[1], :ii_a[2], :ii_a[3];
    ...
}
```

繰り返しになりますが、以下の例は

```
EXEC SQL BEGIN DECLARE SECTION;
int ii_a[8];
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE curl CURSOR FOR SELECT ii FROM t3;
EXEC SQL OPEN curl;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    /* WRONG */
    EXEC SQL FETCH FROM curl INTO :ii_a;
    ...
}
```

この場合は正しく動作しません。なぜなら、配列型のカラムをホストの配列変数に直接対応させることはできないからです。

もうひとつの回避策は、配列をホスト変数の `char[]` または `VARCHAR[]` 型に文字列表現として保存することです。この表現方法についての詳細は [項8.15.2](#) を参照してください。このことは、配列にはホストプログラム内で自然な形ではアクセスできないことを意味しています（文字列表現を解析する追加処理が無ければ）。

33.4.5.2. 複合型

複合型はECPGでは直接はサポートされていませんが、簡単な回避方法が利用可能です。利用可能なワークアラウンドは、先に配列において説明されたものと似ています: 各属性に個別にアクセスするか、外部の文字列表現を使います。

以降の例のため、以下の型とテーブルを仮定します:

```
CREATE TYPE comp_t AS (intval integer, textval varchar(32));
CREATE TABLE t4 (compval comp_t);
INSERT INTO t4 VALUES ( (256, 'PostgreSQL') );
```

もっとも分かりやすい解決法は、各属性に個別にアクセスすることです。以下のプログラムは、`comp_t` 型の各要素を個別に選択することによってサンプルのテーブルからデータを受け取ります:

```
EXEC SQL BEGIN DECLARE SECTION;
int intval;
varchar textval[33];
EXEC SQL END DECLARE SECTION;

/* Put each element of the composite type column in the SELECT list. */
EXEC SQL DECLARE curl CURSOR FOR SELECT (compval).intval, (compval).textval FROM t4;
EXEC SQL OPEN curl;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    /* Fetch each element of the composite type column into host variables. */
    EXEC SQL FETCH FROM curl INTO :intval, :textval;

    printf("intval=%d, textval=%s\n", intval, textval.arr);
}

EXEC SQL CLOSE curl;
```

この例を拡張して、`FETCH` コマンドの値を格納するホスト変数を一つの構造体にまとめることができます。構造体の形のホスト変数の詳細については [項33.4.4.3.2](#) を参照してください。構造体に変更するために、この例は以下のように変更することができます。二つのホスト変数 `intval` と `textval` を `comp_t` 構造体のメンバー変数とし、構造体を `FETCH` コマンドで指定します。


```
EXEC SQL BEGIN DECLARE SECTION;
typedef struct
{
    int intval;
    varchar textval[33];
} comp_t;

comp_t compval;
EXEC SQL END DECLARE SECTION;

/* Put each element of the composite type column in the SELECT list. */
EXEC SQL DECLARE curl CURSOR FOR SELECT (compval).intval, (compval).textval FROM t4;
EXEC SQL OPEN curl;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    /* Put all values in the SELECT list into one structure. */
    EXEC SQL FETCH FROM curl INTO :compval;

    printf("intval=%d, textval=%s\n", compval.intval, compval.textval.arr);
}

EXEC SQL CLOSE curl;
```

構造体が `FETCH` コマンドで使われていますが、属性名は `SELECT` 句において各々が指定されています。これは、複合型の値のすべての属性を示す `*` を用いることで拡張することができます。

```
...
EXEC SQL DECLARE curl CURSOR FOR SELECT (compval).* FROM t4;
EXEC SQL OPEN curl;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    /* Put all values in the SELECT list into one structure. */
    EXEC SQL FETCH FROM curl INTO :compval;

    printf("intval=%d, textval=%s\n", compval.intval, compval.textval.arr);
}
...
```

この方法であれば、ECPGが複合型そのものを理解できないとしても、複合型はほぼシームレスに構造体に対応させることができます。

最後に、`char[]` または `VARCHAR[]` 型のホスト変数に外部の文字列表現として複合型の値を格納することもできます。しかし、この方法ではホストプログラムから値のフィールドにアクセスするのは簡単ではありません。

33.4.5.3. ユーザ定義の基本型

新しいユーザ定義の基本型は、ECPGでは直接的にはサポートされていません。外部の文字列表現、`char[]` または `VARCHAR[]` 型のホスト変数を使うことができ、この解決法は多くの型について確かに適切かつ十分です。

以下に [項35.11](#) に含まれる `complex` 型を使った例を示します。 `complex` from この型の外部文字列表現は `(%lf,%lf)` で、[項35.11](#) の `complex_in()` 関数および `complex_out()` 関数で定義されています。以下の例は、カラム `a` と `b` に、`complex` 型の値 `(1,1)` および `(3,3)` を挿入し、その後、それらをテーブルから `SELECT` します。

```
EXEC SQL BEGIN DECLARE SECTION;
varchar a[64];
varchar b[64];
EXEC SQL END DECLARE SECTION;

EXEC SQL INSERT INTO test_complex VALUES ('(1,1)', '(3,3)');

EXEC SQL DECLARE curl CURSOR FOR SELECT a, b FROM test_complex;
EXEC SQL OPEN curl;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    EXEC SQL FETCH FROM curl INTO :a, :b;
    printf("a=%s, b=%s\n", a.arr, b.arr);
}

EXEC SQL CLOSE curl;
```

この例は、以下の結果を示します。

```
a=(1,1), b=(3,3)
```

その他の回避方法は、ユーザ定義型をECPGにおいて直接的に使うことを避けることであり、ユーザ定義型とECPGが扱えるプリミティブ型を変換する関数またはキャストを作成することです。ただし、型のキャスト、特に暗黙のものは型システムにおいて慎重に導入されなければなりません。

例を示します。

```
CREATE FUNCTION create_complex(r double, i double) RETURNS complex
LANGUAGE SQL
IMMUTABLE
AS $$ SELECT $1 * complex '(1,0\')' + $2 * complex '(0,1\')' $$;
```

この定義の後、以下の例は

```
EXEC SQL BEGIN DECLARE SECTION;
double a, b, c, d;
EXEC SQL END DECLARE SECTION;

a = 1;
b = 2;
c = 3;
d = 4;

EXEC SQL INSERT INTO test_complex VALUES (create_complex(:a, :b), create_complex(:c, :d));
```

以下と同じ効果をもたらします。

```
EXEC SQL INSERT INTO test_complex VALUES ('(1,2\')', '(3,4\')');
```

33.4.6. 指示子

上の例ではNULL値を扱いません。実際、取り出し例では、もしデータベースからNULL値が取り出された場合にはエラーが発生します。データベースへNULL値を渡す、または、データベースからNULL値を取り出すためには、第二のホスト変数指定をデータを格納するホスト変数それぞれに追加しなければなりません。第二のホスト変数は指示子と呼ばれ、データがNULLかどうかを表すフラグが含まれます。NULLの場合、実際のホスト変数の値は無視されます。以下に、NULL値の取り出しを正しく扱う例を示します。

```
EXEC SQL BEGIN DECLARE SECTION;
VARCHAR val;
int val_ind;
EXEC SQL END DECLARE SECTION;

...

EXEC SQL SELECT b INTO :val :val_ind FROM test1;
```

値がNULLでなければ、指示子変数`val_ind`は0となります。値がNULLならば負の値となります。

指示子は他の機能を持ちます。指示子の値が正ならば、値がNULLではありませんが、ホスト変数に格納する際に一部切り詰められたことを示します。

プリプロセッサ `ecpg` に引数 `-r no_indicator` が渡された場合、"no-indicator" モードで動作します。no-indicator モードでは、指示子変数が指定されなかった場合、（入力および出力において）文字列型に対して空の文字列としてnull値が、整数型に対してはもっとも小さな値が割り当てられます（例えば、`int` の場合 `INT_MIN` です）。

[前のページ](#)
SQLコマンドの実行

[ホーム](#)
[上に戻る](#)

[次のページ](#)
動的SQL

33.5. 動的SQL

多くの場合、アプリケーションが実行しなければならないSQL文は、アプリケーションを作成する段階で決まります。しかし、中には、SQL文が実行時に構成されることや外部ソースで提供されることがあります。このような場合、SQL文を直接Cソースコードに埋め込むことはできません。しかし、文字列変数として提供される任意のSQL文を呼び出すことができる機能が存在します。

33.5.1. 結果セットを伴わないSQL文の実行

任意のSQL文を実行するもっとも簡単な方法は、EXECUTE IMMEDIATE コマンドを使用することです。例を示します：

```
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "CREATE TABLE test1 (...);";
EXEC SQL END DECLARE SECTION;

EXEC SQL EXECUTE IMMEDIATE :stmt;
```

EXECUTE IMMEDIATE は結果セットを返却しないSQL文に使用することができます (e.g., DDL, INSERT, UPDATE, DELETE)。結果を受け取るSQL文 (e.g., SELECT) をこの方法で実行することはできません。次の節で、その実行方法を説明します。

33.5.2. 入力パラメータを伴うSQL文の実行

任意のSQL文を実行するより強力な方法は、一度プリペアをし、その後でプリペアド・ステートメントを実行したいところで実行することです。また、SQL文を汎用化した形でプリペアし、パラメータを置き換えることで特定のSQL文を実行させることも可能です。SQL文をプリペアする時、後でパラメータとして置き換えたいところには疑問符を記述してください。以下に例を示します。

```
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "INSERT INTO test1 VALUES(?, ?);";
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE mystmt FROM :stmt;
...
EXEC SQL EXECUTE mystmt USING 42, 'foobar';
```

プリペアド・ステートメントが必要なくなった時、割当てを解除しなければなりません。

```
EXEC SQL DEALLOCATE PREPARE name;
```

33.5.3. 結果セットを返却するSQL文の実行

単一行を編訳するSQL文を実行するには、EXECUTE を使うことができます。結果を保存するには、INTO 句を追加します。

```
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "SELECT a, b, c FROM test1 WHERE a > ?";
int v1, v2;
VARCHAR v3[50];
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE mystmt FROM :stmt;
...
EXEC SQL EXECUTE mystmt INTO :v1, :v2, :v3 USING 37;
```

EXECUTE コマンドはINTO句、USING句、この両方を持つことも、どちらも持たないこともできます。

クエリが2行以上の結果を返すことが想定される場合、以下の例のようにカーソルを使う必要があります。（カーソルの詳細については [項33.3.2](#) を参照してください）

```
EXEC SQL BEGIN DECLARE SECTION;
char dbaname[128];
char datname[128];
char *stmt = "SELECT u.username as dbaname, d.datname "
            " FROM pg_database d, pg_user u "
            " WHERE d.datdba = u.usesysid";
EXEC SQL END DECLARE SECTION;

EXEC SQL CONNECT TO testdb AS con1 USER testuser;

EXEC SQL PREPARE stmt1 FROM :stmt;
```

```
EXEC SQL DECLARE cursor1 CURSOR FOR stmt1;
EXEC SQL OPEN cursor1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    EXEC SQL FETCH cursor1 INTO :dbaname,:datname;
    printf("dbaname=%s, datname=%s¥n", dbaname, datname);
}

EXEC SQL CLOSE cursor1;

EXEC SQL COMMIT;
EXEC SQL DISCONNECT ALL;
```

[前のページ](#)

ホスト変数の使用

[ホーム](#)
[上に戻る](#)

[次のページ](#)

pgtypes ライブラリ

33.6. pgtypes ライブラリ

pgtypesライブラリはPostgreSQLデータベースの型をCプログラムで使えるようにC言語に対応させます。また、これらの型を使用したCの基本的な計算を行う関数も提供します。この計算には、PostgreSQLサーバを使用しません。以下の例を参照してください。

```
EXEC SQL BEGIN DECLARE SECTION;
    date datel;
    timestamp tsl, tsout;
    interval ivl;
    char *out;
EXEC SQL END DECLARE SECTION;

PGTYPESdate_today(&datel);
EXEC SQL SELECT started, duration INTO :tsl, :ivl FROM datetbl WHERE d=:datel;
PGTYPEStimestamp_add_interval(&tsl, &ivl, &tsout);
out = PGTYPEStimestamp_to_asc(&tsout);
printf("Started + duration: %s\n", out);
free(out);
```

33.6.1. numeric 型

numeric 型では任意の精度での計算機能を提供します。PostgreSQLサーバにおける対応する型については[項8.1](#)を参照してください。任意の精度を持つために、この変数は動的に拡張、縮小できなければなりません。これが、PGTYPESnumeric_newやPGTYPESnumeric_free関数では、ヒープ領域上にのみしか numeric 変数を作成できない理由です。decimal 型も似ていますが精度が限定されていますので、ヒープ領域以外にスタック領域上でも作成可能です。

以下の関数は numeric 型で使うことができます：

PGTYPESnumeric_new

新規割当ての numeric 型へのポインタを要求します。

```
numeric *PGTYPESnumeric_new(void);
```

PGTYPESnumeric_free

numeric 型を解放し、そのメモリをすべて解放します。

```
void PGTYPESnumeric_free(numeric *var);
```

PGTYPESnumeric_from_asc

文字列表記から numeric 型に変換します。

```
numeric *PGTYPESnumeric_from_asc(char *str, char **endptr);
```

有効な書式の例を示します。-2、.794、+3.44、592.49E07、-32.84e-4。値への変換に成功した場合、有効なポインタが返されます。失敗した場合は NULL ポインタが返されます。現在ECPGは文字列全体を解析しますので、現時点では*endptr内に最初の無効な文字のアドレスを格納することをサポートしません。このためendptrを安全に NULL にすることができます。

PGTYPESnumeric_to_asc

numeric 型numの文字列表現を持つ、mallocで割り当てられた文字列へのポインタを返します。

```
char *PGTYPESnumeric_to_asc(numeric *num, int dscale);
```

numeric の値は、必要に応じて四捨五入され、dscale 桁の十進数で出力されます。

PGTYPESnumeric_add

2つの numeric 変数を加算し、3番目の numeric 変数に格納します。

```
int PGTYPESnumeric_add(numeric *var1, numeric *var2, numeric *result);
```

この関数は変数var1と変数var2を加算し、その結果をresultに格納します。成功時0を、エラー時-1を返します。

PGTYPESnumeric_sub

2つの numeric 型変数を減算し、3番目の numeric 型変数に結果を格納します。

```
int PGTYPEStnumeric_sub(numeric *var1, numeric *var2, numeric *result);
```

この関数は変数`var1`から変数`var2`を差し引きます。演算結果を変数`result`に格納します。成功時0を、エラー時-1を返します。

PGTYPEStnumeric_mul

2つの numeric 型変数を乗算し、3番目の numeric 型変数に結果を格納します。

```
int PGTYPEStnumeric_mul(numeric *var1, numeric *var2, numeric *result);
```

この関数は変数`var1`と変数`var2`を掛け合わせます。演算結果を変数`var3`に格納します。成功時0を、エラー時-1を返します。

PGTYPEStnumeric_div

2つの numeric 型変数で除算し、3番目の numeric 型変数に結果を格納します。

```
int PGTYPEStnumeric_div(numeric *var1, numeric *var2, numeric *result);
```

この関数は変数`var1`を変数`var2`で割ります。演算結果を変数`var3`に格納します。成功時0を、エラー時-1を返します。

PGTYPEStnumeric_cmp

2つの numeric 型変数を比較します。

```
int PGTYPEStnumeric_cmp(numeric *var1, numeric *var2)
```

この関数は2つの numeric 型変数を比較します。エラーの場合`INT_MAX`が返ります。成功時、この関数は以下のいずれかを返します:

- `var1`が`var2`より大きければ1。
- `var1`が`var2`より小さければ-1。
- `var1`が`var2`と等しければ0。

PGTYPEStnumeric_from_int

int型変数を numeric 型変数に変換します。

```
int PGTYPEStnumeric_from_int(signed int int_val, numeric *var);
```

この関数は符号付整数型の変数を受け、numeric 型変数`var`内に格納します。成功時0、失敗時-1が返ります。

PGTYPEStnumeric_from_long

long int型変数を numeric 型変数に変換します。

```
int PGTYPEStnumeric_from_long(signed long int long_val, numeric *var);
```

この関数は符号付long int型の変数を受け、numeric 型変数`var`内に格納します。成功時0、失敗時-1が返ります。

PGTYPEStnumeric_copy

numeric 型変数を他の numeric 型変数にコピーします。

```
int PGTYPEStnumeric_copy(numeric *src, numeric *dst);
```

この関数は、`src`が指し示す変数の値を`dst`が指し示す変数にコピーします。成功時0、失敗時-1が返ります。

PGTYPEStnumeric_from_double

double型の変数を numeric 型変数に変換します。

```
int PGTYPEStnumeric_from_double(double d, numeric *dst);
```

この関数はdouble型の変数を受け、変換結果を`dst`が指し示す変数内に格納します。成功時0、失敗時-1が返ります。

PGTYPEStnumeric_to_double

numeric 型変数をdouble型に変換します。

```
int PGTYPEStnumeric_to_double(numeric *nv, double *dp)
```

この関数は`nv`が指し示す numeric 型変数の値を`dp`が指し示すdouble型変数に変換します。成功時0、オーバーフローを含むエラーが発生した時-1が返ります。オーバーフローが発生した場合はさらに、グローバル変数`errno`は`PGTYPEStNUM_OVERFLOW`に設定されます。

PGTYPESnumeric_to_int

numeric 型変数をint型に変換します。

```
int PGTYPESnumeric_to_int(numeric *nv, int *ip);
```

この関数はnvが指し示す numeric 型変数の値をipが指し示す整数型変数に変換します。 成功時0、オーバーフローを含むエラーが発生した時-1が返ります。 オーバーフローが発生した場合はさらに、グローバル変数errnoはPGTYPES_NUM_OVERFLOWに設定されます。

PGTYPESnumeric_to_long

numeric 型変数をlong型に変換します。

```
int PGTYPESnumeric_to_long(numeric *nv, long *lp);
```

この関数はnvが指し示す numeric 型変数の値をlpが指し示すlong変数に変換します。 成功時0、オーバーフローを含むエラーが発生した時-1が返ります。 オーバーフローが発生した場合はさらに、グローバル変数errnoはPGTYPES_NUM_OVERFLOWに設定されます。

PGTYPESnumeric_to_decimal

numeric 型変数を decimal 型に変換します。

```
int PGTYPESnumeric_to_decimal(numeric *src, decimal *dst);
```

この関数はsrcが指し示す numeric 型変数の値をdstが指し示す decimal 型変数に変換します。 成功時0、オーバーフローを含むエラーが発生した時-1が返ります。 オーバーフローが発生した場合はさらに、グローバル変数errnoはPGTYPES_NUM_OVERFLOWに設定されます。

PGTYPESnumeric_from_decimal

decimal 型変数を numeric 型に変換します。

```
int PGTYPESnumeric_from_decimal(decimal *src, numeric *dst);
```

この関数はsrcが指し示す decimal 型変数の値をdstが指し示す numeric 型変数に変換します。 成功時0、エラーが発生した時-1が返ります。 decimal 型は制限付の numeric 型として実装されていますので、この変換ではオーバーフローは起きません。

33.6.2. 日付型

Cの日付型を使用して、プログラムからSQLの日付型を取り扱うことができます。 PostgreSQLサーバにおける対応する型については[項8.5](#)を参照してください。

日付型を操作するために以下の関数を使用することができます：

PGTYPESdate_from_timestamp

タイムスタンプから日付部分を取り出します。

```
date PGTYPESdate_from_timestamp(timestamp dt);
```

この関数は唯一の引数としてタイムスタンプを受け、そこから日付部分を取り出します。

PGTYPESdate_from_asc

テキスト表現から日付型に変換します。

```
date PGTYPESdate_from_asc(char *str, char **endptr);
```

この関数はCのchar*型文字列strとCのchar*型文字列endptrへのポインタを受け付けます。 現在ECPGは文字列全体を解析しますので、現時点では*endptrに最初の無効な文字のアドレスを格納することをサポートしません。 このためendptrを安全にNULLにすることができます。

この関数が常にMDY書式の日付を前提としている点に注意してください。 現在ECPGにはこれを変更するための変数がありません。

[表33-2](#)に許される入力書式を示します。

表 33-2. 有効なPGTYPESdate_from_ascの入力書式

入力	結果
January 8, 1999	January 8, 1999
1999-01-08	January 8, 1999
1/8/1999	January 8, 1999
1/18/1999	January 18, 1999
01/02/03	February 1, 2003
1999-Jan-08	January 8, 1999
Jan-08-1999	January 8, 1999
08-Jan-1999	January 8, 1999
99-Jan-08	January 8, 1999
08-Jan-99	January 8, 1999
08-Jan-06	January 8, 2006
Jan-08-99	January 8, 1999
19990108	ISO 8601; January 8, 1999
990108	ISO 8601; January 8, 1999
1999.008	年と年内日数
J2451187	Jユリウス日
January 8, 99 BC	紀元前99年

PGTYPESdate_to_asc

日付型変数のテキスト表現を返します。

```
char *PGTYPESdate_to_asc(date dDate);
```

この関数は唯一の引数として日付型dDateを受付けます。この関数は1999-01-18、つまりYYYY-MM-DD書式で日付を出力します。

PGTYPESdate_julmdy

日付型の変数から、日、月、年の値を取り出します。

```
void PGTYPESdate_julmdy(date d, int *mdy);
```

この関数は日付型のdと、3つの整数値を持つ配列mdyへのポインタを受付けます。この変数名はその並びを表し、mdy[0]には月数、mdy[1]には日数が、mdy[2]には年が入ります。

PGTYPESdate_mdyjul

日付の日、月、年を指定した3つの整数からなる配列から日付値を作成します。

```
void PGTYPESdate_mdyjul(int *mdy, date *jdate);
```

この関数は、1番目の引数として3つの整数からなる配列(mdy)、2番目の引数として処理結果を格納する日付型の変数へのポインタを受付けます。

PGTYPESdate_dayofweek

日付値から週内日数を表す数を返します。

```
int PGTYPESdate_dayofweek(date d);
```

この関数は唯一の引数としてdate型変数dを受け、その日付の週内日数を表す整数を返します。

- 0 - 日曜
- 1 - 月曜
- 2 - 火曜
- 3 - 水曜
- 4 - 木曜
- 5 - 金曜
- 6 - 土曜

PGTYPESdate_today

現在の日付を取得します。

```
void PGTYPESdate_today(date *d);
```

この関数は現在の日付に設定されるdate型変数(d)を指し示すポインタを受付けます。

```
PGTYPESdate_fmt_asc
```

date型変数を書式マスクを使用したテキスト表現に変換します。

```
int PGTYPESdate_fmt_asc(date dDate, char *fmtstring, char *outbuf);
```

この関数は変換対象のdate型(dDate)、書式マスク(fmtstring)、日付のテキスト表現を格納するための文字列(outbuf)を受付けます。

成功時に0、エラーが発生した場合は負の値が返ります。

以下のリテラルを使用して、フィールドを指定することができます。

- dd - 月内の日数。
- mm - 年内の月数。
- yy - 二桁表記の年数
- yyyy - 四桁表記の年数
- ddd - 日の名前（省略形）
- mmm - 月の名前（省略形）

他の文字はすべて出力文字列にそのままコピーされます。

表33-3にいくつかの可能性のある書式を示します。 この関数の使用法に関するアイデアを提供しています。 出力行はすべて同じ日付、1959年11月23日に基づいています。

表 33-3. 有効なPGTYPESdate_fmt_ascの入力書式

書式	結果
mmddyy	112359
ddmmyy	231159
yyymmdd	591123
yy/mm/dd	59/11/23
yy mm dd	59 11 23
yy.mm.dd	59.11.23
.mm.yyyy.dd.	.11.1959.23.
mmm. dd, yyyy	Nov. 23, 1959
mmm dd yyyy	Nov 23 1959
yyyy dd mm	1959 23 11
ddd, mmm. dd, yyyy	Mon, Nov. 23, 1959
(ddd) mmm. dd, yyyy	(Mon) Nov. 23, 1959

```
PGTYPESdate_defmt_asc
```

書式マスクを使用してCのchar*文字列からdate型の値に変換します。

```
int PGTYPESdate_defmt_asc(date *d, char *fmt, char *str);
```

この関数は、処理結果を格納するための日付型へのポインタ(d)、日付を解析するための書式マスク(fmt)、日付のテキスト表現を含むCのchar*文字列(str)を受付けます。 テキスト表現は書式マスクに合った表現であることが仮定されています。 しかし、文字列と書式マスクを1:1に対応付けする必要はありません。 この関数は並んだ順番に解析し、年の位置を表すyyまたはyyyyを、月の位置を表すmmを、日の位置を表すddを検索します。

表33-4はいくつかの可能性のある書式を示します。 これはこの関数の使用方法に関するアイデアを提供します。

表 33-4. 有効なrdefmtdateの入力書式

書式	文字列	結果
ddmmyy	21-2-54	1954-02-21
ddmmyy	2-12-54	1954-12-02
ddmmyy	20111954	1954-11-20
ddmmyy	130464	1964-04-13

mmm.dd.yyyy	MAR-12-1967	1967-03-12
yy/mm/dd	1954, February 3rd	1954-02-03
mmm.dd.yyyy	041269	1969-04-12
yy/mm/dd	In the year 2525, in the month of July, mankind will be alive on the 28th day	2525-07-28
dd-mm-yy	I said on the 28th of July in the year 2525	2525-07-28
mmm.dd.yyyy	9/14/58	1958-09-14
yy/mm/dd	47/03/29	1947-03-29
mmm.dd.yyyy	oct 28 1975	1975-10-28
mmddyy	Nov 14th, 1985	1985-11-14

33.6.3. timestamp型

Cのタイムスタンプ型を使用してプログラムからSQLのタイムスタンプ型データを扱うことができます。 PostgreSQLにおける対応する型については[項8.5](#)を参照してください。

以下の関数を使用してタイムスタンプ型を扱うことができます：

PGTYPEStimestamp_from_asc

テキスト表現のタイムスタンプをタイムスタンプ型変数に変換します。

```
timestamp PGTYPEStimestamp_from_asc(char *str, char **endptr);
```

この関数は変換対象の文字列(str)とC char*へのポインタ(endptr)を受付けます。 現在ECPGは文字列全体を解析しますので、現時点では*endptrに最初の無効な文字の場所を格納することサポートしません。 このためendptrを安全にNULL にすることができます。

この関数は成功時変換後のタイムスタンプを返します。 エラー時、PGTYPEStimestamp_invalidが返され、errnoがPGTYPES_TS_BAD_TIMESTAMPに設定されます。 この値についての重要な注意書きについてPGTYPEStimestamp_invalidを参照してください。

通常、入力文字列には許される日付指定の任意の組み合わせ、空白文字、許される時間指定を含むことができます。 時間帯はECPGでサポートされていない点に注意してください。 変換することはできますが、例えばPostgreSQLサーバが行うような計算を行うことはできません。 時間帯指定は警告無しに無視されます。

[表33-5](#)に入力文字列の例をいくつか示します。

表 33-5. 有効なPGTYPEStimestamp_from_ascの入力書式

入力	結果
1999-01-08 04:05:06	1999-01-08 04:05:06
January 8 04:05:06 1999 PST	1999-01-08 04:05:06
1999-Jan-08 04:05:06.789-8	1999-01-08 04:05:06.789 (時間帯指定は無視されます。)
J2451187 04:05-08:00	1999-01-08 04:05:00 (時間帯指定は無視されます。)

PGTYPEStimestamp_to_asc

date型をC char*文字列に変換します。

```
char *PGTYPEStimestamp_to_asc(timestamp tstamp);
```

この関数はtimestamp型のtstampを唯一の引数として受け、timestamp型のテキスト表現を含む割り当てられた文字列を返します。

PGTYPEStimestamp_current

現在のタイムスタンプを取り出します。

```
void PGTYPEStimestamp_current(timestamp *ts);
```

この関数は現在のタイムスタンプを取り出し、tsが指し示すtimestamp型変数に格納します。

PGTYPEStimestamp_fmt_asc

書式マスクを使用してtimestamp型変数をC char*に変換します。

```
int PGTYPEStimestamp_fmt_asc(timestamp *ts, char *output, int str_len, char *fmtstr);
```

この関数は、最初の引数として変換対象のtimestamp型 (ts)を、出力バッファのポインタ(output)、出力バッファで割当て可能な最大長 (str_len)、変換に使用する書式マスク(fmtstr)を受付けます。

成功するこの関数は0を返します。 エラーが発生した場合は負の値が返ります。

書式マスクには以下の書式指定を使用することができます。 書式指定はlibcの`strftime`関数で使用されるものと同じです。 書式指定以外は出力バッファにコピーされます。

- `%A` - 各言語の曜日名称に置換されます。
- `%a` - 各言語の曜日略称に置換されます。
- `%B` - 各言語の月名称に置換されます。
- `%b` - 各言語の月略称に置換されます。
- `%C` - 年を100で割った10進数に置換されます。1桁の場合は先頭に0が付与されます。
- `%c` - 各言語の日付時刻表現に置換されます。
- `%D` - `%m/%d/%y`と同じです。
- `%d` - 月内の10進日数(01-31)に置換されます。
- `%E*` `%O*` - POSIXロケール拡張です。 `%Ec` `%EC` `%Ex` `%EX` `%Ey` `%EY` `%Od` `%Oe` `%OH` `%OI` `%Om` `%OM` `%OS` `%Ou` `%OU` `%OV` `%Ow` `%OW` `%Oy` という並びは別の表現を提供するものと仮定されています。

さらに、`%OB`は、（日に関する仕様がないうで単体で使用される）別の月名を表すものとして実装されています。

- `%e` - 月内10進日数(1-31)に置換されます。1桁の場合は前に空白が付けられます。
- `%F` - `%Y-%m-%d`と同じです。
- `%G` - 世紀付の10進数として年に置換されます。この年は週の部分がより多く含まれます。（月曜が週の最初の日です。）
- `%g` - `%G`同様に年に置換されますが、世紀の部分を除く10進数(00-99)になります。
- `%H` - 10進の時間(24時間単位)に置換されます(00-23)。
- `%h` - `%b`と同じです。
- `%I` - 10進の時間（12時間単位）に置換されます(01-12)。
- `%j` - 10進の年内日数に置換されます(001-366)。
- `%k` - 10進の時間(24時間単位)に置換されます(0-23)。1桁の場合は先頭に空白が付けられます。
- `%l` - 10進の時間（12時間単位）に置換されます(1-12)。1桁の場合は先頭に空白が付けられます。
- `%M` - 10進の分数に置換されます(00-59)。
- `%m` - 10進の月数に置換されます(01-12)。
- `%n` - 改行に置換されます。
- `%O*` - `%E*`と同じです。
- `%p` - 各言語の"午前"または"午後"に適切に置換されます。
- `%R` - `%H:%M`と同じです。
- `%r` - `%I:%M:%S` `%p`と同じです。
- `%S` - 10進の秒数に置換されます(00-60)。
- `%s` - エポック、UTCからの秒数に置換されます。
- `%T` - `%H:%M:%S`と同じです。
- `%t` - タブに置換されます。
- `%U` - 10進の週番号（日曜が週の先頭です）に置換されます。

- %u - 10進の週番号（月曜が週の先頭です）に置換されます。
- %V - 10進の年内の週番号（月曜が週の先頭です）に置換されます(01-53)。新しい年で、1月1日を含む週が4日以上存在する場合、その週が1となります。さもなければ、この週は前年の週となり、次の週が1となります。
- %v - %e-%b-%Yと同じです。
- %W - 10進の年内の週番号（月曜が週の先頭です）に置換されます(00-53)。
- %w - 10進の週内日数（日曜が週の先頭です）に置換されます(0-6)。
- %X - 各言語の時間表現に置換されます。
- %x - 各言語の日付表現に置換されます。
- %Y - 10進の世紀付年に置換されます。
- %y - 10進の世紀なし年に置換されます(00-99)。
- %Z - 時間帯名称に置換されます。
- %z - UTCからの時間帯オフセットに置換されます。UTCより東では正符号が先頭に付き、西では負符号が付きます。それぞれ2桁の時間と分がその後が続きますが、その区切りはありません。（この形式はRFC 822の日付ヘッダでよく使用されます。）
- %+ - 各言語の日付時刻表現に置換されます。
- %-* - GNU libc拡張です。数値出力を行う際に何も文字を詰めません。
- \$_* - GNU libcの拡張です。明示的に空白文字を使用して文字を詰めます。
- %0* - GNU libcの拡張です。明示的に0を使用して文字を詰めます。
- %% - %に置換されます。

PGTYPETimestamp_sub

タイムスタンプの減算を行い、その結果をinterval型の変数に格納します。

```
int PGTYPETimestamp_sub(timestamp *ts1, timestamp *ts2, interval *iv);
```

この関数は`ts1`が指し示すタイムスタンプ型変数から`ts2`が指し示すタイムスタンプ型変数を差し引き、`iv`が指し示すinterval型変数に結果を格納します。

成功すると、この関数は0を返し、エラーが発生した場合は負の値を返します。

PGTYPETimestamp_defmt_asc

書式マスクを使用して、テキスト表現からtimestamp値へ変換します。

```
int PGTYPETimestamp_defmt_asc(char *str, char *fmt, timestamp *d);
```

この関数は`str`変数内に格納されたタイムスタンプのテキスト表現、`fmt`変数内に格納された使用される書式マスクを受け付けます。結果は`d`が指し示す変数内に格納されます。

書式マスク`fmt`が NULL ならば、この関数はデフォルトの書式マスク`%Y-%m-%d %H:%M:%S`を使用するようになります。

これは[PGTYPETimestamp_fmt_asc](#)関数の逆です。使用できる書式マスク項目についてはその文書を参照してください。

PGTYPETimestamp_add_interval

timestamp型変数にinterval型変数を加算します。

```
int PGTYPETimestamp_add_interval(timestamp *tin, interval *span, timestamp *tout);
```

この関数はtimestamp型変数`tin`へのポインタとinterval型変数`span`へのポインタを受け付けます。これは、interval値をtimestamp値に加算し、その結果のtimestamp値を`tout`が指し示す変数に格納します。

成功するとこの関数は0を返します。エラーが発生した場合は負の値を返します。

PGTYPETimestamp_sub_interval

timestamp型変数からinterval型変数の値を引きます。


```
int PGTYPEStimestamp_sub_interval(timestamp *tin, interval *span, timestamp *tout);
```

この関数は`tin`が指し示す`timestamp`型変数から`span`が指し示す`interval`型変数を引きます。結果は`tout`が指し示す変数に保存されます。

成功するとこの関数は0を、エラーが発生した場合は負の値を返します。

33.6.4. interval型

Cにおける`interval`型を用いることにより、プログラムからSQLの`interval`型のデータを扱うことができます。PostgreSQLサーバにおける対応する型については[項8.5](#)を参照してください。

以下の関数を使用して、`interval`型を扱うことができます。

`PGTYPEStimestamp_sub_interval`

新しく割り当てた`interval`型変数へのポインタを返します。

```
interval *PGTYPEStimestamp_sub_interval(interval *span, timestamp *tout);
```

`PGTYPEStimestamp_sub_interval`

以前に割り当てられた`interval`型変数のメモリを解放します。

```
void PGTYPEStimestamp_sub_interval(interval *intvl);
```

`PGTYPEStimestamp_sub_interval`

テキスト表現から`interval`型に変換します。

```
interval *PGTYPEStimestamp_sub_interval(char *str, char **endptr);
```

この関数は入力文字列`str`を変換し、割り当てられた`interval`型へのポインタを返します。現在ECPGは文字列全体を解析しますので、現時点では`*endptr`に最初の無効な文字のアドレスを格納することをサポートしません。このため`endptr`を安全に `NULL` にすることができます。

`PGTYPEStimestamp_sub_interval`

`interval`型変数をテキスト表現に変換します。

```
char *PGTYPEStimestamp_sub_interval(interval *span);
```

この関数は`span`が指し示す`interval`型変数をC `char*`に変換します。出力は`@ 1 day 12 hours 59 mins 10 secs`のようになります。

`PGTYPEStimestamp_sub_interval`

`interval`型変数をコピーします。

```
int PGTYPEStimestamp_sub_interval(interval *intvlsrc, interval *intvldest);
```

この関数は、`intvlsrc`が指し示す`interval`型変数を `intvldest`が指し示す変数にコピーします。事前に格納先の変数用のメモリを割り当てる必要があることに注意してください。

33.6.5. decimal型

`decimal`型は`numeric`型に似ています。しかし、その最大精度は30有効桁に制限されています。ヒープ上にしか作成できない`numeric`型と比べ、`decimal`型はスタックまたはヒープ上に作成することができます。（このためには`PGTYPEStimestamp_sub_interval`および`PGTYPEStimestamp_sub_interval`関数を使用します。） [項33.15](#)で説明するInformix互換モードでは`decimal`型を扱う関数がより多く存在します。

以下の関数を使用して`decimal`型を扱うことができます。これらは`libcompat`ライブラリに含まれるものだけではありません。

`PGTYPEStimestamp_sub_interval`

新しく割り当てられた`decimal`型変数へのポインタを要求します。

```
decimal *PGTYPEStimestamp_sub_interval(void);
```

`PGTYPEStimestamp_sub_interval`

`decimal`型を解放し、そのメモリをすべて解放します。

```
void PGTYPEStimestamp_sub_interval(decimal *var);
```

33.6.6. pgtypeslibのerrno値

PGTYPES_NUM_BAD_NUMERIC

引数はnumeric型変数（またはnumeric型変数へのポインタ）を含んでいるはずですが、実際のメモリ上の表現は無効でした。

PGTYPES_NUM_OVERFLOW

オーバーフローが発生しました。 numeric型はほぼ任意の精度を扱うことができますので、numeric型変数から他の型への変換ではオーバーフローが発生する可能性があります。

PGTYPES_NUM_UNDERFLOW

アンダーフローが発生しました。 numeric型はほぼ任意の精度を扱うことができますので、numeric型変数から他の型への変換ではアンダーフローが発生する可能性があります。

PGTYPES_NUM_DIVIDE_ZERO

ゼロ除算をしようとしてしました。

PGTYPES_DATE_BAD_DATE

PGTYPESdate_from_asc関数に無効な日付文字列が渡されました。

PGTYPES_DATE_ERR_EARGS

PGTYPESdate_defmt_asc関数に無効な引数が渡されました。

PGTYPES_DATE_ERR_ENOSHORTDATE

PGTYPESdate_defmt_asc関数により入力文字列内に無効なトークンが見つかりました。

PGTYPES_INTVL_BAD_INTERVAL

PGTYPESinterval_from_asc関数に無効な内部文字列が渡されました。 もしくはPGTYPESinterval_to_asc関数に無効な内部値が渡されました。

PGTYPES_DATE_ERR_ENOTDMY

PGTYPESdate_defmt_asc関数内の日/月/年の代入において不整合がありました。

PGTYPES_DATE_BAD_DAY

PGTYPESdate_defmt_asc関数により無効な月内日数が見つかりました。

PGTYPES_DATE_BAD_MONTH

PGTYPESdate_defmt_asc関数によって無効な月値が見つかりました。

PGTYPES_TS_BAD_TIMESTAMP

PGTYPEStimestamp_from_asc関数に無効なタイムスタンプ文字列が渡されました。 もしくはPGTYPEStimestamp_to_asc関数に無効なtimestamp値が渡されました。

PGTYPES_TS_ERR_EINFTIME

コンテキスト内で扱うことができない、無限なタイムスタンプ値がありました。

33.6.7. pgtypeslibの特殊な定数

PGTYPESInvalidTimestamp

無効なタイムスタンプを表すtimestamp型の値です。 これは解析エラーの場合にPGTYPEStimestamp_from_asc関数によって返されます。 timestampデータ型の内部表現のため、PGTYPESInvalidTimestampはまた同時に有効なタイムスタンプでもあります。 これは1899-12-31 23:59:59に設定されます。 エラーを検知するためには、PGTYPEStimestamp_from_ascを呼び出す度にその後、PGTYPESInvalidTimestampを試験するだけでなく、errno != 0も試験してください。

33.7. 記述子領域の使用

SQL記述子領域はSELECT、FETCH、DESCRIBE文の結果を処理する、より洗練された手法です。SQL記述子領域は1行のデータをメタデータ項目と一緒に1つのデータ構造体としてグループ化します。特に動的SQL文を実行する場合は結果列の性質が前もってわかりませんので、メタデータが有用です。PostgreSQLは記述子領域を使用するための2つの方法、名前付きSQL記述子領域とC構造化SQLDA、を提供します。

33.7.1. 名前付きSQL記述子領域

名前付きSQL記述子領域は、記述子全体に関する情報を持つヘッダと、基本的に結果行内の1つの列を記述する、1つ以上の項目記述子領域から構成されます。

SQL記述子領域を使用可能にするためには、それを以下のように割り当てなければなりません。

```
EXEC SQL ALLOCATE DESCRIPTOR identifier;
```

この識別子は記述子領域の"変数名"として使用されます。記述子が不要になったら、以下のように解放してください。

```
EXEC SQL DEALLOCATE DESCRIPTOR identifier;
```

記述子領域を使用するには、INTO句内の格納対象として、ホスト変数を列挙するのではなく、記述子領域を指定してください。

```
EXEC SQL FETCH NEXT FROM mycursor INTO SQL DESCRIPTOR mydesc;
```

結果セットが空の場合であっても、記述子領域には問い合わせのメタデータ、つまりフィールド名、が含まれます。

まだ実行されていないプリペARED・クエリでは、結果セットのメタデータを入手するためにDESCRIBEを使用することができます。

```
EXEC SQL BEGIN DECLARE SECTION;
char *sql_stmt = "SELECT * FROM table1";
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE stmt1 FROM :sql_stmt;
EXEC SQL DESCRIBE stmt1 INTO SQL DESCRIPTOR mydesc;
```

PostgreSQL 9.0より前では、SQLキーワードは省略可能でした。このためDESCRIPTORおよびSQL DESCRIPTORは名前付きSQL記述子領域を生成しました。これは強制事項になり、SQLキーワードを省略すると、SQLDA記述子領域を生成します。

[項33.7.2](#)を参照してください。

DESCRIBEおよびFETCH文では、INTOおよびUSINGキーワードを同じように使用することができます。これらは結果セットと記述子領域内のメタデータを生成します。

さて、どうやって記述子領域からデータを取り出すのでしょうか。この記述子領域を名前付きフィールドを持つ構造体とみなすことができます。ヘッダからフィールド値を取り出し、それをホスト変数に格納するには、以下のコマンドを使用します。

```
EXEC SQL GET DESCRIPTOR name :hostvar = field;
```

今のところ、COUNTというヘッダフィールドが1つだけ定義されています。これは、記述子領域に存在する項目数を表すものです（つまり、結果内に含まれる列数です）。このホスト変数は整数型でなければなりません。項目記述子領域からフィールドを取り出すには、以下のコマンドを使用します。

```
EXEC SQL GET DESCRIPTOR name VALUE num :hostvar = field;
```

numはリテラル整数、もしくは整数を持つホスト変数を取ることができます。取り得るフィールドは以下の通りです。

CARDINALITY (整数)

結果セット内の行数です。

DATA

実際のデータ項目です（したがってこのフィールドのデータ型は問い合わせに依存します）。

DATETIME_INTERVAL_CODE (整数)

TYPEが9の場合、DATETIME_INTERVAL_CODEは、DATEでは1、TIMEでは2、TIMESTAMPでは3、TIME WITH TIME ZONEでは4、TIMESTAMP WITH TIME ZONEでは5という値を取ります。

DATETIME_INTERVAL_PRECISION (整数)

未実装です。

INDICATOR (整数)

(NULL値や値の切り詰めを示す) 指示子です。

KEY_MEMBER (整数)

実装されていません。

LENGTH (整数)

データの文字列の長さです。

NAME (文字列)

列名です。

NULLABLE (整数)

実装されていません。

OCTET_LENGTH (整数)

データの文字表現のバイト長です。

PRECISION (整数)

(numeric型用の) 精度です。

RETURNED_LENGTH (整数)

データの文字数です。

RETURNED_OCTET_LENGTH (整数)

データの文字表現のバイト長です。

SCALE (整数)

(numeric型用の) 桁です。

TYPE (整数)

列のデータ型の数値コードです。

EXECUTE、DECLARE および OPEN 文では、INTO および USING の効果は異なります。 また、問い合わせやカーソル用の入力パラメータを提供するために記述子領域は手作業で構築することができます。 USING SQL DESCRIPTOR *name* は入力パラメータとパラメータ付きの問い合わせに渡す方法です。 名前付き SQL 記述子領域を構築する SQL 文は以下の通りです。

```
EXEC SQL SET DESCRIPTOR name VALUE num field = :hostvar;
```

PostgreSQLは、1つのFETCH文内の1レコードを複数取り出し、ホスト変数に格納することをサポートします。 この場合ホスト変数は配列であると仮定されます。

```
EXEC SQL BEGIN DECLARE SECTION;
int id[5];
EXEC SQL END DECLARE SECTION;

EXEC SQL FETCH 5 FROM mycursor INTO SQL DESCRIPTOR mydesc;

EXEC SQL GET DESCRIPTOR mydesc VALUE 1 :id = DATA;
```

33.7.2. SQLDA記述子領域

SQLDA記述子領域は、問い合わせの結果セットとメタデータを取り出すために使用可能なC言語の構造体です。 1つの構造体には結果セットの1レコードが格納されます。

```
EXEC SQL include sqlda.h;
sqlda_t          *mysqlda;
```

```
EXEC SQL FETCH 3 FROM mycursor INTO DESCRIPTOR mysqlda;
```

SQLキーワードが省略されていることに注意してください。 [項33.7.1](#)のINTOおよびUSINGの使用状況に関する段落はここで多少追加して適用します。 DESCRIBE文では、INTOが使用されている場合DESCRIPTORキーワードは完全に省略可能です。

```
EXEC SQL DESCRIBE prepared_statement INTO mysqlda;
```

SQLDAを使用するプログラムの一般的な流れは以下の通りです。

- 1 11 問い合わせをブリエアし、そのカーソルを宣言します。
- 1 11 結果セット用のSQLDAを宣言します。
- 1 11 入力パラメータ用のSQLDAを宣言し、初期化（メモリ割り当て、パラメータの設定）します。
- 1 11 入力用SQLDAでカーソルを開きます。
- 1 11 カーソルから行を取り出し、出力用SQLDAに格納します。
- 1 11 出力用SQLDAから値をホスト変数に（必要に応じて変換を行い）読み取ります。
- 1 11 カーソルを閉じます。
- 1 11 入力用SQLDAに割り当てられたメモリ領域を解放します。

33.7.2.1. SQLDAのデータ構造

SQLDAはsqllda_t、sqlvar_t、struct sqlnameという3つの種類のデータ構造を使用します。

ティップ: PostgreSQLのSQLDAはIBM DB2ユニバーサルデータベースのものと似たデータ構造を持ちます。 このため、DB2のSQLDAに関する技術情報の一部はPostgreSQLのSQLDAの理解のより良い助けになるでしょう。

33.7.2.1.1. sqllda_t構造

sqllda_t構造体は実際のSQLDAの型です。 これは1つのレコードを保持します。 そして2つ以上のsqllda_t構造体をdesc_nextフィールド内においてポインタを使ってリンクリスト内でつなげることができます。 こうして行の順序付き集合を表現します。 このため、2つ以上の行を取り出す時、アプリケーションは各sqllda_tノードのdesc_nextポインタを追うことでそれらを読み取ることができます。

sqllda_tの定義は以下の通りです。

```
struct sqllda_struct
{
    char            sqldaid[8];
    long            sqldabc;
    short           sqln;
    short           sqld;
    struct sqllda_struct *desc_next;
    struct sqlvar_struct sqlvar[1];
};

typedef struct sqllda_struct sqllda_t;
```

フィールドの意味は以下の通りです。

sqldaid

ここには"SQLDA "文字列リテラルが含まれます。

sqldabc

ここにはバイト単位の割り当てられた領域のサイズが含まれます。

sqln

USINGキーワードを使用してOPEN、DECLARE、EXECUTE文に渡される場合、ここにはパラメータ付き問い合わせの入力パラメータ数が含まれます。 SELECT、EXECUTE、FETCH文の出力として使用される場合、この値はsqld文と同じです。

sqld

ここには結果セットのフィールド数が含まれます。

desc_next

問い合わせが複数のレコードを返す場合、複数結び付いたSQLDA構造体が返されます。 `desc_next` にリスト内の次の項目を指し示すポインタが保持されます。

`sqlvar`

これは結果セット内の列の配列です。

33.7.2.1.2. `sqlvar_t`構造体

`sqlvar_t`構造体は列の値と型や長さなどのメタデータを保持します。 この型の定義は以下の通りです。

```
struct sqlvar_struct
{
    short      sqltype;
    short      sqlllen;
    char       *sqldata;
    short      *sqlind;
    struct sqlname sqlname;
};

typedef struct sqlvar_struct sqlvar_t;
```

フィールドの意味は以下の通りです。

`sqltype`

ここにはフィールドの型識別子が含まれます。 値については`ecpgtype.h`の`enum ECPGttype`を参照してください。

`sqlllen`

フィールドのバイナリ長が含まれます。 例えば`ECPGt_int`では4バイトです。

`sqldata`

データそのものを指し示します。 データ書式は[項33.4.4](#)で説明します。

`sqlind`

データのNULL指示子を指し示します。 0は非NULLを、-1はNULLを意味します。

`sqlname`

フィールドの名前です。

33.7.2.1.3. `struct sqlname`構造体

`struct sqlname`構造体は列名を保持します。 `sqlvar_t`構造体のメンバとして使用されます。 構造体の定義は以下の通りです。

```
#define NAMEDATALEN 64

struct sqlname
{
    short      length;
    char       data[NAMEDATALEN];
};
```

フィールドの意味は以下の通りです。

`length`

フィールド名の長さが含まれます。

`data`

実際のフィールド名が含まれます。

33.7.2.2. SQLDAを使用した結果セットの取り出し

SQLDAを通して問い合わせの結果を取り出す一般的な手順は以下に示します。

- 1 11 結果セットを受けとるための`sqllda_t`構造体を宣言します。
- 1 11 宣言したSQLDAを指定した問い合わせを処理するために`FETCH/EXECUTE/DESCRIBE`を実行します。
- 1 11 `sqllda_t`構造体のメンバ`sqln`を検索することにより結果セット内のレコード数を検査します。
- 1 11 `sqllda_t`構造体のメンバ`sqlvar[0]`、`sqlvar[1]`などから各列の値を入手します。

1 11 `sqllda_t`構造体のメンバ`desc_next`ポインタを追い、次の行（`sqllda_t`構造体）に進みます。

1 11 必要なだけ上を繰り返します。

以下にSQLDAを通して結果セットを取り出す例を示します。

まず、結果セットを受け取る`sqllda_t`構造体を宣言します。

```
sqllda_t *sqllda1;
```

次にコマンド内にSQLDAを指定します。 以下は`FETCH`コマンドの例です。

```
EXEC SQL FETCH NEXT FROM curl INTO DESCRIPTOR sqllda1;
```

行を取り出すためにリンクリストを追うループを実行します。

```
sqllda_t *cur_sqllda;

for (cur_sqllda = sqllda1;
     cur_sqllda != NULL;
     cur_sqllda = cur_sqllda->desc_next)
{
    ...
}
```

ループの内側では、行の列データ（`sqlvar_t`構造体）それぞれを取り出す別のループを実行します。

```
for (i = 0; i < cur_sqllda->sqlld; i++)
{
    sqlvar_t v = cur_sqllda->sqlvar[i];
    char *sqldata = v.sqldata;
    short sqllen = v.sqllen;
    ...
}
```

列の値を入手するために、`sqlvar_t`構造体のメンバ`sqltype`の値を検査します。 そして、列の型に応じて、`sqlvar`フィールドからホスト変数にデータをコピーするための適切な方法に切り替えます。

```
char var_buf[1024];

switch (v.sqltype)
{
    case ECPGt_char:
        memset(&var_buf, 0, sizeof(var_buf));
        memcpy(&var_buf, sqldata, (sizeof(var_buf) <= sqllen ? sizeof(var_buf) - 1 : sqllen));
        break;

    case ECPGt_int: /* integer */
        memcpy(&intval, sqldata, sqllen);
        snprintf(var_buf, sizeof(var_buf), "%d", intval);
        break;

    ...
}
```

33.7.2.3. SQLDAを使用した問い合わせパラメータ渡し

プリペアド・クエリに入力パラメータを渡すためにSQLDAを使用する、一般的な手順は以下の通りです。

1 11 プリペアド・クエリ（プリペアド・ステートメント）を作成します。

1 11 入力用SQLDAとして`sqllda_t`構造体を宣言します。

1 11 入力用SQLDA用にメモリ領域を（`sqllda_t`構造体として）割り当てます。

1 11 割り当てたメモリに入力値を設定（コピー）します。

1 11 入力用SQLDAを指定してカーソルを開きます。

以下に例を示します。

まずプリペアド・ステートメントを作成します。

```
EXEC SQL BEGIN DECLARE SECTION;
char query[1024] = "SELECT d.oid, * FROM pg_database d, pg_stat_database s WHERE d.oid = s.datid AND
(d.datname = ? OR d.oid = ?)";
EXEC SQL END DECLARE SECTION;
```

```
EXEC SQL PREPARE stmt1 FROM :query;
```

次にSQLDA用にメモリを割り当て、`sqlda_t`構造体のメンバ変数`sqln`に入力パラメータ数を設定します。プリペアド・クエリで2つ以上の入力パラメータが必要な場合、アプリケーションは(パラメータ数 - 1) * `sizeof(sqlvar_t)`で計算される追加のメモリ空間を割り当てなければなりません。ここで示す例では2つの入力パラメータ用にメモリ空間を割り当てます。

```
sqlda_t *sqlda2;

sqlda2 = (sqlda_t *) malloc(sizeof(sqlda_t) + sizeof(sqlvar_t));
memset(sqlda2, 0, sizeof(sqlda_t) + sizeof(sqlvar_t));

sqlda2->sqln = 2; /* 入力変数の個数 */
```

メモリを割り当てた後、`sqlvar[]`配列にパラメータ値を格納します。（これは、SQLDAが結果セットを受け取る時に列値を取り出すために使用した配列と同じです。）この例では、入力パラメータは文字列型を持つ"postgres"と整数型を持つ1です。

```
sqlda2->sqlvar[0].sqltype = ECPGt_char;
sqlda2->sqlvar[0].sqldata = "postgres";
sqlda2->sqlvar[0].sqlllen = 8;

int intval = 1;
sqlda2->sqlvar[1].sqltype = ECPGt_int;
sqlda2->sqlvar[1].sqldata = (char *) &intval;
sqlda2->sqlvar[1].sqlllen = sizeof(intval);
```

ここまでで設定したSQLDAを指定するカーソルを開くことで、入力パラメータはプリペアド・ステートメントに渡されます。

```
EXEC SQL OPEN cur1 USING DESCRIPTOR sqlda2;
```

最後に、問い合わせ結果を受け取るために使用するSQLDAとは異なり、入力用SQLDAの使用後、割り当てたメモリ空間を明示的に解放しなければなりません。

```
free(sqlda2);
```

33.7.2.4. SQLDAを使用するサンプルアプリケーション

以下に、システムカタログから入力パラメータにより指定されたデータベースの統計情報にアクセスし取り出す方法を示す、プログラム例を示します。

このアプリケーションは、`pg_database`と`pg_stat_database`システムテーブルをデータベースOIDで結合し、2つの入力パラメータ（データベース`postgres`とOID1）により取り出されるデータベース統計情報を読み取り、表示します。

まず、入力用のSQLDAと出力用のSQLDAを宣言します。

```
EXEC SQL include sqlda.h;

sqlda_t *sqldal; /* an output descriptor */
sqlda_t *sqlda2; /* an input descriptor */
```

次に、データベースに接続し、プリペアド・ステートメントを作成し、プリペアド・ステートメント用のカーソルを宣言します。

```
int
main(void)
{
    EXEC SQL BEGIN DECLARE SECTION;
    char query[1024] = "SELECT d.oid,* FROM pg_database d, pg_stat_database s WHERE d.oid=s.datid AND (
d.datname=? OR d.oid=? )";
    EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO testdb AS con1 USER testuser;

    EXEC SQL PREPARE stmt1 FROM :query;
    EXEC SQL DECLARE cur1 CURSOR FOR stmt1;
```

次に、入力パラメータのために入力用SQLDA内にいくつかの値を格納します。入力用SQLDAのためのメモリを割り当て、入力パラメータの個数を`sqln`に設定します。型、値、値の長さを`sqlvar`構造体内の`sqltype`、`sqldata`、`sqlllen`に格納します。

```
/* Create SQLDA structure for input parameters. */
sqlda2 = (sqlda_t *) malloc(sizeof(sqlda_t) + sizeof(sqlvar_t));
memset(sqlda2, 0, sizeof(sqlda_t) + sizeof(sqlvar_t));
sqlda2->sqln = 2; /* number of input variables */

sqlda2->sqlvar[0].sqltype = ECPGt_char;
sqlda2->sqlvar[0].sqldata = "postgres";
sqlda2->sqlvar[0].sqlllen = 8;

intval = 1;
```

```

sqllda2->sqlvar[1].sqltype = ECPGt_int;
sqllda2->sqlvar[1].sqldata = (char*)&intval;
sqllda2->sqlvar[1].sqlllen = sizeof(intval);

```

入力用SQLDAを設定し終えた後、入力用SQLDAを付けたカーソルを開きます。

```

/* Open a cursor with input parameters. */
EXEC SQL OPEN curl USING DESCRIPTOR sqllda2;

```

開いたカーソルから出力用SQLDA内に行を取り込みます。（一般的に結果セット内の行をすべて取り込むためには、ループ内でFETCHを繰り返し呼び出さなければなりません。）

```

while (1)
{
    sqllda_t *cur_sqllda;

    /* Assign descriptor to the cursor */
    EXEC SQL FETCH NEXT FROM curl INTO DESCRIPTOR sqllda1;

```

次に、sqllda_t構造体のリンクリストを追うことで、SQLDAから取り込んだレコードを取り出します。

```

for (cur_sqllda = sqllda1 ;
     cur_sqllda != NULL ;
     cur_sqllda = cur_sqllda->desc_next)
{
    ...

```

最初のレコードから各列を読み取ります。列数はsqlldに、最初の列の実データはsqlvar[0]に格納されています。どちらもsqllda_t構造体のメンバです。

```

/* Print every column in a row. */
for (i = 0; i < sqllda1->sqlld; i++)
{
    sqlvar_t v = sqllda1->sqlvar[i];
    char *sqldata = v.sqldata;
    short sqlllen = v.sqlllen;

    strncpy(name_buf, v.sqlname.data, v.sqlname.length);
    name_buf[v.sqlname.length] = '\0';

```

ここで、列データがv変数内に格納されました。列の型についてv.sqltypeを検索しながら、すべてのデータをホスト変数にコピーします。

```

switch (v.sqltype) {
    int intval;
    double doubleval;
    unsigned long long int longlongval;

    case ECPGt_char:
        memset(&var_buf, 0, sizeof(var_buf));
        memcpy(&var_buf, sqldata, (sizeof(var_buf) <= sqlllen ? sizeof(var_buf)-1 : sqlllen));
        break;

    case ECPGt_int: /* integer */
        memcpy(&intval, sqldata, sqlllen);
        snprintf(var_buf, sizeof(var_buf), "%d", intval);
        break;

    ...

    default:
        ...
}

printf("%s = %s (type: %d)\n", name_buf, var_buf, v.sqltype);
}

```

すべてのレコードを処理した後カーソルを閉じ、データベースとの接続を切断します。

```

EXEC SQL CLOSE curl;
EXEC SQL COMMIT;

EXEC SQL DISCONNECT ALL;

```

プログラム全体を[例33-1](#)に示します。

例 33-1. SQLDAプログラムの例

```

#include <stdlib.h>
#include <string.h>
#include <stdlib.h>

```

```

#include <stdio.h>
#include <unistd.h>

EXEC SQL include sqlda.h;

sqlda_t *sqlda1; /* descriptor for output */
sqlda_t *sqlda2; /* descriptor for input */

EXEC SQL WHENEVER NOT FOUND DO BREAK;
EXEC SQL WHENEVER SQLERROR STOP;

int
main(void)
{
    EXEC SQL BEGIN DECLARE SECTION;
    char query[1024] = "SELECT d.oid,* FROM pg_database d, pg_stat_database s WHERE d.oid=s.datid AND (
d.datname=? OR d.oid=? )";

    int intval;
    unsigned long long int longlongval;
    EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO uptimedb AS con1 USER uptime;

    EXEC SQL PREPARE stmt1 FROM :query;
    EXEC SQL DECLARE curl CURSOR FOR stmt1;

    /* Create a SQLDA structure for an input parameter */
    sqlda2 = (sqlda_t *)malloc(sizeof(sqlda_t) + sizeof(sqlvar_t));
    memset(sqlda2, 0, sizeof(sqlda_t) + sizeof(sqlvar_t));
    sqlda2->sqln = 2; /* a number of input variables */

    sqlda2->sqlvar[0].sqltype = ECPGt_char;
    sqlda2->sqlvar[0].sqldata = "postgres";
    sqlda2->sqlvar[0].sqlllen = 8;

    intval = 1;
    sqlda2->sqlvar[1].sqltype = ECPGt_int;
    sqlda2->sqlvar[1].sqldata = (char *) &intval;
    sqlda2->sqlvar[1].sqlllen = sizeof(intval);

    /* Open a cursor with input parameters. */
    EXEC SQL OPEN curl USING DESCRIPTOR sqlda2;

    while (1)
    {
        sqlda_t *cur_sqlda;

        /* Assign descriptor to the cursor */
        EXEC SQL FETCH NEXT FROM curl INTO DESCRIPTOR sqlda1;

        for (cur_sqlda = sqlda1 ;
            cur_sqlda != NULL ;
            cur_sqlda = cur_sqlda->desc_next)
        {
            int i;
            char name_buf[1024];
            char var_buf[1024];

            /* Print every column in a row. */
            for (i=0 ; i<cur_sqlda->sqld ; i++)
            {
                sqlvar_t v = cur_sqlda->sqlvar[i];
                char *sqldata = v.sqldata;
                short sqlllen = v.sqlllen;

                strncpy(name_buf, v.sqlname.data, v.sqlname.length);
                name_buf[v.sqlname.length] = '\0';

                switch (v.sqltype)
                {
                    case ECPGt_char:
                        memset(&var_buf, 0, sizeof(var_buf));
                        memcpy(&var_buf, sqldata, (sizeof(var_buf)<=sqlllen ? sizeof(var_buf)-1 : sqlllen));

                        break;

                    case ECPGt_int: /* integer */
                        memcpy(&intval, sqldata, sqlllen);
                        snprintf(var_buf, sizeof(var_buf), "%d", intval);
                        break;

                    case ECPGt_long_long: /* bigint */
                        memcpy(&longlongval, sqldata, sqlllen);
                        snprintf(var_buf, sizeof(var_buf), "%lld", longlongval);
                        break;
                }
            }
        }
    }
};

```

```

        default:
        {
            int i;
            memset(var_buf, 0, sizeof(var_buf));
            for (i = 0; i < sqllen; i++)
            {
                char tmpbuf[16];
                snprintf(tmpbuf, sizeof(tmpbuf), "%02x ", (unsigned char) sqldata[i]);
                strncat(var_buf, tmpbuf, sizeof(var_buf));
            }
            break;
        }

        printf("%s = %s (type: %d)\n", name_buf, var_buf, v.sqltype);
    }

    printf("\n");
}

EXEC SQL CLOSE curl;
EXEC SQL COMMIT;

EXEC SQL DISCONNECT ALL;

return 0;
}

```

この例の出力は以下のようなものになるはずです（一部の数値は変動します）。

```

oid = 1 (type: 1)
datname = template1 (type: 1)
datdba = 10 (type: 1)
encoding = 0 (type: 5)
datistemplate = t (type: 1)
dataallowconn = t (type: 1)
datconnlimit = -1 (type: 5)
datlastsysoid = 11510 (type: 1)
datfrozensid = 379 (type: 1)
dattablespace = 1663 (type: 1)
datconfig = (type: 1)
datacl = {=c/uptime,uptime=CTc/uptime} (type: 1)
datid = 1 (type: 1)
datname = template1 (type: 1)
numbackends = 0 (type: 5)
xact_commit = 113606 (type: 9)
xact_rollback = 0 (type: 9)
blks_read = 130 (type: 9)
blks_hit = 7341714 (type: 9)
tup_returned = 38262679 (type: 9)
tup_fetched = 1836281 (type: 9)
tup_inserted = 0 (type: 9)
tup_updated = 0 (type: 9)
tup_deleted = 0 (type: 9)

oid = 11511 (type: 1)
datname = postgres (type: 1)
datdba = 10 (type: 1)
encoding = 0 (type: 5)
datistemplate = f (type: 1)
dataallowconn = t (type: 1)
datconnlimit = -1 (type: 5)
datlastsysoid = 11510 (type: 1)
datfrozensid = 379 (type: 1)
dattablespace = 1663 (type: 1)
datconfig = (type: 1)
datacl = (type: 1)
datid = 11511 (type: 1)
datname = postgres (type: 1)
numbackends = 0 (type: 5)
xact_commit = 221069 (type: 9)
xact_rollback = 18 (type: 9)
blks_read = 1176 (type: 9)
blks_hit = 13943750 (type: 9)
tup_returned = 77410091 (type: 9)
tup_fetched = 3253694 (type: 9)
tup_inserted = 0 (type: 9)
tup_updated = 0 (type: 9)
tup_deleted = 0 (type: 9)

```

33.8. エラー処理

本節では、埋め込みSQLプログラムにおいて、例外条件や警告をどのように扱うことができるかについて説明します。 このために、共に使用できる2つの機能があります。

- WHENEVER コマンドを使用して、警告条件、エラー条件を扱うようにコールバックを設定することができます。
- エラーまたは警告に関する詳細情報は `sqlca` 変数から入手することができます。

33.8.1. コールバックの設定

エラーや警告を受け取る簡単な手法の1つは、特定の条件が発生する度に特定の動作を実行するように設定することです。 一般的には以下ようになります。

```
EXEC SQL WHENEVER condition action;
```

condition は以下のいずれかを取ることができます。

SQLERROR

SQL文の実行中にエラーが発生する度に、指定した動作が呼び出されます。

SQLWARNING

SQL文の実行中に警告が発生する度に、指定した動作が呼び出されます。

NOT FOUND

SQL文が0行を受け取る、もしくは0行に影響する時、指定した動作が呼び出されます。（この条件はエラーではありませんが、これを特別に扱いたい場合があります。）

action は以下のいずれかを取ることができます。

CONTINUE

これは、実際のところ、その条件が無視されることを意味します。 これがデフォルトです。

```
GOTO label
GO TO label
```

指定したラベルに移動します（Cの `goto` 文を使用します）。

SQLPRINT

標準エラーにメッセージを出力します。 これは、単純なプログラムやプロトタイプ作成時に役に立ちます。 メッセージの詳細は設定できません。

STOP

プログラムを終了させる `exit(1)` を呼び出します。

DO BREAK

Cの `break` 文を実行します。 これはループ内、もしくは `switch` 文内でのみ使用しなければなりません。

```
CALL name (args)
DO name (args)
```

指定した引数で、指定したC関数を呼び出します。

標準SQLでは `CONTINUE` と `GOTO`（と `GO TO`）のみを提供しています。

簡単なプログラムで使ってみたくなるような例を以下に示します。 警告が発生した場合に簡単なメッセージを表示し、エラーが発生した場合にプログラムを中断します。

```
EXEC SQL WHENEVER SQLWARNING SQLPRINT;
EXEC SQL WHENEVER SQLERROR STOP;
```

`EXEC SQL WHENEVER` 文はCの構文ではなく、SQLプリプロセッサのディレクティブです。 設定したエラーもしくは警告動作は、最初の `EXEC SQL WHENEVER` と条件が発生させたSQL文の間で、同一条件に異なる動作が設定されない限り、ハンドラを設定した

箇所より後にある、すべての埋め込みSQL文に適用されます。Cプログラムの制御フローは関係しません。ですので、以下の2つのCプログラムの抜粋はどちらも望み通りの動作を行います。

```
/*
 * WRONG
 */
int main(int argc, char *argv[])
{
    ...
    if (verbose) {
        EXEC SQL WHENEVER SQLWARNING SQLPRINT;
    }
    ...
    EXEC SQL SELECT ...;
    ...
}

/*
 * WRONG
 */
int main(int argc, char *argv[])
{
    ...
    set_error_handler();
    ...
    EXEC SQL SELECT ...;
    ...
}

static void set_error_handler(void)
{
    EXEC SQL WHENEVER SQLERROR STOP;
}
```

33.8.2. sqlca

より強力にエラーを扱うために、埋め込みSQLインタフェースは以下の構造体を持つ`sqlca`（SQL通信領域）という名前のグローバル変数を提供します。

```
struct
{
    char sqlcaid[8];
    long sqlabc;
    long sqlcode;
    struct
    {
        int sqlerrml;
        char sqlerrmc[SQLERRMC_LEN];
    } sqlerrm;
    char sqlerrp[8];
    long sqlerrd[6];
    char sqlwarn[8];
    char sqlstate[5];
} sqlca;
```

（マルチスレッド化されたプログラムでは、各スレッドは自動的に`sqlca`のコピーを独自に持ちます。これは標準Cの`errno`グローバル変数の扱いと同様に動作します。）

`sqlca`は警告とエラーの両方を対象としています。1つのSQL文の実行時に複数の警告やエラーが発生した場合、`sqlca`は最後のものに関する情報のみを含みます。

直前のSQL文でエラーがなければ、`sqlca.sqlcode`は0に、`sqlca.sqlstate`は"00000"になります。警告やエラーが発生した場合は、`sqlca.sqlcode`は負の値に、`sqlca.sqlstate`は"00000"以外になります。正の`sqlca.sqlcode`は、直前の問い合わせが0行を返したなどの無害な条件を示します。`sqlca.sqlcode`と`sqlca.sqlstate`は2つの異なるエラーコードスキーマです。後で詳細に説明します。

直前のSQL文が成功すると、`sqlca.sqlerrd[1]`は処理された行のOIDが、もしあれば、格納されます。また、もしそのコマンドで適切ならば、`sqlca.sqlerrd[2]`は処理された、もしくは返された行数が格納されます。

エラーもしくは警告の場合、`sqlca.sqlerrm.sqlerrmc`には、そのエラーを説明する文字列が格納されます。`sqlca.sqlerrm.sqlerrml`フィールドには`sqlca.sqlerrm.sqlerrmc`に格納されたエラーメッセージ長が格納されます（`strlen()`の結果です。おそらくCプログラマは必要としないでしょう）。一部のメッセージは固定長の`sqlerrmc`配列には長過ぎることに注意してください。この場合は切り詰められます。

警告の場合、`sqlca.sqlwarn[2]`は`w`に設定されます（他のすべての場合では、これは`w`以外の何かに設定されます）。`sqlca.sqlwarn[1]`が`w`に設定された場合、ホスト変数に代入する際に値が切り詰められています。他の要素が警告を示すように設定された場合、`sqlca.sqlwarn[0]`は`w`に設定されます。

今のところ、sqlcaid、sqlcabc、sqlerrpならびにsqlerrdとsqlwarnの上記以外の要素は有用な情報を持ちません。

sqlcaは標準SQLでは定義されていません。しかし、複数の他のSQLデータベースシステムで実装されています。その定義は基本部分は似ていますが、移植性を持つアプリケーションを作成する場合は実装の違いを注意して調査しなければなりません。

ここでWHENEVERとsqlcaを組み合わせて使用して、エラーが発生した時にsqlcaの内容を表示する、1つの例を示します。これはおそらく、より"ユーザ向け"のエラー処理を組み込む前の、アプリケーションのデバッグまたはプロトタイプで有用です。

```
EXEC SQL WHENEVER SQLERROR CALL print_sqlca();

void
print_sqlca()
{
    fprintf(stderr, "==== sqlca ====\n");
    fprintf(stderr, "sqlcode: %ld\n", sqlca.sqlcode);
    fprintf(stderr, "sqlerrm.sqlerrml: %d\n", sqlca.sqlerrm.sqlerrml);
    fprintf(stderr, "sqlerrm.sqlerrmc: %s\n", sqlca.sqlerrm.sqlerrmc);
    fprintf(stderr, "sqlerrd: %ld %ld %ld %ld %ld %ld\n", sqlca.sqlerrd[0], sqlca.sqlerrd[1], sqlca.sqlerrd[2],
sqlca.sqlerrd[3], sqlca.sqlerrd[4], sqlca.sqlerrd[5]);
    fprintf(stderr, "sqlwarn: %d %d %d %d %d %d %d %d\n", sqlca.sqlwarn[0], sqlca.sqlwarn[1],
sqlca.sqlwarn[2],
                                sqlca.sqlwarn[3], sqlca.sqlwarn[4],
sqlca.sqlwarn[5],
                                sqlca.sqlwarn[6], sqlca.sqlwarn[7]);
    fprintf(stderr, "sqlstate: %5s\n", sqlca.sqlstate);
    fprintf(stderr, "=====\n");
}
```

結果は以下ようになります（ここでのエラーはテーブル名の誤記述によるものです。）。

```
==== sqlca ====
sqlcode: -400
sqlerrm.sqlerrml: 49
sqlerrm.sqlerrmc: relation "pg_databasep" does not exist on line 38
sqlerrd: 0 0 0 0 0 0
sqlwarn: 0 0 0 0 0 0 0 0
sqlstate: 42P01
=====
```

33.8.3. SQLSTATE対SQLCODE

sqlca.sqlstateとsqlca.sqlcodeはエラーコードを提供する異なる2つの機構です。共に標準SQLから派生されたものですが、SQLCODEはSQL-92版では廃れたものとされ、以降の版から削除されました。したがって、新規アプリケーションではSQLSTATEを使用することを強く勧めます。

SQLSTATEは5要素の文字配列です。この5文字は、各種のエラー条件、警告条件のコードを表現する数字、大文字から構成されます。SQLSTATEは階層を持った機構です。最初の2文字は条件を汎化したクラスを示し、残り3文字は汎化クラスの副クラスを示します。成功状態は00000というコードで示されます。SQLSTATEコードのほとんどは標準SQLで定義されています。PostgreSQLサーバは本質的にSQLSTATEエラーコードをサポートしています。したがって、すべてのアプリケーションでこのエラーコードを使用することで、高度な一貫性を達成することができます。詳細については[付録A](#)を参照してください。

廃止されたエラーコードの機構であるSQLCODEは単なる整数です。0という値は成功を意味し、正の値は追加情報を持った成功を、負の値はエラーを示します。標準SQLでは、直前のコマンドが0行を返す、もしくは0行に影響したことを示す+100という正の値のみを定義しています。負の値は規定されていません。したがって、この機構では低い移植性しか達成できず、また、コード体系も階層を持っていません。歴史的に、PostgreSQLの埋め込みSQLプロセッサには、いくつかの特殊なSQLCODEの値が専用に割り当てられていました。以下に、その数値とそのシンボル名の一覧を示します。これらは他のSQL実装への移植性がないことを忘れないでください。アプリケーションのSQLSTATE機構への移行を簡易化するために、対応するSQLSTATEも示しています。しかし、2つのしくみの間の関係は1対1ではなく1対多です（実際は多対多です）。ですので、場合ごとに[付録A](#)に示したグローバルな各SQLSTATEを参照しなければなりません。

以下は割り当て済みのSQLCODEです。

0 (ECPG_NO_ERROR)

エラーがないことを示す。(SQLSTATE 00000)

100 (ECPG_NOT_FOUND)

これは、最後に実行したコマンドが取り出した、または、処理した行がゼロ行であったこと、あるいは、カーソルの最後であることを示す、害のない条件です。(SQLSTATE 02000)

以下のように、カーソルをループ内で処理する時、ループを中断する時を検知する方法として、このコードを使用することができます。

```

while (1)
{
    EXEC SQL FETCH ... ;
    if (sqlca.sqlcode == ECPG_NOT_FOUND)
        break;
}

```

しかし、WHENEVER NOT FOUND DO BREAKはこれを内部で効率的に行います。このため、通常、外部で明示的に記述する利点はありません。

-12 (ECPG_OUT_OF_MEMORY)

仮想メモリ不足を示します。この数値は-ENOMEMとして定義します。(SQLSTATE YE001)

-200 (ECPG_UNSUPPORTED)

ライブラリが把握していない何かをプリプロセッサが生成したことを示します。おそらく、互換性がないプリプロセッサとライブラリのバージョンを使用しています。(SQLSTATE YE002)

-201 (ECPG_TOO_MANY_ARGUMENTS)

コマンドの想定より多くのホスト変数が指定されたことを意味します。(SQLSTATE 07001もしくは07002)

-202 (ECPG_TOO_FEW_ARGUMENTS)

コマンドの想定よりも少ないホスト変数が指定されたことを意味します。(SQLSTATE 07001もしくは07002)

-203 (ECPG_TOO_MANY_MATCHES)

問い合わせが複数行を返したけれども、SQL文では1つの結果の格納の準備だけしかしていなかったことを意味します(例えば、指定された変数が配列ではなかった)。(SQLSTATE 21000)

-204 (ECPG_INT_FORMAT)

ホスト変数の型がintですが、データベース内のデータ型が異なり、その値をintとして解釈させることができませんでした。ライブラリはこの変換にstrtol()を使用します。(SQLSTATE 42804)

-205 (ECPG_UINT_FORMAT)

ホスト変数の型がunsigned intですが、データベース内のデータ型が異なり、その値をunsigned intとして解釈させることができませんでした。ライブラリはこの変換にstrtoul()を使用します。(SQLSTATE 42804)

-206 (ECPG_FLOAT_FORMAT)

ホスト変数の型がfloatですが、データベース内のデータ型が異なり、その値をfloatとして解釈させることができませんでした。ライブラリはこの変換にstrtod()を使用します。(SQLSTATE 42804)

-207 (ECPG_NUMERIC_FORMAT)

ホスト変数の型がnumericですが、データベース内のデータ型が異なり、その値をnumericとして解釈させることができませんでした。(SQLSTATE 42804)

-208 (ECPG_INTERVAL_FORMAT)

ホスト変数の型がintervalであり、データベース内のデータが他の型であり、interval値として解釈することができない値を含みます。(SQLSTATE 42804)

-209 (ECPG_DATE_FORMAT)

ホスト変数の型がdateであり、データベース内のデータが他の型であり、date値として解釈することができない値を含みます。(SQLSTATE 42804)

-210 (ECPG_TIMESTAMP_FORMAT)

ホスト変数の型がtimestampであり、データベース内のデータが他の型であり、timestamp値として解釈することができない値を含みます。

-211 (ECPG_CONVERT_BOOL)

これは、ホスト変数の型がboolですが、データベース内のデータが't'でも'f'でもなかったことを意味します。(SQLSTATE 42804)

-212 (ECPG_EMPTY)

PostgreSQLサーバに送信されたSQL文が空でした（通常埋め込みSQLプログラムでは発生しません。ですので、これは内部エラーを示しているかもしれません）。（SQLSTATE YE002）

-213 (ECPG_MISSING_INDICATOR)

NULL値が返されましたが、NULL用の指示子変数が与えられていませんでした。（SQLSTATE 22002）

-214 (ECPG_NO_ARRAY)

配列が必要な箇所に普通の変数が使用されていました。（SQLSTATE 42804）

-215 (ECPG_DATA_NOT_ARRAY)

配列値が必要な箇所にデータベースが普通の変数を返しました。（SQLSTATE 42804）

-220 (ECPG_NO_CONN)

存在しない接続にプログラムがアクセスしようとしてしました。（SQLSTATE 08003）

-221 (ECPG_NOT_CONN)

存在するが開いていない接続にプログラムがアクセスしようとしてしました（これは内部エラーです）。（SQLSTATE YE002）

-230 (ECPG_INVALID_STMT)

使用しようとしたSQL文がプリペアされていませんでした。（SQLSTATE 26000）

-239 (ECPG_INFORMIX_DUPLICATE_KEY)

重複キーエラー。一意性制約違反（Informix互換モード）。（SQLSTATE 23505）

-240 (ECPG_UNKNOWN_DESCRIPTOR)

指定した記述子が見つかりませんでした。使用しようとしたSQL文はプリペアされていませんでした。（SQLSTATE 33000）

-241 (ECPG_INVALID_DESCRIPTOR_INDEX)

記述子のインデックスが範囲外でした。（SQLSTATE 07009）

-242 (ECPG_UNKNOWN_DESCRIPTOR_ITEM)

無効な記述子項目が要求されました。（これは内部エラーです。）（SQLSTATE YE002）

-243 (ECPG_VAR_NOT_NUMERIC)

動的なSQL文の実行時にデータベースが数値を返しましたが、ホスト変数が数値ではありませんでした。（SQLSTATE 07006）

-244 (ECPG_VAR_NOT_CHAR)

動的なSQL文の実行時にデータベースが数値以外を返しましたが、ホスト変数が数値でした。（SQLSTATE 07006）

-284 (ECPG_INFORMIX_SUBSELECT_NOT_ONE)

副問い合わせの結果が単一行ではありません（Informix互換モード）。（SQLSTATE 21000）

-400 (ECPG_PGSQL)

PostgreSQLサーバで何らかのエラーが発生しました。このメッセージはPostgreSQLサーバからのエラーメッセージを含みます。

-401 (ECPG_TRANS)

PostgreSQLサーバがトランザクションのコミットやロールバックを始めることができないことを通知しました。（SQLSTATE 08007）

-402 (ECPG_CONNECT)

データベースへの接続試行に失敗しました。(SQLSTATE 08001)

-403 (ECPG_DUPLICATE_KEY)

重複キーエラー。一意性制約違反。(SQLSTATE 23505)

-404 (ECPG_SUBSELECT_NOT_ONE)

副問い合わせの結果が単一行ではありません。(SQLSTATE 21000)

-602 (ECPG_WARNING_UNKNOWN_PORTAL)

無効なカーソル名が指定されました。(SQLSTATE 34000)

-603 (ECPG_WARNING_IN_TRANSACTION)

トランザクションが進行中です。(SQLSTATE 25001)

-604 (ECPG_WARNING_NO_TRANSACTION)

活動中(進行中)のトランザクションがありません。(SQLSTATE 25P01)

-605 (ECPG_WARNING_PORTAL_EXISTS)

既存のカーソル名が指定されました。(SQLSTATE 42P03)

[前のページ](#)

記述子領域の使用

[ホーム](#)

[上に戻る](#)

[次のページ](#)

プリプロセッサ指示子

33.9. プリプロセッサ指示子

ecpg プリプロセッサがファイルを解析および処理する方法を変更することができる、プリプロセッサ指示子が複数あります。

33.9.1. ファイルのインクルード

埋め込み SQL プログラムに外部ファイルをインクルードするには、以下を使用します。

```
EXEC SQL INCLUDE filename;
EXEC SQL INCLUDE <filename>;
EXEC SQL INCLUDE "filename";
```

埋め込み SQL プリプロセッサは、`filename.h` という名前のファイルを探し、その前処理を行い、最終的に C 出力の中に含めます。このようにして、ヘッダファイル内の埋め込み SQL 文が正しく扱われます。

ecpg プリプロセッサは以下の順番で複数のディレクトリからファイルを検索します。

- カレントディレクトリ
- `/usr/local/include`
- ビルド時に設定された PostgreSQL のインクルードディレクトリ (e.g., `/usr/local/pgsql/include`)
- `/usr/include`

しかし `EXEC SQL INCLUDE "filename"` が使われる場合、現在のディレクトリのみが検索されます。

各ディレクトリの中で、プリプロセッサはまず指定されたファイル名を探します。見つからなければ（指定されたファイル名がこの接尾辞を持っていない限り）ファイル名に `.h` を付けて再検索します。

`EXEC SQL INCLUDE` は以下とは異なることに注意してください。

```
#include <filename.h>
```

このファイルには SQL コマンド用前処理が行われませんためです。当然ながら、他のヘッダファイルをインクルードする C の `#include` ディレクティブを使用することができます。

注意: 通常の SQL の大文字小文字の区別規則に従う `EXEC SQL INCLUDE` コマンドの一部であったとしても、インクルードファイルの名前は **大文字小文字が区別されます**。

33.9.2. define および undef 指示子

C で既知の `#define` 指示子と同様、埋め込み SQL でも似たような概念を持ちます。

```
EXEC SQL DEFINE name;
EXEC SQL DEFINE name value;
```

このため、以下のように名前を定義することができます。

```
EXEC SQL DEFINE HAVE_FEATURE;
```

また、定数を定義することもできます。

```
EXEC SQL DEFINE MYNUMBER 12;
EXEC SQL DEFINE MYSTRING 'abc';
```

事前の定義を削除するには `undef` を使用します。

```
EXEC SQL UNDEF MYNUMBER;
```

当然、C の `#define` や `#undef` を埋め込み SQL プログラムで使用することは可能です。違いは宣言した値をどこで評価されるかです。 `EXEC SQL DEFINE` を使用する場合は、ecpg プリプロセッサがその定義を評価し、その値を置換します。例えば、

```
EXEC SQL DEFINE MYNUMBER 12;
...
EXEC SQL UPDATE Tbl SET col = MYNUMBER;
```

と記載した場合、ecpg による置換がすでに行われていますので、C コンパイラでは `MYNUMBER` という名前や識別子を参照すること

はありません。 埋め込みSQL問い合わせで使用する予定の定数に`#define`を使用することはできませんので注意してください。
この場合、埋め込みSQLプリプロセッサがこの宣言を参照することができないためです。

33.9.3. `ifdef`、`ifndef`、`else`、`elif`、`endif`指示子

以下の指示子を使用して、コンパイルするコード部分を選択することができます。

```
EXEC SQL ifdef name;
```

name を検査し、その*name*が`EXEC SQL define name`で作成されていた場合に後続の行を処理します。

```
EXEC SQL ifndef name;
```

name を検査し、その*name*が`EXEC SQL define name`で作成されていない場合に後続の行を処理します。

```
EXEC SQL else;
```

`EXEC SQL ifdef name`または`EXEC SQL ifndef name`で導入されたセクションの代替セクションを開始します。

```
EXEC SQL elif name;
```

name を検査し、その*name*が`EXEC SQL define name`で作成されている場合に代替セクションを開始します。

```
EXEC SQL endif;
```

代替セクションを終了します。

以下に例を示します。

```
EXEC SQL ifndef TZVAR;
EXEC SQL SET TIMEZONE TO 'GMT';
EXEC SQL elif TZNAME;
EXEC SQL SET TIMEZONE TO TZNAME;
EXEC SQL else;
EXEC SQL SET TIMEZONE TO TZVAR;
EXEC SQL endif;
```

[前のページ](#)
エラー処理

[ホーム](#)
[上に戻る](#)

[次のページ](#)
埋め込みSQLプログラムの処理

33.10. 埋め込みSQLプログラムの処理

ここまでで、埋め込みSQL Cプログラムの作成方法は理解できたと思います。ここからはそのコンパイル方法についてお話しします。コンパイルの前に、そのファイルを埋め込みSQLCに通します。これは、使用するSQL文を特別な関数呼び出しに変換します。コンパイル後、必要な関数を持つ特別なライブラリとリンクしなければなりません。これらの関数は引数から情報を取り出し、libpqを使用してそのSQLを実行し、出力用に指定された引数にその結果を格納します。

プリプロセッサプログラムは`ecpg`という名前で、通常PostgreSQLのインストールに含まれています。通常、埋め込みSQLプログラムの拡張子は`.pgc`とします。`prog1.pgc`という名前のプログラムファイルがある場合、単純に以下を呼び出すことで前処理を行うことができます。

```
ecpg prog1.pgc
```

これは`prog1.c`という名前のファイルを作成します。入力ファイルがこの提案パターンに従った名前でない場合、`-o` オプションを使用して明示的に出力ファイルを指定することができます。

前処理後のファイルは普通にコンパイルできます。以下に例を示します。

```
cc -c prog1.c
```

生成されたCソースファイルはPostgreSQLインストールに付随するヘッダファイルをインクルードします。ですので、デフォルトで検索されない場所にPostgreSQLをインストールした場合は、コンパイル用のコマンドラインに`-I/usr/local/pgsql/include`のようなオプションを追加しなければなりません。

埋め込みSQLプログラムをリンクするためには、以下のように、`libecpg`ライブラリを含めなければなりません。

```
cc -o myprog prog1.o prog2.o ... -lecpg
```

繰り返しになりますが、コマンドラインに`-L/usr/local/pgsql/lib`といったオプションを追加する必要があるかもしれません。

大規模プロジェクトの構築処理をmakeを使用して管理している場合、以下の暗黙規則をMakefileに含めておくと便利です。

```
ECPG = ecpg

%.c: %.pgc
    $(ECPG) $<
```

`ecpg` コマンドの完全な構文は[ecpg](#)に説明があります。

デフォルトでは`ecpg`はスレッドセーフです。しかしクライアントコードのコンパイル時に他のスレッド関連のコマンドラインオプションを使用する必要があるかもしれません。

33.11. ライブラリ関数

`libecpg` ライブラリには基本的に、埋め込みSQLコマンドで表現される機能を実装するために使用する"隠された"関数が含まれています。しかし、直接呼び出すことができる便利な関数もあります。これによりコードが移植不可能になることに注意してください。

- `ECPGdebug(int on, FILE *stream)` は第1引数が0以外で渡された場合、デバッグログを有効にします。デバッグログは`stream`に出力されます。このログには、すべての入力変数が挿入されたすべてのSQL文と、PostgreSQLサーバが返した結果が含まれます。SQL文のエラーを見つける時に非常に役に立ちます。

注意: Windowsでは、`ecpg` ライブラリとアプリケーションが異なるフラグでコンパイルされると、この関数の呼び出しは、`FILE` ポインタの内部表現が異なるため、アプリケーションをクラッシュさせる可能性があります。特に、そのライブラリを使用するすべてのライブラリとすべてのアプリケーションに対して、`multithreaded/single-threaded`、`release/debug`、および`static/dynamic`フラグは同じでなければなりません。

- `ECPGget_PGconn(const char *connection_name)` は、指定された名前で識別されるライブラリデータベース接続ハンドルを返します。`connection_name`の設定が`NULL`の場合、現在の接続ハンドルが返されます。接続ハンドルを識別できない場合、関数は`NULL`を返します。必要ならば返される接続ハンドルを使用して、任意の`libpq`の他の関数を呼び出すことができます。

注意: `libpq`関数を直接使用して`ecpg`からデータベース接続ハンドルを操作することは推奨されません。

- `ECPGtransactionStatus(const char *connection_name)` は、`connection_name`で識別される指定接続の現在のトランザクション状態を返します。返される状態コードの詳細については[項31.2](#)と`libpq`の`PQtransactionStatus()`を参照してください。
- `ECPGstatus(int lineno, const char* connection_name)` はデータベースに接続している場合は真を、さもなければ偽を返します。単一の接続を使用している場合は`connection_name`を`NULL`とすることができます。

33.12. ラージオブジェクト

ラージオブジェクトはECPGで直接サポートされていません。しかしECPGアプリケーションは、`ECPGget_PGconn()` 関数を呼び出して必要な`PGconn`を入手して、`libpq`ラージオブジェクト関数を介してラージオブジェクトを操作することができます。（しかし`ECPGget_PGconn()` 関数の使用と`PGconn`を直接触ることは非常に注意して行わなければなりません。理想を言えば他のECPGデータベースアクセス呼び出しと混在させないようにしてください。）

`ECPGget_PGconn()` に関しては[項33.11](#)を参照してください。ラージオブジェクト関数インタフェースについては[第32章](#)を参照してください。

ラージオブジェクト関数をトランザクションブロック内で呼び出さなければなりません。このため自動コミットが無効な場合、`BEGIN` コマンドを明示的に発行しなければなりません。

[例33-2](#)では、ECPGアプリケーション内でラージオブジェクトの作成、書き出し、読み取りを行う方法を示すプログラム例を示します。

例 33-2. ラージオブジェクトにアクセスするECPGプログラム

```
#include <stdio.h>
#include <stdlib.h>
#include <libpq-fe.h>
#include <libpq/libpq-fs.h>

EXEC SQL WHENEVER SQLERROR STOP;

int
main(void)
{
    PGconn      *conn;
    Oid          loid;
    int          fd;
    char         buf[256];
    int          buflen = 256;
    char         buf2[256];
    int          rc;

    memset(buf, 1, buflen);

    EXEC SQL CONNECT TO testdb AS con1;

    conn = ECPGget_PGconn("con1");
    printf("conn = %p\n", conn);

    /* create */
    loid = lo_create(conn, 0);
    if (loid < 0)
        printf("lo_create() failed: %s", PQerrorMessage(conn));

    printf("loid = %d\n", loid);

    /* write test */
    fd = lo_open(conn, loid, INV_READ|INV_WRITE);
    if (fd < 0)
        printf("lo_open() failed: %s", PQerrorMessage(conn));

    printf("fd = %d\n", fd);

    rc = lo_write(conn, fd, buf, buflen);
    if (rc < 0)
        printf("lo_write() failed\n");

    rc = lo_close(conn, fd);
    if (rc < 0)
        printf("lo_close() failed: %s", PQerrorMessage(conn));

    /* read test */
    fd = lo_open(conn, loid, INV_READ);
    if (fd < 0)
        printf("lo_open() failed: %s", PQerrorMessage(conn));

    printf("fd = %d\n", fd);

    rc = lo_read(conn, fd, buf2, buflen);
    if (rc < 0)
        printf("lo_read() failed\n");
```

```
rc = lo_close(conn, fd);
if (rc < 0)
    printf("lo_close() failed: %s", PQerrorMessage(conn));

/* check */
rc = memcmp(buf, buf2, buflen);
printf("memcmp() = %d\n", rc);

/* cleanup */
rc = lo_unlink(conn, loid);
if (rc < 0)
    printf("lo_unlink() failed: %s", PQerrorMessage(conn));

EXEC SQL COMMIT;
EXEC SQL DISCONNECT ALL;
return 0;
}
```

[前のページ](#)

ライブラリ関数

[ホーム](#)
[上に戻る](#)

[次のページ](#)

C++アプリケーション

33.13. C++アプリケーション

ECPGはC++アプリケーションを多少の制限がありますがサポートします。本節ではいくつかの注意を説明します。

ecpg プリプロセッサはC（またはCのようなもの）と埋め込みSQLコマンドで記述された入力ファイルを取り、埋め込みSQLコマンドをC言語の小塊に変換し、最終的に.cファイルを作成します。ecpg が生成するC言語の小塊で使用するライブラリ関数のヘッダファイル定義は、C++で使用される場合extern "C" { ... }で囲まれます。このためC++でも継ぎ目なく動作するはずです。

しかし一般的には、ecpg プリプロセッサはCのみを理解しています。C++言語の特殊な構文や予約語を取り扱いません。このため、C++に特化した複雑な機能を使用するC++アプリケーションコードの中に記述された埋め込みSQLコードの一部は、正しく前処理することに失敗する、または想定通りに動作しないかもしれません。

C++アプリケーションで埋め込みSQLコードを使用する安全な方法は、ECPGの呼び出しをC++アプリケーションコードがデータベースにアクセスするために呼び出し、残りのC++コードとまとめてリンクされる、Cモジュール内に隠蔽することです。

[項33.13.2](#)を参照してください。

33.13.1. ホスト変数のスコープ

ecpg プリプロセッサはCにおける変数のスコープを理解しています。C言語では、変数のスコープはコードブロックに基づきますので、どちらかといえば単純です。しかしC++では クラスメンバ変数は宣言場所とは異なるコードブロック内で参照されます。このためecpg プリプロセッサはクラスメンバ変数のスコープを理解していません。

例えば、以下の場合、ecpg プリプロセッサはtestメソッド内のdbname変数の定義を見つけることができません。このためエラーになります。

```

class TestCpp
{
    EXEC SQL BEGIN DECLARE SECTION;
    char dbname[1024];
    EXEC SQL END DECLARE SECTION;

public:
    TestCpp();
    void test();
    ~TestCpp();
};

TestCpp::TestCpp()
{
    EXEC SQL CONNECT TO testdb1;
}

void Test::test()
{
    EXEC SQL SELECT current_database() INTO :dbname;
    printf("current_database = %s\n", dbname);
}

TestCpp::~~TestCpp()
{
    EXEC SQL DISCONNECT ALL;
}

```

このコードは以下のようなエラーになります。

```

ecpg test_cpp.pgc
test_cpp.pgc:28: ERROR: variable "dbname" is not declared

```

このスコープ問題を回避するためには、testメソッドを中間格納領域としてローカル変数を使用するように変更することができます。しかしこの手法は悪い回避策でしかありません。コードを醜くしますし性能も劣化させます。

```

void TestCpp::test()
{
    EXEC SQL BEGIN DECLARE SECTION;
    char tmp[1024];
    EXEC SQL END DECLARE SECTION;

    EXEC SQL SELECT current_database() INTO :tmp;
    strncpy(dbname, tmp, sizeof(tmp));

    printf("current_database = %s\n", dbname);
}

```



```
}
```

33.13.2. 外部のCモジュールを用いたC++アプリケーションの開発

C++における`ecpg`の技術的な制限を理解しているのであれば、ECPG機能を使用するC++アプリケーションを実現するためには、リンク段階でCオブジェクトとC++オブジェクトをリンクする方が、C++コード内で埋め込みSQLコマンドを直接記述することより優れているという結論に至るでしょう。本節では、簡単な例を用いて、C++アプリケーションコードから埋め込みSQLコマンドを分離する方法について説明します。この例では、アプリケーションはC++で実装し、PostgreSQLサーバに接続するためにCおよびECPGを使用します。

Cファイル（*.pgc）、ヘッダファイル、C++ファイルという3種類のファイルを作成しなければなりません。

test_mod.pgc

C内に埋め込まれたサブルーチンモジュールです。プリプロセッサにより`test_mod.c`に変換されます。

```
#include "test_mod.h"
#include <stdio.h>

void
db_connect()
{
    EXEC SQL CONNECT TO testdb1;
}

void
db_test()
{
    EXEC SQL BEGIN DECLARE SECTION;
    char dbname[1024];
    EXEC SQL END DECLARE SECTION;

    EXEC SQL SELECT current_database() INTO :dbname;
    printf("current_database = %s\n", dbname);
}

void
db_disconnect()
{
    EXEC SQL DISCONNECT ALL;
}
```

test_mod.h

Cモジュール（test_mod.pgc）内の関数宣言を持つヘッダファイルです。test_cpp.cppでインクルードされます。このファイルは、C++モジュールからリンクされますので、宣言を囲む`extern "C"`ブロックを持たなければなりません。

```
#ifdef __cplusplus
extern "C" {
#endif

void db_connect();
void db_test();
void db_disconnect();

#ifdef __cplusplus
}
#endif
```

test_cpp.cpp

mainルーチンとこの例でのC++クラスを含む、アプリケーションの主要コードです。

```
#include "test_mod.h"

class TestCpp
{
public:
    TestCpp();
    void test();
    ~TestCpp();
};

TestCpp::TestCpp()
{
    db_connect();
}

void
TestCpp::test()
{
}
```

```

        db_test();
    }

    TestCpp::~TestCpp()
    {
        db_disconnect();
    }

    int
    main(void)
    {
        TestCpp *t = new TestCpp();

        t->test();
        return 0;
    }

```

アプリケーションを構築するためには、以下の処理を行います。 `ecpg` を実行して `test_mod.pgc` を `test_mod.c` に変換します。そしてCコンパイラを用いて `test_mod.c` をコンパイルし `test_mod.o` を生成します。

```

ecpg -o test_mod.c test_mod.pgc
cc -c test_mod.c -o test_mod.o

```

次にC++コンパイラを用いて `test_cpp.cpp` をコンパイルし `test_cpp.o` を生成します。

```

c++ -c test_cpp.cpp -o test_cpp.o

```

最後に、C++コンパイラドライバを用いて `test_cpp.o` および `test_mod.o` というオブジェクトファイルを実行形式ファイルにリンクします。

```

c++ test_cpp.o test_mod.o -lecp -o test_cpp

```

[前のページ](#)
ラジオオブジェクト

[ホーム](#)
[上に戻る](#)

[次のページ](#)
埋め込みSQLコマンド

33.14. 埋め込みSQLコマンド

目次

[ALLOCATE DESCRIPTOR](#) -- SQL記述子領域を割り当てます。
[CONNECT](#) -- データベース接続を確立します。
[DEALLOCATE DESCRIPTOR](#) -- SQL記述子領域の割り当てを解除します。
[DECLARE](#) -- カーソルを定義します。
[DESCRIBE](#) -- プリペアド・ステートメントまたは結果セットに関する情報を入手します。
[DISCONNECT](#) -- データベース接続を終了します。
[EXECUTE IMMEDIATE](#) -- SQL文を動的にプリペアし、実行します。
[GET DESCRIPTOR](#) -- SQL記述子領域から情報を入手します。
[OPEN](#) -- 動的カーソルを開きます。
[PREPARE](#) -- 実行のためにSQL文をプリペアします。
[SET AUTOCOMMIT](#) -- 現在のセッションの自動コミット動作を設定します。
[SET CONNECTION](#) -- データベース接続を選択します。
[SET DESCRIPTOR](#) -- SQL記述子領域に情報を設定します。
[TYPE](#) -- 新しいデータ型を定義します。
[VAR](#) -- 変数を定義します。
[WHENEVER](#) -- SQL文により特定の分類の条件が発生する時に行う動作を指定します。

本節では、埋め込みSQL固有のSQLコマンドをすべて説明します。また、言及がない限り、埋め込みSQLでも使用することができる、[参照1, SQLコマンド](#)に列挙されたSQLコマンドを参照してください。

[前のページ](#)

C++アプリケーション

[ホーム](#)[上に戻る](#)[次のページ](#)

ALLOCATE DESCRIPTOR

ALLOCATE DESCRIPTOR

名前

ALLOCATE DESCRIPTOR -- SQL記述子領域を割り当てます。

概要

```
ALLOCATE DESCRIPTOR name
```

説明

ALLOCATE DESCRIPTOR は、PostgreSQL サーバとホストプログラムとの間のデータ交換のために使用することができる、新しい名前付き SQL 記述子領域を割り当てます。

記述子領域は、使用した後で DEALLOCATE DESCRIPTOR コマンドを使用して解放しなければなりません。

パラメータ

name

SQL 記述子の名前です。 大文字小文字を区別します。 これは SQL 識別子またはホスト変数になることができます。

例

```
EXEC SQL ALLOCATE DESCRIPTOR mydesc;
```

互換性

ALLOCATE DESCRIPTOR は標準 SQL で規定されています。

関連項目

[DEALLOCATE DESCRIPTOR](#), [GET DESCRIPTOR](#), [SET DESCRIPTOR](#)

CONNECT

名前

CONNECT -- データベース接続を確立します。

概要

```
CONNECT TO connection_target [ AS connection_name ] [ USER connection_user_name ]  
CONNECT TO DEFAULT  
CONNECT connection_user_name  
DATABASE connection_target
```

説明

CONNECT コマンドはクライアントとPostgreSQLサーバとの間の接続を確立します。

パラメータ

connection_target

connection_target 以下の複数の形式の1つを使用して、接続する対象サーバを指定します。

[*database_name*] [@*host*] [:*port*]

TCP/IPを介した接続。

unix:postgresql://*host* [:*port*] / [*database_name*] [?*connection_option*]

Unix ドメインソケットを介した接続。

tcp:postgresql://*host* [:*port*] / [*database_name*] [?*connection_option*]

TCP/IPを介した接続。

SQL文字列定数

上記形式のいずれかで記述された値を持ちます。

ホスト変数

上記形式のいずれかで記述された値を持つ`char[]`または`VARCHAR[]`型のホスト変数。

connection_object

他のコマンドで参照することができる、このデータベース接続の識別子です。省略可能です。これはSQL識別子またはホスト変数とすることができます。

connection_user

データベース接続用のユーザ名です。

このパラメータは、*user_name/password*、*user_name IDENTIFIED BY password*、*user_name USING password*のいずれかの形式を使用して、ユーザ名とパスワードを指定することができます。

ユーザ名とパスワードは、SQL識別子、文字列定数、ホスト変数とすることができます。

DEFAULT

libpqで定義された、デフォルトの接続パラメータすべてを使用します。

例

以下に接続パラメータを指定する複数の種類を示します。

```
EXEC SQL CONNECT TO "connectdb" AS main;
EXEC SQL CONNECT TO "connectdb" AS second;
EXEC SQL CONNECT TO "unix:postgresql://200.46.204.71/connectdb" AS main USER connectuser;
EXEC SQL CONNECT TO "unix:postgresql://localhost/connectdb" AS main USER connectuser;
EXEC SQL CONNECT TO 'connectdb' AS main;
EXEC SQL CONNECT TO 'unix:postgresql://localhost/connectdb' AS main USER :user;
EXEC SQL CONNECT TO :db AS :id;
EXEC SQL CONNECT TO :db USER connectuser USING :pw;
EXEC SQL CONNECT TO @localhost AS main USER connectdb;
EXEC SQL CONNECT TO REGRESSDB1 as main;
EXEC SQL CONNECT TO AS main USER connectdb;
EXEC SQL CONNECT TO connectdb AS :id;
EXEC SQL CONNECT TO connectdb AS main USER connectuser/connectdb;
EXEC SQL CONNECT TO connectdb AS main;
EXEC SQL CONNECT TO connectdb@localhost AS main;
EXEC SQL CONNECT TO tcp:postgresql://localhost/ USER connectdb;
EXEC SQL CONNECT TO tcp:postgresql://localhost/connectdb USER connectuser IDENTIFIED BY connectpw;
EXEC SQL CONNECT TO tcp:postgresql://localhost:20/connectdb USER connectuser IDENTIFIED BY connectpw;
EXEC SQL CONNECT TO unix:postgresql://localhost/ AS main USER connectdb;
EXEC SQL CONNECT TO unix:postgresql://localhost/connectdb AS main USER connectuser;
EXEC SQL CONNECT TO unix:postgresql://localhost/connectdb USER connectuser IDENTIFIED BY "connectpw";
EXEC SQL CONNECT TO unix:postgresql://localhost/connectdb USER connectuser USING "connectpw";
EXEC SQL CONNECT TO unix:postgresql://localhost/connectdb?connect_timeout=14 USER connectuser;
```

以下にホスト変数を使用して接続パラメータを指定する方法を示すプログラム例を示します。

```
int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    char *dbname      = "testdb";      /* database name */
    char *user        = "testuser";    /* connection user name */
    char *connection  = "tcp:postgresql://localhost:5432/testdb";
                                      /* connection string */
    char ver[256];      /* buffer to store the version string */
EXEC SQL END DECLARE SECTION;

    ECPGdebug(1, stderr);

    EXEC SQL CONNECT TO :dbname USER :user;
    EXEC SQL SELECT version() INTO :ver;
    EXEC SQL DISCONNECT;

    printf("version: %s\n", ver);

    EXEC SQL CONNECT TO :connection USER :user;
    EXEC SQL SELECT version() INTO :ver;
    EXEC SQL DISCONNECT;

    printf("version: %s\n", ver);

    return 0;
}
```

互換性

CONNECT は標準SQLで規定されていますが、接続パラメータの書式は実装に特化しています。

関連項目

[DISCONNECT](#), [SET CONNECTION](#)

[前のページ](#)

ALLOCATE DESCRIPTOR

[ホーム](#)
[上に戻る](#)

[次のページ](#)

DEALLOCATE DESCRIPTOR

DEALLOCATE DESCRIPTOR

名前

DEALLOCATE DESCRIPTOR -- SQL記述子領域の割り当てを解除します。

概要

```
DEALLOCATE DESCRIPTOR name
```

説明

DEALLOCATE DESCRIPTOR は名前付きSQL記述子領域の割り当てを解除します。

パラメータ

name

割り当てを解除する記述子の名前です。 大文字小文字を区別します。 これはSQL識別子またはホスト変数にすることができます。

例

```
EXEC SQL DEALLOCATE DESCRIPTOR mydesc;
```

互換性

DEALLOCATE DESCRIPTOR は標準SQLで規定されています。

関連項目

[ALLOCATE DESCRIPTOR](#), [GET DESCRIPTOR](#), [SET DESCRIPTOR](#)

DECLARE

名前

DECLARE -- カーソルを定義します。

概要

```
DECLARE cursor_name [ BINARY ] [ INSENSITIVE ] [ [ NO ] SCROLL ] CURSOR [ { WITH | WITHOUT } HOLD ] FOR
prepared_name
DECLARE cursor_name [ BINARY ] [ INSENSITIVE ] [ [ NO ] SCROLL ] CURSOR [ { WITH | WITHOUT } HOLD ] FOR
query
```

説明

DECLARE は、プリペARED・ステートメントの結果セット全体を繰り返し処理するカーソルを宣言します。このコマンドは直接的なDECLARE SQL コマンドとは多少異なる意味を持ちます。こちらは問い合わせを実行し、取り出し用の結果セットの準備を行います。埋め込みSQLコマンドでは、問い合わせの結果セット全体を繰り返す"ループ変数"の名前を宣言するだけです。実際の実行はOPEN コマンドでカーソルが開いた時に起こります。

パラメータ

cursor_name

カーソル名です。大文字小文字を区別します。これはSQL識別子またはホスト変数とすることができます。

prepared_name

プリペARED・クエリの名前です。SQL識別子またはホスト変数のいずれかです。

query

このカーソルで返される行を供給するSELECTまたはVALUES コマンドです。

カーソルオプションの意味については[DECLARE](#)を参照してください。

例

以下に問い合わせ用のカーソルを宣言する例を示します。

```
EXEC SQL DECLARE C CURSOR FOR SELECT * FROM My_Table;
EXEC SQL DECLARE C CURSOR FOR SELECT Item1 FROM T;
EXEC SQL DECLARE curl CURSOR FOR SELECT version();
```

プリペARED・ステートメント用のカーソルを宣言する例を示します。

```
EXEC SQL PREPARE stmt1 AS SELECT version();
EXEC SQL DECLARE curl CURSOR FOR stmt1;
```

互換性

DECLARE は標準SQLで規定されています。

関連項目

[OPEN](#), [CLOSE](#), [DECLARE](#)

DESCRIBE

名前

DESCRIBE -- プリペアド・ステートメントまたは結果セットに関する情報を入手します。

概要

```
DESCRIBE [ OUTPUT ] prepared_name USING [ SQL ] DESCRIPTOR descriptor_name
DESCRIBE [ OUTPUT ] prepared_name INTO [ SQL ] DESCRIPTOR descriptor_name
DESCRIBE [ OUTPUT ] prepared_name INTO sqllda_name
```

説明

DESCRIBE は、実際に行を取り込むことなく、プリペアド・ステートメントに含まれる結果列に関するメタデータ情報を取り出します。

パラメータ

prepared_name

プリペアド・ステートメントの名前です。これはSQL識別子またはホスト変数とすることができます。

descriptor_name

記述子の名前です。大文字小文字を区別します。これはSQL識別子またはホスト変数とすることができます。

sqllda_name

SQLDA変数の名前です。

例

```
EXEC SQL ALLOCATE DESCRIPTOR mydesc;
EXEC SQL PREPARE stmt1 FROM :sql_stmt;
EXEC SQL DESCRIBE stmt1 INTO SQL DESCRIPTOR mydesc;
EXEC SQL GET DESCRIPTOR mydesc VALUE 1 :charvar = NAME;
EXEC SQL DEALLOCATE DESCRIPTOR mydesc;
```

互換性

DESCRIBE は標準SQLで規定されています。

関連項目

[ALLOCATE DESCRIPTOR](#), [GET DESCRIPTOR](#)

DISCONNECT

名前

DISCONNECT -- データベース接続を終了します。

概要

```
DISCONNECT connection_name
DISCONNECT [ CURRENT ]
DISCONNECT DEFAULT
DISCONNECT ALL
```

説明

DISCONNECT はデータベースとの接続（またはすべての接続）を閉じます。

パラメータ

connection_name

CONNECT コマンドで確立したデータベース接続の名前です。

CURRENT

直前に開いた接続またはSET CONNECTION コマンドで設定された接続のいずれかである、"現在の"接続を閉じます。これはDISCONNECT に引数が与えられなかった場合のデフォルトです。

DEFAULT

デフォルトの接続を閉じます。

ALL

開いているすべての接続を閉じます。

例

```
int
main(void)
{
    EXEC SQL CONNECT TO testdb AS DEFAULT USER testuser;
    EXEC SQL CONNECT TO testdb AS con1 USER testuser;
    EXEC SQL CONNECT TO testdb AS con2 USER testuser;
    EXEC SQL CONNECT TO testdb AS con3 USER testuser;

    EXEC SQL DISCONNECT CURRENT; /* close con3 */
    EXEC SQL DISCONNECT DEFAULT; /* close DEFAULT */
    EXEC SQL DISCONNECT ALL;     /* close con2 and con1 */

    return 0;
}
```

互換性

DISCONNECT は標準SQLで規定されています。

関連項目

[CONNECT](#), [SET CONNECTION](#)

EXECUTE IMMEDIATE

名前

EXECUTE IMMEDIATE -- SQL文を動的にプリペアし、実行します。

概要

```
EXECUTE IMMEDIATE string
```

説明

EXECUTE IMMEDIATE は動的に指定されたSQL文を、結果行を受け取ることなく、即座にプリペアし実行します。

パラメータ

string

実行するSQL文を含むC文字列リテラルまたはホスト変数です。

例

以下に、EXECUTE IMMEDIATE と command ホスト変数を使用して INSERT を実行する例を示します。

```
sprintf(command, "INSERT INTO test (name, amount, letter) VALUES ('db: 'r1'', 1, 'f')");  
EXEC SQL EXECUTE IMMEDIATE :command;
```

互換性

EXECUTE IMMEDIATE は標準SQLで規定されています。

GET DESCRIPTOR

名前

GET DESCRIPTOR -- SQL記述子領域から情報を入手します。

概要

```
GET DESCRIPTOR descriptor_name :cvariable = descriptor_header_item [, ... ]
GET DESCRIPTOR descriptor_name VALUE column_number :cvariable = descriptor_item [, ... ]
```

説明

GET DESCRIPTORはSQL記述子領域から問い合わせ結果セットに関する情報を取り出し、それをホスト変数に格納します。記述子領域は通常、このコマンドを使用してホスト言語変数に情報を転送する前に、FETCHまたはSELECTを用いて値が投入されます。

このコマンドには2つの構文があります。1番目の構文では、そのまま結果セットに適用されている記述子の"ヘッダ"項目を取り出します。行数が1つの例です。列番号を追加のパラメータとして必要とする2番目の構文では特定の列に関する情報を取り出します。例えば、列名と列の実際の値です。

パラメータ

descriptor_name

記述子の名前です。

descriptor_header_item

どのヘッダ情報を取り出すかを識別するトークンです。結果セット内の列数を入手するCOUNTのみが現在サポートされています。

column_number

情報を取り出す列の番号です。1から数えます。

descriptor_item

どの列に関する情報を取り出すかを識別するトークンです。サポートされる項目のリストについては[項33.7.1](#)を参照してください。

cvariable

記述子領域から取り出したデータを受け取るホスト変数です。

例

この例は結果セット内の列数を取り出します。

```
EXEC SQL GET DESCRIPTOR d :d_count = COUNT;
```

この例は最初の列のデータ長を取り出します。

```
EXEC SQL GET DESCRIPTOR d VALUE 1 :d_returned_octet_length = RETURNED_OCTET_LENGTH;
```

この例は、2番目の列のデータ本体を文字列として取り出します。

```
EXEC SQL GET DESCRIPTOR d VALUE 2 :d_data = DATA;
```

以下は、SELECT current_database();を実行し、列数、列のデータ長、列のデータを表示する手続き全体を示す例です。

```
int
main(void)
{
```

```

EXEC SQL BEGIN DECLARE SECTION;
    int d_count;
    char d_data[1024];
    int d_returned_octet_length;
EXEC SQL END DECLARE SECTION;

EXEC SQL CONNECT TO testdb AS con1 USER testuser;
EXEC SQL ALLOCATE DESCRIPTOR d;

/* Declare, open a cursor, and assign a descriptor to the cursor */
EXEC SQL DECLARE cur CURSOR FOR SELECT current_database();
EXEC SQL OPEN cur;
EXEC SQL FETCH NEXT FROM cur INTO SQL DESCRIPTOR d;

/* Get a number of total columns */
EXEC SQL GET DESCRIPTOR d :d_count = COUNT;
printf("d_count          = %d\n", d_count);

/* Get length of a returned column */
EXEC SQL GET DESCRIPTOR d VALUE 1 :d_returned_octet_length = RETURNED_OCTET_LENGTH;
printf("d_returned_octet_length = %d\n", d_returned_octet_length);

/* Fetch the returned column as a string */
EXEC SQL GET DESCRIPTOR d VALUE 1 :d_data = DATA;
printf("d_data              = %s\n", d_data);

/* Closing */
EXEC SQL CLOSE cur;
EXEC SQL COMMIT;

EXEC SQL DEALLOCATE DESCRIPTOR d;
EXEC SQL DISCONNECT ALL;

return 0;
}

```

この例を実行すると、結果は以下のようになります。

```

d_count          = 1
d_returned_octet_length = 6
d_data           = testdb

```

互換性

GET DESCRIPTORは標準SQLで規定されています。

関連項目

[ALLOCATE DESCRIPTOR](#), [SET DESCRIPTOR](#)

[前のページ](#)

EXECUTE IMMEDIATE

[ホーム](#)
[上に戻る](#)

[次のページ](#)

OPEN

OPEN

名前

OPEN -- 動的カーソルを開きます。

概要

```
OPEN cursor_name
OPEN cursor_name USING value [, ... ]
OPEN cursor_name USING SQL DESCRIPTOR descriptor_name
```

説明

OPEN はカーソルを開き、省略することができますが、実際の値をカーソル定義内のプレースホルダにバインドします。カーソルは事前に DECLARE コマンドを用いて宣言されていなければなりません。OPEN の実行により問い合わせがサーバ上で実行を開始されます。

パラメータ

cursor_name

開くカーソルの名前です。これはSQL識別子またはホスト変数とすることができます。

value

カーソル内のプレースホルダにバインドされる値です。これは、SQL定数、ホスト変数、指示子を持つホスト変数とすることができます。

descriptor_name

カーソル内のプレースホルダにバインドされる値を含む記述子の名前です。これはSQL識別子またはホスト変数とすることができます。

例

```
EXEC SQL OPEN a;
EXEC SQL OPEN d USING 1, 'test';
EXEC SQL OPEN c1 USING SQL DESCRIPTOR mydesc;
EXEC SQL OPEN :curname1;
```

互換性

OPEN は標準SQLで規定されています。

関連項目

[DECLARE](#), [CLOSE](#)

[前のページ](#)

GET DESCRIPTOR

[ホーム](#)
[上に戻る](#)

[次のページ](#)
PREPARE

PREPARE

名前

PREPARE -- 実行のためにSQL文をプリペアします。

概要

```
PREPARE name FROM string
```

説明

PREPARE は実行用に文字列として動的に指定されたSQL文をプリペアします。これは、埋め込みプログラム内でも使用することができる、直接的なPREPARE SQL文とは異なります。EXECUTE コマンドを使用して、どちらの種類のプリペアド・ステートメントを実行することができます。

パラメータ

prepared_name

プリペアド・クエリ用の識別子です。

string

Cリテラル文字列、プリペアするSQL文を含むホスト変数、SELECT/INSERT/UPDATE/DELETEのいずれかです。

例

```
char *stmt = "SELECT * FROM test1 WHERE a = ? AND b = ?";

EXEC SQL ALLOCATE DESCRIPTOR outdesc;
EXEC SQL PREPARE foo FROM :stmt;

EXEC SQL EXECUTE foo USING SQL DESCRIPTOR indesc INTO SQL DESCRIPTOR outdesc;
```

互換性

PREPARE は標準SQLで規定されています。

関連項目

[EXECUTE](#)

[前のページ](#)
OPEN

[ホーム](#)
[上に戻る](#)

[次のページ](#)
SET AUTOCOMMIT

SET AUTOCOMMIT

名前

SET AUTOCOMMIT -- 現在のセッションの自動コミット動作を設定します。

概要

```
SET AUTOCOMMIT { = | TO } { ON | OFF }
```

説明

`SET AUTOCOMMIT` は現在のデータベースセッションの自動コミット動作を設定します。 デフォルトでは埋め込みSQLプログラムは自動コミットモードではありません。 このため`COMMIT` コマンドを必要ところで明示的に発行しなければなりません。 このコマンドはセッションを、個々のSQL文それぞれが暗黙的にコミットされる、自動コミットモードに変更することができます。

互換性

`SET AUTOCOMMIT` はPostgreSQL ECPGの拡張です。

[前のページ](#)

PREPARE

[ホーム](#)
[上に戻る](#)

[次のページ](#)

SET CONNECTION

SET CONNECTION

名前

SET CONNECTION -- データベース接続を選択します。

概要

```
SET CONNECTION [ TO | = ] connection_name
```

説明

SET CONNECTION は、上書きされない限りすべてのコマンドが使用する、"現在の"データベース接続を設定します。

パラメータ

connection_name

CONNECT コマンドで確立したデータベース接続の名前です。

DEFAULT

接続をデフォルトの接続に設定します。

例

```
EXEC SQL SET CONNECTION TO con2;  
EXEC SQL SET CONNECTION = con1;
```

互換性

SET CONNECTION は標準SQLで規定されています。

関連項目

[CONNECT](#), [DISCONNECT](#)

[前のページ](#)

SET AUTOCOMMIT

[ホーム](#)
[上に戻る](#)

[次のページ](#)

SET DESCRIPTOR

SET DESCRIPTOR

名前

SET DESCRIPTOR -- SQL 記述子領域に情報を設定します。

概要

```
SET DESCRIPTOR descriptor_name descriptor_header_item = value [, ... ]  
SET DESCRIPTOR descriptor_name VALUE number descriptor_item = value [, ...]
```

説明

SET DESCRIPTOR は SQL 記述子領域に値を投入します。その後、通常、記述子領域はプリペアド・クエリ実行においてパラメータをバインドするために使用されます。

このコマンドは 2 つの構文があります。最初の構文は、特定のデータと独立した、記述子の"ヘッダ"に適用します。2 番目の構文は、番号で識別される特定のデータに値を割り当てます。

パラメータ

descriptor_name

記述子の名前です。

descriptor_header_item

設定するヘッダ情報項目を識別するトークンです。記述子項目数を設定する COUNT のみが現在サポートされています。

number

設定する記述子項目の番号です。番号は 1 から数えます。

descriptor_item

記述子内のどの項目の情報を設定するかを識別するトークンです。サポートされる項目のリストについては[項33.7.1](#)を参照してください。

value

記述子項目に格納する値です。これは SQL 定数またはホスト変数とすることができます。

例

```
EXEC SQL SET DESCRIPTOR indesc COUNT = 1;  
EXEC SQL SET DESCRIPTOR indesc VALUE 1 DATA = 2;  
EXEC SQL SET DESCRIPTOR indesc VALUE 1 DATA = :val1;  
EXEC SQL SET DESCRIPTOR indesc VALUE 2 INDICATOR = :val1, DATA = 'some string';  
EXEC SQL SET DESCRIPTOR indesc VALUE 2 INDICATOR = :val2null, DATA = :val2;
```

互換性

SET DESCRIPTOR は標準 SQL で規定されています。

関連項目

[ALLOCATE DESCRIPTOR](#), [GET DESCRIPTOR](#)

TYPE

名前

TYPE -- 新しいデータ型を定義します。

概要

```
TYPE type_name IS ctype
```

説明

TYPE コマンドは新しいCの型を定義します。 これは宣言セクションにtypedefを記述することと同じです。

ecpgが-cオプション付きで実行された場合にのみこのコマンドは認識されます。

パラメータ

type_name

新しい型の名前です。 これは有効なCの型名でなければなりません。

ctype

Cの型指定です。

例

```
EXEC SQL TYPE customer IS
struct
{
    varchar name[50];
    int     phone;
};

EXEC SQL TYPE cust_ind IS
struct ind
{
    short    name_ind;
    short    phone_ind;
};

EXEC SQL TYPE c IS char reference;
EXEC SQL TYPE ind IS union { int integer; short smallint; };
EXEC SQL TYPE intarray IS int[AMOUNT];
EXEC SQL TYPE str IS varchar[BUFFERSIZ];
EXEC SQL TYPE string IS char[11];
```

以下にEXEC SQL TYPEを使用するプログラム例を示します。

```
EXEC SQL WHENEVER SQLERROR SQLPRINT;

EXEC SQL TYPE tt IS
struct
{
    varchar v[256];
    int     i;
};

EXEC SQL TYPE tt_ind IS
struct ind {
    short    v_ind;
    short    i_ind;
};

int
main(void)
```

```
{
EXEC SQL BEGIN DECLARE SECTION;
    tt t;
    tt_ind t_ind;
EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO testdb AS con1;

    EXEC SQL SELECT current_database(), 256 INTO :t:t_ind LIMIT 1;

    printf("t.v = %s¥n", t.v.arr);
    printf("t.i = %d¥n", t.i);

    printf("t_ind.v_ind = %d¥n", t_ind.v_ind);
    printf("t_ind.i_ind = %d¥n", t_ind.i_ind);

    EXEC SQL DISCONNECT con1;

    return 0;
}
```

このプログラムの出力は以下のようになります。

```
t.v = testdb
t.i = 256
t_ind.v_ind = 0
t_ind.i_ind = 0
```

互換性

TYPE コマンドはPostgreSQLの拡張です。

[前のページ](#)
SET DESCRIPTOR

[ホーム](#)
[上に戻る](#)

[次のページ](#)
VAR

VAR

名前

VAR -- 変数を定義します。

概要

```
VAR varname IS ctype
```

説明

VAR コマンドはホスト変数を定義します。 これは宣言セクション内の通常のC変数定義と同じです。

パラメータ

varname

Cの変数名です。

ctype

Cの型指定です。

例

```
EXEC SQL VAR vc IS VARCHAR[10];  
EXEC SQL VAR boolvar IS bool;
```

互換性

VAR コマンドはPostgreSQLの拡張です。

WHENEVER

名前

WHENEVER -- SQL文により特定の分類の条件が発生する時に行う動作を指定します。

概要

```
WHENEVER { NOT FOUND | SQLERROR | SQLWARNING } action
```

説明

SQL実行の結果において特殊な状態（行がない、SQL警告またはSQLエラー）で呼び出される動作を定義します。

パラメータ

パラメータの説明については[項33.8.1](#)を参照してください。

例

```
EXEC SQL WHENEVER NOT FOUND CONTINUE;
EXEC SQL WHENEVER NOT FOUND DO BREAK;
EXEC SQL WHENEVER SQLWARNING SQLPRINT;
EXEC SQL WHENEVER SQLWARNING DO warn();
EXEC SQL WHENEVER SQLERROR sqlprint;
EXEC SQL WHENEVER SQLERROR CALL print2();
EXEC SQL WHENEVER SQLERROR DO handle_error("select");
EXEC SQL WHENEVER SQLERROR DO sqlnotice(NULL, NONO);
EXEC SQL WHENEVER SQLERROR DO sqlprint();
EXEC SQL WHENEVER SQLERROR GOTO error_label;
EXEC SQL WHENEVER SQLERROR STOP;
```

以下は、結果セットを通したループ処理を扱うためにWHENEVER NOT FOUND BREAKを使用する典型的なアプリケーションです。

```
int
main(void)
{
    EXEC SQL CONNECT TO testdb AS con1;
    EXEC SQL ALLOCATE DESCRIPTOR d;
    EXEC SQL DECLARE cur CURSOR FOR SELECT current_database(), 'hoge', 256;
    EXEC SQL OPEN cur;

    /* when end of result set reached, break out of while loop */
    EXEC SQL WHENEVER NOT FOUND DO BREAK;

    while (1)
    {
        EXEC SQL FETCH NEXT FROM cur INTO SQL DESCRIPTOR d;
        ...
    }

    EXEC SQL CLOSE cur;
    EXEC SQL COMMIT;

    EXEC SQL DEALLOCATE DESCRIPTOR d;
    EXEC SQL DISCONNECT ALL;

    return 0;
}
```

互換性

WHENEVERは標準SQLで規定されていますが、ほとんどの動作はPostgreSQLの拡張です。

33.15. Informix互換モード

ecpgをInformix互換モードというモードで動作させることができます。このモードが有効ならば、Informix E/SQL用のInformixプリプロセッサであるかのように動作します。一般的にいうと、これにより埋め込みSQLコマンドを導入する際にEXEC SQLプリミティブの代わりにドル記号を使用することができます。

```
$int j = 3;
$CONNECT TO :dbname;
$CREATE TABLE test(i INT PRIMARY KEY, j INT);
$INSERT INTO test(i, j) VALUES (7, :j);
$COMMIT;
```

注意: \$とその後に続くinclude、define、ifdefなどのプリプロセッサ指示子の間に空白文字を含めてはなりません。こうしないと、プリプロセッサはトークンをホスト変数として解析します。

INFORMIX、INFORMIX_SEという2つの互換モードがあります。

互換モードを使用するプログラムをリンクする際、ECPGに同梱されるlibcompatとリンクすることを忘れないでください。

以前に説明した構文上の飾りの他に、Informix互換モードでは、入力、出力、データ変換関数、E/SQLからECPGで既知の埋め込みSQL文変換に関する関数もいくつか移植しています。

Informix互換モードはECPGのpgtypeslibライブラリと密接に関係しています。pgtypeslibはSQLデータ型とCホストプログラム内のデータ型を対応付けし、ほとんどのInformix互換モードで追加された関数を使用してこれらのCホストプログラム型を操作することができます。しかし、互換範囲は制限されています。これはInformixの動作を真似ることはしません。これを使用して、多少は同じ名前と同じ基本動作を行う関数进行操作、提供できますが、Informixを使用しているのであれば、完全な置き換えにはなりません。さらに一部のデータ型は異なります。例えば、PostgreSQLの日付時刻やinterval型ではYEAR TO MINUTEのような範囲を持ちませんので、これらはECPGではサポートできないことがわかります。

33.15.1. 追加の型

右側を切り詰めた文字列データを格納するInformixの特別な"string"仮定型はtypedefを使用せずともInformixモードでサポートされるようになりました。実際Informixモードでは、ECPGはtypedef sometype string;を含むソースファイルの処理を拒絶します。

```
EXEC SQL BEGIN DECLARE SECTION;
string userid; /* this variable will contain trimmed data */
EXEC SQL END DECLARE SECTION;

EXEC SQL FETCH MYCUR INTO :userid;
```

33.15.2. 追加または存在しない埋め込みSQL文

CLOSE DATABASE

このSQL文は現在の接続を閉じます。実際、これはECPGのDISCONNECT CURRENTと同義です。

```
$CLOSE DATABASE; /* close the current connection */
EXEC SQL CLOSE DATABASE;
```

FREE cursor_name

InformixのESQL/Cと比べECPGの動作方法に違いがあります(つまり純粋に文法の変換がどの段階で行われ、背後の実行時ライブラリにどの段階で依存するか)ので、ECPGにはFREE cursor_name文はありません。このためECPGにおいて、DECLARE CURSORがカーソル名を使用する実行時ライブラリ内の関数呼び出しに変換されません。これはECPG実行時ライブラリ内ではSQLカーソルの実行状況を保持しておらず、PostgreSQLサーバ内のみで保持していることを意味します。

FREE statement_name

FREE statement_nameはDEALLOCATE PREPARE statement_nameの類義語です。

33.15.3. Informix互換SQLDA記述子領域

Informix互換モードは[項33.7.2](#)の説明と異なる構造体をサポートします。以下を参照してください。

```

struct sqlvar_compat
{
    short    sqltype;
    int      sqllen;
    char     *sqldata;
    short    *sqlind;
    char     *sqlname;
    char     *sqlformat;
    short    sqltype;
    short    sqlilen;
    char     *sqliddata;
    int      sqlxid;
    char     *sqltypename;
    short    sqltypelen;
    short    sqlownerlen;
    short    sqlsourcetype;
    char     *sqlownername;
    int      sqlsourceid;
    char     *sqlilongdata;
    int      sqlflags;
    void     *sqlreserved;
};

struct sqlda_compat
{
    short    sqld;
    struct sqlvar_compat *sqlvar;
    char     desc_name[19];
    short    desc_occ;
    struct sqlda_compat *desc_next;
    void     *reserved;
};

typedef struct sqlvar_compat    sqlvar_t;
typedef struct sqlda_compat    sqlda_t;

```

大域的な属性を以下に示します。

sqld

SQLDA記述子内のフィールド数です。

sqlvar

フィールド単位の属性へのポインタです。

desc_name

未使用です。ゼロバイトで埋められます。

desc_occ

割り当てられた構造体のサイズです。

desc_next

結果セットに複数のレコードが含まれる場合、次のSQLDA構造体へのポインタです。

reserved

未使用のポインタでNULLが含まれます。 Informix互換のために保持されます。

フィールド毎の属性を以下に示します。 これらはsqlvar配列内に格納されます。

sqltype

フィールドの型です。 定数はsqltypes.h内に記載されています。

sqllen

フィールドデータ長です。

sqldata

フィールドデータへのポインタです。 このポインタはchar *型です。 指し示されるデータはバイナリ書式です。 以下に例を示します。

```

int intval;

switch (sqldata->sqlvar[i].sqltype)
{
    case SQLINTEGER:

```

```

        intval = *(int *)sqldata->sqlvar[i].sqldata;
        break;
    } ...

```

sqlind

NULL指示子へのポインタです。 DESCRIBEまたはFETCHで返される場合、常に有効なポインタです。 EXECUTE ... USING sqlda; への入力として使用される場合、NULLポインタ値はこのフィールドの値が非NULLであることを意味します。 さもなくば、有効なポインタとsqlitypeは適切に設定されなければなりません。 以下に例を示します。

```

    if (*(int2 *)sqldata->sqlvar[i].sqlind != 0)
        printf("value is NULL\n");

```

sqlname

フィールド名です。 ゼロ終端の文字列です。

sqlformat

Informixでは予約されています。 このフィールドのPQffformat() の値です。

sqlitype

NULL指示子データの型です。 サーバからデータが返される場合は常にSQLSMINTです。 パラメータ付き問い合わせでSQLDAが使用される時、データは集合型にしたがって扱われます。

sqlilen

NULL指示子データの長さです。

sqlxid

フィールドの拡張型で、PQftype() の結果です。

```

sqltypename
sqltypelen
sqlownerlen
sqlsourcetype
sqlownername
sqlsourceid
sqlflags
sqlreserved

```

未使用です。

sqlilongdata

sqlilenが32キロバイトより大きい場合sqldataと同じです。

以下に例を示します。

```

EXEC SQL INCLUDE sqlda.h;

sqlda_t      *sqlda; /* This doesn't need to be under embedded DECLARE SECTION */

EXEC SQL BEGIN DECLARE SECTION;
char *prep_stmt = "select * from table1";
int i;
EXEC SQL END DECLARE SECTION;

...

EXEC SQL PREPARE mystmt FROM :prep_stmt;

EXEC SQL DESCRIBE mystmt INTO sqlda;

printf("# of fields: %d\n", sqlda->sqld);
for (i = 0; i < sqlda->sqld; i++)
    printf("field %d: %\"s¥\"¥n", sqlda->sqlvar[i]->sqlname);

EXEC SQL DECLARE mycursor CURSOR FOR mystmt;
EXEC SQL OPEN mycursor;
EXEC SQL WHENEVER NOT FOUND GOTO out;

while (1)
{
    EXEC SQL FETCH mycursor USING sqlda;
}

EXEC SQL CLOSE mycursor;

free(sqlda); /* The main structure is all to be free(),

```

```
* sqllda and sqllda->sqlvar is in one allocated area */
```

より詳細についてはsqllda.hヘッダファイルとsrc/interfaces/ecpg/test/compat_informix/sqllda.pgcリグレッションテストを参照してください。

33.15.4. 追加関数

decadd

2つのdecimal型変数を加算します。

```
int decadd(decimal *arg1, decimal *arg2, decimal *sum);
```

この関数は、decimal型の最初の演算項目(arg1)へのポインタ、decimal型の2番目の演算項目(arg2)へのポインタ、加算結果を格納するdecimal型値(sum)へのポインタを受け付けます。成功すると、この関数は0を返します。オーバーフローが発生した場合はECPG_INFORMIX_NUM_OVERFLOWが、アンダーフローの場合はECPG_INFORMIX_NUM_UNDERFLOWが返ります。その他の失敗が発生した場合は-1が返し、errnoにはpgtypeslibにおける対応するerrno番号が設定されます。

deccmp

2つのdecimal型変数を比較します。

```
int deccmp(decimal *arg1, decimal *arg2);
```

この関数は、最初のdecimal値(arg1)へのポインタ、2番目のdecimal値(arg2)へのポインタを受け付け、どちらが大きいかを示すint値を返します。

- arg1が指し示す値がarg2が指し示す値より大きければ1。
- arg1が指し示す値がarg2が指し示す値より小さければ-1。
- arg1が指し示す値とarg2が指し示す値が同じならば0。

deccopy

decimal値をコピーします。

```
void deccopy(decimal *src, decimal *target);
```

この関数は、最初の引数としてコピー元のdecimal値(src)へのポインタ、2番目の引数としてdecimal型のコピー先構造体(target)へのポインタを受け付けます。

deccvasc

ASCII表現からdecimal型に値を変換します。

```
int deccvasc(char *cp, int len, decimal *np);
```

この関数は、変換対象の文字列表現を持つ文字列(cp)へのポインタとその文字列長lenを受け付けます。npはこの操作結果を格納するdecimal型の値へのポインタです。

有効な書式の例は以下の通りです。 -2、 .794、 +3.44、 592.49E07、 -32.84e-4。

この関数は成功時0を返します。オーバーフローやアンダーフローが発生した場合はECPG_INFORMIX_NUM_OVERFLOWやECPG_INFORMIX_NUM_UNDERFLOWが返されます。ASCII表現の解析ができなかった場合はECPG_INFORMIX_BAD_NUMERICが、指数部分の解析に問題がある場合はECPG_INFORMIX_BAD_EXPONENTが返されます。

deccvdbl

double型の値をdecimal型の値に変換します。

```
int deccvdbl(double dbl, decimal *np);
```

この関数は、最初の引数として変換対象のdouble型の変数(dbl)を受け付けます。2番目の引数(np)として、この関数は操作結果を格納するdecimal型変数へのポインタを受け付けます。

この関数は成功時に0を返します。変換が失敗した場合は負の値が返ります。

deccvint

int型の値をdecimal型の値に変換します。

```
int deccvint(int in, decimal *np);
```


この関数は最初の引数として、変換対象のint型変数(`in`)を受付けます。 2番目の引数(`np`)として、この関数は変換結果を格納するdecimal型変数へのポインタを受付けます。

この関数は成功時に0を返します。 変換が失敗した場合は負の値が返ります。

deccvlong

long型の値をdecimal型の値に変換します。

```
int deccvlong(long lng, decimal *np);
```

この関数は最初の引数として、変換対象のlong型変数(`lng`)を受付けます。 2番目の引数(`np`)として、この関数は変換結果を格納するdecimal型変数へのポインタを受付けます。

この関数は成功時に0を返します。 変換が失敗した場合は負の値が返ります。

decdiv

2つのdecimal型変数の除算を行います。

```
int decdiv(decimal *n1, decimal *n2, decimal *result);
```

この関数は、1番目の演算項目(`n1`)と2番目の演算項目(`n2`)となる変数のポインタを受付け、`n1/n2`を計算します。`result`は、操作結果を格納する変数へのポインタです。

成功時0が返され、除算の失敗時には負の値が返されます。 オーバーフローやアンダーフローが発生した場合、この関数はそれぞれECPG_INFORMIX_NUM_OVERFLOW、ECPG_INFORMIX_NUM_UNDERFLOWを返します。 0割りが発生した場合はECPG_INFORMIX_DIVIDE_ZEROが返されます。

decmul

2つのdecimal型変数を乗算します。

```
int decmul(decimal *n1, decimal *n2, decimal *result);
```

この関数は、1番目の演算項目(`n1`)と2番目の演算項目(`n2`)となる変数のポインタを受付け、`n1*n2`を計算します。`result`は、操作結果を格納する変数へのポインタです。

成功時0が返され、除算の失敗時には負の値が返されます。 オーバーフローやアンダーフローが発生した場合、この関数はそれぞれECPG_INFORMIX_NUM_OVERFLOW、ECPG_INFORMIX_NUM_UNDERFLOWを返します。

decsub

10進数型値同士の引算を行います。

```
int decsub(decimal *n1, decimal *n2, decimal *result);
```

この関数は、1番目の演算項目(`n1`)と2番目の演算項目(`n2`)となる変数のポインタを受付け、`n1-n2`を計算します。`result`は、操作結果を格納する変数へのポインタです。

成功時0が返され、除算の失敗時には負の値が返されます。 オーバーフローやアンダーフローが発生した場合、この関数はそれぞれECPG_INFORMIX_NUM_OVERFLOW、ECPG_INFORMIX_NUM_UNDERFLOWを返します。

dectoasc

decimal型変数をC char* 文字列のASCII表現に変換します。

```
int dectoasc(decimal *np, char *cp, int len, int right)
```

この関数はdecimal型変数(`np`)のポインタを受け付け、テキスト表現に変換します。 `cp`は変換結果を格納するためのバッファです。 `right`パラメータは、decimal小数点の右側の何桁を出力するかを指定します。 結果はこの10進桁数で丸められます。 `right`を-1にすることで、すべての有効な桁数が出力されるようになります。 `len`で示す出力バッファ長が、最後のNULL文字を含むテキスト表現を格納するには不十分であった場合、結果には*という1文字が格納され、-1が返されます。

この関数は、`cp`バッファが小さすぎる場合に-1を返します。 メモリ不足の場合はECPG_INFORMIX_OUT_OF_MEMORYを返します。

dectodbl

decimal型変数をdoubleに変換します。

```
int dectodbl(decimal *np, double *dblp);
```

この関数は変換対象のdecimal型変数(`np`)のポインタと処理結果を格納するdouble変数(`dblp`)へのポインタを受け付けま

す。

成功時0が、変換失敗時負の値が返されます。

dectoint

decimal型変数を整数型に変換します。

```
int dectoint(decimal *np, int *ip);
```

この関数は変換対象のdecimal型変数(`np`)のポインタと処理結果を格納するoint型変数(`lp`)へのポインタを受け付けます。

成功時0が、変換失敗時負の値が返されます。 オーバーフローが発生した場合はECPG_INFORMIX_NUM_OVERFLOWが返されます。

このECPGの実装はInformixの実装と異なることに注意してください。 Informixでは、整数範囲に-32767から32767までという制限をしていますが、ECPGでの制限はアーキテクチャに依存(-INT_MAX .. INT_MAX)します。

dectolong

decimal型変数をlong型に変換します。

```
int dectolong(decimal *np, long *lngp);
```

この関数は変換対象のdecimal型変数(`np`)のポインタと処理結果を格納するlong変数(`lngp`)へのポインタを受け付けます。

成功時0が、変換失敗時負の値が返されます。 オーバーフローが発生した場合はECPG_INFORMIX_NUM_OVERFLOWが返されます。

このECPGの実装はInformixの実装と異なることに注意してください。 Informixでは、整数範囲に-2,147,483,647から2,147,483,647までという制限をしていますが、ECPGでの制限はアーキテクチャに依存(-LONG_MAX .. LONG_MAX)します。

rdatestr

date型をC char*文字列に変換します。

```
int rdatestr(date d, char *str);
```

この関数は2つの引数を受け付けます。 最初の引数は変換対象のdate型(`d`)、2番目は変換後の文字列へのポインタです。 出力書式は常にyyyy-mm-ddですので、少なくとも11文字（NULL終端を含む）を結果文字列に割り当てなければなりません。

この関数は成功時0を、エラー時負の値を返します。

このECPGの実装はInformixの実装と異なることに注意してください。 Informixでは、環境変数により書式を変更できますが、ECPGでは出力書式を変更することはできません。

rstrdate

date型のテキスト表現を解析します。

```
int rstrdate(char *str, date *d);
```

この関数は、変換対象のdate型のテキスト表現(`str`)とdate型変数のポインタ(`d`)を受け付けます。 この関数では書式マスクを指定することができません。 Informixのデフォルトの書式マスクであるmm/dd/yyyyを使用します。 内部的には、この関数はrdefmtdateを使用して実装しています。 したがってrstrdateは速くありません。 もし選択肢があるのであれば、書式マスクを明示的に指定することができるrdefmtdateを選択すべきです。

この関数はrdefmtdateと同様の値を返します。

rtoday

現在の日付を（date型で）入手します。

```
void rtoday(date *d);
```

この関数はdate型変数(`d`)へのポインタを受け付け、そこに現在の日付を格納します。

内部的には、この関数はPGTYPESdate_today関数を使用します。

rjulmdy

date型変数から、日、月、年の値を取り出します。

```
int rjulmdy(date d, short mdy[3]);
```

この関数は日付`d`、3つのshort integer型の値からなる配列`mdy`へのポインタを受付けます。この変数名はその並びを表し、`mdy[0]`には月数、`mdy[1]`には日数が、`mdy[2]`には年が入ります。

現在この関数は常に0を返します。

内部的にはこの関数は[PGTYPESdate_julmdy](#)関数を使用します。

rdefmtdate

書式マスクを使用して、文字列をdate型の値に変換します。

```
int rdefmtdate(date *d, char *fmt, char *str);
```

この関数は、処理結果を格納するためのdate型へのポインタ(`d`)、日付を解析するための書式マスク(`fmt`)、dateのテキスト表現を含むCのchar*文字列(`str`)を受付けます。テキスト表現は書式マスクに合った表現であることが仮定されています。しかし、文字列と書式マスクを1:1に対応付けする必要はありません。この関数は並んだ順番に解析し、年の位置を表す`yy`または`yyyy`を、月の位置を表す`mm`を、日の位置を表す`dd`を検索します。

この関数は以下の値を返します。

- 0 - 関数が正常に終了しました。
- ECPG_INFORMIX_ENOSHORTDATE - 日付に、日、月、年を区切る文字がありませんでした。この場合、入力文字列は6バイト、8バイトのいずれかでなければなりません、そうではありませんでした。
- ECPG_INFORMIX_ENOTDMY - 書式文字列が正しく年月日の順番を示していません。
- ECPG_INFORMIX_BAD_DAY - 入力文字列に有効な日が含まれていません。
- ECPG_INFORMIX_BAD_MONTH - 入力文字列に有効な月が含まれていません。
- ECPG_INFORMIX_BAD_YEAR - 入力文字列に有効な年が含まれていません。

内部的には、この関数は[PGTYPESdate_defmt_asc](#)関数を使用して実装しています。この関数の説明には、入力例の表がありますので、こちらも参照してください。

rfmtdate

書式マスクを使用してdate型変数をテキスト表現に変換します。

```
int rfmtdate(date d, char *fmt, char *str);
```

この関数は変換対象の日付(`d`)、書式マスク(`fmt`)、日付のテキスト表現を格納する文字列(`str`)を受付けます。

成功時0、エラーが発生した場合は負の値が返されます。

内部的にはこの関数は[PGTYPESdate_fmt_asc](#)関数を使用します。例が記載されていますので、こちらも参照してください。

rmidyjul

日付の日、月、年を表す3つのshort integer型からなる配列から日付型の値を作成します。

```
int rmidyjul(short mdy[3], date *d);
```

この関数は3つのshort integer型からなる配列(`mdy`)と処理結果を格納するdate型変数へのポインタを受付けます。

現在この関数は常に0を返します。

内部的にはこの関数は[PGTYPESdate_mdyjul](#)関数を使用して実装しています。

rdayofweek

日付型の値の週内日数を示す値を返します。

```
int rdayofweek(date d);
```

この関数はdate型変数`d`をその唯一の引数として受け、その日付の週内日数を示す整数を返します。

- 0 - 日曜

- 1 - 月曜
- 2 - 火曜
- 3 - 水曜
- 4 - 木曜
- 5 - 金曜
- 6 - 土曜

内部的にはこの関数は [PGTYPESdate_dayofweek](#) 関数を使用して実装しています。

`dtcurrent`

現在のタイムスタンプを取り出します。

```
void dtcurrent(timestamp *ts);
```

この関数は現在のタイムスタンプを受け取り、`ts` が指し示すタイムスタンプ型変数に格納します。

`dtcvasc`

テキスト表現からtimestamp型変数にタイムスタンプを解析します。

```
int dtcvasc(char *str, timestamp *ts);
```

この関数は対象の文字列(`str`)と処理結果を格納するtimestamp型変数(`ts`)へのポインタを受け付けます。

この関数は成功時0を返し、エラー時負の値を返します。

内部的にはこの関数は[PGYPEStimestamp_from_asc](#)関数を使用します。 入力例の表がありますので、こちらも参照してください。

`dtcvfmtasc`

書式マスクを使用してタイムスタンプのテキスト表現をtimestamp型変数に変換します。

```
dtcvfmtasc(char *inbuf, char *fmtstr, timestamp *dtvalue)
```

この関数は、対象とする文字列(`inbuf`)、使用する書式マスク(`fmtstr`)、処理結果を格納するtimestamp変数(`dtvalue`)へのポインタを受け付けます。

この関数は[PGYPEStimestamp_defmt_asc](#)関数を使用して実装されています。 使用可能な書式指定のリストがありますので、こちらも参照してください。

この関数は成功時に0を、エラー時負の値を返します。

`dtsub`

timestamp型同士で減算を行い、interval型変数を返します。

```
int dtsub(timestamp *ts1, timestamp *ts2, interval *iv);
```

この関数は`ts1` が指し示すtimestamp型変数から`ts2` が指し示すtimestamp型変数を引き去ります。 結果は`iv` が指し示すinterval型変数に格納されます。

成功時この関数は0を返し、エラー時負の値を返します。

`dttoasc`

timestamp型変数をC char*文字列に変換します。

```
int dttoasc(timestamp *ts, char *output);
```

この関数は対象のtimestamp型変数(`ts`)へのポインタ、処理結果を格納する文字列(`output`)を受け付けます。 これは`ts`を標準SQLに従うテキスト表現 (YYYY-MM-DD HH:MM:SSとして定義) に変換します。

成功時この関数は0を返し、エラー時負の値を返します。

`dttofmtasc`

書式マスクを使用してtimestamp型変数をC char*に変換します。

```
int dttofmtasc(timestamp *ts, char *output, int str_len, char *fmtstr);
```

この関数は、最初の引数として変換対象のタイムスタンプ(*ts*)を、出力バッファのポインタ(*output*)、出力バッファで割当て可能な最大長 (*str_len*)、変換に使用する書式マスク(*fmtstr*)を受付けます。

成功時この関数は0を返します。エラーが発生した場合は負の値を返します。

内部的に、この関数は[PGTYPEStimestamp fmt asc](#)関数を使用します。 使用できる書式マスクに関する情報がありますので、こちらも参照してください。

intoasc

interval型変数をC char*文字列に変換します。

```
int intoasc(interval *i, char *str);
```

この関数は、変換対象のinterval型変数(*i*)へのポインタ、処理結果を格納する文字列(*str*)を受付けます。 これは*i*を標準SQLに従うテキスト表現 (YYYY-MM-DD HH:MM:SSとして定義) に変換します。

成功時、この関数は0を返します。 エラーが発生した場合は負の値を返します。

rfmtlong

long integer値を書式マスクを使用してテキスト表現に変換します。

```
int rfmtlong(long lng_val, char *fmt, char *outbuf);
```

この関数は、long型の値*lng_val*、書式マスク*fmt*、出力バッファ*outbuf*へのポインタを受付けます。 これはlong型の値を書式マスクに従ってテキスト表現に変換します。

書式マスクは以下の書式指定文字を組み合わせることができます。

- * (アスタリスク) - この位置が空白ならばアスタリスクで埋めます。
- & (アンパサンド) - この位置が空白ならば0で埋めます。
- # - 先頭のゼロを空白に変換します。
- < - 文字列内で数値を左そろえます。
- , (カンマ) - 4桁以上の数値をカンマで区切った3桁にグループ化します。
- . (ピリオド) - この文字は数値から小数部分を区別します。
- - (マイナス) - 数値が負の場合、マイナス記号を付けます。
- + (プラス) - 数値が正の場合プラス記号を付けます。
- (- これは負の値の先頭のマイナス記号を置き換えます。 マイナス記号は現れません。
-) - この文字はマイナス記号を置き換え、負の値の最後に出力します。
- \$ - 通貨記号

rupshift

文字列を大文字に変換します。

```
void rupshift(char *str);
```

この関数は文字列へのポインタを受付け、すべての小文字を大文字に変換します。

byleng

文字列内の文字数を返します。 ただし、末尾の空白は数えません。

```
int byleng(char *str, int len);
```

この関数は最初の引数として、固定長の文字列(*str*)を、2番目の引数としてその文字列長 (*len*)想定しています。 これは、文字列から末尾の空白を取り除いた、有効文字の数を返します。

ldchar

固定長の文字列をNULL終端の文字列に複製します。

```
void ldchar(char *src, int len, char *dest);
```

この関数はコピー対象の固定長の文字列(`src`)、文字列長(`len`)、格納先メモリ(`dest`)へのポインタを受付けます。`dest`が指し示す文字列には少なくとも`len+1`バイトを割り当てなければならない点に注意してください。この関数は多くても`len`バイトを新しい場所にコピーします。(元の文字列が末尾に空白文字を持つ場合に少なくなります。)そして、NULL終端を付与します。

rgetmsg

```
int rgetmsg(int msgnum, char *s, int maxsize);
```

この関数は存在しますが、現在実装されていません。

rtypalign

```
int rtypalign(int offset, int type);
```

この関数は存在しますが、現在実装されていません。

rtypmsize

```
int rtypmsize(int type, int len);
```

この関数は存在しますが、現在実装されていません。

rtypwidth

```
int rtypwidth(int sqltype, int sqllen);
```

この関数は存在しますが、現在実装されていません。

rsetnull

変数にNULLを設定します。

```
int rsetnull(int t, char *ptr);
```

この関数は、変数の種類を示す整数とC `char*`にキャストした変数自体へのポインタを受付けます。

以下の種類が存在します。

- `CCHARTYPE` - `char`または `char*`型の変数用
- `CSHORTTYPE` - `short int`型の変数用
- `CINTTYPE` - `int`型の変数用
- `CBOOLTYPE` - `boolean`型の変数用
- `CFLOATTYPE` - `float`型の変数用
- `CLONGTYPE` - `long`型の変数用
- `CDOUBLETTYPE` - `double`型の変数用
- `CDECIMALTYPE` - `decimal`型の変数用
- `CDATETYPE` - `date`型の変数用
- `CDTIMETYPE` - `timestamp`型の変数用

以下にこの関数の呼び出し例を示します。

```
$char c[] = "abc          ";
$short s = 17;
$int i = -74874;

rsetnull(CCHARTYPE, (char *) c);
rsetnull(CSHORTTYPE, (char *) &s);
rsetnull(CINTTYPE, (char *) &i);
```

risnull

変数がNULLか検査します。

```
int risnull(int t, char *ptr);
```

この関数は検査する変数の種類(`t`)、変数(`ptr`)へのポインタを受付けます。後者は`char*`にキャストする必要があること

に注意してください。 取り得る変数種類については [rsetnull](#)関数を参照してください。

この関数の使用方法の例を示します。

```
$char c[] = "abc          ";
$short s = 17;
$int i = -74874;

risnull(CCHARTYPE, (char *) c);
risnull(CSHORTTYPE, (char *) &s);
risnull(CINTTYPE, (char *) &i);
```

33.15.5. 追加の定数

ここで示す定数はすべてエラーを示すものであり、負の値を表すように定義されていることに注意してください。 また、他の定数の説明では、現在の実装で定数が表す数値がわかります。 しかし、この数値に依存してはなりません。 しかし、これらのすべてが負の値であることに依存することは可能です。

ECPG_INFORMIX_NUM_OVERFLOW

計算時にオーバーフローが発生した場合、関数はこの値を返します。 内部的には-1200（Informixの定義）と定義されています。

ECPG_INFORMIX_NUM_UNDERFLOW

計算時にアンダーフローが発生した場合、関数はこの値を返します。 内部的には-1201（Informixの定義）と定義されています。

ECPG_INFORMIX_DIVIDE_ZERO

計算時にゼロ除算が発生した場合、関数はこの値を返します。 内部的には-1202（Informixの定義）と定義されています。

ECPG_INFORMIX_BAD_YEAR

日付の解析時に年の値が不正であった場合、関数はこの値を返します。 内部的には-1204（Informixの定義）と定義されています。

ECPG_INFORMIX_BAD_MONTH

日付の解析時に月の値が不正であった場合、関数はこの値を返します。 内部的には-1205（Informixの定義）と定義されています。

ECPG_INFORMIX_BAD_DAY

日付の解析時に日の値が不正であった場合、関数はこの値を返します。 内部的には-1206（Informixの定義）と定義されています。

ECPG_INFORMIX_ENOSHORTDATE

解析処理が短縮日付表現を必要にしているが、正しい長さの日付文字列が得られなかった場合、関数はこの値を返します。 内部的には-1209（Informixの定義）と定義されています。

ECPG_INFORMIX_DATE_CONVERT

日付の書式付けの時にエラーが発生した場合、関数はこの値を返します。 内部的には-1210（Informixの定義）と定義されています。

ECPG_INFORMIX_OUT_OF_MEMORY

操作時にメモリが不足した場合、関数はこの値を返します。 内部的には-1211（Informixの定義）と定義されています。

ECPG_INFORMIX_ENOTDMY

解析処理が書式マスク（`mmddyy`のような）が存在することを前提としているが、すべてのフィールドが正しく列挙されていない場合、関数はこの値を返します。 内部的には-1212（Informixの定義）と定義されています。

ECPG_INFORMIX_BAD_NUMERIC

解析処理がエラーのため数値のテキスト表現を解析できなかった場合や数値変数の少なくとも1つが無効のため数値変数を使用した計算を完了できなかった場合、関数はこの値を返します。 内部的には-1213（Informixの定義）と定義されています。

ECPG_INFORMIX_BAD_EXPONENT

解析処理がが指数の解析を行うことができなかった場合、関数はこの値を返します。 内部的には-1216（Informixの定義）と定義されています。

ECPG_INFORMIX_BAD_DATE

解析処理が日付を解析できなかった場合、関数はこの値を返します。 内部的には-1218（Informixの定義）と定義されています。

ECPG_INFORMIX_EXTRA_CHARS

解析処理が追加の文字列を解析できなかった場合、関数はこの値を返します。 内部的には-1264（Informixの定義）と定義されています。

[前のページ](#)
WHENEVER

[ホーム](#)
[上に戻る](#)

[次のページ](#)
内部

33.16. 内部

本節では内部的なECPGの動作を説明します。この情報はECPGの使用方法を理解する手助けとして有用なことがあります。

`ecpg`によって出力に書き込まれる最初の4行は固定されています。2行はコメントで、残り2行はライブラリとのインタフェースのために必要なインクルード行です。その後、プリプロセッサはファイル全体を読み取り、出力に書き出します。通常は、単にすべてそのまま出力に書き出します。

`EXEC SQL`を検出すると、間に入り、それを変更します。このコマンドは`EXEC SQL`で始まり、`;`で終わります。この間のすべてはSQL文として扱われ、変数の置換のために解析されます。

変数置換は、シンボルがコロン(:)から始まる場合に発生します。その名前の変数が、`EXEC SQL DECLARE`セクションで事前に宣言された変数の中から検索されます。

ライブラリ内で最も重要な関数は`ECPGdo`です。これが、ほとんどのコマンドの実行を管理します。可変長の引数をとります。すべてのプラットフォームで問題にならないことを祈っていますが、これは50程度の引数まで簡単に追加できます。

引数を以下に示します。

行番号

元の行の行番号です。エラーメッセージ内でのみ使用されます。

文字列

発行すべきSQLコマンドです。入力変数、つまり、コンパイル時に未知だったがそのコマンド内に与えるべき変数によって変更されます。変数が文字列内に挿入される箇所は?となっています。

入力変数

すべての入力変数は10個の引数を作成します(後述)。

`ECPGt_EOIT`

入力変数がもうないことを表す`enum`です。

出力変数

すべての出力変数は10個の引数を作成します(後述)。これらの変数は関数によって埋められます。

`ECPGt_EORT`

変数がもうないことを表す`enum`です。

SQLコマンドの一部となるすべての変数に対して、この関数は以下の10個の引数を生成します。

- 1 11 特別シンボルとしての型。
- 1 11 値へのポインタ、もしくはポインタのポインタ。
- 1 11 変数が`char`か`varchar`の場合はそのサイズ。
- 1 11 配列の要素数(配列取り出し用)。
- 1 11 配列の次の要素のオフセット(配列取り出し用)。
- 1 11 特別シンボルとしての指示子変数の型。
- 1 11 指示子変数へのポインタ。
- 1 11 0
- 1 11 指示子配列内の要素数(配列取り出し用)。
- 1 11 指示子配列内の次要素へのオフセット(配列取り出し用)。

すべてのSQLコマンドがこの方法で扱われるわけではないことに注意してください。例えば、以下のカーソルを開くSQL文は出

力にコピーされません。

```
EXEC SQL OPEN cursor;
```

その代わりにカーソルのDECLAREコマンドがOPENコマンドの場所で使用されます。実際にこのコマンドがカーソルを開くからです。

以下に、foo.pgcファイルに対するプリプロセッサの出力を完全に説明する例を示します（プリプロセッサのバージョンによって詳細が異なっているかもしれません）。

```
EXEC SQL BEGIN DECLARE SECTION;
int index;
int result;
EXEC SQL END DECLARE SECTION;

...
EXEC SQL SELECT res INTO :result FROM mytable WHERE index = :index;
```

これは以下に翻訳されます。

```
/* Processed by ecpg (2.6.0) */
/* These two include files are added by the preprocessor */
#include <ecpgtype.h>;
#include <ecpglib.h>;

/* exec sql begin declare section */

#line 1 "foo.pgc"

    int index;
    int result;
/* exec sql end declare section */
...
ECPGdo(__LINE__, NULL, "SELECT res FROM mytable WHERE index = ?      ",
        ECPGt_int,&(index),1L,1L,sizeof(int),
        ECPGt_NO_INDICATOR, NULL , 0L, 0L, 0L, ECPGt_EOIT,
        ECPGt_int,&(result),1L,1L,sizeof(int),
        ECPGt_NO_INDICATOR, NULL , 0L, 0L, 0L, ECPGt_EORT);
#line 147 "foo.pgc"
```

ここで可読性のためにインデントを付けています。プリプロセッサが行ったものではありません。

[前のページ](#)

Informix互換モード

[ホーム](#)
[上に戻る](#)

[次のページ](#)
情報スキーマ

第 34 章 情報スキーマ

目次

- 34.1. [スキーマ](#)
- 34.2. [データ型](#)
- 34.3. [information_schema_catalog_name](#)
- 34.4. [administrable_role_authorizations](#)
- 34.5. [applicable_roles](#)
- 34.6. [attributes](#)
- 34.7. [character_sets](#)
- 34.8. [check_constraint_routine_usage](#)
- 34.9. [check_constraints](#)
- 34.10. [collations](#)
- 34.11. [collation_character_set_applicability](#)
- 34.12. [column_domain_usage](#)
- 34.13. [column_options](#)
- 34.14. [column_privileges](#)
- 34.15. [column_udt_usage](#)
- 34.16. [columns](#)
- 34.17. [constraint_column_usage](#)
- 34.18. [constraint_table_usage](#)
- 34.19. [data_type_privileges](#)
- 34.20. [domain_constraints](#)
- 34.21. [domain_udt_usage](#)
- 34.22. [domains](#)
- 34.23. [element_types](#)
- 34.24. [enabled_roles](#)
- 34.25. [foreign_data_wrapper_options](#)
- 34.26. [foreign_data_wrappers](#)
- 34.27. [foreign_server_options](#)
- 34.28. [foreign_servers](#)
- 34.29. [foreign_table_options](#)
- 34.30. [foreign_tables](#)
- 34.31. [key_column_usage](#)
- 34.32. [parameters](#)
- 34.33. [referential_constraints](#)
- 34.34. [role_column_grants](#)
- 34.35. [role_routine_grants](#)
- 34.36. [role_table_grants](#)
- 34.37. [role_udt_grants](#)
- 34.38. [role_usage_grants](#)
- 34.39. [routine_privileges](#)
- 34.40. [routines](#)
- 34.41. [schemata](#)
- 34.42. [sequences](#)
- 34.43. [sql_features](#)
- 34.44. [sql_implementation_info](#)
- 34.45. [sql_languages](#)
- 34.46. [sql_packages](#)
- 34.47. [sql_parts](#)
- 34.48. [sql_sizing](#)
- 34.49. [sql_sizing_profiles](#)
- 34.50. [table_constraints](#)
- 34.51. [table_privileges](#)
- 34.52. [tables](#)
- 34.53. [triggered_update_columns](#)
- 34.54. [triggers](#)
- 34.55. [udt_privileges](#)
- 34.56. [usage_privileges](#)
- 34.57. [user_defined_types](#)
- 34.58. [user_mapping_options](#)

- 34.59. [user_mappings](#)
- 34.60. [view_column_usage](#)
- 34.61. [view_routine_usage](#)
- 34.62. [view_table_usage](#)
- 34.63. [views](#)

情報スキーマは、現在のデータベースで定義されたオブジェクトについての情報を持つビューの集合から構成されます。情報スキーマは標準SQLで定義されています。したがって、PostgreSQLに特化し、実装上の事項にならって作成されるシステムカタログとは異なり、移植性があり、安定性を保持できるものと期待することができます。しかし、情報スキーマのビューには、PostgreSQL固有の機能についての情報が含まれていません。これに問い合わせを行うためには、システムカタログもしくはPostgreSQL固有のビューに問い合わせを行う必要があります。

注意: 制約情報についてデータベースに問い合わせるとき、一行を返すことが想定される標準に準拠したクエリが数行の結果を返す場合があります。これは、制約名がスキーマ内で一意になることをSQL標準が要求しているのに対して、PostgreSQLはこの制約を強制しないためです。PostgreSQLは自動生成される制約の名前がスキーマ内で重複することを防ぎますが、ユーザは重複する名前を指定することができます。

この問題

は、`check_constraint_routine_usage`や`check_constraints`、`domain_constraints`、`referential_constraints`といった情報スキーマビューを検索するときに表面化することがあります。いくつかの他のビューにも同様の問題がありますが、重複行を識別する助けになるテーブル名を含んでいます。例えば`constraint_column_usage`や`constraint_table_usage`、`table_constraints`などです。

[前のページ](#)
内部

[ホーム](#)
[上に戻る](#)

[次のページ](#)
スキーマ

34.1. スキーマ

情報スキーマ自身は、`information_schema` という名前のスキーマです。このスキーマは自動的にすべてのデータベース内に存在します。このスキーマの所有者は、クラスタ内の最初のデータベースユーザであり、当然このユーザは、スキーマの削除を含むスキーマについてのすべての権限を持ちます（しかし、削除したとしても拙訳できる領域はわずかです）。

デフォルトでは、情報スキーマはスキーマの検索パスには含まれません。ですので、修飾した名前情報スキーマ内のすべてのオブジェクトにアクセスする必要があります。情報スキーマ内の一部のオブジェクトの名前はユーザアプリケーションでも使用されるような一般的な名前であるため、情報スキーマをパスに追加する場合は注意しなければなりません。

34.2. データ型

情報スキーマのビューの列では、情報スキーマ内で定義された特殊なデータ型を使用します。これらは、通常の組み込み型の上位にあたる単純なドメインとして定義されます。情報スキーマ外部で操作する時にはこれらの型を使用してはなりません。しかし、情報スキーマを検索するようなアプリケーションではこれらの型への用意をしておかなければなりません。

これらの型を以下に記します。

`cardinal_number`

非負の整数です。

`character_data`

(最大文字長の指定がない) 文字列です。

`sql_identifier`

文字列です。この型はSQL識別子用に使用され、`character_data`はその他の任意のテキストデータに使用されます。

`time_stamp`

`timestamp with time zone`型の上位ドメインです。

`yes_or_no`

`YES`か`NO`のいずれかを持つ文字列ドメインです。情報スキーマ内で論理(真/偽)データを表すために使用されます。(情報スキーマは`boolean`型が標準SQLに追加される前に考案されました。このため情報スキーマの後方互換性を維持するために、この記法が必要です。)

情報スキーマ内の列はすべてこれら5つの型のいずれかを取ります。

34.3. information_schema_catalog_name

information_schema_catalog_nameは、常に現在のデータベース名（SQL用語では現在のカタログ）を持つ1行1列からなるテーブルです。

表 34-1. information_schema_catalog_nameの列

名前	データ型	説明
catalog_name	sql_identifier	この情報スキーマを持つデータベースの名前です。

34.4. administrable_role_authorizations

administrable_role_authorizationsビューは、現在のユーザがアドミンオプションを持つすべてのロールを識別します。

表 34-2. administrable_role_authorizationsの列

名前	データ型	説明
grantee	sql_identifier	このロールのメンバ資格を付与したロールの名前です。（現在のユーザかもしれませんが、入れ子状のロールメンバ資格の場合は異なるロールかもしれません。）
role_name	sql_identifier	ロール名です。
is_grantable	yes_or_no	常にYESです。

34.5. applicable_roles

applicable_rolesビューは、その権限を現在のユーザが使用することができるすべてのロールを識別します。これは、現在のユーザから問題のロールに付与されたロールの連鎖が存在することを意味します。現在のユーザ自身もまた適用可能なロールです。適用可能なロール群は通常権限の検査に使用されます。

表 34-3. applicable_rolesの列

名前	データ型	説明
grantee	sql_identifier	このロールのメンバ資格を付与したロールの名前です。（現在のユーザかもしれませんが、入れ子状のロールメンバ資格の場合は異なるロールかもしれません。）
role_name	sql_identifier	ロールの名前です。
is_grantable	yes_or_no	付与者がこのロールにアドミンオプションを持つ場合YES、さもなければNOです。

34.6. attributes

`attributes` ビューには、データベース内で定義された複合データ型の属性に関する情報が含まれます。（このビューが、PostgreSQLコンテキスト内でよく呼び出される属性である、テーブル列に関する情報を持たない点に注意してください。）現在のユーザが（所有者であるかまたは複合データ型に対する権限を持っていて）アクセスする権限を持つ属性のみが表示されます。

表 34-4. `attributes` の列

名前	データ型	説明
<code>udt_catalog</code>	<code>sql_identifier</code>	データ型を含むデータベースの名前です。（常に現在のデータベースです。）
<code>udt_schema</code>	<code>sql_identifier</code>	データ型を含むスキーマの名前です。
<code>udt_name</code>	<code>sql_identifier</code>	データ型の名前です。
<code>attribute_name</code>	<code>sql_identifier</code>	属性の名前です。
<code>ordinal_position</code>	<code>cardinal_number</code>	データ型の属性の序数位置です。（1から始まります。）
<code>attribute_default</code>	<code>character_data</code>	属性のデフォルト式です。
<code>is_nullable</code>	<code>yes_or_no</code>	属性がNULLを持つことができる場合はYES、さもなければNOです。
<code>data_type</code>	<code>character_data</code>	属性のデータ型が組み込み型の場合、そのデータ型です。何らかの配列の場合、ARRAYです。（この場合、 <code>element_types</code> ビューを参照してください。）さもなければUSER-DEFINEDです。（この場合、型は <code>attribute_udt_name</code> と関連する列により識別されます。）
<code>character_maximum_length</code>	<code>cardinal_number</code>	<code>data_type</code> が文字列またはビット列を識別する場合、その宣言された最大長です。他のデータ型または最大長が宣言されていない場合はNULLです。
<code>character_octet_length</code>	<code>cardinal_number</code>	<code>data_type</code> が文字列を識別する場合、オクテット（バイト）単位で表したデータの最大長です。他のデータ型ではNULLです。最大オクテット長は宣言された文字最大長(上述)とサーバ符号化方式に依存します。
<code>character_set_catalog</code>	<code>sql_identifier</code>	PostgreSQLでは利用できない機能に適用されるものです。
<code>character_set_schema</code>	<code>sql_identifier</code>	PostgreSQLでは利用できない機能に適用されるものです。
<code>character_set_name</code>	<code>sql_identifier</code>	PostgreSQLでは利用できない機能に適用されるものです。
<code>collation_catalog</code>	<code>sql_identifier</code>	属性の照合を含むデータベース(常に現在のデータベース)の名前で、デフォルトであるか属性のデータ型が照合可能でない場合はNULLです。
<code>collation_schema</code>	<code>sql_identifier</code>	属性の照合を含むスキーマの名前で、デフォルトであるか属性のデータ型が照合可能でない場合はNULLです。
<code>collation_name</code>	<code>sql_identifier</code>	属性の照合の名前で、デフォルトであるか属性のデータ型が照合可能でない場合はNULLです。
<code>numeric_precision</code>	<code>cardinal_number</code>	<code>data_type</code> が数値型を識別する場合、この列は属性の型の（宣言された、あるいは暗黙的な）精度です。この精度は有効桁を意味します。 <code>numeric_precision_radix</code> 列の指定に従って、(10を基とした)10進数、または(2を基とした)2進数表記で表現されます。他のデータ型ではこの列はNULLです。
<code>numeric_precision_radix</code>	<code>cardinal_number</code>	<code>data_type</code> が数値型を識別する場合、この列は、 <code>numeric_precision</code> および <code>numeric_scale</code> で表現される値の基が何かを識別します。この値は2または10です。他のデータ型ではこの列はNULLです。
<code>numeric_scale</code>	<code>cardinal_number</code>	<code>data_type</code> が数値型を識別する場合、この列は、属性の型の（宣言された、あるいは暗黙的な）位取りが含まれます。位取りは少数点以下の有効桁数を意味します。 <code>numeric_precision_radix</code> 列の指定に従って、(10を基とした)10進数、または(2を基とした)2進数表記で表現されます。他のデータ型ではこの列はNULLです。
<code>datetime_precision</code>	<code>cardinal_number</code>	<code>data_type</code> が日付、時刻、タイムスタンプ、または間隔型を示す場合、この列は（宣言されたか暗黙的な）この属性に対する分数秒精度を包含します。つまり、2番目の値の小数点に続く保存された10進桁数です。他

		の全ての型に対してこの列はNULLです。
interval_type	character_data	もしdata_typeがインターバル型を示す場合、この列はこの属性のインターバル値がどのフィールドを含むかの仕様を含みます。例えば、YEAR TO MONTH、DAY TO SECONDなどです。もしフィールド制約が指定されていない(インターバルが全てのフィールドを受け付ける)場合や、他の全てのデータ型の場合はこのフィールドはNULLです。
interval_precision	cardinal_number	PostgreSQLで利用できない機能に適用されるものです。(インターバル型の属性の秒未満の精度についてはdatetime_precisionを参照してください)
attribute_udt_catalog	sql_identifier	属性のデータ型が定義されたデータベースの名前です。(常に現在のデータベースです。)
attribute_udt_schema	sql_identifier	属性のデータ型が定義されたスキーマの名前です。
attribute_udt_name	sql_identifier	属性のデータ型の名前です。
scope_catalog	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
scope_schema	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
scope_name	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
maximum_cardinality	cardinal_number	常にNULLです。PostgreSQLでは配列の次数は無制限だからです。
dtd_identifier	sql_identifier	列のデータ型記述子の、テーブルに属するデータ型記述子内で一意な識別子です。この識別子のインスタンスを結合する時に、主に有用です。(識別子の書式仕様は定義されておらず、今後のバージョンで同一性を維持する保証もありません。)
is_derived_reference_attribute	yes_or_no	PostgreSQLで利用できない機能に適用されるものです。

後述の[項34.16](#)も参照してください。ビューの構造が似ていますし、一部の列では更なる情報も記載されています。

34.7. character_sets

このcharacter_setsビューは、現在のデータベースで利用可能な文字セットを識別します。PostgreSQLはひとつのデータベース内で複数の文字セットをサポートしないので、このビューは常にデータベースエンコーディングの一行だけを表示します。

以下の用語のSQL標準での使われ方に注意してください。

文字集合

例えばUNICODEやUCS、LATIN1といった抽象的な文字集合です。SQLオブジェクトとしてはできませんが、このビューで参照できます。

文字符号化形式

文字集合の符号化方式です。ほとんどの古い文字集合はひとつの符号化形式を使うため、それらについては分離した名称はありません(たとえば、LATIN1はLATIN1集合に適用可能な符号化形式です)。しかし例えばUnicodeにはUTF8、UTF16などの符号化形式があります (PostgreSQLでは一部だけサポートしています)。符号化形式はSQLオブジェクトとして表にできませんが、このビューで参照できます。

文字セット

文字集合、文字符号化方式とデフォルトの照合を識別する順序名前付きのSQLオブジェクトです。定義済みの文字セットは、一般的に符号化形式と同じ名前を持ちますが、ユーザは他の名前を定義することができます。例えば、文字セットUTF8は一般的に文字集合UCS、符号化形式UTF8と何らかのデフォルト照合を識別します。

PostgreSQLにおける"encoding"は、文字セットまたは文字符号化形式のいずれかと考えられます。これらは同じ名前を持ち一つのデータベースでは一つだけ存在できます。

表 34-5. character_setsの列

名前	データ型	説明
character_set_catalog	sql_identifier	文字セットはスキーマオブジェクトとして実装されていないので、この列はNULLです。
character_set_schema	sql_identifier	文字セットはスキーマオブジェクトとして実装されていないので、この列はNULLです。
character_set_name	sql_identifier	文字セットの名前で、現在はデータベースエンコーディングを表示するように実装されています。
character_repertoire	sql_identifier	文字集合で、エンコーディングがUTF8の場合はUCSを、それ以外の場合は単にエンコーディング名を表示します。
form_of_use	sql_identifier	文字符号化形式で、データベースエンコーディングと同じです。
default_collate_catalog	sql_identifier	デフォルト照合を含むデータベース(いずれかの照合が識別された場合は常に現在のデータベース)の名前です。
default_collate_schema	sql_identifier	デフォルト照合を含むスキーマの名前です。
default_collate_name	sql_identifier	デフォルト照合の名前です。デフォルト照合は、現在のデータベースのCOLLATEとCTYPE設定に一致する照合として識別されます。そのような照合が存在しない場合は、この列や対応するスキーマやカタログの列はNULLです。

34.8. check_constraint_routine_usage

check_constraint_routine_usageは検査制約で使用する処理（関数およびプロシージャ）を識別します。 現在有効なロールが所有する処理のみが表示されます。

表 34-6. check_constraint_routine_usageの列

名前	データ型	説明
constraint_catalog	sql_identifier	制約が含まれるデータベースの名前です（常に現在のデータベースです）。
constraint_schema	sql_identifier	制約が含まれるスキーマの名前です。
constraint_name	sql_identifier	制約の名前です。
specific_catalog	sql_identifier	関数が含まれるデータベースの名前です（常に現在のデータベースです）。
specific_schema	sql_identifier	関数が含まれるスキーマの名前です。
specific_name	sql_identifier	関数の"仕様名称"です。 詳細は 項34.40 を参照してください。

34.9. check_constraints

check_constraints ビューには、現在有効なロールが所有している、テーブル上もしくはドメイン上のどちらかにある、全ての検査制約が含まれます（テーブルもしくはドメインの所有者がこの制約の所有者です）。

表 34-7. check_constraints の列

名前	データ型	説明
constraint_catalog	sql_identifier	この制約を持つデータベースの名前です（常に現在のデータベースです）。
constraint_schema	sql_identifier	この制約を持つスキーマの名前です。
constraint_name	sql_identifier	制約の名前です。
check_clause	character_data	この検査制約の検査式です。

34.10. collations

collationsビューは現在のデータベースで利用可能な照合を含みます

表 34-8. collationsの列

名前	データ型	説明
collation_catalog	sql_identifier	照合を含むデータベース(常に現在のデータベース)の名前です。
collation_schema	sql_identifier	照合を含むスキーマの名前です。
collation_name	sql_identifier	デフォルト照合の名前です。
pad_attribute	character_data	常にNO PADです。(もう一方のPAD SPACEはPostgreSQLではサポートされていません。)

34.11. collation_character_set_applicability

collation_character_set_applicabilityビューは、利用可能な照合がどの文字セットに適用可能かを識別します。PostgreSQLでは、データベースごとに一つの文字セットしか存在しない([項34.7](#)の説明を参照してください)ので、このビューは有益な情報を提供しません。

表 34-9. collation_character_set_applicabilityの列

名前	データ型	説明
collation_catalog	sql_identifier	照合を含むデータベース(常に現在のデータベース)の名前です。
collation_schema	sql_identifier	照合を含むスキーマの名前です。
collation_name	sql_identifier	デフォルト照合の名前です。
character_set_catalog	sql_identifier	文字セットはスキーマオブジェクトとして実装されていないので、この列はNULLです。
character_set_schema	sql_identifier	文字セットはスキーマオブジェクトとして実装されていないので、この列はNULLです。
character_set_name	sql_identifier	文字セットの名前です。

34.12. column_domain_usage

column_domain_usage ビューは、現在のデータベース内で定義され、現在有効なロールが所有するあるドメインを使用する（テーブルもしくはビューの）全ての列を識別します。

表 34-10. column_domain_usage の列

名前	データ型	説明
domain_catalog	sql_identifier	ドメインを持つデータベースの名前です（常に現在のデータベースです）。
domain_schema	sql_identifier	ドメインを持つスキーマの名前です。
domain_name	sql_identifier	ドメインの名前です。
table_catalog	sql_identifier	テーブルを持つデータベースの名前です（常に現在のデータベースです）。
table_schema	sql_identifier	テーブルを持つスキーマの名前です。
table_name	sql_identifier	テーブルの名前です。
column_name	sql_identifier	列の名前です。

34.13. column_options

column_optionsビューは、現在のデータベースにある外部テーブルの列に定義された全てのオプションを含みます。現在のユーザが(所有者であるかまたは権限を持っていて)アクセスする権限を持つ外部テーブル列のみが表示されます。

表 34-11. column_optionsの列

名前	データ型	説明
table_catalog	sql_identifier	外部テーブルが含まれるデータベースの名前です。（常に現在のデータベースです）。
table_schema	sql_identifier	外部テーブルを含むスキーマの名前です。
table_name	sql_identifier	外部テーブルの名前です。
column_name	sql_identifier	列の名前です。
option_name	sql_identifier	オプションの名前です。
option_value	character_data	オプションの値です。

34.14. column_privileges

column_privilegesビューは、現在有効なロールに対し、または現在有効なロールによって、列に与えられた権限を全て識別します。列と許可を与えた者、許可を受けた者の組み合わせごとに1行があります。

権限がテーブル全体に付与されていた場合、このビューでは各列に権限が付与された場合と同じように表示されます。しかし、SELECT、INSERT、UPDATE、REFERENCESといった列単位で設定可能な種類の権限のみを対象範囲とします。

表 34-12. column_privilegesの列

名前	データ型	説明
grantor	sql_identifier	権限を与えたユーザの名前です。
grantee	sql_identifier	権限を与えられたロールの名前です。
table_catalog	sql_identifier	その列を含むテーブルを持つデータベースの名前です（常に現在のデータベースです）。
table_schema	sql_identifier	その列を含むテーブルを持つスキーマの名前です。
table_name	sql_identifier	その列を含むテーブルの名前です。
column_name	sql_identifier	列の名前です。
privilege_type	character_data	権限の種類です。 SELECT、INSERT、UPDATE、もしくはREFERENCESです。
is_grantable	yes_or_no	この権限を付与可能な場合はYES、さもなければNOです。

34.15. column_udt_usage

column_udt_usageビューは、現在有効なロールが所有するデータ型を使用する全ての列を識別します。 PostgreSQLでは、組み込みデータ型がユーザ定義型同様に振舞いますので、組み込みデータ型も同様にここに含まれます。 詳細は[項34.16](#)も参照してください。

表 34-13. column_udt_usageの列

名前	データ型	説明
udt_catalog	sql_identifier	列データ型（もし適用されていたら背後にあるドメインの型）を定義したデータベースの名前です（常に現在のデータベースです）。
udt_schema	sql_identifier	列データ型（もし適用されていたら背後にあるドメインの型）を定義したスキーマの名前です。
udt_name	sql_identifier	列データ型（もし適用されていたら背後にあるドメインの型）の名前です。
table_catalog	sql_identifier	そのテーブルを持つデータベースの名前です（常に現在のデータベースです）。
table_schema	sql_identifier	そのテーブルを持つスキーマの名前です。
table_name	sql_identifier	テーブルの名前です。
column_name	sql_identifier	列の名前です。

34.16. columns

columnsビューには、データベース内の全てのテーブル列（もしくはビューの列）についての情報が含まれます。システム列（oidなど）は含まれません。現在のユーザが（所有者である、権限を持っているなどの方法で）アクセスできる列のみが示されます。

表 34-14. columnsの列

名前	データ型	説明
table_catalog	sql_identifier	そのテーブルを持つデータベースの名前です（常に現在のデータベースです）。
table_schema	sql_identifier	そのテーブルを持つスキーマの名前です。
table_name	sql_identifier	テーブルの名前です。
column_name	sql_identifier	列の名前です。
ordinal_position	cardinal_number	テーブル内の列の位置（1から始まる序数）です。
column_default	character_data	列のデフォルト式です。
is_nullable	yes_or_no	列がNULLを持つことができる場合はYES、NULLを持つことができなければNOです。非NULL制約は、列にNULLを持たせないようにする方法の1つですが、その他にも存在します。
data_type	character_data	組み込み型の場合、列のデータ型、配列の場合、ARRAY（この場合はelement_typesビューを参照してください）、さもなくば、USER-DEFINEDです（この場合、型はudt_nameと関連する列で識別されます）。列がドメインに基づくものであれば、その列はドメインの背後にある型を示します（そして、ドメインはdomain_nameと関連する列によって識別されます）。
character_maximum_length	cardinal_number	data_typeが、文字、もしくはビット文字列型を表す場合、宣言された最大の長さです。その他の型や最大長が宣言されていない場合はNULLになります。
character_octet_length	cardinal_number	data_typeが文字型を示す場合、オクテット（バイト）単位の最大長です。他の全てのデータ型ではNULLです。最大オクテット長は宣言された文字最大長(上述)とサーバ符号化方式に依存します。
numeric_precision	cardinal_number	data_typeが数値型を示す場合、ここには、その列の型の（宣言された、もしくは暗黙的な）精度が含まれます。この精度は有意な桁数を示します。numeric_precision_radix列の指定に従い、10進数（10を底）、もしくは2進数（2を底）で表現されます。この列は、他の全ての型ではNULLです。
numeric_precision_radix	cardinal_number	data_typeが数値型を示す場合、この列は、numeric_precisionとnumeric_scaleの表現の底とする数を示します。2もしくは10を取ります。この列は、他の全ての型ではNULLです。
numeric_scale	cardinal_number	data_typeが高精度数値型を示す場合、ここには、その列の型の（宣言された、あるいは暗黙的な）位取りが含まれます。位取りとは、小数点より右側の有意な桁数です。numeric_precision_radix列の指定に従い、10進数（10を底）、もしくは2進数（2を底）で表現されます。この列は、他の全ての型ではNULLです。
datetime_precision	cardinal_number	data_typeが日付、時刻、タイムスタンプ、間隔型を示す場合、この列の型の秒以下の（宣言された、または暗黙的な）精度、つまり、秒値の小数点以降に保持する10進桁数、です。他のすべての型の場合ではこの列はNULLです。
interval_type	character_data	もしdata_typeがインターバル型を示す場合、この列はこの属性のインターバル値がどのフィールドを含むかの仕様を含みます。例えば、YEAR TO MONTH、DAY TO SECONDなどです。もしフィールド制約が指定されていない(インターバルが全てのフィールドを受け付ける)場合や、他の全てのデータ型の場合はこのフィールドはNULLです。
interval_precision	cardinal_number	PostgreSQLで利用できない機能に適用されるものです。(インターバル型の属性の秒未満の精度についてはdatetime_precisionを参照してください)
character_set_catalog	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
character_set_schema	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
character_set_name	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。

collation_catalog	sql_identifier	列の照合を含むデータベース(常に現在のデータベース)の名前で、デフォルトであるか属性のデータ型が照合可能でない場合はNULLです。
collation_schema	sql_identifier	属性の照合を含むスキーマの名前で、デフォルトであるか属性のデータ型が照合可能でない場合はNULLです。
collation_name	sql_identifier	列の照合の名前で、デフォルトであるか列のデータ型が照合可能でない場合はNULLです。
domain_catalog	sql_identifier	列がドメイン型の場合、そのドメインを定義したデータベースの名前です（常に現在のデータベースです）。さもなくば、NULLです。
domain_schema	sql_identifier	列がドメイン型の場合、そのドメインを定義したスキーマの名前です。さもなくば、NULLです。
domain_name	sql_identifier	列がドメイン型の場合、そのドメインの名前です。さもなくば、NULLです。
udt_catalog	sql_identifier	列データ型（もし適切ならば背後にあるドメインの型）を定義したデータベースの名前です（常に現在のデータベースです）。
udt_schema	sql_identifier	列データ型（もし適切ならば背後にあるドメインの型）を定義したスキーマの名前です。
udt_name	sql_identifier	列データ型（もし適切ならば背後にあるドメインの型）の名前です。
scope_catalog	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
scope_schema	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
scope_name	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
maximum_cardinality	cardinal_number	常にNULLです。 PostgreSQLでは配列の最大次数は常に無制限だからです。
dtd_identifier	sql_identifier	列のデータ型記述子の、テーブルに属するデータ型記述子内で一意な識別子です。この識別子のインスタンスを結合する時に、主に有用です（識別子の書式仕様は定義されておらず、今後のバージョンで同一性を維持する保証もありません）。
is_self_referencing	yes_or_no	PostgreSQLで利用できない機能に適用されるものです。
is_identity	yes_or_no	PostgreSQLで利用できない機能に適用されるものです。
identity_generation	character_data	PostgreSQLで利用できない機能に適用されるものです。
identity_start	character_data	PostgreSQLで利用できない機能に適用されるものです。
identity_increment	character_data	PostgreSQLで利用できない機能に適用されるものです。
identity_maximum	character_data	PostgreSQLで利用できない機能に適用されるものです。
identity_minimum	character_data	PostgreSQLで利用できない機能に適用されるものです。
identity_cycle	yes_or_no	PostgreSQLで利用できない機能に適用されるものです。
is_generated	character_data	PostgreSQLで利用できない機能に適用されるものです。
generation_expression	character_data	PostgreSQLで利用できない機能に適用されるものです。
is_updatable	yes_or_no	列が更新可能な場合YES、さもなくばNOです。（ベーステーブルの列は常に更新可能です。ビューの列では不要です。）

データ型は、複数の方法でSQLにより定義でき、さらにPostgreSQLにはデータ型定義に別の方法も追加されていますので、情報スキーマにおけるデータ型表現は多少難しくなっています。 `data_type` 列は、列の背後にある組み込み型を識別できるようになっていなければなりません。 PostgreSQLでは、型が `pg_catalog` システムカタログスキーマで定義されていることを意味します。 よく知られた組み込み型を特別に扱うことができるアプリケーション（例えば、数値型を異なる書式にする、精度列内のデータを使用する）の場合、この列が有用な場合があります。 `udt_name`、`udt_schema`、`udt_catalog` 列は、たとえドメインに基づいた列であっても、常に列の背後にあるデータ型を識別します（PostgreSQLは、組み込み型をユーザ定義型と同様に扱いますので、ここには組み込み型も現れます。これは標準SQLからの拡張です）。 アプリケーションが型に従って異なる処理を行う場合、これらの列を使用しなければなりません。 このような場合、本当はドメインに基づいている列なのかどうかは関係ないからです。 列がドメインに基づく場合、ドメインの識別子は `domain_name`、`domain_schema`、`domain_catalog` 列に保持されます。 関連するデータ型と列の組み合わせを作りたい場合や、ドメインを別の型として扱いたい場合は、`coalesce(domain_name, udt_name)` などとすることができます。

34.17. constraint_column_usage

constraint_column_usageビューは、現在のデータベースで制約を使用する全ての列を識別します。現在有効なロールが所有するテーブル内の列のみが表示されます。 検査制約では、このビューは検査式で使用する列を識別します。 外部キー制約では、このビューは外部キーを参照する列を識別します。一意性制約もしくはプライマリーキー制約では、このビューは制約される列を識別します。

表 34-15. constraint_column_usage 列

名前	データ型	説明
table_catalog	sql_identifier	ある制約で使用する列を持つデータベースの名前です（データベースの名前です）。
table_schema	sql_identifier	ある制約で使用する列を持つテーブルを含むスキーマの名前です。
table_name	sql_identifier	ある制約で使用する列を持つテーブルの名前です。
column_name	sql_identifier	ある制約で使用する列の名前です。
constraint_catalog	sql_identifier	制約を持つデータベースの名前です（常に現在のデータベースです）。
constraint_schema	sql_identifier	制約を持つスキーマの名前です。
constraint_name	sql_identifier	制約の名前です。

34.18. constraint_table_usage

constraint_table_usageビューは、ある制約で使用され、かつ、現在有効なロールが所有する、現在のデータベース内の全てのテーブルを識別します（これは、全てのテーブル制約とそれを定義したテーブルを識別するtable_constraintsとは異なります）。外部キー制約では、このビューは外部キーが参照するテーブルを識別します。一意性制約もしくはプライマリキー制約では、このビューは単に制約が属するテーブルを識別します。検査制約と非NULL制約はこのビューには含まれません。

表 34-16. constraint_table_usageの列

名前	データ型	説明
table_catalog	sql_identifier	ある制約で使用されるテーブルを持つデータベースの名前です（常に現在のデータベースです）。
table_schema	sql_identifier	ある制約で使用されるテーブルを持つスキーマの名前です。
table_name	sql_identifier	ある制約で使用されるテーブルの名前です。
constraint_catalog	sql_identifier	制約を持つデータベースの名前です（常に現在のデータベースです）。
constraint_schema	sql_identifier	制約を持つスキーマの名前です。
constraint_name	sql_identifier	制約の名前です。

34.19. data_type_privileges

data_type_privilegesビューは、記述子が示すオブジェクトの所有者である、何かしらの権限を持っているといった方法で現在のユーザがアクセスできる全てのデータ型記述子を識別します。あるデータ型がテーブル列やドメイン、関数（パラメータとして、あるいは戻り値として）の定義内で使用されると、そのデータ型記述子は生成され、そして、データ型がそのインスタンスでどのように使用されるか（例えば、もし適切ならば、宣言された最大長）についての情報が格納されます。各データ型記述子は、1つのオブジェクト（テーブル、ドメイン、関数）に割り当てられたデータ型記述子の中で一意となる任意の識別子が割り振られます。このビューはおそらくアプリケーションではあまり使用されませんが、情報スキーマ内の他のビューを定義する際に使用されます。

表 34-17. data_type_privilegesの列

名前	データ型	説明
object_catalog	sql_identifier	記述子が示すオブジェクトを持つデータベースの名前です（常に現在のデータベースです）。
object_schema	sql_identifier	記述子が示すオブジェクトを持つスキーマの名前です。
object_name	sql_identifier	記述子が示すオブジェクトの名前です。
object_type	character_data	記述子が示すオブジェクトの種類です。TABLE（データ型記述子はそのテーブルの列に属します）、DOMAIN（データ型記述子はそのドメインに属します）、ROUTINE（データ型記述子はその関数のパラメータあるいは戻り値データ型に属します）のいずれかです。
dtd_identifier	sql_identifier	データ型記述子の識別子です。これは同一オブジェクトに対するデータ型記述子の中で一意なものです。

34.20. domain_constraints

domain_constraintsビューには、現在有効なロールが所有するドメインに属する全ての制約があります。現在のユーザが（所有者である、権限を持っているなどの方法で）アクセスできるドメインのみが示されます。

表 34-18. domain_constraintsの列

名前	データ型	説明
constraint_catalog	sql_identifier	制約を持つデータベースの名前です（常に現在のデータベースです）。
constraint_schema	sql_identifier	制約を持つスキーマの名前です。
constraint_name	sql_identifier	制約の名前です。
domain_catalog	sql_identifier	ドメインを持つデータベースの名前です（常に現在のデータベースです）。
domain_schema	sql_identifier	ドメインを持つスキーマの名前です。
domain_name	sql_identifier	ドメインの名前です。
is_deferrable	yes_or_no	制約が遅延可能ならばYES。さもなければNO。
initially_deferred	yes_or_no	制約が遅延可能で初期状態が遅延であればYES。さもなければNO。

34.21. domain_udt_usage

domain_udt_usageビューは、現在有効なロールが所有するデータ型に基づいたすべてのドメインを識別します。 PostgreSQLでは組み込みデータ型はユーザ定義型と同様に振舞いますので、ここにも同様に現れることに注意してください。

表 34-19. domain_udt_usageの列

名前	データ型	説明
udt_catalog	sql_identifier	ドメインデータ型を定義したデータベースの名前です（常に現在のデータベースです）。
udt_schema	sql_identifier	ドメインデータ型を定義したスキーマの名前です。
udt_name	sql_identifier	ドメインデータ型の名前です。
domain_catalog	sql_identifier	ドメインを持つデータベースの名前です（常に現在のデータベースです）。
domain_schema	sql_identifier	ドメインを持つスキーマの名前です。
domain_name	sql_identifier	ドメインの名前です。

34.22. domains

domainsビューには、現在のデータベースで定義された全てのドメインが含まれます。現在のユーザが（所有者である、権限を持っているなどの方法で）アクセスできるドメインのみが示されます。

表 34-20. domainsの列

名前	データ型	説明
domain_catalog	sql_identifier	ドメインを持つデータベースの名前です（常に現在のデータベースです）。
domain_schema	sql_identifier	ドメインを持つスキーマの名前です。
domain_name	sql_identifier	ドメインの名前です。
data_type	character_data	組み込み型の場合は、ドメインのデータ型、何らかの配列の場合はARRAYです（後者の場合はelement_typesビューを参照してください）。さもなければ、USER-DEFINEDです（この場合、その型はudt_nameと関連する列で識別されます）。
character_maximum_length	cardinal_number	ドメインが、文字もしくはビット文字列型の場合、宣言された最大長です。他のデータ型、あるいは最大長の宣言がない場合はNULLです。
character_octet_length	cardinal_number	ドメインが文字型の場合、1つのデータの可能最大長をオクテット（バイト）で示します。他のデータ型の場合はNULLです。最大オクテット長は宣言された文字最大長(上述)とサーバ符号化方式に依存します。
character_set_catalog	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
character_set_schema	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
character_set_name	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
collation_catalog	sql_identifier	ドメインの照合を含むデータベース(常に現在のデータベース)の名前で、デフォルトであるかドメインのデータ型が照合可能でない場合はNULLです。
collation_schema	sql_identifier	ドメインの照合を含むスキーマの名前で、デフォルトであるかドメインのデータ型が照合可能でない場合はNULLです。
collation_name	sql_identifier	ドメインの照合の名前で、デフォルトであるかドメインのデータ型が照合可能でない場合はNULLです。
numeric_precision	cardinal_number	ドメインが数値型の場合、この列は、そのドメインの型の（宣言された、もしくは暗黙的な）精度を持ちます。この精度は有効桁数を示すものです。numeric_precision_radix列が示す通り、10進数（10を底）でも2進数（2を底）でも表現することができます。他の全てのデータ型では、この列はNULLです。
numeric_precision_radix	cardinal_number	ドメインが数値型の場合、この列は、numeric_precisionとnumeric_scaleで表現されるその列の値の底数がどちらかを示します。2もしくは10の値となります。他の全てのデータ型では、この列はNULLです。
numeric_scale	cardinal_number	ドメインが高精度数値型の場合、この列は、そのドメインの型の（宣言された、もしくは暗黙的な）位取りを持ちます。位取りは、小数点より右側の有効桁数を示すものです。numeric_precision_radix列の指定に従い、10進数（10を底）、もしくは2進数（2を底）で表現されます。他の全ての型ではこの列はNULLです。
datetime_precision	cardinal_number	data_typeが日付、時刻、タイムスタンプ、間隔型を示す場合、この列はこのドメインの型で（宣言された、または暗黙的な）秒の端数の精度、つまり、秒値の小数点以下で保持される10進の桁数です。他の全ての型の場合はNULLです。
interval_type	character_data	もしdata_typeがインターバル型を示す場合、この列はこのドメインのインターバル値がどのフィールドを含むかの仕様を含みます。例えば、YEAR TO MONTH、DAY TO SECONDなどです。もしフィールド制約が指定されていない(インターバルが全てのフィールドを受け付ける)場合や、他の全てのデータ型の場合はこのフィールドはNULLです。
interval_precision	cardinal_number	PostgreSQLで利用できない機能に適用されるものです。（インターバル型のドメインの秒未満の精度についてはdatetime_precisionを参照してください）
domain_default	character_data	ドメインのデフォルト式です。

udt_catalog	sql_identifier	ドメインデータ型を定義したデータベースの名前です（常に現在のデータベースです）。
udt_schema	sql_identifier	ドメインデータ型を定義したスキーマの名前です。
udt_name	sql_identifier	ドメインデータ型の名前です。
scope_catalog	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
scope_schema	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
scope_name	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
maximum_cardinality	cardinal_number	常にNULLです。 PostgreSQLでは配列の最大次数は常に無制限だからです。
dtd_identifier	sql_identifier	そのドメインに属するデータ型記述子間で一意な、データ型記述子の識別子です（ドメインには1つのデータ型記述子しかありませんのでこれは些細なものです）。これは主に、こうした識別子の他のインスタンスを結合する時に有用です（識別子の書式の仕様は定義されておらず、将来のバージョンでそのまま維持されるかどうかは保証されません）。

34.23. element_types

element_typesには、配列の要素のデータ型記述子が含まれます。テーブル列、複合データ型属性、ドメイン、関数パラメータ、関数の戻り値が配列型であると宣言された場合、情報スキーマの各ビューでは、data_type列にARRAYだけが含まれます。配列の要素の型についての情報を取り出すには、各ビューとこのビューを結合することで可能です。例えば、テーブルの列のデータ型と、もし適切ならば、配列の要素型を表示するには、以下のように行います。

```
SELECT c.column_name, c.data_type, e.data_type AS element_type
FROM information_schema.columns c LEFT JOIN information_schema.element_types e
    ON ((c.table_catalog, c.table_schema, c.table_name, 'TABLE', c.dtd_identifier)
        = (e.object_catalog, e.object_schema, e.object_name, e.object_type, e.collection_type_identifier))
WHERE c.table_schema = '...' AND c.table_name = '...'
ORDER BY c.ordinal_position;
```

このビューは、所有者である、適切な権限を持っているといった方法で、現在のユーザがアクセスできるオブジェクトのみが含まれます。

表 34-21. element_typesの列

名前	データ型	説明
object_catalog	sql_identifier	記述される配列を使用するオブジェクトを持つデータベースの名前です（常に現在のデータベースです）。
object_schema	sql_identifier	記述される配列を使用するオブジェクトを持つスキーマの名前です。
object_name	sql_identifier	記述される配列を使用するオブジェクトの名前です。
object_type	character_data	記述される配列を使用するオブジェクトの種類です。TABLE（その配列がテーブルの列によって使用される）、USER-DEFINED TYPE（その配列が複合データ型の属性によって使用される）、DOMAIN（その配列がドメインによって使用される）、ROUTINE（その配列が関数のパラメータ、もしくは戻り値の型によって使用される）のいずれかです。
collection_type_identifier	sql_identifier	記述される配列のデータ型記述子の識別子です。他の情報スキーマビューのdtd_identifier列と結合するのに使用してください。
data_type	character_data	組み込み型の場合は配列要素のデータ型です。さもなくば、USER-DEFINEDです（この場合、型はudt_nameと関連する列で識別されます）。
character_maximum_length	cardinal_number	常にNULLです。この情報は、PostgreSQLにおける配列要素のデータ型には当てはまらないからです。
character_octet_length	cardinal_number	常にNULLです。この情報は、PostgreSQLにおける配列要素のデータ型には当てはまらないからです。
character_set_catalog	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
character_set_schema	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
character_set_name	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
collation_catalog	sql_identifier	要素データ型の照合を含むデータベース(常に現在のデータベース)の名前で、デフォルトであるか要素のデータ型が照合可能でない場合はNULLです。
collation_schema	sql_identifier	要素データ型の照合を含むスキーマの名前で、デフォルトであるか要素のデータ型が照合可能でない場合はNULLです。
collation_name	sql_identifier	要素データ型の照合の名前で、デフォルトであるか要素のデータ型が照合可能でない場合はNULLです。
numeric_precision	cardinal_number	常にNULLです。この情報は、PostgreSQLにおける配列要素のデータ型には当てはまらないからです。
numeric_precision_radix	cardinal_number	常にNULLです。この情報は、PostgreSQLにおける配列要素のデータ型には当てはまらないからです。
numeric_scale	cardinal_number	常にNULLです。この情報は、PostgreSQLにおける配列要素のデータ型には当てはまらないからです。
datetime_precision	cardinal_number	常にNULLです。この情報は、PostgreSQLにおける配列要素のデータ型には当てはまらないからです。
interval type	character_data	常にNULLです。この情報は、PostgreSQLにおける配列要素のデータ型には

		当てはまらないからです。
interval_precision	cardinal_number	常にNULLです。 この情報は、PostgreSQLにおける配列要素のデータ型には当てはまらないからです。
domain_default	character_data	未実装です。
udt_catalog	sql_identifier	要素のデータ型を定義したデータベースの名前です （常に現在のデータベースです）。
udt_schema	sql_identifier	要素のデータ型を定義したスキーマの名前です。
udt_name	sql_identifier	要素のデータ型の名前です。
scope_catalog	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
scope_schema	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
scope_name	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
maximum_cardinality	cardinal_number	常にNULLです。 PostgreSQLでは配列の最大次数は常に無制限だからです。
dtd_identifier	sql_identifier	要素のデータ型記述子の識別子です。 現在は用途はありません。

[前のページ](#)
domains

[ホーム](#)
[上に戻る](#)

[次のページ](#)
enabled_roles

34.24. enabled_roles

enabled_rolesビューは、現在"有効なロール"を識別します。 有効なロールは、現在のユーザと自動継承によって有効なロールに付与されたすべてのロールとして再帰的に定義されます。 言い換えると、これは、現在のユーザが直接または間接的に属するメンバ資格の継承により自動的に持つすべてのロールです。

権限検査では、"適用可能なロール"群が適用されます。 これは、有効なロールよりも広範になる可能性があります。 したがって一般的にはこのビューよりもapplicable_rolesビューを使用する方が良いでしょう。 このビューも参照してください。

表 34-22. enabled_rolesの列

名前	データ型	説明
role_name	sql_identifier	ロールの名前です。

34.25. foreign_data_wrapper_options

foreign_data_wrapper_optionsビューには現在のデータベース内の外部データラッパー用に定義されたすべてのオプションが含まれます。 > 現在のユーザが（所有者または何らかの権限を持つことで）アクセス可能な外部データラッパーのみが表示されます。

表 34-23. foreign_data_wrapper_options Columns

名前	データ型	説明
foreign_data_wrapper_catalog	sql_identifier	外部データラッパーを定義したデータベースの名前です。（常に現在のデータベースです。）
foreign_data_wrapper_name	sql_identifier	外部データラッパーの名前です。
option_name	sql_identifier	オプションの名前です。
option_value	character_data	オプションの値です。

34.26. foreign_data_wrappers

foreign_data_wrappersビューには現在のデータベース内で定義された、すべての外部データラッパーが含まれます。現在のユーザが（所有者である、または、何らかの権限を持つことにより）アクセス可能な外部データラッパーのみが表示されます。

表 34-24. foreign_data_wrappersの列

名前	データ型	説明
foreign_data_wrapper_catalog	sql_identifier	外部データラッパーを含むデータベース名です。（常に現在のデータベースです。）
foreign_data_wrapper_name	sql_identifier	外部データラッパーの名前です。
authorization_identifier	sql_identifier	外部サーバの所有者の名前です。
library_name	character_data	この外部データラッパーを実装するライブラリの名前です。
foreign_data_wrapper_language	character_data	この外部データラッパーを実装するのに使用される言語です。

34.27. foreign_server_options

foreign_server_optionsビューには、現在のデータベース内の外部サーバ用に定義されたすべてのオプションが含まれます。現在のユーザが（所有者である、または何らかの権限を持つことにより）アクセス可能な外部サーバのみが表示されます。

表 34-25. foreign_server_optionsの列

名前	データ型	説明
foreign_server_catalog	sql_identifier	外部サーバを定義したデータベースの名前です。（常に現在のデータベースです。）
foreign_server_name	sql_identifier	外部サーバの名前
option_name	sql_identifier	オプションの名前
option_value	character_data	オプションの値

34.28. foreign_servers

foreign_serversビューには、現在のデータベース内で定義されたすべての外部サーバが含まれます。現在のユーザが（所有者である、または何らかの権限を持つことにより）アクセス可能な外部サーバのみが表示されます。

表 34-26. foreign_serversの列

名前	データ型	説明
foreign_server_catalog	sql_identifier	外部サーバを定義したデータベースの名前です。（常に現在のデータベースです。）
foreign_server_name	sql_identifier	外部サーバの名前です。
foreign_data_wrapper_catalog	sql_identifier	外部サーバで使用する外部データラッパーを含むデータベースの名前です。（常に現在のデータベースです。）
foreign_data_wrapper_name	sql_identifier	外部サーバにより使用される外部データラッパーの名前です。
foreign_server_type	character_data	作成時点で指定されている場合、その外部サーバの種類情報です。
foreign_server_version	character_data	作成時点で指定されている場合、その外部サーバのバージョン情報です。
authorization_identifier	sql_identifier	外部サーバの所有者名です。

34.29. foreign_table_options

foreign_table_optionsビューは、現在のデータベースの外部テーブルに定義された全てのオプションを含みます。(所有者であるか何らかの権限を持っていて)現在のユーザがアクセスできる外部テーブルだけが表示されます。

表 34-27. foreign_table_optionsの列

名前	データ型	説明
foreign_table_catalog	sql_identifier	外部テーブルを含むデータベースの名前です（常に現在のデータベースです）。
foreign_table_schema	sql_identifier	外部テーブルを含むスキーマの名前です。
foreign_table_name	sql_identifier	外部テーブルの名前です。
foreign_server_catalog	sql_identifier	外部サーバが定義されているデータベースの名前です（常に現在のデータベースです）。
foreign_server_name	sql_identifier	外部サーバの名前です。
option_name	sql_identifier	オプションの名前です。
option_value	character_data	オプションの値です。

34.30. foreign_tables

foreign_tablesビューは、現在のデータベースで定義されている全ての外部テーブルを含みます。(所有者であるか何らかの権限を持っていて)現在のユーザがアクセスできる外部テーブルだけが表示されます。

表 34-28. foreign_tablesの列

名前	データ型	説明
foreign_table_catalog	sql_identifier	外部テーブルを含むデータベースの名前です（常に現在のデータベースです）。
foreign_table_schema	sql_identifier	外部テーブルを含むスキーマの名前です。
foreign_table_name	sql_identifier	外部テーブルの名前です。
foreign_server_catalog	sql_identifier	外部サーバが定義されているデータベースの名前です（常に現在のデータベースです）。
foreign_server_name	sql_identifier	外部サーバの名前です。

34.31. key_column_usage

key_column_usageビューは、現在のデータベースにおいて、ある一意性制約、プライマリー制約、外部キー制約によって制限を受けている全ての列を識別します。 検査制約はこのビューには含まれません。 現在のユーザが所有者である、または何らかの権限を持ち、アクセスできるテーブル内のこうした列のみがここに示されます。

表 34-29. key_column_usageの列

名前	データ型	説明
constraint_catalog	sql_identifier	制約を持つデータベースの名前です。（常に現在のデータベースです。）
constraint_schema	sql_identifier	制約を持つスキーマの名前です。
constraint_name	sql_identifier	制約の名前です。
table_catalog	sql_identifier	この制約によって制限を受ける列を持つテーブルを持つデータベースの名前です。（常に現在のデータベースです。）
table_schema	sql_identifier	この制約によって制限を受ける列を持つテーブルを持つスキーマの名前です。
table_name	sql_identifier	この制約によって制限を受ける列を持つテーブルの名前です。
column_name	sql_identifier	この制約によって制限を受ける列の名前です。
ordinal_position	cardinal_number	制約キー内の列の位置を（1から始まる）序数で表したものです。
position_in_unique_constraint	cardinal_number	外部キー制約では、一意性制約内部の被参照列の位置の序数（1から始まります）です。 その他の場合はNULLです。

34.32. parameters

parametersビューには、現在のデータベースにある全ての関数のパラメータ（引数）についての情報があります。現在のユーザが（所有している、あるいはある権限を持っているといった方法で）アクセスできる関数についてののみが示されます。

表 34-30. parametersの列

名前	データ型	説明
specific_catalog	sql_identifier	関数を持つデータベースの名前です。（常に現在のデータベースです。）
specific_schema	sql_identifier	関数を持つスキーマの名前です。
specific_name	sql_identifier	関数の"仕様名称"です。詳細は 項34.40 を参照してください。
ordinal_position	cardinal_number	関数の引数リストにおけるパラメータの位置の序数（1から始まる）です。
parameter_mode	character_data	入力パラメータではIN、出力パラメータではOUT、入出力パラメータではINOUTです。
is_result	yes_or_no	PostgreSQLで利用できない機能に適用されるものです。
as_locator	yes_or_no	PostgreSQLで利用できない機能に適用されるものです。
parameter_name	sql_identifier	名前付きパラメータです。無名のパラメータの場合はNULLです。
data_type	character_data	組み込み型の場合、パラメータのデータ型です。何らかの配列の場合はARRAYです（この場合、 <code>element_types</code> ビューを参照してください）。さもなくば、USER-DEFINEDです（この場合、型は <code>udt_name</code> と関連する列で識別されます）。
character_maximum_length	cardinal_number	常にNULLです。PostgreSQLでは、この情報はパラメータデータ型に適用されないからです。
character_octet_length	cardinal_number	常にNULLです。PostgreSQLでは、この情報はパラメータデータ型に適用されないからです。
character_set_catalog	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
character_set_schema	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
character_set_name	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
collation_catalog	sql_identifier	常にNULLです。PostgreSQLでは、この情報はパラメータデータ型に適用されないからです。
collation_schema	sql_identifier	常にNULLです。PostgreSQLでは、この情報はパラメータデータ型に適用されないからです。
collation_name	sql_identifier	常にNULLです。PostgreSQLでは、この情報はパラメータデータ型に適用されないからです。
numeric_precision	cardinal_number	常にNULLです。PostgreSQLでは、この情報はパラメータデータ型に適用されないからです。
numeric_precision_radix	cardinal_number	常にNULLです。PostgreSQLでは、この情報はパラメータデータ型に適用されないからです。
numeric_scale	cardinal_number	常にNULLです。PostgreSQLでは、この情報はパラメータデータ型に適用されないからです。
datetime_precision	cardinal_number	常にNULLです。PostgreSQLでは、この情報はパラメータデータ型に適用されないからです。
interval_type	character_data	常にNULLです。PostgreSQLでは、この情報はパラメータデータ型に適用されないからです。
interval_precision	cardinal_number	常にNULLです。PostgreSQLでは、この情報はパラメータデータ型に適用されないからです。
udt_catalog	sql_identifier	パラメータのデータ型を定義したデータベースの名前です（常に現在のデータベースです）。
udt_schema	sql_identifier	パラメータのデータ型を定義したスキーマの名前です。
udt_name	sql_identifier	パラメータのデータ型の名前です。

scope_catalog	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
scope_schema	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
scope_name	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
maximum_cardinality	cardinal_number	常にNULLです。 PostgreSQLでは配列の最大次数は常に無制限だからです。
dtd_identifier	sql_identifier	関数に属するデータ型記述子内で一意なパラメータのデータ型記述子の識別子です。 これは主に、こうした識別子の他のインスタンスと結合する時に有用です（識別子の書式の仕様は定義されておらず、また、今後のバージョンでも同一のままであるという保証也没有せん）。

[前のページ](#)
key_column_usage

[ホーム](#)
[上に戻る](#)

[次のページ](#)
referential_constraints

34.33. referential_constraints

referential_constraintsビューには、現在のデータベース内にある全ての参照（外部キー）制約があります。現在のユーザが、参照テーブルに（所有者、またはSELECT以外の何らかの権限を持つという方法で）書き込みアクセスを持つと言う事でこれらの制約は示されます。

表 34-31. referential_constraintsの列

名前	データ型	説明
constraint_catalog	sql_identifier	制約を持つデータベースの名前です（常に現在のデータベースです）。
constraint_schema	sql_identifier	制約を持つスキーマの名前です。
constraint_name	sql_identifier	制約の名前です。
unique_constraint_catalog	sql_identifier	外部キー制約が参照する一意性制約、もしくはプライマリキー制約を持つデータベースの名前です（常に現在のデータベースです）。
unique_constraint_schema	sql_identifier	外部キー制約が参照する一意性制約、もしくはプライマリキー制約を持つスキーマの名前です。
unique_constraint_name	sql_identifier	外部キー制約が参照する一意性制約、もしくはプライマリキー制約の名前です。
match_option	character_data	外部キー制約の一致オプションです。 FULL、PARTIAL、NONEのいずれかです。
update_rule	character_data	外部キー制約の更新規則です。 CASCADE、SET NULL、SET DEFAULT、RESTRICT、NO ACTIONのいずれかです。
delete_rule	character_data	外部キー制約の削除規則です。 CASCADE、SET NULL、SET DEFAULT、RESTRICT、NO ACTIONのいずれかです。

34.34. role_column_grants

role_column_grantsビューは、譲与者または被譲与者が現在有効なロールである場合、列に付与された全ての権限を識別します。 詳細な情報はcolumn_privilegesの中にあります。 このビューとcolumn_privilegesとの間の実質的な違いは、このビューでは現在のユーザがPUBLICに与えられた権限によりアクセスできるようになった列を省略していることです。

表 34-32. role_column_grantsの列

名前	データ型	説明
grantor	sql_identifier	権限を与えたロールの名前です。
grantee	sql_identifier	権限が与えられたロールの名前です。
table_catalog	sql_identifier	列を含むテーブルを持つデータベースの名前です （常に現在のデータベースです）。
table_schema	sql_identifier	列を含むテーブルを持つスキーマの名前です。
table_name	sql_identifier	列を含むテーブルの名前です。
column_name	sql_identifier	列の名前です。
privilege_type	character_data	権限の種類です。 SELECT、INSERT、UPDATE、REFERENCESのいずれかです。
is_grantable	yes_or_no	権限を付与可能であればYESです。さもなければNOです。

34.35. role_routine_grants

role_routine_grantsビューは、現在有効なロールが譲与者、または被譲与者で関数上に与えられた全ての権限を識別します。詳細な情報はroutine_privilegesの中にあります。 このビューとroutine_privilegesとの間の実質的な違いは、このビューでは現在のユーザがPUBLICに与えられた権限によりアクセスできるようになった関数を省略していることです。

表 34-33. role_routine_grantsの列

名前	データ型	説明
grantor	sql_identifier	権限を与えたロールの名前です。
grantee	sql_identifier	権限が与えられたロールの名前です。
specific_catalog	sql_identifier	関数を持つデータベースの名前です（常に現在のデータベースです）。
specific_schema	sql_identifier	関数を持つスキーマの名前です。
specific_name	sql_identifier	関数の"仕様名称"です。 詳細は 項34.40 を参照してください。
routine_catalog	sql_identifier	関数を持つデータベースの名前です（常に現在のデータベースです）。
routine_schema	sql_identifier	関数を持つスキーマの名前です。
routine_name	sql_identifier	関数の名前です（オーバーロードされている場合は重複する可能性があります）。
privilege_type	character_data	常にEXECUTE です（関数用の唯一の権限です）。
is_grantable	yes_or_no	権限が付与可能な場合YESです。さもなければNOです。

34.36. role_table_grants

role_table_grantsビューは、現在有効なロールが譲与者または被譲与者であるテーブルやビュー上に与えられた全ての権限を識別します。詳細な情報はtable_privilegesの中にあります。このビューとtable_privilegesとの間の実質的な違いは、このビューでは現在のユーザがPUBLICに与えられた権限によりアクセスできるようになったテーブルを省略していることです。

表 34-34. role_table_grantsの列

名前	データ型	説明
grantor	sql_identifier	権限を与えたロールの名前です。
grantee	sql_identifier	権限が与えられたロールの名前です。
table_catalog	sql_identifier	テーブルを持つデータベースの名前です（常に現在のデータベースです）。
table_schema	sql_identifier	テーブルを持つスキーマの名前です。
table_name	sql_identifier	テーブルの名前です。
privilege_type	character_data	権限の種類は、SELECT、INSERT、UPDATE、DELETE、TRUNCATE、REFERENCES、またはTRIGGERのいずれかです。
is_grantable	yes_or_no	権限が付与可能な場合YESです。さもなければNOです。
with_hierarchy	yes_or_no	SQL標準では、WITH HIERARCHY OPTIONは、継承テーブル階層に対するある操作を許可する独立した（副次）権限です。PostgreSQLでは、これはSELECT権限に含まれているため、この列は権限がSELECTの場合はYES、それ以外の場合はNOです。

34.37. role_udt_grants

role_udt_grantsビューは、現在有効なロールが付与者または被付与者である、ユーザ定義型に付与されたUSAGE権限を識別することを意図したものです。それ以上の情報はudt_privilegesで見つかります。このビューとudt_privilegesビューとの間の実質的な違いは、このビューでは現在のユーザがPUBLICに与えられた権限によりアクセスできるようになったオブジェクトを省略していることだけです。PostgreSQLではデータ型は実際の権限を持たず、PUBLICに対する暗黙の権限付与しか持たないため、このビューは空です。

表 34-35. role_udt_grants Columns

Name	Data Type	Description
grantor	sql_identifier	The name of the role that granted the privilege
grantee	sql_identifier	The name of the role that the privilege was granted to
udt_catalog	sql_identifier	Name of the database containing the type (always the current database)
udt_schema	sql_identifier	Name of the schema containing the type
udt_name	sql_identifier	Name of the type
privilege_type	character_data	Always TYPE USAGE
is_grantable	yes_or_no	YES if the privilege is grantable, NO if not

34.38. role_usage_grants

role_usage_grantsビューは、譲与者または被譲与者が現在有効なロールである多くの種類のオブジェクトに対し、USAGE権限を識別します。 詳細な情報はusage_privilegesの中にあります。 このビューとusage_privilegesビューとの間の実質的な違いは、このビューでは現在のユーザがPUBLICに与えられた権限によりアクセスできるようになったオブジェクトを省略していることだけです。

表 34-36. role_usage_grantsの列

名前	データ型	説明
grantor	sql_identifier	権限を与えたロールの名前
grantee	sql_identifier	権限が与えられたロールの名前
object_catalog	sql_identifier	オブジェクトを持つデータベースの名前（常に現在のデータベースです）。
object_schema	sql_identifier	適用されればオブジェクトを持つスキーマの名前で、そうでなければ空文字列
object_name	sql_identifier	オブジェクトの名前です。
object_type	character_data	COLLATIONまたはDOMAINまたはFOREIGN DATA WRAPPERまたはFOREIGN SERVERまたはSEQUENCE
privilege_type	character_data	常にUSAGEです。
is_grantable	yes_or_no	権限が付与可能であればYESです。さもなければNOです。

34.39. routine_privileges

routine_privilegesビューは、現在有効なロールに与えられた権限、あるいは現在有効なロールによって関数に与えられた権限を全て識別します。 関数、権限の譲与者と被譲与者の組み合わせごとに1行あります。

表 34-37. routine_privilegesの列

名前	データ型	説明
grantor	sql_identifier	権限を与えたロールの名前です。
grantee	sql_identifier	権限が与えられたロールの名前です。
specific_catalog	sql_identifier	関数を持つデータベースの名前です（常に現在のデータベースです）。
specific_schema	sql_identifier	関数を持つスキーマの名前です。
specific_name	sql_identifier	関数の"仕様名称"です。 詳細は 項34.40 を参照してください。
routine_catalog	sql_identifier	関数を持つデータベースの名前です（常に現在のデータベースです）。
routine_schema	sql_identifier	関数を持つスキーマの名前です。
routine_name	sql_identifier	関数の名前です（オーバーロードされている場合は重複する場合があります）。
privilege_type	character_data	常にEXECUTE です（関数用の唯一の権限です）。
is_grantable	yes_or_no	権限が付与可能な場合YESです。さもなければNOです。

34.40. routines

`routines`ビューには現在のデータベース内の全ての関数があります。表示される関数は、現在のユーザが（所有者である、何らかの権限を持っているといった方法で）アクセスできるものだけです。

表 34-38. `routines`の列

名前	データ型	説明
<code>specific_catalog</code>	<code>sql_identifier</code>	関数を持つデータベースの名前です（常に現在のデータベースです）。
<code>specific_schema</code>	<code>sql_identifier</code>	関数を持つスキーマの名前です。
<code>specific_name</code>	<code>sql_identifier</code>	関数の"仕様名称"です。これは、その関数の実際の名前がオーバーロードされていたとしても、スキーマ内の関数を一意に識別する名前です。仕様名称の書式は定義されておらず、特定の関数名の他のインスタンスと比較するためにのみ使用されます。
<code>routine_catalog</code>	<code>sql_identifier</code>	関数を持つデータベースの名前です（常に現在のデータベースです）。
<code>routine_schema</code>	<code>sql_identifier</code>	関数を持つスキーマの名前です。
<code>routine_name</code>	<code>sql_identifier</code>	関数の名前です（オーバーロードされている場合重複することがあります）。
<code>routine_type</code>	<code>character_data</code>	常に <code>FUNCTION</code> です（将来他の関数種類が含まれる可能性があります）。
<code>module_catalog</code>	<code>sql_identifier</code>	PostgreSQLで利用できない機能に適用されるものです。
<code>module_schema</code>	<code>sql_identifier</code>	PostgreSQLで利用できない機能に適用されるものです。
<code>module_name</code>	<code>sql_identifier</code>	PostgreSQLで利用できない機能に適用されるものです。
<code>udt_catalog</code>	<code>sql_identifier</code>	PostgreSQLで利用できない機能に適用されるものです。
<code>udt_schema</code>	<code>sql_identifier</code>	PostgreSQLで利用できない機能に適用されるものです。
<code>udt_name</code>	<code>sql_identifier</code>	PostgreSQLで利用できない機能に適用されるものです。
<code>data_type</code>	<code>character_data</code>	関数の戻り値が組み込み型の場合、そのデータ型です。何らかの配列の場合は <code>ARRAY</code> です（この場合は、 <code>element_types</code> ビューを参照してください）。さもなくば、 <code>USER-DEFINED</code> です（この場合、その型は <code>type_udt_name</code> と関連する列によって識別されます）。
<code>character_maximum_length</code>	<code>cardinal_number</code>	常にNULLです。PostgreSQLではこの情報は戻り値のデータ型に当てはまりません。
<code>character_octet_length</code>	<code>cardinal_number</code>	常にNULLです。PostgreSQLではこの情報は戻り値のデータ型に当てはまりません。
<code>character_set_catalog</code>	<code>sql_identifier</code>	PostgreSQLで利用できない機能に適用されるものです。
<code>character_set_schema</code>	<code>sql_identifier</code>	PostgreSQLで利用できない機能に適用されるものです。
<code>character_set_name</code>	<code>sql_identifier</code>	PostgreSQLで利用できない機能に適用されるものです。
<code>collation_catalog</code>	<code>sql_identifier</code>	常にNULLです。PostgreSQLではこの情報は戻り値のデータ型に当てはまりません。
<code>collation_schema</code>	<code>sql_identifier</code>	常にNULLです。PostgreSQLではこの情報は戻り値のデータ型に当てはまりません。
<code>collation_name</code>	<code>sql_identifier</code>	常にNULLです。PostgreSQLではこの情報は戻り値のデータ型に当てはまりません。
<code>numeric_precision</code>	<code>cardinal_number</code>	常にNULLです。PostgreSQLではこの情報は戻り値のデータ型に当てはまりません。
<code>numeric_precision_radix</code>	<code>cardinal_number</code>	常にNULLです。PostgreSQLではこの情報は戻り値のデータ型に当てはまりません。
<code>numeric_scale</code>	<code>cardinal_number</code>	常にNULLです。PostgreSQLではこの情報は戻り値のデータ型に当てはまりません。

datetime_precision	cardinal_number	常にNULLです。PostgreSQLではこの情報は戻り値のデータ型に当てはまりません。
interval_type	character_data	常にNULLです。PostgreSQLではこの情報は戻り値のデータ型に当てはまりません。
interval_precision	cardinal_number	常にNULLです。PostgreSQLではこの情報は戻り値のデータ型に当てはまりません。
type_udt_catalog	sql_identifier	関数の戻り値のデータ型が定義されたデータベースの名前です（常に現在のデータベースです）。
type_udt_schema	sql_identifier	関数の戻り値のデータ型が定義されたスキーマの名前です。
type_udt_name	sql_identifier	関数の戻り値のデータ型の名前です。
scope_catalog	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
scope_schema	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
scope_name	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
maximum_cardinality	cardinal_number	常にNULLです。PostgreSQLでは配列の最大次数は常に無制限だからです。
dtd_identifier	sql_identifier	この関数に属するデータ型記述子中で一意な、関数の戻り値のデータ型のデータ型記述子の識別子です。これは主に、そうした識別子の他のインスタンスと結合する際に有用です。（識別子の書式仕様は定義されておらず、将来のバージョンで同じままであるという保証ありません。）
routine_body	character_data	関数がSQL関数ならばSQL、さもなければEXTERNALです。
routine_definition	character_data	関数のソーステキストです（現在有効なロールがその関数の所有者でなければNULLです）。（標準SQLに従うと、routine_bodyがSQLの場合にのみ適用されます。しかし、PostgreSQLでは、関数が作成された時に指定されたソーステキストが常に含まれます。）
external_name	character_data	関数がC関数の場合関数の外部名（リンクシンボル）、さもなければNULLです。（これはroutine_definitionで示した値と同じになるように動作します。）
external_language	character_data	その関数を作成した言語です。
parameter_style	character_data	常にGENERALです（標準SQLでは他のパラメータ様式も定義していますが、これらはPostgreSQLでは使用できません）。
is_deterministic	yes_or_no	関数が不変である（標準SQLでは決定性があると呼びます）と宣言されている場合YES、さもなければNOです。（情報スキーマ経由ではPostgreSQLで利用できる他の変動レベルを問い合わせることはできません。）
sql_data_access	character_data	常に、関数がSQLデータを変更することができることを意味するMODIFIESです。この情報はPostgreSQLでは有用ではありません。
is_null_call	yes_or_no	その関数の引数のいずれかがNULLの場合に、自動的にNULLを返す場合はYES、さもなければNOです。
sql_path	character_data	PostgreSQLで利用できない機能に適用されるものです。
schema_level_routine	yes_or_no	常にYESです。（この反対はユーザ定義の種類による方法となります。これはPostgreSQLでは使用できない機能です。）
max_dynamic_result_sets	cardinal_number	PostgreSQLで利用できない機能に適用されるものです。
is_user_defined_cast	yes_or_no	PostgreSQLで利用できない機能に適用されるものです。
is_implicitly_invocable	yes_or_no	PostgreSQLで利用できない機能に適用されるものです。
security_type	character_data	現在のユーザ権限で関数が動作する場合INVOKER、定義したユーザの権限で関数が動作する場合DEFINERです。
to_sql_specific_catalog	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
to_sql_specific_schema	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
to_sql_specific_name	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
as_locator	yes_or_no	PostgreSQLで利用できない機能に適用されるものです。
created	time stamp	

		PostgreSQLで利用できない機能に適用されるものです。
last_altered	time_stamp	PostgreSQLで利用できない機能に適用されるものです。
new_savepoint_level	yes_or_no	PostgreSQLで利用できない機能に適用されるものです。
is_udt_dependent	yes_or_no	現在は常にNOです。もう一方のYESはPostgreSQLで利用できない機能に適用されるものです。
result_cast_from_data_type	character_data	PostgreSQLで利用できない機能に適用されるものです。
result_cast_as_locator	yes_or_no	PostgreSQLで利用できない機能に適用されるものです。
result_cast_char_max_length	cardinal_number	PostgreSQLで利用できない機能に適用されるものです。
result_cast_char_octet_length	character_data	PostgreSQLで利用できない機能に適用されるものです。
result_cast_char_set_catalog	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
result_cast_char_set_schema	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
result_cast_char_set_name	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
result_cast_collation_catalog	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
result_cast_collation_schema	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
result_cast_collation_name	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
result_cast_numeric_precision	cardinal_number	PostgreSQLで利用できない機能に適用されるものです。
result_cast_numeric_precision_radix	cardinal_number	PostgreSQLで利用できない機能に適用されるものです。
result_cast_numeric_scale	cardinal_number	PostgreSQLで利用できない機能に適用されるものです。
result_cast_datetime_precision	character_data	PostgreSQLで利用できない機能に適用されるものです。
result_cast_interval_type	character_data	PostgreSQLで利用できない機能に適用されるものです。
result_cast_interval_precision	cardinal_number	PostgreSQLで利用できない機能に適用されるものです。
result_cast_type_udt_catalog	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
result_cast_type_udt_schema	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
result_cast_type_udt_name	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
result_cast_scope_catalog	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
result_cast_scope_schema	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
result_cast_scope_name	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
result_cast_maximum_cardinality	cardinal_number	PostgreSQLで利用できない機能に適用されるものです。
result_cast_dtd_identifier	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。

34.41. schemata

schemataビューには、現在のデータベースの中で現在有効なロールが所有するすべてのスキーマがあります。

表 34-39. schemataの列

名前	データ型	説明
catalog_name	sql_identifier	スキーマを持つデータベースの名前です（常に現在のデータベースです）。
schema_name	sql_identifier	スキーマの名前です。
schema_owner	sql_identifier	スキーマの所有者の名前です。
default_character_set_catalog	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
default_character_set_schema	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
default_character_set_name	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
sql_path	character_data	PostgreSQLで利用できない機能に適用されるものです。

34.42. sequences

`sequences` ビューは、現在のデータベース内で定義されたすべてのシーケンスがあります。現在のユーザが（所有者である、または、何らかの権限を持つために）アクセスできるシーケンスのみが表示されます。

表 34-40. `sequences` の列

名前	データ型	説明
<code>sequence_catalog</code>	<code>sql_identifier</code>	シーケンスを含むデータベースの名前です（常に現在のデータベースです）。
<code>sequence_schema</code>	<code>sql_identifier</code>	シーケンスを含むスキーマの名前です。
<code>sequence_name</code>	<code>sql_identifier</code>	シーケンスの名前です。
<code>data_type</code>	<code>character_data</code>	シーケンスのデータ型です。現在PostgreSQLでは常に <code>bigint</code> です。
<code>numeric_precision</code>	<code>cardinal_number</code>	この列は（宣言された、あるいは、暗黙的な）シーケンスデータ型の精度（上述）です。精度は有効桁数を意味します。 <code>numeric_precision_radix</code> の指定に従い、これは10進数（10を基とする）または2進数（2を基とする）で表現されます。
<code>numeric_precision_radix</code>	<code>cardinal_number</code>	この列は、 <code>numeric_precision</code> および <code>numeric_scale</code> で表現される値の基が何かを識別します。値は2または10です。
<code>numeric_scale</code>	<code>cardinal_number</code>	この列はシーケンスデータ型の（宣言された、または、暗黙的な）位取り（上述）です。位取りは、小数点以下の有効桁数を意味します。 <code>numeric_precision_radix</code> の指定に従い、これは10進数（10を基とする）または2進数（2を基とする）で表現されます。
<code>start_value</code>	<code>character_data</code>	シーケンスの開始値です。
<code>minimum_value</code>	<code>character_data</code>	シーケンスの最小値です。
<code>maximum_value</code>	<code>character_data</code>	シーケンスの最大値です。
<code>increment</code>	<code>character_data</code>	シーケンスの増加値です。
<code>cycle_option</code>	<code>yes_or_no</code>	シーケンスが周回する場合はYES、それ以外はNOです。

SQL標準に従い、開始、最小、最大および増加の値が文字列で返されることに注意してください。

34.43. sql_features

sql_featuresビューには、標準SQLで定義された公式な機能のどれがPostgreSQLでサポートされているかについての情報があります。これは[付録D](#)内に記された情報と同じものです。また、ここには背景について追加情報がいくつかあります。

表 34-41. sql_featuresの列

名前	データ型	説明
feature_id	character_data	機能の識別文字列です。
feature_name	character_data	機能の説明的な名前です。
sub_feature_id	character_data	副機能の識別子です。副機能がない場合は空の文字列です。
sub_feature_name	character_data	副機能の説明的な名前です。副機能がない場合は空の文字列です。
is_supported	yes_or_no	現在のバージョンのPostgreSQLで機能が完全にサポートされている場合はYESです。さもなくばNOです。
is_verified_by	character_data	常にNULLです。PostgreSQL開発グループは機能が準拠しているかどうかについて公式な試験を行っていないからです。
comments	character_data	おそらく機能のサポート状況についてのコメントです。

34.44. sql_implementation_info

sql_implementation_infoビューには、標準SQLで実装依存で定義するものとされている各種状況に関する情報があります。この情報は主にODBCインタフェース環境での使用を意図しています。他のインタフェースのユーザはおそらくこの情報があまり役に立たないものとするでしょう。このため、個々の実装情報項目をここでは記載しません。ODBCインタフェースの説明でそれらを説明します。

表 34-42. sql_implementation_infoの列

名前	データ型	説明
implementation_info_id	character_data	実装情報項目の識別文字列です。
implementation_info_name	character_data	実装情報項目の説明的な名前です。
integer_value	cardinal_number	実装情報項目の値です。その値がcharacter_valueにある場合はNULLです。
character_value	character_data	実装情報項目の値です。その値がinteger_valueにある場合はNULLです。
comments	character_data	おそらく実装情報項目に関するコメントです。

34.45. sql_languages

sql_languages テーブルには、PostgreSQLでサポートされる、各SQL言語バインディングに対して1行が含まれます。PostgreSQLは直接的SQLとC言語による埋め込みSQLをサポートします。これがこのテーブルから理解できるすべてです。

このテーブルはSQL標準 SQL:2008から削除されたので、SQL:2003以降の標準を参照するエントリはありません。

表 34-43. sql_languagesの列

名前	データ型	説明
sql_language_source	character_data	言語定義のソースの名前です。 常に標準SQLを表すISO 9075です。
sql_language_year	character_data	sql_language_source で参照される標準が承認された年です。
sql_language_conformance	character_data	言語バインディングの標準準拠レベルです。 ISO 9075:2003では、これは常にCOREです。
sql_language_integrity	character_data	常にNULLです（この値は標準SQLの初期のバージョンでは重要です）。
sql_language_implementation	character_data	常にNULLです。
sql_language_binding_style	character_data	DIRECTもしくはEMBEDDEDのいずれかの言語バインディング様式です。
sql_language_programming_language	character_data	バインディング様式がEMBEDDEDの場合はプログラミング言語です。 さもなくばNULLです。 PostgreSQLではC言語のみをサポートします。

34.46. sql_packages

sql_packages テーブルには、どの標準SQLで定義された機能パッケージがPostgreSQLでサポートされているかに関する情報があります。 機能パッケージの背景情報については[付録D](#)を参照してください。

表 34-44. sql_packagesの列

名前	データ型	説明
feature_id	character_data	パッケージの識別文字列です。
feature_name	character_data	パッケージの説明的な名前です。
is_supported	yes_or_no	パッケージが現在のバージョンのPostgreSQLで完全にサポートされている場合はYES、さもなくばNOです。
is_verified_by	character_data	常にNULLです。 PostgreSQL開発グループは機能が準拠しているかどうかについて公式な試験を行っていないからです。
comments	character_data	おそらくパッケージのサポート状況に関するコメントです。

34.47. sql_parts

sql_partsテーブルは、標準SQLのどの部分がPostgreSQLでサポートされているかに関する情報を持ちます。

表 34-45. sql_partsの列

名前	データ型	説明
feature_id	character_data	部品番号を含む識別文字列です。
feature_name	character_data	部品の説明的な名称です。
is_supported	yes_or_no	PostgreSQLの現在のバージョンで部品が完全にサポートされている場合YES、さもなければNOです。
is_verified_by	character_data	常にNULLです。 PostgreSQL開発グループは公式な機能準拠検査を行いません。
comments	character_data	おそらく部品のサポート状況に関するコメントです。

34.48. sql_sizing

sql_sizing テーブルには、PostgreSQL中の各種サイズ制限と上限値に関する情報があります。 この情報は主にODBCインタフェース環境での使用を意図しています。 他のインタフェースのユーザはおそらくこの情報があまり役に立たないものと考えられるでしょう。 このため、個々のサイズ調整項目はここでは記載しません。 ODBCインタフェースの説明でそれらを説明します。

表 34-46. sql_sizingの列

名前	データ型	説明
sizing_id	cardinal_number	サイズ調整項目の識別子です。
sizing_name	character_data	サイズ調整項目の説明的な名前です。
supported_value	cardinal_number	サイズ調整項目の値です。サイズに制限がない場合や決定できない場合はゼロです。 サイズ調整項目を適用する機能がサポートされない場合はNULLです。
comments	character_data	おそらくサイズ調整項目に関するコメントです。

34.49. sql_sizing_profiles

sql_sizing_profiles テーブルには、標準SQLの各種プロファイルで要求されるsql_sizing値に関する情報があります。PostgreSQLではSQLプロファイルをまったく追いかけていませんので、このテーブルは空です。

表 34-47. sql_sizing_profilesの列

名前	データ型	説明
sizing_id	cardinal_number	サイズ調整項目の識別子です。
sizing_name	character_data	サイズ調整項目の説明的な名前です。
profile_id	character_data	プロファイルの識別文字列です。
required_value	cardinal_number	サイズ調整項目用のSQLプロファイルで要求される値です。 プロファイルでそのサイズ調整項目に制限を設けていない場合はゼロ、プロファイルでサイズ調整項目を適用する機能を要求していない場合はNULLです。
comments	character_data	おそらくプロファイル内のサイズ調整項目に関するコメントです。

34.50. table_constraints

table_constraintsビューには、現在のユーザが所有する、または何らかのSELECT以外の権限を持つテーブルに属する全ての制約があります。

表 34-48. table_constraintsの列

名前	データ型	説明
constraint_catalog	sql_identifier	制約を持つデータベースの名前です （常に現在のデータベースです）。
constraint_schema	sql_identifier	制約を持つスキーマの名前です。
constraint_name	sql_identifier	制約の名前です。
table_catalog	sql_identifier	テーブルを持つデータベースの名前です （常に現在のデータベースです）。
table_schema	sql_identifier	テーブルを持つスキーマの名前です。
table_name	sql_identifier	テーブルの名前です。
constraint_type	character_data	制約の種類です。 CHECK、FOREIGN KEY、PRIMARY KEY、UNIQUEのいずれかです。
is_deferrable	yes_or_no	制約が遅延可能な場合はYES、さもなければNOです。
initially_deferred	yes_or_no	制約が遅延可能、かつ、初期状態が遅延の場合はYES、さもなければNOです。

34.51. table_privileges

table_privilegesビューは、現在有効なロールに対し、または現在有効なロールによって、テーブルもしくはビューに与えられた権限を全て識別します。 テーブル、譲与者、被譲与者の組み合わせごとに1行があります。

表 34-49. table_privilegesの列

名前	データ型	説明
grantor	sql_identifier	権限を与えたロールの名前です。
grantee	sql_identifier	権限を与えられたロールの名前です。
table_catalog	sql_identifier	テーブルを持つデータベースの名前です（常に現在のデータベースです）。
table_schema	sql_identifier	テーブルを持つスキーマの名前です。
table_name	sql_identifier	テーブルの名前です。
privilege_type	character_data	権限の種類は、SELECT、INSERT、UPDATE、DELETE、TRUNCATE、REFERENCESもしくはTRIGGERです。
is_grantable	yes_or_no	この権限を付与可能な場合はYES、さもなければNOです。
with_hierarchy	yes_or_no	SQL標準では、WITH HIERARCHY OPTIONは、継承テーブル階層に対するある操作を許可する独立した（副次）権限です。 PostgreSQLでは、これはSELECT権限に含まれているため、この列は権限がSELECTの場合はYES、それ以外の場合はNOです。

34.52. tables

tablesビューには、現在のデータベースで定義された全てのテーブルとビューがあります。現在のユーザが（所有している、何らかの権限を持っているといった方法で）アクセスできるテーブルとビューのみが表示されます。

表 34-50. tablesの列

名前	データ型	説明
table_catalog	sql_identifier	テーブルを持つデータベースの名前です（常に現在のデータベースです）。
table_schema	sql_identifier	テーブルを持つスキーマの名前です。
table_name	sql_identifier	テーブルの名前です。
table_type	character_data	テーブルの種類です。永続テーブル（普通のテーブルの種類）ではBASE TABLE、ビューではVIEW、外部テーブルではFOREIGN TABLE、一時テーブルではLOCAL TEMPORARY です。
self_referencing_column_name	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
reference_generation	character_data	PostgreSQLで利用できない機能に適用されるものです。
user_defined_type_catalog	sql_identifier	テーブルが型付けされたテーブルの場合、背後のデータ型を持つデータベース(常に現在のデータベース)の名前です。さもなければNULLです。
user_defined_type_schema	sql_identifier	テーブルが型付けされたテーブルである場合、背後のデータ型を含むスキーマの名前です。さもなければNULLです。
user_defined_type_name	sql_identifier	テーブルが型付けされたテーブルである場合、背後のデータ型の名前です。さもなければNULLです。
is_insertable_into	yes_or_no	テーブルが挿入可能な場合YES、さもなければNOです。（ベーステーブルは常に挿入可能ですが、ビューはそうとも限りません。）
is_typed	yes_or_no	テーブルが型付けされたテーブルの場合YES、そうでなければNOです。
commit_action	character_data	未実装です。

34.53. triggered_update_columns

列リスト(UPDATE OF column1, column2など)を指定する現在のデータベース内のトリガに関して、triggered_update_columnsビューはこれらの列を識別します。 列リストを指定しないトリガはこのビューには含まれません。 これらの列の内、現在のユーザが所有するまたはSELECT以外の何らかの権限を持つもののみが示されます。

表 34-51. triggered_update_columnsの列

名前	データ型	説明
trigger_catalog	sql_identifier	トリガを持つデータベースの名前です(常に現在のデータベースです)。
trigger_schema	sql_identifier	トリガを持つスキーマの名前です。
trigger_name	sql_identifier	トリガの名前です。
event_object_catalog	sql_identifier	トリガが定義されたテーブルを持つデータベースの名前です。(常に現在のデータベースです。)
event_object_schema	sql_identifier	トリガが定義されたテーブルを持つスキーマの名前です。
event_object_table	sql_identifier	トリガが定義されたテーブルの名前です。
event_object_column	sql_identifier	トリガが定義された列の名前です。

34.54. triggers

triggersビューには、現在のデータベース内で、現在のユーザが所有するあるいは何らかのSELECT以外の権限を持つテーブルまたはビューに定義された、全てのトリガがあります。

表 34-52. triggersの列

名前	データ型	説明
trigger_catalog	sql_identifier	トリガを持つデータベースの名前です（常に現在のデータベースです）。
trigger_schema	sql_identifier	トリガを持つスキーマの名前です。
trigger_name	sql_identifier	トリガの名前です。
event_manipulation	character_data	トリガを発するイベントです（INSERT、UPDATEもしくはDELETEです）。
event_object_catalog	sql_identifier	トリガが定義されたテーブルを持つデータベースの名前です（常に現在のデータベースです）。
event_object_schema	sql_identifier	トリガが定義されたテーブルを持つスキーマの名前です。
event_object_table	sql_identifier	トリガが定義されたテーブルの名前です。
action_order	cardinal_number	未実装です。
action_condition	character_data	トリガのWHEN条件です。なければNULLです（現在有効なロールが所有していないテーブルの場合もNULLです）。
action_statement	character_data	トリガによって実行される文です（現在は常にEXECUTE PROCEDURE function(...)です）。
action_orientation	character_data	トリガの発行が処理行ごとか文ごとかを識別します（ROWもしくはSTATEMENTです）。
action_timing	character_data	トリガを発する時期です（BEFORE、AFTERもしくはINSERT OFです）。
action_reference_old_table	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
action_reference_new_table	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
action_reference_old_row	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
action_reference_new_row	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
created	time_stamp	PostgreSQLで利用できない機能に適用されるものです。

PostgreSQLにおけるトリガには、標準SQLと比べ、2つの非互換があり、これらは情報スキーマの表現に影響を与えます。1つ目は、PostgreSQLではトリガ名は、独立したスキーマオブジェクトではなく、それぞれのテーブル内で局所的であることです。そのため、別のテーブルに属している場合、1つのスキーマ内でトリガ名を重複させることができます。

（trigger_catalogとtrigger_schemaは実際、そのトリガが定義されたテーブルに属する値となります。）2つ目は、PostgreSQLではトリガは複数のイベントで発行できる点です（例えばON INSERT OR UPDATEです）。一方、標準SQLでは1つのみしか許されません。トリガが複数のイベントで発行するように定義された場合、それぞれのイベントで1行という形で、情報スキーマ内では複数の行として表現されます。これらの2つの問題の結果、triggersビューのプライマリキーは実際、標準SQLで定義された(trigger_catalog, trigger_schema, trigger_name)ではなく、(trigger_catalog, trigger_schema, event_object_table, trigger_name, event_manipulation)となります。それでもなお、標準SQLに従う（スキーマ内でトリガ名を一意とし、トリガに対し1種類のイベントしか持たせないという）手法でトリガを定義していれば、これは影響ありません。

注意: PostgreSQL 9.1 より前は、このビューの列の action_timing、action_reference_old_table、action_reference_new_table、action_reference_old_row、action_reference_new_row はそれぞれ condition_timing、condition_reference_old_table、condition_reference_new_table、condition_reference_old_row、condition_reference_new_row という名前でした。これらの命名は SQL:1999標準におけるものです。新しい名前はSQL:2003以降に準拠しています。

34.55. udt_privileges

udt_privilegesビューは、現在有効なロールが付与者または被付与者である、ユーザ定義型に付与されたUSAGE権限を識別します。列、付与者、被付与者の組み合わせごとに行があります。このビューは複合データ型のみを表示します（理由は[項34.57](#)を参照してください）。ドメイン権限については[項34.56](#)を参照してください。

表 34-53. udt_privileges Columns

Name	Data Type	Description
grantor	sql_identifier	Name of the role that granted the privilege
grantee	sql_identifier	Name of the role that the privilege was granted to
udt_catalog	sql_identifier	Name of the database containing the type (always the current database)
udt_schema	sql_identifier	Name of the schema containing the type
udt_name	sql_identifier	Name of the type
privilege_type	character_data	Always TYPE USAGE
is_grantable	yes_or_no	YES if the privilege is grantable, NO if not

34.56. usage_privileges

usage_privilegesビューは、現在有効なロールに、もしくは現在有効なロールによって与えられた、各種オブジェクト上のUSAGE権限を識別します。これは今のところ、PostgreSQLでは照合、ドメイン、外部データラッパ、外部サーバ、およびシーケンスに適用します。オブジェクトと許可を与えた者、許可を受けた者の組み合わせごとに1行があります。

PostgreSQLでは、照合は実際の権限を所有しませんので、このビューは全ての照合に対して所有者からPUBLICに与えられた暗黙の付与できないUSAGE権限を示します。しかし、その他のオブジェクトの種類は実際の権限を示します。

PostgreSQLでは、シーケンスはUSAGEに加えてSELECTとUPDATE権限もサポートします。これらは非標準であるため、情報スキーマのビューでは参照できません。

表 34-54. usage_privilegesの列

名前	データ型	説明
grantor	sql_identifier	権限を付与されたロール名
grantee	sql_identifier	権限がそれに与えられたロール名
object_catalog	sql_identifier	オブジェクトを持つデータベースの名前です（常に現在のデータベースです）。
object_schema	sql_identifier	適用されるオブジェクトを持つスキーマの名前。そうでなければ空文字列
object_name	sql_identifier	オブジェクトの名前です。
object_type	character_data	COLLATIONまたはDOMAINまたはFOREIGN DATA WRAPPERまたはFOREIGN SERVERまたはSEQUENCE
privilege_type	character_data	常にUSAGEです。
is_grantable	yes_or_no	権限が付与可能であればYES、そうでなければNO

34.57. user_defined_types

user_defined_typesビューは、現在は現在のデータベースで定義された全ての複合データ型を含みます。表示される型は、現在のユーザが（所有者である、何らかの権限を持っているといった方法で）アクセスできるものだけです。

SQLは二種類のユーザ定義データ型を知っています。構造化型（PostgreSQLでは複合データ型として知られています）と特殊型（PostgreSQLでは実装されていません）。将来を見越して、user_defined_type_category列をこれらを区別するために使用します。PostgreSQLの拡張である基本型や列挙型といった他のユーザ定義型はここには表示されません。ドメインについては代わりに[項34.22](#)を参照してください。

表 34-55. user_defined_typesの列

名前	データ型	説明
user_defined_type_catalog	sql_identifier	型を持つデータベースの名前です（常に現在のデータベースです）。
user_defined_type_schema	sql_identifier	型を持つスキーマの名前です。
user_defined_type_name	sql_identifier	型の名前です。
user_defined_type_category	character_data	現在は常にSTRUCTUREDです。
is_instantiable	yes_or_no	PostgreSQLで利用できない機能に適用されるものです。
is_final	yes_or_no	PostgreSQLで利用できない機能に適用されるものです。
ordering_form	character_data	PostgreSQLで利用できない機能に適用されるものです。
ordering_category	character_data	PostgreSQLで利用できない機能に適用されるものです。
ordering_routine_catalog	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
ordering_routine_schema	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
ordering_routine_name	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
reference_type	character_data	PostgreSQLで利用できない機能に適用されるものです。
data_type	character_data	PostgreSQLで利用できない機能に適用されるものです。
character_maximum_length	cardinal_number	PostgreSQLで利用できない機能に適用されるものです。
character_octet_length	cardinal_number	PostgreSQLで利用できない機能に適用されるものです。
character_set_catalog	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
character_set_schema	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
character_set_name	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
collation_catalog	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
collation_schema	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
collation_name	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
numeric_precision	cardinal_number	PostgreSQLで利用できない機能に適用されるものです。
numeric_precision_radix	cardinal_number	PostgreSQLで利用できない機能に適用されるものです。
numeric_scale	cardinal_number	PostgreSQLで利用できない機能に適用されるものです。
datetime_precision	cardinal_number	PostgreSQLで利用できない機能に適用されるものです。
interval_type	character_data	PostgreSQLで利用できない機能に適用されるものです。
interval_precision	cardinal_number	PostgreSQLで利用できない機能に適用されるものです。
source_dtd_identifier	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。
ref_dtd_identifier	sql_identifier	PostgreSQLで利用できない機能に適用されるものです。

34.58. user_mapping_options

user_mapping_options ビューは現在のデータベースでユーザマッピングに定義された全てのオプションを含みます。現在のユーザが対応する外部サーバへのアクセスを所有しているとき（所有者または何らかの権限を持っていることで）表示されます。

表 34-56. user_mapping_optionsの列

名前	データ型	説明
authorization_identifier	sql_identifier	マップされているユーザ名、またはマッピングがpublicであればPUBLICです。
foreign_server_catalog	sql_identifier	このマッピングにより使用される外部サーバが定義されたデータベース名です（常に現在のデータベースです）。
foreign_server_name	sql_identifier	このマッピングに使用される外部サーバ名です。
option_name	sql_identifier	オプションの名前です。
option_value	character_data	オプションの値です。この列は、現在のユーザがマップされたユーザか、マッピングがPUBLIC用かつ現在のユーザがサーバ所有者、もしくは現在のユーザがスーパーユーザでない限りNULLを示します。ユーザマッピングオプションとして格納されているパスワード情報を保護するためです。

34.59. user_mappings

user_mappingsビューは現在のデータベースで定義されたユーザマッピングすべてを含みます。現在のユーザが対応する外部サーバへアクセスを持っているとき（所有者か、何らかの権限を持っていることとして）それらのユーザのマッピングのみを示します。

表 34-57. user_mappingsの列

名前	データ型	説明
authorization_identifier	sql_identifier	マップされているユーザ名、またはマッピングがpublicの場合PUBLICです。
foreign_server_catalog	sql_identifier	このマッピングにより使用される外部サーバが定義されたデータベース名です（常に現在のデータベースです）。
foreign_server_name	sql_identifier	このマッピングで使用された外部サーバ名です。

34.60. view_column_usage

view_column_usageビューは、ビューの問い合わせ式（ビューを定義するSELECT文）で使用される全ての列を識別します。現在有効なロールがその列を含むテーブルの所有者であるもののみが含まれます。

注意: システムテーブルの列は含まれません。 これはいつか修正しなければなりません。

表 34-58. view_column_usageの列

名前	データ型	説明
view_catalog	sql_identifier	ビューを持つデータベースの名前です（常に現在のデータベースです）。
view_schema	sql_identifier	ビューを持つスキーマの名前です。
view_name	sql_identifier	ビューの名前です。
table_catalog	sql_identifier	ビューで使用される列を持つテーブルが含まれるデータベースの名前です（常に現在のデータベースです）。
table_schema	sql_identifier	ビューで使用される列を持つテーブルが含まれるスキーマの名前です。
table_name	sql_identifier	ビューで使用される列を持つテーブルの名前です。
column_name	sql_identifier	ビューで使用される列の名前です。

34.61. view_routine_usage

view_routine_usageビューは、ビューの問い合わせ式（ビューを定義するSELECT文）で使用されるすべてのルーチン（関数およびプロシージャ）を識別します。 現在有効なロールが所有するルーチンのみが含まれます。

表 34-59. view_routine_usageの列

名前	データ型	説明
table_catalog	sql_identifier	ビューを持つデータベースの名前です。（常に現在のデータベースです。）
table_schema	sql_identifier	ビューを持つスキーマの名前です。
table_name	sql_identifier	ビューの名前です。
specific_catalog	sql_identifier	関数を持つデータベースの名前です。（常に現在のデータベースです。）
specific_schema	sql_identifier	関数を持つスキーマの名前です。
specific_name	sql_identifier	関数の"仕様名称"です。 詳細は 項34.40 を参照してください。

34.62. view_table_usage

view_table_usageビューは、ビューの問い合わせ式（ビューを定義するSELECT文）で使用されるすべてのテーブルを識別します。現在有効なロールがそのテーブルの所有者であるもののみが含まれます。

注意: システムテーブルは含まれません。これはいつか修正しなければなりません。

表 34-60. view_table_usageの列

名前	データ型	説明
view_catalog	sql_identifier	ビューを持つデータベースの名前です（常に現在のデータベースです）。
view_schema	sql_identifier	ビューを持つスキーマの名前です。
view_name	sql_identifier	ビューの名前です。
table_catalog	sql_identifier	ビューで使用されるテーブルを持つデータベースの名前です（常に現在のデータベースです）。
table_schema	sql_identifier	ビューで使用されるテーブルを持つスキーマの名前です。
table_name	sql_identifier	ビューで使用されるテーブルの名前です。

34.63. views

viewsビューには、現在のデータベースで定義されたすべてのビューがあります。現在のユーザが（所有者である、何らかの権限を持っているといった方法で）アクセスすることができるビューのみが表示されます。

表 34-61. viewsの列

名前	データ型	説明
table_catalog	sql_identifier	ビューを持つデータベースの名前です（常に現在のデータベースです）。
table_schema	sql_identifier	ビューを持つスキーマの名前です。
table_name	sql_identifier	ビューの名前です。
view_definition	character_data	ビューを定義する問い合わせ式です（現在有効なロールがビューの所有者でない場合はNULLです）。
check_option	character_data	PostgreSQLで利用できない機能に適用されるものです。
is_updatable	yes_or_no	ビューが更新可能（UPDATEおよびDELETEが可能）な場合YES、さもなければNOです。
is_insertable_into	yes_or_no	ビューが挿入可能（INSERTが可能）な場合YES、さもなければNOです。
is_trigger_updatable	yes_or_no	ビューにINSTEAD OF UPDATEトリガーが定義されている場合はYES、さもなければNOです。
is_trigger_deletable	yes_or_no	ビューにINSTEAD OF DELETEトリガーが定義されている場合はYES、さもなければNOです。
is_trigger_insertable_into	yes_or_no	ビューにINSTEAD OF INSERTトリガーが定義されている場合はYES、さもなければNOです。

V. サーバプログラミング

ここでは、ユーザ定義の関数、データ型、演算子、トリガなどを使用してサーバの機能を拡張する方法について説明します。これらはおそらく、PostgreSQLに関するユーザ向けの文書を理解した後にのみたどり着く先進的な話題です。また、最後の数章でPostgreSQLに附属するサーバサイドのプログラミング言語についても説明します。同時にサーバサイドのプログラミング言語に関する一般的な問題についても説明します。サーバサイドのプログラミング言語の章に進む前に、少なくとも、[第35章](#)（関数も説明しています）の最初の数節を読破することは重要です。

目次

35章 [SQLの拡張](#)

- 35.1. [拡張の作用法](#)
- 35.2. [PostgreSQLの型システム](#)
- 35.3. [ユーザ定義関数](#)
- 35.4. [問い合わせ言語（SQL）関数](#)
- 35.5. [関数のオーバーロード](#)
- 35.6. [関数の変動性分類](#)
- 35.7. [手続き型言語関数](#)
- 35.8. [内部関数](#)
- 35.9. [C言語関数](#)
- 35.10. [ユーザ定義の集約](#)
- 35.11. [ユーザ定義の型](#)
- 35.12. [ユーザ定義の演算子](#)
- 35.13. [演算子最適化に関する情報](#)
- 35.14. [インデックス拡張機能へのインタフェース](#)
- 35.15. [関連するオブジェクトを拡張としてパッケージ化](#)
- 35.16. [拡張構築基盤](#)

36章 [トリガ](#)

- 36.1. [トリガ動作の概要](#)
- 36.2. [データ変更の可視性](#)
- 36.3. [Cによるトリガ関数の作成](#)
- 36.4. [完全なトリガの例](#)

37章 [ルールシステム](#)

- 37.1. [問い合わせツリーとは](#)
- 37.2. [ビューとルールシステム](#)
- 37.3. [INSERT、UPDATE、DELETEについてのルール](#)
- 37.4. [ルールと権限](#)
- 37.5. [ルールおよびコマンドの状態](#)
- 37.6. [ルール対トリガ](#)

38章 [手続き言語](#)

- 38.1. [手続き言語のインストール](#)

39章 [PL/pgSQL - SQL手続き言語](#)

- 39.1. [概要](#)
- 39.2. [PL/pgSQLの構造](#)
- 39.3. [宣言](#)
- 39.4. [式](#)
- 39.5. [基本的な文](#)
- 39.6. [制御構造](#)
- 39.7. [カーソル](#)
- 39.8. [エラーとメッセージ](#)
- 39.9. [トリガプロシージャ](#)
- 39.10. [PL/pgSQLの秘訣](#)
- 39.11. [PL/pgSQLによる開発向けのヒント](#)
- 39.12. [Oracle PL/SQLからの移植](#)

40章[PL/Tcl - Tcl手続き言語](#)

- 40.1. [概要](#)
- 40.2. [PL/Tcl関数と引数](#)
- 40.3. [PL/Tclにおけるデータの値](#)
- 40.4. [PL/Tclにおけるグローバルデータ](#)
- 40.5. [PL/Tclからのデータベースアクセス](#)
- 40.6. [PL/Tclのトリガプロシージャ](#)
- 40.7. [モジュールとunknownコマンド](#)
- 40.8. [Tclプロシージャ名](#)

41章[PL/Perl - Perl手続き言語](#)

- 41.1. [PL/Perl関数と引数](#)
- 41.2. [PL/Perlにおけるデータ値](#)
- 41.3. [組込み関数](#)
- 41.4. [PL/Perlにおけるグローバルな値](#)
- 41.5. [信頼されたPL/Perlおよび信頼されないPL/Perl](#)
- 41.6. [PL/Perlトリガ](#)
- 41.7. [PL/Perlの内部](#)

42章[PL/Python - Python手続き言語](#)

- 42.1. [Python 2対Python 3](#)
- 42.2. [PL/Python関数](#)
- 42.3. [データ値](#)
- 42.4. [データの共有](#)
- 42.5. [匿名コードブロック](#)
- 42.6. [トリガ関数](#)
- 42.7. [データベースアクセス](#)
- 42.8. [明示的サブトランザクション](#)
- 42.9. [ユーティリティ関数](#)
- 42.10. [環境変数](#)

43章[サーバプログラミングインタフェース](#)

- 43.1. [インタフェース関数](#)
- 43.2. [インタフェースサポート関数](#)
- 43.3. [メモリ管理](#)
- 43.4. [データ変更の可視性](#)
- 43.5. [例](#)

第 35 章 SQL の拡張

目次

- 35.1. [拡張の作用法](#)
- 35.2. [PostgreSQL の型システム](#)
 - 35.2.1. [基本型](#)
 - 35.2.2. [複合型](#)
 - 35.2.3. [ドメイン](#)
 - 35.2.4. [疑似型](#)
 - 35.2.5. [多様型](#)
- 35.3. [ユーザ定義関数](#)
- 35.4. [問い合わせ言語 \(SQL\) 関数](#)
 - 35.4.1. [SQL 関数用の引数](#)
 - 35.4.2. [基本型を使用する SQL 関数](#)
 - 35.4.3. [複合型を使用する SQL 関数](#)
 - 35.4.4. [出力パラメータを持つ SQL 関数](#)
 - 35.4.5. [可変長引数を取る SQL 関数](#)
 - 35.4.6. [引数にデフォルト値を持つ SQL 関数](#)
 - 35.4.7. [テーブルソースとしての SQL 関数](#)
 - 35.4.8. [集合を返す SQL 関数](#)
 - 35.4.9. [TABLE を返す SQL 関数](#)
 - 35.4.10. [多様 SQL 関数](#)
 - 35.4.11. [照合順序を持つ SQL 関数](#)
- 35.5. [関数のオーバーロード](#)
- 35.6. [関数の変動性分類](#)
- 35.7. [手続き型言語関数](#)
- 35.8. [内部関数](#)
- 35.9. [C 言語関数](#)
 - 35.9.1. [動的ロード](#)
 - 35.9.2. [C 言語関数における基本型](#)
 - 35.9.3. [Version 0 呼び出し規約](#)
 - 35.9.4. [Version 1 呼び出し規約](#)
 - 35.9.5. [コードの作成](#)
 - 35.9.6. [動的にロードされる関数のコンパイルとリンク](#)
 - 35.9.7. [複合型引数](#)
 - 35.9.8. [行 \(複合型\) を返す](#)
 - 35.9.9. [集合を返す](#)
 - 35.9.10. [引数と戻り値の多様性](#)
 - 35.9.11. [変形関数](#)
 - 35.9.12. [共有メモリと LWLocks](#)
 - 35.9.13. [拡張への C++ の利用](#)
- 35.10. [ユーザ定義の集約](#)
- 35.11. [ユーザ定義の型](#)
- 35.12. [ユーザ定義の演算子](#)
- 35.13. [演算子最適化に関する情報](#)
 - 35.13.1. [COMMUTATOR](#)
 - 35.13.2. [NEGATOR](#)
 - 35.13.3. [RESTRICT](#)
 - 35.13.4. [JOIN](#)
 - 35.13.5. [HASHES](#)
 - 35.13.6. [MERGES](#)
- 35.14. [インデックス拡張機能へのインタフェース](#)
 - 35.14.1. [インデックスメソッドと演算子クラス](#)
 - 35.14.2. [インデックスメソッドのストラテジ](#)

- 35.14.3. [インデックスメソッドのサポートルーチン](#)
- 35.14.4. [例](#)
- 35.14.5. [演算子クラスと演算子族](#)
- 35.14.6. [システムの演算子クラスに対する依存性](#)
- 35.14.7. [順序付け演算子](#)
- 35.14.8. [演算子クラスの特殊な機能](#)

35.15. [関連するオブジェクトを拡張としてパッケージ化](#)

- 35.15.1. [拡張のファイル](#)
- 35.15.2. [拡張の再配置性](#)
- 35.15.3. [拡張設定テーブル](#)
- 35.15.4. [拡張の更新](#)
- 35.15.5. [拡張の例](#)

35.16. [拡張構築基盤](#)

本節では以下を追加することでPostgreSQLのSQL問い合わせ言語をどのように拡張できるかを説明します。

- 関数 ([項35.3](#)から)。
- 集約 ([項35.10](#)から)。
- データ型 ([項35.11](#)から)。
- 演算子 ([項35.12](#)から)。
- インデックス用演算子クラス ([項35.14](#)から)。
- 関連オブジェクトのパッケージ ([項35.15](#)から)。

[前のページ](#)
サーバプログラミング

[ホーム](#)
[上に戻る](#)

[次のページ](#)
拡張の作用法

35.1. 拡張の作用法

PostgreSQLは、自分自身の操作がカタログに定義された方法で駆動されているため拡張が可能です。もし標準のリレーショナルデータベースシステムに慣れ親しんでいるのであれば、システムカタログとして一般に知られている中に、データベース、テーブル、列などの情報が格納されていることは知っていると思います。（システムによってはデータディクショナリと呼ぶものもあります。）このカタログはユーザの目には他のテーブルと同じように見えますが、DBMSは内部情報をそこに格納しているのです。PostgreSQLと標準的なリレーショナルデータベースシステムの重要な違いは、PostgreSQLはカタログにより多くの情報を格納するということです。テーブルと列に関する情報だけではなく、データ型、関数、アクセスメソッドなどの情報も格納されています。これらのテーブルはユーザが変更できます。そして、PostgreSQLは操作をこれらのテーブルに基づいて行うので、PostgreSQLはユーザによって拡張することができるのです。これに対して、一般のデータベースシステムでは、ソースコード内にハードコーディングされたプロシージャを変えるか、DBMSベンダによって特別に書かれたモジュールをロードしなければ拡張することができません。

さらにPostgreSQLサーバは動的ローディングによってユーザの作成したコードを取り入れることができます。つまり、ユーザが新しい型か関数を実装するオブジェクトコードファイル（例えば共有ライブラリ）を指定することができ、PostgreSQLは要求された時にロードします。SQLで作成されたコードをサーバに追加するのはさらに簡単です。このように演算を"その場で"変えることができるため、PostgreSQLは新しいアプリケーションや格納構造をラピッドプロトタイプする場合に適しています。

35.2. PostgreSQLの型システム

PostgreSQLのデータ型は、基本型、複合型、ドメイン、疑似型に分類されます。

35.2.1. 基本型

基本型は`int4`のように、SQL言語レベル以下で実装されたものです（通常はCのような低レベル言語で作成されます）。一般的にこれらは抽象データ型とも呼ばれるものに対応します。PostgreSQLは、ユーザによって提供された関数を通して、こうした型に対する操作のみが可能です。そして、こうした型の動作のみをユーザが記述する範囲まで理解することができます。さらに基本型は、スカラと配列型に分類されます。各スカラ型に対応する配列型は自動的に作成され、そこにはそのスカラ型の可変長の配列を保持できます。

35.2.2. 複合型

ユーザがテーブルを作成すると、複合型、もしくは行型が作成されます。関連するテーブルを持たない"スタンドアロン"の複合型を[CREATE TYPE](#)を使用して定義することもできます。複合型は関連したフィールド名を持つ基本型の単なるリストです。複合型の値は行もしくはフィールド値のレコードです。ユーザはSQL問い合わせによってその構成フィールドにアクセスすることができます。複合型のより詳細については[項8.16](#)を参照してください。

35.2.3. ドメイン

ドメインは、特定の基本型に基づいたもので、多くの目的では、その基本型と交換可能です。しかし、ドメインは背後にある基本型で許可される範囲内で値の有効範囲を制限する制約を持つことができます。

ドメインは[CREATE DOMAIN](#) SQLコマンドを使用して作成可能です。ドメインの作成と使用については本章では説明しません。

35.2.4. 疑似型

特殊な目的用に数個の"疑似型"があります。疑似型はテーブルの列や複合型の属性として現れることはありません。しかし、関数の引数や結果型を宣言する際に使用することができます。これは、型システム内で特殊な関数クラスを識別するための機構を提供します。[表8-24](#)に既存の疑似型を列挙します。

35.2.5. 多様型

特殊な用途を持つ疑似型には、`anyelement`と`anyarray`、`anynonarray`、`anyenum`および`anyrange`の5つがあります。これらはまとめて多様型と呼ばれます。これらの型を使用すると宣言された関数は全て、多様関数と呼ばれます。多様関数は多くの異なるデータ型を操作することができます。データ型の指定は、特定の呼び出しに実際に渡されるデータ型によって決定されます。

多様引数と結果は互いに結び付いており、多様関数を呼び出す問い合わせが解析される時に特定のデータ型が決定されます。`anyelement`として宣言された位置（引数もしくは戻り値）にはそれぞれ、任意の実データ型を指定することができますが、1つの呼び出しでは、これらはすべて同一の実データ型でなければなりません。`anyarray`として宣言された位置には、任意の配列データ型を持つことができます。しかし、同様にこれらはすべて同じデータ型でなければなりません。また同様に`anyrange`として宣言された位置はすべて同じ範囲型でなければなりません。さらに`anyarray`と宣言された位置と`anyelement`と宣言された位置の両方がある場合、`anyarray`の位置の実際の配列型は、その要素の型が`anyelement`位置に現れる型と同じでなければなりません。同様に`anyrange`と宣言された位置と`anyelement`と宣言された位置の両方がある場合、`anyrange`の位置の実際の範囲型は、その範囲の派生元型が`anyelement`位置に現れる型と同じでなければなりません。`anynonarray`は、実際の型が配列型であってはならないという制限が加わっている点を除き、`anyelement`とまったく同様に扱われます。`anyenum`は、実際の型が列挙型でなければなければならないという制約が加わっている点を除き、`anyelement`とまったく同様に扱われます。

このように、2つ以上の引数位置が多様型と宣言されると、全体の効果として、実引数型の特定の組み合わせのみが許されるようになります。例えば、`equal(anyelement, anyelement)`と宣言された関数は、2つの引数が同じデータ型である限り、任意の入力値を2つ取ることになります。

関数の戻り値を多様型として宣言する時、少なくとも1つの引数位置も多様でなければなりません。そして引数として与えられる実データ型がその呼び出しの実結果型を決定します。例えば、配列添字機構がなかったとすると、`subscript(anyarray, integer) returns anyelement`として添字機構を実装する関数を定義できます。この宣言には、最初の実引数は配列型になり、パーサはこの最初の実引数の型より正しい結果型を推論することができます。他にも例えば、`f(anyarray) returns anyenum`と宣言された関数は列挙型の配列のみを受け付けます。

`anynonarray`型と`anyenum`型が、別個の型変数を表していないことに注意してください。これは`anyelement`と同じ型で、追加の制約が付いているだけです。例えば、`f(anyelement, anyenum)`として関数を宣言することは、`f(anyenum, anyenum)`と宣言することと同一です。両方の実引数は同じ列挙型でなければなりません。

可変長引数の関数（[項35.4.5](#)で説明する可変個の引数を取る関数）を多様とすることができます。最後のパラメータを`VARIADIC anyarray`と宣言することで実現されます。引数を一致させ、実際の結果型を決めるために、こうした関数は`anynonarray`パラメータを適切な個数記述した場合と同様に動作します

[前のページ](#)
拡張の作用法

[ホーム](#)
[上に戻る](#)

[次のページ](#)
ユーザ定義関数

35.3. ユーザ定義関数

PostgreSQLは4種類の関数を提供します。

- 問い合わせ言語関数（SQLで作成された関数） ([項35.4](#))
- 手続き型言語関数（PL/pgSQLやPL/Tclなどで作成された関数） ([項35.7](#))
- 内部関数 ([項35.8](#))
- C言語関数 ([項35.9](#))

すべての関数は、基本型、複合型、またはこの組み合わせを引数（パラメータ）として受け付けることが可能です。また、すべての関数は基本型または複合型を返すことが可能です。関数は、基本型の集合または複合型の集合を返すように定義することもできます。

多くの関数は特定の疑似型（多様型として）を引数としたり返したりすることができます。しかし、利用できる機能は様々です。詳細は各関数種類の説明を参照してください。

SQL関数の定義の方法が最も簡単ですので、そちらから説明します。SQL関数にある概念のほとんどは、他の種類の関数にも適用できます。

本章の全体に関して、その例をより理解するために、[CREATE FUNCTION](#)コマンドのマニュアルページを一読することが有用です。本章の例のいくつかはPostgreSQLソース配布物内のsrc/tutorialディレクトリにあるfuncs.sqlとfuncs.cでも参照することができます。

35.4. 問い合わせ言語 (SQL) 関数

SQL関数は、任意のSQL文のリストを実行し、そのリストの最後の問い合わせの結果を返します。単純な（集合ではない）場合、最後の問い合わせの結果の最初の行が返されます。（複数行の結果のうちの"最初の行"は、ORDER BYを使用しない限り定義付けることができないことを覚えておってください。）最後の問い合わせが何も行を返さない時はNULL値が返されます。

他にも、SQL関数は、`SETOF sometype`型を返すように指定すること、または同意の`RETURNS TABLE(columns)`と宣言することにより、集合を返すように宣言することもできます。この場合、最後の問い合わせの結果のすべての行が返されます。詳細は後で説明します。

SQL関数の本体は、セミコロンで区切ったSQL文のリストでなければなりません。最後の文の後のセミコロンは省略可能です。関数が`void`を返すものと宣言されていない限り、最後の文は`SELECT`、または`RETURNING`句を持つ`INSERT`、`UPDATE`、または`DELETE`でなければなりません。

SQL言語で作成された、任意のコマンド群はまとめて、関数として定義することができます。SELECT問い合わせ以外に、データ変更用の問い合わせ（つまり、`INSERT`、`UPDATE`、`DELETE`）やその他のSQLコマンドを含めることができます。（SQL関数では`COMMIT`、`SAVEPOINT`などのトランザクション制御コマンドおよび`VACUUM`などのユーティリティコマンドは使用することはできません。）しかし、最後のコマンドは、関数の戻り値型として定義したものを返す`SELECT`、または`RETURNING`句があるものでなければなりません。その他にも、何か動作をさせるが、有用な値を返さないSQL関数を定義したいのであれば、`void`を返すものと定義することで実現可能です。たとえば、以下の関数は`emp`テーブルから負の給料となっている行を削除します。

```
CREATE FUNCTION clean_emp() RETURNS void AS '
    DELETE FROM emp
    WHERE salary < 0;
' LANGUAGE SQL;

SELECT clean_emp();

clean_emp
-----
(1 row)
```

`CREATE FUNCTION`コマンドの構文では、関数本体は文字列定数として作成される必要があります。この文字列定数の記述には、通常、ドル引用符付け（[項4.1.2.4](#)）が最も便利です。文字列定数を単一引用符で括る通常の構文では、関数本体中で使用される単一引用符（`'`）とバックスラッシュ（`\`）（エスケープ文字列構文を仮定）を二重にしなければなりません（[項4.1.2.1](#)を参照）。

35.4.1. SQL関数用の引数

SQL関数の引数は関数本体内で名前または番号を用いて参照することができます。両方の方法の例を後で示します。

名前を使用するためには、関数引数を名前を持つものとして宣言し、その名前を関数本体内で記述するだけです。引数名が関数内の現在のSQLコマンドにおける任意の列名と同じ場合は、列名が優先されます。これを上書きするために、`function_name.argument_name`のように、引数名を関数自身の名前を付けて修飾してください。（もしこれも修飾された列名と競合する場合は、列名が優先されます。SQLコマンド内でテーブルに他の別名を付けることで、この曖昧さを防止することができます。）

古い番号による方法では、引数は関数本体内で`$n`という構文を用いて表すことができます。つまり、`$1`は第1引数を示し、`$2`は第2引数のようになります。これは特定の引数が名前付きで宣言されているかどうかに関係なく動作します。

引数が複合型の場合、`argname.fieldname`や`$1.fieldname`のようなドット表記を用いて引数の属性にアクセスすることができます。繰り返しになりますが、引数名を持つ形式で曖昧さが発生する場合には関数名で引数名を修飾してください。

SQL関数の引数は、識別子としてではなく、データ値としてのみ使用することができます。したがって、例えば

```
INSERT INTO mytable VALUES ($1);
```

は正しいものですが、以下は動作しません。

```
INSERT INTO $1 VALUES (42);
```

注意: SQL関数の引数を参照するために名前を使用できる機能は、PostgreSQL 9.2で追加されました。これより古いサーバ内で使われる関数は`$n`記法を使用しなければなりません。

35.4.2. 基本型を使用するSQL関数

最も簡単なSQL関数は、引数を取らずに単にintegerのような基本型を返すものです。

```
CREATE FUNCTION one() RETURNS integer AS $$
    SELECT 1 AS result;
$$ LANGUAGE SQL;

-- 文字列リテラルの別の構文では
CREATE FUNCTION one() RETURNS integer AS '
    SELECT 1 AS result;
' LANGUAGE SQL;

SELECT one();

    one
-----
    1
```

関数本体内で関数の結果用に列の別名を（`result`という名前）で定義したことに注目してください。しかし、この列の別名はこの関数の外部からは可視ではありません。したがって、その結果は`result`ではなく、`one`というラベルで表示されています。

基本型を引数として取る、SQL関数を定義することはほとんどの場合簡単です。

```
CREATE FUNCTION add_em(x integer, y integer) RETURNS integer AS $$
    SELECT x + y;
$$ LANGUAGE SQL;

SELECT add_em(1, 2) AS answer;

    answer
-----
    3
```

この他に、引数に名前を付けることを省くことができます。この場合は番号を使用します。

```
CREATE FUNCTION add_em(integer, integer) RETURNS integer AS $$
    SELECT $1 + $2;
$$ LANGUAGE SQL;

SELECT add_em(1, 2) AS answer;

    answer
-----
    3
```

以下にもう少し役に立つ関数を示します。これは銀行口座からの引き落としに使用できます。

```
CREATE FUNCTION tf1 (accountno integer, debit numeric) RETURNS integer AS $$
    UPDATE bank
        SET balance = balance - debit
        WHERE accountno = tf1.accountno;
    SELECT 1;
$$ LANGUAGE SQL;
```

以下のように、ユーザはこの関数を使用して、口座番号17から100ドルを引き出すことが可能です。

```
SELECT tf1(17, 100.0);
```

この例では、第一引数の名前に`accountno`を選びましたが、これは`bank`テーブルの列の名前と同じです。`UPDATE`コマンドの中では、`accountno`は`bank.accountno`列を参照しますので、引数を参照するためには`tf1.accountno`を使用しなければなりません。もちろんこれは、引数に別の名前を使用することで防ぐことができます。

実際には、関数の結果を定数1よりもわかりやすい形にするために、以下のように定義するとよいでしょう。

```
CREATE FUNCTION tf1 (accountno integer, debit numeric) RETURNS integer AS $$
    UPDATE bank
        SET balance = balance - debit
        WHERE accountno = tf1.accountno;
    SELECT balance FROM bank WHERE accountno = tf1.accountno;
$$ LANGUAGE SQL;
```

これは残高を調整し、更新後の残高を返します。同じことは`RETURNING`を使用して1つのコマンドで行えます。

```
CREATE FUNCTION tf1 (accountno integer, debit numeric) RETURNS integer AS $$
    UPDATE bank
        SET balance = balance - debit
        WHERE accountno = tf1.accountno
    RETURNING balance;
$$ LANGUAGE SQL;
```

35.4.3. 複合型を使用するSQL関数

関数の引数に複合型を記述した場合、必要な引数だけを指定するだけではなく、必要とする引数の属性（フィールド）も指定する必要があります。例えば、`emp`が従業員データを持つテーブルとすると、この名前はそのテーブル内の各行を表す複合型の名前でもあります。以下に示す`double_salary`関数は、該当する従業員の給料が倍増したらどうなるかを計算します。

```
CREATE TABLE emp (  
    name          text,  
    salary        numeric,  
    age           integer,  
    cubicle       point  
);  
  
INSERT INTO emp VALUES ('Bill', 4200, 45, '(2,1)');  
  
CREATE FUNCTION double_salary(emp) RETURNS numeric AS $$  
    SELECT $1.salary * 2 AS salary;  
$$ LANGUAGE SQL;  
  
SELECT name, double_salary(emp.*) AS dream  
FROM emp  
WHERE emp.cubicle ~= point '(2,1)';  
  
name | dream  
-----+-----  
Bill | 8400
```

`$1.salary`という構文を使用して、引数の行値の1フィールドを選択していることに注目してください。また、`*`を使用した`SELECT`コマンドの呼び出しでは、複合型の値として、現在のテーブル行全体を表すテーブル名を使用していることにも注目してください。別の方法として、テーブル行は以下のようにテーブル名だけを使用して参照することができます。

```
SELECT name, double_salary(emp) AS dream  
FROM emp  
WHERE emp.cubicle ~= point '(2,1)';
```

しかし、この使用方法は混乱しやすいためお勧めしません。

その場で複合型の引数値を作成することが便利な場合があります。これは`ROW`式で行うことができます。例えば、以下のようにして関数に渡すデータを調整することができます。

```
SELECT name, double_salary(ROW(name, salary*1.1, age, cubicle)) AS dream  
FROM emp;
```

複合型を返す関数を作成することもできます。以下に単一の`emp`行を返す関数の例を示します。

```
CREATE FUNCTION new_emp() RETURNS emp AS $$  
    SELECT text 'None' AS name,  
           1000.0 AS salary,  
           25 AS age,  
           point '(2,2)' AS cubicle;  
$$ LANGUAGE SQL;
```

ここでは、各属性を定数で指定していますが、この定数を何らかの演算に置き換えることもできます。

関数を定義する上で、2つの重要な注意点を以下に示します。

- 問い合わせにおける選択リストの順番は、複合型と関連したテーブル内で現れる列の順番と正確に一致する必要があります。（上で行ったように列に名前を付けても、システムは認識しません。）
- 複合型定義と一致するように式を型キャストしなければなりません。さもなくば、以下のようなエラーとなります。

```
ERROR:  function declared to return emp returns varchar instead of text at column 1
```

同じ関数を以下のように定義することもできます。

```
CREATE FUNCTION new_emp() RETURNS emp AS $$  
    SELECT ROW('None', 1000.0, 25, '(2,2)')::emp;  
$$ LANGUAGE SQL;
```

ここで、正しい複合型の単一の列を単に返す`SELECT`を記述しました。今回の例ではこれはより優れたものとはいえませんが、例えば、必要な複合値を返す他の関数を呼び出して結果を計算しなければならない場合など、便利な解法になることがあります。

以下の2つの方法でこの関数を直接呼び出すことができます。

```
SELECT new_emp();  
  
new_emp  
-----  
(None,1000.0,25,"(2,2)")
```

```
SELECT * FROM new_emp();

name | salary | age | cubicle
-----+-----+-----+-----
None | 1000.0 | 25 | (2,2)
```

2番目の方法については、[項35.4.7](#)でより詳しく説明します。

複合型を返す関数を使用する時に、その結果から1つのフィールド（属性）のみを使用したいという場合があります。これは、以下のような構文で行うことができます。

```
SELECT (new_emp()).name;

name
-----
None
```

パーサが混乱しないように、括弧を追加する必要があります。括弧なしで行おうとすると、以下のような結果になります。

```
SELECT new_emp().name;
ERROR: syntax error at or near "."
LINE 1: SELECT new_emp().name;
                        ^
```

また、関数表記を使用して属性を抽出することもできます。簡単に説明すると、`attribute(table)`と`table.attribute`という表記方法のどちらでも使用できるということです。

```
SELECT name(new_emp());

name
-----
None

-- これは
-- SELECT emp.name AS youngster FROM emp WHERE emp.age < 30;
-- と同一です。

SELECT name(emp) AS youngster FROM emp WHERE age(emp) < 30;

youngster
-----
Sam
Andy
```

ティップ: 関数表記と属性表記の同等性によって、"計算処理されたフィールド"を模擬するために複合型に対する関数を使用することができます。例えば、上記で定義された`double_salary(emp)`を使用して、次のように記述することができます。

```
SELECT emp.name, emp.double_salary FROM emp;
```

これを利用するアプリケーションは、`double_salary`が実際にはテーブルの列ではないことを直接気にする必要はありません。（また、ビューを用いて計算処理されたフィールドを模擬することも可能です。）

このように動作しますので、単一の複合型の引数を取る関数に、その複合型のいずれかのフィールドの名前と同じ名前を付けることはお奨めしません。

複合型を結果として返す関数を使用する他の方法は、その結果を、その行型を入力として受け付ける関数に渡す、以下のような方法です。

```
CREATE FUNCTION getname(emp) RETURNS text AS $$
    SELECT $1.name;
$$ LANGUAGE SQL;

SELECT getname(new_emp());
getname
-----
None
(1 row)
```

複合型の結果を返す関数の使用方法として他にも、テーブル関数として呼び出す方法があります。これは[項35.4.7](#)で説明します。

35.4.4. 出力パラメータを持つSQL関数

関数の結果の記述方法には、他にも出力パラメータを使用して定義する方法があります。以下に例を示します。

```
CREATE FUNCTION add_em (IN x int, IN y int, OUT sum int)
AS 'SELECT x + y'
```

```
LANGUAGE SQL;

SELECT add_em(3,7);
   add_em
-----
      10
(1 row)
```

[項35.4.2](#)で示したadd_em版と基本的な違いはありません。 複数列を返す関数を定義する簡単な方法を提供することが出力パラメータの本来の価値です。 以下に例を示します。

```
CREATE FUNCTION sum_n_product (x int, y int, OUT sum int, OUT product int)
AS 'SELECT x + y, x * y'
LANGUAGE SQL;

SELECT * FROM sum_n_product(11,42);
   sum | product
-----+-----
    53 |     462
(1 row)
```

これは基本的に、関数結果用の無名の複合型の作成を行います。 上の例では、

```
CREATE TYPE sum_prod AS (sum int, product int);

CREATE FUNCTION sum_n_product (int, int) RETURNS sum_prod
AS 'SELECT $1 + $2, $1 * $2'
LANGUAGE SQL;
```

と同じ最終結果になります。 しかし、独立した複合型定義に悩まされることがなくなり、便利であるともいえます。 出力パラメータに割り振られた名前が単なる飾りではなく、無名複合型の列名を決定するものであることに注意してください。（出力パラメータの名前を省略した場合、システム自身が名前を選びます。）

SQLからこうした関数を呼び出す時、出力パラメータが呼び出し側の引数リストに含まれないことに注意してください。 PostgreSQLでは入力パラメータのみが関数の呼び出しシグネチャを定義するとみなしているためです。 これはまた、関数を削除することなどを目的に関数を参照する場合、入力パラメータのみが考慮されることを意味しています。 上の関数は、次のいずれかの方法で削除することができます。

```
DROP FUNCTION sum_n_product (x int, y int, OUT sum int, OUT product int);
DROP FUNCTION sum_n_product (int, int);
```

パラメータには、IN（デフォルト）、OUT、INOUT、またはVARIADICという印を付与できます。 INOUTパラメータは、入力パラメータ（呼び出し引数リストの一部）と出力パラメータ（結果のレコード型の一部）の両方を提供します。 VARIADICパラメータは入力パラメータですが、次に説明するように特別に扱われます。

35.4.5. 可変長引数を取るSQL関数

すべての"オプションの"引数が同じデータ型の場合、SQL関数は可変長の引数を受け付けるように宣言できます。 オプションの引数は配列として関数に渡されます。 この関数は最後のパラメータをVARIADICと印を付けて宣言されます。 このパラメータは配列型であるとして宣言されなければなりません。 例をあげます。

```
CREATE FUNCTION mleast(VARIADIC arr numeric[]) RETURNS numeric AS $$
    SELECT min($1[i]) FROM generate_subscripts($1, 1) g(i);
$$ LANGUAGE SQL;

SELECT mleast(10, -1, 5, 4.4);
   mleast
-----
      -1
(1 row)
```

実際、VARIADICの位置以降の実引数はすべて、あたかも以下のように記述したかのように、1次元の配列としてまとめられます。

```
SELECT mleast(ARRAY[10, -1, 5, 4.4]);      -- 動作しません
```

しかし、実際にこのように記述することはできません。 少なくとも、この関数定義に一致しません。 VARIADIC印の付いたパラメータは、自身の型ではなく、その要素型が1つ以上存在することに一致します。

時として、variadic関数に既に構築された配列を渡せることは有用です。 1つのvariadic関数が、自身の配列パラメータを他のものに渡したいとき特に便利です。 呼び出しにVARIADICを指定することで行えます。

```
SELECT mleast(VARIADIC ARRAY[10, -1, 5, 4.4]);
```

これは関数のvariadicパラメータがその要素型に拡張するのを防ぎます。 その結果、配列引数値が標準的にマッチされるようになります。 VARIADICは関数呼び出しの最後の実引数としてのみ付加できます。

variadicパラメータから生成される配列要素パラメータは、それ自身にはまったく名前を持たないものとして扱われます。これは、名前付き引数（[項4.3](#)）を使用して可変長の関数を呼び出すことができないことを意味します。ただし、VARIADICを指定する場合は例外です。たとえば、

```
SELECT mleast(VARIADIC arr := ARRAY[10, -1, 5, 4.4]);
```

は動作しますが、

```
SELECT mleast(arr := 10);
SELECT mleast(arr := ARRAY[10, -1, 5, 4.4]);
```

次は動作しません。

35.4.6. 引数にデフォルト値を持つSQL関数

一部またはすべての入力引数にデフォルト値を持つ関数を宣言することができます。デフォルト値は、関数が実際の引数の数に足りない数の引数で呼び出された場合に挿入されます。引数は実引数リストの終端から省略することができますので、デフォルト値を持つパラメータの後にあるパラメータはすべて、同様にデフォルト値を持たなければなりません。（名前付きの引数記法を使用してこの制限を緩和させることもできますが、まだ位置引数記法が実用的に動作できることが強制されています。）

以下に例を示します。

```
CREATE FUNCTION foo(a int, b int DEFAULT 2, c int DEFAULT 3)
RETURNS int
LANGUAGE SQL
AS $$
    SELECT $1 + $2 + $3;
$$;

SELECT foo(10, 20, 30);
   foo
-----
    60
(1 row)

SELECT foo(10, 20);
   foo
-----
    33
(1 row)

SELECT foo(10);
   foo
-----
    15
(1 row)

SELECT foo(); -- fails since there is no default for the first argument
ERROR:  function foo() does not exist
```

=記号をDEFAULTキーワードの代わりに使用することもできます。

35.4.7. テーブルソースとしてのSQL関数

すべてのSQL関数は問い合わせのFROM句で使用できますが、複合型を返す関数に特に便利です。関数が基本型を返すよう定義されている場合、テーブル関数は1列からなるテーブルを作成します。関数が複合型を返すよう定義されている場合、テーブル関数は複合型の列のそれぞれに対して1つの列を作成します。

以下に例を示します。

```
CREATE TABLE foo (fooid int, foosubid int, fooname text);
INSERT INTO foo VALUES (1, 1, 'Joe');
INSERT INTO foo VALUES (1, 2, 'Ed');
INSERT INTO foo VALUES (2, 1, 'Mary');

CREATE FUNCTION getfoo(int) RETURNS foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT *, upper(fooname) FROM getfoo(1) AS t1;

   fooid | foosubid | fooname | upper
-----+-----+-----+-----
      1 |         1 | Joe     | JOE
(1 row)
```

例からわかる通り、関数の結果の列を通常のテーブルの列と同じように扱うことができます。

この関数の結果得られたのは1行のみであることに注意してください。これは`SETOF`を指定しなかったためです。これについては次節で説明します。

35.4.8. 集合を返すSQL関数

SQL関数が`SETOF sometype`を返すよう宣言されている場合、関数の最後の問い合わせは最後まで実行され、各出力行は結果集合の要素として返されます。

この機能は通常、関数を`FROM`句内で呼び出す時に使用されます。この場合、関数によって返される各行は、問い合わせによって見えるテーブルの行になります。例えば、テーブル`foo`の内容が上記と同じであれば以下ようになります。

```
CREATE FUNCTION getfoo(int) RETURNS SETOF foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT * FROM getfoo(1) AS t1;
```

この出力は以下の通りです。

fooid	foosubid	fooname
1	1	Joe
1	2	Ed

(2 rows)

また、以下のように出力パラメータで定義された列を持つ複数の行を返すことも可能です。

```
CREATE TABLE tab (y int, z int);
INSERT INTO tab VALUES (1, 2), (3, 4), (5, 6), (7, 8);

CREATE FUNCTION sum_n_product_with_tab (x int, OUT sum int, OUT product int)
RETURNS SETOF record
AS $$
    SELECT $1 + tab.y, $1 * tab.y FROM tab;
$$ LANGUAGE SQL;

SELECT * FROM sum_n_product_with_tab(10);
sum | product
-----+-----
11  | 10
13  | 30
15  | 50
17  | 70
(4 rows)
```

ここで重要な点は、関数が1行だけではなく複数行を返すことを示すために`RETURNS SETOF record`を記述しなければならない点です。出力パラメータが1つしか存在しない場合は、`record`ではなく、そのパラメータの型を記述してください。

現在、集合を返す関数は問い合わせの選択リスト内でも呼び出すことができます。問い合わせ自身によって生成する各行に対し、集合を返す関数が呼び出され、関数の結果集合の各要素に対して出力行が生成されます。ただし、この機能は現在では推奨されておらず、今後のリリースでは削除される可能性があります。以下は、選択リストから集合を返す関数の例です。

```
CREATE FUNCTION listchildren(text) RETURNS SETOF text AS $$
    SELECT name FROM nodes WHERE parent = $1
$$ LANGUAGE SQL;

SELECT * FROM nodes;
name | parent
-----+-----
Top  |
Child1 | Top
Child2 | Top
Child3 | Top
SubChild1 | Child1
SubChild2 | Child1
(6 rows)

SELECT listchildren('Top');
listchildren
-----
Child1
Child2
Child3
(3 rows)

SELECT name, listchildren(name) FROM nodes;
name | listchildren
```

```

-----+-----
Top      | Child1
Top      | Child2
Top      | Child3
Child1   | SubChild1
Child1   | SubChild2
(5 rows)

```

最後のSELECTにおいて、Child2とChild3などが出力行に表示されていないことに注意してください。これは、listchildrenがこの入力に対して空の集合を返すため出力行が生成されないからです。

注意: もし関数の最後のコマンドがRETURNINGを持つINSERT、UPDATE、またはDELETEである場合、関数がSETOF付きで宣言されていない、または呼び出す問い合わせがすべての結果行を取り出さなくても、そのコマンドは完了まで実行されます。RETURNING句で生成される余計な行はすべて警告無しに削除されますが、コマンド対象のテーブルの変更はそれでも起こります（そして、関数から戻る前にすべて完了します）。

35.4.9. TABLEを返すSQL関数

集合を返すものとして関数を宣言するには、他にも方法があります。RETURNS TABLE(columns)構文を使用することです。これは1つ以上のOUTパラメータを使い、さらに、関数をSETOF record（または、適切ならば単一の出力パラメータの型のSETOF）を返すものと印を付けることと等価です。この記法は標準SQLの最近の版で規定されたものですので、SETOFを使用するより移植性がより高いかもしれません。

例えば前述の合計と積の例はこのように書けます。

```

CREATE FUNCTION sum_n_product_with_tab (x int)
RETURNS TABLE(sum int, product int) AS $$
    SELECT $1 + tab.y, $1 * tab.y FROM tab;
$$ LANGUAGE SQL;

```

RETURNS TABLE記法と一緒に、明示的OUTまたはINOUTパラメータを使用することはできません。すべての出力列をTABLEリストに含めなければなりません。

35.4.10. 多様SQL関数

SQL関数は、多様型anyelement、anyarray、anynonarray、anyenumおよびanyrangeを受け付け、返すように宣言することができます。多様関数の詳細説明については[項35.2.5](#)を参照してください。以下のmake_array多様関数は、任意の2つのデータ型要素から配列を作成します。

```

CREATE FUNCTION make_array(anyelement, anyelement) RETURNS anyarray AS $$
    SELECT ARRAY[$1, $2];
$$ LANGUAGE SQL;

SELECT make_array(1, 2) AS intarray, make_array('a'::text, 'b') AS textarray;
 intarray | textarray
-----+-----
 {1,2}   | {a,b}
(1 row)

```

'a'::textという型キャストを使用して、引数がtext型であることを指定していることに注目してください。これは引数が単なる文字列リテラルである場合に必要です。さもないと、unknown型として扱われてしまうため、無効なunknownの配列を返そうとしてしまいます。型キャストがないと、以下のようなエラーが発生します。

```

ERROR:  could not determine polymorphic type because input has type "unknown"

```

固定の戻り値型を持ちながら多様引数を持つことは許されますが、逆は許されません。以下に例を示します。

```

CREATE FUNCTION is_greater(anyelement, anyelement) RETURNS boolean AS $$
    SELECT $1 > $2;
$$ LANGUAGE SQL;

SELECT is_greater(1, 2);
 is_greater
-----
 f
(1 row)

CREATE FUNCTION invalid_func() RETURNS anyelement AS $$
    SELECT 1;
$$ LANGUAGE SQL;
ERROR:  cannot determine result data type
DETAIL:  A function returning a polymorphic type must have at least one polymorphic argument.

```

出力引数を持つ関数でも多様性を使用することができます。以下に例を示します。

```
CREATE FUNCTION dup (f1 anyelement, OUT f2 anyelement, OUT f3 anyarray)
AS 'select $1, array[$1,$1]' LANGUAGE SQL;

SELECT * FROM dup(22);
  f2 |      f3
-----+-----
  22 | {22,22}
(1 row)
```

多様性はvariadic関数とともに使用できます。例をあげます。

```
CREATE FUNCTION anyleast (VARIADIC anyarray) RETURNS anyelement AS $$
    SELECT min($1[i]) FROM generate_subscripts($1, 1) g(i);
$$ LANGUAGE SQL;

SELECT anyleast(10, -1, 5, 4);
  anyleast
-----
        -1
(1 row)

SELECT anyleast('abc'::text, 'def');
  anyleast
-----
      abc
(1 row)

CREATE FUNCTION concat_values(text, VARIADIC anyarray) RETURNS text AS $$
    SELECT array_to_string($2, $1);
$$ LANGUAGE SQL;

SELECT concat_values('|', 1, 4, 2);
  concat_values
-----
      1|4|2
(1 row)
```

35.4.11. 照合順序を持つSQL関数

SQL関数が照合順序の変更が可能なデータ型のパラメータを1つ以上持つ場合、それぞれの関数呼び出しに対して、実引数に割り当てられた照合順序に応じて、照合順序が識別されます。照合順序の識別に成功した（つまり、暗黙的な照合順序がすべての引数で競合しない）場合、すべての照合順序の変更が可能なパラメータは暗黙的に照合順序を持つものとして扱われます。これは関数内の照合順序に依存する操作の振舞いに影響します。例えば、上記の`anyleast`を使って考えます。

```
SELECT anyleast('abc'::text, 'ABC');
```

この結果はデータベースのデフォルト照合順序に依存します。cロケールでは`ABC`という結果になりますが、他の多くのロケールでは`abc`になります。使用される照合順序を`COLLATE`句を付与することで強制することができます。例を以下に示します。

```
SELECT anyleast('abc'::text, 'ABC' COLLATE "C");
```

この他、呼び出し元の照合順序とは関係なく特定の照合順序で動作する関数にしたければ、関数定義において必要な所に`COLLATE`句を付けてください。以下の`anyleast`では、文字列を比較する際に常に`en_US`を使用します。

```
CREATE FUNCTION anyleast (VARIADIC anyarray) RETURNS anyelement AS $$
    SELECT min($1[i] COLLATE "en_US") FROM generate_subscripts($1, 1) g(i);
$$ LANGUAGE SQL;
```

しかし、もし照合順序の変更ができないデータ型が与えられた場合にエラーになってしまうことに注意してください。

実引数全体で共通の照合順序を識別できない場合、SQL関数はパラメータがそのデータ型のデフォルト照合順序（通常はデータベースのデフォルトの照合順序ですが、ドメイン型のパラメータでは異なる可能性があります）を持つものとみなします。

照合順序の変更ができるパラメータの動作は、テキストのデータ型にのみ適用できる、限定された多様性と考えることができません。

[前のページ](#)
ユーザ定義関数

[ホーム](#)
[上に戻る](#)

[次のページ](#)
関数のオーバーロード

35.5. 関数のオーバーロード

使用する引数が異なるのであれば、同じSQL名の関数を1つ以上定義することができます。つまり、関数名はオーバーロードが可能です。問い合わせが実行された時、サーバは与えられた引数のデータ型と数によって呼び出すべき関数を決定します。またオーバーロードを使用して、有限個の可変長引数を持つ関数を模擬することができます。

オーバーロード関数を作成する時、曖昧さが発生しないように注意しなければなりません。例えば、以下のような関数を考えてみます。

```
CREATE FUNCTION test(int, real) RETURNS ...
CREATE FUNCTION test(smallint, double precision) RETURNS ...
```

`test(1, 1.5)`のような平凡な入力でも、どちらの関数を呼び出すのかはすぐには明確ではありません。現在実装されている解決規則は[第10章](#)にて説明していますが、この動作に巧妙に依存するようにシステムを設計することは推奨しません。

一般的に、1つの複合型の引数を取る関数は、その型の属性（フィールド）と同じ名前を持つてはいけません。

`attribute(table)`が`table.attribute`と等価とみなされることを思い出してください。複合型に対する関数と複合型の属性との間に曖昧さがあるような場合、属性の方が常に使用されます。この振舞いは関数名をスキーマで修飾する（つまり、`schema.func(table)`）ことにより変更することができますが、競合する名前を使用しないことで問題を防ぐ方が良いでしょう。

可変長引数を取る関数と可変長引数を取らない関数の間に、他にも競合する可能性があります。例えば、`foo(numeric)`と`foo(VARIADIC numeric[])`の両方を作成することが可能です。この場合、単一の数値引数を取った呼び出し、例えば`foo(10.1)`をどちらに一致するものとすべきか不明瞭です。検索パスのより前にある関数が使われる、もし2つの関数が同一スキーマにあれば可変長引数を取らない関数が優先されるというのが、この場合の規則です。

C言語関数をオーバーロードする場合、さらに制限があります。オーバーロードされた関数群内の各関数のCの名前は、内部的に動的ロード可能かに関係なく他のすべての関数のCの名前と異なる必要があります。この規則に反した場合は、この動作は移植性がありません。実行時リンクエラーになるかもしれませんし、関数群のどれか（たいていは内部関数）が呼び出されるかもしれません。CREATE FUNCTION SQLコマンドの別形式の`AS`句は、SQL関数名とCソースコード内の関数名とを分離します。以下に例を示します。

```
CREATE FUNCTION test(int) RETURNS int
AS 'filename', 'test_1arg'
LANGUAGE C;
CREATE FUNCTION test(int, int) RETURNS int
AS 'filename', 'test_2arg'
LANGUAGE C;
```

ここでのC関数の名前は多くの取り得る規約の1つを反映しています。

35.6. 関数の変動性分類

すべての関数は変動性区分を持ちます。取り得る区分は、VOLATILE、STABLE、もしくはIMMUTABLEです。[CREATE FUNCTION](#) コマンドで分類の指定がなければデフォルトでVOLATILEになります。変動性に関する分類は、その関数の動作に関するオプティマイザへの約束事です。

- VOLATILE関数は、データベースの変更を含む、すべてを行うことができます。同一引数で続けて呼び出したとしても異なる結果を返すことができます。オプティマイザはこうした関数の振舞いに対する前提を持ちません。VOLATILE関数を使用した問い合わせは、その行の値を必要とするすべての行においてその関数を再評価します。
- STABLE関数はデータベースを変更することができません。また、単一の文内ですべての行に対して同一の引数を渡した場合に同一の結果を返すことが保証されています。この区分により、オプティマイザは複数の関数の呼び出しを1つの呼び出しに最適化することができます。特に、インデックススキャン条件内でこうした関数を含んだ式を使用することは安全です。（インデックススキャンは行ごとに一度ではなく、一度だけ比較値の評価を行いますので、インデックススキャン条件内でVOLATILE関数を使用することは意味がありません。）
- IMMUTABLE関数はデータベースを変更することができません。また、同一引数に対する呼び出しは常に同一の結果を返すことが保証されています。問い合わせが定数の引数でこうした関数を呼び出した場合、オプティマイザはこの関数を事前に評価することができます。例えば、`SELECT ... WHERE x = 2 + 2` といった問い合わせは、`SELECT ... WHERE x = 4` のように単純化することができます。これは、整数加算演算子の基になる関数がIMMUTABLEとして宣言されているためです。

最適化の結果を最善にするためには、関数に対して有効かつ最も厳密な変動性区分を付けなければなりません。

副作用を持つ関数はすべてVOLATILEと付けなければなりません。こうした関数は最適化することができないためです。関数が副作用を持たなかったとしても、単一問い合わせ内で値が変動する場合はVOLATILEと付けなければなりません。例えば、`random()`、`currval()`、`timeofday()` などです。

その他の重要な例は、`current_timestamp` 系列の関数は、それらの値がトランザクション内で変わらないことから、STABLEと見なされます。

計画作成を行い、すぐに実行されるような単一の対話式問い合わせを考えた場合、相対的にSTABLE区分とIMMUTABLE区分との違いはあまりありません。このような場合、関数が計画作成中に一度実行されるか、問い合わせ実行中に一度実行されるかがあまり問題になりません。しかし、計画が保存され、後で再利用される場合は大きな違いが現れます。本来ならば関数が計画作成段階で早めに定数を保持することができない場合にIMMUTABLEを付けると、その後この計画を使用する時に古くて意味のない値が再利用されてしまうことになります。これは、プリペアド文や計画をキャッシュする関数言語（PL/pgSQLなど）を使用する場合は危険です。

SQLもしくは標準手続き言語で作成された関数では、変動性分類で決定される2番目に重要な性質があります。すなわち、その関数を呼び出すSQLコマンドによりなされてきたすべてのデータ変更の可視性です。VOLATILE関数はそのような変更を捕らえますが、STABLEまたはIMMUTABLE関数はそうしません。この動作はMVCC（[第13章](#)を参照）のスナップショット処理の動作を使用して実装されています。STABLEとIMMUTABLE関数は、呼び出す問い合わせの開始時点で成立したスナップショットを使用しますが、VOLATILE関数はそれぞれの問い合わせの実行開始時点の作りたてのスナップショットを取得します。

注意: しかし、C言語で作成された関数は、どのようにでもスナップショットを管理することができますが、通常C関数でもこのように動作させることは良い考えです。

このスナップショット処理の動作のため、同時実行の問い合わせによって別途変更されている可能性があるテーブルから選択していたとしても、SELECTコマンドのみを含む関数は、安全にSTABLEとすることができます。PostgreSQLは、呼び出し元の問い合わせに対して確立されたスナップショットを使用してSTABLE関数のすべてのコマンドを実行します。したがってその問い合わせの間、データベースに対して固定された視点で値を参照することになります。

IMMUTABLE関数内のSELECTコマンドも同様のスナップショット処理の動作を使用します。ただし、一般的に、IMMUTABLE関数内でデータベースのテーブルを検索（SELECT）することは勧められません。テーブルの内容が変わってしまった場合にその普遍性が壊れてしまうためです。しかし、PostgreSQLでは強制的に検索（SELECT）できないようにはしていません。

よくあるエラーは、設定パラメータに依存する結果となる関数にIMMUTABLEと付けることです。例えば、タイムスタンプを操作する関数は、おそらく[TimeZone](#)の設定に依存した結果になります。こうした関数は、安全のため代わりにSTABLEと付けてください。

注意: PostgreSQLリリース8.0より前のリリースでは、STABLE関数とIMMUTABLE関数がデータベースを変更できないという必要条件がシステムによって禁止されていませんでした。リリース8.0以降では、この区分のSQL関数や手続き言語関数に対してSELECT以外のSQLコマンドを含めないことを必要条件とすることにより、これを禁止して

います。（こうした関数はまだデータベースを変更するVOLATILE関数を呼び出すことができますので、これは防弾条件として完全ではありません。 これを行うと、STABLEもしくはIMMUTABLE関数は、そのスナップショットからそれらが隠されていることから、呼び出した関数によるデータベースの変更に気がつきません。）

[前のページ](#)[関数のオーバーロード](#)[ホーム](#)[上に戻る](#)[次のページ](#)[手続き型言語関数](#)

35.7. 手続き型言語関数

PostgreSQL では SQL や C 言語以外の言語でユーザ定義の関数を作成することができます。これらの他の言語は一般に手続き言語 (PL) と呼ばれます。手続き言語は PostgreSQL サーバ に組み込まれておらず、ロード可能モジュールとして提供されています。詳細は [第38章](#) と以下の章を参照してください。

35.8. 内部関数

内部関数とは、C で作成された、PostgreSQL サーバに静的にリンクされた関数です。関数定義の "本体" では関数の C 言語における名前を指定します。この名前を SQL での使用のために宣言される名前と同じにする必要はありません。（後方互換性のため、C 言語関数名が SQL 名と同じであるという意味として、空の本体も受け付けられます。）

通常、サーバに存在するすべての内部関数は、データベースクラスタの初期化（[項17.2](#) 参照）の際に宣言されますが、ユーザは `CREATE FUNCTION` を使用して、内部関数の別名をさらに作成することができます。内部関数は `internal` という言語名を付けた `CREATE FUNCTION` によって宣言されます。例えば、`sqrt` 関数の別名を作成するには以下のようにします。

```
CREATE FUNCTION square_root(double precision) RETURNS double precision
AS 'dsqrt'
LANGUAGE internal
STRICT;
```

（ほとんどの内部関数は "strict" として宣言されることを想定しています。）

注意: "定義済みの" 関数のすべてが上の意味での "内部" ではありません。SQL で作成された定義済み関数もあります。

35.9. C言語関数

ユーザ定義の関数はC（もしくはC++のようなCと互換性のある言語）で作成することができます。そのような関数は動的ロード可能オブジェクト（共有ライブラリとも呼ばれます）としてコンパイルされ、必要に応じてサーバにロードされます。動的ロード機能が、“C言語”関数を“内部”関数と区別するものです。コーディング方法は基本的に両方とも同じです。（したがって、標準内部関数ライブラリはユーザ定義のC関数のコーディング例の豊富な情報源となります。）

現在、2つの異なる呼び出し規約がC関数で使用されています。より新しい“Version 1”呼び出し規約は、以下に示すように、その関数用に呼び出しマクロPG_FUNCTION_INFO_V1()を書くことで示されます。このマクロが存在しなければ、旧形式 (“Version 0”) の関数であることを示します。どちらの場合もCREATE FUNCTIONで指定する言語名はcです。旧形式の関数は移植性の問題と機能の不足のために勧められません。これは現在、互換性の理由のために存在しています。

35.9.1. 動的ロード

特定のロード可能オブジェクト内のユーザ定義の関数がセッションで最初に呼び出されると、動的ローダは、その関数を呼び出すことができるように、オブジェクトファイルをメモリ内に読み込みます。そのため、ユーザ定義のC関数用のCREATE FUNCTIONはその関数について、ロード可能オブジェクトファイルの名前とオブジェクトファイル中の呼び出される特定の関数のC名称（リンクシンボル）という2つの情報を指定しなければなりません。C名称が明示的に指定されなかった場合、SQLにおける関数名と同じものと仮定されます。

CREATE FUNCTIONコマンドで与えられた名前に基づいて、共有オブジェクトファイルの場所を見つける際に以下のアルゴリズムが使用されます。

- 1 11 名前が絶対パスの場合、指定されたファイルが読み込まれます。
- 1 11 名前が\$libdirという文字列から始まる場合、その部分はPostgreSQLパッケージのライブラリディレクトリで置き換えられます。このディレクトリはビルド時に決定されます。
- 1 11 名前にディレクトリ部分がない場合、そのファイルはdynamic_library_path設定変数で指定されたパス内から検索されます。
- 1 11 上記以外の場合（ファイルがパス内に存在しない場合や相対ディレクトリ部分を持つ場合）、動的ローダは指定された名前をそのまま使用し、ほとんどの場合は失敗します。（これは現在の作業ディレクトリに依存するため信頼できません。）

ここまでの流れがうまくいかなかった場合、プラットフォーム独自の共有ライブラリファイル拡張子（多くの場合.so）が指定された名前に追加され、再度この流れを試みます。同様に失敗した場合は、読み込みは失敗します。

共有ライブラリを\$libdirから相対的に、もしくは動的ライブラリパスの通った所に配置することを推奨します。異なる場所に新しいインストールを配置する場合にバージョンアップを簡単にします。\$libdirが示す実際のディレクトリはpg_config --pkglibdirコマンドを使用することでわかります。

PostgreSQLサーバの実効ユーザIDはロード予定のファイルのパスまで到達できなければなりません。よくある失敗として、postgresユーザに対して読み込み、実行、または両方の権限がそのファイルとその上位ディレクトリに与えられていないことがあります。

どの場合でも、CREATE FUNCTIONコマンドに与えたファイル名はそのままシステムカタログに保存されます。ですので、もしそのファイルを再度読み込む必要がある場合、同じ処理が適用されます。

注意: PostgreSQLはC関数を自動的にコンパイルしません。CREATE FUNCTIONコマンドで参照する前に、そのオブジェクトファイルはコンパイルされていなければなりません。さらなる情報については[項35.9.6](#)を参照してください。

確実に、動的にロードされるモジュールが互換性がないサーバにロードされないように、PostgreSQLは、そのファイルに適切な内容を持つ“魔法のブロック”が含まれているかどうか検査します。これによりサーバは、メジャーバージョンが異なるPostgreSQL用にコンパイルされたモジュールなど、明確に互換性がないことを検知することができます。魔法のブロックはPostgreSQL 8.2から要求されています。魔法のブロックを含めるためには、以下をモジュールのソースファイルに一度（一度だけ）、fmgr.hヘッダファイルをincludeさせた後で、記述してください。

```
#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif
```

そのコードをリリース8.2より前のPostgreSQL用にコンパイルする必要がなければ、#ifdefテストを省略することができます。

最初に使用された後も、動的にロードされたオブジェクトファイルはメモリ内に保持されます。 同一セッションにおいてそのファイル内の関数をその後に呼び出した場合、シンボルテーブルの検索に要する小さなオーバーヘッドしかかかりません。 例えば再コンパイルした後など、そのオブジェクトファイルを強制的に再度読み込ませる必要がある場合は、新しいセッションを開始してください。

省略することもできますが、動的にロードされるファイルに初期化処理関数と最終処理関数を含めることができます。 `_PG_init` という関数がファイルに存在すると、この関数はファイルがロードされた直後に呼び出されます。 この関数は引数を取らず `void` 型を返さなければなりません。 `_PG_fini` という関数がファイルに存在すると、この関数はファイルがアンロードされる直前に呼び出されます。 この関数も同様に引数を取らず `void` 型を返さなければなりません。 `_PG_fini` がファイルのアンロード時にのみ呼び出されるものであり、処理の終了時に呼び出されるものではないことに注意してください。（現在、アンロードは無効となっていますので、決して発生しません。将来変更される可能性があります。）

35.9.2. C言語関数における基本型

C言語関数の作成方法を理解するためには、PostgreSQLが基本データ型を内部でどのように表現し、どのようにそれらを関数とやり取りしているかを理解する必要があります。 内部的にPostgreSQLは基本型を"メモリの小さな塊"とみなします。 ある型を定義するユーザ定義関数は、言い換えると、PostgreSQLがそれを操作できる方法を定義します。 つまり、PostgreSQLはデータの格納、ディスクからの取り出しのみを行い、データの入力や処理、出力にはユーザ定義関数を使用します。

基本型は下記の3つのいずれかの内部書式を使用しています。

- 固定長の値渡し
- 固定長の参照渡し
- 可変長の参照渡し

値渡しは、1、2、4バイト長の型のみで使用することができます（使用するマシンの `sizeof(Datum)` が8の場合は8バイトも使用できます）。 データ型を定義する際、その型がすべてのアーキテクチャにおいて同一の大きさ（バイト数）となるように定義するように注意してください。 例えば、`long` 型はマシンによっては4バイトであったり、8バイトであったりして危険ですが、`int` 型はほとんどのUnixマシンでは4バイトです。 Unixマシンにおける `int4` の理論的な実装は以下のようになります。

```
/* 4 バイト整数、値渡し */
typedef int int4;
```

一方、任意の大きさの固定長の型は参照として引き渡すことができます。 例として以下にPostgreSQLの型の実装サンプルを示します。

```
/* 16 バイト構造体、参照渡し */
typedef struct
{
    double   x, y;
} Point;
```

それらの型のポインタのみがPostgreSQL関数の入出力時に使用できます。 それらの型の値を返すためには、`palloc()` を使用して正しい大きさのメモリ領域を割り当て、そのメモリ領域に値を入力し、そのポインタを返します。（また、入力引数の1つと同じ型かつ同じ値を返したいのであれば、`palloc`を行う手間を省くことができます。 この場合は入力値へのポインタを単に返してください。）

最後に、すべての可変長型は参照として引き渡す必要があります。 また、すべての可変長型は正確に4バイトの不透明な `length` フィールドから始まる必要があります。 このフィールドは `SET_VARSIZE` で設定されます。決して直接このフィールドを設定してはいけません。 その型に格納されるすべてのデータは `length` フィールドのすぐ後のメモリ領域に置かれる必要があります。 `length` フィールドにはその構造体の総長が格納されます。つまり、`length` フィールドそのものもその大きさに含まれます。

その他の重要な点は、データ型の値の中で初期化されていないビットを残さないことです。 例えば、構造体内に存在する可能性がある整列用のパディングバイトを注意してすべてゼロクリアしてください。 こうしないと、独自データ型の論理的に等価な定数がプランナにより一致しないものと判断され、（不正確ではありませんが）非効率的な計画をもたらすかもしれません。

警告

参照渡しの入力値の内容を決して変更しないでください。 指定したポインタがディスクバッファを直接指し示している可能性がよくありますので、変更すると、ディスク上のデータを破壊してしまうかもしれません。 この規則の唯一の例外について[項35.10](#)で説明します。

例えば、`text` 型を定義するには、下記のように行えます。

```
typedef struct {
```



```
int4 length;
char data[1];
} text;
```

ここで宣言されたdataフィールドは、明らかにすべての取り得る文字列を保持できる長さではありません。C言語では可変長の構造体を定義することは不可能ですので、Cコンパイラは配列の添字の範囲検査を行わないという事実に依存します。必要な領域量を割り当て、あたかも正しい長さで宣言されたかのように、配列としてアクセスするだけです。（この手法はよく使用されます。C言語に関する多くの書籍で説明されています。）

可変長型を操作する時、正確な大きさのメモリを割り当て、lengthフィールドを正確に設定することに注意する必要があります。例えば、40バイトをtext構造体に保持させたい場合、下記のようなコードを使用します。

```
#include "postgres.h"
...
char buffer[40]; /* our source data */
...
text *destination = (text *) palloc(VARHDRSZ + 40);
SET_VARSIZE(destination, VARHDRSZ + 40);
memcpy(destination->data, buffer, 40);
...
```

VARHDRSZはsizeof(int32)と同一ですが、可変長型のオーバーヘッド分の大きさを参照する時には、VARHDRSZマクロを使用する方が好ましい形式とみなされています。また長さフィールドを単なる代入ではなくSET_VARSIZEマクロを使用して設定しなければなりません。

表35-1は、PostgreSQLの組み込み型を使用するC言語関数を作成する時の、Cの型とSQL型との対応を規定したものです。"定義場所"列では、型定義を取り出すためにインクルードしなければならないヘッダファイルを示しています。（実際の定義は一覧中のファイルからインクルードされた、別のファイルであるかもしれません。ユーザは定義されたインタフェースを厳守することを推奨されています。）postgres.hには必ず必要になる多くのものが宣言されていますので、ソースファイルの中で必ず初めにこのファイルをインクルードしなければならないことに注意してください。

表 35-1. 組み込みSQL型に相当するCの型

SQL型	C 言語型	定義場所
abstime	AbsoluteTime	utils/nabstime.h
boolean	bool	postgres.h（コンパイラで組み込み済みの可能性があります）
box	BOX*	utils/geo_decls.h
bytea	bytea*	postgres.h
"char"	char	（コンパイラで組み込み済み）
character	BpChar*	postgres.h
cid	CommandId	postgres.h
date	DateADT	utils/date.h
smallint (int2)	int2 or int16	postgres.h
int2vector	int2vector*	postgres.h
integer (int4)	int4 or int32	postgres.h
real (float4)	float4*	postgres.h
double precision (float8)	float8*	postgres.h
interval	Interval*	datatype/timestamp.h
lseg	LSEG*	utils/geo_decls.h
name	Name	postgres.h
oid	Oid	postgres.h
oidvector	oidvector*	postgres.h
path	PATH*	utils/geo_decls.h
point	POINT*	utils/geo_decls.h
regproc	regproc	postgres.h
reltime	RelativeTime	utils/nabstime.h
text	text*	postgres.h
tid	ItemPointer	storage/itemptr.h
time	TimeADT	utils/date.h
time with time zone	TimeTzADT	utils/date.h
timestamp	Timestamp*	datatype/timestamp.h
tinterval	TimeInterval	utils/nabstime.h
varchar	VarChar*	postgres.h
xid	TransactionId	postgres.h

ここまでで基本型に関してあり得る構造体のすべてを記述しましたので、実際の関数の例をいくつか示すことができます。

35.9.3. Version 0 呼び出し規約

まず最初に、現在は非推奨ですが理解しやすいので、"古いスタイル"の呼び出し規約を説明します。Version-0メソッドでは、C関数の引数と結果は、通常のCのプログラムの記述の方法と同じような形式で行いますが、上記の説明のように、各SQLのデータ型に対するC言語での表現を注意して使用してください。

以下にいくつか例を示します。

```
#include "postgres.h"
#include <string.h>
#include "utils/geo_decls.h"

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

/* 値渡し */

int
add_one(int arg)
{
    return arg + 1;
}

/* 固定長の参照渡し */

float8 *
add_one_float8(float8 *arg)
{
    float8      *result = (float8 *) palloc(sizeof(float8));

    *result = *arg + 1.0;

    return result;
}

Point *
makepoint(Point *pointx, Point *pointy)
{
    Point      *new_point = (Point *) palloc(sizeof(Point));

    new_point->x = pointx->x;
    new_point->y = pointy->y;

    return new_point;
}

/* 可変長の参照渡し */

text *
copytext(text *t)
{
    /*
     * VARSIZEは構造体の総長をバイト数で表したものです。
     */
    text *new_t = (text *) palloc(VARSIZE(t));
    SET_VARSIZE(new_t, VARSIZE(t));
    /*
     * VARDATAは構造体のデータ領域へのポインタです。
     */
    memcpy((void *) VARDATA(new_t), /* コピー先 */
           (void *) VARDATA(t),    /* コピー元 */
           VARSIZE(t) - VARHDRSZ); /* バイト数 */
    return new_t;
}

text *
concat_text(text *arg1, text *arg2)
{
    int32 new_text_size = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
    text *new_text = (text *) palloc(new_text_size);

    SET_VARSIZE(new_text, new_text_size);
    memcpy(VARDATA(new_text), VARDATA(arg1), VARSIZE(arg1) - VARHDRSZ);
    memcpy(VARDATA(new_text) + (VARSIZE(arg1) - VARHDRSZ),
           VARDATA(arg2), VARSIZE(arg2) - VARHDRSZ);
    return new_text;
}
```

上のコードがfuncs.cというファイルに用意され、共有オブジェクトとしてコンパイル済みであるとする、以下のようなコマン

ドでPostgreSQLの関数を定義することができます。

```
CREATE FUNCTION add_one(integer) RETURNS integer
AS 'DIRECTORY/funcs', 'add_one'
LANGUAGE C STRICT;

-- "add_one"というSQL関数名をオーバーロードしていることに注意
CREATE FUNCTION add_one(double precision) RETURNS double precision
AS 'DIRECTORY/funcs', 'add_one_float8'
LANGUAGE C STRICT;

CREATE FUNCTION makepoint(point, point) RETURNS point
AS 'DIRECTORY/funcs', 'makepoint'
LANGUAGE C STRICT;

CREATE FUNCTION copytext(text) RETURNS text
AS 'DIRECTORY/funcs', 'copytext'
LANGUAGE C STRICT;

CREATE FUNCTION concat_text(text, text) RETURNS text
AS 'DIRECTORY/funcs', 'concat_text'
LANGUAGE C STRICT;
```

ここで、*DIRECTORY*は共有ライブラリファイルのディレクトリ（例えば、本節で使用する例のコードが含まれるPostgreSQLチュートリアルディレクトリ）を表します。（AS句中では単に'funcs'を使用し、後で*DIRECTORY*を検索パスに追加の方がより良い方法です。 どの場合でも、一般的に.soや.slが使用される、共有ライブラリ用のシステム独特の拡張子を省略することができます。）

ここで、関数を"厳密（strict）"と指定していることに注目してください。 これは、もし入力された値がNULLであった場合に、システムが自動的に返り値もNULLであるとみなすことを意味します。 これを行うことによって、関数のコードで入力値がNULLであるかどうかの検査を行う必要がなくなります。 これがなければ、参照渡し引数それぞれに対してヌルポインタについての検査を行うなど、NULL値の明示的な検査を行う必要性が出てきます。（値渡し引数に関しては、検査を行う方法は存在しません。）

この呼び出し規約は単純ですが、この方法は移植性の面であまり優れていません。 この方法でint型より小さいデータ型を渡す部分で問題を抱えているアーキテクチャも存在します。 また、関数の結果としてNULLを返す簡単な方法はありません。 その上、NULL引数をうまく処理する方法としては、関数を厳密なものにする以外方法はありません。 次に説明するVersion-1規約ではこれらの問題が解決されています。

35.9.4. Version 1 呼び出し規約

Version-1呼び出し規約では、引数と結果の引き渡しの複雑さをなくするためにマクロを使用しています。 Version-1関数のC言語宣言は必ず下記のように行います。

```
Datum funcname(PG_FUNCTION_ARGS)
```

さらに、マクロ呼び出し

```
PG_FUNCTION_INFO_V1(funcname);
```

が同じソースファイルに書かれている必要があります。（一般には、関数の直前に書かれます。） PostgreSQLではすべての内部関数はVersion-1であると認識するので、このマクロの呼び出しはinternal言語関数では必要ありません。 しかし、動的にロードされる関数では必要です。

Version-1関数では、それぞれの実引数は、引数のデータ型に合ったPG_GETARG_XXX() マクロを使用して取り出され、結果は戻り値の型に合ったPG_RETURN_XXX() マクロを使用して返されます。 PG_GETARG_XXX() は、その引数として、取り出す関数引数の番号（ゼロから始まります）を取ります。 PG_RETURN_XXX() は、その引数として、実際に返す値を取ります。

上記と同じ関数をVersion-1形式で記述したものを以下に示します。

```
#include "postgres.h"
#include <string.h>
#include "fmgr.h"
#include "utils/geo_decls.h"

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

/* 値渡し */

PG_FUNCTION_INFO_V1(add_one);

Datum
add_one(PG_FUNCTION_ARGS)
{
```

```

    int32    arg = PG_GETARG_INT32(0);

    PG_RETURN_INT32(arg + 1);
}

/* 固定長の参照渡し */

PG_FUNCTION_INFO_V1(add_one_float8);

Datum
add_one_float8(PG_FUNCTION_ARGS)
{
    /* FLOAT8用のマクロは参照渡しという性質を隠します */
    float8    arg = PG_GETARG_FLOAT8(0);

    PG_RETURN_FLOAT8(arg + 1.0);
}

PG_FUNCTION_INFO_V1(makepoint);

Datum
makepoint(PG_FUNCTION_ARGS)
{
    /* このPoint型の参照渡しという性質は隠されていません */
    Point      *pointx = PG_GETARG_POINT_P(0);
    Point      *pointy = PG_GETARG_POINT_P(1);
    Point      *new_point = (Point *) palloc(sizeof(Point));

    new_point->x = pointx->x;
    new_point->y = pointy->y;

    PG_RETURN_POINT_P(new_point);
}

/* 可変長の参照渡し */

PG_FUNCTION_INFO_V1(copytext);

Datum
copytext(PG_FUNCTION_ARGS)
{
    text      *t = PG_GETARG_TEXT_P(0);
    /*
     * VARSIZEは構造体の総長をバイト数で表したものです。
     */
    text      *new_t = (text *) palloc(VARSIZE(t));
    SET_VARSIZE(new_t, VARSIZE(t));
    /*
     * VARDATAは構造体のデータ領域へのポインタです。
     */
    memcpy((void *) VARDATA(new_t), /* コピー先 */
           (void *) VARDATA(t), /* コピー元 */
           VARSIZE(t) - VARHDRSZ); /* バイト数 */
    PG_RETURN_TEXT_P(new_t);
}

PG_FUNCTION_INFO_V1(concat_text);

Datum
concat_text(PG_FUNCTION_ARGS)
{
    text      *arg1 = PG_GETARG_TEXT_P(0);
    text      *arg2 = PG_GETARG_TEXT_P(1);
    int32 new_text_size = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
    text      *new_text = (text *) palloc(new_text_size);

    SET_VARSIZE(new_text, new_text_size);
    memcpy(VARDATA(new_text), VARDATA(arg1), VARSIZE(arg1) - VARHDRSZ);
    memcpy(VARDATA(new_text) + (VARSIZE(arg1) - VARHDRSZ),
           VARDATA(arg2), VARSIZE(arg2) - VARHDRSZ);
    PG_RETURN_TEXT_P(new_text);
}

```

CREATE FUNCTION コマンドはVersion-0と同じものです。

一見Version-1のコーディング規約は無意味なものに見えるかもしれませんが。しかし、マクロが必要のない情報を隠蔽しているので、多数の改良が行われています。例えば、`add_one_float8`のコードでは、`float8`が参照渡しであることを意識する必要がなくなっています。また別の例としては、可変長型の`GETARG`マクロは"TOASTされた"（圧縮または行外）値をより効率的に取り出すことができます。

Version-1関数の1つの大きな改善点は、NULLの入力/結果の処理能力です。 `PG_ARGISNULL(n)` マクロにより関数は各入力 がNULLであるかどうかの検査を行うことができます。（もちろんこれは、"厳密"と宣言されていない関数でのみ必要です。） `PG_GETARG_xxx()` マクロと同様、入力引数の番号はゼロから始まります。 引数がNULLでないことを確認するまで

は、`PG_GETARG_xxx()`の実行は控えなければなりません。結果としてNULLを返す場合は、`PG_RETURN_NULL()`を実行します。これは、厳密な関数と厳密でない関数の両方で使用可能です。

新しい形式のインタフェースでは、その他のオプションとして`PG_GETARG_xxx()`マクロの変形を2つ提供しています。1つ目の`PG_GETARG_xxx_COPY()`によって、安全に書き込むことができる指定引数のコピーが確実に返されます。（通常のマクロは、物理的にテーブルに格納されている値へのポインタを返すことがあるので、書き込んではいけません。`PG_GETARG_xxx_COPY()`マクロの結果は書き込み可能であることが保証されています。）2つ目の変形は、引数を3つ取る`PG_GETARG_xxx_SLICE()`マクロからなります。1つ目は関数の引数の番号（上記の通り）です。2つ目と3つ目は、オフセットと返されるセグメントの長さです。オフセットはゼロから始まり、負の長さは残りの値を返すことを要求します。これらのマクロを使用すると、ストレージ種類が"external"（外部）である大きな値の一部へアクセスする際に非常に効果的です。（列のストレージ種類は`ALTER TABLE tablename ALTER COLUMN colname SET STORAGE storagetype`を使用して指定できます。`storagetype`は、`plain`、`external`、`extended`、または`main`のいずれかです。）

最後に、Version-1関数呼び出し規約では、結果集合（[項35.9.9](#)）を返すこと、およびトリガ関数（[第36章](#)）と手続型言語の呼び出しハンドラ（[第49章](#)）を実装することができます。また、Version-1コードは、標準Cの関数呼び出しプロトコルの制約を守りますので、Version-0よりも移植性があります。詳細についてはソース配布物内の`src/backend/utils/fmgr/README`を参照してください。

35.9.5. コードの作成

より先進的な話題に入る前に、PostgreSQL C言語関数のコーディングについての規則をいくつか説明します。C言語以外の言語で記述した関数をPostgreSQLに組み込み込むことは可能であるかもしれませんが、例えばC++、FORTRANやPascalといった言語はC言語と同じ呼び出し規約に従いませんので、多くの場合、（可能であったとしても）困難です。それはつまり、他の言語では同じ方法で関数に引数を渡したり、関数から結果を返すことを行わないということです。このため、C言語関数は実際にC言語で書かれているものと仮定します。

C関数の作成と構築の基本規則を以下に示します。

- `pg_config --includedir-server`を使用して、使用中のシステム（もしくはユーザが実行するシステム）にてPostgreSQLサーバのヘッダファイルがインストールされた場所を見つけます。
- PostgreSQLに動的にロードできるように独自コードをコンパイル/リンクする時には常に、特別なフラグが必要となります。特定のオペレーティングシステムにおけるコンパイル/リンク方法については[項35.9.6](#)を参照してください。
- 忘れずに[項35.9.1](#)で説明した"魔法のブロック"を共有ライブラリで定義してください。
- メモリを割り当てる際、Cライブラリの`malloc`と`free`ではなく、PostgreSQLの`palloc`と`pfree`を使用してください。`palloc`で割り当てられたメモリは各トランザクションの終わりに自動的に解放され、メモリリークを防ぎます。
- `memset`を使用して、構造体を必ずゼロクリアしてください（または最初の段階で`palloc0`を用いて割り当ててください）。構造体の各フィールドを割り当てたとしても、ゴミの値を持つ整列用のパディング（構造体内の穴）があるかもしれません。こうしないと、ハッシュインデックスやハッシュ結合をサポートすることが困難です。ハッシュを計算するには、データ構造体内の有意なビットのみを取り出す必要があるためです。プランナはまた時折ビット単位の等価性を用いて定数の比較を行います。このため論理的にな値がビット単位で等価でない場合に望まない計画になってしまう可能性があります。
- ほとんどのPostgreSQLの内部型は`postgres.h`に宣言されています。一方、関数管理インタフェース（`PG_FUNCTION_ARGS`など）は`fmgr.h`で宣言されています。したがって、少なくともこの2つのファイルをインクルードする必要があります。移植性に関する理由により、`postgres.h`をその他のシステムヘッダファイル、ユーザヘッダファイルよりも先にインクルードしておくことが最善です。`postgres.h`をインクルードすることは`eelog.h`、`palloc.h`もインクルードすることになります。
- オブジェクトファイルで定義されているシンボル名は、互いに、またはPostgreSQLサーバの実行ファイルで定義されているものと異なっている必要があります。これに関するエラーが表示される場合は、関数名または変数名を変更する必要があります。

35.9.6. 動的にロードされる関数のコンパイルとリンク

Cで書かれたPostgreSQLの拡張関数を使うためには、サーバが動的にロードできるように特別な方法でコンパイルとリンクを行う必要があります。正確には共有ライブラリを作る必要があります。

本節の説明以上の詳しい情報はオペレーティングシステムのドキュメント、特にCコンパイラ`cc`とリンクエディタ`ld`のマニュアルページを参照してください。さらに、PostgreSQLのソースコードの`contrib`ディレクトリにいくつか実例があります。しかし、もしこれらの例に頼るとPostgreSQLソースコードが利用できることに依存したモジュールが作られてしまいます。

共有ライブラリの作成は一般的に実行プログラムのリンクに類似しています。まずソースファイルがオブジェクトファイルにコ

ンパイルされ、そのオブジェクトファイル同士がリンクされます。これらのオブジェクトファイルは位置独立なコード（PIC）として作られる必要があります。それは概念的には、実行プログラムから呼び出される時にメモリの適当な場所に置くことができるということです（実行プログラム用として作られたオブジェクトファイルはそのようにはコンパイルされません）。共有ライブラリをリンクするコマンドは実行プログラムのリンクと区別するための特別なフラグがあります（少なくとも理論上ではそのようになっています。他のシステムではもっと醜い実際が見受けられます）。

次の例ではソースコードは`foo.c`ファイルにあると仮定し、`foo.so`という共有ライブラリを作るとします。中間のオブジェクトファイルは特別な記述がない限り`foo.o`と呼ばれます。共有ライブラリは1つ以上のオブジェクトファイルを持つことができますが、ここでは1つしか使いません。

FreeBSD

PICを作るためのコンパイラフラグは`-fpic`です。共有ライブラリを作るコンパイラフラグは`-shared`です。

```
gcc -fpic -c foo.c
gcc -shared -o foo.so foo.o
```

これはFreeBSDのバージョン3.0に適用されます。

HP-UX

PICを作るためのシステムコンパイラのコンパイラフラグは`+z`です。GCCを使う場合は`-fpic`です。共有ライブラリのためのリンクフラグは`-b`です。したがって、以下のようになります。

```
cc +z -c foo.c
```

または

```
gcc -fpic -c foo.c
```

そして

```
ld -b -o foo.sl foo.o
```

HP-UXは他のほとんどのシステムと異なり共有ライブラリに`.sl`という拡張子を使います。

IRIX

PICがデフォルトで、特別なコンパイラオプションは何も必要ありません。共有ライブラリを作るためのリンクオプションは`-shared`です。

```
cc -c foo.c
ld -shared -o foo.so foo.o
```

Linux

PICを作るためのコンパイラフラグは`-fpic`です。いくつかのプラットフォームでは状況によって`-fpic`が動作しない場合は`-fPIC`を使わなければいけません。さらに詳しい情報についてはGCCのマニュアルを参照してください。共有ライブラリを作るコンパイラフラグは`-shared`です。完全な例は下記ようになります。

```
cc -fpic -c foo.c
cc -shared -o foo.so foo.o
```

Mac OS X

例を以下に示します。開発者用ツールがインストールされていることが前提です。

```
cc -c foo.c
cc -bundle -flat_namespace -undefined suppress -o foo.so foo.o
```

NetBSD

PICを作るためのコンパイラフラグは`-fpic`です。ELFシステムでは`-shared`コンパイラフラグを使用して共有ライブラリをリンクします。より古い非ELFシステムでは`ld -Bshareable`が使われます。

```
gcc -fpic -c foo.c
gcc -shared -o foo.so foo.o
```

OpenBSD

PICを作成するためのコンパイラフラグは`-fpic`です。共有ライブラリをリンクするには`ld -Bshareable`を使用します。

```
gcc -fpic -c foo.c
ld -Bshareable -o foo.so foo.o
```

Solaris

PICを作るためのコンパイラフラグはSunコンパイラでは`-KPIC`で、GCCでは`-fpic`です。共有ライブラリをリンクするためには、どちらのコンパイラでもコンパイラオプションは`-G`で、GCCの場合、代わりに`-shared`オプションを使うこともできます。

```
cc -KPIC -c foo.c
cc -G -o foo.so foo.o
```

もしくは

```
gcc -fpic -c foo.c
gcc -G -o foo.so foo.o
```

Tru64 UNIX

PICはデフォルトで、コンパイルコマンドは通常のもので。リンクのためには特別なオプション付きの`ld`を使用します。

```
cc -c foo.c
ld -shared -expect_unresolved '*' -o foo.so foo.o
```

システムのコンパイラではなくGCCを使う場合も同じ手順です。特別のオプションは必要ありません。

UnixWare

PICを作るためのコンパイラフラグはSCOコンパイラでは`-KPIC`で、GCCでは`-fpic`です。共有ライブラリのリンクは、SCOコンパイラではコンパイラオプションは`-G`で、GCCでは`-shared`です。

```
cc -K PIC -c foo.c
cc -G -o foo.so foo.o
```

もしくは

```
gcc -fpic -c foo.c
gcc -shared -o foo.so foo.o
```

ティップ: これがあまりに難しいようであれば、[GNU Libtool](#)の使用を検討すべきです。これはプラットフォームの違いを、統一されたインタフェースで判らないようにします。

これで完成した共有ライブラリファイルはPostgreSQLにロードすることができます。CREATE FUNCTION コマンドにファイル名を指定する時には、中間オブジェクトファイルではなく共有ライブラリファイル名の名前を与えてください。システムの標準共有ライブラリ用の拡張子（通常`.so`あるいは`.sl`）はCREATE FUNCTION で省略することができ、そして移植性を最も高くするため通常は省略されます。

サーバがライブラリファイルをどこに見つけるかに関しては[項35.9.1](#)を見直してください。

35.9.7. 複合型引数

複合型ではCの構造体のような固定のレイアウトがありません。複合型のインスタンスはNULLフィールドを持つことができます。さらに、複合型で継承階層の一部であるものは、同じ継承階層の他のメンバとは異なるフィールドを持つこともできます。そのため、PostgreSQLはC言語から複合型のフィールドにアクセスするための関数インタフェースを提供します。

以下のような問い合わせに答える関数を書こうとしていると仮定します。

```
SELECT name, c_overpaid(emp, 1500) AS overpaid
FROM emp
WHERE name = 'Bill' OR name = 'Sam';
```

Version 0呼び出し規約を使用すると、`c_overpaid`は以下のように定義できます。

```
#include "postgres.h"
#include "executor/executor.h" /* GetAttributeByName() 用 */

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

bool
c_overpaid(HeapTupleHeader t, /* empの現在の行 */
           int32 limit)
{
    bool isnull;
    int32 salary;
```



```

    salary = DatumGetInt32(GetAttributeByName(t, "salary", &isnull));
    if (isnull)
        return false;
    return salary > limit;
}

```

Version 1で作成すると、上の関数は以下ようになります。

```

#include "postgres.h"
#include "executor/executor.h" /* GetAttributeByName() 用 */

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

PG_FUNCTION_INFO_V1(c_overpaid);

Datum
c_overpaid(PG_FUNCTION_ARGS)
{
    HeapTupleHeader t = PG_GETARG_HEAPTUPLEHEADER(0);
    int32 limit = PG_GETARG_INT32(1);
    bool isnull;
    Datum salary;

    salary = GetAttributeByName(t, "salary", &isnull);
    if (isnull)
        PG_RETURN_BOOL(false);
    /* この他、salaryがNULLの場合用にPG_RETURN_NULL()を行った方が良いでしょう */

    PG_RETURN_BOOL(DatumGetInt32(salary) > limit);
}

```

`GetAttributeByName`は、指定された行から属性を返す、PostgreSQLシステム関数です。これには3つの引数があります。それらは、関数に渡された`HeapTupleHeader`型の引数、求められた属性の名前、属性がNULLであるかどうかを通知する返りパラメータです。`GetAttributeByName`は適切な`DatumGetXXX()`マクロを使用して適切なデータ型に変換可能な`Datum`型の値を返します。このNULLフラグが設定されている場合、戻り値の意味がないことに注意し、この結果で何かを行おうとする前に常に、NULLフラグを検査してください。

対象列を名前ではなく列番号で選択する`GetAttributeByNum`もあります。

下記のコマンドで`c_overpaid`関数をSQLで宣言します。

```

CREATE FUNCTION c_overpaid(emp, integer) RETURNS boolean
AS 'DIRECTORY/funcs', 'c_overpaid'
LANGUAGE C STRICT;

```

入力引数がNULLかどうかを検査する必要があるように`STRICT`を使用していることに注意してください。

35.9.8. 行（複合型）を返す

C言語関数から行もしくは複合型の値を返すために、複合型の複雑な作成のほとんどを隠蔽するマクロや関数を提供する、特別なAPIを使用することができます。このAPIを使用するためには、ソースファイルで以下をインクルードする必要があります。

```

#include "funcapi.h"

```

複合型のデータ値（以降"タプル"と記す）を作成する2つの方法があります。`Datum`値の配列から作成する方法、もしくはタプルのある列の型の入力変換関数に渡すことができるC文字列の配列から作成することです。どちらの方法でも、まずタプル構造体用の`TupleDesc`記述子を手、あるいは作成しなければなりません。`Datum`を使用する場合は、`TupleDesc`を`BlessTupleDesc`に渡し、各行に対して`heap_form_tuple`を呼び出します。C文字列を使用する場合は、`TupleDesc`を`TupleDescGetAttInMetadata`に渡し、各行に対して`BuildTupleFromCStrings`を呼び出します。タプルの集合を返す関数の場合、この設定段階を最初の関数呼び出しで一度にまとめて行うことができます。

必要な`TupleDesc`の設定用の補助関数はいくつかあります。ほとんどの複合型を返す関数での推奨方法は、以下の関数を呼び出し、呼び出し元の関数自身に渡される`fcinfo`構造体と同じものを渡すことです。

```

TypeFuncClass get_call_result_type(FunctionCallInfo fcinfo,
                                   Oid *resultTypeId,
                                   TupleDesc *resultTupleDesc)

```

（これにはもちろん、version 1呼び出し規約を使用していることが必要です。）`resultTypeId`をNULLとすることも、ローカル変数のアドレスを指定して関数の戻り値型のOIDを受け取ることができます。`resultTupleDesc`はローカルな`TupleDesc`変数のアドレスでなければなりません。結果が`TYPEFUNC_COMPOSITE`かどうかを確認してください。`TYPEFUNC_COMPOSITE`であった場

合、`resultTupleDesc`には必要な`TupleDesc`が格納されています。（`TYPEFUNC_COMPOSITE`ではなかった場合、"レコード型を受け付けられない文脈でレコードを返す関数が呼び出されました"というエラーを報告することができます。）

ティップ: `get_call_result_type`は、多様性関数の結果の実際の型を解決することができます。 ですので、複合型を返す関数だけではなく、スカラの多様結果を返す関数でも有意です。 `resultTypeId`出力は主にスカラの多様結果を返す関数で有意です。

注意: `get_call_result_type`は、`get_expr_result_type`と似たような関数で、関数呼び出しで想定される出力型を式のツリー構造として解決します。 関数自身以外から結果型を決定したい場合に、これを使用することができます。 また、`get_func_result_type`という関数もあります。 これは関数のOIDが利用できる場合にのみ使用することができます。 しかし、これらの関数は、`record`型を返すものと宣言された関数では使用できません。 また、`get_func_result_type`は多様型を解決することができません。 したがって、優先して`get_call_result_type`を使用すべきです。

古く、廃止予定の`TupleDesc`を入手するための関数を以下に示します。

```
TupleDesc RelationNameGetTupleDesc(const char *relname)
```

これを指名したリレーシヨンの行型用の`TupleDesc`を取り出すために使用してください。 また、

```
TupleDesc TypeGetTupleDesc(Oid typeoid, List *colaliases)
```

これを型のOIDに基づいて`TupleDesc`を取り出すために使用してください。 これは、基本型もしくは複合型の`TupleDesc`を取り出すために使用可能です。 これは`record`を返す関数ではうまく動作しません。 また、多様型を解決することもできません。

`TupleDesc`を獲得した後に、`Datum`を使用する場合は以下を呼び出してください。

```
TupleDesc BlessTupleDesc(TupleDesc tupdesc)
```

C文字列を使用する場合は以下を呼び出してください。

```
AttInMetadata *TupleDescGetAttInMetadata(TupleDesc tupdesc)
```

集合を返す関数を作成する場合は、これらの関数の結果を`FuncCallContext`構造体に格納してください。 それぞれ`tuple_desc`と`attinmeta`を使用します。

`Datum`を使用する場合は、ユーザデータを`Datum`形式に格納した`HeapTuple`を構築するために以下を使用します。

```
HeapTuple heap_form_tuple(TupleDesc tupdesc, Datum *values, bool *isnull)
```

C文字列を使用する場合は、ユーザデータをC文字列形式に格納した`HeapTuple`を構築するために以下を使用します。

```
HeapTuple BuildTupleFromCStrings(AttInMetadata *attinmeta, char **values)
```

`values`は行の各属性を1要素としたC文字列の配列です。 各C文字列は、属性のデータ型用の入力関数が受け付け可能な形式でなければなりません。 属性の値をNULL値として返すためには、`values`配列の対応するポインタにNULLを設定してください。 この関数は返す行それぞれに対して繰り返し呼び出す必要があります。

関数から返すタプルを構築し終わったら、それを`Datum`に変換しなければなりません。 以下を使用して、`HeapTuple`を有効な`Datum`に変換してください。

```
HeapTupleGetDatum(HeapTuple tuple)
```

単一行のみを返すのであれば、この`Datum`を直接返すことができます。 さもなくば、集合を返す関数における現在の戻り値として使用することができます。

次節に例を示します。

35.9.9. 集合を返す

C言語関数から集合（複数行）を返す機能のために特殊なAPIが用意されています。 集合を返す関数は、Version 1呼び出し規約に従う必要があります。 また、ソースファイルは上述の通り`funcapi.h`をインクルードする必要があります。

集合を返す関数（SRF）は返される項目ごとに呼び出されます。 そのため、SRFは、過去の操作を記憶して呼び出しの度に次の項目を返すために十分な状態を保っている必要があります。 この処理を制御を補助するための`FuncCallContext`構造体が備わっています。 関数内では、複数の呼び出しにまたがる`FuncCallContext`へのポインタを保持するには、`fcinfo->flinfo->fn_extra`を使用します。

```
typedef struct
{
    /*
     * 既に行われた呼び出しの回数。
     */
}
```

```

/*
 * SRF_FIRSTCALL_INIT() によってcall_cntrが0に初期化され、
 * SRF_RETURN_NEXT() が呼び出される度に増分されます。
 */
uint32 call_cntr;

/*
 * 省略可能 : 呼び出しの最大数
 *
 * max_callsは、便宜上用意されているだけで、設定は省略可能です。
 * 設定されていなければ、関数が終了したことを知るための別の方法を
 * 用意する必要があります。
 */
uint32 max_calls;

/*
 * 省略可能 : 結果スロットへのポインタ
 *
 * これは廃止され、後方互換性、すなわち非推奨のTupleDescGetSlot() を使用する
 * ユーザ定義のSRFのためにだけ存在します。
 */
TupleTableSlot *slot;

/*
 * 省略可能 : 様々なユーザによるコンテキスト情報へのポインタ
 *
 * user_fctxは、関数の呼び出し間の任意のコンテキスト情報
 * を取得するためのユーザ独自の構造へのポインタとして使用されます。
 */
void *user_fctx;

/*
 * 省略可能 : 属性型入力メタ情報を含んだ構造体へのポインタ
 *
 * attinmeta はタプル（つまり複合データ型）を返す際に使用され、
 * 基本データ型を返す場合には必要ありません。
 * BuildTupleFromCStrings() を使用して返されるタプルを作成する場合にのみ必要です。
 */
AttInMetadata *attinmeta;

/*
 * 複数の呼び出しで必要とされる構造体に使われるメモリコンテキスト
 *
 * multi_call_memory_ctxは、SRF_FIRSTCALL_INIT() によってに設定され、
 * SRF_RETURN_DONE() がクリーンアップの際に使用します。
 * これはSRFの複数呼び出しで再利用される全てのメモリ用に最も適切なメモリコンテキストです。
 */
MemoryContext multi_call_memory_ctx;

/*
 * 省略可能: タプル説明を含む構造体へのポインタ。
 * tuple_descはタプル（つまり複合データ型）を返す場合に使用され、BuildTupleFromCStrings()
 * ではなくheap_form_tuple() を使用してタプルを作成する場合にのみ必要です。
 * 通常ここに格納されるTupleDescは最初にBlessTupleDesc() を最初に実行したものでなければなり
 * ません。
 */
TupleDesc tuple_desc;

} FuncCallContext;

```

SRFはいくつかの関数およびマクロを使用してFuncCallContext構造体を自動的に操作します（またfn_extraで検索することを想定します）。

```
SRF_IS_FIRSTCALL()
```

を使用して、その関数呼び出しが初回のものであるか、2回目以降であるかを判断します。 最初の呼び出し（のみ）で、

```
SRF_FIRSTCALL_INIT()
```

を使用して、FuncCallContextを初期化します。 最初の呼び出しを含むすべての呼び出しで、

```
SRF_PERCALL_SETUP()
```

を使用して、FuncCallContextを使用するための適切な設定を行い、以前の受け渡しから残っている結果データを消去します。

関数で返すべきデータがある場合は、

```
SRF_RETURN_NEXT(funcctx, result)
```

を使用して、そのデータを呼び出し側に返します。（先に説明した通り resultはDatum型、つまり1つの値またはタプルである必要があります。） 最後に、関数がデータを返し終わったら、

```
SRF_RETURN_DONE(funcctx)
```

を使用してSRFを片付け、終了します。

SRFの呼び出し時に現行になっているメモリコンテキストは一時的なコンテキストで、各呼び出しの間に消去されます。つまり `palloc` を使用して割り当てたもののすべてを `pfree` する必要はありません。これらはいずれ消去されるものだからです。しかし、データ構造体を複数の呼び出しに渡って使用するように割り当てる場合は、どこか別の場所に置いておく必要があります。`multi_call_memory_ctx` によって参照されるメモリコンテキストは、SRFの実行が終わるまで使用可能にしなければならないデータの保管場所として適しています。つまり、ほとんどの場合、最初の呼び出しのセットアップ中に `multi_call_memory_ctx` へ切り替える必要があるということです。

完全な疑似コードの例を示します。

```
Datum
my_set_returning_function(PG_FUNCTION_ARGS)
{
    FuncCallContext *funcctx;
    Datum          result;
    further declarations as needed

    if (SRF_IS_FIRSTCALL())
    {
        MemoryContext oldcontext;

        funcctx = SRF_FIRSTCALL_INIT();
        oldcontext = MemoryContextSwitchTo(funcctx->multi_call_memory_ctx);
        /* 一度限りのセットアップコードがここに入ります: */
        user code
        if returning composite
            build TupleDesc, and perhaps AttInMetadata
        endif returning composite
        user code
        MemoryContextSwitchTo(oldcontext);
    }

    /* 毎回実行するセットアップコードがここに入ります: */
    user code
    funcctx = SRF_PERCALL_SETUP();
    user code

    /* これは、終了したかどうかをテストする方法の1つです: */
    if (funcctx->call_cntr < funcctx->max_calls)
    {
        /* ここで、別の項目を返します: */
        user code
        obtain result Datum
        SRF_RETURN_NEXT(funcctx, result);
    }
    else
    {
        /* これで項目を返し終わりました。後はクリーンアップするだけです。 */
        user code
        SRF_RETURN_DONE(funcctx);
    }
}
```

複合型を返す単純なSRFの完全な例は以下の通りです。

```
PG_FUNCTION_INFO_V1(retcomposite);

Datum
retcomposite(PG_FUNCTION_ARGS)
{
    FuncCallContext *funcctx;
    int              call_cntr;
    int              max_calls;
    TupleDesc        tupdesc;
    AttInMetadata    *attinmeta;

    /* 関数の最初の呼び出し時にのみ実行 */
    if (SRF_IS_FIRSTCALL())
    {
        MemoryContext oldcontext;

        /* 呼び出し間で永続化する関数コンテキストを作成 */
        funcctx = SRF_FIRSTCALL_INIT();

        /* 複数関数呼び出しに適切なメモリコンテキストへの切り替え */
        oldcontext = MemoryContextSwitchTo(funcctx->multi_call_memory_ctx);

        /* 返されるタブルの合計数 */
        funcctx->max_calls = PG_GETARG_UINT32(0);

        /* 結果型用のタブル記述子を作成 */
        if (get_call_result_type(fcinfo, NULL, &tupdesc) != TYPEFUNC_COMPOSITE)
```

```

ereport (ERROR,
        (errcode(ERRCODE_FEATURE_NOT_SUPPORTED),
         errmsg("function returning record called in context "
                "that cannot accept type record")));

/*
 * 後で未加工のC文字列からタプルを作成するために必要となる
 * 属性メタデータの生成
 */
attinmeta = TupleDescGetAttInMetadata(tupdesc);
funcctx->attinmeta = attinmeta;

MemoryContextSwitchTo(oldcontext);
}

/* 全ての関数呼び出しで実行 */
funcctx = SRF_PERCALL_SETUP();

call_cntr = funcctx->call_cntr;
max_calls = funcctx->max_calls;
attinmeta = funcctx->attinmeta;

if (call_cntr < max_calls) /* 他にも送るものがある場合 */
{
    char          **values;
    HeapTuple      tuple;
    Datum          result;

    /*
     * 返すタプルを構築するためのvalues配列を用意します。
     * これは、後で適切な入力関数で処理される
     * C文字列の配列でなければなりません。
     */
    values = (char **) palloc(3 * sizeof(char *));
    values[0] = (char *) palloc(16 * sizeof(char));
    values[1] = (char *) palloc(16 * sizeof(char));
    values[2] = (char *) palloc(16 * sizeof(char));

    snprintf(values[0], 16, "%d", 1 * PG_GETARG_INT32(1));
    snprintf(values[1], 16, "%d", 2 * PG_GETARG_INT32(1));
    snprintf(values[2], 16, "%d", 3 * PG_GETARG_INT32(1));

    /* タプルの作成 */
    tuple = BuildTupleFromCStrings(attinmeta, values);

    /* タプルをdatumに変換 */
    result = HeapTupleGetDatum(tuple);

    /* クリーンアップ (これは必須ではありません) */
    pfree(values[0]);
    pfree(values[1]);
    pfree(values[2]);
    pfree(values);

    SRF_RETURN_NEXT(funcctx, result);
}
else /* 何も残っていない場合 */
{
    SRF_RETURN_DONE(funcctx);
}
}

```

以下にこの関数をSQLで宣言する一例を示します。

```

CREATE TYPE __retcomposite AS (f1 integer, f2 integer, f3 integer);

CREATE OR REPLACE FUNCTION retcomposite(integer, integer)
RETURNS SETOF __retcomposite
AS 'filename', 'retcomposite'
LANGUAGE C IMMUTABLE STRICT;

```

他にも以下のようにOUTパラメータを使用する方法もあります。

```

CREATE OR REPLACE FUNCTION retcomposite(IN integer, IN integer,
OUT f1 integer, OUT f2 integer, OUT f3 integer)
RETURNS SETOF record
AS 'filename', 'retcomposite'
LANGUAGE C IMMUTABLE STRICT;

```

この方法では、関数の出力型は形式上無名の`record`型になることに注意してください。

ソース配布物内の[contrib/tablefunc](#)モジュールのディレクトリには、集合を返す関数のより多くの例があります。

35.9.10.

引数と戻り値の多様性

C言語関数は、`anyelement`、`anyarray`、`anynonarray`、`anyenum`および`anyrange`多様型を受け付ける、または返すように宣言することができます。多様関数の詳細な説明は[項35.2.5](#)を参照してください。関数の引数もしくは戻り値が多様型として定義される時、関数の作成者は前もって呼び出しにおけるデータ型や返すべきデータ型が何であるかを知ることができません。Version-1 C関数で引数の実データ型と、返すべきと想定された型を発見できるための2つのルーチンが`fmgr.h`に用意されています。このルーチンは`get_fn_expr_rettype(FmgrInfo *flinfo)`と`get_fn_expr_argtype(FmgrInfo *flinfo, int argnum)`という名前です。これらは結果もしくは引数型のOIDを返します。ただし、もし情報が利用できなければ`InvalidOid`を返します。`flinfo`構造体は通常`fcinfo->flinfo`としてアクセスされます。`argnum`パラメータは0から始まります。また、`get_fn_expr_rettype`の代わりに`get_call_result_type`を使用することもできます。

例えば、任意の型の単一要素を受け付け、その型の1次元配列を返す関数を考えてみます。

```
PG_FUNCTION_INFO_V1(make_array);
Datum
make_array(PG_FUNCTION_ARGS)
{
    ArrayType *result;
    Oid        element_type = get_fn_expr_argtype(fcinfo->flinfo, 0);
    Datum      element;
    bool        isnull;
    int16       typlen;
    bool        typbyval;
    char        typalign;
    int         ndims;
    int         dims[MAXDIM];
    int         lbs[MAXDIM];

    if (!OidIsValid(element_type))
        elog(ERROR, "could not determine data type of input");

    /* 与えられた要素がNULLかどうか注意しつつ、要素を取り出します。*/
    isnull = PG_ARGISNULL(0);
    if (isnull)
        element = (Datum) 0;
    else
        element = PG_GETARG_DATUM(0);

    /* 次元数は1 */
    ndims = 1;
    /* 要素を1つ */
    dims[0] = 1;
    /* 下限は1 */
    lbs[0] = 1;

    /* この要素型に関する必要情報を取り出す。*/
    get_typlenbyvalalign(element_type, &typlen, &typbyval, &typalign);

    /* ここで配列を作成 */
    result = construct_md_array(&element, &isnull, ndims, dims, lbs,
                               element_type, typlen, typbyval, typalign);

    PG_RETURN_ARRAYTYPE_P(result);
}
```

以下のコマンドはSQLで`make_array`関数を宣言します。

```
CREATE FUNCTION make_array(anyelement) RETURNS anyarray
AS 'DIRECTORY/funcs', 'make_array'
LANGUAGE C IMMUTABLE;
```

C言語関数でのみ使用できる多様性の変異体があります。"any"型のパラメータを取るように宣言できます。（この型名は、SQL予約語でもあるため二重引用符で括らなくてはならないことに注意してください。）これは、他の"any"引数が同じ型になることを強要することも、関数の結果型の決定を支援することもない点を除いて、`anyelement`のように動作します。C言語関数は最終パラメータが`VARIADIC "any"`であるように宣言可能です。これは任意の型の1つ以上の実引数と一致します（同じ型である必要はありません）。これらの引数は、通常のvariadic関数で起こったように、配列の中にまとめられません。それらは単に別々に関数に渡されるだけです。`PG_NARGS()`マクロと上に記載したメソッドは、この機能を使用するとき実際の引数とその型を決定するため使用されなければなりません。

35.9.11. 変形関数

一部の関数呼び出しでは、関数固有の属性に基づいて計画作成を単純化できます。例えば、`int4mul(n, 1)`は`n`だけに単純化することができます。こうした関数固有の最適化を定義するためには、変形関数を作成し、そのOIDを主関数の`pg_proc`項目の`protransform`フィールドに格納します。変形関数は`protransform(internal) RETURNS internal`というSQLシグネチャを持たなければなりません。引数、実際は`FuncExpr *`は、主関数の呼び出しを表すダミーノードです。変形関数の式ツリー学習によって、式ツリーで表されるすべての可能性がある実際の呼び出しを単純化した式ツリーで置き換えることができることが証明

された場合、単純化した式を構築し返します。 さもなければ、(SQLのNULLではなく) `NULL` ポインタを返します。

PostgreSQLが変形関数によって単純化できる場合に主関数を呼び出さないことは保証されません。 単純化した式と実際の主関数の呼び出しとで、厳密に等価であることを確実にしてください。

現在、セキュリティ上の懸念から、この機能がSQLレベルでユーザに見えることはありません。 このため、これは組み込み関数の最適化での使用のみで実用的です。

35.9.12. 共有メモリとLWLocks

アドインはLWLocks（軽量ロック）とサーバ起動時に共有メモリの割り当てを保持することができます。

[shared_preload_libraries](#)で指定して、こうしたアドインの共有ライブラリを事前にロードしなければなりません。 共有メモリは、その `_PG_init` 関数で以下を呼び出すことで保持されます。

```
void RequestAddinShmemSpace(int size)
```

LWLocksはその `_PG_init` 関数で以下を呼び出すことで保持されます。

```
void RequestAddinLWLocks(int n)
```

競合状態の可能性を防止するために、割り当てられた共有メモリへの接続やその初期化時に、以下のように各バックエンドで `AddinShmemInitLock` 軽量ロックを使用しなければなりません。

```
static mystruct *ptr = NULL;

if (!ptr)
{
    bool    found;

    LWLockAcquire(AddinShmemInitLock, LW_EXCLUSIVE);
    ptr = ShmemInitStruct("my struct name", size, &found);
    if (!found)
    {
        initialize contents of shmem area;
        acquire any requested LWLocks using:
        ptr->mylockid = LWLockAssign();
    }
    LWLockRelease(AddinShmemInitLock);
}
```

35.9.13. 拡張へのC++の利用

以下のガイドラインに従うことで、PostgreSQLの拡張を構築するためC++モードのコンパイラを利用できます。

- バックエンドからアクセスされる関数はすべてバックエンドに対してCインタフェースを提供しなければなりません。 このC関数はC++関数を呼び出すことができます。 例えば、バックエンドからアクセスされる関数には `extern C` リンクが必要です。 これはバックエンドとC++コードの間でポインタとして渡される関数にも必要です。
- 適切な解放メソッドを使ってメモリを解放してください。 例えば、ほとんどのバックエンドメモリは `malloc()` で確保されますので、`free()` を使って解放してください。 この場合にC++の `delete()` を使うと失敗するでしょう。
- 例外がCコードへ伝播しないようにしてください（`extern C` 関数すべての最上位ですべての例外を捕捉するブロックを使ってください）。 メモリ不足のようなイベントにより例外が発生する可能性がありますので、C++コードが何も例外を発生させない場合であっても、これは必要です。 例外はすべて捕捉しなければなりません。 そして適切なエラーをCインタフェースに渡してください。 可能であれば、例外を完全に除去できるように `-fno-exceptions` を付けてC++をコンパイルしてください。 その場合、例えば `new()` で返されるNULLの検査など、C++コード内で失敗の検査を行わなければなりません。
- C++コードからバックエンド関数を呼び出す場合には、C++呼び出しスタック内にC言語互換構造体（POD）のみが含まれていることを確認してください。 バックエンドのエラーは、非PODオブジェクトを持つC++呼び出しスタックを適切に戻すことができない、長距離 `longjmp()` を生成しますので、これは必要です。

まとめると、バックエンドとやりとりするための壁の役割を担う `extern C` 関数の背後にC++コードを配置して、例外、メモリ、呼び出しスタックそれぞれの漏れを避けるのが最善です。

35.10. ユーザ定義の集約

PostgreSQLにおける集約関数は、状態値と状態遷移関数で表現されています。つまり、1つの集約は入力行が処理される度に更新される状態値を使用して演算されます。新しい集約関数を定義するためには、状態値のデータ型、状態の初期値、そして状態遷移関数を選択します。状態遷移関数はただの普通の関数で、集約以外の文脈でも使うことができます。求められる集約の結果が遷移する状態値の中に保持しなければならないデータと違う場合は、最終関数も指定することができます。

したがって、集約のユーザに見える引数と結果のデータ型に加え、引数と結果の型のどちらとも違う可能性がある内部状態値のデータ型があります。

もし最終関数を使わない集約を定義したとすると、それぞれの行の列値で変動する関数で計算された結果の集約ができます。`sum`はそのような集約の一例です。`sum`は0から始まり、常に現在の行の値をその時点までの総和に追加します。例えば、もし`sum`集約を複素数のデータ型で動作するようにしたければ、そのデータ型の加算関数だけが必要になります。集約定義は以下のようになります。

```
CREATE AGGREGATE sum (complex)
(
    sfunc = complex_add,
    stype = complex,
    initcond = '(0,0)'
);

SELECT sum(a) FROM test_complex;

      sum
-----
(34,53.9)
```

(関数のオーバーライド機能に依存していることに注意してください。`sum`という名前の集約関数は複数存在しますが、PostgreSQLは列の型`complex`に適用できる`sum`関数を見つけ出すことができます。)

上記の`sum`の定義は、もし非NULLの入力値がなければ0（初期状態）を返します。その場合本来は代わりにNULLを返したいのではないかと思いますし、標準SQLでは`sum`がそのように動作することを期待しています。そうするためには単に`initcond`句を省略すれば、初期状態がNULLになります。通常はそうすると、`sfunc`がNULL状態の入力をチェックしなければいけないということを意味します。しかし`sum`や、その他`max`、`min`のような単純な集約にとっては、最初の非NULL入力値を状態変数に挿入し、2番目の非NULL入力値で状態遷移関数を当てはめ始めれば十分です。PostgreSQLは、もし初期状態がNULLで状態遷移関数が"厳密 (strict)"と設定されている場合、自動的にそれを実行します（つまりNULL入力では呼び出されないようになります）。

もう1つの"厳密"な状態遷移関数のデフォルト動作としては、NULL入力値が現れると前の状態値が変わらずに維持されるということがあります。したがって、NULL値は無視されます。もしNULL入力に対し他の動作が必要な場合は、状態遷移関数を厳密として定義しないで、NULL入力の検査を行うようにコーディングし、必要なことをすればよいのです。

`avg`（平均値計算）はもっと複雑な集約の一例です。それには2つの変動する状態が必要になります。入力の合計と入力数のカウントです。最終的な結果はこれらの値を割算することによって得られます。平均値計算は典型的に配列を状態遷移値として使って実装されます。例えば、`avg(float8)`の組み込みの実装は以下のようになっています。

```
CREATE AGGREGATE avg (float8)
(
    sfunc = float8_accum,
    stype = float8[],
    finalfunc = float8_avg,
    initcond = '{0,0,0}'
);
```

(`float8_accum`は、入力の総和と個数だけではなく二乗和も蓄積しますので、2要素ではなく、3要素の配列を必要とします。`avg`以外の何らかの集約でも使用できるようにするためです。)

集約関数は多様状態遷移関数や多様最終関数を使用することができます。これにより、同じ関数を使用して複数の集約を実装することができます。[項35.2.5](#)に多様関数の説明があります。もう少し細かく言うと、集約関数自身は、単一の集約定義で複数の入力データ型を扱うことができるように、多様入力型と多様状態型を指定することができるということです。以下に多様集約の例を示します。

```
CREATE AGGREGATE array_accum (anyelement)
(
    sfunc = array_append,
    stype = anyarray,
    initcond = '{}'
```

ここで、集約を行う呼び出し用の実際の状態型は、その実際の入力型がその要素となる、配列型です。 集約の動作はその型の配列に全ての入力を連結することです。 （組み込みの集約関数`array_agg`は、この定義での動作よりもより良い性能で、類似の機能を提供します。）

以下に異なる2つの実データ型を引数として使用した出力例を示します。

```
SELECT attrelid::regclass, array_accum(attname)
FROM pg_attribute
WHERE attnum > 0 AND attrelid = 'pg_tablespace'::regclass
GROUP BY attrelid;

attrelid      | array_accum
-----+-----
pg_tablespace | {spcname,spcowner,spcacl,spcoptions}
(1 row)
```

```
SELECT attrelid::regclass, array_accum(atttypid::regtype)
FROM pg_attribute
WHERE attnum > 0 AND attrelid = 'pg_tablespace'::regclass
GROUP BY attrelid;

attrelid      | array_accum
-----+-----
pg_tablespace | {name,oid,aclitem[],text[]}
(1 row)
```

C言語で作成された関数は、例えば以下のように`AggCheckCallContext`を呼び出して、集約の遷移関数または最終関数が呼び出されたかを検出することができます。

```
if (AggCheckCallContext(fcinfo, NULL))
```

この検査を行う理由の1つとして、遷移関数に対してこれが真の場合、先頭の入力は一時的な遷移中の値であるはずであり、新規に割り当ててコピーを持つことなくそのまま変更しても安全であることが分かります。 例として`int8inc()`を参照してください。 （これは関数内で参照渡しの入力を安全に変更できる唯一の場合です。 特に集約最終関数はいかなる場合でもそれらの入力を変更してはなりません。 と言うのは、時として、同じ最終遷移値上で再実行されることがあるからです。）

さらに詳しい情報は、[CREATE AGGREGATE](#) コマンドを参照してください。

35.11. ユーザ定義の型

[項35.2](#)に述べられているように、PostgreSQLは、新しい型をサポートするように拡張することができます。本節では、SQL言語以下のレベルで定義されるデータ型である基本型を新しく定義する方法について説明します。新しい基本型の作成には、低レベル言語、通常Cで作成された型を操作する関数の実装が必要です。

本節で使用する例は、ソース配布物内のsrc/tutorialディレクトリにcomplex.sqlとcomplex.cという名前で置いてあります。この例の実行方法についてはディレクトリ内のREADMEを参照してください。

ユーザ定義データ型では必ず入力関数と出力関数が必要です。これらの関数は、その型が（ユーザによる入力とユーザへの出力のための）文字列としてどのように表現されるかと、その型がメモリ中でどう構成されるかを決定します。入力関数は引数としてヌル終端文字列を取り、その型の（メモリ中の）内部表現を返します。出力関数は引数としてその型の内部表現を取り、ヌル終端文字列を返します。単に格納するだけではなく、その型に操作を加えたいのであれば、その型に持たせたいすべての操作を実装した関数をさらに提供しなければなりません。

例えば、複素数を表現するcomplex型を定義することを考えます。おそらく、次のようなC構造体で複素数をメモリ中で表現することがごく自然な方法です。

```
typedef struct Complex {
    double    x;
    double    y;
} Complex;
```

単一のDatum値で扱うには大き過ぎるので、これは参照渡し型にしなければなりません。

この型の外部文字列表現として(x,y)形式の文字列を使用することを選択します。

入出力関数、特に出力関数を作成することは困難ではありません。しかし、この型の外部表現文字列を定義する時、その表現のための完全で堅牢なパーサを入力関数として作成しなければなりません。以下に例を示します。

```
PG_FUNCTION_INFO_V1(complex_in);

Datum
complex_in(PG_FUNCTION_ARGS)
{
    char        *str = PG_GETARG_CSTRING(0);
    double      x,
               y;
    Complex     *result;

    if (sscanf(str, " ( %lf , %lf )", &x, &y) != 2)
        ereport(ERROR,
                (errcode(ERRCODE_INVALID_TEXT_REPRESENTATION),
                 errmsg("invalid input syntax for complex: %s", str)));

    result = (Complex *) palloc(sizeof(Complex));
    result->x = x;
    result->y = y;
    PG_RETURN_POINTER(result);
}
```

出力関数は以下のように簡単にできます。

```
PG_FUNCTION_INFO_V1(complex_out);

Datum
complex_out(PG_FUNCTION_ARGS)
{
    Complex     *complex = (Complex *) PG_GETARG_POINTER(0);
    char        *result;

    result = (char *) palloc(100);
    snprintf(result, 100, "(%g,%g)", complex->x, complex->y);
    PG_RETURN_CSTRING(result);
}
```

入出力関数は各々の逆関数になるように注意しなければなりません。そうしないと、データをファイルにダンプし、それを読み戻そうとする際に、深刻な問題が発生するでしょう。これは特に浮動小数点数が関係する際によく発生する問題です。

省略することができますが、ユーザ定義型はバイナリ入出力関数を提供することができます。バイナリ入出力は通常テキスト入出力より高速ですが、テキスト入出力より移植性がありません。テキスト入出力と同様に、外部バイナリ表現を正確に定義することは作成者の責任です。ほとんどの組み込みデータ型は、マシンに依存しないバイナリ表現を提供しようとしています。

`complex`型では`float8`型のバイナリ入出力コンバータを元にします。

```
PG_FUNCTION_INFO_V1(complex_recv);

Datum
complex_recv(PG_FUNCTION_ARGS)
{
    StringInfo    buf = (StringInfo) PG_GETARG_POINTER(0);
    Complex      *result;

    result = (Complex *) palloc(sizeof(Complex));
    result->x = pq_getmsgfloat8(buf);
    result->y = pq_getmsgfloat8(buf);
    PG_RETURN_POINTER(result);
}

PG_FUNCTION_INFO_V1(complex_send);

Datum
complex_send(PG_FUNCTION_ARGS)
{
    Complex      *complex = (Complex *) PG_GETARG_POINTER(0);
    StringInfoData buf;

    pq_begintypsend(&buf);
    pq_sendfloat8(&buf, complex->x);
    pq_sendfloat8(&buf, complex->y);
    PG_RETURN_BYTEA_P(pq_endtypsend(&buf));
}
```

入出力関数を作成し共有ライブラリ内にコンパイルすれば、SQLで`complex`型を定義することができます。まずシェル型として宣言します。

```
CREATE TYPE complex;
```

これは、入出力関数を定義する時にこの型を参照することができるプレースホルダとして動作します。この後以下のように、入出力関数を定義することができます。

```
CREATE FUNCTION complex_in(cstring)
RETURNS complex
AS 'filename'
LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_out(complex)
RETURNS cstring
AS 'filename'
LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_recv(internal)
RETURNS complex
AS 'filename'
LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_send(complex)
RETURNS bytea
AS 'filename'
LANGUAGE C IMMUTABLE STRICT;
```

最後にデータ型の完全な定義を提供することができます。

```
CREATE TYPE complex (
    internallength = 16,
    input = complex_in,
    output = complex_out,
    receive = complex_recv,
    send = complex_send,
    alignment = double
);
```

新しい基本型を定義すると、PostgreSQLは自動的にその型の配列のサポートを提供します。配列型は通常、基本型の名前の前にアンダースコア文字_が付いた名前になります。

データ型が存在するようになると、そのデータ型に対する有用な操作を提供する関数を宣言することができます。そしてその関数を使用する演算子も定義できます。また、必要に応じて、そのデータ型用のインデックスをサポートするための演算子クラスも作成することができます。こうした追加層については後の節で説明します。

データ型の値により（内部形式で）容量が変動する場合、そのデータ型をTOAST可能としなければなりません。（[項56.2](#)を参照してください。）ヘッダのオーバーヘッドを減らすことでTOASTは小さなデータに対しても容量を抑えることができますので、データが常に圧縮し外部に格納するには小さ過ぎる場合でも、これを行わなければなりません。

このためには、内部表現が可変長データの標準レイアウトに従っていなければなりません。先頭の4バイトは`char[4]`フィールド

で、直接アクセスされることは決してありません（慣習的に`vl_len`と呼ばれます）。`SET_VARSIZE()`を使用してデータの容量をこのフィールドに格納し、また、`VARSIZE()`を使用してこのフィールドを取り出さなければなりません。そのデータ型を扱うC関数は常に、`PG_DETOAST_DATUM`を使用して、渡されたTOAST化値を注意深く展開しなければなりません。（通常、こうした詳細は型独自の`GETARG_DATATYPE_P`マクロを定義して隠蔽します。）その後、`CREATE TYPE`コマンドを実行する際に、内部長を`variable`と指定し、また、適当な格納オプションを選択してください。

整列が（単なる特定の関数向けやデータ型が常にバイト単位の整列を規定しているため）重要でない場合、`PG_DETOAST_DATUM`のオーバーヘッドの一部を省くことができます。代わりに`PG_DETOAST_DATUM_PACKED`を使用してください（通常は`PG_DETOAST_DATUM_PACKED`マクロを定義することで隠蔽されます）。そして、`VARSIZE_ANY_EXHDR`および`VARDATA_ANY`マクロを使用して、圧縮されている可能性があるデータにアクセスしてください。繰り返しますが、これらのマクロから返されるデータは、たとえデータ型定義で整列を規定していたとしても、整列されません。整列が重要であれば、通常の`PG_DETOAST_DATUM`インタフェースを介して実行してください。

注意: 古めのコードではしばしば`vl_len`を`char[4]`ではなく`int32`として宣言しています。この構造体定義が少なくとも`int32`で整列されたフィールドを持っている限り、これは問題ありません。しかし、整列されていない可能性があるデータを扱う場合に、こうした構造体定義を使用することは危険です。データが実際に整列されていると仮定することをコンパイラの規則としているかもしれず、この場合、整列に厳密なアーキテクチャではコアダンプしてしまいます。

詳細については[CREATE TYPE](#)コマンドの説明を参照してください。

[前のページ](#)

ユーザ定義の集約

[ホーム](#)

[上に戻る](#)

[次のページ](#)

ユーザ定義の演算子

35.12. ユーザ定義の演算子

演算子は裏側で実際の作業を行う関数を呼び出す"構文上の飾り"です。ですから、演算子を作成する前にまずこの基礎となる関数を作成する必要があります。しかし、演算子は単なる構文上の飾りではありません。問い合わせプランナによる演算子を使用する問い合わせの最適化を補助する追加情報をやり取りするからです。次節では、この追加情報について重点的に説明します。

PostgreSQLでは左単項演算子、右単項演算子、二項演算子をサポートしています。演算子はオーバーロード可能です。つまり、同じ演算子名をオペランドの数と型が異なる演算子に対して使用することができるということです。問い合わせが実行されると、システムは与えられたオペランドの数と型より呼び出すべき演算子を決定します。

以下に2つの複素数の加算を行う演算子を作成する例を示します。既に`complex`型の定義 ([項35.11](#)を参照) を作成していることを前提としています。まず、実作業を行う関数が必要です。その後、演算子を定義できます。

```
CREATE FUNCTION complex_add(complex, complex)
  RETURNS complex
  AS 'filename', 'complex_add'
  LANGUAGE C IMMUTABLE STRICT;

CREATE OPERATOR + (
  leftarg = complex,
  rightarg = complex,
  procedure = complex_add,
  commutator = +
);
```

これで以下のような問い合わせを実行できるようになります。

```
SELECT (a + b) AS c FROM test_complex;

      c
-----
(5.2,6.05)
(133.42,144.95)
```

ここでは二項演算子をどのように作成するのかを示しました。単項演算子を作成するには、単に`leftarg` (左単項の場合) もしくは`rightarg` (右方単項の場合) を省略するだけです。 `procedure` 句と引数用の句のみが `CREATE OPERATOR` での必須項目です。例で示した`commutator` 句は省略可能で、問い合わせオプティマイザへのヒントとなります。 `commutator` やその他のオプティマイザへのヒントについての詳細は次節で説明します。

35.13. 演算子最適化に関する情報

PostgreSQLの演算子定義では、システムに演算子がどう振舞うかに関する有用なことを通知する、いくつかのオプション句を持つことができます。これらの句により演算子を使用する問い合わせの実行速度がかなり向上しますので、これらの句は適切な時には常に提供しなければなりません。しかし、提供する時にはそれらが正しいことを確認しなければいけません！間違っ最最適化用の句を使用すると、問い合わせの低速化、わけのわからないおかしい出力、その他有害な事が起こり得ます。最適化用の句についてわからなければ、使用しなくても構いません。使用された時よりも問い合わせの実行が遅くなるかもしれないというだけです。

PostgreSQLの今後のバージョンで、最適化用の句はさらに追加される可能性があります。ここで説明するものはすべて、バージョン9.2.4で有効なものです。

35.13.1. COMMUTATOR

COMMUTATOR句が与えられた場合、それは定義する演算子の交代演算子となる演算子の名前です。取り得る全ての入力値 x 、 y に対して、 $(x \ A \ y)$ が $(y \ B \ x)$ と等しい時、演算子Aは演算子Bの交代演算子であると言います。また、BはAの交代演算子となることにも注意してください。例えば、通常、特定のデータ型用の演算子 $< >$ は互いの交代演算子になります。また、通常、演算子 $+$ は自身が交代演算子となります。しかし、通常、演算子 $-$ は交代演算子を持ちません。

交代可能な演算子の左オペランドの型は、その交代演算子の右オペランドの型と同一で、その逆もまた同様です。したがって、PostgreSQLで交代演算子を検索する時に必要なものは交代演算子の名前のみになりますので、COMMUTATOR句でそれのみを与えておけば十分です。

インデックスや結合句で使用される演算子では交代演算子の情報を提供することが必須です。これにより、問い合わせ最適マイザがその句を他の種類の実行計画で必要とされる形式に"ひっくり返す"ことができるためです。例えば、 $tab1.x = tab2.y$ のようなWHERE句を持った問い合わせを考えてみます。ここで $tab1.x$ と $tab2.y$ はユーザ定義型で、 $tab2.y$ にはインデックスが付いていると仮定します。最適マイザは、この句を $tab2.y = tab1.x$ という形にひっくり返す方法を知らない限り、インデックススキャンを生成できません。インデックススキャン機構は演算子の左側にインデックス付けされた列があることを想定しているためです。PostgreSQLは簡単にこの変形が有効であると前提しません。=演算子の作成者がこれが有効であることを、交換演算子情報を持つ演算子であると印付けて指定しなければなりません。

自己交代演算子を定義する場合は、単にそれを指定するだけです。交代演算子の対を定義する場合は少し複雑になります。最初に他の未定義のものを参照するものをどう定義するのかということが問題となります。この問題には下記の2つの解決方法があります。

- 1つ目の方法は、最初の演算子を定義する際にCOMMUTATOR句を省略し、2番目の演算子の定義では、COMMUTATOR句に最初の演算子を与えるという方法です。PostgreSQLは交代演算子が対になっていることがわかっていますので、2番目の定義を見た時に、自動的に最初の定義に戻ってその未定義になっているCOMMUTATOR句を設定します。
- もう1つの方法は、両方の定義にCOMMUTATOR句を含めるというもっと素直な方法です。PostgreSQLは最初の定義を処理する際に、COMMUTATORが存在しない演算子を参照していることがわかると、システムはその演算子用の仮のエントリをシステムカタログに作成します。この仮エントリには、PostgreSQLがこの時点で推定できる、演算子名、左オペランドの型、右オペランドの型、および結果の型についてのみの有効なデータが入ります。最初の演算子のカタログエントリはこの仮エントリに結び付きます。その後、2番目の演算子が定義されたら、システムはその仮エントリに2番目の定義から得られる追加情報を更新します。更新される前に仮の演算子を使用すると、エラーメッセージが出力されます。

35.13.2. NEGATOR

NEGATOR句が与えられた場合、それは定義する演算子の否定子となる演算子の名前です。入力値 x と y の取り得るすべての値に対して両方の演算子が論理値を返し、 $(x \ A \ y)$ がNOT $(x \ B \ y)$ と等しい場合、演算子Aは演算子Bの否定子であると言います。また、BはAの否定子でもあることに注意してください。例えば、ほとんどのデータ型では $< >=$ は否定子の対となります。演算子が自身の否定子になることは決してありません。

交代演算子と異なり、単項演算子の対は互いに否定子として有効に指定されます。つまりすべての x に対して $(A \ x)$ がNOT $(B \ x)$ と等しいことを意味します。右単項演算子でも同様です。

ある演算子の否定子は、その演算子定義の左オペランド、右オペランドと同じ型を持たなければなりません。ですので、COMMUTATOR句と同様に演算子の名前のみNEGATOR句で与えるだけで済みます。

NOT $(x = y)$ という式を $x <> y$ という形に単純化させることが可能なので、否定子があると問い合わせ最適マイザにとって非常に役に立ちます。他の再配置の結果としてNOT操作が挿入されることがありますので、この現象は思ったより頻繁に起こり

ます。

否定子の対は、上記の交代演算子のペアで説明した方法と同じ方法で定義することができます。

35.13.3. RESTRICT

`RESTRICT`句が与えられた場合、それは、その演算子用の制限選択評価関数を指定します。（演算子名ではなく関数名であることに注意してください。） `RESTRICT`句は`boolean`型を返す二項演算子に対してのみ有効です。 制限選択評価の目的は、現在の演算子と特定の定数値についての`WHERE`句

```
column OP constant
```

の条件を満たすテーブル内の行の割合を推測することです。 この形式を持った`WHERE`句によって、どのくらいの行が除外されるのかを通知することで、オプティマイザの手助けをします。（定数値が左項にあったら何が起るかという疑問が生じるかもしれませんが、それは`COMMUTATOR`が存在する理由の1つでもあります。）

新しい制限選択評価関数の記述方法は本章の内容を超えていますが、幸いなことに、数多いユーザ定義の演算子に対し通常いくつかのシステム標準の評価関数を使用すれば事足ります。 システム標準の制限評価関数には下記のものがあります。

- =用の`eqsel`
- <>用の`neqsel`
- <もしくは<=用の`scalarltsel`
- >もしくは>=用の`scalargtsel`

こうした分類になっていることを奇妙に思うかもしれませんが、次のようなことを想定すればそれなりの意味があることが理解できるでしょう。 `=`は特にテーブル内の行の小さな部分を受け付けます。 `<>`は特に小さな部分を除きます。 `<`は、指定した定数がテーブル列の取る値の範囲のどの辺りにあるのかに依存する量の部分を受け付けます（これはよく発生するものです。 `ANALYZE`によって収集される情報で、選択評価関数でできるように作成されます）。 `<=`は、同じ定数との比較において<よりも少しだけ大きな部分を受け付けます。 特に大雑把な推測以上のことを行うのは適切ではありませんので、区別する価値がないと言えるくらい似通った値です。 `>`と`>=`についても同様なことが言えます。

非常に高いもしくは低い選択性を所有する演算子が、まったく等しいか等しくないかにかかわらず、`eqsel`または`neqsel`を使用しないことも往々にして可能です。 例えば、近似等号用の幾何演算子はテーブルのエントリの小部分にのみに合致すると仮定して`eqsel`を使用します。

範囲比較のために数値スカラに変換することに多少の有意性があるデータ型を比較するために、`scalarltsel`と`scalargtsel`を使用することも可能です。 できれば`src/backend/utils/adt/selfuncs.c`の`convert_to_scalar()`のルーチンで理解できるところにデータ型を追加してください（今後、このルーチンは`pg_type`システムカタログの列で識別された、データ型ごとの関数で置き換えられなければならないかもしれませんが、まだ行われていません）。 これを行わなくても動きますが、オプティマイザは本来の推測機能を十分発揮することができません。

さらに`src/backend/utils/adt/geo_selfuncs.c`には、幾何演算子に対する選択評価関数`areasel`、`positionsel`、`contsel`があります。 本章の執筆時点では、これらは単なるスタブですが、ともかく使いたい（あるいは改良したい）こともあるでしょう。

35.13.4. JOIN

`JOIN`句が与えられた場合、それはその演算子用の結合選択評価関数の名前を指定します。（これが演算子名ではなく関数名であることに注意してください。） `JOIN`句は`boolean`型を返す二項演算子に対してのみ有効です。 結合選択評価の目的は、現在の演算子について、`WHERE`句

```
table1.column1 OP table2.column2
```

を満たすテーブルの組み合わせの行の割合を推測することです。 `RESTRICT`句の使用と同様、これはいくつかの取り得る結合手順のうち、どれが最も仕事量が少ないように考えられるのかをオプティマイザに計算させることで、大きなオプティマイザへの援助となります。

以前と同様、本章でも結合選択評価関数の作成方法は説明しません。 しかし適用できるものがあれば、単に標準の評価関数を使用することをお勧めします。

- =用の`eqjoin`
- <>用の`neqjoin`
- <もしくは<=用の`scalarltjoin`
- >もしくは>=用の`scalargtjoin`
- 2次元面積を基にした比較用の`areajoin`

2次元位置を基にした比較用の`positionjoin`sel

2次元含有関係を基にした比較用の`contjoin`sel

35.13.5. HASHES

`HASHES`句が存在する場合、それはシステムに対して、この演算子に基づいた結合にハッシュ結合方法を使っても問題がないことを伝えます。`HASHES`句は`boolean`型を返す二項演算子にのみ有効です。実際には、この演算子はあるデータ型またはデータ型の組み合わせの等価性を表現しなければなりません。

ハッシュ結合の基礎となっている仮定は、結合演算子は左項と右項の値が同じハッシュコードを持つ時にのみ真を返すことができるということです。2つの値が異なるハッシュのバケットに置かれた場合、結合演算子の結果が必ず偽であるという仮定を、結合は暗黙的に行い、それらを比べることをしません。したがって、何らかの等価性を表さない演算子に`HASHES`句を指定することはまったく意味がありません。ほとんどの場合、両辺に同一のデータ型をとる演算子に対してハッシュ機能をサポートすることが現実的です。しかし時として、2つ以上のデータ型に対して互換的なハッシュ関数、つまり、値自体が異なる表現形態を持っていたとしても"等しい"値に対して同一のハッシュコードを生成する関数を設計することもできます。例えば、サイズが異なる整数に対するハッシュでは、この性質を調整することで大変単純になります。

`HASHES`印を付けるためには、結合演算子はハッシュインデックスの演算子族内になければなりません。演算子を作成する時には参照する演算子族がまだ存在しませんので、演算子の作成時にこれは強制されていません。しかし、演算子族が存在しない場合に、この演算子をハッシュ結合で使用しようとすると、実行時に失敗します。システムは、演算子の入力データ型用のデータ型特有のハッシュ関数を検索するために、演算子族を必要とします。もちろん、演算子族を作成する前に適切なハッシュ関数を作成しなければなりません。

ハッシュ関数を準備する時には注意が必要です。マシンに依存することから、ハッシュ結合が適切な処理を行わずに失敗することがあるからです。例えば、データ型が不要な部分を埋めるビットを持つ可能性がある構造体である場合、（推奨する戦略である、他の演算子と関数を作成して、不要なビットが常にゼロになることを保証しない限り、）その構造体全体を単に`hash_any`に渡すことはできません。この他の例として、IEEE浮動小数点標準を満たすマシンでは、マイナス0とプラス0は異なる値（異なるビット列）になりますが、この比較は等価と定義されます。浮動小数点数値がマイナス0を持つ可能性があるのであれば、それがプラス0と同じハッシュコードを確実に生成するような処置が必要です。

ハッシュ結合可能な演算子は、同一演算子族内に存在する交代演算子を持たなければなりません。（2つの入力データ型が同じ場合はその演算子自体が交代演算子となります。異なる場合は関連する等価性演算子となります。）これを満たさないと、演算子の使用時にプランナエラーが発生します。また、複数のデータ型をサポートするハッシュ演算子族に対して、データ型の組み合わせすべてに対する等価性演算子を持たせることを推奨します（必要ではありません）。これにより、より優れた最適化が可能になります。

注意: ハッシュ結合可能演算子の基となる関数は`immutable`もしくは`stable`でなければなりません。`volatile`の場合、システムはその演算子を決してハッシュ結合に使用しません。

注意: ハッシュ結合可能演算子の基となる関数が厳密（`strict`）な場合、その関数は完全、つまり2つの非NULL入力に対して、真または偽を返し、決してNULLを返さないものである必要があります。この規則に従わないと、`IN`操作におけるハッシュ最適化は間違った結果を生成する可能性があります。（特に、標準に従うとNULLが正しい答えになるところで`IN`は偽を返すかもしれません。もしくは、NULLという結果に対する準備をしていないといったエラーを生成するかもしれません。）

35.13.6. MERGES

`MERGES`句が存在する場合、それはシステムに対して、この演算子に基づいた結合にマージ結合方法を使っても問題がないことを伝えます。`MERGES`句は`boolean`型を返す二項演算子にのみ有効です。実際には、演算子がデータ型またはデータ型の組み合わせの等価性を表すものであることが必要です。

マージ結合は、左側のテーブル、右側のテーブルを順序よくソートし、並列にスキャンするという考えに基づいています。したがって、両データ型には完全な順序付け機能が必要であり、結合演算子はソート順で"同じ場所"にある値の対のみを成功したものと認める必要がある場合があります。実際問題として、これは、結合演算子は等価性のような振舞いをしなければならないことを意味しています。しかし、マージ結合は論理的な互換性を持つ別の2つのデータ型を取ることができます。例えば、`smallint`対`integer`の等価性演算子はマージ結合が可能です。両方のデータ型を論理的な互換性を保つ順番にソートする演算子のみが必要です。

`MERGES`印を付けるためには、結合演算子は、`btree`インデックス演算子族の等価性メンバとして存在しなければなりません。演算子を作成する時には参照する演算子族がまだ存在しませんので、演算子の作成時にこれは強制されていません。しかし、対応する演算子族が存在しない限り、実際にマージ結合に使用されることはありません。このように、`MERGES`フラグは、プランナが対応する演算子族を検索すべきかどうかを決定する際のヒントとして動作します。

マージ結合可能な演算子は、同一演算子族内に存在する交代演算子を持たなければなりません。（2つの入力データ型が同じ場合はその演算子自体が交代演算子となります。異なる場合は関連する等価性演算子となります。）これを満たさないと、演算子

の使用時にプランナエラーが発生します。 また、複数のデータ型をサポートする**btree** 演算子族に対して、データ型の組み合わせすべてに対する等価性演算子を持たせることを推奨します（必要ではありません）。 これにより、より優れた最適化が可能になります。

注意: マージ結合可能演算子の背後にある関数は**immutable**もしくは**stable**でなければなりません。 **volatile**の場合、システムはその演算子を決してマージ結合に使用しようとはしません。

[前のページ](#)
ユーザ定義の演算子

[ホーム](#)
[上に戻る](#)

[次のページ](#)
インデックス拡張機能へのインタフェース

35.14. インデックス拡張機能へのインタフェース

これまでのところでは、新しい型や新しい関数、および新しい演算子をどの様に定義するかについて説明してきました。しかしながら、新しい型の列に対するインデックスをまだ作成することができません。このためには、新しいデータ型に対する演算子クラスを定義する必要があります。本節では、複素数を値の絶対値の昇順にソートし格納するB-treeインデックスメソッドを使った新しい演算子クラスについての実行例を用いて、演算子クラス の概念を説明します。

演算子クラスを演算子族にまとめ、意味的に互換性を持つクラス間の関係を表すことができます。1つのデータ型のみが含まれる場合、演算子クラスで十分です。そこでまずこうした状況に注目し、その後で演算子族に戻ります。

35.14.1. インデックスメソッドと演算子クラス

pg_amテーブルには各インデックスメソッド（内部ではアクセスメソッドとして知られています）に対して1つの行が含まれています。テーブルへの通常のアクセスのサポートはPostgreSQLに組み込まれていますが、すべてのインデックスメソッドは、pg_amで記述されています。必要とされるインタフェースルーチンを定義した後、pg_amに行を作成することによって、新しいインデックスメソッドを追加することができます。しかし、この方法についての説明は本章での範囲を超えています（[第52章](#)を参照してください）。

インデックスメソッドのルーチンには、直接的にインデックスメソッドが演算するデータ型の情報は何も与えられていません。代わりに、演算子クラスが、特定のデータ型の操作においてインデックスメソッドを使用する必要がある演算の集合を識別します。演算子クラスという名前の由来は、それらが指定するものの1つにインデックスで使用できる（つまり、インデックススキャン条件に変換できる）WHERE句演算子の集合があるからです。また、演算子クラスは、インデックスメソッドの内部演算で必要な、しかしインデックスで使用できるWHERE句演算子には直接的には対応しない、サポートプロシージャをいくつか指定することができます。

同じ入力データ型およびインデックスメソッドに対して複数の演算子クラスを定義することが可能です。これにより、1つのデータ型に対して、複数のインデックス付けセマンティックの集合を定義することができます。例えば、B-treeインデックスでは、処理するデータ型ごとにソート順を定義する必要があります。複素数データ型では、複素数の絶対値によりデータをソートするB-tree演算子クラスと、実部の数値によりソートするB-tree演算子クラスを持つといった方法は、有用かもしれません。通常は演算子クラスの1つが一般的に最も有用であると判断され、そのデータ型およびインデックスメソッドに対するデフォルトの演算子クラスとして設定されます。

複数の異なるインデックスメソッドに、同一の演算子クラス名を使用することができます（例えば、B-treeとハッシュインデックスメソッドは、両方ともint4_opsという名前の演算子クラスを持つことができます）。ただし、そのような各クラスは独立した実体であり、別々に定義される必要があります。

35.14.2. インデックスメソッドのストラテジ

演算子クラスに関連付けられている演算子は、"ストラテジ番号"により識別されます。"ストラテジ番号"は、演算子クラスのコンテキスト内における各演算子のセマンティクスを識別するためのものです。例えば、B-treeの場合、キーが小さい方から大きい方へ厳密に並んでいなければなりません。したがって、B-treeに関しては、"より小さい"および"以上"のような演算子は興味深いと言えます。PostgreSQLではユーザが演算子を定義できるため、PostgreSQLは演算子の名前（例えば<や>=）を見つけても、その演算子がどのような比較を行うかを判断することはできません。その代わり、インデックスメソッドは"ストラテジ"の集合を定義します。"ストラテジ"は汎用演算子と考えることができます。各演算子クラスは、特定のデータ型およびインデックスセマンティクスの解釈において、実際のどの演算子が各ストラテジに対応しているかを指定します。

[表35-2](#)に示すように、B-treeインデックスメソッドではストラテジを5つ定義します。

表 35-2. B-treeストラテジ

演算	ストラテジ番号
小なり	1
以下	2
等しい	3
以上	4
大なり	5

ハッシュインデックスは等価性のみをサポートします。したがって、[表35-3](#)に示すように、ストラテジを1つのみ定義します。

表 35-3. ハッシュストラテジ

演算	ストラテジ番号
等しい	1

GiSTインデックスはより柔軟です。 固定のストラテジの集合をまったく持ちません。 代わりに、特定のGiST演算子クラス
の"consistent"サポートルーチンが、ストラテジ番号が何を意味するかを解釈します。 例えば、複数の組み込みのGiSTイン
デックス演算子クラスは、2次元幾何オブジェクトをインデックス付けし、[表35-4](#)で示される"R-tree"ストラテジを提供しま
す。 この内4個は2次元に対する（重複、合同、含有、被含有）試験です。 残りの内4個はX方向のみに対する、残り4個はY方
向のみに対する同一の試験を提供します。

表 35-4. GiSTによる2次元の"R-tree"ストラテジ

演算	ストラテジ番号
完全に左側	1
右側にはみ出さない	2
重なる	3
左側にはみ出さない	4
完全に右側	5
同じ	6
含む	7
含まれる	8
上側にはみ出さない	9
完全に下側	10
完全に上側	11
下側にはみ出さない	12

SP-GiSTインデックスは柔軟性という点でGiSTと似ており、固定のストラテジ群を持ちません。 その代わりに、各演算子クラ
スのサポートルーチンが演算子クラスの定義に従ってストラテジ番号を解釈します。 例えば、組み込みの点に対する演算子クラ
スで使用されるストラテジ番号を[表35-5](#)に示します。

表 35-5. SP-GiSTの点に関するストラテジ

演算	ストラテジ番号
厳密に左側	1
厳密に右側	5
同一	6
含有される	8
厳密に下	10
厳密に上	11

GINインデックスは、いずれも固定のストラテジ群を持たないという点で、GiSTおよびSP-GiSTインデックスと似ています。
その代わりに、各演算子クラスのサポートルーチンが演算子クラスの定義に従ってストラテジ番号を解釈します。 例えば、組み
込みの配列に対する演算子クラスで使用されるストラテジ番号を[表35-6](#)に示します。

表 35-6. GIN 配列のストラテジ

演算	ストラテジ番号
重複	1
含有	2
含有される	3

等しい	4

上記の演算子はすべて論理値を返すことに注意してください。 実際、インデックスで использоваться ためにWHEREの最上位レベルで現れなければなりませんので、インデックスメソッド検索演算子として定義された、すべての演算子の戻り値の型はbooleanでなければなりません。 （一部のインデックスアクセスメソッドは、通常論理型の値を返さない順序付け演算子もサポートします。この機能については[項35.14.7](#)で説明します。）

35.14.3. インデックスメソッドのサポートルーチン

ストラテジは通常、システムがインデックスを使う方法を判断するために十分な情報ではありません。 実際には、インデックスメソッドが動作するためには、さらにサポートルーチンを必要とします。 例えばB-treeインデックスメソッドは、2つのキーを比較し、より大きいのか、等しいのか、より小さいのかを決定できなければなりません。 同様に、ハッシュインデックスは、キー値のハッシュコードを計算できなければなりません。 これらの操作はSQLコマンドの条件内で使用される演算子とは対応しません。 これらはインデックスメソッドで内部的に使用される管理用ルーチンです。

ストラテジと同じように、演算子クラスにより、与えられたデータ型およびセマンティックス解釈に対して、どの特定の関数がこれらの各役割を果たすべきであるかが識別されます。 インデックスメソッドは必要な関数の集合を定義し、演算子クラスは、これらをインデックスメソッドで指定された"サポート関数番号"に代入することによって、使用すべき正しい関数を識別します。

[表35-7](#)に示すように、B-treeでは1つのサポート関数が必須ですが、演算子クラス作成者のオプションとして提供されるもう1つのサポート関数を持つことができます。

表 35-7. B-treeサポート関数

関数	サポート番号
2つのキーを比較し、最初のキーが2番目のキーより小さいか、等しいか、大きいかを示す、0未満、0、もしくは0より大きい整数を返します。	1
utils/sortsupport.hに記載した、C言語から呼び出し可能なソートサポート関数のアドレスを返します(省略可能)。	2

[表35-8](#)に示すように、ハッシュインデックスでは単一のサポート関数が必要です。

表 35-8. ハッシュサポート関数

関数	サポート番号
キーのハッシュ値を計算	1

[表35-9](#)に示すように、GiSTインデックスは7つのサポート関数を必要とし、また、8番目は省略可能です。（詳細については[第53章](#)を参照してください。）

表 35-9. GiSTサポート関数

関数	説明	サポート番号
consistent	キーが問い合わせ条件を満たすかどうかを決定します。	1
union	キー集合の和集合を計算します。	2
compress	キーまたはインデックス付けされる値の圧縮表現を計算します。	3
decompress	圧縮されたキーを伸張した表現を計算します。	4
penalty	指定された副ツリーキーを持つ副ツリーに新しいキーを挿入する時のペナルティを計算します。	5
picksplit	ページのどのエントリを新しいページに移動させるかを決定し、結果ページ用の統合キーを計算します。	6
equal	2つのキーを比較し、等しければ真を返します。	7
distance	キーと問い合わせ値との間の距離を決定します（省略可能）。	8

[表35-10](#)に示すように、SP-GiSTインデックスでは5つのサポート関数が必要です。（詳細については[第54章](#)を参照してください）

い。)

表 35-10. SP-GiSTサポート関数

関数	説明	サポート番号
config	演算子クラスに関する基本情報を提供します。	1
choose	新しい値を内部タプルに挿入する方法を決定します。	2
picksplit	値集合を分割する方法を決定します。	3
inner_consistent	ある問い合わせでサブパーティションの検索が必要かどうか決定します。	4
leaf_consistent	キーが問い合わせ修飾子を満たすかどうか決定します。	5

表35-11に示すように、GINインデックスは、4つのサポート関数を必要とし、また、5番目は省略可能です。（詳細については第55章を参照してください。）

表 35-11. GINサポート関数

関数	説明	サ ポ ー ト 番 号
compare	2つのキーを比較し、0未満、0、0より大きな整数を返します。 それぞれ最初のキーの方が大きい、等しい、小さいを示します。	1
extractValue	インデックス付けされる値からキーを抽出します。	2
extractQuery	問い合わせ条件からキーを抽出します。	3
consistent	問い合わせ条件に一致する値かどうかを決定します。	4
comparePartial	問い合わせからの部分キーとインデックスからのキーを比較し、それぞれ、GINがこのインデックス項目を無視しなければならないか、一致する項目として扱わなければならないか、インデックススキャンを中止しなければならないかを示す、ゼロより小さい、ゼロ、ゼロより大きい整数値のいずれかを返します(省略可能)。	5

検索演算子と異なり、サポート関数は特定のインデックスメソッドが想定するデータ型、例えばB-tree用の比較関数の場合、符号付き整数を返します。 同様に各サポート関数に渡す引数の数と型はインデックスメソッドに依存します。 B-Treeとハッシュでは、比較関数とハッシュ処理サポート関数はその演算子クラスに含まれる演算子と同じ入力データ型を取りますが、GIN、SP-GiST、およびGiSTサポート関数のほとんどはそうではありません。

35.14.4. 例

ここまでで概念について説明してきました。 ここで、新しい演算子クラスを作成する有用な例を紹介します。（この例を作業できるように、ソース配布物内のsrc/tutorial/complex.cとsrc/tutorial/complex.sqlにコピーがあります。） この演算子クラスは、複素数をその絶対値による順番でソートする演算子をカプセル化します。 ですので、その名前にcomplex_abs_opsを選びました。 最初に演算子の集合が必要になります。 演算子を定義する処理は項35.12で説明しました。 B-tree上の演算子クラスでは、以下の演算子が必要です。

- 絶対値による、小なり（ストラテジ1）
- 絶対値による、以下（ストラテジ2）
- 絶対値による、等しい（ストラテジ3）
- 絶対値による、以上（ストラテジ4）
- 絶対値による、大なり（ストラテジ5）

比較演算子の関連する集合を定義する時にエラーを発生を最小にする方法は、まず、B-tree比較サポート関数を作成し、その後、他の関数をサポート関数に対する1行のラップとして作成することです。 これにより、境界となる条件で一貫性のない結果を得る確率が減少します。 この手法に従って、まず以下を作成します。

```
#define Mag(c) ((c)->x*(c)->x + (c)->y*(c)->y)

static int
complex_abs_cmp_internal(Complex *a, Complex *b)
{
    double    amag = Mag(a),
              bmag = Mag(b);
```



```

    if (amag < bmag)
        return -1;
    if (amag > bmag)
        return 1;
    return 0;
}

```

これで、小なり関数は以下ようになります。

```

PG_FUNCTION_INFO_V1(complex_abs_lt);

Datum
complex_abs_lt(PG_FUNCTION_ARGS)
{
    Complex    *a = (Complex *) PG_GETARG_POINTER(0);
    Complex    *b = (Complex *) PG_GETARG_POINTER(1);

    PG_RETURN_BOOL(complex_abs_cmp_internal(a, b) < 0);
}

```

他の4関数での違いは、内部関数の結果とゼロとをどのように比べるかだけです。

次に、関数と、この関数に基づく演算子をSQLで宣言します。

```

CREATE FUNCTION complex_abs_lt(complex, complex) RETURNS bool
AS 'filename', 'complex_abs_lt'
LANGUAGE C IMMUTABLE STRICT;

CREATE OPERATOR < (
    leftarg = complex, rightarg = complex, procedure = complex_abs_lt,
    commutator = >, negator = >=,
    restrict = scalarlttsel, join = scalarltjoinssel
);

```

正しく交代演算子と否定演算子を指定する他、適切な制限選択性関数と結合関数を指定することが重要です。 さもないと、オプティマイザはインデックスを効率的に使用することができません。 小なり、等価、大なりの場合に異なる選択性関数を使用しなければならないことに注意してください。

他にも注意すべきことがここで発生します。

- 例えば、`complex`型を両オペランドに取る`=`という名前の演算子を1つしか作成できません。 この場合、`complex`用の他の`=`演算子を持てません。 しかし、実際にデータ型を作成しているとしたら、おそらく、複素数の（絶対値の等価性ではない）通常の等価性演算を行う`=`を欲するでしょう。 この場合、`complex_abs_eq`用の演算子名に別の名前を使用しなければなりません。
- PostgreSQLでは異なる引数のデータ型であれば同じSQL名の演算子を使うことができますが、Cでは1つの名前で1つのグローバル関数が使えただけです。 ですから、C関数は`abs_eq`のような単純な名前にするべきではありません。 通常は、他のデータ型の関数と衝突しないように、C関数名にデータ型名を入れておくことを勧めます。
- `abs_eq`関数のSQL名は、PostgreSQLが引数のデータ型によって同じ名前を持つ他のSQL関数から区別してくれることを期待して作ることができます。 ここでは例を簡単にするために、関数にCレベルとSQLレベルで同じ名前を与えています。

次のステップは、B-treeに必要なサポートルーチンの登録です。 これを実装するCコードは、演算子関数と同じファイルに入っています。 以下は、関数をどのように宣言するかを示します。

```

CREATE FUNCTION complex_abs_cmp(complex, complex)
RETURNS integer
AS 'filename'
LANGUAGE C IMMUTABLE STRICT;

```

これまでで、必要な演算子およびサポートルーチンを持つようになりました。 最後に演算子クラスを作成することができます。

```

CREATE OPERATOR CLASS complex_abs_ops
DEFAULT FOR TYPE complex USING btree AS
    OPERATOR          1          < ,
    OPERATOR          2          <= ,
    OPERATOR          3          = ,
    OPERATOR          4          >= ,
    OPERATOR          5          > ,
    FUNCTION           1          complex_abs_cmp(complex, complex);

```

これで終わりです！ これで`complex`列にB-treeインデックスを作って使用することが可能になったはずです。

以下のように、演算子エントリをより冗長に記述することができます。

```

OPERATOR          1          < (complex, complex) ,

```

しかし、演算子が、演算子クラスの定義と同一のデータ型を取る場合、このような記述をする必要はありません。

上記の例は、ユーザがこの新しい演算子クラスを `complex` データ型のデフォルトの B-tree 演算子クラスにしようとしていると仮定しています。このようにしない場合、`DEFAULT` という単語を取り除いてください。

35.14.5. 演算子クラスと演算子族

これまでは暗黙的に、演算子クラスは1つのデータ型のみを扱うものと仮定してきました。確かに特定のインデックス列にはたった1つのデータ型しかあり得ませんが、異なるデータ型の値とインデックス列の比較を行うインデックス操作はよく役に立ちます。また、演算子クラスと関連したデータ型を跨る演算子を使用できる場合、他のデータ型は独自の関連した演算子クラスを持つことがよくあります。SQL 問い合わせを最適化する際にプランナを補助することができますので、関連したクラスを明示的に関連付けることは（どのように動作するかに関する知識をプランナは多く持ちますので、特に B-Tree 演算子クラスで）有用です。

こうした要望に応えるために PostgreSQL は演算子族という概念を使用します。演算子族は1つ以上の演算子クラスから構成されます。また、演算子族全体に属するが、演算子族内の個々のクラスには属さないインデックス可能演算子や対応するサポート関数を含めることもできます。こうした演算子や関数を、特定のクラスに束縛されていないことから、演算子族内で"自由"であると呼びます。通常、各演算子クラスは1つのデータ型演算子を持ちますが、データ型を跨る演算子は演算子族内で自由になります。

演算子族内の演算子と関数はすべて、意味的な互換性を持たなければなりません。この互換性についての必要条件はインデックスメソッドによって設定されます。このため、演算子族の特定の部分集合を演算子クラスとして選び出す方法に疑問を持つかもしれません。実際多くの目的では、クラスの分類は不適切で、演算子族が唯一の興味深いグループ化です。演算子クラスを定義する理由は、どれだけ多くの演算子族が何らかのインデックスをサポートするために必要かを指定することです。ある演算子クラスを使用するインデックスが存在する場合、演算子クラスはそのインデックスを削除しない限り削除することができます。しかし、演算子族の他の部分、すなわち、他の演算子クラスや自由な演算子を削除することができます。したがって、演算子クラスは、特定のデータ型に対するインデックスを操作する上で理論上必要となる最少の演算子と関数の集合を含むように指定すべきです。そして、関連するが基本的なものではない演算子を演算子族の自由なメンバとして追加することができます。

例えば PostgreSQL には `integer_ops` という組み込みの B-Tree 演算子族があります。ここには `bigint (int8)`、`integer (int4)`、`smallint (int2)` 型の列上へのインデックスにそれぞれ対応した `int8_ops`、`int4_ops`、`int2_ops` という演算子クラスが含まれています。また、上記の型の内任意の2つの型を比較できるように、この演算子族にはデータ型を跨る比較演算子も含まれます。このため、上記の型のいずれかに対するインデックスを他の型の値との比較の際に使用することができます。この演算子族は以下の定義により多重化されています。

```
CREATE OPERATOR FAMILY integer_ops USING btree;

CREATE OPERATOR CLASS int8_ops
DEFAULT FOR TYPE int8 USING btree FAMILY integer_ops AS
-- standard int8 comparisons
OPERATOR 1 < ,
OPERATOR 2 <= ,
OPERATOR 3 = ,
OPERATOR 4 >= ,
OPERATOR 5 > ,
FUNCTION 1 btint8cmp(int8, int8) ,
FUNCTION 2 btint8sortsupport(internal) ;

CREATE OPERATOR CLASS int4_ops
DEFAULT FOR TYPE int4 USING btree FAMILY integer_ops AS
-- standard int4 comparisons
OPERATOR 1 < ,
OPERATOR 2 <= ,
OPERATOR 3 = ,
OPERATOR 4 >= ,
OPERATOR 5 > ,
FUNCTION 1 btint4cmp(int4, int4) ,
FUNCTION 2 btint4sortsupport(internal) ;

CREATE OPERATOR CLASS int2_ops
DEFAULT FOR TYPE int2 USING btree FAMILY integer_ops AS
-- standard int2 comparisons
OPERATOR 1 < ,
OPERATOR 2 <= ,
OPERATOR 3 = ,
OPERATOR 4 >= ,
OPERATOR 5 > ,
FUNCTION 1 btint2cmp(int2, int2) ,
FUNCTION 2 btint2sortsupport(internal) ;

ALTER OPERATOR FAMILY integer_ops USING btree ADD
-- cross-type comparisons int8 vs int2
OPERATOR 1 < (int8, int2) ,
OPERATOR 2 <= (int8, int2) ,
OPERATOR 3 = (int8, int2) ,
```

```

OPERATOR 4 >= (int8, int2) ,
OPERATOR 5 > (int8, int2) ,
FUNCTION 1 btint82cmp(int8, int2) ,

-- cross-type comparisons int8 vs int4
OPERATOR 1 < (int8, int4) ,
OPERATOR 2 <= (int8, int4) ,
OPERATOR 3 = (int8, int4) ,
OPERATOR 4 >= (int8, int4) ,
OPERATOR 5 > (int8, int4) ,
FUNCTION 1 btint84cmp(int8, int4) ,

-- cross-type comparisons int4 vs int2
OPERATOR 1 < (int4, int2) ,
OPERATOR 2 <= (int4, int2) ,
OPERATOR 3 = (int4, int2) ,
OPERATOR 4 >= (int4, int2) ,
OPERATOR 5 > (int4, int2) ,
FUNCTION 1 btint42cmp(int4, int2) ,

-- cross-type comparisons int4 vs int8
OPERATOR 1 < (int4, int8) ,
OPERATOR 2 <= (int4, int8) ,
OPERATOR 3 = (int4, int8) ,
OPERATOR 4 >= (int4, int8) ,
OPERATOR 5 > (int4, int8) ,
FUNCTION 1 btint48cmp(int4, int8) ,

-- cross-type comparisons int2 vs int8
OPERATOR 1 < (int2, int8) ,
OPERATOR 2 <= (int2, int8) ,
OPERATOR 3 = (int2, int8) ,
OPERATOR 4 >= (int2, int8) ,
OPERATOR 5 > (int2, int8) ,
FUNCTION 1 btint28cmp(int2, int8) ,

-- cross-type comparisons int2 vs int4
OPERATOR 1 < (int2, int4) ,
OPERATOR 2 <= (int2, int4) ,
OPERATOR 3 = (int2, int4) ,
OPERATOR 4 >= (int2, int4) ,
OPERATOR 5 > (int2, int4) ,
FUNCTION 1 btint24cmp(int2, int4) ;

```

この定義は演算子ストラテジ関数番号とサポート関数番号を"上書き"していることに注意してください。各番号は演算子族内で複数回現れます。特定番号のインスタンスがそれぞれ異なる入力データ型を持つ限り、これは許されます。入力型の両方が演算子クラスの入力型と同じインスタンスは、演算子クラスの主演算子および主サポート関数であり、ほとんどの場合、演算子クラスの自由メンバではなく演算子クラスの一部として宣言しなければなりません。

B-Tree演算子族では、演算子族内のすべての演算子はソート互換、つまり、演算子族でサポートされるデータ型すべてに渡って推移規則、" $A = B$ かつ $B = C$ ならば $A = C$ "および" $A < B$ かつ $B < C$ ならば $A < C$ "を維持しなければなりません。さらに、演算子族内で表現される型の間の暗黙的キャストとバイナリキャストは、関連するソート順序を変更してはいけません。演算子族内の各演算子では、演算子と同じデータ型の2つのデータ型を取るサポート関数が存在しなければなりません。演算子族を完結させること、つまり、データ型の組み合わせそれぞれに対する演算子をすべて含めることを推奨します。各演算子クラスは、自身のデータ型に対してデータ型を跨らない演算子とサポート関数だけを含めなければなりません。

複数データ型のハッシュ演算子族を構築するには、演算子族でサポートされるデータ型それぞれに対する互換性を持つハッシュサポート関数を作成しなければなりません。ここで、互換性とは、関数とその演算子族の等価性演算子で等価であるとみなされる任意の2つの値では同一のハッシュコードが生成されることを保証することを意味します。通常、型が異なる物理表現を持つ場合、これを実現することは困難ですが、実現可能な場合もあります。さらに、暗黙的またはバイナリ変換により、ある演算子族で表現されるデータ型から同じ演算子族で表現されるデータ型に値をキャストしても、計算されたハッシュ値を変更してはいけません。データ型1つに対してサポート関数が1つしか存在しないことに注意してください。等価性演算子ごとに1つではありません。演算子族を完結させること、つまり、データ型の組み合わせそれぞれに対する等価性演算子をすべて含めることを推奨します。各演算子クラスは、自身のデータ型に対してデータ型を跨らない演算子とサポート関数だけを含めなければなりません。

GiST、SP-GiST、GINインデックスではデータ型を跨る操作についての明示的な記法はありません。サポートされる演算子群は単に指定演算子クラスの主サポート関数が扱うことができるものです。

注意: PostgreSQL 8.3より前のバージョンでは演算子族という概念はありませんでした。そのため、インデックスで使用する予定のデータ型を跨る演算子はすべて、インデックスの演算子クラスに結びつけなければなりませんでした。この手法もまだ使用できますが、インデックスの依存性を広げる点、および、両データ型が同一演算子族内で演算子を持つ場合、プランナがデータ型を跨った比較をより効率的に扱うことができる点より、廃止予定です。

35.14.6. システムの演算子クラスに対する依存性

PostgreSQLは演算子クラスを、単にインデックスで利用できるかどうかだけではなく、多くの方式で演算子の性質を推定するために使用します。したがって、データ型の列をインデックス付けするつもりがなくても、演算子クラスを作成した方が良い可能性があります。

具体的には、`ORDER BY`や`DISTINCT`など、値の比較とソートが必要とするSQL機能があります。ユーザ定義のデータ型に対してこの機能を実装するために、PostgreSQLはそのデータ型用のデフォルトのB-tree演算子クラスを検索します。この演算子クラスの"等価判定"メンバが、`GROUP BY`や`DISTINCT`用の値の等価性についてのシステムの意向を定義し、この演算子クラスによって強制されるソート順序が、デフォルトの`ORDER BY`順序を定義します。

また、ユーザ定義型の配列の比較は、デフォルトのB-tree演算子クラスによって定義されるセマンティックに依存します。

データ型用のデフォルトのB-tree演算子クラスが存在しないと、システムはデフォルトのハッシュ演算子クラスを検索します。しかし、この種類の演算子クラスは等価性のみを提供しますので、実際にこれは、配列等価性のサポートだけに対して十分です。

データ型用のデフォルトの演算子クラスが存在しない場合に、こうしたSQL機能をデータ型に使用しようとする、"演算子クラスを識別できなかった"といったエラーとなります。

注意: PostgreSQLバージョン7.4より前までは、ソートやグループ化操作は暗黙的に`=`、`<`、`>`という名前の演算子を使用していました。この新しい、デフォルトの演算子クラスに依存する振舞いによって、特定の名前を持つ演算子の振舞いについて何らかの仮定を立てることを防止しています。

他の重要な点として、ハッシュ演算子族内に現れる演算子がハッシュ結合、ハッシュ集約、関連する最適化の候補となる場合があります。使用するハッシュ関数を識別するため、ここでのハッシュ演算子族は基本的なものです。

35.14.7. 順序付け演算子

一部のインデックスアクセスメソッド（現時点ではGiSTのみ）は順序付け演算子という概念をサポートします。これまで説明してきたものは検索演算子でした。検索演算子は、`WHERE indexed_column operator constant`を満たすすべての行を見つけるために、インデックスを検索可能にするためのものです。一致した行がどの順序で返されるかについては保証がないことに注意してください。反対に、順序付け演算子は返すことができる行集合を限定しますが、その順序を決定します。順序付け演算子は、`ORDER BY indexed_column operator constant`で表される順序で行を返すために、インデックスをスキャン可能にするためのものです。このように順序付け演算子を定義する理由は、その演算子が距離を測るものであれば最近傍検索をサポートすることです。例えば以下のような問い合わせを考えます。

```
SELECT * FROM places ORDER BY location <-> point '(101,456)' LIMIT 10;
```

これは指定した対象地点に最も近い10地点を見つけ出します。`<->`は順序付け演算子ですので、`location`列上のGiSTインデックスは、これを効率的に行うことができます。

検索演算子が論理値結果を返さなければなりませんが、順序付け演算子は普通、距離を表す浮動小数点や数値型など、何らかの他の型を返します。この型は通常、インデックス対象のデータ型と同じにはなりません。異なるデータ型の動作についての固定化された前提を防ぐために、順序付け演算子の定義では、結果データ型のソート順序を指定するB-tree演算子族の名前を必要とします。前節で述べたように、B-tree演算子族はPostgreSQLの順序付け記法を定義します。ですのでこれは自然な表現です。`point`に対する`<->`演算子は`float8`を返しますので、演算子クラスを作成するコマンド内で以下のように指定します。

```
OPERATOR 15 <-> (point, point) FOR ORDER BY float_ops
```

ここで`float_ops`は、`float8`に対する操作を含んだ組込みの演算子族です。この宣言は、インデックスが`<->`演算子の値が増加する方向で行を返すことができることを表しています。

35.14.8. 演算子クラスの特特殊な機能

演算子クラスには、まだ説明していない2つの特殊な機能があります。説明していない主な理由は、最もよく使用するインデックスメソッドでは、これらがあまり有用ではないためです。

通常、演算子を演算子クラス（または演算子族）のメンバとして宣言すると、インデックスメソッドでその演算子を使用し、`WHERE`条件を満たす行の集合を正確に抽出することができます。以下に例を示します。

```
SELECT * FROM table WHERE integer_column < 4;
```

この式は、整数列にB-treeインデックスを使用することにより、正確に満たすことができます。しかし、一致する行へ厳密ではなくとも導く手段としてインデックスが有用である場合があります。例えば、GiSTインデックスで、幾何オブジェクトの境界ボックスのみを格納したとします。その結果、多角形のような長方形でないオブジェクトとの重なりをテストする`WHERE`条件は正確に満たすことができません。もっとも、このインデックスを使用して、対象オブジェクトの境界ボックスに重なる境界ボックスを持つオブジェクトを検索し、さらに、検索されたオブジェクトのみに対して正確に重なるかどうかをテストすること

はできます。この筋書きを適用する場合、インデックスは演算子に対して"非可逆"と言われます。非可逆インデックス検索は、ある行が問い合わせ条件を実際に満足するかしないかの時に`recheck`フラグを返すインデックスメソッドを持つことで実装されます。コアシステムは、そこで有効なマッチとして行が返されるか否かを確認するために、抽出された行に対して元の問い合わせ条件を検査します。この手法はインデックスがすべての必要な行を返すことが保証された上で、元の演算子呼び出しを実行することによって除外することができる、いくつか余分な行を返す可能性がある場合に動作します。非可逆検索を提供するインデックス方式（現時点ではGiST、SP-GiSTおよびGIN）は個々の演算子クラスのサポート関数が`recheck`フラグを設定することを許可します。このためこれは原則的に演算子クラスの機能です。

再度、多角形のような複雑なオブジェクトの境界ボックスのみをインデックスに格納している状況を考えてみてください。この場合、インデックスエントリに多角形全体を格納するのは、それほど有用なことではありません。単に、より単純な`box`型のオブジェクトを格納した方が良いかもしれません。このような状況は、`CREATE OPERATOR CLASS`の`STORAGE`オプションによって表現することができます。例えば、以下のように記述します。

```
CREATE OPERATOR CLASS polygon_ops
  DEFAULT FOR TYPE polygon USING gist AS
  ...
  STORAGE box;
```

現時点では、GiSTインデックスメソッドとGINインデックスメソッドが、列のデータ型と異なる`STORAGE`型をサポートしています。`STORAGE`が使用された場合、GiSTの`compress`および`decompress`サポートルーチンは、データ型を変換する必要があります。GINでは、`STORAGE`型は"キー"の値の型を識別します。通常これはインデックス付けされる列の型とは異なります。例えば、整数配列の列用の演算子クラスは単なる整数をキーとして持つかもしれませんが。GINの`extractValue`および`extractQuery`サポートルーチンが、インデックス付けされた値からキーを取り出す責任をおいます。

35.15. 関連するオブジェクトを拡張としてパッケージ化

PostgreSQLへの有用な拡張は通常、複数のSQLオブジェクトを含んでいます。例えば、新しいデータ型は新しい関数、新しい演算子、おそらく新しいインデックス演算子クラスを必要とします。これらのオブジェクトをすべて単一のパッケージとしてまとめることは、データベース管理を単純化するために役に立ちます。PostgreSQLではこうしたパッケージを拡張とよびます。拡張を定義するためには、少なくとも、拡張のオブジェクトを作成するためのSQLコマンドを含むスクリプトファイル、拡張自身の数個の基本属性を指定する制御ファイルが必要です。また拡張がCコードを含む場合、通常Cコードで構築された共有ライブラリが存在します。これらのファイルがあれば、単純な[CREATE EXTENSION](#)コマンドがそのオブジェクトをデータベース内に読み込みます。

拡張を使用する主な利点は、SQLスクリプトを実行するだけでデータベースに"粗な"なオブジェクトの群をロードできることではなく、PostgreSQLが拡張のオブジェクトをまとめたものと理解できることです。単一の[DROP EXTENSION](#)コマンドでオブジェクトすべてを削除することができます（個々の"アンインストール"スクリプトを保守する必要はありません）。もっと有用なことは、pg_dumpが拡張の個々のメンバオブジェクトを削除してはならないことを把握していることです。代わりにダンプ内にはCREATE EXTENSION コマンドだけが含まれます。これは、古いバージョンよりも多くのまたは異なるオブジェクトを含む可能性がある、拡張の新しいバージョンへの移行を大きく単純化します。しかし、こうしたダンプを新しいデータベースにロードする際には、拡張の制御ファイル、スクリプトファイル、その他のファイルが利用できるようにしておく必要があります。

PostgreSQLはユーザに、拡張全体を削除させる以外に、拡張内に含まれる個々のオブジェクトを削除させません。また、拡張のメンバオブジェクトの定義を変更する（例えば関数ではCREATE OR REPLACE FUNCTIONを介して変更する）ことはできますが、変更した定義はpg_dumpによりダンプされないことに留意してください。こうした変更は通常、同時に拡張のスクリプトファイルにも同じ変更を行った場合のみ認識することができます。（しかし後述するように設定データを持つテーブルに対しては特殊な準備があります。）

また拡張機構は、拡張に含まれるSQLオブジェクトの定義を調整するパッケージ調整スクリプトを準備しています。例えば、拡張のバージョン1.1でバージョン1.0と比べて1つの関数を追加し、他の関数本体を変更する場合、拡張の作成者はこれらの2つの変更のみを行う更新スクリプトを提供することができます。そしてALTER EXTENSION UPDATE コマンドを使用して、これらの変更を適用し、指定されたデータベース内に実際にインストールされた拡張のバージョンが何かを記録します。

拡張のメンバとなり得るSQLオブジェクトの種類を[ALTER EXTENSION](#)で説明します。拡張は1つのデータベースの中でのみ認識されますので、データベース、ロール、テーブル空間などデータベースクラスタ全体のオブジェクトは拡張のメンバにすることができないことに注意してください。（拡張のスクリプトでこうしたオブジェクトを生成することは禁止されていませんが、作成したとしても、拡張の一部として記録されません。）また、テーブルは拡張のメンバになることができますが、インデックスなどそれに付随するオブジェクトは拡張の直接的なメンバとはみなされません。

35.15.1. 拡張のファイル

[CREATE EXTENSION](#) コマンドは各拡張に関して、拡張と同じ名前に.control という拡張子を持つファイル名である必要がある、制御ファイルに依存します。また、このファイルはインストレーションのSHAREDIR/extensionディレクトリ内に存在しなければなりません。また少なくとも1つの、extension--version.sql という命名規約（例えばfoo拡張のバージョン1.0ではfoo--1.0.sql）に従ったSQLスクリプトファイルが存在しなければなりません。デフォルトでは、このスクリプトファイルもSHAREDIR/extensionディレクトリに格納されますが、制御ファイルでスクリプトファイルを別のディレクトリに指定することができます。

拡張の制御ファイルのファイル書式はpostgresql.confファイルと同じです。すなわち、parameter_name = value という代入を1行当たり1つ記述します。空行および#から始まるコメントが許されます。単一の単語または数字ではない値にはすべて引用符で確実にくくってください。

制御ファイルは以下のパラメータを設定することができます。

directory (string)

拡張のSQLスクリプトファイルを含むディレクトリです。絶対パスで指定されていない限り、この名前はインストレーションのSHAREDIRディレクトリからの相対パスになります。デフォルトの動作はdirectory = 'extension' と指定した場合と同じです。

default_version (string)

拡張のデフォルトのバージョン（CREATE EXTENSION でバージョン指定がない場合にインストールされるバージョン）です。これは省略することができますが、その場合VERSION オプションがないCREATE EXTENSION は失敗します。ですので通常省略しようとは思わないでしょう。

`comment (string)`

拡張に関するコメント（任意の文字列）です。この他の方法として、スクリプトファイル内で[COMMENT](#)コマンドを使用してコメントを設定することができます。

`encoding (string)`

スクリプトファイルで使用される文字セット符号化方式です。スクリプトファイルに何らかの非ASCII文字が含まれる場合に指定しなければなりません。指定がなければ、ファイルはデータベース符号化方式であると仮定されます。

`module_pathname (string)`

このパラメータの値でスクリプトファイル内の`MODULE_PATHNAME`の出現箇所が置換されます。設定されていない場合は置換は行われません。通常これは、スクリプトファイル内で共有ライブラリの名前を直接書き込む必要がなくなるように`$libdir/shared_library_name`に設定され、C言語関数では`CREATE FUNCTION`コマンド中で`MODULE_PATHNAME`を使用します。

`requires (string)`

拡張が依存する拡張の名前のリストです。例えば`requires = 'foo, bar'`です。対象の拡張がインストールできるようになる前に、これらの拡張がインストールされていなければなりません。

`superuser (boolean)`

このパラメータが`true`（デフォルト）の場合、スーパーユーザのみが拡張を作成または新しいバージョンに更新することができます。`false`に設定されている場合は、インストール内でコマンドを実行するまたはスクリプトを更新するために必要な権限のみが必要とされます。

`relocatable (boolean)`

拡張を最初に作成した後に拡張により含まれるオブジェクトを別のスキーマに移動することができる場合、拡張は再配置可能です。デフォルトは`false`、つまり、拡張は再配置可能ではありません。詳しくは後で説明します。

`schema (string)`

このパラメータは再配置可能ではない拡張に対してのみ設定することができます。拡張が指名したスキーマのみにロードされ、他にはロードされないことを強制します。詳しくは後で説明します。

主制御ファイル`extension.control`に加え、拡張は`extension--version.control`という形の名前の副制御ファイルを持つことができます。これらを提供する場合は、スクリプトファイルディレクトリに格納しなければなりません。副制御ファイルは主制御ファイルと同じ書式に従います。拡張の対応するバージョンをインストールまたは更新する時、副制御ファイル内で設定されるパラメータはいずれも、主制御ファイルを上書きします。しかし`directory`および`default_version`パラメータは副制御ファイルで設定することはできません。

拡張のSQLスクリプトファイルにはトランザクション制御コマンド（`BEGIN`、`COMMIT`など）およびトランザクションブロックの内側で実行することができないコマンド（`VACUUM`など）を除く任意のSQLコマンドを含めることができます。スクリプトファイルが暗黙的にトランザクションブロック内で実行されるためです。

拡張のSQLスクリプトファイルには、`\echo`から始まる行を含めることができます。この行は拡張の機構では無視されます（コメントとして扱われます）。これは、このスクリプトが`CREATE EXTENSION`（後述のスクリプト例を参照）ではなく`psql`に渡された場合にエラーを発生するために一般的に使用するために用意されたものです。これがないと、ユーザは間違っただけでなく、"まとまっていまい"オブジェクトとして拡張の内容をロードしてしまい、復旧が多少困難な状態になる可能性があります。

スクリプトファイルは指定した符号化方式で認められる任意の文字を含めることができますが、PostgreSQLが制御ファイルの符号化方式が何かを把握する方法がありませんので、制御ファイルにはASCII文字のみを含めなければなりません。実際には、拡張のコメントに非ASCII文字を含めたい場合にのみ、これが問題になります。このような場合には、制御ファイルの`comment`を使用せず、代わりにコメントを設定するためにスクリプトファイル内で`COMMENT ON EXTENSION`を使用することを勧めます。

35.15.2. 拡張の再配置性

ユーザは拡張に含まれるオブジェクトを拡張の作成者が考えていたスキーマとは別のスキーマにロードしたいとよく考えます。再配置性に関して3つのレベルがサポートされます。

- 完全な再配置可能な拡張は、いつでも、データベースにロードされた後であっても、他のスキーマに移動させることができます。これは、自動的にすべてのメンバオブジェクトを新しいスキーマに名前を変更する、`ALTER EXTENSION SET`

SCHEMAを用いて行います。通常これは、拡張がオブジェクトが含まれるスキーマが何かに関して内部的な仮定を持たない場合のみ可能です。また、拡張のオブジェクト（手続き言語など何らかのスキーマに属さないオブジェクトは無視して）はすべて最初に1つのスキーマ内に存在しなければなりません。制御ファイル内で`relocatable = true`と設定することで、完全な再配置可能と印付けます。

- 拡張はインストール処理の間再配置可能ですが、その後再配置することはできません。通常これは、拡張のスクリプトファイルが、SQL関数用の`search_path`属性の設定など、対象のスキーマを明示的に参照する必要がある場合です。こうした拡張では、制御ファイルで`relocatable = false`と設定し、スクリプトファイル内で対象のスキーマを参照するために`@extschema@`を設定してください。この文字列の出現箇所はすべて、スクリプトが実行される前に、実際の対象のスキーマ名に置換されます。ユーザは`CREATE EXTENSION`の`SCHEMA`オプションを使用して対象のスキーマを設定することができます。
- 拡張が再配置をまったくサポートしない場合、制御ファイルで`relocatable = false`を設定し、かつ、`schema`を意図している対象スキーマの名前に設定してください。これは、制御ファイル内で指定されたスキーマと同じ名前が指定されていない限り、`CREATE EXTENSION`の`SCHEMA`オプションの指定を阻止します。この選択は通常、拡張が`@extschema@`を使用して置き換えることができないスキーマ名について内部的な仮定を持つ場合に必要です。`@extschema@`置換機構はこの場合でも使用することができますが、スキーマ名が制御ファイルによって決定されますので、用途は限定されます。

すべての場合において、スクリプトファイルは対象のスキーマを指し示すようにあらかじめ設定した`search_path`を用いて実行されます。つまり`CREATE EXTENSION`は以下と同じことを行います。

```
SET LOCAL search_path TO @extschema@;
```

これによりスクリプトファイルで作成されるオブジェクトを対象のスキーマ内に格納することができます。スクリプトファイルは要望に応じて`search_path`を変更することができますが、一般的には望まれません。`CREATE EXTENSION`の実行後、`search_path`は以前の設定に戻されます。

対象のスキーマは制御ファイル内の`schema`パラメータがあればこのパラメータにより決定されます。このパラメータがなければ、`CREATE EXTENSION`の`SCHEMA`があればこの値で決まり、これ以外の場合は現在のデフォルトのオブジェクト生成用スキーマ（呼び出し元の`search_path`の最初のもの）になります。制御ファイルの`schema`パラメータが使用される時、対象のスキーマが存在しない場合は作成されますが、これ以外の2つの場合ではすでに存在しなければなりません。

何らかの事前に必要な拡張が制御ファイル内の`requires`に列挙されていた場合、それらの対象スキーマが`search_path`の初期設定に追加されます。これにより新しい拡張のスクリプトファイルからそれらのオブジェクトが可視になります。

再配置不可能な拡張は複数スキーマにまたがるオブジェクトを含めることができますが、通常、外部使用を意図したオブジェクトはすべて単一スキーマに格納することが望まれます。この単一スキーマが拡張の対象のスキーマとみなされます。こうした調整は依存する拡張を作成する間、デフォルトの`search_path`設定を都合に合わせて扱います。

35.15.3. 拡張設定テーブル

一部の拡張は、拡張をインストールした後でユーザにより追加または変更される可能性があるデータを持つ設定テーブルを含みます。通常、テーブルが拡張の一部である場合、テーブル定義もその内容も`pg_dump`によりダンプされません。しかしこの振舞いは設定テーブルの場合望まれません。ユーザによってなされたデータ変更はダンプ内に含まれなければなりません。さもないとダンプしリストアした後で拡張の動作が変わってしまいます。

この問題を解消するために、拡張のスクリプトファイルでは設定テーブルとして作成されるテーブルに印を付け、`pg_dump`にテーブルの内容をダンプに含める（定義は含まれません）ようにさせることができます。このためには、以下の例のようにテーブルを作成した後に`pg_extension_config_dump(regclass, text)`関数を呼び出してください。

```
CREATE TABLE my_config (key text, value text);

SELECT pg_catalog.pg_extension_config_dump('my_config', '');
```

任意数のテーブルをこの方法で印付けることができます。

`pg_extension_config_dump`の第2引数が空文字列である場合、テーブルのすべての内容が`pg_dump`によりダンプされます。これは、拡張のスクリプトによって作成された初期段階においてテーブルが空である場合のみ正しいものです。テーブルの中で初期データとユーザが提供したデータが混在する場合、`pg_extension_config_dump`の第2引数においてダンプすべきデータを選択する`WHERE`条件を提供します。以下に例を示します。

```
CREATE TABLE my_config (key text, value text, standard_entry boolean);

SELECT pg_catalog.pg_extension_config_dump('my_config', 'WHERE NOT standard_entry');
```

このようにした後、拡張のスクリプトで作成される行のみで`standard_entry`が確実に真になるようにします。

初期状態で提供される行がユーザによって変更されるようなもっと複雑な状況では、設定テーブルに対するトリガを作成して、変更された行が正しく印付けられることを確実にするように取り扱うことができます。

`pg_extension_config_dump`を再度呼び出すことにより、設定テーブルに関連付いたフィルタ条件を変更することができます。（通常これは拡張の更新スクリプト内で役に立つでしょう。） 設定ファイルからテーブルを取り除くように印付ける方法は、`ALTER EXTENSION ... DROP TABLE`を用いてテーブルを拡張から分離するしかありません。

35.15.4. 拡張の更新

拡張機構の1つの利点は、拡張のオブジェクトを定義するSQLコマンドの更新を簡便に管理する方法を提供していることです。これは、拡張のインストール用スクリプトのリリース版それぞれにバージョン名称またはバージョン番号を関連付けることで行われます。さらに、ユーザにあるバージョンから次のバージョンへ動的にデータベースを更新させることができるようにしたい場合、あるバージョンから次のバージョンまでの間に行われる必要な変更を行う更新スクリプトを提供しなければなりません。更新スクリプトは`extension--oldversion--newversion.sql`というパターンに従った名前（例えば、`foo--1.0--1.1.sql`は`foo`拡張のバージョン1.0からバージョン1.1に変更するコマンドを含みます。）を持たなければなりません。

適切な更新スクリプトが利用可能である場合、`ALTER EXTENSION UPDATE` コマンドはインストール済みの拡張を指定した新しいバージョンへ更新します。更新スクリプトは、`CREATE EXTENSION`がインストール用スクリプト向けに提供する環境と同じ環境で実行されます。具体的には`search_path`は同じ方法で設定され、スクリプトにより作成される新しいオブジェクトはすべて自動的に拡張に追加されます。

拡張が副制御ファイルを持つ場合、更新スクリプトで使用される制御パラメータは、スクリプトの対象の（新しい）バージョンに関連付けられたものになります。

更新機構を使用して、オブジェクトの"粗"集合から拡張に変換するという、特別かつ重大な状況を解消することができます。拡張機構がPostgreSQLに（9.1で）追加されるようになる前では、パッケージ化されずに単に詰めあわされたオブジェクトを作成する拡張モジュールを多くのユーザが作成していました。こうしたオブジェクトを持つデータベースが存在する場合、どのようにすればこれらのオブジェクトを適切にパッケージ化された拡張に変換できるでしょうか？ 削除した後で普通に`CREATE EXTENSION`を行うことも1つの方法ですが、オブジェクトに依存関係がある（例えば拡張により作成されたデータ型のテーブル列が存在する場合など）場合は好まれません。こうした状況を解消する方法は、空の拡張を作成し、`ALTER EXTENSION ADD`を使用して、既存のオブジェクトそれぞれを拡張に関連づけ、最後にパッケージ化されていないリリースに存在しないが現在のバージョンの拡張には存在する新しいオブジェクトを作成するという方法です。`CREATE EXTENSION`は`FROM old_version`オプションでこの状況をサポートします。この場合、通常のインストール用スクリプトは実行されず、代わりに`extension--old_version--target_version.sql`という名前の更新スクリプトが実行されるようになります。`old_version`として使用するダミーのバージョン名の選択は拡張の作成者に任せられていますが、`unpackaged`がよく使われる規約です。拡張形式に更新できるようにしたい過去のバージョンが複数存在する場合、それらを識別できるように複数のダミーバージョン番号を使用してください。

`ALTER EXTENSION`は、要求される更新を実現するために更新スクリプトを連続して実行することができます。例えば`foo--1.0--1.1.sql`と`foo--1.1--2.0.sql`のみが利用可能であるとすると、現在1.0がインストールされている時にバージョン2.0への更新が要求された場合、`ALTER EXTENSION`はこれらを順番に適用します。

PostgreSQLはバージョン名称の特性についてまったく仮定を行いません。例えば1.0の次が1.1であるかどうかを把握しません。これは利用可能なバージョン名をかみ合わせ、もっとも少ない数の更新スクリプトを適用するために必要な経路を続けるだけです。（バージョン名には、`--`を含まず先頭または最後に`-`が付かなければ、任意の文字を取ることができます。）

"ダウングレード"スクリプトを提供することが便利な場合があります。例えば`foo--1.1--1.0.sql`は、バージョン1.1に関連した変更を基に戻すことができます。この場合、ダウングレードスクリプトがより短いパスを生成するために、予期せず適用されてしまう可能性に注意してください。複数のバージョンをまたがって更新する"近道"更新スクリプトと近道の開始バージョンへのダウングレードスクリプトが存在する場合に危険性があります。ダウングレードしてから近道となる更新スクリプトを実行する方が、バージョンを1つずつ進めるよりも少ない処理で済んでしまうかもしれません。ダウングレードスクリプトが取り返しがつかないオブジェクトを何か削除してしまう場合、望まない結果になってしまいます。

想定外の更新経路かどうかを検査するためには、以下のコマンドを使用してください。

```
SELECT * FROM pg_extension_update_paths('extension_name');
```

これは指定した拡張の個々の既知のバージョン名の組み合わせをそれぞれ、元のバージョンから対象のバージョンへ進む時に取られる更新経路順、またはもし利用できる更新経路がなければ`NULL`を付けて、表示します。経路は`--`を区切り文字として使用したテキスト形式で表示されます。配列形式の方が良ければ`regexp_split_to_array(path, '--')`を使用することができます。

35.15.5. 拡張の例

ここでは、SQLのみの拡張の完全な例を示します。"k"と"v"という名称の2つの要素からなる複合型であり、そのスロットには任意の型の値を格納することができるものです。格納の際テキスト以外の値は自動的にテキストに変換されます。

`pair--1.0.sql`スクリプトファイルは以下のようになります。

```
-- complain if script is sourced in psql, rather than via CREATE EXTENSION
¥echo Use "CREATE EXTENSION pair" to load this file. ¥quit
```

```

CREATE TYPE pair AS ( k text, v text );

CREATE OR REPLACE FUNCTION pair(anyelement, text)
RETURNS pair LANGUAGE SQL AS 'SELECT ROW($1, $2)::pair';

CREATE OR REPLACE FUNCTION pair(text, anyelement)
RETURNS pair LANGUAGE SQL AS 'SELECT ROW($1, $2)::pair';

CREATE OR REPLACE FUNCTION pair(anyelement, anyelement)
RETURNS pair LANGUAGE SQL AS 'SELECT ROW($1, $2)::pair';

CREATE OR REPLACE FUNCTION pair(text, text)
RETURNS pair LANGUAGE SQL AS 'SELECT ROW($1, $2)::pair';

CREATE OPERATOR ~> (LEFTARG = text, RIGHTARG = anyelement, PROCEDURE = pair);
CREATE OPERATOR ~> (LEFTARG = anyelement, RIGHTARG = text, PROCEDURE = pair);
CREATE OPERATOR ~> (LEFTARG = anyelement, RIGHTARG = anyelement, PROCEDURE = pair);
CREATE OPERATOR ~> (LEFTARG = text, RIGHTARG = text, PROCEDURE = pair);

```

pair.control 制御ファイルは以下のようになります。

```

# pair extension
comment = 'A key/value pair data type'
default_version = '1.0'
relocatable = true

```

これらの2つのファイルを正しいディレクトリにインストールするためにメイクファイルを作成する必要はほとんどありませんが、以下を含むMakefileを使用することができます。

```

EXTENSION = pair
DATA = pair--1.0.sql

PG_CONFIG = pg_config
PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)

```

このメイクファイルは[項35.16](#)で説明するPGXSに依存します。make install コマンドは制御ファイルとスクリプトファイルをpg_configで報告される正しいディレクトリにインストールします。

ファイルがインストールされた後、[CREATE EXTENSION](#) コマンドを使用してオブジェクトを任意の特定のデータベースにロードしてください。

[前のページ](#)

インデックス拡張機能へのインタフェース

[ホーム](#)
[上に戻る](#)

[次のページ](#)
拡張構築基盤

35.16. 拡張構築基盤

PostgreSQL 拡張モジュールの配布を考えているのであれば、移植可能な構築システムを準備することはかなり難しいものになるかもしれません。このため PostgreSQL インストールは単純な拡張モジュールをすでにインストールされているサーバに対して簡単に構築することができるように、PGXS と呼ばれる拡張向けの構築基盤を提供します。PGXS は主に C コードを含む拡張を意図していますが、SQL のみからなる拡張でも使用することができます。PGXS が PostgreSQL と相互に作用する任意のソフトウェアを構築するために使用できるような万能な構築システムを意図したものではないことに注意してください。これは単に、単純なサーバ拡張用の一般的な構築規則を自動化するものです。より複雑なパッケージでは、独自の構築システムを作成する必要があるかもしれません。

独自の拡張で PGXS 基盤を使用するためには、簡単なメークファイルを作成する必要があります。このメークファイルの中で、いくつか変数を設定し、最後に大域的な PGXS メークファイルをインクルードする必要があります。以下に `isbn_issn` という名称の拡張モジュールを構築する例を示します。このモジュールはいくつかの C コードを含む共有ライブラリ、拡張の制御ファイル、SQL スクリプト、ドキュメントテキストファイルから構成されます。

```
MODULES = isbn_issn
EXTENSION = isbn_issn
DATA = isbn_issn--1.0.sql
DOCS = README.isbn_issn

PG_CONFIG = pg_config
PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)
```

最後の 3 行は常に同じです。ファイルのこの前に変数の設定と独自の make ルールを記載してください。

以下の 3 個の変数の 1 つを構築対象に指定してください。

MODULES

同じ家系のソースファイルから構築される共有ライブラリのリストです。（このリストにはライブラリ接頭辞を含めないでください。）

MODULE_big

複数のソースファイルから構築される共有ライブラリです。（`OBJS` にオブジェクトファイルを列挙します。）

PROGRAM

構築する実行プログラムです。（`OBJS` にオブジェクトファイルを列挙します。）

以下の変数も設定することができます。

EXTENSION

拡張の名前です。各名前に対して、`prefix/share/extension` にインストールされる `extension.control` を提供しなければなりません。

MODULEDIR

DATA および DOCS ファイルのインストール先となるはずの `prefix/share` 副ディレクトリです。（設定がない場合、デフォルトは `EXTENSION` が設定されている場合は `extension` に、設定されていない場合は `contrib` になります。）

DATA

`prefix/share/$MODULEDIR` にインストールされる様々なファイルです。

DATA_built

`prefix/share/$MODULEDIR` にインストールされる、最初に構築しなければならない様々なファイルです。

DATA_TSEARCH

`prefix/share/tsearch_data` 以下にインストールされる様々なファイルです。

DOCS

`prefix/doc/$MODULEDIR` 以下にインストールされる様々なファイルです。

SCRIPTS

`prefix/bin`

にインストールされるスクリプトファイルです（バイナリファイルではありません）。

SCRIPTS_built

`prefix/bin`にインストールされる、最初に構築しなければならないスクリプトファイルです（バイナリファイルではありません）。

REGRESS

リグレッション試験ケース（接尾辞がない）のリストです。後述します。

REGRESS_OPTS

`pg_regress`に渡す追加オプションです。

EXTRA_CLEAN

`make clean`で削除される追加ファイルです。

PG_CPPFLAGS

`CPPFLAGS`に追加されます。

PG_LIBS

`PROGRAM`のリンク行に追加されます。

SHLIB_LINK

`MODULE_big`リンク行に追加されます。

PG_CONFIG

構築対象のPostgreSQLインストレーション用の`pg_config`プログラムへのパスです。（通常は`PATH`内の最初に見つかる`pg_config`が単純に使用されます）

このメイクファイルを`Makefile`として拡張を保管するディレクトリ内に保管してください。その後コンパイルするために`make`を、モジュールをインストールするために`make install`を行うことができます。デフォルトでは、`PATH`の中で最初に見つかる`pg_config`プログラムが対応するPostgreSQLインストレーション用に拡張はコンパイルされ、インストールされます。メイクファイルまたは`make`のコマンドラインのいずれかで`PG_CONFIG`を別の`pg_config`プログラムを指し示すように設定することで、別のインストレーションを使用することができます。

注意
<code>PG_CONFIG</code> の変更は、PostgreSQL 8.3以降に対して構築する場合にのみ動作します。より古いリリースでは、 <code>pg_config</code> 以外の何かを設定しても動作しません。構築対象のインストレーションを選択するように <code>PATH</code> を変更しなければなりません。

`REGRESS`変数に列挙されたスクリプトは、`make install`を実行した後で`make installcheck`によって呼び出すことができる、作成したモジュールのリグレッション試験で使用されます。これが動作するためには、PostgreSQLサーバが実行していなければなりません。`REGRESS`変数に列挙されたスクリプトは、拡張のディレクトリ内の`sql/`という名前の副ディレクトリ内に存在しなければなりません。これらのファイルは`.sql`という拡張子を持たなければなりません。この拡張子はメイクファイル内の`REGRESS`リストには含まれません。また試験ごとに`expected/`という名前の副ディレクトリ内に想定出力を内容として含む、同じシステムに`.out`拡張子を付けた名前のファイルがなければなりません。`make installcheck`は`psql`を用いて各試験スクリプトを実行し、結果出力が想定ファイルに一致するかどうか比較します。何らかの差異は`diff -c`書式で`regression.diffs`に書き出されます。想定ファイルがない試験を実行しようとするとき"問題"として報告されます。このためすべての想定ファイルがあることを確認してください。

ティップ: 想定ファイルを作成する最も簡単な方法は、空のファイルを作成し、試験を実行する（当然差異が報告されます）ことです。`results/`ディレクトリ内で見つかる実際の結果ファイルを確認し、テストの想定結果と合致するのであれば、`expected/`にコピーしてください。

第 36 章 トリガ

目次

- 36.1. [トリガ動作の概要](#)
- 36.2. [データ変更の可視性](#)
- 36.3. [Cによるトリガ関数の作成](#)
- 36.4. [完全なトリガの例](#)

本章ではトリガ関数の作成に関する一般的な情報を示します。トリガ関数は、PL/pgSQL ([第39章](#))、PL/Tcl ([第40章](#))、PL/Perl ([第41章](#))、PL/Python ([第42章](#))など、利用可能な手続き言語のほとんどで作成することができます。本章を読んだ後、好みの手続き言語に関する章を参照して、トリガ作成に関する言語特有の詳細を確認すべきです。

また、C言語でトリガ関数を作成することができます。しかし、ほとんどの方は、手続き言語のいずれかで作成する方が簡単であることに気づくでしょう。現時点では、普通のSQL関数言語ではトリガ関数を作成することはできません。

36.1. トリガ動作の概要

トリガとは、データベースが、ある特定の操作が行われた時に常に自動的に実行しなければならない特定の機能に関する規定です。トリガはテーブルとビューの双方に付与することができます。

テーブル上では、トリガを `INSERT`、`UPDATE` または `DELETE` 操作の前後に、行を変更する度、あるいは SQL 文ごとに行うように定義することができます。 `UPDATE` トリガについては、特定のカラムが `UPDATE` 文の `SET` 句の対象になった時のみ発動するよう設定することができます。また、トリガは `TRUNCATE` 文についても実行できます。トリガイベントが起こると、トリガ関数がそのイベントを扱う適切な時点で呼び出されます。

ビュー上では、トリガを `INSERT`、`UPDATE` または `DELETE` 操作の代わりに実行するものとして定義できます。 `INSTEAD OF` トリガは、ビュー内の変更を行うために必要となる行それぞれに対して一度発行されます。元になっているテーブルへの必要な変更の実施、そして必要に応じて、ビュー上で見えるであろう変更された行を返却するのは、トリガ関数の責任です。ビューへのトリガは、SQL 文ごとに、`INSERT`、`UPDATE` または `DELETE` 操作の前後で実行させるよう定義することができます。

トリガ関数は、トリガ自体が作成される前までに定義しておく必要があります。トリガ関数は、引数を取らない、`trigger` 型を返す関数として宣言される必要があります（トリガ関数は、通常の関数で使われる引数という形ではなく、`TriggerData` 構造体で入力を受け取ります）。

適切なトリガ関数が作成されると、[CREATE TRIGGER](#) を使用してトリガを構築することができます。同一のトリガ関数を複数のトリガに使用することができます。

PostgreSQL は、行単位のトリガと文単位のトリガの両方を提供します。行単位のトリガでは、トリガを発行した文によって影響を受ける行ごとにトリガ関数が呼び出されます。反対に、文単位のトリガでは、適切な文が実行された時に、その文で何行が影響を受けたかどうかは関係なく、一度だけ呼び出されます。特に、行に影響を与えない文であっても、適切な文単位のトリガがあれば実行されます。この2種類のトリガはそれぞれ行レベルトリガと文レベルトリガと呼ばれることがあります。 `TRUNCATE` に対するトリガは文レベルトリガのみに定義することができます。ビューでは、文レベルでのみ、処理の前後での実行をするよう定義することができます。一方、`INSERT`、`UPDATE` または `DELETE` の代わりに実行するトリガは行レベルでのみ定義できます。

また、トリガはそれらが操作の前、後または代替りのどれで実行されるかに応じて分けられます。これらはそれぞれ `BEFORE` トリガ、`AFTER` トリガ、そして `INSTEAD OF` トリガと呼ばれます。文レベルの `BEFORE` トリガは、文が何かを始める前に自然に発行され、文レベルの `AFTER` トリガは文の本当に最後に発行されます。これらのタイプのトリガはテーブルまたはビューで定義されるでしょう。行レベルの `BEFORE` トリガは、特定の行が操作される直前に発行され、行レベルの `AFTER` トリガは文の終わり（ただし、全ての文レベルの `AFTER` トリガの前）に発行されます。これらのタイプのトリガはテーブルにのみ定義されるでしょう。行レベルの `INSTEAD OF` トリガはビューにのみ定義され、ビュー上の行ごとに操作が必要と判断された場合に即座に発行されます。

文単位のトリガによって呼び出されるトリガ関数は常に `NULL` を返さなければなりません。行単位のトリガによって呼び出されるトリガ関数は呼び出し元のエクゼキュータにテーブル行（`HeapTuple` 型の値）を返すように選択することができます。操作前に発行された行レベルのトリガでは以下の選択肢があります。

- `NULL` を返して、現在の行への操作を飛ばすことができます。これは、エクゼキュータにトリガの元になった行レベルの操作（特定のテーブル行の挿入、更新、削除）を行わないよう指示します。
- 行レベルの `INSERT` および `UPDATE` トリガの場合のみ、返される行が挿入される、もしくは実際に更新される行になります。これにより、トリガ関数で、挿入される行もしくは更新される行を変更することができます。

これらの動作をさせたくない行レベルの `BEFORE` トリガについては、渡された行（つまり、`INSERT` および `UPDATE` トリガでは `NEW` 行、`DELETE` の場合は `OLD` 行）と同じ行結果を返すように気を付ける必要があります。

行レベルの `INSTEAD OF` トリガは、ビューの元となった元テーブルのデータをまったく変更しないことを表す `NULL`、または、渡されたビューの行（`INSERT` と `UPDATE` 操作の場合 `NEW` 行、`DELETE` 操作の場合 `OLD` 行）を返さなければなりません。非 `NULL` の戻り値は、そのトリガがビューにおいて必要なデータ変更を実行したことを通知するために使用されます。これにより影響を受けた行数を数えるカウンタは増加されます。 `INSERT` と `UPDATE` 操作では、トリガは戻す前に `NEW` 行を変更することができます。これは `INSERT RETURNING` または `UPDATE RETURNING` で返されるデータを変更しますので、ビューが提供されたデータと正確に同じ結果を返さない場合に有益です。

操作の後に発生する行レベルトリガでは戻り値は無視されますので、これらは `NULL` を返すことができます。

同一リレーション、同一イベントに対して1つ以上のトリガが定義された場合、トリガはその名前のアルファベット順に発生します。 `BEFORE` トリガと `INSTEAD OF` トリガの場合では、各トリガで返される、変更された可能性がある行が次のトリガの入力となります。もし、ある `BEFORE` トリガや `INSTEAD OF` トリガが `NULL` を返したら、(いまのところ)操作はその行で中断し、残りのトリガは発生しません。

トリガ定義は、トリガを発動するかどうかをWHEN句の論理条件で指定することも可能です。行レベルトリガにおいて、WHEN条件は行の列の古い値と(あるいは)新しい値を検索することができます。(あまり有用ではありませんが、文レベルトリガでもWHEN条件で同じことができます。)BEFOREトリガでは、実質的にトリガ関数の開始時と同じ条件で検査できるように、WHEN条件の評価が関数の実施直前になされます。しかしAFTERトリガでは、WHEN条件の評価は行の更新直後に行われ、文の終わり(コミット時)にトリガを発動するためのイベントを待ち行列に入れるかどうかを決めます。そのため、あるAFTERトリガのWHEN条件が真を返さなかった場合は、イベントを待ち行列に入れる必要も文の終わりに行を再取得する必要もありません。これは、大量の行の変更が発生するけれども、トリガがその内の少数の行に対してのみ発動させる必要がある、といった文の処理速度を大幅に上げる効果があります。INSTEAD OFトリガはWHEN条件をサポートしていません。

通常、行レベルのBEFOREトリガは、挿入あるいは更新される予定のデータの検査や変更のために使用されます。例えば、BEFOREトリガは、timestamp型の列に現在時刻を挿入するために、あるいは行の2つの要素の整合性を検査するために使用される可能性があります。行レベルのAFTERトリガは、ほとんど常識的に他のテーブルに更新を伝播させるために、あるいは他のテーブルとの整合性を検査するために使用されます。こうした仕事の切り分け理由は、AFTERトリガは行の最終値を見ることができ、BEFOREトリガは見ることができないという点です。トリガをBEFOREにするかAFTERにするかを決める時に特別な理由がないのであれば、操作の情報を行が終わるまで保持する必要がない分、BEFOREを使う方が効率的です。

トリガ関数がSQLコマンドを処理する場合、これらの問い合わせがトリガを再度発行することがあります。これはカスケードされたトリガと呼ばれます。カスケードの段数に直接的な制限はありません。カスケードの場合、同じトリガを再帰的に呼び出すことが可能です。例えば、INSERTトリガで同じテーブルに追加の行を挿入する問い合わせが実行された場合、その結果としてINSERTトリガが再度発行されます。こうした状況で無限再帰を防ぐのは、トリガプログラマの責任です。

トリガを定義する時、そのトリガ用の引数を指定することができます。トリガ定義に引数を含めた目的は、似たような要求の異なるトリガに同じ関数を呼び出すことができるようにすることです。例えば、2つの列名を引数とし、片方に現在のユーザをもう片方に現在のタイムスタンプを取る、汎化トリガ関数があるとします。適切に作成すれば、この関数が特定のトリガの発行元となるテーブルに依存することはなくなります。同じ関数を使用して、例えば、トランザクションテーブルに作成記録を自動的に登録させるために、適切な列を持つ任意のテーブルのINSERTイベントに使用することができます。また、UPDATEとして定義すれば、最終更新イベントを追跡するために使用することも可能です。

トリガをサポートするプログラミング言語はそれぞれ独自の方法で、トリガ関数で利用できるトリガの入力データを作成します。この入力データにはトリガイベント種類(例えばINSERTやUPDATEなど、CREATE TRIGGERで指定された全ての引数)が含まれます。行レベルトリガの入力データには、INSERTおよびUPDATEトリガの場合はNEW行が、UPDATEおよびDELETEトリガの場合はOLD行が含まれます。文レベルトリガには現在、文によって変更される個々の行を検査するための手段がありません。

[前のページ](#)
トリガ

[ホーム](#)
[上に戻る](#)

[次のページ](#)
データ変更の可視性

36.2. データ変更の可視性

トリガ関数内で SQL コマンドを実行し、このコマンドがトリガの元となったテーブルにアクセスする場合、データの可視性規則に注意する必要があります。この規則が、SQL コマンドがトリガの発行原因となったデータ変更を見ることができるかどうかを決定するからです。簡単に以下に示します。

- 文レベルトリガは次に示す簡単な可視性規則に従います。文によってなされた変更は、文の前に呼び出される文レベルトリガでは不可視です。一方、文レベルの `AFTER` トリガでは全ての変更が可視です。
- 当然ながら行レベルの `BEFORE` トリガ内の SQL コマンドでは、トリガの発生原因となったデータ変更（挿入、更新、削除）はまだ発生していませんので、可視ではありません。
- しかし、行レベルの `BEFORE` トリガで実行される SQL コマンドは、その外側のコマンドで以前に処理された行へのデータ変更の影響を見ることになるでしょう。これらの変更イベントの順序は一般的に予測できませんので、注意が必要です。複数行に影響する SQL コマンドはどのような順番でもその行を更新することができます。
- 同様に、行レベルの `INSTEAD OF` トリガは、同じ外側のコマンドで以前に処理された `INSTEAD OF` トリガによる変更結果を見ることになるでしょう。
- 行レベルの `AFTER` トリガが発生すると、その外側のコマンドによってなされた全ての変更は既に完了していますので、呼び出されたトリガ関数から可視になります。

もし、あなたのトリガが標準的な手続き型言語のいずれかで記述されている時、上記の可視性は関数が `VOLATILE` で定義されている場合のみ適用されます。 `STABLE`、もしくは `IMMUTABLE` で定義されている関数は、どのようなケースにおいても、呼び出しコマンドによる変更は見ないでしょう。

データ可視性規則に関する詳細は [項43.4](#) にあります。 [項36.4](#) の例にこの規則を示します。

36.3. C によるトリガ関数の作成

本節ではトリガ関数とのインタフェースについて低レベルな詳細を説明します。この情報はC言語でトリガ関数を作成する時のみ必要です。高レベルな言語で作成すれば、こうした詳細は代わりに扱ってもらえます。たいていの場合、Cでトリガを作成する前に手続き言語を使用することを検討すべきです。各手続き言語の文書で、その言語を使用したトリガ言語の作成方法を説明します。

トリガ関数は"version 1"関数マネージャインタフェースを使わなくてはなりません。

関数がトリガマネージャから呼び出される時は、通常の引数が渡されるのではなく、TriggerData構造体を指す"context"ポインタが渡されます。C関数は、トリガマネージャから呼び出されたのかどうかを以下のマクロを実行することで検査することができます。

```
CALLED_AS_TRIGGER(fcinfo)
```

これは以下に展開されます。

```
((fcinfo)->context != NULL && IsA((fcinfo)->context, TriggerData))
```

もしこれが真を返す場合、fcinfo->contextをTriggerData *型にキャストし、指されたTriggerData構造体を使用することは安全です。その関数は、TriggerData構造体やそれが指すどのようなデータも変更してはいけません。

struct TriggerDataはcommands/trigger.hの中で定義されています。

```
typedef struct TriggerData
{
    NodeTag      type;
    TriggerEvent tg_event;
    Relation     tg_relation;
    HeapTuple    tg_trigtuple;
    HeapTuple    tg_newtuple;
    Trigger      *tg_trigger;
    Buffer        tg_trigtuplebuf;
    Buffer        tg_newtuplebuf;
} TriggerData;
```

メンバは下記のように定義されています。

type

常にT_TriggerDataです。

tg_event

その関数が呼び出されたイベントを記述します。tg_eventを調べるためには下記のマクロを使うことができます。

```
TRIGGER_FIRED_BEFORE(tg_event)
```

トリガがBEFOREで発行された場合真を返します。

```
TRIGGER_FIRED_AFTER(tg_event)
```

トリガがAFTERで発行された場合真を返します。

```
TRIGGER_FIRED_INSTEAD(tg_event)
```

トリガがINSTEAD OFで発行された場合真を返します。

```
TRIGGER_FIRED_FOR_ROW(tg_event)
```

トリガが行レベルのイベントで発行された場合真を返します。

```
TRIGGER_FIRED_FOR_STATEMENT(tg_event)
```

トリガが文レベルのイベントで発行された場合真を返します。

```
TRIGGER_FIRED_BY_INSERT(tg_event)
```

トリガがINSERTコマンドで発行された場合真を返します。

```
TRIGGER_FIRED_BY_UPDATE(tg_event)
```

トリガがUPDATEコマンドで発行された場合真を返します。

TRIGGER_FIRED_BY_DELETE(tg_event)

トリガがDELETEコマンドで発行された場合真を返します。

TRIGGER_FIRED_BY_TRUNCATE(tg_event)

トリガがTRUNCATEコマンドで発行された場合真を返します。

tg_relation

トリガの発行元のリレーションを記述する構造体へのポインタです。この構造体についての詳細は、utils/rel.hを参照してください。最も興味深いのは、tg_relation->rd_att（リレーションタブルの記述子）とtg_relation->rd_rel->relnameです（リレーション名、これはchar*ではなくNameDataです。名前のコピーが必要な場合は、char*を得るためにSPI_getrelname(tg_relation)を使用してください）。

tg_trigtuple

トリガが発行された行へのポインタです。これは挿入される、削除される、あるいは更新される行です。もしINSERT/DELETEでこのトリガが発行された時、この行を別のもので置き換えたくない（INSERTの場合）場合や、その操作を飛ばしたくない場合は、これをこの関数から返してください。

tg_newtuple

トリガがUPDATEで発行された場合は、行の新しいバージョンへのポインタです。INSERTもしくはDELETEの場合は、NULLです。UPDATEイベントの時、この行を別のもので置き換えたくない場合や操作を飛ばしたくない場合は、これをこの関数から返してください。

tg_trigger

以下のようにutils/reltrigger.hで定義された、Trigger構造体へのポインタです。

```
typedef struct Trigger
{
    Oid          tgoid;
    char         *tgname;
    Oid          tgfoid;
    int16        tgtype;
    char         tgenabled;
    bool         tgisinternal;
    Oid          tgconstrrelid;
    Oid          tgconstrindid;
    Oid          tgconstraint;
    bool         tgdeferrable;
    bool         tginitdeferred;
    int16        tgnargs;
    int16        tgnattr;
    int16        *tgattr;
    char         **tgargs;
    char         *tgqual;
} Trigger;
```

ここで、tgnameがトリガの名前、tgnargsがtgargs内の引数の数、tgargsはCREATE TRIGGER文で指定された引数へのポインタの配列です。他のメンバは内部でのみ使用されます。

tg_trigtuplebuf

tg_trigtupleを含むバッファです。そうしたタプルが存在しない場合やディスクバッファ内に格納されていない場合はInvalidBufferです。

tg_newtuplebuf

tg_newtupleを含むバッファです。そうしたタプルが存在しない場合やディスクバッファ内に格納されていない場合はInvalidBufferです。

トリガ関数はHeapTupleポインタもしくはNULLポインタ（SQLのNULLではありません。したがって、isNullは真にはなりません）のどちらかを返さなければなりません。操作対象の行を変更したくない場合は、注意して、tg_trigtupleかtg_newtupleの適切な方を返してください。

[前のページ](#)

データ変更の可視性

[ホーム](#)

[上に戻る](#)

[次のページ](#)

完全なトリガの例

36.4. 完全なトリガの例

C言語で作成したトリガ関数に関するとても簡単な例をここに示します（手続き言語で作成したトリガの例は、その手続き言語の文書に記載されています。）

`trigf` 関数は、`ttest` テーブル内にある行数を報告し、問い合わせが`x`にNULL値を挿入しようとしていた場合は、その操作を飛ばします（つまり、このトリガは、トランザクションを中断させないNOT NULL制約のような動作をします。）

まず、以下のようにテーブルを定義します。

```
CREATE TABLE ttest (
    x integer
);
```

以下がトリガ関数のソースコードです。

```
#include "postgres.h"

#include "executor/spi.h"          /* これはSPIを使用する場合に必要なもの */
#include "commands/trigger.h"     /* これはトリガに必要なもの */
#include "utils/rel.h"            /* これはリレーションに必要なもの */

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

extern Datum trigf(PG_FUNCTION_ARGS);

PG_FUNCTION_INFO_V1(trigf);

Datum
trigf(PG_FUNCTION_ARGS)
{
    TriggerData *trigdata = (TriggerData *) fcinfo->context;
    TupleDesc   tupdesc;
    HeapTuple   rettupple;
    char        *when;
    bool        checknull = false;
    bool        isnull;
    int         ret, i;

    /* トリガとして呼び出されたかどうかを確認 */
    if (!CALLED_AS_TRIGGER(fcinfo))
        elog(ERROR, "trigf: not called by trigger manager");

    /* エグゼキュータに返すタプル */
    if (TRIGGER_FIRED_BY_UPDATE(trigdata->tg_event))
        rettupple = trigdata->tg_newtuple;
    else
        rettupple = trigdata->tg_trigtuple;

    /* NULL値をチェック */
    if (!TRIGGER_FIRED_BY_DELETE(trigdata->tg_event)
        && TRIGGER_FIRED_BEFORE(trigdata->tg_event))
        checknull = true;

    if (TRIGGER_FIRED_BEFORE(trigdata->tg_event))
        when = "before";
    else
        when = "after ";

    tupdesc = trigdata->tg_relation->rd_att;

    /* SPIマネージャに接続 */
    if ((ret = SPI_connect()) < 0)
        elog(ERROR, "trigf (fired %s): SPI_connect returned %d", when, ret);

    /* テーブル中の行数を取得 */
    ret = SPI_exec("SELECT count(*) FROM ttest", 0);

    if (ret < 0)
```

```

        elog(ERROR, "trigf (fired %s): SPI_exec returned %d", when, ret);

/* count(*)はint8を返す。変換に注意してください*/
i = DatumGetInt64(SPI_getbinval(SPI_tuptable->vals[0],
                                SPI_tuptable->tupdesc,
                                1,
                                &isnull));

    elog (INFO, "trigf (fired %s): there are %d rows in ttest", when, i);

    SPI_finish();

    if (checknull)
    {
        SPI_getbinval(rettuple, tupdesc, 1, &isnull);
        if (isnull)
            rettupple = NULL;
    }

    return PointerGetDatum(rettuple);
}

```

ソースコードをコンパイル ([項35.9.6](#)を参照してください) した後に、以下の様に関数とトリガを宣言します。

```

CREATE FUNCTION trigf() RETURNS trigger
AS 'filename'
LANGUAGE C;

CREATE TRIGGER tbefore BEFORE INSERT OR UPDATE OR DELETE ON ttest
FOR EACH ROW EXECUTE PROCEDURE trigf();

CREATE TRIGGER tafter AFTER INSERT OR UPDATE OR DELETE ON ttest
FOR EACH ROW EXECUTE PROCEDURE trigf();

```

これで、トリガの操作を確認することができます。

```

=> INSERT INTO ttest VALUES (NULL);
INFO:  trigf (fired before): there are 0 rows in ttest
INSERT 0 0

```

-- 挿入操作は飛ばされ、また、AFTERトリガも発行されません。

```

=> SELECT * FROM ttest;
 x
---
(0 rows)

```

```

=> INSERT INTO ttest VALUES (1);
INFO:  trigf (fired before): there are 0 rows in ttest
INFO:  trigf (fired after ): there are 1 rows in ttest
                                ^^^^^^^^

```

可視性の説明を思い出してください。

```

INSERT 167793 1
vac=> SELECT * FROM ttest;
 x
---
 1
(1 row)

```

```

=> INSERT INTO ttest SELECT x * 2 FROM ttest;
INFO:  trigf (fired before): there are 1 rows in ttest
INFO:  trigf (fired after ): there are 2 rows in ttest
                                ^^^^^^

```

可視性の説明を思い出してください。

```

INSERT 167794 1
=> SELECT * FROM ttest;
 x
---
 1
 2
(2 rows)

```

```

=> UPDATE ttest SET x = NULL WHERE x = 2;
INFO:  trigf (fired before): there are 2 rows in ttest
UPDATE 0
=> UPDATE ttest SET x = 4 WHERE x = 2;
INFO:  trigf (fired before): there are 2 rows in ttest
INFO:  trigf (fired after ): there are 2 rows in ttest
UPDATE 1
vac=> SELECT * FROM ttest;
 x
---
 1
 4

```

```
(2 rows)

=> DELETE FROM ttest;
INFO:  trigf (fired before): there are 2 rows in ttest
INFO:  trigf (fired before): there are 1 rows in ttest
INFO:  trigf (fired after ): there are 0 rows in ttest
INFO:  trigf (fired after ): there are 0 rows in ttest
                                ^^^^^^
```

可視性の説明を思い出してください。

```
DELETE 2
=> SELECT * FROM ttest;
      x
----
(0 rows)
```

src/test/regress/regress.cと[spi](#)にはもっと複雑な例があります。

[前のページ](#)

Cによるトリガ関数の作成

[ホーム](#)
[上に戻る](#)

[次のページ](#)
ルールシステム

第 37章 ルールシステム

目次

- 37.1. [問い合わせツリーとは](#)
- 37.2. [ビューとルールシステム](#)
 - 37.2.1. [SELECT ルールの動き](#)
 - 37.2.2. [非SELECT 文のビュールール](#)
 - 37.2.3. [PostgreSQLにおけるビューの能力](#)
 - 37.2.4. [ビューの更新について](#)
- 37.3. [INSERT、UPDATE、DELETE についてのルール](#)
 - 37.3.1. [更新ルールの動作](#)
 - 37.3.2. [ビューとの協調](#)
- 37.4. [ルールと権限](#)
- 37.5. [ルールおよびコマンドの状態](#)
- 37.6. [ルール対トリガ](#)

本章ではPostgreSQLのルールシステムについて説明します。本番で稼働するルールシステムは概念としては単純ですが、実際に使ってみると、わかりにくいところが少なからずあります。

通常それらはストアドプロシージャとトリガですが、他のいくつかのデータベースシステムは能動的データベースルールを定義しています。PostgreSQLでは関数とトリガとして実装されています。

ルールシステム（より正確に言うと問い合わせ書き換えルールシステム）はストアドプロシージャとトリガとはまったく異なります。ルールシステムはルールを参照して問い合わせを修正し、修正した問い合わせを、計画作成と実行のために問い合わせプランナに渡します。これは非常に強力なため、問い合わせ言語プロシージャ、ビューあるいはバージョンなど多くのパターンで 사용할 ことができます。このルールシステムの基礎理論と能力は [On Rules, Procedures, Caching and Views in Database Systems](#) および [A Unified Framework for Version Modeling Using Production Rules in a Database System](#) で解説されています。

37.1. 問い合わせツリーとは

どのようにルールシステムが機能するかを理解するためには、ルールがどのように起動され、その入力と結果は何かを理解しなければなりません。

ルールシステムは問い合わせパーサとプランナの中に位置します。ルールシステムは、入力としてパーサの出力、単一の問い合わせツリー、および何らかの特別な情報を持つ問い合わせツリーでもあるユーザ定義の書き換えルールを取り、結果として0個以上の問い合わせツリーを生成します。ルールシステムの入力と出力は常にパーサ自体でも生成することができるもので、参照する対象は基本的にSQL文として表現できるものです。

では問い合わせツリーとは何でしょうか。それは、SQL文を構成する個々の部品を別々に記憶した、SQL文の内部表現です。`debug_print_parse`、`debug_print_rewritten`、もしくは`debug_print_plan`設定パラメータを設定していれば、サーバログ内で問い合わせツリーを見ることができます。ルールアクションも`pg_rewrite`システムカタログ内に問い合わせツリーとして格納されています。これはログ出力のように整形されていませんが、まったく同じ情報を持っています。

問い合わせツリーそのものを読むためにはある程度の経験が必要です。ルールシステムを理解するためには問い合わせツリーのSQL表現で十分ですので、ここではその読み方までは教えません。

本章の問い合わせツリーのSQL表現形式を読む時に必要なのは、問い合わせツリー構造の中に分解された、ある文の部品を識別できることです。問い合わせツリーには以下の部品があります。

コマンド種類

これはどのコマンド (`SELECT`、`INSERT`、`UPDATE`、`DELETE`) が構文解析ツリーを作ったかを示す単純な値です。

範囲テーブル

範囲テーブルは問い合わせで使われるリレーションのリストです。`SELECT`文ではこれは`FROM`キーワードの後で与えられるリレーションになります。

範囲テーブルのそれぞれの項目はテーブルもしくはビューを識別し、問い合わせの別の部品ではどんな名前でも呼び出されるかを示します。問い合わせツリーでは範囲テーブルの項目は名前よりも番号で参照されることが多いため、ここではSQL文とは違い、重複する名前があるかということは問題になりません。これはルールの範囲テーブルがマージされた後に起こる可能性があります。本章の例ではその状況を含んでいません。

結果リレーション

問い合わせの結果が格納されるリレーションを識別する範囲テーブルへのインデックスです。

`SELECT`問い合わせは結果リレーションを持ちません。(`SELECT INTO`の場合は特別ですが、`INSERT ... SELECT`が付いた`CREATE TABLE`とほぼ同じですので、ここでは個別には説明しません。)

`INSERT`、`UPDATE`、`DELETE`コマンドでは、結果リレーションは変更が有効になるテーブル (もしくはビュー) です。

目的リスト

目的リストは問い合わせの結果を定義する式のリストです。`SELECT`の場合、この式は問い合わせの最終結果を構築するものです。これらは`SELECT`と`FROM`キーワードの間にある式に対応します (*は単にリレーションの全ての列名の省略です。これはパーサによって個別の列に展開されますので、ルールシステムが見ることはありません)。

`DELETE`コマンドは結果を返しませんので、通常目的リストは必要ありません。その代わりに、ルールシステムは空の目的リストに特別なCTID項目を追加し、エクゼキュータが削除すべき行を見つけられるようにします。(CTIDは結果リレーションが通常のテーブルの場合に追加されます。もしビューであれば[項37.2.4](#)で述べるように、代わりに行全体の変数が追加されます。)

`INSERT`問い合わせでは、目的リストは結果リレーションへ入る新規の行を示します。これは`VALUES`句か`INSERT ... SELECT`の中の`SELECT`句の式です。書き換え処理の最初のステップでは、元の問い合わせでは割り当てられず、デフォルト値となっている列の目的リストの項目を追加します。残った列 (値が与えられていない列、かつデフォルト値を持たない列) は全て、プランナによって定数`NULL`式で埋められます。

`UPDATE`コマンドでは、目的リストは古いものを置き換えるべき新しい行を示します。ルールシステムではコマンド内の`SET column = expression`部分にある式だけを持っています。プランナは、古い行から新しい行へ値をコピーする式を挿入することにより、抜けている列を処理します。`DELETE`の場合と同様、エクゼキュータが更新すべき行を見つけられるように、ルールシステムはCTIDもしくは行全体の変数を追加します。

目的リストの各項目は、定数値、範囲テーブル内のリレーション中の1つの列を指し示す変数、パラメータ、または、関数呼び出し、定数、変数、演算子などにより作られた式のツリーを保持します。

条件

問い合わせの条件は目的リストの項目に含まれている式によく似た式です。この式の結果は、最終的な結果の行を得るための（INSERT、UPDATE、DELETEまたはSELECT）演算を実行すべきかどうかを示すブール値です。それはSQL文の中のWHERE句です。

結合ツリー

問い合わせの結合ツリーはFROM句の構造を表します。SELECT ... FROM a, b, cのような単純な問い合わせでは、結合ツリーは単なるFROM項目のリストです。なぜならこれらはどんな順番で結合しても構わないからです。しかしJOIN式、特に外部結合が使われた場合は、その結合が示す順番通りに結合しなければいけません。この場合結合ツリーはJOIN式の構造を表します。特定のJOIN句と関連付けられた制約（ONもしくはUSING式からのもの）はこれらの結合ツリーノードに付加された条件として格納されます。頂点レベルのWHERE式を頂点レベルの結合ツリー項目に付加された条件として格納することも便利です。ですから、結合ツリーはSELECTのFROM句とWHERE句の両方を表しているわけです。

その他

ORDER BY句のような、問い合わせツリーのその他の部品は、ここでは取り上げません。ルールシステムはルールを適用している時にそこで項目を入れ替えることもあります。これはルールシステムの基本とはあまり関係しません。

[前のページ](#)
ルールシステム

[ホーム](#)
[上に戻る](#)

[次のページ](#)
ビューとルールシステム

37.2. ビューとルールシステム

PostgreSQLにおけるビューはルールシステムを使って実装されています。 実際、

```
CREATE VIEW myview AS SELECT * FROM mytab;
```

と

```
CREATE TABLE myview (same column list as mytab);
CREATE RULE "RETURN" AS ON SELECT TO myview DO INSTEAD
SELECT * FROM mytab;
```

の2つのコマンドの間には基本的な違いはありません。 と言うのは、`CREATE VIEW`コマンドによって内部的にまったく同じコマンドが行われるからです。 これには副作用もあります。 その1つはPostgreSQLシステムカタログのビューについての情報はテーブルの情報とまったく同一であるということです。 そのため、パーサにとってはテーブルとビューの間に違いは一切ありません。 これらは同じもの、つまりリレーションです。

37.2.1. SELECTルールの動き

たとえコマンドが`INSERT`、`UPDATE`、`DELETE`などであっても、`ON SELECT`ルールは全ての問い合わせに対し最後に適用されます。 そして、このルールは他のコマンド種類のルールと異なるセマンティックを持っていて、問い合わせツリーを新規に生成せずに、そこにあるものを修正します。 したがって`SELECT`ルールが一番初めに記述されなければいけません。

現在のところ、`ON SELECT`ルールでは1つのアクションしか許されず、それは`INSTEAD`である無条件の`SELECT`アクションでなければいけません。 この制約は、一般のユーザが何をしても、ルールシステムが堅牢であるために必要であり、`ON SELECT`のルールはビュー同様の動作に限定されます。

本章の例として挙げているのは、ちょっとした演算をする2つの結合のビューと、次にこれらの機能を利用するいくつかのビューです。 最終結果が何らかの魔法の機能によりあたかも実テーブルのように振舞うビューになるように、初めの2つのビューのうちの1つは、`INSERT`、`UPDATE`、`DELETE`操作に対するルールを後で追加することでカスタマイズされます。 初めて学ぶための例としては決して簡単ではなく先に進むことを躊躇させるかもしれませんが、多くの別々の例を持ち出して頭の混乱を招くよりも、全ての論点をステップごとに追う1つの例を挙げる方が良いでしょう。

この例では、2つの整数から小さな値を返すちょっとした`min`関数を必要とします。 関数の生成は以下のようにします。

```
CREATE FUNCTION min(integer, integer) RETURNS integer AS $$
SELECT CASE WHEN $1 < $2 THEN $1 ELSE $2 END
$$ LANGUAGE SQL STRICT;
```

最初の2つのルールシステムの説明で必要とする実テーブルを以下に示します。

```
CREATE TABLE shoe_data (
    shoename    text,          -- プライマリキー
    sh_avail    integer,       -- 在庫
    sl_color    text,          -- 望ましい靴紐の色
    sl_minlen   real,          -- 靴紐の最短サイズ
    sl_maxlen   real,          -- 靴紐の最長サイズ
    sl_unit     text           -- 長さの単位
);

CREATE TABLE shoelace_data (
    sl_name     text,          -- プライマリキー
    sl_avail    integer,       -- 在庫
    sl_color    text,          -- 靴紐の色
    sl_len      real,          -- 靴紐の長さ
    sl_unit     text           -- 長さの単位
);

CREATE TABLE unit (
    un_name     text,          -- プライマリキー
    un_fact     real           -- cmに変換するファクタ
);
```

これでわかるかもしれませんが、これらは靴屋のデータを表しています。

ビューを以下のように作成します。

```
CREATE VIEW shoe AS
SELECT sh.shoename,
       sh.sh_avail,
       sh.sl_color,
```

```

        sh.slminlen,
        sh.slminlen * un.un_fact AS slminlen_cm,
        sh.slmaxlen,
        sh.slmaxlen * un.un_fact AS slmaxlen_cm,
        sh.slunit
    FROM shoe_data sh, unit un
    WHERE sh.slunit = un.un_name;

CREATE VIEW shoelace AS
    SELECT s.sl_name,
           s.sl_avail,
           s.sl_color,
           s.sl_len,
           s.sl_unit,
           s.sl_len * u.un_fact AS sl_len_cm
    FROM shoelace_data s, unit u
    WHERE s.sl_unit = u.un_name;

CREATE VIEW shoe_ready AS
    SELECT rsh.shoename,
           rsh.sh_avail,
           rsl.sl_name,
           rsl.sl_avail,
           min(rsh.sh_avail, rsl.sl_avail) AS total_avail
    FROM shoe rsh, shoelace rsl
    WHERE rsl.sl_color = rsh.slcolor
           AND rsl.sl_len_cm >= rsh.slminlen_cm
           AND rsl.sl_len_cm <= rsh.slmaxlen_cm;

```

shoelaceビュー（今ある一番簡単なビュー）用のCREATE VIEWコマンドは、shoelaceリレーションと、問い合わせ範囲テーブルの中でshoelaceリレーションが参照される時はいつでも、適用されるべき書き換えルールの存在を示す項目をpg_rewriteに作ります。ルールはルール条件（SELECTルールは現在持つことができませんので、非SELECTルールのところで取り上げます）を持たないINSTEADです。ルール条件は問い合わせ条件とは異なることに注意してください！ルールアクションは問い合わせ条件を持っています。このルールアクションは、ビュー作成コマンド内のSELECTのコピーである、1つの問い合わせツリーです。

注意: pg_rewrite項目のNEWとOLDに対する2つの特別な範囲テーブル項目はSELECTルールには関係ありません。

ではここでunit、shoe_data、shoelace_dataにデータを入れ、ビューに簡単な問い合わせを行います。

```

INSERT INTO unit VALUES ('cm', 1.0);
INSERT INTO unit VALUES ('m', 100.0);
INSERT INTO unit VALUES ('inch', 2.54);

INSERT INTO shoe_data VALUES ('sh1', 2, 'black', 70.0, 90.0, 'cm');
INSERT INTO shoe_data VALUES ('sh2', 0, 'black', 30.0, 40.0, 'inch');
INSERT INTO shoe_data VALUES ('sh3', 4, 'brown', 50.0, 65.0, 'cm');
INSERT INTO shoe_data VALUES ('sh4', 3, 'brown', 40.0, 50.0, 'inch');

INSERT INTO shoelace_data VALUES ('sl1', 5, 'black', 80.0, 'cm');
INSERT INTO shoelace_data VALUES ('sl2', 6, 'black', 100.0, 'cm');
INSERT INTO shoelace_data VALUES ('sl3', 0, 'black', 35.0, 'inch');
INSERT INTO shoelace_data VALUES ('sl4', 8, 'black', 40.0, 'inch');
INSERT INTO shoelace_data VALUES ('sl5', 4, 'brown', 1.0, 'm');
INSERT INTO shoelace_data VALUES ('sl6', 0, 'brown', 0.9, 'm');
INSERT INTO shoelace_data VALUES ('sl7', 7, 'brown', 60, 'cm');
INSERT INTO shoelace_data VALUES ('sl8', 1, 'brown', 40, 'inch');

SELECT * FROM shoelace;

```

sl_name	sl_avail	sl_color	sl_len	sl_unit	sl_len_cm
sl1	5	black	80	cm	80
sl2	6	black	100	cm	100
sl7	7	brown	60	cm	60
sl3	0	black	35	inch	88.9
sl4	8	black	40	inch	101.6
sl8	1	brown	40	inch	101.6
sl5	4	brown	1	m	100
sl6	0	brown	0.9	m	90

(8 rows)

これは、ビューに対する最も簡単なSELECTですので、この機会にビュールールの基本を説明します。SELECT * FROM shoelaceはパーサによって処理され、次の問い合わせツリーが生成されます。

```

SELECT shoelace.sl_name, shoelace.sl_avail,
       shoelace.sl_color, shoelace.sl_len,
       shoelace.sl_unit, shoelace.sl_len_cm
FROM shoelace shoelace;

```

このツリーがルールシステムに伝えられます。ルールシステムは範囲テーブルを参照し、何らかのリレーションに対してルールが存在するか調べます。shoelace（現時点では唯一のビュー）についての範囲テーブル項目を処理する際、問い合わせツリーで_RETURNルールを検出します。

```

SELECT s.sl_name, s.sl_avail,
       s.sl_color, s.sl_len, s.sl_unit,
       s.sl_len * u.un_fact AS sl_len_cm
FROM shoelace old, shoelace new,
     shoelace_data s, unit u
WHERE s.sl_unit = u.un_name;

```

ビューを展開するために、リライタは単純にルールアクション問い合わせツリーを持つ副問い合わせ範囲テーブルの項目を作り、ビューを参照していた元の範囲テーブルを置き換えます。書き換えられた結果の問い合わせツリーは、以下のように入力した場合とほぼ同じです。

```

SELECT shoelace.sl_name, shoelace.sl_avail,
       shoelace.sl_color, shoelace.sl_len,
       shoelace.sl_unit, shoelace.sl_len_cm
FROM (SELECT s.sl_name,
            s.sl_avail,
            s.sl_color,
            s.sl_len,
            s.sl_unit,
            s.sl_len * u.un_fact AS sl_len_cm
      FROM shoelace_data s, unit u
      WHERE s.sl_unit = u.un_name) shoelace;

```

しかし1つだけ違いがあります。副問い合わせの範囲テーブルが2つの余分な項目 `shoelace old` と `shoelace new` を持っていることです。これらの項目は副問い合わせの結合ツリーや目的リストで参照されませんので、直接問い合わせでは使われません。リライタはそれらを使用して、ビューを参照した範囲テーブルの項目に元々存在したアクセス権限確認情報を格納します。この方法で、書き換えられた問い合わせで直接ビューを使用していなくても、エクゼキュータはユーザがそのビューにアクセスするための正しい権限を持っているか確認します。

これが最初に適用されるルールです。ルールシステムは最上位の問い合わせの残り（この例ではこれ以上ありません）の範囲テーブルの項目をチェックし続けます。そしてルールシステムは、追加された副問い合わせの範囲テーブルの項目がビューを参照するかを再帰的に確認します（しかし `old` や `new` は展開しません。そうでなければ無限再帰になってしまいます！）。この例では `shoelace_data` や `unit` 用の書き換えルールはありません。ですから書き換えは完結し、上記がプランナに渡される最終的な結果となります。

さて、店に置いてある靴紐（の色とサイズ）に一致する靴が店にあるか、完全に一致する靴紐の在庫数が2以上あるかどうかを把握する問い合わせを書いてみましょう。

```

SELECT * FROM shoe_ready WHERE total_avail >= 2;

shoename | sh_avail | sl_name | sl_avail | total_avail
-----+-----+-----+-----+-----
sh1      |         2 | sl1     |         5 |         2
sh3      |         4 | sl7     |         7 |         4
(2 rows)

```

今回のパーサの出力は以下の問い合わせツリーです。

```

SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
FROM shoe_ready shoe_ready
WHERE shoe_ready.total_avail >= 2;

```

最初に適用されるルールは `shoe_ready` ビュー用のもので、問い合わせツリーにおける結果は以下のようになります。

```

SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
FROM (SELECT rsh.shoename,
            rsh.sh_avail,
            rsl.sl_name,
            rsl.sl_avail,
            min(rsh.sh_avail, rsl.sl_avail) AS total_avail
      FROM shoe rsh, shoelace rsl
      WHERE rsl.sl_color = rsh.slcolor
            AND rsl.sl_len_cm >= rsh.slminlen_cm
            AND rsl.sl_len_cm <= rsh.slmaxlen_cm) shoe_ready
WHERE shoe_ready.total_avail >= 2;

```

同じように、`shoe` と `shoelace` 用のルールは副問い合わせの範囲テーブルとして代用され、3レベルの最終問い合わせツリーへと導きます。

```

SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
FROM (SELECT rsh.shoename,
            rsh.sh_avail,
            rsl.sl_name,

```

```

    rsl.sl_avail,
    min(rsh.sh_avail, rsl.sl_avail) AS total_avail
FROM (SELECT sh.shoename,
             sh.sh_avail,
             sh.slcolor,
             sh.slminlen,
             sh.slminlen * un.un_fact AS slminlen_cm,
             sh.slmaxlen,
             sh.slmaxlen * un.un_fact AS slmaxlen_cm,
             sh.slunit
      FROM shoe_data sh, unit un
     WHERE sh.slunit = un.un_name) rsh,
(SELECT s.sl_name,
      s.sl_avail,
      s.sl_color,
      s.sl_len,
      s.sl_unit,
      s.sl_len * u.un_fact AS sl_len_cm
      FROM shoelace_data s, unit u
     WHERE s.sl_unit = u.un_name) rsl
WHERE rsl.sl_color = rsh.slcolor
      AND rsl.sl_len_cm >= rsh.slminlen_cm
      AND rsl.sl_len_cm <= rsh.slmaxlen_cm) shoe_ready
WHERE shoe_ready.total_avail > 2;

```

次にプランナはこのツリーを2レベルの問い合わせツリーに縮めます。一番下のSELECTコマンドは別々に処理する必要がありませんので2つ目のSELECTに"引っ張り上げ"られます。しかし2つ目のSELECTは集約関数を持つため、頂点からは区別されます。もしそれらを引っ張り上げてしまうと一番上のSELECTの動作を変えてしまうことになり、それはしたくありません。しかし、問い合わせツリーを縮めるという最適化を、書き換えシステム自身で意識する必要はありません。

37.2.2. 非SELECT文のビュールール

これまでのビュールールの説明では問い合わせツリーの2つの詳細について触れませんでした。それらは、コマンドタイプと結果リレーションです。実際、コマンドタイプはビュールールでは必要とされませんが、結果リレーションがビューの場合には特別な考慮が必要です。結果リレーションは問い合わせリライタの動作に影響するかもしれません。

SELECTと他のコマンドに対する問い合わせツリーの間には大きな違いはありません。それらは明らかに違うコマンドタイプを持っていて、SELECT以外のコマンドでは、結果リレーションは結果の格納先となる範囲テーブルの項目を指し示します。それ以外ではまったく同じです。ですから、aとbの列を持つテーブルt1、t2に対する以下の2つの文の問い合わせツリーは、ほとんど同じです。

```

SELECT t2.b FROM t1, t2 WHERE t1.a = t2.a;

UPDATE t1 SET b = t2.b FROM t2 WHERE t1.a = t2.a;

```

以下に、具体的に示します。

- 範囲テーブルには、テーブルt1とt2に対する項目があります。
- 目的リストにはテーブルt2に対する範囲テーブル項目のb列を指し示す1つの変数があります。
- 条件式は、範囲テーブルの両項目のa列の等価性を比較します。
- 結合ツリーはt1とt2の単純な結合を表しています。

結果として、両方の問い合わせツリーは似たような実行計画になります。それらはともに2つのテーブルの結合です。UPDATEではt1から抜けている列はプランナによって目的リストに追加され、最終の問い合わせツリーは、以下のようになります。

```

UPDATE t1 SET a = t1.a, b = t2.b FROM t2 WHERE t1.a = t2.a;

```

そして、結合を実行したエグゼキュータは、

```

SELECT t1.a, t2.b FROM t1, t2 WHERE t1.a = t2.a;

```

の結果集合とまったく同じ結果集合を作成します。とは言ってもUPDATEにはちょっとした問題があります。結合を行うエグゼキュータの計画の部分は、結合の結果が何に向けられているかに関与しません。エグゼキュータは単に結果となる行の集合を作成するだけです。1つはSELECTコマンドでもう1つはUPDATEコマンドであるという事実は、エグゼキュータの中のより上位で扱われます。そこでは、これがUPDATEであるとわかっていて、この結果がテーブルt1に入らなければいけないことを知っています。しかし、そこにあるどの行が新しい行によって置換されなければならないのでしょうか。

この問題を解決するため、UPDATE文（DELETE文の場合も同様）の目的リストに別の項目が付け加えられます。それは、現在のタプルID（CTID）です。これはその行のファイルブロック番号とブロック中の位置を持つシステム列です。テーブルがわかっている場合、CTIDを使用して、元のt1行を抽出して更新することができます。CTIDを目的リストに追加すると、問い合わせは以下のようになります。


```
SELECT t1.a, t2.b, t1.ctid FROM t1, t2 WHERE t1.a = t2.a;
```

では、PostgreSQLの別の詳細説明に入りましょう。 テーブルの行は上書きされませんので、ROLLBACK処理は速いのです。 UPDATEでは、（CTIDを取り除いた後）テーブルに新しい結果行が挿入され、CTIDが指し示す古い行の行ヘッダ内のcmaxとxmax項目は現在のコマンドカウンタと現在のトランザクションIDに設定されます。 このようにして、古い行は隠され、トランザクションがコミットされた後、vacuum掃除機が不必要になった行をそのうちに削除できます。

これらの詳細が全部理解できれば、どんなコマンドに対してもまったく同じようにしてビューのルールを簡単に適用することができます。 そこには差異がありません。

37.2.3. PostgreSQLにおけるビューの能力

ここまでで、ルールシステムがどのようにビューの諸定義を元の問い合わせツリーに組み入れるかを解説しました。 第2の例では、1つのビューからの単純なSELECTによって、最終的に4つのテーブルを結合する問い合わせツリーが生成されました（unitは違った名前でも2回使われました）。

ビューをルールシステムで実装する利点は、どのテーブルをスキャンすべきか、それらのテーブル間の関連性、ビューからの制約条件、元の問い合わせ条件に関する情報を全て、プランナが1つの問い合わせツリーの中に持っていることです。 元の問い合わせが既にビューに対する結合である時も同様です。 プランナはここでどれが問い合わせ処理の最適経路かを決定しなければなりません。 プランナは保持する情報が多ければ多いほど、より良い決定を下すことができます。 そしてPostgreSQLに実装されているルールシステムはこれが現時点で、提供されている全ての情報であることを保証します。

37.2.4. ビューの更新について

ビューがINSERT、UPDATE、DELETEなどの目的リレーションとして名付けられた場合はどうなるのでしょうか？ 単純に上で説明したような置換をすると、結果リレーションが副問い合わせの範囲テーブル項目を指す問い合わせツリーができますが、それでは使えません。 その代わり、リライタは操作がビューのINSTEAD OFトリガによって扱われると仮定します。（もしそのようなトリガがなければ、エクゼキュータはエラーを発生します。） この場合、書き換えは少々違う形で行われます。 INSERTに対しては、リライタはビューに全く何もせず、問い合わせの結果リレーションをそのままにします。 UPDATEとDELETEに対しては、コマンドが更新もしくは削除しようとする"古い"行を生成するためにビュー問い合わせを展開する必要があります。 そのため、ビューは通常通り展開されますが、もう一つの展開されない範囲テーブル項目が結果リレーションとしてビューを表す問い合わせに追加されます。

ここで起こる問題はビューで更新される行をどのように特定するかということです。 結果リレーションがテーブルの場合、更新する行の物理的な位置を特定するために特別なCTID項目が目的リストに追加されることを思い出して下さい。 ビューの行には実際の物理的な位置がないため、ビューにはCTIDがありませんので、これは結果リレーションがビューの場合には上手くいきません。 その代わり、UPDATEやDELETE操作では、特別な行全体の項目が目的リストに追加されていて、それはビューからすべての列を含むように展開されています。 エクゼキュータこの値を使って"古い"行をINSTEAD OFトリガに提供します。 新旧の行の値に基づいて更新するものを計算するのはトリガの責任です。

ビューを更新するINSTEAD OFトリガがなければ、ビューを独力で自動的に更新できませんのでエラーが発生します。 これを変えるために、ビューでINSERT、UPDATE、DELETEコマンドの動作を変えるルールを定義できます。 これらのルールは、ビューではなくコマンドを書き換えます。 通常は1つもしくは複数のテーブルを更新するコマンドに書き換えます。 それが次節の論題になります。

ルールが最初に評価され、元の問い合わせが計画され実行される前にそれを書き換えることに注意して下さい。 そのためビューにINSTEAD OFトリガとINSERTやUPDATEやDELETEに関するルールがあった場合、ルールが最初に評価され、その結果よってトリガが全く使われないかもしれません。

[前のページ](#)
問い合わせツリーとは

[ホーム](#)
[上に戻る](#)

[次のページ](#)
INSERT、UPDATE、DELETEについてのルール

37.3. INSERT、UPDATE、DELETEについてのルール

INSERT、UPDATE、DELETEに定義するルールは前節で解説したビューのルールとはまったく異なります。第一点として、これらのCREATE RULEコマンドでは以下を行うことができます。

- アクションがないルールも可能です。
- 複数のアクションを持てます。
- INSTEADもしくはALSO（デフォルト）を取ることができます。
- 疑似リレーションNEWとOLDが役立つようになります。
- ルール条件を持たせることができます。

第二点として、その場所で問い合わせツリーを変更しません。その代わりに新規の0個以上の問い合わせツリーを生成して、オリジナルを破棄することができます。

37.3.1. 更新ルールの動作

```
CREATE [ OR REPLACE ] RULE name AS ON event
  TO table [ WHERE condition ]
  DO [ ALSO | INSTEAD ] { NOTHING | command | ( command ; command ... ) }
```

上記の構文を覚えておいてください。以下では、更新ルールはINSERT、UPDATE、DELETEに定義されたルールを意味します。

更新ルールは、問い合わせツリーの結果リレーションとコマンド種類がCREATE RULEで与えられるオブジェクトとイベントと等しい場合にルールシステムによって適用されます。更新ルールに対してルールシステムは問い合わせツリーのリストを生成します。最初は問い合わせツリーリストは空です。0（NOTHING キーワード）、1つまたは複数のアクションが有効です。簡単にするため、ここでは1つのアクションのルールを取り上げます。このルールは条件を持っても持っていなくても構いませんし、またINSTEADかALSO（デフォルト）のいずれかを取ることができます。

ルール条件とはどんなものでしょうか。それはルールのアクションを行わなければならない時と、行ってはならない時を指定する条件です。基本的に（特別な意味合いを持った）オブジェクトとして与えられるリレーションであるNEW疑似リレーションかOLD疑似リレーション、または、その両者のみをこの条件は参照することができます。

1アクションのルールに対し、以下の問い合わせツリーを生成する3つの場合があります。

ALSOまたはINSTEADで条件がない場合。

元の問い合わせツリーの条件が追加された、ルールアクションからの問い合わせツリー

条件付き、かつALSO

ルール条件と元の問い合わせツリーの条件が追加された、ルールアクションからの問い合わせツリー

条件付き、かつINSTEAD

ルール条件と元の問い合わせツリーの条件が追加された、ルールアクションからの問い合わせツリーとルール条件の否定条件が追加された元の問い合わせツリー

最後に、もしルールがALSOの場合、変更されていない元の問い合わせツリーがリストに付け加えられます。条件付きのINSTEADルールのみが既に元の構文解析ツリーに追加をしているので、最後は1つのアクションを持つルールに対して1つもしくは2つの問い合わせツリーにたどり着きます。

ON INSERTルールでは、元の問い合わせは、（INSTEADにより止められていない限り）ルールによって追加されたアクションの前に実行されます。これにより、アクションは挿入された行（複数可）を参照することができます。しかし、ON UPDATEとON DELETEルールでは、元の問い合わせはルールによって追加されたアクションの後に実行されます。これは、アクションが更新される予定の、または削除される予定の行を参照できることを保証します。さもないと、条件に一致する行を見つけることができないためにアクションが作動しなくなる可能性が起きます。

ルールアクションで生成された問い合わせツリーは、再度書き換えシステムに渡され、相当数の問い合わせツリーの結果をもたらす、より多くのルールの適用を受けることもあります。ですから、ルールのアクションはルール自身とは異なるコマンド種類か、異なる結果リレーションを持っていなければなりません。さもないと、この再帰的处理により無限ループに陥ってしまいます。（ルールの再帰展開は検知され、エラーとして報告されます。）

pg_rewriteシステムカタログのアクションにある問い合わせツリーは単なるテンプレートです。 これらはNEWとOLDに対する範囲テーブルの項目を参照することができるため、使用される前に何らかの置換措置がとられていなければなりません。 NEWを参照する全てに対し、元の問い合わせの目的リストは対応する項目があるかどうか検索されます。 項目が見つかった場合には、その項目式が参照を置き換えます。 項目がなかった場合、NEWはOLDと同じ意味になる（UPDATEの場合）か、NULLによって置き換えられます（INSERTの場合）。 OLDに対する参照は全て結果リレーションである範囲テーブルの項目への参照に置き換えられます。

更新ルールの適用が終わると、システムはそこで作られた構文解析ツリーにビュールールを適用します。 ビューは、新しい更新アクションを挿入できないため、ビュー書き換えの結果に更新ルールを適用する必要はありません。

37.3.1.1. 最初のルール、ステップバイステップ

shoelace_dataリレーションのsl_avail列の変化を追跡してみたいと思います。 そこでログ用テーブルと、shoelace_dataに対して行われるUPDATEをログに記録するルールを用意しました。

```
CREATE TABLE shoelace_log (
    sl_name    text,          -- 変更された靴紐
    sl_avail   integer,       -- 新しい現在値
    log_who    text,          -- 誰が行ったか
    log_when   timestamp      -- いつ行ったか
);

CREATE RULE log_shoelace AS ON UPDATE TO shoelace_data
WHERE NEW.sl_avail <> OLD.sl_avail
DO INSERT INTO shoelace_log VALUES (
    NEW.sl_name,
    NEW.sl_avail,
    current_user,
    current_timestamp
);
```

ここで誰かが以下を実行します。

```
UPDATE shoelace_data SET sl_avail = 6 WHERE sl_name = 'sl7';
```

ログテーブルを見てみましょう。

```
SELECT * FROM shoelace_log;

 sl_name | sl_avail | log_who | log_when
-----+-----+-----+-----
 sl7     |        6 | Al      | Tue Oct 20 16:14:45 1998 MET DST
(1 row)
```

思った通りの結果が出ました。 以下に裏で何が起こったのかを説明します。 パーサがまず以下の構文解析ツリーを生成しました。

```
UPDATE shoelace_data SET sl_avail = 6
FROM shoelace_data shoelace_data
WHERE shoelace_data.sl_name = 'sl7';
```

以下のルール条件式

```
NEW.sl_avail <> OLD.sl_avail
```

および、以下のアクションを持つON UPDATEのlog_shoelaceルールがあります。

```
INSERT INTO shoelace_log VALUES (
    new.sl_name, new.sl_avail,
    current_user, current_timestamp )
FROM shoelace_data new, shoelace_data old;
```

(通常、INSERT ... VALUES ... FROMを書くことはできないのでちょっと奇妙に見えるかもしれませんが。 このFROM句は単にnewとoldの問い合わせツリーの範囲テーブル項目があることを示しているだけです。 これらは、INSERT コマンドの問い合わせツリー中の変数から参照されるために必要なのです。)

このルールは条件付きのALSOルールですので、ルールシステムは変更されたルールアクションと元の問い合わせツリーという2つの問い合わせツリーを返さなければなりません。 第1の段階で元の問い合わせの範囲テーブルはルールアクション問い合わせツリーに取り込まれます。 そして、次の結果を生みます。

```
INSERT INTO shoelace_log VALUES (
    new.sl_name, new.sl_avail,
    current_user, current_timestamp )
FROM shoelace_data new, shoelace_data old,
shoelace_data shoelace_data;
```

第2段階で、以下のようにルール条件が付け加えられます。 これにより、この結果集合はsl_availが変更した行に限定されます。

```
INSERT INTO shoelace_log VALUES (
    new.sl_name, new.sl_avail,
    current_user, current_timestamp )
FROM shoelace_data new, shoelace_data old,
    shoelace_data shoelace_data
WHERE new.sl_avail <> old.sl_avail;
```

(INSERT ... VALUESはWHERE句を持たないため、これはさらに奇妙です。 しかし、プランナとエグゼキュータには問題はありません。 これらはどのみちINSERT ... SELECTのために同じ機能をサポートしなければなりません。)

第3段階で、以下のように元の問い合わせツリーの条件が付け加えられ、結果集合は元の問い合わせで変更された行のみにさらに限定されます。

```
INSERT INTO shoelace_log VALUES (
    new.sl_name, new.sl_avail,
    current_user, current_timestamp )
FROM shoelace_data new, shoelace_data old,
    shoelace_data shoelace_data
WHERE new.sl_avail <> old.sl_avail
    AND shoelace_data.sl_name = 'sl7';
```

第4段階では、以下のように元の問い合わせツリーの目的リスト項目、または結果リレーシンの該当する変数参照で、NEWの参照を置換します。

```
INSERT INTO shoelace_log VALUES (
    shoelace_data.sl_name, 6,
    current_user, current_timestamp )
FROM shoelace_data new, shoelace_data old,
    shoelace_data shoelace_data
WHERE 6 <> old.sl_avail
    AND shoelace_data.sl_name = 'sl7';
```

第5段階は、以下のようにOLD参照を結果リレーシン参照に置き換えます。

```
INSERT INTO shoelace_log VALUES (
    shoelace_data.sl_name, 6,
    current_user, current_timestamp )
FROM shoelace_data new, shoelace_data old,
    shoelace_data shoelace_data
WHERE 6 <> shoelace_data.sl_avail
    AND shoelace_data.sl_name = 'sl7';
```

これで終わりです。このルールはALSOのため、元の問い合わせツリーも出力します。 まとめると、ルールシステムからの出力は以下の文に対応する2つの問い合わせツリーのリストです。

```
INSERT INTO shoelace_log VALUES (
    shoelace_data.sl_name, 6,
    current_user, current_timestamp )
FROM shoelace_data
WHERE 6 <> shoelace_data.sl_avail
    AND shoelace_data.sl_name = 'sl7';

UPDATE shoelace_data SET sl_avail = 6
WHERE sl_name = 'sl7';
```

この2つは順番通りに処理され、正確にルールが定義した通りです。

元の問い合わせが例えば下記のような場合に、置換と追加された条件は、ログには何も書かれないことを確実にします。

```
UPDATE shoelace_data SET sl_color = 'green'
WHERE sl_name = 'sl7';
```

この場合、元の問い合わせツリーの目的リストにはsl_availの項目がありませんので、NEW.sl_availがshoelace_data.sl_availに置き換えられます。 その結果、このルールによって以下のような特別の問い合わせが生成されます。

```
INSERT INTO shoelace_log VALUES (
    shoelace_data.sl_name, shoelace_data.sl_avail,
    current_user, current_timestamp )
FROM shoelace_data
WHERE shoelace_data.sl_avail <> shoelace_data.sl_avail
    AND shoelace_data.sl_name = 'sl7';
```

そしてこの条件は決して真にはなりません。

もし元の問い合わせが複数の行を変更してもうまくいきます。 ですから、誰かが下記のようなコマンドを実行したとします。

```
UPDATE shoelace_data SET sl_avail = 0
WHERE sl_color = 'black';
```

この場合、実際には4行が更新されます（s11、s12、s13およびs14）。しかしs13は既にsl_avail = 0を持っています。この場合、元の問い合わせツリーの条件を満たさず、その結果、以下のような特別の問い合わせツリーがルールによって生成されます。

```
INSERT INTO shoelace_log
SELECT shoelace_data.sl_name, 0,
       current_user, current_timestamp
FROM shoelace_data
WHERE 0 <> shoelace_data.sl_avail
AND shoelace_data.sl_color = 'black';
```

この構文解析ツリーは確実に3つの新しいログ項目を挿入します。これはまったく正しい動作です [訳注：s13行はWHERE 0 != shoelace_data.sl_avail条件を満たさない（0!=0）ので、実際に更新される4行-1の3行分のログ項目が挿入されます]。

ここで元の構文解析ツリーが最後に実行されるということが重要な理由がわかります。もしUPDATEが先に実行されたとしたら、全ての行は0にセットされ、0 <> shoelace_data.sl_availである行をログ書き込み時のINSERTの段階で見つけられなくなります。

37.3.2. ビューとの協調

誰かがビューに対してINSERT、UPDATE、DELETEを発行するといった、前述の可能性からビューレレーションを保護する簡単な方法は、それらの構文解析ツリーを破棄してしまうことです。このために以下のルールを作ることができます。

```
CREATE RULE shoe_ins_protect AS ON INSERT TO shoe
DO INSTEAD NOTHING;
CREATE RULE shoe_upd_protect AS ON UPDATE TO shoe
DO INSTEAD NOTHING;
CREATE RULE shoe_del_protect AS ON DELETE TO shoe
DO INSTEAD NOTHING;
```

誰かがshoeビューレレーションに上記の操作を行おうとすると、ルールシステムはルールを適用します。ルールにはアクションがなく、かつ、INSTEADですから、結果の問い合わせツリーリストは空になります。ルールシステムの処理が完了した後に最適化されるものや実行されるべきものが何も残っていませんので、問い合わせ全体が無効になります。

より洗練されたルールシステムの使用法は、実テーブルに適切な操作を行う問い合わせツリーへの書き換えを行うルールを作ることです。shoelaceビューにこれを適用するために以下のルールを作ります。

```
CREATE RULE shoelace_ins AS ON INSERT TO shoelace
DO INSTEAD
INSERT INTO shoelace_data VALUES (
    NEW.sl_name,
    NEW.sl_avail,
    NEW.sl_color,
    NEW.sl_len,
    NEW.sl_unit
);

CREATE RULE shoelace_upd AS ON UPDATE TO shoelace
DO INSTEAD
UPDATE shoelace_data
SET sl_name = NEW.sl_name,
    sl_avail = NEW.sl_avail,
    sl_color = NEW.sl_color,
    sl_len = NEW.sl_len,
    sl_unit = NEW.sl_unit
WHERE sl_name = OLD.sl_name;

CREATE RULE shoelace_del AS ON DELETE TO shoelace
DO INSTEAD
DELETE FROM shoelace_data
WHERE sl_name = OLD.sl_name;
```

このビュー上でRETURNING問い合わせをサポートしたい場合、ビューの行を計算するRETURNING句を含むルールを作成しなければなりません。これは通常、単一テーブルに対するビューでは非常に簡単ですが、shoelaceのようなビューを結合する場合は多少やっかいです。挿入する場合を例として以下に示します。

```
CREATE RULE shoelace_ins AS ON INSERT TO shoelace
DO INSTEAD
INSERT INTO shoelace_data VALUES (
    NEW.sl_name,
    NEW.sl_avail,
    NEW.sl_color,
    NEW.sl_len,
    NEW.sl_unit
)
```

```

RETURNING
    shoelace_data.*,
    (SELECT shoelace_data.sl_len * u.un_fact
     FROM unit u WHERE shoelace_data.sl_unit = u.un_name);

```

この1つのルールが、ビューに対するINSERT問い合わせとINSERT RETURNING問い合わせルールをサポートすることに注意してください。 INSERTではRETURNING句が無視されるだけです。

ここで店には不定期に靴紐のケースが分厚い送り状とともに届けられると仮定します。しかし、毎回毎回手作業でshoelaceビューを更新したくはありません。代わりに、送り状から品目を挿入するテーブルと特殊な仕掛けを持つテーブルの2つの小さなテーブルを用意します。 以下はそれらを作成するコマンドです。

```

CREATE TABLE shoelace_arrive (
    arr_name    text,
    arr_quant   integer
);

CREATE TABLE shoelace_ok (
    ok_name     text,
    ok_quant    integer
);

CREATE RULE shoelace_ok_ins AS ON INSERT TO shoelace_ok
DO INSTEAD
UPDATE shoelace
    SET sl_avail = sl_avail + NEW.ok_quant
    WHERE sl_name = NEW.ok_name;

```

これで、送り状のデータをshoelace_arriveテーブルに投入することができます。

```

SELECT * FROM shoelace_arrive;

arr_name | arr_quant
-----+-----
sl3      |         10
sl6      |         20
sl8      |         20
(3 rows)

```

そして現在のデータをチェックします。

```

SELECT * FROM shoelace;

sl_name | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
-----+-----+-----+-----+-----+-----
sl1     |         5 | black   |      80 | cm      |         80
sl2     |         6 | black   |     100 | cm      |        100
sl7     |         6 | brown   |      60 | cm      |         60
sl3     |         0 | black   |      35 | inch    |        88.9
sl4     |         8 | black   |      40 | inch    |       101.6
sl8     |         1 | brown   |      40 | inch    |       101.6
sl5     |         4 | brown   |       1 | m       |        100
sl6     |         0 | brown   |      0.9 | m       |         90
(8 rows)

```

さて、届いた靴紐を移動します。

```

INSERT INTO shoelace_ok SELECT * FROM shoelace_arrive;

```

そして結果を確認します。

```

SELECT * FROM shoelace ORDER BY sl_name;

sl_name | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
-----+-----+-----+-----+-----+-----
sl1     |         5 | black   |      80 | cm      |         80
sl2     |         6 | black   |     100 | cm      |        100
sl7     |         6 | brown   |      60 | cm      |         60
sl4     |         8 | black   |      40 | inch    |       101.6
sl3     |        10 | black   |      35 | inch    |        88.9
sl8     |        21 | brown   |      40 | inch    |       101.6
sl5     |         4 | brown   |       1 | m       |        100
sl6     |        20 | brown   |      0.9 | m       |         90
(8 rows)

SELECT * FROM shoelace_log;

sl_name | sl_avail | log_who | log_when
-----+-----+-----+-----
sl7     |         6 | Al      | Tue Oct 20 19:14:45 1998 MET DST
sl3     |        10 | Al      | Tue Oct 20 19:25:16 1998 MET DST
sl6     |        20 | Al      | Tue Oct 20 19:25:16 1998 MET DST
sl8     |        21 | Al      | Tue Oct 20 19:25:16 1998 MET DST

```

```
(4 rows)
```

1つのINSERT ... SELECTからこの結果になるには、長い道のりがあります。本章での問い合わせツリーの変形に関する説明はこれが最後です。まず、以下のようなパーサの出力があります。

```
INSERT INTO shoelace_ok
SELECT shoelace_arrive.arr_name, shoelace_arrive.arr_quant
FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok;
```

最初のshoelace_ok_insルールが適用され、結果は以下のようになります。

```
UPDATE shoelace
SET sl_avail = shoelace.sl_avail + shoelace_arrive.arr_quant
FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
shoelace_ok old, shoelace_ok new,
shoelace shoelace
WHERE shoelace.sl_name = shoelace_arrive.arr_name;
```

そして、shoelace_okに対する元のINSERTを破棄します。この書き換えられた問い合わせは再びルールシステムに渡されて、2番目に適用されるshoelace_updルールは以下を生成します。

```
UPDATE shoelace_data
SET sl_name = shoelace.sl_name,
    sl_avail = shoelace.sl_avail + shoelace_arrive.arr_quant,
    sl_color = shoelace.sl_color,
    sl_len = shoelace.sl_len,
    sl_unit = shoelace.sl_unit
FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
shoelace_ok old, shoelace_ok new,
shoelace shoelace, shoelace old,
shoelace new, shoelace_data shoelace_data
WHERE shoelace.sl_name = shoelace_arrive.arr_name
AND shoelace_data.sl_name = shoelace.sl_name;
```

これは再びINSTEADルールですので、以前の問い合わせツリーは破棄されます。この問い合わせはshoelaceビューを引き続き使用していることに注意してください。しかし、この段階ではルールシステムは終了していないため、引き続き_RETURNルールが適用され、下記のようになります。

```
UPDATE shoelace_data
SET sl_name = s.sl_name,
    sl_avail = s.sl_avail + shoelace_arrive.arr_quant,
    sl_color = s.sl_color,
    sl_len = s.sl_len,
    sl_unit = s.sl_unit
FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
shoelace_ok old, shoelace_ok new,
shoelace shoelace, shoelace old,
shoelace new, shoelace_data shoelace_data,
shoelace old, shoelace new,
shoelace_data s, unit u
WHERE s.sl_name = shoelace_arrive.arr_name
AND shoelace_data.sl_name = s.sl_name;
```

最後に、log_shoelaceルールが適用され、以下のような特別な問い合わせツリーが生成されます。

```
INSERT INTO shoelace_log
SELECT s.sl_name,
    s.sl_avail + shoelace_arrive.arr_quant,
    current_user,
    current_timestamp
FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
shoelace_ok old, shoelace_ok new,
shoelace shoelace, shoelace old,
shoelace new, shoelace_data shoelace_data,
shoelace old, shoelace new,
shoelace_data s, unit u,
shoelace_data old, shoelace_data new
shoelace_log shoelace_log
WHERE s.sl_name = shoelace_arrive.arr_name
AND shoelace_data.sl_name = s.sl_name
AND (s.sl_avail + shoelace_arrive.arr_quant) <> s.sl_avail;
```

この後、ルールシステムはルールを使い切り、生成された問い合わせツリーを返します。

そして、以下のSQL文と等価となる2つの最終問い合わせツリーで終結します。

```
INSERT INTO shoelace_log
SELECT s.sl_name,
    s.sl_avail + shoelace_arrive.arr_quant,
    current_user,
    current_timestamp
FROM shoelace_arrive shoelace_arrive, shoelace_data shoelace_data,
```



```

        shoelace_data s
WHERE s.sl_name = shoelace_arrive.arr_name
      AND shoelace_data.sl_name = s.sl_name
      AND s.sl_avail + shoelace_arrive.arr_quant <> s.sl_avail;

UPDATE shoelace_data
      SET sl_avail = shoelace_data.sl_avail + shoelace_arrive.arr_quant
FROM shoelace_arrive shoelace_arrive,
      shoelace_data shoelace_data,
      shoelace_data s
WHERE s.sl_name = shoelace_arrive.sl_name
      AND shoelace_data.sl_name = s.sl_name;

```

結果は、1つのリレーションから来たデータが別のリレーションに挿入され、3つ目のリレーションの更新に変更され、4つ目の更新と5つ目への最終更新のログ記録に変更され、最終的に2つの問い合わせに縮小されます。

ちょっと見苦しい小さな事項があります。 でき上がった2つの問い合わせを見ると、1つに縮小されたはずの `shoelace_data` リレーションが範囲テーブルに二度出てきます。 プランナは処理をしないので、`INSERT` のルールシステムの出力に対する実行計画は次のようになります。

```

入れ子状ループ
-> マージ結合
    -> シーケンシャルスキャン
        -> ソート
            -> sに対するシーケンシャルスキャン
    -> シーケンシャルスキャン
        -> ソート
            -> shoelace_arriveに対するシーケンシャルスキャン
-> shoelace_dataに対するシーケンシャルスキャン

```

一方、余計な範囲テーブル項目を省略することで、以下のようにログテーブルにまったく同じ項目が作られます。

```

マージ結合
-> シーケンシャルスキャン
    -> ソート
        -> sに対するシーケンシャルスキャン
-> シーケンシャルスキャン
    -> ソート
        -> shoelace_arriveに対するシーケンシャルスキャン

```

ですから、ルールシステムは、まったく必要のない `shoelace_data` リレーションに対する余計なスキャンを一度行うことになりま
す。 そして `UPDATE` でも同様な不要なスキャンが再度実行されます。 しかしながら、これらを全て可能にするのは大変な仕事で
す。

最後にPostgreSQLのルールシステムとその効力を示しましょう。 例えば、まったく売れそうもない靴紐をデータベースに追加
してみます。

```

INSERT INTO shoelace VALUES ('sl9', 0, 'pink', 35.0, 'inch', 0.0);
INSERT INTO shoelace VALUES ('sl10', 1000, 'magenta', 40.0, 'inch', 0.0);

```

全ての靴に合わない色が `shoelace` 項目にあるかどうかを検査するビューを作成したいと考えます。 ビューは以下のようになります。

```

CREATE VIEW shoelace_mismatch AS
  SELECT * FROM shoelace WHERE NOT EXISTS
    (SELECT shoename FROM shoe WHERE slcolor = sl_color);

```

この出力は以下のようになります。

```

SELECT * FROM shoelace_mismatch;

sl_name | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
-----+-----+-----+-----+-----+-----
sl9     |         0 | pink    | 35     | inch    | 88.9
sl10    |      1000 | magenta | 40     | inch    | 101.6

```

ここで、合う靴がない靴紐のうち、在庫がないものをデータベースから削除するように設定してみます。 これはPostgreSQLで
は困難な作業ですので、直接削除しません。 代わりに、以下のようにもう1つビューを作成します。

```

CREATE VIEW shoelace_can_delete AS
  SELECT * FROM shoelace_mismatch WHERE sl_avail = 0;

```

そして、以下を行います。

```

DELETE FROM shoelace WHERE EXISTS
  (SELECT * FROM shoelace_can_delete
    WHERE sl_name = shoelace.sl_name);

```

さあできました。

```
SELECT * FROM shoelace;
```

sl_name	sl_avail	sl_color	sl_len	sl_unit	sl_len_cm
s11	5	black	80	cm	80
s12	6	black	100	cm	100
s17	6	brown	60	cm	60
s14	8	black	40	inch	101.6
s13	10	black	35	inch	88.9
s18	21	brown	40	inch	101.6
s110	1000	magenta	40	inch	101.6
s15	4	brown	1	m	100
s16	20	brown	0.9	m	90

(9 rows)

合計4つのネスト/結合されたビューを副問い合わせの条件として持ち、その中の1つはビューを含む副問い合わせ条件を持ち、かつ演算を施されたビューの列が使われる場合の、ビューに対するDELETEが、実テーブルから要求されたデータを削除する単一の問い合わせツリーに書き換えられます。

このような構造が必要な状況は実社会ではほとんどないと思われます。しかし、実際に動くことを確認できれば安心できます。

37.4. ルールと権限

PostgreSQLのルールシステムによる問い合わせの書き換えによって、オリジナルの問い合わせで使われたものではない他のテーブル/ビューがアクセスされます。更新ルールを使うことによってテーブルへの書き込みアクセスを包含することができます。

書き換えルールに別々の所有者はいません。リレーション（テーブルまたはビュー）の所有者は自動的にそれに定義された書き換えルールの所有者となります。PostgreSQLのルールシステムはデフォルトのアクセス制御システムの振舞いを変更します。ルールによって使用されるリレーションは、ルールを起動したユーザの権限ではなく、ルール所有者の権限でチェックされます。このことは、ユーザは問い合わせで明記するテーブル/ビューに対しての権限だけあればよいことを示しています。

例えば、以下のようにします。あるユーザが、いくつかは個人用の、その他は事務所で秘書が利用するための、電話番号のリストを持っていたとします。ユーザは次のようにして構築することができます。

```
CREATE TABLE phone_data (person text, phone text, private boolean);
CREATE VIEW phone_number AS
  SELECT person, CASE WHEN NOT private THEN phone END AS phone
  FROM phone_data;
GRANT SELECT ON phone_number TO secretary;
```

そのユーザ（とデータベースのスーパーユーザ）以外は`phone_data`テーブルにアクセスできません。しかし、`GRANT`により秘書は`phone_number`ビューに対し`SELECT`できます。ルールシステムは`phone_number`からの`SELECT`操作を`phone_data`からの`SELECT`操作に書き換えます。そのユーザは`phone_number`の所有者、したがってルールの所有者ですから、`phone_data`の読み込みに対するアクセスはそのユーザの権限に従ってチェックされ、問い合わせを受け付けてもよいことになります。`phone_number`へのアクセスもチェックされますが、これは呼び出したユーザに対して行われますので、秘書とユーザ以外は使うことができません。

権限はルールごとにチェックされます。ですから秘書だけが今のところ公開の電話番号を参照することができます。しかし、秘書は別のビューを作成し、それに`PUBLIC`に対するアクセス許可を与えることができます。こうすると秘書のビューを通して誰もが`phone_number`データを見ることができます。秘書ができないことは`phone_data`に直接アクセスするビューを作ることです（実際には作成はできますが、アクセスは全て、権限チェックで拒絶されます）。そして、秘書が独自の`phone_number`ビューを開いたことにユーザが気付いた時点で、秘書の権限を取り上げることができます。秘書のビューへのアクセスは即座に失敗に終わります。

このルールごとのチェックがセキュリティホールになると考える人がいるかもしれませんが、実際にはそうではありません。もしこのように機能しないとすると、秘書は`phone_number`と同じ列を持ったテーブルを用意して、1日1回データをそこにコピーするかもしれません。そうすると、データは彼のものですから、誰にアクセス権を与えようが彼の自由です。`GRANT`は"あなたを信用しています"ということです。信用している誰かがこのようなことを行っただけの場合は、考えを変えて`REVOKE`してください。

上に示したような手法を使ってある列の内容を隠すのにビューは使えますが、`security_barrier`フラグが設定されていない限り、見えない行にあるデータを隠す信頼できる方法ではない事に注意してください。例えば、以下のビューは安全ではありません。

```
CREATE VIEW phone_number AS
  SELECT person, phone FROM phone_data WHERE phone NOT LIKE '412%';
```

ルールシステムが`phone_number`からの`SELECT`を`phone_data`からの`SELECT`に書き換え、`phone`が412で始まらない項目のみが必要だという条件を追加しますので、このビューは安全に見えます。しかし、ユーザが自身の関数を作成できるのであれば、`NOT LIKE`式の前にユーザ定義の関数を実行するようプランナを説得することは難しくありません。例えば以下の通りです。

```
CREATE FUNCTION tricky(text, text) RETURNS bool AS $$
BEGIN
  RAISE NOTICE '% => %', $1, $2;
  RETURN true;
END
$$ LANGUAGE plpgsql COST 0.000000000000000000000001;

SELECT * FROM phone_number WHERE tricky(person, phone);
```

プランナはより高価な`NOT LIKE`の前に安価な`tricky`関数を実行することを選びますので、`phone_data`テーブルの人と電話番号はすべて`NOTICE`として表示されます。たとえユーザが新しい関数を定義できない場合でも、同様の攻撃で組み込み関数が使えます。（例えば、ほとんどの型変換関数は生成するエラーメッセージを入力値に含んでいます。）

同様の考慮は更新ルールにも適用できます。前節の例において、データベースのテーブルの所有者は`shoelace`ビューに対し、誰かに`SELECT`、`INSERT`、`UPDATE`、`DELETE`権限を与えることができます。しかし、`shoelace_log`に対しては`SELECT`だけです。ログ項目を書き込むルールアクションは支障なく実行され、また、他のユーザはログ項目を見ることができます。しかし、他のユーザは項目を捏造したり、既に存在する項目を操作する、あるいは削除することはできません。この場合、`shoelace_log`を参照しているルールは条件のない`INSERT`だけです。操作の順序を変えるようにプランナを説得することでルールを破壊する可能性

はありません。これはより複雑な状況では正しくないかもしれません。

ビューが行レベルセキュリティを提供する場合には、そのビューにはsecurity_barrier属性を付与すべきです。これは、悪意を持って選ばれた関数や演算子が、ビューが適用されるより前に取り出された行に対して実行されないようにします。例えば、上記のビューが次のように定義された場合、これは安全です。

```
CREATE VIEW phone_number WITH (security_barrier) AS
SELECT person, phone FROM phone_data WHERE phone NOT LIKE '412%';
```

security_barrierを付けて定義されたビューは、このオプションなしのビューよりも性能の劣るクエリ実行プランを生成します。一般的に、最も高速だが、セキュリティ上の問題があるクエリ実行プランを破棄しなければならないという状況は不可避免です。そのため、このオプションはデフォルトでは有効になっていません。

プランナは副作用のない関数をもう少し柔軟に扱います。これらの関数はLEAKPROOF属性を持っており、等価演算子など、単純で広く用いられる演算子も多く含まれます。利用者に対して不可視な行に対するこれら関数の呼び出しはいかなる情報も漏えいさせないため、プランナはこれらの関数をどのような場所でも安全に実行させる事ができます。一方、例えばオーバーフローやゼロ除算のよう、受け取った引数の値に応じてエラーを発生させるような関数はleak-proofではありません。これがもしセキュリティ・ビューの条件句でフィルタリングされるより前に適用されたなら、不可視行に関する何か重要な情報を与える事ができてしまいます。

ビューがsecurity_barrier属性付きで定義されたとしても、それは限定的な意味で安全である、つまり不可視な行の内容が潜在的に安全でない関数に渡される事がないという事を意図しているにすぎません。例えば、EXPLAINを用いてクエリ実行プランを見たり、ビューに対するクエリ実行時間を計測したり、利用者は不可視な行に対して何らかの推測を行うかもしれません。悪意の攻撃者は不可視データの量を推測したり、分散や最頻値（なぜなら、これらは統計情報を通じて実行プランの選択、ひいては実行時間に影響するかもしれません）に関する何らかの情報を得る事ができるかもしれません。もし、この種の"隠れチャネル"に対する対策が必要であれば、とにかくデータに対するアクセスを許可するのは得策ではありません。

37.5. ルールおよびコマンドの状態

PostgreSQLサーバでは、受け取った問い合わせのそれぞれに対して、`INSERT 149592 1`のようなコマンド状態文字列を返します。これは簡単ですが、ルールが使用されていない場合には十分なものです。しかし、問い合わせがルールにより書き換えられた場合、どのようになるでしょうか。

ルールはコマンド状態に以下のように影響を与えます。

- 問い合わせに無条件の`INSTEAD`ルールが使用されていない場合、元々与えられていた問い合わせが実行され、そのコマンド状態は通常通り返されます（しかし、条件付き`INSTEAD`ルールが使用されている場合、その条件の否定が元の問い合わせに追加されることに注意してください。これにより、処理する行の数が減り、その結果報告される状態が影響を受けるかもしれません）。
- 問い合わせに無条件の`INSTEAD`ルールが使用されている場合、元の問い合わせはまったく実行されません。この場合、サーバでは、（条件付きもしくは無条件の）`INSTEAD`ルールによって挿入され、かつ、元の問い合わせと同じ種類（`INSERT`、`UPDATE`または`DELETE`）の最後の問い合わせについてコマンド状態を返します。この条件に合致する問い合わせがルールによって追加されない場合、返されるコマンド状態は、元の問い合わせの種類と行数およびOIDフィールドに0が表示されます。

（このシステムはPostgreSQL 7.3で確立されました。それ以前のバージョンでのコマンド状態はルールが存在する場合異なる結果を表示する可能性があります。）

後者の場合、プログラマは、有効ルールの中でアルファベット順の最後のルール名を与えることによって、必要な`INSTEAD`ルールを最後に実行することができます。そして、そのことによって、コマンド状態が確実にそのルールで設定されるようになります。

37.6. ルール対トリガ

トリガによって行われる多くの操作はPostgreSQLのルールシステムで実装可能です。ルールで実装できないものの1つはある種の制約、特に外部キーに関してです。もし他のテーブルに列の値がなかった場合、条件ルールでコマンドをNOTHINGに書き換えてしまうことも可能ですが、これではデータがだまって消去されてしまい、良いアイデアとは言えません。有効な値かどうかのチェックが必要で、無効な値についてはエラーメッセージを表示する必要があるなら、このことは今のところトリガを使って行わなければなりません。

この章ではビューを更新するのにルールを使うことに焦点を当ててきました。この章の更新ルールの例はすべてビューのINSTEAD OFトリガを使っても実装できます。特に更新を実行するのに複雑な論理が要求される場合には、そのようなトリガを書くことはしばしばルールを書くよりも簡単です。

どちらでも実装できる事項に関してどちらがベストかはデータベースの使用法によります。トリガは各行に対して一度起動します。ルールは問い合わせを修正するか追加の問い合わせを生成します。ですから、1つの文が多くの行に影響を与える場合、1つの行を処理する度に呼び出され、何をするかを何度も再決定しなければならないトリガよりも、追加の問い合わせを1つ発行するルールの方がほとんどの場合高速になります。しかし、トリガ方式は概念的にルールシステムよりかなり単純であり、初心者は簡単に正しく扱うことができます。

ここで、ある状況下でルールとトリガのどちらを選択するかを示す例を挙げます。例えば、2つのテーブルがあるとします。

```
CREATE TABLE computer (
    hostname      text,      -- インデックスあり
    manufacturer  text      -- インデックスあり
);

CREATE TABLE software (
    software      text,      -- インデックスあり
    hostname      text      -- インデックスあり
);
```

2つのテーブルにはともに数千の行があって、hostname上のインデックスは一意です。ルール/トリガは削除されたホストを参照する、softwareの行を削除する制限を実装しなければなりません。トリガの場合は以下のコマンドを使用します。

```
DELETE FROM software WHERE hostname = $1;
```

computerから削除された行1つひとつに対してこのトリガが呼び出されますので、このコマンドの準備を行い、計画を保存し、パラメータとしてhostnameを渡すことができます。ルールの場合は以下のように作成されます。

```
CREATE RULE computer_del AS ON DELETE TO computer
DO DELETE FROM software WHERE hostname = OLD.hostname;
```

ここで別の類の削除を考えてみましょう。

```
DELETE FROM computer WHERE hostname = 'mypc.local.net';
```

上のような場合では、computerはインデックスにより（高速に）スキャンされます。トリガによってこのコマンドが発行された場合もインデックススキャンが使用されます（高速です）。ルールによる追加コマンドは以下のようになります。

```
DELETE FROM software WHERE computer.hostname = 'mypc.local.net'
AND software.hostname = computer.hostname;
```

適切なインデックスが設定されていますので、プランナは以下の計画を作成します。

```
入れ子状ループ
-> computerに対しcomp_hostidxを使用したインデックススキャン
-> softwareに対しsoft_hostidxを使用したインデックススキャン
```

ですので、トリガとルールの実装間での速度差はあまりありません。

次の削除処理ではhostnameがoldで始まる2,000台全てのcomputerを削除しようと思います。方法として2つの有効な問い合わせがあって、1つは以下のようなものです。

```
DELETE FROM computer WHERE hostname >= 'old'
AND hostname < 'ole'
```

ルールによって追加されるコマンドは以下のようになります。

```
DELETE FROM software WHERE computer.hostname >= 'old' AND computer.hostname < 'ole'
AND software.hostname = computer.hostname;
```

これに対する計画は以下のようになります。

ハッシュ結合

```
-> softwareに対するシーケンシャルスキャン
-> ハッシュ
-> computerに対するcomp_hostidxを使用するインデックススキャン
```

もう1つのコマンドは以下のようなものです。

```
DELETE FROM computer WHERE hostname ~ '^old';
```

これにより、ルールによって追加されるコマンド用の実行計画は以下のようになります。

入れ子状ループ

```
-> computerに対するcomp_hostidxを使用するインデックススキャン
-> softwareに対するsoft_hostidxを使用するインデックススキャン
```

これが示していることは、ANDで結合された複数の検索条件が存在する場合、プランナは正規表現版のコマンドでは行っていることですが、computer上のhostnameに対する検索条件をsoftware上のインデックススキャンにも同様に使用できることを理解しないということです。トリガは削除されるべき2,000台の旧式コンピュータのそれぞれについて1回呼び出され、結果computer上で1回のインデックススキャンとsoftware上で2,000回のインデックススキャンが行われます。ルールによる実装ではインデックスを使用する2つの問い合わせによって実行されます。シーケンシャルスキャンの場合でもルールがより速いかどうかはsoftwareテーブルの大きさに依存します。参照する全てのインデックスブロックがすぐにキャッシュに現れるとしても、トリガによるSPIマネージャ経由の2,000回のコマンドの実行には時間を要します。

最後のコマンドを見てみましょう。

```
DELETE FROM computer WHERE manufacturer = 'bim';
```

この文でもcomputerから多くの行が削除される結果となります。ですので、ここでもトリガはエグゼキュータを通して多くのコマンドを実行することになります。ルールで作成されるコマンドは以下のようなものです。

```
DELETE FROM software WHERE computer.manufacturer = 'bim'
AND software.hostname = computer.hostname;
```

このコマンド用の計画もまた前回同様2つのインデックススキャンの入れ子状ループとなります。computerの別のインデックスを使用する点のみが異なります。

入れ子状ループ

```
-> computerに対するcomp_manufidxを使用するインデックススキャン
-> softwareに対するsoft_hostidxを使用するインデックススキャン
```

いずれの場合においても、ルールシステムが生成する追加コマンドは影響を受ける行数からは多かれ少なかれ独立しています。

まとめると、問い合わせ結果が大きく、プランナがうまく結合条件を設定できないような状況下でのみルールはトリガに比べて明らかに遅くなります。

[前のページ](#)

ルールおよびコマンドの状態

[ホーム](#)

[上に戻る](#)

[次のページ](#)

手続き言語

第 38章手続き言語

PostgreSQLでは、SQLやC言語以外の言語でユーザ定義の関数を作成することができます。これらの他の言語は一般に手続き言語（PL）と呼ばれます。手続き言語で関数が記述されていた場合、データベースサーバにはその関数のソースをどのように解釈すればよいかについての知識が組み込まれていません。代わりに、その処理はその言語を解釈する特別なハンドラに引き渡されます。そのハンドラは解析、構文分析、実行などすべてのことを行うこともできますし、PostgreSQLと存在するプログラミング言語の実装との"橋渡し"ともなり得ます。ハンドラそのものはC言語関数で、他のC言語関数と同様に、共有オブジェクトにコンパイルされ、要求に応じてロードされます。

現在PostgreSQLの標準配布物では、PL/pgSQL（[第39章](#)）、PL/Tcl（[第40章](#)）、PL/Perl（[第41章](#)）、PL/Python（[第42章](#)）という4つの手続き言語があります。さらに、コア配布物には含まれない手続き言語があります。[付録H](#)でその見つけ方を説明します。ユーザは他の言語を定義することもできます。新しい手続き言語の開発について、その基礎を[第49章](#)で説明します。

38.1. 手続き言語のインストール

手続き言語は、それらが使用されるデータベースすべてに"インストール"されている必要があります。しかし、`template1` データベースにインストールされた手続き言語は、`template1` 内の項目は `CREATE DATABASE` によってコピーされますので、その後で作成されたすべてのデータベースで自動的に使用できます。したがって、データベース管理者はどのデータベースにどの言語を使用するかを決定できますし、デフォルトで使用できる言語も決定できます。

標準配布物で提供される言語では、その言語を現在のデータベースにインストールするには `CREATE EXTENSION language_name` の実行のみが必要です。シェルコマンドラインから [createlang](#) プログラムを使用してこれを行うこともできます。例えば、PL/Perl を `template1` データベースにインストールするには、下記のように実行してください。

```
createlang plperl template1
```

下記の手作業は、拡張機能としてパッケージ化されていない言語をインストールする場合にのみ行うことを推奨します。

手続き言語の手作業によるインストール方法

手続き言語を次の5段階でデータベースにインストールすることができます。この作業はデータベースのスーパーユーザで行う必要があります。ほとんどの場合、必要なSQLコマンドは"拡張機能"のインストールスクリプトとしてパッケージ化されていますので、この作業を実行するのに `CREATE EXTENSION` が利用できます。

- 1 11 その言語ハンドラ用の共有オブジェクトがコンパイルされ、適切なライブラリディレクトリにインストールされている必要があります。これは、通常のユーザ定義のC関数を作成してインストールする時と同じです。 [項35.9.6](#)を参照してください。実際のプログラミング言語エンジンを提供する外部ライブラリに、言語ハンドラが依存していることがよくあります。この場合はそのライブラリもインストールしなければなりません。

- 1 11 ハンドラは下記のコマンドで宣言されなければなりません。

```
CREATE FUNCTION handler_function_name ()
  RETURNS language_handler
  AS 'path-to-shared-object'
  LANGUAGE C;
```

`language_handler` という特別な戻り値の型は、この関数が定義済みのSQLデータ型を返さず、SQL文では直接使用できないことをデータベースシステムに伝えます。

- 1 11 省略可能ですが、言語ハンドラは、この言語で書かれた無名コードブロック ([DO](#) コマンド) を実行する"インライン"ハンドラ関数を提供することができます。インラインハンドラ関数が言語により提供されるのであれば、以下のようなコマンドで宣言されます。

```
CREATE FUNCTION inline_function_name (internal)
  RETURNS void
  AS 'path-to-shared-object'
  LANGUAGE C;
```

- 1 11 省略可能ですが、言語ハンドラは、実際に実行することなく関数定義の正確性を検査する"有効性検査"関数を提供することができます。もし存在すれば、有効性検査関数は `CREATE FUNCTION` で呼び出されます。有効性検査関数が言語により提供されるのであれば、以下のようなコマンドで宣言されます。

```
CREATE FUNCTION validator_function_name (oid)
  RETURNS void
  AS 'path-to-shared-object'
  LANGUAGE C STRICT;
```

- 1 11 最終的に、PLは下記のコマンドで宣言されなければいけません。

```
CREATE [TRUSTED] [PROCEDURAL] LANGUAGE language-name
  HANDLER handler_function_name
  [INLINE inline_function_name]
  [VALIDATOR validator_function_name] ;
```

`TRUSTED` というオプションキーワードは、ユーザがアクセス権を持たないデータに対して、その言語がアクセス権を持たないことを指定します。`TRUSTED` である言語は（スーパーユーザ権限を持たない）一般ユーザ用に設計されており、安全に関数やトリガプロシージャを作成できます。PL関数はデータベースサーバの内部で実行されますので、`TRUSTED` フラグはデータベースサーバ内部やファイルシステムへのアクセスを持たない言語のみが使わなければなりません。

PL/pgSQLとPL/Tcl、PL/Perl言語はTRUSTEDと考えられています。提供される機能が無制限に設計されているPL/TclU、PL/PerlU、PL/PythonU言語については、TRUSTEDを指定してはなりません。

[例38-1](#)に、手作業によるインストール手順がPL/Perl言語でどのように動作するかを示します。

例 38-1. PL/Perlの手作業によるインストール

以下のコマンドは、データベースサーバにPL/Perl言語の呼び出しハンドラ関数用の共有ライブラリの存在場所を通知します。

```
CREATE FUNCTION plperl_call_handler() RETURNS language_handler AS
'$libdir/plperl' LANGUAGE C;
```

PL/Perlはインラインハンドラ関数と有効性検査関数を有していますので、以下のようにも宣言できます。

```
CREATE FUNCTION plperl_inline_handler(internal) RETURNS void AS
'$libdir/plperl' LANGUAGE C;
```

```
CREATE FUNCTION plperl_validator(oid) RETURNS void AS
'$libdir/plperl' LANGUAGE C STRICT;
```

以下のコマンドは、直前に宣言された関数を、言語属性がplperlである関数やトリガプロシージャ用に呼び出さなければならないことを定義します。

```
CREATE TRUSTED PROCEDURAL LANGUAGE plperl
HANDLER plperl_call_handler
INLINE plperl_inline_handler
VALIDATOR plperl_validator;
```

デフォルトのPostgreSQLインストールでは、PL/pgSQL言語用のハンドラは構築され、"library"ディレクトリにインストールされます。さらに、PL/pgSQL言語自体がデータベースすべてにインストールされます。Tclのサポート付きで構築した場合、PL/TclとPL/TclU用のハンドラも構築され同じ場所にインストールされますが、言語自体はデフォルトではどのデータベースにもインストールされません。同様に、Perlサポート付きで構築した場合はPL/PerlとPL/PerlUハンドラが、Pythonサポート付きで構築した場合はPL/PythonUハンドラが構築され、インストールされますが、言語自体はデフォルトではインストールされません。

[前のページ](#)

手続き言語

[ホーム](#)

[上に戻る](#)

[次のページ](#)

PL/pgSQL - SQL手続き言語

第 39章PL/pgSQL - SQL手続き言語

目次

39.1. [概要](#)

- 39.1.1. [PL/pgSQLを使用することの利点](#)
- 39.1.2. [引数と結果データ型のサポート](#)

39.2. [PL/pgSQLの構造](#)

39.3. [宣言](#)

- 39.3.1. [関数引数の宣言](#)
- 39.3.2. [ALIAS](#)
- 39.3.3. [型のコピー](#)
- 39.3.4. [行型](#)
- 39.3.5. [レコード型](#)
- 39.3.6. [PL/pgSQL変数の照合](#)

39.4. [式](#)

39.5. [基本的な文](#)

- 39.5.1. [代入](#)
- 39.5.2. [結果を伴わないコマンドの実行](#)
- 39.5.3. [1行の結果を返す問い合わせの実行](#)
- 39.5.4. [動的コマンドの実行](#)
- 39.5.5. [結果ステータスの取得](#)
- 39.5.6. [まったく何もしない](#)

39.6. [制御構造](#)

- 39.6.1. [関数からの復帰](#)
- 39.6.2. [条件分岐](#)
- 39.6.3. [単純なループ](#)
- 39.6.4. [問い合わせ結果による繰り返し](#)
- 39.6.5. [配列を巡回](#)
- 39.6.6. [エラーの捕捉](#)

39.7. [カーソル](#)

- 39.7.1. [カーソル変数の宣言](#)
- 39.7.2. [カーソルを開く](#)
- 39.7.3. [カーソルの使用](#)
- 39.7.4. [カーソル結果に対するループ](#)

39.8. [エラーとメッセージ](#)

39.9. [トリガプロシージャ](#)

39.10. [PL/pgSQLの秘訣](#)

- 39.10.1. [変数置換](#)
- 39.10.2. [計画のキャッシュ](#)

39.11. [PL/pgSQLによる開発向けのヒント](#)

39.12. [Oracle PL/SQLからの移植](#)

- 39.12.1. [移植例](#)
- 39.12.2. [その他の注意事項](#)
- 39.12.3. [付録](#)

39.1. 概要

PL/pgSQLは、PostgreSQLデータベースシステム用の読み込み可能な手続き言語です。PL/pgSQLの設計目的は、次のような読み込み可能な手続き言語でした。

- 関数とトリガプロシージャを作成するために使用できること
- SQL言語に制御構造を追加すること
- 複雑な演算が可能であること
- 全てのユーザ定義型、関数、演算子を継承すること
- サーバによって信頼できるものと定義できること
- 使いやすいこと

PL/pgSQLで作成した関数は、組み込み関数が使えるところであれば、どこでも使用できます。例えば、複雑な条件のある演算処理関数の作成が可能ですし、作成した関数を使用して演算子を定義することも、インデックス式にその関数を使用することも可能です。

PostgreSQL 9.0以降ではPL/pgSQLはデフォルトでインストールされます。しかしこれはまだロード可能なモジュールですので、特にセキュリティに厳しい管理者は削除することもできます。

39.1.1. PL/pgSQLを使用することの利点

SQLはPostgreSQLおよびその他のほとんどのリレーショナルデータベースが問い合わせ言語として使用している言語です。移植性があり、習得が容易です。しかし、あらゆるSQL文はデータベースサーバによって個々に実行されなければいけません。

これはクライアントアプリケーションに対して以下のようなことを要求しています。まず、データベースサーバに問い合わせを送信します。次にそれが処理されるのを待ちます。次に、結果を取得して処理します。次に若干の計算を行います。そして、サーバに次の問い合わせを送信します。クライアントがデータベースサーバマシンと異なるマシンの場合、プロセス間通信を招き、ネットワークオーバーヘッドを起こすでしょう。

PL/pgSQLを使うことで、計算と複数の問い合わせをデータベースサーバ内部でひとまとめに実行することができます。このように、手続き言語の能力とSQLの使いやすさを持ち合わせているにもかかわらず、クライアント/サーバ通信のオーバーヘッドをかなり節約できます。

- クライアント・サーバ間の余計なやり取りを排除する。
- クライアントサーバ間において、クライアントに不必要な中間結果の整理と転送を不要とする。
- 一連の問い合わせに、複数の解析が不要である。

これにより、ストアドプロシージャを使用しないアプリケーションに比較して、かなり性能を向上させることができます。

また、PL/pgSQLではSQL全てのデータ型、演算子、関数を使用することができます。

39.1.2. 引数と結果データ型のサポート

PL/pgSQLで作成された関数は、サーバでサポートされる任意のスカラデータ型や配列データ型を引数として受け付けることができ、また、これらの型を結果として返すことができます。また、任意の、名前で指定された複合型（行型）を受け付けたり、返したりすることもできます。さらに、[項7.2.1.4](#)で説明されているように、PL/pgSQL関数が`record`を返すように宣言することも可能で、この場合、結果は行型で、その列は呼び出す問い合わせの中での指定で決まります。

PL/pgSQL関数は`VARIADIC`記号を使用して可変長の引数を受け付けられるように宣言することができます。これは[項35.4.5](#)で論議したように、SQL関数と全く同じ方法で動作します。

また、PL/pgSQL関数を、`anyelement`、`anyarray`、`anynonarray`、`anyenum`、`anyrange`多様型を受け付けたり、返したりするように宣言することもできます。[項35.2.5](#)の説明の通り、多様型関数で扱われる実際のデータ型は呼び出しごとに変動することができます。例を[項39.3.1](#)に示します。

PL/pgSQL関数は、1つのインスタンスとして返すことができる任意のデータ型の"集合"（テーブル）を返すように宣言できま

す。こうした関数は、結果集合の必要な要素に対して `RETURN NEXT` を実行すること、または問い合わせの評価結果を得るために `RETURN QUERY` を使用することで、その出力を生成します。

最後に、有用な戻り値を持たない場合、PL/pgSQL関数は、`void`を返すように宣言することができます。

PL/pgSQL関数は戻り値の型を明確に指定する代わりに、出力パラメータと共に宣言することもできます。これは言語に対して基本的な能力を追加するものではありませんが、特に複数の値を返す時にしばしば便利です。 `RETURNS TABLE` 表記は `RETURNS SETOF` の代わりとして使用できます。

関連する例は [項39.3.1](#) および [項39.6.1](#) にあります。

[前のページ](#)

PL/pgSQL - SQL手続き言語

[ホーム](#)
[上に戻る](#)

[次のページ](#)
PL/pgSQLの構造

39.2. PL/pgSQLの構造

PL/pgSQLはブロック構造の言語です。関数定義のテキスト全体はブロックでなければなりません。ブロックは以下のように定義されます。

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
END [ label ];
```

ブロック内の宣言や文はそれぞれ、セミコロンで終わります。上に示したように、他のブロック内に出現するブロックのENDの後にはセミコロンが必要ですが、関数本体を完結する最後のENDには不要です。

ティップ: BEGINの直後にセミコロンを書くことも、同じように間違いです。これは不正であり、構文エラーとなります。

labelが必要となるのは、EXIT文が使用されるブロックを特定したい場合、またはブロック内で宣言された変数名を修飾したい場合だけです。ENDの後にラベルを配置する時は、そのブロックの先頭ラベルと一致させなければなりません。

全てのキーワードは大文字と小文字を区別しません。識別子は二重引用符でくくられていない限り、通常のSQLコマンドと同様に、暗黙的に小文字に変換されます。

PL/pgSQLコード内では、通常のSQLと同じ方法のコメントが動作します。二重のダッシュ(--)はその行末までをコメントとするコメントを開始します。/*はコメントブロックの始まりを意味し、次に*/が現れるまでをコメントとします。ブロックコメントは入れ子になります。

ブロックの文節内の全ての文は副ブロックになることができます。副ブロックは論理的なグループ分けや変数を文の小さな集まりに局所化するのに使用できます。副ブロックにおいて宣言された変数は、副ブロック内部では外側のブロックにおける同名の変数を遮蔽しますが、外側のラベルを変数名に付加して指定すればアクセスできます。以下に例を示します。

```
CREATE FUNCTION somefunc() RETURNS integer AS $$
DECLARE
    quantity integer := 30;
BEGIN
    RAISE NOTICE 'Quantity here is %', quantity; -- Quantity here is 30と表示
    quantity := 50;
    --
    -- 副ブロックの作成
    --
    DECLARE
        quantity integer := 80;
    BEGIN
        RAISE NOTICE 'Quantity here is %', quantity; -- Quantity here is 80と表示
    END;
    RAISE NOTICE 'Quantity here is %', quantity; -- Quantity here is 50と表示

    RETURN quantity;
END;
$$ LANGUAGE plpgsql;
```

注意: PL/pgSQL関数の本体を囲む、隠れた"外側のブロック"が存在します。この隠れたブロックにおいて、関数のパラメータがあれば宣言をして、同様にFOUNDのような特殊な変数(項39.5.5を参照)を提供します。この外側のブロックのラベルは関数名となります。つまりパラメータと特殊な変数は関数名によって修飾することを意味します。

PL/pgSQLにおける文をまとめるためのBEGIN/ENDとトランザクション制御用の同名のSQLコマンドとを取り違えないようにすることが重要です。PL/pgSQLのBEGIN/ENDは単にまとめるためのもので、トランザクションを始めたり終わらせたりしません。関数とトリガプロシージャは常に外側の問い合わせで確立されたトランザクションの内側で実行されます。トランザクションの実行させる文脈はありませんので、これらはトランザクションを開始することもコミットすることもできません。しかし、EXCEPTION句を含むブロックは外側のトランザクションに影響しないでロールバックできるサブトランザクションを、実質的に作成できます。これについては項39.6.6を参照してください。

39.3. 宣言

ブロック内で使用される全ての変数はそのブロックの宣言部で宣言されなければなりません。（唯一の例外は、`FOR`ループである整数値の範囲に渡って繰り返されるループ変数で、これは、自動的に整数型変数として宣言されます。同様に、カーソルの結果に対して繰り返し適用される`FOR`ループのループ変数はレコード変数として自動的に宣言されます。）

PL/pgSQL変数は、`integer`、`varchar`、`char`といった、任意のSQLデータ型を持つことができます。

変数宣言の例を以下に示します。

```
user_id integer;
quantity numeric(5);
url varchar;
myrow tablename%ROWTYPE;
myfield tablename.columnname%TYPE;
arow RECORD;
```

変数宣言の一般的な構文は以下の通りです。

```
name [ CONSTANT ] type [ COLLATE collation_name ] [ NOT NULL ] [ { DEFAULT | := } expression ];
```

`DEFAULT`句が指定された場合、ブロックに入った時に変数に代入される初期値を指定します。`DEFAULT`句が指定されない場合、変数はSQLのNULL値に初期化されます。`CONSTANT`オプションにより、そのブロック内でその値が不変になるように、その変数への初期化後の代入は禁止されます。`COLLATE`オプションは、変数として使用するための照合を指定します（[項39.3.6](#)を参照してください）。`NOT NULL`が指定された場合、NULL値の代入は実行時エラーになります。`NOT NULL`として宣言した変数は全て、非NULLのデフォルト値を指定しなければなりません。

変数のデフォルト値はブロックに入る度に評価され、変数に代入されます（関数を呼び出す時に一度だけではありません）。ですから、例えば`now()`を`timestamp`型の変数に代入することで、その変数には関数をプリコンパイルした時刻ではなく、関数呼び出し時の現在時刻が格納されます。

例：

```
quantity integer DEFAULT 32;
url varchar := 'http://mysite.com';
user_id CONSTANT integer := 10;
```

39.3.1. 関数引数の宣言

関数に渡されるパラメータの名前には`$1`、`$2`という識別子が付けられます。省略することもできますが、`$n`というパラメータ名に別名を宣言することができ、可読性が向上します。別名、数字による識別子の両方とも引数の値を参照する時に使用することができます。

別名を作成する方法は2つあり、望ましい方法は`CREATE FUNCTION`コマンドの中でパラメータを命名するものです。以下に例を示します。

```
CREATE FUNCTION sales_tax(subtotal real) RETURNS real AS $$
BEGIN
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

他の方法は、宣言構文を用いて別名を明確に宣言するものです。これは PostgreSQL 8.0より前では、唯一の方法です。

```
name ALIAS FOR $n;
```

以下にこの方法による例を示します。

```
CREATE FUNCTION sales_tax(real) RETURNS real AS $$
DECLARE
    subtotal ALIAS FOR $1;
BEGIN
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

注意: この二例は完全に同等ではありません。最初の例では、`subtotal`を`sales_tax.subtotal`で参照できますが、次の例ではできません（その代わりに、内部ブロックにラベルを付与すれば、`subtotal`をラベルで修飾することができます）。

さらに数例を示します。

```
CREATE FUNCTION instr(vvarchar, integer) RETURNS integer AS $$
DECLARE
    v_string ALIAS FOR $1;
    index ALIAS FOR $2;
BEGIN
    -- v_string とインデックスを使用した何らかの演算を行なう
END;
$$ LANGUAGE plpgsql;

CREATE FUNCTION concat_selected_fields(in_t sometablename) RETURNS text AS $$
BEGIN
    RETURN in_t.f1 || in_t.f3 || in_t.f5 || in_t.f7;
END;
$$ LANGUAGE plpgsql;
```

PL/pgSQL関数が出力パラメータと共に宣言されると、通常の入力パラメータと同様に、出力パラメータには $\$n$ というパラメータ名と任意の別名が与えられます。出力パラメータは実質的には最初がNULL値の変数であり、関数の実行中に値が指定されるはずでず。出力パラメータの最後の値は戻り値です。例えば、消費税の例題は、次のようにすることもできます。

```
CREATE FUNCTION sales_tax(subtotal real, OUT tax real) AS $$
BEGIN
    tax := subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

RETURNS realを省略したことに注意してください。含めることもできますが、冗長になります。

出力パラメータは複数の値を返す時に最も有用になります。簡単な例題を示します。

```
CREATE FUNCTION sum_n_product(x int, y int, OUT sum int, OUT prod int) AS $$
BEGIN
    sum := x + y;
    prod := x * y;
END;
$$ LANGUAGE plpgsql;
```

[項35.4.4](#)で述べたように、この方法は関数の結果に対する匿名のレコード型を実質的に作成します。RETURNS句が与えられた時は、RETURNS recordと言わなければなりません。

PL/pgSQL関数を宣言する他の方法として、RETURNS TABLEを伴うことが挙げられます。以下に例を示します。

```
CREATE FUNCTION extended_sales(p_itemno int)
RETURNS TABLE(quantity int, total numeric) AS $$
BEGIN
    RETURN QUERY SELECT quantity, quantity * price FROM sales
                  WHERE itemno = p_itemno;
END;
$$ LANGUAGE plpgsql;
```

これは、1つ、またはそれ以上のOUTパラメータを宣言すること、およびRETURNS SETOF 何らかのデータ型を指定することと全く等価です。

PL/pgSQL関数の戻り値が多様型 (anyelement、anyarray、anynonarray、anyenumまたはanyrange) として宣言されると、特別な $\$0$ パラメータが作成されます。このデータ型が、実際の入力型から推定 ([項35.2.5](#)を参照) された関数の実際の戻り値型です。 $\$0$ はNULLで初期化され、関数内で変更することができます。ですので、必須ではありませんが、これを戻り値を保持するために使用しても構いません。また $\$0$ に別名を付与することもできます。例えば、以下の関数は+演算子を持つ任意のデータ型に対して稼働します。

```
CREATE FUNCTION add_three_values(v1 anyelement, v2 anyelement, v3 anyelement)
RETURNS anyelement AS $$
DECLARE
    result ALIAS FOR $0;
BEGIN
    result := v1 + v2 + v3;
    RETURN result;
END;
$$ LANGUAGE plpgsql;
```

1つ以上の出力パラメータを多様型として宣言することにより、同様の結果を得ることができます。この場合、特殊な $\$0$ パラメータは使用されません。出力パラメータ自身が同じ目的を果たします。以下に例を示します。

```
CREATE FUNCTION add_three_values(v1 anyelement, v2 anyelement, v3 anyelement,
                                OUT sum anyelement)
AS $$
BEGIN
    sum := v1 + v2 + v3;
END;
```

```
$$ LANGUAGE plpgsql;
```

39.3.2. ALIAS

```
newname ALIAS FOR oldname;
```

`ALIAS` 構文は前節で示したものより一般的です。関数の引数だけではなく、任意の変数に別名を宣言することができます。この現実的な使用は主に、トリガプロシージャにおける`NEW`や`OLD`など、前もって決まった名前の変数に別の名前を割り当てることです。

以下に例を示します。

```
DECLARE
prior ALIAS FOR old;
updated ALIAS FOR new;
```

`ALIAS` は同じオブジェクトを命名する2つの異なる手段を提供しますので、無制限に使用すると混乱を招くかもしれません。前もって決まっている名前を上書きする目的に限定して使用することが最善です。

39.3.3. 型のコピー

```
variable%TYPE
```

`%TYPE` は変数やテーブル列のデータ型を提供します。これを使用してデータベース値を保持する変数を宣言することができます。例えば、`users` テーブルに `user_id` という列があるものとします。 `users.user_id` と同じデータ型の変数を宣言するには、以下のように記述します。

```
user_id users.user_id%TYPE;
```

`%TYPE` を使用することで、参照する構造のデータ型を把握する必要がなくなります。また、これが最も重要なことですが、参照される項目のデータ型が将来変更された（例えば、`user_id` のテーブル定義を `integer` から `real` に変更した）場合でも、関数定義を変更する必要をなくすることができます。

内部変数用のデータ型は呼び出す度に変わるかもしれないので `%TYPE` は特に多様関数で有用です。関数の引数や結果用のプロセスホルダに `%TYPE` を適用することで、適切な変数を作成することができます。

39.3.4. 行型

```
name table_name%ROWTYPE;
name composite_type_name;
```

複合型の変数は、行変数（または行型変数）と呼ばれます。こういった変数には、問い合わせの列集合が変数の型宣言と一致する限り、`SELECT` や `FOR` 問い合わせの結果の行全体を保持することができます。行変数の個々のフィールド値には、例えば、`rowvar.field` といったドット記法を使用してアクセスすることができます。

`table_name%ROWTYPE` という記法を使用して、既存のテーブルやビューの行と同じ型を持つ行変数を宣言することができます。もしくは、複合型の名前を付与して宣言することができます。（全てのテーブルは、同じ名前の関連する複合型を持ちますので、実際のところ PostgreSQL では、`%ROWTYPE` と書いても書かなくても問題にはなりません。しかし、`%ROWTYPE` の方がより移植性が高まります。）

関数へのパラメータとして複合型（テーブル行全体）を取ることができます。その場合、対応する識別子 `$n` は行変数であり、そのフィールドを、例えば、`$1.user_id` で選択することができます。

テーブル行のユーザ定義の属性のみに行型変数でアクセスすることができます。OID やその他のシステム属性にはアクセスできません（ビューからの行があり得るためです）。行型のフィールドは、例えば `char(n)` などのテーブルのフィールドの大きさやデータ型の精度を継承します。

以下に複合型を使用する例を示します。 `table1` 及び `table2` は、少なくとも言及するフィールドを有する既存のテーブルです。

```
CREATE FUNCTION merge_fields(t_row table1) RETURNS text AS $$
DECLARE
    t2_row table2%ROWTYPE;
BEGIN
    SELECT * INTO t2_row FROM table2 WHERE ... ;
    RETURN t_row.f1 || t2_row.f3 || t_row.f5 || t2_row.f7;
END;
$$ LANGUAGE plpgsql;

SELECT merge_fields(t.*) FROM table1 t WHERE ... ;
```

39.3.5.

レコード型

```
name RECORD;
```

レコード変数は行型変数と似ていますが、事前に定義された構造を持っていません。これはSELECTやFORコマンドの間で代入された行の実際に行構造を取ります。レコード変数の副構造は、代入を行う度に変更できます。つまり、レコード変数は、最初に代入されるまで副構造を持たず、したがって、フィールドへのアクセスを試みると実行時エラーが発生します。

RECORDは本当のデータ型ではなく、単なるプレースホルダであることに注意してください。PL/pgSQL関数がrecord型を返す時、この関数ではレコード変数を使用してその結果を保持することができますが、これはレコード変数としての概念とはまったく異なることを認識すべきです。両方とも、関数の作成段階では実際に行構造は不明です。しかし、レコード変数はその場でその行構造を変更できるにもかかわらず、recordを返す関数では呼び出し元の問い合わせが解析された時点で実際の構造は決定されます。

39.3.6. PL/pgSQL変数の照合

PL/pgSQL関数が照合可能なデータ型のパラメータを1つ以上保有する場合、[項22.2](#)に記述したように、実際の引数に割り当てられた照合に従って、関数呼び出し毎に照合が識別されます。照合の識別に成功した場合（すなわち、引数の間に事実上の照合における衝突がない場合）、照合可能な全てのパラメータは事実上の照合を有するとして扱われます。これは関数内部において、照合に依存する操作の作用に影響します。以下の例を考えてください。

```
CREATE FUNCTION less_than(a text, b text) RETURNS boolean AS $$
BEGIN
    RETURN a < b;
END;
$$ LANGUAGE plpgsql;

SELECT less_than(text_field_1, text_field_2) FROM table1;
SELECT less_than(text_field_1, text_field_2 COLLATE "C") FROM table1;
```

第一の使用法においてless_thanは、text_field_1とtext_field_2の比較のための通常の照合として用いられます。第二の使用法においては、c照合として用いられます。

さらに、識別された照合は、照合可能なデータ型の全ての局所変数の照合としても仮定されます。したがって、この関数は下に記述する関数と差異なく作動します。

```
CREATE FUNCTION less_than(a text, b text) RETURNS boolean AS $$
DECLARE
    local_a text := a;
    local_b text := b;
BEGIN
    RETURN local_a < local_b;
END;
$$ LANGUAGE plpgsql;
```

照合可能なデータ型のパラメータが存在しない場合、または、それらで共通する照合順序を識別できない場合、パラメータと局所変数は自身のデータ型のデフォルトの照合順序（通常これはデータベースのデフォルトの照合順序ですが、ドメイン型の変数の場合は異なるかもしれません）を使用します。

照合可能なデータ型の局所変数は、宣言内でCOLLATEオプションを含めることにより、別の照合と関連づけることができます。例を示します。

```
DECLARE
    local_a text COLLATE "en_US";
```

このオプションは上記ルールにより、変数に他の方法で付与されるはずであった照合を上書きします。

また当然ながら、強制的に特定の操作において特定の照合順序を使用したい場合、明示的なCOLLATE句を関数内部に記述することができます。例を示します。

```
CREATE FUNCTION less_than_c(a text, b text) RETURNS boolean AS $$
BEGIN
    RETURN a < b COLLATE "C";
END;
$$ LANGUAGE plpgsql;
```

単純な SQL コマンドで起こるように、これはテーブルの列、パラメータ、または式の中の局所変数に関連づけられた照合を上書きします

39.4. 式

PL/pgSQL文で使用される式は全て、サーバの主SQLエクゼキュータを使用して処理されます。例えば、以下のPL/pgSQL文

```
IF expression THEN ...
```

が記述された時、PL/pgSQLは

```
SELECT expression
```

を主SQLエンジンに供給して、上式を評価します。[項39.10.1](#)において詳細を説明したように、`SELECT` コマンドの形成においてPL/pgSQL変数名は、その都度パラメータによって置換されます。これにより、`SELECT`の問い合わせ計画は一度だけ準備することができ、その後の評価で異なった変数値を代入して再利用されます。すなわち、式の最初の使用においては、実質的に`PREPARE`コマンドと同等です。例えば、2つの整数変数`x`と`y`を宣言して、

```
IF x < y THEN ...
```

という条件式を記述すると背後では

```
PREPARE statement_name(integer, integer) AS SELECT $1 < $2;
```

と同等なプリペアドステートメントが作成されます。そして`IF`文を実行する度にPL/pgSQLの最新の変数値をパラメータ値として供給して、このプリペアドステートメントに対して`EXECUTE`を行います。通常この詳細は、PL/pgSQLユーザにとって重要ではありませんが、この知識は問題点の解析に有用です。それ以外の情報は、[項39.10.2](#)に記述されています。

39.5. 基本的な文

本節および次節では、明示的にPL/pgSQLで解釈される、全ての種類の文について説明します。これらの種類の文として認められないものは全て、SQLコマンドであると仮定され、[項39.5.2](#)および[項39.5.3](#)において記述したように、メインデータベースエンジンに送信され実行されます。

39.5.1. 代入

値をPL/pgSQL変数に代入する場合は以下のように記述します。

```
variable := expression;
```

上述した通り、このような文中にある式は、メインデータベースエンジンに送信されるSELECT SQLコマンドによって評価されます。式は1つの値を生成しなければなりません(変数が行変数またはレコード変数の場合は行値となるかもしれませんが)。対象の変数は単純な変数(ブロック名で修飾可能)、行変数またはレコード変数のフィールド、または単純な変数またはフィールドとなる配列要素とすることができます。

式の結果データ型が変数のデータ型に一致しない場合、または、変数が(char(20)のように)特定の大きさ/精度を持つ場合、結果の値はPL/pgSQLインタプリタによって、結果の型の出力関数と変数の型の入力関数を使用して暗黙的に変換されます。これにより、結果値の文字列形式を入力関数で受け付けることができない場合に、入力関数において実行時エラーが発生する可能性があることに注意してください。

例：

```
tax := subtotal * 0.06;
my_record.user_id := 20;
```

39.5.2. 結果を伴わないコマンドの実行

例えば、RETURNING句のないINSERTのように、行を返さない任意のSQLのコマンドについては、単にそのコマンドを記述することによってPL/pgSQL関数の内部でコマンドを実行できます。

コマンドテキストに現れる全てのPL/pgSQL変数名は、パラメータとして扱われます。その後、実行時のパラメータ値として、その時点の変数値が提供されます。これは以前に述べた式に関する処理と同じです。[項39.10.1](#)を参照してください。

SQLコマンドがこのように実行されると、[項39.10.2](#)に記述したように、PL/pgSQLはコマンドのために、実行計画をキャッシュして再利用します。

式またはSELECT問い合わせを評価して結果を破棄することが、役に立つ場合があります。例えば、関数の呼び出しにおいて、副次的な成果を取得できるが、結果は無用である場合です。このような時PL/pgSQLでは、PERFORM文を使用してください。

```
PERFORM query;
```

これはqueryを実行し、その結果を破棄します。SQLのSELECT文と同じ方法でqueryを記述しますが、最初のキーワードSELECTをPERFORMに置き換えてください。WITH問い合わせに対しては、PERFORMを使用して、問い合わせをカッコ内に配置してください。(この場合、問い合わせは1行だけ返すことができます。)結果を返さないコマンドと同様に、PL/pgSQL変数は問い合わせ内に置き換えられ、計画は同様にキャッシュされます。また、特殊な変数であるFOUNDは問い合わせ結果が1行でも生成された場合は真に設定され、生成されない場合は偽に設定されます([項39.5.5](#)を参照してください)。

注意: 直接SELECTを記述すれば、この結果を得ることができるかもしれませんが、現時点でこれを行う方法はPERFORMしかありません。SELECTのように行を返すSQLコマンドは、エラーとして拒絶されます。なお、INTO句を有する時は例外であり、次節で説明します。

以下に例を示します。

```
PERFORM create_mv('cs_session_page_requests_mv', my_query);
```

39.5.3. 1行の結果を返す問い合わせの実行

(多分、複数列の)1行を返すSQLコマンドの結果は、レコード変数、行型の変数、スカラ変数のリストに代入することができます。これは、基本的なSQLコマンドを記述して、それにINTO句を追加することによって行われます。以下に例を示します。

```
SELECT select_expressions INTO [STRICT] target FROM ...;
INSERT ... RETURNING expressions INTO [STRICT] target;
```



```
UPDATE ... RETURNING expressions INTO [STRICT] target;  
DELETE ... RETURNING expressions INTO [STRICT] target;
```

ここで、*target*はレコード変数、行変数、あるいは、単純な変数とレコード/行変数のフィールドをカンマで区切ったリストです。 PL/pgSQL変数により残りの問い合わせが置換され、行を返さないコマンドにおいて述べたように計画がキャッシュされます。 このように作動するのは、RETURNINGを伴ったINSERT/UPDATE/DELETEとSELECTおよび行セットの結果を返すユーティリティコマンド（例えば、EXPLAIN）です。 INTO句以外では、SQLコマンドはPL/pgSQLの外部に記述したものと同じです。

ティップ: 通常のPostgreSQLのSELECT INTO文では、INTOの対象は新しく作成されるテーブルです。 しかし、INTOを伴ったSELECTでは、この解釈が通常と大きく異なることに注意してください。 PL/pgSQL関数内部でSELECTの結果からテーブルを作成したい場合は、CREATE TABLE ... AS SELECT構文を使用してください。

行または変数リストが対象に使用された場合、列数とデータ型において問い合わせの結果と対象の構造が正確に一致しなければなりません。 さもないと、実行時エラーが発生します。 レコード変数が対象の場合は、問い合わせ結果の列の行型に自身を自動的に設定します。

INTO句はSQLコマンドのほとんど任意の場所に記述することができます。 習慣的には、SELECT文においてはselect_expressionsの直前または直後に、他のコマンドにおいては文の終わりに記述されます。 将来のバージョンでPL/pgSQLのパースがより厳格になる場合に備えて、この習慣に従うことを推奨します。

INTO句においてSTRICTが指定されない場合、*target*は問い合わせが返す最初の行となり、行を返さない時はNULLとなります。（"最初の行"とはORDER BYを使用しないと定義できないことに注意してください。） 2行目以降の行の結果は、全て破棄されます。 以下のように、特殊なFOUND変数（[項39.5.5](#)を参照してください）を調べて、行が返されたかどうかを検査することができます。

```
SELECT * INTO myrec FROM emp WHERE empname = myname;  
IF NOT FOUND THEN  
    RAISE EXCEPTION 'employee % not found', myname;  
END IF;
```

STRICTオプションが指定された場合、問い合わせは正確に1行を返さなければなりません。 さもないと、行がない時はNO_DATA_FOUND、2行以上が返った時はTOO_MANY_ROWSという実行時エラーが生じます。 エラーを捕捉したい時は、例外ブロックを使用できます。 以下に例を示します。

```
BEGIN  
    SELECT * INTO STRICT myrec FROM emp WHERE empname = myname;  
    EXCEPTION  
        WHEN NO_DATA_FOUND THEN  
            RAISE EXCEPTION 'employee % not found', myname;  
        WHEN TOO_MANY_ROWS THEN  
            RAISE EXCEPTION 'employee % not unique', myname;  
END;
```

STRICTを指定したコマンドが成功すると、FOUND変数は常に真に設定されます。

STRICTが指定されない場合でも、RETURNINGを伴ったINSERT/UPDATE/DELETEが2行以上を返した時は、エラーとなります。 なぜなら、どの1行を返すか決定するORDER BYのようなオプションが存在しないからです。

注意: STRICTオプションは、OracleのPL/SQLのSELECT INTOおよび関連した文に対応します。

SQLの問い合わせが返す複数行の結果を処理したい場合は、[項39.6.4](#)を参照してください。

39.5.4. 動的コマンドの実行

PL/pgSQL関数の内部で、動的コマンド、つまり実行する度に別のテーブルや別のデータ型を使用するコマンドを生成したいということがよくあるでしょう。 PL/pgSQLが通常行うコマンドの計画のキャッシュは（[項39.10.2](#)で述べたように）このような状況では動作しません。 この種の問題を扱うために、以下のEXECUTE文が用意されています。

```
EXECUTE command-string [ INTO [STRICT] target ] [ USING expression [, ... ] ];
```

ここで、*command-string*は実行されるコマンドを含む（text型の）文字列を生成する式です。 オプションの*target*はレコード変数、行変数、あるいは、単純な変数とレコード/行変数のフィールドをカンマで区切ったリストで、その中にコマンドの結果が格納されます。 オプションのUSING式は コマンドに挿入される値を与えます。

PL/pgSQL変数は、この演算用のコマンド文字列へ置換されません。 必要な変数の値はすべてコマンド文字列を作成する時に埋め込まなければなりません。 もしくは、以下に説明するパラメータを使用することもできます。

また、EXECUTEを介して実行されるコマンド計画をキャッシュすることはありません。 代わりに、コマンドは文が実行されるとき常に計画されます。 したがって、異なるテーブルと列に対する操作を実行できるように、コマンド文字列を関数内部で動的に作成することができます。

INTO句は、行を返すSQLコマンドの結果を代入すべき場所を指定します。行または変数リストが用いられる時、それは問い合わせの結果の構造と正確に一致しなければなりません（レコード変数が使用される時、自動的に結果の構造と一致するように自身を構築させます）。複数の行が返された時、最初の行だけがINTO変数に代入されます。1行も返されない時、NULLがINTO変数に代入されます。INTO句が指定されない時、問い合わせの結果は捨てられます。

STRICTオプションが指定された時、問い合わせの結果が正確に1行の場合を除き、エラーとなります。

コマンド文字列はパラメータ値を使用可能で、それらは\$1、\$2等としてコマンドの中で参照されます。これらの記号はUSINGで与えられる値を参照します。この方式はデータの値をテキストとしてコマンド文字列の中に挿入する際、よく好まれます。それは値をテキストに変換、そしてその逆を行う場合の実行時オーバーヘッドを防止するとともに、引用符付けするとか、エスケープをする必要がないため、SQLインジェクション攻撃に対してより襲われにくくなります。以下に例を示します。

```
EXECUTE 'SELECT count(*) FROM mytable WHERE inserted_by = $1 AND inserted <= $2'
INTO c
USING checked_user, checked_date;
```

パラメータ記号はデータ値のみ使用可能です。もし動的に決定されるテーブルや列名を使用したい場合、テキストでコマンド文字列にそれらを挿入する必要があります。例えば、先行する問い合わせが、動的に選択されたテーブルに対して処理される必要がある時は、次のようにします。

```
EXECUTE 'SELECT count(*) FROM '
|| tabname::regclass
|| ' WHERE inserted_by = $1 AND inserted <= $2'
INTO c
USING checked_user, checked_date;
```

他にもパラメータ記号はSELECT、INSERT、UPDATE、DELETEコマンドでしか動作しない、という制限があります。他の種類の文(一般的にユーティリティ文と呼ばれます)では、単なるデータ値であったとしてもテキストの値として埋め込まなければなりません。

最初の例のように、単純な定数コマンドとUSINGパラメータを使ったEXECUTEは、コマンドを直接PL/pgSQLで書いて、PL/pgSQL変数を自動的に置換したものと機能的に同じです。重要な差異として、EXECUTEが現在のパラメータ値に特化した計画を生成し、コマンドを実行する度に計画を再作成することです。一方、PL/pgSQLはその他に一般的な計画を作成し、再使用に備えキャッシュします。最適な計画がパラメータ値に大きく依存する場合、一般的な計画が選択されないことを確保するために、EXECUTEの使用は助けになります。

SELECT INTOはEXECUTEでは現在サポートされません。代わりに、普通のSELECTコマンドを実行し、EXECUTEの一部としてINTOを記述してください。

注意: PL/pgSQL EXECUTE文はPostgreSQLサーバでサポートされているEXECUTE SQL文とは関連がありません。サーバのEXECUTE文はPL/pgSQL関数内で使用することはできません（使用する必要もありません）。

例 39-1. 動的問い合わせの中の値の引用符付け

動的コマンドを使用する時、しばしばPL/pgSQLでは単一引用符をエスケープしなければなりません。関数本体における固定のテキストを引用符付けする推奨方法は、ドル引用符を使用する方法です。（ドル引用符を用いない旧式のコードを保有している場合は、[項39.11.1](#)の概要を参照することが、理解しやすいコードへの変換作業の手助けになります）。

作成した問い合わせに挿入すべき動的な値は、それ自身の内部に引用符を含む可能性があるため、注意深い処理が必要です。以下に例を示します（ここでは関数にドル引用符を用いる方法を使用すると仮定しているので、引用符を二重化する必要はありません）。

```
EXECUTE 'UPDATE tbl SET '
|| quote_ident(colname)
|| ' = '
|| quote_literal(newvalue)
|| ' WHERE key = '
|| quote_literal(keyvalue);
```

この例は、quote_identとquote_literal関数（[項9.4](#)を参照）の使用方法を示しています。安全のため、列またはテーブル識別子を含む式は動的問い合わせに挿入する前にquote_identを介して渡されなくてはなりません。組み立てられるコマンドの中のリテラル文字列となるはずの値を含む式は、quote_literalを介して渡されなければなりません。これらの関数は、すべての特殊文字を適切にエスケープして埋め込んだ、二重引用符または単一引用符で囲まれた入力テキストを返すために、適切な手順を踏みます。

quote_literalはSTRICTラベル付けされているため、NULL引数で呼び出された場合、常にNULLを返します。上記の例で、newvalueまたはkeyvalueがNULLの場合、動的問合せ文字列全体がNULLとなり、EXECUTEからのエラーを導きます。quote_nullable関数を使用することで、この問題を回避することができます。その動作は、NULL引数付きで呼び出された場合に文字列NULLを返すことを除いてquote_literalと同一です。以下に例を示します。

```
EXECUTE 'UPDATE tbl SET '
```

```

|| quote_ident(colname)
|| ' = '
|| quote_nullable(newvalue)
|| ' WHERE key = '
|| quote_nullable(keyvalue);

```

NULLの可能性のある値を処理するのであれば、通常`quote_literal`の代わりに`quote_nullable`を使用しなければなりません。

いつものように、問い合わせの中のNULL値は意図しない結果を確実にもたさないよう配慮をしなければなりません。例えば次のようなWHERE句の結果はどうなるのでしょうか。

```
'WHERE key = ' || quote_nullable(keyvalue)
```

これは`keyvalue`がNULLである限り成功しません。その理由は、等価演算子`=`をNULLオペランドで使用するとその結果は常にNULLとなるからです。NULLを通常のキーの値と同じように動作させたい場合、上記を、以下のように書き換えなければなりません。

```
'WHERE key IS NOT DISTINCT FROM ' || quote_nullable(keyvalue)
```

(現時点では、`IS NOT DISTINCT FROM`は=よりもより効率性が少なく扱われますので、必要に迫られた場合以外は行わないようにしてください。NULLと`IS DISTINCT`についての更なる情報は[項9.2](#)を参照してください。)

ドル引用符は固定のテキストを引用符付けする場合のみ有用であるということに注意してください。この例を次のように記述するのは非常に悪い考えです。

```

EXECUTE 'UPDATE tbl SET '
|| quote_ident(colname)
|| ' = $$'
|| newvalue
|| '$$ WHERE key = '
|| quote_literal(keyvalue);

```

なぜなら、`newvalue`の内容がたまたま`$$`を含む時は、途中で次の処理へ移ってしまうからです。同様の不測事態は、ドル引用符の他の区切り文字を選んだ時も起こります。したがって、テキストの内容を把握していない時は、安全にテキストを引用符付けするために、`quote_literal`、`quote_nullable`、または`quote_ident`関数を適切に使用しなければなりません。

動的 SQL 文は`format`関数を使用しても安全に作成できます ([項9.4](#)参照)。例を示します。

```
EXECUTE format('UPDATE tbl SET %I = %L WHERE key = %L', colname, newvalue, keyvalue);
```

`format`関数は`USING`句と共に使用できます。

```
EXECUTE format('UPDATE tbl SET %I = $1 WHERE key = $2', colname)
USING newvalue, keyvalue;
```

この形式はさらに有効です。なぜなら、`newvalue`と`keyvalue`パラメータは、テキスト型に変換されないからです。

動的問い合わせと`EXECUTE`の長大な例は[例39-8](#)にあります。それは新しい関数を定義するために`CREATE FUNCTION`コマンドを組み立て実行するものです。

39.5.5. 結果ステータスの取得

コマンドの効果を判断するにはいくつか方法があります。最初の方法は以下のような形式の`GET DIAGNOSTICS`を使用する方法です。

```
GET [ CURRENT ] DIAGNOSTICS variable = item [ , ... ];
```

このコマンドによってシステムステータスインジケータを取り出すことができます。各`item`は、指定された変数（これは受け取るために正しいデータ型でなければなりません）に代入されるステータス値を識別するキーワードです。現在使用可能なステータス項目は、SQLエンジンに送信された最終SQLコマンドにより処理された行数を示す`ROW_COUNT`、および、最も最近のSQLコマンドにより挿入された最後の行のOIDを示す`RESULT_OID`です。`RESULT_OID`はOIDを保有するテーブルへの`INSERT`コマンドの後でのみ有意であることに注意してください。

以下に例を示します。

```
GET DIAGNOSTICS integer_var = ROW_COUNT;
```

コマンドの効果を判断する2番目の方法は、`FOUND`という`boolean`型の特殊な変数を検査することです。PL/pgSQLの各関数呼び出しで使用される際、`FOUND`は最初は偽に設定されています。以下のように、それぞれの文の種類によって設定が変更されます。

- `SELECT INTO`文は、行が代入された場合は真、返されなかった場合は偽を`FOUND`に設定します。

- `PERFORM` 文は、1つ以上の行が生成（破棄）された場合は真、まったく生成されなかった場合は偽を `FOUND` に設定します。
- `UPDATE`、`INSERT`、および `DELETE` 文は、少なくとも1行が影響を受けた場合は真、まったく影響を受けなかった場合は偽を `FOUND` に設定します。
- `FETCH` 文は、行が返された場合は真、まったく返されなかった場合は偽を `FOUND` に設定します。
- `MOVE` 文は、カーソルの移動が成功したが場合は真、失敗した場合は偽を `FOUND` に設定します。
- `FOR` 文または `FOREACH` 文は、1回以上繰り返しが行われた場合は真、行われなかった場合は偽を `FOUND` に設定します。
`FOUND` はループが終了した際、このように設定されます。ループ実行中はループ文による `FOUND` の変更はありません。ただし、ループ本体内の他種類の文を実行することによって、変更されるかもしれません。
- `RETURN QUERY` と `RETURN QUERY EXECUTE` 文は、問い合わせが行を1つでも返せば真、行が返されなければ偽を設定します。

他のPL/pgSQL文は `FOUND` の状態を変更しません。特に、`EXECUTE` は `GET DIAGNOSTICS` の出力を変更しますが、`FOUND` を変更しないことに注意してください。

`FOUND` はそれぞれのPL/pgSQL関数内部のローカル変数です。`FOUND` に対して行われた全ての変更は、現在の関数にのみ影響します。

39.5.6. まったく何もしない

何もしないプレースホルダ文が有用になることがあります。例えば、`IF/THEN/ELSE` 文の一部が空文であることを明示したい時です。このような目的には `NULL` 文を使用します。

```
NULL;
```

例えば、次の2つのコードは同等です。

```
BEGIN
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN
        NULL; -- 誤りを無視する
END;
```

```
BEGIN
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN -- 誤りを無視する
END;
```

どちらが望ましいと思うかは、好みの問題です。

注意: OracleのPL/SQLでは無記述の文は許されませんので、こうした状況では `NULL` 文が必須です。しかしPL/pgSQLでは無記述の文が許可されています。

[前のページ](#)
式

[ホーム](#)
[上に戻る](#)

[次のページ](#)
制御構造

39.6. 制御構造

制御構造はおそらくPL/pgSQLの最も有用（かつ重要）な部分です。PL/pgSQLの制御構造を使用して、PostgreSQLのデータを非常に柔軟、強力に操作することができます。

39.6.1. 関数からの復帰

関数からデータを返すために使用できるコマンドが2つあります。RETURNおよびRETURN NEXTです。

39.6.1.1. RETURN

```
RETURN expression;
```

式を持つRETURNは関数を終了し、*expression*の値を呼び出し元に返します。この形式は集合を返さないPL/pgSQL関数で使用されます。

スカラ型を返す場合は任意の式を使用することができます。この式の結果は、代入のところで説明したように、自動的に関数の戻り値の型にキャストされます。複合（行）値を返すためには、*expression*としてレコードもしくは行変数を記述しなければなりません。

出力パラメータを持った関数を宣言した時は、式の無いRETURNを記述してください。その時点における出力パラメータの値が返されます。

voidを返すように関数を宣言した場合でも、関数を直ちに抜け出すためにRETURNを使用できますが、RETURNの後に式を記述しないでください。

関数の戻り値は未定義とさせたままにすることはできません。制御が、RETURN文が見つからない状態で関数の最上位のブロックの終わりまで達した時、実行時エラーが発生します。しかし、この制限は出力パラメータを持った関数及びvoidを返す関数には当てはまりません。このような場合は最上位のブロックが終わった時、RETURN文が自動的に実行されます。

39.6.1.2. RETURN NEXTおよびRETURN QUERY

```
RETURN NEXT expression;
RETURN QUERY query;
RETURN QUERY EXECUTE command-string [ USING expression [, ...] ];
```

PL/pgSQL関数がSETOF sometypeを返すように宣言した場合、後続の処理が多少違います。この場合、戻り値の個々の項目は、RETURN NEXTコマンドまたはRETURN QUERYコマンドで指定されます。そして、引数のない最後のRETURNコマンドにより、関数が実行を終了したことが示されます。RETURN NEXTは、スカラ型および複合型の両方で使用することができます。複合型の場合、結果の"テーブル"全体が返されます。RETURN QUERYは、問い合わせを実行した結果を関数の結果集合に追加します。RETURN NEXTとRETURN QUERYは、単一の集合を返す関数の中で自由に混合できます。この場合、連結されたものが結果となります。

実際には、RETURN NEXTおよびRETURN QUERYは関数から戻りません。単に関数の結果集合に行を追加しているだけです。そして、その実行はPL/pgSQL関数内の次の文に継続します。RETURN NEXTまたはRETURN QUERYコマンドが連続して実行されると、結果集合が作成されます。最後の、引数を持つてはならないRETURNにより、関数の終了を制御します（または制御を関数の最後に移すことができます）。

RETURN QUERYにはRETURN QUERY EXECUTEという亜種があり、それは問い合わせが動的に実行されることを指定します。パラメータ式を、EXECUTEコマンド内と全く同じように、USINGによって演算された問い合わせ文字列に挿入することができます。

出力パラメータを持つ関数を宣言した時は、式の無いRETURN NEXTだけを記述してください。実行の度に、その時点における出力パラメータの値が、関数からの戻り値のために結果の行として保存されます。出力パラメータを持つ集合を返す関数を作成するためには、出力パラメータが複数の時はSETOF recordを返すように関数を宣言し、単一のsometype型の出力パラメータの時はSETOF sometypeを返すように関数を宣言しなければならないことに注意してください。

RETURN NEXTを使用する関数の例を以下に示します。

```
CREATE TABLE foo (fooid INT, foosubid INT, fooname TEXT);
INSERT INTO foo VALUES (1, 2, 'three');
INSERT INTO foo VALUES (4, 5, 'six');

CREATE OR REPLACE FUNCTION getAllFoo() RETURNS SETOF foo AS
$BODY$
DECLARE
```

```

        r foo%rowtype;
BEGIN
    FOR r IN SELECT * FROM foo
    WHERE fooid > 0
    LOOP
        -- can do some processing here
        RETURN NEXT r; -- return current row of SELECT
    END LOOP;
    RETURN;
END
$BODY$
LANGUAGE 'plpgsql' ;

SELECT * FROM getallfoo();

```

注意: 上記のように、`RETURN NEXT`および`RETURN QUERY`の現在の実装では、関数から返される前に結果集合全体を保管します。これにより、PL/pgSQL関数が非常に大量の結果集合を返した場合、性能が低下する可能性があります。メモリの枯渇を避けるため、データはディスクに書き込まれます。しかし、関数自体は結果集合全体が生成されるまでは戻りません。将来のPL/pgSQLのバージョンでは、この制限を受けずに集合を返す関数をユーザが定義できるようになるかもしれません。現在、データがディスクに書き込まれ始まる時点は`work_mem`設定変数によって制御されています。大量の結果集合を保管するのに十分なメモリがある場合、管理者はこのパラメータの値を大きくすることを考慮すべきです。

39.6.2. 条件分岐

`IF`と`CASE`文はある条件に基づいて代わりのコマンドを実行させます。PL/pgSQLには、以下のような3つの`IF`の形式があります。

- `IF ... THEN`
- `IF ... THEN ... ELSE`
- `IF ... THEN ... ELSE IF`
- `IF ... THEN ... ELSIF ... THEN ... ELSE`

また、以下のような2つの`CASE`の形式があります。

- `CASE ... WHEN ... THEN ... ELSE ... END CASE`
- `CASE WHEN ... THEN ... ELSE ... END CASE`

39.6.2.1. IF-THEN

```

IF boolean-expression THEN
    statements
END IF;

```

`IF-THEN`文は最も単純な`IF`の形式です。`THEN`と`END IF`の間の文が条件が真の場合に実行されます。さもなければそれらは飛ばされます。

例：

```

IF v_user_id <> 0 THEN
    UPDATE users SET email = v_email WHERE user_id = v_user_id;
END IF;

```

39.6.2.2. IF-THEN-ELSE

```

IF boolean-expression THEN
    statements
ELSE
    statements
END IF;

```

`IF-THEN-ELSE`文は`IF-THEN`に加え、条件評価が偽の場合に実行すべき代替となる文の集合を指定することができます。（これには条件が`NULL`と評価した場合も含まれることに注意してください。）

例：

```

IF parentid IS NULL OR parentid = ''
THEN
    RETURN fullname;
ELSE

```

```

        RETURN hp_true_filename(parentid) || '/' || fullname;
END IF;

IF v_count > 0 THEN
    INSERT INTO users_count (count) VALUES (v_count);
    RETURN 't';
ELSE
    RETURN 'f';
END IF;

```

39.6.2.3. IF-THEN-ELSIF

```

IF boolean-expression THEN
    statements
[ ELSEIF boolean-expression THEN
    statements
[ ELSEIF boolean-expression THEN
    statements
    ...]]
[ ELSE
    statements ]
END IF;

```

選択肢が2つだけではなくより多くなる場合があります。IF-THEN-ELSIFは、順番に複数の代替手段を検査する、より便利な方法を提供します。IF条件は最初の真である結果が見つかるまで連続して検査されます。そして関連した文が実行され、その後END IF以降の次の文に制御が渡されます。（以降にあるIF条件の検査はすべて実行されません。）全てのIF条件が真でない場合、ELSEブロックが（もし存在すれば）実行されます。機能的には、IF-THEN-ELSE-IF-THENコマンドを入れ子にしたものと同じですが、必要なEND IFは1つだけです。

以下に例を示します。

```

IF number = 0 THEN
    result := 'zero';
ELSIF number > 0 THEN
    result := 'positive';
ELSIF number < 0 THEN
    result := 'negative';
ELSE
    -- ふうむ、残る唯一の可能性はその値がNULLであることだ
    result := 'NULL';
END IF;

```

ELSIFキーワードはELSEIFのように書くことができます。

同じ作業を遂行する別の方法は、以下の例のようにIF-THEN-ELSE文を入れ子にすることです。

```

IF demo_row.sex = 'm' THEN
    pretty_sex := 'man';
ELSE
    IF demo_row.sex = 'f' THEN
        pretty_sex := 'woman';
    END IF;
END IF;

```

しかし、この方法はそれぞれのIFに対応するEND IFの記述が必要です。従って、多くの選択肢がある場合ELSIFを使用するよりも厄介です。

39.6.2.4. 単純なCASE

```

CASE search-expression
    WHEN expression [, expression [ ... ]] THEN
        statements
    [ WHEN expression [, expression [ ... ]] THEN
        statements
        ... ]
    [ ELSE
        statements ]
END CASE;

```

CASEの単純な形式はオペランドの等価性にもとづく条件的実行を提供します。search-expressionは（一度だけ）評価され、その後WHEN句内のそれぞれのexpressionと比較されます。一致するものが見つかったら、関連したstatementsが実行され、END CASEの次の文に制御が渡されます。（以降のくWHEN式は評価されません。）一致するものが見つからない場合、ELSE statementsが実行されますが、ELSEが無いときはCASE_NOT_FOUND例外を引き起こします。

以下は簡単な例です。

```

CASE x
    WHEN 1, 2 THEN
        msg := 'one or two';

```



```

ELSE
    msg := 'other value than one or two';
END CASE;

```

39.6.2.5. 検索付きCASE

```

CASE
    WHEN boolean-expression THEN
        statements
    [ WHEN boolean-expression THEN
        statements
        ... ]
    [ ELSE
        statements ]
END CASE;

```

CASEの検索された形式は論理値式の真の結果に基づく条件付き実行を提供します。それぞれのWHEN句の*boolean-expression*はtrueとなる1つが見つかるまで順番に評価されます。その後、関連する*statements*が実行され、その結果END CASEの次の文に制御が渡されます。（以降のWHEN式は評価されません。）真となる結果が見つからない場合、ELSE *statements*が実行されますが、ELSEが存在しないときはCASE_NOT_FOUND例外を引き起こします。

以下は簡単な例です。

```

CASE
    WHEN x BETWEEN 0 AND 10 THEN
        msg := 'value is between zero and ten';
    WHEN x BETWEEN 11 AND 20 THEN
        msg := 'value is between eleven and twenty';
END CASE;

```

この形式のCASEは、判定基準が省略されたELSE句に達した場合に何もしないのではなくエラーなる点を除き、IF-THEN-ELSIFと全く同一です。

39.6.3. 単純なループ

LOOP、EXIT、CONTINUE、WHILE、FOR、FOREACH文を使用して、PL/pgSQL関数で、一連のコマンドを繰り返すことができます。

39.6.3.1. LOOP

```

[ <<label>> ]
LOOP
    statements
END LOOP [ label ];

```

LOOPは、EXIT文またはRETURN文によって終了されるまで無限に繰り返される、条件なしのループを定義します。省略可能な*label*は、入れ子状になったループにおいてEXITおよびCONTINUE文がどのレベルの入れ子を参照するかを指定するために使用されます。

39.6.3.2. EXIT

```

EXIT [ label ] [ WHEN boolean-expression ];

```

*label*が指定されない場合、最も内側のループを終わらせ、END LOOPの次の文がその後に実行されます。*label*が指定された場合、それは現在またはその上位の入れ子になったループやブロックのラベルである必要があります。その後、指名されたループまたはブロックを終わらせ、そのループまたはブロックの対応するENDの次の文に制御を移します。

WHENが指定された場合、*boolean-expression*が真の場合のみループの終了が起こります。さもなければ、EXITの後の行に制御が移ります。

EXITは、すべての種類のループと共に使用できます。条件なしのループでの使用に限定されません。

BEGINブロックと共に使用した時、EXITによりブロックの次の文に制御が移ります。この目的のためにラベルが使用されなければならないことに注意してください。ラベル無しのEXITはBEGINブロックに対応するとは決して考えられません。（これは、ラベル無しのEXITがBEGINブロックに対応することを許容するPostgreSQLの8.4より前のリリースからの変更です。）

例：

```

LOOP
    -- 何らかの演算
    IF count > 0 THEN
        EXIT; -- ループを抜け出す
    END IF;
END LOOP;

```



```

LOOP
  -- 何らかの演算
  EXIT WHEN count > 0;  -- 上例と同じ結果
END LOOP;

<<ablock>>
BEGIN
  -- 何らかの演算
  IF stocks > 100000 THEN
    EXIT ablock;  -- これによりBEGINブロックを抜け出す
  END IF;

  -- stokcs > 100000 であればここでの演算は省略
END;

```

39.6.3.3. CONTINUE

```
CONTINUE [ label ] [ WHEN boolean-expression ];
```

*label*が無い場合、すぐ外側のループの次の繰り返しが開始されます。すなわち、ループ本体の残りの文は飛ばされて、他のループの繰り返しが必要かどうかを決めるため、制御がループ制御式(もし存在すれば)に戻ります。*label*が存在する場合、実行を継続するループのラベルを指定します。

WHENが指定された場合、*boolean-expression*が真の場合のみループにおける次の繰り返しが始まります。さもなければ、CONTINUEの後の行に制御が移ります。

CONTINUEは全ての種類のループで使用可能です。条件なしのループに限定されません。

例

```

LOOP
  -- 何らかの演算
  EXIT WHEN count > 100;
  CONTINUE WHEN count < 50;
  -- 50から100を数える、何らかの演算
END LOOP;

```

39.6.3.4. WHILE

```

[ <<label>> ]
WHILE boolean-expression LOOP
  statements
END LOOP [ label ];

```

WHILE文は*boolean-expression*の評価が真である間、一連の文を繰り返します。条件式は、ループ本体に入る前にのみ検査されます。

以下に例を示します。

```

WHILE amount_owed > 0 AND gift_certificate_balance > 0 LOOP
  -- ここで演算をいくつか行います。
END LOOP;

WHILE NOT done LOOP
  -- ここで演算をいくつか行います。
END LOOP;

```

39.6.3.5. FOR (整数FORループ)

```

[ <<label>> ]
FOR name IN [ REVERSE ] expression .. expression [ BY expression ] LOOP
  statements
END LOOP [ label ];

```

この形式のFORは整数値の範囲内で繰り返すループを生成します。*name*変数はinteger型として自動的に定義され、ループ内部のみで存在します(ループ外部で定義しても、ループ内部では全て無視されます)。範囲の下限、上限として与えられる2つの式はループに入った時に一度だけ評価されます。BY句を指定しない時の繰り返し刻みは1ですが、BY句を用いて指定でき、ループに入った時に一度だけ評価されます。REVERSEが指定された場合、繰り返し刻みの値は加算されるのではなく、繰り返しごとに減算されます。

整数FORループの例を以下に示します。

```

FOR i IN 1..10 LOOP
  -- i はループ内で 1、2、3、4、5、6、7、8、9、10 の値を取ります。
END LOOP;

```

```

FOR i IN REVERSE 10..1 LOOP
    -- i はループ内で 10、9、8、7、6、5、4、3、2、1 の値を取ります。
END LOOP;

FOR i IN REVERSE 10..1 BY 2 LOOP
    -- i はループ内で 10、8、6、4、2 の値を取ります。
END LOOP;

```

下限が上限よりも大きい（`REVERSE`の場合はより小さい）場合、ループ本体はまったく実行されません。エラーは発生しません。

`label`を`FOR`ループに付加することにより、`label`を用いて修飾した名前の整数ループ変数を参照できます。

39.6.4. 問い合わせ結果による繰り返し

別の種類の`FOR`ループを使用して、問い合わせの結果を繰り返し、そのデータを扱うことができます。以下に構文を示します。

```

[ <<label>> ]
FOR target IN query LOOP
    statements
END LOOP [ label ];

```

`target`は、レコード変数、行変数またはカンマで区切ったスカラ変数のリストです。`target`には順次、`query`の結果の全ての行が代入され、各行に対してループ本体が実行されます。以下に例を示します。

```

CREATE FUNCTION cs_refresh_mviews() RETURNS integer AS $$
DECLARE
    mviews RECORD;
BEGIN
    RAISE NOTICE 'Refreshing materialized views...';

    FOR mviews IN SELECT * FROM cs_materialized_views ORDER BY sort_key LOOP
        -- ここで"mviews"はcs_materialized_viewsの1つのレコードを持ちます
        RAISE NOTICE 'Refreshing materialized view %s ...', quote_ident(mviews.mv_name);
        EXECUTE 'TRUNCATE TABLE ' || quote_ident(mviews.mv_name);
        EXECUTE 'INSERT INTO '
            || quote_ident(mviews.mv_name) || ' '
            || mviews.mv_query;
    END LOOP;

    RAISE NOTICE 'Done refreshing materialized views.';
    RETURN 1;
END;
$$ LANGUAGE plpgsql;

```

このループが`EXIT`文で終了した場合、最後に割り当てられた行の値はループを抜けた後もアクセスすることができます。

この種類の`FOR`文の`query`としては、呼び出し元に行を返すSQLコマンドをすべて使用できます。通常は`SELECT`ですが、`RETURNING`句を持つ`INSERT`、`UPDATE`または`DELETE`も使用できます。`EXPLAIN`などのユーティリティコマンドも作動します。

PL/pgSQL変数は問い合わせテキストに置き換えられます。問い合わせ計画は、[項39.10.1](#)および[項39.10.2](#)で述べたように、再利用のためにキャッシュされます。

`FOR-IN-EXECUTE`文は行を繰り返すもう1つの方法です。

```

[ <<label>> ]
FOR target IN EXECUTE text_expression [ USING expression [, ... ] ] LOOP
    statements
END LOOP [ label ];

```

この方法は、問い合わせのソースが文字列式で指定される点を除き、前の形式と似ています。この式は`FOR`ループの各項目で評価され、再計画が行われます。これにより、プログラマは、通常の`EXECUTE`文と同じように事前に計画された問い合わせによる高速性と、動的な問い合わせの持つ柔軟性を選択することができます。`EXECUTE`の場合と同様、パラメータ値は`USING`により動的コマンドに挿入できます。

結果を通して繰り返さなければならない問い合わせを指定するもう1つの方法として、カーソルの宣言があります。これは[項39.7.4](#)で説明します。

39.6.5. 配列を巡回

`FOREACH`ループは`FOR`ループにとってもよく似ています。しかし、SQL 問い合わせが抽出した行を繰り返す代わりに、配列の要素を繰り返します。（一般的に`FOREACH`は、複合値で表現される構成要素の巡回を意味しますが、配列でない複合値も巡回する亜種が将来は追加されるかもしれません。）配列を巡回する`FOREACH`文を示します。

```

[ <<label>> ]

```

```
FOREACH target [ SLICE number ] IN ARRAY expression LOOP
    statements
END LOOP [ label ];
```

SLICEがない、またはSLICE 0が指定された場合、ループは*expression*によって評価されて作成された配列の各要素を繰り返します。 *target*変数が各要素の値に順次割り当てられ、各要素に対してループ本体が実行されます。 整数配列の要素を巡回する例を示します。

```
CREATE FUNCTION sum(int[]) RETURNS int8 AS $$
DECLARE
    s int8 := 0;
    x int;
BEGIN
    FOREACH x IN ARRAY $1
    LOOP
        s := s + x;
    END LOOP;
    RETURN s;
END;
$$ LANGUAGE plpgsql;
```

配列の次元数に関係なく、要素は格納した順番で処理されます。 通常*target*は単一の変数ですが、複合値（レコード）の配列を巡回するときは、変数のリストも可能です。 その場合、配列の各要素に対して、変数は複合値（レコード）の列から連続的に割り当てられます。

正のSLICE値を持つ場合、FOREACHは単一の要素ではなく多次元配列の低次元部分配列を通して繰り返します。 SLICE値は、配列の次元数より小さい整数定数でなければなりません。 *target*変数は配列でなければなりません。 この変数は、配列値から連続した部分配列を受けとります ここで部分配列はSLICEで指定した次数となります。 以下に1次元の部分配列を通した繰り返しの例を示します。

```
CREATE FUNCTION scan_rows(int[]) RETURNS void AS $$
DECLARE
    x int[];
BEGIN
    FOREACH x SLICE 1 IN ARRAY $1
    LOOP
        RAISE NOTICE 'row = %', x;
    END LOOP;
END;
$$ LANGUAGE plpgsql;

SELECT scan_rows(ARRAY[[1,2,3],[4,5,6],[7,8,9],[10,11,12]]);

NOTICE:  row = {1,2,3}
NOTICE:  row = {4,5,6}
NOTICE:  row = {7,8,9}
NOTICE:  row = {10,11,12}
```

39.6.6. エラーの捕捉

デフォルトでは、PL/pgSQL関数の内部で発生したエラーは関数の実行を中止し、実際に上位のトランザクションをアボートします。 BEGINブロックおよびEXCEPTION句を使用すれば、エラーを捕捉してその状態から回復できます。 その構文は通常のBEGINブロックの構文を拡張したものです。

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
EXCEPTION
    WHEN condition [ OR condition ... ] THEN
        handler_statements
    [ WHEN condition [ OR condition ... ] THEN
        handler_statements
        ... ]
END;
```

エラーが発生しない時、この形式のブロックは単に全ての*statements*を実行し、ENDの次の文に制御が移ります。 しかし、*statements*の内部でエラーが発生すると、それ以後の*statements*の処理は中断され、EXCEPTIONリストに制御が移ります。 そしてリストの中から、発生したエラーと合致する最初の*condition*を探します。 合致するものがあれば、対応する*handler_statements*を実行し、ENDの次の文に制御が移ります。 合致するものがなければ、EXCEPTION句が存在しないのと同じで、エラーは外側に伝播します。 EXCEPTIONを含んだ外側のブロックはエラーを捕捉できますが、失敗すると関数の処理は中断されます。

全ての*condition*の名前は[付録A](#)に示したもののいずれかを取ることができます。 分類名はそこに分類される全てのエラーに合致します。 OTHERSという特別の状態名はQUERY_CANCELED以外の全てのエラーに合致します。 (QUERY_CANCELEDを名前で捕捉することは可能ですが、賢明ではありません。) 状態名は大文字小文字を区別しません。 同時に、エラー状態はSQLSTATEコードで

指定可能です。例えば以下は等価です。

```
WHEN division_by_zero THEN ...
WHEN SQLSTATE='2Z012' THEN ...
```

エラーが該当する`handler_statements`内部で新たに発生した時、`EXCEPTION`句はそのエラーを捕捉できずエラーは外側に伝播します。なお、上位の`EXCEPTION`句はそのエラーを捕捉できます。

`EXCEPTION`句がエラーを捕捉した時、PL/pgSQL関数のローカル変数はエラーが起こった後の状態を保ちます。しかし、ブロック内部における永続的なデータベースの状態は、ロールバックされます。そのような例を以下に示します。

```
INSERT INTO mytab(firstname, lastname) VALUES('Tom', 'Jones');
BEGIN
    UPDATE mytab SET firstname = 'Joe' WHERE lastname = 'Jones';
    x := x + 1;
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN
        RAISE NOTICE 'caught division_by_zero';
        RETURN x;
END;
```

制御が変数`y`の代入に移ると、`division_by_zero`エラーとなり、`EXCEPTION`句がそのエラーを捕捉します。`RETURN`文による関数の戻り値は、1を加算した後の`x`の値となりますが、`UPDATE`コマンドによる結果はロールバックされます。しかし、前のブロックの`INSERT`コマンドはロールバックされません。したがって、データベースの内容の最終結果はTom Jonesであり、Joe Jonesではありません。

ティップ: `EXCEPTION`句を含んだブロックの実行に要する時間は、含まないブロックに比べてとても長くなります。したがって、必要のない時に`EXCEPTION`を使用してはいけません。

例 39-2. UPDATE/INSERTの例外

これは`UPDATE`または`INSERT`の実行における例外処理を使用した適当な例題です。

```
CREATE TABLE db (a INT PRIMARY KEY, b TEXT);

CREATE FUNCTION merge_db(key INT, data TEXT) RETURNS VOID AS
$$
BEGIN
    LOOP
        -- 最初にキーを更新する
        UPDATE db SET b = data WHERE a = key;
        IF found THEN
            RETURN;
        END IF;
        -- キーが存在しないので、キーの挿入を試行する
        -- 他者がすでに同一のキーを挿入していたならば
        -- 一意性に違反する欠陥となります
        BEGIN
            INSERT INTO db(a,b) VALUES (key, data);
            RETURN;
        EXCEPTION WHEN unique_violation THEN
            -- 何もしないで、更新を再試行します
            END;
    END LOOP;
END;
$$
LANGUAGE plpgsql;

SELECT merge_db(1, 'david');
SELECT merge_db(1, 'dennis');
```

このコーディングでは`unique_violation`エラーの原因が`INSERT`によるものであり、テーブルのトリガ関数内部の`INSERT`によるものでないと仮定します。また、テーブルに2つ以上の一意インデックスが存在した場合、どちらのインデックスがエラーの原因になろうと操作を再試行するので、誤作動となります。捕捉したエラーが予測したものであるか検証するために、次節で記述するエラー情報を利用すれば、より安全となります。

39.6.6.1. エラーに関する情報の取得

例外ハンドラはしばしば、起こった特定のエラーを識別する必要があります。現在の例外に関する情報を取得する方法は2つあります。特殊な変数と`GET STACKED DIAGNOSTICS`コマンドです。

例外ハンドラの内部では、特殊な変数`SQLSTATE`変数が起こった例外に対応したエラーコード（表A-1のエラーコード表を参照してください）を保有します。特殊な変数`SQLERRM`は例外に関連したエラーメッセージを保有します。これらの変数は、例外ハンドラの外部では定義されていません。

例外ハンドラの内部では、`GET STACKED DIAGNOSTICS` コマンドを使用して、現在の例外に関する情報を取り出すこともできます。次のようなやり方となります。

```
GET STACKED DIAGNOSTICS variable = item [ , ... ];
```

Each *item* is a key word identifying a status value to be assigned to the specified variable (which should be of the right data type to receive it). The currently available status items are shown in [表39-1](#). --> 各*item*は、指定された変数（これは受け取るために正しいデータ型でなければなりません）に代入される状態値を識別するキーワードです。現在使用可能なステータス項目は[表39-1](#)に表示されています。

表 39-1. エラーの診断値

名前	型	説明
RETURNED_SQLSTATE	text	例外のSQLSTATEエラーコード
MESSAGE_TEXT	text	例外の主要なメッセージのテキスト
PG_EXCEPTION_DETAIL	text	例外の詳細なメッセージのテキスト、存在する場合
PG_EXCEPTION_HINT	text	例外のヒントとなるメッセージのテキスト、存在する場合
PG_EXCEPTION_CONTEXT	text	呼び出しスタックを記述するテキストの行

例外が項目の値を設定しない場合、空文字列が返されます。

以下に例を示します。

```
DECLARE
    text_var1 text;
    text_var2 text;
    text_var3 text;
BEGIN

    -- 例外を引き起こす処理
    ...
EXCEPTION WHEN OTHERS THEN
    GET STACKED DIAGNOSTICS text_var1 = MESSAGE_TEXT,
                           text_var2 = PG_EXCEPTION_DETAIL,
                           text_var3 = PG_EXCEPTION_HINT;

END;
```

39.7. カーソル

問い合わせ全体を一度に実行するのではなく、カーソルを設定して、問い合わせをカプセル化し、問い合わせの結果を一度に数行ずつ読み取ることができます。これを行う理由の1つは、結果内に多数の行がある場合のメモリの枯渇を防ぐことです。（しかし、PL/pgSQLユーザは通常これを心配する必要はありません。FORループは自動的にカーソルを内部的に使用してメモリの問題を防ぐからです。）より興味深い使用方法として、呼び出し元が行を読み取ることができるように、作成されたカーソルへの参照を返す方法があります。これにより、関数から大量の行集合を返す際の効率が向上します。

39.7.1. カーソル変数の宣言

PL/pgSQLにおけるカーソルへのアクセスは全て、カーソル変数を経由します。カーソル変数は、常に特殊な`refcursor`データ型です。カーソル変数を作成する1つの方法は、単に`refcursor`型の変数として宣言することです。他の方法は、カーソル宣言構文を使用することです。以下にその一般形を示します。

```
name [ [ NO ] SCROLL ] CURSOR [ ( arguments ) ] FOR query;
```

（Oracleとの互換性のため、FORはISに置き換えることができます。）もしSCROLLを指定すれば、カーソルは逆方向に移動できます。もしNO SCROLLを指定すれば、逆方向の行の取り出しはできません。どちらも指定しない時、逆方向に取り出しできるかは問い合わせに依存します。もしargumentsがあれば、name datatypeをカンマで区切ったリストで、与えられた問い合わせ内のパラメータ値として置換される名前を定義します。その名前に実際に置換される値は、カーソルを開いた時点より後に指定されます。

以下に例を示します。

```
DECLARE
    curs1 refcursor;
    curs2 CURSOR FOR SELECT * FROM tenk1;
    curs3 CURSOR (key integer) FOR SELECT * FROM tenk1 WHERE unique1 = key;
```

これら3つの変数は全て`refcursor`データ型を持ちますが、最初のものは全ての問い合わせに使用でき、2番目には完全な問い合わせが既にバウンドされています（結び付けられています）。また、最後のものには、パラメータ付きの問い合わせがバウンドされています（keyはカーソルが開いた時に整数パラメータ値に置き換えられます）。curs1変数は、特定の問い合わせに結び付けられていませんので、アンバウンドであると呼ばれます。

39.7.2. カーソルを開く

カーソルを使用して行を取り出す前に、開かれる必要があります。（これはDECLARE CURSOR SQLコマンドの動作と同じです。）PL/pgSQLには3種類のOPEN文があり、そのうちの2つはアンバウンドカーソル変数を使用し、残りの1つはバウンドカーソル変数を使用します。

注意: バウンドカーソル変数は[項39.7.4](#)で説明されているFOR文で、明示的にカーソルを開かなくても使用することができます。

39.7.2.1. OPEN FOR query

```
OPEN unbound_cursorvar [ [ NO ] SCROLL ] FOR query;
```

カーソル変数は開かれ、実行するよう指定した問い合わせが付与されます。既に関いたカーソルを開くことはできず、また、アンバウンドカーソル変数として（つまり、単なる`refcursor`変数として）宣言されていなければなりません。この問い合わせはSELECT文であるか、または（EXPLAINのように）何らかの行を返すものでなければなりません。この問い合わせは、他のPL/pgSQLのSQL文と同様の方法で扱われます。PL/pgSQLの変数名は置き換えられ、問い合わせ計画は再利用できるようにキャッシュされます。PL/pgSQL変数がカーソルを使用する問い合わせに代入された時、変数はOPEN時の値となり、その後の変更はカーソルの動きに影響しません。SCROLLおよびNO SCROLLオプションの意味はバウンドカーソルと同様です。

以下に例を示します。

```
OPEN curs1 FOR SELECT * FROM foo WHERE key = mykey;
```

39.7.2.2. OPEN FOR EXECUTE

```
OPEN unbound_cursorvar [ [ NO ] SCROLL ] FOR EXECUTE query_string
[ USING expression [, ... ] ];
```

カーソル変数は開かれ、実行するよう指定した問い合わせが付与されます。既に関いたカーソルを開くことはできず、また、ア

ンバウンドカーソル変数として（つまり、単なる`refcursor`変数として）宣言されていなければなりません。問い合わせは、`EXECUTE` コマンドと同じ方法による文字列式として指定されます。通常と同様に、これにより、次回に実行する際に違った問い合わせを計画できる柔軟性が得られます([項39.10.2](#)参照)。また、変数置換がコマンド文字列上で行われないことも意味します。`EXECUTE`と同様に`USING`を介して動的コマンドにパラメータ値を挿入することができます。`SCROLL`および`NO SCROLL`オプションの意味はバウンドカーソルと同様です。

以下に例を示します。

```
OPEN curs1 FOR EXECUTE 'SELECT * FROM ' || quote_ident(tabname)
|| ' WHERE col1 = $1' USING keyvalue;
```

この例では、テーブル名は問い合わせ内にテキストとして埋め込まれます。このためSQLインジェクションに対抗するために`quote_ident()`の使用を勧めます。`col1`との比較値は`USING`経由で埋め込まれますので、引用符を付ける必要がありません。

39.7.2.3. バウンドカーソルを開く

```
OPEN bound_cursorvar [ ( [ argument_name := ] argument_value [, ...] ) ];
```

宣言時に問い合わせが結び付いたカーソル変数を開くために使用される`OPEN`の形式です。既に開いたカーソルを開くことはできません。実引数の式のリストはカーソルが引数を取るものと宣言された場合にのみ現れます。これらの値は問い合わせの中で置き換えられます。バウンドカーソルの問い合わせ計画は常にキャッシュ可能とみなされます。この場合、`EXECUTE`と等価なものはありません。`SCROLL`および`NO SCROLL`を`OPEN`において指定できないことに注意してください。カーソル移動の仕様はすでに決まっているからです。

位置的表記または記名的表記を使用して、引数の値を渡すことができます。位置的表記では、全ての引数が順番に指定されます。記名的表記では、引数の式と区別するために`:=`を使用して、各々の引数の名前が指定されます。[項4.3](#)に記述した関数呼び出しと同様に、位置的表記と記名的表記を混用できます。

例を示します（ここでは上例のカーソル宣言を使用します）。

```
OPEN curs2;
OPEN curs3(42);
OPEN curs3(key := 42);
```

変数の代入はバウンドカーソルの問い合わせで行われるため、カーソルへ値を渡す方法が2つあります。`OPEN`コマンドの明確な引数とするものと、問い合わせにおけるPL/pgSQL変数として暗黙的に参照するものです。しかし、バウンドカーソルの宣言より前に宣言した変数だけが代入されます。どちらの場合も、`OPEN`の実行時に変数値が決まります。例えば、上例の`curs3`と同じ結果を取得する方法を、以下に示します。

```
DECLARE
    key integer;
    curs4 CURSOR FOR SELECT * FROM tenk1 WHERE unique1 = key;
BEGIN
    key := 42;
    OPEN curs4;
```

39.7.3. カーソルの使用

カーソルを開いてから、ここで説明する文を使用してカーソルを扱うことができます。

これらの操作は、カーソルを開始するために開いた関数内で行う必要はありません。関数から`refcursor`値を返し、呼び出し元でそのカーソルの操作をさせることもできます。（内部的には`refcursor`値は、カーソルへの有効な問い合わせを持つポータルの名前を示す単なる文字列です。この名前は、ポータルを壊すことなく、他の`refcursor`型の変数に代入することで、他に渡すことができます。）

全てのポータルは、暗黙的にトランザクションの終わりで閉ざされます。したがって、`refcursor`値はそのトランザクションの終わりまでの間のみ開いたカーソルへの参照として有効です。

39.7.3.1. FETCH

```
FETCH [ direction { FROM | IN } ] cursor INTO target;
```

`FETCH`は`SELECT INTO`と同様に、カーソルから次の行を抽出し、対象に格納します。対象とは、行変数、レコード変数、または単純な変数をカンマで区切ったリストです。`SELECT INTO`の場合と同様、特殊な`FOUND`変数を検査することで、行が入手できたかどうかを確認することができます。

`direction`句は複数行を取り出すことができるコマンドを除き、SQL [FETCH](#)で許可されたどのようなコマンドも可能です。すなわち、以下のものです。NEXT、PRIOR、FIRST、LAST、ABSOLUTE count、RELATIVE count、FORWARDまたはBACKWARD。`direction`句の省略は、NEXTの指定と同じです。`SCROLL`オプションを用いてカーソルを宣言または開かないと、`direction`の値による逆方向への移動の要求は失敗します。

`cursor`名は、開いているカーソルのポータルを参照する`refcursor`変数名でなければなりません。

例：

```
FETCH curs1 INTO rowvar;
FETCH curs2 INTO foo, bar, baz;
FETCH LAST FROM curs3 INTO x, y;
FETCH RELATIVE -2 FROM curs4 INTO x;
```

39.7.3.2. MOVE

```
MOVE [ direction { FROM | IN } ] cursor;
```

`M MOVE` コマンドは、データを取り出さないでカーソルの位置を変更します。移動先の行を返さないでカーソルの位置だけを変更することを除けば、`FETCH` コマンドと同一の働きをします。`SELECT INTO`と同様に、特殊な変数`FOUND`を用いて、移動先に行が存在するかどうかを検査できます。

`direction`句は、[FETCH](#) SQLコマンドで許されている以下の値のいずれかを取ることができます。 `NEXT`、`PRIOR`、`FIRST`、`LAST`、`ABSOLUTE count`、`RELATIVE count`、`ALL`、`FORWARD [ count | ALL ]`、または`BACKWARD [ count | ALL ]` `direction`句の省略は、`NEXT`の指定と同じです。`SCROLL`オプションを用いてカーソルを宣言または開かないと、`direction`の値による逆方向への移動の要求は失敗します。

例：

```
M MOVE curs1;
MOVE LAST FROM curs3;
MOVE RELATIVE -2 FROM curs4;
MOVE FORWARD 2 FROM curs4;
```

39.7.3.3. UPDATE/DELETE WHERE CURRENT OF

```
UPDATE table SET ... WHERE CURRENT OF cursor;
DELETE FROM table WHERE CURRENT OF cursor;
```

カーソルの位置をテーブルの行に変更すれば、カーソルによって特定した行を更新または消去できます。カーソル問い合わせは何が許されているのか（特にグルーブ化しないとき）についての制限があり、それはカーソル内で`FOR UPDATE`を使用することが最善です。より詳細については[DECLARE](#)マニュアルページを参照下さい。

以下に例を示します。

```
UPDATE foo SET dataval = myval WHERE CURRENT OF curs1;
```

39.7.3.4. CLOSE

```
CLOSE cursor;
```

`CLOSE`はポータルの背後にあるカーソルを閉じます。これを使用してトランザクションの終わりよりも前にリソースを解放することができ、また、カーソル変数を解放し、再度開くことができます。

例：

```
CLOSE curs1;
```

39.7.3.5. カーソルを返す

PL/pgSQL関数では、呼び出し元にカーソルを返すことができます。この方法は、関数から複数行または複数列を返す場合、特にその結果集合が非常に大きい場合に有用です。これを行うには、関数はカーソルを開き、呼び出し元にカーソル名を返します（もしくは、もし呼び出し元でポータル名がわかっているならば、単純に指定されたポータル名を使用してカーソルを開きます）。これにより、呼び出し元はカーソルから行を取り出すことができるようになります。カーソルは呼び出し元で閉じることができます。または、トランザクションが終了した際に自動的に閉じられます。

カーソル用のポータル名は、プログラマが指定するか、または自動的に生成されます。ポータル名を指定するには、開く前に、単に`refcursor`変数に文字列を代入します。`refcursor`変数の文字列値は`OPEN`によって、背後のポータル名として使用されます。しかし、`refcursor`変数が`NULL`の場合、`OPEN`は自動的に既存のポータルと競合しない名前を生成し、それを`refcursor`変数に代入します。

注意：バウンドカーソル変数は、その名前を表現する文字列値で初期化されます。そのため、プログラマがカーソルを開く前に代入により上書きしない限り、ポータル名はカーソル変数と同じになります。しかし、アンバウンドカーソル変数の初期値はデフォルトで`NULL`です。そのため、上書きされていない場合に自動的に生成される一意な名前を受け取ります。

以下の例は、呼び出し元でカーソル名を指定する方法を示しています。

```
CREATE TABLE test (col text);
INSERT INTO test VALUES ('123');

CREATE FUNCTION reffunc(refcursor) RETURNS refcursor AS '
BEGIN
    OPEN $1 FOR SELECT col FROM test;
    RETURN $1;
END;
' LANGUAGE plpgsql;

BEGIN;
SELECT reffunc('funcursor');
FETCH ALL IN funcursor;
COMMIT;
```

以下の例では、自動的に生成されたカーソル名を使用しています。

```
CREATE FUNCTION reffunc2() RETURNS refcursor AS '
DECLARE
    ref refcursor;
BEGIN
    OPEN ref FOR SELECT col FROM test;
    RETURN ref;
END;
' LANGUAGE plpgsql;

-- カーソルを使用するには、トランザクション内部である必要があります。
BEGIN;
SELECT reffunc2();

        reffunc2
-----
<unnamed cursor 1>
(1 row)

FETCH ALL IN "<unnamed cursor 1>";
COMMIT;
```

以下の例は単一関数から複数のカーソルを返す方法を示しています。

```
CREATE FUNCTION myfunc(refcursor, refcursor) RETURNS SETOF refcursor AS $$
BEGIN
    OPEN $1 FOR SELECT * FROM table_1;
    RETURN NEXT $1;
    OPEN $2 FOR SELECT * FROM table_2;
    RETURN NEXT $2;
END;
$$ LANGUAGE plpgsql;

-- カーソルを使用するには、トランザクション内部である必要があります。
BEGIN;

SELECT * FROM myfunc('a', 'b');

FETCH ALL FROM a;
FETCH ALL FROM b;
COMMIT;
```

39.7.4. カーソル結果に対するループ

カーソルで返される行に対して反復することができるFOR文の亜種があります。構文は以下のようになります。

```
[ <<label>> ]
[ <<label>> ]
FOR recordvar IN bound_cursorvar [ ( [ argument_name := ] argument_value [, ...] ) ] LOOP
    statements
END LOOP [ label ];
```

カーソル変数は宣言されたとき、何らかの問い合わせとバウンドされていなければならない、また既に開かれていてはなりません。FOR文は自動的にカーソルを開き、ループから抜けたときに再度閉じます。実際の引数値式のリストは、カーソルが引数を取ることを宣言された場合に限ってのみ出現できます。これらの値は、OPEN過程と同じ方法で、問い合わせの中で置換されます（[項39.7.2.3](#)を参照してください）。

recordvar変数は、record型として自動的に定義され、ループ内でのみ存在します（存在するいかなる変数名の定義もループ内では無視されます）。カーソルによって返されたそれぞれの行はこのレコード変数に引き続いて割り当てられ、ループ本体が実行されます。

39.8. エラーとメッセージ

RAISE文を使用してメッセージを報告し、エラーを発生することができます。

```
RAISE [ level ] 'format' [, expression [, ... ]] [ USING option = expression [, ... ] ];
RAISE [ level ] condition_name [ USING option = expression [, ... ] ];
RAISE [ level ] SQLSTATE 'sqlstate' [ USING option = expression [, ... ] ];
RAISE [ level ] USING option = expression [, ... ] ;
RAISE ;
```

levelオプションはエラーの深刻度を指定します。使用可能なレベルはDEBUG、LOG、INFO、NOTICE、WARNINGおよびEXCEPTIONで、EXCEPTIONがデフォルトです。EXCEPTIONはエラーを発生させ、現在のトランザクションをアボートします。他のレベルは異なる優先度レベルのメッセージを生成するだけです。特定の優先度のエラーメッセージがクライアントに報告するか、サーバログに書き込むか、またはその両方は[log_min_messages](#)および[client_min_messages](#)設定変数によって制御されます。詳細については、[第18章](#)を参照してください。

もしあればlevelの後にformatを記述することができます（これは評価式ではなく、単純文字列リテラルでなければなりません）。書式文字列は報告されるエラーメッセージテキストを指定します。書式文字列内では、%は次の省略可能な引数の値の文字列表現で書き換えられます。%%と記述することで%リテラルを表すことができます。

以下の例では、v_job_idの値は文字列内の%を置き換えます。

```
RAISE NOTICE 'Calling cs_create_job(%)', v_job_id;
```

USINGに続いて、option = expression項目を記載することで、エラー報告に追加の情報を加えることができます。使用可能なoptionキーワードはMESSAGE、DETAIL、HINT、およびERRCODEで、それぞれのexpressionは任意の文字列で値付けされる式を取ることができます。MESSAGEはエラーメッセージテキストを設定します（このオプションはUSINGの前に書式文字列を含むRAISE形式では使用できません）。DETAILはエラー詳細メッセージを、一方HINTはヒントを出力します。ERRCODEは、[付録A](#)で示されている状況名、または直接的に5文字によるSQLSTATEコードのいずれかの報告すべきエラーコード（SQLSTATE）を指定します。

以下の例は、与えられたエラーメッセージとヒントを付けてトランザクションをアボートします。

```
RAISE EXCEPTION 'Nonexistent ID --> %', user_id
USING HINT = 'Please check your user ID';
```

以下の2つの例は、SQLSTATEを設定する等価な方法を示しています。

```
RAISE 'Duplicate user ID: %', user_id USING ERRCODE = 'unique violation';
RAISE 'Duplicate user ID: %', user_id USING ERRCODE = '23505';
```

主引数が報告されるべき状況名、またはSQLSTATEである場合、2番目のRAISE構文があります。例を示します。

```
RAISE division_by_zero;
RAISE SQLSTATE '22012';
```

この構文において、USINGは独自のエラーメッセージ、詳細、またはヒントを供給するように使用できます。先の例と同じことを行う別の方法は次のようになります。

```
RAISE unique_violation USING MESSAGE = 'Duplicate user ID: ' || user_id;
```

他にも亜種があり、RAISE USING または RAISE level USING と記述して、全て一括してUSINGリスト内に書き加えます。

最後のRAISE亜種はパラメータを全く取りません。この形式はBEGINブロックのEXCEPTION句で使用されるのみです。これは、現在処理中のエラーを再発生させます。

注意: PostgreSQL 9.1より前のバージョンでは、パラメータのないRAISEは稼働している例外ハンドラを含むブロックからのエラーの再発生と解釈されました。したがって、例外ハンドラの中で入れ子となったEXCEPTION句は、RAISEが入れ子となったEXCEPTION句のブロック内にあるときでも、エラーを捕捉できないことになりました。これは驚くべきことであり、オラクルのPL/SQLと非互換でした。

RAISE EXCEPTIONコマンド内で状況名もSQLSTATEも指定されない場合、デフォルトはRAISE_EXCEPTION (P0001)を使用します。メッセージテキストが指定されない場合、デフォルトは状況名、またはSQLSTATEをメッセージテキストとして使用します。

注意: SQLSTATEコードでエラーコードを指定する場合、事前に定義されたエラーコードに制約されることはありません。00000以外の5桁の数字かASCIIの大文字からなるどんなエラーコードも選択できます。3つのゼロで終わるエラーコードの出力を避けるように推奨されています。言うのは、そこには分類コードがあり、それらは全ての分類から捕捉することによってのみ補足可能だからです。

39.9. トリガプロシージャ

PL/pgSQLはトリガプロシージャの定義に使用できます。トリガプロシージャは、`CREATE FUNCTION`コマンドを使って、`trigger`という戻り値の型を持った引数のない関数として作成されます。その関数は、たとえ、`CREATE TRIGGER`にて引数を取るものとしていたとしても、引数を持たないものと宣言しなければなりません。トリガの引数は、後述する通り、`TG_ARGV`経由で渡されます。

PL/pgSQL関数がトリガとして呼び出された場合、いくつかの特殊な変数が自動的に最上位レベルのブロックで作成されます。それらを以下に示します。

`NEW`

`RECORD` データ型。この変数は行レベルのトリガでの`INSERT/UPDATE`操作によって更新された、新しいデータベースの行を保持します。この変数は文レベルのトリガ、および`DELETE`操作では`NULL`です。

`OLD`

`RECORD` データ型。この変数は、行レベルのトリガでの`UPDATE/DELETE`操作によって更新される前のデータベースの行を保持します。この変数は文レベルのトリガ、および`INSERT`操作では`NULL`です。

`TG_NAME`

`name` データ型。実際に発行されたトリガの名前を持つ変数。

`TG_WHEN`

`text` データ型。トリガの定義に依存した`BEFORE`、`AFTER`、または`INSTEAD OF`という文字列。

`TG_LEVEL`

`text` データ型。トリガの定義に依存した`ROW` または `STATEMENT` という文字列。

`TG_OP`

`text` データ型。トリガを起動した操作を示す、`INSERT`、`UPDATE`、`DELETE`、または`TRUNCATE`という文字列。

`TG_RELID`

`oid` データ型。このトリガの呼び出し元になるテーブルのオブジェクトID。

`TG_RELNAME`

`name` データ型。このトリガの呼び出し元になるテーブルの名前。将来これは廃止されそうです。代わりに`TG_TABLE_NAME`を使用してください。

`TG_TABLE_NAME`

`name` データ型。このトリガの呼び出し元になるテーブルの名前。

`TG_TABLE_SCHEMA`

`name` データ型。このトリガの呼び出し元になるテーブルのスキーマ名。

`TG_NARGS`

`integer` 型。 `CREATE TRIGGER` 文におけるトリガプロシージャに与えられる引数の数。

`TG_ARGV[]`

`text` 型の配列型。 `CREATE TRIGGER` 文での引数。このインデックスは0から始まります。無効なインデックス（0未満や`tg_nargs`以上）は、`NULL`値という結果になります。

トリガ関数は`NULL`または、トリガの発行元になったテーブルの構造を正確に持ったレコード/行を返さなければなりません。

`BEFORE`として発行された行レベルトリガが`NULL`を返す場合には、トリガマネージャに実際の行への操作を取りやめるように通知します（つまり、その後にトリガが発行されず、その`INSERT/UPDATE/DELETE`はその行に対して実行されません）。非`NULL`値を返す場合には、その操作はその行値で処理されます。元の`NEW`の値と異なる行値を返すことは、挿入、更新される値を変更します。したがってトリガ関数が行値を変更せずにトリガ処理を普通に成功させたい場合は、`NEW`(またはその等価な値)を返さなければなりません。格納する行を変更するために、`NEW`の個々の値を直接置き換え、変更した`NEW`を返すことも、新しいレコード/行を完全に作成して返すことも可能です。`DELETE`に対する`BEFORE`トリガの場合、返される値は直接的な影響を与えません

`DELETE`

`NEW`

が、トリガ動作を継続させるためには非NULLを返さなければなりません。トリガでは がNULLであり、これを返すことは通常無意味であることに注意して下さい。DELETEトリガにおける通常の慣例はOLDを返すことです。

INSTEAD OFトリガ（これは常に行レベルトリガであり、ビューに対してのみ使用可能です）は、まったく更新を行わなかったためにこの行に対する残りの操作を飛ばさなければならない（つまり後続のトリガは発行されず、トリガの発生元のINSERT/UPDATE/DELETEにおいて影響を受けた行数として数えられない）ことを通知するNULLを返すことができます。この他の場合、トリガが要求された操作を実行したことを通知するために、非NULLの値を返さなければなりません。INSERTおよびUPDATE操作では、戻り値は、トリガ関数がINSERT RETURNINGおよびUPDATE RETURNINGをサポートするために変更しているかもしれない、NEWとなるはずですが（これは後続のトリガに渡される行値にも影響します）。DELETE操作では、戻り値はOLDとなるはずですが。

行レベルのAFTERトリガ、文レベルのBEFOREまたはAFTERトリガの戻り値は常に無視されます。NULLとしても構いません。しかし、これらの種類のトリガでも、エラーを発生させることで操作全体を中断させることが可能です。

[例39-3](#)にPL/pgSQLのトリガプロシージャの例を示します。

例 39-3. PL/pgSQLトリガプロシージャ

このトリガの例では、テーブルの行が挿入または更新された時には必ず、現在のユーザ名と時刻がその行に入っていることを確実にします。そして、従業員名が与えられていることとその給料が正の値であることを確認します。

```
CREATE TABLE emp (
    empname text,
    salary integer,
    last_date timestamp,
    last_user text
);

CREATE FUNCTION emp_stamp() RETURNS trigger AS $emp_stamp$
BEGIN
    -- empnameとsalaryが与えられていることをチェック
    IF NEW.empname IS NULL THEN
        RAISE EXCEPTION 'empname cannot be null';
    END IF;
    IF NEW.salary IS NULL THEN
        RAISE EXCEPTION '% cannot have null salary', NEW.empname;
    END IF;

    -- 支払時に問題が起こらないように
    IF NEW.salary < 0 THEN
        RAISE EXCEPTION '% cannot have a negative salary', NEW.empname;
    END IF;

    -- 誰がいつ変更したかを記録
    NEW.last_date := current_timestamp;
    NEW.last_user := current_user;
    RETURN NEW;
END;
$emp_stamp$ LANGUAGE plpgsql;

CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
FOR EACH ROW EXECUTE PROCEDURE emp_stamp();
```

テーブルにおける変更のログを取る他の方法は、挿入、更新または削除の各々に対する行を保有する新テーブルを作成することです。この方法はテーブルにおける変更の監査と考えることができます。

[例39-4](#)はPL/pgSQLによる監査用トリガプロシージャの一例を示します。

例 39-4. PL/pgSQLによる監査用のトリガプロシージャ

このトリガの例では、empテーブルにおける行の挿入、更新または削除のどれもがemp_auditテーブルの中へ確実に記録（すなわち監査）されます。現在時刻とユーザ名は、行った操作の種類とともにemp_auditの行の中に記録されます。

```
CREATE TABLE emp (
    empname text NOT NULL,
    salary integer
);

CREATE TABLE emp_audit(
    operation char(1) NOT NULL,
    stamp timestamp NOT NULL,
    userid text NOT NULL,
    empname text NOT NULL,
    salary integer
);
```



```

CREATE OR REPLACE FUNCTION process_emp_audit() RETURNS TRIGGER AS $emp_audit$
BEGIN
--
-- empで行った操作を反映する行をemp_auditに作成
-- 操作の種類を決定するために、特殊な変数TG_OPを活用
--
IF (TG_OP = 'DELETE') THEN
    INSERT INTO emp_audit SELECT 'D', now(), user, OLD.*;
    RETURN OLD;
ELSIF (TG_OP = 'UPDATE') THEN
    INSERT INTO emp_audit SELECT 'U', now(), user, NEW.*;
    RETURN NEW;
ELSIF (TG_OP = 'INSERT') THEN
    INSERT INTO emp_audit SELECT 'I', now(), user, NEW.*;
    RETURN NEW;
END IF;
RETURN NULL; -- AFTERトリガですので、結果は無視されます
END;
$emp_audit$ LANGUAGE plpgsql;

CREATE TRIGGER emp_audit
AFTER INSERT OR UPDATE OR DELETE ON emp
FOR EACH ROW EXECUTE PROCEDURE process_emp_audit();

```

前例の変形では、各エントリが最終修正された時を表示するため、主テーブルを監査テーブルに結合したビューを使用します。この方法でもテーブルの変化の監査証跡を全て記録できますが、監査証跡から抽出した各エントリの最終修正のタイムスタンプ表示することにより、監査証跡の簡単なビューを表示することにもなります。例39-5で示すものは、PL/pgSQLを用いたビューの監査トリガの例です。

例 39-5. 監査のためのPL/pgSQLビュートリガプロシージャ

この例では、ビューを更新可能とし、その行の挿入、更新、削除をemp_auditテーブルに確実に記録（つまり監査）するためにビューに対するトリガを使用します。現在時刻とユーザ名が実行された操作種類と一緒に記録されます。ビューは各行の最終更新時間を表示します。

```

CREATE TABLE emp (
    empname      text PRIMARY KEY,
    salary       integer
);

CREATE TABLE emp_audit(
    operation    char(1) NOT NULL,
    userid       text NOT NULL,
    empname      text NOT NULL,
    salary       integer,
    stamp        timestamp NOT NULL
);

CREATE VIEW emp_view AS
    SELECT e.empname,
           e.salary,
           max(ea.stamp) AS last_updated
    FROM emp e
    LEFT JOIN emp_audit ea ON ea.empname = e.empname
    GROUP BY 1, 2;

CREATE OR REPLACE FUNCTION update_emp_view() RETURNS TRIGGER AS $$
BEGIN
--
-- 要求された操作を emp に実行し
-- emp_audit に行を作成し
-- emp の変化を反映する
--
IF (TG_OP = 'DELETE') THEN
    DELETE FROM emp WHERE empname = OLD.empname;
    IF NOT FOUND THEN RETURN NULL; END IF;

    OLD.last_updated = now();
    INSERT INTO emp_audit VALUES('D', user, OLD.*);
    RETURN OLD;
ELSIF (TG_OP = 'UPDATE') THEN
    UPDATE emp SET salary = NEW.salary WHERE empname = OLD.empname;
    IF NOT FOUND THEN RETURN NULL; END IF;

    NEW.last_updated = now();
    INSERT INTO emp_audit VALUES('U', user, NEW.*);
    RETURN NEW;
ELSIF (TG_OP = 'INSERT') THEN
    INSERT INTO emp VALUES(NEW.empname, NEW.salary);

    NEW.last_updated = now();
    INSERT INTO emp_audit VALUES('I', user, NEW.*);

```

```

        RETURN NEW;
    END IF;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER emp_audit
INSTEAD OF INSERT OR UPDATE OR DELETE ON emp_view
FOR EACH ROW EXECUTE PROCEDURE update_emp_view();

```

トリガの使用目的の1つは、あるテーブルのサマリテーブルを維持することです。結果のサマリテーブルは、元のテーブルに代わって、ある種の問い合わせに対して使用でき、しばしば実行時間を大幅に縮小します。通常この手法は、計測または観測データ（ファクトテーブルと言います）を非常に大きくなるかもしれない、データウェアハウスに使用されます。データウェアハウス内のファクトテーブルに対してサマリテーブルを維持するPL/pgSQLのトリガプロシージャの例を[例39-6](#)に示します。

例 39-6. サマリテーブルを維持するためのPL/pgSQLトリガプロシージャ

ここに述べるスキーマの一部はRalph Kimballによる*The Data Warehouse Toolkit*の*Grocery Store*の例に基づいています。

```

--
-- time dimensionとsales factの主テーブル
--
CREATE TABLE time_dimension (
    time_key          integer NOT NULL,
    day_of_week       integer NOT NULL,
    day_of_month       integer NOT NULL,
    month              integer NOT NULL,
    quarter            integer NOT NULL,
    year              integer NOT NULL
);
CREATE UNIQUE INDEX time_dimension_key ON time_dimension(time_key);

CREATE TABLE sales_fact (
    time_key          integer NOT NULL,
    product_key       integer NOT NULL,
    store_key          integer NOT NULL,
    amount_sold        numeric(12,2) NOT NULL,
    units_sold         integer NOT NULL,
    amount_cost        numeric(12,2) NOT NULL
);
CREATE INDEX sales_fact_time ON sales_fact(time_key);

--
-- sales by timeのサマリテーブル
--
CREATE TABLE sales_summary_bytime (
    time_key          integer NOT NULL,
    amount_sold        numeric(15,2) NOT NULL,
    units_sold         numeric(12) NOT NULL,
    amount_cost        numeric(15,2) NOT NULL
);
CREATE UNIQUE INDEX sales_summary_bytime_key ON sales_summary_bytime(time_key);

--
-- 更新、挿入および削除によりサマリテーブルの列を修正する関数とトリガ
--
CREATE OR REPLACE FUNCTION maint_sales_summary_bytime() RETURNS TRIGGER
AS $maint_sales_summary_bytime$
    DECLARE
        delta_time_key          integer;
        delta_amount_sold       numeric(15,2);
        delta_units_sold        numeric(12);
        delta_amount_cost       numeric(15,2);
    BEGIN
        -- 増加または減少量を算出
        IF (TG_OP = 'DELETE') THEN

            delta_time_key = OLD.time_key;
            delta_amount_sold = -1 * OLD.amount_sold;
            delta_units_sold = -1 * OLD.units_sold;
            delta_amount_cost = -1 * OLD.amount_cost;

        ELSIF (TG_OP = 'UPDATE') THEN

            -- time keyを変更する更新を禁止します
            -- （削除 + 挿入の方法により大部分の変更を行うため
            --   それほど厄介ではありません）。
            IF ( OLD.time_key != NEW.time_key) THEN
                RAISE EXCEPTION 'Update of time_key : % -> % not allowed',
                                OLD.time_key, NEW.time_key;
            END IF;

```

```

        delta_time_key = OLD.time_key;
        delta_amount_sold = NEW.amount_sold - OLD.amount_sold;
        delta_units_sold = NEW.units_sold - OLD.units_sold;
        delta_amount_cost = NEW.amount_cost - OLD.amount_cost;

ELSIF (TG_OP = 'INSERT') THEN

        delta_time_key = NEW.time_key;
        delta_amount_sold = NEW.amount_sold;
        delta_units_sold = NEW.units_sold;
        delta_amount_cost = NEW.amount_cost;

END IF;

-- サマリテーブルの行を挿入または新しい値で更新します。
<<insert_update>>
LOOP
    UPDATE sales_summary_bytime
        SET amount_sold = amount_sold + delta_amount_sold,
            units_sold = units_sold + delta_units_sold,
            amount_cost = amount_cost + delta_amount_cost
        WHERE time_key = delta_time_key;

    EXIT insert_update WHEN found;

BEGIN
    INSERT INTO sales_summary_bytime (
        time_key,
        amount_sold,
        units_sold,
        amount_cost)
        VALUES (
            delta_time_key,
            delta_amount_sold,
            delta_units_sold,
            delta_amount_cost
        );

    EXIT insert_update;

EXCEPTION

    WHEN UNIQUE VIOLATION THEN
        -- 何もいません
    END;
END LOOP insert_update;

RETURN NULL;

END;
$maint_sales_summary_bytime$ LANGUAGE plpgsql;

CREATE TRIGGER maint_sales_summary_bytime
AFTER INSERT OR UPDATE OR DELETE ON sales_fact
FOR EACH ROW EXECUTE PROCEDURE maint_sales_summary_bytime();

INSERT INTO sales_fact VALUES(1,1,1,10,3,15);
INSERT INTO sales_fact VALUES(1,2,1,20,5,35);
INSERT INTO sales_fact VALUES(2,2,1,40,15,135);
INSERT INTO sales_fact VALUES(2,3,1,10,1,13);
SELECT * FROM sales_summary_bytime;
DELETE FROM sales_fact WHERE product_key = 1;
SELECT * FROM sales_summary_bytime;
UPDATE sales_fact SET units_sold = units_sold * 2;
SELECT * FROM sales_summary_bytime;

```

39.10. PL/pgSQLの秘訣

本節では、PL/pgSQL利用者の知識として重要な、実装の詳細を述べます。

39.10.1. 変数置換

PL/pgSQL関数内のSQL文および式は変数および関数のパラメータを参照することができます。背後では、PL/pgSQLはこうした参照を問い合わせパラメータに置き換えます。パラメータまたは列参照が文法的に許されているところでのみパラメータは置換されます。極端な場合として、以下のよろしくないプログラミングスタイルの例を考えてみましょう。

```
INSERT INTO foo (foo) VALUES (foo);
```

最初に現れる`foo`の場所は文法的にはテーブル名でなければなりません。このため関数が`foo`という名前の変数を持っていたとしても、置換されません。2番目の場所はテーブルの列名でなければなりません。このためこれも置換されません。3番目の場所のみが関数の変数参照の候補です。

注意: 9.0より前のPostgreSQLでは、3つの場合すべてにおいて変数を置換しようとし、構文エラーを引き起こしました。

変数名は文法的にはテーブル列名と違いがありませんので、テーブルを参照する文の中であいまいさが出る可能性があります。与えられた名前はテーブル列を意味するのでしょうか、それとも変数なのでしょうか。前の例を次のように変えてみましょう。

```
INSERT INTO dest (col) SELECT foo + bar FROM src;
```

ここでは、`dest`および`src`はテーブル名でなければなりません。また、`col`は`dest`の列でなければなりません。また、`foo`および`bar`は理論上関数の変数かもしれませんが、`src`の列かもしれません。

デフォルトでPL/pgSQLはSQL文における名前が変数かテーブル列のいずれかを参照可能な場合にエラーを報告します。変数または列の名前を変更することやあいまいな参照を修飾すること、PL/pgSQLにどちらを優先して解釈するかを通知することで、こうした問題を解消することができます。

最も簡単な解法は変数名または列名を変更することです。一般的なコーディング法として、列の命名とPL/pgSQL変数の命名とで規約を分ける方法があります。例えば、一貫して関数の変数は`v_something`という名前とし、列名は`v_`で始まらないようにすれば、競合は起こりません。

その他、あいまいな参照を明確にするために修飾することができます。上の例では、`src.foo`によりテーブル列への参照についてあいまいさが解消します。あいまい性のない変数参照を行うためには、ラベル付けしたブロック内で変数を宣言し、そのブロックのラベルを使用します([項39.2](#)参照)。以下に例を示します。

```
<<block>>
DECLARE
    foo int;
BEGIN
    foo := ...;
    INSERT INTO dest (col) SELECT block.foo + bar FROM src;
```

ここで`block.foo`は`src`に`foo`列があったとしても、変数を意味することになります。`FOUND`などの特別な変数を含め、関数パラメータを関数名で修飾することができます。これらは暗黙的に関数名をラベル名とした上位ブロック内で宣言されているためです。

PL/pgSQLの大規模な本体コードにおける、すべてのあいまいな参照を修正することが現実的ではない場合があります。こうした場合、PL/pgSQLにあいまいな参照を変数として解決すべき(この動作はPostgreSQL 9.0より前のPL/pgSQLの動作と互換性を持ちます)、または、テーブル列参照として解決すべき(Oracleなどの他のシステムと互換性を持ちます)と指定することができます。

システム全体に対してこの動作を変更するためには`plpgsql.variable_conflict`設定パラメータを`error`、`use_variable`、`use_column`のいずれかに設定します(`error`が標準配布におけるデフォルトです)。このパラメータは以降のPL/pgSQL関数の文のコンパイルに影響しますが、現在のセッションでコンパイル済みの文には影響を与えません。この設定を変更することで、PL/pgSQLの動作において予期できない変化が発生することがありますので、これはスーパーユーザのみが変更することができます。

また、関数テキストの先頭に以下の特殊なコマンドの1つをいれることで、関数単位で動作を設定することもできます。

```
#variable_conflict error
#variable_conflict use_variable
#variable_conflict use_column
```

これらのコマンドを記述した関数に対してのみ、コマンドは影響を与え、`plpgsql.variable_conflict`の設定を上書きします。以下に例を示します。

```
CREATE FUNCTION stamp_user(id int, comment text) RETURNS void AS $$
#variable_conflict use_variable
DECLARE
    curtime timestamp := now();
BEGIN
    UPDATE users SET last_modified = curtime, comment = comment
    WHERE users.id = id;
END;
$$ LANGUAGE plpgsql;
```

UPDATEコマンドにおいて、`curtime`、`comment`および`id`は、`users`に同名の列があるか否かに関わらず、関数の変数またはパラメータを参照します。テーブル列を参照させるためにWHERE句において`users.id`と参照を修飾する必要があったことに注意して下さい。しかしUPDATEリストの対象としての`comment`への参照は修飾させる必要がありませんでした。これは文法的に`users`の列でなければならないためです。以下のように`variable_conflict`の設定に依存せずに同じ関数を作成することもできます。

```
CREATE FUNCTION stamp_user(id int, comment text) RETURNS void AS $$
<<fn>>
DECLARE
    curtime timestamp := now();
BEGIN
    UPDATE users SET last_modified = fn.curtime, comment = stamp_user.comment
    WHERE users.id = stamp_user.id;
END;
$$ LANGUAGE plpgsql;
```

変数置換はEXECUTEコマンドまたはその亜種におけるコマンド文字列の中では起こりません。そのようなコマンドに可変値を挿入する時は、[項39.5.4](#)に述べたように、文字列の値を構成するものの一部とするかUSINGを使用してください。

今のところ変数置換は、SELECTとINSERTとUPDATEとDELETEコマンドの中だけで作動します。メインSQLエンジンが問い合わせパラメータをこれらのコマンドでしか許可しないからです。他の種類の文（通常ユーティリティ文といいます）において可変名または可変値を使用するには、文字列としてユーティリティ文を構成しEXECUTEしてください。

39.10.2. 計画のキャッシュ

PL/pgSQLインタプリタは、初めてその関数が（各セッションで）呼び出された時に、関数のソーステキストを解析し、バイナリ形式の命令ツリーを内部で作成します。この命令ツリーは完全にPL/pgSQL文構造に変換されますが、関数内部の個々のSQL式とSQLコマンドは即座に変換されません。

各式やSQLコマンドが初めてその関数で実行される時に、PL/pgSQLインタプリタは（SPIマネージャのSPI_prepare関数を使用して、プリペアドステートメントを作成するためにコマンドを解析します。その後その式やコマンドが行われる時には、そのプリペアドステートメントを再利用します。こうして、めったに分岐されない条件付きコードパスを持つ関数では、現在のセッションで実行されないそれらのコマンドの解析によるオーバーヘッドを背負いこむことはありません。欠点は特定の式や問い合わせのエラーが、関数の該当部分が実行されるまで検出されないことです。（典型的な構文エラーは、最初の解釈において検出されますが、それより深いエラーは、実行の時まで検出されません）。

PL/pgSQLは（正確にはSPIマネージャは）さらに特定のプリペアドステートメントに関する実行計画のキャッシュを試行できます。キャッシュした実行計画が使用されなかった場合、プリペアドステートメントが呼び出される度に新しい実行計画が作成され、選択した実行計画を最適にするために、最新のパラメータ値（すなわちPL/pgSQLの変数値）が使用されます。プリペアドステートメントがパラメータを持たないか何回も使用される場合、SPIマネージャは特定のパラメータ値に依存しない一般的な実行計画の作成を考え、再使用のためにキャッシュします。典型的には、これは参照したPL/pgSQLの変数値が、実行計画にさほど影響しない場合にだけ起こります。それならば、毎回の実行計画の作成の方が優れています。

このようにPL/pgSQLはプリペアドステートメントおよび時には実行計画を保存しますので、PL/pgSQL関数内に直接現れるSQLコマンドは実行の度に同じテーブルとフィールドを参照しなければなりません。つまり、SQLコマンドにて、テーブルやフィールドの名前としてパラメータを使用することができません。実行の度に新しく実行計画を作成する無駄を覚悟で、PL/pgSQLのEXECUTE文を使った動的問い合わせを構成することで、この制限を回避できます。

レコード変数の変わりやすいという性質はこの接続において別の問題となります。レコード変数のフィールドが式や文の中で使用される場合、そのフィールドのデータ型を関数を呼び出す度に更新してはいけません。それぞれの式が最初に実行された時のデータ型を使用して、その式が解析されているからです。必要な場合EXECUTEを使用してこの問題を回避することができます。

同一の関数が2つ以上のテーブルのトリガとして使用される場合、PL/pgSQLはテーブルごとのプリペアドステートメントをキャッシュします。すなわち、各々のトリガ関数とテーブルの組ごとにキャッシュするのであり、トリガ関数ごとではありません。このため、データ型の変更に伴う問題の一部を軽減します。例えば、別のテーブルにある異なったデータ型であっても、`key`と命名した列に対してトリガ関数は有効に作動します。

同様に、多様型の引数を持った関数は、実際に呼び出す引数の型の組み合わせごとに別々のプリペアドステートメントをキャッシュします。そのため、データ型の差異が原因で予期しない失敗が起こることはありません。

プリペアドステートメントのキャッシュにより、時間に依存する値の解釈の結果に違いが現れることがあります。例えば、以下の2つの関数の結果は異なります。

```
CREATE FUNCTION logfunc1(logtxt text) RETURNS void AS $$
BEGIN
    INSERT INTO logtable VALUES (logtxt, 'now');
END;
$$ LANGUAGE plpgsql;
```

および

```
CREATE FUNCTION logfunc2(logtxt text) RETURNS void AS $$
DECLARE
    curtime timestamp;
BEGIN
    curtime := 'now';
    INSERT INTO logtable VALUES (logtxt, curtime);
END;
$$ LANGUAGE plpgsql;
```

logfunc1の場合では、PostgreSQLのメインパーサは、INSERTを解析する時に、logtableの対象列の型から'now'をtimestampと解釈しなければならないことを把握しています。こうして、パーサはINSERTが解析された時点で'now'をtimestamp定数に変換し、その定数値をその後のセッションの有効期間におけるlogfunc1の全ての呼び出しで使用します。言うまでもありませんが、これはプログラマが意図した動作ではありません。now()またはcurrent_timestamp関数の使用が優れています。

logfunc2の場合では、PostgreSQLのメインパーサは'now'の型を決定することができません。そのため、nowという文字列を持つtext型のデータ値を返します。curtimeローカル変数に代入する時に、PL/pgSQLインタプリタはこの文字列をtext_outとtimestamp_in関数を変換に使用してtimestamp型にキャストします。ですから、演算されたタイムスタンプは、プログラマが意図した通り、実行の度に更新されます。この方法でたまたま意図した通り動くけれど、それほど効率的ではありません。ですから、now()関数の使用の方が優れています。

[前のページ](#)

トリガプロシージャ

[ホーム](#)
[上に戻る](#)

[次のページ](#)

PL/pgSQLによる開発向けのヒント

39.11. PL/pgSQLによる開発向けのヒント

PL/pgSQLで開発する1つの良い方法は、関数を作成するのに自分の好きなテキストエディタを使い、もう1つのウィンドウでpsqlを使用して関数を読み込ませて試験を行うことです。この方法で行う場合にはCREATE OR REPLACE FUNCTIONを使用して関数を作成する方が良いでしょう。この方法でファイルを再読み込みすると、関数定義を更新することができます。例えば以下のようにします。

```
CREATE OR REPLACE FUNCTION testfunc(integer) RETURNS integer AS $$
    ....
$$ LANGUAGE plpgsql;
```

psqlを実行し、以下のように関数定義ファイルを読み込み、または再読み込みすることができます。

```
¥i filename.sql
```

その後すぐに、関数を試験するためにSQLコマンドを発行することができます。

PL/pgSQLにおける開発のもう1つの良い方法は、手続き言語の開発機能を持つGUIデータベースアクセスツールを使用することです。他にもありますが、pgAdminがこうしたツールの一例です。こうしたツールは、単一引用符をエスケープさせたり、関数の作り直しやデバッグが容易に行えたりする便利な機能をよく持っています。

39.11.1. 引用符の扱い

PL/pgSQL関数のコードはCREATE FUNCTION内で文字列リテラルとして指定されます。単一引用符で囲む通常のやり方で文字列リテラルを記述する時、関数本体内部の全ての単一引用符を二重化しなければなりません。同様に、全てのバックスラッシュを二重化しなければなりません。なお、文字列としてエスケープする構文が使用されると仮定します。引用符を単に重ねるやり方は最も冗長であり、簡単に想像できると思いますが、複雑な状態では数個以上の隣接した引用符が必要となるため、コードを率直には理解しにくくなります。それに代わって推奨されるのは、関数本体を"ドル引用符"の文字列リテラルとして記述することです(項4.1.2.4を見てください)。ドル引用符を用いるやり方では他の引用符を二重化する必要はありませんが、それぞれの入れ子になった階層ごとに異なったドル引用符による区切り符号を用いなければなりません。例えば、CREATE FUNCTIONコマンドを以下のように記述しても構いません。

```
CREATE OR REPLACE FUNCTION testfunc(integer) RETURNS integer AS $PROC$
    ....
$PROC$ LANGUAGE plpgsql;
```

このやり方では、SQLコマンドの中で単純なリテラル文字列に対して引用符を使用でき、文字列として集積したSQLコマンドの断片を区切るために\$\$を使用できます。もし\$\$を含んだテキストを引用符で囲む時は、\$Q\$のような記述を使用できます。

以下の表はドル引用符を用いない時の引用符の記述法を示したものです。ドル引用符を用いる以前における引用符の記述を理解するのに、この表は役立つと思われます。

1つの引用符

関数本体の先頭と末尾。以下に例を示します。

```
CREATE FUNCTION foo() RETURNS integer AS '
    ....
' LANGUAGE plpgsql;
```

関数本体内部では引用符は必ずペアで現れます。

2つの引用符

関数本体内部の文字列リテラル用。以下に例を示します。

```
a_output := 'Blah';
SELECT * FROM users WHERE f_name='foobar';
```

ドル引用符を用いる時は、次のように記述します。

```
a_output := 'Blah';
SELECT * FROM users WHERE f_name='foobar';
```

どちらもPL/pgSQLパーサから見ると同一となります。

4つの引用符

39.12. Oracle PL/SQLからの移植

本節ではOracle®からPostgreSQLへアプリケーションを移植する開発者の手助けとなるように、PostgreSQLのPL/pgSQL言語とOracleのPL/SQL言語の違いについて説明します。

PL/pgSQLは多くの点でPL/SQLに似ています。それはブロックで構成されていて、厳格な言語であり、全ての変数は宣言されなければならない点です。代入やループ、条件分岐も同様です。PL/SQLからPL/pgSQLに移植する際に注意しなければならない、主な違いを以下に示します。

- SQLコマンド内に使用された名前が、テーブルの列名または関数の変数への参照のどちらにもなり得る場合、PL/SQLは列名として処理します。これはPL/pgSQLにおける`plpgsql.variable_conflict = use_column`時の動作に対応しますが、[項39.10.1](#)の説明通り、これはデフォルトではありません。初期段階において、そのようなあいまいさを避けることが最善です。しかしこの動作に依存するコードの量が多いものを移植しなければならない場合、`variable_conflict`を使用することが最善の解法かもしれません。
- PostgreSQLの関数本体は文字列リテラルとして書かなければなりません。したがって、関数本体内部でドル引用符を使用するか、単一引用符をエスケープする必要があります。[\(項39.11.1を参照してください\)](#)。
- パッケージの代わりに、スキーマを使用して関数群をグループにまとめてください。
- パッケージがないため、パッケージレベルの変数也没有ありません。これは幾分か厄介なことです。代わって、セッションごとの状態を一時テーブル内部に保存できます。
- `REVERSE`を付けた整数`FOR`ループの処理は異なります。PL/SQLでは最後の数から最初の数へ減少しながら処理しますが、PL/pgSQLでは最初の数から最後の数へ減少しながら処理します。移植において、ループの両端となる最初の数と最後の数を交換する必要があります。この非互換性は不幸なことです、変わりそうもありません。[\(項39.6.3.5を見てください\)](#)。
- 問い合わせ上の`FOR`ループは（カーソルを除いて）も異なって処理されます。対象の変数は宣言されなければならないませんが、PL/SQLは常にそれらを暗黙的に宣言します。この優位点は変数値をループを抜けてからでも依然としてアクセスできることです。
- カーソル変数の使用に対する様々な表記上の違いがあります。

39.12.1. 移植例

[例39-7](#)に簡単な関数のPL/SQLからPL/pgSQLへの移植方法を示します。

例 39-7. 簡単な関数のPL/SQLからPL/pgSQLへの移植

以下はOracle PL/SQLの関数です。

```
CREATE OR REPLACE FUNCTION cs_fmt_browser_version(v_name varchar,
                                                    v_version varchar)
RETURN varchar IS
BEGIN
    IF v_version IS NULL THEN
        RETURN v_name;
    END IF;
    RETURN v_name || '/' || v_version;
END;
/
show errors;
```

この関数を通じて、PL/pgSQLとの違いを見てみましょう。

- 関数プロトタイプ内の`RETURN`キーワード（関数本体ではありません）はPostgreSQLでは`RETURNS`になります。同様に`IS`は`AS`になります。PL/pgSQL以外の言語でも関数を記述できるため、`LANGUAGE`句が必要となります。
- PostgreSQLは関数本体を文字列リテラルと考えます。したがって、それを囲むドル引用符または他の引用符が必要です。これは/`で終了するOracleの方法の代替です。`
- PostgreSQLには`show errors`コマンドはありません。また、エラーが自動的に表示されるため、必要ありません。

それではPostgreSQLに移植されると、この関数がどのようなになるか見てみましょう。

```

CREATE OR REPLACE FUNCTION cs_fmt_browser_version(v_name varchar,
                                                    v_version varchar)
RETURNS varchar AS $$
BEGIN
    IF v_version IS NULL THEN
        RETURN v_name;
    END IF;
    RETURN v_name || '/' || v_version;
END;
$$ LANGUAGE plpgsql;

```

例39-8は、他の関数を生成する関数を移植する方法、ならびに、その結果発生する引用符問題を扱う方法を示します。

例 39-8. 他の関数を生成するPL/SQLをPL/pgSQLに移植

以下の手続きは、SELECT文からの行を取って、効率のためにIF文で結果を巨大な関数に埋め込んでいます。

以下はOracle版です。

```

CREATE OR REPLACE PROCEDURE cs_update_referrer_type_proc IS
    CURSOR referrer_keys IS
        SELECT * FROM cs_referrer_keys
        ORDER BY try_order;

    func_cmd VARCHAR(4000);
BEGIN
    func_cmd := 'CREATE OR REPLACE FUNCTION cs_find_referrer_type(v_host IN VARCHAR,
        v_domain IN VARCHAR, v_url IN VARCHAR) RETURN VARCHAR IS BEGIN';

    FOR referrer_key IN referrer_keys LOOP
        func_cmd := func_cmd ||
            ' IF v_' || referrer_key.kind
            || ' LIKE ''' || referrer_key.key_string
            || ''' THEN RETURN ''' || referrer_key.referrer_type
            || '''; END IF;';
    END LOOP;

    func_cmd := func_cmd || ' RETURN NULL; END;';

    EXECUTE IMMEDIATE func_cmd;
END;
/
show errors;

```

この関数をPostgreSQLで記述するとこうなるでしょう。

```

CREATE OR REPLACE FUNCTION cs_update_referrer_type_proc() RETURNS void AS $func$
DECLARE
    referrer_keys CURSOR IS
        SELECT * FROM cs_referrer_keys
        ORDER BY try_order;
    func_body text;
    func_cmd text;
BEGIN
    func_body := 'BEGIN';

    FOR referrer_key IN referrer_keys LOOP
        func_body := func_body ||
            ' IF v_' || referrer_key.kind
            || ' LIKE ' || quote_literal(referrer_key.key_string)
            || ' THEN RETURN ' || quote_literal(referrer_key.referrer_type)
            || '; END IF;';
    END LOOP;

    func_body := func_body || ' RETURN NULL; END;';

    func_cmd :=
        'CREATE OR REPLACE FUNCTION cs_find_referrer_type(v_host varchar,
            v_domain varchar,
            v_url varchar)

            RETURNS varchar AS '
        || quote_literal(func_body)
        || ' LANGUAGE plpgsql;' ;

    EXECUTE func_cmd;
END;
$func$ LANGUAGE plpgsql;

```

関数本体を別途作成し、それをquote_literalに渡して本体内の引用符を二重化する方法に注目してください。新規の関数を定義する時ドル引用符の使用が安全とは限らないため、この方法が必要となります。それはreferrer_key.key_stringの領域に、どのような文字列が書き込まれているか不明だからです。(referrer_key.kindは常に信用できるhostかdomainかurlであると仮定しますが、どんなものでもreferrer_key.key_stringになり得るので、ドル記号を含む可

性能があります。) この関数はOracle版より実際に改善されています。それは`referrer_key.key_string`または`referrer_key.referrer_type`が引用符を含む時、おかしいコードを生成しないからです。

[例39-9](#)は、OUTパラメータを持ち、文字列操作を行う関数の移植方法を示します。PostgreSQLには組み込みの`instr`関数はありませんが、他の関数を組み合わせることで作成できます。[項39.12.3](#)に、移植を簡略化できるように`instr`のPL/pgSQLによる実装を示します。

例 39-9. 文字列操作を行い、outパラメータを持つPL/SQLプロシージャのPL/pgSQLへの移植

以下のOracle PL/SQLプロシージャは、URLを解析していくつかの要素（ホスト、パス、問い合わせ）を返します。

以下はOracle版です。

```
CREATE OR REPLACE PROCEDURE cs_parse_url(
    v_url IN VARCHAR,
    v_host OUT VARCHAR, -- この値は戻されます
    v_path OUT VARCHAR, -- この値も戻されます
    v_query OUT VARCHAR) -- この値も戻されます
IS
    a_pos1 INTEGER;
    a_pos2 INTEGER;
BEGIN
    v_host := NULL;
    v_path := NULL;
    v_query := NULL;
    a_pos1 := instr(v_url, '//');

    IF a_pos1 = 0 THEN
        RETURN;
    END IF;
    a_pos2 := instr(v_url, '/', a_pos1 + 2);
    IF a_pos2 = 0 THEN
        v_host := substr(v_url, a_pos1 + 2);
        v_path := '/';
        RETURN;
    END IF;

    v_host := substr(v_url, a_pos1 + 2, a_pos2 - a_pos1 - 2);
    a_pos1 := instr(v_url, '?', a_pos2 + 1);

    IF a_pos1 = 0 THEN
        v_path := substr(v_url, a_pos2);
        RETURN;
    END IF;

    v_path := substr(v_url, a_pos2, a_pos1 - a_pos2);
    v_query := substr(v_url, a_pos1 + 1);
END;
/
show errors;
```

これをPostgreSQLで記述すると、以下のようになります。

```
CREATE OR REPLACE FUNCTION cs_parse_url(
    v_url IN VARCHAR,
    v_host OUT VARCHAR, -- この値は戻されます
    v_path OUT VARCHAR, -- この値も戻されます
    v_query OUT VARCHAR) -- この値も戻されます
AS $$
DECLARE
    a_pos1 INTEGER;
    a_pos2 INTEGER;
BEGIN
    v_host := NULL;
    v_path := NULL;
    v_query := NULL;
    a_pos1 := instr(v_url, '//');

    IF a_pos1 = 0 THEN
        RETURN;
    END IF;
    a_pos2 := instr(v_url, '/', a_pos1 + 2);
    IF a_pos2 = 0 THEN
        v_host := substr(v_url, a_pos1 + 2);
        v_path := '/';
        RETURN;
    END IF;

    v_host := substr(v_url, a_pos1 + 2, a_pos2 - a_pos1 - 2);
    a_pos1 := instr(v_url, '?', a_pos2 + 1);

    IF a_pos1 = 0 THEN
```

```

        v_path := substr(v_url, a_pos2);
        RETURN;
    END IF;

    v_path := substr(v_url, a_pos2, a_pos1 - a_pos2);
    v_query := substr(v_url, a_pos1 + 1);
END;
$$ LANGUAGE plpgsql;

```

この関数は以下のように使用できます。

```
SELECT * FROM cs_parse_url('http://foobar.com/query.cgi?baz');
```

[例39-10](#)は、Oracleに特化した多くの機能を使用したプロシージャの移植方法を示します。

例 39-10. PL/SQL プロシージャのPL/pgSQLへの移植

以下はOracle版です。

```

CREATE OR REPLACE PROCEDURE cs_create_job(v_job_id IN INTEGER) IS
    a_running_job_count INTEGER;
    PRAGMA AUTONOMOUS_TRANSACTION; (1)
BEGIN
    LOCK TABLE cs_jobs IN EXCLUSIVE MODE; (2)

    SELECT count(*) INTO a_running_job_count FROM cs_jobs WHERE end_stamp IS NULL;

    IF a_running_job_count > 0 THEN
        COMMIT; -- ロックを解放(3)
        raise_application_error(-20000,
            'Unable to create a new job: a job is currently running.');
    END IF;

    DELETE FROM cs_active_job;
    INSERT INTO cs_active_job(job_id) VALUES (v_job_id);

    BEGIN
        INSERT INTO cs_jobs (job_id, start_stamp) VALUES (v_job_id, sysdate);
    EXCEPTION
        WHEN dup_val_on_index THEN NULL; -- 既存であっても問題なし
    END;
    COMMIT;
END;
/
show errors

```

このようなプロシージャはvoid型を返すPostgreSQL関数に簡単に変換することができます。 以下のようなことを教えてくれることもあって、このプロシージャは特におもしろいものです。

(1)

PostgreSQLにはPRAGMA文はありません。

(2)

PL/pgSQLでLOCK TABLEを使用するならば、トランザクションの終了を呼び出すまでロックが解放されません。

(3)

PL/pgSQL関数内部では、COMMITを使用できません。 関数は外部のトランザクションの一部として実行されるため、COMMITは関数の実行の終了を意味するからです。 しかし、LOCK TABLEによって掛けたロックはエラーが起こった時解放されるので、この例題では特に必要ありません。

それでは、このプロシージャをPL/pgSQLに移植することができた方法を見てみましょう。

```

CREATE OR REPLACE FUNCTION cs_create_job(v_job_id integer) RETURNS void AS $$
DECLARE
    a_running_job_count integer;
BEGIN
    LOCK TABLE cs_jobs IN EXCLUSIVE MODE;

    SELECT count(*) INTO a_running_job_count FROM cs_jobs WHERE end_stamp IS NULL;

    IF a_running_job_count > 0 THEN
        RAISE EXCEPTION 'Unable to create a new job: a job is currently running'; (1)
    END IF;

    DELETE FROM cs_active_job;
    INSERT INTO cs_active_job(job_id) VALUES (v_job_id);

    BEGIN
        INSERT INTO cs_jobs (job_id, start_stamp) VALUES (v_job_id, now());
    EXCEPTION
        WHEN unique_violation THEN (2)
        -- 既存であっても問題なし

```

```
END;
END;
$$ LANGUAGE plpgsql;
```

(1)

基本のRAISE *exception_name*である場合は同様に操作できますが、RAISE構文はOracleにおける文とかなり異なります。

(2)

PL/pgSQLがサポートする例外の名称は、Oracleと異なります。 提供する例外の名称は、はるかに広範囲です（[付録A](#)を参照してください）。 今のところ、ユーザ定義の例外名称を宣言できません。 しかし代わりにユーザが選択したSQLSTATE値を返すことができます。

このプロシージャはOracle版に比べて機能的な違いがあります。 それはcs_jobsテーブルの排他的ロックがトランザクションの終了まで継続することです。 同様に、関数呼び出しの後で（例えばエラーが原因で）アボートすると、プロシージャの影響はロールバックされます。

39.12.2. その他の注意事項

本節では、Oracle PL/SQL関数のPostgreSQLへの移植における、その他の注意事項を説明します。

39.12.2.1. 例外後の暗黙的ロールバック

PL/pgSQLにおいてEXCEPTION句が例外を捕捉すると、BEGIN以降のそのブロックにおけるデータベースの変更が自動的にロールバックされます。 すなわち、OracleのPL/SQLによる以下のプログラムと同等の処理が実行されます。

```
BEGIN
    SAVEPOINT s1;
    ... code here ...
EXCEPTION
    WHEN ... THEN
        ROLLBACK TO s1;
        ... code here ...
    WHEN ... THEN
        ROLLBACK TO s1;
        ... code here ...
END;
```

このような方式でSAVEPOINTとROLLBACK TOを使用したOracleのプロシージャの書き換えは簡単です。 単にSAVEPOINTとROLLBACK TOの処理を削除すればよいだけです。 これと異なった方式でSAVEPOINTとROLLBACK TOを使用したプロシージャの時は、それに応じた工夫が必要になると思われます。

39.12.2.2. EXECUTE

PL/pgSQLのEXECUTEはPL/SQL版とよく似ています。 しかし[項39.5.4](#)で説明されているquote_literalとquote_identを使うことを覚えておかなければいけません。 これらの関数を使用しない限りEXECUTE 'SELECT * from \$1';という構文の動作には、信頼性がありません。

39.12.2.3. PL/pgSQL関数の最適化

PostgreSQLには実行を最適化するために2つの関数生成修飾子があります。 変動性（同じ引数が与えられた場合常に同じ結果が返します）と"厳密性"（引数のいずれかにNULLが含まれる場合NULLを返します）です。 詳細は[CREATE FUNCTION](#)を参照してください。

これらの最適化属性を利用するためには、CREATE FUNCTION文を以下のようにします。

```
CREATE FUNCTION foo(...) RETURNS integer AS $$
...
$$ LANGUAGE plpgsql STRICT IMMUTABLE;
```

39.12.3. 付録

本節には、移植作業を簡略化するために使用できる、Oracle互換のinstr関数のコードがあります。

```
--
-- Oracleのものと同じ動作をするinstr関数
-- 構文: instr(string1,string2,[n],[m]) ただし、[]は省略可能なパラメータ
--
-- string1内のn番目の文字からm番目の文字まででstring2を探します。
-- nが負の場合、逆方向に検索します。 mが渡されなかった場合は、
-- 1とみなします（最初の文字から検索を始めます）。
```

--

```
CREATE FUNCTION instr(vvarchar, varchar) RETURNS integer AS $$
DECLARE
    pos integer;
BEGIN
    pos:= instr($1, $2, 1);
    RETURN pos;
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;
```

```
CREATE FUNCTION instr(string varchar, string_to_search varchar, beg_index integer)
RETURNS integer AS $$
DECLARE
    pos integer NOT NULL DEFAULT 0;
    temp_str varchar;
    beg integer;
    length integer;
    ss_length integer;
BEGIN
    IF beg_index > 0 THEN
        temp_str := substring(string FROM beg_index);
        pos := position(string_to_search IN temp_str);

        IF pos = 0 THEN
            RETURN 0;
        ELSE
            RETURN pos + beg_index - 1;
        END IF;
    ELSE
        ss_length := char_length(string_to_search);
        length := char_length(string);
        beg := length + beg_index - ss_length + 2;

        WHILE beg > 0 LOOP
            temp_str := substring(string FROM beg FOR ss_length);
            pos := position(string_to_search IN temp_str);

            IF pos > 0 THEN
                RETURN beg;
            END IF;

            beg := beg - 1;
        END LOOP;

        RETURN 0;
    END IF;
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;
```

```
CREATE FUNCTION instr(string varchar, string_to_search varchar,
    beg_index integer, occur_index integer)
RETURNS integer AS $$
DECLARE
    pos integer NOT NULL DEFAULT 0;
    occur_number integer NOT NULL DEFAULT 0;
    temp_str varchar;
    beg integer;
    i integer;
    length integer;
    ss_length integer;
BEGIN
    IF beg_index > 0 THEN
        beg := beg_index;
        temp_str := substring(string FROM beg_index);

        FOR i IN 1..occur_index LOOP
            pos := position(string_to_search IN temp_str);

            IF i = 1 THEN
                beg := beg + pos - 1;
            ELSE
                beg := beg + pos;
            END IF;

            temp_str := substring(string FROM beg + 1);
        END LOOP;

        IF pos = 0 THEN
            RETURN 0;
        ELSE
            RETURN beg;
        END IF;
    ELSE

```



```
ss_length := char_length(string_to_search);
length := char_length(string);
beg := length + beg_index - ss_length + 2;

WHILE beg > 0 LOOP
    temp_str := substring(string FROM beg FOR ss_length);
    pos := position(string_to_search IN temp_str);

    IF pos > 0 THEN
        occur_number := occur_number + 1;

        IF occur_number = occur_index THEN
            RETURN beg;
        END IF;
    END IF;

    beg := beg - 1;
END LOOP;

RETURN 0;
END IF;
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;
```

[前のページ](#)

PL/pgSQLによる開発向けのヒント

[ホーム](#)
[上に戻る](#)

[次のページ](#)

PL/Tcl - Tcl手続き言語

第 40 章 PL/Tcl - Tcl 手続き言語

目次

- 40.1. [概要](#)
- 40.2. [PL/Tcl 関数と引数](#)
- 40.3. [PL/Tcl におけるデータの値](#)
- 40.4. [PL/Tcl におけるグローバルデータ](#)
- 40.5. [PL/Tcl からのデータベースアクセス](#)
- 40.6. [PL/Tcl のトリガプロシージャ](#)
- 40.7. [モジュールと unknown コマンド](#)
- 40.8. [Tcl プロシージャ名](#)

PL/Tcl とは、PostgreSQL データベースシステムにロード可能な手続き言語で、[Tcl 言語](#) を使った関数とトリガプロシージャを作成できます。

40.1. 概要

PL/Tclは、いくつか制限がありますが、C言語で書かれた関数と同じような能力を提供します。さらに、Tclで利用できる、強力な文字列処理ライブラリを持っています。

すべてがTclインタプリタの安全なコンテキスト内で実行されるという制約はやむを得ないものですが、逆に良い制約でもあります。安全なTclの制約付きのコマンドセットに、SPIを使ってデータベースにアクセスするコマンドと、`elog()`を使ってメッセージを処理するためのコマンドなどの、わずかなコマンドが追加されています。C関数では可能ですが、PL/Tclにはデータベースサーバ内部にアクセスする方法や、PostgreSQLサーバプロセスの権限によるOSレベルのアクセスを行う方法はありません。この結果、非特権データベースユーザがこの言語を信頼して使用することができます。つまり、無制限の権限は与えられません。

その他の注意すべき実装上の制約として、Tcl関数を使用して新しいデータ型用の入出力関数を作成することはできません。

例えば、メールを送るTcl関数が必要な場合など、安全なTclに制約されないTcl関数を書くことが望ましい場合があります。このような場合、`PL/TclU`（信頼されないTcl）というPL/Tclの亜種を使用します。これは、完全なTclインタプリタが使用されているという点以外の違いはありません。`PL/TclU`を使用する場合は、信頼されていない手続き言語としてインストールする必要があります。そうすることによって、データベースのスーパーユーザのみが関数を作成できるようになります。PL/TclU関数ではデータベース管理者としてログインしたユーザができるあらゆることの実行が可能となるので、作成する際に、この関数が意図された以外のことを行わないように細心の注意を払う必要があります。

インストール時にTclサポートの設定が指定されていれば、PL/TclとPL/TclU呼び出しハンドラの共有オブジェクトコードは自動的に作成され、PostgreSQLのライブラリディレクトリにインストールされます。PL/TclまたはPL/TclUの一方あるいは両方を特定のデータベースにインストールしたい場合は、`CREATE EXTENSION`または`createlang`プログラムを使用してください。例えば、`createlang pltcl dbname`または`createlang pltclu dbname`です。

40.2. PL/Tcl関数と引数

PL/Tcl言語で関数を作成するには、以下の標準的な[CREATE FUNCTION](#)構文を使用してください。

```
CREATE FUNCTION funcname (argument-types) RETURNS return-type AS $$
    # PL/Tcl関数本体
    $$ LANGUAGE pltcl;
```

PL/TclUでも、言語にpltcluを指定しなければならない点以外は同様です。

関数本体は、単なる小さなTclスクリプトです。関数が呼び出された時、引数の値はTclスクリプトに\$1 ... \$nという変数として渡されます。結果は通常通りreturn文を使用してTclのコードから返されます。

例えば、2つの整数のうち大きな方を返す関数は以下のように定義できます。

```
CREATE FUNCTION tcl_max(integer, integer) RETURNS integer AS $$
    if {$1 > $2} {return $1}
    return $2
    $$ LANGUAGE pltcl STRICT;
```

STRICT句に注意してください。これによりプログラマは、入力にNULL値が与えられた場合を検討する手間を省くことができます。NULLが渡された場合、関数はまったく呼び出されず、単にNULLという結果が自動的に返されます。

厳密 (strict) でない関数では、引数の実際の値がNULLである場合、対応する\$*n*変数は空文字列に設定されます。ある引数がNULLかどうかを検出するためには、argisnull関数を使用してください。例えば、引数の片方がNULL、もう片方が非NULLであって、NULLではなく、非NULLの引数の方を返すtcl_maxを考えると、以下のようになります。

```
CREATE FUNCTION tcl_max(integer, integer) RETURNS integer AS $$
    if {[argisnull 1]} {
        if {[argisnull 2]} { return_null }
        return $2
    }
    if {[argisnull 2]} { return $1 }
    if {$1 > $2} {return $1}
    return $2
    $$ LANGUAGE pltcl;
```

上で示した通り、NULL値をPL/Tcl関数から返すためには、return_nullを実行してください。これは、関数が厳密かどうかに関係なく、実行することができます。

複合型の引数は、Tcl配列として関数に渡されます。配列の要素名は複合型の属性名です。渡された行の属性がNULL値の場合、その属性は配列内には現れません。以下に例を示します。

```
CREATE TABLE employee (
    name text,
    salary integer,
    age integer
);

CREATE FUNCTION overpaid(employee) RETURNS boolean AS $$
    if {200000.0 < $1(salary)} {
        return "t"
    }
    if {$1(age) < 30 && 100000.0 < $1(salary)} {
        return "t"
    }
    return "f"
    $$ LANGUAGE pltcl;
```

今のところ、複合型の戻り値を返す機能や集合を返す機能はサポートされていません。

今のところPL/Tclはドメイン型を完全にサポートしていません。ドメインをその背後にあるスカラ型と同様に扱います。これは、ドメインに関連する制約が強制されないことを意味しています。これは関数の引数では問題ありませんが、PL/Tcl関数をドメイン型を返すように宣言した場合は危険です。

40.3. PL/Tcl におけるデータの値

PL/Tcl 関数コードに与えられる引数の値は、単に、テキスト形式（`SELECT` 文によりそれを表示した場合と同じ形式）に変換された入力引数です。逆に、`return` コマンドは、その関数宣言における戻り値の型の入力書式として受け付けることができる、任意の文字列を受け付けます。このように PL/Tcl 関数ではすべての値は単なるテキスト文字列です。

40.4. PL/Tclにおけるグローバルデータ

ある関数の複数の呼び出し間で保持される、もしくは、異なる関数間で共有されるような、いくつかのグローバルデータを持つことが有意な場合があります。これはPL/Tclで簡単に実現できますが、理解する必要がある制限がいくつかあります。

セキュリティ上の理由のため、PL/Tclは、任意のSQLロールによって呼び出された関数をそのロール用の別のTclインタプリタで実行します。これにより、他のユーザのPL/Tcl関数の処理によってあるユーザへの事故または悪意の干渉を防止します。こうしたインタプリタはそれぞれ任意の"グローバル"なTcl変数を持ちます。したがって、同じSQLロールにより実行されていれば、2つのPL/Tcl関数は同じグローバル変数を共有します。単一セッション内で (`SECURITY DEFINER` 関数または `SET ROLE` などを通して) 複数のSQLロールでコードを実行するアプリケーションでは、PL/Tcl関数が確実にデータを共有できるように明示的な処理を行う必要があるかもしれません。このためには、通信しなければならない関数が同一ユーザで所有されていること、および、それが `SECURITY DEFINER` として印がついていることを確実にしてください。当然ながら、こうした関数が意図しない動作を行うために使用できないことに注意しなければなりません。

セッション内で使用されるすべてのPL/TclU関数は、当然ながらPL/Tcl関数とは別のインタプリタですが、同一のTclインタプリタ内で実行されます。このためPL/TclU関数間ではグローバルデータは自動的に共有されます。すべてのPL/TclU関数は同じ信頼レベル、すなわちデータベーススーパーユーザで実行されますので、これはセキュリティ上危険とはみなされません。

PL/Tcl関数が予期しない相互作用に巻き込まれないようにするために、`upvar` コマンドを使用することによって、各関数でアクセスできるグローバルな配列を作成することができます。この変数のグローバル名は関数の内部名で、ローカル名は `GD` となります。関数の永続局所データでは `GD` を使用することを推奨します。複数の関数で共用させる予定の値に対してのみ、通常のTclのグローバル変数を使用してください。(`GD` 配列が特定のインタプリタ内のみでグローバルであることに注意してください。このため、これらは上記のセキュリティ制限を迂回することはありません。)

後述の `spi_execp` の例の中に `GD` の使用例があります。

40.5. PL/Tclからのデータベースアクセス

下記のコマンドは、PL/Tcl関数内からデータベースアクセスを行う時に使用できるコマンドです。

```
spi_exec ?-count n? ?-array name? command ?loop-body?
```

文字列として与えられたSQL問い合わせを実行します。 コマンド内のエラーは、エラーの発生となります。 さもなければ、この`spi_exec`の戻り値はコマンドによって処理（選択、挿入、更新、削除）された行数、または、コマンドがユーティリティ文の場合はゼロとなります。 さらに、コマンドが`SELECT`文の場合、選択された列の値は以下のようにTclの変数に格納されます。

`-count` オプションの値は、`spi_exec`に対し、そのコマンドで処理する最大行数を指示します。 これにより、問い合わせをカーソルとして設定し、`FETCH n`を実行することと同じことができます。

コマンドが`SELECT`文の場合、その結果得られた列の値は、列名にちなんだ名前のTcl変数に格納されます。 `-array`オプションが付与された場合は、列の値は指定した名前の連想配列に格納され、その配列のインデックスとして列名が使用されます。

問い合わせ文が`SELECT`文、かつ、`loop-body`スクリプトが付与されなかった場合、結果のうち最初の行だけがTcl変数に格納されます。 他にも行があったとしても、それらは無視されます。 問い合わせが行を返さなかった場合は、変数への格納は発生しません（`spi_exec`の戻り値を検査することで、これを検出することができます）。 以下に例を示します。

```
spi_exec "SELECT count(*) AS cnt FROM pg_proc"
```

これは、`$cnt` Tcl変数を、`pg_proc`システムカタログの行数に設定します。

`loop-body`オプション引数が付与された場合、それは、問い合わせの結果内の行それぞれに対して一度だけ実行される小さなTclスクリプトです（`loop-body`は`SELECT`以外の問い合わせで付与された場合は無視されます）。 処理中の行の列値は、各繰り返しの前にTcl変数に格納されます。 以下に例を示します。

```
spi_exec -array C "SELECT * FROM pg_class" {  
    elog DEBUG "have table $C(relname)"  
}
```

これは、`pg_class`の各行に対してログメッセージを出力します。 この機能は他のTclの繰り返し構文でも同様に動作します。 特にループ本体内の`continue`と`break`は通常通り動作します。

問い合わせの結果、列がNULLであった場合、対象となる変数は代入されずに、"未設定状態"になります。

```
spi_prepare query typelist
```

後の実行のために問い合わせ計画の準備、保存を行います。 保存された計画は現在のセッションが終了するまで保持されます。

問い合わせはパラメータ、つまり、計画が実際に実行される時に常に与えられる値用のプレースホルダを持つことができます。 問い合わせ文字列の中では、`$1 ... $n`というシンボルを使用して引数を参照してください。 問い合わせがパラメータを使用する場合、Tclのリストとしてパラメータの型名を指定する必要があります。（パラメータを使用しない場合は`typelist`には空のリストを指定してください。）

`spi_prepare`の戻り値は問い合わせIDです。 このIDは後に`spi_execp`を呼び出す時に使用されます。 使用例については`spi_execp`を参照してください。

```
spi_execp ?-count n? ?-array name? ?-nulls string? queryid ?value-list? ?loop-body?
```

`spi_prepare`により事前に準備された問い合わせを実行します。 `queryid`は`spi_prepare`により返されたIDです。 その問い合わせがパラメータを参照する場合、`value-list`を与える必要があります。 これは、そのパラメータの実際の値を持つTclのリストです。 このリストの長さは、事前に`spi_prepare`で指定した引数型のリストの長さと同じでなければなりません。 問い合わせにパラメータがない場合は、`value-list`を省略してください。

`-nulls`オプションの値は、空白文字と'`n`'という文字からなる文字列で、`spi_execp`に対し、どの引数がNULL値かを示します。 指定された場合、その文字列の長さは`value-list`の長さと正確に一致していなければなりません。 指定されない場合は、すべてのパラメータの値は非NULLです。

問い合わせとそのパラメータをどこで指定するのかという点を除き、`spi_execp`は`spi_exec`と同様に動作します。 `-count`、`-array`、`loop-body`オプションも、そして、結果の値も同じです。

ここで、プリペアド計画を使用した、PL/Tcl関数の例を示します。


```
CREATE FUNCTION t1_count(integer, integer) RETURNS integer AS $$
  if {[ info exists GD(plan) ]} {
    # 最初の呼び出しでは保存する計画を準備します。
    set GD(plan) [ spi_prepare ¥
      "SELECT count(*) AS cnt FROM t1 WHERE num >= ¥$1 AND num <= ¥$2" ¥
      [ list int4 int4 ] ]
  }
  spi_execp -count 1 $GD(plan) [ list $1 $2 ]
  return $cnt
$$ LANGUAGE pltcl;
```

`spi_prepare`に与える問い合わせ文字列の内側では、`$n`記号が確実にそのまま`spi_prepare`に渡され、Tcl変数の代入による置き換えが起こらないようにバックスラッシュが必要です。

`spi_lastoid`

直前の`spi_exec`または`spi_execp`によるコマンドが単一行のINSERT文であり、かつ、更新されるテーブルがOIDを持つ場合、そのコマンドによって挿入された行のOIDを返します。（さもなければ、ゼロを返します。）

`quote string`

指定された文字列内のすべての単一引用符とバックスラッシュ文字を二重化します。`spi_exec`や`spi_prepare`で与えられたSQL問い合わせに挿入される予定の文字列を安全に引用符付けするために、これを使用することができます。例えば、以下のような問い合わせ文字列を考えます。

```
"SELECT '$val' AS ret"
```

ここで、`val` Tcl変数に`doesn't`が実際に含まれているものとします。これは最終的に以下の問い合わせ文字列になってしまいます。

```
SELECT 'doesn't' AS ret
```

これでは、`spi_exec`または`spi_prepare`の実行中に解析エラーが発生してしまいます。正しく稼働させるには、実行したい問い合わせは以下のようにしなければなりません。

```
SELECT 'doesn''t' AS ret
```

これは、PL/Tclでは以下により形成することができます。

```
"SELECT '[ quote $val ]' AS ret"
```

`spi_execp`の持つ1つの利点は、パラメータはSQL問い合わせ文字列の一部として解析されることがありませんので、このようにパラメータの値を引用符付けする必要がないことです。

`elog level msg`

ログまたはエラーメッセージを発行します。使用できるレベルは、DEBUG、LOG、INFO、NOTICE、WARNING、ERROR、およびFATALです。ERRORはエラー状態が発生します。その上位レベルのTclコードで例外が捕捉されなければ、このエラーは問い合わせ呼び出し処理の外部へ伝播され、その結果、現在のトランザクションもしくはサブトランザクションはアボートされます。これは実質的にTclの`error`コマンドと同一です。FATALはトランザクションをアボートし、現在のセッションを停止させます。（PL/Tcl関数においてこのエラーレベルを使用すべき理由はおそらく存在しませんが、完全性のために用意されています。）他のレベルは、異なる重要度のメッセージを生成するだけです。

[log_min_messages](#)と[client_min_messages](#)設定パラメータは、特定の重要度のメッセージをクライアントに報告するか、サーバのログに書き出すか、あるいはその両方かを制御します。詳細については[第18章](#)を参照してください。

[前のページ](#)

PL/Tclにおけるグローバルデータ

[ホーム](#)
[上に戻る](#)

[次のページ](#)

PL/Tclのトリガプロシージャ

40.6. PL/Tcl のトリガプロシージャ

トリガプロシージャを PL/Tcl で作成することができます。PostgreSQL では、トリガとして呼び出されるプロシージャは、`trigger` 型の戻り値を返す引数のない関数として宣言する必要があります。

トリガマネージャからの情報は、以下の変数内に格納されてプロシージャ本体に渡されます。

<code>\$TG_name</code>	<code>CREATE TRIGGER</code> 文によるトリガ名。
<code>\$TG_relid</code>	そのトリガプロシージャ呼び出しが発生したテーブルのオブジェクト ID。
<code>\$TG_table_name</code>	そのトリガプロシージャ呼び出しが発生したテーブルの名前。
<code>\$TG_table_schema</code>	そのトリガプロシージャ呼び出しが発生したテーブルのスキーマ。
<code>\$TG_relatts</code>	先頭に空のリスト要素を持つ、テーブルの列名の Tcl リスト。Tcl の <code>lsearch</code> コマンドを使用して、そのリストから列名を検索することで、最初の列を 1 とした要素番号が返されます。これは、PostgreSQL での通常の列の番号付けと同じです。（また空のリスト要素は、右側の列の属性番号を正しくするために、削除された列の位置に現れます。）
<code>\$TG_when</code>	トリガイベントの種類に応じた、 <code>BEFORE</code> 、 <code>AFTER</code> または <code>INSTEAD OF</code> という文字列。
<code>\$TG_level</code>	トリガイベントの種類に応じた、 <code>ROW</code> または <code>STATEMENT</code> という文字列。
<code>\$TG_op</code>	トリガイベントの種類に応じた、 <code>INSERT</code> 、 <code>UPDATE</code> 、 <code>DELETE</code> 、または <code>TRUNCATE</code> の文字列。
<code>\$NEW</code>	<code>INSERT</code> / <code>UPDATE</code> 動作の場合は新しいテーブル行の値を、 <code>DELETE</code> 動作の場合は空を持つ連想配列。配列のインデックスは列名です。 <code>NULL</code> の列はこの配列内には現れません。文レベルのトリガに対しては設定されません。
<code>\$OLD</code>	<code>UPDATE</code> / <code>DELETE</code> 動作の場合は古いテーブル行の値を、 <code>INSERT</code> 動作の場合は空を持つ連想配列。配列のインデックスは列名です。 <code>NULL</code> の列はこの配列内には現れません。文レベルのトリガに対しては設定されません。
<code>\$args</code>	<code>CREATE TRIGGER</code> 文で指定された、プロシージャへの引数の Tcl リスト。この引数は、プロシージャ本体から <code>\$1 ... \$n</code> としてもアクセスすることができます。

トリガプロシージャからの戻り値は、`OK` という文字列、`SKIP` という文字列、`array get` Tcl コマンドによって返されるリストの内の 1 つを取ることができます。戻り値が `OK` の場合、トリガを発行した操作 (`INSERT`/`UPDATE`/`DELETE`) は正常に処理されます。`SKIP` はトリガマネージャにこの行に対する操作を何も出力せずに中止するように通知します。リストが返された場合は、PL/Tcl に対し、変更した行を挿入するようにトリガマネージャに返すことを通知します。これは、`$NEW` 内で与えられる行ではなく変更された行が挿入される、行レベルの `BEFORE INSERT` または `UPDATE` トリガ、または、返される行が `INSERT RETURNING` および `UPDATE RETURNING` コマンドをサポートするために使用される、行レベルの `INSTEAD OF INSERT` または `UPDATE` トリガでのみ有効です。その他の種類のトリガでは戻り値は無視されます。

ここで、テーブル内の整数値としてその行に対する更新数を記録させる、小さなトリガプロシージャの例を示します。新規の行が挿入された場合は、その値はゼロに初期化され、その後の各更新操作時に 1 が加算されます。

```
CREATE FUNCTION trigfunc_modcount() RETURNS trigger AS $$
    switch $TG_op {
        INSERT {
            set NEW($1) 0
```

```
    }
    UPDATE {
        set NEW($1) $OLD($1)
        incr NEW($1)
    }
    default {
        return OK
    }
}
return [array get NEW]
$$ LANGUAGE pltcl;

CREATE TABLE mytab (num integer, description text, modcnt integer);

CREATE TRIGGER trig_mytab_modcount BEFORE INSERT OR UPDATE ON mytab
FOR EACH ROW EXECUTE PROCEDURE trigfunc_modcount('modcnt');
```

トリガプロシージャ自身は列名を認識していない点に注目してください。これはトリガの引数として与えられます。これにより、このトリガプロシージャを別のテーブルで再利用することができます。

[前のページ](#)

PL/Tclからのデータベースアクセス

[ホーム](#)
[上に戻る](#)

[次のページ](#)

モジュールとunknown コマンド

40.7. モジュールとunknownコマンド

PL/Tclでは、使用時に自動的にTclのコードを読み込む機能があります。これは、Tclコードのモジュールを含むと仮定される`pltcl_modules`という特殊なテーブルを認識します。このテーブルが存在する場合、そのテーブルから`unknown`モジュールが取り出され、データベースセッション内でPL/Tclが最初に行われる直前にTclインタプリタに読み込まれます。（セッション内で複数のインタプリタが使用される場合、これは個々のTclインタプリタそれぞれで起こります。[項40.4](#)を参照してください。）

実際、`unknown`モジュールには必要な任意の初期化コードを含めることができますが、通常は、そこにTcl "unknown"プロシージャを定義します。このプロシージャはTclが呼び出されたプロシージャ名を認識できなかった場合に常に呼び出されます。このプロシージャのPL/Tcl標準バージョンでは、必要なプロシージャを定義している`pltcl_modules`からのモジュール検索を試みます。プロシージャが検出された場合、インタプリタに読み込まれ、その後、元々試みられたプロシージャ呼び出しを実行することが許されます。二次的な`pltcl_modfuncs`テーブルは、どの関数がどのモジュールで定義されているかに関するインデックスを提供します。これにより検索がかなり高速になります。

PostgreSQLの配布には、これらのテーブル管理用のサポートスクリプト、`pltcl_loadmod`、`pltcl_listmod`および`pltcl_delmod`が含まれています。同様に標準の`unknown`モジュールのソースが`share/unknown.pltcl`に含まれています。自動読み込み機構をサポートさせるためには、あらかじめ各データベースにこのモジュールを読み込ませる必要があります。

`pltcl_modules`および`pltcl_modfuncs`テーブルは全ユーザから読み取り可能でなければなりません。しかし、その所有者をデータベース管理者とし、データベース管理者のみが書き込み可能とする方が良いでしょう。セキュリティ対策としてPL/Tclは`pltcl_modules`をスーパーユーザが所有していない限りそれを無視します(このため`unknown`モジュールの読み込みを行いません)。しかし、十分信頼できるのであればこのテーブル上の更新権限を他のユーザに付与することができます。

40.8. Tcl プロシージャ名

PostgreSQL では、その関数の引数の数または引数の型が異なっていれば、同じ関数名を異なる関数定義に使用することが可能です。しかし、Tcl ではプロシージャ名の重複は許されません。PL/Tcl では、プロシージャ名の一部に `pg_proc` システムテーブルにあるその関数のオブジェクトID を持たせた内部的な Tcl プロシージャ名を作成することでこれに対応しています。したがって、こういった異なる引数の型を持つ同じ名前の PostgreSQL 関数は、異なる Tcl プロシージャになります。PL/Tcl プログラムから見ますと、通常は問題にはなりません、デバッグの際に表面に現れます。

第 41章 PL/Perl - Perl手続き言語

目次

- 41.1. [PL/Perl関数と引数](#)
- 41.2. [PL/Perlにおけるデータ値](#)
- 41.3. [組込み関数](#)
 - 41.3.1. [PL/Perlからのデータベースアクセス](#)
 - 41.3.2. [PL/Perlのユーティリティ関数](#)
- 41.4. [PL/Perlにおけるグローバルな値](#)
- 41.5. [信頼されたPL/Perlおよび信頼されないPL/Perl](#)
- 41.6. [PL/Perlトリガ](#)
- 41.7. [PL/Perlの内部](#)
 - 41.7.1. [設定](#)
 - 41.7.2. [制限および存在しない機能](#)

PL/Perlは[Perlプログラミング言語](#)を使用してPostgreSQL関数を作成することができる、ロード可能な手続き言語です。

PL/Perlを使用する主たる利点は、ストアードプロシージャの中で、さまざまな"文字列操作"やPerlで使用可能な関数を使用できるという点です。複雑な文字列解析は、PL/pgSQLで提供される文字列関数や制御構造体を使用するよりPerlを使用する方が簡単に行うことができます。

PL/Perlを特定のデータベースにインストールするには、`CREATE EXTENSION plperl`を使用するか、シェルのコマンドラインから`createlang plperl dbname`を使用してください。

ティップ: 言語を`template1`にインストールすると、その後に作成されるデータベース全てにその言語は自動的にインストールされます。

注意: ソースパッケージを使用するユーザは、インストール作業時にPL/Perlを特別に使用可能にする必要があります。（詳細については、[第15章](#)を参照してください。） バイナリパッケージを使用する場合は、別個のサブパッケージにPL/Perlが入っている可能性があります。

41.1. PL/Perl関数と引数

PL/Perl言語で関数を作成するには、以下の標準的な[CREATE FUNCTION](#)構文を使用してください。

```
CREATE FUNCTION funcname (argument-types) RETURNS return-type AS $$
# PL/Perl関数本体
$$ LANGUAGE plperl;
```

関数本体は通常のPerlのコードです。実際、PL/Perlの糊付けコードは、これをPerlのサブルーチンの内部に格納します。PL/Perl関数はスカラコンテキストとして呼び出されます。このためリストを返すことはできません。後述の通り、参照を返すことによりスカラ以外の値（配列、レコード、集合）を返すことができます。

またPL/Perlは[DO](#)文で呼び出される匿名コードブロックをサポートします。

```
DO $$
# PL/Perl code
$$ LANGUAGE plperl;
```

匿名コードブロックは引数を取りません。また何らかの値を返したとしても破棄されます。その他は関数と同様に動作します。

注意: Perl、特にその閉ざされたスコープで局所変数を参照するような場合では、名前付きの入れ子状サブルーチンの使用は危険です。PL/Perl関数はサブルーチン内に格納されますので、内部に記述した名前付きのサブルーチンはすべて入れ子にされます。一般的に、コード参照を介して呼び出す匿名サブルーチンを作成する方がかなり安全です。詳細はperldiagマニュアルページ内のVariable "%s" will not stay sharedおよびVariable "%s" is not availableを参照してください。またはインターネットで"perl nested named subroutine"を検索してください。

[CREATE FUNCTION](#)コマンドの構文では、関数本体は文字列定数として記述されることを必須としています。通常、文字列定数にはドル引用符付け（[項4.1.2.4](#)を参照）を使用することが最も便利です。エスケープ文字列構文E''を使用することを選択した場合、関数本体で使用される単一引用符（'）とバックスラッシュ（\）をすべて二重にしなければなりません（[項4.1.2.1](#)を参照）。

引数と結果は他のPerlサブルーチンと同様に扱われます。引数は@_の中に渡され、結果値はreturn、または、その関数で最後に評価された式として返されます。

例えば、2つの整数のうち大きな方を返す関数は以下のように定義できます。

```
CREATE FUNCTION perl_max (integer, integer) RETURNS integer AS $$
if ($_[0] > $_[1]) { return $_[0]; }
return $_[1];
$$ LANGUAGE plperl;
```

注意: PL/Perl内部での使用のため、引数はデータベースの符号化方式からUTF-8に変換され、返されるときにUTF-8からデータベースの符号化方式に戻されます。

SQLのNULL値が関数に渡された場合、その引数値はPerlにおける"未定義"として現れます。上の関数定義では、NULL値が入力された場合うまく動作しないでしょう（実際はそれがゼロであるかのように動作するでしょう）。STRICTを関数定義に加えることで、PostgreSQLの動作をより合理的にすることができます。NULL値が渡された場合、関数はまったく呼び出されず、単にNULLという結果が自動的に返されます。他の方法として、関数本体で未定義な入力进行检查することもできます。例えば、perl_maxの引数の片方がNULL、もう片方が非NULLの場合に、NULL値ではなく非NULLの引数を返すようにするとします。

```
CREATE FUNCTION perl_max (integer, integer) RETURNS integer AS $$
my ($x, $y) = @_;
if (not defined $x) {
    return undef if not defined $y;
    return $y;
}
return $x if not defined $y;
return $x if $x > $y;
return $y;
$$ LANGUAGE plperl;
```

上で示した通り、PL/Perl関数からSQLのNULL値を返すためには、未定義値を返すようにしてください。これは、関数が厳密かどうかに関係なく、実行することができます。

関数引数の内で参照ではないものは、対応するデータ型向けのPostgreSQLの標準的な外部テキスト表現で表された文字列です。通常の数値やテキスト型では、Perlは正確に処理を行いますので、通常プログラマは心配することはありません。しかし、この他の場合では、引数をPerlでより使用しやすいうように変換する必要があります。例えば、decode_bytea関数はbytea型の引

数をエスケープしないバイナリに変換するために使用することができます。

同様に、PostgreSQLに戻される値を外部テキスト表現書式で表さなければなりません。例えば、bytea型の戻り値をバイナリデータにエスケープするためにencode_byteaを使用することができます。

Perlは、PostgreSQLの配列をPerl配列への参照として返すことができます。以下に例を示します。

```
CREATE OR REPLACE function returns_array()
RETURNS text[][] AS $$
    return [['a"b','c,d'],['e¥¥f','g']];
$$ LANGUAGE plperl;

select returns_array();
```

PerlはPostgreSQLの配列をblessされたPostgreSQL::InServer::ARRAYオブジェクトとして渡します。9.1より過去のPostgreSQLで作成されたPerlコードを実行させるための後方互換性のため、このオブジェクトは配列への参照または文字列として扱うことができます。以下に例を示します。

```
CREATE OR REPLACE FUNCTION concat_array_elements(text[]) RETURNS TEXT AS $$
my $arg = shift;
my $result = "";
return undef if (!defined $arg);

# 配列への参照として
for (@$arg) {
    $result .= $_;
}

# 文字列としても働く
$result .= $arg;

return $result;
$$ LANGUAGE plperl;

SELECT concat_array_elements(ARRAY['PL','/', 'Perl']);
```

注意: Perlプログラマの常識のように、多次元配列は低次元配列の参照への参照として表現されます。

複合型の引数はハッシュへの参照として関数に渡されます。ハッシュのキーは複合型の属性名です。以下に例を示します。

```
CREATE TABLE employee (
    name text,
    basesalary integer,
    bonus integer
);

CREATE FUNCTION empcomp(employee) RETURNS integer AS $$
my ($emp) = @_;
return $emp->{basesalary} + $emp->{bonus};
$$ LANGUAGE plperl;

SELECT name, empcomp(employee.*) FROM employee;
```

必要な属性を持つハッシュの参照を返すという同じ方法で、PL/Perl関数は複合型の結果を返すことができます。以下に例を示します。

```
CREATE TYPE testrowperl AS (f1 integer, f2 text, f3 text);

CREATE OR REPLACE FUNCTION perl_row() RETURNS testrowperl AS $$
    return {f2 => 'hello', f1 => 1, f3 => 'world'};
$$ LANGUAGE plperl;

SELECT * FROM perl_row();
```

宣言された結果データ型の任意の列の内、ハッシュ内に存在しないものはNULL値として返されます。

また、PL/Perl関数はスカラ型の配列や複合型の配列を返すこともできます。通常ならば、起動処理の高速化とメモリ内の結果セット全体を待ち行列に保持できることから、1度に1行を返す方がよいでしょう。以下に示すreturn_nextを使用して、これを行うことができます。最後のreturn_nextの後で、returnまたはreturn undef（推奨）を記述しなければならないことに注意してください。

```
CREATE OR REPLACE FUNCTION perl_set_int(int)
RETURNS SETOF INTEGER AS $$
    foreach (0..$_[0]) {
        return_next($_);
    }
    return undef;
$$ LANGUAGE plperl;

SELECT * FROM perl_set_int(5);
```

```
CREATE OR REPLACE FUNCTION perl_set()
RETURNS SETOF testrowperl AS $$
    return_next({ f1 => 1, f2 => 'Hello', f3 => 'World' });
    return_next({ f1 => 2, f2 => 'Hello', f3 => 'PostgreSQL' });
    return_next({ f1 => 3, f2 => 'Hello', f3 => 'PL/Perl' });
    return undef;
$$ LANGUAGE plperl;
```

小規模な結果セットでは、それぞれ単純な型、配列型、複合型に対応する、スカラ、配列への参照、ハッシュへの参照を含む配列への参照を返すことができます。以下に、配列への参照として結果セット全体を返す単純な例をいくつか示します。

```
CREATE OR REPLACE FUNCTION perl_set_int(int) RETURNS SETOF INTEGER AS $$
    return [0..$_[0]];
$$ LANGUAGE plperl;

SELECT * FROM perl_set_int(5);

CREATE OR REPLACE FUNCTION perl_set() RETURNS SETOF testrowperl AS $$
    return [
        { f1 => 1, f2 => 'Hello', f3 => 'World' },
        { f1 => 2, f2 => 'Hello', f3 => 'PostgreSQL' },
        { f1 => 3, f2 => 'Hello', f3 => 'PL/Perl' }
    ];
$$ LANGUAGE plperl;

SELECT * FROM perl_set();
```

コード内で`strict`プラグマを使用したいのであればいくつか選択肢があります。一時的に大域的に使用するために、`SET plperl.use_strict`を真にすることができます。このパラメータは、その後のPL/Perl関数のコンパイルに影響しますが、現在のセッションでコンパイル済みの関数には影響しません。永続的に大域的に使用するためには、`postgresql.conf`ファイル内で`plperl.use_strict`を真に設定します。

特定の関数で永続的に使用するためには単純に以下を関数本体の先頭に記載してください。

```
use strict;
```

また、Perlのバージョンが5.10.0以上であれば`use`で`feature`プラグマが利用可能です。

[前のページ](#)

PL/Perl - Perl手続き言語

[ホーム](#)
[上に戻る](#)

[次のページ](#)

PL/Perlにおけるデータ値

41.2. PL/Perlにおけるデータ値

PL/Perl関数のコードに渡される引数値は、単に(`SELECT` 文で表示される場合と同様の)テキスト形式に変換された入力引数です。反対に`return`および`return_next`コマンドは、関数の宣言された戻り値の型で受け付け可能な入力書式で表された任意の文字列を受け付けます。

41.3. 組込み関数

41.3.1. PL/Perlからのデータベースアクセス

Perl関数からデータベースそのものにアクセスするには以下の関数で行います。

```
spi_exec_query(query [, max-rows])
```

`spi_exec_query`はSQLコマンドを実行し、行セット全体をハッシュへの参照を要素とする配列への参照として返します。結果が相対的に小規模であることが分かっている場合にのみこのコマンドを使用してください。以下に最大行数オプションを持った問い合わせ (`SELECT` コマンド) の例を示します。

```
$rv = spi_exec_query('SELECT * FROM my_table', 5);
```

これは`my_table`テーブルから5行までを返します。 `my_table`に`my_column`列がある場合、結果の第`$i`行の列値を以下のように取り出すことができます。

```
$foo = $rv->{rows}[$i]->{my_column};
```

`SELECT`問い合わせから返される行の総数は以下のようにアクセスできます。

```
$nrows = $rv->{processed}
```

以下は他の種類のコマンドを使用する例です。

```
$query = "INSERT INTO my_table VALUES (1, 'test')";
$rv = spi_exec_query($query);
```

この後、以下のようにコマンドステータス (例えば`SPI_OK_INSERT`) にアクセスすることができます。

```
$res = $rv->{status};
```

影響を受けた行数を取り出すには以下を行います。

```
$nrows = $rv->{processed};
```

以下に複雑な例を示します。

```
CREATE TABLE test (
    i int,
    v varchar
);

INSERT INTO test (i, v) VALUES (1, 'first line');
INSERT INTO test (i, v) VALUES (2, 'second line');
INSERT INTO test (i, v) VALUES (3, 'third line');
INSERT INTO test (i, v) VALUES (4, 'immortal');

CREATE OR REPLACE FUNCTION test_munge() RETURNS SETOF test AS $$
my $rv = spi_exec_query('select i, v from test;');
my $status = $rv->{status};
my $nrows = $rv->{processed};
foreach my $rn (0 .. $nrows - 1) {
    my $row = $rv->{rows}[$rn];
    $row->{i} += 200 if defined($row->{i});
    $row->{v} =~ tr/A-Za-z/a-zA-Z/ if (defined($row->{v}));
    return_next($row);
}
return undef;
$$ LANGUAGE plperl;

SELECT * FROM test_munge();
```

```
spi_query(command)
spi_fetchrow(cursor)
spi_cursor_close(cursor)
```

`spi_query`および`spi_fetchrow`は、大規模になる可能性がある行セット用、または、行を順番通りに返したい場合向けに組み合わせて動作します。 `spi_fetchrow`は`spi_query`と一緒になければ動作しません。 組み合わせて使用する方法について、以下の例で示します。

```
CREATE TYPE foo_type AS (the_num INTEGER, the_text TEXT);

CREATE OR REPLACE FUNCTION lotsa_md5 (INTEGER) RETURNS SETOF foo_type AS $$
use Digest::MD5 qw(md5 hex);
```

```

my $file = '/usr/share/dict/words';
my $t = localtime;
elog(NOTICE, "opening file $file at $t" );
open my $fh, '<', $file # ooh, it's a file access!
    or elog(ERROR, "cannot open $file for reading: $!");
my @words = <$fh>;
close $fh;
$t = localtime;
elog(NOTICE, "closed file $file at $t");
chomp(@words);
my $row;
my $sth = spi_query("SELECT * FROM generate_series(1,$_[0]) AS b(a)");
while (defined ($row = spi_fetchrow($sth))) {
    return_next({
        the_num => $row->{a},
        the_text => md5_hex($words[rand @words])
    });
}
return;
$$ LANGUAGE plperl;

SELECT * from lotsa_md5(500);

```

通常`spi_fetchrow`は、読み取る行がなくなったことを示す`undef`が返されるまで繰り返されるはずです。`spi_fetchrow`が`undef`を返すと`spi_query`で返されるカーソルは自動的に解放されます。すべての行を読み取りたくない場合は代わりに`spi_cursor_close`を呼び出してカーソルを解放してください。これに失敗するとメモリリークという結果になります。

```

spi_prepare(command, argument types)
spi_query_prepared(plan, arguments)
spi_exec_prepared(plan [, attributes], arguments)
spi_freeplan(plan)

```

`spi_prepare`、`spi_query_prepared`、`spi_exec_prepared`、`spi_freeplan`は、プリペアド問い合わせ用に同様の機能を実装します。`spi_prepare`は番号付き引数ブレースホルダ(\$1、\$2など)を持つ問い合わせ文字列と引数の型を表す文字列リストを受け付けます。

```

$plan = spi_prepare('SELECT * FROM test WHERE id > $1 AND name = $2',
                    'INTEGER', 'TEXT');

```

`spi_prepare`を呼び出すことで問い合わせ計画が準備されると、`spi_exec_query`により返されるものと同様の結果となる`spi_exec_prepared`や`spi_query`とまったく同じカーソルが返される`spi_query_prepared`(このカーソルは後で`spi_fetchrow`に渡すことができます)の中で、その計画を問い合わせ文字列の代わりに使用することができます。`spi_exec_prepared`の省略可能な第二パラメータは属性のハッシュ参照です。現在サポートされる唯一の属性は、問い合わせで返される最大行数を設定する`limit`です。

プリペアド問い合わせの利点は、1つの準備された計画を複数回使用して問い合わせを実行することができるという点です。計画が不要になった後、`spi_freeplan`を使用して、計画を解放することができます。

```

CREATE OR REPLACE FUNCTION init() RETURNS VOID AS $$
    $_SHARED{my_plan} = spi_prepare('SELECT (now() + $1)::date AS now',
                                     'INTERVAL');
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION add_time( INTERVAL ) RETURNS TEXT AS $$
    return spi_exec_prepared(
        $_SHARED{my_plan},
        $_[0]
    )->{rows}->[0]->{now};
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION done() RETURNS VOID AS $$
    spi_freeplan( $_SHARED{my_plan});
    undef $_SHARED{my_plan};
$$ LANGUAGE plperl;

SELECT init();
SELECT add_time('1 day'), add_time('2 days'), add_time('3 days');
SELECT done();

    add_time | add_time | add_time
-----+-----+-----
2005-12-10 | 2005-12-11 | 2005-12-12

```

`spi_prepare`内のパラメータ添字が\$1、\$2、\$3などを介して定義されることに注意してください。そのため、検出困難な不具合が簡単に発生することになる二重引用符内での問い合わせ文字列宣言はやめてください。

他の例は、`spi_exec_prepared`における省略可能なパラメータの使用について示しています。

```

CREATE TABLE hosts AS SELECT id, ('192.168.1.'||id)::inet AS address
FROM generate_series(1,3) AS id;
977/2500

```

```

CREATE OR REPLACE FUNCTION init_hosts_query() RETURNS VOID AS $$
    $_SHARED{plan} = spi_prepare('SELECT * FROM hosts
                                WHERE address << $1', 'inet');
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION query_hosts(inet) RETURNS SETOF hosts AS $$
    return spi_exec_prepared(
        $_SHARED{plan},
        {limit => 2},
        $_[0]
    )->{rows};
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION release_hosts_query() RETURNS VOID AS $$
    spi_freeplan($_SHARED{plan});
    undef $_SHARED{plan};
$$ LANGUAGE plperl;

SELECT init_hosts_query();
SELECT query_hosts('192.168.1.0/30');
SELECT release_hosts_query();

      query_hosts
-----
(1,192.168.1.1)
(2,192.168.1.2)
(2 rows)

```

41.3.2. PL/Perlのユーティリティ関数

`elog(level, msg)`

ログまたはエラーメッセージを発行します。使用できるレベルは、DEBUG、LOG、INFO、NOTICE、WARNING、およびERRORです。ERRORはエラー状態を発生します。その上位のPerlコードでこのエラーを捕捉しない場合、エラーは問い合わせの呼び出し元まで伝播し、その結果、現在のトランザクションもしくはサブトランザクションはアボートします。これは実質Perlのdieコマンドと同じです。他のレベルは、異なる重要度のメッセージを生成するだけです。[log_min_messages](#)と[client_min_messages](#)設定パラメータは、特定の重要度のメッセージをクライアントに報告するか、サーバのログに書き出すか、あるいはその両方かを制御します。詳細は[第18章](#)を参照してください。

`quote_literal(string)`

与えられた文字列を、SQL文の文字列内で文字列リテラルとして使用するために適切に引用符付けして返します。埋め込まれた単一引用符およびバックスラッシュは適切に二重化されます。quote_literalは入力がundefならばundefを返すことに注意してください。引数がundefの可能性があるのであれば、quote_nullableの方が適しています。

`quote_nullable(string)`

与えられた文字列を、SQL文の文字列内で文字列リテラルとして使用するために適切に引用符付けして返します。引数がundefの場合引用符付けされない文字列"NULL"を返します。埋め込まれた単一引用符およびバックスラッシュは適切に二重化されます。

`quote_ident(string)`

与えられた文字列を、SQL文の文字列内で識別子として使用するために適切に引用符付けして返します。必要な場合(つまり文字列に識別子用ではない文字列が含まれる、または、大文字小文字を保持する場合)のみ引用符が付けられます。埋め込まれた引用符は適切に二重化されます。

`decode_bytea(string)`

与えられた文字列の内容を表す、エスケープのないバイナリデータを返します。これはbytea符号化でなければなりません。

`encode_bytea(string)`

与えられた文字列の内容をバイナリデータ形式で符号化したbyteaを返します。

`encode_array_literal(array)`
`encode_array_literal(array, delimiter)`

参照先の配列の内容を、配列リテラル書式で表した文字列として返します([項8.15.2](#)参照)。配列への参照でない場合は引数の値は変更されません。配列リテラルの要素間の区切り文字は指定がない、または、undefの場合、デフォルトで", "です。

`encode_typed_literal(value, typename)`

Perl変数を2番目の引数として渡されたデータ型の値に変換し、その値の文字列表現を返します。入れ子状の配列や複合

型の値を正しく扱います。

`encode_array_constructor(array)`

参照先の配列の内容を配列生成書式で表した文字列として返します([項4.2.12](#)参照)。個々の値は`quote_nullable`を使用して引用符付けされます。配列への参照でない場合は、`quote_nullable`を使用して引用符付けされた引数の値が返されます。

`looks_like_number(string)`

与えられた文字列の内容がPerlの流儀で数値でありそうな場合に真値を、さもなくば偽を返します。引数が`undef`ならば`undef`を返します。先頭の空白、末尾の空白は無視されます。`Inf`および`Infinity`は数値とみなします。

`is_array_ref(argument)`

指定された引数が配列参照として扱うことができる場合、つまり、引数の`ref`が`ARRAY`または`PostgreSQL::InServer::ARRAY`の場合、真を返します。さもなくば偽を返します。

[前のページ](#)

PL/Perlにおけるデータ値

[ホーム](#)
[上に戻る](#)

[次のページ](#)

PL/Perlにおけるグローバルな値

41.4. PL/Perlにおけるグローバルな値

現在のセッションの有効期間中の関数呼び出し間でデータ（コード参照を含む）を受け渡しするためにグローバルな`%_SHARED`ハッシュを使用することができます。

データの共有について簡単な例を以下に示します。

```
CREATE OR REPLACE FUNCTION set_var(name text, val text) RETURNS text AS $$
    if ($_SHARED{$_[0]} = $_[1]) {
        return 'ok';
    } else {
        return "cannot set shared variable $_[0] to $_[1]";
    }
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION get_var(name text) RETURNS text AS $$
    return $_SHARED{$_[0]};
$$ LANGUAGE plperl;

SELECT set_var('sample', 'Hello, PL/Perl!  How''s tricks?');
SELECT get_var('sample');
```

以下は、コード参照を使用した、多少複雑な例です。

```
CREATE OR REPLACE FUNCTION myfuncs() RETURNS void AS $$
    $_SHARED{myquote} = sub {
        my $arg = shift;
        $arg =~ s/(['\$\{\}])/\\$1/g;
        return "$arg";
    };
$$ LANGUAGE plperl;

SELECT myfuncs(); /* 関数の初期化 */

/* 引用符関数を使用する関数を作成 */

CREATE OR REPLACE FUNCTION use_quote(TEXT) RETURNS text AS $$
    my $text_to_quote = shift;
    my $qfunc = $_SHARED{myquote};
    return &$qfunc($text_to_quote);
$$ LANGUAGE plperl;
```

（可読性を犠牲にすると、上は`return $_SHARED{myquote}->($_[0]);`という1行のみで置き換えることができます。）

セキュリティ上の理由により、PL/Perlは、あるロールで呼び出された関数をそのロール用に独立したPerlインタプリタ内で実行します。これにより、あるユーザの事故または悪意によって他のユーザのPL/Perl関数の動作が干渉されてしまうことを防ぎます。こうしたインタプリタはそれぞれ独自の`%_SHARED`などのグローバル状態を持ちます。したがって、同一のSQLロールによって実行された場合のみ、2つのPL/Perl関数は同じ`%_SHARED`値を共有します。1つのセッション内で複数のSQLロールの元でコードを（`SECURITY DEFINER`経由、`SET ROLE`の使用など）実行するアプリケーションでは、確実にPL/Perl関数が`%_SHARED`を介してデータを共有することができるように、明示的な処理を行う必要があります。このためには、通信しなければならない関数が同じユーザによって所有されること、および`SECURITY DEFINER`と印付けられていることを確実にしなければなりません。当然ながらこうした関数が意図していないことを行うために使用することができないように注意しなければなりません。

41.5. 信頼されたPL/Perlおよび信頼されないPL/Perl

通常、PL/Perlは`plperl`という名前で"信頼された"プログラミング言語としてインストールされます。この設定では、セキュリティを確保するためにPerlの特定の操作は無効にされます。一般的には、制限される操作は環境に作用するものです。これには、ファイルハンドル操作や`require`、`use`（外部モジュール用）が含まれます。C関数では可能ですが、Perlでは、データベースサーバ内部にアクセスする方法や、サーバプロセスの権限によるOSレベルのアクセスを行う方法はありません。この結果、データベースの全ての非特権ユーザはこの言語を使用することができます。

セキュリティ上の理由により許されていないファイルシステム操作を行うため、うまく動作しない関数の例を以下に示します。

```
CREATE FUNCTION badfunc() RETURNS integer AS $$
  my $tmpfile = "/tmp/badfile";
  open my $fh, '>', $tmpfile
    or elog(ERROR, qq{could not open the file "$tmpfile": $!});
  print $fh "Testing writing to a file\n";
  close $fh or elog(ERROR, qq{could not close the file "$tmpfile": $!});
  return 1;
$$ LANGUAGE plperl;
```

許されていない操作の使用が検証機能によって検出されますので、この関数の作成は失敗します。

制限のないPerl関数の作成が望ましい場合があります。例えば、Perl 関数を使用してメールを送信するような場合です。このような場合を扱うために、PL/Perlを"信頼されない"言語（通常PL/PerlUと呼ばれます）としてインストールすることもできます。この場合は完全なPerl言語を使用することができます。言語がインストールされた場合、`plperlU`という言語名によって、信頼されないPL/Perlの亜種が選択されます。

PL/PerlU関数の作成者は、その関数を不必要なことに使用できないように注意する必要があります。この関数は、データベース管理者としてログインしたユーザが実行できることを全て実行できるからです。データベースシステムはデータベースのスーパーユーザにのみ信頼されない言語による関数作成を許可していることに注意してください。

上記の関数が、スーパーユーザによって`plperlU`言語を使用して作成された場合、実行は可能となります。

同じ方法で、言語を`plperl`ではなく`plperlU`と指定することで、Perl内に作成された匿名コードブロックは制限された操作を使用することができます。ただし呼び出し元はスーパーユーザでなければなりません。

注意: PL/Perl関数はSQLロール毎に別々のPerlインタプリタ内で実行されますが、あるセッションで実行されるPL/PerlU関数はすべて、単一のPerlインタプリタ（PL/Perl関数用に使用されるインタプリタのいずれかではありません）内で実行されます。これによりPL/PerlU関数はデータを自由に共有することができます。しかしPL/Perl関数とPL/PerlU関数の間で通信することはできません。

注意: Perlは適切なフラグ、すなわち`usemultiplicity`または`useithreads`を付けて構築していない限り、1つのプロセス内で複数のインタプリタをサポートすることはできません。（実際にスレッドの使用が必要でなければ`usemultiplicity`を勧めます。詳細は`perlembed`マニュアルページを参照してください。）PL/Perlがこの方法で構築されていないPerlのコピーを使用する場合、1つのセッション内で1つのPerlインタプリタしか持つことができません。このため、1つのセッションでは、PL/PerlU関数、もしくは、すべて同一のSQLロールで呼び出されるPL/Perl関数のいずれかのみを実行することができます。

41.6. PL/Perlトリガ

PL/Perlを使用してトリガ関数を作成することができます。トリガ関数では、`$_TD`というハッシュへの参照に、現在のトリガイベントに関する情報が含まれています。`$_TD`は大域変数であり、各トリガ呼び出しに対して局所的な値を別々に取り出します。以下に`$_TD`というハッシュへの参照のフィールドを示します。

```
$_TD->{new}{foo}
```

NEWのfoo列値。

```
$_TD->{old}{foo}
```

OLDのfoo列値。

```
$_TD->{name}
```

呼び出されたトリガの名前。

```
$_TD->{event}
```

トリガイベント。 INSERT、UPDATE、DELETE、TRUNCATE、もしくはUNKNOWN。

```
$_TD->{when}
```

トリガがいつ呼び出されたか。 BEFORE、AFTER、INSTEAD OFもしくはUNKNOWN。

```
$_TD->{level}
```

トリガレベル。 ROW、STATEMENT、もしくはUNKNOWN。

```
$_TD->{relid}
```

トリガの発行元テーブルのOID。

```
$_TD->{table_name}
```

トリガの発行元テーブルの名前。

```
$_TD->{relname}
```

トリガの発行元テーブルの名前。これは廃止予定で、将来のリリースで削除される可能性があります。代わりに`$_TD->{table_name}`を使用してください。

```
$_TD->{table_schema}
```

トリガの発行元テーブルが存在するスキーマの名前。

```
$_TD->{argc}
```

トリガ関数の引数の数。

```
@{$_TD->{args}}
```

トリガ関数の引数。`$_TD->{argc}`が0の場合は存在しません。

行レベルトリガは以下のいずれかを返すことができます。

```
return;
```

操作を実行します。

```
"SKIP"
```

操作を実行しません。

```
"MODIFY"
```

トリガ関数によってNEW行が変更されたことを示します。

以下はトリガ関数の例で、ここまでの説明の一部を例証するものです。

```
CREATE TABLE test (
  i int,
```

```
        v varchar
    );

CREATE OR REPLACE FUNCTION valid_id() RETURNS trigger AS $$
    if (($_TD->{new}{i} >= 100) || ($_TD->{new}{i} <= 0)) {
        return "SKIP";      # INSERT/UPDATEコマンドを取消します。
    } elsif ($_TD->{new}{v} ne "immortal") {
        $_TD->{new}{v} .= "(modified by trigger)";
        return "MODIFY";    # 行を変更し、INSERT/UPDATEコマンドを実行します。
    } else {
        return;              # INSERT/UPDATEコマンドを実行します。
    }
$$ LANGUAGE plperl;

CREATE TRIGGER test_valid_id_trig
    BEFORE INSERT OR UPDATE ON test
    FOR EACH ROW EXECUTE PROCEDURE valid_id();
```

[前のページ](#)

信頼されたPL/Perlおよび信頼されないPL/Perl

[ホーム](#)
[上に戻る](#)

[次のページ](#)
PL/Perlの内部

41.7. PL/Perlの内部

41.7.1. 設定

本節ではPL/Perlに影響する設定パラメータを列挙します。

`plperl.on_init (string)`

Perlインタプリタが最初に初期化され、`plperl`または`plperl_u`での使用のための準備がなされる前に実行されるperlコードを指定します。このコードが実行される時にはSPI関数を利用できません。このコードがエラーで失敗した場合、インタプリタの初期化は中断され、呼び出し元の問い合わせに伝わり、現在のトランザクションまたはサブトランザクションがアボートすることになります。

このPerlコードは単一文字列に制限されます。長いコードをモジュール化し、`on_init`文字列でロードすることができます。以下に例を示します。

```
plperl.on_init = 'require "plperlinit.pl"
plperl.on_init = 'use lib "/my/app"; use MyApp::PgInit;'
```

`plperl.on_init`により直接または間接的に読み込まれるモジュールはすべて、`plperl`により使用可能になります。これはセキュリティの危険性が発生する可能性があります。どんなモジュールが読み込まれたかを確認するためには以下を使用します。

```
DO 'elog(WARNING, join ", ", sort keys %INC)' LANGUAGE plperl;
```

`plperl`ライブラリが`shared_preload_libraries`に含まれている場合、初期化はpostmaster内部で起こります。この場合、postmasterが不安定になる危険が出てくるため、一層の考慮が必要です。この機能を使用できるようにした大きな理由は、`plperl.on_init`でロードされるPerlモジュールはpostmaster起動時点のみでロードされなければならないためです。このため個々のデータベースセッション内にロードというオーバーヘッドをもたらすことなく即座に利用できるようになります。しかし、データベースセッションで最初に使用されるPerlインタプリタ（PL/PerlUまたはPL/Perl関数を呼び出す最初のSQLロール用のPL/Perl）に対してのみ、このオーバーヘッドを防ぐことができる点に注意してください。データベースセッション内でその後作成されるPerlインタプリタはすべて、新たに`plperl.on_init`を実行する必要があります。また、postmasterプロセス内で作成されるPerlインタプリタは子プロセスに伝播されませんので、Windowsにおける事前ロードには何かを節約することはまったくありません。

このパラメータは`postgresql.conf`ファイルまたはサーバのコマンドラインでのみ設定可能です。

`plperl.on_plperl_init (string)`

`plperl.on_plperl_u_init (string)`

これらのパラメータはそれぞれ、`plperl`または`plperl_u`用にPerlインタプリタを特化する時に実行されるPerlコードを指定します。これは、データベースセッション内でPL/PerlまたはPL/PerlU関数が最初に実行される時、または、他の言語が呼び出されたため、あるいは新しいSQLロールでPL/Perl関数が呼び出されたために追加のインタプリタを呼び出す必要があった時に起こります。この後に`plperl.on_init`による初期化が行われます。このコードを実行する時にはSPI関数を利用できません。`plperl.on_plperl_init`内のPerlコードはインタプリタを"権限で制限した"後に実行されます。このためPerlコードは信頼できる操作のみを行うことができます。

コードがエラーで失敗した場合、初期化は中断され、呼び出し元にエラーが伝わります。その結果現在のトランザクションまたはサブトランザクションはアボートします。Perl内ですで行われた処理は取り消されません。言語が再度使用される時、初期化は新しいインタプリタの中で再度試行されます。

スーパーユーザのみがこれらの設定を変更することができます。これらの設定はセッション内で変更することができますが、このような変更は関数を実行するためにすでに使用されたPerlインタプリタには影響を与えません。

`plperl.use_strict (boolean)`

真の場合、その後のPL/Perl関数のコンパイルは`strict`プラグマが有効になります。このパラメータは現在のセッションでコンパイル済みの関数には影響しません。

41.7.2. 制限および存在しない機能

現時点では、以下の機能はPL/Perlにありません。各機能の寄稿を歓迎します。

PL/Perl関数は互いに直接呼び出すことができません。

- SPIはまだ完全に実装されていません。
- `spi_exec_query`を使用して、非常に大規模なデータセットを取り出そうとする場合、これらがすべてメモリ内に保存されることに注意しなければなりません。上で示した通り、`spi_query/spi_fetchrow`を使用することで、これを避けることができます。

集合を返す関数が大規模な行セットを`return`を介してPostgreSQLに返す場合、同様の問題が起こります。前述の通り、この問題も`return_next`を使用して行毎に返すことで避けることができます。

- セッションが正常に終了した時、致命的なエラーによるものでなければ、定義された任意の`END`ブロックが実行されます。現在、その他の動作は行われません。特にファイルハンドルは自動的に吐き出されません。またオブジェクトも自動的に破棄されません。

[前のページ](#)

PL/Perlトリガ

[ホーム](#)
[上に戻る](#)

[次のページ](#)

PL/Python - Python手続き言語

第 42章PL/Python - Python手続き言語

目次

- 42.1. [Python 2対Python 3](#)
- 42.2. [PL/Python関数](#)
- 42.3. [データ値](#)
 - 42.3.1. [データ型の対応付け](#)
 - 42.3.2. [NullとNone](#)
 - 42.3.3. [配列、リスト](#)
 - 42.3.4. [複合型](#)
 - 42.3.5. [集合を返す関数](#)
- 42.4. [データの共有](#)
- 42.5. [匿名コードブロック](#)
- 42.6. [トリガ関数](#)
- 42.7. [データベースアクセス](#)
 - 42.7.1. [データベースアクセス関数](#)
 - 42.7.2. [エラーの捕捉](#)
- 42.8. [明示的サブトランザクション](#)
 - 42.8.1. [サブトランザクションのコンテキスト管理](#)
 - 42.8.2. [Pythonの旧式バージョン](#)
- 42.9. [ユーティリティ関数](#)
- 42.10. [環境変数](#)

PL/Python手続き言語を使用してPostgreSQLの関数を[Python言語](#)で作成することができます。

PL/Pythonを特定のデータベースにインストールするには、`CREATE EXTENSION plpythonu`を使用するか、またはシェルのコマンドラインから`createlang plpython dbname`を使用してください(ただし[項42.1](#)も参照してください)。

ティップ: 言語を`template1`にインストールすると、その後に作成されるデータベース全てにその言語は自動的にインストールされます。

PostgreSQL 7.4より、PL/Pythonは"信頼されない"、つまり、ユーザが実行可能なことを制限する方法を提供しない言語としてのみ利用可能です。したがって、`plpythonu`という名前に変更されました。Pythonで新しい安全な実行手法が開発されたら、将来信頼できる`plpython`の亜種は利用可能になるかもしれません。データベース管理者としてログインしたユーザにより行えることをすべて行うことができますので、信頼されないPL/Pythonによる関数開発者は、その関数は不必要なものを行うために使用できないことに注意しなければなりません。スーパーユーザのみが`plpythonu`などの信頼されない言語で関数を作成することができます。

注意: ソースパッケージを使用するユーザは、インストール処理の過程でPL/Pythonの構築が有効になるように指定する必要があります。(詳細については、インストール手順を参照してください。) バイナリパッケージを使用する場合は、別のサブパッケージにPL/Pythonが入っている可能性があります。

42.1. Python 2対Python 3

PL/PythonはPython 2およびPython 3言語の両方をサポートします。(PostgreSQLのインストール手順では、サポートするPythonの細かなマイナーバージョンに関して、より正確な情報が記載されています。) Python 2及びPython 3言語には重要な側面でいくつか互換性がないため、混在を防ぐためにPL/Pythonでは以下の命名ならびに移行計画が使用されています。

- `plpython2u` という名前のPostgreSQL言語はPython 2言語に基づいたPL/Pythonを実装します。
- `plpython3u` という名前のPostgreSQL言語はPython 3言語に基づいたPL/Pythonを実装します。
- `plpythonu` という名前の言語はデフォルトのPython言語(現時点ではPython 2)に基づいたPL/Pythonを実装します。(このデフォルトはどのローカルにインストールされたPythonがその"デフォルト"とみなされるか、例えば `/usr/bin/python` が何を示すか、とは独立しています。) PythonコミュニティにおけるPython 3への移行の進行状況に依存しますが、おそらく近い将来のPostgreSQLのデフォルトはPython 3に変わります。

この計画は、[PEP 394](#)内の、`python` コマンドの命名と移行に関する推奨に類似しています。

Python 2用のPL/PythonかPython 3用のPL/Python、またはその両方が利用できるかどうかは、構築時の設定またはインストールしたパッケージに依存します。

ティップ: どのバージョンのPythonがインストール時に検出されるか、または、`PYTHON` 環境変数を使用してバージョンを明示的に設定することで、構築される亜種が決まります。[項15.4](#)を参照してください。1つのインストールで両方のPL/Pythonを利用可能にするためには、ソースツリーで`configure`と構築を2回行う必要があります。

この結果以下のような使用方法と移行戦略となります。

- 既存のユーザおよび現時点でPython 3に興味を持たないユーザは`plpythonu` という名前の言語を使用し、当分の間何も変更する必要はありません。最終的なPython 3への移行を簡単にするために、Python 2.6/2.7への移行を介してコードを少しずつ"将来に備える"ことを勧めます。

実際には、多くのPL/Python関数はそのまま、またはわずかな変更を行うことでPython 3に移行されます。

- Python 2に大きく依存するコードがあることが分かっていて、変更する予定がないユーザは`plpython2u` 言語名を使用することができます。これは、PostgreSQLでPython 2のサポートが完全になくなる、非常に先の将来まで動作し続けます。
- Python 3に挑戦したいユーザは、現在の標準では永久に動作し続ける`plpython3u` 言語名を使用できます。遠い将来、Python 3がデフォルトになった時、美的な理由から"3"という文字はなくなることになるでしょう。
- Python 3のみのオペレーティングシステム環境を構築したい、恐れを知らぬユーザは、[pg_pltemplate](#)の内容を変更して、`plpythonu`が`plpython3u`を示すようにすることができます。ただし、世間一般と互換性がないインストールを作成していることを覚えておいてください。

また、Python 3への移植に関する情報については[Python 3.0における新機能](#)文書を参照してください。

Python 2を基にしたPL/PythonとPython 3を基にしたPL/Pythonを同じセッションで使用することはできません。動的モジュール内のシンボルが相反するため、こうするとPostgreSQLサーバプロセスがクラッシュしてしまうためです。あるセッション内でPythonのメジャーバージョンが混在させないための検査があり、不一致が見つかったらセッションは中断されます。しかし別々のセッションからであれば、同じデータベースにおいて異なるPL/Pythonの両方を使用することができます。

42.2. PL/Python関数

PL/Pythonで作成された関数は標準的な[CREATE FUNCTION](#)構文で宣言されます。

```
CREATE FUNCTION funcname (argument-list)
    RETURNS return-type
AS $$
    # PL/Python function body
$$ LANGUAGE plpythonu;
```

関数本体は単なるPythonスクリプトです。関数が呼び出されると、引数は`args[]`リストの要素として渡されます。名前付きの引数も通常の変数としてPythonスクリプトに渡されます。通常、名前付き引数の方が可読性が高くなります。結果は、Pythonコードから通常の方法、`return`または`yield`（結果セット文の場合）で返されるものです。戻り値を提供しない場合、Pythonはデフォルトの`None`を返します。PL/PythonはPythonの`None`をSQLのNULL値に変換します。

たとえば、2つの整数の内大きな数を返す関数は以下のように定義することができます。

```
CREATE FUNCTION pymax (a integer, b integer)
    RETURNS integer
AS $$
    if a > b:
        return a
    return b
$$ LANGUAGE plpythonu;
```

関数定義の本体として提供されたPythonのコードはPythonの関数に変換されます。例えば上の例は以下ようになります。

```
def __plpython_procedure_pymax_23456():
    if a > b:
        return a
    return b
```

ここで、23456はPostgreSQLにより割り当てられたこの関数のOIDです。

引数はグローバル変数として設定されます。Pythonのスコープ規則のため、これは、ブロック内でグローバルとして再宣言されていない限り、関数内で引数変数に変数名自身を含む式の値として再代入できないという難解な結果をもたらします。例えば以下は動作しません。

```
CREATE FUNCTION pystrip(x text)
    RETURNS text
AS $$
    x = x.strip() # error
    return x
$$ LANGUAGE plpythonu;
```

`x`への代入は、`x`をブロック全体に対するローカル変数にしようとし、そして、代入の右辺の`x`がPL/Pythonの関数パラメータではなく、まだ割り当てられていないローカル変数`x`を参照するためです。`global`文を使用することで、動作するようになります。

```
CREATE FUNCTION pystrip(x text)
    RETURNS text
AS $$
    global x
    x = x.strip() # ok now
    return x
$$ LANGUAGE plpythonu;
```

しかし、PL/Pythonのこうした詳細な実装に依存しないようにすることを勧めます。関数パラメータは読み取りのみとして扱うことを勧めます。

42.3. データ値

一般的にいて、PL/Pythonの目標はPostgreSQLとPythonの世界の間で"自然な"対応付けを提供することです。これは以下のようなデータの対応付けを形成します。

42.3.1. データ型の対応付け

関数引数は、以下のようにPostgreSQLの型から対応するPython型に変換されます。

- PostgreSQLの`boolean`はPythonの`bool`に変換されます。
- PostgreSQLの`smallint`および`int`はPythonの`int`に変換されます。PostgreSQLの`bigint`は、Python 2では`long`に、Python 3では`int`に変換されます。
- PostgreSQLの`real`、`double`および`numeric`はPythonの`float`に変換されます。`numeric`では情報が失われ、間違った結果をもたらす可能性があることに注意してください。将来のリリースでこれは修正されるかもしれません。
- PostgreSQLの`bytea`は、Python 2では`str`に、Python 3では`bytes`に変換されます。Python 2では文字列は文字符号化方式を持たないバイト列として扱われるはずですが。
- PostgreSQLの文字列型を含む、上記以外のデータ型はすべてPythonの`str`に変換されます。Python 2ではこの文字列はPostgreSQLのサーバ符号化方式で表されます。Python 3ではすべての文字列と同様にUnicode文字列となります。
- スカラ型以外については後述します。

関数の戻り値は、以下のようにPostgreSQLの宣言された戻り値データ型に変換されます。

- PostgreSQLの戻り値の型が`boolean`の場合、戻り値はPythonの規約に従った真に対して評価されます。つまり、0や空文字列は偽です。`'f'`が真となることには注意してください。
- PostgreSQLの戻り値の型が`bytea`の場合、戻り値は文字列(Python 2)またはbytes(Python 3)に、それぞれ対応するPythonのビルトインを使用して変換され、その結果が`bytea`に変換されます。
- その他のPostgreSQLの戻り値型では、返されるPythonの値はPythonのビルトイン`str`を使用して文字列に変換され、その結果がPostgreSQLデータ型の入力関数に渡されます。

Python 2における文字列はPostgreSQLに渡される時にPostgreSQLサーバの符号化方式でなければなりません。現在のサーバ符号化方式で無効な文字列はエラーとなりますが、符号化方式の不一致がすべて検知されることはありません。このため正確に行われなかった場合にはゴミデータという結果になります。Unicode文字列は自動的に正しい符号化方式に変換されます。このためこれを使用することがより安全でより便利です。Python 3における文字列はすべてUnicode文字列です。

- スカラデータ型以外については後述します。

宣言されたPostgreSQLの戻り値型と実際に返されるオブジェクトのPythonデータ型との間の論理的な不整合が伝わらないことに注意してください。値はいかなる場合でも変換されます。

42.3.2. NullとNone

SQLのNULL値が関数に渡されると、その引数値はPython.では`None`となります。例えば、[項42.2](#)に示された`pymax`関数の定義では、NULL入力に対して間違った結果が返されます。関数定義に`STRICT`を付与してPostgreSQLを、NULL値が渡された場合にその関数を呼び出さず、自動的に単にNULL結果を返すという、より理想的に動作させることができます。他に、関数本体でNULL入力を検査することもできます。

```
CREATE FUNCTION pyman (a integer, b integer)
  RETURNS integer
AS $$
  if (a is None) or (b is None):
    return None
  if a > b:
    return a
  return b
$$ LANGUAGE plpythonu;
```

上で示したように、PL/Python関数からSQL NULL値を返すには、`None`という値を返してください。関数を厳密とした場合でも

厳密としない場合でも、これを行うことができます。

42.3.3. 配列、リスト

SQL配列値はPythonのリストとしてPL/Pythonに渡されます。 PL/Python関数の外部にSQL配列値を返すためには、Pythonのシーケンス、例えばリストかタプルを返します。

```
CREATE FUNCTION return_arr()
  RETURNS int[]
AS $$
return (1, 2, 3, 4, 5)
$$ LANGUAGE plpythonu;

SELECT return_arr();
   return_arr
-----
      {1,2,3,4,5}
(1 row)
```

Pythonでは、文字列はシーケンスであることに注意してください。 これは予想できない影響を与えることがあります が、Pythonプログラマには慣れたものでしょう。

```
CREATE FUNCTION return_str_arr()
  RETURNS varchar[]
AS $$
return "hello"
$$ LANGUAGE plpythonu;

SELECT return_str_arr();
   return_str_arr
-----
      {h,e,l,l,o}
(1 row)
```

42.3.4. 複合型

複合型の引数はPythonのマップとして渡されます。 マップの要素名は複合型の属性名です。 渡された行の属性値がNULLの場合、マップ上ではNoneという値となります。 以下に例を示します。

```
CREATE TABLE employee (
  name text,
  salary integer,
  age integer
);

CREATE FUNCTION overpaid (e employee)
  RETURNS boolean
AS $$
if e["salary"] > 200000:
    return True
if (e["age"] < 30) and (e["salary"] > 100000):
    return True
return False
$$ LANGUAGE plpythonu;
```

Python関数から行または複合型を返す方法は複数存在します。 以下の例では

```
CREATE TYPE named_value AS (
  name text,
  value integer
);
```

を前提とします。 複合型の結果は以下のように返されます。

シーケンス型（タプルまたはリスト。ただしインデック付けができないためset は不可）

返されるシーケンスオブジェクトは、結果の複合型が持つフィールドと同じ項目数をもたなければなりません。 0というインデックスの項目が複合型の最初のフィールド、1が次のフィールド、...となります。 以下に例を示します。

```
CREATE FUNCTION make_pair (name text, value integer)
  RETURNS named_value
AS $$
return [ name, value ]
# or alternatively, as tuple: return ( name, value )
$$ LANGUAGE plpythonu;
```

任意の列でSQL NULL値を返すには、対応する位置にNoneを挿入します。

マップ（辞書）

結果型の列の値は、列名をキーとして持つマップから取り出されます。 以下に例を示します。

```
CREATE FUNCTION make_pair (name text, value integer)
  RETURNS named_value
AS $$
  return { "name": name, "value": value }
$$ LANGUAGE plpythonu;
```

余計な辞書のキーと値の組み合わせは無視されます。 存在しないキーはエラーとして扱われます。 任意の列でSQL NULLを返すためには、対応する列名をキーとしてNoneを挿入してください。

オブジェクト（__getattr__メソッドを提供する任意のオブジェクト）

これはマップと同じように動作します。 以下に例を示します。

```
CREATE FUNCTION make_pair (name text, value integer)
  RETURNS named_value
AS $$
  class named_value:
    def __init__ (self, n, v):
      self.name = n
      self.value = v
  return named_value(name, value)

  # or simply
  class nv: pass
  nv.name = name
  nv.value = value
  return nv
$$ LANGUAGE plpythonu;
```

OUTパラメータを用いる関数もサポートされています。 以下に例を示します。

```
CREATE FUNCTION multiout_simple(OUT i integer, OUT j integer) AS $$
return (1, 2)
$$ LANGUAGE plpythonu;

SELECT * FROM multiout_simple();
```

42.3.5. 集合を返す関数

また、PL/Python関数はスカラまたは複合型の集合を返すこともできます。 返されるオブジェクトは内部的にイテレータに変換されるため、複数の実現方法があります。 以下の例では、以下の複合型が存在することを仮定します。

```
CREATE TYPE greeting AS (
  how text,
  who text
);
```

集合という結果は以下から返されます。

シーケンス型（タプル、リスト、セット）

```
CREATE FUNCTION greet (how text)
  RETURNS SETOF greeting
AS $$
  # return tuple containing lists as composite types
  # all other combinations work also
  return ( [ how, "World" ], [ how, "PostgreSQL" ], [ how, "PL/Python" ] )
$$ LANGUAGE plpythonu;
```

イテレータ（__iter__メソッドとnextメソッドを提供する任意のオブジェクト）

```
CREATE FUNCTION greet (how text)
  RETURNS SETOF greeting
AS $$
  class producer:
    def __init__ (self, how, who):
      self.how = how
      self.who = who
      self.ndx = -1

    def __iter__ (self):
      return self

    def next (self):
      self.ndx += 1
      if self.ndx == len(self.who):
```

```
        raise StopIteration
    return ( self.how, self.who[self.ndx] )

    return producer(how, [ "World", "PostgreSQL", "PL/Python" ])
$$ LANGUAGE plpythonu;
```

ジェネレータ(yield)

```
CREATE FUNCTION greet (how text)
    RETURNS SETOF greeting
AS $$
    for who in [ "World", "PostgreSQL", "PL/Python" ]:
        yield ( how, who )
$$ LANGUAGE plpythonu;
```

警告

Pythonの[bug #1483133](#)のため、一部のPython 2.4デバッグ版 (`--with-pydebug`オプション付きで設定/コンパイルされたPython) が、集合結果を返すためにイテレータを使用する場合にPostgreSQLサーバをクラッシュさせることがわかっています。 未パッチのFedora 4にはこの不具合があります。 Python運用版やパッチ適用済みのFedora 4ではこの問題は起こりません。

(`RETURNS SETOF record`を使用して) `OUT`パラメータを持つ集合を返す関数もサポートされます。 以下に例を示します。

```
CREATE FUNCTION multiout_simple_setof(n integer, OUT integer, OUT integer) RETURNS SETOF record AS $$
    return [(1, 2)] * n
$$ LANGUAGE plpythonu;

SELECT * FROM multiout_simple_setof(3);
```

[前のページ](#)
PL/Python関数

[ホーム](#)
[上に戻る](#)

[次のページ](#)
データの共有

42.4. データの共有

グローバルなSD辞書は、関数呼び出し間のデータ保存のために使用することができます。この変数はプライベートな静的データです。グローバルなGD辞書は、共有データであり、セッション内の全てのPython関数で使用することができます。注意して使用してください。

各関数は、Pythonインタプリタ内で自身の実行環境を入手します。そのため、`myfunc`によるグローバルデータと関数の引数は`myfunc2`から使用することはできません。上記で説明した通り、GD辞書内のデータは例外です。

[前のページ](#)

データ値

[ホーム](#)
[上に戻る](#)[次のページ](#)

匿名コードブロック

42.5. 匿名コードブロック

PL/Pythonは[DO](#)文で呼び出される匿名コードブロックもサポートします。

```
DO $$  
    # PL/Python code  
$$ LANGUAGE plpythonu;
```

匿名コードブロックは引数を持たず、また、何か値を返したとしても破棄されます。その他は関数とまったく同様に動作します。

42.6. トリガ関数

トリガとして関数を使用した場合、`TD` 辞書にトリガに関連した値が格納されます。

```
TD["event"]
```

次のイベントが文字列として格納されます。 `INSERT`、`UPDATE`、`DELETE`、`TRUNCATE`

```
TD["when"]
```

`BEFORE`、`AFTER`、または `INSTEAD OF` のいずれかが格納されます。

```
TD["level"]
```

`ROW` または `STATEMENT` が格納されます。

```
TD["new"]  
TD["old"]
```

行レベルトリガにおいてトリガイベントに依存して、これらのフィールドの片方または両方に対応するトリガ行が格納されます。

```
TD["name"]
```

トリガ名が格納されます。

```
TD["table_name"]
```

トリガの発生元のテーブルの名前が格納されます。

```
TD["table_schema"]
```

トリガの発生元のテーブルのスキーマが格納されます。

```
TD["relid"]
```

トリガの発生元テーブルのOIDが格納されます。

```
TD["args"]
```

`CREATE TRIGGER` に引数が含まれていた場合、その引数は `TD["args"][0]` から `TD["args"][n-1]` までの範囲で使うことができます。

`TD["when"]` が `BEFORE` または `INSTEAD OF` で、かつ、`TD["level"]` が `ROW` の場合、Python の関数から、行が変更されないことを示す `None` または `"OK"`、イベントを中断したことを示す `"SKIP"` を返すことができます。 また、`TD["event"]` が `INSERT` または `UPDATE` の場合、行を変更したことを示す `"MODIFY"` を返すことができます。 さもないと、戻り値は無視されます。

42.7. データベースアクセス

PL/Python言語モジュールは自動的に`plpy`というPythonモジュールをインポートします。 このモジュールの関数と定数は、`plpy.foo`のように作成したPythonコードから使用することができます。

42.7.1. データベースアクセス関数

`plpy`モジュールはデータベースコマンドを実行するために数個の関数を用意しています。

```
plpy.execute(query [, max-rows])
```

`plpy.execute`を、問い合わせ文字列および省略可能な行数制限引数を付けて呼び出すと、問い合わせが実行され、結果オブジェクトとして問い合わせ結果が返ります。

結果オブジェクトはリストもしくは辞書オブジェクトをエミュレートします。 結果オブジェクトは、行番号や列名によってアクセスすることができます。 例を示します。

```
rv = plpy.execute("SELECT * FROM my_table", 5)
```

これは、`my_table`から5行までを返します。 `my_table`に`my_column`列が存在する場合、その列には以下のようにアクセスできます。

```
foo = rv[i]["my_column"]
```

戻った行数はビルトイン`len`関数を使用して取得できます。

結果オブジェクトには以下のメソッドが追加されています。

```
nrows()
```

コマンドによる処理の行数を返します。 戻った行数と同じとは限らないことに注意してください。 例えば、`UPDATE`コマンドではゼロでない値を返しますが、行を戻すことはありません（`RETURNING`を使用したときは別です）。

```
status()
```

`SPI_execute()` 関数の戻り値を返します。

```
colnames()
coltypes()
coltypmods()
```

各々、列名のリスト、列の型OIDのリスト、列に関する型独自の型修飾子のリストを返します。

`RETURNING`を持たない`UPDATE`や`DROP TABLE`など、結果セットを生成しないコマンドによる結果オブジェクトに対して呼び出された場合、これらのメソッドは例外を発生します。 しかし、ゼロ行の結果セットに対してこれらのメソッドを使用することには問題ありません。

結果オブジェクトは変更できます。

`plpy.execute`を呼び出すことにより、結果セット全体がメモリ内に読み込まれることに注意してください。 結果セットが比較的小さいことが確実な場合だけ、この関数を使用してください。 大規模な結果を取り込む場合の過渡なメモリ使用に関する危険を回避したい場合は、`plpy.execute`ではなく`plpy.cursor`を使用してください。

```
plpy.prepare(query [, argtypes])
plpy.execute(plan [, arguments [, max-rows]])
```

`plpy.prepare`は問い合わせの実行計画を準備します。 問い合わせ内にパラメータ参照がある場合、問い合わせ文字列および引数型のリストとともに呼び出されます。 例を示します。

```
plan = plpy.prepare("SELECT last_name FROM my_users WHERE first_name = $1", ["text"])
```

`text`は`$1`として渡される変数の型です。 問い合わせにパラメータを渡さない場合、2番目の引数は省略可能です。

文を準備した後、それを実行するために関数`plpy.execute`の亜種を使用します。

```
rv = plpy.execute(plan, ["name"], 5)
```

実行計画を（問い合わせ文字列ではなく）最初の引数として渡してください。 問い合わせに代入する値のリストを、2番

目の引数として渡してください。問い合わせにパラメータがない場合、2番目の引数は省略可能です。3番目の引数は、前に述べた省略可能な行数制限引数です。

問い合わせパラメータおよび結果行のフィールドは[項42.3](#)で示した通り、PostgreSQLとPythonのデータ型の間で変換されます。現在複合型がサポートされていないという例外があります。複合型は問い合わせパラメータとしては拒絶され、問い合わせの結果に存在する場合は文字列に変換されます。後者の問題の回避方法として、結果行のフィールドとしてではなく結果行として複合型が存在するように問い合わせを書き換えることができる場合があります。他に、結果として生じる文字列を手作業で解析して分離することもできますが、この方法は時代に左右されてしまいますので推奨されません。

PL/Pythonモジュールを使用して準備した計画は自動的に保存されます。これが何を意味するのかについてはSPIの文書([第43章](#))を参照してください。これを複数呼び出しにおいて効果的に使用するためには、永続的な格納用辞書であるSDまたはGD([項42.4](#)を参照)のいずれかを使用する必要があります。例を示します。

```
CREATE FUNCTION usesavedplan() RETURNS trigger AS $$
    plan = SD.setdefault("plan", plpy.prepare("SELECT 1"))
    # rest of function
$$ LANGUAGE plpythonu;
```

```
plpy.cursor(query)
plpy.cursor(plan [, arguments])
```

`plpy.cursor`関数は`plpy.execute`と同じ引数を受け取り(行数制限引数を除いた)カーソルオブジェクトとして返します。これにより大規模な結果セットをより小さな塊の中で処理することができます。`plpy.execute`の場合と同様、問い合わせ文字列または引数リスト付きの計画オブジェクトを使用することができます。

カーソルオブジェクトは、整数パラメータを受け、結果オブジェクトを返す`fetch`メソッドを提供します。`fetch`を呼び出す度に、返されるオブジェクトには次の一群の行が含まれます。この行数はパラメータ値より多くなることはありません。全ての行が出し尽くされると、`fetch`は空の結果オブジェクトを返すようになります。カーソルオブジェクトはまた、すべての行を出し尽くすまで一度に1行を生成する[イテレータインタフェース](#)を提供します。この方法で取り出されたデータは結果オブジェクトとしては返されず、1つの辞書が単一の結果行に対応する辞書群として返されます。

大きなテーブルのデータを処理する、2つの方法の例を示します。

```
CREATE FUNCTION count_odd_iterator() RETURNS integer AS $$
    odd = 0
    for row in plpy.cursor("select num from largetable"):
        if row['num'] % 2:
            odd += 1
    return odd
$$ LANGUAGE plpythonu;

CREATE FUNCTION count_odd_fetch(batch_size integer) RETURNS integer AS $$
    odd = 0
    cursor = plpy.cursor("select num from largetable")
    while True:
        rows = cursor.fetch(batch_size)
        if not rows:
            break
        for row in rows:
            if row['num'] % 2:
                odd += 1
    return odd
$$ LANGUAGE plpythonu;

CREATE FUNCTION count_odd_prepared() RETURNS integer AS $$
    odd = 0
    plan = plpy.prepare("select num from largetable where num % $1 <> 0", ["integer"])
    rows = list(plpy.cursor(plan, [2]))

    return len(rows)
$$ LANGUAGE plpythonu;
```

カーソルは自動的に処分されます。しかし、カーソルが保有していた資源を明示的に開放したい場合は、`close`メソッドを使用してください。閉じた後、カーソルからこれ以上取り込むことはできません。

ティップ: `plpy.cursor`によって作成されたオブジェクトと、[PythonデータベースAPI仕様](#)において定義されたDB-APIカーソルとを混同しないでください。名称以外の共通点はありません。

42.7.2. エラーの捕捉

データベースにアクセスする関数はエラーに遭遇し、エラーが関数をアボートして例外を発生させる原因となります。`plpy.execute`および`plpy.prepare`は、デフォルトでは関数を終了させる`plpy.SPIError`のサブクラスのインスタンスを発生させることができます。このエラーは、`try/except`構文を使用して、Pythonの他の例外と同様に処理できます。例を示します。

```
CREATE FUNCTION try_adding_joe() RETURNS text AS $$
    try:
        plpy.execute("INSERT INTO users(username) VALUES ('joe')")
    except plpy.SPIError:
        "うまくいかなかった" を返す
    else:
        "Joeが追加された" を返す
$$ LANGUAGE plpythonu;
```

発生される例外の実クラスはエラーを引き起こした特定の条件と対応します。 [表A-1](#)にあり得る条件のリストがありますので参照してください。 `plpy.spiexceptions` モジュールはPostgreSQLの条件それぞれに対して、その条件名に因んだ名前の例外クラスを定義しています。 例え

ば `division_by_zero` は `DivisionByZero`、 `unique_violation` は `UniqueViolation` に、 `fdw_error` は `FdwError` などのようになります。 これらの例外クラスはそれぞれ `SPIError` を継承したものです。 このように分離することで特定のエラーをより簡単に扱うことができるようになります。 以下に例を示します。

```
CREATE FUNCTION insert_fraction(numerator int, denominator int) RETURNS text AS $$
from plpy import spiexceptions
try:
    plan = plpy.prepare("INSERT INTO fractions (frac) VALUES ($1 / $2)", ["int", "int"])
    plpy.execute(plan, [numerator, denominator])
except spiexceptions.DivisionByZero:
    return "denominator cannot equal zero"
except spiexceptions.UniqueViolation:
    return "already have that fraction"
except plpy.SPIError, e:
    return "other error, SQLSTATE %s" % e.sqlstate
else:
    return "fraction inserted"
$$ LANGUAGE plpythonu;
```

`plpy.spiexceptions` モジュールからの全ての例外は `SPIError` を継承するため、例外を処理する `except` 句は全てのデータベースアクセスエラーを捕捉することに注意してください。

異なったエラー条件を処理する代りの方法として、 `SPIError` 例外を捕捉して、例外オブジェクトの `sqlstate` 属性を調べることにより、 `except` ブロック内部の明細なエラー条件を決定できます。 この属性は "SQLSTATE" エラーコードを含む文字列値です。 この方法は、ほぼ同じ機能を提供します。

[前のページ](#)
トリガ関数

[ホーム](#)
[上に戻る](#)

[次のページ](#)
明示的サブランザクション

42.8. 明示的サブトランザクション

[項42.7.2](#)で説明したデータベースアクセスによって引き起こるエラーからの復旧は、操作の中の1つが失敗する前に、一部の操作が成功し、エラーからの復旧の後一貫性のないデータが残ってしまうという望ましくない状態を導く可能性があります。PL/Pythonは明示的サブトランザクションにより、この問題の解法を提供します。

42.8.1. サブトランザクションのコンテキスト管理

2つの口座の間の振替えを実装する関数を考えてみます。

```
CREATE FUNCTION transfer_funds() RETURNS void AS $$
try:
    plpy.execute("UPDATE accounts SET balance = balance - 100 WHERE account_name = 'joe'")
    plpy.execute("UPDATE accounts SET balance = balance + 100 WHERE account_name = 'mary'")
except plpy.SPIError, e:
    result = "error transferring funds: %s" % e.args
else:
    result = "funds transferred correctly"
plan = plpy.prepare("INSERT INTO operations (result) VALUES ($1)", ["text"])
plpy.execute(plan, [result])
$$ LANGUAGE plpythonu;
```

2番目のUPDATE文が例外を発生させる結果となった場合、この関数はエラーを記録しますが、それにもかかわらず最初のUPDATEはコミットされます。言い換えると、資金はジョーの口座から引き落とされますが、メアリーの口座には移転しません。

こうした問題を防ぐために、`plpy.execute`呼び出しを明示的なサブトランザクションで囲むことができます。`plpy`モジュールは、`plpy.subtransaction()`関数で作成される明示的なサブトランザクションを管理するための補助オブジェクトを提供します。この関数によって作成されるオブジェクトは[コンテキストマネージャインタフェース](#)を実装します。明示的なサブトランザクションを使用して、上の関数を以下のように書き換えることができます。

```
CREATE FUNCTION transfer_funds2() RETURNS void AS $$
try:
    with plpy.subtransaction():
        plpy.execute("UPDATE accounts SET balance = balance - 100 WHERE account_name = 'joe'")
        plpy.execute("UPDATE accounts SET balance = balance + 100 WHERE account_name = 'mary'")
except plpy.SPIError, e:
    result = "error transferring funds: %s" % e.args
else:
    result = "funds transferred correctly"
plan = plpy.prepare("INSERT INTO operations (result) VALUES ($1)", ["text"])
plpy.execute(plan, [result])
$$ LANGUAGE plpythonu;
```

`try/catch`の使用がまだ必要なことに注意してください。さもないと例外がPythonスタックの最上位まで伝播され、関数全体がPostgreSQLエラーにより中断され、この結果、`operations`テーブルには挿入されるはずの行が存在しないことになります。サブトランザクションのコンテキストマネージャはエラーを捕捉しません。これはそのスコープの内側で実行されるデータベース操作すべてが、原子的にコミットされるかロールバックされるかだけを保証します。サブトランザクションブロックのロールバックは、データベースアクセスを元にしたエラーによって引き起こる例外だけではなく、何らかの種類の例外終了でも起こります。明示的なサブトランザクションブロックの内側で発生した通常のPython例外も同様にサブトランザクションをロールバックさせます。

42.8.2. Pythonの旧式バージョン

デフォルトでは、`with`キーワードを使用したコンテキストマネージャ構文はPython 2.6で利用可能です。これより古いバージョンのPythonでPL/Pythonを使用する場合でも、透過性はありませんが、明示的なサブトランザクションを使用することができます。サブトランザクションマネージャの`__enter__`および`__exit__`関数を、`enter`および`exit`という便利な別名を使用して、呼び出すことができます。資金の振替えを行う関数の例は以下のように記述できます。

```
CREATE FUNCTION transfer_funds_old() RETURNS void AS $$
try:
    subxact = plpy.subtransaction()
    subxact.enter()
    try:
        plpy.execute("UPDATE accounts SET balance = balance - 100 WHERE account_name = 'joe'")
        plpy.execute("UPDATE accounts SET balance = balance + 100 WHERE account_name = 'mary'")
    except:
        import sys
        subxact.exit(*sys.exc_info())
```

```
        raise
    else:
        subxact.exit(None, None, None)
except plpy.SPIError, e:
    result = "error transferring funds: %s" % e.args
else:
    result = "funds transferred correctly"

plan = plpy.prepare("INSERT INTO operations (result) VALUES ($1)", ["text"])
plpy.execute(plan, [result])
$$ LANGUAGE plpythonu;
```

注意: コンテキストマネージャはPython 2.5で実装されましたが、このバージョンで`with`構文を使用するためには[future文](#)を使用する必要があります。しかし実装上の問題のためPL/Python関数ではfuture文を使用することができません。

[前のページ](#)

データベースアクセス

[ホーム](#)

[上に戻る](#)

[次のページ](#)

ユーティリティ関数

42.9. ユーティリティ関数

plpyでは、`plpy.debug(msg)`、`plpy.log(msg)`、`plpy.info(msg)`、`plpy.notice(msg)`、`plpy.warning(msg)`、`plpy.error(msg)`、および`plpy.fatal(msg)`関数を提供しています。 `plpy.error`および`plpy.fatal`は、実際にPythonの例外を発生させます。これが捕捉されない場合、呼び出し中の問い合わせ外部に伝わり、その結果、現在のトランザクションもしくはサブトランザクションがアボートします。 `raise plpy.Error(msg)`および`raise plpy.Fatal(msg)`は、それぞれ`plpy.error`および`plpy.fatal`の呼び出しと同じです。他の関数は異なる重要度のメッセージを生成するだけです。

[log_min_messages](#)と[client_min_messages](#)設定変数は、特定の重要度のメッセージをクライアントに報告するか、サーバのログに書き出すか、あるいはその両方かを制御します。詳細は[第18章](#)を参照してください。

この他のユーティリティ関数群には`plpy.quote_literal(string)`、`plpy.quote_nullable(string)`および`plpy.quote_ident(string)`があります。これらは[項9.4](#)で説明する組込みの引用符付け関数と同等です。これらはその場限りの問い合わせを構築する時に有用です。 [例39-1](#)の動的SQLと同等なPL/Pythonを以下に示します。

```
plpy.execute("UPDATE tbl SET %s = %s WHERE key = %s" % (
    plpy.quote_ident(colname),
    plpy.quote_nullable(newvalue),
    plpy.quote_literal(keyvalue)))
```

42.10. 環境変数

Pythonインタプリタにより受け付けられる環境変数の一部はまた、PL/Pythonの動作を変更するために使用することができます。これらは例えば起動スクリプト内など主PostgreSQLサーバプロセスの環境で設定される必要があります。利用可能な環境変数はPythonのバージョンに依存します。詳細に付いてはPythonの文書を参照してください。適切なバージョンのPythonであることが前提ですが、本章の執筆時点では以下の環境変数がPL/Pythonに影響を与えます。

- PYTHONHOME
- PYTHONPATH
- PYTHON2K
- PYTHONOPTIMIZE
- PYTHONDEBUG
- PYTHONVERBOSE
- PYTHONCASEOK
- PYTHONDONTWRITEBYTECODE
- PYTHONIOENCODING
- PYTHONUSERBASE
- PYTHONHASHSEED

(`python` マニュアルページに列挙された環境変数の一部はコマンドラインインタプリタでのみ影響を与え埋め込みPythonインタプリタには影響しないというPL/Pythonの制御を超えたPythonの詳細実装があるようです。)

第 43章サーバプログラミングインタフェース

目次

43.1. [インタフェース関数](#)

[SPI_connect](#) -- SPIマネージャにプロシージャを接続する
[SPI_finish](#) -- プロシージャをSPIマネージャから切断する
[SPI_push](#) -- 再帰的にSPIを使用できるようにSPIスタックをプッシュする
[SPI_pop](#) -- 再帰的なSPIの使用から復帰できるようにSPIスタックをポップする
[SPI_execute](#) -- コマンドを実行する
[SPI_exec](#) -- 読み書きコマンドを実行する
[SPI_execute_with_args](#) -- 行外のパラメータを持つコマンドを実行する
[SPI_prepare](#) -- 文を準備する。文の実行はまだ行わない
[SPI_prepare_cursor](#) -- 文を準備する。まだ実行は行わない
[SPI_prepare_params](#) -- 文を準備する。まだ実行は行わない
[SPI_getargcount](#) -- `SPI_prepare`により準備した文に必要とされる引数の数を返す
[SPI_getargtypeid](#) -- `SPI_prepare`で準備された文で指定される引数のデータ型のOIDを返す
[SPI_is_cursor_plan](#) -- `SPI_prepare`で準備された文が`SPI_cursor_open`で使える場合にtrueを返す
[SPI_execute_plan](#) -- `SPI_prepare`で準備された文を実行する
[SPI_execute_plan_with_paramlist](#) -- `SPI_prepare`で準備された文を実行する
[SPI_execp](#) -- 読み書きモードで文を実行する
[SPI_cursor_open](#) -- `SPI_prepare`で作成された文を使用したカーソルを設定する
[SPI_cursor_open_with_args](#) -- 問い合わせとパラメータを使ってカーソルを設定する
[SPI_cursor_open_with_paramlist](#) -- パラメータを使ってカーソルを設定する
[SPI_cursor_find](#) -- 既存のカーソルを名前を検索する
[SPI_cursor_fetch](#) -- カーソルから数行を取り出す
[SPI_cursor_move](#) -- カーソルを移動する
[SPI_scroll_cursor_fetch](#) -- カーソルから一部の行を取り出す。
[SPI_scroll_cursor_move](#) -- カーソルを移動する
[SPI_cursor_close](#) -- カーソルを閉じる
[SPI_keepplan](#) -- 準備済み文を保持する
[SPI_saveplan](#) -- 準備済み文を保存する

43.2. [インタフェースサポート関数](#)

[SPI_fname](#) -- 指定した列番号に対する列名を決定する
[SPI_fnumber](#) -- 指定した列名から列番号を決定する
[SPI_getvalue](#) -- 指定された列の文字列値を返す
[SPI_getbinval](#) -- 指定した列のバイナリ値を返す
[SPI_gettype](#) -- 指定された列のデータ型名を返す
[SPI_gettypeid](#) -- 指定された列のデータ型のOIDを返す
[SPI_getrelname](#) -- 指定されたリレーシヨンの名前を返す
[SPI_getnspname](#) -- 指定されたリレーシヨンの名前空間を返す。

43.3. [メモリ管理](#)

[SPI_palloc](#) -- 上位エグゼキュータコンテキスト内にメモリを割り当てる
[SPI_realloc](#) -- 上位エグゼキュータコンテキスト内にメモリを再割り当てる
[SPI_pfree](#) -- 上位エグゼキュータコンテキスト内のメモリを解放する
[SPI_copytuple](#) -- 上位エグゼキュータ内に行のコピーを作成する
[SPI_returntuple](#) -- Datumとしてタプルを返す準備をする
[SPI_modifytuple](#) -- 与えられた行の選択フィールドを置き換えた行を作成する
[SPI_freetuple](#) -- 上位エグゼキュータコンテキスト内に割り当てられた行を解放する
[SPI_freetupletable](#) -- `SPI_execute`や類似の関数によって生成された行セットを解放する
[SPI_freeplan](#) -- 以前に保存した準備済み文を解放する

43.4. [データ変更の可視性](#)

43.5. [例](#)

サーバプログラミングインタフェース (SPI) は、ユーザ定義のC関数からSQL問い合わせを実行する機能をユーザに提供します。SPIはパーサ、プランナ、エグゼキュータへのアクセスを単純化したインタフェース関数の集合です。また、SPIは多少のメモリ管理を行います。

注意: 利用可能な手続き言語は、プロシージャからSQLコマンドを実行するための各種手段を提供します。これらのほとんどは、SPIを基にしていますので、この文書はこれらの言語のユーザにとっても有用な場合があります。

誤解を防ぐために、これ以降、"関数"をSPIインタフェース関数の意味で、"プロシージャ"をSPIを呼び出すユーザ定義のC関数の意味で使うことにします。

コマンドがSPIの失敗を起こした場合、その制御はプロシージャには戻らないことに注意してください。それどころか、プロシージャを実行していたトランザクションもしくは副トランザクションはロールバックされます（これはSPI関数のほとんどでエラーを返す規約があることから奇妙に思われるかもしれませんが。しかし、こうした規約はSPI関数自身でエラーを検知した時にのみ適用されるものです）。失敗する可能性があるSPI呼び出しを囲む副トランザクションを独自に用意することで、エラーの後の制御を戻すことができます。要求される機構がまだ流動的であるため、これはまだ文書化されていません。

SPI関数は成功時に非負の結果を（戻り値、もしくは後述のSPI_result グローバル変数の中に）返します。エラー時、負の結果もしくはNULLを返します。

SPIを使用するソースコードファイルではexecutor/spi.hヘッダファイルをincludeしなければなりません。

[前のページ](#)
環境変数

[ホーム](#)
[上に戻る](#)

[次のページ](#)
インタフェース関数

43.1. インタフェース関数

目次

[SPI_connect](#) -- SPIマネージャにプロシージャを接続する
[SPI_finish](#) -- プロシージャをSPIマネージャから切断する
[SPI_push](#) -- 再帰的にSPIを使用できるようにSPIスタックをプッシュする
[SPI_pop](#) -- 再帰的なSPIの使用から復帰できるようにSPIスタックをポップする
[SPI_execute](#) -- コマンドを実行する
[SPI_exec](#) -- 読み書きコマンドを実行する
[SPI_execute_with_args](#) -- 行外のパラメータを持つコマンドを実行する
[SPI_prepare](#) -- 文を準備する。文の実行はまだ行わない
[SPI_prepare_cursor](#) -- 文を準備する。まだ実行は行わない
[SPI_prepare_params](#) -- 文を準備する。まだ実行は行わない
[SPI_getargcount](#) -- [SPI_prepare](#)により準備した文に必要なとされる引数の数を返す
[SPI_getargtypeid](#) -- [SPI_prepare](#)で準備された文で指定される引数のデータ型のOIDを返す
[SPI_is_cursor_plan](#) -- [SPI_prepare](#)で準備された文が[SPI_cursor_open](#)で使用できる場合にtrueを返す
[SPI_execute_plan](#) -- [SPI_prepare](#)で準備された文を実行する
[SPI_execute_plan_with_paramlist](#) -- [SPI_prepare](#)で準備された文を実行する
[SPI_execp](#) -- 読み書きモードで文を実行する
[SPI_cursor_open](#) -- [SPI_prepare](#)で作成された文を使用したカーソルを設定する
[SPI_cursor_open_with_args](#) -- 問い合わせとパラメータを使ってカーソルを設定する
[SPI_cursor_open_with_paramlist](#) -- パラメータを使ってカーソルを設定する
[SPI_cursor_find](#) -- 既存のカーソルを名前で検索する
[SPI_cursor_fetch](#) -- カーソルから数行を取り出す
[SPI_cursor_move](#) -- カーソルを移動する
[SPI_scroll_cursor_fetch](#) -- カーソルから一部の行を取り出す。
[SPI_scroll_cursor_move](#) -- カーソルを移動する
[SPI_cursor_close](#) -- カーソルを閉じる
[SPI_keepplan](#) -- 準備済み文を保持する
[SPI_saveplan](#) -- 準備済み文を保存する

SPI_connect

名前

SPI_connect -- SPI マネージャにプロシージャを接続する

概要

```
int SPI_connect(void)
```

説明

`SPI_connect` はプロシージャ呼び出しから SPI マネージャへの接続を開きます。SPI を経由してコマンドを実行させる場合、この関数を呼び出さなければなりません。SPI ユーティリティ関数の中には、未接続のプロシージャから呼び出し可能なものがあります。

既にプロシージャが接続済みの場合、`SPI_connect` は `SPI_ERROR_CONNECT` エラーを返します。これは、`SPI_connect` を呼び出したプロシージャが、直接それ自身で `SPI_connect` を呼び出す別のプロシージャを呼び出す場合に起こります。SPI を経由して呼び出される SQL コマンドが SPI を使用する他の関数を呼び出す場合、SPI マネージャの再帰的な呼び出しは許可されていますが、`SPI_connect` と `SPI_finish` を直接入れ子に呼び出すことは禁止されています（しかし、`SPI_push` および `SPI_pop` を参照してください）。

戻り値

`SPI_OK_CONNECT`

成功した場合。

`SPI_ERROR_CONNECT`

エラーが発生した場合。

SPI_finish

名前

SPI_finish -- プロシージャをSPIマネージャから切断する

概要

```
int SPI_finish(void)
```

説明

`SPI_finish` は既存のSPIマネージャへの接続を切断します。 プロシージャの現在の呼び出し期間内で必要なSPI操作が完了した後この関数を呼び出さなければなりません。 しかし、`elog(ERROR)` 経由でトランザクションを中断させる場合は、この関数が何を行うかを気にする必要はありません。 その場合、SPIは自動的に自身を整理します。

`SPI_finish` が、有効な接続がない状態で呼び出された場合、`SPI_ERROR_UNCONNECTED` が返されます。 この場合根本的な問題はありません。 SPIマネージャは特に行うことがないことを意味します。

戻り値

`SPI_OK_FINISH`

適切に切断された場合。

`SPI_ERROR_UNCONNECTED`

未接続のプロシージャから呼び出された場合。

SPI_push

名前

SPI_push -- 再帰的にSPIを使用できるようにSPIスタックをプッシュする

概要

```
void SPI_push(void)
```

説明

SPI_pushは、それ自体がSPIを使用する可能性がある他のプロシージャを実行する前に呼び出されなければなりません。SPI_pushの後、SPIは"接続"状態ではなくなります。新しくSPI_connectを実行しない限り、SPI関数の呼び出しは拒絶されます。これにより確実にプロシージャのSPI状態と呼び出した別のプロシージャのSPI状態を明確に分離できます。他のプロシージャから復帰した後にSPI_popを呼び出して、そのプロシージャのSPI状態にアクセスを戻してください。

SPI_executeとその関連関数は、制御をSQL実行エンジンに戻す前に自動的にSPI_push同等の処理を行うことに注意してください。ですので、こうした関数を使用する際には気にする必要はありません。SPI_connect呼び出しを行う可能性がある任意のコードを直接呼び出す場合のみ、SPI_pushとSPI_popを呼び出さなければなりません。

SPI_pop

名前

SPI_pop -- 再帰的なSPIの使用から復帰できるようにSPIスタックをポップする

概要

```
void SPI_pop(void)
```

説明

SPI_pop は、SPI呼び出しスタックから以前の環境をポップします。 SPI_push を参照してください。

[前のページ](#)[SPI_push](#)[ホーム](#)[上に戻る](#)[次のページ](#)[SPI_execute](#)

SPI_execute

名前

SPI_execute -- コマンドを実行する

概要

```
int SPI_execute(const char * command, bool read_only, long count)
```

説明

SPI_executeは指定したSQLコマンドを、count行分実行します。read_onlyがtrueの場合、そのコマンドは読み取りのみでなければなりません、多少のオーバーヘッドが削減されます。

この関数は接続済みのプロシージャからのみ呼び出し可能です。

countが0の場合、そのコマンドを、適用される全ての行に対して実行します。countが0より多ければ、countを超えない数の行が取り出されます。問い合わせにLIMIT句と追加するのと同じように、countに達すれば、実行は止まります。例えば、

```
SPI_execute("SELECT * FROM foo", true, 5);
```

は、テーブルから多くても5行しか取り出しません。この制限はコマンドが実際に行を返した場合にのみ有効なことに注意して下さい。例えば

```
SPI_execute("INSERT INTO foo SELECT * FROM bar", false, 5);
```

は、countパラメータを無視して、barからすべての行を挿入します。しかし、

```
SPI_execute("INSERT INTO foo SELECT * FROM bar RETURNING **", false, 5);
```

は、5番目のRETURNINGの結果行を取り出した後に実行が止まりますので、多くても5行を挿入するだけです。

複数のコマンドを1つの文字列として渡すことができます。SPI_executeは最後に実行したコマンドの結果を返します。count制限は（最後の結果が返されたただけとしても）それぞれのコマンドに独立に適用されます。この制限はルールによって生成される隠れたコマンドには適用されません。

read_onlyがfalseの場合、文字列内の各コマンドを実行する前にSPI_executeはコマンドカウンタを増分し、新しいスナップショットを作成します。このスナップショットは、現在のトランザクション隔離レベルがSERIALIZABLEまたはREPEATABLE READの場合は変更されません。しかし、READ COMMITTEDモードでは、このスナップショットは更新され、他のセッションで新しくコミットされたトランザクションの結果を各コマンドから参照できます。これは、そのコマンドがデータベースを変更する場合、一貫性の維持に重要です。

read_onlyがtrueの場合は、SPI_executeはスナップショットもコマンドカウンタも更新しません。さらに、普通のSELECTコマンドのみをコマンド文字列内に記述することができます。このコマンドは、その前後の問い合わせによって事前に確立済みのスナップショットを使用して実行されます。この実行モードは読み書きモードよりもコマンドごとのオーバーヘッドが省略される分多少高速です。また、これにより本当に安定（stable）な関数を構築することができます。つまり、連続した実行は全て同じスナップショットを使用しますので、結果は変わらないということです。

一般的に、SPIを使用する1つの関数内で読み取りのみコマンドと読み書きコマンドを混在させることは勧めません。読み取りのみの問い合わせでは、読み書き問い合わせでなされたデータベースの更新結果を参照しないいため、非常に混乱した動作に陥ることがあります。

（最後の）コマンドが実行した実際の行数は、SPI_processedグローバル変数に返されます。関数の戻り値がSPI_OK_SELECT、SPI_OK_INSERT_RETURNING、SPI_OK_DELETE_RETURNING、またはSPI_OK_UPDATE_RETURNINGの場合、SPITupleTable *SPI_tuptableグローバルポインタを使用して、結果の行にアクセスすることができます。また、一部のユーティリティコマンド（EXPLAINなど）は行セットを返しますが、この場合もSPI_tuptableにはその結果が含まれます。

SPITupleTable構造体は以下のように定義されています。

```
typedef struct
{
    MemoryContext tuptabcxt; /* 結果テーブルのメモリコンテキスト */
```

```

uint32      allocated;          /* 割り当て済みのvalsの数 */
uint32      free;              /* 解放されたvalsの数 */
TupleDesc   tupdesc;          /* 行記述子 */
HeapTuple   *vals;             /* 行 */
} SPITupleTable;

```

`vals`が行へのポインタの配列です（有効な項目数は`SPI_processed`で判明します）。`tupdesc`は行を扱うSPI関数に渡すことができる行記述子です。`tuptabcxt`、`allocated`、`free`はSPI呼び出し元での使用を意図していない内部的なフィールドです。

`SPI_finish`は、現在のプロシージャで割り当てられた`SPITupleTable`をすべて解放します。`SPI_freetuptable`を呼び出して解放する場合、特定の結果テーブルを早めに解放することができます。

引数

```
const char * command
```

実行するコマンドを含む文字列。

```
bool read_only
```

読み取りのみの実行の場合`true`。

```
long count
```

返される行の最大数。無制限なら0。

戻り値

コマンドの実行に成功した場合、以下のいずれかの（非負の）値が返されます。

```
SPI_OK_SELECT
```

`SELECT`（`SELECT INTO`を除く）が実行された場合。

```
SPI_OK_SELINTO
```

`SELECT INTO`が実行された場合。

```
SPI_OK_INSERT
```

`INSERT`が実行された場合。

```
SPI_OK_DELETE
```

`DELETE`が実行された場合。

```
SPI_OK_UPDATE
```

`UPDATE`が実行された場合。

```
SPI_OK_INSERT_RETURNING
```

`INSERT RETURNING`が実行された場合。

```
SPI_OK_DELETE_RETURNING
```

`DELETE RETURNING`が実行された場合。

```
SPI_OK_UPDATE_RETURNING
```

`UPDATE RETURNING`が実行された場合。

```
SPI_OK_UTILITY
```

ユーティリティコマンド（`CREATE TABLE`など）が実行された場合。

```
SPI_OK_REWRITTEN
```

[ルール](#)によって（例えば、`UPDATE`が`INSERT`になったような）あるコマンドが他の種類のコマンドに書き換えられた場合です。

エラーの場合、以下のいずれかの負の値が返されます。

```
SPI_ERROR_ARGUMENT
```

`command`がNULL、あるいは`count`が0未満の場合。

SPI_ERROR_COPY

`COPY TO stdout`あるいは`COPY FROM stdin`が試行された場合。

SPI_ERROR_TRANSACTION

トランザクション操作を行うコマンド（`BEGIN`、`COMMIT`、`ROLLBACK`、`SAVEPOINT`、`PREPARE TRANSACTION`、`COMMIT PREPARED`、`ROLLBACK PREPARED`、およびこれらの亜種）が試行された場合。

SPI_ERROR_OPUNKNOWN

コマンド種類が不明な場合（起きてはなりません）。

SPI_ERROR_UNCONNECTED

未接続なプロシージャから呼び出された場合。

注意

SPI問い合わせ実行関数はすべて`SPI_processed`と`SPI_tuptable`の両方を変更します（ポインタのみで、構造体の内容は変更しません）。`SPI_exec`や他の問い合わせ実行関数の結果テーブルを後の呼び出しでまたがってアクセスしたいのであれば、これら2つのグローバル変数を局所的なプロシージャ変数に保存してください。

[前のページ](#)

SPI_pop

[ホーム](#)
[上に戻る](#)

[次のページ](#)

SPI_exec

SPI_exec

名前

SPI_exec -- 読み書きコマンドを実行する

概要

```
int SPI_exec(const char * command, long count)
```

説明

SPI_execは、常にread_onlyパラメータをfalseとしたSPI_executeと同じです。

引数

```
const char * command
```

実行するコマンドを含む文字列。

```
long count
```

返される行の最大数。無制限なら0。

戻り値

SPI_executeを参照してください。

[前のページ](#)
SPI_execute

[ホーム](#)
[上に戻る](#)

[次のページ](#)
SPI_execute_with_args

SPI_execute_with_args

名前

SPI_execute_with_args -- 行外のパラメータを持つコマンドを実行する

概要

```
int SPI_execute_with_args(const char *command,
                          int nargs, Oid *argtypes,
                          Datum *values, const char *nulls,
                          bool read_only, long count)
```

説明

SPI_execute_with_argsは外部から供給されるパラメータへの参照を含むコマンドを実行します。 コマンドテキストはパラメータを\$*n*として参照し、呼び出しはこうしたシンボル毎にデータ型と値を指定します。 read_onlyとcountはSPI_executeと同じ解釈をします。

SPI_executeと比較して、このルーチンの主たる利点は、データ値を面倒な引用やエスケープを要せずコマンドに埋め込むことができることで、従ってSQLインジェクション攻撃の危険性を軽減します。

後にSPI_execute_planが続いたSPI_prepareでも同様の結果が得られますが、この関数を使用するときには、提供された特定のパラメータ値に対して問い合わせ計画が必ずカスタマイズされます。 1回限りの問い合わせ実行に対しては、この関数を選ぶべきです。 多くの異なったパラメータを持つ同一のコマンドを実行する場合、再計画のコストと独自計画による利益に依存して、どちらか一方の方法がより早くなります。

引数

const char * command

コマンド文字列

int nargs

(\$1、\$2など) 入力パラメータの数

Oid * argtypes

パラメータのデータ型のOIDを含む配列

Datum * values

実パラメータ値の配列

const char * nulls

どのパラメータがnullかを記述する配列

nullsがNULLであれば、SPI_execute_with_argsはどのパラメータもnullでないと看做す

bool read_only

読み取りのみの実行の場合true

long count

返される行の最大数。無制限なら0。

Return Value

戻り値はSPI_executeと同じです。

成功した場合SPI_processedとSPI_tuptableはSPI_executeと同様に設定されます。

[前のページ](#)
SPI_exec

[ホーム](#)
[上に戻る](#)

[次のページ](#)
SPI_prepare

SPI_prepare

名前

SPI_prepare -- 文を準備する。文の実行はまだ行わない

概要

```
SPIPlanPtr SPI_prepare(const char * command, int nargs, Oid * argtypes)
```

説明

SPI_prepareは指定したコマンド用の準備済み文を作成し、それを返します。しかし、そのコマンドは実行しません。その準備済み文はSPI_execute_planを使って後で繰り返し実行できます。

同じ、あるいは類似のコマンドが繰り返し実行される場合、一度だけ解析を計画作成を行うことには一般に利点があります。また、コマンドの実行計画を再利用することにはさらに利点があるかも知れません。SPI_prepareはコマンド文字列を、解析結果をカプセル化した準備済み文に変換します。実行の度に独自計画を生成するのが役に立たないと分かった場合には、準備済み文は実行計画をキャッシュする場所も提供します。

プリペアドコマンドは、通常のコマンド内の定数となる場所を（\$1、\$2などの）パラメータで記述することで一般化することができます。そしてパラメータの実際の値は、SPI_execute_plan が呼び出される時に指定されます。これにより、プリペアドコマンドは、パラメータがない場合に比べ、より広範な状況で使用できるようになります。

SPI_finishは文用に割り当てられたメモリを解放しますので、SPI_prepareで返される文は、そのプロシージャの現在の呼び出し内でのみ使用することができます。しかし、関数SPI_keepplanやSPI_saveplanを使用して長期間文を保存することもできます。

引数

const char * command

コマンド文字列

int nargs

（\$1、\$2など）入力パラメータ数

Oid * argtypes

パラメータのデータ型のOIDを持つ配列へのポインタ

戻り値

SPI_prepareはSPIPlanへの非NULLのポインタを返します。ここでSPIPlanは準備済み文を表すopaque構造体です。エラーの場合、NULLが返され、SPI_executeで使用されるエラーコードと同じコードの1つがSPI_resultに設定されます。しかし、commandがNULLの場合や、nargsが0未満の場合、nargsが0より大きくかつargtypesがNULLの場合は、SPI_ERROR_ARGUMENTに設定されます。

注意

パラメータが定義されていなければ、SPI_execute_planが最初に使用された時に一般的な計画が作成され、以降の実行すべてでも利用されます。パラメータがあれば、始めの何回かのSPI_execute_planの使用で、与えられたパラメータの値に固有の独自計画が作成されます。同じ準備済み文が十分に使用された後、SPI_execute_planは一般的な計画を作成し、独自計画よりもそれほど高価でなければ、毎回再計画する代わりに一般的な計画を使い始めるようになります。このデフォルトの動作が不適切であれば、SPI_prepare_cursorにCURSOR_OPT_GENERIC_PLANまたはCURSOR_OPT_CUSTOM_PLANフラグを設定することで、それぞれ一般的な計画が独自計画を強制的に利用するよう変更できます。

この関数は接続済みのプロシージャからのみ呼び出してください。

`SPIPlanPtr`は`spi.h`内で`opaque`構造体型へのポインタとして宣言されています。たいていの場合将来のバージョンのPostgreSQLでそのコードが壊れてしまうため、この内容に直接アクセスすることは避けてください。

そのデータ構造はもはや実行計画を含むとは限りませんので、`SPIPlanPtr`という名前はいくらか歴史的なものです。

[前のページ](#)

`SPI_execute_with_args`

[ホーム](#)
[上に戻る](#)

[次のページ](#)

`SPI_prepare_cursor`

SPI_prepare_cursor

名前

SPI_prepare_cursor -- 文を準備する。まだ実行は行わない

概要

```
SPIPlanPtr SPI_prepare_cursor(const char * command, int nargs,
                               Oid * argtypes, int cursorOptions)
```

説明

SPI_prepare_cursorは、プランナの"カーソルオプション"パラメータを指定できる点を除き、SPI_prepareと同じです。これはDeclareCursorStmtのoptionsフィールド用にnodes/parsenodes.hで示された値を持つビットマスクです。SPI_prepareでは常にカーソルオプションをゼロとして扱います。

引数

const char * command

コマンド文字列

int nargs

(\$1、\$2など) 入力パラメータ数

Oid * argtypes

パラメータのデータ型のOIDを持つ配列へのポインタ

int cursorOptions

カーソルオプションの整数ビットマスク。ゼロはデフォルトの動作を引き起こします

戻り値

SPI_prepare_cursorはSPI_prepareと同じ戻り値の規則を持ちます。

注釈

cursorOptionsに指定できるビットに

は、CURSOR_OPT_SCROLL、CURSOR_OPT_NO_SCROLL、CURSOR_OPT_FAST_PLAN、CURSOR_OPT_GENERIC_PLAN、CURSOR_OPT_CUSTOM_PLANがあります。特にCURSOR_OPT_HOLDは無視される点に注意してください。

SPI_prepare_params

名前

SPI_prepare_params -- 文を準備する。まだ実行は行わない

概要

```
SPIPlanPtr SPI_prepare_params(const char * command,
                              ParserSetupHook parserSetup,
                              void * parserSetupArg,
                              int cursorOptions)
```

説明

SPI_prepare_paramsは指定したコマンドの準備済み文を作成し返します。しかしそのコマンドを実行しません。この関数はSPI_prepare_cursorと同じですが、呼び出し元が外部パラメータ参照の解析を制御するパーサフック関数を指定できる点が追加されています。

引数

const char * command

コマンド文字列

ParserSetupHook parserSetup

パーサフック設定関数

void * parserSetupArg

parserSetupに渡される引数

int cursorOptions

カーソルオプションの整数ビットマスク。ゼロはデフォルトの動作を引き起こします

戻り値

SPI_prepare_paramsはSPI_prepareと同じ戻り値の規則を持ちます。

[前のページ](#)[SPI_prepare_cursor](#)[ホーム](#)
[上に戻る](#)[次のページ](#)[SPI_getargcount](#)

SPI_getargcount

名前

SPI_getargcount -- SPI_prepareにより準備した文に必要とされる引数の数を返す

概要

```
int SPI_getargcount(SPIPlanPtr plan)
```

説明

SPI_getargcount は、SPI_prepareにより準備された文を実行する時に必要とされる引数の数を返します。

引数

SPIPlanPtr plan

(SPI_prepareで返される) 準備済み文です。

戻り値

planで想定される引数の数です。 planがNULLまたは無効な場合はSPI_resultにSPI_ERROR_ARGUMENTが設定され、-1が返されます。

SPI_getargtypeid

名前

SPI_getargtypeid -- SPI_prepare で準備された文で指定される引数のデータ型のOIDを返す

概要

```
Oid SPI_getargtypeid(SPIPlanPtr plan, int argIndex)
```

説明

SPI_getargtypeid は、SPI_prepare で準備された文における argIndex 番目の引数の型を表すOIDを返します。 インデックス0は最初の引数を示します。

引数

SPIPlanPtr plan

(SPI_prepare で返される) 準備済み文

int argIndex

0から始まる引数のインデックス

戻り値

指定したインデックスにおける引数の型OIDです。 plan が NULL または無効、あるいは argIndex が 0 未満、plan で宣言された引数の数以上の場合、SPI_result に SPI_ERROR_ARGUMENT が設定され、InvalidOid が返されます。

SPI_is_cursor_plan

名前

SPI_is_cursor_plan -- SPI_prepareで準備された文がSPI_cursor_openで使える場合にtrueを返す

概要

```
bool SPI_is_cursor_plan(SPIPlanPtr plan)
```

説明

SPI_prepareで準備済み文がSPI_cursor_openへの引数として渡すことができる場合、SPI_is_cursor_planはtrueを返します。渡すことができない場合はfalseを返します。この基準は、planが単一のコマンドであり、かつ、そのコマンドが呼び出し元にタプルを返すことです。例えば、INTO句を含んでいないSELECTは可能です。そして、RETURNING句を含む場合のみUPDATEも可能です。

引数

SPIPlanPtr plan

(SPI_prepareで返される) 準備済み文

戻り値

planがカーソルを生成することができるかどうかを示すtrueもしくはfalseです。そしてSPI_resultをゼロに設定します。解答を決定することができない場合（例えばplanがNULL、または無効な場合、もしくはSPI未接続時に呼び出された場合）はSPI_resultに適切なエラーコードが設定され、falseが返されます。

SPI_execute_plan

名前

SPI_execute_plan -- SPI_prepare で準備された文を実行する

概要

```
int SPI_execute_plan(SPIPlanPtr plan, Datum * values, const char * nulls,
                    bool read_only, long count)
```

説明

SPI_execute_plan は、SPI_prepare もしくは類似の関数で準備された文を実行します。 read_only と count は SPI_execute と同様の解釈がなされます。

引数

SPIPlanPtr plan

(SPI_prepare で返される) 準備済み文

Datum * values

実パラメータ値の配列。 文の引数の数と同じ長さでなければなりません

const char * nulls

どのパラメータが NULL であるかを示す配列。 文の引数の数と同じ長さでなければなりません。 n は NULL 値を表します (values 内の項目は無視されます)。 空白は非 NULL の値を表します (values 内の項目は有効です)。

nulls が NULL の場合、SPI_execute_plan は全てのパラメータが NULL ではないとみなします。

bool read_only

読み取りのみの実行の場合 true

long count

返される行の最大数。無制限なら 0。

戻り値

戻り値は、SPI_execute 同様のものに加え、以下のエラー（負）の結果を取ることがあります。

SPI_ERROR_ARGUMENT

plan が NULL または無効、あるいは、count が 0 未満の場合

SPI_ERROR_PARAM

values が NULL、かつ、plan がパラメータ付きで準備された場合

成功時、SPI_processed と SPI_tuptable が SPI_execute と同様に設定されます。

SPI_execute_plan_with_paramlist

名前

SPI_execute_plan_with_paramlist -- SPI_prepare で準備された文を実行する

概要

```
int SPI_execute_plan_with_paramlist(SPIPlanPtr plan,
                                     ParamListInfo params,
                                     bool read_only,
                                     long count)
```

説明

SPI_execute_plan_with_paramlist は SPI_prepare で準備された文を実行します。この関数は SPI_execute_plan と同じですが、問い合わせに渡されるパラメータ値に関する情報が別途存在する点が異なります。ParamListInfo 表現は、すでに利用可能な形式で値を渡すために便利です。また ParamListInfo で指定されたフック関数経由での動的なパラメータ群の使用をサポートします。

引数

SPIPlanPtr plan

(SPI_prepare で返される) 準備済み文

ParamListInfo params

パラメータの型と値からなるデータ構造。なければヌル。

bool read_only

読み取りのみの実行の場合 true

long count

返される行の最大数。無制限なら 0。

戻り値

戻り値は SPI_execute_plan と同じです。

成功時、SPI_processed と SPI_tuptable が SPI_execute_plan と同様に設定されます。

SPI_execp

名前

SPI_execp -- 読み書きモードで文を実行する

概要

```
int SPI_execp(SPIPlanPtr plan, Datum * values, const char * nulls, long count)
```

説明

SPI_execp は、常に read_only パラメータを false とした SPI_execute_plan と同じです。

引数

SPIPlanPtr plan

(SPI_prepare で返される) 準備済み文

Datum * values

実パラメータ値の配列。 文の引数の数と同じ長さでなければなりません。

const char * nulls

どのパラメータが NULL であるかを示す配列。 文の引数の数と同じ長さでなければなりません。 n は NULL 値を表します (values 内の項目は無視されます)。 空白は非 NULL の値を表します (values 内の項目は有効です)。

nulls が NULL の場合、SPI_execp は全てのパラメータが NULL ではないとみなします。

long count

返される行の最大数。無制限なら 0。

戻り値

SPI_execute_plan を参照してください。

成功時、SPI_execute 同様に SPI_processed と SPI_tuptable が設定されます。

SPI_cursor_open

名前

SPI_cursor_open -- SPI_prepareで作成された文を使用したカーソルを設定する

概要

```
Portal SPI_cursor_open(const char * name, SPIPlanPtr plan,
                        Datum * values, const char * nulls,
                        bool read_only)
```

説明

SPI_cursor_openは、SPI_prepareによって準備された文を実行するカーソル（内部的にはポータル）を設定します。このパラメータはSPI_execute_planの対応するパラメータと同じ意味を持ちます。

文を直接実行するのではなくカーソルを使用することには2つの利点があります。1つ目は、結果行を一度に少なく取り出し、多くの行を返す問い合わせでのメモリの過使用を防ぐことができる点です。2つ目は、ポータルは現在のプロシージャの外部でも有効である点です（実際、現在のトランザクションの終端まで有効とすることができます）。プロシージャの呼び出し元にポータルの名前を返すことで、結果として行セットを返す手段を提供します。

渡されるパラメータデータはカーソルのポータルにコピーされます。そのため、カーソルが存在している間にそのデータを解放することができます。

引数

const char * name

ポータルの名前、あるいはシステムに名前を決定させる場合はNULL

SPIPlanPtr plan

（SPI_prepareで返される）準備済み文

Datum * values

実パラメータ値の配列。文の引数の数と同じ長さでなければなりません。

const char * nulls

どのパラメータがNULLであるかを示す配列。文の引数の数と同じ長さでなければなりません。nはNULL値を表します（values内の項目は無視されます）。空白は非NULLの値を表します（values内の項目は有効です）。

nullsがNULLの場合、SPI_curcor_openは全てのパラメータがNULLではないとみなします。

bool read_only

読み取りのみの実行の場合true

戻り値

カーソルを含むポータルへのポインタ。戻り値の規約にはエラーを表すものがないことに注意してください。エラーはすべてelog経由で報告されます。

SPI_cursor_open_with_args

名前

SPI_cursor_open_with_args -- 問い合わせとパラメータを使ってカーソルを設定する

概要

```
Portal SPI_cursor_open_with_args(const char *name,
                                  const char *command,
                                  int nargs, Oid *argtypes,
                                  Datum *values, const char *nulls,
                                  bool read_only, int cursorOptions)
```

説明

SPI_cursor_open_with_argsは特定の問い合わせを実行するカーソル（内部的にはポータル）を設定します。ほとんどのパラメータはSPI_prepare_cursorとSPI_cursor_openに対応するパラメータと同じ意味を持っています。

1 回限りの問い合わせ実行に対しては、後にSPI_cursor_openが続いたSPI_prepare_cursorよりも、この関数を選ぶべきです。多くの異なったパラメータを持つ同一のコマンドを実行する場合、再計画のコストと独自計画による利益に依存して、どちらか一方の方法がより早くなります。

渡されたパラメータデータはカーソルのポータルにコピーされますので、カーソルが存在している間は解放することができません。

引数

const char * name

ポータルの名前、またはシステムに名前を選択させるNULL

const char * command

コマンド文字列

int nargs

（\$1、\$2など）入力パラメータの数

Oid * argtypes

パラメータのデータ型のOIDを含む配列

Datum * values

実パラメータ値の配列

const char * nulls

どのパラメータがnullかを記述する配列

nullsがNULLであれば、SPI_cursor_open_with_argsはどのパラメータもnullでないとみなします。

bool read_only

読み取りのみの実行の場合true

int cursorOptions

カーソルオプションの整数ビットマスク。ゼロの場合はデフォルトの動作

戻り値

カーソルを含んだポータルへのポインタ。 エラーを返す規約がないことに注意してください。 すべてのエラーはe_{log}で報告されます。

[前のページ](#)

SPI_cursor_open

[ホーム](#)

[上に戻る](#)

[次のページ](#)

SPI_cursor_open_with_paramlist

SPI_cursor_open_with_paramlist

名前

SPI_cursor_open_with_paramlist -- パラメータを使ってカーソルを設定する

概要

```
Portal SPI_cursor_open_with_paramlist(const char *name,
                                       SPIPlanPtr plan,
                                       ParamListInfo params,
                                       bool read_only)
```

説明

SPI_cursor_open_with_paramlistはSPI_prepareで準備された文を実行するカーソル(内部的にはポータル)を設定します。この関数はSPI_cursor_openと同じですが、問い合わせに渡されるパラメータ値に関する情報が別途存在することが異なります。ParamListInfo表現は、すでに利用可能な形式で値を渡すために便利です。またParamListInfoで指定されたフック関数経由での動的なパラメータ群の使用をサポートします。

渡されるパラメータデータはカーソルのポータルにコピーされます。そのため、カーソルが存在している間にそのデータを解放することができます。

引数

const char * name

ポータルの名前、あるいはシステムに名前を決定させる場合はNULL

SPIPlanPtr plan

(SPI_prepareで返される) 準備済み文

ParamListInfo params

パラメータの型と値からなるデータ構造。 なければヌル。

bool read_only

読み取りのみの実行の場合true

戻り値

カーソルを含むポータルへのポインタ。 戻り値の規約にはエラーを表すものがないことに注意してください。 エラーはすべてelog経由で報告されます。

SPI_cursor_find

名前

SPI_cursor_find -- 既存のカーソルを名前で検索する

概要

```
Portal SPI_cursor_find(const char * name)
```

説明

`SPI_cursor_find` は既存のカーソルを名前で検索します。これは主に、他の何らかの関数でテキストとして返されたカーソル名
の名前解決の際に使用されます。

引数

```
const char * name
```

ポータルの名前

戻り値

指定された名前のポータルへのポインタ。見つからない場合は `NULL` です。

SPI_cursor_fetch

名前

SPI_cursor_fetch -- カーソルから数行を取り出す

概要

```
void SPI_cursor_fetch(Portal portal, bool forward, long count)
```

説明

SPI_cursor_fetchはカーソルから数行を取り出します。これは、FETCH SQLコマンドと部分的に等価です。（詳細機能についてはSPI_scroll_cursor_fetchを参照してください。）

引数

Portal portal

カーソルを持つポータル

bool forward

前方方向の取り出しの場合、真。後方方向の場合は偽。

long count

取り出す最大行数。

戻り値

成功時、SPI_processedとSPI_tuptableがSPI_execute同様に設定されます。

注釈

カーソルの計画がCURSOR_OPT_SCROLLオプションを付けて作成されなかった場合、後方方向の取り出しは失敗する可能性があります。

SPI_cursor_move

名前

SPI_cursor_move -- カーソルを移動する

概要

```
void SPI_cursor_move(Portal portal, bool forward, long count)
```

説明

SPI_cursor_move はカーソル内で、数行を飛ばします。これは`MOVE` SQLコマンドと部分的に等価です。（詳細機能についてはSPI_scroll_cursor_moveを参照してください。）

引数

Portal portal

カーソルを持つポータル

bool forward

前方方向の移動の場合、真。後方方向の場合は偽。

long count

移動する最大行数。

注釈

カーソルの計画がCURSOR_OPT_SCROLLオプション付きで作成されなかった場合、後方方向への移動は失敗する可能性があります。

[前のページ](#)

SPI_cursor_fetch

[ホーム](#)
[上に戻る](#)[次のページ](#)

SPI_scroll_cursor_fetch

SPI_scroll_cursor_fetch

名前

SPI_scroll_cursor_fetch -- カーソルから一部の行を取り出す。

概要

```
void SPI_scroll_cursor_fetch(Portal portal, FetchDirection direction,
                             long count)
```

説明

SPI_scroll_cursor_fetchはカーソルから行の一部を取り出します。 これはSQLコマンド`FETCH`と等価です。

引数

Portal portal

カーソルを含むポータル

FetchDirection direction

`FETCH_FORWARD`、`FETCH_BACKWARD`、`FETCH_ABSOLUTE`、`FETCH_RELATIVE`のいずれか

long count

`FETCH_FORWARD`または`FETCH_BACKWARD`では取り出す行数。 `FETCH_ABSOLUTE`では取り出す行の絶対番号。
`FETCH_RELATIVE`では取り出す行の相対的番号。

戻り値

成功時、SPI_execute同様にSPI_processedとSPI_tuptableが設定されます。

注釈

directionパラメータおよびcountパラメータの解釈の詳細についてはSQL [FETCH](#)コマンドを参照してください。

カーソルの計画がCURSOR_OPT_SCROLLオプション付きで作成されていない場合、`FETCH_FORWARD`以外の方向値は失敗する可能性があります。

SPI_scroll_cursor_move

名前

SPI_scroll_cursor_move -- カーソルを移動する

概要

```
void SPI_scroll_cursor_move(Portal portal, FetchDirection direction,
                             long count)
```

説明

SPI_scroll_cursor_move はカーソル内の行の一部を飛ばします。これはSQLコマンド `MOVE` と等価です。

引数

Portal portal

カーソルを含むポータル

FetchDirection direction

FETCH_FORWARD、FETCH_BACKWARD、FETCH_ABSOLUTE、FETCH_RELATIVE のいずれか

long count

FETCH_FORWARD または FETCH_BACKWARD では移動する行数。 FETCH_ABSOLUTE では移動する行の絶対番号。
FETCH_RELATIVE では移動する行の相対的番号。

戻り値

成功時、SPI_execute 同様に SPI_processed が設定されます。この関数は行を返しませんので、SPI_tuptable は NULL に設定されます。

注釈

direction パラメータおよび count パラメータの解釈の詳細についてはSQL [FETCH](#) コマンドを参照してください。

カーソルの計画が CURSOR_OPT_SCROLL オプション付きで作成されていない場合、FETCH_FORWARD 以外の方向値は失敗する可能性があります。

SPI_cursor_close

名前

SPI_cursor_close -- カーソルを閉じる

概要

```
void SPI_cursor_close(Portal portal)
```

説明

SPI_cursor_close は事前に作成されたカーソルを閉じ、そのポータル用の領域を解放します。

トランザクションの終了時に全ての開いているカーソルが自動的に閉ざされます。 SPI_cursor_close は、リソースの解放をより早めに行いたい場合にのみ呼び出す必要があります。

引数

Portal portal

カーソルを持つポータル

SPI_keepplan

名前

SPI_keepplan -- 準備済み文を保持する

概要

```
int SPI_keepplan(SPIPlanPtr plan)
```

説明

SPI_keepplan は渡された (SPI_prepare で準備された) 文を SPI_finish とトランザクションマネージャで解放されないメモリ内に保存します。これは、現在のセッションにおける、その後のプロシージャの呼び出しで準備済み文を再利用できる機能を提供します。

引数

SPIPlanPtr plan

保存する準備済み文

戻り値

成功時は0。 plan が NULL または無効な場合は SPI_ERROR_ARGUMENT

注意

渡された文はポインタの調整により永続的記憶領域に再配置されます(データコピーは不要です)。後ほど削除したければ、SPI_freeplan を実行してください。

SPI_saveplan

名前

SPI_saveplan -- 準備済み文を保存する

概要

```
SPIPlanPtr SPI_saveplan(SPIPlanPtr plan)
```

説明

SPI_saveplanは渡された（SPI_prepareで準備された）文をSPI_finishとトランザクションマネージャで解放されないメモリ内にコピーします。そして、コピーした文のポインタを返します。これは、現在のセッションにおける、その後のプロシージャの呼び出しで準備済み文を再利用できる機能を提供します。

引数

SPIPlanPtr plan

保存する準備済み文

戻り値

コピーした文へのポインタ。失敗した場合はNULLです。エラー時、SPI_resultは以下のように設定されます。

SPI_ERROR_ARGUMENT

planがNULL、または無効な場合

SPI_ERROR_UNCONNECTED

未接続のプロシージャから呼び出された場合

注意

渡された元の文は解放されません。ですので、SPI_finishを行うまでのメモリリークを防ぎたいければSPI_freeplanを実行してください。

準備済み文のデータ構造を物理的にコピーする必要なく、ほとんど同じ結果をもたらしますので、たいていの場合、この関数よりもSPI_keepplanの方が好ましいです。

43.2. インタフェースサポート関数

目次

[SPI_fname](#) -- 指定した列番号に対する列名を決定する
[SPI_fnumber](#) -- 指定した列名から列番号を決定する
[SPI_getvalue](#) -- 指定された列の文字列値を返す
[SPI_getbinval](#) -- 指定した列のバイナリ値を返す
[SPI_gettype](#) -- 指定された列のデータ型名を返す
[SPI_gettypeid](#) -- 指定された列のデータ型のOIDを返す
[SPI_getrelname](#) -- 指定されたリレーシヨンの名前を返す
[SPI_getnsname](#) -- 指定されたリレーシヨンの名前空間を返す。

以下で説明する関数は、`SPI_execute`や他のSPI関数で返される結果セットから情報を取り出すためのインタフェースを提供します。

本節で説明する関数は全て、接続、未接続のプロシージャのどちらからでも使用することができます。

[前のページ](#)[SPI_saveplan](#)[ホーム](#)
[上に戻る](#)[次のページ](#)[SPI_fname](#)

SPI_fname

名前

SPI_fname -- 指定した列番号に対する列名を決定する

概要

```
char * SPI_fname(TupleDesc rowdesc, int colnumber)
```

説明

`SPI_fname` は指定した列の列名のコピーを返します（名前のコピーが不要になった場合 `pfree` を使用してその領域を解放することができます）。

引数

`TupleDesc rowdesc`

入力行の記述

`int colnumber`

（1から始まる）列番号

戻り値

列の名前です。 `colnumber` が範囲外の場合は `NULL` です。 エラー時、 `SPI_result` は `SPI_ERROR_NOATTRIBUTE` に設定されます。

SPI_fnumber

名前

SPI_fnumber -- 指定した列名から列番号を決定する

概要

```
int SPI_fnumber(TupleDesc rowdesc, const char * colname)
```

説明

SPI_fnumber は指定した名前の列の列番号を返します。

colname が (oid などの) システム列を参照する場合、適切な負の列番号が返されます。呼び出し元は、エラーを検知するために戻り値が SPI_ERROR_NOATTRIBUTE と正確に同一であるかどうかを注意して検査しなければなりません。システム列を拒絶させなければ、結果が 0 あるいは 0 未満かを検査するという方法は、正しくありません。

引数

TupleDesc rowdesc

入力行の記述

const char * colname

列名

戻り値

(1 から始まる) 列番号。 指定された名前の列が見つからなければ、SPI_ERROR_NOATTRIBUTE です。

SPI_getvalue

名前

SPI_getvalue -- 指定された列の文字列値を返す

概要

```
char * SPI_getvalue(HeapTuple row, TupleDesc rowdesc, int colnumber)
```

説明

SPI_getvalue は指定された列の値の文字列表現を返します。

結果は、palloc を使用して割り当てられたメモリ内に返されます（不要になった段階で、pfree を使用してメモリを解放することができます）。

引数

HeapTuple row

検査対象の入力行

TupleDesc rowdesc

入力行の記述

int colnumber

（1から始まる）列番号

戻り値

列の値。列がNULLの場合、あるいはcolnumberが範囲外の場合はNULLです（SPI_resultがSPI_ERROR_NOATTRIBUTEに設定されます）。利用できる出力関数が存在しない場合は、NULLです（SPI_resultがSPI_ERROR_NOOUTFUNCに設定されます）。

SPI_getbinval

名前

SPI_getbinval -- 指定した列のバイナリ値を返す

概要

```
Datum SPI_getbinval(HeapTuple row, TupleDesc rowdesc, int colnumber,
                    bool * isnull)
```

説明

SPI_getbinval は指定された列の値を内部形式で (Datum として) 返します。

この関数はデータ用に新しい領域を確保しません。 参照渡し of データ型の場合、戻り値は渡された行の内部を示すポインタとなります。

引数

HeapTuple row

対象とする入力行

TupleDesc rowdesc

入力行の記述

int colnumber

(1から始まる) 列番号

bool * isnull

列のNULL値についてのフラグ

戻り値

列のバイナリ値が返されます。 isnull で指し示される変数は、列がNULLならば真に、さもなくば、偽に設定されます。

エラー時、SPI_result は SPI_ERROR_NOATTRIBUTE に設定されます。

SPI_gettype

名前

SPI_gettype -- 指定された列のデータ型名を返す

概要

```
char * SPI_gettype(TupleDesc rowdesc, int colnumber)
```

説明

SPI_gettype は指定された列のデータ型名のコピーを返します（不要になった段階で、pfree を使用して名前のコピーを解放することができます）。

引数

TupleDesc rowdesc

int colnumber

戻り値

指定された列のデータ型名。 エラー時は NULL です。 エラー時、SPI_result は SPI_ERROR_NOATTRIBUTE に設定されます。

SPI_gettypeid

名前

SPI_gettypeid -- 指定された列のデータ型のOIDを返す

概要

```
Oid SPI_gettypeid(TupleDesc rowdesc, int colnumber)
```

説明

SPI_gettypeidは指定された列のデータ型のOIDを返します。

引数

TupleDesc rowdesc

入力行の記述

int colnumber

(1から始まる) 列番号

戻り値

指定された列のデータ型のOIDです。 エラー時はInvalidOidです。 エラー時、SPI_resultはSPI_ERROR_NOATTRIBUTEに設定されます。

SPI_getrelname

名前

SPI_getrelname -- 指定されたリレーシヨンの名前を返す

概要

```
char * SPI_getrelname(Relation rel)
```

説明

SPI_getrelnameは指定リレーシヨンの名前のコピーを返します（不要になった段階で、pfreeを使用して名前のコピーを解放することができます）。

引数

Relation rel

入力リレーシヨン

戻り値

指定されたリレーシヨンの名前です。

SPI_getnspname

名前

SPI_getnspname -- 指定されたリレーシヨンの名前空間を返す。

概要

```
char * SPI_getnspname(Relation rel)
```

説明

SPI_getnspname は、指定した Relation が属する名前空間名のコピーを返します。これはリレーシヨンのスキーマと同じです。作業終了時に、この関数の戻り値に対して pfree を行わなければなりません。

引数

Relation rel

入力リレーシヨン

戻り値

指定したリレーシヨンの名前空間の名称です。

43.3. メモリ管理

目次

[SPI_palloc](#) -- 上位エグゼキュータコンテキスト内にメモリを割り当てる
[SPI_realloc](#) -- 上位エグゼキュータコンテキスト内にメモリを再割り当てる
[SPI_pfree](#) -- 上位エグゼキュータコンテキスト内のメモリを解放する
[SPI_copytuple](#) -- 上位エグゼキュータ内に行のコピーを作成する
[SPI_returntuple](#) -- Datumとしてタプルを返す準備をする
[SPI_modifytuple](#) -- 与えられた行の選択フィールドを置き換えた行を作成する
[SPI_freetuple](#) -- 上位エグゼキュータコンテキスト内に割り当てられた行を解放する
[SPI_freetable](#) -- [SPI_execute](#)や類似の関数によって生成された行セットを解放する
[SPI_freeplan](#) -- 以前に保存した準備済み文を解放する

PostgreSQLは、メモリコンテキスト内にメモリを確保します。これは、様々な場所で、必要な有効期間がそれぞれ異なるような割り当てを管理する便利な方法を提供します。コンテキストを破壊することで、そこで割り当てられた全てのメモリを解放します。したがって、メモリリークを防ぐための個々のオブジェクトの追跡を維持することは不要です。その代わり、相対的に少量のコンテキストを管理する必要があります。 `palloc`と関連する関数は"現在の"コンテキストからメモリを確保します。

`SPI_connect`は新しくメモリコンテキストを作成し、それを現在のコンテキストとします。 `SPI_finish`は直前の現在のメモリコンテキストを戻し、`SPI_connect`で作成されたコンテキストを破壊します。これらの動作により、プロシージャ内で割り当てが行われる一時的なメモリがプロシージャの終了時に回収され、メモリリークが防止されることが保証されます。

しかし、(参照渡しの変数の値といった)プロシージャが割り当てられたメモリ内のオブジェクトを返す必要がある場合、少なくともSPIに接続していない期間は、`palloc`を使用してメモリを確保することができません。これを試行すると、そのオブジェクトは`SPI_finish`で解放されてしまい、プロシージャは正しく動作しないでしょう。この問題を解決するには、`SPI_palloc`を使用して、戻り値となるオブジェクト用のメモリを確保してください。 `SPI_palloc`は"上位エグゼキュータコンテキスト"内にメモリを割り当てます。このメモリコンテキストは、`SPI_connect`が呼び出された時点において現在のコンテキストだったものであり、プロシージャの戻り値用のコンテキストとしてまさに正しいものです。

プロシージャがSPIに未接続な状態で`SPI_palloc`が呼び出されると、普通の`palloc`と同じように動作します。プロシージャがSPIマネージャに接続する前では、現在のメモリコンテキストは上位エグゼキュータコンテキストであり、そのため、プロシージャから`palloc`やSPIユーティリティ関数経由で割り当てられた全ては、このコンテキストで作成されます。

`SPI_connect`が呼び出されると、`SPI_connect`によって作成されるプロシージャ固有のコンテキストが現在のコンテキストに作成されます。 `palloc`、`realloc`、SPIユーティリティ関数 (`SPI_copytuple`、`SPI_returntuple`、`SPI_modifytuple`、`SPI_palloc`は除きます) によって作成される割り当ては全て、このコンテキスト内に作成されます。プロシージャがSPIマネージャから (`SPI_finish`経由で) 切断した時、現在のコンテキストは上位エグゼキュータコンテキストに戻され、プロシージャメモリコンテキスト内で割り当てられたメモリは全て解放され、二度と使用することができません。

本節で説明する関数は全て、接続状態、未接続状態のどちらのプロシージャから使用することができます。未接続のプロシージャでは、背後にある通常のサーバ関数 (`palloc`など) と同様に動作します。

SPI_palloc

名前

SPI_palloc -- 上位エグゼキュータコンテキスト内にメモリを割り当てる

概要

```
void * SPI_palloc(Size size)
```

説明

SPI_palloc は上位エグゼキュータコンテキスト内にメモリを割り当てます。

引数

Size size

割り当てる領域のバイト数

戻り値

指定サイズの新しい格納領域へのポインタ

SPI_realloc

名前

SPI_realloc -- 上位エグゼキュータコンテキスト内にメモリを再割り当てる

概要

```
void * SPI_realloc(void * pointer, Size size)
```

説明

SPI_reallocは、以前にSPI_mallocを使用して割り当てられたメモリセグメントのサイズを変更します。

この関数はもはや通常のreallocとは異なるものではありません。単に既存コードの後方互換性のために保持されています。

引数

void * pointer

変更する既存の領域へのポインタ

Size size

割り当てる領域のバイト数

戻り値

指定サイズに新規に割り当てられ、既存領域の内容をコピーした領域へのポインタ

[前のページ](#)
SPI_malloc

[ホーム](#)
[上に戻る](#)

[次のページ](#)
SPI_pfree

SPI_pfree

名前

SPI_pfree -- 上位エグゼキュータコンテキスト内のメモリを解放する

概要

```
void SPI_pfree(void * pointer)
```

説明

SPI_pfreeは、以前にSPI_pallocやSPI_repalloccを使用して割り当てられたメモリを解放します。

この関数はもはや通常のpfreeとは異なるものではありません。単に既存コードの後方互換性のために保持されています。

引数

```
void * pointer
```

解放する既存の領域へのポインタ

SPI_copytuple

名前

SPI_copytuple -- 上位エグゼキュータ内に行のコピーを作成する

概要

```
HeapTuple SPI_copytuple(HeapTuple row)
```

説明

`SPI_copytuple`は上位エグゼキュータコンテキスト内に行のコピーを作成します。これは通常、トリガから変更した行を返す時に使用されます。複合型を返すものと宣言された関数では、代わりに`SPI_returntuple`を使用してください。

引数

HeapTuple row

コピーされる行

戻り値

コピーされた行。 tupleがNULLの場合のみNULLです。

SPI_returntuple

名前

SPI_returntuple -- Datumとしてタプルを返す準備をする

概要

```
HeapTupleHeader SPI_returntuple(HeapTuple row, TupleDesc rowdesc)
```

説明

`SPI_returntuple` は上位エグゼキュータコンテキスト内に行の複製を作成し、それを行型の `Datum` 形式で返します。返されるポインタは、返す前に `PointerGetDatum` を使用して `Datum` に変換することのみが必要です。

この関数は複合型を返すものと宣言された関数に対して使用しなければなりません。トリガでは使用されません。トリガで変更された行を返すには `SPI_copytuple` を使用してください。

引数

`HeapTuple row`

コピーされる行

`TupleDesc rowdesc`

行の記述子（最も効率的にキャッシュを行えるように毎回同一の記述子を渡してください）

戻り値

コピーされた行を指し示す `HeapTupleHeader` です。 `row` もしくは `rowdesc` が `NULL` の場合のみ `NULL` が返ります。

SPI_modifytuple

名前

SPI_modifytuple -- 与えられた行の選択フィールドを置き換えた行を作成する

概要

```
HeapTuple SPI_modifytuple(Relation rel, HeapTuple row, int ncols,
                          int * colnum, Datum * values, const char * nulls)
```

説明

SPI_modifytuple は、選択された列は新しい値で置き換え、その他の位置は元の行の列をコピーした、新しい行を作成します。入力行は変更されません。

引数

Relation rel

行の行記述子のソースとしてのみ使用されます。（行記述子ではなくリレーションを渡すことは設計ミスです。）

HeapTuple row

変更される行

int ncols

colnum 配列の列番号の数

int * colnum

変更される列番号の配列（列番号は1から始まります）

Datum * values

指定された列の新しい値

const char * Nulls

もしあれば、どの値がNULLかを表します（書式についてはSPI_execute_planを参照してください）。

戻り値

変更された新しい行。上位エクゼキュータコンテキストに割り当てられます。rowがNULLの場合のみNULLです。

エラー時、SPI_result が以下のように設定されます。

SPI_ERROR_ARGUMENT

relがNULLの場合、rowがNULLの場合、ncolsが0以下の場合、colnumがNULLの場合、valuesがNULLの場合。

SPI_ERROR_NOATTRIBUTE

colnumが無効な列番号を持つ場合（0以下、rowの列数以上）。

SPI_freetuple

名前

SPI_freetuple -- 上位エクゼキュータコンテキスト内に割り当てられた行を解放する

概要

```
void SPI_freetuple(HeapTuple row)
```

説明

SPI_freetupleは以前に上位エクゼキュータコンテキスト内に割り当てられた行を解放します。

この関数はもはや通常のheap_freetupleとは異なるものではありません。 単に既存コードの後方互換性のために保持されています。

引数

HeapTuple row

解放する行

SPI_freetuptable

名前

SPI_freetuptable -- SPI_execute や類似の関数によって生成された行セットを解放する

概要

```
void SPI_freetuptable(SPITupleTable * tuptable)
```

説明

SPI_freetuptable は、以前に SPI_execute などの SPI コマンド実行関数によって作成された行セットを解放します。 そのため、この関数は通常 SPI_tupletable グローバル変数を引数として呼び出されます。

この関数は SPI プロシージャが複数のコマンドを実行する必要があり、かつ、初期のコマンドの結果を終わりまで保持したくない場合に有用です。 解放されない行セットは、SPI_finish 時に全て解放されることに注意してください。

引数

```
SPITupleTable * tuptable
```

解放する行セットへのポインタ

SPI_freeplan

名前

SPI_freeplan -- 以前に保存した準備済み文を解放する

概要

```
int SPI_freeplan(SPIPlanPtr plan)
```

説明

SPI_freeplan は、以前に SPI_prepare から返された、あるいは SPI_keepplan や SPI_saveplan で保存された準備済み文を解放します。

引数

SPIPlanPtr plan

解放する文へのポインタ

戻り値

成功時は 0。 plan が NULL または無効な場合、 SPI_ERROR_ARGUMENT です。

43.4. データ変更の可視性

SPI（や他の任意のC関数）を使用する関数内のデータの可視性は、以下の規則に従います。

- SQLコマンドの実行中、そのコマンドで行われたデータ変更はそのコマンドからは不可視です。例えば、

```
INSERT INTO a SELECT * FROM a;
```

では、挿入された行はSELECT部からは不可視です。

- コマンドCで行われた変更は、Cの後に開始された全てのコマンドからは可視です。Cの内側（処理中）に開始したかCの処理後に開始したかは関係ありません。
- SQLコマンドによって呼び出される関数（普通の関数やトリガ関数）の内側で、SPIを使用して実行されるコマンドは、SPIに渡される読み書きフラグに応じて上の規則のいくつかに従います。読み取りのみモードで実行されるコマンドは、呼び出し中のコマンドの変更は不可視であるという最初の規則に従います。読み書きモードで実行されるコマンドは、今までに行われた変更はすべて可視であるという2番目の規則に従います。
- 標準の手続き言語は全て、関数の変動属性に応じてSPI読み書きモードを設定します。STABLEおよびIMMUTABLE関数のコマンドは、読み取りのみモードで行われ、VOLATILE関数のコマンドは読み書きモードで行われます。C言語関数の作者はこの規約を無視することができますが、それはほとんどの場合勧められません。

次節には、これら規則の適用についてを示す例があります。

43.5. 例

本節には、SPIを使用する非常に簡単な例があります。 `execq` プロシージャは1つ目の引数としてSQLコマンドを、2つ目の引数として行数を取り、`SPI_exec`コマンドを実行し、そのコマンドで処理された行数を返します。SPIのより複雑な例はソースツリー内の`src/test/regress/regress.c`と[spi](#)モジュールにあります。

```
#include "postgres.h"

#include "executor/spi.h"
#include "utils/builtins.h"

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

int execq(text *sql, int cnt);

int
execq(text *sql, int cnt)
{
    char *command;
    int ret;
    int proc;

    /* 与えられたテキストオブジェクトをC文字列に変換 */
    command = text_to_cstring(sql);

    SPI_connect();

    ret = SPI_exec(command, cnt);

    proc = SPI_processed;
    /*
     * 何らかの行が取り出された場合は、行をelog(INFO) を使用して表示
     */
    if (ret > 0 && SPI_tuptable != NULL)
    {
        TupleDesc tupdesc = SPI_tuptable->tupdesc;
        SPITupleTable *tuptable = SPI_tuptable;
        char buf[8192];
        int i, j;

        for (j = 0; j < proc; j++)
        {
            HeapTuple tuple = tuptable->vals[j];

            for (i = 1, buf[0] = 0; i <= tupdesc->natts; i++)
                snprintf(buf + strlen(buf), sizeof(buf) - strlen(buf), " %s",
                        SPI_getvalue(tuple, tupdesc, i),
                        (i == tupdesc->natts) ? " " : " |");
            elog(INFO, "EXECQ: %s", buf);
        }

        SPI_finish();
        pfree(command);

        return (proc);
    }
}
```

(例をよりわかりやすくするためにこの関数は呼び出し規約Version-0を使用しています。 実際のアプリケーションでは、新しいVersion-1インタフェースを使用すべきです。)

以下は、コンパイルし共有ライブラリ ([項35.9.6](#)を参照) を作成した後で、関数を宣言する方法です。

```
CREATE FUNCTION execq(text, integer) RETURNS integer
AS 'filename'
LANGUAGE C;
```

以下はセッションの例です。

```
=> SELECT execq('CREATE TABLE a (x integer)', 0);
 execq
-----
      0
(1 row)

=> INSERT INTO a VALUES (execq('INSERT INTO a VALUES (0)', 0));
```

```

INSERT 0 1
=> SELECT execq('SELECT * FROM a', 0);
INFO: EXECQ: 0      -- execqによって挿入された
INFO: EXECQ: 1      -- execqによって返され、上位のINSERTによって挿入された

execq
-----
      2
(1 row)

=> SELECT execq('INSERT INTO a SELECT x + 2 FROM a', 1);
execq
-----
      1
(1 row)

=> SELECT execq('SELECT * FROM a', 10);
INFO: EXECQ: 0
INFO: EXECQ: 1
INFO: EXECQ: 2      -- 指定された、0 + 2という1つの行のみが挿入された

execq
-----
      3              -- 10は最大値を示すのみで、3が実際の行数です。
(1 row)

=> DELETE FROM a;
DELETE 3
=> INSERT INTO a VALUES (execq('SELECT * FROM a', 0) + 1);
INSERT 0 1
=> SELECT * FROM a;
x
---
      1              -- aテーブルに行がない(0) + 1
(1 row)

=> INSERT INTO a VALUES (execq('SELECT * FROM a', 0) + 1);
INFO: EXECQ: 1
INSERT 0 1
=> SELECT * FROM a;
x
---
      1
      2              -- aテーブルに1行あり + 1
(2 rows)

--   これはデータ変更に関する可視性規則を説明します。

=> INSERT INTO a SELECT execq('SELECT * FROM a', 0) * x FROM a;
INFO: EXECQ: 1
INFO: EXECQ: 2
INFO: EXECQ: 1
INFO: EXECQ: 2
INFO: EXECQ: 2
INSERT 0 2
=> SELECT * FROM a;
x
---
      1
      2
      2              -- 2 行 * 1 (xは最初の行)
      6              -- 3 行 (2 + 挿入された 1) * 2 (2行目のx)
(4 rows)
      ^^^^^^
      別の呼び出しで execq() で可視な行

```

VI. リファレンス

このリファレンス内の項目では、対応する主題に関する信頼できる、完全な、形式の整った要約を、適切な長さで提供することが意図されています。 PostgreSQLの使用に関する詳細は、物語、チュートリアル、例形式として、本書の他の部分にて説明されています。 各リファレンスページで挙げたクロスリファレンスを参照してください。

このリファレンスは伝統的な"man"ページとしても入手できます。

目次

I. SQLコマンド

[ABORT](#) -- 現在のトランザクションをアボートする
[ALTER AGGREGATE](#) -- 集約関数定義を変更する
[ALTER COLLATION](#) -- 照合順の定義を変更する
[ALTER CONVERSION](#) -- 変換の定義を変更する
[ALTER DATABASE](#) -- データベースを変更する
[ALTER DEFAULT PRIVILEGES](#) -- デフォルトのアクセス権限を定義する
[ALTER DOMAIN](#) -- ドメイン定義を変更する
[ALTER EXTENSION](#) -- 拡張の定義を変更する
[ALTER FOREIGN DATA WRAPPER](#) -- 外部データラップの定義を変更する
[ALTER FOREIGN TABLE](#) -- 外部テーブルの定義を変更
[ALTER FUNCTION](#) -- 関数定義を変更する
[ALTER GROUP](#) -- ロールの名前またはメンバ資格を変更する
[ALTER INDEX](#) -- インデックス定義を変更する
[ALTER LANGUAGE](#) -- 手続き言語の定義を変更する
[ALTER LARGE OBJECT](#) -- ラージオブジェクトの定義を変更する
[ALTER OPERATOR](#) -- 演算子の定義を変更する
[ALTER OPERATOR CLASS](#) -- 演算子クラスの定義を変更する
[ALTER OPERATOR FAMILY](#) -- 演算子族の定義を変更する
[ALTER ROLE](#) -- データベースロールを変更する
[ALTER SCHEMA](#) -- スキーマ定義を変更する
[ALTER SEQUENCE](#) -- シーケンスジェネレータの定義を変更する
[ALTER SERVER](#) -- 外部サーバの定義を変更する
[ALTER TABLE](#) -- テーブル定義を変更する
[ALTER TABLESPACE](#) -- テーブル空間の定義を変更する
[ALTER TEXT SEARCH CONFIGURATION](#) -- テキスト検索設定の定義を変更する
[ALTER TEXT SEARCH DICTIONARY](#) -- テキスト検索辞書の定義を変更する
[ALTER TEXT SEARCH PARSER](#) -- テキスト検索パーサの定義を変更する
[ALTER TEXT SEARCH TEMPLATE](#) -- テキスト検索テンプレートの定義を変更する
[ALTER TRIGGER](#) -- トリガ定義を変更する
[ALTER TYPE](#) -- 型定義を変更する
[ALTER USER](#) -- データベースロールを変更する
[ALTER USER MAPPING](#) -- ユーザマップの定義を変更する
[ALTER VIEW](#) -- ビュー定義を変更する
[ANALYZE](#) -- データベースに関する統計を集計する
[BEGIN](#) -- トランザクションブロックを開始する
[CHECKPOINT](#) -- トランザクションログのチェックポイントを強制的に実行する
[CLOSE](#) -- カーソルを閉じる
[CLUSTER](#) -- インデックスに従ってテーブルをクラスタ化する
[COMMENT](#) -- オブジェクトのコメントを定義する、または変更する
[COMMIT](#) -- 現在のトランザクションをコミットする
[COMMIT PREPARED](#) -- 二相コミット用に事前に準備されたトランザクションをコミットする
[COPY](#) -- ファイルとテーブルの間でデータをコピーする
[CREATE AGGREGATE](#) -- 新しい集約関数を定義する
[CREATE CAST](#) -- 新しいキャストを定義する
[CREATE COLLATION](#) -- 新しい照合順序を定義する
[CREATE CONVERSION](#) -- 新しい符号化方式変換の定義
[CREATE DATABASE](#) -- 新しいデータベースを作成する
[CREATE DOMAIN](#) -- 新しいドメインを定義する
[CREATE EXTENSION](#) -- 拡張をインストールする
[CREATE FOREIGN DATA WRAPPER](#) -- 新しい外部データラップを定義する

[CREATE FOREIGN TABLE](#) -- 新しい外部テーブルを定義する
[CREATE FUNCTION](#) -- 新しい関数を定義する
[CREATE GROUP](#) -- 新しいデータベースロールを定義する
[CREATE INDEX](#) -- 新しいインデックスを定義する
[CREATE LANGUAGE](#) -- 新しい手続き言語を定義する
[CREATE OPERATOR](#) -- 新しい演算子を定義する
[CREATE OPERATOR CLASS](#) -- 新しい演算子クラスを定義する
[CREATE OPERATOR FAMILY](#) -- 新しい演算子族を定義する
[CREATE ROLE](#) -- 新しいデータベースロールを定義する
[CREATE RULE](#) -- 新しい書き換えルールを定義する
[CREATE SCHEMA](#) -- 新しいスキーマを定義する
[CREATE SEQUENCE](#) -- 新しいシーケンスジェネレータを定義する
[CREATE SERVER](#) -- 新しい外部サーバを定義する
[CREATE TABLE](#) -- 新しいテーブルを定義する
[CREATE TABLE AS](#) -- 問い合わせの結果によって新しいテーブルを定義する
[CREATE TABLESPACE](#) -- 新しいテーブル空間を定義する
[CREATE TEXT SEARCH CONFIGURATION](#) -- 新しいテキスト検索設定を定義する
[CREATE TEXT SEARCH DICTIONARY](#) -- 新しいテキスト検索辞書を定義する
[CREATE TEXT SEARCH PARSER](#) -- 新しいテキスト検索パーサを定義する
[CREATE TEXT SEARCH TEMPLATE](#) -- 新しいテキスト検索テンプレートを定義する
[CREATE TRIGGER](#) -- 新しいトリガを定義する
[CREATE TYPE](#) -- 新しいデータ型を定義する
[CREATE USER](#) -- 新しいデータベースロールを定義する
[CREATE USER MAPPING](#) -- 外部サーバのユーザマップを新しく定義します。
[CREATE VIEW](#) -- 新しいビューを定義する
[DEALLOCATE](#) -- 準備された文の割り当てを解除する
[DECLARE](#) -- カーソルを定義する
[DELETE](#) -- テーブルから行を削除する
[DISCARD](#) -- セッションの状態を破棄する
[DO](#) -- 無名コードブロックを実行します。
[DROP AGGREGATE](#) -- 集約関数を削除する
[DROP CAST](#) -- キャストを削除する
[DROP COLLATION](#) -- 照合順序を削除
[DROP CONVERSION](#) -- 変換を削除する
[DROP DATABASE](#) -- データベースを削除する
[DROP DOMAIN](#) -- ドメインを削除する
[DROP EXTENSION](#) -- 拡張を削除
[DROP FOREIGN DATA WRAPPER](#) -- 外部データラップを削除する
[DROP FOREIGN TABLE](#) -- 外部テーブルを削除
[DROP FUNCTION](#) -- 関数を削除する
[DROP GROUP](#) -- データベースロールを削除する
[DROP INDEX](#) -- インデックスの削除
[DROP LANGUAGE](#) -- 手続き言語の削除
[DROP OPERATOR](#) -- 演算子を削除する
[DROP OPERATOR CLASS](#) -- 演算子クラスを削除する
[DROP OPERATOR FAMILY](#) -- 演算子族を削除する
[DROP OWNED](#) -- データベースロールにより所有されるデータベースオブジェクトを削除します。
[DROP ROLE](#) -- データベースロールを削除する
[DROP RULE](#) -- 書き換えルールを削除する
[DROP SCHEMA](#) -- スキーマを削除する
[DROP SEQUENCE](#) -- シーケンスを削除する
[DROP SERVER](#) -- 外部サーバの記述子を削除する
[DROP TABLE](#) -- テーブルを削除する
[DROP TABLESPACE](#) -- テーブル空間を削除する
[DROP TEXT SEARCH CONFIGURATION](#) -- テキスト検索設定を削除する
[DROP TEXT SEARCH DICTIONARY](#) -- テキスト検索辞書を削除する
[DROP TEXT SEARCH PARSER](#) -- テキスト検索パーサを削除する
[DROP TEXT SEARCH TEMPLATE](#) -- テキスト検索テンプレートを削除する
[DROP TRIGGER](#) -- トリガを削除する
[DROP TYPE](#) -- データ型の削除
[DROP USER](#) -- データベースロールを削除する
[DROP USER MAPPING](#) -- 外部サーバ用のユーザマップを削除します。
[DROP VIEW](#) -- ビューを削除する
[END](#) -- 現在のトランザクションをコミットする
[EXECUTE](#) -- 準備された文を実行する

[EXPLAIN](#) -- 問い合わせ文の実行計画を表示する
[FETCH](#) -- カーソルを使用してテーブルから行を抽出する
[GRANT](#) -- アクセス権限を定義する
[INSERT](#) -- テーブルに新しい行を作成する
[LISTEN](#) -- 通知を監視する
[LOAD](#) -- 共有ライブラリファイルの読み込みを行う
[LOCK](#) -- テーブルをロックする
[MOVE](#) -- カーソルの位置を決める
[NOTIFY](#) -- 通知を生成する
[PREPARE](#) -- 実行する文を準備する
[PREPARE TRANSACTION](#) -- 二相コミット用に現在のトランザクションを準備する
[REASSIGN OWNED](#) -- あるデータベースロールにより所有されたデータベースオブジェクトの所有権を変更する
[REINDEX](#) -- インデックスを再構築する
[RELEASE SAVEPOINT](#) -- 設定済みのセーブポイントを破棄する
[RESET](#) -- 実行時パラメータの値をデフォルト値に戻す
[REVOKE](#) -- アクセス権限を取り消す
[ROLLBACK](#) -- 現在のトランザクションをアボートする
[ROLLBACK PREPARED](#) -- 事前に二相コミット用に準備されたトランザクションを取り消します。
[ROLLBACK TO SAVEPOINT](#) -- セーブポイントまでロールバックする
[SAVEPOINT](#) -- 現在のトランザクション内に新規にセーブポイントを定義する
[SECURITY LABEL](#) -- オブジェクトに適用するセキュリティラベルを定義または変更する
[SELECT](#) -- テーブルもしくはビューから行を検索する
[SELECT INTO](#) -- 問い合わせの結果からの新しいテーブルを定義する
[SET](#) -- 実行時パラメータを変更する
[SET CONSTRAINTS](#) -- 現在のトランザクションの制約検査のタイミングを設定する
[SET ROLE](#) -- 現在のセッションにおける現在のユーザ識別子を設定する
[SET SESSION AUTHORIZATION](#) -- セッションのユーザ識別子、現在のセッションの現在のユーザ識別子を設定する
[SET TRANSACTION](#) -- 現在のトランザクションの特性を設定する
[SHOW](#) -- 実行時パラメータの値を表示する
[START TRANSACTION](#) -- トランザクションブロックを開始する
[TRUNCATE](#) -- 1テーブルまたはテーブル群を空にする
[UNLISTEN](#) -- 通知の監視を停止する
[UPDATE](#) -- テーブルの行を更新する
[VACUUM](#) -- データベースの不要領域の回収とデータベースの解析（オプション）を行う
[VALUES](#) -- 行セットを計算します

II. [PostgreSQLクライアントアプリケーション](#)

[clusterdb](#) -- PostgreSQLデータベースをクラスタ化する
[createdb](#) -- 新しいPostgreSQLデータベースを作成する
[createlang](#) -- PostgreSQL手続き言語をインストールする
[createuser](#) -- 新しいPostgreSQLユーザアカウントを定義する
[dropdb](#) -- PostgreSQLデータベースを削除する
[droplang](#) -- PostgreSQL手続き言語を削除する
[dropuser](#) -- PostgreSQLユーザアカウントを削除する
[ecpg](#) -- 埋め込みSQL用Cプリプロセッサを使用する
[pg_basebackup](#) -- PostgreSQLクラスタのベースバックアップを取得
[pg_config](#) -- インストールしたPostgreSQLバージョンの情報を提供する
[pg_dump](#) -- PostgreSQLデータベースをスクリプトファイルまたは他のアーカイブファイルへ抽出する
[pg_dumpall](#) -- PostgreSQLデータベースクラスタをスクリプトファイルへ抽出する
[pg_receivexlog](#) -- PostgreSQLクラスタからトランザクションログをストリームする
[pg_restore](#) -- pg_dumpによって作成されたアーカイブファイルからPostgreSQLデータベースをリストアする
[psql](#) -- PostgreSQL対話的ターミナル
[reindexdb](#) -- PostgreSQLデータベースのインデックスを再作成します。
[vacuumdb](#) -- PostgreSQLデータベースの不要領域の回収と解析を行う

III. [PostgreSQLサーバアプリケーション](#)

[initdb](#) -- PostgreSQLのデータベースクラスタを新しく作成する
[pg_controldata](#) -- PostgreSQLデータベースクラスタの制御情報を表示する
[pg_ctl](#) -- PostgreSQLサーバの初期化、起動、停止、制御
[pg_resetxlog](#) -- PostgreSQLデータベースクラスタの先行書き込みログやその他の制御情報を初期化する
[postgres](#) -- PostgreSQLデータベースサーバ
[postmaster](#) -- PostgreSQLデータベースサーバ

I. SQLコマンド

ここではPostgreSQLでサポートされるSQLコマンドのリファレンス情報があります。"SQL"ごとに、その言語の一般的な目的、標準に従う情報、各コマンドの互換性が対応するリファレンスページからわかります。

目次

[ABORT](#) -- 現在のトランザクションをアボートする
[ALTER AGGREGATE](#) -- 集約関数定義を変更する
[ALTER COLLATION](#) -- 照合順の定義を変更する
[ALTER CONVERSION](#) -- 変換の定義を変更する
[ALTER DATABASE](#) -- データベースを変更する
[ALTER DEFAULT PRIVILEGES](#) -- デフォルトのアクセス権限を定義する
[ALTER DOMAIN](#) -- ドメイン定義を変更する
[ALTER EXTENSION](#) -- 拡張の定義を変更する
[ALTER FOREIGN DATA WRAPPER](#) -- 外部データラップの定義を変更する
[ALTER FOREIGN TABLE](#) -- 外部テーブルの定義を変更
[ALTER FUNCTION](#) -- 関数定義を変更する
[ALTER GROUP](#) -- ロールの名前またはメンバ資格を変更する
[ALTER INDEX](#) -- インデックス定義を変更する
[ALTER LANGUAGE](#) -- 手続き言語の定義を変更する
[ALTER LARGE OBJECT](#) -- ラージオブジェクトの定義を変更する
[ALTER OPERATOR](#) -- 演算子の定義を変更する
[ALTER OPERATOR CLASS](#) -- 演算子クラスの定義を変更する
[ALTER OPERATOR FAMILY](#) -- 演算子族の定義を変更する
[ALTER ROLE](#) -- データベースロールを変更する
[ALTER SCHEMA](#) -- スキーマ定義を変更する
[ALTER SEQUENCE](#) -- シーケンスジェネレータの定義を変更する
[ALTER SERVER](#) -- 外部サーバの定義を変更する
[ALTER TABLE](#) -- テーブル定義を変更する
[ALTER TABLESPACE](#) -- テーブル空間の定義を変更する
[ALTER TEXT SEARCH CONFIGURATION](#) -- テキスト検索設定の定義を変更する
[ALTER TEXT SEARCH DICTIONARY](#) -- テキスト検索辞書の定義を変更する
[ALTER TEXT SEARCH PARSER](#) -- テキスト検索パーサの定義を変更する
[ALTER TEXT SEARCH TEMPLATE](#) -- テキスト検索テンプレートの定義を変更する
[ALTER TRIGGER](#) -- トリガ定義を変更する
[ALTER TYPE](#) -- 型定義を変更する
[ALTER USER](#) -- データベースロールを変更する
[ALTER USER MAPPING](#) -- ユーザマップの定義を変更する
[ALTER VIEW](#) -- ビュー定義を変更する
[ANALYZE](#) -- データベースに関する統計を集計する
[BEGIN](#) -- トランザクションブロックを開始する
[CHECKPOINT](#) -- トランザクションログのチェックポイントを強制的に実行する
[CLOSE](#) -- カーソルを閉じる
[CLUSTER](#) -- インデックスに従ってテーブルをクラスタ化する
[COMMENT](#) -- オブジェクトのコメントを定義する、または変更する
[COMMIT](#) -- 現在のトランザクションをコミットする
[COMMIT PREPARED](#) -- 二相コミット用に事前に準備されたトランザクションをコミットする
[COPY](#) -- ファイルとテーブルの間でデータをコピーする
[CREATE AGGREGATE](#) -- 新しい集約関数を定義する
[CREATE CAST](#) -- 新しいキャストを定義する
[CREATE COLLATION](#) -- 新しい照合順序を定義する
[CREATE CONVERSION](#) -- 新しい符号化方式変換の定義
[CREATE DATABASE](#) -- 新しいデータベースを作成する
[CREATE DOMAIN](#) -- 新しいドメインを定義する
[CREATE EXTENSION](#) -- 拡張をインストールする
[CREATE FOREIGN DATA WRAPPER](#) -- 新しい外部データラップを定義する
[CREATE FOREIGN TABLE](#) -- 新しい外部テーブルを定義する
[CREATE FUNCTION](#) -- 新しい関数を定義する
[CREATE GROUP](#) -- 新しいデータベースロールを定義する
[CREATE INDEX](#) -- 新しいインデックスを定義する
[CREATE LANGUAGE](#) -- 新しい手続き言語を定義する

[CREATE OPERATOR](#) -- 新しい演算子を定義する
[CREATE OPERATOR CLASS](#) -- 新しい演算子クラスを定義する
[CREATE OPERATOR FAMILY](#) -- 新しい演算子族を定義する
[CREATE ROLE](#) -- 新しいデータベースロールを定義する
[CREATE RULE](#) -- 新しい書き換えルールを定義する
[CREATE SCHEMA](#) -- 新しいスキーマを定義する
[CREATE SEQUENCE](#) -- 新しいシーケンスジェネレータを定義する
[CREATE SERVER](#) -- 新しい外部サーバを定義する
[CREATE TABLE](#) -- 新しいテーブルを定義する
[CREATE TABLE AS](#) -- 問い合わせの結果によって新しいテーブルを定義する
[CREATE TABLESPACE](#) -- 新しいテーブル空間を定義する
[CREATE TEXT SEARCH CONFIGURATION](#) -- 新しいテキスト検索設定を定義する
[CREATE TEXT SEARCH DICTIONARY](#) -- 新しいテキスト検索辞書を定義する
[CREATE TEXT SEARCH PARSER](#) -- 新しいテキスト検索パーサを定義する
[CREATE TEXT SEARCH TEMPLATE](#) -- 新しいテキスト検索テンプレートを定義する
[CREATE TRIGGER](#) -- 新しいトリガを定義する
[CREATE TYPE](#) -- 新しいデータ型を定義する
[CREATE USER](#) -- 新しいデータベースロールを定義する
[CREATE USER MAPPING](#) -- 外部サーバのユーザマップを新しく定義します。
[CREATE VIEW](#) -- 新しいビューを定義する
[DEALLOCATE](#) -- 準備された文の割り当てを解除する
[DECLARE](#) -- カーソルを定義する
[DELETE](#) -- テーブルから行を削除する
[DISCARD](#) -- セッションの状態を破棄する
[DO](#) -- 無名コードブロックを実行します。
[DROP AGGREGATE](#) -- 集約関数を削除する
[DROP CAST](#) -- キャストを削除する
[DROP COLLATION](#) -- 照合順序を削除
[DROP CONVERSION](#) -- 変換を削除する
[DROP DATABASE](#) -- データベースを削除する
[DROP DOMAIN](#) -- ドメインを削除する
[DROP EXTENSION](#) -- 拡張を削除
[DROP FOREIGN DATA WRAPPER](#) -- 外部データラップを削除する
[DROP FOREIGN TABLE](#) -- 外部テーブルを削除
[DROP FUNCTION](#) -- 関数を削除する
[DROP GROUP](#) -- データベースロールを削除する
[DROP INDEX](#) -- インデックスの削除
[DROP LANGUAGE](#) -- 手続き言語の削除
[DROP OPERATOR](#) -- 演算子を削除する
[DROP OPERATOR CLASS](#) -- 演算子クラスを削除する
[DROP OPERATOR FAMILY](#) -- 演算子族を削除する
[DROP OWNED](#) -- データベースロールにより所有されるデータベースオブジェクトを削除します。
[DROP ROLE](#) -- データベースロールを削除する
[DROP RULE](#) -- 書き換えルールを削除する
[DROP SCHEMA](#) -- スキーマを削除する
[DROP SEQUENCE](#) -- シーケンスを削除する
[DROP SERVER](#) -- 外部サーバの記述子を削除する
[DROP TABLE](#) -- テーブルを削除する
[DROP TABLESPACE](#) -- テーブル空間を削除する
[DROP TEXT SEARCH CONFIGURATION](#) -- テキスト検索設定を削除する
[DROP TEXT SEARCH DICTIONARY](#) -- テキスト検索辞書を削除する
[DROP TEXT SEARCH PARSER](#) -- テキスト検索パーサを削除する
[DROP TEXT SEARCH TEMPLATE](#) -- テキスト検索テンプレートを削除する
[DROP TRIGGER](#) -- トリガを削除する
[DROP TYPE](#) -- データ型の削除
[DROP USER](#) -- データベースロールを削除する
[DROP USER MAPPING](#) -- 外部サーバ用のユーザマップを削除します。
[DROP VIEW](#) -- ビューを削除する
[END](#) -- 現在のトランザクションをコミットする
[EXECUTE](#) -- 準備された文を実行する
[EXPLAIN](#) -- 問い合わせ文の実行計画を表示する
[FETCH](#) -- カーソルを使用してテーブルから行を抽出する
[GRANT](#) -- アクセス権限を定義する
[INSERT](#) -- テーブルに新しい行を作成する
[LISTEN](#) -- 通知を監視する

[LOAD](#) -- 共有ライブラリファイルの読み込みを行う
[LOCK](#) -- テーブルをロックする
[MOVE](#) -- カーソルの位置を決める
[NOTIFY](#) -- 通知を生成する
[PREPARE](#) -- 実行する文を準備する
[PREPARE TRANSACTION](#) -- 二相コミット用に現在のトランザクションを準備する
[REASSIGN OWNED](#) -- あるデータベースロールにより所有されたデータベースオブジェクトの所有権を変更する
[REINDEX](#) -- インデックスを再構築する
[RELEASE SAVEPOINT](#) -- 設定済みのセーブポイントを破棄する
[RESET](#) -- 実行時パラメータの値をデフォルト値に戻す
[REVOKE](#) -- アクセス権限を取り消す
[ROLLBACK](#) -- 現在のトランザクションをアボートする
[ROLLBACK PREPARED](#) -- 事前に二相コミット用に準備されたトランザクションを取り消します。
[ROLLBACK TO SAVEPOINT](#) -- セーブポイントまでロールバックする
[SAVEPOINT](#) -- 現在のトランザクション内に新規にセーブポイントを定義する
[SECURITY LABEL](#) -- オブジェクトに適用するセキュリティラベルを定義または変更する
[SELECT](#) -- テーブルもしくはビューから行を検索する
[SELECT INTO](#) -- 問い合わせの結果からの新しいテーブルを定義する
[SET](#) -- 実行時パラメータを変更する
[SET CONSTRAINTS](#) -- 現在のトランザクションの制約検査のタイミングを設定する
[SET ROLE](#) -- 現在のセッションにおける現在のユーザ識別子を設定する
[SET SESSION AUTHORIZATION](#) -- セッションのユーザ識別子、現在のセッションの現在のユーザ識別子を設定する
[SET TRANSACTION](#) -- 現在のトランザクションの特性を設定する
[SHOW](#) -- 実行時パラメータの値を表示する
[START TRANSACTION](#) -- トランザクションブロックを開始する
[TRUNCATE](#) -- 1テーブルまたはテーブル群を空にする
[UNLISTEN](#) -- 通知の監視を停止する
[UPDATE](#) -- テーブルの行を更新する
[VACUUM](#) -- データベースの不要領域の回収とデータベースの解析（オプション）を行う
[VALUES](#) -- 行セットを計算します

[前のページ](#)
リファレンス

[ホーム](#)
[上に戻る](#)

[次のページ](#)
ABORT

ABORT

名前

ABORT -- 現在のトランザクションをアボートする

概要

ABORT [WORK | TRANSACTION]

説明

`ABORT` は現在のトランザクションをロールバックし、そのトランザクションで行われた全ての更新を廃棄します。 このコマンドの動作は標準SQLの [ROLLBACK](#) コマンドと同一であり、歴史的な理由のためだけに存在しています。

パラメータ

WORK
TRANSACTION

省略可能なキーワードです。何も効果がありません。

注釈

トランザクションを正常に終了させる場合は [COMMIT](#) を使用してください。

トランザクションの外部で `ABORT` を発行しても害はありませんが、警告メッセージが表示されます。

例

全ての変更をアボートします。

```
ABORT;
```

互換性

このコマンドはPostgreSQLの拡張で、歴史的な理由で存在します。 `ROLLBACK` は、このコマンドと等価な標準SQLコマンドです。

関連項目

[BEGIN](#), [COMMIT](#), [ROLLBACK](#)

ALTER AGGREGATE

名前

ALTER AGGREGATE -- 集約関数定義を変更する

概要

```
ALTER AGGREGATE name ( argtype [ , ... ] ) RENAME TO new_name
ALTER AGGREGATE name ( argtype [ , ... ] ) OWNER TO new_owner
ALTER AGGREGATE name ( argtype [ , ... ] ) SET SCHEMA new_schema
```

説明

ALTER AGGREGATE は集約関数の定義を変更します。

ALTER AGGREGATE を使用するには集約関数の所有者でなければなりません。 集約関数のスキーマを変更するには、新しいスキーマにおける CREATE 権限も必要です。 所有者を変更するには、直接または間接的に新しい所有者ロールのメンバでなければなりません。 また、そのロールは集約関数のスキーマにおいて CREATE 権限を持たなければなりません。 （この制限により強制的に、所有者の変更に伴い集約関数の削除や再作成ができなくなるといった問題が起こらないようになります。 しかし、スーパーユーザはすべての集約関数の所有者を変更することができます。）

パラメータ

name

既存の集約関数の名前です（スキーマ修飾名も可）。

argtype

集約関数が演算する入力データ型です。 引数を持たない集約関数を参照するには、入力データ型のリストに*と記載してください。

new_name

新しい集約関数の名前です。

new_owner

新しい集約関数の所有者です。

new_schema

集約関数の新しいスキーマです。

例

integer 型用の myavg 集約関数の名前を my_average に変更します。

```
ALTER AGGREGATE myavg(integer) RENAME TO my_average;
```

integer 型用の myavg 集約関数の所有者を joe に変更します。

```
ALTER AGGREGATE myavg(integer) OWNER TO joe;
```

integer 型用の myavg 集約関数を myschema スキーマに移動します。

```
ALTER AGGREGATE myavg(integer) SET SCHEMA myschema;
```

互換性

標準SQLにはALTER AGGREGATE文はありません。

関連項目

[CREATE AGGREGATE](#), [DROP AGGREGATE](#)

[前のページ](#)
ABORT

[ホーム](#)
[上に戻る](#)

[次のページ](#)
ALTER COLLATION

ALTER COLLATION

名前

ALTER COLLATION -- 照合順の定義を変更する

概要

```
ALTER COLLATION name RENAME TO new_name
ALTER COLLATION name OWNER TO new_owner
ALTER COLLATION name SET SCHEMA new_schema
```

説明

ALTER COLLATION は照合順の定義を変更します。

ALTER COLLATION を使用するためには照合順を所有していなければなりません。所有者を変更するためには、新しい所有ロールの直接あるいは間接的なメンバでなければならず、かつ、そのロールが照合順のスキーマにおける CREATE 権限を持たなければなりません（この制限により強制的に所有者の変更が、照合順の削除と再作成を行ってもできないことを行わないようにします。ただし、スーパーユーザはすべての照合順の所有者を変更することができます）。

パラメータ

name

既存の照合順の名前（スキーマ修飾可）です。

new_name

照合順の新しい名前です。

new_owner

照合順の新しい所有者です。

new_schema

照合順の新しいスキーマです。

例

de_DE 照合順の名前を german に変更します。

```
ALTER COLLATION "de_DE" RENAME TO german;
```

en_US 照合順の所有者を joe に変更します。

```
ALTER COLLATION "en_US" OWNER TO joe;
```

互換性

標準SQLには ALTER COLLATION 文はありません。

関連項目

[CREATE COLLATION](#), [DROP COLLATION](#)

ALTER CONVERSION

名前

ALTER CONVERSION -- 変換の定義を変更する

概要

```
ALTER CONVERSION name RENAME TO new_name
ALTER CONVERSION name OWNER TO new_owner
ALTER CONVERSION name SET SCHEMA new_schema
```

説明

ALTER CONVERSION は変換の定義を変更します。

ALTER CONVERSION を使用するには変換の所有者でなければなりません。所有者を変更するには、直接または間接的に新しいロールのメンバでなければなりません。 また、そのロールは変換のスキーマにおいて CREATE 権限を持たなければなりません。（この制限により強制的に、所有者の変更に伴い変換の削除や再作成ができなくなるといった問題が起こらないようになります。しかし、スーパーユーザはすべての変換の所有者を変更することができます。）

パラメータ

- name*
既存の変換の名前です（スキーマ修飾名も可）。
- new_name*
新しい変換の名前です。
- new_owner*
新しい変換の所有者です。
- new_schema*
変換の新しいスキーマです。

例

```
iso_8859_1_to_utf8 変換の名前を latin1_to_unicode へ変更します。
ALTER CONVERSION iso_8859_1_to_utf8 RENAME TO latin1_to_unicode;

iso_8859_1_to_utf8 変換の所有者を joe に変更します。
ALTER CONVERSION iso_8859_1_to_utf8 OWNER TO joe;
```

互換性

標準SQLには ALTER CONVERSION 文はありません。

関連項目

[CREATE CONVERSION](#), [DROP CONVERSION](#)

ALTER DATABASE

名前

ALTER DATABASE -- データベースを変更する

概要

```
ALTER DATABASE name [ [ WITH ] option [ ... ] ]
```

ここで`option`は以下の通りです。

```
CONNECTION LIMIT conlimit

ALTER DATABASE name RENAME TO new_name

ALTER DATABASE name OWNER TO new_owner

ALTER DATABASE name SET TABLESPACE new_tablespace

ALTER DATABASE name SET configuration_parameter { TO | = } { value | DEFAULT }
ALTER DATABASE name SET configuration_parameter FROM CURRENT
ALTER DATABASE name RESET configuration_parameter
ALTER DATABASE name RESET ALL
```

説明

ALTER DATABASEはデータベースの属性を変更します。

最初の構文はデータベース毎の設定を変更します。（詳細は後述します。） データベース所有者とスーパーユーザのみがこの設定を変更することができます。

2番目の構文は、データベースの名前を変更します。 データベースの名前を変更できるのは、データベースの所有者またはスーパーユーザのみです。 ただし、スーパーユーザではない所有者はCREATEDB権限を所有していなければなりません。 現在のデータベースの名前を変更することはできません（必要ならば、別のデータベースに接続してください）。

3番目の構文は、データベースの所有者を変更します。 所有者を変更するにはデータベースを所有し、かつ、新しい所有者ロールの間接的あるいは直接的なメンバでなければなりません。 さらに、CREATEDB権限も持たなければなりません。（スーパーユーザはこれらの権限を自動的に持っていることに注意してください。）

4番目の構文は、データベースのデフォルトのテーブル空間を変更します。 データベースの所有者またはスーパーユーザのみがこれを行うことができます。 また、新しいテーブル空間における作成権限を持つ必要があります。 このコマンドはデータベースの古いデフォルトのテーブル空間から新しいテーブル空間にテーブルまたはインデックスを物理的にすべて移動します。 デフォルト以外のテーブル空間にあるテーブルまたはインデックスには影響がないことに注意してください。

残りの構文は、PostgreSQLデータベースにおける実行時設定変数のセッションのデフォルト値を変更します。 指定された値が、コマンド実行後にデータベースで開始される新規セッションのデフォルト値になります。 データベース固有のデフォルト値によって、`postgresql.conf` ファイルに記述されている設定や`postgres` コマンドラインから受け取った設定は上書きされます。 データベースにおけるセッションのデフォルト値を変更できるのは、データベースの所有者またはスーパーユーザのみです。 この方法では設定できない変数や、スーパーユーザのみが設定できる変数も存在します。

パラメータ

name

属性変更の対象となるデータベースの名前です。

conlimit

データベースへの最大同時接続数です。 -1は無制限を意味します。

new_name

新しいデータベース名です。

new_owner

新しいデータベースの所有者です。

new_tablespace

新しいデータベースのデフォルトのテーブル空間です。

configuration_parameter
value

指定した設定パラメータについて、データベースのセッションのデフォルト値を指定した値に設定します。
valueがDEFAULTの場合、あるいは等価なRESETが使用されている場合、データベース固有の設定は無効になり、新しいセッションではシステム全体のデフォルト設定が継承されます。全てのデータベース固有の設定をクリアするには、RESET ALLを使用してください。SET FROM CURRENTは、データベース固有の値としてセッションにおけるパラメータの現在値を保管します。

設定可能なパラメータ名とその値に関する詳細については[SET](#)および[第18章](#)を参照してください。

注釈

データベースではなく特定のロールにセッションのデフォルト値を関連付けることもできます。[ALTER ROLE](#)を参照してください。設定が競合する場合には、ロール固有の設定により、データベース固有の設定が上書きされます。

例

test データベース内のインデックススキャンをデフォルトで無効にします。

```
ALTER DATABASE test SET enable_indexscan TO off;
```

互換性

ALTER DATABASE文はPostgreSQLの拡張です。

関連項目

[CREATE DATABASE](#), [DROP DATABASE](#), [SET](#), [CREATE TABLESPACE](#)

[前のページ](#)
ALTER CONVERSION

[ホーム](#)
[上に戻る](#)

[次のページ](#)
ALTER DEFAULT PRIVILEGES

ALTER DEFAULT PRIVILEGES

名前

ALTER DEFAULT PRIVILEGES -- デフォルトのアクセス権限を定義する

概要

```
ALTER DEFAULT PRIVILEGES
    [ FOR { ROLE | USER } target_role [, ...] ]
    [ IN SCHEMA schema_name [, ...] ]
    abbreviated_grant_or_revoke
```

ここで*abbreviated_grant_or_revoke*は以下のいずれかです。

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
    ON TABLES
    TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]
```

```
GRANT { { USAGE | SELECT | UPDATE }
    [, ...] | ALL [ PRIVILEGES ] }
    ON SEQUENCES
    TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]
```

```
GRANT { EXECUTE | ALL [ PRIVILEGES ] }
    ON FUNCTIONS
    TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]
```

```
GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON TYPES
    TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]
```

```
REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
    ON TABLES
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]
```

```
REVOKE [ GRANT OPTION FOR ]
    { { USAGE | SELECT | UPDATE }
    [, ...] | ALL [ PRIVILEGES ] }
    ON SEQUENCES
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]
```

```
REVOKE [ GRANT OPTION FOR ]
    { EXECUTE | ALL [ PRIVILEGES ] }
    ON FUNCTIONS
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]
```

```
REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON TYPES
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]
```

説明

ALTER DEFAULT PRIVILEGESにより今後作成されるオブジェクトに適用される権限を設定することができます。（既存のオブジェクトに割り当てられている権限には影響しません。）現時点ではテーブル（ビュー、外部テーブルを含む）、シーケンス、関数、型（ドメインを含む）用の権限のみを変更可能です。

ユーザ自身、ユーザがメンバとして属するロールにより作成されるオブジェクト向けのデフォルト権限を変更することができます。権限は大域的に（つまり現在のデータベース内に作成されるすべてのオブジェクトに対して）設定することも、指定したスキーマ内に作成されるオブジェクトのみに対して設定することもできます。スキーマ単位で指定されるデフォルト権限は、大域的な個々の種類のオブジェクト用のデフォルト権限に追加されます。

[GRANT](#)の説明通り、任意の種類のオブジェクトのデフォルト権限は通常、オブジェクト所有者に対して付与可能な権限すべてを

付与します。また、同様にPUBLICに対して一部の権限を付与することもできます。しかしこの動作はALTER DEFAULT PRIVILEGESを用いて大域デフォルト権限により変更可能です。

パラメータ

target_role

現在のロールがメンバとして属する、既存のロールの名前です。FOR ROLEを省略した場合、現在のロールと仮定されます。

schema_name

既存のスキーマの名前です。各target_roleは指定したスキーマそれぞれに対してCREATE権限を持つ必要があります。IN SCHEMAを省略した場合、大域的なデフォルト権限が変更されます。

role_name

権限を付与または取り上げる、既存のロールの名前です。このパラメータ、およびabbreviated_grant_or_revoke内の他のパラメータは、GRANTやREVOKEの説明通りに動作します。ただし、指名したオブジェクトではなくオブジェクトクラス全体に対して権限を設定する点が異なります。

注釈

デフォルト権限としてすでに割り当てられている情報を入手するためにはpsqlの\ddpコマンドを使用してください。権限の値の意味は、GRANTの\dpの説明と同じです。

デフォルト権限を変更したロールを削除したい場合、デフォルト権限の項目を取り除くために、そのデフォルト権限の変更を元に戻すかDROP OWNED BYを使用する必要があります。

例

myschemaスキーマ内に今後作成されるすべてのテーブル（およびビュー）に対して、全員にSELECT権限を付与します。また、webuserロールにはそれらに挿入できるようにします。

```
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema GRANT SELECT ON TABLES TO PUBLIC;
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema GRANT INSERT ON TABLES TO webuser;
```

今後作成されるテーブルが通常以外の権限を持たないように、上を元に戻します。

```
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema REVOKE SELECT ON TABLES FROM PUBLIC;
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema REVOKE INSERT ON TABLES FROM webuser;
```

今後adminロールにより作成されるすべての関数について、通常関数に付与される、全員に対するEXECUTE権限を取り除きます。

```
ALTER DEFAULT PRIVILEGES FOR ROLE admin REVOKE EXECUTE ON FUNCTIONS FROM PUBLIC;
```

互換性

標準SQLにはALTER DEFAULT PRIVILEGES文はありません。

関連項目

[GRANT](#), [REVOKE](#)

ALTER DOMAIN

名前

ALTER DOMAIN -- ドメイン定義を変更する

概要

```
ALTER DOMAIN name
    { SET DEFAULT expression | DROP DEFAULT }
ALTER DOMAIN name
    { SET | DROP } NOT NULL
ALTER DOMAIN name
    ADD domain_constraint [ NOT VALID ]
ALTER DOMAIN name
ALTER DOMAIN name
    DROP CONSTRAINT [ IF EXISTS ] constraint_name [ RESTRICT | CASCADE ]
ALTER DOMAIN name
    RENAME CONSTRAINT constraint_name TO new_constraint_name
ALTER DOMAIN name
    VALIDATE CONSTRAINT constraint_name
ALTER DOMAIN name
    VALIDATE CONSTRAINT constraint_name
ALTER DOMAIN name
    OWNER TO new_owner
ALTER DOMAIN name
    RENAME TO new_name
ALTER DOMAIN name
    SET SCHEMA new_schema
```

説明

ALTER DOMAINは既存ドメインの定義を変更します。 以下に示す副構文があります。

SET/DROP DEFAULT

この構文はドメインのデフォルト値の設定または削除を行います。 指定したデフォルト値は、その後のINSERTコマンドのみに適用されることに注意してください。 そのドメインを使用したテーブルの既存の行には影響を与えません。

SET/DROP NOT NULL

この構文はドメインがNULL値を持つことができるかどうかを変更します。 SET NOT NULLを実行できるのは、ドメインを使用する列にNULL値が含まれていない場合のみです。

ADD *domain_constraint* [NOT VALID]

この構文はCREATE DOMAINと同一の構文を使って、ドメインに新しい制約を付与します。 新しい制約がドメインに追加された時、そのドメインを使用するすべての列が新しく追加された制約に対して検査されます。 これらの検査をNOT VALIDオプションを使用して新しい制約を追加することで抑制することができます。 ALTER DOMAIN ... VALIDATE CONSTRAINTを使用することで、後で制約を有効にすることができます。 新しく挿入または更新された行については、NOT VALIDと印が付いていたとしても、常にすべての制約が検査されます。 NOT VALIDはCHECK制約でのみ受け付けられます。

DROP CONSTRAINT [IF EXISTS]

この構文はドメイン上の制約を削除します。 IF EXISTSが指定された場合、制約が存在しなくてもエラーになりません。 この場合は代わりに注意メッセージが発生します。

RENAME CONSTRAINT

この構文はドメイン上の制約名を変更します。

VALIDATE CONSTRAINT

この構文は、以前にNOT VALIDとして追加された制約を検証します。 つまり、そのドメインを使用する列の中のすべてのデータが指定された制約を満たすかどうかを検証します。

OWNER

この構文はドメインの所有者を指定したユーザに変更します。

RENAME

この構文はドメインの名前を変更します。

SET SCHEMA

この構文はドメインのスキーマを変更します。 ドメインに関連する制約はすべて新しいスキーマに移動します。

`ALTER DOMAIN`を使用するにはドメインを所有していなければなりません。 ドメインのスキーマを変更するには、新しいスキーマにおける`CREATE`権限を持たなければなりません。 所有者を変更するには、直接または間接的に新しいロールのメンバでなければなりません。 また、そのロールはドメインのスキーマにおいて`CREATE`を持たなければなりません。 （この制限により強制的に、所有者の変更に伴いドメインの削除や再作成ができなくなるといった問題が起こらないようになります。 しかし、スーパーユーザはすべてのドメインの所有者を変更することができます。）

パラメータ

name

変更対象となる既存のドメインの名前です（スキーマ修飾名も可）。

domain_constraint

ドメイン用の新しいドメイン制約です。

constraint_name

削除または名前を変更する既存の制約名です。

`NOT VALID`

制約の妥当性について既存の列を検証しません。

`CASCADE`

その制約に依存するオブジェクトを自動的に削除します。

`RESTRICT`

依存するオブジェクトがある場合、制約の削除要求を拒否します。 これがデフォルトの動作です。

new_name

ドメインの新しい名前です。

new_constraint_name

制約の新しい名前です。

new_owner

ドメインの新しい所有者となるユーザの名前です。

new_schema

ドメインの新しいスキーマです。

注釈

今のところ、`ALTER DOMAIN ADD CONSTRAINT`と`ALTER DOMAIN SET NOT NULL`は、指定されたドメインおよびその派生ドメインがデータベース内のいずれかのテーブルの複合型の列で使用されていた場合、失敗します。 これは少なくとも、こうした入れ子になった列に対する新しい制約を検証できるように改良されなければなりません。

例

ドメインに`NOT NULL`制約を付与します。

```
ALTER DOMAIN zipcode SET NOT NULL;
```

ドメインからNOT NULL制約を削除します。

```
ALTER DOMAIN zipcode DROP NOT NULL;
```

ドメインにCHECK制約を付与します。

```
ALTER DOMAIN zipcode ADD CONSTRAINT zipchk CHECK (char_length(VALUE) = 5);
```

ドメインからCHECK制約を削除します。

```
ALTER DOMAIN zipcode DROP CONSTRAINT zipchk;
```

ドメイン状の検査制約の名前を変更します。

```
ALTER DOMAIN zipcode RENAME CONSTRAINT zipchk TO zip_check;
```

ドメインを新しいスキーマに移動します。

```
ALTER DOMAIN zipcode SET SCHEMA customers;
```

互換性

ALTER DOMAIN文は標準SQLに準拠しています。ただし、OWNER、RENAME、SET SCHEMA、VALIDATE CONSTRAINT構文は例外で、PostgreSQLの拡張です。ADD CONSTRAINT構文のNOT VALID句もPostgreSQLの拡張です。

関連項目

[CREATE DOMAIN](#), [DROP DOMAIN](#)

[前のページ](#)

ALTER DEFAULT PRIVILEGES

[ホーム](#)
[上に戻る](#)

[次のページ](#)

ALTER EXTENSION

ALTER EXTENSION

名前

ALTER EXTENSION -- 拡張の定義を変更する

概要

```

ALTER EXTENSION name UPDATE [ TO new_version ]
ALTER EXTENSION name SET SCHEMA new_schema
ALTER EXTENSION name ADD member_object
ALTER EXTENSION name DROP member_object

```

ここで`member_object`は以下の通りです。

```

AGGREGATE agg_name (agg_type [, ...] ) |
CAST (source_type AS target_type) |
COLLATION object_name |
CONVERSION object_name |
DOMAIN object_name |
FOREIGN DATA WRAPPER object_name |
FOREIGN TABLE object_name |
FUNCTION function_name ( [ [ argmode ] [ argname ] argtype [, ...] ] ) |
OPERATOR operator_name (left_type, right_type) |
OPERATOR CLASS object_name USING index_method |
OPERATOR FAMILY object_name USING index_method |
[ PROCEDURAL ] LANGUAGE object_name |
SCHEMA object_name |
SEQUENCE object_name |
SERVER object_name |
TABLE object_name |
TEXT SEARCH CONFIGURATION object_name |
TEXT SEARCH DICTIONARY object_name |
TEXT SEARCH PARSER object_name |
TEXT SEARCH TEMPLATE object_name |
TYPE object_name |
VIEW object_name

```

説明

ALTER EXTENSION はインストールされた拡張の定義を変更します。 複数の副構文があります。

UPDATE

この構文は拡張を新しいバージョンに更新します。 拡張は、現在インストールされているバージョンから要求するバージョンに変更することができる、適切な更新スクリプト（またはスクリプト群）を提供しなければなりません。

SET SCHEMA

この構文は拡張のオブジェクトを別のスキーマに移動します。 このコマンドを成功させるためには、拡張は再配置可能でなければなりません。

ADD *member_object*

この構文は既存のオブジェクトを拡張に追加します。 これは主に拡張の更新スクリプトで有用です。 オブジェクトはその後拡張のメンバとして扱われます。 特に、オブジェクトの削除は拡張の削除によってのみ可能です。

DROP *member_object*

この構文は拡張からメンバオブジェクトを削除します。 これは主に拡張の更新スクリプトで有用です。 オブジェクトは削除されません。 拡張との関連がなくなるだけです。

これらの操作の詳細については[項35.15](#)を参照してください。

ALTER EXTENSION を使用するためには拡張の所有者でなければなりません。 同様にADD/DROP 構文では追加されるオブジェクトまたは削除されるオブジェクトの所有者でなければなりません。

パラメータ

name

インストールされた拡張の名前です。

new_version

更新したい新しい拡張のバージョンです。これは識別子または文字列リテラルのいずれかで記述することができます。指定がない場合、`ALTER EXTENSION UPDATE`は拡張の制御ファイル内でデフォルトバージョンとして示されるものへの更新を試行します。

new_schema

拡張の新しいスキーマです。

object_name
agg_name
function_name
operator_name

拡張に追加する、または、拡張から削除するオブジェクトの名前です。テーブル、集約、ドメイン、外部テーブル、関数、演算子、演算子クラス、演算子族、シーケンス、全文検索オブジェクト、型、ビューの名前はスキーマ修飾可能です。

agg_type

集約関数が操作する入力データ型です。引数を持たない集約関数を参照するためには、入力データ型のリストの部分で*と記述してください。

source_type

キャストの変換元データ型の名前です。

target_type

キャストの変換先データ型の名前です。

argmode

関数引数のモードで`IN`、`OUT`、`INOUT`、`VARIADIC`のいずれかです。省略時のデフォルトは`IN`です。関数を識別するためには入力引数だけが必要です、実際のところ`ALTER EXTENSION`は`OUT`引数を考慮しないことに注意してください。このため`IN`、`INOUT`および`VARIADIC`引数を列挙するだけで十分です。

argname

関数引数の名前です。関数を識別するためには入力引数だけが必要です、実際のところ`ALTER EXTENSION`は引数名を考慮しないことに注意してください。

argtype

もしあれば関数引数のデータ型（スキーマ修飾可）です。

left_type
right_type

演算子の引数のデータ型（スキーマ修飾可）です 前置または後置演算子における存在しない引数には`NONE`と記述してください。

`PROCEDURAL`

これは無意味な単語です。

例

`hstore` 拡張をバージョン2.0まで更新します。

```
ALTER EXTENSION hstore UPDATE TO '2.0';
```

`hstore` 拡張のスキーマを`utils`に変更します。

```
ALTER EXTENSION hstore SET SCHEMA utils;
```

`hstore` 拡張に既存の関数を追加します。

```
ALTER EXTENSION hstore ADD FUNCTION populate_record(anyelement, hstore);
```

互換性

ALTER EXTENSIONはPostgreSQLの拡張です。

関連項目

[CREATE EXTENSION](#), [DROP EXTENSION](#)

[前のページ](#)

ALTER DOMAIN

[ホーム](#)

[上に戻る](#)

[次のページ](#)

ALTER FOREIGN DATA WRAPPER

ALTER FOREIGN DATA WRAPPER

名前

ALTER FOREIGN DATA WRAPPER -- 外部データラップの定義を変更する

概要

```
ALTER FOREIGN DATA WRAPPER name
    [ HANDLER handler_function | NO HANDLER ]
    [ VALIDATOR validator_function | NO VALIDATOR ]
    [ OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ] ) ]
ALTER FOREIGN DATA WRAPPER name OWNER TO new_owner
ALTER FOREIGN DATA WRAPPER name RENAME TO new_name
```

説明

ALTER FOREIGN DATA WRAPPERは外部データラップの定義を変更します。 このコマンドの最初の構文はサポート関数または外部データラップの一般的なオプションを変更します。（少なくとも1つの句が必要です。） 第2の構文は外部データラップの所有者を変更します。

スーパーユーザのみが外部データラップを変更することができます。 さらにスーパーユーザのみが外部データラップを所有することができます。

パラメータ

name

既存の外部データラップの名前です。

HANDLER *handler_function*

外部データラップ用の新しいハンドラ関数を指定します。

NO HANDLER

これを使用して外部データラップがハンドラ関数を持たないことを指定します。

ハンドラを持たない外部データラップを使用する外部テーブルにはアクセスできないことに注意してください。

VALIDATOR *validator_function*

外部データラップ用の新しい検証関数を指定します。

検証関数を変更した後、外部データラップのオプション、サーバ、ユーザマップが無効になることがあることに注意してください。 外部データラップを使用する前にこれらのオプションが正しいことを確実にすることはユーザの責任です。

NO VALIDATOR

これは、外部データラップが検証関数を持たないことを指定するために使用されます。

OPTIONS ([ADD | SET | DROP] *option* ['*value*'] [, ...])

外部データラップ用のオプションを変更します。 ADD、SET、DROPは実行する動作を指定します。 明示的な動作の指定がなければADDとみなされます。 オプション名は一意でなければなりません。 また名前と値は外部データラップの検証関数を使用して検証されます。

new_owner

外部データラップの新しい所有者のユーザ名です。

new_name

外部データラップの新しい名前です。

例

`dbi`外部データラッパを変更し、`foo`オプションを追加し、`bar`オプションを削除します。

```
ALTER FOREIGN DATA WRAPPER dbi OPTIONS (ADD foo '1', DROP 'bar');
```

外部データラッパ`dbi`の検証関数を`bob.myvalidator`に変更します。

```
ALTER FOREIGN DATA WRAPPER dbi VALIDATOR bob.myvalidator;
```

互換性

`ALTER FOREIGN DATA WRAPPER`はISO/IEC 9075-9 (SQL/MED)に従います。ただし、`HANDLER`、`VALIDATOR`、`OWNER` `TO`、`RENAME`句は拡張です。

関連項目

[CREATE FOREIGN DATA WRAPPER](#), [DROP FOREIGN DATA WRAPPER](#)

[前のページ](#)

ALTER EXTENSION

[ホーム](#)

[上に戻る](#)

[次のページ](#)

ALTER FOREIGN TABLE

ALTER FOREIGN TABLE

名前

ALTER FOREIGN TABLE -- 外部テーブルの定義を変更

概要

```

ALTER FOREIGN TABLE [ IF EXISTS ] name
    action [, ... ]
ALTER FOREIGN TABLE [ IF EXISTS ] name
    RENAME [ COLUMN ] column_name TO new_column_name
ALTER FOREIGN TABLE [ IF EXISTS ] name
    RENAME TO new_name
ALTER FOREIGN TABLE [ IF EXISTS ] name
    SET SCHEMA new_schema

```

ここで*action*は以下のいずれかです。

```

ADD [ COLUMN ] column_name data_type [ NULL | NOT NULL ]
DROP [ COLUMN ] [ IF EXISTS ] column_name [ RESTRICT | CASCADE ]
ALTER [ COLUMN ] column_name [ SET DATA ] TYPE data_type
ALTER [ COLUMN ] column_name { SET | DROP } NOT NULL
ALTER [ COLUMN ] column_name SET STATISTICS integer
ALTER [ COLUMN ] column_name SET ( attribute_option = value [, ... ] )
ALTER [ COLUMN ] column_name RESET ( attribute_option [, ... ] )
ALTER [ COLUMN ] column_name OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ] )
OWNER TO new_owner
OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ] )

```

説明

ALTER FOREIGN TABLE は既存の外部テーブルの定義を変更します。 以下のように複数の副構文があります。

ADD COLUMN

この構文は、[CREATE FOREIGN TABLE](#)と同じ文法を使用して、外部テーブルに新しい列を追加します。

DROP COLUMN [IF EXISTS]

この構文は外部テーブルから列を削除します。 ビューなど何らかのその他のテーブルがこの列に依存する場合、CASCADE を付けなければなりません。 IF EXISTS が指定された場合、列が存在しなくてもエラーになりません。 この場合注意が代わりに発せられます。

IF EXISTS

外部テーブルが存在しない場合でもエラーとしません。 この場合注意メッセージが発生します。

SET DATA TYPE

この構文は外部テーブルの列の型を変更します。

SET/DROP NOT NULL

列にNULL値を許すか許さないかどうか印を付けます。

SET STATISTICS

この構文は、この後の[ANALYZE](#)操作における列単位の統計情報収集対象を設定します。 詳細については[ALTER TABLE](#)の類似の構文を参照してください。

```

SET ( attribute_option = value [, ... ] )
RESET ( attribute_option [, ... ] )

```

この構文は属性単位のオプションを設定または未設定状態にします。 詳細については[ALTER TABLE](#)における類似の構文を参照してください。

OWNER

この構文は外部テーブルの所有者を指定ユーザに変更します。

RENAME

RENAME構文は外部テーブルの名前または外部テーブル内の個々の列の名前を変更します。

SET SCHEMA

この構文は外部テーブルを別のスキーマに移動します。

OPTIONS ([ADD | SET | DROP] option ['value'] [, ...])

外部テーブルもしくはその列の1つに対するオプションを変更します。ADD、SET、DROPは実行する操作を指定します。明示的な操作指定がない場合ADDと仮定されます。重複したオプション名は許されません。（しかしテーブルオプションと列オプションとで同じ名前を持たせることは問題ありません。）またオプションの名前と値は外部データラップのライブラリを使用して検証されます。

RENAMEおよびSET SCHEMA以外の操作はすべて、複数変更項目リストにまとめて並行に適用することができます。例えば、複数の列の追加、複数の列の型変更、またはその両方を単一のコマンドで行うことができます。

ALTER FOREIGN TABLEを使用するためにはテーブルの所有者でなければなりません。また外部テーブルのスキーマを変更するためには、新しいスキーマに対してCREATE権限を持っていなければなりません。所有者を変更するためには、新しく所有者となるロールの直接的または間接的なメンバでなければなりません。また新しく所有者となるロールはテーブルのスキーマに対してCREATE権限を持っていなければなりません。（これらの制限により強制的に所有者の変更が、テーブルの削除と再作成を行ってもできないことを行わないようにします。）列の追加または列の型の変更を行うためには、そのデータ型に対するUSAGE権限も必要です。

パラメータ

name

変更対象の既存外部テーブルの名前（スキーマ修飾可）です。

column_name

新しい列または既存の列の名前です。

new_column_name

既存の列に対する新しい名前です。

new_name

テーブルの新しい名前です。

data_type

新しい列のデータ型、または既存の列に対する新しいデータ型です。

CASCADE

削除される列に依存するオブジェクト（この列を参照するビューなど）を自動的に削除します。

RESTRICT

何らかの依存するオブジェクトが存在する場合削除を拒否します。これがデフォルトの動作です。

new_owner

テーブルの新しい所有者のユーザ名です。

new_schema

テーブルの移動先となるスキーマの名前です。

注意

COLUMNキーワードはノイズであり、省略可能です。

ADD COLUMNまたはDROP COLUMNにより列が追加、削除された時、NOT NULL制約が追加された時、SET DATA TYPEにより列の型が変更された時、外部サーバとの一貫性は検査されません。確実にテーブル定義をリモート側に合わせることはユーザの責任です。

有効なパラメータに関する詳しい説明については[CREATE FOREIGN TABLE](#)を参照してください。

例

列を非NULLと印付けします。

```
ALTER FOREIGN TABLE distributors ALTER COLUMN street SET NOT NULL;
```

外部テーブルのオプションを変更します。

```
ALTER FOREIGN TABLE myschema.distributors OPTIONS (ADD opt1 'value', SET opt2, 'value2', DROP opt3 'value3');
```

互換性

ADD、DROP、SET DATA TYPE構文は標準SQLに準拠します。他の構文は標準SQLに対するPostgreSQLの拡張です。単一のALTER FOREIGN TABLEコマンドに複数の操作を指定する機能も拡張です。

ALTER FOREIGN TABLE DROP COLUMNを用いて外部テーブルの列だけを削除してゼロ列のテーブルとして残すことができます。これは拡張であり、SQLではゼロ列の外部テーブルを許しません。

ALTER FUNCTION

名前

ALTER FUNCTION -- 関数定義を変更する

概要

```
ALTER FUNCTION name ( [ [ argmode ] [ argname ] argtype [, ...] ] )  
    action [ ... ] [ RESTRICT ]  
ALTER FUNCTION name ( [ [ argmode ] [ argname ] argtype [, ...] ] )  
    RENAME TO new_name  
ALTER FUNCTION name ( [ [ argmode ] [ argname ] argtype [, ...] ] )  
    OWNER TO new_owner  
ALTER FUNCTION name ( [ [ argmode ] [ argname ] argtype [, ...] ] )  
    SET SCHEMA new_schema
```

ここで、*action*は以下のいずれかです。

```
CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT  
IMMUTABLE | STABLE | VOLATILE | [ NOT ] LEAKPROOF  
[ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER  
COST execution_cost  
ROWS result_rows  
SET configuration_parameter { TO | = } { value | DEFAULT }  
SET configuration_parameter FROM CURRENT  
RESET configuration_parameter  
RESET ALL
```

説明

ALTER FUNCTIONは関数定義を変更します。

ALTER FUNCTIONを使用するには関数の所有者でなければなりません。関数のスキーマを変更するには、新しいスキーマにおけるCREATE権限も必要です。所有者を変更するには、直接または間接的に新しいロールのメンバでなければなりません。また、そのロールは関数のスキーマにおいてCREATEを持たなければなりません。（この制限により、強制的に、所有者の変更が関数の削除と再作成で行うことができない何らかの処理を行わないようになります。しかし、スーパーユーザはすべての関数の所有者を変更することができます。）

パラメータ

name

既存の関数名です（スキーマ修飾名も可）。

argmode

引数のモードで、IN、OUT、INOUT、VARIADICのいずれかです。省略された場合のデフォルトはINです。関数の識別を行うには入力引数のみが必要ですので、実際にはALTER FUNCTIONがOUT引数を見捨てることに注意してください。ですので、IN、INOUTおよびVARIADIC引数を列挙することで十分です。

argname

引数の名前です。関数の識別を行うには引数のデータ型のみが必要ですので、実際にはALTER FUNCTIONは引数の名前を見捨てることに注意してください。

argtype

もしあれば、その関数の引数のデータ型（複数可）です。（スキーマで修飾可能です）

new_name

新しい関数名です。

new_owner

新しい関数の所有者です。関数に`SECURITY DEFINER`が指定されている場合、その後は新しい所有者の権限で関数が実行されることに注意してください。

new_schema

関数の新しいスキーマです。

`CALLLED ON NULL INPUT`
`RETURNS NULL ON NULL INPUT`
`STRICT`

`CALLLED ON NULL INPUT`は、引数の一部またはすべてが`NULL`の場合に関数が呼び出されるように変更します。`RETURNS NULL ON NULL INPUT`もしくは`STRICT`は、引数の一部が`NULL`の場合に関数が呼び出されないように変更します。代わりに自動的に`NULL`という結果が仮定されます。詳細は[CREATE FUNCTION](#)を参照してください。

`IMMUTABLE`
`STABLE`
`VOLATILE`

関数の揮発性を指定した設定に変更します。関数の揮発性に関する詳細については[CREATE FUNCTION](#)を参照してください。

`[ EXTERNAL ] SECURITY INVOKER`
`[ EXTERNAL ] SECURITY DEFINER`

関数のセキュリティを定義者にするか否かを変更します。`EXTERNAL`キーワードはSQLとの互換性のためのものであり、無視されます。この機能の詳細については[CREATE FUNCTION](#)を参照してください。

`LEAKPROOF`

関数を漏洩防止関数とみなすか否かを変更します。この機能に関する詳細については[CREATE FUNCTION](#)を参照してください。

`COST` *execution_cost*

関数の推定実行コストを変更します。詳細については[CREATE FUNCTION](#)を参照してください。

`ROWS` *result_rows*

集合を返す関数で返される推定行数を変更します。詳細については[CREATE FUNCTION](#)を参照してください。

configuration_parameter
value

関数呼び出し時に設定パラメータに対して行われる代入を追加または変更します。*value*が`DEFAULT`、またはそれと等価な`RESET`が使用された場合、関数の局所的な設定は削除されます。このため、関数はその環境内に存在する値で実行されます。すべての関数の局所的な設定を整理したければ`RESET ALL`を使用してください。`SET FROM CURRENT`は、セッションにおけるパラメータの現在値を関数起動時に適用される値として保管します。

設定可能なパラメータとその値に関する詳細については、[SET](#)および[第18章](#)を参照してください。

`RESTRICT`

標準SQLとの互換性のためのものであり、無視されます。

例

`integer`型用の`sqrt`関数の名前を`square_root`に変更します。

```
ALTER FUNCTION sqrt(integer) RENAME TO square_root;
```

`integer`型用の`sqrt`関数の所有者を`joe`に変更します。

```
ALTER FUNCTION sqrt(integer) OWNER TO joe;
```

`integer`型用の`sqrt`関数のスキーマを`maths`に変更します。

```
ALTER FUNCTION sqrt(integer) SET SCHEMA maths;
```

関数用に検索パスを自動的に設定するように調整します。

```
ALTER FUNCTION check_password(text) SET search_path = admin, pg_temp;
```

関数用の`search_path`の自動設定を無効にします。

```
ALTER FUNCTION check_password(text) RESET search_path;
```

呼び出し元で使用する検索パスでこの関数が実行されるようになります。

互換性

この文の一部は標準SQLのALTER FUNCTION文に従っています。 標準ではより多くの関数の属性を変更できますが、関数名の変更、関数を定義者の権限で実行するかどうかの変更、関数と設定パラメータ値の関連付け、関数の所有者やスキーマ、揮発性の変更を行う機能はありません。 また、標準ではRESTRICTキーワードを必須としていますが、PostgreSQLでは省略可能です。

関連項目

[CREATE FUNCTION](#), [DROP FUNCTION](#)

前のページ ALTER FOREIGN TABLE	ホーム 上に戻る	次のページ ALTER GROUP

ALTER GROUP

名前

ALTER GROUP -- ロールの名前またはメンバ資格を変更する

概要

```
ALTER GROUP group_name ADD USER user_name [, ... ]
ALTER GROUP group_name DROP USER user_name [, ... ]

ALTER GROUP group_name RENAME TO new_name
```

説明

ALTER GROUP はユーザグループの属性を変更します。 後方互換性のために受け付けられていますが、このコマンドは廃止予定です。 グループ（ユーザも同様）は、より一般化されたロールという概念に置き換えられたからです。

最初の2つの構文は、ユーザをグループに追加もしくはグループから削除します（この場合、任意のロールを"ユーザ"部分として、または"グループ" 部分として使用することができます）。 この種の構文は、実のところ、"グループ"として指名されたロール内のメンバ資格の付与、削除と同じです。 ですので、[GRANT](#)や[REVOKE](#)を使用する方法を薦めます。

3番目の構文はグループの名前を変更します。 これは、[ALTER ROLE](#)を使用したロール名の変更とまったく同じです。

パラメータ

group_name

変更するグループ（ロール）の名前です。

user_name

グループに追加または削除されるユーザ（ロール）です。 指定するユーザは存在するものでなければいけません。 ALTER GROUP は、ユーザの作成も削除も行いません。

new_name

新しいグループ名です。

例

ユーザをグループに追加します。

```
ALTER GROUP staff ADD USER karl, john;
```

ユーザをグループから削除します。

```
ALTER GROUP workers DROP USER beth;
```

互換性

標準SQLにはALTER GROUP 文はありません。

関連項目

[GRANT](#), [REVOKE](#), [ALTER ROLE](#)

ALTER INDEX

名前

ALTER INDEX -- インデックス定義を変更する

概要

```
> ALTER INDEX [ IF EXISTS ] name RENAME TO new_name
> ALTER INDEX [ IF EXISTS ] name SET TABLESPACE tablespace_name
> ALTER INDEX [ IF EXISTS ] name SET ( storage_parameter = value [, ... ] )
> ALTER INDEX [ IF EXISTS ] name RESET ( storage_parameter [, ... ] )
```

説明

ALTER INDEX は既存のインデックスの定義を変更します。 以下のような副構文が存在します。

IF EXISTS

インデックスが存在しない場合にエラーとしません。 この場合注意メッセージが発生します。

RENAME

このRENAME 構文は、インデックスの名前を変更します。 格納されたデータには影響しません。

SET TABLESPACE

この構文は、インデックスのテーブル空間を指定したテーブル空間に変更し、インデックスに関連する（複数の）データファイルを移動します。 [CREATE TABLESPACE](#) も参照してください。

SET (*storage_parameter* = *value* [, ...])

この構文は、インデックスに対し、インデックスメソッド固有の1つ以上の格納パラメータを変更します。 設定可能なパラメータについては[CREATE INDEX](#)を参照してください。 このコマンドにより、インデックスの内容は即座に変更されないことに注意してください。 パラメータによりませんが、予定の効果をを得るために[REINDEX](#)を使用してインデックスを再構築しなければならない場合もあります。

RESET (*storage_parameter* [, ...])

この構文は、1つ以上のインデックスメソッド固有の格納パラメータをデフォルト値に再設定します。 SET 同様、インデックスを完全に更新するためにREINDEXが必要になる場合があります。

パラメータ

name

変更対象の既存のインデックスの名前です（スキーマ修飾名も可）。

new_name

インデックスの新しい名前です。

tablespace_name

インデックスの移動先のテーブル空間です。

storage_parameter

インデックスメソッド固有の格納パラメータの名前です。

value

インデックスメソッド固有の格納パラメータの新しい値です。 パラメータに応じてこれが数値になることも文字列になることもあります。

注釈

これらの操作は[ALTER TABLE](#)を使用して行うこともできます。 実際には、`ALTER INDEX`は、`ALTER TABLE`のインデックス用構文の別名に過ぎません。

以前は`ALTER INDEX OWNER`という種類の構文がありましたが、（警告の上）無視されるようになりました。 インデックスの所有者は基のテーブルの所有者と異なるものにすることができません。 テーブルの所有者を変更すると自動的にインデックスの所有者も変わります。

システムカタログ用インデックスに対する変更は許されていません。

例

既存のインデックスの名前を変更します。

```
ALTER INDEX distributors RENAME TO suppliers;
```

インデックスを別のテーブル空間に移動します。

```
ALTER INDEX distributors SET TABLESPACE fasttablespace;
```

インデックスのフィルファクタを変更します（インデックスメソッドがフィルファクタをサポートしていることを仮定します）。

```
ALTER INDEX distributors SET (fillfactor = 75);
REINDEX INDEX distributors;
```

互換性

`ALTER INDEX`はPostgreSQLの拡張です。

関連項目

[CREATE INDEX](#), [REINDEX](#)

ALTER LANGUAGE

名前

ALTER LANGUAGE -- 手続き言語の定義を変更する

概要

```
ALTER [ PROCEDURAL ] LANGUAGE name RENAME TO new_name
ALTER [ PROCEDURAL ] LANGUAGE name OWNER TO new_owner
```

説明

ALTER LANGUAGE は、手続き言語の定義を変更します。 言語名の変更および言語の所有者の変更のみが可能です。 ALTER LANGUAGE を使用するためにはスーパーユーザまたは言語の所有者でなければなりません。

パラメータ

name

言語の名前です。

new_name

新しい言語の名前です。

new_owner

新しい言語の所有者です。

互換性

標準SQLにはALTER LANGUAGE 文はありません。

関連項目

[CREATE LANGUAGE](#), [DROP LANGUAGE](#)

[前のページ](#)

ALTER INDEX

[ホーム](#)

[上に戻る](#)

[次のページ](#)

ALTER LARGE OBJECT

ALTER LARGE OBJECT

名前

ALTER LARGE OBJECT -- ラージオブジェクトの定義を変更する

概要

```
ALTER LARGE OBJECT large_object_oid OWNER TO new_owner
```

説明

ALTER LARGE OBJECT はラージオブジェクトの定義を変更します。 唯一の機能は新しい所有者の割り当てです。 ALTER LARGE OBJECT を使用するためにはスーパーユーザまたはラージオブジェクトの所有者でなければなりません。

パラメータ

large_object_oid

変更対象のラージオブジェクトのOIDです。

new_owner

ラージオブジェクトの新しい所有者です。

互換性

標準SQLにはALTER LARGE OBJECT 文はありません。

関連項目

[第32章](#)

[前のページ](#)

ALTER LANGUAGE

[ホーム](#)
[上に戻る](#)

[次のページ](#)

ALTER OPERATOR

ALTER OPERATOR

名前

ALTER OPERATOR -- 演算子の定義を変更する

概要

```
ALTER OPERATOR name ( { left_type | NONE } , { right_type | NONE } ) OWNER TO new_owner
ALTER OPERATOR name ( { left_type | NONE } , { right_type | NONE } ) SET SCHEMA new_schema
```

説明

ALTER OPERATOR は演算子の定義を変更します。 現在、演算子の所有者の変更のみが可能です。

ALTER OPERATOR を使用するには演算子の所有者でなければなりません。 所有者を変更するには、直接または間接的に新しい所有者ロールのメンバでなければなりません。 また、そのロールが演算子のスキーマにおいて CREATE を持たなければなりません。（この制限により、強制的に、所有者の変更が演算子の削除と再作成で行うことができない何らかの処理を行わないようになります。 しかし、スーパーユーザはすべての演算子の所有者を変更することができます。）

パラメータ

name

既存の演算子の名前です（スキーマ修飾名も可）。

left_type

演算子の左オペランドのデータ型です。 左オペランドがない演算子の場合は NONE を指定します。

right_type

演算子の右オペランドのデータ型です。 右オペランドがない演算子の場合は NONE を指定します。

new_owner

新しい演算子の所有者です。

new_schema

演算子の新しいスキーマです。

例

text 型用の独自の演算子 a @@ b の所有者を変更します。

```
ALTER OPERATOR @@ (text, text) OWNER TO joe;
```

互換性

標準 SQL には ALTER OPERATOR 文はありません。

関連項目

[CREATE OPERATOR](#), [DROP OPERATOR](#)

ALTER OPERATOR CLASS

名前

ALTER OPERATOR CLASS -- 演算子クラスの定義を変更する

概要

```
ALTER OPERATOR CLASS name USING index_method RENAME TO new_name
ALTER OPERATOR CLASS name USING index_method OWNER TO new_owner
ALTER OPERATOR CLASS name USING index_method SET SCHEMA new_schema
```

説明

ALTER OPERATOR CLASS は演算子クラスの定義を変更します。

ALTER OPERATOR CLASS を使用するには演算子クラスの所有者でなければなりません。所有者を変更するには、直接または間接的に新しい所有者ロールのメンバでなければなりません。また、そのロールが演算子クラスのスキーマにおいて CREATE を持たなければなりません。（この制限により、強制的に、所有者の変更が演算子クラスの削除と再作成で行うことができない何らかの処理を行わないようになります。しかし、スーパーユーザはすべての演算子クラスの所有者を変更することができます。）

パラメータ

name

既存の演算子クラスの名前です（スキーマ修飾名も可）。

index_method

演算子クラス用のインデックスメソッドの名前です。

new_name

新しい演算子クラス名です。

new_owner

新しい演算子クラスの所有者です。

new_schema

演算子クラスの新しいスキーマです。

互換性

標準SQLには ALTER OPERATOR CLASS 文はありません。

関連項目

[CREATE OPERATOR CLASS](#), [DROP OPERATOR CLASS](#), [ALTER OPERATOR FAMILY](#)

ALTER OPERATOR FAMILY

名前

ALTER OPERATOR FAMILY -- 演算子族の定義を変更する

概要

```
ALTER OPERATOR FAMILY name USING index_method ADD
{ OPERATOR strategy_number operator_name ( op_type, op_type ) [ FOR SEARCH | FOR ORDER BY
sort_family_name ]
  | FUNCTION support_number [ ( op_type [ , op_type ] ) ] function_name ( argument_type [ , ... ] )
} [, ... ]
ALTER OPERATOR FAMILY name USING index_method DROP
{ OPERATOR strategy_number ( op_type [ , op_type ] )
  | FUNCTION support_number ( op_type [ , op_type ] )
} [, ... ]
ALTER OPERATOR FAMILY name USING index_method RENAME TO new_name
ALTER OPERATOR FAMILY name USING index_method OWNER TO new_owner
ALTER OPERATOR FAMILY name USING index_method SET SCHEMA new_schema
```

説明

ALTER OPERATOR FAMILYは演算子族の定義を変更します。 演算子やサポート関数を演算子族に追加することやそれらを演算子族から削除すること、演算子族の名前や所有者を変更することが可能です。

ALTER OPERATOR FAMILYを使用して演算子とサポート関数が演算子族に追加される時、これらは演算子族内の特定の演算子クラスの一部とはならず、単に演算子族内で"自由"なものになります。 これは、これらの演算子と関数が演算子族と意味的な互換性を持つが、特定のインデックスの正しい動作には必要とされないことを意味します。 （代わりに必要な演算子と関数は演算子クラスの一部として宣言しなければなりません。 [CREATE OPERATOR CLASS](#)を参照してください。） PostgreSQLでは演算子族の自由なメンバをいつでも演算子族から削除することができます。 しかし演算子クラス内のメンバは、クラス全体と依存するインデックスすべてを削除する以外に削除することはできません。 通常、単一データ型の演算子と関数は、特定のデータ型に対するインデックスをサポートするために必要ですので、演算子クラスの一部となります。 一方、データ型を跨る演算子と関数は、演算子族内の自由なメンバとなります。

ALTER OPERATOR FAMILYを使用するには、スーパーユーザでなければなりません（おかしな演算子族定義はサーバを混乱させクラッシュさせることがありますので、この制限がなされています）。

現時点ではALTER OPERATOR FAMILYは、インデックスメソッドで必要とされる演算子族がすべての演算子と関数を含んでいるかどうかを検査しません。 また、演算子と関数が自己制約集合を形成しているかどうかも検査しません。 有効な演算子族を定義することはユーザの責任です。

詳細は[項35.14](#)を参照してください。

パラメータ

name

既存の演算子族の名前です（スキーマ修飾可）。

index_method

演算子族が対象とするインデックスメソッドの名前です。

strategy_number

演算子族と関連した演算子に対するインデックスメソッドの戦略番号です。

operator_name

演算子族と関連した演算子の名前です（スキーマ修飾可）。

op_type

`OPERATOR`句では演算子の入力データ型、または左単項演算子、右単項演算子を表す `NONE` です。 `CREATE OPERATOR CLASS` と類似の構文と異なり、入力データ型を常に指定しなければなりません。

`ADD FUNCTION` では、関数の入力データ型と異なる場合、関数がサポートする予定の入力データ型です。 関数の入力データ型は常に正しく使用するデータ型であるため、B-Tree圧縮関数およびHash関数では `op_type` を指定する必要がありません。 B-Treeソートサポート関数とGiST、SP-GiST、GIN演算子クラスのすべての関数では、関数が使用する入力データ型を指定する必要があります。

`DROP FUNCTION` 句では、関数がサポートする予定の入力データ型を指定しなければなりません。

sort_family_name

順序付け演算子に関連するソート順序を記述する、既存の `btree` 演算子族の名前（スキーマ修飾も可）です。

`FOR SEARCH` も `FOR ORDER BY` も指定されない場合、`FOR SEARCH` がデフォルトです。

support_number

演算子族に関連する関数用のインデックスメソッドのサポートプロシージャの番号です。

function_name

演算子族用のインデックスメソッドのサポートプロシージャとなる関数の名前です（スキーマ修飾名でも可）。

argument_type

関数のパラメータのデータ型です。

new_name

演算子族の新しい名前です。

new_owner

演算子族の新しい所有者です。

new_schema

演算子族の新しいスキーマです。

`OPERATOR` と `FUNCTION` 句は任意の順番で記述できます。

注釈

`DROP` 構文が、戦略番号またはサポート番号と入力データ型という、演算子族の"スロット"のみを指定していることに注意してください。 そのスロットに存在する演算子または関数の名前については言及されません。 また、 `DROP FUNCTION` では、指定する型は関数がサポートする予定の入力データ型です。 GiST、SP-GiSTおよびGINインデックスでは、関数の実際の入力引数の型と関連しない可能性があります。

インデックス機構は使用する前に関数のアクセス権限を検査しません。 演算子族内の関数や演算子を含めることは、公的な実行権限を与えることと同じです。 これは通常、演算子族内で使用される関数では問題になりません。

演算子をSQL関数で定義してはいけません。 SQL関数はよく、呼び出し元の問い合わせ内でインライン展開されます。 すると、オプティマイザが問い合わせがインデックスに一致するかどうか認識できなくなります。

PostgreSQL 8.4より前までは、`OPERATOR` 句に `RECHECK` オプションを含めることができました。 インデックス演算子に"損失がある"かどうかは実行時にその場で決定されるようになりましたので、これはサポートされなくなりました。 これにより、演算子に損失があるかもしれないしないかもしれないような場合を効率的に扱うことができるようになりました。

例

以下のコマンド例は、データ型を跨る演算子とサポート関数を `int4` と `int2` データ型用のB-Tree演算子クラスをすでに含む演算子族に追加します。

```
ALTER OPERATOR FAMILY integer_ops USING btree ADD

-- int4 vs int2
OPERATOR 1 < (int4, int2) ,
OPERATOR 2 <= (int4, int2) ,
OPERATOR 3 = (int4, int2) ,
```

```
OPERATOR 4 >= (int4, int2) ,
OPERATOR 5 > (int4, int2) ,
FUNCTION 1 btint42cmp(int4, int2) ,

-- int2 vs int4
OPERATOR 1 < (int2, int4) ,
OPERATOR 2 <= (int2, int4) ,
OPERATOR 3 = (int2, int4) ,
OPERATOR 4 >= (int2, int4) ,
OPERATOR 5 > (int2, int4) ,
FUNCTION 1 btint24cmp(int2, int4) ;
```

これらの項目を再度削除します。

```
ALTER OPERATOR FAMILY integer_ops USING btree DROP
```

```
-- int4 vs int2
OPERATOR 1 (int4, int2) ,
OPERATOR 2 (int4, int2) ,
OPERATOR 3 (int4, int2) ,
OPERATOR 4 (int4, int2) ,
OPERATOR 5 (int4, int2) ,
FUNCTION 1 (int4, int2) ,

-- int2 vs int4
OPERATOR 1 (int2, int4) ,
OPERATOR 2 (int2, int4) ,
OPERATOR 3 (int2, int4) ,
OPERATOR 4 (int2, int4) ,
OPERATOR 5 (int2, int4) ,
FUNCTION 1 (int2, int4) ;
```

互換性

標準SQLにはALTER OPERATOR FAMILY文はありません。

関連項目

[CREATE OPERATOR FAMILY](#), [DROP OPERATOR FAMILY](#), [CREATE OPERATOR CLASS](#), [ALTER OPERATOR CLASS](#), [DROP OPERATOR CLASS](#)

[前のページ](#)

ALTER OPERATOR CLASS

[ホーム](#)
[上に戻る](#)

[次のページ](#)
ALTER ROLE

ALTER ROLE

名前

ALTER ROLE -- データベースロールを変更する

概要

```
ALTER ROLE name [ [ WITH ] option [ ... ] ]
```

ここで`option`は以下の通りです。

```
SUPERUSER | NOSUPERUSER
| CREATEDB | NOCREATEDB
| CREATEROLE | NOCREATEROLE
| CREATEUSER | NOCREATEUSER
| INHERIT | NOINHERIT
| LOGIN | NOLOGIN
| REPLICATION | NOREPLICATION
| CONNECTION LIMIT connlimit
| [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
| VALID UNTIL 'timestamp'
```

```
ALTER ROLE name RENAME TO new_name
```

```
ALTER ROLE name [ IN DATABASE database_name ] SET configuration_parameter { TO | = } { value | DEFAULT }
ALTER ROLE name [ IN DATABASE database_name ] SET configuration_parameter FROM CURRENT
ALTER ROLE name [ IN DATABASE database_name ] RESET configuration_parameter
ALTER ROLE name [ IN DATABASE database_name ] RESET ALL
```

説明

ALTER ROLEはPostgreSQLロールの属性を変更します。

このコマンドの最初の構文では、[CREATE ROLE](#)で指定可能な多くのロール属性を変更することができます。（指定し得るすべての属性に対応していますが、メンバ資格の追加および削除用のオプションはありません。追加および削除には[GRANT](#)と[REVOKE](#)を使用してください。）このコマンドで指定しなかった属性は以前の設定のまま残ります。データベーススーパーユーザはすべてのロールのすべての属性を変更することができます。CREATEROLE権限を持つロールは、スーパーユーザおよびレプリケーション以外のロールに対してすべての属性を変更することができます。通常のロールは自身のパスワードのみを変更することができます。

2番目の構文ではロールの名前を変更することができます。データベーススーパーユーザはすべてのロールの名前を変更できます。CREATEROLE権限を持つロールはスーパーユーザ以外のロールに対してその名前を変更できます。現在のセッションユーザの名前を変更することはできません。（必要ならば別のユーザで接続してください）。MD5暗号化パスワードではロール名を暗号用のソルトとして使用しますので、パスワードがMD5で暗号化されている場合、ロール名を変更するとパスワードは空になります。

残りの構文では、全データベース用、またはIN DATABASE句が指定された場合はそのデータベース用のセッションに対するロールの設定変数についてのセッションデフォルトを変更します。その後、ロールが新しいセッションを始めると常に、`postgresql.conf`内の設定や`postgres`コマンドラインより受け取った設定は上書きされ、指定された値がセッションのデフォルトとなります。これはログイン時のみに発生します。[SET ROLE](#)または[SET SESSION AUTHORIZATION](#)を実行しても新しい設定値は設定されません。全データベースに対する設定群は、ロールに割り当てたデータベース固有の設定により上書きされます。スーパーユーザはすべてのユーザのセッションのデフォルトを変更することができます。CREATEROLE権限を持つロールはスーパーユーザ以外のロールのデフォルトを変更することができます。通常のロールは自身のデフォルトのみを設定することができます。設定変数の中にはこの方法で変更できないものがあります。また、スーパーユーザがこのコマンドを発行した時にのみ変更できるものもあります。

パラメータ

name

属性を変更するロールの名前です。

SUPERUSER

```
NOSUPERUSER
CREATEDB
NOCREATEDB
CREATEROLE
NOCREATEROLE
CREATEUSER
NOCREATEUSER
INHERIT
NOINHERIT
LOGIN
NOLOGIN
REPLICATION
NOREPLICATION
CONNECTION LIMIT conlimit
PASSWORD password
ENCRYPTED
UNENCRYPTED
VALID UNTIL 'timestamp'
```

これらの句は、元々[CREATE ROLE](#)で設定された属性を変更します。 詳細は[CREATE ROLE](#)のマニュアルページを参照してください。

new_name

ロールの新しい名前です。

database_name

設定変数を設定する対象のデータベースの名前です。

configuration_parameter
value

指定した設定パラメータに対して、ロールのセッションデフォルトを指定した値に設定します。 *value*が`DEFAULT`、または`RESET`が使用されていた場合、ロール固有の変数設定は削除され、新しいセッションではロールはシステム全体のデフォルト設定を継承します。 すべてのロール固有の設定を削除するには`RESET ALL`を使用してください。 `SET FROM CURRENT`はセッションのパラメータ値をロール固有の値として保管します。 `IN DATABASE`が指定された場合、設定パラメータは指定されたロールとデータベースのみで設定または削除されます。

ロール固有の変数設定はログイン時のみに影響を与えます。 [SET ROLE](#)および[SET SESSION AUTHORIZATION](#)はロール固有の変数設定を処理しません。

取り得るパラメータ名とその値に関する詳細は[SET](#)および[第18章](#)を参照してください。

注釈

新規にロールを追加するには[CREATE ROLE](#)を使用してください。 また、ロールを削除するには[DROP ROLE](#)を使用してください。

`ALTER ROLE`ではロールのメンバ資格を変更できません。 メンバ資格の変更には[GRANT](#)および[REVOKE](#)を使用してください。

このコマンドで暗号化しないパスワードを指定するときには注意しなければなりません。 パスワードはサーバに平文で送信されます。 クライアントのコマンド履歴やサーバのログにこれが残ってしまうかもしれません。 [psql](#)には`\password`コマンドがあります。 これを使用してロールのパスワードを平文のパスワードをさらすことなく変更することができます。

ロールではなくデータベースにセッションのデフォルトを結びつけることもできます。 [ALTER DATABASE](#)を参照してください。 競合する場合、データベースとロールの組み合わせに固有な設定はロール固有の設定を上書きし、ロール固有の設定はデータベース固有の設定を上書きします。

例

ロールのパスワードを変更します。

```
ALTER ROLE davide WITH PASSWORD 'hu8jmn3';
```

ロールのパスワードを削除します。

```
ALTER ROLE davide WITH PASSWORD NULL;
```

パスワードの有効期限を変更します。 以下では、2015年5月4日正午（UTCから1時間進んだタイムゾーン付き）にパスワードが無効となるように変更しています。

```
ALTER ROLE chris VALID UNTIL 'May 4 12:00:00 2015 +1';
```


パスワードの有効期限を無効にします。

```
ALTER ROLE fred VALID UNTIL 'infinity';
```

ロールに他のロールの作成権限と新しいデータベースの作成権限を与えます。

```
ALTER ROLE miriam CREATEROLE CREATEDB;
```

ロールに[maintenance_work_mem](#)パラメータ用のデフォルトとは異なる設定を与えます。

```
ALTER ROLE worker_bee SET maintenance_work_mem = 100000;
```

ロールにデータベース固有の[client_min_messages](#)パラメータ用のデフォルトとは異なる設定を与えます。

```
ALTER ROLE fred IN DATABASE devel SET client_min_messages = DEBUG;
```

互換性

ALTER ROLE文はPostgreSQLの拡張です。

関連項目

[CREATE ROLE](#), [DROP ROLE](#), [SET](#)

[前のページ](#)

ALTER OPERATOR FAMILY

[ホーム](#)

[上に戻る](#)

[次のページ](#)

ALTER SCHEMA

ALTER SCHEMA

名前

ALTER SCHEMA -- スキーマ定義を変更する

概要

```
ALTER SCHEMA name RENAME TO new_name
ALTER SCHEMA name OWNER TO new_owner
```

説明

ALTER SCHEMA はスキーマ定義を変更します。

ALTER SCHEMA を使用するにはスキーマの所有者でなければなりません。 スキーマ名を変更するには、そのデータベースのCREATE 権限を持たなければなりません。 所有者を変更するには、直接または間接的に新しい所有者ロールのメンバでなければなりません。 また、そのロールがデータベースにおいてCREATE を持たなければなりません。 （スーパーユーザがこれらの権限をすべて自動的に持つことに注意してください。）

パラメータ

name

既存のスキーマの名前です。

new_name

新しいスキーマの名前です。 pg_ から始まる名前は、システムスキーマとして予約されているため使用することができません。

new_owner

新しいスキーマの所有者です。

互換性

標準SQLにはALTER SCHEMA 文はありません。

関連項目

[CREATE SCHEMA](#), [DROP SCHEMA](#)

[前のページ](#)

ALTER ROLE

[ホーム](#)
[上に戻る](#)

[次のページ](#)

ALTER SEQUENCE

ALTER SEQUENCE

名前

ALTER SEQUENCE -- シーケンスジェネレータの定義を変更する

概要

```

ALTER SEQUENCE [ IF EXISTS ] name [ INCREMENT [ BY ] increment ]
    [ MINVALUE minvalue | NO MINVALUE ] [ MAXVALUE maxvalue | NO MAXVALUE ]
    [ START [ WITH ] start ]
    [ RESTART [ [ WITH ] restart ] ]
    [ CACHE cache ] [ [ NO ] CYCLE ]
    [ OWNED BY { table_name.column_name | NONE } ]
ALTER SEQUENCE [ IF EXISTS ] name OWNER TO new_owner
ALTER SEQUENCE [ IF EXISTS ] name RENAME TO new_name
ALTER SEQUENCE [ IF EXISTS ] name SET SCHEMA new_schema

```

説明

ALTER SEQUENCEは、既存のシーケンスジェネレータのパラメータを変更します。 ALTER SEQUENCEで指定されなかったパラメータについては、以前の設定が保持されます。

ALTER SEQUENCEを使用するには、シーケンスの所有者でなければなりません。 シーケンスのスキーマを変更するには、新しいスキーマにおけるCREATE権限も持たなければなりません。 所有者を変更するには、新しく所有者となるロールの直接または間接的なメンバでなければなりません。 またそのロールはシーケンスのスキーマ上にCREATE権限を持たなければなりません。 （これらの制限は、シーケンスの削除または再作成によりユーザが実行できないことを、所有者の変更により実行されないことを確実にするためのものです。 しかし、スーパーユーザはすべてのシーケンスの所有者を変更することができます。）

パラメータ

name

変更するシーケンスの名前です（スキーマ修飾名も可）。

IF EXISTS

シーケンスが存在しない場合にエラーとしません。 この場合、注意メッセージが発生します。

increment

INCREMENT BY *increment*句は省略可能です。 正の値が指定された時は昇順のシーケンス、負の値が指定された時は降順のシーケンスを作成します。 指定がない場合、以前の増分値が保持されます。

minvalue NO MINVALUE

MINVALUE *minvalue*はシーケンスジェネレータが生成する最小値を決定します。 NO MINVALUEが指定された場合、昇順の時は1、降順の時は-(2⁶³-1)がデフォルトになります。 どちらのオプションも指定されていないければ、現在の最小値が保持されます。

maxvalue NO MAXVALUE

MAXVALUE *maxvalue*はシーケンスが生成する最大値を決定します。 NO MAXVALUEが指定された場合、昇順の時は2⁶³-1、降順の時は-1がデフォルトになります。 どちらのオプションも指定されていないければ、現在の最大値が保持されます。

start

省略可能なSTART WITH *start*句は、記録されているシーケンスの開始値を変更します。 これは現在のシーケンス値に影響しません。 単に将来実行されるALTER SEQUENCE RESTART コマンドが使用する値を設定するだけです。

restart

省略可能なRESTART WITH *restart*句は、シーケンスの現在値を変更します。 これはis_called = false付きでsetval関

数を呼び出すことと同じです。指定した値は次の`nextval`呼出時に返されます。`restart`を付けずに`RESTART`と記述することは、`CREATE SEQUENCE`で記録、または前回`ALTER SEQUENCE START WITH`で設定された開始値を指定したことと同じです。

cache

`CACHE` *cache*句を使用すると、アクセスを高速化するために、シーケンス番号を事前に割り当て、メモリに保存しておくことができます。最小値は1です（一度に生成する値が1つだけなので、キャッシュがない状態になります）。指定がなければ、以前のキャッシュ値が保持されます。

`CYCLE`

`CYCLE`キーワードを使用すると、シーケンスが限界値（昇順の場合は`maxvalue`、降順の場合は`minvalue`）に達した時、そのシーケンスを周回させることができます。このキーワードは省略可能です。限界値に達した時、次に生成される番号は、昇順の場合は`minvalue`、降順の場合は`maxvalue`になります。

`NO CYCLE`

`NO CYCLE`キーワードが指定されると、シーケンスの限界値に達した後の`nextval`呼び出しは全てエラーとなります。このキーワードは省略可能です。`CYCLE`も`NO CYCLE`も指定されていない場合は、以前の周回動作が保持されます。

`OWNED BY` *table_name.column_name*
`OWNED BY` `NONE`

`OWNED BY`オプションにより、シーケンスは指定されたテーブル列に関連付けられ、その列（やテーブル全体）が削除されると、自動的にシーケンスも同様に削除されるようになります。指定があると、以前に指定されたシーケンスの関連は、指定された関連に置き換えられます。指定するテーブルは、シーケンスと同一所有者でなければならず、また、同一のスキーマ内に存在しなければなりません。`OWNED BY NONE`を指定することで、既存の関連は削除され、シーケンスは"独立"したものになります。

new_owner

シーケンスの新しい所有者のユーザ名です。

new_name

シーケンスの新しい名称です。

new_schema

シーケンスの新しいスキーマです。

注釈

同じシーケンスから番号を取得するトランザクションの同時実行ブロックを防ぐために、シーケンス生成パラメータに関する`ALTER SEQUENCE`の操作はロールバックできません。`ALTER SEQUENCE`による変更は即座に反映され、元に戻すことはできません。しかし、`OWNED BY`、`OWNER TO`、`RENAME TO`および`SET SCHEMA`句は通常のカatalog更新となり、ロールバックすることができます。

`ALTER SEQUENCE`は、コマンドを実行したバックエンド以外のバックエンドにおける`nextval`に対しては、すぐには効力を発揮しません。これらのバックエンドは事前に割り当てられた（キャッシュされた）値を持っており、この値を全て使い果たした後、変更されたシーケンス生成パラメータを検知します。コマンドを実行したバックエンドには、即座に変更が反映されます。

`ALTER SEQUENCE`はシーケンスの`currval`状態には影響しません。（8.3より前のPostgreSQLでは影響を与える場合があります。）

歴史的な理由により`ALTER TABLE`はシーケンスにも使用することができます。しかし、シーケンスに対して許される`ALTER TABLE`の構文は、上で示した構文と等価なものだけです。

例

`serial`シーケンスを105から再開します。

```
ALTER SEQUENCE serial RESTART WITH 105;
```

互換性

ALTER SEQUENCEは、PostgreSQLの拡張であるSTART WITH、OWNED BY、OWNER TO、RENAME TO、SET SCHEMA構文を除いて、標準SQLに従っています。

関連項目

[CREATE SEQUENCE](#), [DROP SEQUENCE](#)

[前のページ](#)
ALTER SCHEMA

[ホーム](#)
[上に戻る](#)

[次のページ](#)
ALTER SERVER

ALTER SERVER

名前

ALTER SERVER -- 外部サーバの定義を変更する

概要

```
ALTER SERVER name [ VERSION 'new_version' ]  
    [ OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ] ) ]  
ALTER SERVER name OWNER TO new_owner  
ALTER SERVER name RENAME TO new_name
```

説明

ALTER SERVER は外部サーバの定義を変更します。 最初の構文はサーバのバージョン文字列、またはサーバの一般的なオプションを変更します。（少なくとも1つの句が必要です。） 第2の構文はサーバの所有者を変更します。

サーバを変更するためには、サーバの所有者でなければなりません。 さらに所有者を変更するためには、サーバを所有し、かつ、直接または間接的に新しい所有者のロールのメンバでなければなりません。 また、サーバの外部データラップに対してUSAGE権限も必要です。（スーパーユーザはこれらの判定基準すべてを自動的に満たしていることに注意してください。）

パラメータ

name

既存のサーバの名前です。

new_version

新しいサーバのバージョンです。

OPTIONS ([ADD | SET | DROP] *option* ['*value*'] [, ...])

サーバのオプションを変更します。 ADD、SET、DROP は行う動作を指定します。 明示的な動作の指定がない場合ADDとみなされます。 オプション名は一意でなければなりません。 また、名前と値はサーバの外部データラップのライブラリを使用して検証されます。

new_owner

外部サーバの新しい所有者のユーザ名です。

new_name

外部サーバの新しい名前です。

例

fooサーバを変更し、接続オプションを追加します。

```
ALTER SERVER foo OPTIONS (host 'foo', dbname 'foodb');
```

fooサーバを変更し、バージョンとhostオプションを変更します。

```
ALTER SERVER foo VERSION '8.4' OPTIONS (SET host 'baz');
```

互換性

ALTER SERVER はISO/IEC 9075-9 (SQL/MED)に従います。 OWNER TO とRENAME構文はPostgreSQLの拡張です。

関連項目

[CREATE SERVER](#), [DROP SERVER](#)

[前のページ](#)

ALTER SEQUENCE

[ホーム](#)

[上に戻る](#)

[次のページ](#)

ALTER TABLE

ALTER TABLE

名前

ALTER TABLE -- テーブル定義を変更する

概要

```

ALTER TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
    action [, ... ]
ALTER TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
    RENAME [ COLUMN ] column_name TO new_column_name
ALTER TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
    RENAME CONSTRAINT constraint_name TO new_constraint_name
ALTER TABLE [ IF EXISTS ] name
    RENAME TO new_name
ALTER TABLE [ IF EXISTS ] name
    SET SCHEMA new_schema

```

ここで、*action*は以下のいずれかです。

```

ADD [ COLUMN ] column_name data_type [ COLLATE collation ] [ column_constraint [ ... ] ]
DROP [ COLUMN ] [ IF EXISTS ] column_name [ RESTRICT | CASCADE ]
ALTER [ COLUMN ] column_name [ SET DATA ] TYPE data_type [ COLLATE collation ] [ USING expression ]
ALTER [ COLUMN ] column_name SET DEFAULT expression
ALTER [ COLUMN ] column_name DROP DEFAULT
ALTER [ COLUMN ] column_name { SET | DROP } NOT NULL
ALTER [ COLUMN ] column_name SET STATISTICS integer
ALTER [ COLUMN ] column_name SET ( attribute_option = value [, ... ] )
ALTER [ COLUMN ] column_name RESET ( attribute_option [, ... ] )
ALTER [ COLUMN ] column_name SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN }
ADD table_constraint [ NOT VALID ]
ADD table_constraint_using_index
VALIDATE CONSTRAINT constraint_name
DROP CONSTRAINT [ IF EXISTS ] constraint_name [ RESTRICT | CASCADE ]
DISABLE TRIGGER [ trigger_name | ALL | USER ]
ENABLE TRIGGER [ trigger_name | ALL | USER ]
ENABLE REPLICA TRIGGER trigger_name
ENABLE ALWAYS TRIGGER trigger_name
DISABLE RULE rewrite_rule_name
ENABLE RULE rewrite_rule_name
ENABLE REPLICA RULE rewrite_rule_name
ENABLE ALWAYS RULE rewrite_rule_name
CLUSTER ON index_name
SET WITHOUT CLUSTER
SET WITH OIDS
SET WITHOUT OIDS
SET ( storage_parameter = value [, ... ] )
RESET ( storage_parameter [, ... ] )
INHERIT parent_table
NO INHERIT parent_table
OF type_name
NOT OF
OWNER TO new_owner
SET TABLESPACE new_tablespace

```

*table_constraint_using_index*は以下のいずれかです。

```

[ CONSTRAINT constraint_name ]
{ UNIQUE | PRIMARY KEY } USING INDEX index_name
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]

```

説明

ALTER TABLE は既存のテーブルの定義を変更します。 このコマンドには、次のような副構文があります。

ADD COLUMN

この構文を使用すると、[CREATE TABLE](#)と同じ構文を使って新しい列をテーブルに追加できます。

DROP COLUMN [IF EXISTS]

この構文を使用すると、テーブルから列を削除できます。 削除する列を含んでいるインデックスおよびテーブル制約も自

動的に削除されます。また、削除する列にテーブル以外（例えば、外部キー制約、ビューなど）が依存している場合、`CASCADE`を付ける必要があります。`IF EXISTS`が指定されている場合、もしその列がなかったとしてもエラーにはなりません。この場合は代わりに注意が出力されます。

`IF EXISTS`

テーブルが存在しない場合でもエラーとしません。この場合は注意メッセージが発行されます。

`SET DATA TYPE`

この構文を使用すると、テーブルの列の型を変更できます。その列を含むインデックスと簡単なテーブル制約は、元々与えられた式を再解析し、新しい型を使用するように自動的に変換されます。省略可能な`COLLATE`句は、新しい列の照合順を指定します。省略時の照合順は新しい列の型のデフォルトになります。`USING`句は、古い列値をどのように新しい値に計算するかを指定します。この句は省略可能です。省略された場合、デフォルトの変換は、古いデータ型から新しいデータ型への代入キャストと同じになります。古いデータ型から新しいデータ型への暗黙キャストあるいは代入キャストがない場合、`USING`句を指定しなければなりません。

`SET/DROP DEFAULT`

これらの構文を使用すると、列のデフォルト値を設定または削除できます。指定したデフォルト値は、変更後に行われる`INSERT`コマンドのみに適用されます。テーブル内の既存の行は変更されません。デフォルト値はビューに対して設定することも可能です。この場合、ビューの`ON INSERT`ルールが適用される前に、ビューの`INSERT`文にデフォルトが挿入されます。

`SET/DROP NOT NULL`

これらの構文は、列の値として`NULL`値を使用できるかどうかを設定します。`SET NOT NULL`は、その列に`NULL`値が1つもない場合にのみ設定可能です。

`SET STATISTICS`

この構文は、コマンド実行後に行なわれる`ANALYZE`操作において、列単位での統計情報収集対象を設定します。対象として、0から10000までの範囲の値を設定可能です。対象を-1に設定すると、システムのデフォルト統計情報対象（`default_statistics_target`）が使用されます。PostgreSQLの問い合わせプランナによる統計情報の使用に関する詳細は、[項14.2](#)を参照してください。

```
SET ( attribute_option = value [, ... ] )
RESET ( attribute_option [, ... ] )
```

この構文は属性単位のアプシヨンの設定または設定解除を行います。現時点では属性単位のアプシヨンで定義されているのは`n_distinct`および`n_distinct_inherited`のみです。これらのアプシヨンは、その後の`ANALYZE`操作により生成される個別値数の推定値を上書きします。`n_distinct`はテーブル自身の統計情報に影響を与え、`n_distinct_inherited`はテーブルとそれを継承した子テーブルから集めた統計情報に影響を与えます。正の値の場合、`ANALYZE`は、その列には、正確に指定された非`NULL`の個別値が存在するものとみなします。負の値の場合、この値は-1以上でなければなりません。が、`ANALYZE`は、その列内の非`NULL`の個別値はテーブルのサイズに線形であるとみなし、推定テーブルサイズに指定した値の絶対値を乗じた値が個別値数であるとみなします。たとえば、-1という値は、列内のすべての値に重複がないことを意味し、-0.5という値は個々の値は平均して2回現れることを意味します。テーブルの行数との乗算は問い合わせ計画を作詞得るまで行われませんので、テーブルサイズが変わり続けるような場合にこれは有用かもしれません。0という値を指定することで、個別値数の推定を通常に戻します。PostgreSQL問い合わせプランナにおける統計情報の使用に関しては[項14.2](#)を参照してください。

`SET STORAGE`

この構文は、列の保管モードを設定します。列をインラインで保持するか補助`TOAST`テーブルに保持するか、また、データを圧縮するかどうかを制御できます。`PLAIN`は、`integer`のような固定長の値に対して使用します。インラインで保持され、圧縮されません。`MAIN`は、インラインで保持されていて、圧縮可能なデータに使用します。`EXTERNAL`は圧縮されていない外部データに使用します。`EXTENDED`は圧縮された外部データに使用します。`EXTENDED`は、`PLAIN`以外の保管をサポートするほとんどのデータ型におけるデフォルトです。`EXTERNAL`を使用すると、非常に長い`text`および`bytea`列に対する部分文字列操作の処理速度が向上しますが、必要な保管容量が増えるというデメリットがあります。`SET STORAGE`自体はテーブルをまったく変更しないことに注意してください。以後のテーブルの更新時に遂行する戦略を設定するだけです。詳細は[項56.2](#)を参照してください。

`ADD table_constraint [ NOT VALID ]`

この構文は、`CREATE TABLE`と同じ構文に加え、現時点では外部キー制約と検査制約でのみ許される`NOT VALID`アプシヨンを使って新しい制約をテーブルに追加します。制約が`NOT VALID`と印付けされた場合、テーブル内のすべての行が制約を満たすかどうかを検証するための、時間がかかるかもしれない初期検査が飛ばされます。制約はその後の挿入や更新に対して強制されます（つまり、外部キー制約の場合、被参照テーブルに一致する行が存在しない限り失敗します。指定された検査制約に一致する新しい行が存在しない限り失敗します）。しかしデータベースは、`VALIDATE CONSTRAINT`アプシヨンを使用して検証されるまで、テーブル内のすべての行で制約が保持されていると仮定しません。

ADD table_constraint_using_index

この構文は、既存の一意性インデックスに基づき、テーブルにPRIMARY KEYまたはUNIQUE制約を新たに追加します。インデックスのすべての列がこの制約に含まれます。

このインデックスには式列や部分インデックスを持つことはできません。 またこれはデフォルトのソート順序を持つB-Treeインデックスでなければなりません。 これらの制限により、このインデックスが通常のADD PRIMARY KEYまたはADD UNIQUEコマンドにより構築されたインデックスと等価であることを確実にします。

PRIMARY KEYが指定され、インデックスの列がすでにNOT NULLと印付けされている場合、このコマンドはこうした列のそれぞれに対してALTER COLUMN SET NOT NULLの実施を試みます。 これは列にNULLが含まれないことを検証するために完全なテーブルスキャンを必要とします。 この他の場合のすべてにおいて、これが最速の操作です。

制約名が提供された場合、インデックスの名前は制約名に合うように変更されます。 提供されない場合は制約にはインデックスと同じ名前が付けられます。

このコマンドの実行後、インデックスは、インデックスが通常のADD PRIMARY KEYまたはADD UNIQUEにより構築された場合と同様に、制約により"所有"されます。 特にこの制約を削除するとインデックスも消えてしまいます。

注意: 既存のインデックスを使用した制約の追加は、長時間のテーブル更新をブロックすることなく新しい制約を追加しなければならない場合に有用になる可能性があります。 このためには、CREATE INDEX CONCURRENTLYを用いてインデックスを作成し、この構文を使用して正式の制約としてインストールしてください。 後述の例を参照してください。

VALIDATE CONSTRAINT

この構文は、過去NOT VALIDとして作成された外部キー制約または検査制約を、これらの制約を満たさない行が存在しないことを確認するためにテーブルをスキャンして、検証します。すでに有効であると記録された制約の場合は何も起こりません。 制約の初期生成時から検証を分離する利点は、制約生成時より検証の方がテーブルロックがより少ないことです。

DROP CONSTRAINT [IF EXISTS]

この構文を使用すると、テーブル上の指定した制約を削除できます。 IF EXISTSが指定された場合、その制約がなくてもエラーになりません。 この場合は代わりに注意が出力されます。

DISABLE/ENABLE [REPLICA | ALWAYS] TRIGGER

この構文を使用すると、テーブルに属するトリガの発行について設定することができます。 無効にされたトリガはシステム上に存在し続けますが、トリガイベントが発生したとしても実行されません。 遅延トリガの場合、有効無効状態の確認は、トリガ関数を実際に実行しようとする時ではなく、イベントの発生時に行われます。 名前でトリガを1つ指定して有効または無効にすることもできますし、テーブル上のすべてのトリガを指定することもできます。 また、ユーザトリガのみを指定することも可能です（このオプションは、外部キー制約、遅延可能な一意性および排他制約を実装するために使用される内部向けに生成される制約トリガを除外します。） 内部向けに生成される制約トリガを有効または無効にするにはスーパーユーザ権限が必要です。 トリガが実行されなかった場合に当然ながら制約の整合性が保証されませんので、注意して実行しなければなりません。 また、トリガ発行機構は[session replication role](#)設定変数の影響を受けます。 単に有効としたトリガでは、リプリケーションロールが"オリジン"（デフォルト）または"ローカル"の場合に発行されます。 ENABLE REPLICAとしたトリガでは、セッションが"リプリカ"モードである場合のみ発行されます。 そして、ENABLE ALWAYSとしたトリガでは、現在のリプリケーションモードに関係なく発行されます。

DISABLE/ENABLE [REPLICA | ALWAYS] RULE

この構文を使用すると、テーブルに属する書き換えルールの実行について設定することができます。 ルールは無効にしてもシステムに残りますが、問い合わせ書き換え時に適用されません。 この意味はトリガの有効化、無効化と同じです。 この設定はON SELECTルールでは無視されます。 現在のセッションがデフォルト以外のリプリケーションモードであったとしても、ビュー操作を維持するために常に適用されます。

CLUSTER ON

この構文は、以後の[CLUSTER](#)操作のデフォルトインデックスを選択します。 実際のテーブルの再クラスタ化は行いません。

SET WITHOUT CLUSTER

この構文は、テーブルから、一番最後に適用された[CLUSTER](#)インデックス指定を削除します。 以後のインデックスを指定しないクラスタ操作に影響を及ぼします。

SET WITH OIDS

この構文はoidシステム列をテーブルに追加します（[項5.4](#)を参照してください）。 テーブルがすでにOIDを持つ場合は何

も行いません。

これが、システム列ではなく `oid` という名前が付いただけの通常の列を追加する `ADD COLUMN oid oid` と同じではないことに注意してください。

`SET WITHOUT OIDS`

この構文は、テーブルから `oid` システム列を削除します。既に `oid` 列が存在しなくても警告が表示されない点を除けば、`DROP COLUMN oid RESTRICT` とまったく同等です。

`SET ( storage_parameter = value [, ... ] )`

この構文は、1つ以上のテーブルの格納パラメータを変更します。設定可能なパラメータに関しては[格納パラメータ](#)を参照してください。このコマンドによってテーブルの内容が即座に変更されない点に注意してください。パラメータによりますが、予定の効果をj得るためにテーブルを書き換える必要がある場合があります。このためには、テーブルを強制的に書き換える[VACUUM FULL](#)、[CLUSTER](#)または `ALTER TABLE` の構文のいずれかを使用してください。

注意: `CREATE TABLE` では `OIDS` を `WITH (storage_parameter)` 構文で指定することができますが、`ALTER TABLE` では `OIDS` を格納パラメータとして扱っていません。OIDの状態を変更するためには代わりに `SET WITH OIDS` および `SET WITHOUT OIDS` 構文を使用してください。

`RESET ( storage_parameter [, ... ] )`

この構文は、1つ以上の格納パラメータをデフォルト値に再設定します。`SET` 同様、テーブル全体を更新するためにテーブルの書き換えが必要になる場合があります。

`INHERIT parent_table`

この構文は、対象テーブルを指定した親テーブルの子テーブルとして追加します。その後に行われる親テーブルへの問い合わせには対象テーブルの項目も含まれます。子テーブルとして追加するためには、対象テーブルには親テーブルと同じ列がすべて含まれていなければなりません。（この他の列を持つこともできます。）これらの列のデータ型は一致している必要があり、親テーブルで `NOT NULL` 制約がある場合は、子テーブルでも同様に `NOT NULL` 制約を持たなければなりません。

また、親テーブルにおいて継承不可と印付けされている（つまり `ALTER TABLE ... ADD CONSTRAINT ... NO INHERIT` 付きで作成された）制約は無視され、これらを除き、親テーブルの `CHECK` 制約すべてについても、一致する制約が子テーブルに存在しなければなりません。一致する子テーブルの制約はすべて継承不可であってはなりません。現時点では `UNIQUE`、`PRIMARY KEY`、`FOREIGN KEY` 制約は無視されますが、将来変更されるかもしれません。

`NO INHERIT parent_table`

この構文は、指定した親テーブルから子テーブル群を削除します。親テーブルへの問い合わせでは、対象としたテーブルからのデータが含まれなくなります。

`OF type_name`

この構文は、`CREATE TABLE OF` で形成されたかのように、テーブルと複合型とを関連付けします。テーブルの列名とその型のリストは、複合型のものと正確に一致していなければなりません。`oid` システム列の有無は異なっても構いません。テーブルは他の任意のテーブルを継承してはいけません。これらの制限により `CREATE TABLE OF` がまったく同じテーブル定義を確実に許すことができます。

`NOT OF`

この構文は型から型付けされたテーブルの関連を取り除きます。

`OWNER`

この形式を使用すると、テーブル、シーケンス、またはビューの所有者を、指定したユーザに変更できます。

`SET TABLESPACE`

この構文を使用すると、テーブルのテーブル空間を指定したテーブル空間に変更し、テーブルに関連するデータファイルを新しいテーブル空間に移動することができます。テーブルにインデックスがあっても移動されません。インデックスを移動するには、別途 `SET TABLESPACE` コマンドを実行します。[CREATE TABLESPACE](#) も参照してください。

`RENAME`

`RENAME` 構文を使用すると、テーブル（もしくは、インデックス、シーケンス、ビュー）の名前、テーブルの個々の列名、テーブルの制約名を変更できます。格納されているデータへの影響はありません。

`SET SCHEMA`

この構文を使用して、テーブルを別のスキーマに移動することができます。関連するインデックスや制約、テーブル列に

より所有されるシーケンスも同様に移動されます。

`RENAME` および `SET SCHEMA` 以外の全ての操作は、結合して複数の変更リストにまとめて、並行に処理することができます。例えば、複数の列の追加、型の変更を単一のコマンドで実行することができます。これは特に巨大なテーブルでは便利です。変更のために必要なテーブル全体の走査が1回で済むからです。

`ALTER TABLE` コマンドを使用するには、変更するテーブルを所有している必要があります。テーブルのスキーマを変更するには、新しいスキーマにおける `CREATE` 権限も持たなければなりません。親テーブルに新しい子テーブルを追加するには、親テーブルも同様に所有している必要があります。また、所有者を変更するには、新しい所有ロールの直接あるいは間接的なメンバでなければならず、かつ、そのロールがテーブルのスキーマにおける `CREATE` 権限を持たなければなりません（この制限により強制的に所有者の変更が、テーブルの削除と再作成を行ってもできないことを行わないようにします。ただし、スーパーユーザはすべてのテーブルの所有者を変更することができます）。列の追加、列の型の変更、`OF` 句の使用を行うためには、データ型に対する `USAGE` 権限を持たなければなりません。

パラメータ

name

変更対象となる既存のテーブルの名前です（スキーマ修飾名も可）。テーブル名の前に `ONLY` が指定された場合、そのテーブルのみが変更されます。`ONLY` が指定されていない場合、そのテーブルおよび（もしあれば）そのテーブルを継承する全てのテーブルが更新されます。省略することもできますが、テーブル名の後に `*` を指定することで、明示的に継承するテーブルも含まれることを示すことができます。

column_name

新規または既存の列の名前です。

new_column_name

既存の列の新しい名前です。

new_name

テーブルの新しい名前です。

type

新しい列のデータ型、もしくは既存の列に対する新しいデータ型です。

table_constraint

テーブルの新しいテーブル制約です。

constraint_name

削除する既存の制約の名前です。

`CASCADE`

削除された列や制約に依存しているオブジェクト（例えば、削除された列を参照しているビューなど）を、自動的に削除します。

`RESTRICT`

依存しているオブジェクトがある場合、列または制約の削除要求を拒否します。これがデフォルトの動作です。

trigger_name

有効または無効にする単一のトリガの名前です。

`ALL`

テーブルに属するすべてのトリガを有効または無効にします。（外部キー制約、遅延可能な一意性および排他制約を実装するために使用される、内部向けに生成される制約トリガが含まれる場合、スーパーユーザ権限が必要です。）

`USER`

外部キー制約、遅延可能な一意性および排他制約を実装するために使用される、内部向けに生成されるトリガを除く、テーブルに属するトリガすべてを有効または無効にします。

index_name

指定したインデックス名でテーブルをクラスタ化するように印を付けます。

`storage_parameter`

テーブルの格納パラメータの名前です。

`value`

テーブルの格納パラメータの新しい値です。 パラメータによりこれは数値となることも文字列となることもあります。

`parent_table`

このテーブルに関連付ける、または、このテーブルから関連付けを取り除く親テーブルです。

`new_owner`

テーブルの新しい所有者のユーザ名です。

`new_tablespace`

テーブルを移動する先のテーブル空間の名前です。

`new_schema`

テーブルを移動する先のスキーマの名前です。

注釈

`COLUMN` キーワードはノイズであり、省略可能です。

`ADD COLUMN` によって列を追加した時、テーブル内の既存行に追加された列は、全てデフォルト値（`DEFAULT` が指定されていない場合は `NULL`）で初期化されます。

`NULL` ではないデフォルト値を持つ列を追加したり、既存の列の型を変更するには、テーブルとインデックス全体の書き換えが必要になります。 例外として、`USING` 句が列の内容を変更せず、かつ、古いデータ型が新しい型とバイナリ変換可能であるか新しい型全体に対する制約のないドメインである場合、テーブルの書き換えは必要ありません。 しかし影響を受ける列に対するインデックスはすべて再構築されなければなりません。 システム `oid` 列の追加や削除も同様にテーブル全体の書き換えが必要です。 テーブルが巨大な場合、テーブル、インデックスまたはその両方の再構築に非常に時間がかかる可能性があります。 また、一時的に2倍のディスク容量が必要とされます。

`CHECK` あるいは `NOT NULL` 制約を追加する時は、既存の行が制約に従うかどうかを検証するためにテーブルの走査が必要になります。

単一の `ALTER TABLE` 内に複数の変更を指定できるオプションを提供する主な理由は、複数のテーブル走査や書き換えを1回のテーブル走査にまとめることができるようにすることです。

`DROP COLUMN` 構文は、列を物理的には削除せず、SQLを操作する上で不可視にします。 このコマンドを実行した後、テーブルに挿入または更新が行われると、削除した列には `NULL` が格納されます。 したがって、列の削除は短時間でできます。 しかし、削除された列が占めていた領域がまだ回収されていないため、テーブルのディスク上のサイズはすぐには小さくなりません。 この領域は、その後既存の行が更新されるタイミングで回収されます。（システム `oid` 列を削除する場合はこれらの文は適用されません。 これは即座の書き換えが行われます。）

テーブルの書き換えを即座に強制するために、[VACUUM FULL](#)、[CLUSTER](#)、書き換えを強制する `ALTER TABLE` 構文を使用することができます。 これにより、見た目にはテーブルに有意な変更はありませんが、使用されなくなったデータが取り除かれます。

`SET DATA TYPE` の `USING` オプションでは、その行の古い値を含め、どのような式でも指定できます。 つまり、変換対象の列と同様に、その他の列も参照することができます。 そのため、一般的な変換を `SET DATA TYPE` 構文で行うことができます。 この柔軟性のため、`USING` 式は列のデフォルト値には（仮に存在していても）適用されません。 結果が定数式にならない可能性があるためです（デフォルト値は定数式でなければいけません）。 したがって、古い型から新しい型への暗黙キャストや代入キャストが存在しない場合、`USING` が指定されていても、`SET DATA TYPE` がデフォルト値の変換に失敗する可能性があります。 この場合は、`DROP DEFAULT` でデフォルト値を削除し、`ALTER TYPE` を実行した後で、`SET DEFAULT` を使用して再度適切なデフォルト値を指定してください。 変更対象の列を持つインデックスと制約も同様の配慮が必要です。

そのテーブルを継承するテーブルがある場合、子テーブルに同じ処理を実行しなければ、親テーブルに対する列の追加、列の名前、列の型、継承された制約の名前の変更を実行することはできません。 つまり、`ALTER TABLE ONLY` コマンドは受け付けられません。 この制限により、子テーブルの列が常に親テーブルと一致していることが保証されます。

再帰的な `DROP COLUMN` 操作では、子テーブルが他の親テーブルからその列を継承しておらず、かつ、独立した列定義を持っていない場合のみ、その子テーブルの列を削除します。 再帰的でない `DROP COLUMN`（例えば、`ALTER TABLE ONLY ... DROP COLUMN` など）操作では、継承された列は削除されません。 削除する代わりに、その列は継承されておらず独立して定義されているという

印を付けます。

TRIGGER、CLUSTER、OWNER および TABLESPACE は子テーブルに再帰的に伝わりません。つまり、常に ONLY が指定されているかのように動作します。制約の追加は、NO INHERIT 印がない CHECK 制約に関してのみ再帰的に伝わります。

システムカタログテーブルについては、いかなる部分の変更も許可されていません。

有効なパラメータの詳しい説明は [CREATE TABLE](#) を参照してください。 [第5章](#)に、継承に関するさらに詳しい情報があります。

例

varchar 型の列をテーブルに追加します。

```
ALTER TABLE distributors ADD COLUMN address varchar(30);
```

テーブルから列を削除します。

```
ALTER TABLE distributors DROP COLUMN address RESTRICT;
```

1つの操作で既存の2つの列の型を変更します。

```
ALTER TABLE distributors
  ALTER COLUMN address TYPE varchar(80),
  ALTER COLUMN name TYPE varchar(100);
```

USING 句を使用して、Unixタイムスタンプを持つinteger型の列を timestamp with time zone に変更します。

```
ALTER TABLE foo
  ALTER COLUMN foo_timestamp SET DATA TYPE timestamp with time zone
  USING
    timestamp with time zone 'epoch' + foo_timestamp * interval '1 second';
```

上と同じです。ただし、その列は、自動的に新しいデータ型にキャストされないデフォルト式を持ちます。

```
ALTER TABLE foo
  ALTER COLUMN foo_timestamp DROP DEFAULT,
  ALTER COLUMN foo_timestamp TYPE timestamp with time zone
  USING
    timestamp with time zone 'epoch' + foo_timestamp * interval '1 second',
  ALTER COLUMN foo_timestamp SET DEFAULT now();
```

既存の列の名前を変更します。

```
ALTER TABLE distributors RENAME COLUMN address TO city;
```

既存のテーブルの名前を変更します。

```
ALTER TABLE distributors RENAME TO suppliers;
```

既存の制約の名前を変更します。

```
ALTER TABLE distributors RENAME CONSTRAINT zipchk TO zip_check;
```

列に非NULL制約を付与します。

```
ALTER TABLE distributors ALTER COLUMN street SET NOT NULL;
```

列から非NULL制約を削除します。

```
ALTER TABLE distributors ALTER COLUMN street DROP NOT NULL;
```

テーブルとその子テーブルすべてにCHECK制約を付与します。

```
ALTER TABLE distributors ADD CONSTRAINT zipchk CHECK (char_length(zipcode) = 5);
```

そのテーブルのみに適用され、その子テーブルには適用されない検査制約を追加します。

```
ALTER TABLE distributors ADD CONSTRAINT zipchk CHECK (char_length(zipcode) = 5) NO INHERIT;
```

(検査制約はこの後作成される子テーブルにも継承されません。)

テーブルとその子テーブルからCHECK制約を削除します。

```
ALTER TABLE distributors DROP CONSTRAINT zipchk;
```

1つのテーブルのみから検査制約を削除します。


```
ALTER TABLE ONLY distributors DROP CONSTRAINT zipchk;
```

（この検査制約はすべての子テーブルで残ったままです。）

テーブルに外部キー制約を付与します。

```
ALTER TABLE distributors ADD CONSTRAINT distfk FOREIGN KEY (address) REFERENCES addresses (address);
```

テーブルに（複数列の）一意性制約を付与します。

```
ALTER TABLE distributors ADD CONSTRAINT dist_id_zipcode_key UNIQUE (dist_id, zipcode);
```

自動的に命名されるプライマリキー制約をテーブルに付与します。 1つのテーブルが持てるプライマリキーは1つだけであることを注意してください。

```
ALTER TABLE distributors ADD PRIMARY KEY (dist_id);
```

テーブルを別のテーブル空間に移動します。

```
ALTER TABLE distributors SET TABLESPACE fasttablespace;
```

テーブルを別のスキーマに移動します。

```
ALTER TABLE myschema.distributors SET SCHEMA yourschema;
```

インデックスを再構築している間の更新をブロックすることなく、プライマリキー制約を再作成します。

```
CREATE UNIQUE INDEX CONCURRENTLY dist_id_temp_idx ON distributors (dist_id);
ALTER TABLE distributors DROP CONSTRAINT distributors_pkey,
ADD CONSTRAINT distributors_pkey PRIMARY KEY USING INDEX dist_id_temp_idx;
```

互換性

（`USING INDEX`がない）`ADD`、`DROP`、`SET DEFAULT`、`SET DATA TYPE`（`USING`がない）構文は標準SQLに従います。他の構文は標準SQLに対するPostgreSQLの拡張です。また、単一の`ALTER TABLE`コマンド内に複数の操作を指定する機能もPostgreSQLの拡張です。

`ALTER TABLE DROP COLUMN`を使って、1つしか列がないテーブルから列を削除して、列がないテーブルを作成することができます。これはPostgreSQLの拡張です。SQLでは、列を持たないテーブルは認められていません。

関連項目

[CREATE TABLE](#)

[前のページ](#)

ALTER SERVER

[ホーム](#)

[上に戻る](#)

[次のページ](#)

ALTER TABLESPACE

ALTER TABLESPACE

名前

ALTER TABLESPACE -- テーブル空間の定義を変更する

概要

```
ALTER TABLESPACE name RENAME TO new_name
ALTER TABLESPACE name OWNER TO new_owner
ALTER TABLESPACE name SET ( tablespace_option = value [, ... ] )
ALTER TABLESPACE name RESET ( tablespace_option [, ... ] )
```

説明

ALTER TABLESPACE はテーブル空間の定義を変更します。

ALTER TABLESPACE を使用するにはテーブル空間の所有者でなければなりません。所有者を変更するには、直接または間接的に新しい所有者ロールのメンバでなければなりません。（スーパーユーザがこれらの権限を自動的に持つことに注意してください。）

パラメータ

name

既存のテーブル空間の名前です。

new_name

新しいテーブル空間の名前です。 pg_ から始まる名前は、システムのテーブル空間用に予約されているため使用することができません。

new_owner

新しいテーブル空間の所有者です。

tablespace_parameter

設定または設定解除するテーブル空間パラメータです。現時点では seq_page_cost および random_page_cost パラメータのみが利用可能です。特定のテーブル空間にいずれかの値を設定することにより、同一の名前の設定パラメータ (seq_page_cost、random_page_cost 参照) で決定されるテーブル空間内のテーブルからページを読み取る、プランナの通常のコスト推定値が上書きされます。これはあるテーブル空間が他の I/O サブシステムに比べ低速または高速なディスク上にある場合に有用となるかもしれません。

例

index_space テーブル空間を fast_raid という名前に変更します。

```
ALTER TABLESPACE index_space RENAME TO fast_raid;
```

index_space テーブル空間の所有者を変更します。

```
ALTER TABLESPACE index_space OWNER TO mary;
```

互換性

標準 SQL には ALTER TABLESPACE 文はありません。

関連項目

[CREATE TABLESPACE](#), [DROP TABLESPACE](#)

[前のページ](#)

ALTER TABLE

[ホーム](#)

[上に戻る](#)

[次のページ](#)

ALTER TEXT SEARCH
CONFIGURATION

ALTER TEXT SEARCH CONFIGURATION

名前

ALTER TEXT SEARCH CONFIGURATION -- テキスト検索設定の定義を変更する

概要

```
ALTER TEXT SEARCH CONFIGURATION name
    ADD MAPPING FOR token_type [, ... ] WITH dictionary_name [, ... ]
ALTER TEXT SEARCH CONFIGURATION name
    ALTER MAPPING FOR token_type [, ... ] WITH dictionary_name [, ... ]
ALTER TEXT SEARCH CONFIGURATION name
    ALTER MAPPING REPLACE old_dictionary WITH new_dictionary
ALTER TEXT SEARCH CONFIGURATION name
    ALTER MAPPING FOR token_type [, ... ] REPLACE old_dictionary WITH new_dictionary
ALTER TEXT SEARCH CONFIGURATION name
    DROP MAPPING [ IF EXISTS ] FOR token_type [, ... ]
ALTER TEXT SEARCH CONFIGURATION name RENAME TO new_name
ALTER TEXT SEARCH CONFIGURATION name OWNER TO new_owner
ALTER TEXT SEARCH CONFIGURATION name SET SCHEMA new_schema
```

説明

ALTER TEXT SEARCH CONFIGURATIONはテキスト検索設定の定義を変更します。 トークン型から辞書への対応付けの変更、または、設定名称の変更、設定の所有者の変更を行うことができます。

ALTER TEXT SEARCH CONFIGURATIONを使用するためには、設定の所有者でなければなりません。

パラメータ

name

既存のテキスト検索設定の名称（スキーマ修飾可）です。

token_type

設定のパーサが発行するトークン型の名称です。

dictionary_name

指定したトークン型（複数可）で考慮されるテキスト検索辞書の名称です。 複数の辞書が列挙された場合、指定された順序で考慮されます。

old_dictionary

対応付けにて置換されるテキスト検索辞書の名称です。

new_dictionary

*old_dictionary*を置き換えるテキスト検索辞書の名称です。

new_name

新しいテキスト検索設定の名称です。

new_owner

新しいテキスト検索設定の所有者です。

new_schema

テキスト検索設定の新しいスキーマです。

ADD MAPPING FOR構文は指定したトークン型で考慮される辞書のリストをインストールします。 既にそのトークン型に対する対応付けが存在する場合はエラーになります。 ALTER MAPPING FOR構文は、まず既存の対象トークン型に対する対応付けを削除す

る点を除き、同一です。ALTER MAPPING REPLACE構文は、すべてのold_dictionaryをnew_dictionaryで置き換えます。FORがあれば、これは指定したトークン型に対してのみ行われ、なければ、設定におけるすべての対応付けに対して行われます。DROP MAPPING構文は指定したトークン型（複数可）に対するすべての辞書を削除します。この結果、このテキスト検索設定ではこれらの型のトークンが無視されるようになります。IF EXISTSがない限り、トークン型に対する対応付けが存在しない場合はエラーになります。

例

次の例は、my_config内でenglishが使用されるすべてに対し、english辞書をswedish辞書で置換します。

```
ALTER TEXT SEARCH CONFIGURATION my_config
ALTER MAPPING REPLACE english WITH swedish;
```

互換性

標準SQLにはALTER TEXT SEARCH CONFIGURATION文はありません。

関連項目

[CREATE TEXT SEARCH CONFIGURATION](#), [DROP TEXT SEARCH CONFIGURATION](#)

[前のページ](#)
ALTER TABLESPACE

[ホーム](#)
[上に戻る](#)

[次のページ](#)
ALTER TEXT SEARCH DICTIONARY

ALTER TEXT SEARCH DICTIONARY

名前

ALTER TEXT SEARCH DICTIONARY -- テキスト検索辞書の定義を変更する

概要

```
ALTER TEXT SEARCH DICTIONARY name (  
    option [ = value ] [, ... ]  
)  
ALTER TEXT SEARCH DICTIONARY name RENAME TO new_name  
ALTER TEXT SEARCH DICTIONARY name OWNER TO new_owner  
ALTER TEXT SEARCH DICTIONARY name SET SCHEMA new_schema
```

説明

ALTER TEXT SEARCH DICTIONARY はテキスト検索辞書の定義を変更します。辞書のテンプレート固有のオプションの変更、辞書の名称、辞書の所有者を変更することができます。

ALTER TEXT SEARCH DICTIONARY を使用するには、辞書の所有者でなければなりません。

パラメータ

name

既存のテキスト検索辞書の名称（スキーマ修飾可）です。

option

この辞書に設定される、テンプレート固有のオプションの名称です。

value

テンプレート固有のオプションで使用される、新しい値です。等号記号と値が省略された場合、そのオプションの以前の設定は辞書から削除され、デフォルト値の使用を許可します。

new_name

新しいテキスト検索辞書の名称です。

new_owner

新しいテキスト検索辞書の所有者です。

new_schema

テキスト検索辞書の新しいスキーマです。

テンプレート固有のオプションは任意の順序で記述することができます。

例

次の例は、雪だるま式に増加する辞書のストップワードを変更します。他のパラメータはそのまま変更されません。

```
ALTER TEXT SEARCH DICTIONARY my_dict ( StopWords = newrussian );
```

次の例は、言語オプションをドイツ語に変更し、ストップワードオプションを完全に消去します。

```
ALTER TEXT SEARCH DICTIONARY my_dict ( language = dutch, StopWords );
```

次の例は、実際には何も変更しませんが、辞書の定義を"更新"します。

```
ALTER TEXT SEARCH DICTIONARY my_dict ( dummy );
```

（無効なオプションが存在しても、オプションを消去するコードでエラーとしないため、これは動作します。） この手品は、辞書用の設定ファイルを変更する際に有用です。 このALTERにより既存のデータベースセッションは強制的に設定ファイルを再度読み込みます。 こうしないと、以前に読み込んだ設定ファイルを再読み込みすることはありません。

互換性

標準SQLにはALTER TEXT SEARCH DICTIONARY文はありません。

関連項目

[CREATE TEXT SEARCH DICTIONARY](#), [DROP TEXT SEARCH DICTIONARY](#)

前のページ	ホーム	次のページ
ALTER TEXT SEARCH CONFIGURATION	上に戻る	ALTER TEXT SEARCH PARSER

ALTER TEXT SEARCH PARSER

名前

ALTER TEXT SEARCH PARSER -- テキスト検索パーサの定義を変更する

概要

```
ALTER TEXT SEARCH PARSER name RENAME TO new_name
ALTER TEXT SEARCH PARSER name SET SCHEMA new_schema
```

説明

ALTER TEXT SEARCH PARSER はテキスト検索パーサの定義を変更します。現時点では、パーサ名称の変更機能のみがサポートされています。

ALTER TEXT SEARCH PARSER を使用するためにはスーパーユーザでなければなりません。

パラメータ

name

既存のテキスト検索パーサの名称（スキーマ修飾可）です。

new_name

新しいテキスト検索パーサの名称です。

new_schema

全文検索パーサの新しいスキーマです。

互換性

標準SQLにはALTER TEXT SEARCH PARSER 文はありません。

関連項目

[CREATE TEXT SEARCH PARSER](#), [DROP TEXT SEARCH PARSER](#)

[前のページ](#)

ALTER TEXT SEARCH DICTIONARY

[ホーム](#)
[上に戻る](#)

[次のページ](#)

ALTER TEXT SEARCH TEMPLATE

ALTER TEXT SEARCH TEMPLATE

名前

ALTER TEXT SEARCH TEMPLATE -- テキスト検索テンプレートの定義を変更する

概要

```
ALTER TEXT SEARCH TEMPLATE name RENAME TO new_name
ALTER TEXT SEARCH TEMPLATE name SET SCHEMA new_schema
```

説明

ALTER TEXT SEARCH TEMPLATE はテキスト検索テンプレートの定義を変更します。現時点では、テンプレート名称の変更機能のみがサポートされています。

ALTER TEXT SEARCH TEMPLATE を使用するためにはスーパーユーザでなければなりません。

パラメータ

name

既存のテキスト検索テンプレートの名称（スキーマ修飾可）です。

new_name

新しいテキスト検索テンプレートの名称です。

new_schema

テキスト検索テンプレートの新しいスキーマです。

互換性

標準SQLにはALTER TEXT SEARCH TEMPLATE 文はありません。

関連項目

[CREATE TEXT SEARCH TEMPLATE](#), [DROP TEXT SEARCH TEMPLATE](#)

[前のページ](#)

ALTER TEXT SEARCH PARSER

[ホーム](#)
[上に戻る](#)

[次のページ](#)

ALTER TRIGGER

ALTER TRIGGER

名前

ALTER TRIGGER -- トリガ定義を変更する

概要

```
ALTER TRIGGER name ON table_name RENAME TO new_name
```

説明

ALTER TRIGGER を使用すると、既存のトリガのプロパティを変更できます。 RENAME 句を使用すると、トリガ定義を変更せずに、指定されたテーブルのトリガ名を変更することができます。

トリガのプロパティを変更するには、トリガで処理されるテーブルを所有している必要があります。

パラメータ

name

変更の対象となる既存のトリガの名前です。

table_name

このトリガで処理されるテーブルの名前です。

new_name

既存のトリガの新しい名前です。

注釈

トリガを一時的に有効または無効にする機能は [ALTER TABLE](#) が提供します。 ALTER TRIGGER ではありません。 ALTER TRIGGER には、一度にテーブルのトリガを有効または無効にするオプションを表現する、簡便な方法がないからです。

例

既存のトリガの名前を変更します。

```
ALTER TRIGGER emp_stamp ON emp RENAME TO emp_track_chgs;
```

互換性

ALTER TRIGGER は、標準SQLに対するPostgreSQLの拡張です。

関連項目

[ALTER TABLE](#)

[前のページ](#)

ALTER TEXT SEARCH TEMPLATE

[ホーム](#)
[上に戻る](#)

[次のページ](#)
ALTER TYPE

ALTER TYPE

名前

ALTER TYPE -- 型定義を変更する

概要

```
ALTER TYPE name action [, ... ]
ALTER TYPE name OWNER TO new_owner
ALTER TYPE name RENAME ATTRIBUTE attribute_name TO new_attribute_name
ALTER TYPE name RENAME TO new_name [ CASCADE | RESTRICT ]
ALTER TYPE name SET SCHEMA new_schema
ALTER TYPE name ADD VALUE new_enum_value [ { BEFORE | AFTER } existing_enum_value ]
```

ここで`action`は以下のいずれかです。

```
ADD ATTRIBUTE attribute_name data_type [ COLLATE collation ] [ CASCADE | RESTRICT ]
DROP ATTRIBUTE [ IF EXISTS ] attribute_name [ CASCADE | RESTRICT ]
ALTER ATTRIBUTE attribute_name [ SET DATA ] TYPE data_type [ COLLATE collation ] [ CASCADE | RESTRICT ]
```

説明

ALTER TYPEは既存の型の定義を変更します。 複数の副構文があります。

ADD ATTRIBUTE

この構文は、[CREATE TYPE](#)と同じ構文を用いて、複合型に新しい属性を追加します。

DROP ATTRIBUTE [IF EXISTS]

この構文は複合型から属性を削除します。 IF EXISTSが指定された時にその属性が存在しなくてもエラーにはなりません。 この場合は代わりに注意が発せられます。

SET DATA TYPE

この構文は複合型の属性の型を変更します。

OWNER

この構文は型の所有者を変更します。

RENAME

この構文は型の名前または複合型の個々の属性の名前を変更します。

SET SCHEMA

この構文は型を他のスキーマに移動します。

ADD VALUE [BEFORE | AFTER]

この構文は列挙型に新しい値を追加します。 BEFOREまたはAFTERによって列挙型の順番における新しい値の場所が指定されていない場合、新しい項目は値リストの最後に格納されます。

CASCADE

自動的に変更される型で型付けされたテーブルとその子テーブルにこの操作を伝播します。

RESTRICT

変更対象の型がテーブルの型付けに使用されている場合に拒絶します。 これがデフォルトです。

ADD ATTRIBUTE、DROP ATTRIBUTE、ALTER ATTRIBUTE操作は複数の変更リストにまとめて、並行して適用することができます。例えば、複数の属性の追加、複数の属性の変更、またはその両方を1つのコマンドで実行することができます。

ALTER TYPEを使用するには型の所有者でなければなりません。 型のスキーマを変更するには、新しいスキーマにおけるCREATE権限も必要です。 所有者を変更するには、直接または間接的に新しい所有者ロールのメンバでなければなりません。 また、そのロールが型のスキーマにおいてCREATEを持たなければなりません。 （この制限により、強制的に、所有者の変更が型の削除と再

作成で行うことができない何らかの処理を行わないようになります。しかし、スーパーユーザはすべての型の所有者を変更することができます。） 属性を追加または属性の型の変更を行うためには、そのデータ型に対するUSAGE権限を持たなければなりません。

パラメータ

name

変更対象の既存の型の名前です（スキーマ修飾名も可）。

new_name

新しい型の名前です。

new_owner

新しい型の所有者のユーザ名です。

new_schema

型の新しいスキーマです。

attribute_name

追加、変更または削除する属性の名前です。

new_attribute_name

変名する属性の新しい名前です。

data_type

追加する属性のデータ型、または、変更する属性の新しい型です。

new_enum_value

列挙型リストの値に追加する新しい値です。 すべての列挙型のリテラル同様、引用符を付けなければなりません。

existing_enum_value

列挙型の並び順序において新しい値をその直前または直後に追加する、既存の列挙型の値です。 すべての列挙型のリテラル同様、引用符を付けなければなりません。

注意

ALTER TYPE ... ADD VALUE（列挙型に新しい値を追加する構文）はトランザクションブロック内で実行することはできません。

追加された列挙型の値を含む比較は、列挙型の元々の要素のみを含む比較よりも低速になることがあります。 通常これは、新しい値のソート位置がリストの最後ではなく BEFORE または AFTER を用いて設定された場合のみで起こります。 しかし最後に新しい値が追加された場合であっても起こる可能性があります。（これは、OIDカウンタが元の列挙型を作成してから"周回"した場合に起こります。） この低下は通常は大きくありません。 しかしこれが問題であれば、列挙型を削除し再作成する、あるいはデータベースをダンプし再ロードすることで最適な性能まで戻すことができます。

例

データ型の名前を変更します。

```
ALTER TYPE electronic_mail RENAME TO email;
```

email 型の所有者をjoeに変更します。

```
ALTER TYPE email OWNER TO joe;
```

email 型のスキーマをcustomersに変更します。

```
ALTER TYPE email SET SCHEMA customers;
```

型に新しい属性を追加します。

```
ALTER TYPE compfoo ADD ATTRIBUTE f3 int;
```

列挙型の特定のソート位置に新しい値を追加します。

```
ALTER TYPE colors ADD VALUE 'orange' AFTER 'red';
```

互換性

属性の追加および削除を行う構文は標準SQLの一部です。 他の構文はPostgreSQLの拡張です。

関連項目

[CREATE TYPE](#), [DROP TYPE](#)

[前のページ](#)
ALTER TRIGGER

[ホーム](#)
[上に戻る](#)

[次のページ](#)
ALTER USER

ALTER USER

名前

ALTER USER -- データベースロールを変更する

概要

```
ALTER USER name [ [ WITH ] option [ ... ] ]
```

*option*は次の通りです。

```
    SUPERUSER | NOSUPERUSER
| CREATEDB | NOCREATEDB
| CREATEROLE | NOCREATEROLE
| CREATEUSER | NOCREATEUSER
| INHERIT | NOINHERIT
| LOGIN | NOLOGIN
| REPLICATION | NOREPLICATION
| CONNECTION LIMIT connlimit
| [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
| VALID UNTIL 'timestamp'
```

```
ALTER USER name RENAME TO new_name
```

```
ALTER USER name SET configuration_parameter { TO | = } { value | DEFAULT }
ALTER USER name SET configuration_parameter FROM CURRENT
ALTER USER name RESET configuration_parameter
ALTER USER name RESET ALL
```

説明

ALTER USERは[ALTER ROLE](#)の別名になりました。

互換性

ALTER USER文は、PostgreSQLの拡張です。 標準SQLでは、ユーザの定義は実装に任されています。

関連項目

[ALTER ROLE](#)

[前のページ](#)

ALTER TYPE

[ホーム](#)
[上に戻る](#)

[次のページ](#)

ALTER USER MAPPING

ALTER USER MAPPING

名前

ALTER USER MAPPING -- ユーザマップの定義を変更する

概要

```
ALTER USER MAPPING FOR { user_name | USER | CURRENT_USER | PUBLIC }  
    SERVER server_name  
    OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ] )
```

説明

ALTER USER MAPPING はユーザマップの定義を変更します。

外部サーバの所有者は任意のユーザに対するそのサーバ向けのユーザマップを変更することができます。 また、サーバ上でUSAGE権限がユーザに付与されていた場合、ユーザは自身の持つユーザ名に対応するユーザマップを変更することができます。

パラメータ

user_name

対応付けするユーザ名です。 CURRENT_USER と USER は現在のユーザ名に一致します。 PUBLIC は現在および将来にシステム上に存在するすべてのユーザに一致させるために使用します。

server_name

ユーザマップのサーバ名です。

OPTIONS ([ADD | SET | DROP] *option* ['*value*'] [, ...])

ユーザマップのオプションを変更します。 新しいオプションは過去に指定されたオプションをすべて上書きします。 ADD、SET、DROP は実行する動作を指定します。 明示的な動作指定がない場合、ADD とみなされます。 オプション名は一意でなければなりません。 またオプションはサーバの外部データラッパにより検証されます。

例

サーバfooのユーザマップbobのパスワードを変更します。

```
ALTER USER MAPPING FOR bob SERVER foo OPTIONS (user 'bob', password 'public');
```

互換性

ALTER USER MAPPING はISO/IEC 9075-9(SQL/MED)に従います。 小さな構文上の問題があります。 標準ではFORキーワードを省略します。 CREATE USER MAPPING と DROP USER MAPPING ではFORを似たような位置で使用し、またIBM DB2（他の主なSQL/MED実装になっています）ではALTER USER MAPPING で必要としていますので、PostgreSQLは、一貫性と相互運用性を目的に、標準と違いを持たせています。

関連項目

[CREATE USER MAPPING](#), [DROP USER MAPPING](#)

ALTER VIEW

名前

ALTER VIEW -- ビュー定義を変更する

概要

```
ALTER VIEW [ IF EXISTS ] name ALTER [ COLUMN ] column_name SET DEFAULT expression
ALTER VIEW [ IF EXISTS ] name ALTER [ COLUMN ] column_name DROP DEFAULT
ALTER VIEW [ IF EXISTS ] name OWNER TO new_owner
ALTER VIEW [ IF EXISTS ] name RENAME TO new_name
ALTER VIEW [ IF EXISTS ] name SET SCHEMA new_schema
ALTER VIEW [ IF EXISTS ] name SET ( view_option_name [= view_option_value] [, ... ] )
ALTER VIEW [ IF EXISTS ] name RESET ( view_option_name [, ... ] )
```

説明

ALTER VIEW はビューの各種補助属性を変更します。（ビューを定義する問い合わせを変更したい場合は CREATE OR REPLACE VIEW を使用してください。）

ALTER VIEW を使用するためには、ビューの所有者でなければなりません。またビューのスキーマを変更するためには、新しいスキーマ上に CREATE 権限を持たなければなりません。さらに所有者を変更するためには、新しく所有者となるロールの直接あるいは間接的なメンバでなければならず、かつ、そのロールはビューのスキーマ上に CREATE 権限を持たなければなりません。（これらの制限は、ビューの削除または再作成によりユーザが実行できないことを、所有者の変更により実行されないことを確実にするためのものです。しかし、スーパーユーザはすべてのビューの所有者を変更することができます。）

パラメータ

name

既存のビューの名前（スキーマ修飾可）です。

IF EXISTS

ビューが存在する場合にエラーとしません。この場合には注意メッセージが発行されます。

SET/DROP DEFAULT

この構文は列のデフォルト値を設定または削除します。INSERT 文に該当列に対する値が指定されていない場合、ビューの ON INSERT ルールが適用される前に、ビューの列に関連づいたデフォルト値はビューに対する INSERT 文に挿入されます。

new_owner

ビューの新しい所有者のユーザ名です。

new_name

ビューの新しい名前です。

new_schema

ビューの新しいスキーマです。

view_option_name

設定または再設定するビューオプションの名前です。

view_option_name

ビューオプションの新しい値です。

注釈

歴史的な理由により、`ALTER TABLE`をビューに対して使用することができます。ただし、使用可能な構文は上記のビューに対して許される構文に対応する`ALTER TABLE`の構文のみです。

例

ビュー`foo`の名前を`bar`に変更します。

```
ALTER VIEW foo RENAME TO bar;
```

互換性

`ALTER VIEW`は標準SQLに対するPostgreSQLの拡張です。

関連項目

[CREATE VIEW](#), [DROP VIEW](#)

前のページ ALTER USER MAPPING	ホーム 上に戻る	次のページ ANALYZE

ANALYZE

名前

ANALYZE -- データベースに関する統計を集計する

概要

```
ANALYZE [ VERBOSE ] [ table_name [ ( column_name [, ...] ) ] ]
```

説明

ANALYZE はデータベース内のテーブルの内容に関する統計情報を集計し、その結果を [pg_statistic](#) システムテーブルに保存します。問い合わせプランナが最も効率の良い問い合わせの実行計画を決定する際、この統計情報が使用されます。

パラメータがない場合、ANALYZE は現在のデータベース内の全てのテーブルを検査します。パラメータがある場合、ANALYZE は指定されたテーブルのみを検査します。さらに列名のリストを与え、その列の統計情報のみを更新することも可能です。

パラメータ

VERBOSE

進行状況の表示を有効にします。

table_name

解析の対象とするテーブルの名前です（スキーマ修飾名も可）。省略された場合、現在のデータベースの中のすべての通常のテーブル（外部テーブル以外）が解析されます。

column_name

解析の対象とする列名です。デフォルトは全ての列です。

出力

VERBOSE が指定された場合、ANALYZE は進捗メッセージとして処理中のテーブルを表示します。さらに、テーブルについての各種統計情報も表示されます。

注釈

外部テーブルは明示的に選択された場合にのみ解析されます。すべての外部データラッパがANALYZEをサポートしているとは限りません。テーブルのラッパがANALYZEをサポートしない場合、コマンドは警告を出力し、何も行いません。

デフォルトのPostgreSQLの設定では、自動バキュームデーモン（[項23.1.6](#)参照）が、データが最初にロードされた時や通常の操作を通して変更された時にテーブルの自動解析まで面倒をみます。もし自動バキュームが無効にしているならばANALYZEは定期的に、もしくは、テーブルの内容に大きな変更がある度に行うことを推奨します。統計情報が正確であれば、プランナが最も適切な問い合わせ計画を選択できるようになります。これによって、問い合わせ処理の速度が向上します。読み取りの多いデータベースでは、[VACUUM](#)とANALYZEは、1日1回、データベースがあまり使用されていない時間帯に実行することが一般的です。（非常に更新が激しい場合、これでは十分ではありません。）

ANALYZE は、対象とするテーブルの読み取りロックのみを必要とします。したがって、そのテーブルに対する他の操作と並行して実行することができます。

通常、ANALYZE によって集計される統計情報には、各列の典型的な値と各列のデータ分布の概要を示す度数分布が含まれます。ANALYZE によってあまり意味がないとみなされた場合（例えば、一意性制約が付加された列では、典型的な値というものは存在しません）や、列のデータ型が適切な演算子をサポートしていない場合は、片方もしくは両方の情報を省略することができます。[第23章](#)に、統計情報についての詳細が記載されています。

巨大なテーブルでは、`ANALYZE`は、全ての行を検査するのではなく、テーブルの中からランダムにサンプルを取り出して使用します。これによって、非常に巨大なテーブルであっても短時間で解析することが可能です。しかし、このようにして得られた統計情報はおよそそのものでしかなく、テーブルの内容に変更がなくても`ANALYZE`を実行する度に変化することに注意してください。これにより、`EXPLAIN`が表示する、プランナの推定コストも多少変化する可能性があります。稀に、このような不確定要素のせいで、プランナが`ANALYZE`を実行した後に異なる問い合わせ計画を選択してしまうことがあります。これを防止するには、以下に示すように`ANALYZE`で収集される統計情報の量を増やしてください。

`default_statistics_target`設定パラメータ変数を調整するか、もしくは`ALTER TABLE ... ALTER COLUMN ... SET STATISTICS`を使用して統計対象を列単位に設定し、解析を列ベースにすることで、解析の範囲を制御することができます（詳しくは[ALTER TABLE](#)を参照してください）。対象値として設定するのは、典型的な値のリストにおけるエントリ数の最大値と度数分布のビンの最大数です。デフォルトの対象値は100です。しかし、この値は、プランナの推定精度と`ANALYZE`の処理時間、`pg_statistic`の占める容量とのトレードオフによって変更可能です。対象値を0に設定すると、その列に関する統計情報の集計は無効になります。決して`WHERE`句、`GROUP BY`句、`ORDER BY`句に使用されない列に対しては、このような設定が有用です。プランナにとってそのような列の統計情報は不要だからです。

解析する列の統計情報対象値の最大値によって、統計情報を作成するために抽出する行数が決定します。対象値を大きくすると、比例して、`ANALYZE`に要する時間とディスク容量が増加します。

`ANALYZE`で推定される値の1つは各列に出現する個別値の個数です。行の部分集合のみしか検査されませんので、統計情報の対象をできる限り大きくしたとしても、この推定値はかなり不正確になることが時々あり得ます。この不正確性が問い合わせ計画を劣化させるような場合、より正確な値を手作業で求めることができ、そして、`ALTER TABLE ... ALTER COLUMN ... SET (n_distinct = ...)`でインストールすることができます（[ALTER TABLE](#)参照）。

解析中のテーブルが複数の子テーブルを持つ場合、`ANALYZE`は2回統計情報を収集します。1回目は親テーブルのみのテーブル行を対象とし、2回目では親テーブルの行とそのすべての子テーブルの行を対象とします。継承ツリー全体をたどる問い合わせの計画作成では、この2回目の統計情報群が必要とされます。しかし自動バキュームデーモンでは、自動的に解析を行うかどうかを決定する際に親テーブル上の挿入や更新のみを考慮します。このテーブルへの挿入や更新がほとんどなければ、継承関係に対する統計情報は手作業で`ANALYZE`を実行しない限り最新状態にはなりません。

解析しようとするテーブルが完全に空である場合、`ANALYZE`はそのテーブルに対する新しい解析情報を更新しません。これまでの統計情報はすべて保持されます。

互換性

標準SQLには`ANALYZE`文はありません。

関連項目

[VACUUM](#), [vacuumdb](#), [項18.4.4](#), [項23.1.6](#)

[前のページ](#)
[ALTER VIEW](#)

[ホーム](#)
[上に戻る](#)

[次のページ](#)
[BEGIN](#)

BEGIN

名前

BEGIN -- トランザクションブロックを開始する

概要

```
BEGIN [ WORK | TRANSACTION ] [ transaction_mode [, ...] ]
```

*transaction_mode*は以下のいずれかです。

```
ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED }  
READ WRITE | READ ONLY  
[ NOT ] DEFERRABLE
```

説明

BEGINはトランザクションブロックを初期化します。BEGINコマンド以降の文は全て、明示的なCOMMITもしくはROLLBACKが与えられるまで、単一のトランザクションの中で実行されます。デフォルト（BEGINがない場合）は、PostgreSQLはトランザクションを"自動コミット"で実行します。つまり、各文はそれぞれ固有のトランザクションの中で実行され、文の終わりで暗黙的にコミットが実行されます（これは実行が成功した場合です。失敗した場合はロールバックされます）。

トランザクションブロック内では、文はより迅速に実行されます。なぜなら、トランザクションの開始やコミットは、CPUとディスクにかなり高い負荷をかけるからです。また、1つのトランザクションで複数の文を実行することは、複数の関連するテーブルを更新する際、確実に一貫性を保つためにも役立ちます。関連する更新の中に完了していないものが存在する中間的な状態が、他のセッションから参照できなくなるからです。

隔離レベル、読み書きモード、または遅延モードが指定されている場合、新しいトランザクションは、[SET TRANSACTION](#)が実行された時と同様の特性を持ちます。

パラメータ

WORK
TRANSACTION

省略可能なキーワードです。これらは何も効果がありません。

BEGIN文のその他のパラメータについては、[SET TRANSACTION](#)を参照してください。

注釈

[START TRANSACTION](#)にはBEGINと同じ機能があります。

トランザクションを終了させるにはCOMMITまたはROLLBACKを使用してください。

トランザクションブロック内でBEGINを発行すると、警告メッセージが表示されます。ただし、トランザクションの状態には影響ありません。トランザクションブロック内でトランザクションを入れ子にするには、セーブポイントを使用してください（詳しくは[SAVEPOINT](#)を参照してください）。

後方互換性の保持のため、連続する*transaction_modes*間のカンマは省略することができます。

例

トランザクションブロックを開始します。

```
BEGIN;
```

互換性

`BEGIN`はPostgreSQLの拡張です。 標準SQLの[START TRANSACTION](#)コマンドと等価です。 このマニュアルページには互換性に関する詳細な情報が含まれます。

`DEFERRABLE transaction_mode`はPostgreSQLの言語拡張です。

埋め込みSQLでは、`BEGIN`というキーワードを異なった目的に使用しています。 データベースアプリケーションを移植する時は、トランザクションの意味で使用されているのかどうかに注意してください。

関連項目

[COMMIT](#), [ROLLBACK](#), [START TRANSACTION](#), [SAVEPOINT](#)

[前のページ](#)
ANALYZE

[ホーム](#)
[上に戻る](#)

[次のページ](#)
CHECKPOINT

CHECKPOINT

名前

CHECKPOINT -- トランザクションログのチェックポイントを強制的に実行する

概要

CHECKPOINT

説明

チェックポイントとは、ログ内の情報を反映するために全てのデータファイルを更新する、トランザクションログのある一時点を指します。チェックポイントによって、全てのデータファイルがディスクに書き出されます。チェックポイントの間に何が起きるかについては、[項29.4](#)を参照してください。

CHECKPOINT コマンドは、コマンドが発行された時、([項18.5.2](#)によって制御される)システムによって予定された通常のチェックポイントを待たず、即座に強制的にチェックポイント処理を行います。通常の運用時にCHECKPOINTが使用されることは想定していません。

リカバリ中に実行された場合、CHECKPOINTは新しくチェックポイントを書き出さずにリスタートポイント([項29.4](#)参照)を強制します。

CHECKPOINTを実行できるのはスーパーユーザのみです。

互換性

CHECKPOINT コマンドは、PostgreSQLの拡張です。

CLOSE

名前

CLOSE -- カーソルを閉じる

概要

```
CLOSE { name | ALL }
```

説明

CLOSE は、開いたカーソルに関連するリソースを解放します。カーソルが閉じられた後は、そのカーソルに対する操作はできません。カーソルは必要がなくなった時点で閉じるべきです。

トランザクションがCOMMITもしくはROLLBACKで終了された時点で、開いている保持不可能カーソルは全て暗黙的に閉じられます。ROLLBACKにより保持可能カーソルを作成したトランザクションをアボートした場合、この保持可能カーソルは暗黙的に閉じられます。作成したトランザクションが正常にコミットされた場合、保持可能カーソルは明示的にCLOSEが実行されるまで、あるいは、クライアントとの接続が切断されるまで、開いたままになります。

パラメータ

name

閉じる対象となる、現在開いているカーソルの名前です。

ALL

すべてのカーソルを閉じます。

注釈

PostgreSQLには明示的なOPENカーソル文がありません。カーソルは宣言された時に開いたとみなされます。カーソルの宣言にはDECLARE文を使用してください。

[pg_cursors](#) システムビューを問い合わせることにより利用可能なすべてのカーソルを確認することができます。

カーソルがセーブポイントの後に閉ざされ、後にロールバックした場合には、CLOSEはロールバックされません。つまり、そのカーソルは閉じたままとなります。

例

liahonaカーソルを閉じます。

```
CLOSE liahona;
```

互換性

CLOSE は標準SQLと完全な互換性を持ちます。ただし、CLOSE ALLはPostgreSQLの拡張です。

関連項目

[DECLARE](#), [FETCH](#), [MOVE](#)

CLUSTER

名前

CLUSTER -- インデックスに従ってテーブルをクラスタ化する

概要

```
CLUSTER [VERBOSE] table_name [ USING index_name ]
CLUSTER [VERBOSE]
```

説明

CLUSTERは、*index_name*で指定されたインデックスに基づき、*table_name*で指定されたテーブルをクラスタ化するように、PostgreSQLに指示します。このインデックスは前もって*table_name*で定義されていなければなりません。

テーブルがクラスタ化されると、それぞれのテーブルはインデックス情報に基づいて物理的に並べ直されます。クラスタ化は、1回限りの操作です。クラスタ化後にテーブルが更新されても、その変更はクラスタ化されません。つまり、新規に追加された行や更新された行は、インデックス順には保管されません。（インデックス順に保管したい場合は、コマンドを再度入力し、定期的に再クラスタ化を行います。また、更新される行は十分な領域が利用可能ならば同一ページ内に保持されますので、テーブルのFILLFACTOR格納パラメータを100%より小さく設定することで、更新処理中のクラスタ順序付けを保護するという点で役に立ちます。）

テーブルがクラスタ化されると、PostgreSQLはクラスタ化に使用されたインデックスを記録します。CLUSTER *table_name*という構文によって、以前と同じインデックスを使用してテーブルを再クラスタ化します。またALTER TABLEのCLUSTERもしくはSET WITHOUT CLUSTER構文を使用して、将来のクラスタ化操作で使用するインデックスを設定したり、任意の過去の設定を取り消すことができます。

パラメータを指定しないでCLUSTERを実行した場合、呼び出したユーザが所有する現在のデータベース内の全てのテーブルを（スーパーユーザの場合は、全てのテーブルを）クラスタ化します。このパラメータを指定しないCLUSTERを、トランザクションブロック内で実行することはできません。

クラスタ化を行っているテーブルは、ACCESS EXCLUSIVEロックを獲得しています。これにより、CLUSTERが終わるまで、そのテーブルに対するデータベース操作（読み書き両方）を防ぐことができます。

パラメータ

table_name

テーブルの名前です（スキーマ修飾名も可）。

index_name

インデックスの名前です。

VERBOSE

各テーブルのクラスタ化を行う時に進行状況報告を出力します。

注釈

テーブル内の1つの行にランダムにアクセスする場合、テーブル内のデータの順序は重要ではありません。しかし、テーブル内の特定のデータにアクセスが集中している場合、それらのデータをグループ化しているインデックスが存在する時は、CLUSTERによる利益を享受できます。テーブルからインデックスの値の範囲や、一致する複数の行を保有する1つのインデックスの値などが知りたい場合、CLUSTERが役に立ちます。一度インデックスが一致する行に対するテーブルページを認識すると、一致する他の全ての行も同じテーブルページに存在する可能性が高いため、ディスクアクセスを減らして問い合わせ処理の速度を向上することができますからです。

CLUSTERは、指定されたインデックスによるインデックススキャン、または（インデックスがB-Treeの場合）シーケンシャルス

キャン後のソートのいずれかを用いて、テーブルを再ソートすることができます。 プランナのコストパラメータと利用可能な統計情報に基づき、より高速な方式の選択を試みます。

インデックススキャンが使用される場合、インデックス順にテーブルデータを並べた、テーブルの一時コピーが作成されます。同様に、テーブルの各インデックスの一時コピーも作成されます。 したがって、ディスクには、少なくともテーブルとインデックスの合計サイズと同じ容量の空き領域が必要です。

シーケンシャルスキャンとソートが使用される場合も一時的なソートファイルが作成されます。 一時的に必要なとなるサイズの最大値はテーブルサイズの倍のサイズにインデックスサイズを加えた値となります。 この方式はインデックススキャンより高速になることが多いのですが、必要なディスク容量に耐えられない場合は、一時的に [enable_sort](#) を `off` にすることで、この方式を無効にすることができます。

クラスタ処理の前に [maintenance_work_mem](#) を程良く大きな値に設定することを勧めます。 （しかしメモリ容量以上は `CLUSTER` 操作用に割り当てられません。）

プランナはテーブルの順序付けに関する統計情報を記録しているため、新しくクラスタ化されたテーブルでは、 [ANALYZE](#) を実行することが推奨されます。 そうしないと、プランナが問い合わせ計画を適切に選択できない可能性があります。

`CLUSTER` はどのインデックスでクラスタ化したかを記録していますので、対象のテーブルを定期的に再クラスタ化できるように、最初にクラスタ化したいテーブルを手作業でクラスタ化し、その後にパラメータをまったく持たない `CLUSTER` を実行する定期的な保守用スクリプトを設定することができます。

例

`employees_ind` インデックスに基づいて、`emp` テーブルをクラスタ化します。

```
CLUSTER employees USING employees_ind;
```

以前に使用したインデックスを使用して、`employees` テーブルをクラスタ化します。

```
CLUSTER employees;
```

データベース内の、以前にクラスタ化されたテーブルを全てクラスタ化します。

```
CLUSTER;
```

互換性

標準SQLには `CLUSTER` 文はありません。

```
CLUSTER index_name ON table_name
```

という構文も、8.3より前のバージョンのPostgreSQLとの互換性のためサポートされます。

関連項目

[clusterdb](#)

[前のページ](#)
CLOSE

[ホーム](#)
[上に戻る](#)

[次のページ](#)
COMMENT

COMMENT

名前

COMMENT -- オブジェクトのコメントを定義する、または変更する

概要

```

COMMENT ON
{
    AGGREGATE agg_name (agg_type [, ...] ) |
    CAST (source_type AS target_type) |
    COLLATION object_name |
    COLUMN relation_name.column_name |
    CONSTRAINT constraint_name ON table_name |
    CONVERSION object_name |
    DATABASE object_name |
    DOMAIN object_name |
    EXTENSION object_name |
    FOREIGN DATA WRAPPER object_name |
    FOREIGN TABLE object_name |
    FUNCTION function_name ( [ [ argmode ] [ argname ] argtype [, ...] ] ) |
    INDEX object_name |
    LARGE OBJECT large_object_oid |
    OPERATOR operator_name (left_type, right_type) |
    OPERATOR CLASS object_name USING index_method |
    OPERATOR FAMILY object_name USING index_method |
    [ PROCEDURAL ] LANGUAGE object_name |
    ROLE object_name |
    RULE rule_name ON table_name |
    SCHEMA object_name |
    SEQUENCE object_name |
    SERVER object_name |
    TABLE object_name |
    TABLESPACE object_name |
    TEXT SEARCH CONFIGURATION object_name |
    TEXT SEARCH DICTIONARY object_name |
    TEXT SEARCH PARSER object_name |
    TEXT SEARCH TEMPLATE object_name |
    TRIGGER trigger_name ON table_name |
    TYPE object_name |
    VIEW object_name
} IS 'text'

```

説明

COMMENTは、データベースオブジェクトに関するコメントを保存します。

各オブジェクトに保存できるコメント文字列は1つだけです。 ですので、コメントを編集するためには、同一オブジェクトに対して新しくCOMMENTコマンドを発行してください。 コメントを削除するには、テキスト文字列の部分にNULLを記述してください。 オブジェクトが削除された時、コメントは自動的に削除されます。

ほとんどの種類のオブジェクトでは、オブジェクトの所有者のみがコメントを設定することができます。 ロールには所有者がありませんので、COMMENT ON ROLEにおける規則は、スーパーユーザロールに対するコメント付けはスーパーユーザでなければならず、スーパーユーザ以外のロールに対するコメント付けはCREATEROLEを持たなければならないとなります。 当然ながらスーパーユーザは何にでもコメントを付けることができます。

コメントは、psqlの\d系のコマンドで表示することができます。

obj_description()、col_description()、shobj_descriptionという名前の、psqlが使用する組み込み関数を使うように構築することで、他のユーザインタフェースを使ってコメントを取り出せるようになります ([表9-54](#)を参照してください)。

パラメータ

object_name
relation_name.column_name
agg_name
constraint_name
function_name

operator_name
rule_name
trigger_name

コメントを付加するオブジェクトの名前です。 テーブル、集約、照合順、変換、ドメイン、外部テーブル、関数、インデックス、演算子、演算子クラス、演算子族、シーケンス、テキスト検索オブジェクト、データ型、ビューの名前は、スキーマ修飾することができます。 列にコメントを付与する場合、*relation_name*はテーブル、ビュー、複合型、外部テーブルを参照するものでなければなりません。

agg_type

集約関数が演算する入力データ型です。 引数を持たない集約関数を参照する場合は、入力データ型のリストに*を指定してください。

source_type

キャストの変換元データ型の名前です。

target_type

キャストの変換先のデータ型の名前です。

argmode

関数の引数のモードで、IN、OUT、INOUT、VARIADICのいずれかです。 省略時のデフォルトはINです。 関数を識別するには入力引数のみが必要です、COMMENT ON FUNCTIONが実際にはOUT引数を無視することに注意してください。 したがって、IN、INOUTおよびVARIADIC引数を列挙することで十分です。

argname

関数の引数の名前です。 関数の識別には引数データ型のみが必要です、COMMENT ON FUNCTIONが実際には引数の名前を無視することに注意してください。

argtype

もしあれば、関数の引数のデータ型（複数可、スキーマ修飾も可）です。

large_object_oid

ラージオブジェクトのOIDです。

left_type
right_type

演算子の引数のデータ型（スキーマ修飾も可）です。 右単項演算子、左単項演算子における存在しない引数についてはNONEと記述してください。

PROCEDURAL

これはノイズワードです。

text

追加するコメントです。文字列リテラルとして記述します。 コメントを削除する場合はNULLを記述します。

注釈

現在、コメントの閲覧に関するセキュリティ機構は存在しません。 データベースに接続したユーザは誰でも、そのデータベース内のオブジェクトのコメントを参照することができます。 データベース、ロール、テーブル空間などの共有オブジェクトに対するコメントは大域的に格納され、クラスタ内の任意のデータベースに接続した任意のユーザが共有オブジェクトに対するコメントをすべて見ることができます。 そのため、コメントにはセキュリティ的に重大な情報を記載してはいけません。

例

mytable テーブルにコメントを付けます。

```
COMMENT ON TABLE mytable IS 'This is my table.';
```

先ほどのコメントを削除します。

```
COMMENT ON TABLE mytable IS NULL;
```


その他の例をいくつか示します。

```
COMMENT ON AGGREGATE my_aggregate (double precision) IS 'Computes sample variance';
COMMENT ON CAST (text AS int4) IS 'Allow casts from text to int4';
COMMENT ON COLLATION "fr_CA" IS 'Canadian French';
COMMENT ON COLUMN my_table.my_column IS 'Employee ID number';
COMMENT ON CONVERSION my_conv IS 'Conversion to UTF8';
COMMENT ON CONSTRAINT bar_col_cons ON bar IS 'Constrains column col';
COMMENT ON DATABASE my_database IS 'Development Database';
COMMENT ON DOMAIN my_domain IS 'Email Address Domain';
COMMENT ON EXTENSION hstore IS 'implements the hstore data type';
COMMENT ON FOREIGN DATA WRAPPER mywrapper IS 'my foreign data wrapper';
COMMENT ON FOREIGN TABLE my_foreign_table IS 'Employee Information in other database';
COMMENT ON FUNCTION my_function (timestamp) IS 'Returns Roman Numeral';
COMMENT ON INDEX my_index IS 'Enforces uniqueness on employee ID';
COMMENT ON LANGUAGE plpython IS 'Python support for stored procedures';
COMMENT ON LARGE OBJECT 346344 IS 'Planning document';
COMMENT ON OPERATOR ^ (text, text) IS 'Performs intersection of two texts';
COMMENT ON OPERATOR - (NONE, integer) IS 'Unary minus';
COMMENT ON OPERATOR CLASS int4ops USING btree IS '4 byte integer operators for btrees';
COMMENT ON OPERATOR FAMILY integer_ops USING btree IS 'all integer operators for btrees';
COMMENT ON ROLE my_role IS 'Administration group for finance tables';
COMMENT ON RULE my_rule ON my_table IS 'Logs updates of employee records';
COMMENT ON SCHEMA my_schema IS 'Departmental data';
COMMENT ON SEQUENCE my_sequence IS 'Used to generate primary keys';
COMMENT ON SERVER myserver IS 'my foreign server';
COMMENT ON TABLE my_schema.my_table IS 'Employee Information';
COMMENT ON TABLESPACE my_tablespace IS 'Tablespace for indexes';
COMMENT ON TEXT SEARCH CONFIGURATION my_config IS 'Special word filtering';
COMMENT ON TEXT SEARCH DICTIONARY swedish IS 'Snowball stemmer for swedish language';
COMMENT ON TEXT SEARCH PARSER my_parser IS 'Splits text into words';
COMMENT ON TEXT SEARCH TEMPLATE snowball IS 'Snowball stemmer';
COMMENT ON TRIGGER my_trigger ON my_table IS 'Used for RI';
COMMENT ON TYPE complex IS 'Complex number data type';
COMMENT ON VIEW my_view IS 'View of departmental costs';
```

互換性

標準SQLにはCOMMENTはありません。

[前のページ](#)
CLUSTER

[ホーム](#)
[上に戻る](#)

[次のページ](#)
COMMIT

COMMIT

名前

COMMIT -- 現在のトランザクションをコミットする

概要

```
COMMIT [ WORK | TRANSACTION ]
```

説明

`COMMIT` は現在のトランザクションをコミットします。 そのトランザクションで行われた全ての変更が他のユーザから見えるようになり、クラッシュが起きても一貫性が保証されるようになります。

パラメータ

`WORK`
`TRANSACTION`

省略可能なキーワードです。何も効果はありません。

注釈

トランザクションをアボートするには[ROLLBACK](#)を使用してください。

トランザクションの外部で`COMMIT`を発行しても特に問題は発生しません。 ただし、警告メッセージが表示されます。

例

現在のトランザクションをコミットし、全ての変更を永続化します。

```
COMMIT;
```

互換性

標準SQLが規定しているのは、`COMMIT` と `COMMIT WORK` という2つの構文のみです。 その他の点は完全な互換性があります。

関連項目

[BEGIN](#), [ROLLBACK](#)

COMMIT PREPARED

名前

COMMIT PREPARED -- 二相コミット用に事前に準備されたトランザクションをコミットする

概要

```
COMMIT PREPARED transaction_id
```

説明

COMMIT PREPARED は準備段階のトランザクションをコミットします。

パラメータ

transaction_id

コミット対象のトランザクションのトランザクション識別子です。

注釈

準備されたトランザクションをコミットするには、元のトランザクションを実行したユーザと同じユーザか、スーパーユーザでなければなりません。しかし、トランザクションを実行したセッションと同じセッションである必要はありません。

このコマンドはトランザクションブロックの内側では実行できません。準備されたトランザクションは即座にコミットされます。

利用可能な準備されたトランザクションはすべて、[pg_prepared_xacts](#) システムビューで列挙されます。

例

foobar トランザクション識別子で識別されるトランザクションをコミットします。

```
COMMIT PREPARED 'foobar';
```

互換性

COMMIT PREPARED は PostgreSQL の拡張です。これは外部のトランザクション管理システムによる利用を意図したものです。トランザクション管理システムの一部 (X/Open XA など) は標準化されていますが、こうしたシステムの SQL 側は標準化されていません。

関連項目

[PREPARE TRANSACTION](#), [ROLLBACK PREPARED](#)

COPY

名前

COPY -- ファイルとテーブルの間でデータをコピーする

概要

```

COPY table_name [ ( column_name [, ...] ) ]
FROM { 'filename' | STDIN }
[ [ WITH ] ( option [, ...] ) ]

COPY { table_name [ ( column_name [, ...] ) ] | ( query ) }
TO { 'filename' | STDOUT }
[ [ WITH ] ( option [, ...] ) ]

```

ここで`option`は以下のいずれかです。

```

FORMAT format_name
OIDS [ boolean ]
DELIMITER 'delimiter_character'
NULL 'null_string'
HEADER [ boolean ]
QUOTE 'quote_character'
ESCAPE 'escape_character'
FORCE_QUOTE { ( column_name [, ...] ) | * }
FORCE_NOT_NULL { column_name [, ...] ) |
ENCODING 'encoding_name'

```

説明

COPY コマンドは、PostgreSQLのテーブルと標準のファイルシステムのファイル間でデータを移動します。COPY TO コマンドはテーブルの内容をファイルにコピーします。また、COPY FROM コマンドは、ファイルからテーブルへとデータをコピーします（この時、既にテーブルにあるデータにコピーした内容を追加します）。また、COPY TO によりSELECT 問い合わせの結果をコピーすることができます。

列リストが指定されている場合、COPY は、指定された列のデータのみのコピーをファイルからまたはファイルへ行います。列リストに含まれていない列がテーブル内にある場合、COPY FROM はそれらの列にデフォルトの値を挿入します。

ファイル名付きのCOPY コマンドは、PostgreSQL サーバに対して直接ファイルへの読み書きをするように命じます。指定したファイルは必ずサーバからアクセスできる必要があります。また、ファイル名はサーバから見たように指定されなければなりません。STDIN や STDOUT が指定された場合、データはクライアントとサーバ間を流れます。

パラメータ

table_name

既存のテーブルの名前です（スキーマ修飾名も可）。

column_name

コピー対象の列リストです。このパラメータは省略可能です。列リストが指定されていない場合は、テーブルの全ての列がコピーされます。

query

指定したSELECT コマンドまたはVALUES コマンドの結果がコピーされます。問い合わせを括弧でくくる必要があることに注意してください。

filename

入出力ファイルの絶対パス名です。Windows ユーザの場合、パス名内にバックスラッシュを使用する際には`\\`文字列を使用し、二重にする必要があるかもしれません。

STDIN

入力クライアントアプリケーションからのものであることを指定します。

STDOUT

出力クライアントアプリケーションへのものであることを指定します。

boolean

選択オプションを有効とするか無効とするかを指定します。 オプションを有効にするには、`TRUE`、`ON`または`1`と、無効にするには`FALSE`、`OFF`または`0`と記述します。 また`boolean`値は省略可能であり、省略時は`TRUE`とみなされます。

FORMAT

読み取りまたは書き込みに使用するデータ書式を選択します。 `text`、`csv`（カンマ区切り値）、または`binary`です。 デフォルトは`text`です。

OIDS

各行のOIDをコピーするよう指定します（OIDを持たないテーブルに`oids`を指定する場合、または、`query`コピーの場合、エラーが発生します）。

DELIMITER

ファイルの各行（ライン）内の列を区切る文字を指定します。 テキスト書式でのデフォルトはタブ文字、`csv`書式ではカンマです。 これは単一の1バイト文字でなければなりません。 このオプションは`binary`書式を使用する場合は許されません。

NULL

NULL値を表す文字列を指定します。 デフォルトは、テキスト書式では`¥N`（バックスラッシュN）、`csv`書式では引用符のない空文字です。 NULL値と空文字列を区別する必要がない場合は、テキスト書式であっても空文字列を使用した方が良いでしょう。 このオプションは`binary`書式を使用する場合は許されません。

注意: `COPY FROM`の場合、この文字列と一致するデータ要素はNULL値として格納されます。 `COPY TO`実行時に使用した同じ文字列を使用しているかどうか確認してください。

HEADER

ヘッダ行を含むファイルを指定します。 このファイルには各列の名前が記載されています。 出力では、先頭行にテーブルの列名が入り、入力では先頭行は無視されます。 このオプションは`csv`書式を使用する場合のみで許されます。

QUOTE

データ値を引用符付けする際に使用される引用符用文字を指定します。 デフォルトは二重引用符です。 これは単一の1バイト文字でなければなりません。 このオプションは`csv`書式を使用する場合のみで許されます。

ESCAPE

`QUOTE`の値がデータ内の文字と一致した場合に、その前に現れなければならない文字を指定します。 デフォルトは`QUOTE`の値です（このためデータ内に引用符用文字があれば二重になります）。 これは単一の1バイト文字でなければなりません。 このオプションは`csv`書式を使用する場合のみ許されます。

FORCE_QUOTE

指定された各列内にある全ての非NULL値を強制的に引用符で囲みます。 NULL出力は引用符で囲まれません。 *が指定された場合、非NULL値はすべての列で引用符付けされます。 このオプションは`COPY TO`において、かつ、`csv`書式を使用する場合のみで許されます。

FORCE_NOT_NULL

指定された列の値をNULL文字列に対して比較しません。 NULL文字列が空であるデフォルトでは、空の値は引用符付けされていなくてもNULLではなく長さが0の文字列として読み取られることを意味します。 このオプションは`COPY FROM`において、かつ、`csv`書式を使用する場合のみで許されます。

ENCODING

ファイルが`encoding_name`で符号化されていることを指定します。 このオプションが省略された場合、現在のクライアント符号化方式が使用されます。 後述の注意を参照してください。

出力

正常に完了した場合、`COPY`コマンドは以下の形式のコマンドタグを返します。

`COPY count`

`count` はコピーされた行数です。

注釈

`COPY` は通常のテーブルに対してのみ使用することができます。ビューに対して使用することはできません。しかし、`COPY (SELECT * FROM viewname) TO ...` と記述することができます。

`COPY` は指定された特定のテーブルのみを扱います。つまり子テーブルへまたは子テーブルからのデータのコピーは行いません。したがって例えば、`COPY table TO` は `SELECT * FROM ONLY table` と同じデータを示します。しかし `COPY (SELECT * FROM table) TO ...` を使用して、継承階層内のすべてのデータをダンプすることができます。

`COPY TO` の場合は値を読み込むテーブルに対する `SELECT` 権限が、`COPY FROM` の場合は値を挿入するテーブルに対する `INSERT` 権限が必要です。コマンド内で列挙された列に対する列権限を持つことで十分です。

`COPY` コマンドで指定するファイルは、クライアントアプリケーションではなく、サーバが直接読み込み/書き込みを行います。したがって、それらのファイルは、クライアントではなく、データベースサーバマシン上に存在するか、または、データベースサーバマシンからアクセス可能である必要があります。さらに、クライアントではなく、PostgreSQL ユーザ（サーバを実行しているユーザID）が、アクセス権限と読み書き権限を持っている必要があります。ファイル名を指定した `COPY` コマンドの実行は、データベースのスーパーユーザのみに許可されています。このコマンドによって、サーバがアクセス権限を持つ全てのファイルの読み込み、書き込みが可能になってしまうためです。

`COPY` は `psql` の `\copy` とは異なるものであることに注意してください。 `\copy` は `COPY FROM STDIN` や `COPY TO STDOUT` を呼び出し、`psql` クライアントからアクセスできるファイルにデータの書き込み/読み込みを行います。したがって、`\copy` コマンドでは、ファイルへのアクセスが可能かどうかと、ファイルに対するアクセス権限の有無は、サーバではなくクライアント側に依存します。

`COPY` でファイル名を指定する時は、常に絶対パスで記述することをお勧めします。`COPY TO` コマンドの場合はサーバによって強制的に絶対パス指定にさせられますが、`COPY FROM` コマンドでは相対パスで指定されたファイルを読み込むことも可能となっています。後者では、クライアントの作業ディレクトリではなく、サーバプロセスの作業ディレクトリ（通常はクラスタのデータディレクトリ）からの相対的なディレクトリとして解釈されます。

`COPY FROM` は、宛先テーブル上で任意のトリガと検査制約を呼び出しますが、ルールは呼び出しません。

`COPY` の入出力は `DateStyle` の影響を受けます。デフォルト以外の `DateStyle` が設定された可能性がある PostgreSQL インストールとの移植を確実に行いたい場合は、`COPY` を使う前に `DateStyle` を `ISO` に設定しなければなりません。また、`IntervalStyle` を `sql_standard` としてデータをダンプすることは避けることを勧めます。負の時間間隔値が別の `IntervalStyle` 設定を持つサーバで誤解釈される可能性があるためです。

たとえデータがクライアント経由ではなくサーバにより直接ファイルから読み書きされるとしても、入力データは `ENCODING` オプションまたは現在のクライアント符号化方式にしたがって解釈され、出力データは `ENCODING` オプションまたは現在のクライアント符号化方式で符号化されます。

`COPY` では、エラーが発生するとすぐに処理を停止します。`COPY TO` コマンドの実行では何ら問題ありませんが、`COPY FROM` の場合は、対象となるテーブルは初めの方の行を既に受け取っています。これらの行は不可視となり、アクセスすることもできませんが、ディスク領域を占有します。したがって、大きなコピー処理に何度も失敗した場合には、無視できないほど無駄なディスク領域が増えてしまいます。この無駄な領域を取り戻すには、`VACUUM` を行う必要があります。

ファイルの形式

テキスト形式

`text` 書式を使用する場合、読み書きされるデータはテーブルの1つの行を1行で表したテキストファイルとなります。行内の列は区切り文字で区切られます。属性値自体は、その属性のデータ型の出力関数で生成された、または、その入力関数で受け付け可能な文字列です。値が `NULL` の列では、代わりに指定された `NULL` 値を表す文字列が使用されます。入力ファイルのいずれかの行にある列数が予期された数と違う場合、`COPY FROM` はエラーを発生します。`OIDS` が指定された場合、`OID` は、ユーザデータの列の前にある、1番目の列として読み書きされます。

データの終了は、バックスラッシュとピリオド (`\.`) のみから構成される1行で表されます。EOF（ファイルの終了）を使えば同じことが実現できるので、ファイルからの読み込みの場合はデータ終了マークは不要です。しかし、3.0以前のクライアントプロトコルを使用したクライアントアプリケーションとデータのコピーを行う場合だけは、読み込み、書き込みを問わず、終了マークが必要です。

バックスラッシュ文字（¥）は、COPY対象データ内で、行や列の区切り文字と判定される可能性があるデータ文字列の引用符付けに使用します。特に、バックスラッシュ自体、改行、使用中の区切り文字などの文字が列の値に含まれている場合は、必ず前にバックスラッシュを付けなければなりません。

指定されたNULL文字列はバックスラッシュを付けずにCOPY TOに送られます。一方、COPY FROMでは、バックスラッシュを削除する前にNULL文字列と入力と比較されます。したがって、¥NといったNULL文字列が実際の¥Nというデータ値と混乱することはあり得ません。（これは¥¥Nとして表現されます。）

COPY FROMは、バックスラッシュで始まる次のような文字の並びを識別します。

文字の並び	表現
¥b	バックスペース (ASCII 8)
¥f	改ページ (ASCII 12)
¥n	改行 (ASCII 10)
¥r	復帰 (ASCII 13)
¥t	タブ (ASCII 9)
¥v	垂直タブ (ASCII 11)
¥数字	バックスラッシュに続き1から3個の8進数の数字をコード番号として指定すると、そのコードが表す文字を指定できます。
¥x <i>digits</i>	バックスラッシュ、x という並びに続き1から2個の16進数の数字をコード番号として指定すると、そのコードが表す文字を指定できます。

現在、COPY TOは、バックスラッシュの後ろに8進数や16進数を付けた形式で文字を出力することはありませんが、上記一覧に存在しない制御文字が含まれている場合は、この形式を使用します。

上表で記載されていないバックスラッシュ付きの文字はすべて、そのまま解釈されます。しかし、不要なバックスラッシュの追加には注意してください。偶然にデータを終わりを示す印（¥.）やヌル文字列（デフォルトでは¥N）と合致する文字列を生成してしまうかもしれないためです。これらの文字列は他のバックスラッシュの処理を行う前に解釈されます。

COPY データを生成するアプリケーションは、データ内の改行と復帰をそれぞれ、¥nと¥rに変換することを強く推奨されています。現在のところ、バックスラッシュと復帰文字でデータ内の復帰を表したり、バックスラッシュと改行文字でデータ内の改行を表すことが可能です。しかし、こういった表現は今後のリリースでは、受け付けられなくなる可能性があります。また、COPYファイルが異なるマシンをまたがって転送される場合、破損するおそれがあります（例えば、UnixからWindowsあるいはその逆）。

COPY TOは各行の行末にUnix形式の改行 ("¥n") を出力します。なお、Microsoft Windowsで稼働するサーバの場合は、サーバ上のファイルへのCOPYの場合にのみ復帰/改行 ("¥r¥n") を出力します。プラットフォームをまたがる一貫性のために、サーバのプラットフォームにかかわらず、COPY TO STDOUTは常に"¥n"を送信します。COPY FROMは、改行、復帰、復帰/改行を行末として扱うことができます。データを意図したバックスラッシュのない改行や復帰によるエラーの危険性を減らすために、COPY FROMは、入力行の行末が全て共通でない場合に警告を発します。

CSV形式

この書式オプションは、スプレッドシートなど他の多くのプログラムで使われるカンマ区切り値 (CSV) ファイル形式をインポート、エクスポートするために使われます。PostgreSQLの標準テキスト書式で使われるエスケープ規則の代わりに、一般的なCSVのエスケープ機構を生成、認識します。

各レコードの値はDELIMITER文字で区切られます。区切り文字、QUOTE文字、NULL文字列、復帰、改行文字を含む値の場合、全体の値の前後にQUOTE文字が付与されます。値の中でQUOTE文字やESCAPE文字が現れる場合、その前にエスケープ用の文字が付与されます。また、FORCE_QUOTEを使用して、特定列内の非NULL値を出力する時に強制的に引用符を付与することもできます。

CSV書式にはNULL値と空文字列とを区別する標準的な方法はありません。PostgreSQLのCOPYでは引用符によってこれを区別しています。NULLはNULLパラメータの文字列として出力され、引用符で囲まれません。一方、NULLパラメータの文字列に一致する非NULL値は引用符で囲まれます。たとえばデフォルトの設定では、NULLは引用符付けのない空文字列として出力され、空文字列のデータ値は2つの引用符 ("") で出力されます。データの読み込みの際も同様の規則に従います。FORCE_NOT_NULLを使用して、特定列に対しNULL入力の比較を行わないようにすることもできます。

CSV書式ではバックスラッシュは特別な文字ではありませんので、データ終端記号¥. はデータ値として現れることがあります。誤った解釈を防ぐために、行内の一項目として¥. というデータ値が現れる場合、出力に自動的に引用符が付けられます。また、入力では引用符で括られた場合データ終端記号として解釈されません。単一の引用符で括られない列で¥. という値を持つ可能性

がある、他のアプリケーションで作成されたファイルをロードしようとする場合、入力ファイル内のこうした値を引用符で括る必要があります。

注意: CSV書式では文字はすべて意味を持ちます。空白文字で括られた単一引用符や`DELIMITER`以外の任意の文字がこうした文字に含まれます。これにより、固定長にするためにCSVの行に空白文字を埋めるシステムから取り出したデータをインポートする時にエラーが発生する可能性があります。このような状況になった場合、PostgreSQLにデータをインポートする前に、そのCSVファイルから行末の空白を除去する前処理が必要になります。

注意: CSV書式は、復帰文字や改行文字が埋め込まれ引用符で囲まれた値を含むCSVファイルを認識し、生成します。したがって、このファイルでは、テキスト書式のように1つのテーブル行が1行で表されているとは限りません。

注意: 奇妙な（時には間違った）CSVファイルを生成するプログラムは多く存在するので、このファイル書式には標準よりも多くの慣習があります。したがって、この機能でインポートできないファイルが存在するかもしれませんし、COPYが他のプログラムで処理できないファイルを生成するかもしれません。

バイナリ形式

binary書式オプションにより、すべてのデータはテキストではなくバイナリ書式で書き込み/読み取りされるようになります。テキストやCSV書式よりも多少高速になりますが、バイナリ書式のファイルはマシンアーキテクチャやPostgreSQLのバージョンをまたがる移植性が落ちます。またバイナリ書式はデータ型に非常に依存します。たとえば、テキスト書式ではうまく動作したとしても、smallint列からバイナリデータを出力し、それをinteger列として読み込むことはできません。

binaryファイルの書式は、ファイルヘッダ、行データを含む0以上のタプル、ファイルトレーラから構成されます。ヘッダとデータはネットワークバイトオーダーです。

注意: 7.4以前のリリースのPostgreSQLでは異なるバイナリファイル形式を使用していました。

ファイルヘッダ

ファイルヘッダは15バイトの固定フィールドとその後に続く可変長ヘッダ拡張領域から構成されます。固定フィールドは以下の通りです。

署名

PGCOPY¥n¥377¥r¥n¥0 という11バイトの並びです。この署名の必須部分にNULLバイトが含まれていることに注意してください（この署名は、8ビットを通過させない転送方式によってファイルが破損した場合、これを容易に識別できるように設計されています。署名は、改行コード変換やNULLバイトの削除、上位ビット落ち、パリティの変更などによって変化します）。

フラグフィールド

このファイル形式の重要な部分となる32ビット整数のビットマスクです。ビットには0（LSB）から31（MSB）までの番号が付いています。このフィールドは、このファイル形式で使用される他の全ての整数フィールドと同様、（最大バイトが最初に現れる）ネットワークバイトオーダーで保存されていることに注意してください。ファイル形式上の致命的な問題を表すために、16-31ビットは予約されています。この範囲に想定外のビットが設定されていることが判明した場合、読み込み先は処理を中断しなければなりません。後方互換における形式の問題を通知するために、0-15ビットは予約されています。この範囲に想定外のビットが設定されていても、読み込み先は無視すべきです。現在、1つのビットだけがフラグビットとして定義されており、残りは0でなければなりません。

ビット16

1ならば、OIDがデータに含まれています。0ならば、含まれていません。

ヘッダ拡張領域長

自分自身を除いた、ヘッダの残り部分のバイト長を示す32ビットの整数です。現在、これは0となっており、すぐ後に最初のタプルが続きます。今後、ヘッダ内に追加データを格納するような形式の変更があるかもしれません。読み込み側では、ヘッダ拡張データの扱いがわからない場合、そのデータをスキップしなければなりません。

ヘッダ拡張領域は、それ自身で認識することができる塊の並びを保持するために用意されています。フラグフィールドは読み込み先に拡張領域の内容を知らせるものではありません。ヘッダ拡張内容の個々の設計は今後のリリースのために残してあります。

この設計によって、後方互換性を維持するヘッダの追加（ヘッダ拡張チャンクの追加や下位フラグビットの設定）と後方互換性

のない変更（変更を通知するための高位フラグビットの設定や必要に応じた拡張領域へのサポート情報追加）の両方に対応できません。

タプル

全てのタプルはタプル内のフィールド数を表す16ビットの整数から始まります（現時点では、テーブル内の全てのタプルは同一のフィールド数を持つことになっていますが、今後、これは変更される可能性があります）。その後、タプル中のそれぞれのフィールドが続きます。これらのフィールドには、先頭にフィールドデータが何バイトあるかを表す32ビット長のワードが付けられています（このワードが表す長さには自分自身は含まれません。したがって、0になることもあります）。特殊な値としてNULLフィールドを表す-1が用意されています。このNULLが指定された場合、値用のバイトはありません。

フィールド間には整列用のパッドやその他の余計なデータはありません。

現在、バイナリ書式のファイル内の全てのデータ値は、バイナリ形式（形式コード1）であると仮定されています。将来の拡張によって、列単位に形式コードを指定するヘッダフィールドが追加されることを予想したものです。

実際のタプルデータとして適切なバイナリ形式を決定するためには、PostgreSQLのソース、特に各列のデータ型用の*send関数と*recv関数（通常配布物内のsrc/backend/utils/adtディレクトリにあります）を調べなければなりません。

このファイルにOIDが含まれる場合、OIDフィールドがフィールド数ワードの直後に続きます。これは、フィールドカウントを持たない点を除いて、通常のフィールドです。特に長さワードがあるため、苦勞することなく、4バイトのOIDも8バイトのOIDも扱うことができます。また、これにより、望ましいと判断されれば、OIDをNULLとして表示することができます。

ファイルトレーラ

ファイルトレーラは、-1を含んだ、16ビットの整数ワードで構成されます。タプルのフィールドカウントワードを使用すれば、このワードを容易に見分けることができます。

読み込み側は、フィールドカウントワードが-1でも、想定した列数でもなかった場合はエラーを報告しなければなりません。これにより、何らかの理由でデータとの同期が取れなくなったことを判定する特別な検査を行うことが可能になります。

例

次の例では、フィールド区切り文字として縦棒（|）を使用してテーブルをクライアントにコピーします。

```
COPY country TO STDOUT (DELIMITER '|');
```

ファイルからcountryテーブルにデータをコピーします。

```
COPY country FROM '/usr1/proj/bray/sql/country_data';
```

名前が'A'から始まる国のみをファイルにコピーします。

```
COPY (SELECT * FROM country WHERE country_name LIKE 'A%') TO '/usr1/proj/bray/sql/a_list_countries.copy';
```

これはSTDINからテーブルにコピーするのに適したデータの例です。

AF	AFGHANISTAN
AL	ALBANIA
DZ	ALGERIA
ZM	ZAMBIA
ZW	ZIMBABWE

各行の空白文字は実際にはタブ文字であることに注意してください。

以下は同一のデータをバイナリ書式で出力したものです。データをUnixユーティリティod -cを使ってフィルタしたものを示します。テーブルには3列あり、最初のデータ型はchar(2)、2番目はtext、3番目はintegerです。全ての行の3列目はNULL値です。

00000000	P	G	C	O	P	Y	¥n	377	¥r	¥n	¥0	¥0	¥0	¥0	¥0	¥0
00000020	¥0	¥0	¥0	¥0	003	¥0	¥0	¥0	002	A	F	¥0	¥0	¥0	013	A
00000040	F	G	H	A	N	I	S	T	A	N	377	377	377	377	¥0	003
00000060	¥0	¥0	¥0	002	A	L	¥0	¥0	¥0	007	A	L	B	A	N	I
00000100	A	377	377	377	377	¥0	003	¥0	¥0	¥0	002	D	Z	¥0	¥0	¥0
00000120	007	A	L	G	E	R	I	A	377	377	377	377	¥0	003	¥0	¥0
00000140	¥0	002	Z	M	¥0	¥0	¥0	006	Z	A	M	B	I	A	377	377
00000160	377	377	¥0	003	¥0	¥0	¥0	002	Z	W	¥0	¥0	¥0	¥b	Z	I
00000200	M	B	A	B	W	E	377	377	377	377	377	377				

互換性

標準SQLにはCOPY文はありません。

以下の構文は、PostgreSQLバージョン9.0より前に使用されていたもので、まだサポートされています。

```
COPY table_name [ ( column_name [, ...] ) ]
FROM { 'filename' | STDIN }
[ [ WITH ]
    [ BINARY ]
    [ OIDS ]
    [ DELIMITER [ AS ] 'delimiter' ]
    [ NULL [ AS ] 'null string' ]
    [ CSV [ HEADER ]
        [ QUOTE [ AS ] 'quote' ]
        [ ESCAPE [ AS ] 'escape' ]
        [ FORCE NOT NULL column_name [, ...] ] ] ] ]

COPY { table_name [ ( column_name [, ...] ) ] | ( query ) }
TO { 'filename' | STDOUT }
[ [ WITH ]
    [ BINARY ]
    [ OIDS ]
    [ DELIMITER [ AS ] 'delimiter' ]
    [ NULL [ AS ] 'null string' ]
    [ CSV [ HEADER ]
        [ QUOTE [ AS ] 'quote' ]
        [ ESCAPE [ AS ] 'escape' ]
        [ FORCE QUOTE { column_name [, ...] | * } ] ] ] ]
```

この構文では、BINARYとCSVがFORMATオプションの引数ではなく、独立したキーワードとして扱われることに注意してください。

以下の構文は、PostgreSQLバージョン7.3より前に使用されていたもので、まだサポートされています。

```
COPY [ BINARY ] table_name [ WITH OIDS ]
FROM { 'filename' | STDIN }
[ [USING] DELIMITERS 'delimiter' ]
[ WITH NULL AS 'null string' ]

COPY [ BINARY ] table_name [ WITH OIDS ]
TO { 'filename' | STDOUT }
[ [USING] DELIMITERS 'delimiter' ]
[ WITH NULL AS 'null string' ]
```

[前のページ](#)

COMMIT PREPARED

[ホーム](#)
[上に戻る](#)

[次のページ](#)

CREATE AGGREGATE

CREATE AGGREGATE

名前

CREATE AGGREGATE -- 新しい集約関数を定義する

概要

```
CREATE AGGREGATE name ( input_data_type [ , ... ] ) (
    SFUNC = sfunc,
    STYPE = state_data_type
    [ , FINALFUNC = ffunc ]
    [ , INITCOND = initial_condition ]
    [ , SORTOP = sort_operator ]
)
```

または以下の旧構文

```
CREATE AGGREGATE name (
    BASETYPE = base_type,
    SFUNC = sfunc,
    STYPE = state_data_type
    [ , FINALFUNC = ffunc ]
    [ , INITCOND = initial_condition ]
    [ , SORTOP = sort_operator ]
)
```

説明

CREATE AGGREGATEは、新しい集約関数を定義します。配布物には基本的、かつ、よく使用される集約関数がいくつか含まれています。これらの集約関数については、[項9.20](#)に文書化されています。新しい型を定義する場合、またはまだ提供されていない集約関数が必要な場合、必要な機能を実現するためにCREATE AGGREGATEを使うことができます。

スキーマ名が付けられている場合（例えば、CREATE AGGREGATE myschema.myagg ...）、集約関数は指定されたスキーマで作成されます。スキーマ名がなければ、集約関数は現在のスキーマで作成されます。

集約関数は名前と入力データ型(複数可)の組み合わせによって識別されます。演算の対象となる入力データ型が異なっていれば、同じスキーマ内に同じ名前の集約関数があっても構いません。1つのスキーマ内では、集約関数の名前と入力データ型(複数可)は、通常関数の名前と入力データ型と異なる必要があります。

集約関数は1つか2つの通常関数から作られます。状態遷移関数*sfunc*と省略可能な最終計算関数*ffunc*です。これらは以下のように使われます。

```
sfunc ( 内部状態, 次のデータ値 ) ----> 次の内部状態
ffunc ( 内部状態 ) ----> 集約の結果
```

PostgreSQLは、集約の現在の内部状態を保持する*stype*データ型の一時変数を作成します。それぞれの入力行に対して、集約引数の値が計算され、現在の状態値と新しい引数値で新しい内部状態変数を計算するために状態遷移関数が呼び出されます。全ての行が処理されると、集約の出力値を計算するために最終関数が1回呼び出されます。最終関数がない場合は、終了時の状態値がそのまま返されます。

集約関数は、内部状態の初期値として初期状態を提供することができます。これはtext型の値としてデータベースに格納されますが、状態値データ型の定数として有効な外部表現でなければいけません。初期状態が与えられていない場合、状態値はNULLから始まります。

状態遷移関数が"厳格 (strict)"と宣言されている場合、NULLを入力値にして呼び出すことはできません。そのような遷移関数では、集約は次のように実行されます。NULL入力値を持つ行は無視されます。（関数は呼び出されず、前回の状態値が保持されます。）初期状態値がNULLである場合、初めて入力行がすべて非NULL入力値であった時にその引数の値で状態値を置き換え、次に入力行がすべて非NULL入力値を持つ時から遷移関数の呼び出しが始まります。このような動作は、maxのような集約を実装するには便利です。ただし、state_data_typeが最初のinput_data_typeと同じ時にのみ有効であることに注意してください。これらの型が異なる時は、非NULL初期値を供給するか、厳格でない遷移関数を使わなければいけません。

状態遷移関数が厳格 (strict) でない場合は、それぞれの入力行に対してその関数が無条件に呼び出されるので、NULL入力とNULL遷移値を自分で処理しなければいけません。これは、関数の作成者が、集約関数におけるNULL値の扱いを完全に制御できることを意味します。

最終関数が"厳格 (strict)"と宣言されていると、終了状態値がNULLの時は、最終関数が呼び出されません。 その場合、NULLという結果が自動的に出力されます（もちろんこれは、厳格な関数の一般的な動作に過ぎません）。 どのような場合でも、最終関数はNULLを返すことができます。 例えば、avgの最終関数は、入力が行だとわかるとNULLを返します。

MINやMAXのような振舞いをする集約では、すべての入力行を走査せずにインデックスを検索することで最適化できることがあります。 このように最適化される集約の場合、ソート演算子を指定することで明示してください。 その演算子で生成されるソート順で集約の最初の要素が生成されなければならないということが基本的な必要条件です。 言い換えると、

```
SELECT agg(col) FROM tab;
```

が

```
SELECT col FROM tab ORDER BY col USING sortop LIMIT 1;
```

と同じでなければならないということです。 更に、集約がNULL入力を見捨てること、および、NULL以外の入力がまったくなかった時にのみNULLという結果を返すことも前提となります。 通常、データ型の<演算子はMINのソート演算子として、また、>演算子はMAXのソート演算子として適切です。 指定した演算子がB-treeインデックス演算子クラスの"より小さい"ストラテジか"より大きい"ストラテジのメンバでない限り、最適化が実際には効果がないことに注意してください。

集約関数を作成するためには、引数の型、状態の型、戻り値の型に対するUSAGE権限と遷移関数と最終関数に対するEXECUTE権限を持たなければなりません。

パラメータ

name

作成する集約関数の名前です（スキーマ修飾名も可）。

input_data_type

集約関数が演算する入力データ型です。 引数が存在しない集約関数を作成するには、入力データ型のリストに*と記載してください（例えば count(*)などの集約です）。

base_type

CREATE AGGREGATEの旧構文では、入力データ型は集約名称の次に記載されたものではなくbasetypeパラメータにより指定されます。 この構文では1つの入力パラメータしか指定できないことに注意してください。 引数を持たない集約を定義するためには、basetypeを"ANY"（*ではありません）と指定してください。

sfunc

それぞれの入力行に対して呼び出される状態遷移関数の名前です。 N引数を持つ集約関数では、sfuncはN+1個の引数を取らなければなりません。 最初の引数はstate_data_type型で、残りはその集約の入力データ型として宣言したものと一致していなければなりません。 この関数はstate_data_type型の値を返さなければなりません。 この関数は、現在の状態値と現在の入力データ値を受け取り、次の状態値を返します。

state_data_type

集約の状態値のデータ型です。

ffunc

最終関数の名前です。最終関数は、全ての入力行に対する処理が終わった後、集約の結果を計算するために呼び出されます。 この関数はstate_data_type型の引数を1つ取らなければなりません。 集約の出力データ型はこの関数の戻り値として定義されます。 ffuncが指定されない場合には、集約の結果として終了時の状態値が使われます。出力型はstate_data_typeになります。

initial_condition

状態値の初期設定です。 データ型state_data_typeとして受け取り可能な文字列定数でなければいけません。 このパラメータが指定されない場合、状態値はNULLから始まります。

sort_operator

MINまたはMAXのような集約に対して関連付いたソート演算子です。 これは単なる演算子の名前です（スキーマで修飾可能）。 この演算子は集約（これは単一引数の集約でなければなりません）と同じ入力データ型を持つと前提されています。

CREATE AGGREGATEのパラメータは、任意の順番で記述することができます。上記の順番で記述する必要はありません。

例

[項35.10](#)を参照してください。

互換性

`CREATE AGGREGATE`はPostgreSQLの言語拡張です。 標準SQLには、ユーザ定義の集約関数を使用する機能はありません。

関連項目

[ALTER AGGREGATE](#), [DROP AGGREGATE](#)

[前のページ](#)
COPY

[ホーム](#)
[上に戻る](#)

[次のページ](#)
CREATE CAST

CREATE CAST

名前

CREATE CAST -- 新しいキャストを定義する

概要

```
CREATE CAST (source_type AS target_type)
  WITH FUNCTION function_name (argument_type [, ...])
  [ AS ASSIGNMENT | AS IMPLICIT ]

CREATE CAST (source_type AS target_type)
  WITHOUT FUNCTION
  [ AS ASSIGNMENT | AS IMPLICIT ]

CREATE CAST (source_type AS target_type)
  WITH INOUT
  [ AS ASSIGNMENT | AS IMPLICIT ]
```

説明

CREATE CASTを使用すると、新しいキャストを定義できます。キャストは、2つのデータ型間の変換処理方法を指定するものです。以下に例を示します。

```
SELECT CAST(42 AS float8);
```

この文を実行すると、事前に指定された関数（この場合float8(int4)）が呼び出され、整数定数42がfloat8型に変換されます（適切なキャストが定義されていない場合、変換処理は失敗します）。

2つのデータ型をバイナリ強制互換とすることができます。これは、関数をまったく呼び出さなくても、"自由に"変換を行うことができることを意味します。ただし、対応する値は、同じ内部表現を使用している必要があります。例えば、データ型textとvarcharには、両方向でバイナリ互換性があります。バイナリ強制互換性は対称関係を必要としていません。例えば、現在の実装ではxmlからtextへのキャストは自由に行うことができますが、逆方向では少なくとも構文検査を行う関数が必要です。（2つの型が両方向でバイナリ強制互換であることは、バイナリ互換性と呼ばれます。）

WITH INOUT構文を使用してI/O変換キャストとしてキャスト定義を行うことができます。I/O変換キャストは、元データ型の出力関数を呼び出し、その結果文字列を対象データ型の入力関数に渡すことで行われます。多くの一般的な場合では、この機能により変換用に別個のキャスト関数を作成する必要がなくなります。I/O変換キャストは通常の変換関数を基にしたキャストと同様に動作します。ただ実装が異なるだけです。

デフォルトでは、キャストは明示的なキャスト要求があった場合のみ発生します。明示的なキャスト要求の構文は、CAST(x AS typename)、もしくは、x::typename式です。

キャストにAS ASSIGNMENTオプションを付けると、対象データ型の列に代入する際、暗黙的にそのキャストを発生させるように設定できます。例えば、foo.f1がtext型の列であるとして。

```
INSERT INTO foo (f1) VALUES (42);
```

integer型をtext型に変換するキャストにAS ASSIGNMENTオプションが付けられていれば、上記のSQL文が実行できます。しかし、AS ASSIGNMENTオプションが付いていなければ、実行できません（一般的に、この種のキャストは代入キャストと呼ばれています）。

キャストにAS IMPLICITオプションを付けると、代入の場合だけでなく、式の中にある場合でも、全てのコンテキストで暗黙的にそのキャストを呼び出すように設定できます。（一般的に、この種のキャストは暗黙キャストと呼ばれています。）例えば次のような問い合わせを考えてみます。

```
SELECT 2 + 4.0;
```

パーサはまず定数にそれぞれintegerとnumericであると印を付けます。システムカタログには、integer + numericという演算子はありませんが、numeric + numericという演算子は存在します。したがって、integerからnumericへのキャストが利用可能であり、そのキャストにAS IMPLICITが付いていればこの問い合わせは成功します（実際このようになっています）。パーサは暗黙的なキャストを行い、問い合わせをあたかも次のように記載されたものとして解決します。

```
SELECT CAST ( 2 AS numeric ) + 4.0;
```


ここで、カタログはまたnumericからintegerへのキャストも提供しています。もしこのキャストにAS IMPLICITが付いていたら（実際は付いていません）、パーサは上のように解釈するか、それとも、numeric定数をintegerにキャストし、integer + integerという演算子を適用するかを選択しなければなりません。どちらがより良いかという知見がなければ、選択をあきらめ、問い合わせがあいまいであると宣告します。2つのキャストの内1つのみが暗黙的であるという事実が、パーサに、numericとintegerが混在する式をnumericとして扱うという適切な解決方法を知らせる方法です。これに関する組み込まれた知見は存在しません。

暗黙キャストは、多用しない方が賢明です。暗黙的キャストを使用し過ぎると、PostgreSQLがコマンドを思わぬ意味に解釈してしまう原因になります。また、複数の解釈が可能なため、コマンドをまったく解読できなくなってしまう可能性もあります。経験的には、2つのデータ型が同一の一般的なデータ型のカテゴリに属しており、変換によって情報が保持される場合のみ、暗黙キャストを呼び出し可能にするのが良い方法と思われます。例えば、int2型からint4型へのキャストは、暗黙キャストにするのが妥当ですが、float8型からint4型へのキャストは、代入キャストのみにすべきと思われます。text型からint4型への変換のような、カテゴリを越えるデータ型のキャストは、明示的にのみ使用するのが適切です。

注意: 上で説明した通り防ぐことができないあいまいさを引き起こす、型の集合の中で複数の暗黙的なキャストを提供することが、有用性や標準との互換性上の理由により必要となることがまれにあります。パーサは、こうした状況でも望ましい動作の提供を補助できる型カテゴリと好まれる型に基づいた発見的手法を用意しています。詳細は[CREATE TYPE](#)を参照してください。

キャストを作成するためには、変換元または変換先（の内の一方）のデータ型を所有し、もう一方の型に対するUSAGE権限を持つ必要があります。また、バイナリ強制互換性を持つキャストを作成できるのは、スーパーユーザでなければなりません。（バイナリ強制互換性があるキャスト変換を誤って使用するとサーバがクラッシュしてしまう可能性が高いことから、この制限が付けられました）。

パラメータ

source_type

キャストする変換元のデータ型の名前です。

target_type

キャストする変換先のデータ型の名前です。

function_name(*argument_type* [, ...])

この関数はキャストを実行するために使用します。関数名はスキーマ修飾することができます。スキーマ修飾されていない場合、関数はスキーマ検索パスから検索されます。関数の結果のデータ型は、キャストの変換先のデータ型と一致する必要があります。引数については後で説明します。

WITHOUT FUNCTION

変換元データ型から変換先データ型への間に、バイナリ強制互換性があることを示します。この場合、キャストを実行するのに関数は必要ありません。

WITH INOUT

キャストが、元データ型の出力関数を呼び出し、その結果文字列を対象データ型の入力関数に渡すことで行われる、I/O変換キャストであることを示します。

AS ASSIGNMENT

代入コンテキストで、暗黙的にキャストを呼び出せることを示します。

AS IMPLICIT

任意のコンテキストで、暗黙的にキャストを呼び出せることを示します。

キャストを実装する関数は1～3個の引数を取ることができます。1番目の引数型はキャストの変換元データ型と同一、または、変換元データ型からのバイナリ強制互換を持つ型でなければなりません。2番目の引数（もしあれば）は、integer型でなければなりません。変換先の型に関連付けられた型修飾子を指定します。型修飾子がない場合は-1を指定します。3番目の引数（もしあれば）は、boolean型でなければなりません。キャストが明示的なキャストであればtrueを、それ以外であればfalseを指定します（奇妙な話ですが、標準SQLでは、明示的キャストと暗黙的キャストとの間で異なる振舞いを要求する場合があります。この引数はそのようなキャストを実装しなければならない関数用に提供されています。独自のデータ型をこの流儀に従うように設計することは勧められません）。

キャスト関数の戻り値は、キャストの対象型と同一またはバイナリ強制互換性を持たなければなりません。

通常、キャストにおける変換元データ型と変換先データ型は異なる必要があります。しかし、2つ以上の引数を持つ関数でキャストを実装した場合は、変換元と変換先とで同一のデータ型を持つキャストを宣言することができます。これは、システムカタログにおいて型固有の長さ強制関数を表現するために使用されています。このように指定された関数は、型の値を強制的に2番目の引数で与えられた型修飾子の値にするために使用されます（文法上、型修飾子を持つのは特定の組み込みデータ型のみなので、この機能は変換先のデータ型がユーザ定義のものである場合には役に立ちません。ここでは、説明を完全にするために言及しています）。

キャストが変換元と変換先のデータ型が異なり、複数の引数を取る関数を持つ場合、あるデータ型から他のデータ型への変換と長さの強制を1つの操作にまとめたものをサポートします。引数を1つしか取らない場合は、型修飾子を使用して型を強制するために、データ型間の変換と修飾子の適用という2つのキャスト操作が必要となります。

ドメイン型へのキャスト、ドメイン型からのキャストは効果がありません。ドメインへのキャストドメインからのキャストは、基となる型と関連したキャストを使用します。

注釈

ユーザ定義のキャストを削除するには[DROP CAST](#)を使用してください。

データ型を双方向に変更可能にするには、双方向のキャストを明示的に宣言する必要があることに注意してください。

ユーザ定義型と標準文字列型（`text`、`varchar`、`char(n)`）、文字列カテゴリ内と定義されたユーザ定義型との間のキャストを作成することは、通常必要ありません。PostgreSQLはこのために自動的なI/O変換キャストを提供します。この文字列への自動キャストは代入キャストとして扱われますが、文字列型からの入出力変換キャストは明示的なキャストのみです。この振舞いは独自のキャストを宣言して自動キャストを置き換えることで変更することができます。しかし、通常このようにするのは、この変換を標準の代入のみまたは明示的のみの設定よりもより呼び出しやすくしたい場合に限られます。他にも、型の入出力関数と異なる動作で変換したいという理由もあるかもしれません。しかし、これは非常に驚かされるものであり、そうすべきかどうか熟考すべきです。（組み込み型のごく一部は実際変換用に異なった振舞いをしますが、ほとんどは標準SQLの仕様のためのものです。）

PostgreSQL 7.3より前のバージョンでは、データ型と同じ名前を持ち、そのデータ型を返し、異なるデータ型の引数を1つ取る関数は、自動的にキャスト関数とみなされていました。この慣習は、スキーマを導入し、システムカタログでバイナリ強制互換性によるキャストを実現可能にしたことにより、廃止されました。組み込みキャスト関数は、まだこの命名規則に従っています。しかし、他のキャスト同様、`pg_cast` システムカタログにキャストとして示されている必要があります。

必須ではありませんが、キャストを実装する関数には変換先のデータ型の名前を付けるという以前からの慣習に従っておくことを推奨します。多くのユーザは`typename(x)`という関数スタイルの記法でデータ型のキャストを行っています。この記法は、キャストを実装している関数の呼び出しに他なりません。キャストとして特別に扱っているわけではないのです。ユーザが作成した変換関数の名前がこの慣習に従っていないと、他のユーザがとまどうことになります。PostgreSQLは引数として異なる型を取る同じ名前の関数をオーバーロードすることができるので、様々な型から特定の変換先型への変換関数の名前を全て変換先の型名にしても特に問題は発生しません。

注意: 実際のところ、先ほどの段落は単純化しすぎたものです。関数呼び出し式が実際の関数と一致しない状態でキャスト要求として扱われる状況が2つ存在します。関数呼び出し`name(x)`が実際の関数に正確に一致せず、`name`がデータ型の名前であり、`pg_cast`が`x`の型からその型へのバイナリ強制互換のキャストを提供する場合、この呼び出しはバイナリ強制互換キャストとして処理されます。この例外は、実際の関数が存在しなくても、関数のような構文でバイナリ強制互換キャストを呼び出すことができるように作成されました。同様に、`pg_cast`に項目がないが、文字列型との間のキャストが存在する場合、この呼び出しは入出力変換キャストとして処理されます。この例外により関数のような構文で入出力変換キャストができるようになります。

注意: この例外にも例外があります。複合型から文字列型へのI/O変換キャストでは関数構文を使用して呼び出すことができず、明示的なキャスト構文（`CAST`記法または`::`記法のいずれか）で記述しなければなりません。この例外は、自動提供I/O変換キャストを導入した後、関数または列参照を意図した時に非常に簡単に間違って呼び出されることが判明したため追加されました。

例

関数`int4(bigint)`を使用した`bigint`型から`int4`型への代入キャストを作成します。

```
CREATE CAST (bigint AS int4) WITH FUNCTION int4(bigint) AS ASSIGNMENT;
```

（このキャストは、システムに既に定義されています。）

互換性

SQLではバイナリ強制互換性があるデータ型や実装関数の追加の引数について規定されていません。さらに、`AS IMPLICIT`は、PostgreSQLの拡張です。これらの点以外では、`CREATE CAST`は標準SQLに準拠しています。

関連項目

[CREATE FUNCTION](#), [CREATE TYPE](#), [DROP CAST](#)

CREATE COLLATION

名前

CREATE COLLATION -- 新しい照合順序を定義する

概要

```
CREATE COLLATION name (  
    [ LOCALE = locale, ]  
    [ LC_COLLATE = lc_collate, ]  
    [ LC_CTYPE = lc_ctype ]  
)  
CREATE COLLATION name FROM existing_collation
```

説明

CREATE COLLATIONは指定したオペレーティングシステムのロケール設定または既存の照合順序をコピーすることで新しい照合順序を定義します。

照合順序を新しく作成するためには、格納先のスキーマにおけるCREATE権限が必要です。

パラメータ

name

照合順序の名前です。照合順序の名前はスキーマ修飾することができます。スキーマ修飾されていない場合、照合順序は現在のスキーマ内で定義されます。照合順序の名前はそのスキーマ内で一意でなければなりません。（システムカタログでは異なる符号化方式に対して同じ名前の照合順序を含めることができます。しかしデータベース符号化方式が異なる場合には無視されます。）

locale

これは同時にLC_COLLATEおよびLC_CTYPEを設定する省略形です。これを指定した場合、これらのパラメータのいずれかを指定することはできません。

lc_collate

LC_COLLATEロケールカテゴリに対して指定したオペレーティングシステムのロケールを使用します。このロケールは現在のデータベース符号化方式に適用可能なものでなければなりません。（正確な規則については[CREATE DATABASE](#)を参照してください。）

lc_ctype

LC_CTYPEロケールカテゴリに対して指定したオペレーティングシステムのロケールを使用します。このロケールは現在のデータベース符号化方式に適用可能なものでなければなりません。（正確な規則については[CREATE DATABASE](#)を参照してください。）

existing_collation

コピーする既存の照合順序の名前です。新しい照合順序は既存のものと同一属性を持ちますが、独立したオブジェクトになります。

注意

ユーザ定義の照合順序を削除するためにはDROP COLLATIONを使用してください。

PostgreSQLにおける照合順序サポートに関しては[項22.2](#)を参照してください。

例

オペレーティングシステムのfr_FR.utf8ロケールから照合順序を作成します。（現在のデータベース符号化方式がUTF8であることが前提です。）

```
CREATE COLLATION french (LOCALE = 'fr_FR.utf8');
```

既存の照合順序から照合順序を作成します。

```
CREATE COLLATION german FROM "de_DE";
```

アプリケーションにおいてオペレーティングシステムに依存しない照合順序の名前を使用することができ、便利になるかもしれません。

互換性

標準SQLにはCREATE COLLATIONが存在しますが、既存の照合順序のコピーに限定されています。新しい照合順序を作成するための構文はPostgreSQLの拡張です。

関連項目

[ALTER COLLATION](#), [DROP COLLATION](#)

[前のページ](#)
CREATE CAST

[ホーム](#)
[上に戻る](#)

[次のページ](#)
CREATE CONVERSION

CREATE CONVERSION

名前

CREATE CONVERSION -- 新しい符号化方式変換の定義

概要

```
CREATE [ DEFAULT ] CONVERSION name
FOR source_encoding TO dest_encoding FROM function_name
```

説明

CREATE CONVERSION を使用すると、新しい文字セット符号化方式間の変換を定義できます。また、DEFAULT として指定された変換は、クライアントとサーバの間での自動的な符号化方式の変換に使用できます。このためには、符号化方式Aから符号化方式Bへ、および、符号化方式Bから符号化方式Aへという2つの変換を定義する必要があります。

変換を作成するためには、その関数のEXECUTE 権限、および、対象となるスキーマ上のCREATE 権限を保持している必要があります。

パラメータ

DEFAULT

DEFAULT 句により、この変換が、指定された変換元から対象となる符号化方式への変換のデフォルトであることが示されます。1つのスキーマ内でデフォルトとされる変換は、符号化方式の組み合わせ1組において1つだけです。

name

変換の名前です。変換名は、スキーマ修飾することができます。スキーマ修飾されていない場合、変換は現在のスキーマで定義されます。変換名は、スキーマ内で一意である必要があります。

source_encoding

変換元の符号化方式名です。

dest_encoding

変換先の符号化方式名です。

function_name

この関数は、変換の実行に使用されます。関数名は、スキーマ修飾することができます。スキーマ修飾されていない場合、関数はパスから検索されます。

関数は、下記のような形式で記述する必要があります。

```
conv_proc(
    integer, -- 変換元符号化方式ID
    integer, -- 変換先符号化方式ID
    cstring, -- 変換元文字列 (NULLで終わるC言語文字列)
    internal, -- 変換先文字列 (NULLで終わるC言語文字列)
    integer -- 変換元文字列長
) RETURNS void;
```

注釈

ユーザ定義の変換を削除するには、DROP CONVERSION を使用します。

変換の作成に必要な権限は、今後のリリースで変更される可能性があります。

例

myfunc関数を使用して、UTF8からLATIN1への符号化方式の変換を作成します。

```
CREATE CONVERSION myconv FOR 'UTF8' TO 'LATIN1' FROM myfunc;
```

互換性

CREATE CONVERSIONは、PostgreSQLの拡張です。 標準SQLにはCREATE CONVERSION文はありませんが、CREATE TRANSLATION文の目的および構文は非常に似たものです。

関連項目

[ALTER CONVERSION](#), [CREATE FUNCTION](#), [DROP CONVERSION](#)

CREATE DATABASE

名前

CREATE DATABASE -- 新しいデータベースを作成する

概要

```
CREATE DATABASE name
[ [ WITH ] [ OWNER [=] user_name ]
  [ TEMPLATE [=] template ]
  [ ENCODING [=] encoding ]
  [ LC_COLLATE [=] lc_collate ]
  [ LC_CTYPE [=] lc_ctype ]
  [ TABLESPACE [=] tablespace_name ]
  [ CONNECTION LIMIT [=] connlimit ] ]
```

説明

CREATE DATABASEは新しいPostgreSQLデータベースを作成します。

データベースを作成するには、スーパーユーザ、もしくはCREATEDBという特別な権限を持つユーザである必要があります。[CREATE USER](#)を参照してください。

デフォルトでは、新しいデータベースは標準システムデータベースtemplate1を複製することによって作成されます。他のテンプレートを指定するには、TEMPLATE *name*と記述します。特に、TEMPLATE template0と記述することで、そのバージョンのPostgreSQLによって定義済みの標準オブジェクトのみを持つ、新しいデータベースを作ることができます。これは、template1に独自にインストールして追加したオブジェクトをコピーしたくない場合に便利です。

パラメータ

name

作成するデータベースの名前です。

user_name

新しいデータベースを所有するユーザのロール名です。デフォルト設定（つまり、コマンドを実行したユーザ）を使用する場合はDEFAULTと指定します。他のロールによって所有されるデータベースを作成するためには、そのロールの直接的または間接的なメンバであるか、スーパーユーザでなければなりません。

template

新しいデータベースを作成元となるテンプレートの名前です。デフォルトテンプレート（template1）を使う場合は、DEFAULTと指定します。

encoding

新しいデータベースで使われる文字セット符号化方式です。文字列定数（例えば'SQL_ASCII'）、整数の符号化方式番号、デフォルトの符号化方式（すなわちテンプレートデータベースの符号化方式）を使うDEFAULTのいずれかを指定します。PostgreSQLサーバでサポートされる文字セットについては[項22.3.1](#)で説明します。その他の制限については後述します。

lc_collate

新しいデータベースで使用する照合順（LC_COLLATE）です。これは、たとえばORDER BYを持つ問い合わせなどにおいて文字列に適用されるソート順やテキスト型の列に対するインデックスで使われる順序に影響します。

lc_ctype

新しいデータベースで使用する文字のクラス（LC_CTYPE）です。これは、たとえば小文字、大文字、数字といった文字の分類に影響します。デフォルトではテンプレートデータベースの文字クラスを使用します。さらなる制限に関しては後で説明します。

`tablespace_name`

新しいデータベースに関連付けされるデフォルトのテーブル空間名です。テンプレートデータベースのテーブル空間を使用する場合は、`DEFAULT`と指定します。このテーブル空間が、このデータベースで作成されるオブジェクトのデフォルトのテーブル空間となります。詳細は[CREATE TABLESPACE](#)を参照してください。

`connlimit`

このデータベースで確立できる同時接続数です。-1（デフォルト）は無制限を意味します。

省略可能なパラメータは、任意の順番で記述できます。上記の順番で記述しなくても構いません。

注釈

`CREATE DATABASE`はトランザクションブロックの内側では実行できません。

ほとんどの場合、"could not initialize database directory"という行が含まれるエラーは、データディレクトリの権限不足、ディスク容量不足などファイルシステムについての問題に関連するものです。

データベースを削除するには[DROP DATABASE](#)を使用してください。

[createdb](#)プログラムは利便性のために提供される、このコマンドのラッププログラムです。

`template1`以外のデータベースの名前をテンプレートに指定してデータベースをコピーすることは可能ですが、これは（まだ）一般的に使用する"`COPY DATABASE`"機能として意図されているわけではありません。主な制限は、コピー中に他のセッションからテンプレートデータベースへの接続ができないことです。`CREATE DATABASE`は、開始した時に他の接続があると失敗します。テンプレートデータベースへの新規接続は`CREATE DATABASE`が完了するまでロックされます。詳細は[項21.3](#)を参照してください。

新しいデータベース用に指定される文字セット符号化方式は選択されたロケール設定（`LC_COLLATE`および`LC_CTYPE`）と互換性がなければなりません。ロケールが`C`（や同等の`POSIX`）であれば、すべての符号化方式が許されますが、他のロケール設定では適切に動作する符号化方式は1つしかありません。（しかしWindowsではUTF-8符号化方式をすべてのロケールで使用することができます。）`CREATE DATABASE`では、ロケール設定に関係なくスーパーユーザが`SQL_ASCII`符号化方式を指定することを許していますが、こうした選択は廃止予定であり、データベース内にロケールと互換性がない符号化方式でデータが格納された場合、文字列関数の誤動作を多く引き起こします。

符号化方式とロケール設定はテンプレートデータベースのこれらの設定と一致しなければなりません。ただし`template0`がテンプレートとして使用される場合は例外です。他のデータベースには指定された符号化方式と一致しないデータを含む可能性や`LC_COLLATE`および`LC_CTYPE`がソート順序に影響するようなインデックスを含む可能性があることがこの理由です。こうしたデータをコピーすると、新しい設定から見るとデータベースが破損する結果となります。しかし`template0`には影響を受けるデータやインデックスが含まれていないことが分かっています。

`CONNECTION LIMIT`は厳密な制限ではありません。データベース向けの接続"スロット"が1つ残っていた時に同時に2つの新しいセッション開始要求があった場合、両方とも失敗する可能性があります。また、この制限はスーパーユーザには強制されません。

例

新しいデータベースを作成します。

```
CREATE DATABASE lusiadas;
```

`salesapp`ユーザを所有者、`salesspace`をデフォルトのテーブル空間として`sales`データベースを作成します。

```
CREATE DATABASE sales OWNER salesapp TABLESPACE salesspace;
```

ISO-8859-1文字セットをサポートする`music`データベースを作成します。

```
CREATE DATABASE music ENCODING 'LATIN1' TEMPLATE template0;
```

この例における`TEMPLATE template0`句は、`template1`の符号化方式がISO-8859-1ではない場合にのみ必要です。符号化方式を変更するには同様に`LC_COLLATE`と`LC_CTYPE`を新たに設定する必要があるかもしれないことに注意してください。

互換性

標準SQLには`CREATE DATABASE`文はありません。データベースは、実装で生成が定義されるカタログと同等のものです。

関連項目

[ALTER DATABASE](#), [DROP DATABASE](#)

[前のページ](#)

CREATE CONVERSION

[ホーム](#)
[上に戻る](#)

[次のページ](#)

CREATE DOMAIN

CREATE DOMAIN

名前

CREATE DOMAIN -- 新しいドメインを定義する

概要

```
CREATE DOMAIN name [ AS ] data_type
    [ COLLATE collation ]
    [ DEFAULT expression ]
    [ constraint [ ... ] ]
```

constraint は、次のようになります。

```
[ CONSTRAINT constraint_name ]
{ NOT NULL | NULL | CHECK (expression) }
```

説明

CREATE DOMAIN は新しいデータドメインを作成します。ドメインとは基本的には、特別な制約（設定可能な値集合に対する制限）を持ったデータ型です。ドメインを定義したユーザが、その所有者となります。

スキーマ名が付けられている場合（例えば、CREATE DOMAIN myschema.mydomain ...）、ドメインは指定されたスキーマで作成されます。スキーマ名が付けられていなければ、そのドメインは現在のスキーマで作成されます。ドメイン名は、そのスキーマ内に存在するデータ型およびドメインの間で、一意である必要があります。

ドメインを使用すると、共通な制約を1箇所にまとめることができ、メンテナンスに便利です。たとえば、E-mailアドレスを格納する列が複数のテーブルで使用されていて、アドレス構文の検証のためすべてが同一のCHECK制約を必要としているような場合です。このような場合、各テーブルに個別に制約を設定するよりも、ドメインを定義してください。

ドメインを作成するためには、基となる型に対するUSAGEを持たなければなりません。

パラメータ

name

作成するドメインの名前です（スキーマ修飾名でも可）。

data_type

ドメインの基となるデータ型です。配列指定子を含めることができます。

collation

省略可能なドメインの照合順です。照合順序の指定がなければ基となるデータ型のデフォルトの照合順序が使用されます。COLLATE が指定される場合、基礎となる型は照合順序が設定可能な型でなければなりません。

DEFAULT *expression*

DEFAULT 句を使用すると、ドメインデータ型の列にデフォルト値を指定できます。任意の無変数式を値とすることができます（ただし、副問い合わせは許可されません）。デフォルト式のデータ型は、そのドメインのデータ型と一致する必要があります。デフォルト値が指定されない場合、デフォルト値はNULL値となります。

デフォルト式は、挿入操作において該当する列に値が指定されなかった場合に使用されます。特定の列に対してデフォルト値が定義された場合、ドメインに関連するデフォルト値は上書きされます。反対に、基となるデータ型に関連するデフォルト値は、ドメインのデフォルト値によって上書きされます。

CONSTRAINT *constraint_name*

制約の名前です（省略可能です）。指定されなければ、システムが名前を生成します。

NOT NULL

このドメインの値としてNULLの使用を通常禁止します。しかしこの制約を持つドメインでは、例えば左外部結合やINSERT INTO tab (domcol) VALUES ((SELECT domcol FROM tab WHERE false))など、NULLとなるドメイン型と一致するように代入された場合、NULLをとることができます。

NULL

ドメインの値としてNULLの使用を許可します。こちらがデフォルトです。

この句は非標準的なSQLデータベースとの互換性を持つために用意されています。新しいアプリケーションでこの句を使用するのはお勧めできません。

CHECK (*expression*)

CHECK句は、整合性制約やドメインの値が満たさなければならない検査を指定します。各制約は、Boolean型の結果を生成する式である必要があります。検査される値を参照するには、VALUEというキーワードを使用すべきです。

現時点では、CHECK式に副問い合わせを含めたり、VALUE以外の変数を参照したりすることはできません。

例

この例では、us_postal_codeデータ型を作成し、その型をテーブル定義の中で使用します。データが有効なUS郵便番号であるかどうかを検証するために正規表現検査が使用されます。

```
CREATE DOMAIN us_postal_code AS TEXT
CHECK(
    VALUE ~ '^¥d{5}$'
OR VALUE ~ '^¥d{5}-¥d{4}$'
);

CREATE TABLE us_snail_addy (
    address_id SERIAL PRIMARY KEY,
    street1 TEXT NOT NULL,
    street2 TEXT,
    street3 TEXT,
    city TEXT NOT NULL,
    postal us_postal_code NOT NULL
);
```

互換性

CREATE DOMAINコマンドは標準SQLに準拠しています。

関連項目

[ALTER DOMAIN](#), [DROP DOMAIN](#)

[前のページ](#)

CREATE DATABASE

[ホーム](#)
[上に戻る](#)

[次のページ](#)

CREATE EXTENSION

CREATE EXTENSION

名前

CREATE EXTENSION -- 拡張をインストールする

概要

```
CREATE EXTENSION [ IF NOT EXISTS ] extension_name
    [ WITH ] [ SCHEMA schema_name ]
    [ VERSION version ]
    [ FROM old_version ]
```

説明

`CREATE EXTENSION`は現在のデータベース内に新しい拡張を読み込みます。読み込み済みの拡張と同じ名前の拡張を読み込むことはできません。

拡張の読み込みは基本的に拡張のスキプトファイルを実行することと同じです。スキプトは通常、関数、データ型、演算子、インデックスサポートメソッドなどのSQLオブジェクトを新しく作成するものです。`CREATE EXTENSION`はさらに作成したすべてのオブジェクト識別子を記録して、`DROP EXTENSION`が発行された時に削除できるようにします。

拡張の読み込みでは、その要素オブジェクトを作成するために必要となるいくつかの権限が必要です。ほとんどの拡張で、これはスーパーユーザもしくはデータベース所有者の権限が必要であることを意味します。`CREATE EXTENSION`を実行するユーザが、後の権限検査の際に拡張の所有者となります。同様にこのユーザは拡張のスキプトにより作成されたすべてのオブジェクトの所有者となります。

パラメータ

`IF NOT EXISTS`

同じ名前の拡張がすでに存在していてもエラーにしません。この場合注意が発せられます。既存の拡張が、現在利用可能なスキプトファイルより作成されるものと何かしら似たものであることは保証されません。

`extension_name`

インストールする拡張の名前です。PostgreSQLは`SHAREDIR/extension/extension_name.control`ファイルから詳細を読み取り、拡張を作成します。

`schema_name`

拡張のオブジェクトをインストールするスキーマの名前です。これにより拡張の内容を再配置させることができます。指定されたスキーマは事前に存在していなければなりません。指定がなく、拡張の制御ファイルでもスキーマを指定していない場合、現在のデフォルトのオブジェクト生成用スキーマが使用されます。

`version`

インストールする拡張のバージョンです。これは識別子あるいは文字列リテラルのいずれかとして記述できます。デフォルトのバージョンは拡張の制御ファイル内で規定したものにになります。

`old_version`

`FROM old_version`は、単なるオブジェクトの集合で拡張としてパッケージ化されていない"古い形式"のモジュールを置き換えるために拡張をインストールする時、この場合のみに指定しなければなりません。このオプションにより`CREATE EXTENSION`は、新しいオブジェクトを作成するのではなく、既存のオブジェクトをパッケージに取り入れる別のインストールスキプトを実行するようになります。すでに存在するオブジェクトを含有するスキーマを`SCHEMA`で指定することに注意してください。

`old_version`で使用する値は拡張の作成者が決定しますので、拡張にアップグレード可能な古い形式のモジュールのバージョンが1つ以上存在する場合に変動する可能性があります。PostgreSQL 9.1より前で提供された標準の追加モジュールでは、モジュールを拡張形式にアップグレードする時`old_version`には`unpackaged`を使用してください。

注意

拡張をデータベースにロードするために `CREATE EXTENSION` を使用できるようになる前に、拡張のサポートファイルがインストールされていなければなりません。 PostgreSQL が提供する拡張のインストールに関する情報は、[追加で提供されるモジュール](#)で説明します。

現在ロードで利用可能な拡張は [pg_available_extensions](#) または [pg_available_extension_versions](#) システムビューで識別できます。

新しい拡張の作成に関しては[項35.15](#)を参照してください。

例

現在のデータベースに [hstore](#) 拡張をインストールします。

```
CREATE EXTENSION hstore;
```

9.1 より前にインストールされた `hstore` を拡張形式に更新します。

```
CREATE EXTENSION hstore SCHEMA public FROM unpackaged;
```

既存の `hstore` オブジェクトをインストールしたスキーマを指定することに注意してください。

互換性

`CREATE EXTENSION` は PostgreSQL の拡張です。

関連項目

[ALTER EXTENSION](#), [DROP EXTENSION](#)

[前のページ](#)

CREATE DOMAIN

[ホーム](#)

[上に戻る](#)

[次のページ](#)

CREATE FOREIGN DATA WRAPPER

CREATE FOREIGN DATA WRAPPER

名前

CREATE FOREIGN DATA WRAPPER -- 新しい外部データラッパを定義する

概要

```
CREATE FOREIGN DATA WRAPPER name
[ HANDLER handler_function | NO HANDLER ]
[ VALIDATOR validator_function | NO VALIDATOR ]
[ OPTIONS ( option 'value' [, ... ] ) ]
```

説明

CREATE FOREIGN DATA WRAPPER は新しい外部データラッパを作成します。 外部データラッパを定義したユーザがその所有者となります。

外部データラッパの名前はデータベース内で一意でなければなりません。

スーパーユーザのみが外部データラッパを作成することができます。

パラメータ

name

作成する外部データラッパの名前です。

HANDLER *handler_function*

handler_function は、事前に登録された、外部テーブル向けの関数実行を受け付けるために呼び出される関数の名前です。 ハンドラ関数は引数を取らず、 `fdw_handler` 型を返すものでなければなりません。

ハンドラ関数を持たない外部データラッパを作成することもできますが、こうしたラッパを使用する外部テーブルは宣言することができるだけでアクセスできません。

VALIDATOR *validator_function*

validator_function は、外部データラッパへ与える一般的なオプションと、その外部データラッパを使用する外部サーバとユーザマップへ与えるオプションを検査するために呼び出される、前もって登録された関数の名前です。 検証関数がない、または `NO VALIDATOR` が指定された場合、オプションは作成時に検査されません。（実装に依存しますが、実行時外部データラッパは無効なオプション指定を無視することも拒絶することもできます。） 検証関数は2つの引数を取らなければなりません。 1つは `text[]` 型で、システムカタログ内に格納されたオプションの配列を含みます。 もう1つは `oid` 型で、オプションを含むシステムカタログのOIDです。 戻り値の型は無視されます。 関数は `ereport()` 関数を使用して無効なオプションを報告しなければなりません。

OPTIONS (*option* 'value' [, ...])

この句は新しい外部データラッパ用のオプションを指定します。 許されるオプション名と値は外部データラッパごとに固有であり、外部データラッパの検証関数を使用して検証されます。 オプション名は一意でなければなりません。

注釈

現在、外部データラッパには基本的な機能しかありません。 外部テーブルの更新をサポートしません。 また問い合わせの最適化は原始的で（ほとんどはラッパに任せられています）。

`postgresql_fdw_validator` という1つの組込みの外部データラッパの検証関数が提供されています。 これは `libpq` 接続パラメータに対応したオプションを受け付けます。

例

無意味な外部データラッパdummyを作成します。

```
CREATE FOREIGN DATA WRAPPER dummy;
```

file_fdw_handlerハンドラ関数を持つ外部データラッパfileを作成します。

```
CREATE FOREIGN DATA WRAPPER file HANDLER file_fdw_handler;
```

いくつかオプションを付けた外部データラッパmywrapperを作成します。

```
CREATE FOREIGN DATA WRAPPER mywrapper
  OPTIONS (debug 'true');
```

互換性

CREATE FOREIGN DATA WRAPPERはISO/IEC 9075-9 (SQL/MED)に従います。ただし、HANDLER句とVALIDATOR句は拡張であり、PostgreSQLでは標準のLIBRARY句とLANGUAGE句は実装されていません。

しかし、SQL/MED機能は全体としてまだ従っていないことに注意してください。

関連項目

[ALTER FOREIGN DATA WRAPPER](#), [DROP FOREIGN DATA WRAPPER](#), [CREATE SERVER](#), [CREATE USER MAPPING](#)

[前のページ](#)

CREATE EXTENSION

[ホーム](#)
[上に戻る](#)

[次のページ](#)

CREATE FOREIGN TABLE

CREATE FOREIGN TABLE

名前

CREATE FOREIGN TABLE -- 新しい外部テーブルを定義する

概要

```
CREATE FOREIGN TABLE [ IF NOT EXISTS ] table_name ( [
    { column_name data_type [ OPTIONS ( option 'value' [, ... ] ) ] [ NULL | NOT NULL ] }
    [, ... ]
] )
    SERVER server_name
    [ OPTIONS ( option 'value' [, ... ] ) ]
```

説明

CREATE FOREIGN TABLEは現在のデータベース内に新しい外部テーブルを作成します。 このテーブルはコマンドを発行したユーザにより所有されます。

スキーマ名が指定された場合（例えばCREATE FOREIGN TABLE myschema.mytable ...）、テーブルは指定されたスキーマ内に作成されます。 さもなくば現在のスキーマ内に作成されます。 外部テーブルの名前は同じスキーマ内にある他の外部テーブル、テーブル、シーケンス、インデックス、ビューと異なるものでなければなりません。

CREATE FOREIGN TABLEはまた自動的に外部テーブルの1行に対応する複合型を表すデータ型を作成します。 したがって外部テーブルは同じスキーマ内の既存のデータ型の名前と同じものを持つことができません。

テーブルを作成するためには、すべての列の型に対するUSAGE権限を持たなければなりません。

パラメータ

IF NOT EXISTS

同じ名前のリレーションがすでに存在していてもエラーとしません。 この場合注意が発せられます。 既存のリレーションが作成しようとしたものと何かしら似たものであることは保証されません。

table_name

作成するテーブルの名前です（スキーマ修飾名でも可）。

column_name

新しいテーブルで作成される列の名前です。

data_type

列のデータ型です。 これには、配列指定子を含めることができます。 PostgreSQLでサポートされるデータ型の情報に関する詳細は[第8章](#)を参照してください。

NOT NULL

その列がNULL値を持たないことを指定します。

NULL

その列がNULL値を持てることを指定します。 これがデフォルトです。

この句は非標準的なSQLデータベースとの互換性のためだけに提供されています。 新しいアプリケーションでこれを使用するのはお勧めしません。

server_name

外部テーブル用の既存サーバの名前です。

OPTIONS (*option* 'value' [, ...])

新しい外部テーブルまたはその列の1つに関連するオプションです。 設定可能なオプションの名前と値は外部データラッパそれぞれに固有なものであり、外部データラッパの検証関数を用いて検証されます。 重複するオプション名は許されません（しかしテーブルオプションと列おオプションでは同じ名前を持足せることはできます）。

例

`film_server`を用いて `films` 外部テーブルを作成します。

```
CREATE FOREIGN TABLE films (  
  code          char(5) NOT NULL,  
  title         varchar(40) NOT NULL,  
  did           integer NOT NULL,  
  date_prod     date,  
  kind          varchar(10),  
  len           interval hour to minute  
)  
SERVER film_server;
```

互換性

`CREATE FOREIGN TABLE` はおおよそ標準SQLに準拠します。 しかし [CREATE TABLE](#) とほとんど同様、`NULL` 制約とゼロ列の外部テーブルが許されます。

関連項目

[ALTER FOREIGN TABLE](#), [DROP FOREIGN TABLE](#), [CREATE TABLE](#), [CREATE SERVER](#)

[前のページ](#)

CREATE FOREIGN DATA WRAPPER

[ホーム](#)

[上に戻る](#)

[次のページ](#)

CREATE FUNCTION

CREATE FUNCTION

名前

CREATE FUNCTION -- 新しい関数を定義する

概要

```

CREATE [ OR REPLACE ] FUNCTION
    name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ] [, ...] ] )
    [ RETURNS rettype
      | RETURNS TABLE ( column_name column_type [, ...] ) ]
{ LANGUAGE lang_name
  | WINDOW
  | IMMUTABLE | STABLE | VOLATILE | [ NOT ] LEAKPROOF
  | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
  | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
  | COST execution_cost
  | ROWS result_rows
  | SET configuration_parameter { TO value | = value | FROM CURRENT }
  | AS 'definition'
  | AS 'obj_file', 'link_symbol'
} ...
[ WITH ( attribute [, ...] ) ]

```

説明

CREATE FUNCTIONは新しい関数を定義します。CREATE OR REPLACE FUNCTIONは、新しい関数の作成、または、既存定義の置換のどちらかを行います。関数を定義するには、ユーザはその言語のUSAGE権限が必要です。

スキーマ名が含まれている場合、関数は指定されたスキーマで作成されます。スキーマ名がなければ、関数は現在のスキーマで作成されます。同じスキーマ内の同じ入力引数データ型を持つ既存の関数の名前は、新しい関数の名前として使用できません。しかし、異なる引数データ型を持つ関数であれば、名前が重複しても構いません（これを、オーバーロードと言います）。

既存の関数定義を入れ替えるには、CREATE OR REPLACE FUNCTIONを使用してください。この方法では関数の名前や引数の型を変更することはできないことに注意してください（これを行った場合、新しく別の関数が作成されます）。また、CREATE OR REPLACE FUNCTIONでは、既存の関数の戻り値の型を変更することはできません。戻り値の型を変更したい場合は、その関数を削除し、再度作成してください。（これは、OUTパラメータを使用している場合、関数を削除しない限りOUTパラメータの型を変更できないことを意味します。）

CREATE OR REPLACE FUNCTIONを使用して既存の関数を置き換える場合、関数の所有者と権限は変わりません。他の関数に関するすべての属性には、そのコマンドで指定された値、または暗黙的な値が代入されます。置き換えるためにはその関数を所有していなければなりません。（これには所有するロールのメンバであることが含まれています。）

関数を削除し再作成した場合、新しい関数は古いものと同じ実体にはなりません。古い関数を参照する、既存のルール、ビュー、トリガなどを削除しなければならないでしょう。関数を参照するオブジェクトを破壊しないように関数定義を変更するには、CREATE OR REPLACE FUNCTIONを使用してください。また、ALTER FUNCTIONを使用して、既存の関数の補助属性のほとんどを変更することができます。

関数を作成したユーザが、その関数の所有者となります。

関数を作成するためには、引数型および戻り値型に対するUSAGE権限を持たなければなりません。

パラメータ

name

作成する関数の名前です（スキーマ修飾名も可）。

argmode

引数のモードで、IN、OUT、INOUT、VARIADICのいずれかです。省略時のデフォルトはINです。OUT引数のみがVARIADICを1つ後に続けることができます。また、RETURNS TABLE記法では、OUTとINOUT引数の両方を使用することは

できません。

argname

引数の名前です。言語の中にはこの名前を関数本体で利用できるものもあります（今のところPL/pgSQLのみです）。他の言語では、関数そのものに注目する限り、入力引数の名前は単なる追加ドキュメントとして扱われます。しかし関数呼び出し時に入力引数の名前を使用することで可読性を高めることができます。（[項4.3](#)参照）どのような場合であっても、出力引数の名前は、結果の行型の列名となりますので重要です。（出力引数の名前を省略した場合、システムはデフォルトの列名を付与します。）

argtype

関数の引数のデータ型です（スキーマ修飾名も可）。基本データ型、複合データ型、ドメイン型、または、テーブル列の型の参照を使用することができます。

また、言語の実装に依存しますが、`cstring`といった"疑似型"を指定できる場合もあります。疑似型は、引数の型の指定が不完全である、もしくは、通常のSQLデータ型の集合を越えていることを示します。

列の型を参照するには、`table_name.column_name%TYPE`と記述します。これを使用すると、テーブル定義が変更されても関数が影響を受けないようにするのに役に立つことがあります。

default_expr

パラメータが指定されなかった場合のデフォルト値として使用される式です。この式はパラメータの引数型と型強制を持たなければなりません。入力パラメータ（`INOUT`を含みます）のみがデフォルト値を持つことができます。デフォルト値を持つパラメータの後ろにあるパラメータはすべて、同様にデフォルト値を持つパラメータでなければなりません。

rettype

関数が返すデータの型です（スキーマ修飾名も可）。基本型、複合型、ドメイン型、または、テーブル列の型の参照を設定することができます。また、実装している言語によりですが、`cstring`のような"疑似型"も指定することが可能です。その関数が値を返すことを想定していない場合は、戻り値として`void`を指定してください。

`OUT`もしくは`INOUT`パラメータが存在する場合、`RETURNS`句を省略することができます。省略しない場合は、出力用パラメータが意味する結果型に従ったもの、つまり、複数の出力用パラメータがあれば`RECORD`、単一の出力用パラメータであればそれと同じ型、でなければなりません。

`SETOF`修飾子は、その関数が、1つではなく複数のアイテムの集合を返すことを示します。

列の型は、`table_name.column_name%TYPE`と記述することで参照されます。

column_name

`RETURNS TABLE`構文における出力列の名前です。これは実際名前付けされた`OUT`パラメータを宣言する別の方法ですが、`RETURNS TABLE`が`RETURNS SETOF`をも意味する点が異なります。

column_type

`RETURNS TABLE`構文における出力列のデータ型です。

lang_name

関数を実装している言語の名前です。このパラメータには、`SQL`、`C`、`internal`、もしくはユーザ定義手続き言語の名前を指定可能です。後方互換を保持するため、この名前を単一引用符で囲むこともできます。

`WINDOW`

`WINDOW`は、この関数が普通の関数ではなくウィンドウ関数であることを示します。現在これはC言語で作成した関数のみに使用することができます。既存の関数定義を置き換える場合、`WINDOW`属性を変更することはできません。

`IMMUTABLE`
`STABLE`
`VOLATILE`

これらの属性は、関数の動作に関する情報を問い合わせオプティマイザに提供します。いずれか1つのキーワードのみ指定できます。指定がない場合は、デフォルトで`VOLATILE`と解釈されます。

`IMMUTABLE`は、データベースに対する変更を行わないこと、および、その関数に同じ引数値を与えた場合に常に同じ結果を返すことを示します。これは、データベースを検索したり、引数リスト中に直接存在しない情報を使用したりしないということです。このオプションが指定された場合、引数が全て定数である関数呼び出しは、即座に関数値と置き換えることができます。

`STABLE`は、データベースに対する変更を行わないこと、および、その関数に同じ引数値を与えた場合、1テーブルスキャ

ン内で常に同じ結果を返すが、SQL文が異なると結果が変わってしまう可能性があることを示します。これは、データベース検索や（現在の時間帯のような）パラメータ変数などに結果が依存する関数に適します。（これは現在のコマンドで変更された行を問い合わせたいAFTERトリガでは不適切です。）また、`current_timestamp`系の関数は、1つのトランザクション内では値が変化しないため、STABLEであることに注意してください。

VOLATILEは、1テーブルスキャン内において関数の値が変化する可能性があるため、最適化できないことを示します。このような意味で変動的（volatile）なデータベース関数は、比較的少数です。例えば、`random()`、`currval()`、`timeofday()`などは変動的な関数です。しかし、例えば`setval()`などの副作用がある関数は、その結果を完全に予測できるとしても、呼び出しを最適化しないよう、VOLATILE（変動的）に分類する必要があることに注意してください。

詳細は[項35.6](#)を参照してください。

LEAKPROOF

LEAKPROOFは、関数が副作用を持たないことを示します。その引数に関する情報を戻り値以外で漏らしません。例えば、一部だけの引数値に関するエラーメッセージを返す関数や何らかのエラーメッセージの中に引数の値を含める関数は漏洩防止とはいえません。問い合わせプランナは、`security_barrier`オプション付きで作成されたビューに漏洩防止関数を含め込みます。[CREATE VIEW](#)および[項37.4](#)を参照してください。このオプションはスーパーユーザによってのみ設定することができます。

CALLED ON NULL INPUT
RETURNS NULL ON NULL INPUT
STRICT

CALLED ON NULL INPUT（デフォルト）を指定すると、引数にNULLが含まれていても、関数が通常通り呼び出されます。その場合は、必要に応じてNULL値を確認し、適切な対応をすることは関数作成者の責任です。

RETURNS NULL ON NULL INPUTもしくはSTRICTを指定すると、関数の引数にNULLがある場合、常にNULLを返します。このパラメータが指定されると、NULL引数がある場合、関数は実行されません。代わりに、NULLという結果が自動的に与えられます。

[EXTERNAL] SECURITY INVOKER
[EXTERNAL] SECURITY DEFINER

SECURITY INVOKERを指定すると、関数を呼び出したユーザの権限で、その関数が実行されます。これがデフォルトです。SECURITY DEFINERを指定すると、関数を作成したユーザの権限で、その関数が実行されます。

EXTERNALキーワードは、SQLとの互換性を保つために許されています。しかし、SQLとは異なり、この機能は外部関数にのみ適用されるわけではないため、このキーワードは省略可能です。

execution_cost

この関数の推定実行コストを表す正数で、単位は[cpu_operator_cost](#)です。関数が集合を返す場合、これは1行当たりのコストとなります。このコストが指定されない場合、C言語および内部関数では1、他のすべての言語では100となります。値をより大きくすると、プランナは必要以上に頻繁に関数を評価しないようになります。

result_rows

プランナが想定する、この関数が返す行数の推定値を表す正数です。これは、関数が集合を返すものと宣言された場合のみ許されます。デフォルト推定値は1000行です。

configuration_parameter
value

SET句により、関数が始まった時に指定した設定パラメータを指定した値に設定し、関数の終了時にそれを以前の値に戻すことができます。SET FROM CURRENTは、セッションにおけるパラメータの現在値を関数が始まった時に適用する値として保管します。

関数にSET句が付いている場合、関数内部で実行されるSET LOCAL コマンドの同一変数に対する効果はその関数に制限されます。設定パラメータの前の値は関数が終了する時に元に戻ります。しかし、通常の（LOCALがない）SET コマンドはSET句を上書きします。これは過去に行われたSET LOCAL コマンドに対してもほぼ同じです。このコマンドの効果は、現在のトランザクションがロールバックされない限り、関数が終了した後も永続化されます。

使用可能なパラメータと値については、[SET](#)および[第18章](#)を参照してください。

definition

関数を定義する文字列定数です。このパラメータの意味は言語に依存します。内部的な関数名、オブジェクトファイルへのパス、SQLコマンド、手続き言語で記述されたテキストなどを指定できます。

関数を定義する文字列を記述する際に、通常の単一引用符ではなく、ドル引用符（[項4.1.2.4](#)参照）を使用すると便利になることが多くあります。ドル引用符を使用しない限り、関数定義内の単一引用符やバックスラッシュは必ず二重にしてエス

ケーブしなければなりません。

`obj_file`, `link_symbol`

この構文の`AS`句は、C言語のソースコード中の関数名がSQL関数の名前と同じでない場合、動的にロードされるC言語関数に使われます。 `obj_file`という文字列は動的にロードできるオブジェクトを含むファイルの名前で、`link_symbol`はそのオブジェクトのリンクシンボル、つまり、C言語ソースコード中の関数の名前です。 リンクシンボルが省略された場合、定義されるSQL関数の名前と同じものと仮定されます。

同一オブジェクトファイルを参照する、`CREATE FUNCTION`呼び出しが繰り返された場合、そのファイルはセッション毎に一度だけロードされます。 （おそらく開発段階で）ファイルをアンロードし再ロードするには、新しいセッションを開始してください。

`attribute`

関数に関する情報を、部分的に選択して指定する伝統的な方法です。 ここで指定できる属性を以下に示します。

`isStrict`

`STRICT`または`RETURNS NULL ON NULL INPUT`と同じです。

`isCachable`

`isCachable`は、`IMMUTABLE`と同じですが、既に廃止されています。 しかし、後方互換性を保持するため、まだ有効とされています。

属性名では、大文字小文字を区別しません。

さらに詳しい外部関数の作成方法については[項35.3](#)を参照してください。

オーバーロード

PostgreSQLは関数のオーバーロードを許可します。 つまり、入力引数の型が異なっていれば、複数の関数に同じ名前を使用することができます。 しかし、全ての関数のC言語における名前は、重複してはいけません。 したがって、オーバーロードするC言語関数には、異なるC言語の名前を与える必要があります （例えば、C言語における名前の一部に引数の型を使用してください）。

同じ名前、同じ入力用パラメータ型を持つ場合、2つの関数は同一であるとみなされます。 `OUT`パラメータは無視されます。 したがって、例えば以下の宣言は競合しています。

```
CREATE FUNCTION foo(int) ...
CREATE FUNCTION foo(int, out text) ...
```

異なる引数型のリストを持つ関数は、作成時に競合するとはみなされませんが、デフォルト値が指定された場合使用時に競合する可能性があります。 例えば以下を考えてみましょう。

```
CREATE FUNCTION foo(int) ...
CREATE FUNCTION foo(int, int default 42) ...
```

`foo(10)`という呼び出しは、どちらの関数を呼び出すべきかに関して曖昧さがあるために失敗します。

注釈

完全なSQL型の構文により入力引数と戻り値が許されます。 しかし、一部の型指定の細部(例えば`numeric`型の精度フィールド)は、背後の関数の実装に責任があり、`CREATE FUNCTION`コマンドにより警告なく飲み込まれてしまいます(つまり認識も強制もされません)。

既存の関数を`CREATE OR REPLACE FUNCTION`を使って置き換える場合、パラメータ名の変更に関して制限があります。 すでに何らかの入力パラメータに割り当てられた名前を変更することはできません。 （しかし、これまで名前を持たなかったパラメータに名前を追加することは可能です。） 複数の出力パラメータが存在する場合、関数の結果を表わす無名複合型の列名を変更することになるため、出力パラメータの名前を変更することはできません。 既存の関数呼び出しが置き換わった時に動作しなくなることを確実に防ぐために、これらの制限がなされています。

関数が`VARIADIC`引数を持つ`STRICT`と宣言された場合、その厳密性検査では、可変長配列全体が非NULLかどうかを検査します。 配列がNULL要素を持っていたとしても関数は呼び出されます。

例

ここでは、初心者向けの簡単な例をいくつか示します。 [項35.3](#)には、より多くの情報と例が記載されています。

```
CREATE FUNCTION add(integer, integer) RETURNS integer
AS 'select $1 + $2;'
LANGUAGE SQL
IMMUTABLE
RETURNS NULL ON NULL INPUT;
```

PL/pgSQLで、引数名を使用して、整数を1増やします。

```
CREATE OR REPLACE FUNCTION increment(i integer) RETURNS integer AS $$
BEGIN
    RETURN i + 1;
END;
$$ LANGUAGE plpgsql;
```

複数の出力用パラメータを持つレコードを返します。

```
CREATE FUNCTION dup(in int, out f1 int, out f2 text)
AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
LANGUAGE SQL;

SELECT * FROM dup(42);
```

上と同じことを、明示的な名前が付いた複合型を使用して、より冗長に行うことができます。

```
CREATE TYPE dup_result AS (f1 int, f2 text);

CREATE FUNCTION dup(int) RETURNS dup_result
AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
LANGUAGE SQL;

SELECT * FROM dup(42);
```

複数列を返す別の方法は、`TABLE`関数を使用することです。

```
CREATE FUNCTION dup(int) RETURNS TABLE(f1 int, f2 text)
AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
LANGUAGE SQL;

SELECT * FROM dup(42);
```

しかし、これは実際には、1つのレコードではなく、レコードの集合を返しますので、`TABLE`関数は上の例とは異なります。

SECURITY DEFINER関数の安全な作成

`SECURITY DEFINER`関数は関数を作成したユーザの権限で実行されますので、その関数を間違って使用できないことを確実にしなければなりません。安全上、[search_path](#)は、信頼できないユーザが書き込み可能なスキーマを除去した形で設定されます。これは、悪意のあるユーザがその関数で使われるオブジェクトを隠すようなオブジェクトを作成することを防ぎます。ここで特に重要なことは、一時テーブルスキーマです。このスキーマはデフォルトで最初に検索され、そして、通常誰でも書き込み可能です。一時スキーマの検索を強制的に最後にすることで、セキュリティを調整できます。このためには、`pg_temp`を[search_path](#)の最後の項目として記載してください。この関数の安全な使用方法を以下に示します。

```
CREATE FUNCTION check_password(uname TEXT, pass TEXT)
RETURNS BOOLEAN AS $$
DECLARE passed BOOLEAN;
BEGIN
    SELECT (pwd = $2) INTO passed
    FROM   pwds
    WHERE  username = $1;

    RETURN passed;
END;
$$ LANGUAGE plpgsql
SECURITY DEFINER
-- 信頼できるスキーマ、その後にpg_tempという順でsearch_pathを安全に設定します。
SET search_path = admin, pg_temp;
```

PostgreSQLバージョン8.3より前では、`SET`オプションは利用できません。このため古い関数には、`search_path`を保管し、設定、そして元に戻すという、多少複雑なロジックが含まれているかもしれません。こうした目的に`SET`オプションを使用すると、かなり簡単になります。

この他に注意すべき点として、新しく作成された関数ではデフォルトで実行権限が`PUBLIC`に付与されていることがあります。（詳細は[GRANT](#)を参照してください。）`SECURITY DEFINER`関数の使用を一部のユーザのみに制限したいことはよくありま

す。このためには、デフォルトのPUBLIC権限を取り上げ、そして、実行権限の付与を選択して行ってください。新しい関数がすべてのユーザに実行可能となる隙間が存在することを防ぐためには、単一ランザクション内で作成と権限設定を行ってください。以下に例を示します。

```
BEGIN;  
CREATE FUNCTION check_password(uname TEXT, pass TEXT) ... SECURITY DEFINER;  
REVOKE ALL ON FUNCTION check_password(uname TEXT, pass TEXT) FROM PUBLIC;  
GRANT EXECUTE ON FUNCTION check_password(uname TEXT, pass TEXT) TO admins;  
COMMIT;
```

互換性

CREATE FUNCTION コマンドはSQL:1999以降で定義されています。PostgreSQLにおけるCREATE FUNCTIONも類似の機能を持ちますが、完全な互換性はありません。属性には移植性がありません。また、使用可能な言語も異なります。

他のデータベースシステムとの互換性のために、argmodeはargnameの前に書くことも後に書くこともできます。しかし、1つ目の方法が標準に従っています。

標準SQLはパラメータのデフォルトを規定していません。DEFAULT キーワードを持つ構文はOracleに由来したもので、標準の流れを多少汲んだものです。SQL/PSMではこれを変数のデフォルト値として使用しています。=を持つ構文はT-SQLおよびFirebirdで使用されています。

関連項目

[ALTER FUNCTION](#), [DROP FUNCTION](#), [GRANT](#), [LOAD](#), [REVOKE](#), [createlang](#)

前のページ	ホーム	次のページ
CREATE FOREIGN TABLE	上に戻る	CREATE GROUP

CREATE GROUP

名前

CREATE GROUP -- 新しいデータベースロールを定義する

概要

```
CREATE GROUP name [ [ WITH ] option [ ... ] ]
```

*option*は次のようになります。

```
    SUPERUSER | NOSUPERUSER
| CREATEDB | NOCREATEDB
| CREATEROLE | NOCREATEROLE
| CREATEUSER | NOCREATEUSER
| INHERIT | NOINHERIT
| LOGIN | NOLOGIN
| [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
| VALID UNTIL 'timestamp'
| IN ROLE role_name [, ...]
| IN GROUP role_name [, ...]
| ROLE role_name [, ...]
| ADMIN role_name [, ...]
| USER role_name [, ...]
| SYSID uid
```

説明

CREATE GROUP は[CREATE ROLE](#)の別名になりました。

互換性

標準SQLにはCREATE GROUPはありません。

関連項目

[CREATE ROLE](#)

[前のページ](#)

CREATE FUNCTION

[ホーム](#)
[上に戻る](#)

[次のページ](#)
CREATE INDEX

CREATE INDEX

名前

CREATE INDEX -- 新しいインデックスを定義する

概要

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ name ] ON table_name [ USING method ]
( { column_name | ( expression ) } [ COLLATE collation ] [ opclass ] [ ASC | DESC ] [ NULLS { FIRST |
LAST } ] [, ...] )
[ WITH ( storage_parameter = value [, ... ] ) ]
[ TABLESPACE tablespace_name ]
[ WHERE predicate ]
```

説明

CREATE INDEX は、指定したテーブルの指定した列(複数可)に対するインデックスを作ります。インデックスは主にデータベースの性能を向上するために使われます（しかし、インデックスの不適切な使用は性能の低下につながる可能性があります）。

インデックスのキーフィールドは、列名、または括弧に囲まれた式として指定されます。インデックスメソッドが複数列に対するインデックスをサポートする場合は、複数のフィールドを指定できます。

インデックスのフィールドとして、テーブル行の1つ以上の列の値を計算する式を指定できます。この機能は、元のデータに何らかの変換を加えた値を基とするデータへの高速なアクセスを行う手段として使用することができます。例えば、upper(col) という計算に基づくインデックスがあれば、WHERE upper(col) = 'JIM' 句ではインデックスを使用することができます。

PostgreSQLはB-tree、ハッシュ、GiST、SP-GiST、GINのインデックスメソッドを用意しています。ユーザが独自にインデックスメソッドを定義することもできますが、これはかなり複雑です。

WHERE 句が存在する場合、部分インデックスが作成されます。部分インデックスは、テーブルの一部、通常は、テーブルの中でよりインデックスが有用な部分のみのエントリを持つインデックスです。例えば、支払済みの注文と未支払の注文を情報として持つテーブルがあり、テーブル全体における未支払の注文の割合が小さく、かつ、頻繁に使用される場合、未支払の注文のみにインデックスを作成することで性能を向上できます。部分インデックスのその他の利用方法として、UNIQUE 付きのWHERE を使用して、テーブルの一部に一意性を強制する例が考えられます。詳細は[項11.8](#)を参照してください。

WHERE 句内の式では、元となるテーブルの列のみを参照できます。しかし、インデックスを付加する列だけではなく、全ての列を使用することができます。また、現在、副問い合わせと集約式については、WHERE で使用することができません。同一の制限は、式で表されたインデックスのフィールドにも適用されます。

インデックスの定義で使用される全ての関数と演算子は、"不変" (immutable) でなければなりません。つまり、結果は入力引数にのみに依存し、（他のテーブルの内容や現時刻などの）外部からの影響を受けてはなりません。この制限によって、インデックスの動作が十分定義されていることが保証されます。インデックス式やWHERE 句にユーザ定義の関数を使用する場合、関数を作成する際、IMMUTABLE（不変）オプションを付けることを忘れないでください。

パラメータ

UNIQUE

インデックスを（既にデータがある状態で）作成する時、テーブルにデータを追加する時に、テーブル内の値が重複していないかをチェックします。重複エントリを生じるデータの挿入または更新はエラーとなります。

CONCURRENTLY

このオプションを使用すると、PostgreSQLは、対象テーブルに対する同時挿入、更新、削除を防止するようなロックを獲得せずにインデックスを作成します。通常のインデックス作成処理では、完了するまで対象テーブルへの書き込みはロックされます（読み取りはロックされません）。このオプションを使用する際に注意しなければならない点が複数あります。[インデックスの同時作成](#)を参照してください。

name

作成するインデックスの名前です。この名前には、スキーマ名を含めることはできません。インデックスは、常にその

親テーブルと同じスキーマに作成されます。この名前を省略すると、PostgreSQLはその親テーブルの名前とインデックス付けされる列名に基づいた適切な名前を選びます。

table_name

インデックスを作成するテーブルの名前です（スキーマ修飾名の場合もあります）。

method

使用するインデックスメソッドの名前です。 `btree`、`hash`、`gist`、`spgist`、`gin`から選択します。デフォルトのメソッドは**`btree`**です。

column_name

テーブルの列の名前です。

expression

テーブル上の1つ以上の列を使用した式です。通常この式は、構文で示した通り括弧で囲む必要があります。しかし、式が関数呼び出し形式になっている場合は括弧を省略することができます。

collation

インデックスで使用する照合順序の名前です。デフォルトではインデックスはインデックス付け対象の列で宣言された照合順序またはインデックス付け対象の式の結果照合順序を使用します。デフォルト以外の照合順序を使用する式を含む問い合わせで、デフォルト以外の照合順序を持つインデックスが有用になるかもしれません。

opclass

演算子クラスの名前です。詳細は下記を参照してください。

`ASC`

（デフォルトの）正方向のソート順を指定します。

`DESC`

逆方向のソート順を指定します。

`NULLS FIRST`

`NULL`を非`NULL`より前にソートすることを指定します。これは`DESC`が指定された場合のデフォルトです。

`NULLS LAST`

`NULL`を非`NULL`より後にソートすることを指定します。これは`DESC`が指定されない場合のデフォルトです。

storage_parameter

インデックス固有の格納パラメータの名前です。 [インデックス格納パラメータ](#)を参照してください。

tablespace_name

インデックスを生成するテーブル空間です。指定されなかった場合、[default tablespace](#)、もし一時テーブル上のインデックスであれば、[temp tablespaces](#)が考慮されます。

predicate

部分インデックス用の制約式です。

インデックス格納パラメータ

省略可能な**`WITH`**句では、インデックスに対し格納パラメータを指定します。インデックスメソッドはそれぞれ固有の設定可能な格納パラメータを持ちます。B-tree、ハッシュ、GiSTおよびSP-GiSTといったインデックスはすべて以下の1つのパラメータを受け付けます。

`FILLFACTOR`

インデックス用のフィルファクタは割合（パーセント）で、インデックスメソッドがインデックスページをまとめ上げる時にどの程度ページを使用するかを決定するものです。B-treeでは、リーフページは初期インデックス構築時と右側（新しい最大キー値を追加する方向）にインデックスを拡張する時にこの割合分ページを使用します。その後ページすべてが完全に使用されると、インデックスの効果が緩やかに劣化するように分割されます。B-treeのデフォルトのフィルファクタは90ですが、10から100までの任意の整数値を設定することができます。テーブルの変化がほとんどない場合、100が最善でインデックスの物理サイズが最小化できます。更新が非常に多い場合は、より小さなフィルファクタを

設定することで、ページ分割の頻度を少なくすることができます。この他のインデックスメソッドでは、フィルファクタをおおよそは同じですが、異なる意味で使用します。メソッドによってフィルファクタのデフォルト値は異なります。

GiSTインデックスではさらに以下のパラメータを受け付けます。

BUFFERING

[項53.3.1](#)で説明するバッファリング構築技術をインデックスを構築する時に使用するかどうかを決定します。OFFで無効に、ONで有効になります。さらにAUTOと指定すると、最初は無効ですが、インデックスサイズが[effective cache size](#)に達した後はその場で有効になります。デフォルトはAUTOです。

GINインデックスでは以下の異なるパラメータを受け付けます。

FASTUPDATE

この設定は[項55.3.1](#)で説明する高速更新技法を使用するかどうかを制御します。これは論理値パラメータであり、ONは高速更新を有効に、OFFは無効にします。（[項18.1](#)の説明のように、ONとOFFの別記載も可能です。）デフォルトはONです。

注意: ALTER INDEXを使用してFASTUPDATEを無効にすることにより、今後の挿入は待機中のインデックス項目リストに入らないようになります。しかし、このコマンド自体はこれまでの項目を吐き出しません。確実に待機中のリストを空にするためには、続いてテーブルをVACUUMする方が良いでしょう。

インデックスの同時作成

インデックスの作成が、通常のデータベース操作に影響を与えることがあります。通常PostgreSQLは、対象テーブルに対する書き込みをロックしてから、対象テーブルを一度スキャンして全体をインデックス付けします。他のトランザクションはテーブルを読み取ることはできますが、対象テーブル内の行を挿入、更新、削除しようとする、インデックス作成が完了するまでブロックされます。活発な運用状態のデータベースシステムの場合、これは重大な影響を与えます。非常に大規模なテーブルに対するインデックス作成は何時間もかかることがあります。また小規模なテーブルであっても、インデックス作成により、運用状態のシステムでは受け入れられない期間、書き込みロックがかかる可能性があります。

PostgreSQLは書き込みをロックしないインデックス作成もサポートしています。CREATE INDEXにCONCURRENTLYオプションをつけることでこの方式が行われます。このオプションを使うと、PostgreSQLはテーブルを2回スキャンしなければなりません。さらに、潜在的にそのインデックスを使用する可能性がある、実行中のすべてのトランザクションが終わるまで待機しなければなりません。したがって、この方式は通常の方式よりも総作業時間がかかり、また、完了するまでの時間が非常に長くなります。しかし、インデックス作成中に通常の操作を行い続けることができますので、この方式は運用環境での新規インデックス作成に有用です。もちろん、インデックス作成によりCPUや入出力に負荷がかかりますので、他の操作が低速になる可能性があります。

同時実行インデックス構築では実際、第一トランザクションでシステムカタログに登録され、第二以降のトランザクションで2つのテーブルスキャンが起こります。2回目のテーブルスキャンが始まった後にそのテーブルを参照するのみのトランザクションであったとしても、2回目のテーブルスキャンが始まった時に有効なトランザクションは、その完了まで、同時実行インデックス作成をブロックする可能性があります。同時実行インデックス作成は、[項45.58](#)節で概要を示す方法を用いて、古いトランザクションが完了することを直列的に待機します。

たとえば一意性インデックスにおける一意性違反など、テーブルスキャン中に問題が発生すると、CREATE INDEXは失敗しますが、"無効な"インデックスが残ってしまいます。こうしたインデックスは完全ではない可能性がありますので、問い合わせの際には無視されます。しかし、更新時にオーバーヘッドがかかります。psqlの\dコマンドでは、こうしたインデックスをINVALIDとして報告します。

```
postgres=# \d tab
          Table "public.tab"
  Column |      Type      | Modifiers
-----+-----+-----
   col   | integer        |
Indexes:
    "idx" btree (col) INVALID
```

こうした場合の推奨復旧方法は、インデックスを削除し、再度CREATE INDEX CONCURRENTLYを実行することです。（他にもREINDEXを使用したインデックスの再作成という方法もあります。しかし、REINDEXは同時作成をサポートしていないので、この方法は魅力がありません。）

この他に一意性インデックスを同時作成する場合の注意事項があります。2回目のテーブルスキャンが始まる時点で、他のトランザクションに対する一意性制約が既に有効になっているという点です。これは、インデックスが使用できるようになる前やインデックス作成が最終的に失敗したとしても、制約違反が他のトランザクションで報告されてしまうことを意味します。また、2回目のスキャン中に失敗した後も、"無効な"インデックスによる一意性制約は強制され続けます。

式インデックスや部分インデックスの同時作成もサポートされています。式の評価中にエラーが発生した場合も、上で説明した

一意性制約違反と同様な状況が発生します。

同一テーブルに対する通常のインデックス作成処理は複数並行して行うことができます。しかし、あるテーブルに対するインデックスの同時作成は一度に1つしか行うことができません。また、どの場合でもインデックス作成中はテーブルスキーマの変更はできません。この他に、通常のCREATE INDEX コマンドはトランザクションブロック内で実行させることができますが、CREATE INDEX CONCURRENTLY は実行させることができないという相違点があります。

注釈

インデックスが、どのような時に使用され、どのような時に使用されないか、また、どのような場合に有用かといった情報については第11章を参照してください。

注意

ハッシュインデックス操作は現在WALに記録されません。このためハッシュインデックスは、データベースクラッシュの後未書き込みな変更があれば、REINDEX を用いて再構築しなければならない可能性があります。また、ハッシュインデックスの変更は最初のベースバックアップ以降のストリーミングまたはファイルベースのレプリケーションでは複製されません。ですので、その後にそのインデックスを使用する問い合わせに対して間違った答えが返ります。このような理由から現在ハッシュインデックスの使用を勧めません。

現在は、B-tree、GiST、GINインデックスメソッドのみが、複数列に対するインデックスをサポートしています。指定できる列は、デフォルトでは32個までです（この制限はPostgreSQL構築の際に変更できます）。現在、B-treeのみが一意的インデックスをサポートしています。

演算子クラスは、インデックスのそれぞれの列に指定することができます。演算子クラスは、その列のインデックスが使う演算子を識別します。例えば、4バイト整数に対するB-treeインデックスには、int4_ops クラスを使います。この演算子クラスには、4バイト整数の比較関数が含まれています。実際の使用では、通常、列のデータ型のデフォルト演算子クラスで十分です。演算子クラスを保持する主な理由は、データ型の中には有意な順序を2つ以上持つものがあるかもしれないからです。例えば、絶対値または実数部のどちらかを使って複素数のデータ型をソートしたい場合があります。これを実現するには、データ型として2つの演算子クラスを定義し、インデックスを作る時に適切なクラスを選択します。演算子クラスについての詳細は、項11.9と項35.14を参照してください。

順序付きスキャンをサポートするインデックスメソッド（現時点ではB-Treeのみ）では、省略可能なASC、DESC、NULLS FIRST、NULLS LAST 句を指定し、インデックスのソート順を変更させることができます。順序付きインデックスは正方向にも逆方向にもスキャンすることができますので、単一列に対するDESCインデックスは通常は有用ではありません。このソート順序はすでに通常のインデックスを使用して実現できます。これらのオプションの価値は、SELECT ... ORDER BY x ASC, y DESCなどの順序指定が混在する問い合わせによって要求されるソート順に一致する、複数列に対するインデックスを作成できる点です。NULLS オプションは、ソート処理を中断するインデックスに基づいた問い合わせにおける"NULLのソート順を低くする"動作をサポートする必要がある場合に有用です。デフォルトの動作は"NULLのソート順を高くする"です。

ほとんどのインデックスメソッドにおいて、インデックス作成速度はmaintenance_work_mem の設定に依存します。より大きな値を設定すると、マシンをスワップ状態にしまわれないように実際に使用できるメモリ量を超えないようにしている限り、インデックス作成が必要となる回数を減少します。ハッシュインデックスでは、effective_cache_size の値もインデックス作成時間に影響を与えます。PostgreSQLは、推定インデックスサイズがeffective_cache_size より大きいか小さいかによって、2つのハッシュインデックス作成方法の内どちらかを使用します。最善の結果を得るために、このパラメータが利用可能なメモリを反映した値に設定されていることを確認してください。また、maintenance_work_mem とeffective_cache_size の和が、マシンのRAM容量から他のプログラムで必要とされる容量を差し引いた容量より小さくなるよう注意してください。

インデックスを削除するには、DROP INDEX を使用してください。

以前のPostgreSQLにはR-treeインデックスメソッドがありました。GiSTメソッドに比べて大きな利点がありませんでしたので、このメソッドは削除されました。古いデータベースからGiSTへの変換を簡単にするため、USING rtree が指定された場合、CREATE INDEX はUSING gist と解釈します。

例

films テーブルのtitle 列にB-treeインデックスを作成します。

```
CREATE UNIQUE INDEX title_idx ON films (title);
```

大文字小文字を区別しない検索が効率的になるように、lower(title) 式に対してインデックスを作成します。

```
CREATE INDEX ON films ((lower(title)));
```

(この例では、インデックス名を省略することを選びました。 このためシステムが名前、通常はfilms_lower_idxという名前を選ぶことになるでしょう。)

デフォルト以外の照合順序でインデックスを作成します。

```
CREATE INDEX title_idx_german ON films (title COLLATE "de_DE");
```

デフォルト以外のNULLのソート順を指定したインデックスを作成します。

```
CREATE INDEX title_idx_nulls_low ON films (title NULLS FIRST);
```

デフォルト以外のフィルファクタを持つインデックスを作成します。

```
CREATE UNIQUE INDEX title_idx ON films (title) WITH (fillfactor = 70);
```

高速更新を無効にしてGINインデックスを作成します。

```
CREATE INDEX gin_idx ON documents_table USING gin (locations) WITH (fastupdate = off);
```

films テーブル上にcode列に対するインデックスを作成します。 また、このインデックスをindexspace テーブル空間内に生成します。

```
CREATE INDEX code_idx ON films (code) TABLESPACE indexspace;
```

変換関数の結果に対するbox操作を効率的に使用できるようにpoint属性にGiSTインデックスを作成します。

```
CREATE INDEX pointloc
ON points USING gist (box(location,location));
SELECT * FROM points
WHERE box(location,location) && '(0,0),(1,1)::box;
```

対象テーブルへの書き込みをロックせずにインデックスを作成します。

```
CREATE INDEX CONCURRENTLY sales_quantity_index ON sales_table (quantity);
```

互換性

CREATE INDEXはPostgreSQLの拡張です。 標準SQLにはインデックスについての規定はありません。

関連項目

[ALTER INDEX](#), [DROP INDEX](#)

[前のページ](#)
CREATE GROUP

[ホーム](#)
[上に戻る](#)

[次のページ](#)
CREATE LANGUAGE

CREATE LANGUAGE

名前

CREATE LANGUAGE -- 新しい手続き言語を定義する

概要

```
CREATE [ OR REPLACE ] [ PROCEDURAL ] LANGUAGE name
CREATE [ OR REPLACE ] [ TRUSTED ] [ PROCEDURAL ] LANGUAGE name
    HANDLER call_handler [ INLINE inline_handler ] [ VALIDATOR valfunction ]
```

説明

CREATE LANGUAGEは新しい手続き言語をPostgreSQLデータベースに登録します。これにより、関数とトリガプロシージャを新しい言語で定義できるようになります。

注意: PostgreSQL 9.1から、ほとんどの手続き言語は"拡張"としてまとめられました。したがってCREATE LANGUAGEではなく[CREATE EXTENSION](#)を使用してインストールすべきです。CREATE LANGUAGEを直接使用することは拡張インストール用スクリプトに限定されなければなりません。おそらくアップグレードの結果"むきだし"の言語がデータベース内に存在する場合、CREATE EXTENSION *langname* FROM unpackagedを用いて拡張に変換することができます。

CREATE LANGUAGEは、言語名とその言語で作成された関数の実行に責任を持つハンドラ関数を、効率的に関連付けます。言語ハンドラについての詳細は、[第49章](#)を参照してください。

CREATE LANGUAGEコマンドには2種類の形式があります。最初の形式は、ユーザは必要な言語の名前のみを単に指定するものです。この場合、PostgreSQLサーバはpg_pltemplateシステムカタログを参照し、正しいパラメータを決定します。2番目の形式では、ユーザは言語名の他に言語のパラメータも入力します。この形式で、pg_pltemplateで定義されていない言語を作成することができますが、この手法は廃棄予定です。

指定された言語名に対応するpg_pltemplateカタログの項目をサーバが見つけると、コマンドに言語のパラメータが含まれていたとしてもカタログのデータを使用します。この動作により、古すぎる言語のサポート関数に関する情報を含む、古いダンプファイルからの読み込みが簡単になります。

通常、新しい言語に登録するには、ユーザはPostgreSQLのスーパーユーザ権限を持たなければなりません。しかし、言語がpg_pltemplateカタログ内に記載され、データベース所有者による作成可能印 (tmpldbacreateが真) が付いていれば、データベース所有者はそのデータベース内に新しい言語に登録することができます。デフォルトで、信頼された言語はデータベース所有者による作成ができます。しかし、これはスーパーユーザによるpg_pltemplateの変更により調整できます。言語作成者が言語の所有者になり、後で削除したり、名前を変えたり、別の所有者を割り当てたりすることができます。

CREATE OR REPLACE LANGUAGEは新しい言語を作成、または、既存の定義を置き換えます。言語がすでに存在する場合、パラメータは指定された値またはpg_pltemplateから取り出された値にしたがって更新されますが、言語の所有権と権限に関する設定は変更されません。また、その言語で作成された既存の関数も以前として有効であるものと仮定されます。言語を作成する際に通常要求される権限に加え、ユーザはスーパーユーザまたは既存言語の所有者でなければなりません。REPLACEの場合、主に言語が存在することを確認するために使用することを意味します。その言語がpg_pltemplate項目を持つ場合、REPLACEは既存の定義を実際にはまったく変更しません。ただし、言語が作成されてからpg_pltemplate項目が変更された通常ではない場合は例外です。

パラメータ

TRUSTED

TRUSTEDは、他の方法ではユーザがアクセスできないデータに対しては、その言語でのアクセスが許されないことを指定します。言語の登録時にこのキーワードを省略すると、PostgreSQLのスーパーユーザ権限を持つユーザのみが、この言語を使って新しい関数を作れるようになります。

PROCEDURAL

これはノイズワードです。

name

新しい手続き言語の名前です。この名前はデータベースの言語の中で一意でなければなりません。

後方互換を保持するため、この名前を単一引用符で囲むこともできます。

HANDLER *call_handler*

*call_handler*は手続き言語の関数を実行するために呼び出される関数の名前で、事前に登録しておく必要があります。このハンドラは、Version-1呼び出し規約に則って、C言語のようなコンパイル言語で書かれている必要があります。また、引数を取らず *language_handler* 型を返す関数として、PostgreSQLに登録されていなければなりません。*language_handler* 型は、単に関数を呼び出しハンドラとして識別するのに使用するプレースホルダ型です。

INLINE *inline_handler*

*inline_handler*はこの言語で無名コードブロックを実行 ([DO](#) コマンド) するために呼び出される、事前に登録された関数の名前です。*inline_handler* 関数が指定されない場合、その言語では無名コードブロックをサポートしません。このハンドラ関数は、[DO](#) コマンドの内部表現となる *internal* 型の引数を1つ取らなければならない、また、通常 *void* を返します。ハンドラの戻り値は無視されます。

VALIDATOR *valfunction*

*valfunction*は、事前に登録された検証用関数の名前です。新しい関数が当該言語で作成された場合、その関数を検証するために呼び出されます。検証用関数が指定されていない場合、新しい関数は作成時にチェックされません。検証用関数は、*oid* 型の引数を1つ取る必要があります。この引数は作成される関数のOIDになります。また、通常 *void* 型を返します。

検証用関数は、通常、関数本体が構文上正しいかどうかを検査するために使用されます。しかし、それだけでなく、関数のプロパティも検査可能です。例えば、その言語が処理できない特定のデータ型が引数に含まれていないかなどがチェックできます。エラーを通知するには、検証用関数で `ereport()` 関数を使用すべきです。関数の戻り値は無視されます。

サーバが指定した言語名に対応する *pg_pltemplate* の項目を見つけた場合、`TRUSTED` オプションとサポート関数名は無視されます。

注釈

[createlang](#)は `CREATE LANGUAGE` に対する簡単なラッププログラムです。これを使用して簡単にシェルコマンドラインから手続き言語をインストールすることができます。

手続き言語を削除するには、[DROP LANGUAGE](#)、または [droplang](#) を使用してください（後者を勧めます）。

システムカタログ *pg_language* ([項45.27](#)参照) には、現在インストールされている言語に関する情報が記録されています。また、[createlang](#) にはインストールされた言語を列挙するオプションがあります。

手続き言語で関数を作成するには、ユーザはその言語に対する `USAGE` 権限を持たなければなりません。デフォルトでは、信頼された言語の `USAGE` は `PUBLIC` に（つまり全員に）付与されています。これは必要に応じて取り除くことができます。

手続き言語は各データベースに局所的です。しかし、言語を *template1* データベースにインストールすることができます。この場合、その後作成されたすべてのデータベースで自動的にその言語は使用できるようになります。

呼び出しハンドラ関数とインラインハンドラ関数（もしあれば）、検証関数（もしあれば）は、サーバが *pg_pltemplate* 内にその言語に関する項目を持っていない場合には、存在しなければなりません。しかし、その項目があったとしても、関数がすでに存在しているとは限りません。データベース内に存在していなければ自動的に定義されます。（言語を実装する共有ライブラリがインストレーションで利用できない場合、これにより `CREATE LANGUAGE` が失敗する可能性があります。）

PostgreSQL 7.3より前のバージョンでは、ハンドラ関数が返すプレースホルダ型を *language_handler* 型ではなく *opaque* 型と宣言する必要がありました。古いダンプファイルのロードをサポートするために、`CREATE LANGUAGE` は *opaque* 型を返すと宣言された関数を受け入れます。しかし、注意を促すメッセージを表示し、宣言された関数の戻り値型を *language_handler* に変換します。

例

標準的な手続き言語を作成する推奨方法は単に以下に行うことです。

```
CREATE LANGUAGE plperl;
```

pg_pltemplateカタログ内に存在しない言語では、以下のような処理が必要です。

```
CREATE FUNCTION plsample_call_handler() RETURNS language_handler
AS '$libdir/plsample'
LANGUAGE C;
CREATE LANGUAGE plsample
HANDLER plsample_call_handler;
```

互換性

CREATE LANGUAGEはPostgreSQLの拡張です。

関連項目

[ALTER LANGUAGE](#), [CREATE FUNCTION](#), [DROP LANGUAGE](#), [GRANT](#), [REVOKE](#), [createlang](#), [droplang](#)

[前のページ](#)

CREATE INDEX

[ホーム](#)

[上に戻る](#)

[次のページ](#)

CREATE OPERATOR

CREATE OPERATOR

名前

CREATE OPERATOR -- 新しい演算子を定義する

概要

```
CREATE OPERATOR name (
    PROCEDURE = function_name
    [, LEFTARG = left_type ] [, RIGHTARG = right_type ]
    [, COMMUTATOR = com_op ] [, NEGATOR = neg_op ]
    [, RESTRICT = res_proc ] [, JOIN = join_proc ]
    [, HASHES ] [, MERGES ]
)
```

説明

CREATE OPERATORは、新しい演算子`name`を定義します。演算子を定義したユーザがその所有者となります。スキーマ名が指定されていた場合、その演算子は指定したスキーマで作成されます。スキーマ名が指定されなかった場合、現在のスキーマで作成されます。

演算子名として使用できるのは、以下に示す文字を使った、NAMEDATALEN-1（デフォルトでは63）文字までの文字列です。

+ - * / < > = ~ ! @ # % ^ & | ` ?

名前の選択には以下に示すいくつかの制約があります。

- と/*は演算子名の一部として使うことができません。コメントの開始とみなされてしまうからです。
- 複数の文字からなる演算子名は、下記の文字のうちの1つ以上を含まない限り、+または-で終わることができません。

~ ! @ # % ^ & | ` ?

例えば、@-は演算子名として許されますが、*-は許されません。この制約により、PostgreSQLがSQLに準拠する問い合わせをトークン同士の間に空白を要求することなしに解析することが可能になります。

- 演算子名として=>を使用することは廃止される予定です。将来のリリースで完全に許されなくなるかもしれません。

演算子!=は入力時に<>に変換されるので、これらの2つの名前は常に等価です。

少なくともLEFTARGとRIGHTARGのどちらかは定義しなければなりません。二項演算子では、両方を定義しなければなりません。右単項演算子ではLEFTARGのみ、左単項演算子ではRIGHTARGのみが定義されなければなりません。

`function_name` プロシージャは、CREATE FUNCTIONを使って事前に定義されていなければなりません。また、指定された型の正しい数の引数（1つか2つ）を受け付けるよう定義する必要があります。

他の句は省略可能な演算子最適化用の句です。これらの意味は[項35.13](#)で説明されています。

演算子を作成するためには、引数の型と戻り値の型に対するUSAGE権限と背後にある関数に対するEXECUTE権限を持たなければなりません。交代演算子または否定子演算子が指定された場合は、これらの演算子を所有していなければなりません。

パラメータ

`name`

定義される演算子の名前です。使用できる文字は上を参照してください。この名前は、例えばCREATE OPERATOR myschema.+ (...)のように、スキーマ修飾可能です。修飾されていなければ、演算子は現在のスキーマで作成されます。異なるデータ型について処理するものであれば、同じスキーマ内の2つの演算子は同じ名前を持つことができます。これをオーバーロードと言います。

`function_name`

演算子を実装するために使用する関数です。

left_type

演算子の左オペランドのデータ型です（左オペランドが存在する場合のみ）。 このオプションは左単項演算子では省略されます。

right_type

演算子の右オペランドのデータ型です（右オペランドが存在する場合のみ）。 このオプションは右単項演算子では省略されます。

com_op

この演算子の交代演算子です。

neg_op

この演算子の否定子です。

res_proc

この演算子の制約選択評価関数です。

join_proc

この演算子の結合選択評価関数です。

HASHES

この演算子がハッシュ結合をサポートできることを示します。

MERGES

この演算子がマージ結合をサポートできることを示します。

スキーマ修飾された演算子名を*com_op*または他のオプション引数に与えるには、以下の例のように`OPERATOR()` 構文を使用してください。

```
COMMUTATOR = OPERATOR(myschema.==) ,
```

注釈

詳細については[項35.12](#)を参照してください。

`CREATE OPERATOR` で演算子の語彙優先順位を指定することはできません。 パーサの優先順位に関する動作は固定であるためです。 詳細な優先順位については[項4.1.6](#)を参照してください。

廃止されたオプション`SORT1`、`SORT2`、`LTCMP`、および`GTCMP` は、マージ結合可能演算子に関連したソート演算子の名前を指定するために使用されていました。 代わりにB-Tree演算子族を検索することで関連する演算子を見つけることができるようになりましたので、これは必要がなくなりました。 これらの内のいずれかのオプションが指定された場合、暗黙的に`MERGES` を真に指定している場合を除き、無視されます。

データベースからユーザ定義の演算子を削除するには[DROP OPERATOR](#)を使用してください。 データベース内の演算子を変更するには[ALTER OPERATOR](#)を使用してください。

例

以下のコマンドは、`box`データ型に対する領域等価性を判定する新しい演算子を定義します。

```
CREATE OPERATOR == (
  LEFTARG = box,
  RIGHTARG = box,
  PROCEDURE = area_equal_procedure,
  COMMUTATOR = ==,
  NEGATOR = !=,
  RESTRICT = area_restriction_procedure,
  JOIN = area_join_procedure,
  HASHES, MERGES
);
```


互換性

CREATE OPERATORはPostgreSQLの拡張です。 標準SQLには、ユーザ定義の演算子についての規定はありません。

関連項目

[ALTER OPERATOR](#), [CREATE OPERATOR CLASS](#), [DROP OPERATOR](#)

[前のページ](#)
CREATE LANGUAGE

[ホーム](#)
[上に戻る](#)

[次のページ](#)
CREATE OPERATOR CLASS

CREATE OPERATOR CLASS

名前

CREATE OPERATOR CLASS -- 新しい演算子クラスを定義する

概要

```
CREATE OPERATOR CLASS name [ DEFAULT ] FOR TYPE data_type
    USING index_method [ FAMILY family_name ] AS
    { OPERATOR strategy_number operator_name [ ( op_type, op_type ) ] [ FOR SEARCH | FOR ORDER BY
sort_family_name ]
      | FUNCTION support_number [ ( op_type [ , op_type ] ) ] function_name ( argument_type [ , ... ] )
      | STORAGE storage_type
    } [ , ... ]
```

説明

CREATE OPERATOR CLASSは新しい演算子クラスを作成します。演算子クラスは、特定のデータ型がインデックスでどのように使用されるかを定義します。演算子クラスにより、データ型およびインデックスメソッドの特定の役割もしくは"戦略"に、どの演算子を使用するかが指定されます。また、インデックスの列に対して演算子クラスが選択される際、演算子クラスによってインデックスメソッドが使用するサポートプロシージャが指定されます。演算子クラスで使用される全ての演算子および関数は、演算子クラスを作成できるようになる前に定義しておく必要があります。

スキーマ名が与えられている場合、その演算子クラスは指定されたスキーマで作成されます。スキーマ名がなければ、演算子クラスは現在のスキーマで作成されます。異なるインデックスメソッドに使用する場合のみ、同じスキーマ内の2つの演算子クラスに同じ名前を付けることができます。

演算子クラスは、それを定義したユーザが所有者となります。現在、演算子クラスを作成するには、スーパーユーザである必要があります。（誤った演算子クラスを定義すると、混乱の元となり、サーバクラッシュの原因とさえなり得るため、この制限が付けられています）。

現在、CREATE OPERATOR CLASSでは、インデックスメソッドに必要な全ての演算子および関数が演算子クラス定義に含まれているかどうか、また、演算子や関数の形式がそれ自身で整合性を持っているか进行检查しません。ユーザの責任において、有効な演算子クラスを定義してください。

関連する演算子クラスを演算子族にまとめることができます。既存の演算子族に新しい演算子クラスを追加するためには、CREATE OPERATOR CLASSでFAMILYオプションを指定してください。このオプションを指定しないと、新しい演算子クラスはそのクラスと同じ名前の演算子族内に置かれます（もしこの演算子族が存在しない場合は演算子族を作成します）。

詳細については[項35.14](#)を参照してください。

パラメータ

name

作成する演算子クラスの名前です。演算子クラス名は、スキーマ修飾することができます。

DEFAULT

DEFAULTを付けると、その演算子クラスが、そのデータ型のデフォルトのインデックス演算子クラスになります。特定のデータ型およびインデックスメソッドのデフォルトにできる演算子クラスは1つのみです。

data_type

この演算子クラスを使用する列のデータ型です。

index_method

この演算子クラスを使用するインデックスメソッドの名前です。

family_name

新規作成クラスの追加先となる既存の演算子族の名前です。 指定されない場合、演算子クラスと同じ名前の演算子族が使用されます（存在しない場合は演算子族を作成します）。

strategy_number

演算子クラスに関連する演算子のインデックスメソッドの戦略番号です。

operator_name

演算子クラスに関連する演算子の名前です（スキーマ修飾名でも可）。

op_type

`OPERATOR`句では、演算子の入力データ型、もしくは、左単項演算子か右単項演算子を表す `NONE` を指定します。 演算子クラスのデータ型と同じである通常の場合、入力データ型は省略可能です。

`FUNCTION`句では、関数の入力データ型（B-Tree比較関数およびハッシュ関数用）またはクラスのデータ型（B-treeソートサポート関数とGiST、SP-GiST、GIN演算子クラスのすべての関数用）と異なる場合、関数がサポートする予定の演算対象データ型です。 これらのデフォルトは常に正確です。 このため、しかしデータ型を跨がる比較をサポートする予定のB-treeソートサポート関数は除き、`FUNCTION`句で *op_type* を指定する必要はありません。

sort_family_name

順序付け演算子に関連したソート順序を記述する、既存の `btree` 演算子族の名前（スキーマ修飾可）です。

`FOR SEARCH` も `FOR ORDER BY` も指定されていない場合、`FOR SEARCH` がデフォルトです。

support_number

演算子クラスに関連する関数用のインデックスメソッドのサポートプロシージャの番号です。

function_name

演算子クラス用のインデックスメソッドのサポートプロシージャとなる関数の名前です（スキーマ修飾名でも可）。

argument_type

関数のパラメータのデータ型です。

storage_type

インデックスに実際に格納されるデータ型です。 通常、このデータ型は列のデータ型と同じです。 しかし、異なるデータ型を許可するインデックスメソッドも存在します（現時点ではGiSTとGIN）。 インデックスメソッドが異なるデータ型の使用を許可していなければ、`STORAGE` 句を指定してはいけません。

`OPERATOR`、`FUNCTION`、`STORAGE` は任意の順番で記述できます。

注釈

インデックス機構は、使用する前に関数に関するアクセス権限を検査しませんので、 関数や演算子を演算子クラスに含めることは、`PUBLIC`に実行権限を与えることと同じです。 通常、演算子クラスで有用な種類の関数ではこれは問題になりません。

演算子はSQL関数で定義してはなりません。 SQL関数は呼び出し元の問い合わせにインライン化されることが多いので、オプティマイザでその問い合わせがインデックスに一致するかどうかを認識できなくなってしまうからです。

PostgreSQL 8.4より前までは、`OPERATOR` 句に `RECHECK` オプションを含めることができました。 インデックス演算子に"損失がある"かどうかは実行時にその場で決定されるようになりましたので、これはサポートされなくなりました。 これにより、演算子に損失があるかもしれないしないかもしれないような場合を効率的に扱うことができるようになりました。

例

以下のコマンド例では、`_int4` データ型（`int4`の配列）のGiSTインデックス演算子クラスを定義しています。 この例の詳細については、[intarray](#) モジュールを参照してください。

```
CREATE OPERATOR CLASS gist__int_ops
  DEFAULT FOR TYPE _int4 USING gist AS
  OPERATOR      3      &&,
  OPERATOR      6      = (anyarray, anyarray),
  OPERATOR      7      @>,
  OPERATOR      8      <@,
```

```
OPERATOR          20      @@ (_int4, query_int),
FUNCTION          1      g_int_consistent(internal, _int4, int, oid, internal),
FUNCTION          2      g_int_union (internal, internal),
FUNCTION          3      g_int_compress (internal),
FUNCTION          4      g_int_decompress (internal),
FUNCTION          5      g_int_penalty (internal, internal, internal),
FUNCTION          6      g_int_picksplit (internal, internal),
FUNCTION          7      g_int_same (_int4, _int4, internal);
```

互換性

CREATE OPERATOR CLASSはPostgreSQLの拡張です。 標準SQLにはCREATE OPERATOR CLASS文はありません。

関連項目

[ALTER OPERATOR CLASS](#), [DROP OPERATOR CLASS](#), [CREATE OPERATOR FAMILY](#), [ALTER OPERATOR FAMILY](#)

CREATE OPERATOR FAMILY

名前

CREATE OPERATOR FAMILY -- 新しい演算子族を定義する

概要

```
CREATE OPERATOR FAMILY name USING index_method
```

説明

CREATE OPERATOR FAMILYは演算子族を新規に作成します。演算子族は、関連する演算子クラスと、おそらく数個の、これらの演算子クラスと互換性があるが個々のインデックスの機能にとっては重要ではない、追加演算子と関数の集合を定義します。（インデックスにとって重要な演算子と関数は、演算子族内で"自由"とするのではなく、対応する演算子クラスにまとめられなければなりません。通常、単一のデータ型に対する演算子は演算子クラスにまとめ、データ型を跨る演算子を両方のデータ型に対する演算子族内で自由とします。）

新しい演算子クラスの初期状態は空です。続いて、含むべき演算子クラスを追加するためにCREATE OPERATOR CLASSコマンドを発行してデータを投入しなければなりません。そして、省略することもできますが、"自由"な演算子と対応するサポート関数を追加するためにALTER OPERATOR FAMILYコマンドを発行しなければなりません。

スキーマ名が指定されると、演算子族は指定したスキーマ内に作成されます。さもなくば、現在のスキーマ内に作成されます。対象とするインデックスメソッドが異なる限り、同一スキーマ内に同じ名前の2つの演算子族を持たせることができます。

演算子族を定義したユーザがその所有者となります。現時点では、作成者はスーパーユーザでなければなりません。演算子族を間違って定義すると、混乱し、サーバがクラッシュすることがありますので、この制限が存在します。

詳細は[項35.14](#)を参照してください。

パラメータ

name

作成する演算子族の名前です。この名前はスキーマ修飾可能です。

index_method

演算子族が対象とするインデックスメソッドの名前です。

互換性

CREATE OPERATOR FAMILYはPostgreSQLの拡張です。標準SQLにはCREATE OPERATOR FAMILY文はありません。

関連項目

[ALTER OPERATOR FAMILY](#), [DROP OPERATOR FAMILY](#), [CREATE OPERATOR CLASS](#), [ALTER OPERATOR CLASS](#), [DROP OPERATOR CLASS](#)

CREATE ROLE

名前

CREATE ROLE -- 新しいデータベースロールを定義する

概要

```
CREATE ROLE name [ [ WITH ] option [ ... ] ]
```

ここで`option`は以下の通りです。

```
SUPERUSER | NOSUPERUSER
| CREATEDB | NOCREATEDB
| CREATEROLE | NOCREATEROLE
| CREATEUSER | NOCREATEUSER
| INHERIT | NOINHERIT
| LOGIN | NOLOGIN
| REPLICATION | NOREPLICATION
| CONNECTION LIMIT connlimit
| [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
| VALID UNTIL 'timestamp'
| IN ROLE role_name [, ...]
| IN GROUP role_name [, ...]
| ROLE role_name [, ...]
| ADMIN role_name [, ...]
| USER role_name [, ...]
| SYSID uid
```

説明

`CREATE ROLE`は、PostgreSQLデータベースクラスタに新しいロールを加えます。 ロールとは、自身でデータベースオブジェクトを所有することができ、データベース権限を持つことができる実体のことです。 ロールは、使用状況に応じて"ユーザ"、"グループ"、もしくは、その両方であるとみなすことができます。 ユーザの管理と認証に関する情報については、[第20章](#)と[第19章](#)を参照してください。 このコマンドを使用するには、`CREATEROLE`権限を持つか、データベースのスーパーユーザでなければなりません。

ロールはデータベースクラスタのレベルで定義されるため、クラスタ内のすべてのデータベースで有効となることに注意してください。

パラメータ

name

新しいロールの名前です。

```
SUPERUSER
NOSUPERUSER
```

これらの句によって、新しいロールが"スーパーユーザ"となるかが決まります。 "スーパーユーザ"はデータベース内のアクセス制限をすべて変更することができます。 スーパーユーザという状態は危険ですので、本当に必要な場合にのみ使用しなければなりません。 新しくスーパーユーザを作成するには、スーパーユーザでなければなりません。 指定されなかった場合のデフォルトは`NOSUPERUSER`です。

```
CREATEDB
NOCREATEDB
```

これらの句はロールのデータベースの作成に関する権限を定義します。 `CREATEDB`が指定された場合、新しくデータベースを作成することができるように、ロールが作成されます。 `NOCREATEDB`を使用した場合、そのロールにはデータベースを作成する権限が与えられません。 指定されなかった場合のデフォルトは`NOCREATEDB`です。

```
CREATEROLE
NOCREATEROLE
```

これらの句は、ロールが新しいロールを作成（つまり`CREATE ROLE`を実行）できるかどうかを決定します。 `CREATEROLE`権限を持つロールはロールを変更することも削除することもできます。 指定されなかった場合のデフォルト

はNOCREATEROLEです。

CREATEUSER
NOCREATEUSER

これらの句は廃止予定ですが、SUPERUSERとNOSUPERUSERの別名として、まだ受け付けられます。単純にCREATEROLEと同じであると考えられるかもしれませんが、実際には異なることに注意してください。

INHERIT
NOINHERIT

これらの句は、ロールがそのロールが属するロールの権限を"継承"するかどうかを決定します。INHERIT属性を持つロールは自動的に、直接またはメンバとして間接的に割り当てられたすべてのデータベース権限を使用します。INHERITがないと、他のロール内のメンバ資格により得られる能力はそのロールへのSET ROLEだけです。他のロールの持つ権限は、SET ROLEを行った後にのみ利用可能です。指定されなかった場合のデフォルトはINHERITです。

LOGIN
NOLOGIN

これらの句により、ロールがログイン可能かどうか、つまり、そのロールをクライアント接続時にセッションを認証するための名前として使用することができるかどうかが決まります。LOGIN属性を持つロールはユーザとみなすことができます。この属性を持たないロールは、データベース権限を管理する際に有用ですが、普通の意味ではユーザとはいえません。指定されなかった場合のデフォルトはNOLOGINです。ただし、CREATE ROLEが別名のCREATE USERで呼び出された場合は例外です。

REPLICATION
NOREPLICATION

これらの句は、ロールがストリーミングレプリケーションを初期化でき、システムをバックアップモードとの遷移が可能かどうかを決定します。REPLICATION属性を持つロールは、非常に高度な権限を持つものです。このため実際にレプリケーションで使用するロールでのみ使用しなければなりません。指定されない場合のデフォルトはNOREPLICATIONです。

CONNECTION LIMIT *conlimit*

ロールがログイン可能である場合、これは、ロールが確立できる最大同時接続数を指定します。-1（デフォルト）は無制限を意味します。

PASSWORD *password*

ロールのパスワードを設定します。（パスワードはLOGIN属性を持つロールでのみ意味がありますが、この属性を持たないロールにも定義することができます。）パスワード認証を行う予定がなければ、このオプションを省略することができます。パスワードの指定がなければ、パスワードがNULLに設定され、そのアカウントでのパスワード認証は常に失敗します。オプションとして、NULLというパスワードを明示的にPASSWORD NULLと記述することができます。

ENCRYPTED
UNENCRYPTED

これらのキーワードを使用すると、パスワードを暗号化してシステムカタログに格納するかどうかを制御できます（指定されていない場合のデフォルトの動作は、[password encryption](#)設定パラメータによって決まります）。指定された文字列が既にMD5暗号化書式である場合は、ENCRYPTED、UNENCRYPTEDのどちらが指定されているかに関係なく、そのまま保存されます（システムでは、暗号化されたパスワードを復号できないからです）。これにより、ダンプ/リストア時に暗号化パスワードを再読み込みすることができます。

古めのクライアントでは、暗号化して保存されたパスワードを使用するために必要なMD5認証機構をサポートしていない場合があることに注意してください。

VALID UNTIL '*timestamp*'

VALID UNTIL句は、ロールのパスワードが無効になるまでの日時を設定します。この句が省略された場合、パスワードは永遠に有効になります。

IN ROLE *role_name*

IN ROLE句には、新しく作成するロールを新規にメンバとして追加する既存の1つ以上のロールを列挙します。（新しく作成するロールを管理者として追加するオプションがないことに注意してください。このためには別途GRANTコマンドを使用してください。）

IN GROUP *role_name*

IN GROUPはIN ROLEの別名で、廃止予定です。

ROLE *role_name*

`ROLE`には、新しく作成するロールのメンバとして自動的に追加する既存の1つ以上のロールを列挙します。（これは新しく作成したロールを"グループ"とします。）

ADMIN `role_name`

ADMIN句は`ROLE`と似ていますが、新しく作成されるロールに指定されたロールがWITH ADMIN OPTIONとして追加される点が異なります。つまり、新しく作成されるロールのメンバ資格を他者に与えることができる権利を、指定されたロールに与えます。

USER `role_name`

USER句は`ROLE`句の別名で廃止予定です。

SYSID `uid`

SYSID句は無視されます。後方互換性を維持するために受け付けられます。

注釈

ロールの属性を変更するには[ALTER ROLE](#)を、ユーザを削除するには[DROP ROLE](#)を使用してください。CREATE ROLEで指定したすべての属性は、後でALTER ROLEコマンドで変更可能です。

グループとして使用しているロールのメンバの追加、および、削除についての推奨方法は、[GRANT](#)と[REVOKE](#)を使用することで。

VALID UNTIL句は、パスワードのみの有効期限を定義します。ユーザアカウントの有効期限ではありません。特に、パスワードを元にしない認証方式でログインを行う場合には、この有効期限は強制されません。

INHERIT属性は、許可可能な権限（つまり、データベースオブジェクトに対するアクセス権限とロールのメンバ資格）の継承を管理します。これは、CREATE ROLEやALTER ROLEで設定される特別なロール属性には適用されません。INHERITが設定されていたとしても、例えば、CREATEDB権限を持つロールのメンバであっても、データベース作成権限は即座に付与されません。データベースを作成する前に[SET ROLE](#)を使用してそのロールにならなければなりません。

後方互換性を維持するため、INHERIT属性はデフォルトです。以前のリリースのPostgreSQLでは、ユーザは常にメンバとなっているすべてのグループの権限でアクセスできました。しかし、NOINHERITの方が標準SQLの規定の意味により合ったものを提供します。

CREATEROLE権限には注意が必要です。CREATEROLE ロールという権限には継承という概念がありません。あるロールが特定の権限を持っていなくても、別のロールを作成できることを意味します。つまり、簡単に自身の持つ権限と異なる権限（スーパーユーザ権限を持つロールは除きます）を持つ別のロールを作成できてしまいます。たとえば、CREATEROLE権限を持ち、CREATEDB権限を持たない"user"というロールが、CREATEDB権限を持つロールを新規に作成することができます。したがって、CREATEROLE権限を持つロールは、ほとんどスーパーユーザ権限を持つロールと同じであるものと考えてください。

PostgreSQLには、CREATE ROLEと同じ機能を持つ（実際にこのコマンドを呼び出しています）[createuser](#)プログラムがあり、コマンドシェルから実行することができます。

CONNECTION LIMITオプションが加える制限は厳密ではありません。もしそのロールに1つだけ接続"スロット"が残っていた時に、ほぼ同時に2つのセッションが新しく始まった場合、両方とも失敗する可能性があります。また、この制限はスーパーユーザには適用されません。

このコマンドで暗号化しないパスワードを指定するときには注意しなければなりません。パスワードはサーバに平文で送信されます。クライアントのコマンド履歴やサーバのログにこれが残ってしまうかもしれません。しかし、[createuser](#)コマンドはパスワードを暗号化して送信します。また、[psql](#)には`\password`コマンドがあります。これを使用して後でパスワードを安全に変更することができます。

例

ログイン可能なロールを作成します。ただし、パスワードはありません。

```
CREATE ROLE jonathan LOGIN;
```

パスワード付きのロールを作成します。

```
CREATE USER davide WITH PASSWORD 'jw8s0F4';
```

CREATE USERはLOGINを意味する点を除き、CREATE ROLEと同一です。

2004年まで有効なパスワードを持つロールを作成します。 2005年に1秒でも入った時点でパスワードは無効になります。

```
CREATE ROLE miriam WITH LOGIN PASSWORD 'jw8s0F4' VALID UNTIL '2005-01-01';
```

データベースを作成でき、かつ、ロールを管理できるロールを作成します。

```
CREATE ROLE admin WITH CREATEDB CREATEROLE;
```

互換性

`CREATE ROLE`文は標準SQLで規定されています。 しかしSQLでは以下の構文のみを要求しています。

```
CREATE ROLE name [ WITH ADMIN role_name ]
```

複数の初期管理者やそのほかの`CREATE ROLE`のオプションはPostgreSQLの拡張です。

標準SQLでは、ユーザとロールという概念を定義し、それらを別の概念としてみなしています。 また、ユーザを定義するコマンドはすべて、各データベース実装で規定するものとしています。 PostgreSQLでは、ユーザとロールを単一の実体に統一することを選択しています。 したがって、ロールは標準よりも非常に多くの省略可能な属性を持っています。

ユーザは`NOINHERIT`属性を与えること、ロールは`INHERIT`属性を与えることで、標準SQLで規定された振舞いをほぼ完全に行うことができます。

関連項目

[SET ROLE](#), [ALTER ROLE](#), [DROP ROLE](#), [GRANT](#), [REVOKE](#), [createuser](#)

[前のページ](#)

CREATE OPERATOR FAMILY

[ホーム](#)

[上に戻る](#)

[次のページ](#)

CREATE RULE

CREATE RULE

名前

CREATE RULE -- 新しい書き換えルールを定義する

概要

```
CREATE [ OR REPLACE ] RULE name AS ON event
TO table_name [ WHERE condition ]
DO [ ALSO | INSTEAD ] { NOTHING | command | ( command ; command ... ) }
```

説明

CREATE RULEにより、指定したテーブルまたはビューに適用するルールを新しく定義できます。CREATE OR REPLACE RULEを使用すると、新しいルールの作成、または、同じテーブルの同じ名前の既存ルールの置き換えのいずれかを実行します。

PostgreSQLのルールシステムを使用すると、データベーステーブルに対する挿入、更新、削除時に本来の操作の代替として実行するアクションを定義できます。簡単に言えば、指定されたテーブルに対して指定されたコマンドが実行された時、ルールによって追加のコマンドが実行されるということです。その他にも、INSTEADルールによって指定されたコマンドを他のコマンドに置き換えたり、まったくコマンドを実行しないようにすることも可能です。また、ルールはテーブルビューを実装するためにも使われます。重要なのは、ルールとは実際にコマンドを変換する仕組み、言い換えれば、コマンドのマクロであることです。ルールによる変換はコマンドの実行前に発生します。各物理行を個別に操作したい場合、ルールではなくトリガを使用する方が良いでしょう。ルールシステムについての詳細は、[第37章](#)に記載されています。

現時点では、ON SELECTルールは、条件を持たないINSTEADルールでなければなりません。また、ON SELECTルールが持つアクションは、単独のSELECTコマンドのみで構成される必要があります。したがって、ON SELECTルールを使えば、効率的にテーブルをビューにすることができます。このビューでは、元のテーブルの内容ではなく、ルールに含まれるSELECTコマンドが返す行が、ビューの内容として提示されます。ただし、テーブル自体を作成し、それにON SELECTルールを定義するよりは、CREATE VIEWコマンドを使用する方をお勧めします。

ON INSERT、ON UPDATE、ON DELETEルールを必要に応じて定義し、ビューに対する更新操作を他のテーブルに対する適切な更新操作に置換することで、更新可能なビューという実在しないオブジェクトを作成することができます。INSERT RETURNINGなどをサポートしたければ、これらのルールに適切なRETURNING句を確実につけてください。この他の方法としてINSTEAD OFトリガを使用して更新可能なビューを実装することができます（[CREATE TRIGGER](#)参照）。

ビューの更新に条件付きルールを使用しようとする場合、落とし穴があります。そのビューに許可するそれぞれの操作に、条件を持たないINSTEADルールを用意する必要があります。ルールが条件付きであったり、INSTEADでない場合、システムは更新操作を拒否します。その場合、システムが、場合によっては、ビューのダミーテーブルに対する操作になる可能性があるともみなすからです。使用する全てのケースについて条件付きルールを作成する場合は、条件を持たないDO INSTEAD NOTHINGルールを追加し、ダミーテーブルに対する更新は呼び出されないことをシステムに明示します。さらに、条件付きルールには、INSTEADと指定しないようにします。これらの条件が満たされた場合、デフォルトのINSTEAD NOTHINGアクションにルールに含まれるアクションが追加されます。（しかし現在、この方法ではRETURNING問い合わせのサポートがうまく動作しません。）

パラメータ

name

作成するルールの名前です。この名前は、同じテーブルの他のルールとは異なる名前にする必要があります。同一テーブルにイベントの種類が同じルールが複数あった場合、ルール名の順番（アルファベット順）に適用されます。

event

イベントとは、SELECT、INSERT、UPDATE、DELETEのいずれかです。

table_name

ルールを適用するテーブルまたはビューの名前です（スキーマ修飾名も可）。

condition

任意のSQL条件式です（`boolean`型を返します）。条件式では、`NEW`および`OLD`以外のテーブルは参照できません。また、集約関数を含めることもできません。

`INSTEAD`

`INSTEAD`は、元のコマンドの代わりにこのコマンドが実行されることを示します。

`ALSO`

`ALSO`は元のコマンドに加えてこのコマンドが実行されることを示します。

`ALSO`も`INSTEAD`も指定されなかった場合、`ALSO`がデフォルトです。

command

ルールのアクションを構成する単一または複数のコマンドです。有効なコマンドは、`SELECT`、`INSERT`、`UPDATE`、`DELETE`、`NOTIFY`です。

*condition*と*command*の内部では、対象とするテーブルの値を参照するために、特別なテーブル名`NEW`と`OLD`を使用できます。`NEW`は、`ON INSERT`と`ON UPDATE`ルールで有効です。挿入または更新される新しい行を参照します。`OLD`は、`ON UPDATE`と`ON DELETE`ルールで有効です。更新または削除される既存の行を参照します。

注釈

テーブルにルールを定義する、または、そのルールを変更するためには所有者でなければなりません。

ビュー上の`INSERT`、`UPDATE`、`DELETE`ルールでは、`RETURNING`句を追加して、ビューの列を返すことができます。ルールが`INSERT` `RETURNING`、`UPDATE` `RETURNING`、`DELETE` `RETURNING`コマンドによって呼び出された場合、この句は出力を計算することに使用されます。`RETURNING`なしでルールが呼び出された場合、ルールの`RETURNING`句は無視されます。現在の実装では、無条件の`INSTEAD`ルールのみが`RETURNING`を含むことができます。さらに、同一イベント用のすべてのルールの中で多くても1つの`RETURNING`句を持つことができます。（これにより確実に、結果を計算するために使用される`RETURNING`句の候補が1つのみになります。）ビュー上の`RETURNING`問い合わせは、利用可能なルールすべてで`RETURNING`が存在しない場合に拒絶されます。

ルールの循環は絶対に避けるよう注意してください。例えば、下記の2つのルール定義それぞれがPostgreSQLに受け入れられた場合、ルールが再帰的に展開されるため、`SELECT`コマンドが、PostgreSQLにエラーを表示させます。

```
CREATE RULE "_RETURN" AS
  ON SELECT TO t1
  DO INSTEAD
    SELECT * FROM t2;

CREATE RULE "_RETURN" AS
  ON SELECT TO t2
  DO INSTEAD
    SELECT * FROM t1;

SELECT * FROM t1;
```

現在、ルールのアクションに`NOTIFY`問い合わせが含まれる場合、`NOTIFY`は無条件に実行されます。つまり、ルールを適用すべき行が存在しなかったとしても、`NOTIFY`が発行されます。例えば、

```
CREATE RULE notify_me AS ON UPDATE TO mytable DO ALSO NOTIFY mytable;

UPDATE mytable SET name = 'foo' WHERE id = 42;
```

では、`id = 42`という条件に一致する行があってもなくても、`UPDATE`の際、1つの`NOTIFY`イベントが送信されます。これは実装上の制限であり、将来のリリースでは修正されるかもしれません。

互換性

問い合わせ書き換えシステム全体同様、`CREATE RULE`はPostgreSQLの拡張です。

[前のページ](#)
CREATE ROLE

[ホーム](#)
[上に戻る](#)

[次のページ](#)
CREATE SCHEMA

CREATE SCHEMA

名前

CREATE SCHEMA -- 新しいスキーマを定義する

概要

```
CREATE SCHEMA schema_name [ AUTHORIZATION user_name ] [ schema_element [ ... ] ]
CREATE SCHEMA AUTHORIZATION user_name [ schema_element [ ... ] ]
```

説明

CREATE SCHEMA を実行すると、現在のデータベースに新しいスキーマが登録されます。 スキーマ名は、現在のデータベースにある既存のスキーマとは異なる名前にする必要があります。

スキーマは、本質的には名前空間です。 スキーマには、名前付きオブジェクト（テーブル、データ型、関数、および演算子）が含まれます。 これらのオブジェクトの名前は、他のスキーマに存在する他のオブジェクトの名前と重複しても構いません。 名前付きオブジェクトには、スキーマ名を接頭辞としてオブジェクト名を"修飾"するか、必要なスキーマを含んだ検索パスを設定することによってアクセスできます。 オブジェクト名を修飾しないで指定した CREATE コマンドは、そのオブジェクトの現在のスキーマ（current_schema 関数で決定される検索パスの先頭部分）で作成されます。

CREATE SCHEMA には、オプションとして、新しいスキーマ内でオブジェクトを作成するためのサブコマンドを付加することができます。 サブコマンドは、本質的にはスキーマ作成後に発行される別コマンドと同じように扱われます。 ただし、AUTHORIZATION 句を使用した場合、作成された全てのオブジェクトの所有者が指定したユーザになるという点で異なります。

パラメータ

schema_name

作成するスキーマの名前です。 省略された場合、*user_name* がスキーマ名として使用されます。 スキーマ名を pg_ から始めることはできません。 このような名前はシステムスキーマ用に予約されているためです。

user_name

新しいスキーマを所有するユーザのロール名です。 省略された場合、デフォルトでは、コマンドを実行したユーザになります。 他のロールを所有者とするスキーマを作成するためには、そのロールの直接的または間接的なメンバであるか、スーパーユーザでなければなりません。

schema_element

そのスキーマ内で作成されるオブジェクトを定義する SQL 文です。 現在、CREATE SCHEMA 内では、CREATE TABLE、CREATE VIEW、CREATE INDEX、CREATE SEQUENCE、CREATE TRIGGER、および GRANT のみが句として使用可能です。 他の種類のオブジェクトは、スキーマ作成後に個別のコマンドを使えば作成できます。

注釈

スキーマを作成するには、実行するユーザが現在のデータベースにおける CREATE 権限を持っている必要があります。（もちろん、スーパーユーザにはこの制限はありません）。

例

スキーマを作成します。

```
CREATE SCHEMA myschema;
```

joe ユーザ用にスキーマを作成します。 このスキーマの名前は joe になります。

```
CREATE SCHEMA AUTHORIZATION joe;
```

スキーマを作成し、その中にテーブルとビューを作成します。

```
CREATE SCHEMA hollywood
CREATE TABLE films (title text, release date, awards text[])
CREATE VIEW winners AS
SELECT title, release FROM films WHERE awards IS NOT NULL;
```

個々のサブコマンドがセミコロンで終わっていないことに注意してください。

以下は、上述のコマンドと等価であり、同じ結果をもたらします。

```
CREATE SCHEMA hollywood;
CREATE TABLE hollywood.films (title text, release date, awards text[]);
CREATE VIEW hollywood.winners AS
SELECT title, release FROM hollywood.films WHERE awards IS NOT NULL;
```

互換性

標準SQLでは、`CREATE SCHEMA`で`DEFAULT CHARACTER SET`句を使用できます。同様に、現在PostgreSQLで使用できるよりも多くのサブコマンドを使用できます。

標準SQLでは、`CREATE SCHEMA`のサブコマンドを任意の順序で記述できます。現在のPostgreSQLの実装では、サブコマンドにおいて下方参照ができない場合があります。そのため、下方参照を避ける目的で、サブコマンドの順序を並べ替える必要が生じる可能性もあります。

標準SQLでは、スキーマの所有者は、常にそのスキーマ内の全てのオブジェクトを所有します。PostgreSQLでは、スキーマ所有者以外のユーザが所有するオブジェクトを、スキーマに含めることができます。このような状態は、スキーマ所有者が、そのスキーマでの`CREATE`権限を他のユーザに与えた場合やスーパーユーザがその中にオブジェクトを作成した場合にのみ発生します。

関連項目

[ALTER SCHEMA](#), [DROP SCHEMA](#)

[前のページ](#)

CREATE RULE

[ホーム](#)
[上に戻る](#)

[次のページ](#)

CREATE SEQUENCE

CREATE SEQUENCE

名前

CREATE SEQUENCE -- 新しいシーケンスジェネレータを定義する

概要

```
CREATE [ TEMPORARY | TEMP ] SEQUENCE name [ INCREMENT [ BY ] increment ]
[ MINVALUE minvalue | NO MINVALUE ] [ MAXVALUE maxvalue | NO MAXVALUE ]
[ START [ WITH ] start ] [ CACHE cache ] [ [ NO ] CYCLE ]
[ OWNED BY { table_name.column_name | NONE } ]
```

説明

CREATE SEQUENCEは、新しいシーケンス番号ジェネレータを作成します。 具体的には、新しく*name*という名前を持つ、1行だけの特殊なテーブルの作成と初期化を行います。 シーケンスジェネレータは、このコマンドを実行したユーザによって所有されます。

スキーマ名が与えられている場合、そのシーケンスは指定されたスキーマで作成されます。 スキーマ名がなければ、シーケンスは現在のスキーマで作成されます。 また、一時シーケンスは特別なスキーマに存在するため、一時シーケンスの作成時にスキーマ名を与えることはできません。 シーケンス名は、同じスキーマ内の他のシーケンス、テーブル、インデックス、ビュー、外部テーブルとは異なる名前にする必要があります。

シーケンスを作成した後、nextval、currval、setval関数を使用してシーケンスを操作します。 この関数については[項9.16](#)を参照してください。

シーケンスを直接更新することはできませんが、以下のような問い合わせは可能です。

```
SELECT * FROM name;
```

これを使用すると、シーケンスのパラメータと現在の状態を確認することができます。 中でも、シーケンスのlast_valueフィールドは全てのセッションで割り当てられた最後の値を示します（もちろんこの値は、他のセッションのnextvalの実行により、表示された時点で既に最新ではない可能性があります）。

パラメータ

TEMPORARYもしくはTEMP

このパラメータが指定された場合、作成するシーケンスオブジェクトがそのセッション専用となり、セッション終了時に自動的に削除されます。 一時シーケンスが存在する場合、同じ名前を持つ既存の永続シーケンスは、スキーマ修飾された名前で参照されない限り、（そのセッションでは）不可視です。

name

作成するシーケンスの名前です（スキーマ修飾名も可）。

increment

INCREMENT BY *increment*句は、現在のシーケンスの値から新しいシーケンス値を作成する際の値の増加量を設定します。 この句は省略可能です。 正の値が指定された時は昇順のシーケンス、負の値が指定された時は降順のシーケンスを作成します。 指定がない場合のデフォルト値は1です。

minvalue

NO MINVALUE

MINVALUE *minvalue*句は、シーケンスとして作成する最小値を指定します。 この句は省略可能です。 この句の指定されなかった場合、もしくは、NO MINVALUEが指定された場合、デフォルトが使用されます。 シーケンスのデフォルトの最小値は、昇順の時は1、降順の時は $-(2^{63}-1)$ です。

maxvalue

NO MAXVALUE

`MAXVALUE maxvalue`句は、シーケンスの最大値を決定します。この句は省略可能です。この句が指定されなかった場合、もしくは`NO MAXVALUE`が指定された場合、デフォルトが使用されます。シーケンスのデフォルトの最小値は、昇順の時は $2^{63}-1$ 、降順の時は -1 です。

`start`

`START WITH start`句を使用すると、任意の数からシーケンス番号を開始することができます。この句は省略可能です。デフォルトでは、シーケンス番号が始まる値は、昇順の場合`minvalue`、降順の場合`maxvalue`になります。

`cache`

`CACHE cache`オプションは、あらかじめ番号を割り当て、メモリに格納しておくシーケンス番号の量を指定します。これによりアクセスを高速にすることができます。最小値は1です（一度に生成する値が1つだけなので、キャッシュがない状態になります）。これがデフォルトになっています。

`CYCLE`
`NO CYCLE`

`CYCLE`オプションを使用すると、シーケンスが限界値（昇順の場合は`maxvalue`、降順の場合は`minvalue`）に達した時、そのシーケンスを周回させることができます。限界値まで達した時、次に生成される番号は、昇順の場合は`minvalue`、降順の場合は`maxvalue`になります。

`NO CYCLE`が指定された場合、シーケンスの限界値に達した後の`nextval`呼び出しは全てエラーになります。`CYCLE`も`NO CYCLE`も指定されていない場合は、`NO CYCLE`がデフォルトとなります。

`OWNED BY table_name.column_name`
`OWNED BY NONE`

`OWNED BY`オプションにより、シーケンスは指定されたテーブル列に関連付けられ、その列（やテーブル全体）が削除されると、自動的にシーケンスも同様に削除されるようになります。指定するテーブルは、シーケンスと同一所有者でなければならず、また、同一のスキーマ内に存在しなければなりません。デフォルトは`OWNED BY NONE`であり、こうした関連付けがないことを示します。

注釈

シーケンスを削除するには`DROP SEQUENCE`を使用してください。

シーケンスは`bigint`演算に基づいています。そのため、8バイト整数の範囲（ -9223372036854775808 から 9223372036854775807 まで）を越えることはできません。比較的古いプラットフォームでは、8バイト整数をサポートするコンパイラがないことがあります。その場合、シーケンスには通常の`integer`演算が使用されます（この場合、範囲は -2147483648 から $+2147483647$ まで）。

シーケンスオブジェクトの`cache`として1より大きな値を設定した場合、そのシーケンスを複数のセッションで同時に使用すると、予想外の結果になる可能性があります。各セッションは、シーケンスオブジェクトへの1回のアクセスの間に、連続するシーケンス値を取得し、キャッシュします。そして、キャッシュした数に応じて、シーケンスオブジェクトの`last_value`を増加させます。この場合、そのセッションは、その後の`cache-1`回に対しては、あらかじめ取得済みのシーケンス値を返し、シーケンスオブジェクトを変更しません。セッションに割り当てられたが使用されなかったシーケンス番号は、セッションの終了時に全て失われるため、結果としてシーケンスに"穴"ができます。

さらに、複数のセッションには個別のシーケンス値が割り当てられることが保証されていますが、全てのセッションが尊重されると、シーケンス値が順番通りにならないことがあります。例えば、`cache`が10の場合を考えます。セッションAでは1から10までを確保し、`nextval=1`を返します。セッションBでは、セッションAが`nextval=2`を返す前に、11から20を確保し、`nextval=11`を返します。したがって、`cache`を1に設定した場合は`nextval`が連続した値であると考えても問題ありませんが、`cache`を1より大きな値に設定した場合は、`nextval`の値が全て異なることのみが保証され、連続した値であることは保証されません。また、`last_value`は、値が`nextval`によって返されたかどうかに関係なく、いずれかのセッションによって確保された最後の値となります。

この他、このようなシーケンスに対して`setval`が実行されても、他のセッションは、それぞれがキャッシュした取得済みの値を全て使い果たすまで、それがわからないことも考慮すべき問題です。

例

101から始まる`serial`という名前の昇順シーケンスを作成します。

```
CREATE SEQUENCE serial START 101;
```

このシーケンスから次の番号を選択します。

```
SELECT nextval('serial');
```

```
nextval
-----
101
```

このシーケンスから次の番号を選択します。

```
SELECT nextval('serial');
```

```
nextval
-----
102
```

このシーケンスをINSERT コマンドで使します。

```
INSERT INTO distributors VALUES (nextval('serial'), 'nothing');
```

COPY FROMの後でシーケンス値を更新します。

```
BEGIN;
COPY distributors FROM 'input_file';
SELECT setval('serial', max(id)) FROM distributors;
END;
```

互換性

以下の例外を除き、CREATE SEQUENCEは標準SQLに従います。

- 標準におけるAS <data type>式はサポートされていません。
- 次の値を取り出すには、標準のNEXT VALUE FOR式ではなくnextval()関数を使します。
- OWNED BY句はPostgreSQLの拡張です。

関連項目

[ALTER SEQUENCE](#), [DROP SEQUENCE](#)

[前のページ](#)

CREATE SCHEMA

[ホーム](#)
[上に戻る](#)

[次のページ](#)

CREATE SERVER

CREATE SERVER

名前

CREATE SERVER -- 新しい外部サーバを定義する

概要

```
CREATE SERVER server_name [ TYPE 'server_type' ] [ VERSION 'server_version' ]  
FOREIGN DATA WRAPPER fdw_name  
[ OPTIONS ( option 'value' [, ... ] ) ]
```

説明

CREATE SERVER は新しい外部サーバを定義します。サーバを定義したユーザがその所有者となります。

外部サーバは通常、外部データラップが外部データリソースにアクセスするために使用する接続情報をカプセル化します。さらに、ユーザマップによりユーザ指定の接続情報が指定される可能性があります。

サーバ名はデータベース内で一意でなければなりません。

サーバを作成するには、使用する外部データラップ上に `USAGE` 権限が必要です。

パラメータ

server_name

作成する外部サーバの名前です。

server_type

省略可能なサーバの種類です。

server_version

省略可能なサーババージョンです。

fdw_name

このサーバを管理する外部データラップの名前です。

OPTIONS (*option* '*value*' [, ...])

この句はサーバのオプションを指定します。オプションは通常、サーバの接続の詳細を定義しますが、実際の名前とその値はサーバの外部データラップに依存します。

注釈

`dblink` モジュール ([dblink](#) 参照) を使用している場合、接続パラメータを表すために、外部サーバ名を `dblink_connect` 関数の引数として使用することができます。より多くの例がありますので、こちらも参照してください。この方法で利用できるようにするためには外部サーバ上に `USAGE` 権限が必要です。

例

組み込みの外部データラップ `default` を使用するサーバ `foo` を作成します。

```
CREATE SERVER foo FOREIGN DATA WRAPPER "default";
```

外部データラップ `pgsql` を使用するサーバ `myserver` を作成します。

```
CREATE SERVER myserver FOREIGN DATA WRAPPER postgresql OPTIONS (host 'foo', dbname 'foodb', port '5432');
```

互換性

CREATE SERVERはISO/IEC 9075-9 (SQL/MED)に従います。

関連項目

[ALTER SERVER](#), [DROP SERVER](#), [CREATE FOREIGN DATA WRAPPER](#), [CREATE USER MAPPING](#)

[前のページ](#)
CREATE SEQUENCE

[ホーム](#)
[上に戻る](#)

[次のページ](#)
CREATE TABLE

CREATE TABLE

名前

CREATE TABLE -- 新しいテーブルを定義する

概要

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT EXISTS ] table_name ( [
    { column_name data_type [ COLLATE collation ] [ column_constraint [ ... ] ]
    | table_constraint
    | LIKE source_table [ like_option ... ] }
    [, ... ]
)
[ INHERITS ( parent_table [, ... ] ) ]
[ WITH ( storage_parameter [= value] [, ... ] ) | WITH OIDS | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE tablespace_name ]

CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT EXISTS ] table_name
    OF type_name [ (
        { column_name WITH OPTIONS [ column_constraint [ ... ] ]
        | table_constraint }
        [, ... ]
    ) ]
[ WITH ( storage_parameter [= value] [, ... ] ) | WITH OIDS | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE tablespace_name ]
```

*column_constraint*には、次の構文が入ります。

```
[ CONSTRAINT constraint_name ]
{ NOT NULL |
  NULL |
  CHECK ( expression ) [ NO INHERIT ] |
  DEFAULT default_expr |
  UNIQUE index_parameters |
  PRIMARY KEY index_parameters |
  REFERENCES reftable [ ( refcolumn ) ] [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ]
    [ ON DELETE action ] [ ON UPDATE action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

また、*table_constraint*には、次の構文が入ります。

```
[ CONSTRAINT constraint_name ]
{ CHECK ( expression ) [ NO INHERIT ] |
  UNIQUE ( column_name [, ... ] ) index_parameters |
  PRIMARY KEY ( column_name [, ... ] ) index_parameters |
  EXCLUDE [ USING index_method ] ( exclude_element WITH operator [, ... ] ) index_parameters [ WHERE (
predicate ) ] |
  FOREIGN KEY ( column_name [, ... ] ) REFERENCES reftable [ ( refcolumn [, ... ] ) ]
    [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ] [ ON DELETE action ] [ ON UPDATE action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

and *like_option* is:

```
{ INCLUDING | EXCLUDING } { DEFAULTS | CONSTRAINTS | INDEXES | STORAGE | COMMENTS | ALL }
```

UNIQUE、PRIMARY KEYおよびEXCLUDE制約内の*index_parameters*は以下の通りです。

```
[ WITH ( storage_parameter [= value] [, ... ] ) ]
[ USING INDEX TABLESPACE tablespace_name ]
EXCLUDE制約内のexclude_elementは以下の通りです。
{ column_name | ( expression ) } [ opclass ] [ ASC | DESC ] [ NULLS { FIRST | LAST } ]
```

説明

CREATE TABLEは、現在のデータベースに新しい空のテーブルを作成します。 作成したテーブルはこのコマンドを実行したユーザが所有します。

スキーマ名が付けられている場合（例えば、CREATE TABLE myschema.mytable ...）、テーブルは指定されたスキーマで作成されます。 スキーマ名がなければ、テーブルは現在のスキーマで作成されます。 また、一時テーブルは特別なスキーマに存在するため、一時テーブルの作成時にスキーマ名を与えることはできません。 テーブル名は、同じスキーマ内の他のテーブル、シー

ケンス、インデックス、ビュー、外部テーブルとは異なる名前にする必要があります。

さらに、`CREATE TABLE`は、作成するテーブルの1行に対応する複合型のデータ型を作成します。したがって、テーブルは、同じスキーマ内の既存のデータ型と同じ名前を持つことができません。

制約句には、挿入、更新操作を行うために、新しい行、または更新する行が満たさなければならない制約（検査項目）を指定します。制約句は省略可能です。制約は、テーブル内で様々な有効な値の集合を定義する際、役に立つSQLオブジェクトです。

制約の定義にはテーブル制約と列制約という2種類があります。列制約は列定義の一部として定義されます。テーブル制約定義は、特定の列とは結びつけられておらず、複数の列を含有することができます。また、全ての列制約はテーブル制約として記述することができます。列制約は、1つの列にのみ影響する制約のための、簡便な記述方法に過ぎません。

テーブルを作成するためには、すべての列の型または`OF`句中の型に対する`USAGE`権限を持たなければなりません。

パラメータ

`TEMPORARY` or `TEMP`

このパラメータが指定された場合、テーブルは一時テーブルとして作成されます。一時テーブルは、そのセッションの終わり、場合によっては、現在のトランザクションの終わり（後述の`ON COMMIT`を参照）に自動的に削除されます。一時テーブルが存在する場合、同じ名前を持つ既存の永続テーブルは、スキーマ修飾名で参照されていない限り、現在のセッションでは非可視になります。一時テーブルで作られるインデックスも、全て自動的に一時的なものとなります。

[autovacuum daemon](#)は一時テーブルにアクセスできません。したがって一時テーブルのバキュームや解析を行うことはできません。このためセッションのSQLコマンドを用いて適切なバキュームと解析を実行しなければなりません。例えば、一時テーブルが複雑な問い合わせで使用される場合、一時テーブルにデータを投入した後にそれに対し`ANALYZE`を実行することを勧めます。

`GLOBAL`または`LOCAL`を`TEMPORARY`や`TEMP`の前に記述することができます（省略可能）。PostgreSQLでは、現在違いがなく、廃止予定です。[互換性](#)を参照してください。

`UNLOGGED`

指定された場合、テーブルはログを取らないテーブルとして作成されます。ログを取らないテーブルに書き出されたデータは先行書き込みログ（[第29章](#)参照）には書き出されません。このため通常のテーブルより相当高速になります。しかしこれらはクラッシュ時に安全ではありません。クラッシュまたは異常停止の後、ログを取らないテーブルは自動的に切り詰められます。またログを取らないテーブルの内容はスタンバイサーバに複製されません。ログを取らないテーブル上に作成されたインデックスはすべて同様に、ログを取らないようになります。しかし現在ログをとらない[GiST indexes](#)はサポートされませんので、ログを取らないテーブル上に作成することはできません。

`IF NOT EXISTS`

同じ名前のリレーションがすでに存在していてもエラーとしません。この場合注意が発せられます。既存のリレーションが作成しようとしたものと何かしら似たものであることは保証されません。

`table_name`

作成するテーブルの名前です（スキーマ修飾名でも可）。

`OF type_name`

指定(スキーマ修飾可能)した複合型から構造を取り出した型付きテーブルを作成します。型付きテーブルはその型に束縛されます。例えば、型が(`DROP TYPE ... CASCADE`付きで)削除されるとそのテーブルは削除されます。

型付きテーブルが作成されると、その列のデータ型は背後の複合型により決定され、`CREATE TABLE`コマンドでは指定されません。しかし`CREATE TABLE`コマンドではテーブルにデフォルトと制約を追加できます。また、格納パラメータの指定も可能です。

`column_name`

新しいテーブルで作成される列の名前です。

`data_type`

列のデータ型です。これには、配列指定子を含めることができます。PostgreSQLでサポートされるデータ型の情報に関する詳細は[第8章](#)を参照してください。

`COLLATE collation`

`COLLATE`句は列（照合順の設定が可能なデータ型でなければなりません）に照合順を割り当てます。指定がなければ、列

のデータ型のデフォルトの照合順が使用されます。

```
INHERITS ( parent_table [, ... ] )
```

`INHERITS`句でテーブルの一覧を指定すると、新しいテーブルは指定されたテーブルの全ての列を自動的に継承します。この句は省略可能です。

`INHERITS`を使用すると、新しい子テーブルとその親テーブル（複数可）との間に永続的な関連が作成されます。通常、親へのスキーマ修飾子は同様に子にも伝播します。また、デフォルトでは、親テーブルの走査結果には子テーブルのデータが含まれます。

複数の親テーブルに同一名の列が存在する場合、それらのデータ型が一致していなければ、エラーとして報告されます。競合がなければ、これらの重複した列は新しいテーブルで1つの列の形に融合されます。新しいテーブルの列名の一覧に継承する列の名前が含まれる場合も、そのデータ型は継承する列のデータ型と一致していなければなりません。さらに、その列定義は1つに融合されます。新しいテーブルで明示的に列のデフォルト値を指定した場合、継承した列宣言における全てのデフォルト値は上書きされます。デフォルト値を指定しなかった場合、親側でデフォルト値が指定されている時は、それらのデフォルト値が全て同じ値でなければなりません。値が違う場合はエラーになります。

`CHECK`制約は、基本的には列と同様の方法でマージされます。複数の親テーブル、新しいテーブル、またはその両方の定義に同じ名前の`CHECK`制約が存在した場合、これらの制約はすべて同じ検査式を持たなければなりません。さもなくば、エラーが報告されます。同じ名前と式を持つ制約は1つのコピーにまとめられます。親テーブルで`NO INHERIT`と印が付いた制約は考慮されません。一意な名前が常に選択されますので、新しいテーブル内の無名の`CHECK`制約はマージされないことに注意してください。

列の`STORAGE`設定もまた親テーブルからコピーされます。

```
LIKE source_table [ like_option ... ]
```

`LIKE`句にテーブルを指定すると、自動的にそのテーブルの全ての列名、そのデータ型、非`NULL`制約が新しいテーブルにコピーされます。

`INHERITS`との違いは、作成した後、新しいテーブルと元のテーブルが完全に分離されることです。元のテーブルへの変更は新しいテーブルには適用されません。また、元のテーブルを操作しても新しいテーブルのデータは見つかりません。

コピーする列のデフォルト式は、`INCLUDING DEFAULTS`が指定された場合にのみコピーされます。デフォルトでは、デフォルト式がコピーされないため、新しいテーブルのコピーされた列はデフォルト値として`NULL`を持つことになります。

非`NULL`制約は常に新しいテーブルにコピーされます。`CHECK`制約は、`INCLUDING CONSTRAINTS`が指定された場合にのみコピーされます。他の種類の制約は決してコピーされません。また、列制約とテーブル制約とを区別しません。制約が要求される場合は、すべての検査制約がコピーされます。

`INCLUDING INDEXES`句を指定しない限り、元のテーブル上のインデックスはまったく新しいテーブルには作成されません。

複製された列定義に関する`STORAGE`設定は、`INCLUDING STORAGE`が指定された場合のみコピーされます。このデフォルトの動作では、`STORAGE`設定は除外され、新しいテーブルにおけるコピーされた列は型固有のデフォルトの設定を持つようになります。`STORAGE`の詳細については[項56.2](#)を参照してください。

コピーされた列、制約、インデックスについてのコメントは`INCLUDING COMMENTS`が指定された場合のみコピーされます。このデフォルトの動作では、コメントは除外され、新しいテーブルにおけるコピーされた列や制約はコメントを持ちません。

`INCLUDING ALL`は`INCLUDING DEFAULTS INCLUDING CONSTRAINTS INCLUDING INDEXES INCLUDING STORAGE INCLUDING COMMENTS`の省略形です。

また、`INHERITS`と異なり、`LIKE`によりコピーされた列や制約は類似の名前の列や制約にまとめられません。同じ名前が明示的に、あるいは他の`LIKE`句で指定された場合、エラーが通知されます。

また`LIKE`句をビュー、外部テーブル、複合型から列をコピーするために使用することができます。不適切なオプション（ビューからの`INCLUDING INDEXES`など）は無視されます。

```
CONSTRAINT constraint_name
```

省略可能な列制約、テーブル制約の名前です。もし、この制約に違反すると、制約名がエラーメッセージに含まれるようになります。ですので、`col must be positive`(正数でなければならない)といった名前の制約名を付与することで、クライアントアプリケーションへ有用な制約情報を渡すことができます。（空白を含む制約名を指定する場合、二重引用符が必要です。）指定されなければ、システムが名前を生成します。

```
NOT NULL
```


その列がNULL値を持っていないことを指定します。

NULL

その列がNULL値を持てることを指定します。これがデフォルトです。

この句は非標準的なSQLデータベースとの互換性のためだけに提供されています。新しいアプリケーションでこれを使用するのはお勧めしません。

CHECK (*expression*) [NO INHERIT]

CHECKは、論理型の結果を生成する、新しい行または更新される行が挿入または更新処理を成功させるために満足しなければならない式を指定します。TRUEまたはUNKNOWNと評価される式は成功します。挿入または更新処理の行がFALSEという結果をもたらす場合はエラー例外が発生し、その挿入または更新によるデータベースの変更は行われません。列制約として指定された検査制約は列の値のみを参照しなければなりません。テーブル制約内の式は複数の列を参照できます。

現時点では、CHECK式には副問い合わせも現在の行の列以外の変数を含めることはできません。

NO INHERITと印が付いた制約は子テーブルには伝搬しません。

DEFAULT *default_expr*

DEFAULT句を列定義に付けると、その列にデフォルトデータ値が割り当てられます。値として指定するのは、任意の無変数式です（副問い合わせと現在のテーブル内の他の列へ交差参照は許可されません）。デフォルト式のデータ型はその列のデータ型と一致する必要があります。

デフォルト式は、全ての挿入操作において、その列に値が指定されていない場合に使用されます。列にデフォルト値がない場合、デフォルト値はNULLになります。

UNIQUE (列制約)

UNIQUE (*column_name* [, ...]) (表制約)

UNIQUE制約は、テーブルの1つまたは複数の列からなるグループが、一意な値のみを持つことができることを指定します。一意性テーブル制約の動作は一意性列制約と同じですが、さらに複数列にまたがる機能を持ちます。

一意性制約では、NULL値は等しいとはみなされません。

それぞれの一意性テーブル制約には、そのテーブルの他の一意性制約もしくはプライマリキー制約によって名付けられた列の集合とは、異なる名前の列の集合を指定しなければなりません（同じ名前を指定すると、同じ制約が2回現れるだけになります）。

PRIMARY KEY (列制約)

PRIMARY KEY (*column_name* [, ...]) (表制約)

プライマリキー制約は、テーブルの1列または複数列が一意な（重複がない）、非NULL値のみを持つことを指定します。技術的には、PRIMARY KEYは単なるUNIQUEとNOT NULLの組み合わせです。しかし、プライマリキーであることは他のテーブルがその列集合を一意な行識別子とみなせることを意味するので、列集合をプライマリキーと特定すると、スキーマ設計に関するメタデータを提供することになります。

列制約であるかテーブル制約であるかにかかわらず、1つのテーブルに指定できるプライマリキーは1つだけです。

プライマリキー制約には、そのテーブルに定義された他の一意性制約で指名された列の集合とは違う組み合わせの列の集合を指定しなければなりません。

EXCLUDE [USING *index_method*] (*exclude_element* WITH *operator* [, ...]) *index_parameters* [WHERE (*predicate*)]

EXCLUDE句は、任意の2行を指定した列(複数可)または式(複数可)を指定した演算子(複数可)を使用して比較した場合、比較結果のすべてがTRUEを返さないことを保障する、除外制約を定義します。指定した演算子のすべてが等価性を試験するものであれば、これはUNIQUE制約と同じです。しかし、通常の一意性制約のほうが高速です。しかし、除外制約では単純な等価性よりも一般的な制約を指定することができます。例えば、テーブル内の2つの行が重複しない円(項8.8参照)を持たないといった制約を<&演算子を使用して指定することができます。

除外制約はインデックスを使用して実装されています。このため指定した演算子はそれぞれ適切な演算子クラス(項11.9参照)で*index_method*インデックスアクセスメソッドと関連付けされていなければなりません。演算子は交換可能でなければなりません。省略可能ですが、各*exclude_element*は演算子クラス、順序付けオプション、またはその両方を指定することができます。これらについてはCREATE INDEXで説明します。

アクセスメソッドはamgettupleをサポートしなければなりません(第52章参照)。現時点では、これはGINを使用できないことを意味します。許されることですが、B-treeやHashインデックスを除外制約で使用する場合に少し問題がありま

す。これが通常の一意性制約がよりよく行わないことを行うためです。このため現実的にはアクセスメソッドは常にGiSTもしくはSP-GiSTとなります。

`predicate`により、除外制約をテーブルの部分集合に指定することができます。内部的には、これは部分インデックスを作成します。述語の前後に括弧が必要であることを注意して下さい。

```
REFERENCES reftable [ ( refcolumn ) ] [ MATCH matchtype ] [ ON DELETE action ] [ ON UPDATE action ] (列制約)
FOREIGN KEY ( column_name [, ... ] ) REFERENCES reftable [ ( refcolumn [, ... ] ) ] [ MATCH matchtype ] [ ON DELETE action ] [ ON UPDATE action ] (表制約)
```

これらの句は、外部キー制約を指定します。外部キー制約は、新しいテーブルの1つまたは複数の列の集合が、被参照テーブルの一部の行の被参照列（複数可）に一致する値を持たなければならないことを指定するものです。`refcolumn`が省略された場合、`reftable`のプライマリキーが使用されます。被参照列は、被参照テーブルにおいて遅延不可の一意性制約もしくはプライマリキー制約を持った列でなければなりません。一時テーブルと永続テーブルとの間で外部キー制約を定義できないことに注意してください。

参照列に挿入された値は、被参照テーブルと被参照列の値に対して、指定した照合型で照会されます。照合型には3種類があります。`MATCH FULL`、`MATCH PARTIAL`、デフォルトでもある`MATCH SIMPLE`照合型です。`MATCH FULL`は全ての外部キー列がNULLとなる場合を除き、複数列外部キーのある列がNULLとなることを許可しません。`MATCH SIMPLE`照合型は、外部キーの他の部分がNULLでない限り、外部キーの一部をNULLとなることを許可します。`MATCH PARTIAL`はまだ実装されていません。

さらに、被参照列のデータが変更された場合、このテーブルの列のデータに何らかの動作が発生します。`ON DELETE`句は、被参照テーブルの被参照行が削除された場合の動作を指定します。同様に`ON UPDATE`句は、被参照テーブルの被参照列が新しい値に更新された場合の動作を指定します。行の更新があった場合でも、被参照列が実際に変更されない場合は、動作は実行されません。制約が遅延可能と宣言されていても、`NO ACTION`検査以外の参照動作は遅延させられません。各句について、以下の動作を指定可能です。

`NO ACTION`

削除もしくは更新が外部キー制約違反となることを示すエラーが発生します。制約が遅延可能な場合、何らかの参照行が存在する限り、このエラーは制約の検査時点で発生します。これはデフォルトの動作です。

`RESTRICT`

削除もしくは更新が外部キー制約違反となることを示すエラーが発生します。検査が遅延できない点を除き、`NO ACTION`と同じです。

`CASCADE`

削除された行を参照している行は全て削除します。また、参照している列の値を、被参照列の新しい値にします。

`SET NULL`

参照する列（複数可）をNULLに設定します。

`SET DEFAULT`

参照する列（複数可）をそのデフォルト値に設定します。

被参照列が頻繁に更新される場合、外部キー列にインデックスを付け、その外部キー列に関連する参照動作がより効率的に実行できるようにする方が良いでしょう。

```
DEFERRABLE
NOT DEFERRABLE
```

制約を遅延させることが可能かどうかを制御します。遅延不可の制約は各コマンドの後すぐに検査されます。遅延可能な制約の検査は、([SET CONSTRAINTS](#)コマンドを使用して) トランザクションの終了時まで遅延させることができます。`NOT DEFERRABLE`がデフォルトです。現在、`UNIQUE`、`PRIMARY KEY`、`EXCLUDE`、`REFERENCES`（外部キー）制約のみがこの句を受け付けることができます。`NOT NULL`および`CHECK`制約は遅延させることができません。

```
INITIALLY IMMEDIATE
INITIALLY DEFERRED
```

制約が遅延可能な場合、この句は制約検査を行うデフォルトの時期を指定します。制約が`INITIALLY IMMEDIATE`の場合、各文の実行後に検査されます。これがデフォルトです。制約が`INITIALLY DEFERRED`の場合、トランザクションの終了時にのみ検査されます。制約検査の時期は[SET CONSTRAINTS](#)コマンドを使用して変更することができます。

```
WITH ( storage_parameter [= value] [, ... ] )
```

この句は、テーブルまたはインデックスに対して省略可能な格納パラメータを指定します。詳細は[格納パラメータ](#)を参照してください。テーブル用の`WITH`には、`oids=true`（もしくは単なる`oids`）を含めて、新しいテーブルの行が行に割り当てられたOID（オブジェクト識別子）を持たなければならないことを指定することもできます。また、`oids=false`を含め

て、OIDを持たないことを指定することもできます。 `oids`が指定されない場合、デフォルトの設定は[default_with_oids](#)設定パラメータに依存します。（新しいテーブルがOIDを持つテーブルから継承する場合、コマンドで`oids=false`と指定しても強制的に`oids=true` となります。）

`oids=false`が明示的または暗黙的に指定されている場合、新しいテーブルはOIDを格納しません。また、挿入される行にはOIDが割り当てられません。このような動作は一般的に有益であると考えられます。それは、OIDの使用を抑え、32ビットのOIDカウンタの回転周期を延長できるためです。カウンタが一周するとOIDの一意性を保証できなくなるので、その有用性を減少させることになります。また、OIDをなくすことで、テーブル1行当たり（ほとんどのマシンで）4バイト分、テーブルをディスクに格納するための容量を軽減するので、多少性能が向上します。

テーブルの作成後にOIDを削除するには、[ALTER TABLE](#)を使用してください。

```
WITH OIDS
WITHOUT OIDS
```

これは古い構文で、それぞれ`WITH (oids)` and `WITH (oids=false)`と同じです。 `oids`の設定と格納パラメータの設定の両方を指定したい場合は、上述の`WITH ( ... )`を使用しなければなりません。

```
ON COMMIT
```

`ON COMMIT`を使用して、トランザクションブロックの終了時点での一時テーブルの動作を制御することができます。以下の3つのオプションがあります。

```
PRESERVE ROWS
```

トランザクションの終了時点で、特別な動作は行われません。これがデフォルトの動作です。

```
DELETE ROWS
```

一時テーブル内の全ての行は、各トランザクションブロックの終わりで削除されます。基本的には、コミットの度に自動的に[TRUNCATE](#)が実行されます。

```
DROP
```

一時テーブルは、現在のトランザクションブロックの終了時点で削除されます。

```
TABLESPACE tablespace_name
```

*tablespace_name*は、新しいテーブルが作成されるテーブル空間名です。指定されていない場合、[default_tablespace](#)、もし一時テーブルの場合は[temp_tablespaces](#)が考慮されます。

```
USING INDEX TABLESPACE tablespace_name
```

この句により、`UNIQUE`、`PRIMARY KEY`、または`EXCLUDE`制約に関連したインデックスを作成するテーブル空間を選択することができます。指定されていない場合、[default_tablespace](#)、もし一時テーブルであれば[temp_tablespaces](#)が考慮されます。

格納パラメータ

`WITH`句により、テーブルおよび`UNIQUE`、`PRIMARY KEY`、または`EXCLUDE`制約と関連づいたインデックスの格納パラメータを指定することができます。インデックスの格納パラメータについては[CREATE INDEX](#)で説明します。現在テーブルで設定可能な格納パラメータの一覧を以下に示します。各パラメータに対して、注記がない限り、さらに`toast`という接頭辞のついた、同一の名前のパラメータがあります。これはもしあれば、テーブルの補助TOASTテーブルの動作を制御するために使用することができます。（TOASTに関する詳細については[項56.2](#)を参照してください。）TOASTテーブルは、`toast.autovacuum_*`の設定群がない場合に親テーブルの`autovacuum`値を継承することに注意してください。

```
fillfactor (integer)
```

テーブルのフィルファクタは10から100までの間の割合（パーセント）です。100（完全にすべて使用）がデフォルトです。より小さな値を指定すると、`INSERT`操作は指定した割合までしかテーブルページを使用しません。各ページの残りの部分は、そのページ内の行更新用に予約されます。これにより`UPDATE`は、元の行と同じページ上に更新済みの行の複製を格納することができるようになります。これは別のページに更新済みの行の複製を格納することよりも効率的です。項目の更新がまったくないテーブルでは、完全にすべてを使用することが最善の選択です。しかし、更新が非常に多いテーブルではより小さめのフィルファクタが適切です。TOASTテーブルではこのパラメータを設定できません。

```
autovacuum_enabled, toast.autovacuum_enabled (boolean)
```

特定のテーブルに対する自動バキュームデーモンを有効または無効にします。真の場合、自動バキュームデーモンは、更新または削除されたタプル数が`autovacuum_vacuum_threshold+autovacuum_vacuum_scale_factor`×リレーション内の推定有効タプル数を超えたときに、特定のテーブルに対する`VACUUM`操作を始めます。同様に、挿入、更新、削除されたタプ

ル数が`autovacuum_analyze_threshold`+`autovacuum_analyze_scale_factor`×リレーション内の推定有効タプル数を超えたときに、`ANALYZE`操作を始めます。 偽の場合、トランザクション周回問題を回避するため以外で自動バキュームは行われません。 周回問題の回避については[項23.1.5](#)を参照してください。 この変数が[autovacuum](#)からの値を継承していることに気付いてください。

`autovacuum_vacuum_threshold, toast.autovacuum_vacuum_threshold (integer)`

特定のテーブルに対して`VACUUM`操作を始める前の、更新または削除されたタプルの最小数です。

`autovacuum_vacuum_scale_factor, toast.autovacuum_vacuum_scale_factor (float4)`

`autovacuum_vacuum_threshold`に加算する`reltuples`用の乗数です。

`autovacuum_analyze_threshold (integer)`

特定のテーブルに対して`ANALYZE`操作を始める前の、挿入、更新または削除されたタプルの最小数です。

`autovacuum_analyze_scale_factor (float4)`

`autovacuum_analyze_threshold`に加算する`reltuples`用の乗数です。

`autovacuum_vacuum_cost_delay, toast.autovacuum_vacuum_cost_delay (integer)`

[autovacuum_vacuum_cost_delay](#)独自パラメータです。

`autovacuum_vacuum_cost_limit, toast.autovacuum_vacuum_cost_limit (integer)`

[autovacuum_vacuum_cost_limit](#)独自パラメータです。

`autovacuum_freeze_min_age, toast.autovacuum_freeze_min_age (integer)`

[vacuum_freeze_min_age](#)独自パラメータです。 テーブル単位の`autovacuum_freeze_min_age`をシステム全体の[autovacuum_freeze_max_age](#)設定の1/2より大きく設定しようとしても、自動バキュームが無視することに注意してください。

`autovacuum_freeze_max_age, toast.autovacuum_freeze_max_age (integer)`

[autovacuum_freeze_max_age](#)独自パラメータです。 テーブル単位の`autovacuum_freeze_max_age`をシステム全体に対する設定より大きく設定しようとしても、自動バキュームが無視することに注意してください。（より小さな値しか設定できません。） `autovacuum_freeze_max_age`を非常に小さく設定する、ゼロに設定することすらできますが、頻繁なバキューム処理が強制されるため通常は勧められません。

`autovacuum_freeze_table_age, toast.autovacuum_freeze_table_age (integer)`

Custom [vacuum_freeze_table_age](#) parameter.

注釈

新規のアプリケーションでOIDを使用するのはお勧めしません。 できる限り、テーブルのプライマリキーとして`SERIAL`や他のシーケンスジェネレータを使用の方が望ましいと考えられます。 しかし、アプリケーションがテーブルの特定の行を識別するためにOIDを使用する場合は、そのテーブルの`oid`列に一意性制約を作成することを推奨します。 これにより、カウンタが一周してしまった場合でも、テーブル内のOIDで一意に行を識別できることが保証されるからです。 OIDがテーブル全体で一意であると考えるのは止めてください。 データベース全体で一意的な識別子が必要な場合は、`tableoid`と行のOIDの組み合わせを使用してください。

タイプ: `oids=false`の使用は、プライマリキーのないテーブルでは推奨されません。 OIDも一意なデータキーも存在しないと、特定行を識別することが難しくなるからです。

PostgreSQLは自動的に各一意性制約とプライマリキー制約に対してインデックスを作成し、その一意性を確実なものにします。 したがって、プライマリキーの列に明示的なインデックスを作成することは必要ありません（詳細については[CREATE INDEX](#)を参照してください）。

現在の実装では、一意性制約とプライマリキーは継承されません。 これは、継承と一意性制約を組み合わせると障害が発生するからです。

テーブルは1600列以上の列を持つことはできません（タプル長の制限により実際の制限はもっと低くなります）。

例

`films` テーブルと `distributors` テーブルを作成します。

```
CREATE TABLE films (  
  code      char(5) CONSTRAINT firstkey PRIMARY KEY,  
  title     varchar(40) NOT NULL,  
  did       integer NOT NULL,  
  date_prod date,  
  kind      varchar(10),  
  len       interval hour to minute  
);  
  
CREATE TABLE distributors (  
  did       integer PRIMARY KEY DEFAULT nextval('serial'),  
  name      varchar(40) NOT NULL CHECK (name <> '')  
);
```

2次元配列を持つテーブルを作成します。

```
CREATE TABLE array_int (  
  vector    int[][]  
);
```

`films` テーブルに一意性テーブル制約を定義します。一意性テーブル制約はテーブルの1つ以上の列に定義することができます。

```
CREATE TABLE films (  
  code      char(5),  
  title     varchar(40),  
  did       integer,  
  date_prod date,  
  kind      varchar(10),  
  len       interval hour to minute,  
  CONSTRAINT production UNIQUE(date_prod)  
);
```

検査列制約を定義します。

```
CREATE TABLE distributors (  
  did       integer CHECK (did > 100),  
  name      varchar(40)  
);
```

検査テーブル制約を定義します。

```
CREATE TABLE distributors (  
  did       integer,  
  name      varchar(40)  
  CONSTRAINT con1 CHECK (did > 100 AND name <> '')  
);
```

`films` テーブルにプライマリーテーブル制約を定義します。

```
CREATE TABLE films (  
  code      char(5),  
  title     varchar(40),  
  did       integer,  
  date_prod date,  
  kind      varchar(10),  
  len       interval hour to minute,  
  CONSTRAINT code_title PRIMARY KEY(code,title)  
);
```

`distributors` テーブルにプライマリーキー制約を定義します。以下の2つの例は同じものです。前者はテーブル制約構文を使用し、後者は列制約構文を使用します。

```
CREATE TABLE distributors (  
  did       integer,  
  name      varchar(40),  
  PRIMARY KEY(did)  
);  
  
CREATE TABLE distributors (  
  did       integer PRIMARY KEY,  
  name      varchar(40)  
);
```

以下では、`name` 列のデフォルト値にリテラル定数を割り当てています。また、`did` 列のデフォルト値として、シーケンスオブジェクトの次の値が生成されるように調整しています。`modtime` のデフォルト値は、その行が挿入された時刻となります。


```
CREATE TABLE distributors (
    name      varchar(40) DEFAULT 'Luso Films',
    did       integer DEFAULT nextval('distributors_serial'),
    modtime   timestamp DEFAULT current_timestamp
);
```

2つのNOT NULL列制約をdistributorsテーブルに定義します。 そのうち1つには明示的な名前を付けています。

```
CREATE TABLE distributors (
    did       integer CONSTRAINT no_null NOT NULL,
    name      varchar(40) NOT NULL
);
```

name列に対し、一意性制約を定義します。

```
CREATE TABLE distributors (
    did       integer,
    name      varchar(40) UNIQUE
);
```

上をテーブル制約として指定します。

```
CREATE TABLE distributors (
    did       integer,
    name      varchar(40),
    UNIQUE(name)
);
```

テーブルとその一意性インデックスに70%のfillファクタを指定して、同じテーブルを作成します。

```
CREATE TABLE distributors (
    did       integer,
    name      varchar(40),
    UNIQUE(name) WITH (fillfactor=70)
)
WITH (fillfactor=70);
```

2つの円の重複を許さない除外制約を持つcirclesテーブルを作成します。

```
CREATE TABLE circles (
    c circle,
    EXCLUDE USING gist (c WITH &&)
);
```

diskv11テーブル空間にcinemasテーブルを作成します。

```
CREATE TABLE cinemas (
    id serial,
    name text,
    location text
) TABLESPACE diskv11;
```

複合型と型付きテーブルを作成します。

```
CREATE TYPE employee_type AS (name text, salary numeric);

CREATE TABLE employees OF employee_type (
    PRIMARY KEY (name),
    salary WITH OPTIONS DEFAULT 1000
);
```

互換性

CREATE TABLEは、以下の一覧を除いて、標準SQLに従います。

一時テーブル

CREATE TEMPORARY TABLEは標準SQLに類似していますが、その効果は同じではありません。 標準では、一時テーブルは一度だけ定義され、それを必要とするセッションごとに自動的に（空の内容で始まる形で）出現します。 PostgreSQLでは、これと異なり、各セッションで独自に、使用する一時テーブル用のCREATE TEMPORARY TABLEコマンドを発行しなければなりません。 これにより、異なるセッションで同じ名前の一時テーブルを異なる目的で使うことができます。 一方、標準の方法では、ある一時テーブル名を持つインスタンスが、全て同一のテーブル構造を持つという制限があります。

標準におけるこのような一時テーブルの動作定義はたいてい無視されています。 この点でのPostgreSQLの動作は、他の多くのSQLデータベースと似ています。

また標準SQLではグローバル一時テーブルとローカル一時テーブルを区別しています。ローカル一時テーブルは各セッション内のSQLモジュールそれぞれ用に内容の集合を分離しますが、その定義はセッション全体で共有されます。PostgreSQLはSQLモジュールをサポートしませんので、PostgreSQLではこの区別は適切ではありません。

互換性を保持するため、PostgreSQLは一時テーブルの宣言においてGLOBALとLOCALキーワードを受け付けますが、これらには現在、何の効果もありません。PostgreSQLの今後のバージョンでは、これらの意味についてより標準に近い実装を取り入れる可能性がありますので、これらのキーワードの使用は勧めません。

一時テーブル用のON COMMIT句もまた、標準SQLに類似していますが、数点の違いがあります。ON COMMIT句が省略された場合、SQLでは、デフォルトの動作はON COMMIT DELETE ROWSであると規定しています。しかし、PostgreSQLでのデフォルトの動作はON COMMIT PRESERVE ROWSです。また、ON COMMIT DROPはSQLにはありません。

非遅延一意性制約

UNIQUEまたはPRIMARY KEY制約が非遅延の場合、PostgreSQLは行が挿入または変更されると即座に一意性を検査します。標準SQLでは一意性は文が完了した時にのみ強制されなければならないと記述しています。これにより、たとえば、1つのコマンドが複数のキー値を更新する時に違いが現れます。標準互換の動作をさせるためには、非遅延（つまりINITIALLY IMMEDIATE）ではなくDEFERRABLEとして制約を宣言してください。これが即座に行われる一意性検査よりかなり低速になる可能性があることに注意してください。

列検査制約

標準SQLでは、CHECK列制約はそれを適用する列のみを参照でき、複数の列を参照できるのはCHECKテーブル制約のみであるとされています。PostgreSQLにはこの制限はありません。列検査制約とテーブル検査制約を同様のものとして扱っています。

EXCLUDE制約

EXCLUDEという種類の制約はPostgreSQLの拡張です。

NULL "制約"

NULL "制約"（実際には非制約）は、標準SQLに対するPostgreSQLの拡張で、他のいくつかのデータベースシステムとの互換性のために含まれています（NOT NULL制約と対称になります）。どんな列に対してもデフォルトとなるため、これはノイズに過ぎません。

継承

INHERITS句による複数継承は、PostgreSQLの言語拡張です。SQL:1999以降では、異なる構文と意味体系による単一継承を定義しています。今のところ、SQL:1999方式の継承はPostgreSQLではサポートされていません。

列を持たないテーブル

PostgreSQLでは、列を持たないテーブルを作成することができます（例えば、CREATE TABLE foo();）。これは標準SQLからの拡張です。標準SQLでは列を持たないテーブルは許されません。列を持たないテーブルそれ自体は役に立ちませんが、これを無効とすると、ALTER TABLE DROP COLUMNに対して奇妙な特例を生成することになります。したがって、この仕様上の制限を無視する方が簡潔であると考えます。

WITH句

WITH句はPostgreSQLの拡張です。格納パラメータもOIDも標準にはありません。

テーブル空間

PostgreSQLのテーブル空間の概念は標準にはありません。したがって、TABLESPACEとUSING INDEX TABLESPACEは、PostgreSQLにおける拡張です。

型付きテーブル

型付きテーブルは標準SQLのサブセットを実装します。 標準に従うと、型付きテーブルは背後の複合型に対応した列の他に"自己参照列"という列も持ちます。 PostgreSQLはこうした自己参照列を明示的にサポートしません。 しかし、OID機能を使用して同様の効果を持たせることができます。

関連項目

[ALTER TABLE](#), [DROP TABLE](#), [CREATE TABLESPACE](#), [CREATE TYPE](#)

[前のページ](#)

CREATE SERVER

[ホーム](#)

[上に戻る](#)

[次のページ](#)

CREATE TABLE AS

CREATE TABLE AS

名前

CREATE TABLE AS -- 問い合わせの結果によって新しいテーブルを定義する

概要

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE table_name
[ ( column_name [, ...] ) ]
[ WITH ( storage_parameter [= value] [, ...] ) | WITH OIDS | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE tablespace_name ]
AS query
[ WITH [ NO ] DATA ]
```

説明

CREATE TABLE AS はテーブルを作成し、SELECT コマンドによって算出されたデータをそのテーブルに格納します。テーブルの列は、SELECT の出力列に結び付いた名前とデータ型を持ちます（ただし、新しい列名を明示したリストを渡すと、この列名を上書きすることができます）。

CREATE TABLE AS はビューの作成と似ていますが、実際にはまったく異なります。CREATE TABLE AS は新しいテーブルを作成し、新しいテーブルの内容を初期化するために一度だけ問い合わせを評価します。それ以降に行われた、問い合わせの元テーブルに対する変更は、新しいテーブルには反映されません。反対に、ビューは問い合わせの度に定義されたSELECT文を再評価します。

パラメータ

GLOBALまたはLOCAL

互換性を保持するため、無視されます。これらのキーワードの使用は廃止予定です。詳細については[CREATE TABLE](#)を参照してください。

TEMPORARY or TEMP

指定された場合、テーブルは一時テーブルとして作成されます。 [CREATE TABLE](#)を参照してください。

UNLOGGED

指定された場合、テーブルはログを取らないテーブルとして作成されます。 [CREATE TABLE](#)を参照してください。

table_name

作成するテーブルの名前です（スキーマ修飾名も可）。

column_name

新しいテーブルにおける列の名前です。列名を指定しない場合は、問い合わせの出力列名を利用します。

WITH (*storage_parameter* [= *value*] [, ...])

この句は、新しいテーブル用の省略可能な格納パラメータを指定します。詳細は[格納パラメータ](#)を参照してください。WITHには、OIDS=TRUE（もしくは単なるOIDS）を含めて、新しいテーブルの行が行に割り当てられたOID（オブジェクト識別子）を持たなければならないことを指定することもできます。また、OIDS=FALSEを含めて、OIDを持たないことを指定することもできます。詳細は[CREATE TABLE](#)を参照してください。

WITH OIDS
WITHOUT OIDS

これらは古い構文で、それぞれWITH (OIDS)とWITH (OIDS=FALSE)と同じです。OIDSと格納パラメータの設定の両方を指定したい場合、上記のWITH (...)構文を使用しなければなりません。

ON COMMIT

トランザクションブロックの終了時の一時テーブルの動作を `ON COMMIT` を使用して制御することができます。以下の3つのオプションがあります。

`PRESERVE ROWS`

トランザクションの終了時に特別な処理は何も行われません。これがデフォルトの動作です。

`DELETE ROWS`

各トランザクションブロックの終了時に、一時テーブルのすべての行が削除されます。具体的には、コミット毎に自動的に [TRUNCATE](#) が行われます。

`DROP`

現在のトランザクションブロックの終了時に一時テーブルは削除されます。

`TABLESPACE` *tablespace_name*

tablespace_name は、新しいテーブルの作成先となるテーブル空間名です。指定がなければ、[default_tablespace](#)、もし一時テーブルの場合は [temp_tablespaces](#) が考慮されます。

query

[SELECT](#)、[TABLE](#)、[VALUES](#) コマンドまたは、あらかじめ準備された `SELECT`、`TABLE` または `VALUES` 問い合わせを実行する [EXECUTE](#) コマンドです。

`WITH` [`NO`] `DATA`

この句は問い合わせで生成されるデータを新しいテーブルにコピーすべきかどうかを指定します。コピーしない場合はテーブル構造のみがコピーされます。デフォルトではデータをコピーします。

注釈

このコマンドは、[SELECT INTO](#) と同じ機能を持ちますが、`SELECT INTO` 構文の他の使用方法と混乱する可能性が少ないため、こちらを使用の方が良いでしょう。さらに、`CREATE TABLE AS` は、`SELECT INTO` が提供する機能のスーパーセットを提供します。

PostgreSQL 8.0 より前は、`CREATE TABLE AS` が生成するテーブルには、常にOIDが含まれていました。PostgreSQL 8.0 では、ユーザが明示的に `CREATE TABLE AS` コマンドの結果にOIDを含めるかどうかを指定できるようになりました。OIDの有無を明示していない場合、[default_with_oids](#) 設定変数が使用されます。PostgreSQL 8.1 では [default_with_oids](#) のデフォルトは偽ですので、デフォルトの振舞いは8.0以前のリリースとは異なります。`CREATE TABLE AS` が生成するテーブルにおいてOIDが必要なアプリケーションでは、確実に希望する動作をさせるために明示的に `WITH (oids)` を付与しなければなりません。

例

`films` の最近の項目のみから構成される、新しい `films_recent` テーブルを作成します。

```
CREATE TABLE films_recent AS
SELECT * FROM films WHERE date_prod >= '2002-01-01';
```

テーブルを完全に複製するために、`TABLE` コマンドを使った短縮形も使用することができます。

```
CREATE TABLE films2 AS
TABLE films;
```

準備された文を使用して、`films` 内の最近の項目のみから構成される `films_recent` 一時テーブルを作成します。この新しいテーブルはOIDを持ち、コミット時に削除されます。

```
PREPARE recentfilms(date) AS
SELECT * FROM films WHERE date_prod > $1;
CREATE TEMP TABLE films_recent WITH (oids) ON COMMIT DROP AS
EXECUTE recentfilms('2002-01-01');
```

互換性

`CREATE TABLE AS` は標準SQLに従います。以下は非標準の拡張です。

- 標準では副問い合わせ句を括弧で囲む必要がありますが、PostgreSQLではこの括弧は省略可能です。

- 標準ではWITH [NO] DATA句は必須ですが、PostgreSQLでは省略可能です。
- PostgreSQLの一時テーブルの扱いは標準とは異なります。 詳細は[CREATE TABLE](#)を参照してください。
- WITH句はPostgreSQLの拡張です。 格納パラメータもOIDも標準にはありません。
- PostgreSQLのテーブル空間という概念は標準にはありません。 したがって、TABLESPACE句は拡張です。

関連項目

[CREATE TABLE](#), [EXECUTE](#), [SELECT](#), [SELECT INTO](#), [VALUES](#)

[前のページ](#)

CREATE TABLE

[ホーム](#)
[上に戻る](#)

[次のページ](#)

CREATE TABLESPACE

CREATE TABLESPACE

名前

CREATE TABLESPACE -- 新しいテーブル空間を定義する

概要

```
CREATE TABLESPACE tablespace_name [ OWNER user_name ] LOCATION 'directory'
```

説明

CREATE TABLESPACE はクラスタ全体で利用できるテーブル空間を新規に登録します。このテーブル空間名は、データベースクラスタ内の既存のテーブル空間名と異なるものでなければなりません。

テーブル空間を使用すると、スーパーユーザが、データベースオブジェクト（テーブルやインデックスなど）が入ったデータファイルを、ファイルシステム上のどの場所に格納するかを別途定義できます。

適切な権限を持つユーザは、CREATE DATABASE や CREATE TABLE、CREATE INDEX、ADD CONSTRAINT コマンドに *tablespace_name* を渡すことで、これらのオブジェクトを指定したテーブル空間に格納することができます。

パラメータ

tablespace_name

作成するテーブル空間の名前です。pg_ から始まる名前はシステムのテーブル空間用に予約されているので使用することはできません。

user_name

テーブル空間を所有するユーザの名前です。省略時はデフォルトでコマンドを実行したユーザとなります。テーブル空間を作成できるのはスーパーユーザのみですが、テーブル空間の所有権を非スーパーユーザに割り当てることは可能です。

directory

テーブル空間用に使用するディレクトリです。このディレクトリは空であるべきです。また、PostgreSQL システムユーザが所有していなければなりません。このディレクトリは絶対パス名で指定する必要があります。

注釈

テーブル空間をサポートしているのは、シンボリックリンクをサポートしているシステムのみです。

トランザクションブロック内で CREATE TABLESPACE を実行することはできません。

例

/data/dbs に dbspace テーブル空間を作成します。

```
CREATE TABLESPACE dbspace LOCATION '/data/dbs';
```

/data/indexes に、genevieve ユーザが所有する indexspace テーブル空間を作成します。

```
CREATE TABLESPACE indexspace OWNER genevieve LOCATION '/data/indexes';
```

互換性

CREATE TABLESPACEはPostgreSQLの拡張です。

関連項目

[CREATE DATABASE](#), [CREATE TABLE](#), [CREATE INDEX](#), [DROP TABLESPACE](#), [ALTER TABLESPACE](#)

[前のページ](#)

CREATE TABLE AS

[ホーム](#)

[上に戻る](#)

[次のページ](#)

CREATE TEXT SEARCH
CONFIGURATION

CREATE TEXT SEARCH CONFIGURATION

名前

CREATE TEXT SEARCH CONFIGURATION -- 新しいテキスト検索設定を定義する

概要

```
CREATE TEXT SEARCH CONFIGURATION name (  
    PARSER = parser_name |  
    COPY = source_config  
)
```

説明

CREATE TEXT SEARCH CONFIGURATION は新しいテキスト検索設定を作成します。テキスト検索設定は、文字列をトークンに分割するためのテキスト検索パーサを指定すると共に、どのトークンを検索対象とするかを決定するために使用する辞書を特定します。

パーサのみが指定された場合、新しいテキスト検索設定の初期状態では、トークン型と辞書との対応付けが存在しません。そのため、すべての単語が無視されます。この設定を実用できるようにするためには、後で ALTER TEXT SEARCH CONFIGURATION コマンドを使用して、対応付けを作成しなければなりません。この他に、既存のテキスト検索設定を複製することも可能です。

スキーマ名が指定された場合、テキスト検索設定は指定されたスキーマ内に作成されます。さもなければ、現在のスキーマ内に作成されます。

テキスト検索設定を定義したユーザが所有者になります。

詳細は [第12章](#) を参照してください。

パラメータ

name

作成するテキスト検索設定の名称です。名前をスキーマ修飾することができます。

parser_name

この設定で使用するテキスト検索パーサの名称です。

source_config

複製するテキスト検索設定の名称です。

注釈

PARSER と COPY オプションを同時に使用することはできません。既存の設定が複製される場合は、パーサの設定も複製されるためです。

互換性

標準SQLには CREATE TEXT SEARCH CONFIGURATION 文はありません。

関連項目

[ALTER TEXT SEARCH CONFIGURATION](#), [DROP TEXT SEARCH CONFIGURATION](#)

CREATE TEXT SEARCH DICTIONARY

名前

CREATE TEXT SEARCH DICTIONARY -- 新しいテキスト検索辞書を定義する

概要

```
CREATE TEXT SEARCH DICTIONARY name (  
    TEMPLATE = template  
    [, option = value [, ... ]]  
)
```

説明

CREATE TEXT SEARCH DICTIONARY は新しいテキスト検索辞書を作成します。テキスト検索辞書は、検索の際に何を対象とし、何を対象としないのかについての方法を指定します。実際に作業を行う関数を指定するテキスト検索テンプレートに、辞書は依存します。通常辞書は、テンプレートの関数の動作の詳細を制御するいくつかのオプションを提供します。

スキーマ名が指定された場合、テキスト検索辞書は指定されたスキーマ内に作成されます。さもなくば、現在のスキーマ内に作成されます。

テキスト検索辞書を定義したユーザが所有者になります。

詳細は[第12章](#)を参照してください。

パラメータ

name

作成するテキスト検索辞書の名称です。名前をスキーマ修飾することができます。

template

この辞書の基本動作を定義するテキスト検索テンプレートの名称です。

option

辞書に対して設定されるテンプレート固有のオプションの名称です。

value

テンプレート固有のオプションで使用される値です。値が単一識別子または数値でない場合、引用符で括らなければなりません。（常に引用符で括ることもできます。）

オプションは任意の順序で記述することができます。

例

次の例で示すコマンドは、非標準のストップワードのリストを持つ、雪だるま式に増加する辞書を作成します。

```
CREATE TEXT SEARCH DICTIONARY my_russian (  
    template = snowball,  
    language = russian,  
    stopwords = myrussian  
);
```

互換性

標準SQLにはCREATE TEXT SEARCH DICTIONARY 文はありません。

関連項目

[ALTER TEXT SEARCH DICTIONARY](#), [DROP TEXT SEARCH DICTIONARY](#)

前のページ CREATE TEXT SEARCH CONFIGURATION	ホーム 上に戻る	次のページ CREATE TEXT SEARCH PARSER

CREATE TEXT SEARCH PARSE

名前

CREATE TEXT SEARCH PARSE -- 新しいテキスト検索パーサを定義する

概要

```
CREATE TEXT SEARCH PARSE name (  
    START = start_function ,  
    GETTOKEN = gettoken_function ,  
    END = end_function ,  
    LEXTYPES = lextypes_function  
    [, HEADLINE = headline_function ]  
)
```

説明

CREATE TEXT SEARCH PARSE は新しいテキスト検索パーサを作成します。 テキスト検索パーサは、テキスト文字列をトークンに分割する方法を定義し、トークンに型（カテゴリ）を割り当てます。 パーサ自体は特別有用なものではありませんが、検索するためには、数個のテキスト検索辞書と共にテキスト検索設定と関連付けされなければなりません。

スキーマ名が指定された場合、テキスト検索パーサは指定されたスキーマ内に作成されます。 さもなくば、現在のスキーマに作成されます。

CREATE TEXT SEARCH PARSE を使用するには、スーパーユーザでなければなりません。（おかしなテキスト検索パーサ定義は混乱を引き起こし、サーバをクラッシュさせる可能性があるため、この制限があります。）

詳細は[第12章](#)を参照してください。

パラメータ

name

作成するテキスト検索パーサの名称です。 この名前はスキーマで修飾することができます。

start_function

パーサの開始関数の名称です。

gettoken_function

パーサの次トークン取り出し関数の名称です。

end_function

パーサの終了関数の名称です。

lextypes_function

パーサのLEXTYPE関数（生成するトークン型集合に関する情報を返す関数）の名称です。

headline_function

パーサの見出し関数（トークン集合を要約する関数）の名称です。

関数名は必要に応じてスキーマで修飾可能です。 各種関数の引数リストは事前にわかりますので、引数型の指定はありません。 見出し関数以外の関数はすべて必要です。

引数は、上で示した順序だけではなく、任意の順序で記述することができます。

互換性

標準SQLにはCREATE TEXT SEARCH PARSER文はありません。

関連項目

[ALTER TEXT SEARCH PARSER](#), [DROP TEXT SEARCH PARSER](#)

[前のページ](#)

CREATE TEXT SEARCH DICTIONARY

[ホーム](#)

[上に戻る](#)

[次のページ](#)

CREATE TEXT SEARCH TEMPLATE

CREATE TEXT SEARCH TEMPLATE

名前

CREATE TEXT SEARCH TEMPLATE -- 新しいテキスト検索テンプレートを定義する

概要

```
CREATE TEXT SEARCH TEMPLATE name (  
    [ INIT = init_function , ]  
    LEXIZE = lexize_function  
)
```

説明

CREATE TEXT SEARCH TEMPLATE は新しいテキスト検索テンプレートを作成します。 テキスト検索テンプレートは、テキスト検索辞書を実装する関数を定義します。 テンプレート自体は有用ではありませんが、使用される辞書として実体化されなければなりません。 通常この辞書はテンプレート関数に渡すパラメータを指定します。

スキーマ名が指定された場合、テキスト検索テンプレートは指定したスキーマに作成されます。 さもなくば、現在のスキーマに作成されます。

CREATE TEXT SEARCH TEMPLATE を使用するには、スーパーユーザでなければなりません。 おかしなテキスト検索テンプレート定義は混乱を引き起こし、サーバをクラッシュさせる可能性があるため、この制限があります。 辞書とテンプレートを分離させた理由は、テンプレートにより辞書定義の"安全でない"側面を隠蔽化することです。 辞書を定義する時に設定できるパラメータは、非特権ユーザが設定しても安全なものです。 このため、辞書の作成では特権操作は必要ありません。

詳細は[第12章](#)を参照してください。

パラメータ

name

作成するテキスト検索テンプレートの名称です。 この名前はスキーマで修飾することができます。

init_function

テンプレートの初期化関数の名称です。

lexize_function

テンプレートの字句化関数の名称です。

関数名は必要に応じてスキーマで修飾可能です。 各種関数の引数リストは事前にわかりますので、引数型の指定はありません。 字句化関数は必須ですが、初期化関数は省略可能です。

引数は、上で示した順序だけではなく、任意の順序で記述することができます。

互換性

標準SQLにはCREATE TEXT SEARCH TEMPLATE 文はありません。

関連項目

[ALTER TEXT SEARCH TEMPLATE](#), [DROP TEXT SEARCH TEMPLATE](#)

CREATE TRIGGER

名前

CREATE TRIGGER -- 新しいトリガを定義する

概要

```
CREATE [ CONSTRAINT ] TRIGGER name { BEFORE | AFTER | INSTEAD OF } { event [ OR ... ] }
ON table_name
[ FROM referenced_table_name ]
[ { NOT DEFERRABLE | [ DEFERRABLE ] { INITIALLY IMMEDIATE | INITIALLY DEFERRED } } ]
[ FOR [ EACH ] { ROW | STATEMENT } ]
[ WHEN ( condition ) ]
EXECUTE PROCEDURE function_name ( arguments )
```

ここでeventは以下のいずれかを取ることができます。

```
INSERT
UPDATE [ OF column_name [, ... ] ]
DELETE
TRUNCATE
```

説明

CREATE TRIGGERは新しいトリガを作成します。 作成したトリガは指定したテーブルまたはビューと関連付けられ、特定のイベントが発生した時に指定した関数function_nameを実行します。

トリガでは、起動のタイミングとして、行への操作が開始される前（制約条件のチェックとINSERT、UPDATEまたはDELETEが行われる前）、操作が完了した後（制約条件がチェックされ、INSERT、UPDATEまたはDELETEが完了した後）、操作の代わり（ビューにおける挿入、更新、削除の場合）のいずれかを指定することができます。 イベントの前または代わりにトリガが起動する場合、そのトリガは対象行に対する操作を省略したり、（INSERTとUPDATEの操作時のみ）挿入する行を変更したりすることができます。 イベントの後にトリガが起動する場合、他のトリガの影響を含む全ての変更が、トリガに対して"可視"状態となります。

FOR EACH ROW付きのトリガは、その操作によって変更される行ごとに1回ずつ呼び出されます。 例えば、10行に影響を与えるDELETE操作は、対象リレーション上のON DELETEトリガを、行が削除される度に1回ずつ、個別に10回呼び出すこととなります。 反対に、FOR EACH STATEMENT付きのトリガは、その操作によって何行変更されたかにかかわらず、任意の操作ごとに1回のみ実行されます（変更対象が0行となる操作でも、適用できるFOR EACH STATEMENTトリガがあれば実行されます）。

トリガイベントのINSTEAD OFとして発行されるように指定されたトリガはFOR EACH ROW印を付けなければなりません。 このためビュー上にのみ定義することができます。 ビューに対するBEFOREおよびAFTERトリガはFOR EACH STATEMENT印を付けなければなりません。

さらに、FOR EACH STATEMENTのみですが、トリガをTRUNCATEで発行するように定義することができます。

以下の表にどの種類のトリガがテーブルとビューに対して使用できるかをまとめます。

いつ	イベント	行レベル	文レベル
BEFORE	INSERT/UPDATE/DELETE	テーブル	テーブルおよびビュー
	TRUNCATE	—	テーブル
AFTER	INSERT/UPDATE/DELETE	テーブル	テーブルおよびビュー
	TRUNCATE	—	テーブル
INSTEAD OF	INSERT/UPDATE/DELETE	ビュー	—
	TRUNCATE	—	—

またトリガ定義では、トリガを発行すべきかどうかを判定する、論理値のWHEN条件を指定することができます。 行レベルのトリガでは、WHEN条件は行の列の古い値、新しい値、またはその両方で検証することができます。 文レベルのトリガでもWHEN条件を持たせることができますが、条件としてテーブル内の何らかの値を参照することができませんので、この機能はあまり有用ではありません

同一イベントに同じ種類の複数のトリガが定義された場合、名前のアルファベット順で実行されます。

`CONSTRAINT` オプションが指定された場合、このコマンドは制約トリガを作成します。これは、[SET CONSTRAINTS](#)を使用してトリガを発行するタイミングを調整することができるという点を除き、通常のトリガと同じです。制約トリガは`AFTER ROW`トリガでなければなりません。トリガイベントを引き起こした文の最後、またはそれを含むトランザクションの最後のいずれかで発行することができます。後者の場合、遅延と呼ばれます。`SET CONSTRAINTS`を使用することで、強制的に待機中の遅延トリガの発行を即座に行わせることができます。制約トリガは、実装する制約に違反した時に例外を発生するものと想定しています。

`SELECT` はまったく行を変更しないため、`SELECT` トリガを作成することはできません。この場合は、ルールやビューの方が適しています。

トリガに関するより詳細については、[第36章](#)を参照してください。

パラメータ

name

新しいトリガに付与する名前です。同じテーブルの他のトリガと異なる名前にする必要があります。名前をスキーマ修飾することはできません。トリガはそのテーブルのスキーマを引き継ぎます。制約トリガの場合、この名前が`SET CONSTRAINTS`を使用してトリガの動作を変更する時に使用されます。

`BEFORE`
`AFTER`
`INSTEAD OF`

関数の呼び出しをイベントの前に行うか後に行うか、それとも代替として行うかを決定します。制約トリガでは`AFTER`としてしか指定することができません。

event

`INSERT`、`UPDATE`、`DELETE`、`TRUNCATE`のいずれかが入ります。このパラメータは、トリガを起動するイベントを指定します。`OR`を使用して、複数のイベントを指定することができます。

`UPDATE` イベントでは、以下の構文を使用して列リストを指定することができます。

```
UPDATE OF column_name1 [, column_name2 ... ]
```

このトリガは`UPDATE` コマンドの対象として列挙された列のいずれかが少なくとも1つの列が指定された場合に発行されます。

`INSTEAD OF UPDATE` イベントは列リストをサポートしません。

table_name

トリガを作成するテーブルまたはビューの名前です（スキーマ修飾名も可）。

referenced_table_name

制約で参照される（おそらくスキーマ修飾された）他のテーブルの名前です。このオプションは外部キー制約で使用されるものであり、一般利用を推奨しません。これは制約トリガでのみ指定することができます。

`DEFERRABLE`
`NOT DEFERRABLE`
`INITIALLY IMMEDIATE`
`INITIALLY DEFERRED`

トリガのデフォルトのタイミングです。これらの制約オプションについては[CREATE TABLE](#) 文書を参照してください。これは制約トリガでのみ指定することができます。

`FOR EACH ROW`
`FOR EACH STATEMENT`

このパラメータは、トリガプロシージャを、トリガイベントによって影響を受ける行ごとに1回起動するか、SQL文ごとに1回のみ起動するかを指定します。どちらも指定されない場合は、`FOR EACH STATEMENT` がデフォルトです。制約トリガでは`FOR EACH ROW` のみ指定することができます。

condition

トリガ関数を実際に実行するか否かを決定する論理式です。`WHEN` が指定された場合、*condition* が `true` を返す場合のみ関数が呼び出されます。`FOR EACH ROW` トリガでは、`WHEN` 条件で、それぞれ `OLD.column_name`、`NEW.column_name` と記述することで、古い行値、新しい行値、またはその両方の列を参照することができます。当然ながら `INSERT` トリガでは `OLD` を参照することはできませんし、`DELETE` トリガでは `NEW` を参照することはできません。

`INSTEAD OF` トリガは `WHEN` 条件をサポートしません。

現時点ではWHEN条件に副問い合わせを含めることはできません。

制約トリガでは、WHEN条件の評価は遅延されず、行の更新操作が行われた直後に発生することに注意してください。この条件が真と評価されなかった場合、トリガは遅延実行タイプのキューに入りません。

function_name

ユーザが提供する関数です。この関数は、引数を取らずtrigger型を返すよう定義します。トリガが起動した時に実行されます。

arguments

トリガ実行時に関数に渡される引数をカンマで区切ったリストです。このパラメータは省略可能です。引数として指定するのは、リテラル文字列定数です。単純な名前および数値定数を記述できますが、全て文字列に変換されます。関数内でこれらの引数にアクセスする方法について調べるためには、トリガ関数を実装した言語の説明を参照してください。通常の関数引数とは異なる場合があります。

注釈

テーブルにトリガを作成するには、ユーザがそのテーブルに対しTRIGGER権限を持っている必要があります。またユーザはトリガ関数に対しEXECUTE権限を持たなければなりません。

トリガを削除するためには[DROP TRIGGER](#)を使用してください。

列指定のトリガ（UPDATE OF column_name構文で定義されたトリガ）は、列挙された列のいずれかがUPDATEコマンドのSETリスト内に対象として指定された場合に発行されます。BEFORE UPDATEトリガにより行の内容になされた変更が考慮されないため、トリガが発行されない場合であっても、列の値を変更することができます。反対に、UPDATE ... SET x = x ...のようなコマンドは、列の値が変更されていない場合であっても、x列に対するトリガは発行されます。

BEFOREトリガにおいてWHEN条件は関数が実行される、またはされそうな場合のみ評価されます。このためWHENの使用はトリガ関数の先頭で同一の条件を試験することと実質違いはありません。この条件で確認できるNEW行が現在の値であり、それまでのトリガで変更されている可能性があることに、特に注意して下さい。またBEFOREトリガのWHEN条件では、まだ設定されていないので、NEW行のシステム列(oidなど)を検査することができません。

AFTERトリガにおいて、WHEN条件は行の更新を行った直後に評価され、文の最後でトリガを発行するためにイベントを保持すべきかどうかを決定します。このためAFTERトリガのWHEN条件は真を返さない場合、イベントを保持する必要もありませんし、文の最後の行を再度取り出す必要もありません。これにより、トリガをわずかな行のみに対して発行する必要がある場合、多くの行を変更する文を非常に高速にすることができます。

PostgreSQL 7.3より前のバージョンでは、トリガ関数の戻り値の型を、trigger型ではなくプレースホルダであるopaque型として宣言する必要がありました。古いダンプファイルのロードをサポートするため、CREATE TRIGGERではopaque型を返すよう宣言された関数を受け入れます。しかし、注意を促すメッセージを表示し、宣言された関数の戻り値型をtriggerに変換します。

例

accountsテーブルの行が更新される直前にcheck_account_update関数を実行します。

```
CREATE TRIGGER check_update
  BEFORE UPDATE ON accounts
  FOR EACH ROW
  EXECUTE PROCEDURE check_account_update();
```

上と同じです。しかし、balance列がUPDATEコマンドの対象として使用された場合のみ実行されます。

```
CREATE TRIGGER check_update
  BEFORE UPDATE OF balance ON accounts
  FOR EACH ROW
  EXECUTE PROCEDURE check_account_update();
```

以下の構文では、balance列が実際に変更された場合のみ関数が実行されます。

```
CREATE TRIGGER check_update
  BEFORE UPDATE ON accounts
  FOR EACH ROW
  WHEN (OLD.balance IS DISTINCT FROM NEW.balance)
  EXECUTE PROCEDURE check_account_update();
```

何か変更された場合のみにaccountsの更新のログを取る関数を呼び出します。

```
CREATE TRIGGER log_update
  AFTER UPDATE ON accounts
  FOR EACH ROW
  WHEN (OLD.* IS DISTINCT FROM NEW.*)
  EXECUTE PROCEDURE log_account_update();
```

ビューの背後にあるテーブルに行を挿入するために、各行に対してview_insert_row関数を実行します。

```
CREATE TRIGGER view_insert
  INSTEAD OF INSERT ON my_view
  FOR EACH ROW
  EXECUTE PROCEDURE view_insert_row();
```

[項36.4](#)には、C言語で作成されたトリガ関数の完全な例があります。

互換性

PostgreSQLにおけるCREATE TRIGGER文は標準SQLのサブセットを実装したものです。ただし、PostgreSQLには、次の機能がありません。

- SQLでは、"old"や"new"、トリガによって起動する動作の定義が使用するテーブルに、別名を定義することができます。（例えば、CREATE TRIGGER ... ON tablename REFERENCING OLD ROW AS somename NEW ROW AS othername ...）。PostgreSQLでは、複数のユーザ定義言語でトリガプロシージャを作成できるので、データへのアクセスは言語固有の方法で扱われます。
- PostgreSQLでは、トリガ動作として、1つのユーザ定義関数の実行しか認めていません。標準では、他のSQLコマンドを複数実行させることができます。例えば、トリガ動作としてCREATE TABLEを実行させることも可能です。この制限を回避する方法は簡単です。必要なコマンドを実行するユーザ定義関数を作成すればよいのです。

SQLでは、複数のトリガは、作成時刻順に起動すべきであると規定しています。PostgreSQLでは名前順であり、より簡単に判断することができます。

SQLでは、数珠繋ぎの削除に対するBEFORE DELETEは、数珠繋ぎのDELETEが完了した後に発行するものと規定しています。PostgreSQLでは、BEFORE DELETEは常に削除操作よりも後に行われます。この方がより一貫性があると考えています。また、参照整合性に関する動作により引き起こる更新を実行している間に、BEFOREトリガが行を更新し、更新を妨げるような場合の動作は非標準的です。これは、制約違反となるかもしれませんが、参照整合性制約に合わないデータを格納してしまうかもしれません。

ORを使用して単一トリガに複数の動作を指定する機能は、標準SQLに対するPostgreSQLの拡張です。

TRUNCATEでのトリガ発行機能、および、ビューに対する文レベルトリガの定義機能は標準SQLに対するPostgreSQLの拡張です。

CREATE CONSTRAINT TRIGGERは標準SQLに対するPostgreSQLの拡張です。

関連項目

[CREATE FUNCTION](#), [ALTER TRIGGER](#), [DROP TRIGGER](#), [SET CONSTRAINTS](#)

[前のページ](#)

CREATE TEXT SEARCH TEMPLATE

[ホーム](#)
[上に戻る](#)

[次のページ](#)
CREATE TYPE

CREATE TYPE

名前

CREATE TYPE -- 新しいデータ型を定義する

概要

```

CREATE TYPE name AS
    ( [ attribute_name data_type [ COLLATE collation ] [, ... ] ] )

CREATE TYPE name AS ENUM
    ( [ 'label' [, ... ] ] )

CREATE TYPE name AS RANGE (
    SUBTYPE = subtype
    [ , SUBTYPE_OPCLASS = subtype_operator_class ]
    [ , COLLATION = collation ]
    [ , CANONICAL = canonical_function ]
    [ , SUBTYPE_DIFF = subtype_diff_function ]
)

CREATE TYPE name (
    INPUT = input_function,
    OUTPUT = output_function
    [ , RECEIVE = receive_function ]
    [ , SEND = send_function ]
    [ , TYPMOD_IN = type_modifier_input_function ]
    [ , TYPMOD_OUT = type_modifier_output_function ]
    [ , ANALYZE = analyze_function ]
    [ , INTERNALLENGTH = { internallength | VARIABLE } ]
    [ , PASSEDBYVALUE ]
    [ , ALIGNMENT = alignment ]
    [ , STORAGE = storage ]
    [ , LIKE = like_type ]
    [ , CATEGORY = category ]
    [ , PREFERRED = preferred ]
    [ , DEFAULT = default ]
    [ , ELEMENT = element ]
    [ , DELIMITER = delimiter ]
    [ , COLLATABLE = collatable ]
)

CREATE TYPE name

```

説明

`CREATE TYPE`は、現在のデータベースで使用できる新しいデータ型を登録します。型を定義したユーザがその所有者となります。

スキーマ名が与えられている場合、型は指定されたスキーマで作成されます。スキーマ名がなければ、その型は現在のスキーマで作成されます。型名は、同じスキーマにある既存の型もしくはドメインとは、異なる名前にする必要があります（さらに、テーブルはデータ型と関連しているため、データ型名は同じスキーマのテーブル名とも競合しないようにしてください）。

上の構文概要に示すように、`CREATE TYPE`には5つの構文があります。これらはそれぞれ、複合型、列挙型、範囲型、基本型、シェル型を作成します。これらの内最初の4個についてはここで順番に説明します。シェル型は、単なる、後で定義される型用のプレースホルダです。これは型名以外のパラメータを持たずに`CREATE TYPE`を実行することで作成されます。シェル型は、それぞれの節で説明するように範囲型と基本型を作成するときの前方参照として必要なものです。

複合型

`CREATE TYPE`の最初の構文を使用すると、複合型を作成できます。複合型は、列名およびデータ型のリストにより指定されます。データ型の照合順序が設定可能である場合、属性の照合順序も指定することができます。複合型は本質的にはテーブルの行型と同じです。しかし、型を定義するだけであれば、`CREATE TYPE`を使用することで、実際のテーブルを作成する必要をなくすることができます。スタンドアロンの複合型は、例えば関数の引数や戻り値の型として使用すると有用です。

複合型を作成するためには、すべての属性型に対して`USAGE`権限を持たなければなりません。

列挙型

`CREATE TYPE`の2つ目の構文を使用すると、[項8.7](#)で説明する列挙型 (enum) を作成します。列挙型は、1つ以上の引用符付きのラベルのリストを取ります。ラベルは`NAMEDATALEN` (PostgreSQLを標準で構築した場合は64バイト) バイトよりも少ない長さでなければなりません。

範囲型

`CREATE TYPE`の三番目の構文は、[項8.17](#)で説明する範囲型を新規に作成します。

範囲型の *subtype* は、関連する (範囲型の値を順序を決定するための) b-tree 演算子クラスを持つ任意の型を取ることができます。通常、派生元型のデフォルトの b-tree 演算子クラスが順序を決定するために使用されます。デフォルト以外の演算子クラスを使用するためには、*subtype_opclass* でその名前を指定してください。派生元型が照合順序変更可能であり、範囲の順序付けでデフォルト以外の照合順序を使用したい場合は、*collation* オプションで使用したい照合順序を指定してください。

省略可能な *canonical* 関数は、定義する範囲型の引数を1つ取り、同じ型の値を返さなければなりません。これは適切な時に範囲値を正規形式に変換するために使用されます。詳細については[項8.17.8](#)を参照してください。範囲型を定義できるようになる前に定義されている必要があるため、*canonical* 関数を作成することは多少厄介です。このためには、まず、名前と所有者以外の属性を持たないプレースホルダであるシェル型を作成しなければなりません。これは、他にパラメータを持たない `CREATE TYPE name` を実行することで行われます。その後、このシェル型を引数と結果として使用する関数を宣言することができます。最後に同じ名前を用いて範囲型を宣言することができます。これは自動的にシェル型の項目と有効な範囲型に置き換えます。

省略可能な *subtype_diff* 関数は、*subtype* 型の2つの値を引数として取り、与えられた2つの値の差異を表す `double precision` 型を返さなければなりません。これは省略することができますが、提供することでその範囲型の列に対する GiST インデックスの効率を大きく向上させることができます。詳細については[項8.17.8](#)を参照してください。

基本型

`CREATE TYPE`の4つ目の構文を使用すると、基本型 (スカラ型) を新しく作成できます。新しい基本型を作成するにはスーパーユーザでなければなりません。(エラーがある型定義が混乱を招き、サーバがクラッシュすることすらあるため、この制限がなされました。)

パラメータは、上述の順番である必要はなく、任意の順番で指定することができます。型を定義する前に、(`CREATE FUNCTION` を用いて) 2つ以上の関数を登録しておく必要があります。サポート関数である *input_function* と *output_function* は必須です。*receive_function* 関数、*send_function* 関数、*type_modifier_input_function* 関数、*type_modifier_output_function* 関数、および *analyze_function* 関数は省略可能です。通常、これらの関数は、C言語やその他の低レベル言語で作成されなければなりません。

input_function は、型のテキストによる外部表現を内部表現形式に変換するものであり、その型用に定義される演算子や関数で使用されます。*output_function* はこの逆の変換を行うものです。入力関数は、1つの `cstring` 型の引数、あるいは、`cstring` 型、`oid` 型、`integer` 型という3つの引数を取るように宣言されます。最初の引数にはC文字列の入力テキスト、2番目には型自体のOID (配列型の場合は例外で要素の型のOIDとなります)、3番目は、判明していれば対象列の `typmod` を渡します (不明な場合は-1を渡します)。この入力関数では、データ型自身の値を返さなければなりません。通常入力関数は `STRICT` として宣言しなければなりません。そうしないと、`NULL` という入力値を読み取った時、`NULL` という最初のパラメータを持って呼び出されます。この場合でも関数は `NULL` を返さなければなりません。さもなくばエラーになります。(こうした状況はほとんどの場合、ドメイン入力関数をサポートすることを意図しています。ドメイン入力関数では `NULL` 入力を拒絶する可能性があります。) 出力関数は、新しいデータ型の引数を1つ取るように宣言しなければなりません。出力関数は、`cstring` 型を返さなければなりません。出力関数は `NULL` 値に対して呼び出されることはありません。

receive_function では、型のバイナリによる外部表現を内部表現に変換します。この関数は省略可能です。この関数が与えられない場合、この型ではバイナリ入力を行うことができません。バイナリ表現の方法は、適度な移植性を保ちつつ、内部表現への変換コストが小さくなるよう選択すべきです (例えば標準の整数データ型は、外部バイナリ表現としてはネットワークバイトオーダーを使用し、内部表現ではマシン固有のバイトオーダーを使用します)。この受信関数では、値が有効かどうかを判定するための適切な検査を行わなければなりません。受信関数は、`internal` 型の引数1つ、または、`internal` 型と `oid`、`integer` 型の3つの引数を取るように宣言されます。最初の引数は受信したバイト文字列を保持する `StringInfo` バッファへのポインタ、省略可能な引数は、入力関数の説明と同じです。そして、データ型自体の値を返す必要があります。通常受信関数は `STRICT` として宣言しなければなりません。そうしないと、`NULL` という入力値を読み取った時、`NULL` という最初のパラメータを持って呼び出されます。この場合でも関数は `NULL` を返さなければなりません。さもなくばエラーになります。(こうした状況はほとんどの場合、ドメイン受信関数をサポートすることを意図しています。ドメイン受信関数では `NULL` 入力を拒絶する可能性があります。) 同様に、*send_function* は、内部表現からバイナリによる外部表現に変換します。この関数も省略可能です。この関数が与えられない場合、この型ではバイナリ出力を行うことができません。この送信関数は、新しいデータ型の引数1つを取るように宣言しなければなりません。送信関数は `bytea` 型を返さなければなりません。送信関数は `NULL` 値に対して呼び出されません。

ここで、新しいデータ型を作成する前に入力関数と出力関数を作成する必要があるのに、どのようにしてそれらの関数で新しいデータ型を戻り値や入力として宣言できるのか、疑問に思われるかもしれません。その答えは、まず型が最初にシェル型として定義されます。この型はプレースホルダ型であり、名称と所有者以外の属性を持ちません。これは、他にパラメータを持たない `CREATE TYPE name` コマンドを発行することで行われます。その後、入出力関数をこのシェル型を参照するように定義することができます。最後に完全な定義を持った `CREATE TYPE` によって、シェル型の項目が完全かつ有効な型定義に置き換わり、新しい型を普通に使用することができるようになります。

`type_modifier_input_function` と `type_modifier_output_function` は省略可能ですが、型が修正子をサポートする場合は必要です。修正子とは、`char(5)` や `numeric(30,2)` などの省略可能な型宣言に付与される制約です。PostgreSQLでは、ユーザ定義型が1つ以上の整数定数または識別子を修正子として取ることができます。しかし、この情報はシステムカタログに格納される時に単一非負の整数にまとめられるものでなければなりません。`type_modifier_input_function` には、`cstring` 型配列の形で宣言された修正子が渡されます。その値について妥当性を検査しなければなりません（不当な場合はエラーとします）。そして、正しい場合は、"typmod"列として格納される、単一非負の `integer` 値を返さなければなりません。型が `type_modifier_input_function` を持たない場合、型修正子は拒否されます。`type_modifier_output_function` は内部的な整数 `typmod` 値をユーザ側の表示に合わせて変換します。この関数は型名に付与する正確な文字列となる `cstring` 値を返さなければなりません。たとえば `numeric` 用の関数では `(30,2)` を返すかもしれません。デフォルトの表示用書式が保管された `typmod` 整数値を括弧で括ったものと一致している場合は、`type_modifier_output_function` を省略することができます。

`analyze_function` は、このデータ型の列に対する、型固有の統計情報の収集を行います。この関数は省略可能です。デフォルトでは、その型用のデフォルトのB-tree演算子クラスがあれば、`ANALYZE` は型の"等価"演算子と"小なり"演算子を使用して統計情報を集めようと試みます。非スカラ型には、この振舞いはあまり適していません。そのため、独自の解析関数を指定すると、この振舞いを上書きすることができます。この解析関数は、`internal` 型の引数を1つ取り、戻り値として `boolean` を返すように宣言する必要があります。解析関数用のAPIの詳細は、`src/include/commands/vacuum.h` にあります。

新しい型の内部表現の詳細を理解しなければならないのは、これらのI/O関数とその型に関連して動作するユーザ定義の関数のみですが、内部表現には、PostgreSQLに対し宣言しなければならない複数の属性値があります。属性の中で最も重要なものは `internallength` です。基本データ型は、`internallength` に正の整数を指定して固定長として作成するだけでなく、`internallength` に `VARIABLE` と設定し可変長として作成することもできます（内部的には、これは `typelen` を -1 に設定することで表現されます）。全ての可変長型の内部表現は、型の値の全長を示す4バイトの整数値から始まらなければなりません。

`PASSEDBYVALUE` フラグは、このデータ型の値が参照ではなく値によって渡されることを示します。このフラグは省略可能です。`Datum` 型のサイズ（ほとんどのマシンでは4バイトだが、まれに8バイト）よりも長い内部表現の型は値で渡すことができます。

`alignment` パラメータは、そのデータ型の格納の際に整列が必要であることを示します。設定可能な値は、1、2、4、8バイト境界での整列と同じです。可変長型の位置配置は最低でも4を持たなければならないことに注意してください。最初の要素として `int4` を持たなければならないからです。

`storage` パラメータを使用することで、可変長データ型を格納する際の戦略を選択することができます（固定長の型には `plain` のみが使用できます）。`plain` を指定すると、その型のデータは常にインラインで格納され、圧縮されません。`extended` を指定すると、システムはまず長いデータ値を圧縮しようとし、それでもまだ長過ぎる場合は値をメインテーブルの行から削除して移動します。`external` はメインテーブルから値を削除して移動することを許しますが、システムはデータを圧縮しようとしません。`main` はデータの圧縮を許し、できるだけ値をメインテーブルから削除しないようにします（行を収めるために他に方法がない場合にはメインテーブルから削除されてしまう可能性があります、`extended` や `external` が指定されたアイテムよりも優先してメインテーブルに残されます）。

`like_type` パラメータは、何らかの既存のデータ型から複製するという、データ型の基本表現プロパティを指定する、別の方法を提供します。`internallength`、`passedbyvalue`、`alignment`、`storage` の値が指定された型から複製されます。（普通好まれません、`LIKE` 句と一緒にこれらの値を指定することで、値を上書きすることも可能です。）新しい型の低レベル実装にある流儀に従った既存の型を"移す"時に、この方法で表現を指定することが特に有用です。

`category` と `preferred` パラメータは、暗黙的なキャストがあいまいな状況で適用されるかどうかについての制御を補助するために使用することができます。各データ型は単一のASCII文字で命名されるカテゴリに属しています。そして、各型はそのカテゴリで"好まれる（preferred）"か否かを持ちます。パーサは（同一のカテゴリ内の他の型からの）好まれる型へのキャストを、この規則が上書きされた関数または演算子の解決に有用な場合に優先します。より詳細は[第10章](#)を参照してください。他の型への、または、ほかの型からの暗黙的なキャストを持たない型では、これらの設定をデフォルトのままにしておくことで十分です。しかし、暗黙的なキャストを持つ関連するグループでは、それらすべてをカテゴリに属し、その1つまたは2つをカテゴリ内で好まれるものとして"最も一般的な"型として選択することが有用となる場合が多くあります。ユーザ定義型を、数値型や文字列型などの既存の組み込みカテゴリに追加する場合に、`category` パラメータは特に有用です。しかし、すべてがユーザ定義型のカテゴリを作成することもできます。カテゴリとして命名されていない任意の大文字を選択してください。

ユーザがそのデータ型の列のデフォルトを `NULL` 以外にしたい場合は、デフォルト値を指定することができます。デフォルト値は `DEFAULT` キーワードで指定してください（この方法で指定されたデフォルト値は、特定の列に付与された、明示的な `DEFAULT` 句によって上書きされる可能性があります）。

データ型が配列であることを示すには、`ELEMENT` キーワードを使用して配列要素の型を指定してください。例えば、4バイト整数

(`int4`) の配列を定義するには、`ELEMENT = int4`と指定してください。 詳細は後述の配列型で説明します。

この型による配列の外部形式における値間の区切り文字を示すために、`delimiter`で特定の文字を設定することができます。 デフォルトの区切り文字はカンマ (',') です。 この区切り文字は、配列要素の型に関係するものであり、配列型自体に関係するものでないことに注意してください。

省略可能な論理型の`collatable`パラメータが真の場合、`COLLATE`句を使用することによって、列定義と型の式は照合順序情報を持つことができます。 照合順序情報を実際に使用するかどうかは、型に対する操作を行う関数実装に任されています。 照合順序を設定可能な型を作成することにより、これが自動的に行われることはほとんどありません。

配列型

ユーザ定義型が作成されると、PostgreSQLは、自動的に、その要素型の名前の前にアンダースコアを付け、必要に応じて`NAMEDATALEN`長より短くなるように切り詰められた名前を持つ、関連する配列型を作成します。 (こうして付けられた名前が既存の型名と競合する場合、競合する名称がなくなるまでこの処理が繰り返されます。) この暗黙的に作成される配列型は可変長で、組み込み入出力関数`array_in`と`array_out`を使用します。 配列型はその要素となる型の所有者とスキーマのなんらかの変更に従い、また、要素となる型が削除された場合に削除されます。

理論的に考えると、「システムが自動的に配列型を正しく作成するのであれば、`ELEMENT` オプションはどうして存在するのだろうか」と不思議に思うでしょう。 `ELEMENT` が意味を持つ、唯一の場合は次のような条件を満たす固定長の型を作成する時です。 その条件とは、内部的に大量の同一の要素からなる配列となっていること、その配列に対して添字を指定して直接アクセスできること、加えて、今後作成する型全体に対する操作がどのようなものであっても、それらから直接アクセスできることです。例えば、`point` 型では、その構成要素である2つの浮動小数点にそれぞれ`point[0]`および`point[1]`を用いてアクセスすることができます。 この機能は、その内部形式が同一の固定長フィールドの正確な並びである、固定長の型でのみ動作することに注意してください。 添字による指定が可能な可変長型は、`array_in`と`array_out`を使用して、一般化された内部表現を持つ必要があります。 歴史的な理由 (明らかに間違いなのですが、変更するには遅すぎたため) により、固定長配列型への要素番号指定は0から始まり、可変長配列の場合は1から始まります。

パラメータ

`name`

作成するデータ型の名前です (スキーマ修飾名も可)。

`attribute_name`

複合型用の属性 (列) 名です。

`data_type`

複合型の列となる、既存のデータ型の名前です。

`collation`

複合型の列または範囲型に関連付けされる、既存の照合順序の名前です。

`label`

列挙型の1つの値に関連付けられるテキスト形式のラベルを表す、文字列リテラルです。

`subtype`

範囲型がその範囲の対象として表現する、要素型の名前です。

`subtype_operator_class`

派生元型のb-tree演算子クラスの名前です。

`canonical_function`

範囲型の正規化関数の名前です。

`subtype_diff_function`

派生元型の差異をとる関数の名前です。

`input_function`

指定された型のテキストによる外部形式のデータを内部形式に変換する関数の名前です。

output_function

指定された型の内部形式のデータをテキストによる外部形式に変換する関数の名前です。

receive_function

指定された型のバイナリによる外部形式のデータを内部形式に変換する関数の名前です。

send_function

指定された型の内部形式のデータをバイナリによる外部形式に変換する関数の名前です。

type_modifier_input_function

型に関する修正子の配列を内部形式に変換する関数の名前です。

type_modifier_output_function

内部形式の型修正子をテキストの外部形式に変換する関数の名前です。

analyze_function

指定したデータ型の統計情報解析を行う関数の名前です。

internallength

新しいデータ型の内部表現のバイト長を表す数値定数です。 デフォルトでは、可変長であるとみなされます。

alignment

データ型の格納整列条件です。 このオプションを指定する場合は、`char`、`int2`、`int4`、`double`のいずれかでなければなりません。 デフォルトは`int4`です。

storage

データ型の格納戦略です。 このオプションを指定する場合は、`plain`、`external`、`extended`、`main`のいずれかでなければなりません。 デフォルトは`plain`です。

like_type

新しい型に同じ表現を持たせる既存のデータ型の名前です。 *internallength*、*passedbyvalue*、*alignment*、*storage*の値が、この`CREATE TYPE`コマンドのどこかで明示的な指定により上書きされない限り、型から複製されます。

category

この型用のカテゴリコード（単一のASCII文字）です。 デフォルトは"ユーザ定義型"を表す'`U`'です。 他の標準カテゴリコードを[表45-51](#)に示します。 独自のカテゴリを作成するために他のASCII文字を選択することもできます。

preferred

この型がカテゴリ内で好まれる型である場合に真、さもなければ偽です。 デフォルトは偽です。 動作に予想外の変化を引き起こしますので既存の型カテゴリに新しく好まれる型を作成することには十分注意してください。

default

そのデータ型のデフォルト値です。 省略された場合、デフォルトはNULLです。

element

配列型を作成する場合、その配列の要素の型を指定します。

delimiter

このデータ型による配列で、値間の区切り文字として使われる文字です。

collatable

この型を操作する時に照合順序情報を使用することができる場合に真を取ります。 デフォルトは偽です。

注釈

一度作成したデータ型の使用には制限はありませんので、基本型または範囲型の作成は型定義で言及した関数の実行権をPUBLICに対して付与することと同じです。 この種の型定義において有用な関数では、これは通常問題になりません。 しかし、型を設計する前に、外部形式から、または、外部形式への変換を行う時に、その関数が"秘密の"情報を必要とするかどうか熟考してください。

PostgreSQLバージョン8.3より前のバージョンでは、生成される配列型の名前は常に要素型の名前の前に1つのアンダースコア文字（`_`）を付けたものになりました。（このため型の名前は他の名前よりも1文字短く制限されていました。）通常はこのように名付けられることは変わりありませんが、最大長の名前の場合やアンダースコアから始まるユーザ定義の型と競合する場合、配列型の名前は変わることがあります。このため、この変換に依存したコードを書くことは避けてください。代わりに、`pg_type.typarray`を使用して、指定した型に関した配列型を特定してください。

アンダースコアから始まる型やテーブル名の使用を避けることが賢明です。サーバは生成された配列型名称をユーザ指定の名前と競合しないように変更しますが、混乱する危険があります。特に古いクライアントソフトウェアを使用する場合、名前がアンダースコアから始まる型を常に配列を表すものと仮定しているかもしれません。

PostgreSQLバージョン8.2より前までは、シェル型を作成する`CREATE TYPE name`構文は存在しません。新規に基本型を作成する方法は、最初に入力関数を作成することでした。この方法では、PostgreSQLはまず新しいデータ型の名称を、入力関数の戻り値型から判断します。この場合、シェル型が暗黙的に作成され、残りの入出力関数の定義で参照することができます。この方法もまだ使用できますが、廃止予定で将来のリリースでなくなる可能性があります。また、関数定義における単純な打ち間違いの結果作成されるシェル型によって起こるカタログの混乱を防止するため、入力関数がCで作成された場合にのみこの方法によってシェル型が作成されます。

PostgreSQL 7.3より前のバージョンでは、関数の下位参照を、プレースホルダとなる疑似データ型である`opaque`型のデータ型名と置き換えることによって、shell型を作成することを完全に、慣習的に避けていました。また、7.3より前のバージョンでは、`cstring`型の引数および結果も`opaque`型として宣言する必要がありました。古いダンプファイルのロードをサポートするため、`CREATE TYPE`では`opaque`型を使用するよう宣言された入出力関数を受け入れます。しかし、注意を促すメッセージを表示し、正しいデータ型を使用するように関数の宣言を変更します。

例

次の例では、複合型を作成し、それを関数定義で使します。

```
CREATE TYPE compfoo AS (f1 int, f2 text);
CREATE FUNCTION getfoo() RETURNS SETOF compfoo AS $$
    SELECT fooid, fooname FROM foo
$$ LANGUAGE SQL;
```

次の例では、列挙型を作成し、それをテーブル定義に使します。

```
CREATE TYPE bug_status AS ENUM ('new', 'open', 'closed');

CREATE TABLE bug (
    id serial,
    description text,
    status bug_status
);
```

次の例では、範囲型を作成します。

```
CREATE TYPE float8_range AS RANGE (subtype = float8, subtype_diff = float8mi);
```

次の例では、`box`基本データ型を作成し、その型をテーブル定義の中で使しています。

```
CREATE TYPE box;

CREATE FUNCTION my_box_in_function(cstring) RETURNS box AS ... ;
CREATE FUNCTION my_box_out_function(box) RETURNS cstring AS ... ;

CREATE TYPE box (
    INTERNALLENGTH = 16,
    INPUT = my_box_in_function,
    OUTPUT = my_box_out_function
);

CREATE TABLE myboxes (
    id integer,
    description box
);
```

`box`型の内部構造が`float4`型が4つ並んだ配列の場合、このように書き換えることもできます。

```
CREATE TYPE box (
    INTERNALLENGTH = 16,
    INPUT = my_box_in_function,
    OUTPUT = my_box_out_function,
    ELEMENT = float4
);
```

このようにすると、`box`値の要素に要素番号でアクセスできます。その他は、上の例と同様の動作をします。

次の例では、ラージオブジェクト型を作成し、テーブル定義にてそれを使用します。

```
CREATE TYPE bigobj (  
    INPUT = lo_filein, OUTPUT = lo_fileout,  
    INTERNALLENGTH = VARIABLE  
);  
CREATE TABLE big_objs (  
    id integer,  
    obj bigobj  
);
```

その他の例は、[項35.11](#)を参照してください。ここには、入力関数、出力関数などを使った例があります。

互換性

複合型を作成する、最初のCREATE TYPE コマンドの構文は標準SQLに従います。他の構文はPostgreSQLの拡張です。標準SQLではまた他のCREATE TYPE 構文を定義していますが、PostgreSQLでは実装されていません。

ゼロ個の要素を持つ複合型を作成する機能は標準から派生したPostgreSQL固有のもの（CREATE TABLE の場合と同様）です。

関連項目

[ALTER TYPE](#), [CREATE DOMAIN](#), [CREATE FUNCTION](#), [DROP TYPE](#)

前のページ CREATE TRIGGER	ホーム 上に戻る	次のページ CREATE USER

CREATE USER

名前

CREATE USER -- 新しいデータベースロールを定義する

概要

```
CREATE USER name [ [ WITH ] option [ ... ] ]
```

ここで`option`は以下の通りです。

```
SUPERUSER | NOSUPERUSER
| CREATEDB | NOCREATEDB
| CREATEROLE | NOCREATEROLE
| CREATEUSER | NOCREATEUSER
| INHERIT | NOINHERIT
| LOGIN | NOLOGIN
| REPLICATION | NOREPLICATION
| CONNECTION LIMIT connlimit
| [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
| VALID UNTIL 'timestamp'
| IN ROLE role_name [, ...]
| IN GROUP role_name [, ...]
| ROLE role_name [, ...]
| ADMIN role_name [, ...]
| USER role_name [, ...]
| SYSID uid
```

説明

CREATE USER は [CREATE ROLE](#) の別名になりました。 唯一の違いは、CREATE USER という名前でコマンドが呼び出されると、デフォルトで LOGIN になり、CREATE ROLE という名前でコマンドが呼び出されると、デフォルトで NOLOGIN となる点です。

互換性

CREATE USER 文は、PostgreSQL の拡張です。 標準 SQL では、ユーザの定義は実装に任されています。

関連項目

[CREATE ROLE](#)

[前のページ](#)

CREATE TYPE

[ホーム](#)

[上に戻る](#)

[次のページ](#)

CREATE USER MAPPING

CREATE USER MAPPING

名前

CREATE USER MAPPING -- 外部サーバのユーザマップを新しく定義します。

概要

```
CREATE USER MAPPING FOR { user_name | USER | CURRENT_USER | PUBLIC }  
SERVER server_name  
[ OPTIONS ( option 'value' [ , ... ] ) ]
```

説明

CREATE USER MAPPINGは外部サーバとユーザの対応付けを定義します。ユーザマップは通常接続情報をカプセル化し、外部データラッパは外部データリソースにアクセスするためにこの情報と外部サーバによりカプセル化した情報を使用します。

外部サーバの所有者は任意のユーザに対するそのサーバ向けのユーザマップを作成することができます。また、サーバ上でUSAGE権限がユーザに付与されていた場合、ユーザは自身の持つユーザ名に対応するユーザマップを作成することができます。

パラメータ

user_name

外部サーバに対応付けされる既存のユーザ名です。CURRENT_USERとUSERは現在のユーザの名前に一致します。PUBLICが指定された場合、ユーザ指定がないマップが適用されたときに使用される公開マップと呼ばれるものが作成されます。

server_name

ユーザマップを作成する対象の既存のサーバの名前です。

OPTIONS (*option* 'value' [, ...])

この句はユーザマップのオプションを指定します。通常オプションはマップにおける実際のユーザ名とパスワードを定義します。オプション名は一意でなければなりません。許されるオプションの名前と値は、サーバの外部データラッパにより異なります。

例

ユーザbobとサーバfooとのユーザマップを作成します。

```
CREATE USER MAPPING FOR bob SERVER foo OPTIONS (user 'bob', password 'secret');
```

互換性

CREATE USER MAPPINGはISO/IEC 9075-9 (SQL/MED)に従います。

関連項目

[ALTER USER MAPPING](#), [DROP USER MAPPING](#), [CREATE FOREIGN DATA WRAPPER](#), [CREATE SERVER](#)

[前のページ](#)

CREATE USER

[ホーム](#)

[上に戻る](#)

[次のページ](#)

CREATE VIEW

CREATE VIEW

名前

CREATE VIEW -- 新しいビューを定義する

概要

```
CREATE [ OR REPLACE ] [ TEMP | TEMPORARY ] VIEW name [ ( column_name [, ...] ) ]  
    [ WITH ( view_option_name [= view_option_value] [, ...] ) ]  
    AS query
```

説明

CREATE VIEW は問い合わせによるビューを定義します。ビューは物理的な実体として存在するものではありません。その代わり、問い合わせでビューが参照される度に、指定された問い合わせが実行されます。

CREATE OR REPLACE VIEW も同様の働きをしますが、このコマンドでは、同じ名前のビューが既に存在している場合、そのビューを置き換えます。新しい問い合わせは、既存のビュー問い合わせが生成する列と同じ列(つまり、同じ順序の同じデータ型の同じ列名)を生成しなければなりません。しかし、そのリストの最後に列を追加しても構いません。出力列を生成する計算をまったく異なるものにしても構いません。

スキーマ名が付けられている場合(例えば、CREATE VIEW myschema.myview ...)、ビューは指定されたスキーマで作成されます。スキーマ名がなければ、そのビューは現在のスキーマで作成されます。一時ビューは特別なスキーマで作成されます。そのため、一時ビューを作成する時にはスキーマ名を付けることはできません。ビュー名は、同じスキーマ内の他のビュー、テーブル、シーケンス、インデックス、外部テーブルとは異なる名前である必要があります。

パラメータ

TEMPORARY or TEMP

これが指定された場合、ビューは一時ビューとして作成されます。現在のセッションが終わった時、一時ビューは自動的に削除されます。一時ビューが存在する間、現在のセッションでは、これと同じ名前の永続リレーションはスキーマで修飾した名前で参照していない限り不可視です。

ビューで参照されるテーブルの一部が一時テーブルであった場合、(TEMPORARY の指定があってもなくても)ビューは一時ビューとして作成されます。

name

作成するビューの名前です(スキーマ修飾名も可)。

column_name

ビューの列名として使用する名前のリストです。このパラメータは省略可能です。省略された場合、問い合わせに由来する名前が使用されます。

WITH (view_option_name [= view_option_value] [, ...])

この句はビュー用の省略可能なパラメータを指定します。現時点でサポートされるパラメータ名は security_barrier のみです。行レベルのセキュリティを提供することを意図したビューでは、これを有効にしなければなりません。詳細については [項37.4](#) を参照してください。

query

ビューの列と行を生成する [SELECT](#) または [VALUES](#) コマンドです。

注釈

現在のところ、ビューは読み取り専用です。システムは、ビューに対する挿入、更新、削除を許可しません。しかし、ビュー

に対する挿入などを他のテーブルに対する処理に適切に書き換える `INSTEAD` トリガを作成することで、更新可能なビューと同じように扱うことができます。 詳細については、[CREATE TRIGGER](#)を参照してください。 他にもルールを作成することでも実現できますが、実際のところトリガの方が理解しやすく正確に使用することができます。

ビューを削除するには、[DROP VIEW](#)文を使用してください。

ビューの列の名前と型は指定通りに割り当てられることに注意してください。 例えば、次のコマンドを見てください。

```
CREATE VIEW vista AS SELECT 'Hello World';
```

この例は、次の2点から好ましくありません。 まず、列の名前がデフォルトの `?column?` になります。さらに、列のデータ型もデフォルトの `unknown` になってしまいます。 ビューの結果として文字リテラルを返したい場合は、次のように指定してください。

```
CREATE VIEW vista AS SELECT text 'Hello World' AS hello;
```

ビューが参照するテーブルにアクセスできるかどうかは、ビューの所有者の権限で決定されます。 これは、場合によっては、背後のテーブルへのアクセスが制限されますが、安全性を提供するために使用することができます。 しかしすべてのビューが不正な改変に対して安全ではありません。 [項37.4](#)を参照してください。 ビュー内で実行される関数については、ビューを使用した問い合わせにおいて、その関数が直接呼び出された場合と同様に扱われます。 したがって、ビューを使用するユーザには、ビュー内で使用される全ての関数を呼び出す権限が必要です。

`CREATE OR REPLACE VIEW`が既存のビューに対して使用されると、ビューを定義するSELECTルールのみが変更されます。 所有者情報、権限、SELECT以外のルールなど他のビューの属性はそのまま変更されません。 置き換えるためにはビューの所有者（所有ロールのメンバである場合も含む）でなければなりません。

例

全てのコメディ映画（Comedy films）からなるビューを作成します。

```
CREATE VIEW comedies AS
SELECT *
FROM films
WHERE kind = 'Comedy';
```

これはビューを作成した時点で `film` テーブル内にある列を持つビューを作成します。 ビューを作成するために `*` が使用されていますが、その後にテーブルに追加された列はビューには含まれません。

互換性

標準SQLの `CREATE VIEW` 文には、さらにいくつかの機能が規定されています。

```
CREATE VIEW name [ ( column_name [, ...] ) ]
AS query
[ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

このSQLコマンドの全てのオプション句について、次に説明します。

`CHECK OPTION`

更新される可能性のあるビューには、このオプションを使用しなければなりません。 ビューに対する全ての `INSERT` と `UPDATE` コマンドは、ビュー定義の条件を満足しているかどうか（新しいデータがビューを通して可視であるかどうか）検査されます。 条件を満たしていない場合、その更新は拒否されます。

`LOCAL`

このビューの整合性を検査します。

`CASCADED`

このビューと依存している全てのビューの整合性を検査します。 `CASCADED` または `LOCAL` のいずれも指定されていない場合は、`CASCADED` として扱われます。

`CREATE OR REPLACE VIEW` は PostgreSQL の言語拡張です。 つまり一時ビューという概念も言語拡張です。

関連項目

[ALTER VIEW](#), [DROP VIEW](#)

DEALLOCATE

名前

DEALLOCATE -- 準備された文の割り当てを解除する

概要

```
DEALLOCATE [ PREPARE ] { name | ALL }
```

説明

DEALLOCATE を使用して、過去に準備されたSQL文の割り当てを解除します。 準備された文を明示的に割り当て解除しなかった場合、セッションが終了した時に割り当てが解除されます。

準備された文に関する詳細については[PREPARE](#)を参照してください。

パラメータ

PREPARE

このキーワードは無視されます。

name

割り当てを解除する、準備された文の名前です。

ALL

準備された文の割り当てをすべて解除します

互換性

DEALLOCATE 文は標準SQLにもありますが、埋め込みSQLでの使用のみに用途が限定されています。

関連項目

[EXECUTE](#), [PREPARE](#)

[前のページ](#)

CREATE VIEW

[ホーム](#)

[上に戻る](#)

[次のページ](#)

DECLARE

DECLARE

名前

DECLARE -- カーソルを定義する

概要

```
DECLARE name [ BINARY ] [ INSENSITIVE ] [ [ NO ] SCROLL ]  
CURSOR [ { WITH | WITHOUT } HOLD ] FOR query
```

説明

DECLAREを使うと、巨大な問い合わせの結果から一度に少数の行を取り出す機能であるカーソルが使用できるようになります。カーソルを作成した後、[FETCH](#)を使用して行を取り出します。

注意: このマニュアルページではSQLコマンドレベルでのカーソルの使用方法について説明します。 PL/pgSQL内でカーソルを使用するつもりであれば、規則が異なりますので、[項39.7](#)を参照してください。

パラメータ

name

作成されるカーソルの名前です。

BINARY

カーソルによるデータの取得が、テキスト形式ではなくバイナリ形式になります。

INSENSITIVE

カーソルから取り出されたデータが、カーソルを作成した後に行われた背後にあるテーブルの更新の影響を受けないことを示します。しかし、これはPostgreSQLのデフォルトの動作です。したがって、このキーワードを使用しても効果はなく、このキーワードは標準SQLとの互換性を保持するために存在しています。

SCROLL
NO SCROLL

SCROLLは、そのカーソルから通常の順序通りでない方法で（例えば後方から）行を取り出可能であることを指定します。問い合わせの実行計画が複雑になると、SCROLLの指定によって問い合わせの実行時間が増大する可能性があります。NO SCROLLは、そのカーソルから順序通りでない方法では行を取り出せないことを指定します。デフォルトでは、いくつかの場合でスクロール可能です。これはSCROLLの指定と同じではありません。詳細は[注釈](#)を参照してください。

WITH HOLD
WITHOUT HOLD

WITH HOLDは、カーソルを生成したトランザクションが正常にコミット処理を行った後も、そのカーソルの使用を続けられることを指定します。WITHOUT HOLDは、カーソルを生成したトランザクションの外部では、そのカーソルを使用できないことを指定します。WITH HOLDもWITHOUT HOLDも指定されない場合、WITHOUT HOLDがデフォルトとなります。

query

カーソルによって返される行を提供する[SELECT](#)または[VALUES](#)コマンドです。

BINARY、INSENSITIVE、SCROLLキーワードは任意の順番で指定することができます。

注釈

通常のカーソルは、SELECTの出力と同じテキスト形式でデータを返します。BINARYは、カーソルがバイナリ形式でデータを返さなければならないことを示します。これによりサーバ、クライアントの両方で変換に関する作業を省くことができますが、プラットフォームに依存するバイナリデータ書式を扱うためのプログラムの作業が大きくなります。例えば、問い合わせが整数の

列から値として1を返す場合、デフォルトのカーソルからは1という文字列を取得することになりますが、バイナリ形式のカーソルからは、内部表現を使った4バイトの値を取得することになります（バイトオーダはビッグエンディアン）。

バイナリ形式のカーソルは注意して使わなければなりません。psqlなどの多くのアプリケーションは、データはテキスト形式で返されるものとみなしており、バイナリ形式のカーソルを扱うことができません。

注意: クライアントアプリケーションが"拡張問い合わせ"プロトコルを使用してFETCHコマンドを発行する場合、テキスト形式とバイナリ形式のどちらでデータを受け取るのかは、バインドプロトコルメッセージで指定します。この選択は、カーソル定義での指定を上書きします。全てのカーソルをテキスト形式/バイナリ形式のどちらでも扱うことができる拡張問い合わせプロトコルでは、バイナリカーソルという概念は旧式なものです。

WITH HOLDが指定されなければ、このコマンドで生成されるカーソルは現在のトランザクションの中でのみ使用することができます。したがって、WITH HOLDのないDECLAREはトランザクションブロックの外側では意味がありません。その場合、カーソルはこの文が完了するまでのみ有効です。そのため、PostgreSQLはトランザクションブロックの外部でこうしたコマンドが使用された場合エラーを報告します。トランザクションブロックを定義するには、[BEGIN](#)と[COMMIT](#)（または[ROLLBACK](#)）を使用してください。

WITH HOLDが指定された場合、カーソルを作成したトランザクションのコミットに成功していても、同一セッション内のその後のトランザクションからそのカーソルにアクセスすることができます（ただし、トランザクションがアボートされた場合、そのカーソルは削除されます）。WITH HOLD付きで作成されたカーソルは、そのカーソルに対して明示的なCLOSEが発行された場合やセッションが終了した時に閉じられます。現在の実装では、保持されたカーソルを使って表される行は、その後のトランザクションでも利用できるように、一時ファイルかメモリ領域にコピーされます。

問い合わせがFOR UPDATEまたはFOR SHAREを含む場合、WITH HOLDを指定することはできません。

カーソルから逆方向にデータを取り出す時には、SCROLLオプションを指定しなければなりません。これは標準SQLでは必須となっています。しかし、以前のバージョンとの互換性を保持するために、PostgreSQLでは、カーソルの問い合わせ計画が単純であり、そのサポートに余計なオーバーヘッドが必要ない場合、SCROLLなしでも逆方向にデータを取り出すことができます。しかし、SCROLLを付けなくても逆方向にデータが取り出せることを利用してアプリケーションを開発するのはお勧めしません。NO SCROLLを指定した場合は、どのような場合でも逆方向に取り出すことはできません。

また、問い合わせがFOR UPDATEまたはFOR SHAREを含む場合は、逆方向の取り出しは許されません。このためこの場合はSCROLLを指定することはできません。

注意

スクロール可能なWITH HOLDカーソルが揮発関数（[項35.6](#)参照）を含む場合、想定しない結果をもたらす可能性があります。これまで取り出した行を再度取り出した時、関数は再実行される可能性があり、この場合おそらく初回と異なる結果をもたらします。こうした問題の回避方法の1つとして、カーソルをWITH HOLDと宣言し、そこから何か行を読み取る前にトランザクションをコミットすることがあります。これにより強制的にカーソルの出力全体が一時領域に具現化され、揮発関数は各行に対して1度しか実行されなくなります。

問い合わせがFOR UPDATEまたはFOR SHAREを含む場合、通常のこのオプションを持つSELECTコマンドと同様、返される行は取り出した時点でロックされます。さらに、返される行はもっとも最新のバージョンになります。したがって、このオプションは、標準SQLで"センシティブカーソル"と呼ばれるものと同じ機能を提供します。（[INSENSITIVE](#)をFOR UPDATEまたはFOR SHAREといっしょに指定するとエラーになります。）

注意

カーソルをUPDATE ... WHERE CURRENT OFまたはDELETE ... WHERE CURRENT OFで使用するつもりならば、FOR UPDATEの使用を通常勧めます。FOR UPDATEを使用することで、取り出し中および更新中に他のセッションが行を変更することを防止します。FOR UPDATEがなければ、カーソル作成後に行が変更された場合に後に行うWHERE CURRENT OFコマンドは効果がなくなります。

FOR UPDATEを使用する他の理由は、"簡単に更新可能"にするためにカーソル問い合わせが標準SQLに合わない場合（具体的にはカーソルは1つのテーブルのみを参照しなければならず、また、グループ化やORDER BYを使用してはならない）、これがないと後に実行されるWHERE CURRENT OFが失敗するかもしれないことです。計画選択の詳細によっては、簡単に更新可能でないカーソルは動作するかもしれませんが、動作しないかもしれません。このため最悪の場合、アプリケーションは試験時に動作するが、運用時に失敗するかもしれません。

FOR UPDATEをWHERE CURRENT OFといっしょに使用しない大きな理由は、カーソルをスクロール可能にする必要がある、または後の更新の影響を受けないようにする（つまり古いデータを表示し続けるようにする）必要がある場合のためです。これが必要ならば、上記の警告に十分注意してください。

標準SQLでは、組み込みSQLにおけるカーソルのみが規定されています。 PostgreSQLサーバはカーソル用のOPEN文を実装していません。 カーソルは宣言された時に開いたものとみなされています。 しかし、PostgreSQL用の埋め込みSQLプリプロセッサであるECPGでは、DECLAREとOPEN文を含め、標準SQLのカーソル規定をサポートしています。

[pg_cursors](#) システムビューを問い合わせることで、利用可能なすべてのカーソルを確認することができます。

例

カーソルを宣言します。

```
DECLARE liahona CURSOR FOR SELECT * FROM films;
```

カーソル使用のより多くの例については[FETCH](#)を参照してください。

互換性

標準SQLでは、デフォルトでカーソルが背後にあるデータの同時実行更新を気づかなければならないかどうかは実装依存であると述べています。 PostgreSQLのカーソルはデフォルトでは気づかず、FOR UPDATEを指定することで気づくことができます。 他の製品では異なる動作をするかもしれません。

標準SQLでは、カーソルを埋め込みSQL内とモジュール内でのみ使用できます。 PostgreSQLでは、対話式にカーソルを使うことができます。

バイナリカーソルはPostgreSQLの拡張です。

関連項目

[CLOSE](#), [FETCH](#), [MOVE](#)

[前のページ](#)
DEALLOCATE

[ホーム](#)
[上に戻る](#)

[次のページ](#)
DELETE

DELETE

名前

DELETE -- テーブルから行を削除する

概要

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
DELETE FROM [ ONLY ] table_name [ * ] [ [ AS ] alias ]
    [ USING using_list ]
    [ WHERE condition | WHERE CURRENT OF cursor_name ]
    [ RETURNING * | output_expression [ [ AS ] output_name ] [, ...] ]
```

説明

DELETEは、指定したテーブルからWHERE句を満たす行を削除します。WHEREがない場合、指定したテーブルの全ての行を削除することになります。この結果、そのテーブルは存在するが中身が空のテーブルになります。

ティップ: PostgreSQLによる拡張であるTRUNCATEは、より高速に、テーブルから全ての行を削除する仕組みを持っています。

データベース内のほかのテーブルに含まれる情報を用いてテーブル内の行を削除する方法には、副問い合わせとUSING句で追加テーブルを指定する方法という2つがあります。どちらの技法が適切かはその状況によります。

省略可能なRETURNING句により、DELETEは実際に削除された各行に基づいて計算された値を返すようになります。そのテーブルの列、USINGで指定された他のテーブルの列、あるいは、その両方を使用した式を計算することができます。RETURNINGリストの構文はSELECTの出力リストと同一です。

削除を実行するには、そのテーブルのDELETE権限が必要です。また、USING句内のテーブルに対するSELECT権限、および、conditionで使用する値を読み取るために、その値が含まれるテーブルに対するSELECT権限も必要です。

パラメータ

with_query

WITH句によりDELETE問い合わせ内で名前で参照可能な1つ以上の副問い合わせを指定することができます。[項7.8](#)と[SELECT](#)を参照してください。

table_name

既存のテーブル名です（スキーマ修飾名も可）。テーブル名の前にONLYが指定された場合、そのテーブルのみで一致する行が削除されます。ONLYが指定されていない場合、そのテーブルおよび（もしあれば）そのテーブルを継承する全てのテーブルから一致する行が削除されます。省略することもできますが、テーブル名の後に*を指定することで、明示的に継承するテーブルも含まれることを示すことができます。

alias

対象テーブルの代理名称です。別名が与えられた場合、実際のテーブル名は完全に隠蔽されます。たとえば、DELETE FROM foo AS fを考えます。このDELETE文の残りの部分ではこのテーブルをfooではなくfとして参照しなければなりません。

using_list

WHERE条件内に他のテーブルの列を許可するための、テーブル式のリストです。これは、SELECT文のFROM句において指定できるテーブルのリストと同様です。例えば、テーブル名の別名も指定することができます。自己結合を設定する目的以外ではusing_listに対象のテーブルを繰り返さないでください。

condition

boolean型の値を返す式です。この式がtrueを返す行のみが削除されます。

cursor_name

`WHERE CURRENT OF` 条件で使用されるカーソルの名前です。削除対象の行は、そのカーソルからもっとも最近に取り出される行です。カーソルは、`DELETE` の対象テーブルに対するグループ化のない問い合わせでなければなりません。`WHERE CURRENT OF` を論理条件といっしょに指定することはできません。`WHERE CURRENT OF` 付きのカーソルの使用に関する情報については[DECLARE](#)を参照してください。

output_expression

各行を削除した後に計算され、`DELETE` によって返される式です。この式には、*table_name* で指名したテーブルまたは `USING` で指定したテーブル（複数可）の任意の列名を使用することができます。すべての列を返す場合は `*` と記載してください。

output_name

返される列で使用される名前です。

出力

正常に終了した場合、`DELETE` コマンドは以下の形式のコマンドタグを返します。

```
DELETE count
```

count は削除した行数です。この数は、`BEFORE DELETE` トリガによって削除が抑止された場合、*condition* に合致した行より少なくなる可能性があることに注意してください。*count* が 0 の場合、*condition* を満たす行が存在しなかったことを示します（これはエラーとはみなされません）。

`DELETE` コマンドが `RETURNING` 句を持つ場合、その結果は、`RETURNING` リストで定義した列と値を持ち、そのコマンドで削除された行全体に対して計算を行う `SELECT` 文の結果と似たものになるでしょう。

注釈

PostgreSQL では、`USING` 句で他のテーブルを指定することで、`WHERE` 条件内で他のテーブルを参照することができます。例えば、指定したプロデューサーが製作した全ての映画を削除する時は、次のようなコマンドを実行します。

```
DELETE FROM films USING producers
WHERE producer_id = producers.id AND producers.name = 'foo';
```

ここでは、`films` と `producers` とを結合して、`films` 行に削除用の印を付けるという作業を行っています。この構文は標準に従ったものではありません。より標準的な方法は以下の通りです。

```
DELETE FROM films
WHERE producer_id IN (SELECT id FROM producers WHERE name = 'foo');
```

副問い合わせ形式より結合形式の方が書き易く、実行が速くなることがあります。

例

`films` テーブルから `musical` 以外を全て削除します。

```
DELETE FROM films WHERE kind <> 'Musical';
```

`films` テーブルを空にします。

```
DELETE FROM films;
```

完了した作業（`status` が `DONE`）を削除し、削除された行を完全に返します。

```
DELETE FROM tasks WHERE status = 'DONE' RETURNING *;
```

`tasks` において `c_tasks` カーソルが現在位置している行を削除します。

```
DELETE FROM tasks WHERE CURRENT OF c_tasks;
```

互換性

このコマンドは標準 SQL に準拠しています。ただし、`USING` 句と `RETURNING` 句は PostgreSQL の拡張です。`DELETE` で `WITH` が使用可能であることも同様に拡張です。

DISCARD

名前

DISCARD -- セッションの状態を破棄する

概要

```
DISCARD { ALL | PLANS | TEMPORARY | TEMP }
```

説明

`DISCARD` はデータベースセッションに関連した内部リソースを解放します。通常これらのリソースはセッション終了時に解放されます。

`DISCARD TEMP` は現在のセッションで作成された一時テーブルをすべて削除します。`DISCARD PLANS` は内部で保管された問い合わせ計画をすべて削除します。`DISCARD ALL` は、一時的なリソースを破棄し、セッションにおける局所的な設定変更を初期化することで、元々の状態に初期化します。

パラメータ

`TEMPORARY` or `TEMP`

現在のセッションで作成された一時テーブルをすべて削除します。

`PLANS`

保管された問い合わせ計画をすべて解放します。

`ALL`

現在のセッションに関連付いた一時的なリソースを解放し、セッションを初期状態に戻します。現時点でこれは、以下に示す一連の文を実行することと同じ効果です。

```
SET SESSION AUTHORIZATION DEFAULT;
RESET ALL;
DEALLOCATE ALL;
CLOSE ALL;
UNLISTEN *;
SELECT pg_advisory_unlock_all();
DISCARD PLANS;
DISCARD TEMP;
```

注釈

`DISCARD ALL` をトランザクションブロック内で実行することはできません。

互換性

`DISCARD` は PostgreSQL の拡張です。

DO

名前

DO -- 無名コードブロックを実行します。

概要

```
DO [ LANGUAGE lang_name ] code
```

説明

DOは無名コードブロック、言い換えると、手続き言語内の一時的な無名関数を実行します。

コードブロックはあたかもパラメータを取らずにvoidを返す関数の本体かのように扱われます。これは解析され、一回実行されます。

省略可能なLANGUAGE句をコードブロックの前または後ろにつけることができます。

パラメータ

code

実行される手続き言語のコードです。これは、CREATE FUNCTIONの場合と同様、文字列リテラルとして指定しなければなりません。ドル記号による引用符付けの使用を勧めます。

lang_name

コードの作成に使用する手続き言語の名前です。省略時のデフォルトはplpgsqlです。

注釈

使用される手続き言語は常に、CREATE LANGUAGEを使用して現在のデータベースにインストールされていなければなりません。plpgsqlはデフォルトでインストールされますが、他の言語はインストールされません。

ユーザは手続き言語に対するUSAGE権限を持たなければなりません。また、言語が信用できない場合はスーパーユーザでなければなりません。これは、その言語における関数作成に必要な権限と同じです。

例

publicスキーマ内のすべてのビューに対するすべての権限をwebuserロールに付与します。

```
DO $$DECLARE r record;
BEGIN
  FOR r IN SELECT table_schema, table_name FROM information_schema.tables
    WHERE table_type = 'VIEW' AND table_schema = 'public'
  LOOP
    EXECUTE 'GRANT ALL ON ' || quote_ident(r.table_schema) || '.' || quote_ident(r.table_name) || ' TO
webuser';
  END LOOP;
END$$;
```

互換性

標準SQLにはDO文はありません。

DROP AGGREGATE

名前

DROP AGGREGATE -- 集約関数を削除する

概要

```
DROP AGGREGATE [ IF EXISTS ] name ( argtype [ , ... ] ) [ CASCADE | RESTRICT ]
```

説明

DROP AGGREGATEを実行すると、既存の集約関数定義を削除することができます。このコマンドを実行するには、現在のユーザがその集約関数を所有している必要があります。

パラメータ

IF EXISTS

集約が存在しない場合でもエラーになりません。この場合注意メッセージが発行されます。

name

既存の集約関数の名前です（スキーマ修飾名も可）。

argtype

集約関数の操作対象となる入力データ型です。引数を持たない関数を参照する場合は、入力データ型の一覧の場所に*を記述してください。

CASCADE

その集約関数に依存しているオブジェクトを自動的に削除します。

RESTRICT

依存しているオブジェクトがある場合、その集約関数の削除要求を拒否します。こちらがデフォルトです。

例

integer型のmyavg集約関数を削除します。

```
DROP AGGREGATE myavg(integer);
```

互換性

標準SQLには、DROP AGGREGATE文はありません。

関連項目

[ALTER AGGREGATE](#), [CREATE AGGREGATE](#)

DROP CAST

名前

DROP CAST -- キャストを削除する

概要

```
DROP CAST [ IF EXISTS ] (source_type AS target_type) [ CASCADE | RESTRICT ]
```

説明

DROP CAST は、以前に定義したキャストを削除します。

キャストを削除するには、変換元または変換先のデータ型を所有する必要があります。これらは、キャストを作成するために必要な権限と同じです。

パラメータ

IF EXISTS

キャストが存在しない場合でもエラーにはなりません。この場合注意メッセージが発行されます。

source_type

キャストする変換元のデータ型の名前です。

target_type

キャストする変換先のデータ型の名前です。

CASCADE
RESTRICT

キャストは依存関係を持たないため、これらのキーワードを指定しても効果はありません。

例

text 型から int 型へのキャストを削除します。

```
DROP CAST (text AS int);
```

互換性

DROP CAST コマンドは、標準SQLに準拠しています。

関連項目

[CREATE CAST](#)

[前のページ](#)

DROP AGGREGATE

[ホーム](#)
[上に戻る](#)

[次のページ](#)

DROP COLLATION

DROP COLLATION

名前

DROP COLLATION -- 照合順序を削除

概要

```
DROP COLLATION [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

説明

DROP COLLATION は事前に定義された照合順序を削除します。 照合順序を削除するためには、その照合順序を所有していなければなりません

パラメータ

IF EXISTS

照合順序が存在しない場合でもエラーになりません。 この場合注意メッセージが発行されます。

name

照合順序の名前です。照合順序名はスキーマ修飾可能です。

CASCADE

照合順序に依存するオブジェクトを自動的に削除します。

RESTRICT

依存するオブジェクトが存在する場合、照合順序の削除を拒絶します。 これがデフォルトです。

例

german という名前の照合順序を削除します。

```
DROP COLLATION german;
```

互換性

PostgreSQLの拡張であるIF EXISTS オプションを除き、DROP COLLATION コマンドは標準SQLに従います。

関連項目

[ALTER COLLATION](#), [CREATE COLLATION](#)

DROP CONVERSION

名前

DROP CONVERSION -- 変換を削除する

概要

```
DROP CONVERSION [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

説明

DROP CONVERSION を使用すると、以前に定義した変換を削除できます。変換を削除するには、その変換を所有していなければなりません。

パラメータ

IF EXISTS

変換が存在しない場合でもエラーになりません。この場合注意メッセージが発行されます。

name

変換の名前です。変換名はスキーマ修飾可能です。

CASCADE
RESTRICT

変換は依存関係を持たないため、これらのキーワードを指定しても効果はありません。

例

myname という名前の変換を削除します。

```
DROP CONVERSION myname;
```

互換性

標準SQLにはDROP CONVERSION 文はありません。しかし、PostgreSQLのCREATE CONVERSION に似たCREATE TRANSLATION 文と
いっしょにDROP TRANSLATION 文があります。

関連項目

[ALTER CONVERSION](#), [CREATE CONVERSION](#)

[前のページ](#)

DROP COLLATION

[ホーム](#)
[上に戻る](#)

[次のページ](#)

DROP DATABASE

DROP DATABASE

名前

DROP DATABASE -- データベースを削除する

概要

```
DROP DATABASE [ IF EXISTS ] name
```

説明

`DROP DATABASE`は、データベースを削除します。そのデータベースの項目をカタログから削除し、データを保存していたディレクトリを削除します。データベースの所有者のみがこのコマンドを実行することができます。また、このコマンドを実行しようとしているユーザ自身や他のユーザが、対象とするデータベースに接続している時は実行できません（このコマンドを実行する時は、`postgres`や他のデータベースに接続してください）。

`DROP DATABASE`は元に戻すことができません。十分注意して使用してください。

パラメータ

`IF EXISTS`

データベースが存在しない場合でもエラーになりません。この場合注意メッセージが発行されます。

name

削除するデータベースの名前です。

注釈

`DROP DATABASE`はトランザクションブロックの内部では実行できません。

対象とするデータベースに接続している間は、このコマンドを実行することができません。したがって、このコマンドのラッパである[dropdb](#)プログラムを使用する方がより便利であるかもしれません。

互換性

標準SQLには`DROP DATABASE`文はありません。

関連項目

[CREATE DATABASE](#)

[前のページ](#)

DROP CONVERSION

[ホーム](#)

[上に戻る](#)

[次のページ](#)

DROP DOMAIN

DROP DOMAIN

名前

DROP DOMAIN -- ドメインを削除する

概要

```
DROP DOMAIN [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

説明

DROP DOMAIN はドメインを削除します。 ドメインを削除できるのは、ドメインの所有者のみです。

パラメータ

IF EXISTS

ドメインが存在しない場合でもエラーにはなりません。 この場合注意メッセージが発行されます。

name

既存のドメインの名前です（スキーマ修飾名も可）。

CASCADE

ドメインに依存するオブジェクト（テーブルの列など）を自動的に削除します。

RESTRICT

依存しているオブジェクトがある場合、そのドメインの削除要求を拒否します。 これがデフォルトです。

例

box ドメインを削除します。

```
DROP DOMAIN box;
```

互換性

このコマンドは、PostgreSQLの拡張であるIF EXISTS オプションを除き、標準SQLに準拠しています。

関連項目

[CREATE DOMAIN](#), [ALTER DOMAIN](#)

[前のページ](#)

DROP DATABASE

[ホーム](#)

[上に戻る](#)

[次のページ](#)

DROP EXTENSION

DROP EXTENSION

名前

DROP EXTENSION -- 拡張を削除

概要

```
DROP EXTENSION [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

説明

DROP EXTENSION はデータベースから拡張を削除します。 拡張を削除すると、その構成オブジェクトも削除されます。

DROP EXTENSION を使用するためにはその拡張を所有していなければなりません。

パラメータ

IF EXISTS

拡張が存在しない場合でもエラーになりません。 この場合注意メッセージが発行されます。

name

インストールされている拡張の名前です。

CASCADE

削除する拡張に依存しているオブジェクトを自動的に削除します。

RESTRICT

依存しているオブジェクト（それ自身が所有するメンバオブジェクトと同じDROP コマンドで指定された他の拡張以外）がある場合に、拡張の削除を拒否します。 こちらがデフォルトです。

例

現在のデータベースからhstore拡張を削除します。

```
DROP EXTENSION hstore;
```

例えば何らかのテーブルがhstore型の列を持つなど、データベース内でhstoreのオブジェクトを使用している場合、このコマンドは失敗します。 こうした依存するオブジェクトがある場合でも強制的に削除するためにはCASCADEを付けてください。

互換性

DROP EXTENSION はPostgreSQLの拡張です。

関連項目

[CREATE EXTENSION](#), [ALTER EXTENSION](#)

DROP FOREIGN DATA WRAPPER

名前

DROP FOREIGN DATA WRAPPER -- 外部データラッパを削除する

概要

```
DROP FOREIGN DATA WRAPPER [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

説明

DROP FOREIGN DATA WRAPPER は既存の外部データラッパを削除します。 このコマンドを実行するためには、現在のユーザは外部データラッパの所有者でなければなりません。

パラメータ

IF EXISTS

外部データラッパが存在しない場合にエラーを発生しません。 この場合警告が発行されます。

name

既存の外部データラッパの名前です。

CASCADE

外部データラッパに依存するオブジェクト（サーバなど）を自動的に削除します。

RESTRICT

外部データラッパに依存するオブジェクトが存在する場合に削除を取りやめます。 これがデフォルトです。

例

外部データラッパ`dbi`を削除します。

```
DROP FOREIGN DATA WRAPPER dbi;
```

互換性

DROP FOREIGN DATA WRAPPER は ISO/IEC 9075-9 (SQL/MED) に従います。 IF EXISTS 句は PostgreSQL の拡張です。

関連項目

[CREATE FOREIGN DATA WRAPPER](#), [ALTER FOREIGN DATA WRAPPER](#)

[前のページ](#)

DROP EXTENSION

[ホーム](#)

[上に戻る](#)

[次のページ](#)

DROP FOREIGN TABLE

DROP FOREIGN TABLE

名前

DROP FOREIGN TABLE -- 外部テーブルを削除

概要

```
DROP FOREIGN TABLE [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

説明

DROP FOREIGN TABLE は外部テーブルを削除します。 外部テーブルの所有者のみが削除することができます。

パラメータ

IF EXISTS

外部テーブルが存在しない場合でもエラーになりません。 この場合注意メッセージが発行されます。

name

削除する外部テーブルの名前です（スキーマ修飾名も可）。

CASCADE

削除する外部テーブルに依存しているオブジェクト（ビューなど）を自動的に削除します。

RESTRICT

依存しているオブジェクトがある場合に、外部テーブルの削除を拒否します。 こちらがデフォルトです。

例

films および distributors という 2 つの外部テーブルを破棄します。

```
DROP FOREIGN TABLE films, distributors;
```

互換性

このコマンドは、ISO/IEC 9075-9 (SQL/MED) では 1 つのコマンドで 1 つの外部テーブルのみを削除できるという点を除き、および、PostgreSQL の拡張である IF EXISTS オプションを除き、この標準に従います。

関連項目

[ALTER FOREIGN TABLE](#), [CREATE FOREIGN TABLE](#)

[前のページ](#)

DROP FOREIGN DATA WRAPPER

[ホーム](#)

[上に戻る](#)

[次のページ](#)

DROP FUNCTION

DROP FUNCTION

名前

DROP FUNCTION -- 関数を削除する

概要

```
DROP FUNCTION [ IF EXISTS ] name ( [ [ argmode ] [ argname ] argtype [, ...] ] )
[ CASCADE | RESTRICT ]
```

説明

DROP FUNCTIONは既存の関数定義を削除します。 このコマンドを実行できるのは、その関数の所有者のみです。 関数の引数の型は必ず指定しなければなりません。 異なる引数を持つ同じ名前の関数が複数存在する可能性があるからです。

パラメータ

IF EXISTS

関数が存在しない場合でもエラーになりません。 この場合注意メッセージが発行されます。

name

既存の関数の名前です（スキーマ修飾名も可）。

argmode

引数のモードで、IN、OUT、INOUT、VARIADICのいずれかです。 省略された場合のデフォルトはINです。 関数の識別を行うには入力引数のみが必要です。 実際にはDROP FUNCTIONがOUT引数を見捨てることに注意してください。 です。 IN、INOUT、およびVARIADIC引数を列挙することで十分です。

argname

引数の名前です。 関数の識別を行うには引数のデータ型のみが必要です。 実際にはDROP FUNCTIONは引数の名前を見捨てることに注意してください。

argtype

もしあれば、その関数の引数のデータ型（複数可）です。 （スキーマで修飾可能です）

CASCADE

関数に依存するオブジェクト（演算子やトリガなど）を自動的に削除します。

RESTRICT

依存しているオブジェクトがある場合、その関数の削除を拒否します。 これがデフォルトです。

例

次のコマンドは平方根関数を削除します。

```
DROP FUNCTION sqrt(integer);
```

互換性

DROP FUNCTION文は標準SQLでも定義されていますが、このコマンドとは互換性がありません。

関連項目

[CREATE FUNCTION](#), [ALTER FUNCTION](#)

[前のページ](#)

DROP FOREIGN TABLE

[ホーム](#)

[上に戻る](#)

[次のページ](#)

DROP GROUP

DROP GROUP

名前

DROP GROUP -- データベースロールを削除する

概要

```
DROP GROUP [ IF EXISTS ] name [, ...]
```

説明

DROP GROUP は [DROP ROLE](#) の別名になりました。

互換性

標準SQLには DROP GROUP 文はありません。

関連項目

[DROP ROLE](#)

[前のページ](#)

DROP FUNCTION

[ホーム](#)
[上に戻る](#)

[次のページ](#)
DROP INDEX

DROP INDEX

名前

DROP INDEX -- インデックスの削除

概要

```
DROP INDEX [ CONCURRENTLY ] [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

説明

DROP INDEX はデータベースシステムから既存のインデックスを削除します。 このコマンドを実行するには、そのインデックスを所有していなければなりません。

パラメータ

CONCURRENTLY

インデックスのテーブルに対して同時に実行される選択、挿入、更新、削除をロックすることなくインデックスを削除します。 通常のDROP INDEX ではテーブルに対する排他ロックを獲得し、インデックスの削除が完了するまで他のアクセスをブロックします。 このオプションを使うと、競合するトランザクションが完了するまでコマンドは待たされます。

このオプションを使用する時に注意すべき、複数の警告があります。 指定できるインデックス名は1つだけであり、また、CASCADE オプションはサポートされません。 （したがってUNIQUE またはPRIMARY KEY をサポートするインデックスをこの方法で削除することはできません。） また、通常のDROP INDEX はトランザクションブロックの中で行うことができますが、DROP INDEX CONCURRENTLY ではできません。

IF EXISTS

インデックスが存在しない場合でもエラーになりません。 この場合注意メッセージが発行されます。

name

削除するインデックスの名前です（スキーマ修飾名も可）。

CASCADE

そのインデックスに依存しているオブジェクトを自動的に削除します。

RESTRICT

依存しているオブジェクトがある場合、そのインデックスの削除を拒否します。 こちらがデフォルトです。

例

次のコマンドはtitle_idx インデックスを削除します。

```
DROP INDEX title_idx;
```

互換性

DROP INDEX はPostgreSQLの言語拡張です。 標準SQLにはインデックスに関する規定はありません。

関連項目

[CREATE INDEX](#)

DROP LANGUAGE

名前

DROP LANGUAGE -- 手続き言語の削除

概要

```
DROP [ PROCEDURAL ] LANGUAGE [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

説明

DROP LANGUAGEは過去に登録された手続き言語の定義を削除します。 DROP LANGUAGEを使用するにはスーパーユーザか言語の所有者でなければなりません。

注意: PostgreSQL 9.1からほとんどの手続き言語は"拡張"にまとめられましたので、DROP LANGUAGEではなく[DROP EXTENSION](#)を使用して削除すべきです。

パラメータ

IF EXISTS

言語が存在しない場合でもエラーになりません。 この場合注意メッセージが発行されます。

name

既存の手続き言語の名前です。 後方互換性を保持するため、この名前は単一引用符で囲むことができます。

CASCADE

その言語に依存するオブジェクト（その言語で記述された関数など）を自動的に削除します。

RESTRICT

依存しているオブジェクトがある場合、その言語の削除を拒否します。 こちらがデフォルトです。

例

次のコマンドはplsampleという手続き言語を削除します。

```
DROP LANGUAGE plsample;
```

互換性

標準SQLにはDROP LANGUAGE文はありません。

関連項目

[ALTER LANGUAGE](#), [CREATE LANGUAGE](#), [droplang](#)

DROP OPERATOR

名前

DROP OPERATOR -- 演算子を削除する

概要

```
DROP OPERATOR [ IF EXISTS ] name ( { left_type | NONE } , { right_type | NONE } ) [ CASCADE | RESTRICT ]
```

説明

DROP OPERATOR はデータベースシステムから既存の演算子を削除します。 このコマンドを実行するには、その演算子の所有者でなければなりません。

パラメータ

IF EXISTS

演算子が存在しない場合でもエラーになりません。 この場合注意メッセージが発行されます。

name

既存の演算子の名前です（スキーマ修飾名も可）。

left_type

演算子の左項のデータ型です。 演算子に左項がない場合は、NONE と記述します。

right_type

演算子の右項のデータ型です。 演算子に右項がない場合は、NONE と記述します。

CASCADE

演算子に依存するオブジェクトを自動的に削除します。

RESTRICT

依存するオブジェクトがある場合、演算子の削除を拒否します。 これがデフォルトです。

例

integer 型の累乗を行う演算子 a^n を削除します。

```
DROP OPERATOR ^ (integer, integer);
```

bit 型のビット列の否定を行う左単項演算子 ~b を削除します。

```
DROP OPERATOR ~ (none, bit);
```

bigint 型の階乗を行う右単項演算子 x! を削除します。

```
DROP OPERATOR ! (bigint, none);
```

互換性

標準SQLにはDROP OPERATOR 文はありません。

関連項目

[CREATE OPERATOR](#), [ALTER OPERATOR](#)

[前のページ](#)

DROP LANGUAGE

[ホーム](#)

[上に戻る](#)

[次のページ](#)

DROP OPERATOR CLASS

DROP OPERATOR CLASS

名前

DROP OPERATOR CLASS -- 演算子クラスを削除する

概要

```
DROP OPERATOR CLASS [ IF EXISTS ] name USING index_method [ CASCADE | RESTRICT ]
```

説明

DROP OPERATOR CLASS は既存の演算子クラスを削除します。 このコマンドを実行するには、演算子クラスの所有者でなければなりません。

DROP OPERATOR CLASS はそのクラスで参照される演算子や関数をまったく削除しません。 演算子クラスに依存するインデックスがある場合、削除を成功させるためには CASCADE を指定する必要があります。

パラメータ

IF EXISTS

演算子クラスが存在しない場合でもエラーにはなりません。 この場合注意メッセージが発行されます。

name

既存の演算子クラスの名前です（スキーマ修飾名も可）。

index_method

この演算子クラスを使用するインデックスアクセスメソッドの名前です。

CASCADE

この演算子クラスに依存しているオブジェクトを自動的に削除します。

RESTRICT

依存しているオブジェクトがある場合に、その演算子クラスの削除を拒否します。 こちらがデフォルトです。

注釈

DROP OPERATOR CLASS は、たとえその演算子族が空になった（特にその演算子族が CREATE OPERATOR CLASS で暗黙的に作成された場合）としても、そのクラスを含む演算子族を削除しません。 空の演算子族は存在しても害はありませんが、整理するために DROP OPERATOR FAMILY を使用してこの演算子族を削除することを勧めます。 この場合、おそらくより良いのは、代わりに DROP OPERATOR FAMILY を使用することです。

例

widget_ops という名前の B-tree 演算子クラスを削除します。

```
DROP OPERATOR CLASS widget_ops USING btree;
```

演算子クラスを使用するインデックスが存在する場合、このコマンドは実行できません。 このようなインデックスを演算子クラスとともに削除するには、CASCADE を指定します。

互換性

標準SQLにはDROP OPERATOR CLASSは存在しません。

関連項目

[ALTER OPERATOR CLASS](#), [CREATE OPERATOR CLASS](#), [DROP OPERATOR FAMILY](#)

[前のページ](#)
DROP OPERATOR

[ホーム](#)
[上に戻る](#)

[次のページ](#)
DROP OPERATOR FAMILY

DROP OPERATOR FAMILY

名前

DROP OPERATOR FAMILY -- 演算子族を削除する

概要

```
DROP OPERATOR FAMILY [ IF EXISTS ] name USING index_method [ CASCADE | RESTRICT ]
```

説明

DROP OPERATOR FAMILYは既存の演算子族を削除します。 このコマンドを実行するためには、その演算子族の所有者でなければなりません。

DROP OPERATOR FAMILYには、その演算子族に含まれるすべての演算子クラスの削除も含まれています。 しかし、演算子族から参照される演算子や関数はまったく削除されません。 この演算子族内の演算子クラスに依存するインデックスが存在する場合、削除を完了させるためにはCASCADEを指定しなければなりません。

パラメータ

IF EXISTS

演算子族が存在しない場合にエラーとしません。 この場合注意メッセージが表示されます。

name

既存の演算子族の名前（スキーマ修飾可）です。

index_method

演算子族が対象とするインデックスアクセスメソッドの名前です。

CASCADE

演算子族に依存するオブジェクトを自動的に削除します。

RESTRICT

何らかのオブジェクトが演算子族に依存している場合、削除を中止します。 これがデフォルトです。

例

B-tree演算子族float_opsを削除します。

```
DROP OPERATOR FAMILY float_ops USING btree;
```

この演算子族内の演算子クラスを使用するインデックスが存在する場合、このコマンドは失敗します。 演算子族に関連するこうしたインデックスも削除する場合はCASCADEを付けてください。

互換性

標準SQLにはDROP OPERATOR FAMILY文はありません。

関連項目

[ALTER OPERATOR FAMILY](#), [CREATE OPERATOR FAMILY](#), [ALTER OPERATOR CLASS](#), [CREATE OPERATOR CLASS](#),

DROP OWNED

名前

DROP OWNED -- データベースロールにより所有されるデータベースオブジェクトを削除します。

概要

```
DROP OWNED BY name [, ...] [ CASCADE | RESTRICT ]
```

説明

DROP OWNED は、現在のデータベース内にある、指定したロールが所有するオブジェクトをすべて削除します。 また、現在のデータベース内にあるオブジェクトと共有オブジェクト（データベース、テーブル空間）に対して指定したロールに与えられた権限も取り上げられます。

パラメータ

name

所有するオブジェクトを削除し、その権限が取り上げられるロールの名称です。

CASCADE

関連するオブジェクトに依存するオブジェクトを自動的に削除します。

RESTRICT

他のデータベースオブジェクトが関連オブジェクトに依存する場合、ロールにより所有されるオブジェクトの削除を取りやめます。 これがデフォルトです。

注釈

DROP OWNED はよく、複数ロールの削除の前処理として使用されます。 DROP OWNED は現在のデータベース内のオブジェクトにのみ影響しますので、このコマンドを通常、削除予定のロールが所有するオブジェクトを持つデータベース毎に実行する必要があります。

CASCADE オプションを使用すると、このコマンドで再帰的に他のユーザが所有するオブジェクトを処理する可能性があります。

[REASSIGN OWNED](#) コマンドは、複数のロールが所有するすべてのデータベースオブジェクトの所有権を再割り当てする、別の方法です。

このロール(複数可)により所有されるデータベースおよびテーブル空間は削除されません。

互換性

DROP OWNED 文はPostgreSQLの拡張です。

関連項目

[REASSIGN OWNED](#), [DROP ROLE](#)

DROP ROLE

名前

DROP ROLE -- データベースロールを削除する

概要

```
DROP ROLE [ IF EXISTS ] name [, ...]
```

説明

DROP ROLEは指定したロール（複数可）を削除します。スーパーユーザロールを削除するには、自身もスーパーユーザでなければなりません。スーパーユーザ以外のロールを削除するには、CREATEROLE権限を持たなければなりません。

データベースクラスタのいずれかから参照されている場合、ロールを削除することができません。削除しようとしてもエラーとなります。ロールを削除する前に、そのロールが所有するオブジェクトすべてを削除（またはその所有権を変更）しなければなりません。また、そのロールが付与した権限も取り上げなければなりません。この目的のためには[REASSIGN OWNED](#)および[DROP OWNED](#)コマンドが有効です。

しかし、ロール内のロールメンバ資格を削除する必要はありません。DROP ROLEは自動的に他のロール内にある対象ロールのメンバ資格を取り上げます。他のロールは削除されることも何らかの影響を受けることもありません。

パラメータ

IF EXISTS

ロールが存在しない場合でもエラーになりません。この場合注意メッセージが発行されます。

name

削除対象のロールの名前です。

注釈

PostgreSQLには、このコマンドと同じ機能を持つ[dropuser](#)プログラムがあります。（実際には、このプログラムはこのコマンドを呼び出しています。）こちらはコマンドシェルから実行することができます。

例

ロールを削除します。

```
DROP ROLE jonathan;
```

互換性

標準SQLではDROP ROLEを定義していますが、一度に1つのロールしか削除することができません。また、PostgreSQLとは異なる権限が必要であると規定しています。

関連項目

[CREATE ROLE](#), [ALTER ROLE](#), [SET ROLE](#)

DROP RULE

名前

DROP RULE -- 書き換えルールを削除する

概要

```
DROP RULE [ IF EXISTS ] name ON table_name [ CASCADE | RESTRICT ]
```

説明

DROP RULE は書き換えルールを削除します。

パラメータ

IF EXISTS

ルールが存在しない場合でもエラーになりません。 この場合注意メッセージが発行されます。

name

削除するルールの名前です。

table_name

そのルールが適用されたテーブルもしくはビューの名前です（スキーマ修飾名も可）。

CASCADE

ルールに依存するオブジェクトを自動的に削除します。

RESTRICT

依存するオブジェクトがある場合、ルールの削除を拒否します。 こちらがデフォルトです。

例

`newrule` という書き換えルールを削除します。

```
DROP RULE newrule ON mytable;
```

互換性

標準SQLにはDROP RULE 文はありません。

関連項目

[CREATE RULE](#)

[前のページ](#)

DROP ROLE

[ホーム](#)

[上に戻る](#)

[次のページ](#)

DROP SCHEMA

DROP SCHEMA

名前

DROP SCHEMA -- スキーマを削除する

概要

```
DROP SCHEMA [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

説明

DROP SCHEMA はデータベースからスキーマを削除します。

スキーマを削除できるのは、所有者またはスーパーユーザのみです。所有者は、スキーマ内に自分が所有していないオブジェクトが含まれていても、そのスキーマ（およびそこに含まれる全てのオブジェクト）を削除できます。

パラメータ

IF EXISTS

スキーマが存在しない場合でもエラーにはなりません。 この場合注意メッセージが発行されます。

name

スキーマの名前です。

CASCADE

スキーマに含まれるオブジェクト（テーブル、関数など）を自動的に削除します。

RESTRICT

スキーマにオブジェクトが含まれている場合は、スキーマの削除を拒否します。 こちらがデフォルトです。

例

データベースから `mystuff` スキーマ、およびそこに含まれる全てのオブジェクトを削除します。

```
DROP SCHEMA mystuff CASCADE;
```

互換性

標準SQLでは一度のコマンド実行につき1つのスキーマしか削除できないという点を除き、および、PostgreSQLの拡張である `IF EXISTS` を除き、`DROP SCHEMA` は、標準SQLと完全な互換性を持ちます。

関連項目

[ALTER SCHEMA](#), [CREATE SCHEMA](#)

DROP SEQUENCE

名前

DROP SEQUENCE -- シーケンスを削除する

概要

```
DROP SEQUENCE [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

説明

DROP SEQUENCE はシーケンス番号ジェネレータを削除します。 シーケンスの削除はその所有者またはスーパーユーザのみが可能です。

パラメータ

IF EXISTS

シーケンスが存在しない場合でもエラーになりません。 この場合、注意メッセージが発行されます。

name

シーケンスの名前です（スキーマ修飾名も可）。

CASCADE

このシーケンスに依存しているオブジェクトを自動的に削除します。

RESTRICT

依存オブジェクトがある場合に、シーケンスの削除を拒否します。 こちらがデフォルトです。

例

serial という名前のシーケンスを削除します。

```
DROP SEQUENCE serial;
```

互換性

標準では1コマンドで1つのシーケンスだけを削除できるという点を除き、および、PostgreSQLの拡張である IF EXISTS オプションを除き、DROP SEQUENCE は標準SQLに準拠します。

関連項目

[CREATE SEQUENCE](#), [ALTER SEQUENCE](#)

[前のページ](#)

DROP SCHEMA

[ホーム](#)

[上に戻る](#)

[次のページ](#)

DROP SERVER

DROP SERVER

名前

DROP SERVER -- 外部サーバの記述子を削除する

概要

```
DROP SERVER [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

説明

`DROP SERVER`は既存の外部サーバ記述子を削除します。 このコマンドを実行するためには、現在のユーザはサーバの所有者でなければなりません。

パラメータ

`IF EXISTS`

サーバが存在しない場合にエラーを発生しません。 この場合警告が発行されます。

name

既存のサーバの名前です。

`CASCADE`

サーバに依存するオブジェクト（ユーザマップなど）を自動的に削除します。

`RESTRICT`

依存するオブジェクトが存在する場合サーバの削除を取りやめます。 これがデフォルトです。

例

存在する場合、サーバ`foo`を削除します。

```
DROP SERVER IF EXISTS foo;
```

互換性

`DROP SERVER`はISO/IEC 9075-9 (SQL/MED)に従います。 `IF EXISTS`句はPostgreSQLの拡張です。

関連項目

[CREATE SERVER](#), [ALTER SERVER](#)

[前のページ](#)

DROP SEQUENCE

[ホーム](#)

[上に戻る](#)

[次のページ](#)

DROP TABLE

DROP TABLE

名前

DROP TABLE -- テーブルを削除する

概要

```
DROP TABLE [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

説明

`DROP TABLE` はデータベースからテーブルを削除します。 テーブル所有者、スキーマ所有者、スーパーユーザのみがテーブルを削除することができます。 テーブルを削除するのではなく、テーブルの行を空にするには、[DELETE](#) または [TRUNCATE](#) を使用してください。

`DROP TABLE` は、削除対象のテーブル内に存在するインデックス、ルール、トリガ、制約を全て削除します。 しかし、ビューや他のテーブルから外部キー制約によって参照されているテーブルを削除するには `CASCADE` を指定する必要があります（`CASCADE` を指定すると、テーブルに依存するビューは完全に削除されます。しかし、外部キー制約によって参照されている場合は、外部キー制約のみが削除され、その外部キーを持つテーブルそのものは削除されません）。

パラメータ

`IF EXISTS`

テーブルが存在しない場合でもエラーにはなりません。 この場合、注意メッセージが発行されます。

name

削除するテーブルの名前です（スキーマ修飾名も可）。

`CASCADE`

削除するテーブルに依存しているオブジェクト（ビューなど）を自動的に削除します。

`RESTRICT`

依存しているオブジェクトがある場合に、テーブルの削除を拒否します。 こちらがデフォルトです。

例

2つのテーブル、`films` と `distributors` を削除します。

```
DROP TABLE films, distributors;
```

互換性

標準では1コマンドで1テーブルのみを削除できるという点を除き、および、PostgreSQLの拡張である `IF EXISTS` オプションを除き、このコマンドは標準SQLに従います。

関連項目

[ALTER TABLE](#), [CREATE TABLE](#)

DROP TABLESPACE

名前

DROP TABLESPACE -- テーブル空間を削除する

概要

```
DROP TABLESPACE [ IF EXISTS ] name
```

説明

DROP TABLESPACE はシステムからテーブル空間を削除します。

テーブル空間を削除できるのは、その所有者もしくはスーパーユーザのみです。 テーブル空間を削除する前に、全てのデータベースオブジェクトが空になっていなければなりません。 現在のデータベース内のオブジェクトが使用していなかったとしても、他のデータベース内のオブジェクトがそのテーブル空間上にあることがあります。 また、活動中のセッションのいずれかの [temp tablespaces](#) のリストにそのテーブル空間が含まれている場合、一時ファイルがそのテーブル空間に存在するため DROP が失敗する可能性があります。

パラメータ

IF EXISTS

テーブル空間が存在しない場合でもエラーになりません。 この場合注意メッセージが発行されます。

name

テーブル空間の名前です。

注釈

トランザクションブロック内で DROP TABLESPACE を実行することはできません。

例

mystuff テーブル空間をシステムから削除します。

```
DROP TABLESPACE mystuff;
```

互換性

DROP TABLESPACE は PostgreSQL の拡張です。

関連項目

[CREATE TABLESPACE](#), [ALTER TABLESPACE](#)

DROP TEXT SEARCH CONFIGURATION

名前

DROP TEXT SEARCH CONFIGURATION -- テキスト検索設定を削除する

概要

```
DROP TEXT SEARCH CONFIGURATION [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

説明

DROP TEXT SEARCH CONFIGURATIONは既存のテキスト検索設定を削除します。 このコマンドを実行するためには、その設定の所有者でなければなりません。

パラメータ

IF EXISTS

テキスト検索設定が存在しない場合でもエラーとしません。 この場合は注記が発行されます。

name

既存のテキスト検索設定の名称（スキーマ修飾可）です。

CASCADE

テキスト検索設定に依存するオブジェクトを自動的に削除します。

RESTRICT

依存するオブジェクトが存在する場合、テキスト検索設定の削除を中止します。 これがデフォルトです。

例

テキスト検索設定のmy_englishを削除します。

```
DROP TEXT SEARCH CONFIGURATION my_english;
```

to_tsvector呼び出し内でこの設定を参照するインデックスが存在する場合、このコマンドは成功しません。 こうしたインデックスをテキスト検索設定と一緒に削除するためにはCASCADEを付けてください。

互換性

標準SQLにはDROP TEXT SEARCH CONFIGURATION文はありません。

関連項目

[ALTER TEXT SEARCH CONFIGURATION](#), [CREATE TEXT SEARCH CONFIGURATION](#)

DROP TEXT SEARCH DICTIONARY

名前

DROP TEXT SEARCH DICTIONARY -- テキスト検索辞書を削除する

概要

```
DROP TEXT SEARCH DICTIONARY [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

説明

DROP TEXT SEARCH DICTIONARYは既存のテキスト検索辞書を削除します。 このコマンドを実行するためには、その辞書の所有者でなければなりません。

パラメータ

IF EXISTS

テキスト検索辞書が存在しない場合でもエラーとしません。 この場合は注記が発行されます。

name

既存のテキスト検索辞書の名称（スキーマ修飾可）です。

CASCADE

テキスト検索辞書に依存するオブジェクトを自動的に削除します。

RESTRICT

依存するオブジェクトが存在する場合、テキスト検索辞書の削除を中止します。 これがデフォルトです。

例

テキスト検索辞書のenglishを削除します。

```
DROP TEXT SEARCH DICTIONARY english;
```

この辞書を使用するテキスト検索設定が存在する場合、このコマンドは成功しません。 こうした設定を辞書と一緒に削除するためにはCASCADEを付けてください。

互換性

標準SQLにはDROP TEXT SEARCH DICTIONARY文はありません。

関連項目

[ALTER TEXT SEARCH DICTIONARY](#), [CREATE TEXT SEARCH DICTIONARY](#)

DROP TEXT SEARCH PARSER

名前

DROP TEXT SEARCH PARSER -- テキスト検索パーサを削除する

概要

```
DROP TEXT SEARCH PARSER [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

説明

DROP TEXT SEARCH PARSERは既存のテキスト検索パーサを削除します。 このコマンドを実行するためには、その辞書のスーパーユーザでなければなりません。

パラメータ

IF EXISTS

テキスト検索パーサが存在しない場合でもエラーとしません。 この場合は注記が発行されます。

name

既存のテキスト検索パーサの名称（スキーマ修飾可）です。

CASCADE

テキスト検索パーサに依存するオブジェクトを自動的に削除します。

RESTRICT

依存するオブジェクトが存在する場合、テキスト検索パーサの削除を中止します。 これがデフォルトです。

例

テキスト検索パーサのmy_parserを削除します。

```
DROP TEXT SEARCH PARSE my_parser;
```

このパーサを使用するテキスト検索設定が存在する場合、このコマンドは成功しません。 こうしたパーサを辞書と一緒に削除するためにはCASCADEを付けてください。

互換性

標準SQLにはDROP TEXT SEARCH PARSE文はありません。

関連項目

[ALTER TEXT SEARCH PARSE](#), [CREATE TEXT SEARCH PARSE](#)

DROP TEXT SEARCH TEMPLATE

名前

DROP TEXT SEARCH TEMPLATE -- テキスト検索テンプレートを削除する

概要

```
DROP TEXT SEARCH TEMPLATE [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

説明

DROP TEXT SEARCH TEMPLATE は既存のテキスト検索テンプレートを削除します。 このコマンドを実行するためには、そのテンプレートのスーパーユーザでなければなりません。

パラメータ

IF EXISTS

テキスト検索テンプレートが存在しない場合でもエラーとしません。 この場合は注記が発行されます。

name

既存のテキスト検索テンプレートの名称（スキーマ修飾可）です。

CASCADE

テキスト検索テンプレートに依存するオブジェクトを自動的に削除します。

RESTRICT

依存するオブジェクトが存在する場合、テキスト検索テンプレートの削除を中止します。 これがデフォルトです。

例

テキスト検索テンプレートの `thesaurus` を削除します。

```
DROP TEXT SEARCH TEMPLATE thesaurus;
```

このテンプレートを使用するテキスト検索辞書が存在する場合、このコマンドは成功しません。 こうした辞書をテンプレートと一緒に削除するためには `CASCADE` を付けてください。

互換性

標準SQLには `DROP TEXT SEARCH TEMPLATE` 文はありません。

関連項目

[ALTER TEXT SEARCH TEMPLATE](#), [CREATE TEXT SEARCH TEMPLATE](#)

DROP TRIGGER

名前

DROP TRIGGER -- トリガを削除する

概要

```
DROP TRIGGER [ IF EXISTS ] name ON table_name [ CASCADE | RESTRICT ]
```

説明

DROP TRIGGER は既存のトリガ定義を削除します。 このコマンドを実行できるのは、トリガが定義されたテーブルの所有者のみです。

パラメータ

IF EXISTS

トリガが存在しない場合でもエラーになりません。この場合注意メッセージが発行されます。

name

削除するトリガの名前です。

table_name

トリガが定義されたテーブルの名前です（スキーマ修飾名も可）。

CASCADE

このトリガに依存しているオブジェクトを自動的に削除します。

RESTRICT

依存しているオブジェクトがある場合に、トリガの削除を拒否します。こちらがデフォルトです。

例

films テーブルにある if_dist_exists というトリガを削除します。

```
DROP TRIGGER if_dist_exists ON films;
```

互換性

PostgreSQL の DROP TRIGGER 文には標準SQLとの互換性がありません。標準SQLでは、トリガ名がテーブルに局所化されていないので、DROP TRIGGER *name* というコマンドが使われています。

関連項目

[CREATE TRIGGER](#)

[前のページ](#)

DROP TEXT SEARCH TEMPLATE

[ホーム](#)
[上に戻る](#)

[次のページ](#)
DROP TYPE

DROP TYPE

名前

DROP TYPE -- データ型の削除

概要

```
DROP TYPE [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

説明

DROP TYPE はユーザ定義型を削除します。 データ型を削除できるのは、その所有者のみです。

パラメータ

IF EXISTS

型が存在しない場合でもエラーになりません。 この場合、注意メッセージが発行されます。

name

削除するデータ型の名前です（スキーマ修飾名も可）。

CASCADE

削除するデータ型に依存するオブジェクト（テーブルの列、関数、演算子など）を自動的に削除します。

RESTRICT

依存しているオブジェクトがある場合に、データ型の削除を拒否します。 こちらがデフォルトです。

例

`box` データ型を削除します。

```
DROP TYPE box;
```

互換性

このコマンドは、PostgreSQLの拡張であるIF EXISTS オプションを除き、標準SQL内の対応するコマンドと似ています。 しかし、PostgreSQLのCREATE TYPE コマンドの多くとデータ型拡張機構は標準SQLとは異なる点に注意してください。

関連項目

[ALTER TYPE](#), [CREATE TYPE](#)

[前のページ](#)

DROP TRIGGER

[ホーム](#)

[上に戻る](#)

[次のページ](#)

DROP USER

DROP USER

名前

DROP USER -- データベースロールを削除する

概要

```
DROP USER [ IF EXISTS ] name [, ...]
```

説明

DROP USER は [DROP ROLE](#) の別名となりました。

互換性

DROP USER 文は PostgreSQL の拡張です。 標準 SQL では、ユーザの定義は実装に任されています。

関連項目

[DROP ROLE](#)

[前のページ](#)

DROP TYPE

[ホーム](#)
[上に戻る](#)

[次のページ](#)

DROP USER MAPPING

DROP USER MAPPING

名前

DROP USER MAPPING -- 外部サーバ用のユーザマップを削除します。

概要

```
DROP USER MAPPING [ IF EXISTS ] FOR { user_name | USER | CURRENT_USER | PUBLIC } SERVER server_name
```

説明

DROP USER MAPPING は既存のユーザマップを外部サーバから削除します。

外部サーバの所有者は任意のユーザに対するそのサーバ向けのユーザマップを削除することができます。 また、サーバ上で `USAGE` 権限がユーザに付与されていた場合、ユーザは自身の持つユーザ名に対応するユーザマップを削除することができます。

パラメータ

`IF EXISTS`

ユーザマップが存在しない場合にエラーを発生しません。 この場合警告が発行されます。

user_name

対応付けされるユーザ名です。 `CURRENT_USER` と `USER` は現在のユーザの名前に一致します。 `PUBLIC` は、システム上の現在および将来のユーザ名すべてに一致させるために使用します。

server_name

ユーザマップのサーバ名です。

例

存在する場合、サーバ `foo` からユーザマップ `bob` を削除します。

```
DROP USER MAPPING IF EXISTS FOR bob SERVER foo;
```

互換性

DROP USER MAPPING は ISO/IEC 9075-9 (SQL/MED) に従います。 `IF EXISTS` 句は PostgreSQL の拡張です。

関連項目

[CREATE USER MAPPING](#), [ALTER USER MAPPING](#)

[前のページ](#)

DROP USER

[ホーム](#)

[上に戻る](#)

[次のページ](#)

DROP VIEW

DROP VIEW

名前

DROP VIEW -- ビューを削除する

概要

```
DROP VIEW [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

説明

DROP VIEW は既存のビューを削除します。 このコマンドを実行できるのは、ビューの所有者のみです。

パラメータ

IF EXISTS

ビューが存在しなかったとしてもエラーになりません。 この場合注意メッセージが発行されます。

name

削除するビューの名前です（スキーマ修飾名も可）。

CASCADE

削除するビューに依存しているオブジェクト（他のビューなど）を自動的に削除します。

RESTRICT

依存するオブジェクトがある場合は、ビューの削除を拒否します。 こちらがデフォルトです。

例

次のコマンドは `kinds` という名前のビューを削除します。

```
DROP VIEW kinds;
```

互換性

標準では1コマンドで1つのビューのみを削除できるという点を除き、およびPostgreSQLの拡張であるIF EXISTS オプションを除き、このコマンドは標準SQLに従っています。

関連項目

[ALTER VIEW](#), [CREATE VIEW](#)

[前のページ](#)

DROP USER MAPPING

[ホーム](#)
[上に戻る](#)

[次のページ](#)
END

END

名前

END -- 現在のトランザクションをコミットする

概要

```
END [ WORK | TRANSACTION ]
```

説明

`END`は現在のトランザクションをコミットします。これにより、そのトランザクションでなされた全ての変更は、他に対して可視状態となります。また、クラッシュが発生したとしても永続性が保証されます。このコマンドは、PostgreSQLの拡張で、[COMMIT](#)と同等です。

パラメータ

WORK
TRANSACTION

省略可能なキーワードです。何も効果がありません。

注釈

トランザクションのアボートには[ROLLBACK](#)を使用してください。

トランザクションの外側で`END`を発行しても問題はありませんが、警告メッセージが表示されます。

例

現在のトランザクションをコミットし、全ての変更を永続化します。

```
END;
```

互換性

`END`はPostgreSQLの拡張です。このコマンドの機能は、標準SQLで規定された[COMMIT](#)と同じです。

関連項目

[BEGIN](#), [COMMIT](#), [ROLLBACK](#)

EXECUTE

名前

EXECUTE -- 準備された文を実行する

概要

```
EXECUTE name [ ( parameter [, ...] ) ]
```

説明

EXECUTE は、前もって準備された文を実行する際に使用します。 準備された文はセッション中にしか存在できないため、事前に同一セッション中のPREPARE 文によって作成されたものでなければなりません。

文を作成したPREPARE 文にパラメータが指定されている場合は、これに適合するパラメータの集合がEXECUTE に渡される必要があります。 そうしないと、エラーになります。 （関数とは異なり）準備された文は、パラメータのデータ型や個数によってオーバーロードされることはありません。 準備された文の名前は、1つのデータベースセッション内で一意でなければなりません。

準備された文の作成方法と使用方法についての詳細は[PREPARE](#)を参照してください。

パラメータ

name

実行する準備された文の名前を指定します。

parameter

準備された文に対するパラメータの実際の値を式で指定します。 この式は、準備された文が生成された時に決定される、そのパラメータのデータ型と互換性のある値を返す必要があります。

出力

EXECUTE で返されるコマンドタグは、EXECUTE ではなく、準備された文のコマンドタグとなります。

例

例は[PREPARE](#)文書の[例節](#)にあります。

互換性

標準SQLにはEXECUTE 文が含まれていますが、これは埋め込みSQLでのみ使用できます。 また、このバージョンのEXECUTE 文では、多少異なる構文が使用されています。

関連項目

[DEALLOCATE](#), [PREPARE](#)

EXPLAIN

名前

EXPLAIN -- 問い合わせ文の実行計画を表示する

概要

```
EXPLAIN [ ( option [, ...] ) ] statement
EXPLAIN [ ANALYZE ] [ VERBOSE ] statement
```

ここで`option`は以下のいずれかを取ることができます。

```
ANALYZE [ boolean ]
VERBOSE [ boolean ]
COSTS [ boolean ]
BUFFERS [ boolean ]
TIMING [ boolean ]
FORMAT { TEXT | XML | JSON | YAML }
```

説明

与えられた文に対して、PostgreSQLプランナが生成する実行計画を表示します。実行計画は、問い合わせ文が参照するテーブル（複数の場合もある）をスキャンする方法（単純なシーケンススキャン、インデックススキャンなど）、複数のテーブルを参照する場合に、各テーブルから取り出した行を結合するために使用する結合アルゴリズムを示すものです。

表示内容の中でも、最も重要なのは、文の実行にかかるコストの見積もりです。これは、プランナが文の実行にかかる時間（任意の、しかし慣習的にはディスクページ抽出を意味するコスト単位で計測）を推測したものです。具体的には、2つの数が表示されます。1つは最初の行が返されるまでのスタートアップコスト、もう1つはすべての行が返されるまでの合計コストです。ほとんどの問い合わせにとって問題となるのは合計コストの方ですが、`EXISTS`副問い合わせなどのコンテキストでは、プランナは合計コストが最も短くなるプランよりも、スタートアップコストが最も短くなるプランを選びます（エクゼキュータは行を1つ取得した後に停止するからです）。さらに、`LIMIT`句を使って問い合わせが返す行数を制限する場合、プランナは実際にはどの計画が一番低コストになるかを概算するため、全体を処理した場合のコストの間で適切な補間を行います。

`ANALYZE`オプションを付けると、計画を作るだけでなく、文が実際に実行されます。この場合は、各計画ノードで費された総経過時間（ミリ秒単位）と実際に返された全行数を含む、実実行時の統計情報が追加表示されます。プランナの推測と実際の値がどの程度近くなるかを確認できるという点で、このオプションは有用です。

重要項目: `ANALYZE`を使用した場合は、文が実際に実行されることを忘れないでください。`EXPLAIN`は`SELECT`が返す出力をまったく表示しませんが、文に伴う副作用は通常通り発生します。`INSERT`、`UPDATE`、`DELETE`、`CREATE TABLE AS`、`EXECUTE`文に対して、データに影響を与えないように`EXPLAIN ANALYZE`を実行したい場合は、以下の方法を使用してください。

```
BEGIN;
EXPLAIN ANALYZE ...;
ROLLBACK;
```

`ANALYZE`および`VERBOSE`オプションのみが、この順序でのみ、オプションリストを括弧で括弧することなく、指定可能です。PostgreSQL 9.0より前までは、括弧がない構文のみがサポートされていました。すべての新しいオプションは括弧付き構文のみでサポートされることを想定しています。

パラメータ

`ANALYZE`

コマンドを実行し、実際の実行時間やその他の統計情報を表示します。このパラメータのデフォルトは`FALSE`です。

`VERBOSE`

計画についての追加情報を出力します。具体的に言うと、計画ツリー、スキーマ修飾テーブル、関数名内の各ノードに対して出力列リストを含めます。常に範囲テーブルの別名を付けて式内の変数を命名し、また常に統計情報が表示される各トリガの名前を出力します。このパラメータのデフォルトは`FALSE`です。

COSTS

各計画ノードの推定起動コストと総コスト、さらに推定行数と各行の推定幅に関する情報を含めます。このパラメータのデフォルトは`TRUE`です。

BUFFERS

バッファの使用状況に関する情報を含めます。具体的には、共有ブロックのヒット数、読み取り数、ダーティブロック数、書き出し数、ローカルブロックのヒット数、ダーティブロック数、読み取り数、書き出し数、一時ブロックの読み取り数、書き出し数が含まれます。ヒットとは、必要な時にキャッシュ内にそのブロックが見つかったため読み取りが避けられたことを意味します。共有ブロックには、通常のテーブルとインデックスからのデータが含まれます。ローカルブロックには、一時テーブルとそのインデックスからのデータが含まれます。一時ブロックには、ソートやハッシュ、マテリアライズ計画ノードで使用される短期間有効なデータが含まれます。ダーティブロック数は、これまでは変更がなかったがその問い合わせによって変更されたブロックの数を示します。書き出しブロック数は、問い合わせ処理の間にバックエンドにより、ダーティ状態だったブロックの内キャッシュから追い出されたブロックの数を示します。上位レベルのノードで表示されるブロック数には、その子ノードすべてで使用されるブロックが含まれます。テキスト形式では、非ゼロの値のみが出力されます。このパラメータは`ANALYZE`パラメータも有効である場合にのみ使用することができます。デフォルトは`FALSE`です。

TIMING

実際のスタートアップ時間とノードで費やされた時間が追加表示されます。一部のシステムでは、システムクロックを何度も読み取るオーバーヘッドのため問い合わせがかなり低速になる可能性があります。このため、実際の時間ではなく実際の行数のみが必要であるのであれば、このパラメータは`FALSE`に設定する方が有益でしょう。このパラメータは`ANALYZE`パラメータも有効である場合にのみ使用することができます。デフォルトは`TRUE`です。

FORMAT

出力形式を指定します。TEXT、XML、JSON、YAMLを指定可能です。TEXT以外の出力にはTEXT出力と同じ情報が含まれていますが、プログラムによる解析がより容易になります。このパラメータのデフォルトは`TEXT`です。

boolean

選択したオプションを有効とするか無効とするかを指定します。オプションを有効にするためには`TRUE`、`ON`、`1`のいずれかを書きます。無効にするには`FALSE`、`OFF`、`0`のいずれかを書きます。`boolean`値は省略可能です。省略時は`TRUE`と仮定されます。

statement

実行計画の表示対象となる、`SELECT`、`INSERT`、`UPDATE`、`DELETE`、`VALUES`、`EXECUTE`、`DECLARE`、`CREATE TABLE AS`のいずれかの文です。

出力

コマンドの結果は、`statement`に対して選択された計画をテキストで説明します。省略することもできますが、実行時の統計情報で注釈が付けられます。[項14.1](#)では出力される情報について説明します。

注釈

PostgreSQL問い合わせプランナが十分な情報を使って問合せを最適化できるようにするには、問い合わせ内で使用されるすべてのテーブルに関する`pg_statistic`のデータを最新状態にしなければなりません。通常[自動バキュームデーモン](#)により自動的に処理されます。しかし最近その内容が大きく変更されたテーブルでは、自動バキュームがその変更を追いつくまで待つのではなく、手作業による[ANALYZE](#)を実行しなければならないでしょう。

実行計画内の各ノードの実行時コストを測定するために、現在の`EXPLAIN ANALYZE`実装は、問い合わせ実行に対し、情報収集のためのオーバーヘッドを加えます。この結果、ある問い合わせについての`EXPLAIN ANALYZE`実行が、普通に問い合わせを実行した場合より非常に時間がかかることがあります。このオーバーヘッドの量は問い合わせの性質と使用するプラットフォームに依存します。実行の間非常に短い時間を必要とする計画ノードに関して、日付を得るためのシステムコールの操作が相対的に低速なプラットフォーム上で最悪な場合が発生します。

例

`integer`列を1つ持ち、10000行が存在するテーブルに対して、単純な問い合わせを行った場合の問い合わせ計画を表示します。

```
EXPLAIN SELECT * FROM foo;
```

```

          QUERY PLAN
-----
Seq Scan on foo   (cost=0.00..155.00 rows=10000 width=4)
(1 row)

```

以下は同じ問い合わせをJSON出力形式で出力したものです。

```

EXPLAIN (FORMAT JSON) SELECT * FROM foo;
          QUERY PLAN
-----
[
  {
    "Plan": {
      "Node Type": "Seq Scan",
      "Relation Name": "foo",
      "Alias": "foo",
      "Startup Cost": 0.00,
      "Total Cost": 155.00,
      "Plan Rows": 10000,
      "Plan Width": 4
    }
  }
]
(1 row)

```

インデックスが存在し、問い合わせのWHERE条件でインデックスを利用できる場合、EXPLAINは異なる計画を表示する可能性があります。

```

EXPLAIN SELECT * FROM foo WHERE i = 4;
          QUERY PLAN
-----
Index Scan using fi on foo   (cost=0.00..5.98 rows=1 width=4)
Index Cond: (i = 4)
(2 rows)

```

以下は同じ問い合わせをYAML形式で表したものです。

```

EXPLAIN (FORMAT YAML) SELECT * FROM foo WHERE i='4';
          QUERY PLAN
-----
- Plan:
  Node Type: "Index Scan"
  Scan Direction: "Forward"
  Index Name: "fi"
  Relation Name: "foo"
  Alias: "foo"
  Startup Cost: 0.00
  Total Cost: 5.98
  Plan Rows: 1
  Plan Width: 4
  Index Cond: "(i = 4)"
(1 row)

```

読者への演習としてXML形式については記載しません。

以下はコスト推定値を出力しない同じ計画です。

```

EXPLAIN (COSTS FALSE) SELECT * FROM foo WHERE i = 4;
          QUERY PLAN
-----
Index Scan using fi on foo
Index Cond: (i = 4)
(2 rows)

```

次に、集約関数を使用した問い合わせに対する問い合わせ計画の例を示します。

```

EXPLAIN SELECT sum(i) FROM foo WHERE i < 10;
          QUERY PLAN
-----
Aggregate   (cost=23.93..23.93 rows=1 width=4)
->  Index Scan using fi on foo   (cost=0.00..23.92 rows=6 width=4)
    Index Cond: (i < 10)
(3 rows)

```

以下は、EXPLAIN EXECUTEによって準備された文に対する実行計画を表示する例です。

```

PREPARE query(int, int) AS SELECT sum(bar) FROM test
WHERE id > $1 AND id < $2
GROUP BY foo;

```

```
EXPLAIN ANALYZE EXECUTE query(100, 200);

                                QUERY PLAN
-----
HashAggregate (cost=39.53..39.53 rows=1 width=8) (actual time=0.661..0.672 rows=7 loops=1)
  -> Index Scan using test_pkey on test (cost=0.00..32.97 rows=1311 width=8) (actual time=0.050..0.395
rows=99 loops=1)
    Index Cond: ((id > $1) AND (id < $2))
  Total runtime: 0.851 ms
(4 rows)
```

もちろん、ここで示した具体的な数値は対象とするテーブルの実際の中身によって変わります。また、この数値や選択された問い合わせ戦略は、プランナの改良のため、PostgreSQLのリリース間で異なる可能性がありますので注意してください。さらに、ANALYZE コマンドは、データの統計情報を推定する際にランダムなサンプリングを行うため、実際のテーブル内の分布が変わっていなくても、新たにANALYZEを実行すると推定コストが変わることがあります。

互換性

標準SQLではEXPLAIN文は定義されていません。

関連項目

ANALYZE

前のページ	ホーム	次のページ
EXECUTE	上に戻る	FETCH

FETCH

名前

FETCH -- カーソルを使用してテーブルから行を抽出する

概要

```
FETCH [ direction [ FROM | IN ] ] cursor_name
```

*direction*は空にするか、次のいずれかを指定します。

```
NEXT
PRIOR
FIRST
LAST
ABSOLUTE count
RELATIVE count
count
ALL
FORWARD
FORWARD count
FORWARD ALL
BACKWARD
BACKWARD count
BACKWARD ALL
```

説明

FETCHは事前に作成されたカーソルを使用して行を取り出します。

カーソルはそれぞれ位置情報を持っており、FETCHはこれを使用します。カーソル位置として取ることができるのは、問い合わせの結果の先頭行の前、結果内の任意の特定の行、結果の最終行の後のいずれかです。カーソルの生成時は、カーソル位置は先頭行の前にあります。行を取り出した後は、カーソル位置は最後に取り出した行にあります。FETCHが利用可能な行の終わりまで達すると、カーソル位置は最終行の後にあります（後方抽出の場合は先頭行の前になります）。FETCH ALLもしくはFETCH BACKWARD ALLでは、カーソルの位置は常に最終行の後か、先頭行の前になります。

NEXT、PRIOR、FIRST、LAST、ABSOLUTE、RELATIVE形式では、カーソルを適切に移動した後、行を1つ取り出します。行が存在しない場合、空の結果が返され、カーソル位置はそのまま先頭行の前か最終行の後に留まります。

FORWARDおよびBACKWARDを使用した構文では、指定数の行を前方もしくは後方方向に取り出します。この時、最後に取り出した行がカーソル位置となります（*count*が利用可能な行数を超えた場合は、全行の後/前になります）。

RELATIVE 0、FORWARD 0、およびBACKWARD 0は全て、カーソルを移動することなく現在の行を取り出します。つまり、一番最後に取り出した行を再度取り出すことになります。カーソルが先頭行の前や最終行の後になければ、これらのコマンドは成功します。先頭行の前や最終行の後にあれば、行は返されません。

注意: このマニュアルページではSQLコマンドレベルでのカーソルの使用方法について説明します。PL/pgSQL内でカーソルを使用するつもりであれば、規則が異なりますので、[項39.7](#)を参照してください。

パラメータ

direction

*direction*は、取り出す方向と取り出す行数を定義します。以下のいずれかを指定できます。

NEXT

次の行を取り出します。これは、*direction*が省略された時のデフォルトです。

PRIOR

1つ前の行を取り出します。

FIRST

問い合わせの先頭行を取り出します（`ABSOLUTE 1`と同じです）。

`LAST`

問い合わせの最終行を取り出します（`ABSOLUTE -1`と同じです）。

`ABSOLUTE count`

問い合わせの`count`番目の行を取り出します。`count`が負ならば、終わりから`abs(count)`番目の行を取り出します。`count`が範囲外の場合、カーソル位置は先頭行の前か最終行の後になります。特に、`ABSOLUTE 0`と指定すると、先頭行の前になります。

`RELATIVE count`

カーソルの現在位置から`count`番目の行を取り出します。`count`が負の場合、現在位置より`abs(count)`行分前の行を取り出します。`RELATIVE 0`と指定すると、現在の行があれば、その行を再度取り出します。

`count`

次の`count`行を取り出します（`FORWARD count`と同じです）。

`ALL`

残っている行を全て取り出します（`FORWARD ALL`と同じです）。

`FORWARD`

1つ次の行を取り出します（`NEXT`と同じです）。

`FORWARD count`

次の`count`行分の行を取り出します。`FORWARD 0`と指定すると、現在の行を再度取り出します。

`FORWARD ALL`

残っている行を全て取り出します。

`BACKWARD`

1つ前の行を取り出します（`PRIOR`と同じです）。

`BACKWARD count`

前の`count`行分の行を（逆方向に走査して）取り出します。`BACKWARD 0`と指定すると、現在の行を再度取り出します。

`BACKWARD ALL`

現在位置より前の行を（逆方向に走査して）全て取り出します。

`count`

取り出す位置や行数を決定する`count`は、整数定数で、符号を付けることができます。`FORWARD`と`BACKWARD`において、`count`に負の値を指定すると、`FORWARD`と`BACKWARD`の意味が入れ替わると考えることができます。

`cursor_name`

開いているカーソルの名前を指定します。

出力

正常に終了すると、`FETCH` コマンドは以下の形式のコマンドタグを返します。

`FETCH count`

`count`は取り出した行数です（0の可能性もあります）。`psql`では取り出した行数を別途表示するため、このコマンドタグは実際には表示されないことに注意してください。

注釈

`FETCH` コマンドとして、`FETCH NEXT`もしくは`FETCH FORWARD`以外を使用する場合、カーソルを`SCROLL`オプション付きで宣言しなければなりません。単純な問い合わせでは、PostgreSQLにより、カーソルが`SCROLL`付きで宣言されていなくても後方向の取り出

しを行うことができますが、この動作に依存すべきではありません。カーソルがNO SCROLL付きで宣言された場合は、後方向の取り出しを行うことができません。

相対的な指定による指定行への移動に比べると、ABSOLUTEによる取り出しは高速ではありません。内部的な実装では、必ず中間の行を全て経由しているからです。絶対指定で負の値を指定した場合、速度はさらに悪化します。まず、最終行を見つけるために最後まで問い合わせを読み取って、その後に最終行から後方に移動するためです。ただし、(FETCH ABSOLUTE 0を使用して) 問い合わせの先頭へ戻る時は高速です。

[DECLARE](#)を使用してカーソルを定義します。データを取り出さずにカーソル位置を変更する場合は[MOVE](#)を使用してください。

例

次の例では、カーソルを使用してテーブル内を行き来しています。

```
BEGIN WORK;

-- カーソルを設定します。
DECLARE liahona SCROLL CURSOR FOR SELECT * FROM films;

-- liahonaカーソルから最初の5行を取り出します。

FETCH FORWARD 5 FROM liahona;

code | title | did | date_prod | kind | len
-----+-----+-----+-----+-----+-----
BL101 | The Third Man | 101 | 1949-12-23 | Drama | 01:44
BL102 | The African Queen | 101 | 1951-08-11 | Romantic | 01:43
JL201 | Une Femme est une Femme | 102 | 1961-03-12 | Romantic | 01:25
P_301 | Vertigo | 103 | 1958-11-14 | Action | 02:08
P_302 | Becket | 103 | 1964-02-03 | Drama | 02:28

-- 1つ前の行を取り出します。
FETCH PRIOR FROM liahona;

code | title | did | date_prod | kind | len
-----+-----+-----+-----+-----+-----
P_301 | Vertigo | 103 | 1958-11-14 | Action | 02:08

-- カーソルを閉じ、トランザクションを終了します。
CLOSE liahona;
COMMIT WORK;
```

互換性

標準SQLでは、埋め込みSQLにおけるFETCHのみが定義されています。上記で説明した各種のFETCHは、ホスト変数への代入ではなく、SELECTの結果であるかのようにデータを返します。この点を除き、FETCHは完全に標準SQLと上位互換性を持ちます。

FORWARDとBACKWARDを持つFETCHの形式や、暗黙的なFORWARDを持つFETCH countとFETCH ALLはPostgreSQLの拡張です。

標準SQLでは、カーソル名の前に付けられるのはFROMのみです。INを使用するオプション、または、どちらも省略することはPostgreSQLの拡張です。

関連項目

[CLOSE](#), [DECLARE](#), [MOVE](#)

GRANT

名前

GRANT -- アクセス権限を定義する

概要

```

GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
        [, ...] | ALL [ PRIVILEGES ] }
ON { [ TABLE ] table_name [, ...]
     | ALL TABLES IN SCHEMA schema_name [, ...] }
TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { { SELECT | INSERT | UPDATE | REFERENCES } ( column_name [, ...] )
        [, ...] | ALL [ PRIVILEGES ] ( column_name [, ...] ) }
ON [ TABLE ] table_name [, ...]
TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { { USAGE | SELECT | UPDATE }
        [, ...] | ALL [ PRIVILEGES ] }
ON SEQUENCE sequence_name [, ...]
     | ALL SEQUENCES IN SCHEMA schema_name [, ...] }
TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { { CREATE | CONNECT | TEMPORARY | TEMP } [, ...] | ALL [ PRIVILEGES ] }
ON DATABASE database_name [, ...]
TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
ON DOMAIN domain_name [, ...]
TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
ON FOREIGN DATA WRAPPER fdw_name [, ...]
TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
ON FOREIGN SERVER server_name [, ...]
TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { EXECUTE | ALL [ PRIVILEGES ] }
ON { FUNCTION function_name ( [ [ argmode ] [ arg_name ] arg_type [, ...] ] ) [, ...]
     | ALL FUNCTIONS IN SCHEMA schema_name [, ...] }
TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
ON LANGUAGE lang_name [, ...]
TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { { SELECT | UPDATE } [, ...] | ALL [ PRIVILEGES ] }
ON LARGE OBJECT loid [, ...]
TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { { CREATE | USAGE } [, ...] | ALL [ PRIVILEGES ] }
ON SCHEMA schema_name [, ...]
TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { CREATE | ALL [ PRIVILEGES ] }
ON TABLESPACE tablespace_name [, ...]
TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
ON TYPE type_name [, ...]
TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT role_name [, ...] TO role_name [, ...] [ WITH ADMIN OPTION ]

```

説明

GRANT には基本的に2つの種類があります。1つはデータベースオブジェクト（テーブル、列、ビュー、外部テーブル、シーケンス、データベース、異種データラップ、異種サーバ、関数、手続き言語、スキーマ、テーブル空間）に対する権限の付与、もう1つはロール内のメンバ資格の付与です。これらの種類は多くの点で似ていますが、説明は別々に行わなければならない程違

いがあります。

データベースオブジェクトに対するGRANT

この種類のGRANT コマンドはデータベースオブジェクトの特定の権限を1つ以上のロールに付与します。既に権限が他のロールに付与されている場合でも、追加として付与されます。

1つ以上のスキーマ内の同じ種類のオブジェクトすべてに対して権限を付与するオプションも存在します。この機能は現在テーブル、シーケンス、関数のみでサポートされます。（しかしALL TABLESにはビューおよび外部テーブルが含まれるとみなされていることに注意してください。）

PUBLIC キーワードは、今後作成されるロールを含む、全てのロールへの許可を示します。PUBLICは、全てのロールを常に含む、暗黙的に定義されたグループと考えることができます。個々のロールは全て、ロールに直接許可された権限、ロールが現在属しているロールに許可された権限、そして、PUBLICに許可された権限を合わせた権限を持っています。

WITH GRANT OPTIONが指定されると、権限の受領者は、その後、他にその権限を与えることができます。グラントオプションがない場合、受領者はこれを行うことができません。グラントオプションはPUBLICには与えることができません。

所有者（通常はオブジェクトを作成したユーザ）はデフォルトで全ての権限を保持しているため、オブジェクトの所有者に権限を許可する必要はありません（ただし、オブジェクトの作成者が、安全性を確保するために自らの権限を取り消すことは可能です）。

オブジェクトを削除する権限や何らかの方法でオブジェクトの定義を変更する権限は、付与可能な権限として扱われません。これらの権限は、所有者固有のものであり、許可したり取り消したりすることはできません。（しかし、オブジェクトを所有するロール内のメンバ関係を付与したり取り消すことで、同等な効果を入力することができます。後で説明します。）所有者は、オブジェクトに対する全てのグラントオプションも暗黙的に保持しています。

PostgreSQLは、一部の種類のオブジェクトに対し、デフォルトの権限をPUBLICに付与します。デフォルトでは、テーブル、列、スキーマ、テーブル空間に関してPUBLICに与えられたアクセス権限はありません。他の種類のオブジェクトでは、PUBLICに与えられる権限は次の通りです。データベースではCONNECTおよびCREATE TEMP TABLE 権限、関数ではEXECUTE 権限、言語ではUSAGE 権限です。もちろんオブジェクトの所有者はデフォルトで与えられた権限と明示的に与えられた権限の両方をREVOKE することができます。（最大限の安全性を得るため、REVOKE コマンドはオブジェクトを作成したトランザクションと同じトランザクション内で発行してください。そうすれば、他のユーザがそのオブジェクトを使用する時間はなくなります）。また、これらの初期デフォルト権限の設定はALTER DEFAULT PRIVILEGES コマンドを使用して変更可能です。

設定可能な権限は以下のものです。

SELECT

任意の列、指定した列リスト、指定したテーブル、ビュー、シーケンスの任意の列に対するSELECTを許可します。また、COPY TOの使用も許可します。UPDATEやDELETEで存在する列を参照するためにも、この権限は必要です。シーケンスでは、この権限によってcurrval関数を使用することができます。ラージオブジェクトでは、この権限によってオブジェクトを読み取ることができます。

INSERT

指定したテーブルへの新規行のINSERTを許可します。列リストが指定された場合は、これらの列のみをINSERTコマンド内で代入することができます（この結果、その他の列はデフォルト値となります）。また、COPY FROMの使用も許可します。

UPDATE

任意の列、指定した列リスト、指定したテーブルに対するUPDATEを許可します。（実際には、単純ではないUPDATE コマンドはすべて、SELECT 権限を同様に必要とします。どの行を更新すべきかを決めるため、または、列の新しい値を計算するため、またはその両方のため、テーブル列を参照する必要があるからです。）SELECT ... FOR UPDATEおよびSELECT ... FOR SHAREも、SELECT 権限に加えて、少なくとも1つの列に対するこの権限を必要とします。シーケンスでは、この権限によりnextvalおよびsetvalの使用が許可されます。ラージオブジェクトでは、この権限によりオブジェクトの書き出しや切り詰めを行うことができます。

DELETE

指定したテーブルからの行のDELETEを許可します。（実際には、単純ではないDELETE コマンドはすべて、SELECT 権限を同様に必要とします。どの行を削除すべきかを決めるためにテーブル列を参照する必要があるからです。）

TRUNCATE

指定したテーブルに対してTRUNCATEを行うことができます。

REFERENCES

外部キー制約を作成するには、参照する側と参照される側の両方の列に対して、この権限を持っていなければなりません。この権限はテーブルのすべての列に付与することも、特定の列のみに付与することもできます。

TRIGGER

指定したテーブル上のトリガの作成を許可します（[CREATE TRIGGER](#)文を参照してください）。

CREATE

対象がデータベースの場合は、データベース内での新規スキーマの作成を許可します。

対象がスキーマの場合は、スキーマ内での新規オブジェクトの作成を許可します。既存のオブジェクトの名前を変更するには、オブジェクトを所有し、かつ、そのオブジェクトが入っているスキーマに対してこの権限を保持していなければなりません。

対象がテーブル空間の場合は、テーブル空間内でのテーブルとインデックス、一時ファイルの作成と、デフォルトのテーブル空間としてこのテーブル空間を持つデータベースの作成を許可します（この権限の取り消しによって既存のデータベースやスキーマの振舞いが変わることはないことに注意してください）。

CONNECT

ユーザは指定されたデータベースに接続することができます。この権限は（`pg_hba.conf`で組み込まれる制限の検査に加え）接続開始時に検査されます。

TEMPORARY TEMP

指定したデータベースの使用中に一時テーブルを作成することを許可します。

EXECUTE

指定された関数、さらにその関数で実装されている任意の演算子の使用を許可します。これは、関数に適用することができる唯一の権限です（この構文は集約関数についても同じように機能します）。

USAGE

手続き言語において、その言語で関数を作成することを許可します。これは、手続き言語に適用することができる唯一の権限です。

スキーマにおいて、指定したスキーマに含まれるオブジェクトへのアクセスを許可します（オブジェクト自体の権限要件が満たされている場合）。基本的には、この権限によってスキーマ内のオブジェクトを"検索"する権限も認められます。この権限がなくても、例えばシステムテーブルを問い合わせることでオブジェクト名は判りますが、SQLを介してアクセスすることができません。また、この権限を取り除いた後でも、存在し続けているバックエンドがこうした検索を以前に実行していた文を持っている可能性があります。このため、これはオブジェクトへのアクセスを防ぐ、完全に安全な方法ではありません。

シーケンスにおいて、この権限は`currval`および`nextval`関数の使用を許可します。

型とドメインにおいて、この権限により、テーブル、関数、その他のスキーマオブジェクトの作成の際に、この型またはドメインを使用できるようになります。（問い合わせにおけるその型の値に対する操作など、一般的な型の"使用"を制御するものではないことに注意してください。その型に依存したオブジェクトが生成されることのみを防止します。この権限の主な目的は、どのユーザが型に依存するものを作成するか、所有者がその型を後で変更することを防止するかどうか、を制御することです。）

異種データラップにおいて、この権限を与えられると異種データラップを使用する新しいサーバを作成することができます。

サーバにおいて、この権限を与えられると、サーバに関連する、自身が所有するユーザのユーザマップを作成、変更、削除を行うことができます。またこの権限を与えられると、サーバに関連するユーザマップのオプションを問い合わせることができます。

ALL PRIVILEGES

利用可能な全ての権限を一度に付与します。`PRIVILEGES`キーワードはPostgreSQLでは省略可能ですが、厳密にはSQLでは必須です。

その他のコマンドの実行に必要な権限は、そのコマンドのリファレンスページにて示されています。

ロールに対するGRANT

この種類のGRANT コマンドは、1つ以上のロール内のメンバ資格を付与します。 これによりロールに付与された権限を各メンバに伝えますので、ロール内のメンバ資格は重要です。

WITH ADMIN OPTIONが指定された場合、メンバはロール内のメンバ資格を他に付与することができるようになります。 また同様にロール内のメンバ資格を取り上げることもできるようになります。 アドミンオプションがないと、一般ユーザは他への権限の付与や取り上げを行うことができません。 しかし、データベーススーパーユーザはすべてのロール内のメンバ資格を誰にでも付与したり、削除したりすることができます。 CREATEROLE 権限を持つロールは、スーパーユーザロール以外のロール内のメンバ資格の付与、取り上げが可能です。

権限の場合と異なり、ロール内のメンバ資格をPUBLICに付与することはできません。 また、このコマンド構文では無意味なGROUP という単語を受け付けないことに注意してください。

注釈

アクセス権限を取り消すには、REVOKE コマンドが使用されます。

PostgreSQL 8.1から、ユーザとグループという概念は、ロールと呼ばれる1種類の実体に統合されました。 そのため、GROUP キーワードを使用して付与者がユーザかグループかどうかを識別する必要はなくなりました。 このコマンドではまだGROUPは許されていますが、何の意味もありません。

ユーザは特定の列やテーブル全体に対する権限を持つ場合にSELECT、INSERTなどを実行することができます。 テーブルレベルの権限を付与してからある列に対する権限を取り除くことは、望まないことでしょう。 テーブルレベルの権限は列レベルの操作による影響を受けません。

オブジェクトの所有者でもなく、そのオブジェクトに何の権限も持たないユーザが、そのオブジェクトの権限をGRANTしようとしても、コマンドの実行は直ちに失敗します。 何らかの権限を持っている限り、コマンドの実行は進行しますが、与えることのできる権限は、そのユーザがGRANT オプションを持つ権限のみです。 GRANT ALL PRIVILEGES 構文は警告メッセージを發します。 一方、その他の構文では、コマンドで名前を指定した権限に関するGRANT オプションを持っていない場合に警告メッセージを發します（原理上、ここまでの説明はオブジェクトの所有者に対しても当てはまりますが、所有者は常に全てのGRANT オプションを保持しているものとして扱われるため、こうした状態は決して起こりません）。

データベーススーパーユーザのみが、オブジェクトに関する権限設定に関係なく、全てのオブジェクトにアクセスできることには注意しなければなりません。 スーパーユーザが持つ権限は、Unixシステムにおけるroot権限に似ています。 rootと同様、絶対に必要な場合以外は、スーパーユーザとして操作を行わないのが賢明です。

スーパーユーザがGRANTやREVOKEの發行を選択した場合、それらのコマンドは対象とするオブジェクトの所有者が發行したかのように実行されます。 特に、こうしたコマンドで与えられる権限は、オブジェクトの所有者によって与えられたものとして表されます。（ロールのメンバ資格では、メンバ資格は含まれるロール自身が与えたものとして表されます。）

GRANTおよびREVOKEは、影響するオブジェクトの所有者以外のロールによって実行することもできますが、オブジェクトを所有するロールのメンバであるか、そのオブジェクトに対しWITH GRANT OPTION権限を持つロールのメンバでなければなりません。 この場合、その権限は、そのオブジェクトの実際の所有者ロールまたはWITH GRANT OPTION権限を持つロールによって付与されたものとして記録されます。例えば、t1テーブルがg1ロールによって所有され、u1がg1ロールのメンバであるとします。この場合、u1はt1上の権限をu2に付与できます。しかし、これらの権限はg1によって直接付与されたものとして現れます。後でg1ロールの他のメンバがこの権限を取り上げることができます。

GRANTを実行したロールが、ロールの持つ複数メンバ資格の経路を通して間接的に必要な権限を持つ場合、どのロールが権限を付与したロールとして記録されるかについては指定されません。こうした場合、SET ROLEを使用して、GRANTを行わせたい特定のロールになることを推奨します。

テーブルへの権限付与によって、SERIAL列によって関連付けされたシーケンスを含む、そのテーブルで使用されるシーケンスへの権限の拡張は自動的に行われません。シーケンスへの権限は別途設定しなければなりません。

既存のテーブルおよび列に対する権限についての情報を得るには、以下の例のようにpsqlの\dp コマンドを使用してください。

```
=> \dp mytable

          Access privileges
Schema | Name      | Type | Access privileges | Column access privileges
-----+-----+-----+-----+-----
public | mytable   | table | miriam=arwdDxt/miriam | coll:
                                     : =r/miriam               :   miriam_rw=rw/miriam
                                     1315/2500
```



```
                                : admin=arw/miriam
(1 row)
```

¥dpで表される項目は、以下のように解釈することができます。

```
rolename=xxxx -- ロールに与えられた権限
=xxxx -- PUBLICに与えられた権限

r -- SELECT (読み取り (read) )
w -- UPDATE (書き込み (write) )
a -- INSERT (追加 (append) )
d -- DELETE
D -- TRUNCATE
x -- REFERENCES
t -- TRIGGER
X -- EXECUTE
U -- USAGE
C -- CREATE
c -- CONNECT
T -- TEMPORARY
arwdDxt -- すべての権限 (テーブル用。他のオブジェクトでは異なります。)
* -- 直前の権限に関するグラントオプション

/yyyy -- この権限を付与したロール
```

上記の例では、mytableテーブルを作成し、次のコマンドを実行した後にmiriamユーザに表示されます。

```
GRANT SELECT ON mytable TO PUBLIC;
GRANT SELECT, UPDATE, INSERT ON mytable TO admin;
GRANT SELECT (coll), UPDATE (coll) ON mytable TO miriam_rw;
```

テーブル以外のオブジェクトでは、その権限を表示することができる他の¥dコマンドがあります。

あるオブジェクトの"アクセス権限"列が空の場合、そのオブジェクトはデフォルトの権限を持っていることを意味します（つまり権限フィールドがNULL）。所有者に対しては、デフォルト権限として常に全ての権限が含まれていますが、オブジェクトの種類によっては、PUBLICの権限が含まれていることがあります。これについては既に説明した通りです。オブジェクトに対する最初のGRANTまたはREVOKEが実行されるとデフォルトの権限がインスタンス化され（例えば{miriam=arwdDxt/miriam}のように作成されます）、与えられた要求によってこれらを変更します。同様に、デフォルト以外の権限を持つ列に対してのみ"列アクセス権限"内で項目が表示されます。（注意：この目的のために、"デフォルト権限"とはそのオブジェクト種類の組み込みのデフォルト権限を意味します。ALTER DEFAULT PRIVILEGESコマンドにより影響をうける権限を持つオブジェクトは常に、このALTERによる影響を含む、明示的な権限項目を持って表示されます。）

所有者が暗黙のうちに持つグラントオプションは、上記のアクセス権限の表示に出力されていない点に注目してください。
*は、グラントオプションが明示的に誰かに与えられた場合にのみ出力されます。

例

filmsテーブルにデータを追加する権限を全てのユーザに与えます。

```
GRANT INSERT ON films TO PUBLIC;
```

kindsビューにおける利用可能な全ての権限を、manuelユーザに与えます。

```
GRANT ALL PRIVILEGES ON kinds TO manuel;
```

上のコマンドをスーパーユーザやkindsの所有者が実行した場合は、全ての権限が付与されますが、他のユーザが実行した場合は、そのユーザがグラントオプションを持つ権限のみが付与されることに注意してください。

adminsロール内のメンバ資格をjoeユーザに与えます。

```
GRANT admins TO joe;
```

互換性

標準SQLに従い、ALL PRIVILEGES内のPRIVILEGESは必須です。標準SQLでは、1つのコマンドによる複数オブジェクトへの権限の設定はサポートしていません。

PostgreSQLでは、オブジェクトの所有者は、自身が持つ権限を取り消すことができます。例えば、テーブル所有者は自身のINSERT、UPDATE、DELETE、TRUNCATE権限を取り消すことで、自分にとってそのテーブルが読み取り専用になるよう変更することができます。これは、標準SQLでは不可能です。PostgreSQLでは、所有者の権限を、所有者自身により与えられたものとして扱っているため、同様に所有者自身で権限を取り消すことができるようになっています。標準SQLでは、所有者の権限は仮想的な"_SYSTEM"実体によって与えられたものとして扱っています。そのため、所有者はその権限を取り消すことができませ

ん。

標準SQLにしたがうと、グラントオプションはPUBLICに対して与えることができます。 PostgreSQLではグラントオプションはロールに対して与えることのみをサポートしています。

標準SQLでは、文字セット、集合、翻訳といったその他の種類のオブジェクトに対して、USAGE権限を付与することができます。

標準SQLでは、シーケンスは、PostgreSQLにおけるnextval関数と等価なNEXT VALUE FOR式の使用を制御するUSAGE権限のみを持ちます。 シーケンスに関するSELECT権限とUPDATE権限はPostgreSQLの拡張です。 シーケンスに関するUSAGE権限がcurrval関数にも適用される点もPostgreSQLの拡張です(この関数自体が拡張です)。

データベース、テーブル空間、スキーマ、言語についての権限はPostgreSQLの拡張です。

関連項目

[REVOKE](#), [ALTER DEFAULT PRIVILEGES](#)

前のページ FETCH	ホーム 上に戻る	次のページ INSERT

INSERT

名前

INSERT -- テーブルに新しい行を作成する

概要

```
[ WITH [ RECURSIVE ] with_query [, ...] ]  
INSERT INTO table_name [ ( column_name [, ...] ) ]  
    { DEFAULT VALUES | VALUES ( { expression | DEFAULT } [, ...] ) [, ...] | query }  
    [ RETURNING * | output_expression [ [ AS ] output_name ] [, ...] ]
```

説明

INSERTはテーブルに新しい行を挿入します。値式を使用して行（複数可）を挿入するだけでなく、問い合わせの結果を使って0行以上の行を挿入することもできます。

対象の列名はどのような順番でも指定できます。列名リストが指定されなかった場合は、宣言時の順番に並べたテーブル内の全ての列がデフォルトとなります。また、VALUES句やqueryでN列のみが指定された場合は、先頭のN列の名前がデフォルトとなります。VALUES句やqueryで提供される値は、明示的または暗黙的な列リストと左から右への順で関連付けられます。

明示的または暗黙的な列リストにない各列にはデフォルト値（デフォルト値が宣言されていればその値、未宣言ならばNULL）が挿入されます。

各列の式が正しいデータ型でない場合は、自動的に型の変換が行われます。

省略可能なRETURNING句により、INSERTは実際に挿入された各行に基づいて計算された値を返すようになります。これは、通番のシーケンス番号など、デフォルトで与えられた値を取り出す時にかなり便利です。しかし、そのテーブルの列を使用した式を指定することができます。RETURNINGリストの構文はSELECTの出力リストと同一です。

テーブルに行を追加するには、そのテーブルに対してINSERT権限を持っている必要があります。列リストが指定された場合は、列挙された列に対するINSERT権限のみが必要です。RETURNING句を使用するには、RETURNINGで使用するすべての列に対するSELECT権限が必要です。queryを使用して問い合わせ結果を元に行を挿入する場合は当然ながら、その問い合わせ内で使われる全てのテーブルまたは列に対してSELECT権限を持っている必要があります。

パラメータ

with_query

WITH句により、INSERT問い合わせ内で名前により参照することができる1つ以上の副問い合わせを指定することができます。 [項7.8](#)と[SELECT](#)を参照してください。

query (SELECT文) ではまた、WITH句を含めることができます。こうした場合、with_queryの集合との両方をquery内で参照することができます。しかしより綿密に入れ子にされますので、第二の問い合わせは述語を取ります。

table_name

既存のテーブルの名前です（スキーマ修飾名も可）。

column_name

*table_name*で指名されたテーブル内の列名です。列名を副フィールドの名前や配列の添え字で修飾することができます。（複合型の列の一部のフィールドのみを挿入しても他のフィールドはNULLのままです。）

DEFAULT VALUES

全ての列に、それぞれのデフォルト値が設定されます。

expression

対応する列に代入する有効な式または値を指定します。

DEFAULT

対応する列にデフォルト値を設定します。

query

挿入する行を提供する問い合わせ（`SELECT`文）を指定します。 構文の説明については[SELECT](#)文を参照してください。

output_expression

各行が挿入された後、計算され`INSERT`により返される式です。 この式には`table_name`で指名されたテーブルの任意の列名を使用することができます。 挿入された行のすべての列を返す場合は`*`と記載してください。

output_name

返される列で使用される名前です。

出力

正常に終了すると、`INSERT`は以下のようなコマンドタグを返します。

```
INSERT oid count
```

`count`は挿入された行数です。 `count`が正確に1であり、対象のテーブルがOIDを持つ場合、`oid`は挿入された行に割り当てられたOIDです。 その他の場合、`oid`は0となります。

`INSERT`コマンドが`RETURNING`句を持つ場合、その結果は、`RETURNING`リストで定義した列と値を持ち、そのコマンドで挿入された行全体に対して計算を行う`SELECT`文の結果と似たものになるでしょう。

例

`films` テーブルに1行を挿入します。

```
INSERT INTO films VALUES
('UA502', 'Bananas', 105, '1971-07-13', 'Comedy', '82 minutes');
```

この例では、`len`列を省略しています。 したがって、ここにはデフォルト値`NULL`が入ります。

```
INSERT INTO films (code, title, did, date_prod, kind)
VALUES ('T_601', 'Yojimbo', 106, '1961-06-16', 'Drama');
```

3つ目の例では、日付列に対して値ではなく`DEFAULT`を使用します。

```
INSERT INTO films VALUES
('UA502', 'Bananas', 105, DEFAULT, 'Comedy', '82 minutes');
INSERT INTO films (code, title, did, date_prod, kind)
VALUES ('T_601', 'Yojimbo', 106, DEFAULT, 'Drama');
```

全てデフォルト値からなる行を挿入します。

```
INSERT INTO films DEFAULT VALUES;
```

複数行の`VALUES`構文を使用して複数行を挿入します。

```
INSERT INTO films (code, title, did, date_prod, kind) VALUES
('B6717', 'Tampopo', 110, '1985-02-10', 'Comedy'),
('HG120', 'The Dinner Game', 140, DEFAULT, 'Comedy');
```

この例では、`films` テーブルと同じ列レイアウトを持つ`tmp_films` テーブルから`films` テーブルへいくつか行を挿入します。

```
INSERT INTO films SELECT * FROM tmp_films WHERE date_prod < '2004-05-07';
```

この例では、配列型の列に挿入しています。

```
-- 三目並べ用の3×3マスのゲーム盤を作成します。
INSERT INTO tictactoe (game, board[1:3][1:3])
VALUES (1, '{{" "," "," "," "},{ " "," "," "," "},{ " "," "," "," "}}');
--上の例の添え字は本当は必要ありません。
INSERT INTO tictactoe (game, board)
VALUES (2, '{{X," "," "},{ " ",O," "},{ " ",X," "}}');
```

`distributors` テーブルに単一行を挿入し、その`DEFAULT`句により生成されたシーケンス番号を返します。

```
INSERT INTO distributors (did, dname) VALUES (DEFAULT, 'XYZ Widgets')
```

```
RETURNING did;
```

Acme社の顧客を担当する営業担当者の売り上げ数を増やし、ログテーブルに更新行全体と更新時刻を記録します。

```
WITH upd AS (  
  UPDATE employees SET sales_count = sales_count + 1 WHERE id =  
    (SELECT sales_person FROM accounts WHERE name = 'Acme Corporation')  
  RETURNING *  
)  
INSERT INTO employees_log SELECT *, current_timestamp FROM upd;
```

互換性

INSERTは標準SQLに準拠します。ただし、RETURNING句とINSERTでWITHが可能であることはPostgreSQLの拡張です。また、標準SQLでは、列名リストが省略された時に、VALUES句またはqueryで一部の列のみを指定することはできません。

制限される可能性があるquery句の機能については、[SELECT](#)にて記述されています。

[前のページ](#)

GRANT

[ホーム](#)

[上に戻る](#)

[次のページ](#)

LISTEN

LISTEN

名前

LISTEN -- 通知を監視する

概要

```
LISTEN channel
```

説明

`LISTEN`は現在のセッションを、通知チャンネル`channel`のリスナとして登録します。現在のセッションが既に指定した通知チャンネルのリスナとして登録されている場合は、何も起こりません。

このセッションまたは同一データベースに接続している別のセッションによって`NOTIFY channel`が実行されると、現在その通知チャンネルを監視している全てのセッションに対して通知されます。次に、各セッションは接続中のクライアントアプリケーションにこれを通知します。

`UNLISTEN`コマンドを使って、セッションに登録された指定通知チャンネルを解除できます。また、セッションの監視登録はそのセッションが終了した時点で自動的に削除されます。

クライアントアプリケーションが通知イベントを検出する方法は、使用しているPostgreSQLアプリケーションプログラミングインタフェースに依存します。libpqライブラリを使用するアプリケーションでは、`LISTEN`を通常のSQLコマンドとして発行し、その後、`PQnotifies`ルーチンを定期的呼び出して通知イベントが受信されたかどうかを調べる必要があります。libpqtc等その他のインタフェースには、通知イベントを扱うより高レベルな方法が用意されています。実際、libpqtcを使ったアプリケーションの場合、プログラマが`LISTEN`や`UNLISTEN`を直接発行する必要すらありません。詳細については、使用中のインタフェースのドキュメントを参照してください。

[NOTIFY](#)には、`LISTEN`および`NOTIFY`についてのより広範な説明があります。

パラメータ

`channel`

通知チャンネルの名前です（任意の識別子）。

注釈

`LISTEN`はトランザクションのコミット時に影響します。`LISTEN`または`UNLISTEN`がトランザクション内で実行され、それがロールバックされた場合、監視している通知チャンネルの集合には変化はありません。

`LISTEN`を実行したトランザクションでは二相コミットの準備を行うことはできません。

例

psqlから、監視/通知処理の設定と実行を行います。

```
LISTEN virtual;
NOTIFY virtual;
Asynchronous notification "virtual" received from server process with PID 8448.
```

互換性

標準SQLに`LISTEN`はありません。

関連項目

[NOTIFY](#), [UNLISTEN](#)

[前のページ](#)
INSERT

[ホーム](#)
[上に戻る](#)

[次のページ](#)
LOAD

LOAD

名前

LOAD -- 共有ライブラリファイルの読み込みを行う

概要

```
LOAD 'filename'
```

説明

LOAD コマンドは、共有ライブラリファイルをPostgreSQLサーバのアドレス空間にロードします。そのファイルが既にロードされていると、このコマンドは何も行いません。C関数を含む共有ライブラリファイルは、その中の1つの関数が呼び出された時に常に、自動的にロードされます。このため通常、明示的なLOADは、関数群を提供するのではなく"フック"を通してサーバの動作を変更するライブラリをロードするためだけに必要となります。

共有ライブラリのファイル名は、[CREATE FUNCTION](#)における共有ライブラリ名と同じ方法で特定されます。特に、検索パスやシステム標準の共有ライブラリファイル名拡張子の自動付与に依存することができます。詳細は[項35.9](#)を参照してください。

非特権ユーザは\$libdir/plugins/にあるライブラリファイルのみをLOADさせることができます。つまり、指定したfilenameはこの文字列から始まらなければなりません。（このディレクトリ以下に確実に"安全な"ライブラリのみをインストールすることはデータベース管理者の責任です。）

互換性

LOADはPostgreSQLの拡張です。

関連項目

[CREATE FUNCTION](#)

[前のページ](#)

[LISTEN](#)

[ホーム](#)

[上に戻る](#)

[次のページ](#)

[LOCK](#)

LOCK

名前

LOCK -- テーブルをロックする

概要

```
LOCK [ TABLE ] [ ONLY ] name [ * ] [, ...] [ IN lockmode MODE ] [ NOWAIT ]
```

*lockmode*には以下のいずれかが入ります。

```
ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE  
| SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE
```

説明

LOCK TABLEはテーブルレベルのロックを取得します。必要であれば競合するロックが解除されるまで待機します。NOWAITが指定された場合は、対象のロックを取得できなくても待機しません。この場合、すぐにロックが取得できなければ、このコマンドを中止し、エラーを出力します。ロックは、一度取得されると現行のトランザクションが完了するまで保持されます（UNLOCK TABLEといったコマンドはありません。ロックが解除されるのは常にトランザクションの終了時です）。

テーブルを参照するコマンドのために自動的にロックを取得する場合、PostgreSQLは常に使用可能な一番弱いロックモードを使用します。LOCK TABLEはより制限の強いロックが必要な場合のために用意されています。例えば、隔離レベルREAD COMMITTEDでトランザクションを実行するアプリケーションで、トランザクションの間中、テーブルのデータを確実に安定させる必要がある場合を考えます。この場合、問い合わせ実行前にテーブル全体にSHAREロックモードを使用します。これにより、データが同時に変更されるのを防ぎ、それ以降のテーブルの読み取りを安定させることができます。なぜならSHAREロックモードは書き込み側が取得するROW EXCLUSIVEロックと競合するので、LOCK TABLE *name* IN SHARE MODE文は、ROW EXCLUSIVEを保持しているトランザクションがコミットまたはロールバックされるのを待つからです。したがって、一度ロックを取得してしまえば、コミットされていない状態の書き込みは存在しないことになります。さらに、ロックを解除するまで他のアプリケーションは書き込みを開始することができません。

REPEATABLE READまたはSERIALIZABLE隔離レベルで実行しているトランザクションで同様の効果を得るには、全てのSELECT文とデータを更新する文を実行する前にLOCK TABLE文を実行する必要があります。REPEATABLE READまたはSERIALIZABLEトランザクション側から参照するデータの状態は、最初にSELECT文またはデータ更新用文が開始された時点で固定されます。後からトランザクション内でLOCK TABLEを実行した場合も同時書き込みを防ぐことはできますが、トランザクションの読み込み対象データの値がコミットされた最新の値であることは保証されません。

このようなトランザクションでテーブル内データを変更する場合は、SHAREモードではなくSHARE ROW EXCLUSIVEロックモードを使用する必要があります。これによって、この種のトランザクションが同時に複数実行されることがなくなります。SHARE ROW EXCLUSIVEを使用しないと、デッドロックが発生する可能性があります。2つのトランザクションの両方が、SHAREモードを取得していながら、実際の更新に必要なROW EXCLUSIVEモードを取得できない状態になる可能性があるためです（トランザクション自身が所有しているロック間は競合しないので、トランザクションはSHAREモードを保持している間もROW EXCLUSIVEを獲得することができます。しかし、他のトランザクションがSHAREモードを保持している時にはROW EXCLUSIVEを獲得することはできません）。デッドロックを回避するには、全てのトランザクションが、必ず同一オブジェクトに対して同一の順番でロックを取得するようにしてください。また、1つのオブジェクトに対して複数のロックモードを呼び出す場合、トランザクションは常に最も制限の強いモードを最初に取得しなければなりません。

ロックモードとロック取得方針についてのより詳細については[項13.3](#)を参照してください。

パラメータ

name

ロックする既存のテーブルの名前です（スキーマ修飾名も可）。テーブル名の前にONLYが指定された場合、そのテーブルのみをロックします。ONLYが指定されない場合、そのテーブルとすべての子テーブル（もしあれば）をロックします。省略することもできますが、テーブル名の後に*を指定することで、明示的に継承するテーブルも含まれることを示すことができます。

LOCK a, b;というコマンドはLOCK TABLE a; LOCK TABLE b;と同じです。テーブルは1つひとつLOCKで指定された順番

でロックされます。

lockmode

ロックモードには、取得するロックと競合するロックを指定します。 ロックモードについては、[項13.3](#)で説明します。

ロックモードを指定しない場合、最も制限が強いACCESS EXCLUSIVEが使用されます。

NOWAIT

LOCK TABLEが競合するロックの解放まで待機しないことを指定します。 指定したロックがすぐに取得できない場合、トランザクションはアボートされます。

注釈

LOCK TABLE ... IN ACCESS SHARE MODEには、対象テーブルのSELECT権限が必要です。 他の形式のLOCKには、テーブルレベルのUPDATE、DELETE、TRUNCATE権限を持たなければなりません。

LOCK TABLEはトランザクションブロックの外側では意味がありません。 文が完了するまでしかロックは保持されません。 したがってPostgreSQLはLOCKがトランザクションブロックの外側で使用された場合にエラーを報告します。 トランザクションブロックを定義するためにはBEGINおよびCOMMIT（またはROLLBACK）を使用してください。

LOCKが扱うのはテーブルレベルのロックのみです。 そのため、モード名にROWが含まれるのは適切ではありません。 これらのモード名によって、普通は、ロックされたテーブル内で行レベルのロックを取得できると思ってしまうでしょう。 また、ROW EXCLUSIVEモードは共有可能なロックです。 LOCK TABLEに関しては、全てのロックモードが同じ意味を持っていることに注意してください。 違うのは、どのモードがどのモードと競合するかという規則だけです。 行レベルでのロックを獲得する方法についてはSELECTマニュアルページの[項13.3.2](#)と[FOR UPDATE/FOR SHARE句](#)を参照してください。

例

外部キーテーブルへの挿入を行う際に、プライマリーキーテーブルへのSHARE ロックを獲得します。

```
BEGIN WORK;
LOCK TABLE films IN SHARE MODE;
SELECT id FROM films
  WHERE name = 'Star Wars: Episode I - The Phantom Menace';
-- レコードがなければROLLBACKしてください。
INSERT INTO films_user_comments VALUES
  (id, 'GREAT! I was waiting for it for so long!');
COMMIT WORK;
```

削除操作を行う際にプライマリーキーテーブルのSHARE ROW EXCLUSIVEロックを取得します。

```
BEGIN WORK;
LOCK TABLE films IN SHARE ROW EXCLUSIVE MODE;
DELETE FROM films_user_comments WHERE id IN
  (SELECT id FROM films WHERE rating < 5);
DELETE FROM films WHERE rating < 5;
COMMIT WORK;
```

互換性

標準SQLにはLOCK TABLEはありません。 その代わりにトランザクションの同時性レベルを指定するSET TRANSACTIONが使用されます。 PostgreSQLはこのコマンドもサポートしています。 詳細は[SET TRANSACTION](#)を参照してください。

ACCESS SHARE、ACCESS EXCLUSIVE、SHARE UPDATE EXCLUSIVEロックモードを除き、PostgreSQLのロックモードとLOCK TABLE構文はOracleのものと互換性があります。

[前のページ](#)
LOAD

[ホーム](#)
[上に戻る](#)

[次のページ](#)
MOVE

MOVE

名前

MOVE -- カーソルの位置を決める

概要

```
MOVE [ direction [ FROM | IN ] ] cursor_name
```

ここで`direction`は空または以下のいずれかを取ることができます。

```
NEXT
PRIOR
FIRST
LAST
ABSOLUTE count
RELATIVE count
count
ALL
FORWARD
FORWARD count
FORWARD ALL
BACKWARD
BACKWARD count
BACKWARD ALL
```

説明

`MOVE`はデータを取り出すことなくカーソルの位置を変更します。 `MOVE`は`FETCH`コマンドとまったく同じように動作しますが、カーソルの位置を変えるだけで行を返しません。

`MOVE`コマンドのパラメータは`FETCH`コマンドと同一です。 構文と使用方法についての詳細は[FETCH](#)を参照してください。

出力

正常に終了すると、`MOVE`は以下の形式のコマンドタグを返します。

```
MOVE count
```

`count`は同じパラメータを与えた`FETCH`コマンドが返すはずの行数です（この値は0の場合もあります）。

例

```
BEGIN WORK;
DECLARE liahona CURSOR FOR SELECT * FROM films;

-- 最初の5行を飛ばします。
MOVE FORWARD 5 IN liahona;
MOVE 5

-- liahonaカーソル内の6行目を抽出します。
FETCH 1 FROM liahona;
code | title | did | date_prod | kind | len
-----+-----+-----+-----+-----+-----
P_303 | 48 Hrs | 103 | 1982-10-22 | Action | 01:37
(1 row)

-- liahonaカーソルを閉じ、トランザクションを終了します。
CLOSE liahona;
COMMIT WORK;
```

互換性

標準SQLにはMOVE文はありません。

関連項目

[CLOSE](#), [DECLARE](#), [FETCH](#)

[前のページ](#)
LOCK

[ホーム](#)
[上に戻る](#)

[次のページ](#)
NOTIFY

NOTIFY

名前

NOTIFY -- 通知を生成する

概要

```
NOTIFY channel [ , payload ]
```

説明

NOTIFY コマンドは、現在のデータベース内で事前にその指定チャンネル名についてLISTEN channel コマンドを実行したクライアントアプリケーションに省略可能な"ペイロード"を持つ通知イベントを送ります。

NOTIFYは同一のPostgreSQLデータベースにアクセスするプロセスの集合に対する単純なプロセス間通信の仕組みを提供します。通知の際にペイロード文字列を送信することができます。また、データベース内のテーブルを使用して通知者から（1つまたは複数の）リスナに追加的なデータを渡すことにより、構造化されたデータを渡す高度な仕組みを構築することができます。

通知イベントとしてクライアントに渡される情報には、通知チャンネル名と通知を行うセッションのサーバプロセスのPID、ペイロード文字列(指定されていなければ空文字列)が含まれます。

与えられたデータベースにおいて使用される通知チャンネル名とその意味についての定義は、データベース設計者に任されています。通知チャンネル名には、データベース内のテーブル名と同じものを使用するのが一般的です。通知イベントは基本的に"私はこのテーブルを変更しました。変更された箇所を確認してください"ということの意味するものです。しかし、NOTIFY コマンドとLISTEN コマンドでは、そのような関連付けは強制されていません。例えば、データベース設計者は、1つのテーブルに対する異なる種類の変更を通知するために、複数の異なる通知チャンネル名を使用することができます。他の方法としてペイロード文字列を使用して各種様々な状況に対応させることもできます。

特定のテーブルが変更されたことを通知するためにNOTIFYを使用する場合、NOTIFYをテーブル更新時に発行されるルール内に配置すると便利です。この場合、通知はテーブルが変更された時に自動的に行われるので、アプリケーションプログラマが通知の実行を忘れるといった事故を防ぐことができます。

NOTIFYとSQLトランザクションの間には、いくつかの重要な相互作用があります。まず、NOTIFYがトランザクション内部で実行された場合、通知イベントはトランザクションがコミットされない限り配送されません。トランザクションがアボートされた場合、NOTIFYだけでなく、そのトランザクション内で行われたコマンドが全て無効化されるので、これは妥当といえます。しかし、通知イベントが即座に配送されることを期待していた場合、当惑するかもしれません。次に、監視中のセッションがトランザクション処理中に通知信号を受け取った場合、そのトランザクションが（コミットもしくはアボートされて）完了するまで、通知イベントは接続しているクライアントに配送されません。この理由も同じです。トランザクションに通知が配送された後にそのトランザクションがアボートされた場合、何とかして通知を取り消したくなりますが、サーバはいったんクライアントに送信した通知を"取り戻す"ことはできません。したがって、通知イベントはトランザクションとトランザクションの合間에만配送されます。この結果、実時間レベルでシグナルを処理するため、NOTIFYを使用するアプリケーションではトランザクション処理を短くしておかなければなりません。

同じペイロード文字列を持つ同じチャンネル名が同一トランザクション内から複数回通知される場合、データベースサーバは1つの通知のみを伝えるように決定します。一方個別のペイロード文字列を持つ通知は常に個別の通知として伝えられます。同様に別のトランザクションからの通知が1つの通知にまとめられることは決してありません。重複する通知インスタンスを後で削除する場合は例外ですが、NOTIFYは同一トランザクションからの通知は送信された順番に配送されることを保証します。また異なるトランザクションからのメッセージがトランザクションのコミット順で配送されることも保証します。

NOTIFYを実行するクライアント自身が、その通知の通知チャンネルを監視していることはよくあります。この場合、同じ通知名を監視する他のセッションに対するのと同じように通知イベントが戻ってきます。アプリケーションのロジックにもよりますが、これは無駄な作業になることがあります。例えば、そのセッションが書き出したばかりのデータベースに対する更新を調べるためにテーブルの再読み込みを行う場合などが考えられます。通知イベントメッセージ内にある通知元セッションのサーバプロセスのPIDと、libpqで得られる自分自身のPIDが同じかどうかかに注意することで、こういった余計な作業を回避できます。PIDが同じであれば、その通知イベントは自分自身から跳ね返ってきたものであり、無視することができます。

パラメータ

channel

シグナルとして送られる通知チャンネル名です（任意の識別子）。

payload

通知と一緒に通信される"ペイロード"文字列です。これは単一の文字列リテラルとして指定されなければなりません。デフォルトの設定では、8000バイト未満でなければなりません。（バイナリデータまたは大規模な情報を渡さなければならないのであれば、データベーステーブル内に格納しレコードのキーを送信することが最善です。）

注釈

送信済みだがすべての監視セッションで処理されていない通知を保持するためのキューが存在します。このキューがいっぱいになると、NOTIFYを呼び出すトランザクションのコミットに失敗します。キューはかなり大きなもの(標準のインストレーションで8ギガバイト)であり、ほとんどすべての環境で十分な大きさであるはずです。しかしセッションがNOTIFYを実行した後に長期間のトランザクションに入った場合、キューから吐き出すところなくなります。キューの半分までたまると、ログファイル内に吐き出しを妨げるセッションを指し示す警告が現れるようになります。この場合、吐き出し処理が進むように、確実にそのセッションでその現在のトランザクションを完了させるようにしなければなりません

NOTIFYを実行したトランザクションでは二相コミットを準備することはできません。

pg_notify

通知を送信するためにpg_notify(text,text)関数を使用することもできます。この関数は第1引数としてチャンネル名、第2引数としてペイロードを取ります。不定のチャンネル名、ペイロードで作業しなければならない場合は、NOTIFYコマンドよりこの関数を使用する方がかなり簡単です。

例

psqlから監視/通知処理の設定と実行を行います。

```
LISTEN virtual;
NOTIFY virtual;
Asynchronous notification "virtual" received from server process with PID 8448.
NOTIFY virtual, 'This is the payload';
Asynchronous notification "virtual" with payload "This is the payload" received from server process with PID 8448.

LISTEN foo;
SELECT pg_notify('fo' || 'o', 'pay' || 'load');
Asynchronous notification "foo" with payload "payload" received from server process with PID 14728.
```

互換性

標準SQLにはNOTIFYはありません。

関連項目

[LISTEN](#), [UNLISTEN](#)

[前のページ](#)
MOVE

[ホーム](#)
[上に戻る](#)

[次のページ](#)
PREPARE

PREPARE

名前

PREPARE -- 実行する文を準備する

概要

```
PREPARE name [ ( data_type [, ...] ) ] AS statement
```

説明

PREPAREはプリペアド文を作成します。プリペアドは、性能を最適化するために利用可能なサーバ側オブジェクトです。**PREPARE**文を実行すると、指定された問い合わせの構文解析、書き換えが行われます。その後、**EXECUTE**文が発行された際に、プリペアド文は実行計画が作成され、実行されます。この作業の分割により構文解析作業が繰り返されることを防止でき、さらに、特定のパラメータ値に合わせた実行計画を提供することができます。

プリペアド文はパラメータ、すなわち文が実行される時に代入される値を取ることができます。プリペアド文を作成する時には`$1`や`$2`などを使用して、位置によりパラメータを参照してください。省略可能ですが、対応するパラメータのデータ型のリストを指定することもできます。パラメータのデータ型の指定がない、または、`unknown`と宣言されている場合、型はパラメータが使用される文脈より(可能ならば)推測されます。文の実行時には、**EXECUTE**文内にこれらのパラメータの実際の値を指定します。詳細は[EXECUTE](#)を参照してください。

プリペアド文は現在のデータベースセッションの期間中にのみ保持されます。セッションが終了すると、プリペアド文は破棄されます。そのため、再び利用する場合は、再作成する必要があります。また、これは、1つのプリペアド文を複数のデータベースクライアントから使用することはできないことを意味します。ただし、各クライアントが個別にプリペアド文を作成することはできます。プリペアド文を手作業で削除するには、[DEALLOCATE](#)コマンドを使用します。

プリペアド文の利点を最大限に発揮できるのは、単一のセッションで同様の問い合わせを多数実行する場合です。パフォーマンスの違いは、文の書き換えや実行計画が複雑なほど顕著になるでしょう。例えば、問い合わせに多数のテーブルの結合が含まれている場合や、いくつかのルールを適用しなければならない場合などが考えられます。書き換えおよび実行計画が比較的単純で、実行コストが高い文の場合は、プリペアド文の効果はそれほど現れないでしょう。

パラメータ

name

個々のプリペアド文に与えられる任意の名前です。この名前は、1つのセッション内で一意でなければいけません。プリペアド文の実行および削除の時に、この名前が使用されます。

data_type

プリペアド文に対するパラメータのデータ型です。特定のパラメータのデータ型の指定がない、または、`unknown`と指定された場合、パラメータが使用される文脈から推測されます。プリペアド文自体の中でこのパラメータを参照する時は、`$1`、`$2`などを使用します。

statement

`SELECT`、`INSERT`、`UPDATE`、`DELETE`、`VALUES`文のいずれかです。

注釈

プリペアド文が何度も実行される場合、サーバは最終的に、毎回計画を再作成するのではなく、一般的な計画を保存し再利用することに決定する可能性があります。プリペアド文がパラメータを持たない場合、これは即座に起こります。パラメータを持つ場合は、一般的な計画がパラメータ値に合わせた計画に比べて大して高価ではないように見える場合に、これが起こります。一般化された計画は典型的には、問い合わせの性能が提供された特定のパラメータ値に大きな影響を受けない場合のみに選ばれます。

プリペアド文に対してPostgreSQLが使用する問い合わせ計画を検証するためには、[EXPLAIN](#)を使用してください。一般的な計

画が使用される場合には、`$n`というパラメータ記号が含まれ、独自計画が使用される場合は現在のパラメータの実値で置換されます。

問い合わせの実行計画や問い合わせの最適化のためにPostgreSQLが収集する統計に関する詳細は、[ANALYZE](#)のドキュメントを参照してください。

[pg_prepared_statements](#) システムビューを問い合わせることによりセッションで利用可能なプリペアド文をすべて確認することができます。

例

INSERT 文に対してプリペアド文を作成し、実行します。

```
PREPARE fooplan (int, text, bool, numeric) AS
    INSERT INTO foo VALUES($1, $2, $3, $4);
EXECUTE fooplan(1, 'Hunter Valley', 't', 200.00);
```

SELECT 文に対してプリペアド文を作成し、実行します。

```
PREPARE usrrptplan (int) AS
    SELECT * FROM users u, logs l WHERE u.usrid=$1 AND u.usrid=l.usrid
    AND l.date = $2;
EXECUTE usrrptplan(1, current_date);
```

第2パラメータのデータ型が指定されていないことに注目してください。 このため`$2`が使用される文脈からデータ型が推測されます。

互換性

標準SQLには`PREPARE`文が含まれていますが、埋め込みSQLでの使用に限られています。 また、標準SQLの`PREPARE`文では多少異なる構文が使用されます。

関連項目

[DEALLOCATE](#), [EXECUTE](#)

[前のページ](#)

NOTIFY

[ホーム](#)
[上に戻る](#)

[次のページ](#)

PREPARE TRANSACTION

PREPARE TRANSACTION

名前

PREPARE TRANSACTION -- 二相コミット用に現在のトランザクションを準備する

概要

```
PREPARE TRANSACTION transaction_id
```

説明

PREPARE TRANSACTIONは、二相コミット用に現在のトランザクションを準備します。このコマンドの後、トランザクションは現在のセッションと関連しなくなります。トランザクションの状態は完全にディスク上に保存され、コミット要求前にデータベースがクラッシュしてしまったとしても、ほとんど確実に正常にコミットできるようになります。

準備された後、そのトランザクションを[COMMIT PREPARED](#)や[ROLLBACK PREPARED](#)によりコミットまたはロールバックすることができます。元のトランザクションを実行したセッションだけではなく、任意のトランザクションからこれらのコマンドを発行することができます。

コマンドを発行したセッションから見ると、PREPARE TRANSACTIONはROLLBACKコマンドと似たような動作をします。実行した後、実行中の現在のトランザクションはなくなり、準備したトランザクションの効果は不可視になります。（そのトランザクションがコミットされた場合に効果が可視になります。）

何らかの原因でPREPARE TRANSACTIONコマンドが失敗した場合、ROLLBACKが行われます。つまり、現在のトランザクションが取り消されます。

パラメータ

transaction_id

後でCOMMIT PREPAREDやROLLBACK PREPAREDでトランザクションを識別するための任意の識別子です。この識別子は文字列リテラルでなければなりません。また、200バイト未満でなければなりません。また、その時点で準備されたトランザクションとして使用されている他の識別子と同じものは使用できません。

注釈

PREPARE TRANSACTIONはアプリケーションや対話式のセッションでの使用を目的としていません。この目的は、外部トランザクションマネージャにより、複数のデータベースやその他のトランザクションを持つリソースを跨るグローバルなトランザクションを原始的に実現できるようにすることです。トランザクションマネージャを作成しているのでなければ、おそらくPREPARE TRANSACTIONを使用してはなりません。

このコマンドはトランザクションブロック内で使用しなければなりません。トランザクションブロックを始めるには、[BEGIN](#)を使用してください。

現時点では、一時テーブルを含む操作を行うトランザクションやカーソルをWITH HOLDで作成するトランザクション、LISTENまたはUNLISTENを実行するトランザクションをPREPAREさせることはできません。準備したトランザクションで便利に使用するには、これらの機能は現在のセッションにあまりに強く結びついているためです。

トランザクションで何らかの実行時パラメータが（LOCALオプションなしの）SETで設定されている場合、その影響はPREPARE TRANSACTIONの後も残ります。また、COMMIT PREPAREDやROLLBACK PREPAREDの後では、その影響はありません。したがって、この意味では、PREPARE TRANSACTIONはROLLBACKよりCOMMITと似た動きであるといえます。

その時点で利用できるすべての準備されたトランザクションは[pg_prepared_xacts](#)システムビューで列挙されます。

注意

トランザクションを長期間準備された状態のままとすることは勧められません。これは格納領域を回収するVACUUM機能と相互作用し、極端な場合では、トランザクションの周回（[項23.1.5](#)参照）を回避するためにデータベースを停止させてしまいます。またトランザクションが保持している何からのロックを保持し続けていることにも注意してください。外部トランザクションマネージャが他のデータベースがコミットの準備をしたと検証した後すぐに、準備されたトランザクションは通常コミットまたはロールバックされることがこの機能の想定している使用方法です。

準備されたトランザクションを追跡し、その終了を即座に確認できるように外部トランザクションマネージャを設定していない場合、[max_prepared_transactions](#)をゼロに設定して準備されたトランザクション機能を無効にしておくことが最善です。こうすれば事故により準備されたトランザクションが作成され、忘れられた結果問題を引き起こすことを防ぐことができます。

例

二相コミット用に現在のトランザクションを準備します。トランザクション識別子として`foobar`を使用します。

```
PREPARE TRANSACTION 'foobar';
```

互換性

PREPARE TRANSACTIONはPostgreSQLの拡張です。これは外部のトランザクション管理システムによる利用を意図したものです。トランザクション管理システムの一部（X/Open XAなど）は標準化されていますが、こうしたシステムのSQL側は標準化されていません。

関連項目

[COMMIT PREPARED](#), [ROLLBACK PREPARED](#)

前のページ PREPARE	ホーム 上に戻る	次のページ REASSIGN OWNED

REASSIGN OWNED

名前

REASSIGN OWNED -- あるデータベースロールにより所有されたデータベースオブジェクトの所有権を変更する

概要

```
REASSIGN OWNED BY old_role [, ...] TO new_role
```

説明

REASSIGN OWNED は、old_roles のいずれかが所有するデータベースオブジェクトの所有権を new_role に変更しようシステムに指示します。

パラメータ

old_role

ロール名です。現在のデータベースでこのロールが所有するすべてのオブジェクトの所有権を new_role に割り当てます。

new_role

対象とするオブジェクトの新しい所有者となるロール名です。

注釈

REASSIGN OWNED は、1 つ以上のロールの削除準備によく使用されます。REASSIGN OWNED は現在のデータベース内のオブジェクトのみに影響を与えますので、通常は、削除対象のロールにより所有されるオブジェクトを有するデータベース毎にこのコマンドを実行する必要があります。

REASSIGN OWNED は元のロールと対象のロール上に権限が必要です。

代わりに [DROP OWNED](#) コマンドを使用して、1 つ以上のロールにより所有されるデータベースオブジェクトすべてを削除することができます。また DROP OWNED は元のロール上の権限のみを必要とすることに注意してください。

REASSIGN OWNED コマンドは、old_roles により所有されていないオブジェクトにおいて old_roles に与えられた権限には影響しません。こうした権限を取り上げるには、DROP OWNED を使用してください。

REASSIGN OWNED コマンドは、そのロールで所有される任意のデータベースの所有権には影響しません。所有権を再割り当てするためには [ALTER DATABASE](#) を使用してください。

互換性

REASSIGN OWNED 文は PostgreSQL の拡張です。

関連項目

[DROP OWNED](#), [DROP ROLE](#), [ALTER DATABASE](#)

REINDEX

名前

REINDEX -- インデックスを再構築する

概要

```
REINDEX { INDEX | TABLE | DATABASE | SYSTEM } name [ FORCE ]
```

説明

REINDEXは、インデックスのテーブルに保存されたデータを使用してインデックスを再構築し、古いインデックスのコピーと置き換えます。以下にREINDEXが使用される状況を示します。

- インデックスが破損してしまい、有効なデータがなくなった場合です。理論的には決して起こらないはずですが、実際には、ソフトウェアのバグやハードウェアの障害によりインデックスが破損することがあります。REINDEXはこの修復手段を提供します。
- インデックスが"膨張状態"、つまり、多くの空、もしくは、ほとんど空のページを持つ状態になっている場合です。この状況は、PostgreSQLのB-treeインデックスが特定のパターンでアクセスされた場合に起こり得ます。REINDEXを使って、使用されないページを取り除いた新しいインデックス作成すると、インデックスの領域消費量を減少することができます。詳細は[項23.2](#)を参照してください。
- インデックスの格納パラメータ（フィルファクタなど）を変更し、この変更が有効かどうかを確認したい場合です。
- CONCURRENTLY オプションをつけたインデックス作成が失敗し、"無効な"インデックスが残った場合です。こうしたインデックスは使用されませんが、REINDEXを使用して簡単に再作成することができます。REINDEXでは同時構築ができないことに注意してください。運用に影響を与えずにインデックスを作成するためには、インデックスを削除してからCREATE INDEX CONCURRENTLY コマンドを再発行しなければなりません。

パラメータ

INDEX

指定したインデックスを再作成します。

TABLE

指定したテーブルの全インデックスを再作成します。テーブルに2次的な"TOAST"テーブルがあっても、同様にインデックスを再作成します。

DATABASE

現在のデータベースのすべてのインデックスを再作成します。共有システムカタログのインデックスも処理されます。この構文のREINDEXをトランザクションブロック内で実行することはできません。

SYSTEM

現在のデータベースのシステムカタログに対するすべてのインデックスを再作成します。共有システムカタログのインデックスも含まれます。ユーザテーブルのインデックスは処理されません。この構文のREINDEXをトランザクションブロック内で実行することはできません。

name

インデックスを再作成するインデックス、テーブル、データベースの名前です。インデックスとテーブルはスキーマで修飾可能です。現状では、REINDEX DATABASEとREINDEX SYSTEMは現在のデータベースのインデックスのみを再作成することができます。そのため、このパラメータは現在のデータベース名と一致する必要があります。

FORCE

このオプションは廃止されました。指定されても無視されます。

注釈

ユーザテーブル上の特定のインデックスに破損の疑いがある場合、`REINDEX INDEX`を使ってそのインデックスを再構築することもできますし、`REINDEX TABLE`を使ってそのテーブルのすべてのインデックスを再構築することもできます。

システムテーブルのインデックスの破損を復旧する場合の手順はより複雑になります。この場合、システムによって破損の可能性があるインデックス自体が使用されないようにすることが重要です（実際は、このようなケースでは、破損したインデックスに依存していたため、サーバプロセスが起動時に強制終了してしまう可能性があります）。安全に復旧させるには、システムカタログ検索時のインデックスの使用を禁止する`-P`オプションを使用してサーバを起動しなければなりません。

考えられる方法の1つは次の方法です。まず、サーバを停止して、コマンドラインから`-P`オプションを指定してシングルユーザ状態のPostgreSQLサーバを起動します。そして、再構成する範囲に応じて、`REINDEX DATABASE`、`REINDEX SYSTEM`、`REINDEX TABLE`、または、`REINDEX INDEX`コマンドを発行します。範囲が不明な場合は、`REINDEX SYSTEM`を使用して、そのデータベースの全てのシステムインデックスを再構成してください。その後、シングルユーザ状態のサーバセッションを停止して、実サーバを再起動します。シングルユーザ状態のサーバインタフェースの操作方法についての詳細は、[postgres](#) マニュアルページを参照してください。

その他、コマンドラインで`-P`を指定して実サーバセッションを起動することもできます。具体的な方法は、クライアントによって異なります。しかし、libpqベースのクライアントであれば、クライアントを起動する前に`PGOPTIONS`環境変数を`-P`に設定すれば実現できます。この方法では他のクライアントを締め出す必要はありませんが、修復が終わるまで破損したデータベースへの他のユーザの接続を防止する方が良いことに注意してください。

`REINDEX`は、インデックスの中身を1から作り直すという点では、インデックスを削除してから再作成する処理と似ています。しかし、ロックに関しては異なります。`REINDEX`はインデックスの元となるテーブルの書き込みをロックしますが、読み込みはロックしません。また、処理中のインデックスに対する排他ロックを取得するので、そのインデックスを使用する読み込みはブロックされます。一方、`DROP INDEX`は瞬間的に元となるテーブルの排他ロックを取得するので、書き込みも読み込みもブロックされます。その後に行う`CREATE INDEX`では書き込みのみをロックし、読み込みはロックしません。インデックスは存在しないので、インデックスを使用する読み込みは発生しません。したがって、読み込みがブロックされることはありませんが、コストが高いシーケンシャルスキャンの使用を強制されることになります。

単一インデックスまたは単一テーブルのインデックス再作成を行うには、そのインデックスまたはテーブルの所有者でなければなりません。データベースに対するインデックス再作成を行うには、データベースの所有者でなければなりません（したがって、この所有者は他のユーザが所有するテーブルのインデックスを再作成することができます）。もちろん、スーパーユーザは常にすべてのインデックス再作成を行うことができます。

PostgreSQL 8.1より前までは、`REINDEX DATABASE`は名前から想像されるすべてのインデックスに対する処理ではなく、システムインデックスのみを処理していました。`REINDEX SYSTEM`で、この以前の処理を行わせることができます。

PostgreSQL 7.4より前までは、`REINDEX TABLE`はTOASTテーブルに対して自動的に処理を行いませんでした。そのため、別のコマンドでインデックスを再作成しなければならませんでした。現在でもこれは可能ですが、冗長です。

例

単一のインデックスを再構築します。

```
REINDEX INDEX my_index;
```

`my_table`テーブル上のすべてのインデックスを再構築します。

```
REINDEX TABLE my_table;
```

システムインデックスが有効かどうかを確認することなく、あるデータベース内の全てのインデックスを再構築します。

```
$ export PGOPTIONS="-P"
$ psql broken_db
...
broken_db=> REINDEX DATABASE broken_db;
broken_db=> ¥q
```

互換性

標準SQLには`REINDEX`はありません。

RELEASE SAVEPOINT

名前

RELEASE SAVEPOINT -- 設定済みのセーブポイントを破棄する

概要

```
RELEASE [ SAVEPOINT ] savepoint_name
```

説明

RELEASE SAVEPOINT は、現在のトランザクションで事前に設定されていたセーブポイントを破棄します。

セーブポイントを破棄すると、ロールバックするポイントとして使用できなくなります。他にユーザの目に付くような動作はありません。このコマンドは、セーブポイントの設定後に実行されたコマンドの効果を取り消すわけではありません（これを行う方法は[ROLLBACK TO SAVEPOINT](#)を参照してください）。不要になったセーブポイントを破棄することにより、システムがトランザクションの終了前に多少のリソースを回収することができます。

また、RELEASE SAVEPOINT が実行されると、指定したセーブポイントの後に設定されたセーブポイントは全て破棄されます。

パラメータ

savepoint_name

破棄するセーブポイントの名前です。

注釈

設定されていないセーブポイント名を指定するとエラーになります。

トランザクションがアボート状態の時には、セーブポイントを解放することはできません。

同じ名前のセーブポイントが複数存在する場合、最後に設定されたセーブポイントが解放されます。

例

セーブポイントを設定し、その後、破棄します。

```
BEGIN;  
  INSERT INTO table1 VALUES (3);  
  SAVEPOINT my_savepoint;  
  INSERT INTO table1 VALUES (4);  
  RELEASE SAVEPOINT my_savepoint;  
COMMIT;
```

上記のトランザクションでは、3と4の両方が挿入されます。

互換性

このコマンドは標準SQLに準拠しています。SQL:2003標準では、SAVEPOINT は必須であると規定されています。PostgreSQL では SAVEPOINT キーワードを省略することができます。

関連項目

RESET

名前

RESET -- 実行時パラメータの値をデフォルト値に戻す

概要

```
RESET configuration_parameter
RESET ALL
```

説明

RESET は実行時パラメータをデフォルト値に戻します。 RESET は下記のように記述することもできます。

```
SET configuration_parameter TO DEFAULT
```

詳細は[SET](#)を参照してください。

デフォルト値とは、現行セッション内でSET コマンドが発行されなかった場合に変数が保持していた値として定義されます。 デフォルト値は、コンパイル時に指定したデフォルト、設定ファイル、コマンドラインオプション、データベースごと、ユーザごとのデフォルト設定のいずれかを元にしています。 これは、もし設定ファイルを元にした値である場合、現在の設定ファイルによって指定される値に再設定されるため、"セッション起動時にそのパラメータが取る値"という定義と若干異なります。 詳細は[第18章](#)を参照してください。

RESET のトランザクションでの振舞いのSET と同じです。 この効果は、トランザクションのロールバックによって取り消されません。

パラメータ

configuration_parameter

設定可能な実行時パラメータの名前です。 利用できるパラメータについては[第18章](#)で説明します。 また、[SET](#) マニュアルページを参照してください。

ALL

設定可能な全ての実行時パラメータをデフォルト値に戻します。

例

timezone 設定変数をデフォルト値に設定します。

```
RESET timezone;
```

互換性

RESET はPostgreSQLの拡張です。

関連項目

[SET](#), [SHOW](#)

REVOKE

名前

REVOKE -- アクセス権を取り消す

概要

```

REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
ON { [ TABLE ] table_name [, ...]
    | ALL TABLES IN SCHEMA schema_name [, ...] }
FROM { [ GROUP ] role_name | PUBLIC } [, ...]
[ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | REFERENCES } ( column_name [, ...] )
    [, ...] | ALL [ PRIVILEGES ] ( column_name [, ...] ) }
ON [ TABLE ] table_name [, ...]
FROM { [ GROUP ] role_name | PUBLIC } [, ...]
[ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { USAGE | SELECT | UPDATE }
    [, ...] | ALL [ PRIVILEGES ] }
ON { SEQUENCE sequence_name [, ...]
    | ALL SEQUENCES IN SCHEMA schema_name [, ...] }
FROM { [ GROUP ] role_name | PUBLIC } [, ...]
[ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { CREATE | CONNECT | TEMPORARY | TEMP } [, ...] | ALL [ PRIVILEGES ] }
ON DATABASE database_name [, ...]
FROM { [ GROUP ] role_name | PUBLIC } [, ...]
[ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
ON DOMAIN domain_name [, ...]
FROM { [ GROUP ] role_name | PUBLIC } [, ...]
[ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
ON FOREIGN DATA WRAPPER fdw_name [, ...]
FROM { [ GROUP ] role_name | PUBLIC } [, ...]
[ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
ON FOREIGN SERVER server_name [, ...]
FROM { [ GROUP ] role_name | PUBLIC } [, ...]
[ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { EXECUTE | ALL [ PRIVILEGES ] }
ON { FUNCTION function_name ( [ [ argmode ] [ arg_name ] arg_type [, ...] ] ) [, ...]
    | ALL FUNCTIONS IN SCHEMA schema_name [, ...] }
FROM { [ GROUP ] role_name | PUBLIC } [, ...]
[ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
ON LANGUAGE lang_name [, ...]
FROM { [ GROUP ] role_name | PUBLIC } [, ...]
[ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { SELECT | UPDATE } [, ...] | ALL [ PRIVILEGES ] }
ON LARGE OBJECT loid [, ...]
FROM { [ GROUP ] role_name | PUBLIC } [, ...]
[ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { CREATE | USAGE } [, ...] | ALL [ PRIVILEGES ] }
ON SCHEMA schema_name [, ...]

```

```

FROM { [ GROUP ] role_name | PUBLIC } [, ...]
[ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
{ CREATE | ALL [ PRIVILEGES ] }
ON TABLESPACE tablespace_name [, ...]
FROM { [ GROUP ] role_name | PUBLIC } [, ...]
[ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
{ USAGE | ALL [ PRIVILEGES ] }
ON TYPE type_name [, ...]
FROM { [ GROUP ] role_name | PUBLIC } [, ...]
[ CASCADE | RESTRICT ]

REVOKE [ ADMIN OPTION FOR ]
role_name [, ...] FROM role_name [, ...]
[ CASCADE | RESTRICT ]

```

説明

`REVOKE` コマンドは、1つ以上のロールに対して、以前に与えた権限を取り消します。 `PUBLIC` キーワードは暗黙的に定義された全ロールからなるグループです。

権限の種類の意味については[GRANT](#) コマンドの説明を参照してください。

全てのロールは、そのロールに直接許可された権限、現在属しているロールに許可された権限、`PUBLIC` に許可された権限という3つの権限を合わせた権限を持っていることに注意してください。したがって、例えば、`PUBLIC` から `SELECT` 権限を取り消すことは、必ずしも全てのロールがそのオブジェクトに対する `SELECT` 権限を失うことを意味しません。権限が直接許可されているロール、あるいは、別ロール経由で許可されているロールは、`SELECT` 権限を持ち続けます。同様にユーザから `SELECT` を取りあげても、`PUBLIC` またはほかのメンバとして属するロールが `SELECT` 権限を持つ場合、`SELECT` の使用を拒否できません。

`GRANT OPTION FOR` が指定された場合、権限自体ではなく、その権限のグラントオプションのみが取り消されます。指定されていなければ、権限とグラントオプションの両方が取り消されます。

グラントオプション付きの権限を保持しているユーザが、その権限を他ユーザに与えていた場合、与えられたユーザが保持する権限は依存権限と呼ばれます。権限を与えたユーザ自身の権限やグラントオプションが取り消された時、その権限に依存権限が存在する場合、`CASCADE` が指定されていると依存権限も取り消されます。指定されていなければ、権限の取り消しが失敗します。この再帰的な権限の取り消しは、ユーザ権限の連鎖を通じて与えられた権限の中でも、`REVOKE` を実行されたユーザから追跡可能な範囲にのみ影響します。したがって、依存権限を持つユーザが他のユーザからも同じ権限を与えられている場合は、`REVOKE` が実行された後もその権限を保持している可能性があります。

テーブルの権限を取り消す場合、対応する列の権限（もしあれば）も自動的に、そのテーブルの各列から取り消されます。一方、ロールがテーブルに対する権限を持つ場合、個々の列から同じ権限を取り消しても影響ありません。

ロールのメンバ資格を取り消す場合、同様に振舞いますが、`GRANT OPTION` ではなく `ADMIN OPTION` が呼び出されます。また、このコマンド構文では無意味な `GROUP` という単語を受け付けけないことに注意してください。

注釈

既存のテーブルや列に付与された権限を表示するためには [psql](#) の `\dp` コマンドを使用してください。この書式については[GRANT](#) を参照してください。テーブル以外のオブジェクトについて、その権限を表示することができる他の `\d` コマンドが存在します。

取り消すことができるのは、ユーザが直接付与した権限のみです。例えば、もし、ユーザAがグラントオプションを付けてユーザBに権限を与え、その後、ユーザBがユーザCにその権限を与えたとすると、ユーザAはユーザCの権限を直接取り消すことはできません。その代わりに、ユーザAがユーザBのグラントオプションを `CASCADE` オプション付きで取り消すことで、ユーザCに与えられた権限を取り消すことができます。別の状況を考えてみます。AとBの両方が同じ権限をCに与えた場合、AはAの与えた権限を取り消すことはできますが、Bの与えた権限を取り消すことはできません。したがって、Cは実質的にその権限を持ち続けることになります。

オブジェクトの所有者以外がそのオブジェクト上の権限に対して `REVOKE` を実行した場合、ユーザがオブジェクトに対して何の権限も持っていないければ、即座にコマンドが失敗します。何らかの権限があればコマンド処理が続行されますが、取り消すことができるのはそのユーザがグラントオプションを持つ権限のみです。`REVOKE ALL PRIVILEGES` 構文をまったく権限を持たない状態で実行すると、警告が出力されます。他の構文の場合は、そのコマンドで指名した権限に対するグラントオプションを持たない状態で実行すると、警告が出力されます（原理上、上記の説明はオブジェクト所有者にも適用されますが、所有者は常に全てのグラントオプションを保持しているので、こうした問題が発生することはありません）。

スーパーユーザがGRANTやREVOKEコマンドを発行した場合、そのコマンドは、対象のオブジェクトの所有者によって発行されたものであるかのように動作します。 根本的には全ての権限はオブジェクトの所有者から渡されるものなので（ただし、グラントオプションの連鎖により間接的に渡される場合もありますが）、スーパーユーザは、全ての権限を取り消すことができます。 ただし、この場合は上述のCASCADEを使用する必要があります。

REVOKEは、影響するオブジェクトの所有者以外のロールによって実行することもできますが、 オブジェクトを所有するロールのメンバであるか、そのオブジェクトに対しWITH GRANT OPTION権限を持つロールのメンバでなければなりません。 この場合、 そのオブジェクトの実際の所有者ロールまたはWITH GRANT OPTION権限を持つロールによって発行されたかのように、このコマンドは実行されます。 例えば、t1テーブルがg1ロールによって所有され、u1がg1ロールのメンバであるとしします。 この場合、u1はg1で付与されたものと記録されている権限を取り上げることができます。 これには、u1が付与した権限とg1ロールの他のメンバによって付与された権限が含まれます。

REVOKEを実行したロールが、ロールの持つ複数メンバ資格の経路を通して間接的に必要な権限を持つ場合、 どのロールが権限を付与したロールとして記録されるかについては指定されません。 こうした場合、SET ROLEを使用して、REVOKEを行わせたい特定のロールになることを推奨します。 こうしないと、意図しないロールによって権限を取り上げることになったり、取り上げ自体が失敗することになったりします。

例

publicに与えたfilmsテーブルに対する挿入権限を取り消します。

```
REVOKE INSERT ON films FROM PUBLIC;
```

kindsビューから、manuelユーザに与えた全ての権限を取り上げます。

```
REVOKE ALL PRIVILEGES ON kinds FROM manuel;
```

これは"自分が与えた全ての権限を取り消す"ことを意味します。

ユーザjoeからロールadmins内のメンバ資格を取り上げます。

```
REVOKE admins FROM joe;
```

互換性

GRANTコマンドの互換性についての注釈はREVOKEにも当てはまります。 標準では、キーワードRESTRICTかCASCADEのどちらかが必須です。 しかし、PostgreSQLではデフォルトでRESTRICTと仮定されます。

関連項目

[GRANT](#)

前のページ RESET	ホーム 上に戻る	次のページ ROLLBACK

ROLLBACK

名前

ROLLBACK -- 現在のトランザクションをアボートする

概要

ROLLBACK [WORK | TRANSACTION]

説明

ROLLBACKは現在のトランザクションをロールバックし、そのトランザクションで行われた全ての更新を廃棄させます。

パラメータ

WORK
TRANSACTION

省略可能なキーワードです。 効果は何もありません。

注釈

トランザクションを正常に終了させるには[COMMIT](#)を使用してください。

トランザクションの外部でROLLBACKを発行しても問題はありません。 ただし、警告メッセージが表示されます。

例

全ての変更をアボートします。

ROLLBACK;

互換性

標準SQLでは、ROLLBACKとROLLBACK WORKの2構文のみが仕様に記載されています。 この点以外は完全に互換性があります。

関連項目

[BEGIN](#), [COMMIT](#), [ROLLBACK TO SAVEPOINT](#)

[前のページ](#)
REVOKE

[ホーム](#)
[上に戻る](#)

[次のページ](#)
ROLLBACK PREPARED

ROLLBACK PREPARED

名前

ROLLBACK PREPARED -- 事前に二相コミット用に準備されたトランザクションを取り消します。

概要

```
ROLLBACK PREPARED transaction_id
```

説明

ROLLBACK PREPARED は、準備段階のトランザクションをロールバックします。

パラメータ

transaction_id

ロールバックさせるトランザクションのトランザクション識別子です。

注釈

準備されたトランザクションをロールバックするには、トランザクションを元々実行したユーザかスーパーユーザでなければなりません。しかし、トランザクションを実行したセッション内で実行する必要はありません。

このコマンドはトランザクションブロック内では実行できません。準備されたトランザクションは即座にロールバックされます。

現在利用できるすべての準備されたトランザクションは [pg_prepared_xacts](#) システムビュー内に列挙されています。

例

トランザクション識別子 `foobar` で識別されるトランザクションをロールバックします。

```
ROLLBACK PREPARED 'foobar';
```

互換性

ROLLBACK PREPARED は PostgreSQL の拡張です。これは外部のトランザクション管理システムによる利用を意図したものです。ただし外部のトランザクション管理システムの中には標準化されたもの (X/Open XA など) もありますが、こうしたシステムでも SQL 側は標準化されていません。

関連項目

[PREPARE TRANSACTION](#), [COMMIT PREPARED](#)

ROLLBACK TO SAVEPOINT

名前

ROLLBACK TO SAVEPOINT -- セーブポイントまでロールバックする

概要

```
ROLLBACK [ WORK | TRANSACTION ] TO [ SAVEPOINT ] savepoint_name
```

説明

セーブポイントの設定後に実行されたコマンドを全てロールバックします。 セーブポイントは有効なまま残るので、必要に応じて、その後再度ロールバックすることができます。

ROLLBACK TO SAVEPOINTは、指定したセーブポイントより後に設定した全てのセーブポイントを暗黙的に破棄します。

パラメータ

savepoint_name

ロールバック先のセーブポイントです。

注釈

セーブポイントの設定後に実行されたコマンドの結果を維持したままセーブポイントを破棄するには、[RELEASE SAVEPOINT](#)を使用してください。

設定されていないセーブポイントの名前を指定するとエラーになります。

カーソルはセーブポイントという観点から見るとトランザクションの外にあるかのように振舞います。 セーブポイントの内部で開かれたカーソルは全て、そのセーブポイントがロールバックした時に閉ざされます。 セーブポイントの前に開かれたカーソルに対しセーブポイント内でFETCHまたはMOVEコマンドを実行した場合、その後、ロールバックされたとしても、カーソルの位置はFETCHの結果、移動した位置から変わりません（つまりFETCHによる位置の移動はロールバックされません）。 また、カーソルのクローズはロールバックしても取り消すことはできません。 しかしカーソルの問い合わせにより発生するその他の副作用（問い合わせにより呼出される揮発性関数の影響など）は、セーブポイント内で実行され、それがロールバックされた場合に、ロールバックされます。 カーソルの実行によってトランザクションのアボートが引き起こされた場合、そのカーソルは実行不可能状態に遷移します。 この場合、トランザクションはROLLBACK TO SAVEPOINTを使用して戻すことができますが、そのカーソルは使用することができません。

例

*my_savepoint*の設定後に実行されたコマンドの効果を取り消します。

```
ROLLBACK TO SAVEPOINT my_savepoint;
```

セーブポイントへのロールバックは、カーソル位置に影響を与えません。

```
BEGIN;

DECLARE foo CURSOR FOR SELECT 1 UNION SELECT 2;

SAVEPOINT foo;

FETCH 1 FROM foo;
?column?
-----
1
```

```
ROLLBACK TO SAVEPOINT foo;

FETCH 1 FROM foo;
?column?
-----
      2

COMMIT;
```

互換性

標準SQLでは、`SAVEPOINT` キーワードは必須です。しかし、PostgreSQLとOracleでは省略することができます。SQLで使用できるのは、`WORK`のみです。`TRANSACTION`は使用できず、`ROLLBACK`の後のノイズワードとして扱われます。また、SQLでは省略可能な`AND [ NO ] CHAIN`句がありますが、これはPostgreSQLでは現在サポートされていません。その他については、このコマンドは標準SQLと互換性を持ちます。

関連項目

[BEGIN](#), [COMMIT](#), [RELEASE SAVEPOINT](#), [ROLLBACK](#), [SAVEPOINT](#)

[前のページ](#)

ROLLBACK PREPARED

[ホーム](#)
[上に戻る](#)

[次のページ](#)
SAVEPOINT

SAVEPOINT

名前

SAVEPOINT -- 現在のトランザクション内に新規にセーブポイントを定義する

概要

```
SAVEPOINT savepoint_name
```

説明

SAVEPOINT は、現在のトランザクション内に新しいセーブポイントを設定します。

セーブポイントとはトランザクション内に付ける特別な印です。セーブポイントを設定しておく、それ以降に実行されたコマンドを全てロールバックし、トランザクションを設定時の状態に戻すことができます。

パラメータ

savepoint_name

新しいセーブポイントに付与する名前です。

注釈

セーブポイントまでロールバックするには[ROLLBACK TO SAVEPOINT](#)を使用してください。セーブポイント後に行われたコマンドの効果を保持したままセーブポイントを破棄するには、[RELEASE SAVEPOINT](#)を使用してください。

セーブポイントはトランザクションブロックの内側のみに設定することができます。1つのトランザクションの中には、複数のセーブポイントを設定することができます。

例

セーブポイントを設定し、その後に実行した全てのコマンドの効果を取り消します。

```
BEGIN;  
  INSERT INTO table1 VALUES (1);  
  SAVEPOINT my_savepoint;  
  INSERT INTO table1 VALUES (2);  
  ROLLBACK TO SAVEPOINT my_savepoint;  
  INSERT INTO table1 VALUES (3);  
COMMIT;
```

上記のトランザクションでは、1と3は挿入されますが、2は挿入されません。

セーブポイントを設定し、その後に破棄します。

```
BEGIN;  
  INSERT INTO table1 VALUES (3);  
  SAVEPOINT my_savepoint;  
  INSERT INTO table1 VALUES (4);  
  RELEASE SAVEPOINT my_savepoint;  
COMMIT;
```

上記のトランザクションでは、3と4の両方が挿入されます。

互換性

SQLでは、同じ名前のセーブポイントが設定された時は、自動的に古い方のセーブポイントを破棄することになっています。
PostgreSQLでは、古いセーブポイントも保持されますが、 ロールバックや解放時には新しい方のセーブポイントが使用されま
す（`RELEASE SAVEPOINT`を用いて新しいセーブポイントが解放されると、再び`ROLLBACK TO SAVEPOINT`や`RELEASE SAVEPOINT`か
ら古いセーブポイントが使用できるようになります）。この点以外は、`SAVEPOINT`は完全にSQLに従っています。

関連項目

[BEGIN](#), [COMMIT](#), [RELEASE SAVEPOINT](#), [ROLLBACK](#), [ROLLBACK TO SAVEPOINT](#)

前のページ	ホーム	次のページ
ROLLBACK TO SAVEPOINT	上に戻る	SECURITY LABEL

SECURITY LABEL

名前

SECURITY LABEL -- オブジェクトに適用するセキュリティラベルを定義または変更する

概要

```
SECURITY LABEL [ FOR provider ] ON
{
    TABLE object_name |
    COLUMN table_name.column_name |
    AGGREGATE agg_name (agg_type [, ...] ) |
    DATABASE object_name |
    DOMAIN object_name |
    FOREIGN TABLE object_name
    FUNCTION function_name ( [ [ argmode ] [ argname ] argtype [, ...] ] ) |
    LARGE OBJECT large_object_oid |
    [ PROCEDURAL ] LANGUAGE object_name |
    ROLE object_name |
    SCHEMA object_name |
    SEQUENCE object_name |
    TABLESPACE object_name |
    TYPE object_name |
    VIEW object_name
} IS 'label'
```

説明

SECURITY LABELはセキュリティラベルをデータベースオブジェクトに適用します。 ラベルプロバイダごとに1つの、任意の数のセキュリティラベルを指定したデータベースオブジェクトに関連付けることができます。 ラベルプロバイダは、register_label_provider関数を使用して自身を登録する、ロード可能なモジュールです。

注意: register_label_providerはSQL関数ではありません。 バックエンドにロードされたCコードからのみ呼び出すことができます。

ラベルプロバイダは、指定されたラベルが有効かどうか、および指定されたオブジェクトにラベルを割り当てることが許されているかどうかを決定します。 また、ラベルプロバイダは指定されたラベルの意味の決定権を持ちます。 PostgreSQLは、ラベルプロバイダがセキュリティラベルを解釈するかしないか、どのように解釈するかに関して制限を持ちません。 単にこれらを格納するための機構を提供するだけです。 実際には、この機能はSE-Linuxなどのラベルベースの強制アクセス制御（MAC）システムと統合できるようにすることを意図したものです。 こうしたシステムにより、すべてのアクセス制御の決定は、ユーザとグループなどの伝統的な任意アクセス制御（DAC）という考えではなく、オブジェクトラベルに基づいて行われます。

パラメータ

```
object_name
table_name.column_name
agg_name
function_name
```

ラベル付けされるオブジェクトの名前です。 テーブル、集約、ドメイン、外部テーブル、関数、シーケンス、型、ビューの名前はスキーマで修飾可能です。

provider

このラベルに関連するプロバイダの名前です。 指定されたプロバイダはロードされていなければならない、かつ、提供されるラベル付け操作と一致しなければなりません。 プロバイダが1つだけロードされていた場合、プロバイダの名前を省略して簡略化することができます。

arg_type

集約関数が操作する入力データ型です。 引数を持たない集約関数を参照する場合は入力データ型のリストには*と記述してください。

argmode

関数引数のモードです。 IN、OUT、INOUT、VARIADICのいずれかです。 省略された場合のデフォルトはINです。 入力引数のみが関数の識別を決定するために必要ですので、実際にはSECURITY LABEL ON FUNCTIONはOUTをまったく考慮しないことに注意してください。 このためIN、INOUT、VARIADICのリストで十分です。

argname

関数引数の名前です。 引数のデータ型のみが関数の識別を決定するために必要ですので、実際にはSECURITY LABEL ON FUNCTIONは引数名をまったく考慮しないことに注意してください。

argtype

関数引数があればそのデータ型です（スキーマ修飾可）。

large_object_oid

ラージオブジェクトのOIDです。

PROCEDURAL

これは意味がない単語です。

label

文字列リテラルで記述された新しいセキュリティラベルです。 セキュリティラベルを削除するためにはNULLと記述します。

例

以下の例はテーブルのセキュリティラベルを変更する方法を示します。

```
SECURITY LABEL FOR selinux ON TABLE mytable IS 'system_u:object_r:sepgsql_table_t:s0';
```

互換性

標準SQLにはSECURITY LABEL コマンドはありません。

関連項目

[sepgsql](#), [dummy_seclabel](#)

[前のページ](#)
SAVEPOINT

[ホーム](#)
[上に戻る](#)

[次のページ](#)
SELECT

SELECT

名前

SELECT, TABLE, WITH -- テーブルもしくはビューから行を検索する

概要

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
      * | expression [ [ AS ] output_name ] [, ...]
      [ FROM from_item [, ...] ]
      [ WHERE condition ]
      [ GROUP BY expression [, ...] ]
      [ HAVING condition [, ...] ]
      [ WINDOW window_name AS ( window_definition ) [, ...] ]
      [ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] select ]
      [ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ] [, ...] ]
      [ LIMIT { count | ALL } ]
      [ OFFSET start [ ROW | ROWS ] ]
      [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ]
      [ FOR { UPDATE | SHARE } [ OF table_name [, ...] ] [ NOWAIT ] [...] ]
```

ここで`from_item`は以下のいずれかです。

```
[ ONLY ] table_name [ * ] [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
( select ) [ AS ] alias [ ( column_alias [, ...] ) ]
with_query_name [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
function_name ( [ argument [, ...] ] ) [ AS ] alias [ ( column_alias [, ...] | column_definition [, ...] ) ]
      function_name ( [ argument [, ...] ] ) AS ( column_definition [, ...] )
      from_item [ NATURAL ] join_type from_item [ ON join_condition | USING ( join_column [, ...] ) ]
```

また`with_query`は以下の通りです。

```
with_query_name [ ( column_name [, ...] ) ] AS ( select | values | insert | update | delete )

TABLE [ ONLY ] table_name [ * ]
```

説明

SELECTは0個以上のテーブルから行を返します。SELECTの一般的な処理は以下の通りです。

- 1 11 WITHリスト内のすべての問い合わせが計算されます。これらは実質、FROMリスト内から参照可能な一時テーブルとして提供されます。FROM内で2回以上参照されるWITH問い合わせは一度のみ計算されます。（後述の[WITH句](#)を参照してください。）
- 1 11 FROMリストにある全要素が計算されます（FROMリストの要素は実テーブルか仮想テーブルのいずれかです）。FROMリストに複数の要素が指定された場合、それらはクロス結合されます（後述の[FROM句](#)を参照してください）。
- 1 11 WHERE句が指定された場合、条件を満たさない行は全て出力から取り除かれます（後述の[WHERE句](#)を参照してください）。
- 1 11 GROUP BY句が指定された場合、1つまたは複数の値が条件に合う行ごとにグループに組み合わせて出力されます。HAVING句が指定された場合、指定した条件を満たさないグループは取り除かれます（後述の[GROUP BY句](#)と[HAVING句](#)を参照してください）。
- 1 11 実際には、選択された各行または行グループに対して、SELECT出力式を使用して計算した結果の行が出力されます（後述の[SELECTリスト](#)を参照してください）。
- 1 11 DISTINCTは結果から重複行を取り除きます。DISTINCT ONは指定した全ての式に一致する行を取り除きます。ALLでは、重複行も含め、全ての候補行を返します（これがデフォルトです。詳しくは、後述の[DISTINCT句](#)を参照してください）。
- 1 11 UNION、INTERSECT、EXCEPT演算子を使用すると、複数のSELECT文の出力を1つの結果集合にまとめることができます。UNION演算子は、両方の結果集合に存在する行と、片方の結果集合に存在する行を全て返します。INTERSECT演算子は、両方の結果集合に存在する行を返します。EXCEPT演算子は、最初の結果集合にあり、2番目の結果集合にない行を返しま

す。ALLが指定されない限り、いずれの場合も、重複する行は取り除かれます。無意味なDISTINCTという単語を付けて、明示的に重複行を除去することを指定することができます。SELECT自体はALLがデフォルトですが、この場合はDISTINCTがデフォルトの動作であることに注意してください。（後述の[UNION句](#)、[INTERSECT句](#)、[EXCEPT句](#)を参照してください。）

- 1 11 ORDER BY句が指定された場合、返される行は指定した順番でソートされます。ORDER BYが指定されない場合は、システムが計算過程で見つけた順番で行が返されます（後述の[ORDER BY句](#)を参照してください）。
- 1 11 LIMIT（またはFETCH FIRST）あるいはOFFSET句が指定された場合、SELECT文は結果行の一部分のみを返します（詳しくは、後述の[LIMIT句](#)を参照してください）。
- 1 11 FOR UPDATEまたはFOR SHARE句を指定すると、SELECT文は引き続き行われる更新に備えて選択行をロックします（詳しくは、後述の[FOR UPDATE/FOR SHARE句](#)を参照してください）。

SELECTコマンド内で使われる列それぞれに対するSELECT権限が必要です。FOR UPDATEまたはFOR SHAREを使用するには、さらに、（選択できるように各テーブルで少なくとも1列に対する）UPDATE権限が必要です。

パラメータ

WITH句

WITH句により主問い合わせ内で名前により参照可能な、1つ以上の副問い合わせを指定することができます。副問い合わせは実質的に主問い合わせの間の一時的なテーブルかビューのように動作します。各副問い合わせはSELECT、VALUES、INSERT、UPDATE、DELETEにすることができます。WITH内でデータ変更文（INSERT、UPDATE、DELETE）を記述する場合は、RETURNING句を含めることが有用です。主問い合わせで読み取られる一時テーブルを形成するのは、RETURNINGの出力であり、文が変更する背後のテーブルではありません。RETURNINGを省くと、文は同様に実行されますが、出力を生成しませんので、主問い合わせでテーブルとして参照することができません。

（スキーマ修飾がない）名前を各WITH問い合わせで指定しなければなりません。省略可能ですが、列名のリストを指定することもできます。これを省略すると、列名は副問い合わせから推定されます。

RECURSIVEが指定されると、SELECT副問い合わせは自身で名前により参照することができます。こうした副問い合わせは以下のような形式でなければなりません。

```
non_recursive_term UNION [ ALL | DISTINCT ] recursive_term
```

ここで再帰的な自己参照はUNIONの右辺に現れなければなりません。問い合わせ当たり1つの再帰的な自己参照のみが許されます。再帰的なデータ変更文はサポートされていませんが、データ変更文で再帰的なSELECTの結果を使用することができます。例は[項7.8](#)を参照してください。

RECURSIVEには他にも、WITH問い合わせが順序通りでなくても構わないという効果があります。問い合わせはリストの後にある別のものを参照することができます。（しかし巡回する参照や相互的な参照は実装されていません。）RECURSIVEがないと、WITH問い合わせは主問い合わせが共通するWITH問い合わせのうち、WITHリストの前方にあるもののみを参照することができます。

WITH問い合わせの鍵となる特性は、これらを主問い合わせが複数回参照していたとしても、主問い合わせの実行当たり一度のみ評価される点です。特にデータ変更文は、主問い合わせがその出力のすべてまたは一部を読み取るかに関係なく、本当に一度のみ実行されることが保証されています。

主問い合わせとWITH問い合わせは（理論上）同時にすべて実行されます。WITH内のデータ変更文によりなされた影響は、RETURNING出力を読み取る以外、問い合わせの他の部分では参照できないことを意味します。こうしたデータ変更文が2つあり、同じ行を変更しようとした場合、その結果は明確ではありません。

追加情報については[項7.8](#)を参照してください。

FROM句

FROM句にはSELECTの対象となるソーステーブルを1つ以上指定します。複数のソースが指定された場合、結果は全てのソースの直積（クロス結合）となります。しかし、通常は制約条件を付けて、直積のごく一部を返すように結果行を限定します。

FROM句には以下の要素を指定できます。

table_name

既存のテーブルもしくはビューの名前です（スキーマ修飾名も可）。テーブル名の前にONLYが指定された場合、そのテーブルのみがスキャンされます。ONLYが指定されない場合、テーブルと（もしあれば）それを継承する全てのテーブルがスキャンされます。省略することもできますが、テーブル名の後に*を指定することで、明示的に継承するテーブルも含まれることを示すことができます。

alias

別名を含むFROMアイテムの代替名です。別名は、指定を簡潔にするため、もしくは、自己結合（同じテーブルを複数回スキャンする結合）の曖昧さをなくすために使われます。別名が指定されている場合は、その別名によって実際のテーブル名または関数名が完全に隠されます。例えば、FROM foo AS fと指定されている場合、以降のSELECT文ではこのFROMアイテムをfooではなくfとして参照する必要があります。テーブルの別名があれば、そのテーブルの複数の列の名前を置き換える列の別名リストを記述することができます。

select

FROM句では、副SELECTを使うことができます。SELECTコマンドの実行中、副SELECTの出力は一時テーブルであるかのように動作します。副SELECTは括弧で囲まれなければなりません。また、必ず別名を与えておかなければなりません。VALUESコマンドをここで使用することもできます。

with_query_name

WITH問い合わせは、問い合わせの名前があたかもテーブル名であるかのように、名前を記述することで参照されます。（実際にはWITH問い合わせは主問い合わせの対象とするテーブルと同じ名前の実テーブルを隠蔽します。必要ならばテーブル名をスキーマ修飾することで同じ名前の実テーブルを参照することができます。）テーブルと同様の方法で別名を提供することができます。

function_name

FROM句では、関数呼び出しを使用することができます（これは特に関数が結果セットを返す場合に有用ですが、任意の関数を使用することもできます）。SELECTコマンドの実行中は、この関数の結果は一時テーブルであるかのように動作します。また、別名を使用することもできます。別名が指定されていれば、さらに列の別名リストを指定して、関数の複合型の戻り値の属性に対する代替名を提供することもできます。関数がrecordデータ型を返すと定義されている場合は、別名すなわちASキーワードと、それに続く(column_name data_type [, ...])という形式の列定義リストが必要です。列定義リストは、関数によって返される実際の列の数およびデータ型に一致していなければなりません。

join_type

以下のいずれかです。

- [INNER] JOIN
- LEFT [OUTER] JOIN
- RIGHT [OUTER] JOIN
- FULL [OUTER] JOIN
- CROSS JOIN

INNERおよびOUTER結合型では、結合条件、すなわち、NATURAL, ON join_condition、USING (join_column [, ...])のいずれか1つのみを指定する必要があります。それぞれの意味は後述します。CROSS JOINでは、これらの句を記述しなくても構いません。

JOIN句は、2つのFROMアイテムを結び付けます。入れ子の順番を決めるために、必要ならば括弧を使用してください。括弧がないと、JOINは左から右へ入れ子にします。どのような場合でもJOINは、カンマで分けられたFROM項目よりも強い結び付きを持ちます。

CROSS JOINとINNER JOINは直積を1つ生成します。これは、FROMの最上位で2つの項目を結合した結果と同一です。しかし、（指定すれば）結合条件によって制限をかけることができます。CROSS JOINはINNER JOIN ON (true)と等価であり、条件によって削除される行はありません。これらの結合型は記述上の便宜のためだけに用意されています。したがって、通常のFROMとWHEREを実行しなければ何も行いません。

LEFT OUTER JOINは、条件に合う直積の全ての行（つまり、その結合条件を満たす全ての組み合わせ）に加え、左側テーブルの中で、右側テーブルには結合条件を満たす行が存在しなかった行のコピーも返します。この左側テーブルの行を結合結果のテーブルの幅に拡張するために、右側テーブルが入る列にはNULL値が挿入されます。マッチする行を決める時は、JOIN句自身の条件のみが考慮されることに注意してください。他の外部結合条件は後で適用されます。

逆に、RIGHT OUTER JOINは、全ての結合行と、左側テーブルに当てはまるものがなかった右側の行（左側はNULLで拡張されています）の1行ずつを返します。左右のテーブルを入れ替えればLEFT OUTER JOINに変換できるので、RIGHT OUTER JOINは記述上の便宜を図るため用意されているに過ぎません。

`FULL OUTER JOIN`は、全ての結合行に加え、一致しなかった左側の行（右側はNULLで拡張）、一致しなかった右側の行（左側はNULLで拡張）を全て返します。

`ON join_condition`

`join_condition`は、結合においてどの行が一致するかを指定する、`boolean`型の値を返す式です（`WHERE`句に類似しています）。

`USING ( join_column [, ...] )`

`USING ( a, b, ... )`句は`ON left_table.a = right_table.a AND left_table.b = right_table.b ...`の省略形です。`USING`は等価な列の両方ではなく片方のみが結合の出力に含まれることを意味します。

`NATURAL`

`NATURAL`は、2つのテーブル内の同じ名前を持つ行を全て指定した`USING`リストの省略形です。

WHERE 句

`WHERE`句は通常以下の形式となります（この句は省略可能です）。

`WHERE condition`

`condition`は、評価の結果として`boolean`型を返す任意の式です。この条件を満たさない行は全て出力から取り除かれます。全ての変数に実際の行の値を代入して、式が真を返す場合、その行は条件を満たすとみなされます。

GROUP BY 句

`GROUP BY`句は通常以下の形式となります（この句は省略可能です）。

`GROUP BY expression [, ...]`

`GROUP BY`は、グループ化のために与えられた式を評価し、結果が同じ値になった行を1つの行にまとめる機能を持ちます。`expression`には、入力列の名前、出力列（`SELECT`リスト項目）の名前/序数、あるいは入力列の値を計算する任意の式を取ることができます。判断がつかない時は、`GROUP BY`の名前は出力列名ではなく入力列名として解釈されます。

集約関数を使用した場合、各グループ内の全ての行を対象に計算が行われ、結果としてグループごとの値が生成されます（一方`GROUP BY`がなければ、集約関数は選択された全ての行を対象に計算を行い、1つの値を生成します）。`GROUP BY`が存在する場合、集約関数内部以外で、グループ化されていない列を参照する場合やグループ化されていない列がグループ化された列に関数依存する場合、`SELECT`リストは無効になります。こうしないとグループ化されていない列について返される値は複数の値になってしまう可能性があるからです。グループ化された列(またはその部分集合)がグループ化されていない列を含むテーブルのプライマリキーである場合、関数依存が存在します。

HAVING 句

`HAVING`は通常以下の形になります（この句は省略可能です）。

`HAVING condition`

`condition`は`WHERE`句で指定するものと同じです。

`HAVING`は、グループ化された行の中で、条件を満たさない行を取り除く機能を持ちます。`HAVING`と`WHERE`は次の点が異なります。`WHERE`が、`GROUP BY`の適用前に個々の行に対してフィルタを掛けるのに対し、`HAVING`は、`GROUP BY`の適用後に生成されたグループ化された行に対してフィルタをかけます。`condition`内で使用する列は、集約関数内で使用されたものを除き、グループ化された列を一意に参照するものでなければなりません。

`HAVING`句があると、`GROUP BY`句がなかったとしても問い合わせはグループ化された問い合わせになります。`GROUP BY`句を持たない問い合わせが集約関数を含む場合も同様です。選択された行はすべて、1つのグループを形成するものとみなされます。また、`SELECT`リストと`HAVING`句では、集約関数が出力するテーブル列しか参照することができません。こうした問い合わせでは、`HAVING`が真の場合には単一の行を、真以外の場合は0行を出力します。

WINDOW 句

省略可能な`WINDOW`句の一般的な構文は以下の通りです。

`WINDOW window_name AS ( window_definition ) [, ...]`

ここで`window_name`は、こののちのウィンドウ定義や`OVER`句で参照することができる名前です。 また、`window_definition`は以下の通りです。

```
[ existing_window_name ]
[ PARTITION BY expression [, ...] ]
[ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ] [, ...] ]
[ frame_clause ]
```

`existing_window_name`が指定された場合、それは`WINDOW`リストの前にある項目を参照しなければなりません。 新しいウィンドウはその範囲指定句をその項目からコピーします。 順序指定句があった場合も同様です。 この場合、新しいウィンドウでは独自の`PARTITION BY`句を指定することはできません。 また、コピーされたウィンドウが`ORDER BY`を持たない場合のみ`ORDER BY`を指定することができます。 新しいウィンドウは常に独自のフレーム句を使用します。 コピーされたウィンドウはフレーム句を指定してはなりません。

`PARTITION BY`リストの要素は[GROUP BY句](#)の要素とほとんど同じように解釈されます。 ただし、こちらは常に単純な式であり、出力列の名前や番号ではないことが異なります。 他にも違いがあり、これらの式は、通常の`GROUP BY`句では許されない、集約関数を含めることができるという点です。 グループ化および集約処理の後にウィンドウ処理が動作するため、これらでは許されています。

同様に、`ORDER BY`リストの要素は[ORDER BY句](#)の要素とほとんど同じように解釈されます。 ただし、この式は常に単純な式であり、出力列の名前や番号ではないことが異なります。

省略可能な`frame_clause`は、（すべてではありませんが）フレームに依存するウィンドウ関数用のウィンドウフレームを定義します。 ウィンドウフレームは、問い合わせの各行(現在の行と呼ばれます)に関連する行の集合です。 `frame_clause`は以下のいずれかを取ることができます。

```
[ RANGE | ROWS ] frame_start
[ RANGE | ROWS ] BETWEEN frame_start AND frame_end
```

ここで`frame_start`と`frame_end`は以下のいずれかを取ることができます。

```
UNBOUNDED PRECEDING
value PRECEDING
CURRENT ROW
value FOLLOWING
UNBOUNDED FOLLOWING
```

`frame_end`が省略された場合デフォルトで`CURRENT ROW`となります。 `frame_start`は`UNBOUNDED FOLLOWING`とすることができない、`frame_end`は`UNBOUNDED PRECEDING`とすることができない、および、上のリストで`frame_end`の選択を`frame_start`の選択より先に行うことができないという制限があります。 例えば`RANGE BETWEEN CURRENT ROW AND value PRECEDING`は許されません。

デフォルトのフレーム化オプションは`RANGE UNBOUNDED PRECEDING`です。 これは`RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`と同じで、パーティションの先頭から`ORDER BY`順序における現在の行の最後のピアまでのすべての行をフレームとします(`ORDER BY`がなければすべての行を意味します)。 一般的に`UNBOUNDED PRECEDING`はフレームがパーティションの先頭から始まることを意味し、同様に`UNBOUNDED FOLLOWING`はフレームがパーティションの最終行で終わることを意味します(`RANGE`モードか`ROWS`かは関係ありません)。 `ROWS`モードでは、`CURRENT ROW`はフレームが現在の行で始まる、または終わることを意味しますが、`RANGE`モードでは、フレームが現在の行の`ORDER BY`順序における最初のピアまたは最後のピアで始まる、または終わることを意味します。 現時点では`value PRECEDING`および`value FOLLOWING`という場合だけは`ROWS`モードだけで許されます。 これらは、現在の行の何行前または何行後にフレームが始まるまたは終わることを示します。 `value`は整数式でなければならない、変数、集約関数、ウィンドウ関数を含めることはできません。 この値は`NULL`または負を取ることはできません。 しかし、現在の行自身を選択するゼロを取ることができます。

`ORDER BY`順序によりその行を一意に順序付けできない場合、`ROWS`が予期できない結果をもたらす可能性があることに注意して下さい。 `RANGE`は、`ORDER BY`順序におけるピアとなる行が同等に扱われる、つまり任意の2つのピアはフレーム内に両方とも存在するか、存在しないかのいずれかとなることが確実になるように設計されています。

`WINDOW`句の目的は、問い合わせの[SELECTリスト](#)または[ORDER BY句](#)に記載されるウィンドウ関数の動作を規定することです。 これらの関数はその`OVER`句において名前で`WINDOW`句の項目を参照することができます。 しかし`WINDOW`句の項目は他で参照されてはなりません。 問い合わせ内で使用されなかったものは、単に無視されます。 ウィンドウ関数呼び出しは`OVER`句でウィンドウ定義を直接規定することができますので、`WINDOW`句を全く使わずにウィンドウ関数を使用することができます。 しかし`WINDOW`句は、同じウィンドウ定義が複数のウィンドウ関数で必要とされる場合に入力量を省くことができます。

ウィンドウ関数に関する詳細については[項3.5](#)、[項4.2.8](#)、[項7.2.4](#)を参照してください。

SELECT リスト

`SELECT` リスト (`SELECT` と `FROM` の間にあるキーワード) は、`SELECT` 文の出力行を形成する式を指定するものです。 この式では、`FROM` 句で処理後の列を参照することができます (通常は実際に参照します)。 `AS output_name` を使用すると、出力列に元の

名前とは別の名前を付けることができます。

テーブルの場合と同様に、SELECTの出力列はすべて名前を持ちます。 簡単なSELECTでは、この名前は列に表示用のラベルを付けるために使用されるだけです。 しかしSELECTが大規模な問い合わせの副問い合わせである場合、大規模な問い合わせ側で副問い合わせで生成された仮想のテーブルの列名としてこの名前が参照されます。 出力列として使用するための名前を指定するためには、列式の後にAS output_nameと記述してください。 （希望する列名がPostgreSQLのキーワード（[付録C](#)を参照）に一致しない場合にのみASを省略することができます。 将来あり得るキーワードの追加に備えるために、常にASを記述する、あるいは、出力名を二重引用符で括弧することを推奨します。） 列名を指定しない場合、名前はPostgreSQLにより自動的に付けられます。 列式が単純な列参照であれば、つけられる名前はその列の名前と同じものです。 より複雑な場合では、関数名または型名が使用されるかもしれません。 さもなければ?columnN?のように生成される名前になるかもしれません。

ORDER BY句とGROUP BY句内で列の値を参照する時も、出力列名を使用できます。 しかし、WHEREやHAVING句では使用できません。 これらでは式を書かなければなりません。

リストには、選択された行の全ての列を表す省略形として、式ではなく*と書くことができます。 また、そのテーブルに由来する列のみを表す省略形として、table_name.*と書くこともできます。 このような場合、ASにより新しい名前を指定することはできません。 出力列名はテーブルの列名と同一になります。

DISTINCT句

SELECT DISTINCTが指定されると、重複する行は全て結果セットから削除されます（重複するグループの中で1行が保持されます）。 SELECT ALLはこの反対で、全ての行が保持されます。 デフォルトはこちらです。

SELECT DISTINCT ON (expression [, ...])は各行集合の中で、指定した式が等しいと評価した最初の行のみを保持します。 DISTINCT ON式は、ORDER BY（上述）と同じ規則で扱われます。 各集合の"最初の行"は、ORDER BYを使用して目的の行が確実に最初に現れるようにしない限り予測することはできないことに注意してください。 例えば、次の例は各地点の最新の気象情報を取り出します。

```
SELECT DISTINCT ON (location) location, time, report
FROM weather_reports
ORDER BY location, time DESC;
```

しかしORDER BYを使用して各地点を時間によって降順にソートしなければ、各地点について得られる情報がいつのものかはわかりません。

DISTINCT ONに指定する式はORDER BYの最も左側の式と一致しなければなりません。 ORDER BY句は、通常、各DISTINCT ONグループの中での行の優先順位を決定する追加的な式を含みます。

UNION句

UNIONは通常以下の形式となります。

```
select_statement UNION [ ALL | DISTINCT ] select_statement
```

select_statementには、ORDER BY、LIMIT、FOR UPDATE、FOR SHARE句を持たない任意のSELECT文が入ります（ORDER BYとLIMITは、括弧で囲めば複式として付与することができます。 括弧がない場合、これらの句は右側に置かれた入力式ではなく、UNIONの結果に対して適用されてしまいます）。

UNION演算子は、2つのSELECT文が返す行の和集合を作成します。 この和集合には、2つのSELECT文の結果集合のいずれか（または両方）に存在する行が全て含まれています。 UNIONの直接のオペランドとなるSELECT文同士が返す列数は、同じでなければなりません。 また、対応する列のデータ型には互換性が存在する必要があります。

ALLオプションが指定されていない限り、UNIONの結果には重複行は含まれません。 ALLを指定するとこのような重複除去が行われません（したがって、通常UNION ALLはUNIONよりかなり高速です。 できるだけALLを使用してください）。 重複行を除去するデフォルトの動作を明示的に指定するためにDISTINCTを記述することができます。

1つのSELECT文に複数のUNION演算子がある場合、括弧がない限り、それらは左から右に評価されます。

現時点では、UNIONの結果やUNIONに対する入力に、FOR UPDATEまたはFOR SHAREを指定することはできません。

INTERSECT句

INTERSECTは通常以下の形式となります。

```
select_statement INTERSECT [ ALL | DISTINCT ] select_statement
```


`select_statement`には、ORDER BY、LIMIT、FOR UPDATE、FOR SHARE句を持たない、任意のSELECT文が入ります。

INTERSECTは、2つのSELECT文が返す行の積集合を計算します。この積集合に含まれるのは、2つのSELECT文の結果集合の両方に存在する行です。

ALLオプションを指定しない限り、INTERSECTの結果に重複行は含まれません。ALLが指定された場合、左側テーブルに m 個、右側テーブルに n 個の重複がある行は、結果集合では $\min(m,n)$ 個出現します。重複行を除去するデフォルトの動作を明示的に指定するためにDISTINCTを記述することができます。

1つのSELECT文に複数のINTERSECT演算子がある場合、括弧がない限り、それらは左から右に評価されます。INTERSECTはUNIONよりも強い結び付きを持ちます。つまり、`A UNION B INTERSECT C`は`A UNION (B INTERSECT C)`と解釈されます。

現時点では、INTERSECTの結果やINTERSECTに対する入力に、FOR UPDATEまたはFOR SHAREを指定することはできません。

EXCEPT句

EXCEPTは通常以下の形式となります。

```
select_statement EXCEPT [ ALL | DISTINCT ] select_statement
```

`select_statement`には、ORDER BY、LIMIT、FOR UPDATE、FOR SHARE句を持たない、任意のSELECT文が入ります。

EXCEPTは、左側のSELECT文の結果には存在し、右側のSELECT文の結果には存在しない行の集合を生成します。

ALLオプションが指定されていない限り、EXCEPTの結果には重複行は含まれません。ALLがある場合、左側テーブルに m 個、右側テーブルに n 個の重複がある行は、結果集合では $\max(m-n,0)$ 個出現します。重複行を除去するデフォルトの動作を明示的に指定するためにDISTINCTを記述することができます。

1つのSELECT文に複数のEXCEPT演算子がある場合、括弧がない限り、それらは左から右に評価されます。EXCEPTの結び付きの強さはUNIONと同じです。

現時点では、EXCEPTの結果やEXCEPTに対する入力に、FOR UPDATEまたはFOR SHAREを指定することはできません。

ORDER BY句

ORDER BY句は通常以下の形式となります（この句は省略可能です）。

```
ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ] [, ...]
```

ORDER BY句を使うと、結果行を指定した式（複数可）に従ってソートすることができます。最も左側の式を使って比較した結果、2つの行が等しいと判断された場合は、1つ右側の式を使って比較します。その結果も等しければ、さらに次の式に進みます。指定した全ての式で等しいと判断された場合は、実装に依存した順番で返されます。

expressionには、出力列（SELECTリスト項目）の名前または序数、あるいは入力列値から形成される任意の式を取ることができます。

序数は、出力列の位置（左から右に割り当てられます）を示します。これを使うと、一意な名前を持たない列の順序を定義することができます。AS句を使用すれば出力列に名前を割り当てることができるので、これはどうしても必要な機能というわけではありません。

また、ORDER BY句には、SELECT出力リストに出現しない列を含む、任意の式を使用できます。したがって、以下の文は有効です。

```
SELECT name FROM distributors ORDER BY code;
```

ただし、UNION、INTERSECT、EXCEPTの結果にORDER BYを適用する場合は、式は使用できず、出力列の名前か序数のみを指定できるという制限があります。

ORDER BYの式として出力列名と入力列名の両方に一致する単なる名前が与えられた場合、ORDER BYはそれを出力列名として扱います。これは、同じ状況におけるGROUP BYの選択とは反対です。この不整合は、標準SQLとの互換性を保持するために発生しています。

ORDER BY中の任意の式の後に、省略可能なキーワードASC（昇順）、DESC（降順）を付加することができます。指定がなければ、デフォルトでASCがあるものとして扱われます。その他、順序を指定する演算子名をUSING句に指定する方法もあります。順序指定演算子は何らかのB-Tree演算子族の小なりまたは大なり演算子でなければなりません。通常、ASCはUSING<と、DESCはUSING>と同じです（ただし、ユーザ定義データ型の作成時には、デフォルトのソート順を定義することができます

す。また、異なる名前の演算子と対応付けすることもできます）。

`NULLS LAST`が指定されると、NULL値はすべての非NULL値の後にソートされます。`NULLS FIRST`が指定されると、NULL値はすべての非NULL値の前にソートされます。どちらも指定されない場合のデフォルト動作は、明示的あるいは暗黙的な`ASC`の場合には`NULLS LAST`、`DESC`がの場合は`NULLS FIRST`です。（したがって、デフォルトでは、NULLが非NULLよりも大きい値であるかのように動作します。）`USING`が指定されると、デフォルトのNULLの順序は、演算子が小なり演算子か大なり演算子によって変わります。

順序付けオプションは直前の演算子にのみ適用されます。たとえば、`ORDER BY x, y DESC`は`ORDER BY x DESC, y DESC`と同一の意味ではありません。

文字型データでは、格納する列に適用された照合順序に従ってソートされます。これは必要に応じて`expression`内に`COLLATE`句を含めることで上書きできます。例えば`ORDER BY mycolumn COLLATE "en_US"`です。より詳細については[項4.2.10](#)および[項22.2](#)を参照してください。

LIMIT 句

LIMIT 句は2つの独立した副句から構成されます。

```
LIMIT { count | ALL }  
OFFSET start
```

`count`には返される行の最大数を、一方、`start`には行を返し始める前に飛ばす行数を指定します。両方とも指定された場合、`start`行分が飛ばされ、そこから数えて`count`行が返されます。

`count`式がNULLと評価された場合、LIMIT ALLとして、つまり制限無しとして扱われます。`start`がNULLと評価された場合、OFFSET 0と同様に扱われます。

SQL:2008では同じ結果を実現する異なる構文が導入されました。PostgreSQLでもサポートしています。以下の構文です。

```
OFFSET start { ROW | ROWS }  
FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY
```

この構文において、`start`または`count`に単一整数定数以外を記述するためには、括弧でくくって記述しなければなりません。`count`をFETCH句で省略した場合、そのデフォルトは1です。ROWおよびROWS、そしてFIRSTおよびNEXTは意味がない単語で、この句に影響を与えることはありません。標準に従うとOFFSET句は、FETCH句と同時に使用する場合、これより前に存在しなければなりません。しかしPostgreSQLは厳密ではなく、どちらが先でも許されます。

LIMITを使う時は、結果行を一意的な順番に強制するORDER BY句を使うとよいでしょう。そうしないと、問い合わせ結果のどの部分が返されるのかがわかりません。10～20行目までを出力するとしても、どの順番で並べた時の10～20行目なのでしょう。ORDER BYを指定しない限り、行が返される順番は不明です。

問い合わせプランナは問い合わせ計画を作成する時にLIMITを考慮するので、LIMITとOFFSETの指定によって異なった計画を得ることになるでしょう。計画が異なれば、異なる順番で行が返ります。したがって、LIMIT/OFFSET値の変更によって異なる結果行を選択しようとする、ORDER BYで順序を並び替えない限り、矛盾した結果を返すことになります。これはバグではありません。「SQLは、ORDER BYで順序を制御されない限り、問い合わせ結果が返す順序を約束しない」という事実の当然の帰結なのです。

厳密的に部分集合の選択を強制するORDER BYがなければ、同じLIMIT問い合わせを繰り返し実行してもテーブル行から異なる部分集合が取り出される可能性すらあります。繰り返しますが、これは不具合ではありません。こうした場合に確定した結果は単に保証されていないのです。

FOR UPDATE / FOR SHARE 句

FOR UPDATEは以下の形式となります。

```
FOR UPDATE [ OF table_name [, ...] ] [ NOWAIT ]
```

深く関連するFOR SHARE句は以下の形式となります。

```
FOR SHARE [ OF table_name [, ...] ] [ NOWAIT ]
```

FOR UPDATEを使用すると、問い合わせによって検索された行が更新用にロックされます。これにより、現行のトランザクションが終了するまでは、これらの行が他のトランザクションによって変更されたり削除されたりすることがなくなります。つまり、現行のトランザクションが終了するまでは、他のトランザクションがこれらの行に対してUPDATE、DELETE、SELECT FOR UPDATEを試行しても拒否されます。また、他のトランザクションからのUPDATE、DELETE、SELECT FOR UPDATEによって選択した行がロックされている場合、SELECT FOR UPDATEを実行しようすると、SELECT FOR UPDATEはそのトランザクションが終了する

のを待ってから、その後行をロックして更新された行を返します（行が削除された場合は返しません）。しかしREPEATABLE READまたはSERIALIZABLEトランザクションの内部では、ロック対象の行がトランザクション開始時から変更されていた場合、エラーが発生します。第13章を参照してください。

FOR SHAREも同様に振舞いますが、入手する行に対し排他的ロックを獲得するのではなく共有ロックを獲得する点異なります。共有ロックにより、他トランザクションによるその行に対するUPDATE、DELETE、SELECT FOR UPDATE操作はブロックされます。しかし、他トランザクションによるSELECT FOR SHARE操作を防ぎません。

他のトランザクションのコミットを待機することなく操作を進めるには、NOWAITオプションを使用してください。NOWAITでは、選択行のロックを即座に獲得できない時、文は待機せずに、エラーを報告します。NOWAITは行レベルロックにのみに適用される点に注意してください。つまり、必要なROW SHAREテーブルレベルロックは通常通りの方法（第13章を参照）で獲得されます。もし、テーブルレベルのロックを待機せずに獲得しなければならないのであれば、最初にLOCKのNOWAITオプションを使用してください。

FOR UPDATEまたはFOR SHARE内に特定のテーブルが指定されている場合は、そのテーブルの行のみがロックされます。SELECT内の他のテーブルは通常通りに読み込まれます。テーブルリストを持たないFOR UPDATEもしくはFOR SHARE句は、その文で使用されるすべてのテーブルに影響を与えます。FOR UPDATEもしくはFOR SHAREがビューまたは副問い合わせで使用された場合、そのビューや副問い合わせで使用されるすべてのテーブルに影響を与えます。しかしFOR UPDATE/FOR SHAREは主問い合わせで参照されるWITH問い合わせには適用されません。WITH問い合わせ内での行ロックを行いたい場合は、WITH問い合わせ内でFOR UPDATEまたはFOR SHAREを指定してください。

異なるロック方式を異なるテーブルに指定する必要があるれば、複数のFOR UPDATEとFOR SHARE句を記述することができます。FOR UPDATE句とFOR SHARE句の両方で、同一のテーブルを記述した（または暗黙的に影響を与えられた）場合、FOR UPDATEとして処理されます。同様に、あるテーブルに影響を与える句のいずれかでNOWAITが指定された場合、そのテーブルはNOWAITとして処理されます。

FOR UPDATEおよびFOR SHAREは、返される行がテーブルのどの行に対応するのが明確に識別できない場合には使用することができません。例えば、集約には使用できません。

FOR UPDATEまたはFOR SHAREがSELECT問い合わせの最上位レベルに存在する場合、ロック対象行は問い合わせが返す行に正確に一致します。結合問い合わせ内の場合、ロック対象行は返される結合行に関連する行となります。さらに、スナップショットを更新した後に問い合わせ条件を満たさなくなった場合は返されなくなりますが、問い合わせのスナップショット時点で問い合わせ条件を満たす行もロックされます。LIMITが使用された場合、制限を満たす行が返されるとロック処理は止まります。（しかし、OFFSETにより飛ばされた行はロックされることに注意してください。）同様に、FOR UPDATEまたはFOR SHAREがカーソル問い合わせで使用された場合、カーソルにより実際に取り込んだ行または過去に処理された行のみがロックされます。

FOR UPDATEまたはFOR SHAREが副SELECTに存在する場合、ロック対象行は副問い合わせの外側の問い合わせで返される行となります。外側の問い合わせからの条件が副問い合わせ実行の最適化に使用される可能性がありますので、これには副問い合わせ自体の検査が提示する行より少なくなるかもしれません。例えば、

```
SELECT * FROM (SELECT * FROM mytable FOR UPDATE) ss WHERE col1 = 5;
```

は、副問い合わせ内では文字として条件が記載されていなくても、col1 = 5を持つ行のみがロックされます。

注意

行をロックしてから後のセーブポイントやPL/pgSQL例外ブロックでその行を編集することは避けてください。後のロールバックでロックが失われてしまいます。以下に例を示します。

BEGIN;
SELECT * FROM mytable WHERE key = 1 FOR UPDATE;
SAVEPOINT s;
UPDATE mytable SET ... WHERE key = 1;
ROLLBACK TO s;

ROLLBACK後、行は、セーブポイント前のロックされた更新前の状態に戻らず、実質ロックされません。現在のトランザクションでロックされた行が更新されたり削除されたりした場合や、共有ロックが排他ロックに向上した場合、これは危険です。すべての場合において、以前のロック状態が忘れられています。トランザクションがその後、元のロックコマンドと続く更新の間の状態にロールバックした場合、その行はロックされていない状態として現れます。これは実装上の問題であり、将来のPostgreSQLリリースで修正する予定です。

注意

ORDER BY句とFOR UPDATE/SHARE句を使用した、READ COMMITTEDトランザクション隔離レベルで実行するSELECTコマンドでは、順序通りにならない行を返す可能性があります。ORDER BYが最初に適用されるためです。このコマンドは結果をソートしますが、その後、1行または複数の行のロック獲得がブロックされる可能性があります。このSELECTのブロックが解除された時点で、順序付け対象の列値の一部が変更されているかもしれません。これによりこうした行が（元の列値という観点では順序

通りではありますが、) 順序通りに現れません。 必要に応じて、これは以下のように副問い合わせ内にFOR UPDATE/SHARE 句を記述することで、回避することができます。

```
SELECT * FROM (SELECT * FROM mytable FOR UPDATE) ss ORDER BY column1;
```

これは、最上位レベルにおけるFOR UPDATEは実際に返される行のみをロックするのに対して、結果としてmytableのすべての行をロックすることに注意してください。 これは、特にORDER BYがLIMITやその他の制限と組み合わせている場合、性能上大きな違いを生み出す可能性があります。 このため、この技法は、順序付け対象の列に対する同時実行の更新が想定され、かつ、厳密にソートされた結果が要求される場合にのみ推奨されます。

REPEATABLE READまたはSERIALIZABLE トランザクション隔離レベルでは、（'40001' というSQLSTATEを持つ）シリアライゼーション失敗が発生します。 このためこれらの隔離レベルでは順序外の行を受け取る可能性はありません。

TABLE コマンド

```
TABLE name
```

というコマンドは以下と完全に同じです。

```
SELECT * FROM name
```

これは、最上位のコマンドとしても複雑な問い合わせの一部として入力を省略する構文の一種としても使用することができます。

例

films テーブルをdistributors テーブルと結合します。

```
SELECT f.title, f.did, d.name, f.date_prod, f.kind
FROM distributors d, films f
WHERE f.did = d.did
```

title	did	name	date_prod	kind
The Third Man	101	British Lion	1949-12-23	Drama
The African Queen	101	British Lion	1951-08-11	Romantic
...				

全ての映画のlen 列を合計しkind 列によって結果をグループ化します。

```
SELECT kind, sum(len) AS total FROM films GROUP BY kind;
```

kind	total
Action	07:34
Comedy	02:58
Drama	14:28
Musical	06:42
Romantic	04:38

全ての映画のlen 列を合計しkind 列によって結果をグループ化し、合計が5時間より少ないグループの合計を表示します。

```
SELECT kind, sum(len) AS total
FROM films
GROUP BY kind
HAVING sum(len) < interval '5 hours';
```

kind	total
Comedy	02:58
Romantic	04:38

次に、結果を2番目の列 (name) の内容に基づいてソートする方法を2つ例示します。

```
SELECT * FROM distributors ORDER BY name;
SELECT * FROM distributors ORDER BY 2;
```

did	name
109	20th Century Fox
110	Bavaria Atelier
101	British Lion
107	Columbia
102	Jean Luc Godard
113	Luso films

```

104 | Mosfilm
103 | Paramount
106 | Toho
105 | United Artists
111 | Walt Disney
112 | Warner Bros.
108 | Westward

```

次の例は、distributors テーブルと actors テーブルの和集合を取得する方法を示しています。さらに、両方のテーブルで結果をWという文字で始まる行のみに限定しています。重複しない行のみが必要なので、ALL キーワードは省略されています。

```

distributors:          actors:
did |      name        id |      name
-----+-----
108 | Westward          1 | Woody Allen
111 | Walt Disney         2 | Warren Beatty
112 | Warner Bros.       3 | Walter Matthau
...
...

SELECT distributors.name
FROM distributors
WHERE distributors.name LIKE 'W%'
UNION
SELECT actors.name
FROM actors
WHERE actors.name LIKE 'W%';

      name
-----
Walt Disney
Walter Matthau
Warner Bros.
Warren Beatty
Westward
Woody Allen

```

次に、FROM 句内での関数の使用方法について、列定義リストがある場合とない場合の両方の例を示します。

```

CREATE FUNCTION distributors(int) RETURNS SETOF distributors AS $$
    SELECT * FROM distributors WHERE did = $1;
$$ LANGUAGE SQL;

SELECT * FROM distributors(111);
did |      name
-----+-----
111 | Walt Disney

CREATE FUNCTION distributors_2(int) RETURNS SETOF record AS $$
    SELECT * FROM distributors WHERE did = $1;
$$ LANGUAGE SQL;

SELECT * FROM distributors_2(111) AS (f1 int, f2 text);
f1 |      f2
-----+-----
111 | Walt Disney

```

以下の例では簡単なWITH 句の使用方法を示します。

```

WITH t AS (
    SELECT random() as x FROM generate_series(1, 3)
)
SELECT * FROM t
UNION ALL
SELECT * FROM t

      x
-----
0.534150459803641
0.520092216785997
0.0735620250925422
0.534150459803641
0.520092216785997
0.0735620250925422

```

WITH 問い合わせが一度だけ評価されることに注意してください。このため3つのランダムな値の2つの集合を得ることになります。

以下の例ではWITH RECURSIVE を使用して、直接の部下しか表示しないテーブルから、従業員Maryの（直接または間接的な）部下とその間接度を見つけ出します。

```

WITH RECURSIVE employee_recursive(distance, employee_name, manager_name) AS (
    SELECT 1, employee_name, manager_name
    FROM employee

```

```

WHERE manager_name = 'Mary'
UNION ALL
SELECT er.distance + 1, e.employee_name, e.manager_name
FROM employee_recursive er, employee e
WHERE er.employee_name = e.manager_name
)
SELECT distance, employee_name FROM employee_recursive;
```

初期条件、続いてUNION、さらに問い合わせの再帰部分という再起問い合わせの典型的な構文に注意してください。問い合わせの再帰部分は最終的にはタプルを返さないことを確実にしてください。さもないと問い合わせは無限にループします。（多くの例については[項7.8](#)を参照してください。）

互換性

当然ながら、SELECT文は標準SQLと互換性があります。しかし、拡張機能や実現されていない機能もいくつかあります。

FROM句の省略

PostgreSQLでは、FROM句を省略することができます。これによって、以下のように単純な式を計算させることができます。

```

SELECT 2+2;

?column?
-----
4
```

他のSQLデータベースでは、このようなSELECTを行うためにはダミーの1行テーブルを使わなければなりません。

FROM句の指定がない場合、問い合わせではデータベーステーブルを参照することができません。例えば、以下の問い合わせは無効です。

```

SELECT distributors.* WHERE distributors.name = 'Westward';
```

PostgreSQLリリース8.1より前まででは、こうした形の問い合わせを受け付け、問い合わせで参照する各テーブルに対する暗黙的な項目を問い合わせのFROM句に追加していました。これは許されなくなりました。

ASキーワードの省略

標準SQLでは、省略可能なキーワードASは、新しい列名が有効な列名（つまり予約済みのキーワードと異なるものすべて）である場合は常に、出力列名の前から省くことができます。PostgreSQLには多少より強い制限があります。新しい列名が予約済みか否かに関わらず何らかのキーワードに一致する場合はASが必要です。推奨する実践方法は、今後のキーワードの追加と競合する可能性に備え、ASを使用する、または出力列名を二重引用符で括ることです。

FROM項目において標準およびPostgreSQLでは、未予約のキーワードである別名の前のASを省略することができます。しかし、構文があいまいになるため、出力名では現実的ではありません。

ONLYと継承関係

標準SQLでは、SELECT * FROM ONLY (tab1), ONLY (tab2) WHERE ...のように、ONLYを記述する時にテーブル名の前後を括弧でくくることを要求します。PostgreSQLではこの括弧を省略可能であるとみなしています。

PostgreSQLでは最後に*を付けることで明示的に子テーブルを含めるというONLYではない動作を指定することができます。標準ではこれを許していません。

（この点はONLYオプションをサポートするすべてのSQLコマンドで同様に適用されます。）

GROUP BYとORDER BYにおける利用可能な名前空間

標準SQL-92では、ORDER BY句で利用できるのは、出力列名か序数のみであり、GROUP BY句で利用できるのは、入力列名からなる式のみです。PostgreSQLは、これらの句で両方が指定できるように拡張されています（ただし、不明瞭さがある場合は標準の解釈が使用されます）。さらに、PostgreSQLではどちらの句にも任意の式を指定できます。式で使われる名前は、常に出力列名ではなく入力列の名前とみなされることに注意してください。

SQL:1999以降では、SQL-92との上位互換性がまったくない、多少異なる定義が採用されています。しかし、ほとんどの場

合、PostgreSQLはSQL:1999と同じ方法でORDER BYやGROUP BYを解釈します。

関数依存性

テーブルのプライマリーキーがGROUP BYリストに含まれる場合に限り、PostgreSQLは（GROUP BYで列を省くことができる）関数依存性を認識します。 標準SQLでは、認識しなければならない追加の条件を規定しています。

WINDOW句の制限

標準SQLではウィンドウ用のframe_clauseオプションをさらに提供します。 現在のPostgreSQLでは上述のオプションのみをサポートします。

LIMITおよびOFFSET

LIMITおよびOFFSET句はPostgreSQL独自の構文ですが、MySQLでも使用されています。 [LIMIT句](#)で説明したように、標準SQL:2008にて同じ機能のOFFSET ... FETCH {FIRST|NEXT} ...が導入されました。 この構文はIBM DB2でも使用されています。（PostgreSQLでは利用できませんが、Oracle用に開発されたアプリケーションでは、これらの句の機能を実装するためによく自動生成されるrownum列を含めるという回避策を使用します。）

FOR UPDATEおよびFOR SHARE

FOR UPDATEは標準SQLに存在しますが、標準では、DECLARE CURSORのオプションとしてしか許されていません。 PostgreSQLでは、副SELECTなど任意のSELECTで許されます。 これは拡張です。 FOR SHAREの亜種、およびNOWAITオプションは標準にはありません。

WITH内のデータ変更文

PostgreSQLではWITH問い合わせとしてINSERT、UPDATEおよびDELETEを使用することができます。 これは標準SQLにはありません。

非標準句

DISTINCT ON句は標準SQLでは定義されていません。

[前のページ](#)

SECURITY LABEL

[ホーム](#)
[上に戻る](#)

[次のページ](#)
SELECT INTO

SELECT INTO

名前

SELECT INTO -- 問い合わせの結果からの新しいテーブルを定義する

概要

```

[ WITH [ RECURSIVE ] with_query [, ...] ]
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    * | expression [ [ AS ] output_name ] [, ...]
INTO [ TEMPORARY | TEMP | UNLOGGED ] [ TABLE ] new_table
[ FROM from_item [, ...] ]
[ WHERE condition ]
[ GROUP BY expression [, ...] ]
[ HAVING condition [, ...] ]
[ WINDOW window_name AS ( window_definition ) [, ...] ]
[ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] select ]
[ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ] [, ...] ]
[ LIMIT { count | ALL } ]
[ OFFSET start [ ROW | ROWS ] ]
[ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ]
[ FOR { UPDATE | SHARE } [ OF table_name [, ...] ] [ NOWAIT ] [...] ]

```

説明

SELECT INTOは新しいテーブルを作成し、そこに問い合わせによって計算したデータを格納します。このデータは通常のSELECTのようにクライアントに返されません。新しいテーブルの列はSELECTの出力列に関連するデータ型と名前を持ちます。

パラメータ

TEMPORARY or TEMP

このオプションが指定された場合、テーブルは一時テーブルとして作成されます。詳細は[CREATE TABLE](#)を参照してください。

UNLOGGED

指定された場合、テーブルはログをとらないテーブルとして作成されます。 [CREATE TABLE](#)を参照してください。

new_table

作成するテーブルの名前です（スキーマ修飾名も可）。

その他のパラメータについては、[SELECT](#)で詳細に説明されています。

注釈

[CREATE TABLE AS](#)は機能的にはSELECT INTOと同じです。 INTO句の解釈が異なるため、SELECT INTOという形式はECPGやPL/pgSQLでは使用できません。そのため、CREATE TABLE AS構文を使用することをお勧めします。さらに、CREATE TABLE ASは、SELECT INTOの機能に加え、さらに多くの機能を提供します。

PostgreSQL 8.1より前は、SELECT INTOで作成されるテーブルはデフォルトでOIDを含んでいました。PostgreSQL 8.1ではこの問題はありません。新しいテーブルにOIDを含めたいければ、[default_with_oids](#)設定変数を有効にする必要があります。WITH OIDS句を付けてCREATE TABLE ASを実行するという別の方法もあります。

例

`films` テーブルの最近の項目のみから構成される、新しい`films_recent` テーブルを作成します。

```
SELECT * INTO films_recent FROM films WHERE date_prod >= '2002-01-01';
```

互換性

標準SQLでは、`SELECT INTO`は新しいテーブルの作成ではなく、選択した値をホストプログラムのスカラ変数とするために使われます。これは実際、ECPG（[第33章](#)を参照）やPL/pgSQL（[第39章](#)を参照）で見られる使用方法です。PostgreSQLにおいて、テーブルを作成する`SELECT INTO`の用法は歴史的なものです。新しいコードでは、テーブルの作成には`CREATE TABLE AS`を使うのが最善です。

関連項目

[CREATE TABLE AS](#)

[前のページ](#)

SELECT

[ホーム](#)

[上に戻る](#)

[次のページ](#)

SET

SET

名前

SET -- 実行時パラメータを変更する

概要

```

SET [ SESSION | LOCAL ] configuration_parameter { TO | = } { value | 'value' | DEFAULT }
SET [ SESSION | LOCAL ] TIME ZONE { timezone | LOCAL | DEFAULT }

```

説明

SET コマンドは実行時設定パラメータを変更します。 [第18章](#)に列挙されている実行時パラメータの多くは稼働中にSET コマンドで変更できます（ただし、変更するためにスーパーユーザ権限が必要な場合があります。また、サーバあるいはセッションの始動後は変更できないパラメータもあります）。SET は現行セッションで使用される値にのみ影響することに注意してください。

SET (または同等のSET SESSION)が発行された後にトランザクションがアボートされると、トランザクションがロールバックした時点でSET コマンドの効力は失われます。一度トランザクションがコミットされると、別のSET コマンドで上書きされない限り、セッションが終了するまでその効果が持続します。

SET LOCAL の効果は、コミットの有無にかかわらず現在のトランザクションが終了するまでしか持続しません。1つのトランザクション内でSET の後にSET LOCAL が続く特殊な例を考えてみましょう。この場合、トランザクションが終了するまではSET LOCAL 値が有効ですが、その後（トランザクションがコミットされると）SET 値が有効になります。

SET もしくはSET LOCAL の効果は、このコマンドより以前のセーブポイントまでロールバックすると取り消されます。

SET LOCAL を同じ変数に対するSET オプション（[CREATE FUNCTION](#)を参照）を持つ関数内で使用する場合、SET LOCAL コマンドの効果は関数終了時に消滅します。つまり、関数が呼び出された時に有効だった値にとにかく戻されます。これによりSET LOCAL は、呼び出し元の値を保管し元に戻すというSET オプションを使用する利点を持ちつつ、関数内で動的または繰り返し変更されるパラメータ用に使用できます。しかし、通常のSET コマンドでは下位の関数のSET オプションを上書きしてしまい、その効果はロールバックしない限り永続します。

注意: PostgreSQLバージョン8.0から8.2まででは、SET LOCAL の効果は、より以前のセーブポイントを解放すること、または、PL/pgSQL例外ブロックから正常終了することで取り消されました。直観的ではないようですが、この動作は変更されました。

パラメータ

SESSION

コマンドの有効範囲が現行セッションであることを指定します（SESSION も LOCAL も指定されていない場合は、これがデフォルトです）。

LOCAL

コマンドの有効範囲が現行のトランザクションのみであることを指定します。COMMIT または ROLLBACK の後は、再びセッションレベルの設定が有効になります。SET LOCAL をBEGIN ブロックの外側で実行すると、トランザクションが即座に終了するため、一見何も実行されていないように見えます。

configuration_parameter

設定可能な実行時パラメータ名です。利用可能なパラメータは、[第18章](#)と以下に示します。

value

パラメータの新しい値です。値として、対象のパラメータで適切となるように、文字列定数、識別子、数字、あるいはこれらをカンマで区切ったリストを指定することができます。DEFAULT と記述することで、パラメータをデフォルト値（つまり、現在のセッションでSET が実行されなかった時に設定される値）に再設定することができます。

[第18章](#)に記載された設定パラメータの他にも、`SET`コマンドを使用して調整するしかないパラメータや特殊な構文を持つパラメータがいくつかあります。

SCHEMA

`SET SCHEMA 'value'`は`SET search_path TO value`の別名です。この構文を使用する場合は1つのスキーマのみを指定することができます。

NAMES

`SET NAMES value`は、`SET client_encoding TO value`の別名です。

SEED

乱数ジェネレータ (`random`関数) 用の内部シードを設定します。-1から1までの浮動小数点数を値として設定できます。その後、この値には $2^{31}-1$ がかけられます。

シードは`setseed`関数を呼び出すことでも設定可能です。

```
SELECT setseed(value);
```

TIME ZONE

`SET TIME ZONE value`は`SET timezone TO value`の別名です。`SET TIME ZONE`構文では、時間帯の指定に特殊な構文を使用できます。有効な値の例を以下に示します。

```
'PST8PDT'
```

カリフォルニア州バークレイ用の時間帯です。

```
'Europe/Rome'
```

イタリア用の時間帯です。

```
-7
```

UTCから西に7時間分ずらした時間帯です (PDTと同じです)。正の値はUTCから東方向です。

```
INTERVAL '-08:00' HOUR TO MINUTE
```

UTCから西に8時間分ずらした時間帯です (PSTと同じです)。

```
LOCAL  
DEFAULT
```

時間帯をユーザのローカルな時間帯 (サーバのデフォルトの`timezone`値) に設定します。

時間帯に関する詳細は[項8.5.3](#)を参照してください。

注釈

`set_config`関数は等価な機能を提供します。[項9.26](#)を参照してください。また、[pg_settings](#)システムビューを更新することで、`SET`と同じことを実行することができます。

例

スキーマの検索パスを設定します。

```
SET search_path TO my_schema, public;
```

日付のスタイルを、"day before month"の伝統的なPOSTGRES入力方式に設定します。

```
SET datestyle TO postgres, dmy;
```

時間帯をカリフォルニア州バークレイに設定します。

```
SET TIME ZONE 'PST8PDT';
```

時間帯をイタリアに設定します。

```
SET TIME ZONE 'Europe/Rome';
```

互換性

`SET TIME ZONE`は標準SQLで定義された構文を拡張したものです。 標準では数値による時間帯オフセットしか使用できないのに対し、PostgreSQLでは、より柔軟に時間帯を指定することができます。 `SET`が持つその他の機能は、全てPostgreSQLの拡張です。

関連項目

[RESET](#), [SHOW](#)

[前のページ](#)
SELECT INTO

[ホーム](#)
[上に戻る](#)

[次のページ](#)
SET CONSTRAINTS

SET CONSTRAINTS

名前

SET CONSTRAINTS -- 現在のトランザクションの制約検査のタイミングを設定する

概要

```
SET CONSTRAINTS { ALL | name [, ...] } { DEFERRED | IMMEDIATE }
```

説明

SET CONSTRAINTSは、現在のトランザクションにおける制約の検査方法を設定します。IMMEDIATE制約は、1つの文の実行が終わるごとに検査されます。DEFERRED制約は、トランザクションがコミットされるまで検査されません。全ての制約は、IMMEDIATEかDEFERREDのどちらかのモードを持ちます。

制約にはその生成時点で、DEFERRABLE INITIALLY DEFERRED、DEFERRABLE INITIALLY IMMEDIATE、NOT DEFERRABLEの3つのうちのいずれかの性質が与えられます。3番目のNOT DEFERRABLE制約は、常にIMMEDIATEモードであり、SET CONSTRAINTSコマンドの影響を受けません。DEFERRABLE INITIALLY DEFERRED制約とDEFERRABLE INITIALLY IMMEDIATE制約の2つは、トランザクションを指定されたモードで開始しますが、トランザクション内でSET CONSTRAINTSを使用するとその振舞いを変更することができます。

制約名のリストを持ったSET CONSTRAINTSが変更するのは、これらの制約のモードのみです（これらは全て遅延可能です）。制約名はそれぞれスキーマ修飾可能です。スキーマ名が指定されていない場合、最初に一致する名前を見つけ出すために現在のスキーマ検索パスが使用されます。SET CONSTRAINTS ALLは遅延可能な全ての制約のモードを変更します。

制約のモードをDEFERREDからIMMEDIATEに変更した場合は、新しい制約モードが遡及的に有効になります。つまりDEFERREDモードであればトランザクションの終了時に検査される未検査のデータ変更が、SET CONSTRAINTSコマンドの実行中に検査されます。もし、この時に何らかの制約違反があった場合、SET CONSTRAINTSは失敗します（そして、制約モードは変更されません）。したがって、SET CONSTRAINTSを利用すれば、トランザクションの特定の時点で強制的に制約の検査を実行することができます。

現在UNIQUE、PRIMARY KEY、REFERENCES（外部キー）、EXCLUDE制約のみがこの設定の影響を受けます。NOT NULLおよびCHECK制約は、行が挿入または変更された時に（文の終了時ではありません）、常に即座に検査されます。DEFERRABLE宣言されていない一意性制約および排除制約も即座に検査されます。

また、"制約トリガ"として宣言されたトリガの発行もこの設定により制御されます。これらは関連する制約が検査されるはずの時に同時に発行されます。

注釈

PostgreSQLでは、スキーマ内で制約名が一意となることを要求していません（テーブル単位での一意性を要求します）ので、指定した制約名に複数が一致する可能性があります。この場合SET CONSTRAINTSは一致するすべてに対して動作します。スキーマ修飾がない名前では、検索パス上のあるスキーマで1つまたは複数の一致があると、パス上で後に見つかったスキーマは無視されます。

このコマンドが変更するのは、現在のトランザクション内の制約の動作のみです。したがって、トランザクションブロック（BEGIN/COMMITの間）の外部でこのコマンドが実行されても場合、何ら効果はありません。

互換性

このコマンドは、標準SQLで定義された動作に準拠しています。ただし、PostgreSQLではNOT NULLおよびCHECK制約に適用できないという制限があります。またPostgreSQLは非遅延一意性制約を、標準が提案する文の終わりにではなく、即座に検査します。

SET ROLE

名前

SET ROLE -- 現在のセッションにおける現在のユーザ識別子を設定する

概要

```
SET [ SESSION | LOCAL ] ROLE role_name
SET [ SESSION | LOCAL ] ROLE NONE
RESET ROLE
```

説明

このコマンドは現在のSQLセッションにおける現在のユーザ識別子を`role_name`に設定します。 ロール名は識別子あるいは文字列リテラルのどちらを使用しても記述することができます。 `SET ROLE`の後、SQLコマンドに対する権限検査は、指名されたロールで普通にログインした場合と同様に行われます。

指定する`role_name`は、現在のセッションユーザがメンバとして属するロールでなければなりません。（セッションユーザがスーパーユーザであった場合、任意のロールを選択することができます。）

`SESSION`および`LOCAL`修飾子は通常の[SET](#)コマンドと同様に動作します。

`NONE`および`RESET`形式は、現在のユーザ識別子を現在のセッションユーザ識別子に戻します。 この形式はすべてのユーザが実行することができます。

注釈

このコマンドを使用して、権限を追加することも制限することもできます。 セッションユーザのロールが`INHERITS`属性を持つ場合、自動的に`SET ROLE`で設定されたすべてのロールの権限を持ちます。 この場合、`SET ROLE`は実際、セッションユーザに直接割り当てられている権限、セッションユーザが属するロールに割り当てられている権限の内、指名されたロールで使用可能な権限を残し、他をすべて削除します。 一方、セッションユーザのロールが`NOINHERITS`属性を持つ場合、セッションユーザに直接割り当てられた権限をすべて削除し、指名されたロールで利用可能な権限を獲得します。

特に、スーパーユーザが非特権ユーザへの`SET ROLE`を行うと、スーパーユーザ権限を失うことになります。

`SET ROLE`の影響は[SET SESSION AUTHORIZATION](#)と似ていますが、行われる権限検査はかなり異なります。 また、`SET SESSION AUTHORIZATION`は、その後に実行する`SET ROLE`コマンドでどのロールに変更できるかを決定しますが、`SET ROLE`を使用してロールを変更した場合、その後に実行する`SET ROLE`コマンドで変更可能なロール群は変更されません。

`SET ROLE`はロールの[ALTER ROLE](#)設定で指定されたセッション変数を処理しません。 これはログイン時のみ適用されます。

`SET ROLE`を`SECURITY DEFINER`関数で使用することはできません。

例

```
SELECT SESSION_USER, CURRENT_USER;

 session_user | current_user 
-----+-----
    peter     |    peter

SET ROLE 'paul';

SELECT SESSION_USER, CURRENT_USER;

 session_user | current_user 
-----+-----
    peter     |    paul
```


互換性

PostgreSQLでは、識別子構文 ("`rolename`") を使用できます。しかし、標準SQLではロール名を文字列リテラルとして記述しなければなりません。SQLでは、トランザクション内でこのコマンドを実行することを許可していません。この制限の理由がありませんので、PostgreSQLではこの制限はありません。`SESSION`、`LOCAL`修飾子、および、`RESET`構文はPostgreSQLの拡張です。

関連項目

[SET SESSION AUTHORIZATION](#)

前のページ	ホーム	次のページ
SET CONSTRAINTS	上に戻る	SET SESSION AUTHORIZATION

SET SESSION AUTHORIZATION

名前

SET SESSION AUTHORIZATION -- セッションのユーザ識別子、現在のセッションの現在のユーザ識別子を設定する

概要

```
SET [ SESSION | LOCAL ] SESSION AUTHORIZATION user_name
SET [ SESSION | LOCAL ] SESSION AUTHORIZATION DEFAULT
RESET SESSION AUTHORIZATION
```

説明

このコマンドはセッションのユーザ識別子、ならびに、現在のSQLセッションにおける現在のユーザ識別子を`user_name`に設定します。ユーザ名は、識別子だけでなく、文字列リテラルとして記述することもできます。このコマンドを使用すると、例えば、一時的に非特権ユーザとなり、その後の特権ユーザに戻るといったことが可能です。

セッションのユーザ識別子はクライアントから渡される（おそらく認証済みの）ユーザ名で初期化されます。現在のユーザ識別子は通常セッションのユーザ識別子と同一ですが、`SECURITY DEFINER`関数や類似の機能によって一時的に変更される可能性があります。[SET ROLE](#)でこれを変更することもできます。現在のユーザ識別子は権限の検査に影響を与えます。

セッションのユーザ識別子は、最初のセッションユーザ（認証されたユーザ）がスーパーユーザ権限を持っている場合にのみ変更できます。スーパーユーザ権限を持っていない場合、認証されたユーザ名以外を指定してもこのコマンドは受け入れられません。

`SESSION`修飾子および`LOCAL`修飾子は、通常の[SET](#)コマンドの場合と同じように機能します。

`DEFAULT`構文および`RESET`構文は、セッションと現在のユーザ識別子を元の認証ユーザに戻します。これらの構文は全てのユーザが実行できます。

Notes

`SET SESSION AUTHORIZATION`を`SECURITY DEFINER`関数で使用することはできません。

例

```
SELECT SESSION_USER, CURRENT_USER;

 session_user | current_user 
-----+-----
peter        | peter

SET SESSION AUTHORIZATION 'paul';

SELECT SESSION_USER, CURRENT_USER;

 session_user | current_user 
-----+-----
paul          | paul
```

互換性

標準SQLでは、`user_name`リテラルの指定方法としてその他の表現を使用できます。しかし、この違いは実用上は重要ではありません。PostgreSQLでは識別子構文 ("`username`")を使用できますが、SQLでは使用できません。SQLではこのコマンドをトランザクション中に実行することができませんが、PostgreSQLでは、禁止する理由が見当たらないため、この制限を付けていません。`SESSION`修飾子および`LOCAL`修飾子は`RESET`構文同様、PostgreSQLの拡張です。

標準SQLでは、このコマンドを実行するために必要な権限は、実装に依存するとされています。

関連項目

[SET ROLE](#)

[前のページ](#)
SET ROLE

[ホーム](#)
[上に戻る](#)

[次のページ](#)
SET TRANSACTION

SET TRANSACTION

名前

SET TRANSACTION -- 現在のトランザクションの特性を設定する

概要

```
SET TRANSACTION transaction_mode [, ...]
SET TRANSACTION SNAPSHOT snapshot_id
SET SESSION CHARACTERISTICS AS TRANSACTION transaction_mode [, ...]
```

ここで`transaction_mode`は以下のいずれかです。

```
ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED }
READ WRITE | READ ONLY
[ NOT ] DEFERRABLE
```

説明

SET TRANSACTIONは現在のトランザクションの特性を設定します。これはその後のトランザクションには影響を及ぼしません。SET SESSION CHARACTERISTICSは、セッションにおけるそれ以後のトランザクションのデフォルトのトランザクション特性を設定します。SET TRANSACTIONにより個々のトランザクションのデフォルト特性を上書きすることができます。

利用可能なトランザクション特性はトランザクションの隔離レベル、トランザクションのアクセスモード（読み書きモードもしくは読み取りのみモード）、遅延モードです。さらに、セッションのデフォルトとしてではなく、現在のトランザクションのみに対してスナップショットを選択することができます。

トランザクションの隔離レベルは、並行して実行中の他のトランザクションが存在する場合、そのトランザクションが見ることができるデータを決定するものです。

READ COMMITTED

1つひとつの文から見ることは、その文が開始される前にコミットされた行のみです。これがデフォルトです。

REPEATABLE READ

現在のトランザクションにおける全ての文は、トランザクションで最初の問い合わせ文またはデータを変更する文が実行される前にコミットされた行だけを見ることができます。

SERIALIZABLE

現在のトランザクションにおける全ての文は、トランザクションで最初の問い合わせ文またはデータを変更する文が実行される前にコミットされた行だけを見ることができます。同時実行のシリアライズブルトランザクションの中で読み取りと書き込みのパターンによって、これらのトランザクションの実行を直列に(同時に一度)行うことができない状況になる場合、その内1つのトランザクションは`serialization_failure`というエラーでロールバックされます。

標準SQLでは、さらにREAD UNCOMMITTEDという1つのレベルを定義しています。PostgreSQLではREAD UNCOMMITTEDはREAD COMMITTEDとして扱われます。

トランザクション隔離レベルは、そのトランザクションにおける最初の問い合わせ文やデータ更新文（SELECT、INSERT、DELETE、UPDATE、FETCH、COPY）が実行された後からは変更することができません。トランザクションの隔離や同時実行制御についての詳細情報は[第13章](#)を参照してください。

トランザクションのアクセスモードは、そのトランザクションが読み書き可能か読み取りのみかを決定します。デフォルトは読み書き可能です。読み取りのみのトランザクションでは、書き込み対象のテーブルが一時テーブルでない限り、INSERT、UPDATE、DELETE、COPY FROMなどのSQLコマンドを実行できません。また、CREATE、ALTER、DROP系の全てのSQLコマンド、COMMENT、GRANT、REVOKE、TRUNCATEは、まったく実行できません。さらに、EXPLAIN ANALYZEとEXECUTEコマンドに上述のコマンドの実行が含まれている場合、これらのコマンドも実行できません。この方法ではディスクへの書き込み防止をまったく行わないので、読み取り専用を高レベルで実現する考え方と言えます。

DEFERRABLE トランザクション属性は、トランザクションがSERIALIZABLEかつREAD ONLYである場合のみ効果があります。あるトランザクションでこれら3つの属性がすべて選択されている場合、最初にスナップショットを獲得する時にブロックされる可能

性があります。その後、そのトランザクションをSERIALIZABLEトランザクションの通常のオーバーヘッドを伴わず、またシリアルライズ処理の失敗を引き起こす恐れやシリアルライズ処理の失敗によりキャンセルされる恐れもなく実行することができます。これは時間がかかるレポート処理やバックアップによく適しています。

SET TRANSACTION SNAPSHOT コマンドにより、既存のトランザクションと同じスナップショットを持つ新しいトランザクションを実行することができます。既存のトランザクションはpg_export_snapshot関数(項9.26.5参照)を使用してそのスナップショットを公開していなければなりません。この関数はスナップショット識別子を返します。この識別子を、どのスナップショットを取り込むかを指定するためにSET TRANSACTION SNAPSHOTに渡さなければなりません。このコマンドでは、この識別子を例えば'000003A1-1'のようにリテラル文字列として記述しなければなりません。トランザクションの開始時に、トランザクションの最初の問い合わせまたはデータ変更文(SELECT、INSERT、DELETE、UPDATE、FETCH、COPY)を行う前でのみ、SET TRANSACTION SNAPSHOTを実行することができます。さらに、そのトランザクションを前もってSERIALIZABLEまたはREPEATABLE READ隔離レベルに設定していなければなりません。(さもないと、READ COMMITTEDではコマンドそれぞれに対して新しいスナップショットを取りますので、このスナップショットは即座に破棄されます。)取り込むトランザクションがSERIALIZABLE隔離レベルを使用している場合、スナップショットを公開したトランザクションもこの隔離レベルを使用しなければなりません。また、読み取り専用ではないシリアルライズバブルトランザクションは、読み取り専用トランザクションから公開されたスナップショットを取り込むことができません。

注釈

SET TRANSACTIONをSTART TRANSACTIONやBEGINの前に実行した場合、そのトランザクションは即座に終了するので、効果はまったく現れません。

BEGINあるいはSTART TRANSACTIONで目的のtransaction_modesを指定すれば、SET TRANSACTIONは不要です。しかしSET TRANSACTION SNAPSHOTではこの選択を行うことができません。

セッションのデフォルトのトランザクションモードは、設定パラメータdefault_transaction_isolation、default_transaction_read_only、default_transaction_deferrableで設定することができます(実際、SET SESSION CHARACTERISTICSは冗長な記述であり、これらの変数をSETで設定するのと変わりありません)。したがって、トランザクションモードのデフォルトは設定ファイルやALTER DATABASEなどで設定可能です。詳細は第18章を参照してください。

例

既存のトランザクションと同じスナップショットを持つトランザクションを新しく開始するためには、まず既存のトランザクションからスナップショットを公開します。以下の例に示すように、これはスナップショット識別子を返します。

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT pg_export_snapshot();
       pg_export_snapshot
-----
000003A1-1
(1 row)
```

そして、新規に開始したトランザクションの先頭のSET TRANSACTION SNAPSHOTでこのスナップショット識別子を渡します。

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SET TRANSACTION SNAPSHOT '000003A1-1';
```

互換性

このコマンドは標準SQLで定義されています。DEFERRABLEトランザクションモードとSET TRANSACTION SNAPSHOT構文は例外であり、PostgreSQLの拡張です。

SERIALIZABLE is the default transaction isolation level in the standard. In PostgreSQL the default is ordinarily READ COMMITTED, but you can change it as mentioned above. --> 標準SQLではデフォルトのトランザクションはSERIALIZABLEです。PostgreSQLでは、通常、READ COMMITTEDがデフォルトですが、これは上述の通り変更可能です。

標準SQLでは、もう1つ、診断領域の大きさというトランザクション特性があり、このコマンドで設定可能です。この概念は組み込みSQL固有のものなので、PostgreSQLサーバには実装されていません。

標準SQLでは、連続するtransaction_modesの間にはカンマが必要です。歴史的な理由よりPostgreSQLではカンマを省略することができます。

SHOW

名前

SHOW -- 実行時パラメータの値を表示する

概要

```
SHOW name
SHOW ALL
```

説明

SHOWは、実行時パラメータの現在の設定を表示します。 これらの変数は、SET文、postgresql.confの編集、PGOPTIONS環境変数（libpqあるいはlibpqを使用したアプリケーションの使用時）、または、postgresサーバの始動時のコマンドラインフラグで設定することができます。 詳細は[第18章](#)を参照してください。

パラメータ

name

実行時パラメータの名前です。 利用可能なパラメータは[第18章](#)と[SET](#)マニュアルページに記載されています。 さらに、表示のみ可能で、変更できない次のようなパラメータがあります。

SERVER_VERSION

サーバのバージョン番号を示します。

SERVER_ENCODING

サーバ側の文字セット符号化方式を表示します。 現時点では、符号化方式はデータベース作成時に決定されるため、このパラメータは表示のみ可能で、変更することができません。

LC_COLLATE

データベースの照合（テキストの順序付け）についてのロケール設定を示します。 現時点では、この設定はデータベース作成時に決定されるため、このパラメータは表示のみ可能で、変更することができません。

LC_CTYPE

文字分類についてのデータベースのロケール設定を表示します。 現時点では、この設定はデータベース作成時に決定されるため、このパラメータは表示のみ可能で、変更することができません。

IS_SUPERUSER

現在のロールがスーパーユーザ権限を持つ場合は真になります。

ALL

全ての設定パラメータの値とその説明を表示します。

注釈

current_setting関数は同等の出力を生成します。 [項9.26](#)を参照してください。 また、[pg_settings](#)システムビューは同じ情報を生成します。

例

DateStyleパラメータの現在の設定を表示します。

```
SHOW DateStyle;
DateStyle
-----
ISO, MDY
(1 row)
```

geqoパラメータの現在の設定を表示します。

```
SHOW geqo;
geqo
-----
on
```


(1 row)

全設定を表示します。

SHOW ALL;

name	setting	description
allow_system_table_mods	off	Allows modifications of the structure of ...
.	.	.
xmloption	content	Sets whether XML data in implicit parsing ...
zero_damaged_pages	off	Continues processing past damaged page headers.

(196 Rows)

互換性

SHOW コマンドはPostgreSQLの拡張です。

関連項目

[SET](#), [RESET](#)

START TRANSACTION

名前

START TRANSACTION -- トランザクションブロックを開始する

概要

```
START TRANSACTION [ transaction_mode [, ...] ]
```

*transaction_mode*には以下のいずれかが入ります。

```
ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED }  
READ WRITE | READ ONLY  
[ NOT ] DEFERRABLE
```

説明

このコマンドは新しいトランザクションブロックを開始します。 隔離レベルや読み取り/書き込みモード、遅延モードを指定すると、[SET TRANSACTION](#)が実行された時のように、新しいトランザクションはそれらの特性を持ちます。 このコマンドの機能は、[BEGIN](#)コマンドと同じです。

パラメータ

この文のパラメータの意味については[SET TRANSACTION](#)を参照してください。

互換性

標準SQLでは、トランザクションブロック開始時のSTART TRANSACTIONコマンドの発行は必須ではありません。 任意のSQLコマンドが暗黙的にブロックを開始するからです。 PostgreSQLでは、START TRANSACTION（もしくはBEGIN）が実行されていない状態で発行されたコマンドは、その直後に、暗黙的なCOMMITが発行されたかのように動作します。 これは"自動コミット"と呼ばれます。 他のリレーショナルデータベースシステムの中にも、簡便性のために自動コミット機能を提供しているものもあります。

DEFERRABLE *transaction_mode*はPostgreSQLの言語拡張です。

標準SQLでは、連続する*transaction_modes*の間にはカンマが必須です。 しかし、PostgreSQLでは歴史的な理由によりカンマを省略することができます。

[SET TRANSACTION](#)の互換性の節も参照してください。

関連項目

[BEGIN](#), [COMMIT](#), [ROLLBACK](#), [SAVEPOINT](#), [SET TRANSACTION](#)

[前のページ](#)

SHOW

[ホーム](#)

[上に戻る](#)

[次のページ](#)

TRUNCATE

TRUNCATE

名前

TRUNCATE -- 1テーブルまたはテーブル群を空にする

概要

```
TRUNCATE [ TABLE ] [ ONLY ] name [ * ] [, ... ]  
[ RESTART IDENTITY | CONTINUE IDENTITY ] [ CASCADE | RESTRICT ]
```

説明

TRUNCATEはテーブル群から全ての行を素早く削除します。各テーブルに対して条件指定のないDELETEコマンドの実行と同じ効果を持ちますが、実際にテーブルを走査しない分、このコマンドの方が高速です。さらに、その後にVACUUM操作を行うことなく、このコマンドはディスク領域を即座に回収します。このコマンドは、大きなテーブルを対象とする場合に最も有用です。

パラメータ

name

空にするテーブルの名前です（スキーマ修飾名も可）。テーブル名の前にONLYが指定されている場合、そのテーブルのみを空にします。ONLYが指定されていない場合、そのテーブルとそのすべての子テーブル（もしあれば）を空にします。省略することもできますが、テーブル名の後に*を指定することで、明示的に継承するテーブルも含まれることを示すことができます。

RESTART IDENTITY

消去されるテーブルの列により所有されるシーケンスを自動的に再起動させます。

CONTINUE IDENTITY

シーケンスの値を変更しません。これがデフォルトです。

CASCADE

指定されたテーブル、または、CASCADEにより削除対象テーブルとされたテーブルを参照する外部キーを持つテーブルすべてを自動的に空にします。

RESTRICT

外部キーにより対象のテーブルを参照するテーブルのいずれかがこのコマンドで指定されていない場合、操作を拒否します。これがデフォルトです。

注釈

テーブルを空にするためにはそのテーブルにTRUNCATE権限を持たなければなりません。

TRUNCATEは操作対象の各テーブルに対するACCESS EXCLUSIVEを獲得します。これは、この他のそのテーブルに対する同時操作をすべてブロックします。RESTART IDENTITYが指定された場合、初期化対象のシーケンスがあると、それは同様に排他ロックされます。テーブルへの同時アクセスが必要ならば、代わりにDELETEコマンドを使用しなければなりません。

そのテーブルが他のテーブルから外部キーで参照されている場合、1つのTRUNCATEでそれらのテーブルをすべて空にするように指定していない限り、TRUNCATEを使用することはできません。このような場合は、有効性を検査するためにテーブルスキャンが必要になりますが、テーブルスキャンを行うのであれば、このコマンドの利点がなくなるからです。CASCADEを使用して、自動的にすべての依存テーブルを含めることができます。しかし、意図しないデータ損失の可能性がありますので、このオプションを使用する時には十分に注意してください。

TRUNCATEは、テーブルに存在するすべてのON DELETEトリガを発行しません。しかし、ON TRUNCATEトリガを発行します。テーブルのいずれかにON TRUNCATEトリガが定義されている場合、何らかの消去が行われる前にすべてのBEFORE TRUNCATEトリガが発

行されます。 また、最後の消去がなされ、シーケンスが初期化された後すべてのAFTER TRUNCATEトリガが発行されます。 トリガは処理されるテーブルの順番（コマンドに列挙されたものが先、その後にカスケードのために追加されたもの）に発行されま

警告

TRUNCATEはMVCC的に安全ではありません。（MVCCに関する一般的な情報については第13章を参照してください。） 削除の後テーブルはすべてのトランザクションから見て、たとえそのトランザクションが始まる前のスナップショットを使用していたとしても、空になります。 削除が始まる前にテーブルを参照しなかったトランザクションに関してのみ問題となります。 そうしたトランザクションでは少なくとも ACCESS SHAREロックを保持しますので、そのトランザクションが終わるまでTRUNCATEはブロックされます。 このため、トランザクションでは、同一テーブルに対する後の問い合わせにおいてその内容に明らかな矛盾は発生しませんが、削除されたテーブルと他のテーブルとの間の内容に目に見える矛盾が発生します。

テーブル内のデータという観点では、TRUNCATEはトランザクション的に安全です。 前後のトランザクションがコミットされなければ消去は安全にロールバックされます。

RESTART IDENTITYが指定された場合、暗黙的にALTER SEQUENCE RESTART操作がトランザクション的に安全に行われます。 つまりそれを囲むトランザクションがコミットされなければ、ロールバックされます。 これは通常のALTER SEQUENCE RESTARTとは異なる動作です。 トランザクションがロールバックされる前に、初期化したシーケンスに対してさらにシーケンス操作を行う場合には注意してください。 シーケンスに対するこれらの操作の影響はロールバックされますが、currval()への影響はロールバックされません。 つまりトランザクションの後、currval()は、シーケンス自体と値とが一貫性のない状態になっていたとしても、失敗したトランザクションの内側で得た最後のシーケンス値を継続して反映します。 これは、失敗したトランザクションの後のcurrval()の通常の動作と同じです。

例

bigtableテーブルおよびfattableテーブルを空にします。

```
TRUNCATE bigtable, fattable;
```

以下も同じですが、ここでは関連するシーケンスジェネレータをすべてリセットします。

```
TRUNCATE bigtable, fattable RESTART IDENTITY;
```

othertableテーブル、および、外部キー制約によりothertableを参照するすべてのテーブルを空にします。

```
TRUNCATE othertable CASCADE;
```

互換性

標準SQL:2008には、TRUNCATE TABLE tablenameという構文のTRUNCATEコマンドが含まれます。 CONTINUE IDENTITY/RESTART IDENTITY句も標準に記載され、関連してはいるのですが、若干異なります。 標準では、このコマンドの同時実行に関する動作の一部は実装に依存するものとされています。 このため、上記注意を検討し、必要に応じて他の実装と比べなければなりません。

UNLISTEN

名前

UNLISTEN -- 通知の監視を停止する

概要

```
UNLISTEN { channel | * }
```

説明

UNLISTENを使うと、既存のNOTIFYイベントの登録を削除することができます。UNLISTENは、現在のPostgreSQLセッションにある、*name*という名前の通知チャンネルのリスナ登録を全て取り消します。ワイルドカード*は特別な条件を表します。これを指定すると、現在のセッションにある全てのリスナ登録が取り消されます。

[NOTIFY](#)には、LISTENとNOTIFYについてのより広範な説明があります。

パラメータ

channel

通知チャンネルの名称です（任意の識別子）。

*

このセッションにおける、全ての監視登録をクリアします。

注釈

監視をまったく行っていない状態でもこのコマンドは実行できます。警告やエラーは表示されません。

セッション終了時に、自動的にUNLISTEN *が実行されます。

UNLISTENを実行したトランザクションは二相コミット用を準備することはできません。

例

登録を行います。

```
LISTEN virtual;  
NOTIFY virtual;  
Asynchronous notification "virtual" received from server process with PID 8448.
```

UNLISTENが実行されると、その後のNOTIFYメッセージは無視されます。

```
UNLISTEN virtual;  
NOTIFY virtual;  
-- NOTIFYイベントを受け取りません。
```

互換性

標準SQLにはUNLISTENコマンドはありません。

関連項目

UPDATE

名前

UPDATE -- テーブルの行を更新する

概要

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
UPDATE [ ONLY ] table_name [ * ] [ [ AS ] alias ]
SET { column_name = { expression | DEFAULT } |
    ( column_name [, ...] ) = ( { expression | DEFAULT } [, ...] ) } [, ...]
[ FROM from_list ]
[ WHERE condition | WHERE CURRENT OF cursor_name ]
[ RETURNING * | output_expression [ [ AS ] output_name ] [, ...] ]
```

説明

UPDATEは、条件を満たす全ての行の指定した列の値を変更します。 SET句には、変更する列のみを指定する必要があります。 SET句にて明示的に指定されなかった列の値は変更されません。

データベース内の他のテーブルの情報を使用してテーブルを変更するには、2つの方法があります。 1つは副問い合わせを使用する方法、もう1つはFROM句で追加するテーブルを指定する方法です。 どちらの方法が適切であるかは状況次第です。

省略可能なRETURNING句により、UPDATEは実際に更新された各行に基づいて計算された値を返すようになります。 そのテーブルの列、FROMで指定された他のテーブルの列、あるいは、その両方を使用した式を計算することができます。 テーブル列の新しい（更新された後の）値が使用されます。 RETURNINGリストの構文はSELECTの出力リストと同一です。

更新を行うためには、そのテーブルまたは少なくとも対象の列上にUPDATE権限を持たなければなりません。 また *expressions* や *condition* で読み込むテーブルに対するSELECT権限も必要になります。

パラメータ

with_query

WITH句によりUPDATE問い合わせ内で名前で参照可能な1つ以上の副問い合わせを指定することができます。 [項7.8](#)と[SELECT](#)を参照してください。

table_name

更新対象のテーブルの名前です（スキーマ修飾名でも可）。 テーブルの前にONLYを指定すると、指名されたテーブルのみで一致する行が更新されます。 ONLYを指定しないと、指名したテーブルから継承されたすべてのテーブルも一致する行が同時に更新されます。 省略することもできますが、テーブル名の後に*を指定して、明示的に子テーブルが含まれることを示すことができます。

alias

対象テーブルの代替名です。 別名が指定されると、テーブルの実際の名前は完全に隠蔽されます。 たとえば、UPDATE foo AS fでは、UPDATE文の残りの部分ではfooではなくfとしてこのテーブルを参照しなければなりません。

column_name

*table_name*で指名されたテーブル内の列名です。 必要に応じて、列名を副フィールド名や配列の指示子で修飾することも可能です。 対象列の指定にはテーブル名を含めないでください。 たとえば、UPDATE tab SET tab.col = 1は無効です。

expression

列に代入する式です。 この式では、テーブル内の対象列やその他の列の変更前の値を使用することができます。

DEFAULT

列にデフォルト値を設定します（デフォルト式が割り当てられていない場合はNULLになります）。

from_list

WHERE 条件や更新用の式において、他のテーブルの列を指定するために使用するテーブル式の集合です。これはSELECT文のFROM句で指定するテーブルのリストに似ています。自己結合を行う場合を除き、from_listに更新対象のテーブルを含める必要はありません（自己結合を行う場合は、from_list内で更新対象のテーブルとその別名を指定しておく必要があります）。

condition

boolean型の値を返す式です。この式がtrueを返す行のみが更新されます。

cursor_name

WHERE CURRENT OF 条件で使用されるカーソルの名前です。更新対象の行は、そのカーソルからもっとも最近に取り出された行です。カーソルはUPDATEの対象テーブルに対するグループ化のない問い合わせでなければなりません。WHERE CURRENT OFを論理条件といっしょに指定することはできません。WHERE CURRENT OF付きのカーソル使用に関する情報についてはDECLAREを参照してください。

output_expression

各行を更新した後に計算され、UPDATEによって返される式です。この式には、table_nameまたはFROMで指定したテーブル（複数可）の任意の列名を使用することができます。すべての列を返す場合は*と記載してください。

output_name

返される列で使用される名前です。

出力

正常に処理が終わると、UPDATE コマンドは以下の形式のコマンドタグを返します。

```
UPDATE count
```

countは、合致した行が変更されなかった行を含む、更新された行数を意味します。BEFORE UPDATE トリガにより更新が抑制された場合に、conditionに合致した行数より少なくなる可能性があることに注意してください。countが0の場合はconditionに一致する行がなかったことを意味します（これはエラーとはみなされません）。

UPDATE コマンドがRETURNING句を持つ場合、その結果は、RETURNING リストで定義した列と値を持ち、そのコマンドで更新された行全体に対して計算を行うSELECT文の結果と似たものになるでしょう。

注釈

FROM句が存在する場合、基本的に、対象テーブルとfrom_listで指定されたテーブルが結合され、この結合の出力行が対象テーブルの更新操作の結果となります。FROM句を使用する場合、更新対象テーブルの1行に対して、結合結果が複数行にならないように注意してください。言い換えると、対象テーブルの個々の行は、他テーブルの複数の行と結合すべきではありません。結合結果が複数行になった場合、対象行の更新には結合結果のいずれか1行のみが使用されますが、どの行が使用されるかは簡単には予測できません。

このような問題があるため、他テーブルの参照は副問い合わせ内のみに留めておいた方がより安全です（ただし、結合よりも可読性や実行速度は低下します）。

例

films テーブルのkind列にあるDrama という単語をDramaticに変更します。

```
UPDATE films SET kind = 'Dramatic' WHERE kind = 'Drama';
```

weather テーブルの特定の行に対し、気温という項目を調整し、降水量をデフォルト値に戻します。

```
UPDATE weather SET temp_lo = temp_lo+1, temp_hi = temp_lo+15, prcp = DEFAULT
WHERE city = 'San Francisco' AND date = '2003-07-03';
```

同じ操作を行い、更新された項目を返します。

```
UPDATE weather SET temp_lo = temp_lo+1, temp_hi = temp_lo+15, prcp = DEFAULT
WHERE city = 'San Francisco' AND date = '2003-07-03'
RETURNING temp_lo, temp_hi, prcp;
```

別の列リスト構文を使用して同様の更新を行います。

```
UPDATE weather SET (temp_lo, temp_hi, prcp) = (temp_lo+1, temp_lo+15, DEFAULT)
WHERE city = 'San Francisco' AND date = '2003-07-03';
```

FROM句構文を使用して、アクメ社を顧客とするセールスマンのセールスカウントを1増加させます。

```
UPDATE employees SET sales_count = sales_count + 1 FROM accounts
WHERE accounts.name = 'Acme Corporation'
AND employees.id = accounts.sales_person;
```

WHERE句で副問い合わせを使用して、同じ操作を行います。

```
UPDATE employees SET sales_count = sales_count + 1 WHERE id =
(SELECT sales_person FROM accounts WHERE name = 'Acme Corporation');
```

新しい商品とその在庫数を挿入します。既にその商品が存在している場合は、代わりに既存商品の在庫数を更新します。 トランザクション全体が失敗することがないようにこの操作を行うには、セーブポイントを使用してください。

```
BEGIN;
-- 何かしらの他の操作を行います。
SAVEPOINT spl;
INSERT INTO wines VALUES('Chateau Lafite 2003', '24');
-- 上記のコマンドが一意キー違反により失敗したとします。
-- この場合、次のコマンドを実行します。
ROLLBACK TO spl;
UPDATE wines SET stock = stock + 24 WHERE winename = 'Chateau Lafite 2003';
-- 他の操作を続けた後、最後に次を実行します。
COMMIT;
```

filmsテーブルにおいて、c_filmsカーソルが現在位置している行のkind列を変更します。

```
UPDATE films SET kind = 'Dramatic' WHERE CURRENT OF c_films;
```

互換性

このコマンドは標準SQLに準拠しています。ただしFROM句およびRETURNING句はPostgreSQLの拡張です。UPDATEでWITHが使用可能であることも同様に拡張です。

標準に従うと、列リスト構文は、副選択のような単一の行値式から代入される列のリストを許可しなければなりません。

```
UPDATE accounts SET (contact_last_name, contact_first_name) =
(SELECT last_name, first_name FROM salesmen
WHERE salesmen.id = accounts.sales_id);
```

これは現時点ではサポートされていません。ソースは独立した式でなければなりません。

他のデータベースシステムには、FROMオプション内で、対象テーブルが再度指定されることを前提として動作するものもあります。これはPostgreSQLにおけるFROMの解釈方法とは異なります。FROM拡張機能を使用するアプリケーションを移植する時は注意してください。

[前のページ](#)

UNLISTEN

[ホーム](#)

[上に戻る](#)

[次のページ](#)

VACUUM

VACUUM

名前

VACUUM -- データベースの不要領域の回収とデータベースの解析（オプション）を行う

概要

```

VACUUM [ ( { FULL | FREEZE | VERBOSE | ANALYZE } [, ...] ) ] [ table_name [ (column_name [, ...] ) ] ]
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] [ table_name ]
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] ANALYZE [ table_name [ (column_name [, ...] ) ] ]

```

説明

VACUUM は、不要タプルが使用する領域を回収します。 PostgreSQLの通常動作では、削除されたタプルや更新によって不要となったタプルは、テーブルから物理的には削除されません。 これらのタプルはVACUUMが完了するまで存在し続けます。 そのため、特に更新頻度が多いテーブルでは、VACUUMを定期的に行う必要があります。

パラメータの指定がない場合、VACUUMは現在のユーザがバキュームできる権限を持つ、現在のデータベース内の全てのテーブルを処理します。 パラメータを指定した場合、VACUUMは指定したテーブルのみを処理します。

VACUUM ANALYZEは、指定したテーブルの1つひとつに対し、VACUUMを行った後、ANALYZEを行います。 このコマンドの組み合わせは、日常的な管理スクリプトで使うと便利です。 処理の詳細に関しては、[ANALYZE](#)を参照してください。

（FULLが指定されていない）通常のVACUUMは、単に領域を回収し、そこを再利用可能な状態に変更します。 この形式のコマンドでは排他的ロックが取得されていないため、テーブルへの通常の読み書き操作と並行して実行することができます。 しかし余った領域はオペレーティングシステムには(ほとんどの場合)返されません。 同じテーブル内で再利用できるように保持されるだけです。 VACUUM FULLでは、テーブルの内容全体を新しいディスクファイルに書き換えるため領域が余ることはなく、オペレーティングシステムに未使用の領域を返すことができます。 この形式では、実行速度がかなり低速になります。 また、処理中のテーブルに対する排他的ロックが必要になります。

オプションリストが括弧でくくられていた場合、オプションを任意の順序で記述することができます。 括弧がないと、オプションは上で示した通りの順番で指定しなければなりません。 括弧付きの構文はPostgreSQL 9.0で追加されました。 カッコがない構文は廃止予定です。

パラメータ

FULL

より多くの領域の回収することができる"完全な"バキュームを選択します。 ただし、通常よりも処理に時間がかかります。 また、テーブルに対する排他ロックが必要です。

またこの方式では、テーブルのコピーを新しく書き出し、操作が終わるまで古いコピーが解放されませんので、余分にディスク領域が必要です、通常、大きな容量がテーブルから回収されなければなら内場合にのみこれが使用されるはずで

FREEZE

積極的にタプルの"凍結"を選択します。 FREEZE指定は、[vacuum_freeze_min_age](#)パラメータをゼロとしてVACUUMを実行することと同じです。

VERBOSE

各テーブルについてバキューム処理の詳細な報告を出力します。

ANALYZE

プランナが使用する統計情報を更新し、問い合わせを実行する最も効率的な方法を決定できるようにします。

table_name

バキューム対象のテーブル名です（スキーマ修飾名も可）。 デフォルトは現在のデータベース内の全テーブルです。

column_name

解析の対象とする列名です。デフォルトは全列です。 列リストが指定された場合はANALYZEを意味します。

出力

VERBOSEが指定された場合、VACUUMは、現在処理中のテーブルを示す進行状況メッセージを表示します。 同様に、テーブルについての各種の統計情報も表示されます。

注釈

テーブルをバキュームするためには、通常はテーブルの所有者もしくはスーパーユーザでなければなりません。しかしデータベースの所有者は共有カタログを除くデータベース内の全テーブルをバキュームすることができます。（共有カタログに関する制限は、データベース全体のVACUUMはスーパーユーザのみが実行可能であることを意味します。） VACUUMは、呼び出したユーザがバキュームするための権限を持たないテーブルはすべて飛ばします。

トランザクションブロック内でVACUUMを実行することはできません。

GINインデックスを持つテーブルでは、VACUUM（全構文）は待ち状態のインデックス挿入を、待ち状態のインデックス構文を主GINインデックス構造内の適切なところに移動させることにより、すべて完了させます。 [項55.3.1](#)を参照してください。

不要となった行を削除するため、実運用状態のデータベースに対しては（少なくとも毎晩）定期的にVACUUMを実行することを推奨します。 また、テーブルに対して多数の行を追加/削除した後は、そのテーブルにVACUUM ANALYZEを発行することを推奨します。 これによりシステムカタログに最近なされた全ての変更が反映されることになり、PostgreSQLの問い合わせプランナが、問い合わせ計画の作成時により良い選択をできるようになります。

FULLオプションを日常的に使用することは推奨しませんが、特殊なケースでは有用となる場合もあります。例えば、テーブル内のほとんど全ての行を削除または更新し、そのテーブルによるディスクの使用量を物理的に縮小させて高速なテーブルスキャンを行いたい場合です。 VACUUM FULLはたいいていの場合、通常のVACUUMよりもテーブルを縮小します。

VACUUMによりI/Oトラフィックがかなり増大しますので、実行中の他のセッションの性能が悪化する可能性があります。 このため、コストベースのバキューム遅延機能の使用を推奨する場合があります。 詳細は[項18.4.4](#)を参照してください。

PostgreSQLには、バキューム保守作業を自動化する"autovacuum"機能があります。 自動バキューム処理および手作業によるバキューム処理に関する詳細については、[項23.1](#)を参照してください。

例

下記の例は、regressionデータベース内のテーブルにVACUUMを実行したものです。

```
regression=# VACUUM (VERBOSE, ANALYZE) onek;
INFO:  vacuuming "public.onek"
INFO:  index "onek_unique1" now contains 1000 tuples in 14 pages
DETAIL:  3000 index tuples were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU 0.01s/0.08u sec elapsed 0.18 sec.
INFO:  index "onek_unique2" now contains 1000 tuples in 16 pages
DETAIL:  3000 index tuples were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU 0.00s/0.07u sec elapsed 0.23 sec.
INFO:  index "onek_hundred" now contains 1000 tuples in 13 pages
DETAIL:  3000 index tuples were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU 0.01s/0.08u sec elapsed 0.17 sec.
INFO:  index "onek_string1" now contains 1000 tuples in 48 pages
DETAIL:  3000 index tuples were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU 0.01s/0.09u sec elapsed 0.59 sec.
INFO:  "onek": removed 3000 tuples in 108 pages
DETAIL:  CPU 0.01s/0.06u sec elapsed 0.07 sec.
INFO:  "onek": found 3000 removable, 1000 nonremovable tuples in 143 pages
DETAIL:  0 dead tuples cannot be removed yet.
There were 0 unused item pointers.
0 pages are entirely empty.
CPU 0.07s/0.39u sec elapsed 1.56 sec.
INFO:  analyzing "public.onek"
INFO:  "onek": 36 pages, 1000 rows sampled, 1000 estimated total rows
VACUUM
```

互換性

標準SQLにはVACUUM文はありません。

関連項目

[vacuumdb](#), [項18.4.4](#), [項23.1.6](#)

前のページ UPDATE	ホーム 上に戻る	次のページ VALUES

VALUES

名前

VALUES -- 行セットを計算します

概要

```
VALUES ( expression [, ...] ) [, ...]
[ ORDER BY sort_expression [ ASC | DESC | USING operator ] [, ...] ]
[ LIMIT { count | ALL } ]
[ OFFSET start [ ROW | ROWS ] ]
[ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ]
```

説明

VALUES は、値の式で指定された行あるいは行の集合を計算します。大きなコマンドの中で"定数テーブル"を作成するために使用することが多いですが、それ単独で使用することも可能です。

複数行を指定した場合は、すべての行の要素数が同じでなければなりません。できあがるテーブル列のデータ型を決定するには、明示的に指定されている型やその列に登場する式から推測できる型を組み合わせで使用します。これは UNION と同じ方式です ([項10.5](#) を参照ください)。

大きなコマンドの中において、SELECT が文法上使える場所ならどこでもVALUESを使用することができます。なぜなら、これは文法上はSELECTと同じ扱いだからです。ORDER BY、LIMIT (、これと等価なFETCH FIRST) そしてOFFSET句でVALUESコマンドを使用することができます。

パラメータ

expression

定数あるいは式です。これを計算した結果が、表 (行セット) の中の指定した場所に挿入されます。VALUES リストを INSERT の最上位レベルで使用する場合は、*expression* を DEFAULT で置き換えることができます。これは、その列のデフォルト値を挿入することを表します。他の場所で VALUES を使用する場合には、DEFAULT は使用できません。

sort_expression

式あるいは整数の定数で、結果の行をソートする方法を表します。この式は、VALUES の結果の列を *column1*、*column2* などのように参照することができます。詳細は [ORDER BY句](#) を参照ください。

operator

ソート用の演算子です。詳細は [ORDER BY句](#) を参照ください。

count

返す行の最大数です。詳細は [LIMIT句](#) を参照ください。

start

結果を返す際に読み飛ばす行数です。詳細は [LIMIT句](#) を参照ください。

注釈

VALUES で大量の行を扱うことは避けるべきです。メモリ不足や性能の劣化を生じさせる可能性があります。VALUES を INSERT の中で使用する場合は特別です。(列の型は INSERT 先のテーブルからわかるので、VALUES のリストを調べて型を推測する必要がないからです) そのため、他の場面に比べて大きなリストを扱っても実用に耐えます。

例

必要最小限の `VALUES` コマンドはこのようになります。

```
VALUES (1, 'one'), (2, 'two'), (3, 'three');
```

これは、列が二つで行が三つの表を返します。事実上、これは次と同じことです。

```
SELECT 1 AS column1, 'one' AS column2
UNION ALL
SELECT 2, 'two'
UNION ALL
SELECT 3, 'three';
```

通常は、`VALUES` は大きな `SQL` コマンドの内部で使います。最もよくあるのは、`INSERT` での使用です。

```
INSERT INTO films (code, title, did, date_prod, kind)
VALUES ('T_601', 'Yojimbo', 106, '1961-06-16', 'Drama');
```

`INSERT` 内で使用する場合には、`VALUES` のリストに `DEFAULT` を指定することができます。これは、値を具体的に指定するのではなくその列のデフォルトを使用することを表します。

```
INSERT INTO films VALUES
('UA502', 'Bananas', 105, DEFAULT, 'Comedy', '82 minutes'),
('T_601', 'Yojimbo', 106, DEFAULT, 'Drama', DEFAULT);
```

`VALUES` は、副`SELECT` が書ける場所ならどこでも使用することができます。例えば `FROM` 句の中などでも使えます。

```
SELECT f.*
FROM films f, (VALUES('MGM', 'Horror'), ('UA', 'Sci-Fi')) AS t (studio, kind)
WHERE f.studio = t.studio AND f.kind = t.kind;

UPDATE employees SET salary = salary * v.increase
FROM (VALUES(1, 200000, 1.2), (2, 400000, 1.4)) AS v (depno, target, increase)
WHERE employees.depno = v.depno AND employees.sales >= v.target;
```

`VALUES` を `FROM` 句の中で使用する場合には、`AS` 句が必須となることに注意しましょう。これは `SELECT` の場合と同様です。`AS` 句ですべての列の名前を指定する必要はありませんが、指定しておくことをお勧めします (`VALUES` のデフォルトの列名は、PostgreSQL においては `column1`、`column2` のようになります。しかし、他のデータベースシステムでは異なるかもしれません)。

`VALUES` を `INSERT` の中で使用する場合は、値の型が挿入先の列のデータ型に自動変換されます。それ以外の場面で使用する際には、正しいデータ型を指定する必要があるかもしれません。値がすべてクォートされたリテラル定数である場合は、最初の値にだけ型を指定しておけば十分です。

```
SELECT * FROM machines
WHERE ip_address IN (VALUES('192.168.0.1 '::inet), ('192.168.0.10'), ('192.168.1.43'));
```

ティップ: 単に `IN` を試したいのなら、上のような `VALUES` クエリを使用するよりも `IN` のスカラリスト形式を使用するほうがよいでしょう。スカラメソッドのリストのほうが記述量が減りますし、たいていはより効率的になります。

互換性

`VALUES` は `SQL` 標準に従います。 `LIMIT` および `OFFSET` は PostgreSQL の拡張です。 [SELECT](#) も参照してください。

関連項目

[INSERT](#), [SELECT](#)

[前のページ](#)
VACUUM

[ホーム](#)
[上に戻る](#)

[次のページ](#)
PostgreSQLクライアントアプリケーション

II. PostgreSQLクライアントアプリケーション

ここでは、PostgreSQLクライアントアプリケーションとユーティリティについてのリファレンス情報があります。これらのコマンドの全てが汎用的なユーティリティではありません。一部は特定の権限を必要とします。これらアプリケーションの共通機能は、データベースサーバが稼働しているかどうかには依存しない、どのホストでも実行できるという点です。

コマンドラインから指定された場合、ユーザ名とデータベース名の英文字は保持されます。空白文字や特殊文字がある場合は引用符付けが必要かもしれません。テーブル名やその他の識別子では文書化されていない限り英文字は保持されませんので、引用符付けが必要かもしれません。

目次

- [clusterdb](#) -- PostgreSQLデータベースをクラスタ化する
- [createdb](#) -- 新しいPostgreSQLデータベースを作成する
- [createlang](#) -- PostgreSQL手続き言語をインストールする
- [createuser](#) -- 新しいPostgreSQLユーザアカウントを定義する
- [dropdb](#) -- PostgreSQLデータベースを削除する
- [droplang](#) -- PostgreSQL手続き言語を削除する
- [dropuser](#) -- PostgreSQLユーザアカウントを削除する
- [ecpg](#) -- 埋め込みSQL用Cプリプロセッサを使用する
- [pg_basebackup](#) -- PostgreSQLクラスタのベースバックアップを取得
- [pg_config](#) -- インストールしたPostgreSQLバージョンの情報を提供する
- [pg_dump](#) -- PostgreSQLデータベースをスクリプトファイルまたは他のアーカイブファイルへ抽出する
- [pg_dumpall](#) -- PostgreSQLデータベースクラスタをスクリプトファイルへ抽出する
- [pg_receivexlog](#) -- PostgreSQLクラスタからトランザクションログをストリームする
- [pg_restore](#) -- pg_dumpによって作成されたアーカイブファイルからPostgreSQLデータベースをリストアする
- [psql](#) -- PostgreSQL対話的ターミナル
- [reindexdb](#) -- PostgreSQLデータベースのインデックスを再作成します。
- [vacuumdb](#) -- PostgreSQLデータベースの不要領域の回収と解析を行う

clusterdb

名前

clusterdb -- PostgreSQL データベースをクラスタ化する

概要

```
clusterdb [connection-option...] [--verbose | -v] [ --table | -t table ] [dbname]
```

```
clusterdb [connection-option...] [--verbose | -v] --all | -a
```

説明

clusterdbは、PostgreSQLデータベース内のテーブルを再クラスタ化するユーティリティです。既にクラスタ化されているテーブルを検索し、前回と同じインデックスを使用して再度クラスタ化します。一度もクラスタ化されていないテーブルは処理されません。

clusterdbは、SQLコマンド [CLUSTER](#) のラップです。クラスタ化を行うのに、このユーティリティを使用しても、これ以外のサーバへのアクセス方法を使用しても、特別な違いはありません。

オプション

clusterdbでは、下記のコマンドライン引数を指定できます。

```
-a  
--all
```

全てのデータベースをクラスタ化します。

```
[-d] dbname  
[--dbname=] dbname
```

クラスタ化するデータベースの名前を指定します。データベース名が指定されておらず、`-a`（または`--all`）も使用されていない場合、データベース名は環境変数 `PGDATABASE` から読み取られます。この変数も設定されていない場合は、接続のために指定されたユーザ名が使用されます。

```
-e  
--echo
```

clusterdbが生成し、サーバに送るコマンドをエコー表示します。

```
-q  
--quiet
```

進行メッセージを表示しません。

```
-t table  
--table=table
```

*table*のみをクラスタ化します。

```
-v  
--verbose
```

処理の間、詳細な情報を出力します。

```
-V  
--version
```

clusterdbのバージョンを表示し、終了します。

```
-?  
--help
```

clusterdbのコマンドライン引数の使用方法を表示し、終了します。

clusterdbは、さらに、下記のコマンドライン引数を接続パラメータとして受け付けます。

```
-h host
--host=host
```

サーバが稼働しているマシンのホスト名を指定します。この値がスラッシュから始まる場合、Unixドメインソケット用のディレクトリとして使用されます。

```
-p port
--port=port
```

サーバが接続を監視するTCPポートもしくはUnixドメインソケットファイルの拡張子を指定します。

```
-U username
--username=username
```

接続するためのユーザ名です。

```
-w
--no-password
```

パスワードの入力を促しません。サーバがパスワード認証を必要とし、かつ、`.pgpass` ファイルなどの他の方法が利用できない場合、接続試行は失敗します。バッチジョブやパスワードを入力するユーザが存在しない場合にこのオプションは有用かもしれません。

```
-W
--password
```

データベースに接続する前に、clusterdbは強制的にパスワード入力を促します。

サーバがパスワード認証を要求する場合clusterdbは自動的にパスワード入力を促しますので、これが重要になることはありません。しかし、clusterdbは、サーバにパスワードが必要かどうかを判断するための接続試行を無駄に行います。こうした余計な接続試行を防ぐために`-w`の入力が有意となる場合もあります。

```
--maintenance-db=dbname
```

どのデータベースをクラスタ化しなければならないかを見つけ出すために接続するデータベースの名前を指定します。指定されなければ`postgres`データベースが使用され、もし存在しなければ`template1`が使用されます。

環境

```
PGDATABASE
PGHOST
PGPORT
PGUSER
```

デフォルトの接続パラメータです。

また、このユーティリティは、他のほとんどのPostgreSQLユーティリティと同様、libpqでサポートされる環境変数を使用します（[項31.14](#)を参照してください）。

診断

問題が発生した場合、考えられる原因とエラーメッセージについては[CLUSTER](#)と[psql](#)を参照してください。データベースサーバは対象ホスト上で稼働していなければなりません。また、libpqフロントエンドライブラリの、あらゆるデフォルトの設定や環境変数が適用されます。

例

`test` データベースをクラスタ化します。

```
$ clusterdb test
```

`xyzyy` データベース内のテーブルの1つ`foo`をクラスタ化します。

```
$ clusterdb --table foo xyzyy
```

関連項目

createdb

名前

createdb -- 新しいPostgreSQLデータベースを作成する

概要

```
createdb [connection-option...] [option...] [dbname [description]]
```

説明

createdbは、新しいPostgreSQLデータベースを作成します。

通常、このコマンドを実行したデータベースユーザが、新しいデータベースの所有者になります。ただし、コマンドを実行するユーザが適切な権限を持っている場合、-o オプションを使用して別のユーザを所有者に指定することができます。

createdbは[CREATE DATABASE](#)というSQLコマンドのラップです。したがって、このユーティリティでデータベースを作成しても、これ以外の方法でサーバにアクセスしてデータベースを作成しても何も違いはありません。

オプション

createdbでは、下記のコマンドライン引数を指定できます。

dbname

作成するデータベースの名前を指定します。この名前はクラスタ内の全てのPostgreSQLデータベースの中で一意でなければなりません。デフォルトでは、現在のシステムユーザと同じ名前でデータベースを作成します。

description

新しく作成されるデータベースに関連付けるコメントを指定します。

```
-D tablespace  
--tablespace=tablespace
```

データベース用のデフォルトのテーブル空間を指定します。（この名前は二重引用符で囲まれた識別子として処理されます。）

```
-e  
--echo
```

createdbが生成し、サーバに送信したコマンドをエコー表示します。

```
-E encoding  
--encoding=encoding
```

このデータベース内で使用する文字符号化方式を指定します。PostgreSQLサーバでサポートされる文字セットについては[項22.3.1](#)で説明します。

```
-l locale  
--locale=locale
```

このデータベースで使用するロケールを指定します。これは--lc-collateと--lc-ctypeの両方を指定することと等価です。

```
--lc-collate=locale
```

このデータベースで使用するLC_COLLATE設定を指定します。

```
--lc-ctype=locale
```

このデータベースで使用するLC_CTYPE設定を指定します。

```
-O owner
```

--owner=*owner*

新しいデータベースの所有者となるデータベースユーザを指定します。（この名前は二重引用符で囲まれた識別子として処理されます。）

-T *template*

--template=*template*

このデータベースの構築に使用するテンプレートデータベースを指定します。（この名前は二重引用符で囲まれた識別子として処理されます。）

-V

--version

createdbのバージョンを表示し、終了します。

-?

--help

createdbのコマンドライン引数の使用方法を表示し、終了します。

オプション-D、-l、-E、-O、および-Tは、基盤となる[CREATE DATABASE](#)というSQLコマンドのオプションにそれぞれ対応しています。詳細はそちらを参照してください。

またcreatedbは、以下のコマンドライン引数を接続パラメータとして受け付けます。

-h *host*

--host=*host*

サーバが稼働しているマシンのホスト名を指定します。この値がスラッシュから始まる場合、Unixドメインソケット用のディレクトリとして使用されます。

-p *port*

--port=*port*

サーバが接続を監視するTCPポートもしくはUnixドメインソケットのファイル拡張子を指定します。

-U *username*

--username=*username*

接続に使用するユーザ名を指定します。

-w

--no-password

パスワードの入力を促しません。サーバがパスワード認証を必要とし、かつ、.pgpassファイルなどの他の方法が利用できない場合、接続試行は失敗します。バッチジョブやパスワードを入力するユーザが存在しない場合にこのオプションは有用かもしれません。

-W

--password

データベースに接続する前に、createdbは強制的にパスワード入力を促します。

サーバがパスワード認証を要求する場合createdbは自動的にパスワード入力を促しますので、これが重要になることはありません。しかし、createdbは、サーバにパスワードが必要かどうかを判断するための接続試行を無駄に行います。こうした余計な接続試行を防ぐために-wの入力が有意となる場合もあります。

--maintenance-db=*dbname*

新しいデータベースを作成する時の接続先となるデータベースの名前を指定します。指定がなければ、postgresデータベースが使用されます。もし存在しなければ（またはこれが作成しようとしているデータベースの名前であれば）template1が使用されます。

環境

PGDATABASE

この値が設定されている場合、コマンドラインで上書きされない限り、設定された値が作成するデータベースの名前になります。

PGHOST

PGPORT

PGUSER

デフォルトの接続パラメータです。コマンドラインでもPGDATABASEでも名前が指定されていない場合は、PGUSERが作成す

るデータベースの名前にもなります。

また、このユーティリティは、他のほとんどのPostgreSQLユーティリティと同様、libpqでサポートされる環境変数を使用します（[項31.14](#)を参照してください）。

診断

問題が発生した場合、考えられる原因とエラーメッセージの説明について[CREATE DATABASE](#)と[psql](#)を参照してください。データベースサーバは対象ホストで稼働していなければなりません。また、libpqフロントエンドライブラリで使用される、全てのデフォルトの接続設定と環境変数が適用されることも覚えておいてください。

例

デフォルトのデータベースサーバを使用してdemoデータベースを作成します。

```
$ createdb demo
```

edenホスト上のポート番号5000のサーバを使用し、LATIN1符号化方式を使用するdemoデータベースを、背後で実行されるコマンドを表示させながら作成します。

```
$ createdb -p 5000 -h eden -E LATIN1 -e demo
CREATE DATABASE demo ENCODING 'LATIN1';
```

関連項目

[dropdb](#), [CREATE DATABASE](#)

[前のページ](#)
clusterdb

[ホーム](#)
[上に戻る](#)

[次のページ](#)
createlang

createlang

名前

createlang -- PostgreSQL 手続き言語をインストールする

概要

```
createlang [connection-option...] langname [dbname]
createlang [connection-option...] --list | -l [dbname]
```

説明

createlang は、PostgreSQL データベースに手続き言語を追加するためのユーティリティです。

createlang は単なる [CREATE LANGUAGE](#) SQL コマンドのラップです。

注意
createlang は廃止予定であり、将来の PostgreSQL リリースで削除される可能性があります。代わりに <code>CREATE EXTENSION</code> コマンドを直接使用することを推奨します。

オプション

createlang では、下記のコマンドライン引数を指定できます。

```
langname
    インストールする手続きプログラミング言語の名前を指定します。（この名前は小文字です。）

[-d] dbname
[--dbname=] dbname
    言語を追加するデータベースを指定します。デフォルトでは、現在のシステムユーザ名と同じ名前のデータベースが使用されます。

-e
--echo
    実行される SQL コマンドを表示します。

-l
--list
    対象データベース内に既にインストールされている言語の一覧を表示します。

-V
--version
    createlang のバージョンを表示し、終了します。

-?
--help
    createlang のコマンドライン引数の使用方法を表示し、終了します。

createlang は、以下のコマンドライン引数も接続パラメータとして受け付けます。

-h host
--host=host
    サーバが稼働しているマシンのホスト名を指定します。この値がスラッシュから始まる場合、Unix ドメインソケット用のディレクトリとして使用されます。
```

```
-p port
--port=port
```

サーバが接続を監視するTCPポートもしくはUnixドメインソケットファイルの拡張子を指定します。

```
-U username
--username=username
```

接続するためのユーザ名を指定します。

```
-w
--no-password
```

パスワードの入力を促しません。サーバがパスワード認証を必要とし、かつ、`.pgpass` ファイルなどの他の方法が利用できない場合、接続試行は失敗します。 バッチジョブやパスワードを入力するユーザが存在しない場合にこのオプションは有用かもしれません。

```
-W
--password
```

データベースに接続する前に、`createlang`は強制的にパスワード入力を促します。

サーバがパスワード認証を要求する場合`createlang`は自動的にパスワード入力を促しますので、これが重要になることはありません。しかし、`createlang`は、サーバにパスワードが必要かどうかを判断するための接続試行を無駄に行います。こうした余計な接続試行を防ぐために`-w`の入力が有意となる場合もあります。

環境

```
PGDATABASE
PGHOST
PGPORT
PGUSER
```

デフォルトの接続パラメータです。

また、このユーティリティは、他のほとんどのPostgreSQLユーティリティと同様、`libpq`でサポートされる環境変数を使用します ([項31.14](#)を参照してください)。

診断

ほとんどのエラーメッセージは、読めば理解できるものとなっています。 エラーメッセージを見ても問題が解決しない場合は、`createlang`を`--echo`オプションを付けて実行してください。表示結果の詳細については、それぞれ対応するSQLコマンドを参照してください。 `libpq`フロントエンドライブラリで 사용되는デフォルトの接続設定と環境変数も適用されます。

注釈

言語を削除するには [droplang](#) を使用してください。

例

`pltcl` 言語を `template1` データベースにインストールします。

```
$ createlang pltcl template1
```

言語を `template1` にインストールすることにより、後で作成されるデータベースにも自動的にインストールされることに注意してください。

関連項目

[droplang](#), [CREATE EXTENSION](#), [CREATE LANGUAGE](#)

[前のページ](#)
createdb

[ホーム](#)
[上に戻る](#)

[次のページ](#)
createuser

createuser

名前

createuser -- 新しいPostgreSQLユーザアカウントを定義する

概要

```
createuser [connection-option...] [option...] [username]
```

説明

createuserは新しいPostgreSQLユーザ（より正確に言えばロール）を作成します。新しいユーザを作成できるのは、スーパーユーザとCREATEROLE権限を持つユーザのみです。したがって、createuserは、スーパーユーザもしくはCREATEROLE権限を持つユーザとして接続可能なユーザによって実行されなければなりません。

新しいスーパーユーザを作成したいのであれば、スーパーユーザとして接続しなければなりません。CREATEROLEだけではいけません。スーパーユーザであるということは、そのデータベースにおけるアクセス権限の検査を素通りできることを意味しています。したがって、スーパーユーザという地位を簡単に与えてはなりません。

createuserはCREATE ROLE SQLコマンドのラップです。このユーティリティによってユーザを作成しても、これ以外の方法でサーバにアクセスしてユーザを作成しても特に違いはありません。

オプション

createuserでは、下記のコマンドライン引数を指定できます。

username

作成するPostgreSQLユーザの名前を指定します。この名前は、そのPostgreSQLインストレーションの中で一意でなければなりません。

```
-c number
--connection-limit=number
```

新しいユーザの最大接続数を設定します。デフォルトでは無制限です。

```
-d
--createdb
```

新しいユーザに対してデータベースの作成を許可します。

```
-D
--no-createdb
```

新しいユーザに対してデータベースの作成を禁止します。これはデフォルトです。

```
-e
--echo
```

createuserが生成しサーバに送信するコマンドを出力します。

```
-E
--encrypted
```

データベースに格納するユーザのパスワードを暗号化します。指定がない場合、デフォルトのパスワードの挙動が使用されます。

```
-i
--inherit
```

新しいロールは自動的にメンバとして属するロールの権限を継承します。これがデフォルトです。

```
-I
--no-inherit
```

新しいロールは自動的にメンバとして属するロールの権限を継承しません。

--interactive

ユーザ名がコマンドラインで指定されない場合、ユーザ名の入力を促し、更に `-d/-D`、`-r/-R`、`-s/-S` オプションがコマンドラインで指定されない場合にはどちらにするか入力を促します。（これはPostgreSQL 9.1までのデフォルトの動作でした。）

-l
--login

新しいユーザに対してログインを許可します。（つまり、このユーザ名をセッション起動時のユーザ識別子として使用することができます。）これがデフォルトです。

-L
--no-login

新しいユーザに対してログインを禁止します。（ログイン権限を持たないロールはデータベース権限管理という面で有意です。）

-N
--unencrypted

データベースに保存するユーザのパスワードを暗号化しません。このオプションが指定されていない場合は、デフォルトのパスワードの挙動が使用されます。

-P
--pwprompt

このオプションが指定されると、`createuser`は新しいユーザのパスワードのプロンプトを表示します。もしパスワード認証を使う予定がなければ、これは必要ありません。

-r
--createrole

新しいユーザに対して新しいロールの作成を許可します。（つまり、このユーザは`CREATEROLE`権限を持つことになります。）

-R
--no-createrole

新しいユーザに対して新しいロールの作成を禁止します。これはデフォルトです。

-s
--superuser

新しいユーザはスーパーユーザになります。

-S
--no-superuser

新しいユーザはスーパーユーザにはなりません。これはデフォルトです。

-V
--version

`createuser`のバージョンを表示し、終了します。

--replication

新しいユーザは`REPLICATION`権限を持ちます。この権限については[CREATE ROLE](#)の文書で多く説明します。

--no-replication

新しいユーザは`REPLICATION`権限を持ちません。この権限については[CREATE ROLE](#)の文書で多く説明します。

-?
--help

`createuser`のコマンドライン引数の使用方法を表示し、終了します。

`createuser`は、以下のコマンドライン引数も接続パラメータとして受け付けます。

-h *host*
--host=*host*

サーバが稼働しているマシンのホスト名を指定します。この値がスラッシュから始まる場合、Unixドメインソケット用のディレクトリとして使用されます。

-p *port*

```
--port=port
```

サーバが接続を監視するTCPポートもしくはUnixドメインソケットファイルの拡張子を指定します。

```
-U username  
--username=username
```

接続に使用するユーザ名です（作成するユーザの名前ではありません）。

```
-w  
--no-password
```

パスワードの入力を促しません。サーバがパスワード認証を必要とし、かつ、`.pgpass` ファイルなどの他の方法が利用できない場合、接続試行は失敗します。 バッチジョブやパスワードを入力するユーザが存在しない場合にこのオプションは有用かもしれません。

```
-W  
--password
```

`createuser`は強制的にパスワード入力を促します。（新しいユーザのパスワードではなく、サーバに接続するためのパスワードです）。

サーバがパスワード認証を要求する場合`createuser`は自動的にパスワード入力を促しますので、これが重要になることはありません。しかし、`createuser`は、サーバにパスワードが必要かどうかを判断するための接続試行を無駄に行います。こうした余計な接続試行を防ぐために`-w`の入力が有意となる場合もあります。

環境

```
PGHOST  
PGPORT  
PGUSER
```

デフォルトの接続パラメータです。

また、このユーティリティは、他のほとんどのPostgreSQLユーティリティと同様、`libpq`でサポートされる環境変数を使用します（[項31.14](#)を参照してください）。

診断

問題が発生した場合、考えられる原因とエラーメッセージの説明については、[CREATE ROLE](#)と[psql](#)を参照してください。データベースサーバは対象ホストで稼働していなければなりません。また、`libpq`フロントエンドライブラリで使用される、全ての接続設定と環境変数が適用されることを覚えておいてください。

例

デフォルトデータベースサーバ上に`joe`というユーザを作成します。

```
$ createuser joe
```

デフォルトデータベースサーバ上に`joe`というユーザを一部の属性入力促されるように作成します。

```
$ createuser --interactive joe  
Shall the new role be a superuser? (y/n) n  
Shall the new role be allowed to create databases? (y/n) n  
Shall the new role be allowed to create more new roles? (y/n) n
```

`eden` ホスト上のポート番号5000のサーバを使って上記と同じ`joe`というユーザを属性を明示的に指定して作成し、背後で実行される問い合わせを表示します。

```
$ createuser -h eden -p 5000 -S -D -R -e joe  
CREATE ROLE joe NOSUPERUSER NOCREATEDB NOCREATEROLE INHERIT LOGIN;
```

`joe`というユーザをスーパーユーザとして作成します。作成時にパスワードを割り当てます。

```
$ createuser -P -s -e joe  
Enter password for new role: xyzzy  
Enter it again: xyzzy  
CREATE ROLE joe PASSWORD 'md5b5f5ba1a423792b526f799ae4eb3d59e' SUPERUSER CREATEDB CREATEROLE INHERIT LOGIN;
```

上の例で、実際には入力した新しいパスワードは画面上に表示されませんが、分かりやすくするために記載しています。上記の通りこのパスワードはクライアントに送信される前に暗号化されます。 `--unencrypted` オプションが使用された場合、パスワード

ドはコマンドの出力に（おそらくはサーバのログなどに）含まれて表示されます。 ですので、もし誰かに画面を覗き見されるかもしれない場合、パスワードを割り当てる際には`-e`を使用しないことを勧めます。

関連項目

[dropuser](#), [CREATE ROLE](#)

[前のページ](#)
createlang

[ホーム](#)
[上に戻る](#)

[次のページ](#)
dropdb

dropdb

名前

dropdb -- PostgreSQLデータベースを削除する

概要

```
dropdb [connection-option...] [option...] dbname
```

説明

dropdbは既存のPostgreSQLデータベースを削除します。 このコマンドを実行できるのは、データベースのスーパーユーザまたはデータベースの所有者のみです。

dropdbは、[DROP DATABASE](#) SQLコマンドのラップです。 このユーティリティを使用しても、これ以外の方法でサーバにアクセスして削除しても、特に違いはありません。

オプション

dropdbは、下記のコマンドライン引数を受け付けます。

dbname

削除するデータベース名を指定します。

`-e`
`--echo`

dropdbが生成し、サーバに送信するコマンドをエコー表示します。

`-i`
`--interactive`

削除を行う前に、確認のためのプロンプトを表示します。

`-V`
`--version`

dropdbのバージョンを表示し、終了します。

`--if-exists`

指定したデータベースが存在しない場合でもエラーとしません。 この場合には注意が発生します。

`-?`
`--help`

dropdbのコマンドライン引数の使用方法を表示し、終了します。

またdropdbは、以下のコマンドライン引数を接続パラメータとして受け付けます。

`-h host`
`--host=host`

サーバが稼働しているマシンのホスト名を指定します。 この値がスラッシュから始まる場合、Unixドメインソケット用のディレクトリとして使用されます。

`-p port`
`--port=port`

サーバが接続を監視するTCPポートもしくはUnixドメインソケットファイルの拡張子を指定します。

`-U username`
`--username=username`

接続するユーザ名を指定します。

```
-w
--no-password
```

パスワードの入力を促しません。サーバがパスワード認証を必要とし、かつ、`.pgpass` ファイルなどの他の方法が利用できない場合、接続試行は失敗します。 バッチジョブやパスワードを入力するユーザが存在しない場合にこのオプションは有用かもしれません。

```
-W
--password
```

データベースに接続する前に、`dropdb`は強制的にパスワード入力を促します。

サーバがパスワード認証を要求する場合`dropdb`は自動的にパスワード入力を促しますので、これが重要になることはありません。 しかし、`dropdb`は、サーバにパスワードが必要かどうかを判断するための接続試行を無駄に行います。 こうした余計な接続試行を防ぐために`-w`の入力が有意となる場合もあります。

```
--maintenance-db=dbname
```

対象データベースを削除するために接続するデータベースの名前を指定します。 指定されない場合は、`postgres`データベースが使用されます。 このデータベースが存在しない場合（またはこのデータベースが削除対象のデータベースである場合）`template1`が使用されます。

環境

```
PGHOST
PGPORT
PGUSER
```

デフォルトの接続パラメータです。

また、このユーティリティは、他のほとんどのPostgreSQLユーティリティと同様、`libpq`でサポートされる環境変数を使用します（[項31.14](#)を参照してください）。

診断

問題が発生した場合、考えられる原因とエラーメッセージについては[DROP DATABASE](#)と[psql](#)を参照してください。 対象ホストでデータベースサーバが稼働していなければなりません。 また、`libpq`のフロントエンドライブラリの、あらゆるデフォルトの設定や環境変数が適用されることを覚えておってください。

例

次のコマンドは、デフォルトのデータベースサーバ上の`demo`データベースを削除します。

```
$ dropdb demo
```

次のコマンドはホスト名`eden`、ポート番号5000で動作しているサーバから`demo`データベースを削除します。 その際、削除の確認、およびバックエンドに送られるコマンドを表示します。

```
$ dropdb -p 5000 -h eden -i -e demo
Database "demo" will be permanently deleted.
Are you sure? (y/n) y
DROP DATABASE demo;
```

関連項目

[createdb](#), [DROP DATABASE](#)

[前のページ](#)
[createuser](#)

[ホーム](#)
[上に戻る](#)

[次のページ](#)
[droplang](#)

droplang

名前

droplang -- PostgreSQL手続き言語を削除する

概要

```
droplang [connection-option...] langname [dbname]
```

```
droplang [connection-option...] --list | -l [dbname]
```

説明

droplangは、PostgreSQLデータベースにあるプログラミング言語を削除するユーティリティです。

droplangは[DROP EXTENSION](#) SQLコマンドの単なるラッパです。

注意
droplangは廃止予定であり、将来のPostgreSQLリリースで削除される可能性があります。代わりにDROP EXTENSIONコマンドを直接使用することを推奨します。

オプション

droplangは、下記のコマンドライン引数を受け付けます。

langname

削除するプログラミング言語の名前を指定します。（名前は小文字です。）

[-d] dbname
[--dbname=] dbname

どのデータベースから言語を削除するのかを指定します。 デフォルトでは、現在のシステムユーザ名と同じ名前のデータベースが使用されます。

-e
--echo

実行されるSQLコマンドを表示します。

-l
--list

対象データベース内に既にインストールされている言語のリストを表示します。

-V
--version

droplangのバージョンを表示し、終了します。

-?
--help

droplangのコマンドライン引数の使用方法を表示し、終了します。

また、droplangでは接続引数として下記のコマンドライン引数を使用できます。

-h host
--host=host

サーバが稼働しているマシンのホスト名を指定します。 ホスト名がスラッシュから始まる場合、Unixドメインソケット用のディレクトリとして使用されます。

```
-p port
--port=port
```

サーバが接続を監視するインターネットTCP/IPポートもしくはUnixドメインソケットのファイル拡張子を指定します。

```
-U username
--username=username
```

接続するためのユーザ名を指定します。

```
-w
--no-password
```

パスワードの入力を促しません。サーバがパスワード認証を必要とし、かつ、`.pgpass` ファイルなどの他の方法が利用できない場合、接続試行は失敗します。 バッチジョブやパスワードを入力するユーザが存在しない場合にこのオプションは有用かもしれません。

```
-W
--password
```

データベースに接続する前に、`droplang`は強制的にパスワード入力を促します。

サーバがパスワード認証を要求する場合`droplang`は自動的にパスワード入力を促しますので、これが重要になることはありません。しかし、`droplang`は、サーバにパスワードが必要かどうかを判断するための接続試行を無駄に行います。こうした余計な接続試行を防ぐために`-w`の入力が有意となる場合もあります。

環境

```
PGDATABASE
PGHOST
PGPORT
PGUSER
```

デフォルトの接続パラメータです。

また、このユーティリティは、他のほとんどのPostgreSQLユーティリティと同様、`libpq`でサポートされる環境変数を使用します ([項31.14](#)を参照してください)。

診断

ほとんどのエラーメッセージは、読めば理解できるものとなっています。エラーメッセージを見ても問題が解決しない場合は、`--echo` オプションを付けて`droplang`を実行してください。表示結果の詳細については、それぞれ対応するSQLコマンドを参照してください。また、`libpq`フロントエンドライブラリで使用するデフォルトの接続設定や環境変数も適用されます。

注釈

言語の追加には[createlang](#)を使用してください。

例

`pltcl` 言語を削除するには下記のコマンドを実行してください。

```
$ droplang pltcl dbname
```

関連項目

[createlang](#), [DROP EXTENSION](#), [DROP LANGUAGE](#)

[前のページ](#)
[dropdb](#)

[ホーム](#)
[上に戻る](#)

[次のページ](#)
[dropuser](#)

dropuser

名前

dropuser -- PostgreSQLユーザアカウントを削除する

概要

dropuser [connection-option...] [option...] [username]

説明

dropuserは、存在するPostgreSQLユーザを削除します。 PostgreSQLユーザを削除することができるのは、スーパーユーザとCREATEROLE権限があるユーザのみです。

dropuserは[DROP ROLE](#) SQLコマンドのラッパです。 このユーティリティを使用してユーザを削除しても、この方法以外の方法でサーバでアクセスしてユーザを削除しても特に違いはありません。

オプション

dropuserは、下記のコマンドライン引数を受け付けます。

username

削除するPostgreSQLユーザ名を指定します。 コマンドラインで指定されず、かつ-i/--interactiveオプションが使用されている場合は、入力を促すプロンプトが表示されます。

-e
--echo

dropuserが生成し、サーバに送信するコマンドを表示します。

-i
--interactive

実際にユーザを削除する前に確認のプロンプトを表示させます。 コマンドラインにてユーザ名が指定されなかった場合にユーザ名の入力を促します。

-V
--version

dropuserのバージョンを表示し、終了します。

--if-exists

ユーザが存在しない場合にエラーを発生しません。 この場合は注意が発生します。

-?
--help

dropuserのコマンドライン引数の使用方法を表示し、終了します。

dropuserは以下のコマンドライン引数も接続パラメータとして受け付けます。

-h host
--host=host

サーバが稼働しているマシンのホスト名を指定します。 この値がスラッシュから始まる場合、Unixドメインソケット用のディレクトリとして使用されます。

-p port
--port=port

サーバが接続を監視するTCPポートもしくはUnixドメインソケットのファイル拡張子を指定します。

-U username

```
--username=username
```

接続に使用するユーザ名です（削除するユーザ名ではありません）。

```
-w  
--no-password
```

パスワードの入力を促しません。サーバがパスワード認証を必要とし、かつ、`.pgpass` ファイルなどの他の方法が利用できない場合、接続試行は失敗します。バッチジョブやパスワードを入力するユーザが存在しない場合にこのオプションは有用かもしれません。

```
-W  
--password
```

データベースに接続する前に、`dropuser`は強制的にパスワード入力を促します。

サーバがパスワード認証を要求する場合`dropuser`は自動的にパスワード入力を促しますので、これが重要になることはありません。しかし、`dropuser`は、サーバにパスワードが必要かどうかを判断するための接続試行を無駄に行います。こうした余計な接続試行を防ぐために`-w`の入力が有意となる場合もあります。

環境

```
PGHOST  
PGPORT  
PGUSER
```

デフォルトの接続パラメータです。

また、このユーティリティは、他のほとんどのPostgreSQLユーティリティと同様、`libpq`でサポートされる環境変数を使用します（[項31.14](#)を参照してください）。

診断

問題が発生した場合、考えられる原因とエラーメッセージについては[DROP ROLE](#)と[psql](#)を参照してください。データベースサーバは対象ホスト上で稼働していなければなりません。また、`libpq`フロントエンドライブラリで使用する、あらゆるデフォルトの設定や環境変数が適用されることを覚えておいてください。

例

デフォルトのデータベースサーバから、`joe`というユーザを削除します。

```
$ dropuser joe
```

ホスト`eden`でポート5000を使用しているサーバから、`joe`というユーザを削除します。オプションによって、背後で実行されるコマンドの表示と、削除前の確認がなされています。

```
$ dropuser -p 5000 -h eden -i -e joe  
Role "joe" will be permanently removed.  
Are you sure? (y/n) y  
DROP ROLE joe;
```

関連項目

[createuser](#), [DROP ROLE](#)

[前のページ](#)
[droplang](#)

[ホーム](#)
[上に戻る](#)

[次のページ](#)
[ecpg](#)

ecpg

名前

ecpg -- 埋め込みSQL用Cプリプロセッサを使用する

概要

ecpg [*option...*] *file...*

説明

ecpg は、C プログラム用の埋め込みSQLプリプロセッサです。SQL 呼び出しを特殊な関数呼び出しに置き換えることによって、埋め込みSQL文を含むCプログラムを、通常のCコードに変換します。これにより、出力ファイルは、どのようなCコンパイラツールを使用しても処理することができます。

ecpg は、コマンドラインで指定される各入力ファイルを対応するCの出力ファイルに変換します。入力ファイルに .pgc という拡張子を付けておくと、出力ファイル名の拡張子が .c に置き換えられるので便利です。入力ファイルの拡張子が .pgc でない場合、そのファイルのフルネームの末尾に .c を追加したものが出力ファイル名となります。出力ファイル名は、-o オプションによって上書きすることもできます。

このマニュアルページでは埋め込みSQL言語については説明しません。 [第33章](#)を参照してください。

オプション

ecpg は、以下のコマンドライン引数を受け付けます。

-c

SQLコードから有効なCコードを自動的に生成します。現在、このオプションはEXEC SQL TYPE に対して使用できます。

-C *mode*

互換モードを設定します。*mode* は INFORMIX もしくは INFORMIX_SE のどちらかを取ることができます。

-D *symbol*

C プリプロセッサのシンボルを定義します。

-i

同様にシステムインクルードファイルも解析します。

-I *directory*

追加のインクルード用パスを指定します。これは、EXEC SQL INCLUDE を使用してインクルードされるファイルを検索する際に使用されます。デフォルトでは順に、. (現行ディレクトリ)、/usr/local/include、コンパイル時に定義されるPostgreSQLのインクルードディレクトリ (デフォルトでは/usr/local/pgsql/include)、/usr/include です。

-o *filename*

ecpg が全ての出力を *filename* に書き込むことを指定します。

-r *option*

実行時の動作を選択します。以下のいずれかを *option* として取ることができます。

no_indicator

指示子を使用せずにNULL値を表す特殊な値を使用します。歴史的にこの方式を使用したデータベースが存在します。

prepare

すべての文を使用する前に準備します。 libecpgは準備済みの文のキャッシュを保持し、再実行される場合に文を再利用します。 キャッシュが満杯になった場合、libecpgは最も古い文を解放します。

questionmarks

互換性という理由でクエションマークをプレースホルダとして許します。 これは長い間デフォルトとして使用されていました。

-t

トランザクションの自動コミットを有効にします。 このモードでは、各SQLコマンドは明示的なトランザクションブロックの内部にない限り、自動的にコミットされます。 デフォルトのモードでは、EXEC SQL COMMITが発行された時にのみコマンドがコミットされます。

-v

バージョンやインクルード用パスなどの補足情報を表示します。

--version

ecpgのバージョンを表示し、終了します。

-?

--help

ecpgのコマンドライン引数の使用方法を表示し、終了します。

注釈

前処理されたCコードファイルをコンパイルする際、コンパイラはPostgreSQLのインクルードディレクトリ内にあるECPGヘッダファイルを検索できるようにしなければなりません。 そのため、コンパイラの呼び出し時に、`-I`オプションを使用しなければならない可能性があります（例：`-I/usr/local/pgsql/include`）。

SQLが埋め込まれたCプログラムには、`-L/usr/local/pgsql/lib -lecp`g リンカオプションを使用するなどして、libecpg ライブラリをリンクする必要があります。

使用するシステムにおいて上記の2つに対応するディレクトリを調べるには、[pg_config](#)を使用します。

例

埋め込みSQLを使用したprog1.pgcというCソースファイルがある場合、次のコマンドを順番に実行すれば、実行可能プログラムを作成することができます。

```
ecpg prog1.pgc
cc -I/usr/local/pgsql/include -c prog1.c
cc -o prog1 prog1.o -L/usr/local/pgsql/lib -lecp
```

[前のページ](#)
dropuser

[ホーム](#)
[上に戻る](#)

[次のページ](#)
pg_basebackup

pg_basebackup

名前

pg_basebackup -- PostgreSQL クラスタのベースバックアップを取得

概要

pg_basebackup [option...]

説明

pg_basebackupは、稼働中のPostgreSQLデータベースクラスタのベースバックアップを取るために使用されます。データベースへの他のクライアントに影響することなく、バックアップが取られます。またこのバックアップはポイントインタイムリカバリ ([項24.3参照](#)) とログシッピングやストリーミングレプリケーションスタンバイサーバ用の開始点 ([項25.2参照](#)) としても使用することができます。

pg_basebackupは、システムは自動的にバックアップモードにし、自動的にバックアップモードから戻ることを実行し、データベースクラスタファイルのバイナリコピーを作成します。バックアップは常にデータベースクラスタ全体のバックアップを取ります。個々のバックアップや個々のデータベースオブジェクトをバックアップすることはできません。個々のデータベースバックアップに関しては[pg_dump](#)などのツールを使用しなければなりません。

バックアップは通常のPostgreSQL接続を経由して、レプリケーションプロトコルを用いて、作成されます。この接続はスーパーユーザまたはREPLICATION権限 ([項20.2参照](#)) を持つユーザが確立しなければなりません。さらにレプリケーション用の接続には[pg_hba.conf](#)における明示的な権限が許されていなければなりません。またサーバで[max_wal_senders](#)を、バックアップ用に少なくとも1つのセッションを残すように十分高く設定する必要があります。

同時にpg_basebackupを複数実行することができます。しかし性能という観点からは、1つのバックアップのみを取り結果をコピーする方が優れています。

pg_basebackupは、マスタからだけではなくスタンバイからもベースバックアップを作成することができます。スタンバイからバックアップを取得するためには、レプリケーション接続を受け付けられるようにスタンバイを設定してください (つまり[max_wal_senders](#)と[hot_standby](#)を設定し、[host-based authentication](#)を設定してください)。またマスタで[full_page_writes](#)を有効にする必要があります。

スタンバイからのオンラインバックアップではいくつかの制限があることに注意してください。

- バックアップ履歴ファイルはバックアップされるデータベースクラスタ内に作成されません。
- バックアップの最後で、バックアップに必要なすべてのWALファイルが保管されること保証はありません。アーカイブリカバリ用にこのバックアップを使用する予定である場合、および、その時点で必要なファイルすべてが利用できることを確実にしたい場合、`-x`オプションを使用してバックアップに含めなければなりません。
- オンラインバックアップ中にスタンバイがマスタに昇格した場合、バックアップは失敗します。
- バックアップに必要なすべてのWALレコードは、必要なだけの完全ページ書き出しを含んでいなければなりません。つまりこれは、マスタで[full_page_writes](#)を有効にし、`archive_command`としてWALファイルから完全ページ書き出しを取り除く[pg_compresslog](#)のようなツールを使用しないことが要求されます。

オプション

以下のコマンドラインオプションは出力の場所と書式を制御します。

```
-D directory
--pgdata=directory
```

出力を書き出すディレクトリです。pg_basebackupは必要ならば、このディレクトリとその親ディレクトリすべてを作成します。ディレクトリはすでに存在してもかまいませんが、存在しかつ空でない場合はエラーになります。

バックアップがtarモードであり、かつ、指定したディレクトリが- (ダッシュ) の場合、tarファイルは[stdout](#)に書き出さ

れます。

このオプションは必須です。

```
-F format
--format=format
```

出力書式を選択します。 *format*には以下のいずれかを取ることができます。

```
p
plain
```

普通のファイルとして、現在のデータディレクトリとテーブル空間と同じレイアウトで、出力を書き出します。 クラスタがテーブル空間を追加で持たない場合、データベース全体が指定したディレクトリに格納されます。 クラスタが追加のテーブル空間を持つ場合は、主データディレクトリは指定したディレクトリ内に格納されますが、他のテーブル空間はすべて、サーバ上の同じ絶対パスに格納されます。

これがデフォルトの書式です。

```
t
tar
```

指定したディレクトリ内にtarファイルとして出力を書き出します。 主データディレクトリは`base.tar`という名前のファイルに書き出され、他のテーブル空間はすべてテーブル空間のOIDに因んだ名前のファイルに書き出されます。

対象ディレクトリとして-（ダッシュ）という値が指定された場合、tarの内容は標準出力に書き出されます。 これは例えばgzipへのパイプ処理に適しています。 これはクラスタが追加テーブル空間を持たない場合のみ行うことができます。

```
-x
--xlog
```

このオプションの使用は、`fetch`方式の`-x`の使用と同一です。

```
-X method
--xlog-method=method
```

必要なトランザクションログファイル（WALファイル）をバックアップに含めます。 これはバックアップ中に生成されたトランザクションログをすべて含めます。 このオプションが指定された場合、ログアーカイブを考慮することなく展開したディレクトリ内でそのままpostmasterを起動することができます。 つまりこれは完全なスタンドアローンバックアップを作成します。

以下のトランザクションログを収集するための方法がサポートされます。

```
f
fetch
```

トランザクションログファイルはバックアップの最後に収集されます。 したがって、[wal_keep_segments](#)パラメータを、バックアップの最後までログが削除されない程度に大きくさせる必要があります。 ログの転送時点でそのログがローテートされていた場合、バックアップは失敗し、使用することができません。

```
s
stream
```

バックアップを作成する時にトランザクションログをストリームします。 これは第二のサーバ接続を開き、バックアップを実行している間、並行してトランザクションログのストリーミングを始めます。 したがって、これは[max_wal_senders](#)で設定されるスロットを最大2つ使用します。 クライアントが受け取ったトランザクションログに追従している限り、このモードを使用すれば、マスタ上に余分に保管されるトランザクションログは必要ありません。

```
-z
--gzip
```

tarファイル出力のデフォルトの圧縮レベルによるgzip圧縮を有効にします。 tarファイルを生成する場合のみ圧縮を利用することができます。

```
-Z level
--compress=level
```

tarファイル出力のgzip圧縮を有効にします。 また圧縮レベル（1から9まで。9が最高の圧縮レベル）を指定します。 tarファイルを生成する場合のみ圧縮を利用することができます。

以下のコマンドラインオプションはバックアップの生成とこのプログラムの実行を制御します。

```
-c fast|spread
```

```
--checkpoint=fast|spread
```

チェックポイントをfastまたはspread（デフォルト）に設定します。

```
-l label  
--label=label
```

バックアップのラベルを設定します。何も指定がない場合、"pg_basebackup base backup"というデフォルト値が使用されます。

```
-P  
--progress
```

進行状況報告を有効にします。これを有効にすると、バックアップ中におおよその進行状況が報告されます。データベースはバックアップ中に変更があるかもしれませんが、これはおおよそでしかなく100%正確に終わらないかもしれません。実際、WALログがバックアップに含まれる場合、データ総量は前もって予測することはできません。このためこの場合、推定対象容量はWALなしの総推定量を過ぎた後増加します。

これが有効な場合、バックアップはまずデータベース全体容量を計算し、その後実際の内容に戻り、送信します。バックアップに要する時間はかなり長くなるかもしれません。具体的にはデータが送られるようになるまでの時間がより長くなります。

```
-v  
--verbose
```

冗長モードを有効にします。開始時および終了段階でいくつか追加の段階が出力されます。また進行状況報告も有効な場合、現在処理中のファイル名も正しく出力されます。

以下のオプションはデータベース接続パラメータを制御します。

```
-h host  
--host=host
```

サーバが稼働しているマシンのホスト名を指定します。この値がスラッシュから始まる場合、Unixドメインソケット用のディレクトリとして使用されます。デフォルトは、設定されていればPGHOST環境変数から取得されます。設定されていなければ、Unixドメインソケット接続と仮定されます。

```
-p port  
--port=port
```

サーバが接続を監視するTCPポートもしくはローカルUnixドメインソケットファイルの拡張子を指定します。デフォルトは、設定されている場合、PGPORT環境変数の値となります。設定されていなければ、コンパイル時のデフォルト値となります。

```
-s interval  
--status-interval=interval
```

サーバに返送する状態パケットの間隔を秒単位で指定します。これは、サーバでレプリケーションタイムアウトが設定されている場合にトランザクションログを(--xlog=streamを用いて)ストリーミングする際に必要であり、監視をより簡単に行うことができます。ゼロという値は状態更新を完全に無効にします。デフォルト値は10秒です。

```
-U username  
--username=username
```

接続ユーザ名です。

```
-w  
--no-password
```

パスワードの入力を促しません。サーバがパスワード認証を必要とし、かつ、.pgpassファイルなどの他の方法が利用できない場合、接続試行は失敗します。バッチジョブやパスワードを入力するユーザが存在しない場合にこのオプションは有用かもしれません。

```
-W  
--password
```

データベースに接続する前に、pg_basebackupは強制的にパスワード入力を促します。

サーバがパスワード認証を要求する場合pg_basebackupは自動的にパスワード入力を促しますので、これが重要になることはありません。しかし、pg_basebackupは、サーバにパスワードが必要かどうかを判断するための接続試行を無駄に行います。こうした余計な接続試行を防ぐために-wの入力が有意となる場合もあります。

以下のその他のオプションも使用することができます。

```
-V  
--version
```

pg_basebackupのバージョンを表示し終了します。

```
-?  
--help
```

pg_basebackupコマンドライン引数の使用方法を表示し、終了します。

環境

他のほとんどのPostgreSQLユーティリティと同様このユーティリティはlibpqでサポートされる環境変数（[項31.14](#)参照）を使用します。

注意

このバックアップには、設定ファイルとサードパーティによりディレクトリに格納された追加ファイルを含む、データディレクトリとテーブル空間内のすべてのファイルが含まれます。 データディレクトリ内には通常のファイルとディレクトリのみが許され、シンボリックリンクや特殊デバイスファイルは許されません。

PostgreSQLがテーブル空間を管理する方式のため、すべての追加テーブル空間のパスはバックアップをリストアした時に常に同一でなければなりません。 しかし主データディレクトリは他の場所に設置することができます。

例

mydbserverで移動するサーバのベースバックアップを作成し、ローカルディレクトリに保管します。 /usr/local/pgsql/data:

```
$ pg_basebackup -h mydbserver -D /usr/local/pgsql/data
```

各テーブル空間につき圧縮したtarファイルを1つ作成するようにローカルサーバをバックアップし、backupディレクトリに保管します。 同時に実行時に進行状況を表示します。

```
$ pg_basebackup -D backup -Ft -z -P
```

単一のテーブル空間を持つローカルデータベースのバックアップを作成し、それをbzip2で圧縮します。

```
$ pg_basebackup -D - -Ft | bzip2 > backup.tar.bz2
```

（データベース内に複数のテーブル空間が存在する場合このコマンドは失敗します。）

関連項目

[pg_dump](#)

[前のページ](#)
ecpg

[ホーム](#)
[上に戻る](#)

[次のページ](#)
pg_config

pg_config

名前

pg_config -- インストールしたPostgreSQLバージョンの情報を提供する

概要

pg_config [*option...*]

説明

pg_configユーティリティは、現在インストールしているバージョンのPostgreSQLの設定パラメータを表示します。これは、例えばPostgreSQLとのインタフェースを持つソフトウェアパッケージが必要なヘッダファイルやライブラリを容易に検出できるように用意されたものです。

オプション

pg_configを使用するためには、以下のオプションを1つ以上指定します。

- bindir
ユーザコマンドの場所を表示します。例えば、psqlプログラムを検索するために使用します。これは通常、pg_configプログラムが存在するディレクトリでもあります。
- docdir
文書ファイルの場所を表示します
- htmldir
HTML文書ファイルの場所を表示します。
- includedir
クライアントインタフェースのCヘッダファイルの場所を表示します。
- pkgincludedir
その他のCヘッダファイルの場所を表示します。
- includedir-server
サーバプログラム作成用のCヘッダファイルの場所を表示します。
- libdir
オブジェクトコードライブラリのディレクトリを表示します。
- pkglibdir
動的ローディング可能なモジュールの場所（この場所をサーバが検索します。このディレクトリには、アーキテクチャに依存する他のデータファイルも存在する可能性があります）。
- localedir
ロケールサポートファイルの場所を表示します（PostgreSQLをロケールサポートなしで構築した場合は空文字列となります）。
- mandir
マニュアルページの場所を表示します。
- sharedir

アーキテクチャ非依存のサポートファイルの場所を表示します。

--sysconfdir

システム全体の設定ファイルの場所を表示します。

--pgxs

拡張用Makefileの場所を表示します。

--configure

PostgreSQLを構築する時にconfigureスクリプトに与えたオプションを表示します。まったく同じ構築条件でPostgreSQLを再作成する時、あるいは、バイナリパッケージの構築時のオプションを知りたい時に有益です（バイナリパッケージには、ベンダ特有のカスタムパッチが含まれていることが多いので注意してください）。後述の例も参照してください。

--cc

PostgreSQLの構築時に使用されたcc変数の値を表示します。使用したCコンパイラが表示されます。

--cppflags

PostgreSQLの構築時に使用されたCPPFLAGS変数の値を表示します。事前処理時に必要としたCコンパイラのスイッチが表示されます。（通常は-Iスイッチです。）

--cflags

PostgreSQLの構築時に使用されたCFLAGS変数の値を表示します。Cコンパイラスイッチが表示されます。

--cflags_sl

PostgreSQLの構築時に使用されたCFLAGS_SL変数の値を表示します。共有ライブラリの構築に使用されたCコンパイラスイッチが表示されます。

--ldflags

PostgreSQLの構築時に使用されたLDFLAGS変数の値を表示します。リンクスイッチが表示されます。

--ldflags_ex

PostgreSQLの構築時に使用されたLDFLAGS_EX変数の値を表示します。実行ファイルの構築のみに使用されたリンクスイッチが表示されます。

--ldflags_sl

PostgreSQLの構築時に使用されたLDFLAGS_SL変数の値を表示します。共有ライブラリの構築のみに使用されたリンクスイッチが表示されます。

--libs

PostgreSQLの構築時に使用されたLIBS変数の値を表示します。これには通常、PostgreSQLにリンクする外部ライブラリ用の-lスイッチが含まれます。

--version

PostgreSQLのバージョンを表示します。

-?
--help

pg_configコマンドライン引数に関する説明を表示し、終了します。

1つ以上のオプションが与えられた場合、指定したオプションの順番に従って1行ずつ情報を表示します。オプションがない場合、すべての利用可能な情報をラベル付きで表示します。

注釈

--includedir-server オプションはPostgreSQL 7.2で新しく導入されました。それまでのリリースでは、サーバ向けのインクルードファイルは、--includedir オプションで表示されるクライアント向けのヘッダと同じ場所にありました。どちらの場合でも対応できるようなパッケージにするには、まず新しいオプションを実行して、その終了ステータス进行检查して成功したかどうかを確認してください。

オプション--docdir、--pkgincludedir、--localedir、--mandir、--sharedir、--sysconfdir、--cc、--cppflags、--

cflags、--cflags_sl、--ldflags、--ldflags_sl、--libs はPostgreSQL 8.1から追加されました。 オプション--
htmldirはPostgreSQL 8.4で追加されました。 オプション--ldflags_exはPostgreSQL 9.0で追加されました。

pg_configが存在しない、PostgreSQL 7.1より前のバージョンでは、同様の設定情報を検索する方法はありませんでした。

例

使用中のPostgreSQLインストレーションの構築時の設定を再生成するには、以下のコマンドを実行します。

```
eval `./configure `pg_config --configure`
```

pg_config --configureの出力にはシェルの引用符が含まれますので、空白を含む引数も正しく表現することができます。 したがって、正しく動作させるためにはevalが必要です。

[前のページ](#)

pg_basebackup

[ホーム](#)
[上に戻る](#)

[次のページ](#)

pg_dump

pg_dump

名前

pg_dump -- PostgreSQLデータベースをスクリプトファイルまたは他のアーカイブファイルへ抽出する

概要

```
pg_dump [connection-option...] [option...] [dbname]
```

説明

pg_dumpはPostgreSQLデータベースをバックアップするユーティリティです。 データベースを使用中であっても一貫性のあるバックアップを作成することができます。 pg_dumpは他のユーザによるデータベースへのアクセス（読み書き）をブロックしません。

ダンプはスクリプト形式、または、アーカイブファイル形式で出力することができます。 スクリプトダンプは、保存した時点の状態のデータベースを再構成するために必要なSQLコマンドが書き込まれた平文ファイルです。 このスクリプトを使ってリストアを行うには`psql`を使用します。 スクリプトファイルを使えば、ダンプを行ったのとは別のマシンや別のアーキテクチャ上でも、データベースを再構築することができます。 また、多少編集すれば他のSQLデータベース製品上でもデータベースの再構築が可能です。

もう1つの形式であるアーカイブファイル形式を使ってデータベースを再構築するには、`pg_restore`を使用しなければなりません。 このファイルを使用すると、pg_restoreがリストア対象を選択したり、リストアするアイテムを並び替えたりできます。 アーカイブファイルもまた、アーキテクチャを越えて移植できるように設計されています。

いずれかのアーカイブファイル形式をpg_restoreと組み合わせて使用する場合は、pg_dumpの柔軟なアーカイブ/転送機構が利用できます。 具体的には、pg_dumpを使用してデータベース全体をバックアップし、pg_restoreを使用して、アーカイブの内容を検査したり、データベースの一部を選択してリストアしたりすることができます。 最も柔軟な出力ファイル形式は"custom"形式（`-Fc`）です。 この形式ではデフォルトで圧縮されます。 また、保存された項目の選択や並び換えを行うことができます。

pg_dumpの実行中は、標準エラーに出力される警告（特に後述の制限に関する警告）が出力されていないか確認してください。

オプション

以下のコマンドラインオプションは出力形式とその内容を制御します。

`dbname`

ダンプするデータベースの名前を指定します。 指定されていない場合は`PGDATABASE`環境変数が使われます。 この変数も設定されていない場合は、接続のために指定されたユーザ名が使用されます。

`-a`
`--data-only`

データのみをダンプし、スキーマ（データ定義）はダンプしません。 テーブルデータ、ラージオブジェクト、シーケンス値がダンプされます。

このオプションは`--section=data`を指定することと似ていますが、歴史的な理由で同一ではありません。

`-b`
`--blobs`

ラージオブジェクトをダンプに含めます。 これはデフォルトの動作ですが、`--schema`、`--table`、`--schema-only`が指定された場合は例外です。 したがって、`-b`オプションは選択的なダンプにラージオブジェクトを追加する場合にのみ有効です。

`-c`
`--clean`

データベースオブジェクトを作成するコマンドの前に、データベースオブジェクトを整理（削除）するコマンドを書き出します。（対象のデータベースの中にオブジェクトがまったくない場合でも、リストア時に害がないエラーがいくつか発生するかもしれません。）

このオプションは平文形式の場合にのみ有効です。アーカイブ形式では、`pg_restore`を呼び出す時にこのオプションを指定することができます。

```
-C
--create
```

初めにデータベース自体を作成するコマンドを出力し、その後、作成したデータベースに接続するコマンドを出力します（このようなスクリプトを使用すると、スクリプトを実行する前に対象のインストレーションの中のどのデータベースに接続すればよいかという問題を考える必要がなくなります）。`--clean`も同時に指定されている場合、このスクリプトは接続する前に対象データベースを削除し再作成します。

このオプションは平文形式の場合にのみ有効です。アーカイブ形式では、`pg_restore`を呼び出す時にこのオプションを指定することができます。

```
-E encoding
--encoding=encoding
```

指定した文字セット符号化方式でダンプを作成します。デフォルトではダンプはデータベースの符号化方式で作成されます。（`PGCLIENTENCODING`環境変数を好みのダンプ時の符号化方式に設定することで、同じ結果を得ることができます。）

```
-f file
--file=file
```

出力を指定のファイルに送ります。ファイルを基にする出力形式ではこのパラメータは省くことができます。省略時は標準出力が使用されます。しかしディレクトリ出力形式の場合は、省略することはできず、ファイルではなく対象ディレクトリを指定します。この場合、ディレクトリは`pg_dump`が生成しますので、事前に存在してはなりません。

```
-F format
--format=format
```

出力形式を選択します。`format`には以下のいずれかを取ることができます。

```
p
plain
```

平文のSQLスクリプトファイルを出力します（デフォルト）。

```
c
custom
```

`pg_restore`への入力に適したカスタム形式アーカイブを出力します。ディレクトリ出力形式と一緒に使用する場合、リストア時に手作業で保管された項目の選択、再順序付けができますので、これはもっとも柔軟な出力形式です。また、この形式はデフォルトで圧縮されます。

```
d
directory
```

`pg_restore`への入力に適したディレクトリ形式のアーカイブを出力します。これは、ダンプされる各テーブルおよびblobごとに1つのファイル、さらに、`pg_restore`から読み取り可能な、機械的に読み取り易い書式でダンプしたオブジェクトを記述する目次ファイルと呼ばれるファイルを持つディレクトリを作成します。ディレクトリ形式アーカイブは標準Unixツールで操作することができます。例えば、未圧縮アーカイブ内のファイルをgzipツールを使用して圧縮することができます。この形式はデフォルトで圧縮されます。

```
t
tar
```

`pg_restore`への入力に適したtar形式のアーカイブを出力します。このtar形式はディレクトリ形式と互換性があります。tar形式アーカイブを展開すると、有効なディレクトリ形式のアーカイブを生成します。しかしtar形式は圧縮をサポートせず、個々のテーブルのサイズに関して8ギガバイトという上限があります。またリストア時にテーブルデータ項目の相対的な順序を変更することはできません。

```
-i
--ignore-version
```

廃止されたオプションであり、現在は無視されます。

```
-n schema
--schema=schema
```

`schema`に一致するスキーマのみをダンプします。これはスキーマ自体とそこに含まれるオブジェクトすべてを選択します。このオプションが指定されなければ、対象データベース内にあるシステム以外のスキーマ全てがダンプされます。複数の`-n`オプションを記述することで、複数のスキーマを選択することができます。また、`schema`パラメータ

はpsqlの`\d`コマンドと同じ規則に従うパターン（[パターン](#)参照）として解釈されます。 ですので、ワイルドカード文字をパターン内に記述することで、複数のスキーマを選択することもできます。 ワイルドカードを使用する時は、シェルによりそのワイルドカードを展開させないように、パターンを引用符で括ってください。 [例](#)を参照してください。

注意: `-n`が指定されると、`pg_dump`は選択したスキーマ内のオブジェクトが依存する可能性がある、その他のデータベースオブジェクトのダンプを行いません。 したがって、スキーマ指定のダンプ結果を初期状態のデータベースに正常にリストアできるかどうかの保証はありません。

注意: `-n`が指定されると、`blob`などの非スキーマオブジェクトはダンプされません。 `--blobs`オプションをつけてダンプを行うことで`blob`も追加されます。

```
-N schema
--exclude-schema=schema
```

`schema`パターンに一致するスキーマをダンプしません。 このパターンは`-n`と同様の規則に従って解釈されます。 `-N`を複数指定して、複数のパターンのいずれかに一致するスキーマを除外することができます。

`-n`と`-N`の両方が指定された場合、少なくとも1つの`-n`に一致し`-N`オプションに一致しないスキーマだけがダンプされます。 `-n`なしで`-N`が指定された場合、`-N`に一致するスキーマが通常のダンプから除外されます。

```
-O
--oids
```

各テーブルのオブジェクト識別子（OID）をデータの一部としてダンプします。 アプリケーションでOID列を（外部キー制約など）何らかの形で使用している場合は、このオプションを使用してください。 その他の場合は、このオプションは使用しないでください。

```
-O
--no-owner
```

オブジェクトの所有権を元のデータベースに一致させるためのコマンドを出力しません。 デフォルトでは、`pg_dump`は、`ALTER OWNER`文または`SET SESSION AUTHORIZATION`文を発行して、作成したデータベースオブジェクトの所有権を設定します。 スーパーユーザ（もしくは、そのスクリプト内の全てのオブジェクトを所有するユーザ）以外のユーザがスクリプトを実行した場合、これらの文は失敗します。 任意のユーザがリストアできるスクリプトを作成するには、`-O`を指定してください。 ただし、この場合は、全てのオブジェクトの所有者がリストアしたユーザとなってしまいます。

このオプションは平文形式の場合にのみ有効です。 アーカイブ形式では、`pg_restore`を呼び出す時にこのオプションを指定することができます。

```
-R
--no-reconnect
```

このオプションは廃止されましたが、後方互換性を保持するため受け入れられます。

```
-s
--schema-only
```

データ定義（スキーマ）のみをダンプし、データはダンプしません。

このオプションは`--data-only`の逆です。 これは`--section=pre-data --section=post-data`を指定することと似ていますが、歴史的な理由のため同一ではありません。

（これと`--schema`オプションと混乱しないでください。 "schema"という単語を異なる意味で使用しています。）

データベース内の一部のみのテーブルのテーブルデータを除外するためには`--exclude-table-data`を参照してください。

```
-S username
--superuser=username
```

トリガを無効にする場合に使用する、スーパーユーザのユーザ名を指定します。 これは`--disable-triggers`を使う場合にのみ使用されます。（通常は このオプションを使うよりも、出力されたスクリプトをスーパーユーザ権限で実行する方が良いでしょう。）

```
-t table
--table=table
```

`table`に一致するテーブル（またはビュー、シーケンス、外部テーブル）のみをダンプします。 複数の`-t`オプションを記述することで複数のテーブルを選択することができます。 また、`table`パラメータはpsqlの`\d`コマンドで使用される規則（[パターン](#)参照）と同じ規則に従うパターンとして解釈されます。 ですので、ワイルドカード文字をパターン内に記述することで、複数のテーブルを選択することもできます。 ワイルドカードを使用する時は、シェルによりそのワイルドカードを展開させないように、パターンを引用符で括ってください。 [例](#)を参照してください。

`-t`が使用されると、`-n`および`-N`オプションの効果はなくなります。 `-t`で選択したテーブルが、これらのオプションとは

関係なくダンプされ、また、非テーブルオブジェクトはダンプされないためです。

注意: `-t`が指定されると、`pg_dump`は選択したテーブル内のオブジェクトが依存する可能性がある他のデータベースオブジェクトのダンプを行いません。したがって、テーブル指定のダンプ結果を初期化されたデータベースに正常にリストアできるかどうかの保証はありません。

注意: `-t`オプションの動作は8.2より前のバージョンのPostgreSQLと完全な互換性はありません。以前は、`-t tab`と記述することで`tab`という名前のテーブルをすべてダンプしていました。しかし現在は、デフォルトの検索パスで可視のものだけがダンプされます。過去の動作を行うためには、`-t '*.tab'`と記述してください。また、特定のスキーマ内のテーブルを選択するためには、以前は`-n sch -t tab`と記述していましたが、`-t sch.tab`などと記述しなければなりません。

```
-T table
--exclude-table=table
```

`tables`パターンに一致するテーブルをまったくダンプしません。このパターンは`-t`と同じ規則に従って解釈されます。`-T`を複数指定することで、複数のパターンに一致するテーブルを除外させることができます。

`-t`と`-T`の両方が指定された場合、少なくとも1つの`-t`オプションに一致し、`-T`オプションに一致しないテーブルのみがダンプされます。`-t`なしで`-T`が指定された場合、通常のダンプから`-T`に一致するテーブルが除外されます。

```
-v
--verbose
```

冗長モードを指定します。これを指定すると、`pg_dump`は、詳細なオブジェクトコメント、開始時刻、終了時刻をダンプファイルに、進行状況メッセージを標準エラーに出力します。

```
-V
--version
```

`pg_dump`のバージョンを表示し、終了します。

```
-x
--no-privileges
--no-acl
```

アクセス権限 (`grant/revoke` コマンド) のダンプを抑制します。

```
-Z 0..9
--compress=0..9
```

使用する圧縮レベルを指定します。ゼロは圧縮無しを意味します。カスタムアーカイブ形式では、これは個々のテーブルデータセグメントの圧縮を指定するもので、デフォルトでは中間レベルで圧縮されます。平文出力では、非ゼロの圧縮レベルの指定によりあたかもgzipに渡されたかのように出力ファイル全体が圧縮されます。しかしデフォルトは圧縮無しです。tarアーカイブ形式では現在圧縮を全くサポートしていません。

```
--binary-upgrade
```

このオプションは現位置でのアップグレード用のユーティリティにより使用されるものです。他の目的での使用は推奨されませんし、サポートもされません。このオプションの動作は、将来注意することなく変更される可能性があります。

```
--column-inserts
--attribute-inserts
```

明示的に列名を付けたINSERTコマンド (`INSERT INTO table (column, ...) VALUES...`) としてデータをダンプします。これによりリストアは非常に遅くなります。主にPostgreSQL以外のデータベースへロード可能なダンプを作成する時に有用です。しかし、このオプションは各行に対して別々のコマンドを生成しますので、一行を再ロードする時にエラーになったとしても、テーブルの内容まるごと失われることなく、一行のみが失われるだけです。

```
--disable-dollar-quoting
```

このオプションは、関数本体用のドル引用符の使用を無効にし、強制的に標準SQLの文字列構文を使用した引用符付けを行います。

```
--disable-triggers
```

このオプションは、データのためのダンプを作成する場合にしか適用されません。データの再ロード中に、`pg_dump`に対し、対象テーブル上のトリガを一時的に無効にするコマンドを出力するよう指示します。このオプションは、データの再ロード中には呼び出したいくない参照整合性検査やその他のトリガがテーブル上にある場合に使用します。

現在のところ、`--disable-triggers`を指定してコマンドを実行するのは、スーパーユーザでなければなりません。そのため、ユーザは`-s`でスーパーユーザを指定するか、あるいは、十分に注意してスーパーユーザ権限でスクリプトを開始する必要があります (後者の方がより望ましい方法です)。

このオプションは平文形式の場合にのみ有効です。他の形式では、`pg_restore`を呼び出す時にこのオプションを指定する

ことができます。

`--exclude-table-data=table`

`table`に一致するすべてのテーブルのデータをダンプしません。パターンは`-t`用の規則と同じ規則にしたがって解釈されます。複数のパターンのいずれかに一致するテーブルを除外することができるように、`--exclude-table-data`を複数回与えることができます。このオプションは、特定のテーブルに関してデータを格納する必要がないがテーブル定義が必要である場合に有用です。

データベース内のすべてのテーブルに関してデータを除外するためには、`--schema-only`を参照してください。

`--inserts`

データを（`COPY`ではなく）`INSERT`コマンドの形式でダンプします。これを行うとリストアに非常に時間がかかります。主にPostgreSQL以外のデータベースへロード可能なダンプを作成する時に有用です。しかし、このオプションは各行に対して別々のコマンドを生成しますので、一行を再ロードする時にエラーになったとしても、テーブルの内容まるごと失われることなく、一行のみが失われるだけです。列の順序を変更した場合はリストアが失敗する可能性があることに注意してください。`--column-inserts`の方がより処理が遅くなりますが、列の順序変更に対して安全です。

`--lock-wait-timeout=timeout`

ダンプの開始時に共有テーブルのロックを永遠に待ちません。代わりに指定した`timeout`内にテーブルロックを獲得できなければ失敗します。タイムアウトには`SET statement_timeout`で受け付けられるすべての書式を指定できます。（使用可能な値はダンプの元となるサーバのバージョンに依存して異なります。しかし、7.3以降ではミリ秒単位の整数値を受け付けられます。7.3より前のサーバからダンプする場合、このオプションは無視されます。）

`--no-security-labels`

セキュリティラベルをダンプしません。

`--no-tablespaces`

テーブル空間を選択するコマンドを出力しません。このオプションを使用すると、すべてのオブジェクトはリストア時のデフォルトのテーブル空間の中に作成されます。

このオプションは平文書式でのみ有効です。アーカイブ書式では、`pg_restore`を呼び出す時にこのオプションを指定することができます。

`--no-unlogged-table-data`

ログを取らないテーブルの内容をダンプしません。このオプションはテーブル定義（スキーマ）をダンプするかどうかには影響しません。そのテーブルデータのダンプを抑制するだけです。スタンバイサーバからダンプを行う場合、ログを取らないテーブル内のデータは常に除外されます。

`--quote-all-identifiers`

強制的にすべての識別子に引用符を付与します。追加のキーワードが導入されている可能性がある将来のバージョンへの移行用にデータベースをダンプする場合に有用かもしれません。

`--section=sectionname`

指名した部分のみをダンプします。部分名は`pre-data`、`data`、`post-data`のいずれかを取ることができます。複数の部分を選択するために、このオプションは複数回指定することができます。デフォルトではすべての部分をダンプします。

`data`部分には、実際のテーブルデータとラージオブジェクトの中身、シーケンス値が含まれます。`Post-data`項目は、インデックス定義、トリガ定義、ルール定義、有効化された検査制約以外の制約定義が含まれます。`Pre-data`項目は、他のすべてのデータ定義項目が含まれます。

`--serializable-deferrable`

使用されるスナップショットがその後のデータベース状態と一貫性を持つことを保証するために、ダンプ時に`serializable`トランザクションを使用します。ダンプが失敗したり、`serialization failure`により他のトランザクションがロールバックしたりする危険がないように、トランザクションストリーム内で異常が発生することがない時点まで待つことでこれを行います。トランザクション隔離および同時実行性の制御については[第13章](#)を参照してください。

このオプションは障害対策のリカバリのみを目的とするダンプでは利点はありません。元のデータベースを継続して更新しながら、レポート処理や他の読み取りのみの負荷分散のためにデータベースのコピーをロードするために使用されるダンプとして有用になります。こうしないと、ダンプには何らかのトランザクションの直列実行が結局コミットした状態と一貫性がない状態が反映される可能性があります。例えば、バッチ処理技術が使用される場合、バッチは、バッチ内で存在するすべての項目を持たないダンプ内でクローズしたものと表示される可能性があります。

`pg_dump`を始めた時に読み書きを行うトランザクションが存在しない場合、このオプションには違いはありません。読

み書きを行うトランザクションが活動している場合、ダンプの起動が確定できない期間、遅延される可能性があります。動き出してから性能は、このスイッチがある場合とない場合とで違いはありません。

```
--use-set-session-authorization
```

オブジェクトの所有権を決定するために、`ALTER OWNER` コマンドの代わりに標準SQLの`SET SESSION AUTHORIZATION` コマンドを出力します。これにより、ダンプの標準への互換性が高まりますが、ダンプ内のオブジェクトの履歴によっては正しくリストアされない可能性が生じます。また、`SET SESSION AUTHORIZATION` を使用したダンプを正しくリストアするためには、確実にスーパーユーザ権限が必要となります。 `ALTER OWNER` で必要な権限はこれよりも少なくなります。

```
-?
--help
```

`pg_dump` のコマンドライン引数の使用方法を表示し、終了します。

以下のコマンドラインオプションは、データベース接続パラメータを制御します。

```
-h host
--host=host
```

サーバが稼働しているマシンのホスト名を指定します。この値がスラッシュから始まる場合、Unix ドメインソケット用のディレクトリとして使用されます。デフォルトは、設定されていれば `PGHOST` 環境変数から取得されます。設定されていなければ、Unix ドメインソケット接続と仮定されます。

```
-p port
--port=port
```

サーバが接続を監視するTCPポートもしくはローカルUnix ドメインソケットファイルの拡張子を指定します。デフォルトは、設定されている場合、`PGPORT` 環境変数の値となります。設定されていなければ、コンパイル時のデフォルト値となります。

```
-U username
--username=username
```

接続ユーザ名です。

```
-w
--no-password
```

パスワードの入力を促しません。サーバがパスワード認証を必要とし、かつ、`.pgpass` ファイルなどの他の方法が利用できない場合、接続試行は失敗します。バッチジョブやパスワードを入力するユーザが存在しない場合にこのオプションは有用かもしれません。

```
-W
--password
```

データベースに接続する前に、`pg_dump` は強制的にパスワード入力を促します。

サーバがパスワード認証を要求する場合 `pg_dump` は自動的にパスワード入力を促しますので、これが重要になることはありません。しかし、`pg_dump` は、サーバにパスワードが必要かどうかを判断するための接続試行を無駄に行います。こうした余計な接続試行を防ぐために `-w` の入力がある場合もあります。

```
--role=rolename
```

ダンプを作成する際に使用するロール名を指定します。このオプションにより `pg_dump` はデータベースに接続した後 `SET ROLE rolename` コマンドを発行するようになります。認証に使用したユーザ（`-U` で指定されたユーザ）が `pg_dump` で必要とされる権限を持たないが、必要な権限を持つロールに切り替えることができる場合に有効です。一部のインストレーションではスーパーユーザとして直接ログインさせないポリシーを取ることがありますが、このオプションを使用することでポリシーに反することなくダンプを作成することができます。

環境

```
PGDATABASE
PGHOST
PGOPTIONS
PGPORT
PGUSER
```

デフォルトの接続パラメータです。

また、このユーティリティは、他のほとんどのPostgreSQLユーティリティと同様、`libpq` でサポートされる環境変数を使用します（[項31.14](#)を参照してください）。

診断

`pg_dump`は内部で`SELECT`文を実行します。`pg_dump`の実行時に問題が発生する場合は、[psql](#)などを使用して、そのデータベースから情報を選択できることを確認してください。また、`libpq`フロントエンドライブラリで使用されるデフォルトの接続設定や環境変数も適用されます。

通常`pg_dump`のデータベースに対する活動は統計情報コレクタにより収集されます。これを望まない場合、`PGOPTIONS`または`ALTER USER`コマンドを使用して`track_counts`パラメータを偽に設定してください。

注釈

データベースクラスタにおいて`template1`データベースに対し独自の変更を行っている場合、`pg_dump`の出力は、確実に空のデータベースにリストアするように注意してください。そうしないと、おそらく追加されたオブジェクトの重複定義によってエラーが発生します。独自の追加が反映されていない空のデータベースを作成するには、`template1`ではなく`template0`をコピーしてください。以下に例を示します。

```
CREATE DATABASE foo WITH TEMPLATE template0;
```

`--disable-triggers`オプションを使用し、データのためのダンプを行う場合、`pg_dump`はデータを挿入する前にユーザテーブルにトリガを無効にする問い合わせを発行し、データの挿入が完了した後で、それらを再び有効にする問い合わせを発行します。リストアが途中で停止した場合、システムカタログが不適切な状態のままになっている可能性があります。

`tar`アーカイブのメンバのサイズは、8ギガバイト未満に制限されています（これは`tar`ファイル形式自体が持っている制限です）。そのため、いずれか1つのテーブルのテキスト表現がこのサイズを越える場合、この形式は使用できません。`tar`アーカイブとその他の出力形式の合計サイズには制限がありません。ただしオペレーティングシステムによる制限がある場合があります。

`pg_dump`が生成するダンプファイルには、オプティマイザが問い合わせ計画を決定する際に使用される統計情報が含まれていません。そのため、最適な性能を発揮するために、ダンプファイルからリストアした後で`ANALYZE`を実行することをお勧めします。詳しくは[項23.1.3](#)および[項23.1.6](#)を参照してください。ダンプファイルはまた`ALTER DATABASE ... SET`コマンドがまったく含まれません。これらの設定はデータベースユーザと他のインストレーション全体の設定に従って [pg_dumpall](#)によってダンプされます。

`pg_dump`は新しいバージョンのPostgreSQLへのデータ移行に使用されますので、`pg_dump`の出力は`pg_dump`のバージョンより新しいバージョンのPostgreSQLデータベースへロード可能と想定できるようになっています。また、`pg_dump`は自身より古いバージョンのPostgreSQLデータベースを読み取ることもできます。（現在はバージョン7.0までのサーバをサポートします。）しかし、`pg_dump`はそのメジャーバージョンより新しいPostgreSQLサーバのダンプを取ることはできません。試そうとしても、無効なダンプが作成される危険はなく、失敗します。また、`pg_dump`の出力がメジャーバージョンが古いサーバにロードできるとは、たとえ同じバージョンのサーバから取得したダンプであっても、保証されていません。より古いサーバへのダンプファイルのロードには、古いサーバでは理解できない構文を削除するために、ダンプファイルの手作業による修正が必要になることがあります。

例

`mydb`という名前のデータベースをSQLスクリプトファイルにダンプします。

```
$ pg_dump mydb > db.sql
```

`newdb`という名前の（新規に作成した）データベースにスクリプトを再ロードします。

```
$ psql -d newdb -f db.sql
```

カスタム形式のアーカイブファイルにデータベースをダンプします。

```
$ pg_dump -Fc mydb > db.dump
```

ディレクトリ形式アーカイブにデータベースをダンプします。

```
$ pg_dump -Fd mydb -f dumpdir
```

`newdb`という名前の（新規に作成した）データベースにアーカイブファイルを再ロードします。

```
$ pg_restore -d newdb db.dump
```

`mytab`という名前の単一のテーブルをダンプします。

```
$ pg_dump -t mytab mydb > db.sql
```


detroit スキーマ内の emp から始まる名前のテーブルをすべてダンプします。ただし、employee_log という名前のテーブルは除きます。

```
$ pg_dump -t 'detroit.emp*' -T detroit.employee_log mydb > db.sql
```

east または west で始まり gsm で終わるスキーマをすべてダンプします。ただし、test という単語を含む場合は除きます。

```
$ pg_dump -n 'east*gsm' -n 'west*gsm' -N '*test*' mydb > db.sql
```

正規表現記法を使用してオプションをまとめた形で同じことを行います。

```
$ pg_dump -n '(east|west)*gsm' -N '*test*' mydb > db.sql
```

ts_ から始まる名前のテーブルを除き、すべてのデータベースオブジェクトをダンプします。

```
$ pg_dump -T 'ts_*' mydb > db.sql
```

大文字または大文字小文字混在の名前を -t などのスイッチに指定するには、名前を二重引用符で括らなければなりません。さもないと小文字に変換されます。（[パターン](#)を参照してください。）しかし、二重引用符はシェルでも特別に扱われますので、これも引用符で括らなければなりません。したがって、大文字小文字混在の名前を持つテーブルを1つダンプするには、以下のようになります。

```
$ pg_dump -t '"MixedCaseName"' mydb > mytab.sql
```

関連項目

[pg_dumpall](#), [pg_restore](#), [psql](#)

[前のページ](#)

pg_config

[ホーム](#)

[上に戻る](#)

[次のページ](#)

pg_dumpall

pg_dumpall

名前

pg_dumpall -- PostgreSQLデータベースクラスタをスクリプトファイルへ抽出する

概要

```
pg_dumpall [connection-option...] [option...]
```

説明

pg_dumpallはクラスタの全てのPostgreSQLデータベースを、1つのスクリプトファイルに書き出す ("ダンプする") ためのユーティリティです。 スクリプトファイルには、データベースのリストアのためにpsqlへの入力として使うことのできるSQLコマンドが含まれています。 リストアはクラスタ内の各データベースに対してpg_dumpを呼び出すことで行われます。 さらに、pg_dumpallは、全てのデータベースに共通するグローバルオブジェクトもダンプします (pg_dumpはこれらのオブジェクトを保存しません)。 現時点では、これにはデータベースユーザとグループ、テーブル空間、データベース全体に適用するアクセス制限などの属性に関する情報が含まれています。

pg_dumpallは全てのデータベースからテーブルを読み込むため、完全なダンプを作成するには、おそらくデータベーススーパーユーザとして接続する必要があると思われます。 さらに、ユーザとグループを追加したり、データベースを作成したりするので、保存されたスクリプトを実行するにはスーパーユーザ権限が必要になるでしょう。

SQLスクリプトは標準出力に書き込まれます。 それをファイルにリダイレクトするためには、[-f|file]オプションまたはシェルの演算子を使用して下さい。

pg_dumpallは、PostgreSQLサーバに何度か接続しなければなりません (データベースごとに接続することになります)。 パスワード認証を使用している場合、その度にパスワード入力が促されます。 そのような場合は~/.pgpass ファイルを用意しておくくと便利です。 詳細は[項31.15](#)を参照してください。

オプション

以下のコマンドラインオプションは内容や出力形式を制御します。

```
-a
--data-only
```

データのみをダンプし、スキーマ (データ定義) をダンプしません。

```
-c
--clean
```

データベースを再作成するコマンドの前に、データベースのクリーンアップ (削除) するコマンドを書き出します。 ロールおよびテーブル空間用のDROP コマンドも同様に追加されます。

```
-f filename
--file=filename
```

出力を指定したファイルに送ります。 これが省略されると標準出力が使用されます。

```
-g
--globals-only
```

グローバルオブジェクト (ロールとテーブル空間) のみをダンプし、データベースのダンプを行いません。

```
-i
--ignore-version
```

廃止されたオプションで、現在では無視されます。

```
-o
--oids
```

各テーブルのオブジェクト識別子 (OID) をデータの一部としてダンプします。 アプリケーションでOID列を (外部キー

制約など) 何らかの形で使用している場合は、このオプションを使用してください。 その他の場合は、このオプションは使用しないでください。

-O
--no-owner

オブジェクトの所有権を元のデータベースに一致させるためのコマンドを出力しません。 デフォルトでは、pg_dumpallはALTER OWNER文またはSET SESSION AUTHORIZATION文を発行して作成したスキーマ要素の所有権を設定します。 スーパーユーザ（もしくは、スクリプト内の全てのオブジェクトを所有するユーザ）以外のユーザがスクリプトを実行した場合、これらの文は失敗します。 任意のユーザがリストアできるスクリプトを作成するには、-oを指定してください。ただし、この場合は、全てのオブジェクトの所有者がリストアしたユーザとなってしまいます。

-r
--roles-only

ロールのみをダンプし、データベースやテーブル空間のダンプを行いません。

-s
--schema-only

オブジェクト定義（スキーマ）のみをダンプし、データをダンプしません。

-S *username*
--superuser=*username*

トリガを無効にする際に使用するスーパーユーザのユーザ名を指定してください。 これは--disable-triggersを使用する場合にのみ使用されます（通常はこのオプションを使うよりも、出力されたスクリプトをスーパーユーザ権限で実行する方が良いでしょう）。

-t
--tablespaces-only

テーブル空間のみをダンプし、データベースやロールのダンプを行いません。

-v
--verbose

冗長モードを指定します。 これを指定すると、pg_dumpallは開始時刻と終了時刻をダンプファイルに、進行メッセージを標準エラーに出力するようになります。 また、これによりpg_dumpの冗長出力が有効になります。

-V
--version

pg_dumpallのバージョンを表示し、終了します。

-x
--no-privileges
--no-acl

アクセス権限のダンプ（grant/revokeコマンド）を行いません。

--binary-upgrade

このオプションは現位置でのアップグレード用のユーティリティにより使用されるものです。 他の目的での使用は推奨されませんし、サポートもされません。 このオプションの動作は、将来注意することなく変更される可能性があります。

--column-inserts
--attribute-inserts

明示的に列名を付けたINSERTコマンド（INSERT INTO *table* (*column*, ...) VALUES...）としてダンプします。 これによりリストアは非常に遅くなります。 主に、PostgreSQL以外のデータベースにロードで可能なダンプを作成する時に有用です。

--disable-dollar-quoting

このオプションは、関数本体用のドル引用符の使用を無効にし、強制的に標準SQLの文字列構文を使用した引用符付けを行います。

--disable-triggers

このオプションは、データのためのダンプを作成する場合だけに使用します。 データのリロード中に、対象とするテーブル上のトリガを一時的に使用不可にするためのコマンドを出力するようpg_dumpallに指示します。 このオプションは、データのリロード中には呼び出したくない参照整合性検査やその他のトリガがテーブル上にある場合に使用します。

現在のところ、--disable-triggersを指定してコマンドを実行するのは、スーパーユーザでなければなりません。 そのため、ユーザは-sでスーパーユーザを指定するか、あるいは、十分に注意してスーパーユーザ権限でスクリプトを開始する必要があります（後者の方がより望ましい方法です）。

--inserts

データを（COPYではなく）INSERTコマンドとしてダンプします。 これを行うとリストアが非常に遅くなります。 主にPostgreSQL以外のデータベースにロード可能なダンプを作成する時に有用です。 列の順序を変更した場合はリストアが失敗する可能性があることに注意してください。 さらに低速になりますが、--column-insertsオプションの方が安全です。

--lock-wait-timeout=*timeout*

ダンプ開始時に共有テーブルロックの獲得のために永遠に待機しません。 代わりに指定した*timeout*の間にテーブルをロックすることができない場合は失敗します。 タイムアウトはSET statement_timeoutで受け付けられる任意の書式で指定することができます。 許される値はダンプ元のサーババージョンに依存して変わります。 しかし、7.3以降のすべてのバージョンでミリ秒単位の整数値は受け付けられます。 このオプションは7.3より前のサーバからダンプされた場合、無視されます。

--no-security-labels

セキュリティラベルをダンプしません。

--no-tablespaces

オブジェクト用にテーブル空間を作成または選択するコマンドを出力しません。 このオプションを付けると、すべてのオブジェクトはリストア時デフォルトであったテーブル空間内に作成されます。

--no-unlogged-table-data

ログを取らないテーブルの内容をダンプしません。 このオプションはテーブル定義（スキーマ）をダンプするかどうかには影響しません。 そのテーブルデータのダンプを抑制するだけです。

--quote-all-identifiers

強制的にすべての識別子に引用符を付与します。 追加のキーワードが導入されている可能性がある将来のバージョンへの移行用にデータベースをダンプする場合に有用かもしれません。

--use-set-session-authorization

ALTER OWNERコマンドの代わりに標準SQLのSET SESSION AUTHORIZATIONコマンドを出力します。 これにより、ダンプの標準への互換性が高まりますが、ダンプ内のオブジェクトの履歴によっては正しくリストアされない可能性があります。

-?
--help

pg_dumpallコマンドライン引数の使用方法を表示し、終了します。

以下のコマンドラインオプションは、データベース接続パラメータを制御します。

-h *host*
--host=*host*

データベースサーバが稼働しているマシンのホスト名を指定します。 この値がスラッシュから始まる場合、Unixドメインソケット用のディレクトリとして使用されます。 デフォルトは、設定されていればPGHOST環境変数から取得されます。 設定されていなければ、Unixドメインソケット接続と仮定されます。

-l *dbname*
--database=*dbname*

グローバルオブジェクトをダンプし、他のどのデータベースをダンプすべきかを見つけるために接続するデータベースの名前を指定します。 指定されなかった場合、postgresが使用されます。 もしこれも存在しない場合はtemplate1が使用されます。

-p *port*
--port=*port*

サーバが接続を監視するTCPポートもしくはローカルUnixドメインソケットファイルの拡張子を指定します。 デフォルトは、設定されている場合、PGPORT環境変数の値になります。 設定されていなければ、コンパイル時のデフォルト値となります。

-U *username*
--username=*username*

接続ユーザ名です。

-w
--no-password

パスワードの入力を促しません。 サーバがパスワード認証を必要とし、かつ、.pgpassファイルなどの他の方法が利用できない場合、接続試行は失敗します。 バッチジョブやパスワードを入力するユーザが存在しない場合にこのオプションは

有用かもしれません。。

```
-W
--password
```

データベースに接続する前に、pg_dumpallは強制的にパスワード入力を促します。

サーバがパスワード認証を要求する場合pg_dumpallは自動的にパスワード入力を促しますので、これが重要になることはありません。しかし、pg_dumpallは、サーバにパスワードが必要かどうかを判断するための接続試行を無駄に行います。こうした余計な接続試行を防ぐために-Wの入力が有意となる場合もあります。

パスワードの入力はダンプするデータベース毎に繰り返し促されます。通常は、手作業のパスワード入力に依存するよりも~/.pgpassを設定する方が良いでしょう。

```
--role=rolename
```

ダンプを作成する際に使用するロール名を指定します。このオプションによりpg_dumpallはデータベースに接続した後にSET ROLE rolenameコマンドを発行するようになります。認証に使用したユーザ（-Uで指定されたユーザ）がpg_dumpallで必要とされる権限を持たないが、必要な権限を持つロールに切り替えることができる場合に有効です。一部のインストレーションではスーパーユーザとして直接ログインさせないポリシーを取ることがありますが、このオプションを使用することでポリシーに反することなくダンプを作成することができます。

環境

```
PGHOST
PGOPTIONS
PGPORT
PGUSER
```

デフォルトの接続パラメータです。

また、このユーティリティは、他のほとんどのPostgreSQLユーティリティと同様、libpqでサポートされる環境変数を使用します（[項31.14](#)を参照してください）。

注釈

pg_dumpallは、内部でpg_dumpを呼び出すので、診断メッセージの一部ではpg_dumpを参照しています。

オプティマイザが正確な統計情報を使用できるように、リストア後は、リストアしたテーブルそれぞれに対してANALYZEを実行することを勧めます。また、vacuumdb -a -zを実行すると、全てのデータベースを解析することができます。

pg_dumpallでは、必要なテーブル空間用のディレクトリがリストア前に存在していることを要求します。存在しないと、デフォルト以外の場所にあるデータベースに関して、そのデータベース生成が失敗します。

例

全てのデータベースを書き出す場合、以下のようにします。

```
$ pg_dumpall > db.out
```

上記のファイルからデータベースを読み込む場合、以下のようにします。

```
$ psql -f db.out postgres
```

（ここではどのデータベースに接続するかということは問題になりません。なぜならpg_dumpallが作成するスクリプトファイルには、保存されたデータベースの作成および接続のためのコマンドが含まれているからです。）

関連項目

考えられるエラーの原因については、[pg_dump](#)を参照してください。

pg_receivexlog

名前

pg_receivexlog -- PostgreSQL クラスタからトランザクションログをストリームする

概要

pg_receivexlog [*option...*]

説明

pg_receivexlogは実行中のPostgreSQLクラスタからトランザクションログをストリームするために使用されます。 トランザクションログはストリーミングレプリケーションプロトコルを使用してストリームされ、ローカルディレクトリのファイルとして書き出されます。 このディレクトリはポイントインタイムリカバリ ([項24.3](#)参照) を用いてリストアする際のアーカイブ場所として使用することができます。

pg_receivexlogはトランザクションログをサーバで生成された時と同じ時刻にストリームし、[archive_command](#)のようにセグメントが完了するまで待機することはありません。 このため、pg_receivexlogを使用する場合には[archive_timeout](#)を設定する必要はありません。

トランザクションログは通常のPostgreSQL接続を経由して、そしてレプリケーションプロトコルを使用して、ストリームされます。 この接続はスーパーユーザまたはREPLICATION権限 ([項20.2](#)参照) を持つユーザで確立されなければなりません。 そしてpg_hba.confで明示的にレプリケーション用の接続を許可しなければなりません。 またサーバではストリーム用に利用できるセッションが少なくとも1つ存在できるように[max_wal_senders](#)を十分大きく設定しなければなりません。

接続が失われた場合、または、致命的ではないエラーで初期確立ができなかった場合、pg_receivexlogは無期限に接続を再試行しできるだけ早くストリーミングを再確立します。 この動作を止めるためには-nパラメータを使用してください。

オプション

以下のコマンドラインオプションにより格納場所や出力書式を制御します。

```
-D directory
--directory=directory
```

出力を書き出すディレクトリです。

このパラメータは必須です。

以下のコマンドラインオプションはプログラムの実行を制御します。

```
-n
--no-loop
```

接続エラー時に繰り返しません。 代わりにエラーですぐに終了します。

```
-v
--verbose
```

冗長モードを有効にします。

以下のコマンドラインオプションはデータベース接続パラメータを制御します。

```
-h host
--host=host
```

サーバが稼働しているマシンのホスト名を指定します。 名前がスラッシュから始まる場合、Unixドメインソケット用のディレクトリとして使用されます。 デフォルトはPGHOST環境変数が設定されていればその値から取られ、設定されていない場合はUnixドメインソケット接続が試行されます。

```
-p port
--port=port
```

サーバが接続を待ち受けるTCPポートまたはUnixドメインソケットファイルの拡張子を指定します デフォルトはPGPORT環境変数が指定されていればその値から取られ、設定されていない場合はコンパイル時のデフォルト値から取られます。

```
-s interval
--status-interval=interval
```

サーバに状態パケットを返答する期間を秒単位で指定します。これは、リプリケーションタイムアウトがサーバで設定されている場合は必須であり、より簡単に監視することができます。ゼロという値は状態の更新を完全に無効にします。デフォルト値は10秒です。

```
-U username
--username=username
```

接続するユーザ名です。

```
-w
--no-password
```

パスワード入力を促しません。サーバがパスワード認証を必要とし、かつ、.pgpassファイルなど他の手段でパスワードが入手できない場合、接続試行は失敗します。このオプションは、パスワードを入力するユーザが存在しないバッチジョブやスクリプトで有用になります。

```
-W
--password
```

pg_receivexlogはデータベースに接続する前にパスワード入力を強制的に促します。

このオプションは重要ではありません。pg_receivexlogは、サーバがパスワード認証を要求した場合に自動的にパスワードを促すためです。しかしpg_receivexlogは、サーバがパスワードを要求するかどうかを確認するために接続試行を浪費します。-wと入力して無駄な接続試行を防止することが有意である場合があります。

以下のこの他のオプションも使用することができます。

```
-v
--version
```

pg_receivexlogのバージョンを表示し、終了します。

```
-?
--help
```

pg_receivexlogコマンドライン引数についてのヘルプを表示し、終了します。

環境

他のほとんどのPostgreSQLユーティリティと同様、このユーティリティはlibpqでサポートされる環境変数（[項31.14参照](#)）を使用します。

注釈

[archive_command](#)の代わりにpg_receivexlogを使用する場合、サーバは、バックアップが適切に保管されていない場合であっても、失敗するコマンドが存在しないため、トランザクションログの再回収を継続します。これは、ファイルがまだ適切に保管できない場合に失敗する[archive_command](#)を持たせることで回避することができます。以下に例を示します。

```
archive_command = 'sleep 5 && test -f /mnt/server/archivedir/%f'
```

始めの待ち時間は、pg_receivexlogが非同期レプリケーションを使用して動作し、そのためマスタからわずかに遅れる可能性があるため、必要です。

例

トランザクションログをmydbserverサーバからストリームし、それを/usr/local/pgsql/archiveローカルディレクトリに格納させます。

```
$ pg_receivexlog -h mydbserver -D /usr/local/pgsql/archive
```


関連項目

[pg_basebackup](#)

[前のページ](#)
pg_dumpall

[ホーム](#)
[上に戻る](#)

[次のページ](#)
pg_restore

pg_restore

名前

pg_restore -- pg_dumpによって作成されたアーカイブファイルからPostgreSQLデータベースをリストアする

概要

```
pg_restore [connection-option...] [option...] [filename]
```

説明

pg_restoreは、[pg_dump](#)によってアーカイブされた平文形式以外のアーカイブファイルを使って、PostgreSQLデータベースをリストアするためのユーティリティです。このコマンドは、データベースを再構成して保存された時点の状態にするために必要なコマンドを発行します。また、pg_restoreは、アーカイブファイルから、リストアする内容を選択したり、リストアする前にアイテムの並び替えを行うこともできます。アーカイブファイルはアーキテクチャに依存しない移植性を持つように設計されています。

pg_restoreの操作には2つのモードがあります。データベース名が指定された場合、pg_restoreはそのデータベースに接続し、アーカイブを直接そのデータベースにリストアします。データベース名が指定されなかった場合は、データベースを再構築するために必要となるSQLコマンドが含まれたスクリプトが作成されます（ファイルもしくは標準出力に書き出されます）。このスクリプトの出力はpg_dumpの平文形式の出力と同じです。実際に、出力を制御するオプションの中には、pg_dumpのオプションに類似したものがあります。

当然ながら、pg_restoreによって、アーカイブファイルに存在しない情報をリストアすることはできません。例えば、アーカイブが"`INSERT`コマンドの形式でデータダンプ"を行うオプションを使用して作成されたものであった場合、pg_restoreは`COPY`文を使用してデータを読み込むことはできません。

オプション

pg_restoreは以下のコマンドライン引数を受け付けます。

filename

リストアするアーカイブファイル（ディレクトリ書式アーカイブの場合はディレクトリ）の場所を指定します。指定がない場合は、標準入力を使用されます。

`-a`
`--data-only`

データのみをリストアし、スキーマ（データ定義）はリストアしません。アーカイブ内にある、テーブルデータ、ラージオブジェクト、シーケンス値がリストアされます。

このオプションは`--section=data`を指定することと似ていますが、歴史的な理由により同一ではありません。

`-c`
`--clean`

再作成前にデータベースオブジェクトを整理（削除）します。（対象のデータベースの中にオブジェクトがまったくない場合でも、害のないメッセージをいくつか出力するかもしれません。）

`-C`
`--create`

リストア前にデータベースを作成します。 `--clean`も同時に指定されている場合、接続する前に対象データベースを削除し再作成します。

このオプションがある場合、`-d`で指定したデータベースは最初の`DROP DATABASE`と`CREATE DATABASE`コマンドの発行時にのみ使用されます。そして、すべてのデータはアーカイブ内に記述された名前のデータベースにリストアされます。

`-d dbname`
`--dbname=dbname`

`dbname` データベースに接続し、このデータベースに直接リストアします。

`-e`
`--exit-on-error`

データベースにSQLコマンドを送信中にエラーが発生した場合、処理を終了します。 デフォルトでは、処理を続行し、リストア処理の最後に発生したエラーの数を表示します。

`-f filename`
`--file=filename`

作成するスクリプト (`-l`を使用した場合はアーカイブの一覧) の出力ファイルを指定します。 デフォルトは標準出力です。

`-F format`
`--format=format`

アーカイブの形式を指定します。 `pg_restore`は形式を自動認識するので、このオプションは必須ではありません。 指定する値は以下のいずれかになります。

`c`
`custom`

アーカイブが`pg_dump`のカスタム形式であることを表します。

`d`
`directory`

アーカイブがディレクトリアーカイブであることを表します。

`t`
`tar`

アーカイブが`tar`アーカイブであることを表します。

`-i`
`--ignore-version`

廃止されたオプションで、現在は無視されます。

`-I index`
`--index=index`

指定したインデックスの定義のみをリストアします。

`-j number-of-jobs`
`--jobs=number-of-jobs`

`pg_restore`のもっとも時間がかかる部分、つまり、データのロード、インデックスの作成、制約の作成部分を複数の同時実行ジョブを使用して実行します。 このオプションは、複数プロセッサマシンで稼働するサーバに大規模なデータベースをリストアする時間を劇的に減らすことができます。

オペレーティングシステムに依存して、各ジョブは1プロセスまたは1スレッドです。 そして別々のサーバへの接続を使用します。

このオプションの最適値はサーバ、クライアント、ネットワークの構成に依存します。 要素にはCPUコア数やディスク構成も含まれます。 試行する最初の値としてサーバのCPUコア数を勧めます。 しかし、多くの場合これより大きな値でもリストア時間を高速化することができます。 当然ながらあまりに大きな値を使用すると、スラッシングのために性能が劣化することになります。

カスタムアーカイブ書式のみがこのオプションをサポートします。 入力ファイルは通常のファイルでなければなりません (例えばパイプはいけません)。 このオプションは、直接データベースサーバに接続するのではなく、スクリプトを生成する場合に無視されます。 また、複数ジョブは`--single-transaction`オプションといっしょに使用することはできません。

`-l`
`--list`

アーカイブの内容を一覧として出力します。 このコマンドが出力する一覧は、`-l`オプションに対する入力として使用することができます。 `-n`や`-t`などのフィルタオプションを`-l`といっしょに使用する場合、一覧出力する項目を制限します。

`-L list-file`
`--use-list=list-file`

`list-file`内で指定したアーカイブ要素のみをリストアします。 また、それらはそのファイルの出現順にリストアされます。 `-n`や`-t`などのフィルタオプションを`-L`といっしょに使用する場合、さらにリストアする項目が制限します。

`list-file`は通常、事前に行った`-l`操作の出力を編集して作成されます。 行の移動や削除、または、行の先頭に`;`を付け

てコメントアウトすることが可能です。 後述の例を参照してください。

```
-n namespace
--schema=Schema
```

指定されたスキーマ内のオブジェクトのみをリストアします。 これは特定のテーブルのみをリストアするために-tオプションと組み合わせることができます。

```
-O
--no-owner
```

オブジェクトの所有者を元のデータベースに合わせるためのコマンドを出力しません。 デフォルトでは、pg_restoreは、ALTER OWNERまたはSET SESSION AUTHORIZATIONを発行して、作成したスキーマ要素の所有者を設定します。 データベースに最初に接続したのがスーパーユーザ（もしくは、そのスクリプト内の全てのオブジェクトを所有するユーザ）でない場合、これらの文は失敗します。 -Oを付与すると、初期接続に任意のユーザ名を使用できるようになります。ただし、この場合は、全てのオブジェクトの所有者がリストアしたユーザになります。

```
-P function-name(argtype [, ...])
--function=function-name(argtype [, ...])
```

指定した関数のみをリストアします。 関数や引数の名前は、ダンプファイルの一覧で出力される通りのスペルで正確に入力するよう注意してください。

```
-R
--no-reconnect
```

このオプションは廃止されました。後方互換性を保持するために受け入れられています。

```
-s
--schema-only
```

アーカイブ内にあるスキーマ項目の範囲でスキーマ（データ定義）のみをリストアし、データ（テーブルの内容）をリストアしません。

このオプションは--data-onlyの逆です。 このオプションは--section=pre-data --section=post-dataを指定することと似ていますが、歴史的な理由により同一ではありません。

（これと--schemaオプションと混乱しないでください。"schema"という単語を異なる意味で使用しています。）

```
-S username
--superuser=username
```

トリガを無効にする場合に使用する、スーパーユーザのユーザ名を指定します。 これは--disable-triggersを使う場合にのみ使用されます。

```
-t table
--table=table
```

指定されたテーブルのみについて、定義、データまたはその両方をリストアします。 -nオプションと組み合わせることでスキーマを指定することができます。

```
-T trigger
--trigger=trigger
```

指定されたトリガだけをリストアします。

```
-v
--verbose
```

冗長モードを指定します。

```
-V
--version
```

pg_restoreのバージョンを表示し、終了します。

```
-x
--no-privileges
--no-acl
```

アクセス権限（grant/revokeコマンド）のリストアを行いません。

```
-1
--single-transaction
```

リストアを単一トランザクションとして実行します（つまり発行するコマンドをBEGIN/COMMITで囲みます）。 これにより確実に、すべてのコマンドが完全に成功するか、まったく変更がなされないかのどちらかになります。 このオプションは--exit-on-errorを意味します。

--disable-triggers

このオプションは、データのみをのダンプを作成する際にしか適用されません。データの再ロード中、pg_restoreに対し、対象テーブル上のトリガを一時的に無効にするコマンドを実行するよう指示します。このオプションは、データの再ロード中には呼び出したくない参照整合性検査やその他のトリガがある場合に使用します。

現在のところ、--disable-triggersを指定してコマンドを実行するのは、スーパーユーザでなければなりません。そのため、ユーザは-sでスーパーユーザを指定するか、あるいはPostgreSQLのスーパーユーザ権限でpg_restoreを実行する必要があります（後者の方がより望ましい方法です）。

--no-data-for-failed-tables

デフォルトでは、関連するテーブルの作成に失敗した（たとえば、既に存在するなどの理由により）としてもテーブルデータオブジェクトはリストアされます。このオプションにより、こうしたテーブルデータは単に無視されるようになります。これは対象のデータベースに目的のテーブルの中身が含まれている時に便利です。たとえばPostGISなどのPostgreSQL拡張用の補助テーブルが既に対象のデータベース内に存在する可能性があります。このオプションを指定すれば、二重ロードや古いデータのロードを防ぐことができます。

このオプションはSQLスクリプト出力を生成する時ではなく、直接データベースにリストアする時のみ効果的です。

--no-security-labels

アーカイブにセキュリティラベルが含まれている場合であっても、セキュリティラベルを戻すコマンドを出力しません。

--no-tablespaces

テーブル空間を選択するコマンドを出力しません。このオプションを付けると、すべてのオブジェクトはリストア時にデフォルトであったテーブル空間内に作成されます。

--section=sectionname

指名された部分のみをリストアします。部分名はpre-data、data、post-dataのいずれかを取ることができます。複数の部分を選択するために、このオプションを複数指定することができます。デフォルトではすべての部分をリストアします。

data部分には、実際のテーブルデータやラージオブジェクト定義が含まれます。Post-data項目は、インデックス定義、トリガ定義、ルール定義、有効化された検査制約以外の制約定義から構成されます。Pre-data項目は、他のすべてのデータ定義項目から構成されます。

--use-set-session-authorization

ALTER OWNER コマンドの代わりに、標準SQLのSET SESSION AUTHORIZATION コマンドを出力して、オブジェクトの所有権を決定します。これにより、ダンプの標準への互換性が高まりますが、ダンプ内のオブジェクトの履歴によっては正しくリストアされない可能性が生じます。

-?
--help

pg_restore コマンドライン引数の使用方法を表示し、終了します。

pg_restoreはさらに以下のコマンドライン引数を接続パラメータとして受け付けます。

-h host
--host=host

サーバが稼働しているマシンのホスト名を指定します。この値がスラッシュから始まる場合、Unixドメインソケット用のディレクトリとして使用されます。デフォルトは、設定されていればPGHOST環境変数から取得されます。設定されていなければ、Unixドメインソケット接続と仮定されます。

-p port
--port=port

サーバが接続を監視するTCPポートもしくはローカルUnixドメインソケットファイルの拡張子を指定します。デフォルトは、設定されている場合、PGPORT環境変数の値となります。設定されていなければ、コンパイル時のデフォルト値となります。

-U username
--username=username

接続ユーザ名です。

-w
--no-password

パスワードの入力を促しません。サーバがパスワード認証を必要とし、かつ、.pgpass ファイルなどの他の方法が利用で

きない場合、接続試行は失敗します。 バッチジョブやパスワードを入力するユーザが存在しない場合にこのオプションは有用かもしれません。

```
-W
--password
```

データベースに接続する前に、pg_restoreは強制的にパスワード入力を促します。

サーバがパスワード認証を要求する場合pg_restoreは自動的にパスワード入力を促しますので、これが重要になることはありません。 しかし、pg_restoreは、サーバにパスワードが必要かどうかを判断するための接続試行を無駄に行います。 こうした余計な接続試行を防ぐために-Wの入力が有意となる場合もあります。

```
--role=rolename
```

リストアを実行する際に使用するロール名を指定します。 このオプションによりpg_restoreはデータベースに接続した後にSET ROLE rolenameコマンドを発行するようになります。 認証に使用したユーザ (-Uで指定されたユーザ) がpg_restoreで必要とされる権限を持たないが、必要な権限を持つロールに切り替えることができる場合に有用です。一部のインストレーションではスーパーユーザとして直接ログインさせないポリシーを取ることがありますが、このオプションを使用することでポリシーに反することなくリストアを行うことができます。

環境

```
PGHOST
PGOPTIONS
PGPORT
PGUSER
```

デフォルトの接続パラメータです。

また、このユーティリティは、他のほとんどのPostgreSQLユーティリティと同様、libpqでサポートされる環境変数を使用します ([項31.14](#)を参照してください) 。

診断

-dオプションによってデータベースに直接接続するよう指定されている場合、pg_restoreは内部でSQL文を実行します。 pg_restoreの実行時に問題が発生する場合は、[psql](#)などを使用して、そのデータベースから情報を選択できることを確認してください。 また、libpqフロントエンドライブラリで使用されるデフォルト接続設定や環境変数もすべて適用されます。

注釈

template1データベースに対し独自の変更を行っている場合、pg_restoreの出力は、確実に空のデータベースにロードするように注意してください。 そうしないと、おそらく追加されたオブジェクトの重複定義によってエラーが発生します。 独自の追加が反映されていない空のデータベースを作成するには、template1ではなくtemplate0をコピーしてください。 以下に例を示します。

```
CREATE DATABASE foo WITH TEMPLATE template0;
```

pg_restoreの制限を以下に示します。

- --disable-triggersオプションを使用して既存のテーブルにデータをリストアする際、pg_restoreは、データを挿入する前に、ユーザテーブル上のトリガを無効にする問い合わせを発行し、データの挿入が完了した後で、それらを再び有効にする問い合わせを発行します。 リストアが途中で停止した場合、システムカタログが不適切な状態のままになっている可能性があります。
- pg_restoreは特定のテーブルのみのラージオブジェクトなどといった、ラージオブジェクトを選択してリストアすることはできません。 アーカイブにラージオブジェクトが含まれる場合、すべてのラージオブジェクトをリストアします。 もし-L、-tなどのオプション経由で除外されていた場合は、全くリストアしません。

pg_dumpの制限についての詳細は、[pg_dump](#)の文書も参照してください。

リストア後は、オプティマイザが有用な統計情報を持つように、リストアしたテーブルそれぞれに対してANALYZEを実行することをお勧めします。 詳しくは[項23.1.3](#)および[項23.1.6](#)を参照してください。

例

mydb という名前のデータベースをカスタム書式のダンプファイルにダンプしているものと仮定します。

```
$ pg_dump -Fc mydb > db.dump
```

データベースを削除し、ダンプファイルから再作成します。

```
$ dropdb mydb
$ pg_restore -C -d postgres db.dump
```

-d オプションのデータベース名には、クラスタに存在するデータベースを指定することができます。pg_restore は、例えば mydb に対する CREATE DATABASE コマンドを発行するためだけに、この名前を使用します。-c を付けると、データは常にダンプファイル内に記載された名前のデータベースにリストアされます。

newdb という新しいデータベースにダンプファイルをリストアします。

```
$ createdb -T template0 newdb
$ pg_restore -d newdb db.dump
```

-c を使用していないことに注目してください。 変わりにリストアするデータベースに直接接続しています。 また、新しいデータベースを template1 ではなく template0 からコピーして作成している点にも注目してください。 確実に初期状態を空にするためです。

データベースのアイテムを並び換えるには、まずこのアーカイブの内容の一覧をダンプしなければなりません。

```
$ pg_restore -l db.dump > db.list
```

一覧ファイルは、ヘッダと各アイテムを1行で表したものから構成されます。

```
;
; Archive created at Mon Sep 14 13:55:39 2009
; dbname: DBDEMOS
; TOC Entries: 81
; Compression: 9
; Dump Version: 1.10-0
; Format: CUSTOM
; Integer: 4 bytes
; Offset: 8 bytes
; Dumped from database version: 8.3.5
; Dumped by pg_dump version: 8.3.8
;
;
; Selected TOC Entries:
;
3; 2615 2200 SCHEMA - public pasha
1861; 0 0 COMMENT - SCHEMA public pasha
1862; 0 0 ACL - public pasha
317; 1247 17715 TYPE public composite pasha
319; 1247 25899 DOMAIN public domain0 pasha
```

セミコロンで始まる行はコメントです。 行の先頭の番号は、各アイテムに割り当てられた内部アーカイブIDを示します。

このファイルの各行に対して、コメントアウト、削除、並び替えを行うことができます。 以下に例を示します。

```
10; 145433 TABLE map_resolutions postgres
;2; 145344 TABLE species postgres
;4; 145359 TABLE nt_header postgres
6; 145402 TABLE species_records postgres
;8; 145416 TABLE ss_old postgres
```

このファイルをpg_restoreの入力として利用すれば、アイテム10と6だけを、この順番でリストアすることができます。

```
$ pg_restore -L db.list db.dump
```

関連項目

[pg_dump](#), [pg_dumpall](#), [psql](#)

[前のページ](#)
pg_receivexlog

[ホーム](#)
[上に戻る](#)

[次のページ](#)
psql

psql

名前

psql -- PostgreSQL対話的ターミナル

概要

```
psql [option...] [dbname [username]]
```

説明

psqlとはPostgreSQLのターミナル型フロントエンドです。対話的に問い合わせを入力し、それをPostgreSQLに対して発行して、結果を確認することができます。また、ファイルから入力を読み込むことも可能です。さらに、スクリプトの記述を簡便化したり、様々なタスクを自動化したりする、いくつかのメタコマンドとシェルに似た各種の機能を備えています。

オプション

```
-a  
--echo-all
```

読み込んだ全ての行を標準出力に表示します。これは対話式モードよりもスクリプト処理の際に有用です。ECHO変数をallに設定するのと同じ意味を持ちます。

```
-A  
--no-align
```

位置揃えなしの出力モードに切り替えます（デフォルトの出力モードは位置揃えありです）。

```
-c command  
--command=command
```

psqlに対し、*command*という1つのコマンド文字列を実行し、終了するよう指示します。このコマンドはシェルスクリプト内で有用です。このオプションを使用すると起動ファイル（psqlrc and ~/.psqlrc）は無視されます。

*command*は、サーバで完全に解析可能な（つまり、psql特有の機能は含まない）コマンド文字列、もしくは、バックスラッシュコマンド1つである必要があります。このため、このオプションではSQLとpsqlメタコマンドを混在させることはできません。これらを同時に使用するには、echo '¥x ¥¥ SELECT * FROM foo;' | psqlのようにパイプを使って文字列をpsqlに渡します（¥¥はメタコマンドの区切り文字です。）。

コマンド文字列が複数のSQLコマンドを含む場合、トランザクションを複数に分けるBEGIN/COMMITコマンドが明示的に文字列内に含まれない限り、それらのコマンドは1つのトランザクションで処理されます。これは、同じ文字列をpsqlの標準入力として渡した場合の動作とは異なります。また最後のSQLコマンドの結果のみが返されます。

```
-d dbname  
--dbname=dbname
```

接続するデータベースの名前を指定します。コマンドラインでオプション以外の最初の引数として*dbname*を指定するのと同じ効力を持ちます。

このパラメータに=記号が含まれる場合、または有効なURI接頭辞(postgresql://またはpostgres://)から始まる場合conninfo文字列として扱われます。詳しくは[項31.1](#)を参照してください。

```
-e  
--echo-queries
```

サーバに送られたコマンドを標準出力にも送ります。ECHO変数をqueriesに設定するのと同じ効力を持ちます。

```
-E  
--echo-hidden
```

¥dやその他のバックスラッシュコマンドによって生成される実際の問い合わせを表示します。これを使って、psqlの内部動作を調べることができます。psql内部からECHO_HIDDEN変数を設定するのと同じ効力を持ちます。

```
-f filename
--file=filename
```

対話式にコマンドを読み取るのではなく、*filename* ファイルをコマンドのソースとして使用します。このファイルの処理が終了した後、psqlは終了します。これは`\i`メタコマンドとほぼ同じ効力を持ちます。

*filename*に`-`（ハイフン）を指定すると、標準入力から読み取られます。

このオプションを指定するのと、`psql < filename`と入力するのでは、微妙に動作が異なります。一般的には、両者とも期待通りの動作を行いますが、`-f`を使用した場合は、エラーメッセージに行番号を付けるなどの機能がいくつか有効になります。また、このオプションを使用した場合、起動時のオーバーヘッドが減少する可能性が若干あります。一方、シェルの入力リダイレクションを使用する方法では、（理論的には）全て手作業で入力した場合の出力とまったく同一な出力になることが保証されます。

```
-F separator
--field-separator=separator
```

*separator*を位置揃えを行わない出力におけるフィールド区切り文字として使用します。`\pset fieldsep`もしくは`\f`と同じ効力を持ちます。

```
-h hostname
--host=hostname
```

サーバを実行しているマシンのホスト名を指定します。この値がスラッシュから始まる場合、Unixドメインソケット用のディレクトリとして使用されます。

```
-H
--html
```

HTML表形式を有効にします。`\pset format html`もしくは`\H`コマンドと同じ効力を持ちます。

```
-l
--list
```

利用可能な全てのデータベースを一覧表示し、終了します。この他の接続に関連しないオプションは無視されます。`\list`メタコマンドと似た効力を持ちます。

```
-L filename
--log-file=filename
```

すべての問い合わせの出力を通常の出力先に出力し、さらにファイル*filename*に書き出します。

```
-n
--no-readline
```

行編集用のreadlineを使用しません。また履歴も使用しません。コピー＆ペースト時のタブ展開を無効にするために有用かもしれません。

```
-o filename
--output=filename
```

全ての問い合わせの出力を*filename*ファイルに書き込みます。これは`\o`コマンドと同じ効力を持ちます。

```
-p port
--port=port
```

サーバが接続監視を行っているTCPポートもしくはローカルUnixドメインソケットファイルの拡張子を指定します。`PGPORT`環境変数の値、環境変数が設定されていない場合はコンパイル時に指定した値（通常は5432）がデフォルト値となります。

```
-P assignment
--pset=assignment
```

`\pset`形式により表示オプションを指定します。ここでは空白ではなく等号を使って名前と値を区切っていることに注意してください。たとえば、出力形式をLaTeXにする場合、`-P format=latex`と入力します。

```
-q
--quiet
```

psqlがメッセージ出力なしで処理を行うように指示します。デフォルトでは、ウェルカム（welcome）メッセージや各種の出力情報が表示されますが、このオプションを使用した場合、これらのメッセージが表示されません。`-c`オプションと併用すると便利です。psql内で`QUIET`変数を設定した場合も、同じ効力を得ることができます。

```
-R separator
--record-separator=separator
```

*separator*を位置揃えを行わない出力におけるレコード区切り文字として使用します。これは`\pset recordsep`コマンドと同じです。

```
-s
--single-step
```

シングルステップモードで実行します。これは各コマンドがサーバに送信される前に、ユーザに対して実行するかキャンセルするかについて確認を求めることを意味します。スクリプトのデバッグを行う時に使用してください。

```
-S
--single-line
```

シングル行モードで実行します。このモードでは、セミコロンと同じように改行もSQLコマンドの終端として扱われます。

注意: このモードはどうしてもこのような方式を使用したいユーザ向けに用意されたものです。無理に使用する必要はありません。特に、1行にSQLとメタコマンドを混在させる場合、経験の浅いユーザにとってその実行順番は必ずしもわかりやすいものではありません。

```
-t
--tuples-only
```

列名と結果の行数フッタなどの表示を無効にします。これは、`¥t` コマンドとまったく同じ効力を持ちます。

```
-T table_options
--table-attr=table_options
```

HTMLの

```
-U username
--username=username
```

デフォルトのユーザではなく`username`ユーザとしてデータベースに接続します（当然、指定したユーザがデータベースに接続する権限を持っている必要があります）。

```
-v assignment
--set=assignment
--variable=assignment
```

`¥set` メタコマンドのように、変数の代入を行います。値がある場合、コマンド上では、名前と値を等号(=)で区切る必要があることに注意してください。等号を指定しないと変数が未設定の状態になります。値が空の変数を設定するには、値を指定しないで等号のみ使用してください。これらの代入は起動時の非常に早い段階で行われます。そのため、内部で使用するために予約されている変数は後で上書きされる可能性があります。

```
-V
--version
```

psqlのバージョンを表示し、終了します。

```
-w
--no-password
```

パスワードの入力を促しません。サーバがパスワード認証を必要とし、かつ、`.pgpass` ファイルなどの他の方法が利用できない場合、接続試行は失敗します。バッチジョブやパスワードを入力するユーザが存在しない場合にこのオプションは有用かもしれません。

このオプションがセッション全体にわたって設定され続けることに注意してください。このため`¥connect`メタコマンドの使用に関しても初期接続試行と同様に影響します。

```
-W
--password
```

データベースに接続する前に、psqlは強制的にパスワード入力を促します。

サーバがパスワード認証を要求する場合psqlは自動的にパスワード入力を促しますので、これが重要になることはありません。しかし、psqlは、サーバにパスワードが必要かどうかを判断するための接続試行を無駄に行います。こうした余計な接続試行を防ぐために`-w`の入力が有意となる場合もあります。

このオプションがセッション全体に対して残ることに注意してください。このため初期接続試行と同様に`¥connect`メタコマンドの使用にも影響を与えます。

```
-x
--expanded
```

拡張テーブル形式モードを有効にします。これは`¥x`コマンドと同じです。

```
-X,
--no-psqlrc
```

起動用ファイル（`psqlrc` ファイルもしくはユーザ用の`~/.psqlrc` ファイル）を読み込みません。

```
-z
--field-separator-zero
```

位置揃えを行わない出力用のフィールド区切り文字をゼロバイトに設定します。

```
-0
--record-separator-zero
```

位置揃えを行わない出力用のレコード区切り文字をゼロバイトに設定します。これは例えば `xargs -0` と連携する時に有用です。

```
-1
--single-transaction
```

`-f` オプションを使用して `psql` がスクリプトを実行する時、このオプションを併記すると、スクリプトを `BEGIN/COMMIT` で囲み、単一トランザクション内でスクリプトを実行します。これにより確実にすべてのコマンドが完全に成功するか、変更がまったく行われないかのいずれかになります。

スクリプト内部で `BEGIN`、`COMMIT`、`ROLLBACK` を使用している場合、このオプションは想定した効果をもたらしません。また、スクリプト内部にトランザクションブロック内部で実行することができないコマンドが含まれている場合、このオプションを指定することで、そのコマンドは失敗（そしてそのためにトランザクション全体が失敗）します。

```
/?
--help
```

`psql` のコマンドライン引数に関するヘルプを表示し、終了します。

終了状態

`psql` は、正常に終了した時には 0 を、`psql` にとって致命的なエラー（メモリ不足やファイルが見つからないなど）が発生した時には 1 を、セッションが対話式でない状態でサーバとの接続が不完全になった時には 2 を、`ON_ERROR_STOP` 変数が設定されている状態でスクリプトでエラーが発生した時には 3 をシェルに返します。

使用方法

データベースへの接続

`psql` は PostgreSQL の正式なクライアントアプリケーションです。データベースに接続するには、接続するデータベース名、ホスト名、サーバのポート番号、接続する際に使用するユーザ名がわかっている必要があります。`psql` では、それらをコマンドラインオプションで指定することができます。接続するデータベース名は `-d`、ホスト名は `-h`、サーバのポート番号は `-p`、接続するユーザ名は `-U` を使用してそれぞれ指定します。オプションに属さない引数がある場合、それはデータベース名（データベース名が与えられている場合にはユーザ名）とみなされます。これらのオプションは全て指定されている必要はありません。指定されていない時はデフォルト値が使用されます。ホスト名を省略した場合、`psql` は Unix ドメインソケット経由でローカルホストにあるサーバに、Unix ドメインソケットを持たないマシンでは `localhost` への TCP/IP 経由で接続します。デフォルトのポート番号はコンパイル時に設定されます。データベースサーバは同じデフォルト値を使用するので、多くの場合、ポートは指定する必要はありません。デフォルトのユーザ名とデータベース名は、Unix のユーザ名です。任意のユーザ名で全てのデータベースに接続できるわけではありません。データベース管理者は、接続権限をユーザに知らせておかねばなりません。

デフォルトがまったく適用できない時は、入力の手間を省くために、環境変数 `PGDATABASE`、`PGHOST`、`PGPORT`、`PGUSER` に適当な値を設定することもできます。（この他の環境変数については、[項31.14](#)を参照してください。）また、`~/.pgpass` ファイルを使用すれば、定常的なパスワードの入力を防ぐことができ、便利です。詳細は[項31.15](#)を参照してください。

他の接続パラメータの指定方法として `conninfo` 文字列または URI があります。これは、データベース名の代わりに使用されます。この機構により、接続全体に関する非常に幅広い制御を行うことができます。

```
$ psql "service=myservice sslmode=require"
$ psql postgresql://dbmaster:5433/mydb?sslmode=require
```

この方法では接続パラメータの検索に、[項31.17](#)で説明する LDAP を使用することもできます。利用できる接続オプションのすべてについての詳細は、[項31.1](#)を参照してください。

何らかの原因（権限がない、指定したホストでサーバが稼働していないなど）で接続ができなかった場合は、`psql` はエラーメッセージを表示し、終了します。

標準入力または標準出力の内最低1つが端末である場合、`psql` はクライアントの符号化方式を "auto" に設定します。これはロケール設定（Unix システムでは `LC_CTYPE` 環境変数）から適切なクライアント符号化方式を決定します。想定した通りに動作しない場合、`PGCLIENTENCODING` 環境変数を使用してクライアント符号化方式を上書きすることができます。

SQLコマンドの入力

通常の操作において、psqlは、psqlが現在接続しているデータベース名の後に=>の文字列が付いたプロンプトを表示します。以下に例を示します。

```
$ psql testdb
psql (9.2.4)
Type "help" for help.

testdb=>
```

プロンプトに対しユーザはSQLコマンドを入力することができます。通常、入力された行は問い合わせ終了を意味するセミコロンに達した時点でサーバへと送信されます。つまり、改行は問い合わせの終了とはみなされません。したがって、わかりやすくするために、コマンドは複数の行にわたって記述することができます。コマンドが送信され問題なく実行されると、画面にコマンドの結果が表示されます。

また、コマンドが実行される度に、psqlは[LISTEN](#)と[NOTIFY](#)によって生成された非同期通知イベントを検査します。

メタコマンド

psql内で入力されたコマンドのうち、バックスラッシュで始まり、引用符で囲まれていないものは、psql自身が実行するpsqlのメタコマンドとして扱われます。これらのコマンドを使うと、データベースを管理したりスクリプトを作成するにあたって、psqlがより便利になります。メタコマンドはよくスラッシュコマンド、またはバックスラッシュコマンドと呼ばれます。

psqlコマンドは、バックスラッシュ、コマンド本体、引数の順につなげた形式になっています。引数とコマンド本体の間と引数間は、空白文字によって分割されています。

引数に空白を含める場合は単一引用符で囲みます。単一引用符を引数に含める場合には、その単一引用符を二重にし、単一引用符で括ってください。単一引用符で囲まれた文字は、C言語と同じような置換の対象となります。このような文字には、`¥n` (改行)、`¥t` (タブ)、`¥b` (後退)、`¥r` (復帰)、`¥f` (改頁)、`¥digits` (8進数で表された文字)、`¥xdigits` (16進数で表された文字)があります。単一引用符で括られたテキスト内でその他の任意の文字の前にバックスラッシュを付けた場合は、その文字が何であろうとその一文字だけとして扱われます。

引数の中で逆引用符(`¥`)で囲まれたテキストは、コマンドラインとして認識され、シェルに渡されます。コマンドの結果(末尾の改行は削除されます)で逆引用符で囲まれたテキストを置き換えます。

[SQL差し替え](#)に示すように、引数の中で引用符で囲まれていないコロン(:)の後にpsql変数名が存在する場合、そこは変数の値に置換されます。

コマンドには、引数としてSQLの(テーブル名などの)識別子を取るものがあります。これらの引数は次のようなSQLの構文規則に従います。引用符を伴わない文字は強制的に小文字になります。しかし、二重引用符(")で囲まれると、大文字小文字変換が行われず、空白文字を識別子内に含めることができます。さらに、二重引用符内では、連続する2つの二重引用符は1つの二重引用符とみなされます。例えば、`FOO"BAR"BAZ`は`fooBARbaz`と解釈され、`"A weird" name"`は`A weird name`になります。

引数の解析は行末または引用符で囲まれていないもう1つのバックスラッシュが見つかったら終了します。引用符がないバックスラッシュは新しいメタコマンドの始まりと解釈されます。`¥¥` (バックスラッシュ2つ)という特別な文字の並びは引数の終わりを意味するので、SQLコマンドが残されている場合は、その解析を続けます。このように、SQLコマンドとpsqlコマンドは1つの行に自由に混合して記述することができます。しかし、あらゆる場合において、メタコマンドの引数は行をまたぐことはできません。

メタコマンドとして、以下のものが定義されています。

`¥a`

現在のテーブルの出力形式が「揃えない」になっていれば「揃える」に、「揃える」になっていれば「揃えない」に設定します。このコマンドは後方互換性を保持するためにあります。より一般的な解決策は`¥pset`を参照してください。

`¥c` or `¥connect [ dbname [ username ] [ host ] [ port ] ]`

PostgreSQLサーバへの接続を新規に確立します。新規接続に成功した場合、以前の接続は閉ざされます。`dbname`、`username`、`host`、`port`のいずれかが省略、あるいは-と指定された場合、対応するパラメータの値はこれまでの接続の値が使用されます。これまで接続されていない場合は、そのパラメータの値にはlibpqのデフォルト値が使用されます。

接続の試行が(ユーザ名の間違いやアクセス拒否などの理由で)失敗した場合、psqlが対話式モードである場合に限つ

て、それまでの接続が保持されます。 非対話式スクリプトを実行している場合は、処理はエラーとなり、即座に停止します。 この実装の違いは、対話モードでは入力ミスに対するユーザの簡便性を考慮し、非対話モードではスクリプトによって間違えたデータベースを操作することを防ぐための安全策を考慮した結果生じています。

```
¥C [ title ]
```

問い合わせ結果として表示されるテーブルタイトルの設定、または、タイトルの設定解除を行います。 このコマンドは、`¥pset title title`と同じ効力を持ちます。 (このコマンド名は"caption"に由来します。 以前はHTMLのテーブルの標題 (caption) を設定するためだけに使われていたためです。)

```
¥cd [ directory ]
```

現在の作業ディレクトリを`directory`に変更します。 引数がない場合は、現在のユーザのホームディレクトリに変更します。

ティップ: 現在の作業ディレクトリを表示するには、`¥! pwd`を使用してください。

```
¥conninfo
```

現在のデータベース接続に関する情報を出力します。

```
¥copy { table [ ( column_list ) ] | ( query ) } { from | to } { filename | stdin | stdout | pstdin | pstdout } [ with ] [ binary ] [ oids ] [ delimiter [ as ] 'character' ] [ null [ as ] 'string' ] [ csv [ header ] [ quote [ as ] 'character' ] [ escape [ as ] 'character' ] [ force quote column_list | * ] [ force not null column_list ] ]
```

フロントエンド (クライアント) コピーを行います。 これは[COPY](#) SQLコマンドを実行する操作ですが、サーバで指定ファイルに対する読み込みまたは書き込みを行うのではなく、psqlがファイルの読み書きや、サーバとローカルファイルシステム間のデータ送信を行います。 この場合、ファイルへのアクセス権限はサーバではなくローカルユーザのものを使用するので、SQLのスーパーユーザ権限は必要ありません。

このコマンドの構文は[COPY](#) SQLコマンドに似ています (詳細はこのコマンドの説明を参照してください) 。 このため、`¥copy` コマンドには特別な解析規則が適用されていることに注意してください。 特に、変数の置換規則やバックslashエスケープは適用されません。

`¥copy ... from stdin | to stdout`はそれぞれコマンドの入力と出力を元に読み書きを行います。 コマンドの発行源と同じところから、`¥.`を読み取るまで、あるいは、ストリームがEOFに達するまで全ての行を読み続けます。 出力はコマンドの出力と同じところに送られます。 psqlの標準入力や標準出力を使用するには、`pstdin`や`pstdout`を使用してください。 これは、SQLスクリプトファイルの内部でテーブルにデータを投入する場合に便利です。

ティップ: この操作はSQLの[COPY](#)コマンドほど効率が良いわけではありません。 これは、全てのデータをクライアント/サーバ接続を通じてやり取りしなければならないからです。 データ量が多い時はSQLコマンドを使用する方が良いでしょう。

```
¥copyright
```

PostgreSQLの著作権および配布条項を表示します。

```
¥d[S+] [ pattern ]
```

`pattern`に一致する各リレーション (テーブル、ビュー、インデックス、シーケンス、外部テーブル) または複合型について、全ての列、列の型、テーブル空間 (デフォルト以外を使用している場合) 、`NOT NULL`やデフォルトなどの特別な属性を表示します。 関連付けられているインデックス、制約、ルールおよびトリガも表示されます。 外部テーブルでは関連する外部サーバも表示されます。 ("パターンのマッチング"については後述の[パターン](#)で定義されています。)

リレーションの一部の種類では、`¥d`は各列について追加の情報を表示します。 例えば、シーケンスでは列の値、インデックスではインデックス式、外部テーブルでは外部データラップのオプションです。

`¥d+`という形式も同一のコマンドを表しますが、より多くの情報を表示します。 こちらでは、テーブルの列に関連付けられたコメントやテーブルにOIDが存在するかどうか、さらにリレーションがビューの場合はビューの定義も表示されます。

デフォルトではユーザが作成したオブジェクトのみが表示されます。 パターンまたは文字`s`修飾子を追加すると、システムオブジェクトが表示に追加されます。

注意: `¥d`が`pattern`引数なしで使用された場合は、`¥dtvsE`と同じ意味となり、可視である全てのテーブル、ビュー、シーケンス、外部テーブルの一覧が表示されます。 これは単に便宜上のものです。

```
¥da[S] [ pattern ]
```

全ての集約関数と、その操作対象となるデータ型、戻り値のデータ型の一覧を表示します。 `pattern`が指定された場合、そのパターンに名前が一致する集約のみが表示されます。 デフォルトではユーザが作成したオブジェクトのみが表示され

ます。パターンまたは文字_s修飾子を追加すると、システムオブジェクトが表示に追加されます。

¥db[+] [[pattern](#)]

テーブル空間を一覧表示します。 *pattern*が指定された場合、そのパターンに名前が一致するテーブル空間のみが表示されます。 コマンド名に+が付与された場合、各オブジェクトに関連付けされた権限についても表示します。

¥dc[S+] [[pattern](#)]

文字セット符号化方式間の変換の一覧を表示します。 *pattern*が指定された場合、そのパターンに名前が一致する変換のみが表示されます。 デフォルトではユーザが作成したオブジェクトのみが表示されます。 システムオブジェクトを含めるためには、パターンまたは_s修飾子を付与してください。 コマンド名に+を付与すると、各オブジェクトに関連する説明を付けて列挙します。

¥dC[+] [[pattern](#)]

型キャストの一覧を表示します。 *pattern*が指定された場合、そのパターンに元データ型または変換先データ型が一致するキャストのみが表示されます。 デフォルトではユーザが作成したオブジェクトのみが表示されます。 パターンまたは文字_s修飾子を追加すると、システムオブジェクトが表示に追加されます。 コマンド名に+を付与すると、各オブジェクトに関連する説明を付けて列挙します。

¥dd[S] [[pattern](#)]

constraint、operator class、operator family、rule、triggerという種類のオブジェクトについての説明を表示します。 他のコメントはすべて、これらのオブジェクト種類用の対応するバックスラッシュコマンドによって表示されます。

¥ddは*pattern*に一致するオブジェクトの説明を表示します。 引数が指定されていない場合は、適切な種類の可視なオブジェクトの説明を表示します。 どちらの場合でも、一覧に表示されるのは説明を持つオブジェクトのみです デフォルトではユーザが作成したオブジェクトのみが表示されます。 パターンまたは文字_s修飾子を追加すると、システムオブジェクトが表示に追加されます。

オブジェクトの説明は[COMMENT](#) SQLコマンドを使用して作成することができます。

¥ddp [[pattern](#)]

デフォルトのアクセス権限設定を列挙します。 組み込みのデフォルトから権限設定が変更されたロール（もし適切ならばスキーマ）ごとに1項目示されます。 *pattern*が指定された場合、パターンに一致するロール名またはスキーマ名の項目のみが列挙されます。

[ALTER DEFAULT PRIVILEGES](#)コマンドを使用して、デフォルトのアクセス権限を設定します。 権限表示の意味は[GRANT](#)で説明します。

¥dD[S+] [[pattern](#)]

ドメインを一覧表示します。 *pattern*が指定されている場合は、パターンに名前が一致するドメインのみが表示されます。 デフォルトではユーザが作成したオブジェクトのみが表示されます。 パターンまたは文字_s修飾子を追加すると、システムオブジェクトが表示に追加されます。 コマンド名に+を付与すると、各オブジェクトに関連する権限と説明を付けて列挙します。

¥dE[S+] [[pattern](#)]

¥di[S+] [[pattern](#)]

¥ds[S+] [[pattern](#)]

¥dt[S+] [[pattern](#)]

¥dv[S+] [[pattern](#)]

このコマンド群において、E、i、s、t、vという文字はそれぞれ、外部テーブル、インデックス、シーケンス、テーブル、ビューを表します。 これらの種類のオブジェクトのリストを入手するために、これらの文字の中の任意の文字またはすべてを指定することができます。 例えば、¥ditはインデックスとテーブルを列挙します。 +がコマンド名に付与された場合、各オブジェクトは、もしあればディスク上の物理容量と関連する説明をつけて列挙されます。 *pattern*が指定されている場合は、パターンに名称が一致する項目のみが表示されます。 デフォルトでは、ユーザが作成したオブジェクトのみが表示されます。 システムオブジェクトを含めるためにはパターンまたは_s修飾子を付与してください。

¥des[+] [[pattern](#)]

外部サーバ（分かりやすく言うと"external servers"）を列挙します。 *pattern*が指定されている場合は、名前がパターンに一致するサーバのみが表示されます。 ¥des+構文が使用された場合、各サーバの完全な説明、サーバのACL、型、バージョン、オプション、説明が表示されます。

¥det[+] [[pattern](#)]

外部テーブル（ニモニック"external tables"）を列挙します。 *pattern*が指定された場合、パターンにテーブル名またはスキーマ名が一致する項目のみが列挙されます。 ¥det+が使用された場合、汎用オプションと外部テーブルの説明も表示されます。

¥deu[+] [[pattern](#)]

ユーザマップ（分かりやすく言うと"外部ユーザ"）を列挙します。 *pattern*が指定されている場合は、名前がパターンに一致するユーザのみが表示されます。 ¥deu+ 構文が使用された場合、各マップについて追加情報が表示されます。

注意

¥deu+ ではリモートユーザのユーザ名とパスワードも表示される可能性があります。 これらを外部に曝さないように注意しなければなりません。

¥dew[+] [[pattern](#)]

外部データラップ（分かりやすく言うと"外部ラップ"）を列挙します。 *pattern*が指定されている場合、名前がパターンに一致する外部データラップのみが表示されます。 ¥dew+ 構文が使用された場合、外部データラップのACL、オプションおよび説明も表示されます。

¥df[antwS+] [[pattern](#)]

関数とその引数と戻り値の型、および、"agg" (集約)、"normal"、"trigger"、"window"で分類される関数の種類の一覧を表示します。 特定種類の関数のみを表示させるには、対応する文字 *a*、*n*、*t*、*w* をコマンドに付けて下さい。 *pattern*が指定されている場合は、そのパターンに名前が一致する関数のみが表示されます。 ¥df+ 構文が使われた場合、各関数の、揮発性、言語、ソースコードや説明を含む付加的情報も表示されます。 デフォルトではユーザが作成したオブジェクトのみが表示されます。 パターンまたは文字 *s* 修飾子を追加すると、システムオブジェクトが表示に追加されます。

ティップ: 特定の型を引数とする関数や特定の型を返す関数を検索するには、ページの検索機能を使用し
て¥dfの出力をスクロールしてください。

¥dF[+] [[pattern](#)]

全文検索設定を列挙します。 *pattern*が指定された場合、このパターンにマッチする名前の設定のみが表示されます。 ¥dF+ 形式が使用された場合、背後の全文検索パーサや各パーサトークン型についての辞書リストなど各設定の完全な説明が表示されます。

¥dFd[+] [[pattern](#)]

全文検索辞書を列挙します。 *pattern*が指定された場合、このパターンにマッチする名前の辞書のみが表示されます。 ¥dFd+ 形式が使用された場合、選択された辞書それぞれについて背後の全文検索テンプレートやオプション値など更なる情報が表示されます。

¥dFp[+] [[pattern](#)]

全文検索パーサを列挙します。 *pattern*が指定された場合、このパターンにマッチする名前のパーサのみが表示されます。 ¥dFp+ 形式が使用された場合、背後の関数や認知されるトークン型のリストなど各パーサの完全な説明が表示されます。

¥dFt[+] [[pattern](#)]

テキスト検索テンプレートを列挙します。 *pattern*が指定された場合、このパターンにマッチする名前のテンプレートのみが表示されます。 ¥dFt+ 形式が使用された場合、テンプレートそれぞれについて背後の関数名など更なる情報が表示されます。

¥dg[+] [[pattern](#)]

データベースロールを一覧表示します。 ("ユーザ"と"グループ"という概念は"ロール"に統合されましたので、このコマンドは¥duと同じものになりました。) *pattern*が指定されている場合は、そのパターンに名前が一致するロールのみが表示されます。 ¥dg+ 構文が使用された場合、ロールそれぞれについて更なる情報が表示されます。 現時点では各ロールのコメントが追加されます。

¥dl

¥lo_listの別名で、ラジオブジェクトの一覧を表示します。

¥dL[S+] [[pattern](#)]

手続き言語を列挙します。 *pattern*を指定すると、パターンに名前が一致する言語のみが表示されます。 デフォルトではユーザが作成した言語のみが表示されます。 *s* 修飾子を追加すると、システムオブジェクトが表示に追加されます。 +がコマンド名に追加すると、呼び出しハンドラ、有効性検証関数、アクセス権限、システムオブジェクトか否かという情報を付けて各言語が列挙されます。

¥dn[S+] [[pattern](#)]

スキーマ（名前空間）の一覧を表示します。 *pattern*を指定すると、パターンに名前が一致するスキーマのみが表示され

ます。デフォルトではユーザが作成したオブジェクトのみが表示されます。パターンまたはs修飾子を追加すると、システムオブジェクトが表示に追加されます。コマンド名の後に+を付加すると、各オブジェクトに関連付けられている権限と説明が（存在すれば）表示されます。

¥do[S] [[pattern](#)]

演算子と、その演算項目と戻り値を一覧表示します。patternを指定すると、パターンに名前が一致する演算子のみが表示されます。デフォルトではユーザが作成したオブジェクトのみが表示されます。パターンまたは文字s修飾子を追加すると、システムオブジェクトが表示に追加されます。

¥dO[S+] [[pattern](#)]

照合順序を列挙します。patternを指定すると、パターンに名前が一致する照合順序のみが表示されます。デフォルトではユーザが作成したオブジェクトのみが表示されます。パターンまたはs修飾子を追加すると、システムオブジェクトが表示に追加されます。コマンド名の後に+を付加すると、各照合順序に関連付けられている説明が（存在すれば）表示されます。現在のデータベースの符号化方式で使用する照合順序のみが表示されます。このため結果は同じインスタレーションにあるデータベースによって異なる可能性があります。

¥dp [[pattern](#)]

テーブル、ビュー、シーケンスを、関連付けられているアクセス権限とともに一覧表示します。patternを指定すると、パターンに名前が一致するテーブル、ビュー、シーケンスのみが表示されます。

[GRANT](#)コマンドと[REVOKE](#)コマンドは、アクセス権限の設定に使われます。権限の表示に関する意味は[GRANT](#)で説明します。

¥drds [[role-pattern](#) [[database-pattern](#)]]

定義済み設定に関する設定を列挙します。これらの設定はロール固有、データベース固有、または両者に対して行うことができます。role-patternおよびdatabase-patternはそれぞれ特定のロールやデータベースを選択するために使用します。省略された、または*が指定された場合、それぞれロール固有でない、または、データベース固有でない設定を含むすべての設定を列挙します。

[ALTER ROLE](#)および[ALTER DATABASE](#)コマンドを使用してロール単位およびデータベース単位の設定を設定します。

¥dT[S+] [[pattern](#)]

データ型を一覧表示します。patternを指定すると、パターンに一致する名前を持つ型のみを表示します。+をコマンド名に付けると、型ごとに、型の内部名、サイズ、enum型では許される値、関連する権限も表示されます。デフォルトではユーザが作成したオブジェクトのみが表示されます。パターンまたは文字s修飾子を追加すると、システムオブジェクトが表示に追加されます。

¥du[+] [[pattern](#)]

データベースロールを一覧表示します。("ユーザ"と"グループ"という概念は"ロール"に統合されましたので、このコマンドは¥duと同じものになりました。) patternが指定されている場合は、そのパターンに名前が一致するロールのみが表示されます。¥dg+構文が使用された場合、ロールそれぞれについて更なる情報が表示されます。現時点では各ロールのコメントが追加されます。

¥dx[+] [[pattern](#)]

インストールされた拡張を列挙します。patternを指定すると、パターンに一致する名前の拡張のみを表示します。¥dx+形式が使用された場合、一致する拡張それぞれについて拡張に属するすべてのオブジェクトが表示されます。

¥eまたは¥edit [*filename*] [*line_number*]

filenameが指定された場合、このファイルが編集されます。エディタを終了した後、その内容は問い合わせバッファにコピーされます。filenameがない場合、現在の問い合わせバッファが一時ファイルにコピーされ、同様に編集されます。

次に、新しい問い合わせバッファは、通常のpsqlの規則に従い、全体を1行として再解析されます（このため、この方法では"スクリプト"を作成できません。この目的のためには¥iを使用してください）。これは問い合わせの終端がセミコロンである（もしくは問い合わせがセミコロンを含む）場合、すぐに実行されることを意味しています。さもなければ、単に問い合わせバッファ内に保持されるだけです。送信するためにはセミコロンまたは¥gを入力、キャンセルするためには¥rを入力してください。

行番号が指定された場合、psqlはファイルまたは問い合わせバッファ内の指定行にカーソルを移します。すべてが数字の単一引数が指定された場合、psqlはそれをファイル名ではなく行番号であると仮定することに注意してください。

ティップ: 使用するエディタを設定、調整する方法については[環境](#)を参照してください。

¥echo *text* [...]

引数を空白で区切り、標準出力に出力し、改行します。 出力するスクリプトのところどころに情報を記載する場合に有効です。 使用例を次に示します。

```
=> \echo `date`  
Tue Oct 26 21:40:57 CEST 1999
```

最初の引数が引用符で囲まれていない`-n`である場合、最後の改行は出力されません。

ティップ: `\o` コマンドを使用して問い合わせの出力先を変更した場合、このコマンドではなく、`\qecho` を使用した方が良いでしょう。

`\ef [ function_description [ line_number ] ]`

このコマンドは指名された関数定義を`CREATE OR REPLACE FUNCTION` コマンド構文で取り出し、編集します。 編集は`\edit`と同様の方法で行われます。 エディタ終了後、更新されたコマンドは問い合わせバッファ内で待機しています。 それを送信するためにはセミコロンか`\g`を入力してください。 取り消す場合は`\r`を入力してください。

対象の関数を名前単体、または、たとえば`foo(integer, text)`のように名前と引数で指定することができます。 同じ名前の関数が複数存在する場合、引数の型を指定しなければなりません。

関数が指定されなかった場合、空の`CREATE FUNCTION`のテンプレートが編集用に表示されます。

行番号が指定された場合、`psql`は関数本体における指定行にカーソルを移動します。（関数本体は通常ファイルの先頭から始まらないことに注意してください。）

ティップ: 使用するエディタを設定、調整する方法については[環境](#)を参照してください。

`\encoding [ encoding ]`

クライアント側の文字セット符号化方式を設定します。 引数を指定しない場合、このコマンドは現在の符号化方式を表示します。

`\f [ string ]`

位置揃えされていない問い合わせの出力用に、フィールドの区切り文字を設定します。 デフォルトは、縦棒（`|`）です。 一般的な出力オプションの設定方法については、`\pset`を参照してください。

`\g [ { filename | command } ]`

現在の問い合わせ入力バッファをサーバに送ります。 オプションを指定すれば、問い合わせ出力を`filename`に格納したり、その出力を別のUnixシェルに渡し、`command`を実行したりすることもできます。 `\g`だけを指定した場合は、セミコロンと実質的に同じです。 `\g`に引数を指定した場合は、`\o`コマンドの"一度限りの"代替手段として使用できます。

`\h`または`\help [ command ]`

指定したSQLコマンドの構文に関するヘルプを表示します。 `command`が指定されていない場合は、`psql`は構文ヘルプが存在する全てのコマンドの一覧を表示します。 `command`をアスタリスク（`*`）にすると、全てのSQLコマンドの構文ヘルプが表示されます。

注意: 入力を簡単にするため、複数の単語からなるコマンドを引用符で囲む必要はありません。 `\help alter table`と入力するだけで十分です。

`\H`

HTML問い合わせ出力形式を有効にします。 HTML形式が有効になっている場合は、デフォルトの位置揃えされたテキスト形式に戻します。 このコマンドは互換性と簡便性のために存在します。 他の出力オプションについては、`\pset`を参照してください。

`\i filename`

`filename`ファイルから入力を読み取り、キーボードから入力された場合と同じように実行します。

注意: 読み取られた行を画面に表示させる場合は、`ECHO`変数を`all`に設定する必要があります。

`\ir filename`

`\ir`コマンドは`\i`と似ていますが、相対ファイル名の解決方法が異なります。 対話モードで実行している場合は2つのコマンドの動作は同一です。 しかし、スクリプトから呼び出す場合、`\ir`は、現在の作業ディレクトリではなく、そのスクリプトの格納ディレクトリから見た相対ファイル名として解釈します。

`\l` (または`\list`)

`\l+` (または`\list+`)

サーバ上の全てのデータベースの名前、所有者、文字セット符号化方式、アクセス権限の一覧を表示します。 コマンド名に+を付けると、データベースのサイズ、デフォルトのテーブル空間、説明も表示します。（サイズ情報は現在のユーザが接続可能なデータベースでのみ表示されます。）

```
¥lo_export loid filename
```

データベースから`loid`というOIDを持つラージオブジェクトを読み取り、`filename`に書き出します。これは`lo_export`サーバ関数とは微妙に異なります。`lo_export`関数は、データベースサーバを実行しているユーザ権限で、サーバ上のファイルシステムに対して動作します。

ティップ: ラージオブジェクトのOIDを確認するには、`¥lo_list`を使用してください。

```
¥lo_import filename [ comment ]
```

ファイルをPostgreSQLのラージオブジェクトに保存します。 オプションで、そのオブジェクトに指定したコメントを関連付けることができます。 下記に例を示します。

```
foo=> ¥lo_import '/home/peter/pictures/photo.xcf' 'a picture of me'
lo_import 152801
```

上の応答は、指定したラージオブジェクトがオブジェクトID 152801として受け付けられたことを示します。 今後この新規作成されたラージオブジェクトにアクセスする場合に、この番号が使用されます。 可読性を高めるために、常に全てのオブジェクトに人間がわかるようなコメントを関連付けることが推奨されます。 `¥lo_list`コマンドではOIDとコメントの両方が表示されます。

このコマンドは、ローカルなユーザによってローカルなファイルシステムに対して動作します。一方、サーバ側の`lo_import`は、サーバのユーザによってサーバ上のファイルシステムに対して動作します。 このコマンドとサーバ側の`lo_import`は、この点で微妙に異なっています。

```
¥lo_list
```

現在データベースに保存されている全てのPostgreSQLラージオブジェクトの一覧を、そのオブジェクトに付けられたコメントと一緒に表示します。

```
¥lo_unlink loid
```

`loid`というOIDが示すラージオブジェクトをデータベースから削除します。

ティップ: ラージオブジェクトのOIDを確認するには、`¥lo_list`を使用してください。

```
¥o [ {filename | |command} ]
```

以降の問い合わせの結果を、`filename`で指定されたファイルに保存するか、または、別のUnixシェルに渡し、`command`を実行します。 引数がない場合、問い合わせの出力は標準出力にリセットされます。

"問い合わせの結果"には、全てのテーブル、コマンドの応答、データベースサーバからの注意メッセージだけでなく、データベースに問い合わせを行う（`¥d`のような）各種バックスラッシュコマンドの出力が含まれます。ただし、エラーメッセージは含まれません。

ティップ: 問い合わせの結果の間にテキストを挿入するには、`¥qecho`を使用してください。

```
¥p
```

現在の問い合わせバッファを標準出力に書き出します。

```
¥password [ username ]
```

指定したユーザ（デフォルトは現在のユーザ）のパスワードを変更します。 このコマンドは、新しいパスワード、暗号化するかどうかを問い合わせ、それを`ALTER ROLE`コマンドとしてサーバに送信します。 これによりコマンド履歴やサーバログなどどこにも新しいパスワードが平文で残りません。

```
¥prompt [ text ] name
```

変数`name`に代入するテキストを入力するようにユーザを促します。 省略可能ですがプロンプト`text`を指定することができます。（複数の単語をプロンプトで使用する場合はそのテキストを単一引用符でくくってください。）

デフォルトでは`¥prompt`は入出力に端末を使用します。 しかし、`-f`コマンドラインスイッチが使用されている場合、`¥prompt`は標準入力、標準出力を使用します。

```
¥pset option [ value ]
```

このコマンドは問い合わせ結果のテーブル出力に影響するオプションを設定します。 `option`には、どのオプションを設定するのかを記述します。 `value`の意味は選択したオプションにより変わります。 以下のオプション別の説明の通り、オブ

ションの中には`value`を省略することでトグルや設定解除を行うものがあります。こうした動作の記載がなければ、`value`を省略すると、単に現在の設定値が表示されることになります。

次は、表示の調整に関するオプションです。

`border`

`value`は数値でなければなりません。基本的には、この数字が多くなればなるほど、表示するテーブルが持つ境界線は増えますが、具体的にはそれぞれの出力形式に依存しています。HTML書式では、この値は直接`border=...`属性に反映されます。他の書式の場合は、0（境界線なし）、1（内側の境界線）、2（テーブル枠）という3つの数値のみ有効です。

`columns`

`wrapped`書式の対象幅を設定し、そして、ページャを必要とする、拡張自動モードにおける縦表示への切替えに十分な幅で出力するかどうかを決定する幅制限を設定します。ゼロ（デフォルト）では、環境変数`COLUMNS`、もし`COLUMNS`が設定されていなければ、検出したスクリーンの幅、により対象幅が制御されます。さらに`columns`がゼロの場合、`wrapped`書式がスクリーン出力にのみ影響を与えることになります。`columns`が非ゼロの場合は、ファイルやパイプへの出力も同様に折り返されます。

`expanded` (または`x`)

`value`を指定する場合は、拡張モードを有効または無効にする`on`または`off`、あるいは`auto`のいずれかでなければなりません。`value`を省略した場合、このコマンドは通常モードと拡張モードをトグルします。拡張モードを有効にした場合、問い合わせ結果は左に列名、右にデータという2つの列で出力されます。このモードは、データが通常の"horizontal（水平）"モードによる画面表示に適していない場合に有用です。自動設定の場合、問い合わせの出力が画面幅より広ければ拡張モードが使用され、さもなければ通常モードが使用されます。自動設定は位置揃え書式または折り返し書式でのみ効果があります。この他の書式では、常に拡張モードが無効の場合と同様に動作します。

`fieldsep`

位置揃えなしの出力書式で使用するフィールド区切り文字を指定します。これにより、例えばタブ区切り、カンマ区切りといった他プログラムがよく使用する形式を作成することができます。タブをフィールド区切り文字として使用するには、`\pset fieldsep '\t'`と入力します。デフォルトのフィールド区切り文字は'|'（縦棒）です。

`fieldsep_zero`

位置揃えなしの出力書式で使用するフィールド区切り文字をゼロバイトに指定します。

`footer`

`value`が指定された場合、それぞれテーブルフッタの表示(`n rows` 数)を有効または無効にする`on`または`off`のいずれかでなければなりません。`value`を省略した場合、このコマンドはフッタの表示、非表示をトグルします。

`format`

出力形式を`unaligned`、`aligned`、`wrapped`、`html`、`latex`、`troff-ms`のいずれかに設定します。一意に判別できる範囲で省略が可能です（この場合、1文字でも十分であることを意味します）。

`unaligned`書式は、表示行の1行に1つの行の全列を、現在有効なフィールド区切り文字で区切って書き出します。これは他のプログラムに読み込ませることを目的とした出力(例えばタブ区切りやカンマ区切り書式)を生成する場合に有用です。

`aligned`書式は、人間が読みやすいように、美しく整形されたテキスト出力です。これがデフォルトです。

`wrapped`書式は`aligned`と似ていますが、データ値を複数行に折り返して対象の列幅に合うように出力します。対象の幅は後述の`columns`オプションにより決定されます。`psql`は列ヘッダタイトルを折り返さないことに注意して下さい。このため`wrapped`書式は列ヘッダに必要な幅全体が対象より長い場合、`aligned`と動作が同じになります。

`html`、`latex`および`troff-ms`書式は対応するマークアップ言語を使用する文書内に含めることを目的とした表を出力します。出力自体は完全な文書ではありません。（HTMLでは大して問題にはならないでしょうが、LaTeXでは完全な文書ラップを持たせなければなりません。）

`linestyle`

境界線の表示形式を`ascii`、`old-ascii`または`unicode`のいずれかに設定します。一意になれば省略形が許されます。（つまり一文字で十分であることを意味します。）デフォルトの設定は`ascii`です。このオプションは`aligned`および`wrapped`出力書式のみで有効です。

`ascii`形式は通常のASCIIを使用します。データ内部の改行は右側余白に+を使用して表します。`wrapped`書式では、改行文字を使わずに、先頭行の右側余白にドット(.)を示し、以降の行の左側余白に繰り返してドットを示して、データを改行します。

`old-ascii`形式は通常のASCII文字を使用して、PostgreSQL 8.4以前で使用されていた方法で整形します。データ内部の改行は列区切りの左側に;記号を使用して表します。データを折り返す際には改行文字を使わず、列区切りの左側に;記号を使用して表します。

`unicode`形式はUnicode矩形描画文字を使用します。データ内部の改行は右側の余白に復帰記号を使用して表します。データを折り返す際には改行文字を使わず、省略記号を先頭行の右側余白に表し、以降の行の左側余白にも繰り返し表します。

`border`設定がゼロより大きい場合、このオプションはまた、境界線を描画する文字も決定します。通常のASCII文字はどのような場合でも動作しますが、Unicode文字の方が認識しやすいという点で表示に優れていると思います。

`null`

`null`値の代わりに表示する文字列を設定します。デフォルトでは何も表示しません。そのため、よく空の文字列と間違ふことがあります。このような場合、`¥pset null '(null)'`を使用する方が良いでしょう。

`numericlocale`

`value`が指定された場合、それぞれ10進数マーカの左に桁のくくりを分離するロケール固有の文字を表示するか否かを指定する、`on`または`off`のいずれかでなければなりません。`value`を省略した場合、このコマンドは通常出力がロケール固有の数値出力かをトグルします。

`pager`

問い合わせおよびpsqlヘルプ出力の際の、ページャプログラムの使用を制御します。`PAGER`環境変数が設定されている場合、出力は指定したプログラムにパイプで渡されます。`PAGER`が設定されていない場合は、プラットフォーム依存のデフォルト (`more`など) が使用されます。

`pager` オプションが`off`の場合、ページャプログラムは使用されません。`pager` オプションが`on`の場合、ページャは適切な場合、つまり出力先が端末であり、その画面に合わない場合に使用されます。また`pager` オプションは`always`に設定することもできます。こうすると画面に合うかどうかに関わらずすべての端末出力でページャが使用されます。`value`を持たない`¥pset pager`はページャの使用をトグルします。

`recordsep`

位置揃えなしの出力書式で使用するレコード (行) の区切り文字を指定します。デフォルトは改行文字です。

`recordsep_zero`

位置揃えなしの出力書式で使用するレコードの区切り文字をゼロバイトに指定します。

`tableattr` (または `T`)

html 出力書式においてHTMLの`table`タグ内に記述する属性を指定します。これを使用して、例えば、`cellpadding`や`bgcolor`を指定することができます。`border`属性は既に`¥pset border`によって処理されているので、このコマンドで`border`を指定する必要はありません。`value`の指定がない場合、テーブル属性の設定は解除されます。

`title`

今後表示される全てのテーブル用にテーブルタイトルを設定します。これは出力に説明のためのタグを付けたい場合に有効です。`value`がない場合、タイトルの設定が解除されます。

`tuples_only` (or `t`)

`value`が指定された場合、それぞれタプルのみを表示を有効または無効にする、`on`または`off`のいずれかでなければなりません。`value`を省略した場合、このコマンドはタプルのみを表示と通常表示をトグルします。通常表示では列のヘッダ、タイトル、各種フッタなどのその他の情報が追加されます。タプルのみモードでは、テーブルの実データのみが表示されます。

例節に、これらの書式がどのように見えるかを示した図があります。

ティップ: `¥pset` には各種のショートカットコマンドがあります。 `¥a`、`¥C`、`¥H`、`¥t`、`¥T`、`¥x`を参照してください。

注意: 引数なしで`¥pset`を呼び出した場合はエラーになります。今後のバージョンでは、現在の表示用オブ

ションの状態が全て表示されるようになる予定です。

`¥q` または `¥quit`

psqlプログラムを終了します。 スクリプトファイルでは、そのスクリプトの実行のみが終了します。

`¥qecho text [ ... ]`

このコマンドは、`¥echo`と同じです。 ただし、`¥o`による設定と同様に、出力が問い合わせ出力チャンネルに書き出される点異なります。

`¥r`

問い合わせバッファをリセット（クリア）します。

`¥s [ filename ]`

コマンドラインの履歴の表示、または`filename`への保存を行います。 `filename`が省略された場合、履歴は標準出力に書き出されます。 このオプションは、コンパイル時に、psqlがGNU Readlineライブラリを使用するように設定されている場合のみ使用可能です。

`¥set [ name [ value [ ... ] ] ]`

`value`、または複数の`value`が与えられた場合はそれらを連結したものを、`name` psql変数に設定します。 第一引数しか指定されない場合は、変数に空の値を設定されます。 変数を未設定とするには、`¥unset`コマンドを使用してください。

引数をまったく取らない`¥set`は、現在設定されているpsql変数すべての名前と値を表示します。

変数名には、文字、数字、アンダースコアを使用することができます。 詳細は、後述の[変数](#)を参照してください。 変数名は大文字小文字を区別します。

必要ならばどのようなものでも任意の変数に設定できますが、psqlはいくつかの変数を特別なものとして扱っています。これらについては変数に関する節にて説明します。

注意: このコマンドはSQLの[SET](#)コマンドとは関係ありません。

`¥setenv [ name [ value ] ]`

`name`環境変数を`value`に設定します。 もし`value`が与えられない場合は、その環境変数を未設定状態にします。 以下に例を示します。

```
testdb=> ¥setenv PAGER less
testdb=> ¥setenv LESS -imx4F
```

`¥sf[+] function_description`

このコマンドは、`CREATE OR REPLACE FUNCTION`コマンドの形式で、指定された関数の定義を抽出し表示します。 この定義は、`¥o`で設定されるように、現在の問い合わせ出力チャンネルに出力されます。

対象の関数は、名前だけまたは、例えば`foo(integer, text)`のように名前と引数で指定することができます。 同じ名前の関数が複数存在する場合は、引数の型を指定しなければなりません。

コマンド名に`+`を付けると、出力行に関数本体の先頭行を1行目と数える行番号が付けられます。

`¥t`

出力列名ヘッダと行数フッタの表示を切り替えます。 このコマンドは`¥pset tuples_only`と同じで、簡便性のために用意されています。

`¥T table_options`

HTML出力書式における`table`タグ内部に記述する属性を指定します。 このコマンドは`¥pset tableattr table_options`と同じ効力を持ちます。

`¥timing [ on | off ]`

パラメータがない場合、各SQL文にかかる時間（ミリ秒単位）の表示の有無を切り替えます。 パラメータがある場合、同じことを設定します。

`¥unset name`

`name` psql変数を未設定状態にします（削除します）。

`¥w filename`

`¥w |command`

現在の問い合わせバッファを、*filename* ファイルに出力するか、もしくは、*command* Unix コマンドにパイプで渡します。

¥x [*on* | *off* | *auto*]

拡張テーブル形式モードを設定またはトグルします。 このコマンドは¥pset expandedと同じ効力を持ちます。

¥z [[pattern](#)]

テーブル、ビュー、シーケンスを、関連付けられているアクセス権限とともに一覧表示します。 *pattern*を指定すると、パターンに名前が一致するテーブル、ビュー、シーケンスのみが表示されます。

これは¥dp ("権限の表示 (display privileges) ") の別名です。

¥! [*command*]

別のシェルを起動するか、もしくは、Unixの*command*コマンドを実行します。 引数はこれ以上解釈されず、そのままシェルに渡されます。

¥?

バックスラッシュ ("¥") コマンドに関するヘルプ情報を表示します。

パターン

¥d コマンドでは、*pattern* パラメータを渡して、表示するオブジェクト名を指定することができます。 パターンが正確にオブジェクト名に一致するというのが最も単純な場合です。 パターン内の文字は、SQL名と同様、通常小文字に変換されます。 例えば¥dt fooはfooという名前のテーブルを表示します。 SQL名と同様、パターンを二重引用符で括ることで小文字への変換が取り止められます。 二重引用符自体をパターン内に含めなければならない場合、二重引用符で括った文字列の中で二重引用符を二重に記載してください。 繰り返しますが、これはSQLの引用符付き識別子の規則に従ったものです。 例えば、¥dt "foo" "bar" はfoo"barという名前のテーブルを表示します (foo"barではありません)。 SQL名と異なる点は、パターンの一部を二重引用符で括ることができる点です。 例えば、¥dt foo"foo"barはfoofoo"barという名前のテーブルを表示します。

pattern パラメータが完全に省略されている場合、¥d コマンドは現在のスキーマ検索パス内で可視的なオブジェクトを全て表示します。 これは*というパターンを使用することと同じです。 (オブジェクトを含むスキーマが検索パス上にあり、同じ種類かつ同じ名前のオブジェクトが検索パス上に存在しない場合、そのオブジェクトは可視であるといえます。 これは文において明示的なスキーマ修飾がない名前でオブジェクトを参照できることと同じです。) 可視か否かに関わらずデータベース内の全てのオブジェクトを表示するには、*. *というパターンを使用します。

パターン内部では、*は (0文字を含む) 任意の文字の並びに一致し、?は任意の1文字に一致します。 (この記法はUnixシェルのファイル名パターンと似ています。) 例えば、¥dt int*は、intから始まる名前を持つテーブルを表示します。 しかし、二重引用符の中では、*と?はその特別な意味を失い、文字そのものに一致することになります。

ドット(.)を含むパターンは、スキーマ名にオブジェクト名が続くパターンとして解釈されます。 例えば、¥dt foo*. *bar*は、fooで始まるスキーマ内のbarを含むテーブルを全て表示します。 ドットがない場合、パターンは現行のスキーマ検索パス内で可視的なオブジェクトのみに一致します。 繰り返しますが、二重引用符で括られた文字列内のドットは特別な意味を失い、文字そのものに一致することになります。

上級者は文字クラスに、例えば任意の数に一致する[0-9]などの正規表現を使用することができます。 上述のように.が区切り文字となる点、*は正規表現の.*記法になる点、?が.になる点、\$がそのまま扱われる点は例外ですが、すべての正規表現の特殊文字が[項9.7.3](#)の規定通りに動作します。 .は?と、R*は(R+|)と、R?は(R|)と記述することで、これらのパターン文字を模擬することができます。 \$を正規表現文字として扱う必要はありません。 通常、正規表現の解釈と異なり、パターンは常に名前全体に一致するためです。 (言い替えると、\$は自動的にパターンに追加されます。) パターンの適用位置を決められない場合は、*を先頭や末尾に記載してください。 二重引用符の内側では、正規表現の特殊文字はその意味を失い、文字そのものに一致することになる点に注意してください。 また、正規表現の特殊文字は、演算子名パターン (つまり¥doの引数) では文字そのものに一致します。

高度な機能

変数

psqlは一般的なUnixコマンドシェルに似た変数の置換機能を提供します。 変数とは名前と値の組み合わせです。 値として任意の長さの任意の文字列を使用できます。 名前は文字(非ラテン文字を含む)、数字、アンダースコアから構成されなければなりません。

変数を設定するには、psqlの¥setメタコマンドを以下のように使用します。

```
testdb=> \set foo bar
```

この例では、`foo`変数を`bar`という値に設定しています。変数の内容を取り出すには、以下のように変数名の前にコロンを付けます。

```
testdb=> \echo :foo
bar
```

これは通常のSQLコマンド内とメタコマンド内の両方で動作します。後述の[SQL差し替え](#)で詳しく説明します。

第二引数を持たない`\set`を呼び出すと、空文字列を値として持つ変数が設定されます。変数を未設定状態にする(つまり削除する)ためには、`\unset`コマンドを使用してください。すべての変数の値を表示するためには、引数をまったく取らない`\set`を呼び出してください。

注意: `\set`の引数は他のコマンドと同じ置換規則に従います。このため、`\set :foo 'something'`のような参照を作成して、Perlにおける"ソフトリンク"やPHPにおける"可変変数"に当たるものを得られます。しかし、残念ながら(あるいは好運にも)、このような構成をうまく使用する方法はありません。一方、`\set bar :foo`のようにして変数をコピーするのは、完全に有効な方法です。

これらの変数の多くは、psqlに特別扱いされています。これらは、変数の値を変更することにより、実行時に変更可能なオプションの設定を表現します。またpsqlの変更可可能な状態を表現しているものもあります。これらの変数を別の目的で使用することもできますが、即座にプログラムの動作がおかしくなる可能性があるため、推奨されません。慣習上、特別視される変数の名前はすべてASCII大文字(と数字とアンダースコア)からなります。将来的な互換性を最大限考慮するために、自分で作成した変数にはこのような変数名を使用しないでください。以下に特別に取り扱われる変数の一覧を示します。

AUTOCOMMIT

この変数の値が`on`の場合(自動コミット有効モード: デフォルト)、各SQLコマンドの実行が成功すると、自動的にコミットされます。コミットを延期するには、`BEGIN`もしくはSQLの`START TRANSACTION`コマンドを入力する必要があります。値が`off`もしくは未設定の場合(自動コミット無効モード)、明示的に`COMMIT`もしくは`END`を発行するまで、SQLコマンドはコミットされません。自動コミット無効モードでは、トランザクションブロック以外でコマンドが発行されると、そのコマンドを実行する前に、自動的に`BEGIN`コマンドが発行されます(ただし、そのコマンド自体が`BEGIN`コマンドやその他のトランザクション制御コマンドである場合、トランザクションブロック内で実行することができないコマンド(`VACUUM`など)である場合は除きます)

注意: したがって、自動コミット無効モードでは、`ABORT`や`ROLLBACK`を発行して、明示的に失敗したトランザクションを放棄しなければなりません。また、コミットせずにセッションを終了した場合は、作業が失われてしまうので注意してください。

注意: PostgreSQLは、伝統的に自動コミット有効モードで動作していましたが、自動コミット無効モードの方がよりSQLの仕様に近いものです。自動コミット無効モードは、システム全体に対する`psqlrc`ファイル、もしくは、個人用の`.psqlrc`ファイルで設定すれば実現できます。

COMP_KEYWORD_CASE

SQLキーワードを補完する時に大文字小文字のどちらを使用するかを決定します。`lower`または`upper`が設定された場合、補完された単語はそれぞれ小文字または大文字になります。`preserve-lower`または`preserve-upper`(デフォルト)が設定された場合、補完された単語は入力済みの文字の大文字小文字を引き継ぎますが、何も入力されていない場合はそれぞれ小文字または大文字に補完されます。

DBNAME

現在接続しているデータベース名です。この変数は(プログラム起動時も含め)データベースに接続する度に設定されますが、これを未設定にすることもできます。

ECHO

`all`を設定した場合、キーボード、もしくは、スクリプトからの取得された全ての行は、解析/実行の前に標準出力に書き出されます。この動作をプログラム起動時に設定するには、`-a`スイッチを使用してください。`queries`を設定した場合、psqlはサーバに送信された問い合わせのみを表示します。これらを切り替えるオプションは`-e`です。

ECHO_HIDDEN

この変数が設定されている場合、バックスラッシュコマンドでデータベースに問い合わせを行う時、最初にその問い合わせが表示されます。これにより、PostgreSQL内部動作について調べたり、自作プログラム内で同様の関数機能を用意したりすることが可能になります(この動作をプログラム起動時に選択するには`-E`スイッチを使用してください)。この変数を`noexec`という値に設定した場合、問い合わせは実際にサーバに送信、実行されずに、単に表示されるだけになります。

ENCODING

現在のクライアント側の文字セット符号化方式です。

FETCH_COUNT

この変数が0より大きな整数値に設定されている場合、`SELECT`問い合わせの結果は、指定した行数の集合として取り出され、表示されます。デフォルトの動作では、表示する前にすべての結果が取り出されます。したがって、結果セットの大きさに比例する形でメモリの使用量が限定されます。この機能を有効とする場合に100から1000までの値がよく使用されます。この機能を使用する際には、既に一部の行が表示されている場合、問い合わせが失敗する可能性があることに注意してください。

ティップ: 任意の出力書式でこの機能を使用することができますが、デフォルトの`aligned`書式は適していません。`FETCH_COUNT`行のグループそれぞれが別々に整形されてしまい、行のグループによって列幅が異なることになるためです。他の出力書式は適切に動作します。

HISTCONTROL

この変数を`ignore_space`に設定した場合、空白文字から始まる行は履歴リストには入りません。`ignore_dups`に設定した場合、これまでの履歴にある行は履歴リストには入りません。`ignore_both`に設定した場合は、上記の2つを組み合わせたものになります。この変数を設定しない場合、または上記以外の値を設定する場合は、対話モードで読まれる全ての行が履歴リストに保存されます。

注意: この機能はBashの機能を真似たものです。

HISTFILE

履歴を格納するために使用されるファイルの名前です。デフォルトは`~/.psql_history`です。例えば、`~/.psqlrc`で以下を記述すると、psqlはデータベース毎に分けて履歴を管理します。

```
¥set HISTFILE ~/.psql_history- :DBNAME
```

注意: この機能はBashの機能を真似たものです。

HISTSIZE

コマンド履歴に保存するコマンド数です。デフォルト値は500です。

注意: この機能はBashの機能を真似たものです。

HOST

接続中のデータベースサーバホストです。この変数は（プログラム起動時も含め）データベースに接続する度に設定されますが、未設定にすることもできます。

IGNOREEOF

この変数を未設定にすると、対話式セッションにEOF文字（通常**Control+D**）が送信された時、psqlが終了します。数値を設定すると、指定された数だけ、送信されたEOF文字を無視してから終了します。数値以外を設定した場合は、デフォルトの10になります。

注意: この機能はBashの機能を真似たものです。

LASTOID

`INSERT`や`¥lo_insert`コマンドによって返された、最後に影響を受けたOIDの値です。この変数は、次のSQLコマンドの結果が表示されるまでの間のみ保証されています。

ON_ERROR_ROLLBACK

`on`の場合、トランザクションブロック内である文がエラーとなった時に、そのエラーは無視され、トランザクションは継続します。`interactive`の場合、対話式セッション内の場合にのみエラーは無視されます。スクリプトファイルを読み込んでいる場合は無視されません。`off`（デフォルト）の場合、トランザクションブロック内の文がエラーになると、トランザクション全体をアボートします。`ON_ERROR_ROLLBACK`が`on`という状態は、トランザクションブロック内で各コマンドの実行直前に暗黙的な`SAVEPOINT`を行い、エラーが起きた時にこのセーブポイントにロールバックすることで実現されています。

ON_ERROR_STOP

デフォルトではエラーの後もコマンド処理は続行されます。この変数が設定されると、代わりに即座に停止します。対話モードではpsqlはコマンドプロンプトに戻ります。これ以外ではpsqlは終了し、エラーコード1を返す致命的エラー条件と区別できるように、エラーコード3を返します。どちらの場合でも、現在実行中のスクリプト（トップレベルのスクリプト、もしあれば関連性を持つ他のスクリプトすべて）は即座に終了します。トップレベルのコマンド文字列が複数のSQLコマンドを含む場合、その時点のコマンドで処理は終了します。

PORT

接続中のデータベースサーバのポートです。 この変数は（プログラム起動時も含め）データベースに接続する度に設定されますが、未設定にすることもできます。

PROMPT1
PROMPT2
PROMPT3

これらの変数は、psqlが発行するプロンプトの見た目を指定します。 後述の[プロンプト](#)を参照してください。

QUIET

この変数は-qコマンドラインオプションと同じ効力を持ちます。 対話式モードではあまり役立ちません。

SINGLELINE

この変数は-sコマンドラインオプションと同じ効力を持ちます。

SINGLESTEP

この変数は-sコマンドラインオプションと同じ効力を持ちます。

USER

接続中のデータベースユーザです。 この変数は（プログラム起動時も含め）データベースに接続する度に設定されますが、未設定にすることもできます。

VERBOSITY

この変数をdefault、verbose、terseのいずれかに設定することで、エラー報告の冗長性を制御できます。

SQL差し替え

psqlの変数には、通常のSQL文やメタコマンドの引数の中で使用（"差し替え：interpolate"）できるという重要な機能があります。 さらにpsqlは、 SQLリテラルと識別子として使用される変数の値が適切に引用符付けされていることを確実にするための機能を提供します。 引用符付けをまったく行わずに差し替えるための構文は、変数名の前にコロンの(:)を付けることです。 以下に例を示します。

```
testdb=> \set foo 'my_table'
testdb=> SELECT * FROM :foo;
```

この例では、問い合わせはmy_tableテーブルに対して行われます。 これが安全ではない場合があることに注意して下さい。 変数の値はそのままコピーされるので、対応のとれていない引用符やバックスラッシュコマンドさえも含めることができます。 挿入した場所で変数が展開された時に、確実に正しい意味になるようにしなければなりません。

値がSQLリテラルや識別子内で使用される場合、それが引用符付けされるように調整することがもっとも安全です。 SQLリテラルとして変数値を引用符付けするためには、単一引用符内にコロンの、その後に変数名を記述してください。 SQL識別子として値を引用符付けするためには、二重引用符内にコロンの、その後に変数名を記述してください。 これらの式は正しく引用符と変数値内に埋め込まれた特殊文字を扱います。 前の例は以下のように記述することでより安全になります。

```
testdb=> \set foo 'my_table'
testdb=> SELECT * FROM : "foo";
```

変数差し替えは、引用符付けされたSQLリテラルと識別子の中では行われません。 したがって':foo'などの式は、変数の値から引用符付けしたリテラルを生成するようには動作しません。（値の中に埋め込まれた引用符を正しく取り扱えませんが、もし動作したとしたら安全ではありません。）

この機能の有効な利用方法の例は、ファイルの内容をテーブル列にコピーする場合も利用することができます。 その際は、ファイルをまず変数に読み込み、引用符付けした文字列として変数名を差し替えます。

```
testdb=> \set content `cat my_file.txt`
testdb=> INSERT INTO my_table VALUES (: 'content');
```

標準非標準の文字列を使用しているのであれば、さらにバックスラッシュを二重にしなければなりません。 以下のように多少ややこしくなります。

```
testdb=> \set content '``` `sed -e "s/\'/\'/g" -e "s/\\\'/\\\\\'/g" < my_file.txt` ```'
```

(my_file.txtにNULバイトが含まれている場合、これはうまく動作しないことに注意してください。 psqlは変数値内に埋め込まれたNULバイトをサポートしません。)

コロンの(:)もSQLコマンド内で正規に使用できますので、 指定した変数が現在設定されていない場合、差し替え時の見かけの置換(:name、:'name'、:"name")は行われません。 コロンのバックスラッシュでエスケープすれば、常に差し替えから保護する

ことができます。

変数用のコロンの構文は、ECPGのような組み込みの問い合わせ言語用の標準SQLとして規定されています。配列の一部の切り出し、および型キャスト用のコロンの構文はPostgreSQLの拡張であり、標準での使用方法と競合することがあります。SQLリテラルまたは識別子として変数の値をエスケープさせる引用符付きコロンの構文はpsqlの拡張です。

プロンプト

psqlが発行するプロンプトは好みに応じてカスタマイズできます。PROMPT1、PROMPT2、PROMPT3という3つの変数はプロンプトの表示内容を示す文字列や特別なエスケープシーケンスを持ちます。プロンプト1はpsqlが新しいコマンドを受け付ける際に発行される通常のプロンプトです。プロンプト2はコマンドがセミコロンで終わっていない、または、引用符が閉じていないためにさらにコマンドの入力が要求されている際に発行されます。プロンプト3はSQLのCOPYコマンドを実行している際、または、端末上で行の値の入力が要求されている際に発行されます。

選択されたプロンプト変数の値はそのまま文字として表示されます。ただし、パーセント（%）が含まれる場合は例外です。この場合は、次の文字に従って、特定のテキストに置換されます。置換対象として定義されているのは次のものです。

%M
データベースサーバの（ドメイン名付きの）完全なホスト名です。その接続がUnixドメインソケットの場合は[local]となります。ただし、Unixドメインソケットがコンパイル時に設定したデフォルトの場所に存在しない場合は、[local:/dir/name]となります。

%m
最初のドットで取り除いたデータベースサーバのホスト名です。その接続がUnixドメインソケットの場合は[local]となります。

%>
データベースサーバが監視するポート番号です。

%n
データベースセッションユーザの名前です（この値の展開結果は、SET SESSION AUTHORIZATIONコマンドの実行によってデータベースセッション中に変わることがあります）。

%/
接続中のデータベース名です。

%~
デフォルトデータベースの場合に~（チルダ）が出力される点を除いて、%/と同じです。

%#
セッションユーザがデータベーススーパーユーザである場合は#、それ以外の場合は>となります（この値の展開結果は、SET SESSION AUTHORIZATIONコマンドの実行によってデータベースセッション中に変わることがあります）。

%R
プロンプト1の場合、通常は=、シングル行モードでは^、データベースとの接続が切れたセッションでは!になります（¥connectが失敗した場合に発生します）。プロンプト2の場合、_、*、単一引用符、二重引用符、ドル記号に置き換えられます。どの文字に置き換えられるかは、psqlが入力を待っている理由（コマンドが終了していない、/* ... */によるコメント行の中にいる、引用符やエスケープされたドル記号の中にいる）によって決まります。プロンプト3の場合、何も表示されません。

%x
トランザクションの状態です。トランザクションブロックの外にいる場合は空文字列に、トランザクションブロックの中にいる場合は*に、失敗したトランザクションブロックの中にいる場合は!に、（接続されていないなど）トランザクションの状態が不定の場合は?になります。

%digits
指定8進数値コードの文字に置換されます。

%: name :
psqlのname変数の値です。詳細は[変数](#)を参照してください。

%`command`

通常の"逆引用符"による置き換えに似た、*command*の出力です。

```
%[ ... %]
```

プロンプトには端末制御文字を含めることができます。 具体的には、色の変更、背景、プロンプトテキストの様式、端末ウィンドウのタイトルなどが指定できます。 Readlineの行編集機能を適切に動作させるためには、印字されない制御文字を%[と%]で囲んで、不可視であることを明示しなければなりません。 この記号の組み合わせはプロンプト内に複数記述することができます。 以下に例を示します。

```
testdb=> \set PROMPT1 '%[033[1;33;40m%]n@%/%R%[033[0m%]%# '
```

これにより、VT100互換の配色可能な端末では、ボールドフォントで(1;)、黄色で点滅する(33;40) プロンプトが表示されます。

プロンプトにパーセント記号を入れる場合は、%%と記述してください。 デフォルトのプロンプトは、プロンプト1と2に'%/%R%# '、プロンプト3に'>> 'を設定した場合と同じです。

注意: この機能はtcshの機能を真似たものです。

コマンドライン編集

psqlは行内編集や繰り返し入力が簡便になるようにReadlineライブラリをサポートしています。 コマンド履歴は、psqlの終了時に自動的に保存され、psqlの起動時に再読み込みされます。 タブによる補間もサポートされていますが、SQLのパースとしてコマンドを解釈して判断してくれるわけではありません。 タブによる補間を何らかの事情により使用したくなければ、ホームディレクトリ以下の.inputrcというファイルに以下のように書き込むことで無効にできます。

```
$if psql
set disable-completion on
$endif
```

(これはpsqlの機能ではなく、Readlineの機能です。 詳細についてはReadlineのドキュメントを参照してください)。

環境

COLUMNS

\pset columnsがゼロの場合、wrapped書式の幅、および、幅出力がページャを必要とするかどうかを決める幅を制御します。 また自動拡張モードでは縦書式に切り替えるべきかどうかを制御します。

PAGER

問い合わせ結果が画面に入り切らない場合、このコマンドによって結果をパイプします。 一般的に指定される値は、more、またはlessです。 デフォルトはプラットフォームによって異なります。 ページャの使用を禁止するには\pset コマンドを使用します。

```
PGDATABASE
PGHOST
PGPORT
PGUSER
```

デフォルトの接続パラメータです ([項31.14](#)を参照)。

```
PSQL_EDITOR
EDITOR
VISUAL
```

\eおよび\efコマンドで使用されるエディタです。 変数はこのリスト順に検索されます。 つまり最初に設定されたものが使用されます。

組込みのデフォルトエディタは、Unixシステムではvi、Windowsシステムではnotepad.exeです。

PSQL_EDITOR_LINENUMBER_ARG

\eまたは\efが行番号引数を付けて使用された場合、この変数は、ユーザのエディタに開始行番号を渡すために使用されるコマンドライン引数を指定します。 Emacsまたはviのようなエディタでは、これはプラス(+)記号です。 オプション名と行番号の間に空白文字が必要な場合は、変数の値の最後に空白文字を含めてください。 以下に例を示します。

```
PSQL_EDITOR_LINENUMBER_ARG='+'
PSQL_EDITOR_LINENUMBER_ARG='--line '
```

Unixシステム上のデフォルトは+です。 (デフォルトのエディタviに対応するものですが、他のよく使われる多くのエ

ディタでも役に立ちます。) 一方Windowsシステムではデフォルトはありません。

PSQL_HISTORY

コマンド履歴ファイルの場所を指定します。 チルダ (~) 展開が行われます。

PSQLRC

ユーザの.psqlrcファイルの場所を指定します。 チルダ (~) 展開が行われます。

SHELL

¥! コマンドが実行するコマンドです。

TMPDIR

一時ファイルを格納するディレクトリです。 デフォルトは/tmpです。

また、このユーティリティは、他のほとんどのPostgreSQLユーティリティと同様、libpqでサポートされる環境変数を使用します ([項31.14](#)を参照してください)。

ファイル

- -xまたは-cオプションが渡されない限り、psqlは処理を始める前に、システム全体用のpsqlrcファイルとユーザ用の~/.psqlrcファイルのコマンドを読み込み、実行しようとします (Windowsにおける、ユーザ用の起動ファイルの名前は%APPDATA%\postgresql\psqlrc.confです)。 システム全体用のファイルの設定に関する情報についてはPREFIX/share/psqlrc.sampleを参照してください。 ¥setやSETコマンドを使用して、好みに応じたクライアントやサーバを設定するために使用することができます。

ユーザの~/.psqlrcファイルの場所もPSQLRC環境変数を介して明示的に設定することができます。

- システム全体用のpsqlrcファイルとユーザ用の~/.psqlrcファイルに、例えば、~/.psqlrc-9.2や~/.psqlrc-9.2.5のように、ハイフン記号とPostgreSQLのメジャーリリース番号またはpsqlのマイナーリリース番号を付与することで、特定バージョンのpsql向けのファイルとすることができます。 一致するバージョンのファイルはバージョン指定のないファイルよりも優先して読み込まれます。
- コマンドライン履歴は~/.psql_historyファイル、Windowsの場合は%APPDATA%\postgresql\psql_historyに格納されます。

履歴ファイルの場所もPSQL_HISTORY環境設定を介して明示的に設定することができます。

注釈

- 旧バージョンのpsqlでは、単一文字のバックスラッシュコマンドの後に、空白を入れずに直接最初の引数を入力することができました。 PostgreSQL 8.4からこれは許されなくなりました。
- psqlが円滑に動作することを保証できるのは、同一バージョンのサーバが対象の時のみです。 異なるバージョンの組み合わせではまったく動かないという意味ではありませんが、大小の様々な問題が発生する可能性があります。 特にバックスラッシュコマンドは、サーバがpsql自身のバージョンより新しいと失敗しやすくなります。 しかし、psql自身よりもサーバが新しくなければならないということなく、¥d系列のバックスラッシュコマンドは7.4までさかのぼるバージョンのサーバで動作するはずです。

Windowsユーザ向けの注意

psqlは"コンソールアプリケーション"としてコンパイルされます。 Windowsのコンソールウィンドウは、システムの他の部分とは異なる符号化方式を使用しているため、psqlで8ビット文字を使用する時には特別な配慮が必要です。 psqlは、コンソール用コードページとして問題があることを検出すると、起動時に警告を発します。 コンソール用コードページを変更するためには、以下の2つが必要です。

- cmd.exe /c chcp 1252と入力して、コードページを設定します (1252はドイツ圏における適切なコードページです。 システムに合わせて変更してください)。 Cygwinを使用しているのであれば、このコマンドを/etc/profileに追加してください。
- コンソール用フォントをLucida Consoleに設定してください。 ラスタフォントは、ANSIコードページでは正しく動作しないためです。

例

最初に、複数行にわたるコマンドの入力例を示します。 プロンプトの変化に注意してください。

```
testdb=> CREATE TABLE my_table (  
testdb(> first integer not null default 0,  
testdb(> second text)  
testdb-> ;  
CREATE TABLE
```

さて、ここでテーブル定義を再度確認してみます。

```
testdb=> \d my_table  
          Table "my_table"  
Attribute | Type      | Modifier  
-----+-----+-----  
first     | integer   | not null default 0  
second    | text      |
```

次に、プロンプトをもっと面白いものに変更してみます。

```
testdb=> \set PROMPT1 '%n%m %~%R%# '  
peter@localhost testdb=>
```

テーブルにデータを入力したものと考えてください。データを見る場合は次のようにします。

```
peter@localhost testdb=> SELECT * FROM my_table;  
first | second  
-----+-----  
1 | one  
2 | two  
3 | three  
4 | four  
(4 rows)
```

`\pset` コマンドを使って、このテーブルの表示を違うタイプに変更することができます。

```
peter@localhost testdb=> \pset border 2  
Border style is 2.  
peter@localhost testdb=> SELECT * FROM my_table;  
+-----+-----+  
| first | second |  
+-----+-----+  
| 1 | one |  
| 2 | two |  
| 3 | three |  
| 4 | four |  
+-----+-----+  
(4 rows)  
  
peter@localhost testdb=> \pset border 0  
Border style is 0.  
peter@localhost testdb=> SELECT * FROM my_table;  
first second  
-----  
1 one  
2 two  
3 three  
4 four  
(4 rows)  
  
peter@localhost testdb=> \pset border 1  
Border style is 1.  
peter@localhost testdb=> \pset format unaligned  
Output format is unaligned.  
peter@localhost testdb=> \pset fieldsep ", "  
Field separator is ", ".  
peter@localhost testdb=> \pset tuples_only  
Showing only tuples.  
peter@localhost testdb=> SELECT second, first FROM my_table;  
one,1  
two,2  
three,3  
four,4
```

その他の方法として、短縮されたコマンドを使用してみます。

```
peter@localhost testdb=> \a \t \x  
Output format is aligned.  
Tuples only is off.
```

```
Expanded display is on.
peter@localhost testdb=> SELECT * FROM my_table;
-[ RECORD 1 ]-
first  | 1
second | one
-[ RECORD 2 ]-
first  | 2
second | two
-[ RECORD 3 ]-
first  | 3
second | three
-[ RECORD 4 ]-
first  | 4
second | four
```

[前のページ](#)
pg_restore

[ホーム](#)
[上に戻る](#)

[次のページ](#)
reindexdb

reindexdb

名前

reindexdb -- PostgreSQLデータベースのインデックスを再作成します。

概要

```
reindexdb [connection-option...] [ --table | -t table] [ --index | -i index] [dbname]
```

```
reindexdb [connection-option...] --all | -a
```

```
reindexdb [connection-option...] --system | -s [dbname]
```

説明

reindexdbは、PostgreSQLデータベース内のインデックスを再作成するユーティリティです。

reindexdbは、[REINDEX](#) SQLコマンドのラッパです。 このユーティリティを使用しても他の方法でサーバにアクセスしても、データベースインデックスの再作成には実質的な違いはありません。

オプション

reindexdbは以下のコマンドライン引数を受け付けます。

```
-a  
--all
```

すべてのデータベースのインデックスを再作成します。

```
[-d] dbname  
[--dbname=] dbname
```

インデックス再作成を行うデータベースの名前を指定します。これが指定されず、かつ、`-a`（または`--all`）も使用されていない場合、`PGDATABASE`環境変数からデータベース名が決まります。これも設定されていない場合は、接続時に指定したユーザ名が使用されます。

```
-e  
--echo
```

reindexdbが生成しサーバに送信したコマンドを表示します。

```
-i index  
--index=index
```

*index*のみを再作成します。

```
-q  
--quiet
```

進行メッセージを表示しません。

```
-s  
--system
```

データベースのシステムカタログのインデックスを再作成します。

```
-t table  
--table=table
```

*table*のインデックスのみを再作成します。

```
-V  
--version
```

reindexdbのバージョンを表示し、終了します。

```
-?
--help
```

reindexdb コマンドライン引数の使用方法を表示し、終了します。

また、reindexdb は、接続パラメータとして以下のコマンドライン引数を受け付けます。

```
-h host
--host=host
```

サーバが稼働しているマシンのホスト名を指定します。ホスト名がスラッシュから始まる場合、Unix ドメインソケット用のディレクトリとして使用されます。

```
-p port
--port=port
```

サーバが接続を監視する TCP ポートもしくは Unix ドメインソケットファイルの拡張子を指定します。

```
-U username
--username=username
```

接続するユーザ名を指定します。

```
-w
--no-password
```

パスワードの入力を促しません。サーバがパスワード認証を必要とし、かつ、`.pgpass` ファイルなどの他の方法が利用できない場合、接続試行は失敗します。バッチジョブやパスワードを入力するユーザが存在しない場合にこのオプションは有用かもしれません。

```
-W
--password
```

データベースに接続する前に、reindexdb は強制的にパスワード入力を促します。

サーバがパスワード認証を要求する場合 reindexdb は自動的にパスワード入力を促しますので、これが重要になることはありません。しかし、reindexdb は、サーバにパスワードが必要かどうかを判断するための接続試行を無駄に行います。こうした余計な接続試行を防ぐために `-w` の入力がある場合もあります。

```
--maintenance-db=dbname
```

どのデータベースを再インデックス付けしなければならないかを見つけ出すために接続するデータベースの名前を指定します。指定されなければ `postgres` データベースが使用され、もし存在しなければ `template1` が使用されます。

環境

```
PGDATABASE
PGHOST
PGPORT
PGUSER
```

デフォルトの接続パラメータです。

また、このユーティリティは、他のほとんどの PostgreSQL ユーティリティと同様、libpq でサポートされる環境変数を使用します ([項31.14](#)を参照してください)。

診断

問題が発生した場合、考えられる原因とエラーメッセージについての説明は [REINDEX](#) と [psql](#) を参照してください。データベースサーバは、指定したホストで稼働している必要があります。また、libpq フロントエンドライブラリのデフォルトの設定や環境変数が適応されることに注意してください。

注釈

reindexdb の実行中には PostgreSQL サーバに何度も接続しなければならないことがありますが、この場合その度にパスワードが必要です。そのような場合は `~/.pgpass` ファイルを用意しておくとう便利です。詳細は [項31.15](#)を参照してください。

例

データベースtestのインデックスを再作成します。

```
$ reindexdb test
```

abcdというデータベース内のテーブルfooのインデックスとインデックスbarを再作成します。

```
$ reindexdb --table foo --index bar abcd
```

関連項目

[REINDEX](#)

[前のページ](#)
psql

[ホーム](#)
[上に戻る](#)

[次のページ](#)
vacuumdb

vacuumdb

名前

vacuumdb -- PostgreSQL データベースの不要領域の回収と解析を行う

概要

```
vacuumdb [connection-option...] [option...] [ --table | -t table [( column [,...] )] ] [dbname]
```

```
vacuumdb [connection-option...] [option...] --all | -a
```

説明

vacuumdbは、PostgreSQLデータベースの不要領域のクリーンアップを行うユーティリティです。また、vacuumdbはPostgreSQLの問い合わせオプティマイザが使用する内部的な統計情報も生成します。

vacuumdbは、SQLの**VACUUM**コマンドのラッパです。このユーティリティを使っても、これ以外の方法でサーバにアクセスしてバキュームや解析を行っても特に違いは生じません。

オプション

vacuumdbでは、下記のコマンドライン引数を指定できます。

```
-a
--all
```

全てのデータベースに対してバキュームを行います。

```
[-d] dbname
[--dbname=] dbname
```

不要領域のクリーンアップ、または、解析を行うデータベース名を指定します。データベース名が指定されておらず、`-a`（または`--all`）も指定されていない場合、データベース名は`PGDATABASE`環境変数から読み取られます。この変数も設定されていない場合は、接続時に指定したユーザ名が使用されます。

```
-e
--echo
```

vacuumdbが生成し、サーバに送るコマンドをエコー表示します。

```
-f
--full
```

"完全な (full) "クリーンアップを行います。

```
-F
--freeze
```

積極的にタプルを"凍結"します。

```
-q
--quiet
```

進行メッセージを表示しません。

```
-t table [(column [,...]) ]
--table=table [(column [,...]) ]
```

`table`のみをクリーンアップ/解析します。列名は`--analyze`や`--analyze-only`オプションがある場合にのみ設定できます。

ティップ: クリーンアップする列を指定する場合は、シェルで括弧をエスケープする必要があるでしょう（後述の例を参照してください）。

```
-v
```

--verbose

処理中に詳細な情報を表示します。

-V
--version

vacuumdbのバージョンを表示し、終了します。

-Z
--analyze

オプティマイザが使用する、データベースの統計情報も算出します。

-Z
--analyze-only

オプティマイザにより使用される統計情報の計算のみを行います（バキュームを行いません）。

-?
--help

vacuumdbのコマンドライン引数の使用方法を表示し、終了します。

vacuumdbには、以下に記す接続パラメータ用のコマンドライン引数も指定することもできます。

-h *host*
--host=*host*

サーバが稼働しているマシンのホスト名を指定します。ホスト名がスラッシュから始まる場合、Unixドメインソケット用のディレクトリとして使用されます。

-p *port*
--port=*port*

サーバが接続を監視するTCPポートもしくはUnixドメインソケットファイルの拡張子を指定します。

-U *username*
--username=*username*

接続するユーザ名を指定します。

-w
--no-password

パスワードの入力を促しません。サーバがパスワード認証を必要とし、かつ、.pgpassファイルなどの他の方法が利用できない場合、接続試行は失敗します。バッチジョブやパスワードを入力するユーザが存在しない場合にこのオプションは有用かもしれません。

-W
--password

データベースに接続する前に、vacuumdbは強制的にパスワード入力を促します。

サーバがパスワード認証を要求する場合vacuumdbは自動的にパスワード入力を促しますので、これが重要になることはありません。しかし、vacuumdbは、サーバにパスワードが必要かどうかを判断するための接続試行を無駄に行います。こうした余計な接続試行を防ぐために-wの入力が有意となる場合もあります。

--maintenance-db=*dbname*

どのデータベースをバキュームしなければならないかを見つけ出すために接続するデータベースの名前を指定します。指定されなければpostgresデータベースが使用され、もし存在しなければtemplate1が使用されます。

環境

PGDATABASE
PGHOST
PGPORT
PGUSER

デフォルトの接続パラメータです。

また、このユーティリティは、他のほとんどのPostgreSQLユーティリティと同様、libpqでサポートされる環境変数を使用します（[項31.14](#)を参照してください）。

診断

問題が発生した場合、考えられる原因とエラーメッセージについての説明は[VACUUM](#)と[psql](#)を参照してください。 データベースサーバは、指定したホストで稼働している必要があります。 また、libpqフロントエンドライブラリのデフォルトの設定や環境変数が適応されることに注意してください。

注釈

vacuumdbの実行中にはPostgreSQLサーバに何度も接続しなければならないことがあります。 この場合その度にパスワードが必要です。 そのような場合は`~/.pgpass`ファイルを用意しておくと便利です。 詳細は[項31.15](#)を参照してください。

例

`test`というデータベースをクリーンアップするには、下記のコマンドを実行します。

```
$ vacuumdb test
```

`bigdb`という名前のデータベースのクリーンアップとオプティマイザ用の解析を行う場合には、下記のコマンドを実行します。

```
$ vacuumdb --analyze bigdb
```

`xyzzzy`という名前のデータベースの`foo`という1つのテーブルだけの掃除と、そのテーブルの`bar`という1つの列にだけ対してオプティマイザ用の解析を行う場合には、下記のコマンドを実行します。

```
$ vacuumdb --analyze --verbose --table 'foo(bar)' xyzzzy
```

関連項目

[VACUUM](#)

[前のページ](#)
reindexdb

[ホーム](#)
[上に戻る](#)

[次のページ](#)
PostgreSQLサーバアプリケーション

III. PostgreSQL サーバアプリケーション

ここでは、PostgreSQL サーバアプリケーションとサポートユーティリティに関するリファレンス情報があります。これらのコマンドは通常データベースサーバが稼働しているホスト上でのみ実行されます。他のユーティリティプログラムの一覧は[参照II](#), [PostgreSQL クライアントアプリケーション](#)にあります。

目次

[initdb](#) -- PostgreSQL のデータベースクラスタを新しく作成する
[pg_controldata](#) -- PostgreSQL データベースクラスタの制御情報を表示する
[pg_ctl](#) -- PostgreSQL サーバの初期化、起動、停止、制御
[pg_resetxlog](#) -- PostgreSQL データベースクラスタの先行書き込みログやその他の制御情報を初期化する
[postgres](#) -- PostgreSQL データベースサーバ
[postmaster](#) -- PostgreSQL データベースサーバ

initdb

名前

initdb -- PostgreSQLのデータベースクラスタを新しく作成する

概要

initdb [*option...*] [--pgdata | -D] *directory*

説明

initdbはPostgreSQLのデータベースクラスタを新しく作成します。データベースクラスタとは、1つのサーバインスタンスで管理されるデータベースの集合です。

データベースクラスタの作成には、データベースのデータを保存するディレクトリの作成、共有カタログテーブル(特定のデータベースではなく、クラスタ全体に所属するテーブル)の生成、そしてtemplate1データベースとpostgresデータベースの作成といった作業が含まれます。後から新しいデータベースを作成する際は、template1データベースの全ての内容がコピーされます。(つまり、template1にインストールしたものすべてが、後に生成した各データベースに自動的にコピーされます。) postgresデータベースは、ユーザ、ユーティリティ、サードパーティ製アプリケーションのデフォルトデータベースとしての使用を意図したものです。

initdbは指定されたデータディレクトリを作成しようと試みますが、多くの場合、そのデータディレクトリの親ディレクトリの所有者はrootなので、権限がないことがあります。このような場合、まず、空のデータディレクトリをrootで作成し、chownを使ってそのディレクトリの所有権限をデータベースのユーザに変えてください。次にsuを使ってデータベースユーザとなります。最後にデータベースユーザとしてinitdbを実行します。

initdbは、サーバプロセスの所有者によって実行されなければなりません。initdbによって作成されるファイルやディレクトリにサーバがアクセスする必要があるからです。サーバをrootとして実行することはできませんので、rootでinitdbを実行してはいけません(実際には、実行しようとしても拒否されます)。

initdbは、データベースクラスタのデフォルトのロケールと文字セット符号化方式を初期化します。文字セット符号化方式、照合順(LC_COLLATE)および文字セットクラス(LC_CTYPE、例えば、大文字、小文字、数字)は、データベース作成時、別々に設定することができます。initdbはtemplate1データベース用のこれらの値を決定します。この値が他のすべてのデータベースのデフォルトとして提供されます。

デフォルトの照合順や文字セットクラスを変更するためには、--lc-collateおよび--lc-ctypeオプションを使用してください。CやPOSIX以外の照合順には性能の劣化が伴います。これらの理由によりinitdb実行時のロケールを正しく選択することが重要です。

残りのロケール関連のものはサーバ起動時に変更することができます。--localeを使用して、照合順や文字セットクラスを含む、すべてのロケール関連のもののデフォルトを設定することも可能です。SHOW ALLを使用して、すべてのサーバのロケール値(lc_*)を表示することができます。詳細は[項22.1](#)に記載しています。

デフォルトの符号化方式を変更するには、--encodingオプションを使用します。詳細は[項22.3](#)に記載しています。

オプション

-A *authmethod*
--auth=*authmethod*

このオプションは、pg_hba.confにてローカルユーザ用に使用される認証方法(host行とlocal行)を指定します。システムのすべてのローカルユーザが信頼できるわけではない場合は、trustを使用しないでください。インストールを簡単にするためにtrustがデフォルトです。

--auth-host=*authmethod*

このオプションはpg_hba.confにてTCP/IP接続経由のローカルユーザ用に使用される認証方法(host行)を指定します。

--auth-local=*authmethod*

このオプションはpg_hba.confにてUnixドメインソケット接続経由のローカルユーザ用に使用される認証方法 (local行) を指定します。

```
-D directory
--pgdata=directory
```

このオプションは、データベースクラスタを格納すべきディレクトリを指定します。initdbが必要とする情報はこれだけですが、PGDATA環境変数を設定しておけば、このオプションで指定する手間が省けます。この方法は、後に同じ変数を使用してデータベースサーバ (postgres) がデータベースディレクトリを参照できるので、効率的です。

```
-E encoding
--encoding=encoding
```

テンプレートデータベースの符号化方式を選択します。特に上書きされない限り、これが今後作成される全てのデータベースのデフォルト符号化方式となります。デフォルトは、ロケールによって決まります。ロケールが機能していない場合は、SQL_ASCIIになります。PostgreSQLサーバによってサポートされる文字セットについては[項22.3.1](#)で説明します。

```
--locale=locale
```

データベースクラスタ用のデフォルトのロケールを設定します。このオプションを指定しない場合は、initdbを実行している環境のロケールが継承されます。ロケールサポートについては[項22.1](#)で説明します。

```
--lc-collate=locale
--lc-ctype=locale
--lc-messages=locale
--lc-monetary=locale
--lc-numeric=locale
--lc-time=locale
```

--localeと似ていますが、指定したカテゴリのロケールのみを設定します。

```
--no-locale
```

--locale=Cと同じです。

```
--pwfile=filename
```

initdbはデータベーススーパーユーザのパスワードをこのファイルから読み取ります。このファイルの最初の行がパスワードとして解釈されます。

```
-T CFG
--text-search-config=CFG
```

デフォルトの全文検索設定を設定します。詳細については[default_text_search_config](#)を参照してください。

```
-U username
--username=username
```

データベースのスーパーユーザのユーザ名を選択します。initdbを実行している実効ユーザの名前がデフォルトです。スーパーユーザの名前はあまり重要ではありませんが、慣習的に使われているpostgresを使うことを (オペレーティングシステムのユーザ名と異なっても) お勧めします。

```
-W
--pwprompt
```

initdbにデータベースのスーパーユーザ権限を与えるためのパスワード入力のプロンプトを表示させます。パスワード認証を行うつもりがない場合は必要ありません。このオプションを指定しても、パスワードの設定を行わない限りパスワード認証は行われません。

```
-X directory
--xlogdir=directory
```

このオプションはトランザクションログの格納ディレクトリを指定します。

この他にも、使用頻度は下がりますが、下記のオプションが使用可能です。

```
-d
--debug
```

ブートストラップバックエンドからのデバッグ情報と、一般の利用者にはおそらく不要なその他の情報を出力します。ブートストラップバックエンドとはinitdbがカタログテーブルを作成する際に使用するプログラムです。このオプションはうんざりするようなログを大量に出力します。

```
-l directory
```

データベースクラスタを初期化する際に、initdbが参照すべき入力ファイルを指定します。これは通常必要ありません。明示的に指定する必要がある場合は、その時に指定するよう要求されます。

```
-n
--noclean
```

デフォルトでは、`initdb`を実行中にエラーが発生し、データベースクラスタを完成できなかった場合に、そのエラーが発生する前に作成された全てのファイルを削除します。このオプションを指定すると、これらのファイルが削除しないで残されるので、デバッグの際にはとても便利です。

その他のオプションを以下に示します。

```
-V
--version
```

`initdb`のバージョンを表示し、終了します。

```
-?
--help
```

`initdb`のコマンドライン引数の使用方法を表示し、終了します。

環境

PGDATA

データベースクラスタを保存するディレクトリを指定します。 `-D` オプションを使用して上書きすることができます。

また、このユーティリティは、他のほとんどのPostgreSQLユーティリティと同様、`libpq`でサポートされる環境変数を使用します（[項31.14](#)を参照してください）。

注釈

`initdb`は`pg_ctl initdb`経由でも呼び出すことができます。

関連項目

[pg_ctl](#), [postgres](#)

[前のページ](#)

PostgreSQLサーバアプリケーション

[ホーム](#)
[上に戻る](#)

[次のページ](#)

`pg_controldata`

pg_controldata

名前

pg_controldata -- PostgreSQL データベース クラスターの制御情報を表示する

概要

pg_controldata [*option*] [*datadir*]

説明

pg_controldata はカタログのバージョンなど `initdb` の際に初期化された情報を表示します。また、WAL（ログ先行書き込み）およびチェックポイント処理に関する情報も表示します。この情報はクラスター全体に関するものであり、特定のデータベースに関するものではありません。

このユーティリティの実行にはデータディレクトリへの読み取りアクセス権限が必要となるため、クラスターを初期化したユーザのみが実行できます。データディレクトリは、コマンドラインや `PGDATA` 環境変数を使用して指定することもできます。このユーティリティは、pg_controldata のバージョンを表示し終了する `-v` および `--version` オプションをサポートします。またサポートされる引数を入力する `-?` および `--help` オプションもサポートします。

環境

PGDATA

デフォルトのデータディレクトリの場所です。

pg_ctl

名前

pg_ctl -- PostgreSQLサーバの初期化、起動、停止、制御

概要

```
pg_ctl init[db] [-s] [-D datadir] [-o initdb-options]
```

```
pg_ctl start [-w] [-t seconds] [-s] [-D datadir] [-l filename] [-o options] [-p path] [-c]
```

```
pg_ctl stop [-W] [-t seconds] [-s] [-D datadir] [-m s[mart] | f[ast] | i[mmediate] ]
```

```
pg_ctl restart [-w] [-t seconds] [-s] [-D datadir] [-c] [-m s[mart] | f[ast] | i[mmediate] ] [-o options]
```

```
pg_ctl reload [-s] [-D datadir]
```

```
pg_ctl status [-D datadir]
```

```
pg_ctl promote [-s] [-D datadir]
```

```
pg_ctl kill signal_name process_id
```

```
pg_ctl register [-N servicename] [-U username] [-P password] [-D datadir] [-S a[uto] | d[emand] ] [-w] [-t seconds] [-s] [-o options]
```

```
pg_ctl unregister [-N servicename]
```

説明

pg_ctlはPostgreSQLデータベースクラスタの初期化、PostgreSQLのデータベースサーバ ([postgres](#)) を起動、停止、再起動する、あるいは稼働中のサーバの状態を表示するためのユーティリティです。サーバは手動で起動することも可能ですが、pg_ctlは、ログ出力のリダイレクトや、ターミナルとプロセスグループの適切な分離などの作業をカプセル化してくれます。さらにシャットダウン制御のための便利なオプションも提供します。

initまたはinitdbモードは新しいPostgreSQLデータベースクラスタを作成します。データベースクラスタとは、単一のサーバインスタンスで管理されるデータベースの集合です。このモードはinitdbコマンドを呼び出します。 [initdb](#)を参照して下さい。

startモードでは、新しいサーバが立ち上がります。サーバはバックグラウンドで起動され、その標準入力 は/dev/null (Windowsの場合はnul) に接続されます。Unix互換のシステムではデフォルトで、サーバの標準出力と標準エラーはpg_ctlの標準出力に転送されます。pg_ctlの標準出力はファイルにリダイレクトするか、例えばrotatelogなどのログ回転プログラムのような別プロセスにパイプで渡すべきです。こうしないと、postgresはその出力を（バックグラウンドから）制御端末に書き出しますので、シェルのプロセスグループから切り離すことができません。Windowsではデフォルトで、サーバの標準出力と標準エラーは端末に送信されます。こうしたデフォルトの動作は-lを用いてサーバの出力をログファイルに追加するように変更することができます。-lまたは出力のリダイレクトのどちらかを使用することを勧めます。

stopモードは、指定されたデータディレクトリで稼働しているサーバを停止（シャットダウン）します。-mオプションでは、3つの異なる停止方式を選択できます。"Smart"モード（デフォルト）は、全てのクライアントが切断し、オンラインバックアップがあればそれが終了するまで待ちます。サーバがホットスタンバイ状態の場合、すべてのクライアント接続が切断された後にリカバリとストリーミングレプリケーションは終了します。"Fast"モードはクライアントが切断するまで待たず、かつ、実行中のオンラインバックアップを終了させます。全ての実行中のトランザクションをロールバックし、クライアントとの接続を強制的に切断した後、サーバを停止します。"Immediate"モードは、クリーンアップ処理なしで、全てのサーバプロセスを即座に中断します。これを指定すると、再起動時にクラッシュリカバリ処理が実行されます。

restartモードは、効果的にサーバを停止して、その後、再起動します。この時、postgresコマンドラインオプションを変更することができます。

reloadモードは、単にpostmasterにSIGHUPシグナルを送り、(postgresql.conf、pg_hba.confなどの) 設定ファイルを再読み込みを実行させます。これにより、完全な再起動をすることなく、設定ファイルのオプションの変更を反映させることができます。

`status` モードは指定したデータディレクトリでサーバが起動しているかどうかを確認します。 起動している場合はPIDと、サーバを起動する時に使われたコマンドラインオプションを表示します。 サーバが稼働していない場合、このプロセスは終了ステータス3を返します。

`promote` モードでは、指定したデータディレクトリで稼働中のスタンバイサーバにリカバリを終了し読み書き操作を始めるようコマンドを送ります。

`kill` モードでは指定したプロセスにシグナルを送信することができます。 これは特に、`kill` コマンドを持たないMicrosoft Windowsで有効です。 サポートするシグナル名の一覧は`--help`を使用して確認してください。

`register` モードにより、Microsoft Windows上でシステムサービスを登録することができます。 `-s` オプションによりサービスの起動種類を`"auto"`（システム起動時に自動的にサービスを起動）か`"demand"`（要望が合った時にサービスを起動）から選択することができます。

`unregister` によりMicrosoft Windows上のシステムサービスを登録解除することができます。 これは過去に`register` コマンドによりなされた変更を元に戻します。

オプション

`-c`
`--core-file`

実現可能なプラットフォームにおいて、サーバクラッシュ時にcoreファイルを生成できるように関連するソフトリソース制限を上げます。 障害が起きたサーバプロセスから得られたスタックトレースを行うことができますので、問題のデバッグや診断の際に有用です。

`-D datadir`
`--pgdata datadir`

データベースファイルのファイルシステム上の場所を指定します。 これが省略された場合、`PGDATA`環境変数が使われません。

`-l filename`
`--log filename`

サーバログ出力を`filename`に追加します。 そのファイルが存在しない場合は作成されます。 `umask`は077に設定されますので、他のユーザからのログファイルへのアクセスはデフォルトでは許可されません。

`-m mode`
`--mode mode`

停止（シャットダウン）モードを指定します。 `mode`は`smart`、`fast`、`immediate`、もしくはこの3つのモード名の最初の1文字をとることができます。 省略時は`smart`が使用されます。

`-o options`

直接`postgres`に渡されるオプションを指定します。

このオプションは、通常、1グループとして渡されることを保証するために、単一引用符または二重引用符で囲まなければなりません。

`-o initdb-options`

`initdb` コマンドに直接渡すオプションを指定します。

このオプションは通常、ひとまとめとして渡すために確実に、単一引用符または二重引用符で囲まなければなりません。

`-p path`

`postgres`の実行プログラムの位置を指定します。 デフォルトでは`postgres`の実行プログラムは`pg_ctl`と同じディレクトリにあるものと仮定されます。 また、このディレクトリに存在しなければ、構築時に指定したインストールディレクトリが使われます。 このオプションは、何か異例なことをしていて`postgres`が見つからないというようなエラーが出ない限り、使う必要はありません。

`init` モードでは、このオプションは同様に`initdb`実行プログラムの場所を指定します。

`-s`
`--silent`

エラーメッセージのみを表示し、その他の情報を表示しません。

`-t`

--timeout

起動または停止の待機が完了するまでの最大の秒数。 デフォルトは60秒です。

-V
--version

pg_ctlのバージョンを表示し、終了します。

-w

起動もしくは停止処理が完了するのを待ちます。 待機は停止処理ではデフォルトのオプションですが、起動処理ではデフォルトではありません。 起動処理を待機している場合、pg_ctlは繰り返しサーバへの接続を試行します。 停止処理を待機している場合、pg_ctlはサーバがPIDファイルを削除するまで待機します。 pg_ctlは、起動もしくは停止が成功したかどうかに基づいて正しい終了コードを返します。

-W

起動処理もしくは停止処理が完了するのを待ちません。 起動と再起動モードのデフォルトです。

-?
--help

pg_ctlコマンドライン引数の説明を表示し、終了します。

Windows用オプション

-N *servicename*

登録するシステムサービスの名前です。 この名前は、システム名としても表示名としても使用されます。

-P *password*

サービスを開始するユーザ用のパスワードです。

-S *start-type*

登録するシステムサービスの起動種類です。 気同種類はauto、demand、またはこれら2単語の先頭の文字のいずれかです。 省略時はautoが使用されます。

-U *username*

サービスを起動するユーザのユーザ名です。 ドメインユーザの場合はDOMAIN¥username 書式を使用してください。

環境

PGDATA

デフォルトのデータ保存場所です。

また、pg_ctlは、他のほとんどのPostgreSQLユーティリティと同様、libpqでサポートされる環境変数を使用します（[項31.14](#)を参照してください）。 この他のサーバ変数については[postgres](#)を参照してください。

ファイル

postmaster.pid

データディレクトリにこのファイルがあるかどうかによって、pg_ctlはサーバが現在稼働中かどうかを決定します。

postmaster.opts

このファイルがデータディレクトリにあれば、pg_ctl（のrestartモード）は、-oで上書きされるものを除き、このファイルの内容をオプションとしてpostgresに渡します。 また、このファイルの内容がstatusモードで表示されます。

例

サーバの起動

サーバを起動します。

```
$ pg_ctl start
```

以下はサーバが接続を植え付けられるようになるまで待機する起動例です。

```
$ pg_ctl -w start
```

ポート5433を使い`fsync`なしでサーバを起動します。

```
$ pg_ctl -o "-F -p 5433" start
```

サーバの停止

サーバを停止するためには以下を使用します。

```
$ pg_ctl stop
```

`-m`オプションによりどのようにバックエンドを停止させるかを制御できます。

```
$ pg_ctl stop -m fast
```

サーバの再起動

サーバの再起動は、サーバを停止してもう一度起動するのとはほぼ同じですが、違うのは、`pg_ctl`が以前起動していたインスタンスに渡されていたコマンドラインオプションを保存し再利用することです。 下記はサーバを再起動する最も簡単な方法です。

```
$ pg_ctl restart
```

再起動の際にサーバのシャットダウンを待ってから起動を行う方法です。

```
$ pg_ctl -w restart
```

以下はポート5433を使って再起動し、再起動後に`fsync`を無効にする方法です。

```
$ pg_ctl -o "-F -p 5433" restart
```

サーバの状態表示

下記は`pg_ctl`からの状態の出力の例です。

```
$ pg_ctl status
pg_ctl: server is running (PID: 13718)
/usr/local/pgsql/bin/postgres "-D" "/usr/local/pgsql/data" "-p" "5433" "-B" "128"
```

これは再起動（restart）モードで呼び出されるコマンドラインです。

関連項目

[initdb](#), [postgres](#)

[前のページ](#)
`pg_controldata`

[ホーム](#)
[上に戻る](#)

[次のページ](#)
`pg_resetxlog`

pg_resetxlog

名前

pg_resetxlog -- PostgreSQLデータベースクラスタの先行書き込みログやその他の制御情報を初期化する

概要

```
pg_resetxlog [-f] [-n] [-o oid] [-x xid] [-e xid_epoch] [-m mxid] [-O mxoff] [-l timelineid,fileid,seg] datadir
```

説明

pg_resetxlogは、先行書き込みログ（WAL）を消去し、さらにオプションでpg_controlファイル内に保存された制御情報の一部を初期化します。この機能は、これらのファイルが破損した場合に必要なことがあります。このような破損などが原因でサーバを起動できない場合の最後の手段としてのみ、この機能を使用してください。

このコマンドを実行すると、サーバが開始できるようになるはずですが、ただし、不完全にコミットされたトランザクションが原因でデータベースのデータに矛盾が起こる可能性があることに注意してください。コマンドの実行後は、データをただちにダンプし、initdbを実行し、リロードすべきです。リロード後、矛盾がないか検査し、必要に応じて修復を行ってください。

このユーティリティの実行にはデータディレクトリへの読み込み/書き込みアクセス権限が必要となるため、サーバをインストールしたユーザのみが実行できます。安全のため、データディレクトリをコマンドラインで指定する必要があります。

pg_resetxlogは、PGDATA環境変数を使用しません。

pg_resetxlogがpg_controlに対する有効なデータを判別できない場合、-f（強制）スイッチを指定すれば強制的に処理を進めることができます。その場合、欠落したデータは無難な値で代用されます。ほとんどフィールドでは適切な値が使用されますが、次のOID、次のトランザクションIDとエポック時間、マルチトランザクションIDとそのオフセット、WAL開始アドレスの値については、手動の操作が必要な場合があります。これらの値は下記で説明するオプションを使用して設定することができます。すべてに対して正しい値を決定できない場合にも-fを使用することができますが、この場合は回復したデータベースを通常よりさらに注意深く検査する必要があります。必ず、ただちにダンプおよびリロードを行ってください。決して、ダンプを行う前にデータ変更などの操作を行ってはなりません。これを行うと、破損状態がさらに悪化します。

-o、-x、-e、-m、-O、-lの各オプションを使用すると、次のOID、次のトランザクションID、次のトランザクションIDのエポック時間、次のマルチトランザクションID、次のマルチトランザクションのオフセット、WAL開始アドレスの値を手動で設定することができます。この操作は、pg_resetxlogがpg_controlの読み込みによって適切な値を判別できない場合にのみ必要です。安全な値を得るには、以下のようにします。

- 次のトランザクションID（-x）用の安全な値は、データディレクトリ以下のpg_clogディレクトリ内で最も大きな数字を持つファイル名を探すことで決定することができます。その値に1を加え、1048576で乗算します。ファイル名は16進数ですので、通常、オプションの値も16進数で指定した方が簡単です。例えば、0011がpg_clogで最も大きなエントリであれば、-x 0x12000000となります（後ろにゼロを5つ付けると、1048576で乗算したことになります）。
- 次のマルチトランザクションID（-m）用の安全な値は、データディレクトリ以下のpg_multixact/offsetsディレクトリ内で最も大きな数字を持つファイル名を探すことで決定することができます。その値に1を加え、65536で乗算します。上と同様、ファイル名は16進数ですので、通常、オプションの値も16進数で指定した方が簡単です。ここでは、4個のゼロを付けてください。
- 次のマルチトランザクションオフセット（-O）用の安全な値は、データディレクトリ以下のpg_multixact/membersディレクトリ内で最も大きな数字を持つファイル名を探すことで決定することができます。その値に1を加え、65536で乗算します。上と同様、ファイル名は16進数ですので、通常、オプションの値も16進数で指定した方が簡単です。ここでは、4個のゼロを付けてください。
- WAL開始アドレス（-l）は、データディレクトリ以下のpg_xlogに現在存在するWALセグメントファイル名のどの番号よりも大きくなければなりません。この名前も16進数で、3つの部分に分かれています。最初の部分は"時系列ID"で、通常、この値は変更すべきではありません。3番目の部分には255（0xFF）より大きい値を付けてはなりません。255を超える時は、2番目の部分に1を加え、3番目の部分を0に戻してください。例えば、pg_xlog内で最大のエントリが00000000100000003200000004Aである場合は、-l 0x1,0x32,0x4Bと指定すれば問題なく動作します。しかし、最大のエントリが00000000100000003A000000FFの場合は、-l 0x1,0x3B,0x0、または、これ以上の値を指定してください。

注意: pg_resetxlog自体はpg_xlog内のファイルを参照し、既存のファイル名の最後を超えた値をデフォルト

トの`-l`設定として選択します。したがって、手作業による`-l`の調整は、オフラインアーカイブ内の項目など`pg_xlog`に現存しないWALセグメントファイルに注意する場合、または、`pg_xlog`の内容が完全に失われている場合にのみ必要とされます。

- データベース内のOIDの最大値よりも大きな次のOIDを決定するには、上記のような簡単な方法はありません。しかし、幸いにも、次のOIDが正しく設定されているかどうかは、それほど重要ではありません。
- `pg_resetxlog`で設定されるフィールドを除き、トランザクションIDのエポック時間は実際にはデータベース内に格納されません。そのため、データベース自身が問題としない限り、任意の値で動作するでしょう。Slony-IなどのリPLICATIONシステムが確実に正しく動作するように、この値を調整しなければならない可能性があります。その場合、適切な値はリPLICATIONシステムの下流で複製されたデータベースの状態から得られるはずです。

`-n`（操作なし）スイッチを指定すると、`pg_resetxlog`は`pg_control`から再構築した値を出力して、何も変更せずに終了します。これは主にデバッグと目的としたツールですが、`pg_resetxlog`を実際に進める前の検査としても有用な場合があります。

`-v`および`--version`オプションは`pg_resetxlog`のバージョンを表示し、終了します。`-?`および`--help`オプションはサポートする引数を表示し、終了します。

注釈

このコマンドは、サーバの稼動中に使用してはいけません。`pg_resetxlog`は、データディレクトリにサーバのロックファイルがあると、実行されません。サーバがクラッシュした場合、ロックファイルがそのまま残される場合があります。その場合は、ロックファイルを削除すれば`pg_resetxlog`を実行することができます。ただし、このコマンドを実行する時は、起動中のサーバプロセスが一切ないことを慎重に確認してください。

[前のページ](#)
pg_ctl

[ホーム](#)
[上に戻る](#)

[次のページ](#)
postgres

postgres

名前

postgres -- PostgreSQLデータベースサーバ

概要

postgres [*option...*]

説明

postgresは、PostgreSQLのデータベースサーバです。クライアントアプリケーションがデータベースに接続するためには、稼働中のpostgresインスタンスに（ネットワークを介して、またはローカルで）接続する必要があります。その後、postgresは接続を取り扱うために別のサーバプロセスを開始します。

1つのpostgresインスタンスは常に1つのデータベースクラスタのデータを管理します。データベースクラスタとは、共通のファイルシステム領域（"データ領域"）に格納されているデータベースの集まりのことです。1つのシステム上で、同時に複数のpostmasterインスタンスを実行することは可能ですが、それらは異なるデータ領域と、異なる接続ポート（下記参照）を使用する必要があります。postgresは、その起動時にデータ領域の場所を知らなければなりません。その場所を-DオプションまたはPGDATA環境変数によって指定しなければなりません。デフォルト値はありません。通常、-DまたはPGDATAは、[initdb](#)で作成されたデータ領域ディレクトリを直接指し示します。他のファイルレイアウトを取ることもできます。[項18.2](#)を参照してください。

デフォルトでは、postgresはフォアグラウンドで起動し、ログメッセージを標準エラー出力に出力します。実運用では、postgresアプリケーションはおそらくマシン起動時にバックグラウンドで起動されるはずです。

またpostgresはシングルユーザモードで呼び出すこともできます。このモードは、主に[initdb](#)による初期化処理の中で使用されます。これを、デバッグのためや災害からの復旧時に使用することもできます。実際には発生するはずのプロセス間通信やロック処理が発生しませんので、シングルユーザモードがサーバのデバッグに確実に適したものではないことに注意してください。シェルからシングルユーザモードで呼び出された場合、ユーザは問い合わせを入力できます。結果は画面上に表示されますが、その形式はエンドユーザ向けではなく開発者向けのものです。シングルユーザモードでは、セッションユーザはユーザID 1と設定され、暗黙的にスーパーユーザの能力がこのユーザに与えられます。このユーザは実際に存在していなくても構いません。このため、シングルユーザモードはシステムカタログの偶発的な損傷などから手動で復旧するために使用することができます。

オプション

postgresには、下記のコマンドライン引数を指定できます。オプションに関する詳細は[第18章](#)を参照してください。また、設定ファイルを使用することによって、これらを入力する手間を省くことができます。一部の（安全な）オプションは接続中のクライアントから設定することもできます。その設定方法はアプリケーションに依存し、また、そのセッションでのみ適用されます。たとえば、PGOPTIONS環境変数を設定すると、libpqを基にしたクライアントはその文字列をサーバに渡します。渡された文字列はpostgresコマンドラインオプションとして扱われます。

一般的な目的

-A 0|1

実行時にアサーション検査を行い、プログラムの不具合を検知するデバッグを補助します。このオプションは、PostgreSQLのコンパイル時にアサーションが有効にされた場合にのみ、使用可能となります。この場合、デフォルトで有効になります。

-B *nbuffers*

サーバプロセスが使用する共有バッファの数を設定します。このパラメータのデフォルト値はinitdbにより自動的に決まります。このオプション指定は[shared_buffers](#)設定パラメータの設定と同じです。

-c *name=value*

名前付き実行時パラメータを設定します。 PostgreSQLでサポートされる設定パラメータは[第18章](#)で説明します。 実際には、他の多くのコマンドラインオプションはこのようなパラメータへの代入を簡略化したものです。 `-c`は複数のパラメータを設定するために複数回使用することができます。

`-C name`

指定された名前の実行時パラメータの値を表示し、終了します、（詳細については[-c](#)を参照してください。） これは実行中のサーバに対して使用することができ、`postgresql.conf`からの値やその起動時に与えられたパラメータにより変更された値を返します。 クラスタが起動した後に与えられたパラメータは反映されません。

このオプションは、設定パラメータ値を問い合わせる[pg_ctl](#)など、サーバインスタンスと連携する他のプログラム向けを意図しています。 ユーザ向けのアプリケーションでは代わりに[SHOW](#)や[pg_settings](#)ビューなどを使用すべきです。

`-d debug-level`

デバッグレベルを設定します。 大きな値が設定されているほど、より多くのデバッグ情報がサーバのログに出力されます。 値として設定できる範囲は、1から5までです。 特定のセッションで[-d 0](#)を渡すこともできます。 この設定により、親の[postgres](#)プロセスのサーバログレベルがこのセッションに伝播されません。

`-D datadir`

ファイルシステム上のデータディレクトリの場所や設定ファイルの場所を指定します。 詳細は[項18.2](#)を参照してください。

`-e`

デフォルトの日付データ形式を"ヨーロッパ式"にします。これにより、日付の入力時の順番はDMYになります。 また、日付の出力時にも、月の前に日にちを置いた書式で表示されるようになります。 詳細は[項8.5](#)を参照してください。

`-F`

性能向上のために[fsync](#)呼び出しを無効とします。 ただしこの場合、システムクラッシュ時にデータが損傷する危険性があります。 このオプションの指定は[fsync](#)設定パラメータを無効にすることと同一です。 このオプションを使用する時は、事前に詳細な文書を一読してください。

`-h hostname`

[postgres](#)がクライアントアプリケーションとの接続を監視するIPホスト名、または、IPアドレスを指定します。 この値には、カンマで区切ったアドレスの一覧だけでなく、全ての利用可能なインタフェースを監視することを意味する*を指定することができます。 空の値を指定すると、IPアドレスをまったく監視しないことを意味します。 この場合、サーバへの接続には、Unixドメインソケットのみが使用されます。 デフォルトではlocalhostのみを監視します。 このオプション指定は[listen_addresses](#)設定パラメータを設定することと同一です。

`-i`

リモートクライアントからのTCP/IP（インターネットドメイン）経由の接続を可能とします。 このオプションを設定しない場合には、ローカルからの接続のみが使用可能となります。 このオプションは、`postgresql.conf`内の[listen_addresses](#)を、あるいは[-h](#)を*に設定することと同じ意味を持ちます。

このオプションは[listen_addresses](#)の全ての機能を実現することができないため廃止が予定されています。 通常は、直接[listen_addresses](#)を設定する方法をお勧めします。

`-k directory`

[postgres](#)がクライアントアプリケーションからの接続を監視するUnixドメインソケットのディレクトリを指定します。 デフォルトでは、通常/tmpとなっていますが、これは構築時に変更できます。

`-l`

SSLを使用して、安全な接続を行います。 このオプションを使用するためには、PostgreSQLのコンパイル時にSSLを有効にする必要があります。 SSLの使用に関する詳細は[項17.9](#)を参照してください。

`-N max-connections`

このサーバが受け付けるクライアント接続数の最大値を設定します。 この値のデフォルト値はinitdbにより自動的に選択されます。 このオプションの指定は[max_connections](#)設定パラメータを設定することと同じ意味を持ちます。

`-o extra-options`

`extra-options`で指定されたコマンドラインオプションは、この[postgres](#)プロセスにより開始された全てのサーバプロセスに引き渡されます。 このオプションの文字列に空白文字が含まれている場合には、文字列全体を引用符で囲む必要があります。

このオプションの使用は非推奨です。 サーバプロセス用のすべてのコマンドラインオプションはpostgresコマンドラインで直接指定することができます。

-p *port*

クライアントアプリケーションからの接続をpostgresが監視するTCP/IPポート、またはローカルUnixドメインソケットファイルの拡張子を指定します。 デフォルトでは、PGPORT環境変数、PGPORTが設定されていない場合はコンパイル時に設定された値（通常5432）が使用されます。 デフォルトのポート以外を指定した場合には、コマンドラインオプション、またはPGPORTを使用して、全てのクライアントアプリケーションが同じポートを指定するようにしなければなりません。

-s

各コマンドの終了時に時間情報や他の統計情報を表示します。 これはベンチマークやバッファ数の調整時の使用に適しています。

-s *work-mem*

一時ディスクファイルで再ソート処理を行う前に内部的なソート処理やハッシュ処理で使用するメモリ量を指定します。 [項18.4.1](#)のwork_mem設定パラメータの説明を参照してください。

-V
--version

postgresのバージョンを表示し、終了します。

--name=*value*

名前付き実行時パラメータを設定します。 -cの省略形式です。

--describe-config

このオプションは、サーバの内部設定変数、その説明、および、デフォルト値をタブ区切りのCOPY書式でダンプ出力します。 これは、管理用ツールからの使用を主目的として設計されました。

-?
--help

postgresのコマンドライン引数に関する説明を表示し、終了します。

開発者向けオプション

ここで説明するオプションは、主にデバッグ用に、まれに深刻な障害を受けたデータベースの復旧を補助するために使われます。 これらを運用状態のデータベースの設定として使用する理由はないはずです。 これらのオプションを列挙しておくのは、PostgreSQLシステム開発者のためです。 さらに、これらのオプションは、いずれも将来のリリースで予告なしに変更・廃止される可能性があります。

-f { *s* | *i* | *o* | *b* | *t* | *n* | *m* | *h* }

特定のスキャンと結合方式の使用を禁止します。 *s*はシーケンシャルスキャン、*i*はインデックススキャンを無効にします。 *o*、*b*、*t*はそれぞれ、インデックスオンリースキャン、ビットマップインデックススキャン、TIDスキャンを無効にします。 *n*はネストループ結合、*m*はマージ結合、*h*はハッシュ結合を無効にします。

シーケンシャルスキャンとネストループ結合は、完全に無効にすることはできません。 -fsオプションと-fnオプションは、単に「他の選択肢があるならば、これらの種類の計画を使わないようにする」ということをオプティマイザに指示するだけです。

-n

これはサーバプロセスが異常終了する問題をデバッグするためのオプションです。 このような状態では、他のすべてのサーバプロセスに対し、終了し、共有メモリやセマフォを再初期化するように通知するのが、通常の戦略です。 エラーが起きたサーバプロセスが、終了する前に、何かしら破損した共有状態を持っていたことが原因かもしれません。 このオプションは、postgresが共有データ構造を再初期化しないように指定します。 知識のあるシステムプログラマであれば、その後にデバッガを使用して共有メモリやセマフォの状態を検証することができます。

-O

システムテーブルの構造の変更を可能にします。 これはinitdbによって使われます。

-P

システムテーブルの読み込みの際にシステムインデックスを無視します システムテーブルの更新の際にはインデックスは更新されます。 破損したシステムインデックスから復旧する場合に有用です。

-t pa[rser] | pl[anner] | e[xecutor]

主要なシステムモジュールに関連する問い合わせに対し、個別に時間に関する統計情報を表示します。 このオプションは-sオプションと一緒に使うことはできません。

-T

これはサーバプロセスが異常終了する問題をデバッグするためのオプションです。 このような状態では、他のすべてのサーバプロセスに対し、終了し、共有メモリやセマフォを再初期化するように通知するのが、通常戦略です。 エラーが起きたサーバプロセスが、終了する前に、何かしら破損した共有状態を持っていたことが原因かもしれません。 このオプションは、postgresが他のすべてのサーバプロセスに対しSIGSTOPシグナルを送信して、終了させるのではなく停止させることを指定します。 これによりシステムプログラマはすべてのサーバプロセスのコアダンプを手作業で収集することができます。

-v protocol

特定のセッションで使われるフロントエンド/バックエンドプロトコルのバージョン番号を指定します。 このオプションは内部使用のみを目的としたものです。

-W seconds

新規サーバプロセスが起動したときに指定した秒数の遅延が発生します。 認証手続きはこの後で行われます。 サーバプロセスにデバッグを接続する時間を提供することが目的です。

シングルユーザモード用のオプション

以下のオプションはシングルユーザモードに対してのみ適用されます。

--single

シングルユーザモードを選択します。 これはコマンドラインで最初の引数でなければなりません。

database

アクセスするデータベースの名称を指定します。 これはコマンドラインの最後の引数でなければなりません。 省略時は、デフォルトでユーザ名になります。

-E

すべてのコマンドをエコー表示します。

-j

改行文字を文の区切り文字として使用することを無効にします。

-r filename

全てのサーバログをfilenameに送ります。 通常のマルチユーザモードでは、このオプションは無視され、stderrがすべてのプロセスで使用されます。

環境

PGCLIENTENCODING

クライアントで使用する文字符号化方式のデフォルトです。 （クライアントは個別に変更することができます。） また、この値は設定ファイルでも設定可能です。

PGDATA

デフォルトのデータディレクトリの場所です。

PGDATESTYLE

[DateStyle](#)実行時パラメータのデフォルト値です。 （この環境変数の使用は廃止予定です。）

PGPORT

デフォルトのポート番号です。 （設定ファイル内で設定するほうが好まれています。）

TZ

サーバのタイムゾーンです。

診断

`semget` や `shmget` について言及するエラーメッセージは、おそらく、適切な共有メモリやセマフォを提供できるようにカーネルを再構築する必要があることを示します。詳細は[項17.4](#)を参照してください。ただし、[shared buffers](#)の値を小さくしてPostgreSQLの共有メモリ消費量を低減する、[max_connections](#)の値を小さくしてセマフォ消費量を低減するといった対策を行うことで、カーネルの再構築を先延ばしすることができます。

他のサーバが既に実行中であることを示すエラーメッセージに対しては、例えば以下のコマンドを使用して、注意深く検査しなければなりません。

```
$ ps ax | grep postgres
```

もしくは

```
$ ps -ef | grep postgres
```

どちらを使用するかは、システムによって異なります。競合するサーバが稼働していないことが確実であれば、メッセージ内で示されたロックファイルを削除して、再度試してください。

ポートをバインドできなかったことを示すエラーメッセージの場合は、PostgreSQL以外のプロセスが既にそのポートを使用している可能性が考えられます。また、`postgres`を停止して、すぐに同じポートを使う設定で再起動した場合、このエラーが出ることがあります。この場合、オペレーティングシステムがポートを閉鎖するまで数秒待ってから再挑戦してください。最後に、オペレーティングシステムが予約しているポート番号を指定した場合も、このエラーが発生することがあります。例えば、Unixの多くのバージョンでは1024より小さいポート番号は"信頼できる (trusted)"とみなされており、Unixのスーパーユーザ以外アクセスできません。

注釈

[pg_ctl](#)ユーティリティコマンドは、`postgres`サーバの起動とシャットダウンを安全かつ簡便にするためのものです。

`postgres`を終了させるためにSIGKILLを使用するのは可能な限り避けてください。これにより、`postgres`が保持しているシステムリソース（共有メモリやセマフォ）を解放する前に終了してしまうことを防止できます。システムリソースが解放されていないと、新しく`postgres`を起動する時に問題が起こる可能性があります。

`postgres`サーバを通常に終了させるために、SIGTERM、SIGINT、SIGQUITを使用することができます。SIGTERMは全てのクライアントが終了するのを待ってから終了させます。SIGINTは強制的に全てのクライアントとの接続を切断します。SIGQUITは適切なシャットダウンを行わないで、即時に終了させます。この場合、次の起動時に復旧処理が行われます。

SIGHUPシグナルによりサーバの設定ファイルを再読み込みします。また、SIGHUPを個々のサーバプロセスに送信することも可能ですが、通常は意味がありません。

実行中の問い合わせを中止するには、そのコマンドを実行しているプロセスにSIGINTシグナルを送信してください。

`postgres`サーバでは、下位サーバプロセスを通常通りに終了させるにはSIGTERMを、通常のクリーンアップを行わずに終了させるにはSIGQUITを使用します。これらはユーザが使用すべきではありません。また、サーバにSIGKILLシグナルを送信するのは好ましくありません。メインの`postgres`はこれをクラッシュとして解釈するので、クラッシュからの標準的な復旧手続きの一環として、強制的に共通の親プロセスを持つ`postgres`プロセスを全て終了させます。

不具合

--オプションはFreeBSDやOpenBSDではうまく動きません。代わりに-cを使用してください。これは対象のオペレーティングシステムの不具合です。もし修正されなければ、将来のPostgreSQLリリースで回避策を提供する予定です。

使用方法

シングルユーザモードのサーバを起動するには、以下のコマンドを使用してください。

```
postgres --single -D /usr/local/pgsql/data other-options my_database
```

データベースディレクトリの正確なパスを-Dで指定します。パスを指定しない時は、必ずPGDATA環境変数を設定しておいてくだ

さい。また、作業対象とするデータベースの名前も指定してください。

通常、シングルユーザモードのサーバでは、改行をコマンド入力の終わりとしみなします。psqlにおけるセミicolonのような高度な機能はありません。コマンドが複数行にわたる場合は、改行を入力する前にバックスラッシュを入力しなければなりません（最終行を除く）。

ただし、`-j` コマンドラインスイッチを使用した場合は、改行がコマンド入力の終わりとはみなされません。サーバは標準入力をファイル終端マーク（EOF）まで読み取り、そこまでの入力を1つのコマンド文字列として処理します。この場合、バックスラッシュと改行の組み合わせは特別扱いされません。

セッションを終了するには、EOF（通常**Control+D**）を入力します。`-j`を使用している場合、終了するためにEOFを2度連続して入力する必要があります。

シングルユーザモードのサーバには、高度な行編集機能が用意されていないことに注意してください（例えばコマンド履歴機能はありません）。またシングルユーザモードは、自動チェックポイントなどのバックグラウンド処理をまったく行いません。

例

デフォルト値を使用するpostgresをバックグラウンドで起動するには、以下を入力してください。

```
$ nohup postgres >logfile 2>&1 </dev/null &
```

例えば1234というポートを指定したpostgresを起動するには、以下を入力してください。

```
$ postgres -p 1234
```

psqlを使用してこのサーバに接続するには、以下のように-pオプションでこのポートを指定してください。

```
$ psql -p 1234
```

またはPGPORT環境変数を設定する必要があります。

```
$ export PGPORT=1234
$ psql
```

名前付き実行時パラメータを以下のいずれかで設定することができます。

```
$ postgres -c work_mem=1234
$ postgres --work-mem=1234
```

どちらの形式でも、`postgresql.conf`内に記載されているかもしれない`work_mem`の値を上書きします。コマンドラインでは、パラメータ内のアンダースコア（`_`）をアンダースコアとしてもダッシュ記号（`-`）としても記載できることに注意してください。おそらく短期間の実験という場合を除き、コマンドラインスイッチに依存するよりも`postgresql.conf`内の設定を変更してパラメータを設定の方が実用的です。

関連項目

[initdb](#), [pg_ctl](#)

[前のページ](#)
pg_resetxlog

[ホーム](#)
[上に戻る](#)

[次のページ](#)
postmaster

postmaster

名前

postmaster -- PostgreSQLデータベースサーバ

概要

postmaster [*option...*]

説明

postmasterはpostgresの廃止予定の別名です。

関連項目

[postgres](#)

[前のページ](#)
postgres

[ホーム](#)
[上に戻る](#)

[次のページ](#)
内部情報

VII. 内部情報

ここでは、PostgreSQL開発者が使用できる情報を分類して説明します。

目次

44章 [PostgreSQL内部の概要](#)

- 44.1. [問い合わせの経路](#)
- 44.2. [接続の確立](#)
- 44.3. [構文解析過程](#)
- 44.4. [PostgreSQLルールシステム](#)
- 44.5. [プランナ/オブティマイザ](#)
- 44.6. [エクゼキュータ](#)

45章 [システムカタログ](#)

- 45.1. [概要](#)
- 45.2. [pg_aggregate](#)
- 45.3. [pg_am](#)
- 45.4. [pg_amop](#)
- 45.5. [pg_amproc](#)
- 45.6. [pg_attrdef](#)
- 45.7. [pg_attribute](#)
- 45.8. [pg_authid](#)
- 45.9. [pg_auth_members](#)
- 45.10. [pg_cast](#)
- 45.11. [pg_class](#)
- 45.12. [pg_constraint](#)
- 45.13. [pg_collation](#)
- 45.14. [pg_conversion](#)
- 45.15. [pg_database](#)
- 45.16. [pg_db_role_setting](#)
- 45.17. [pg_default_acl](#)
- 45.18. [pg_depend](#)
- 45.19. [pg_description](#)
- 45.20. [pg_enum](#)
- 45.21. [pg_extension](#)
- 45.22. [pg_foreign_data_wrapper](#)
- 45.23. [pg_foreign_server](#)
- 45.24. [pg_foreign_table](#)
- 45.25. [pg_index](#)
- 45.26. [pg_inherits](#)
- 45.27. [pg_language](#)
- 45.28. [pg_largeobject](#)
- 45.29. [pg_largeobject_metadata](#)
- 45.30. [pg_namespace](#)
- 45.31. [pg_opclass](#)
- 45.32. [pg_operator](#)
- 45.33. [pg_opfamily](#)
- 45.34. [pg_pltemplate](#)
- 45.35. [pg_proc](#)
- 45.36. [pg_range](#)
- 45.37. [pg_rewrite](#)
- 45.38. [pg_seclabel](#)
- 45.39. [pg_shdepend](#)
- 45.40. [pg_shdescription](#)
- 45.41. [pg_shseclabel](#)
- 45.42. [pg_statistic](#)
- 45.43. [pg_tablespace](#)
- 45.44. [pg_trigger](#)
- 45.45. [pg_ts_config](#)
- 45.46. [pg_ts_config_map](#)

- 45.47. [pg_ts_dict](#)
- 45.48. [pg_ts_parser](#)
- 45.49. [pg_ts_template](#)
- 45.50. [pg_type](#)
- 45.51. [pg_user_mapping](#)
- 45.52. [システムビュー](#)
- 45.53. [pg_available_extensions](#)
- 45.54. [pg_available_extension_versions](#)
- 45.55. [pg_cursors](#)
- 45.56. [pg_group](#)
- 45.57. [pg_indexes](#)
- 45.58. [pg_locks](#)
- 45.59. [pg_prepared_statements](#)
- 45.60. [pg_prepared_xacts](#)
- 45.61. [pg_roles](#)
- 45.62. [pg_rules](#)
- 45.63. [pg_seclabels](#)
- 45.64. [pg_settings](#)
- 45.65. [pg_shadow](#)
- 45.66. [pg_stats](#)
- 45.67. [pg_tables](#)
- 45.68. [pg_timezone_abbrevs](#)
- 45.69. [pg_timezone_names](#)
- 45.70. [pg_user](#)
- 45.71. [pg_user_mappings](#)
- 45.72. [pg_views](#)

46章 [フロントエンド/バックエンドプロトコル](#)

- 46.1. [概要](#)
- 46.2. [メッセージの流れ](#)
- 46.3. [ストリーミングレプリケーションプロトコル](#)
- 46.4. [メッセージのデータ型](#)
- 46.5. [メッセージの書式](#)
- 46.6. [エラーおよび警報メッセージフィールド](#)
- 46.7. [プロトコル2.0からの変更点の要約](#)

47章 [PostgreSQLコーディング規約](#)

- 47.1. [書式](#)
- 47.2. [サーバ内部からのエラーの報告](#)
- 47.3. [エラーメッセージのスタイルガイド](#)

48章 [多言語サポート](#)

- 48.1. [翻訳者へ](#)
- 48.2. [プログラマへ](#)

49章 [手続き言語ハンドラの作成](#)

50章 [外部データラッパの作成](#)

- 50.1. [Foreign Data Wrapper Functions](#)
- 50.2. [外部データラッパのコールバックルーチン](#)
- 50.3. [外部データラッパヘルパ関数](#)
- 50.4. [外部データラッパのクエリプラン作成](#)

51章 [遺伝的問い合わせ最適化](#)

- 51.1. [複雑な最適化問題としての問い合わせ処理](#)
- 51.2. [遺伝的アルゴリズム](#)
- 51.3. [PostgreSQLの遺伝的問い合わせ最適化 \(GEQO\)](#)
- 51.4. [さらに深く知るには](#)

52章 [インデックスアクセスメソッドのインタフェース定義](#)

- 52.1. [インデックス用のカタログ項目](#)
- 52.2. [インデックスアクセスメソッド関数](#)

- 52.3. [インデックススキャン](#)
- 52.4. [インデックスのロック処理に関する検討](#)
- 52.5. [インデックス一意性検査](#)
- 52.6. [インデックスコスト推定関数](#)

53章 [GiSTインデックス](#)

- 53.1. [序文](#)
- 53.2. [拡張性](#)
- 53.3. [実装](#)
- 53.4. [例](#)

54章 [SP-GiSTインデックス](#)

- 54.1. [はじめに](#)
- 54.2. [拡張性](#)
- 54.3. [実装](#)
- 54.4. [例](#)

55章 [GINインデックス](#)

- 55.1. [はじめに](#)
- 55.2. [拡張性](#)
- 55.3. [実装](#)
- 55.4. [GINの小技](#)
- 55.5. [制限](#)
- 55.6. [例](#)

56章 [データベースの物理的な格納](#)

- 56.1. [データベースファイルのレイアウト](#)
- 56.2. [TOAST](#)
- 56.3. [空き領域マップ](#)
- 56.4. [可視性マップ](#)
- 56.5. [初期化フォーク](#)
- 56.6. [データベースページのレイアウト](#)

57章 [BKIバックエンドインタフェース](#)

- 57.1. [BKIファイル形式](#)
- 57.2. [BKIコマンド](#)
- 57.3. [BKIファイルのブートストラップの構成](#)
- 57.4. [例](#)

58章 [プランナが統計情報をどのように使用するか](#)

- 58.1. [行推定の例](#)

第 44章PostgreSQL内部の概要

目次

- 44.1. [問い合わせの経路](#)
- 44.2. [接続の確立](#)
- 44.3. [構文解析過程](#)
 - 44.3.1. [パーサ](#)
 - 44.3.2. [書き換えプロセス](#)
- 44.4. [PostgreSQLルールシステム](#)
- 44.5. [プランナ/オプティマイザ](#)
- 44.6. [エグゼキュータ](#)

著者: 本章は[Enhancement of the ANSI SQL Implementation of PostgreSQL](#)として、ウィーン工科大学にてO.Univ.Prof.Dr. Georg GottlobとUniv.Ass.Mag. Katrin Seyr.の指導の下にStefan Simkovicsが書いた修士論文の一部が基になっています。

本章ではPostgreSQLのバックエンドの内部構造の概要を説明します。次からの節を読んだ後には、問い合わせがどのように処理されるかの概念がつかめているはずです。ここではPostgreSQLの内部操作の詳細な説明は行いません。記述するとしたら文章が膨大になってしまうからです。どちらかと言えば、バックエンドが問い合わせを受け取った時点からクライアントに結果を返す時点の間に引き起こる操作の一般的な流れを理解してもらうのが目的です。

44.1. 問い合わせの経路

ここでは、問い合わせが結果を得るためにたどる過程を簡単に説明します。

- 1 11 アプリケーションプログラムからPostgreSQLサーバに接続が確立されなくてはなりません。 アプリケーションプログラムはサーバに問い合わせを送り、そしてサーバから送り返される結果を待ちます。

- 1 11 構文解析過程で、アプリケーションプログラムから送られた問い合わせの構文が正しいかをチェックし、問い合わせツリーを作成します。

- 1 11 書き換えシステムは構文解析過程で作られた問い合わせツリーを受け取り、問い合わせツリーに適用するための（システムカタログに格納されている）ルールを探します。 そしてルール本体で与えられた変換を実行します。

書き換えシステムの適用例の1つとしてビューの具現化が挙げられます。 ビュー（すなわち仮想テーブル）に対して問い合わせがあると、書き換えシステムが代わってユーザの問い合わせを、ビュー定義で与えられた実テーブルにアクセスする問い合わせに書き換えます。

- 1 11 プランナ/オプティマイザは、（書き換えられた）問い合わせツリーを見て、エグゼキュータに渡すための問い合わせ計画を作ります。

そのためにまず同じ結果をもたらす全ての可能な限りの経路を作ります。 例えば、スキャンの対象となるリレーション上にインデックスがあるとすると2つの経路があります。 1つは単純なシーケンシャルスキャンで、もう1つはインデックスを使ったスキャンです。 次にそれぞれの経路を実行するためのコストが見積もられ、一番コストの小さい経路が選ばれます。 一番コストの小さな経路は、エグゼキュータが実行できるように完全な計画に拡張されます。

- 1 11 エグゼキュータは再帰的に計画ツリー上を進み、計画で示されている方法で行を抽出します。 エグゼキュータはリレーションをスキャンする間保存システムを利用してソートと結合を実行し、検索条件の評価を行い、最後に得られた行を返します。

これからの節では、PostgreSQLの内部制御とデータ構造をより良く理解するために、上に記載した事柄をさらに詳しく説明します。

44.2. 接続の確立

PostgreSQLは単純な「1プロセスに1ユーザ」のクライアント/サーバモデルによって実装されています。このモデルでは1つのサーバプロセスに対し厳密に1つのクライアントプロセスしか存在しません。いくつかの接続が行われるか事前にわからないので、接続要求の度に新しいプロセスを作るマスタープロセスを使わなければなりません。このマスタープロセスは`postgres`と呼ばれ、指定されたTCP/IPポートで入ってくる接続要求を監視します。接続要求を検出すると、`postgres`プロセスは新しいサーバプロセスを生み出します。このサーバのタスクはセマフォと共有メモリを活用してお互いに連絡を取り合い、同時にデータにアクセスしても整合性が保たれるようにします。

クライアントプロセスは[第46章](#)に記載されたPostgreSQLプロトコルを理解できるどんなプログラムでも構いません。多くのクライアントは`libpq` C言語ライブラリに基づいていますが、Java JDBCドライバのようにいくつかの独立したプロトコル実装も存在します。

いったん接続が確立されると、クライアントプロセスはバックエンド（サーバ）に問い合わせを送ることができます。問い合わせは平文で送信されます。つまり、フロントエンド（クライアント）は構文解析を行いません。サーバは問い合わせの構文解析を行い、実行計画を作り、そして計画を実行し、抽出した行を確立された接続を通じてクライアントに返します。

44.3. 構文解析過程

構文解析過程は2つの部から構成されています。

- `gram.y`と`scan.l`で定義されているパーサは、Unixのツールbisonとflexを使って構築されます。
- 変換プロセスは、パーサから返されたデータ構造の変更や追加を行います。

44.3.1. パーサ

パーサは、(平文のASCIIテキストとして渡される) 問い合わせ文字列が正しい構文になっているかチェックしなければいけません。もし構文が正しい場合は構文解析ツリーが作られて返されます。正しくない場合はエラーが返されます。パーサと字句解析はUnixでよく知られたツールのbisonとflexを使用して実装されています。

字句解析はファイル`scan.l`で定義され、識別子やSQLキーワードなどの確認を担当します。検出された全てのキーワードや識別子に対しトークンが生成されパーサに渡されます。

パーサはファイル`gram.y`の中で定義され、文法ルールとルールが実行された時に実行されるアクションの組から構成されています。アクションのコード(実際はC言語コードです)は構文解析ツリーを作るのに使われます。

ファイル`scan.l`はプログラムflexを使ってCのソースファイル`scan.c`に変換されます。そして`gram.y`はbisonを使って`gram.c`に書き換えられます。これらの書き換えが終わると、パーサを作るために通常のCコンパイラが使えるようになります。生成されたCのファイルには絶対に変更を加えないでください。と言うのは次にflexもしくはbison が呼ばれた時に上書きされる可能性があるからです。

注意: ここで言及した書き換えやコンパイルは通常PostgreSQLのソースと一緒に配布される`makefile`を使って自動的に行われます。

bisonまたは`gram.y`で定義される文法ルールの詳細は本稿では説明しきれません。flexやbisonについては本や資料がたくさん出ています。`gram.y`の文法の勉強を始める前にbisonの知識が必須となります。その知識なしではそこで何が起きているのかを理解することは難しいでしょう。

44.3.2. 書き換えプロセス

構文解析過程ではSQLの構文構造に関する固定ルールのみを使って構文解析ツリーを作成します。システムカタログの参照を行わないので、要求されている操作の詳細な語義は理解しません。構文解析が終わった後に入力としてパーサから戻されたツリーを書き換えプロセスが引き受け、どのテーブル、関数、そして演算子が問い合わせによって参照されているのかの判断に必要な語義翻訳を行います。この情報を表すために作成されるデータ構造を問い合わせツリーと呼びます。

語義解釈と入力の構文解釈を切り分ける理由は、システムカタログの参照はトランザクション内でのみ行うことができますが、問い合わせ文字列を受け取ってすぐにトランザクションを開始することは好ましくないと考えられるからです。入力に対する構文解析過程ではトランザクション管理コマンド(`BEGIN`、`ROLLBACK`など)を特定するだけで十分であるとともに、それ以上の分析を行わなくても正しい処理が実行されます。実際の問い合わせ(例えば`SELECT`もしくは`UPDATE`)に関わっているということがわかっていて既にあるトランザクション内にいなければ新規トランザクションを開始することは問題ありません。これ以降に限り書き換えプロセスを起動することができます。

書き換えプロセスで作成された問い合わせツリーはほとんどの箇所で加工されていない構文解析ツリーに構造的には似ていますが、細部では数多くの相違が存在します。例えば、構文解析ツリーの`FuncCall`ノードは構文的には関数呼び出しのように見える何かを表わしています。これは参照された名前が通常の関数になるか集約関数となるかによって`FuncExpr`もしくは`Aggref`に書き換えられることがあります。さらに、列の実際のデータ型と式の結果についての情報が問い合わせツリーに書き加えられます。

44.4. PostgreSQL ルールシステム

PostgreSQLには、ビューと理解の仕方ですらとも取れるビューの更新の仕様に對應する強力なルールシステムがあります。元々PostgreSQLのルールシステムは2つの実装で構成されていました。

- 初めの1つは行レベルの処理を使って動き、エグゼキュータの内部に深く実装されていました。個別の行がアクセスされる度にルールシステムが呼ばれていました。この実装は1995年、最後のBerkeley PostgreSQLプロジェクトの公式リリースがPostgres95へ移行する時に削除されました。
- ルールシステムの2つ目の実装は問い合わせ書き換えと呼ばれる手法です。書き換えシステムは構文解析過程とプランナ/オプティマイザの間にあるモジュールです。この手法は現在でも実装されています。

問い合わせの書き換えについては[第37章](#)にて詳しく論議されますのでここでは取り扱いません。書き換えの入出力ともに問い合わせツリーであることを指摘するのに留めます。と言うのは、ツリー内の表現の仕方や語義をどの程度詳しく判断するかには影響がないからです。書き換えはマクロの拡張と捉えることもできます。

44.5. プランナ/オプティマイザ

プランナ/オプティマイザの役割は最適な実行計画を作ることです。ある与えられたSQL問い合わせは（それがあ問い合わせツリーになるのですが）、同じ結果をもたらす、多くの異なった方法で実際には実行できます。もしもコンピュータの演算として可能であれば、問い合わせオプティマイザは可能な実行計画をすべて検証し、実行するとした場合に一番早く結果をもたらすと想定される実行計画を選択します。

注意: 場合によっては、問い合わせがどう実行されるか、可能性のある全ての手段を検証するため、膨大な時間とメモリ空間を消費する可能性があります。特に数多くの結合操作に問い合わせが関わった時です。相応な（必ずしも最適ではありませんが）問い合わせ計画を、相応な時間内で決定するためPostgreSQLは結合の数が閾値を越えた場合、遺伝的問い合わせオプティマイザ（[第51章](#)参照）を使用します（[geqo_threshold](#)を参照ください）。

このプランナの検索手順は、実際には経路という名前のデータ構造を使用します。経路とは、プランナが決定を行うために必要な情報のみに切り詰めた単なる計画の表現です。最も安価である経路が決定された後、全てが揃った計画ツリーが作成されてエグゼキュータに渡されます。これはつまり、要求されている実行計画はエグゼキュータがそれを実行するために十分な詳しい内容を所有していることを表しています。本節の残りでは、経路と計画の違いについて無視します。

44.5.1. 実行可能な計画の生成

プランナ/オプティマイザは、問い合わせの中で使用される個々のリレーション（テーブル）をスキャンするための計画を生成することから始めます。各リレーション上で利用できるインデックスにより実行可能な計画が決まります。リレーションをシーケンシャルスキャンする可能性は常にありますので、シーケンシャルスキャンを使用する計画は常に作成されます。リレーション上にインデックス（例えばB-treeインデックス）が定義され、問い合わせには`relation.attribute OPR constant`という条件があるとしましょう。もし`relation.attribute`がB-treeインデックスのキーと一致し、`OPR`がインデックスの演算子クラスに列挙されている演算子の1つであれば、リレーションをスキャンするためにB-treeインデックスを使用する別の計画が作られます。さらに他のインデックスが存在し、問い合わせの中で条件がインデックスのキーに一致した場合、なおその上に計画が検討されます。インデックススキャン計画は、問い合わせの（もし存在すれば）`ORDER BY`句に一致するソート順、もしくはマージ結合に便利なソート順を所有するインデックスに対して生成されます（以下を参照してください）。

問い合わせが2つ以上のリレーションの結合を必要とすると、リレーションを結合する計画は、単一のリレーションをスキャンするために全ての実行可能な計画が探し出された後に検討されます。3つの実行可能な結合戦略を示します。

- 入れ子ループ結合： 左側のリレーションの中で見つけられた行ごとに右側のリレーションが1回スキャンされます。この戦略は実装が簡単ですが、時間がかかる場合があります（とは言っても右側のリレーションがインデックススキャンによってスキャン可能であればよい戦略になります。右側のインデックススキャンのキーとして左側のリレーションの現在の行の値を使用することができます。）
- マージ結合： 結合を開始する前に、それぞれのリレーションを結合属性でソートします。そして、2つのリレーションを並行してスキャンし、一致する行を結合行の形にまとめます。それぞれのリレーションがたった1回しかスキャンされなくて済むのでこの結合は魅力的です。要求されるソートは、明示的なソート段階、または、結合キー上のインデックスを使用して適切な順序でリレーションをスキャンすることにより行われます。
- ハッシュ結合： 右側のリレーションがハッシュキーとして結合属性を用いて初めにスキャンされ、ハッシュテーブルに読み込まれます。次に左側のリレーションがスキャンされ、見つかったそれぞれの行に相応しい値が、右側のリレーションの行を探し出すためのハッシュキーとして使われます。

問い合わせが3つ以上のリレーションを含む場合、それぞれ2つの入力を持つ結合段階のツリーによって最終結果を構築しなければなりません。プランナは最も低コストな計画を見つけ出すために、あり得る異なった結合順序を検証します。

問い合わせが[geqo_threshold](#)より少ないリレーションを使用する場合、最適な結合シーケンスを見つけ出すため、完璧に近い検索が行われます。プランナはWHERE条件での対応する結合句が存在する（すなわち、`where rel1.attr1=rel2.attr2`のような制約に対して）、あらゆる2つのリレーション間の結合を優先的に考慮します。結合句のない結合ペアは他に選択のない場合に考慮されます。つまり、ある特定のリレーションが他のどんなリレーションに対しても有効な結合句を持たない場合です。すべての有効な計画はプランナが考慮したすべての結合ペアに対し生成され、最も安価な（と評価された）ものが選択されます。

[geqo_threshold](#)を上回ると、考慮された結合シーケンスは[第51章](#)に記載されているように経験則で決定されます。そうでない時、処理は変わりません。

最終的な計画ツリーは基になっているリレーションのシーケンシャルもしくはインデックススキャン、そして必要に応じて入れ子ループ、マージ、またはハッシュ結合のノード、さらにはソートまたは集約関数計算ノードのような必要とされる補助の手順から構成されます。これらほとんどの計画ノード型は選択（特定の論理演算条件に合致しない行を破棄すること）および射影（与えられた列の値に基づき派生した列の集合を計算すること、つまり必要なところでスカラ式の評価をすること）を行う追

加的能力を持っています。 プランナの1つの責任は、 WHERE 句から選択条件を付加して計画ツリーの最も適切なノードに対し必要とされる出力式を計算することです。

44.6. エクゼキュータ

エクゼキュータは、プランナ/オプティマイザで作成された計画を受け取り、必要な行の集合を抽出するために再帰的に処理します。これは本質的に要求引き寄せ型 (demand-pull) パイプライン機能です。計画ノードが呼ばれる度にもう1つの行を引き渡すか、行を引き渡したことの報告を行わなければなりません。

具体的な例を提供する目的で頂点のノードがMergeJoinノードである場合を想定しましょう。いかなるマージも実行される前に (それぞれの副計画から1つずつ) 2つの行を取ってこなくてははいけません。ですからエクゼキュータは副計画を処理するために自分自身を再帰的に呼び出します (lefttreeに付随する副計画から開始します)。新しい頂点のノード (左の副計画の頂点のノード) はSortノードであるとしましょう。ここでもノード自体が処理される前に入力行を取ってこなくてははいけません。Sortの子ノードは実際のテーブルの読み取りを表現しているSeqScanノードのこともあり得ます。このノードの処理はエクゼキュータにテーブルから行を抽出させ、呼び出しているノードに渡し戻させます。Sortノードはソート対象の全てのノードを取得するために子ノードを繰り返し呼び出します。入力がなくなった時 (子ノードが行ではなくNULLを返してきた時) Sortコードがソートを実行して最終的に最初の出力行を返すことができるようになります。つまりソート順における最初の結果です。後での要求に答えるためソート順に引き渡すことができるように残っている行は保存されます。

MergeJoinノードは同じようにしてその右副計画から最初の行を要求します。そこで2つの行が結合できるかどうか比較されます。もし結合できる場合には呼び出し側に結合された行が返されます。次の呼び出しの時に、もしくは入力された現在の組み合わせが結合できない場合はすぐに、あるテーブルあるいはそれ以外のテーブル (比較の結果に依存して) の次の行に進んで、さらに一致があるかどうか検証されます。最終的にはある副計画もしくは他の計画が使いきられ、MergeJoinノードがこれ以上の結合行を生成できないという意味のNULLを返すことになります。

複雑な問い合わせは多くの階層となった計画ノードに関われますが、概略的な取り扱い方は同じです。それぞれのノードは呼び出される度に次の出力行を計算して返します。それぞれのノードは同時にプランナによって割り当てられたいかなる選択式や射影式でも適用しなければならない責任があります。

エクゼキュータ機構は全ての4つの基本的なSQL型を検証するために用いられます。4つのSQL型とはSELECT、INSERT、UPDATE、そしてDELETEです。SELECTでは、最上位階層のエクゼキュータコードはクライアントから離れて問い合わせ計画によって返されるそれぞれのだけを行を送り返せばよいことになっています。INSERTでは、INSERTで指定された対象となるテーブルに返された行が挿入されます。これはModifyTableと呼ばれる特別な最上位階層の計画ノードの中で行われます。(ある単純なINSERT ... VALUES コマンドは1つのResultノード (これは、結果としての行を1つだけ計算するに留まります) とその上の挿入を実行するためのModifyTableからなる単純な計画ツリーを生成します。しかし、INSERT ... SELECTはエクゼキュータ機構が得意な限りの能力を発揮することを要求する場合があります)。UPDATEではプランナは全ての更新された列の値を含んだ行の演算結果とTID (タプルID、または行ID) を準備します。このデータはModifyTableノードに入力され、ノードでは新しく更新された行の作成と削除された古い行に印を付けるためこの情報を利用します。DELETEでは計画から実際に返されたただ1つの列はTIDで、ModifyTableノードは各対象行を尋ね当てることと削除の印を付けるために単にこのTIDを使用します。

第 45 章 システムカタログ

目次

- 45.1. [概要](#)
- 45.2. [pg_aggregate](#)
- 45.3. [pg_am](#)
- 45.4. [pg_amop](#)
- 45.5. [pg_amproc](#)
- 45.6. [pg_attrdef](#)
- 45.7. [pg_attribute](#)
- 45.8. [pg_authid](#)
- 45.9. [pg_auth_members](#)
- 45.10. [pg_cast](#)
- 45.11. [pg_class](#)
- 45.12. [pg_constraint](#)
- 45.13. [pg_collation](#)
- 45.14. [pg_conversion](#)
- 45.15. [pg_database](#)
- 45.16. [pg_db_role_setting](#)
- 45.17. [pg_default_acl](#)
- 45.18. [pg_depend](#)
- 45.19. [pg_description](#)
- 45.20. [pg_enum](#)
- 45.21. [pg_extension](#)
- 45.22. [pg_foreign_data_wrapper](#)
- 45.23. [pg_foreign_server](#)
- 45.24. [pg_foreign_table](#)
- 45.25. [pg_index](#)
- 45.26. [pg_inherits](#)
- 45.27. [pg_language](#)
- 45.28. [pg_largeobject](#)
- 45.29. [pg_largeobject_metadata](#)
- 45.30. [pg_namespace](#)
- 45.31. [pg_opclass](#)
- 45.32. [pg_operator](#)
- 45.33. [pg_opfamily](#)
- 45.34. [pg_pltemplate](#)
- 45.35. [pg_proc](#)
- 45.36. [pg_range](#)
- 45.37. [pg_rewrite](#)
- 45.38. [pg_seclabel](#)
- 45.39. [pg_shdepend](#)
- 45.40. [pg_shdescription](#)
- 45.41. [pg_shseclabel](#)
- 45.42. [pg_statistic](#)
- 45.43. [pg_tablespace](#)
- 45.44. [pg_trigger](#)
- 45.45. [pg_ts_config](#)
- 45.46. [pg_ts_config_map](#)
- 45.47. [pg_ts_dict](#)
- 45.48. [pg_ts_parser](#)
- 45.49. [pg_ts_template](#)
- 45.50. [pg_type](#)
- 45.51. [pg_user_mapping](#)
- 45.52. [システムビュー](#)
- 45.53. [pg_available_extensions](#)
- 45.54. [pg_available_extension_versions](#)
- 45.55. [pg_cursors](#)
- 45.56. [pg_group](#)
- 45.57. [pg_indexes](#)
- 45.58. [pg_locks](#)

- 45.59. [pg_prepared_statements](#)
- 45.60. [pg_prepared_xacts](#)
- 45.61. [pg_roles](#)
- 45.62. [pg_rules](#)
- 45.63. [pg_seclabels](#)
- 45.64. [pg_settings](#)
- 45.65. [pg_shadow](#)
- 45.66. [pg_stats](#)
- 45.67. [pg_tables](#)
- 45.68. [pg_timezone_abbrevs](#)
- 45.69. [pg_timezone_names](#)
- 45.70. [pg_user](#)
- 45.71. [pg_user_mappings](#)
- 45.72. [pg_views](#)

システムカタログとは、リレーショナルデータベース管理システムがテーブルや列の情報などのスキーマメタデータと内部的な情報を格納する場所です。PostgreSQLのシステムカタログは通常のテーブルです。テーブルを削除したり再作成したり、列の追加および値の挿入や更新をすることは可能ですが、これらの操作でデータベースシステムを台なしにしてしまう可能性もあります。通常手作業でシステムカタログを変更してはいけません。その代わりとしてSQLコマンドを使用します（例えばCREATE DATABASEによりpg_databaseカタログに1行挿入し、ディスク上にデータベースを実際に作成します）。しかしインデックスアクセスメソッドを追加するような特に難易度の高い操作の時などの例外があります。

[前のページ](#)
エグゼキュータ

[ホーム](#)
[上に戻る](#)

[次のページ](#)
概要

45.1. 概要

表45-1にシステムカタログを列挙します。以降システムカタログについてより詳細を説明します。

ほとんどのシステムカタログはデータベースを作成する時にテンプレートデータベースからコピーされデータベースに固有のものとなります。ごく一部のカタログがデータベースクラスタ内の全てのデータベースにわたって物理的に共有されます。これらについては、それぞれのカタログで説明します。

表 45-1. システムカタログ

カタログ名	用途
pg_aggregate	集約関数
pg_am	インデックスアクセスメソッド
pg_amop	アクセスメソッド演算子
pg_amproc	アクセスメソッドサポートプロシージャ
pg_attrdef	列デフォルト値
pg_attribute	テーブル列 ("属性")
pg_authid	認証識別子 (ロール)
pg_auth_members	認証識別子メンバーシップ関係
pg_cast	キャスト (データ型変換)
pg_class	テーブル、インデックス、シーケンス、ビュー ("リレーション")
pg_constraint	検査制約、一意性制約、プライマリーキー制約、外部キー制約
pg_collation	照合順序 (ロケール情報)
pg_conversion	符号化方式変換情報
pg_database	このデータベースクラスタにあるデータベース
pg_db_role_setting	ロール毎およびデータベース毎の設定
pg_default_acl	オブジェクト種類のデフォルト権限
pg_depend	データベースオブジェクト間の依存関係
pg_description	データベースオブジェクトの説明やコメント
pg_enum	列挙型のラベルや値の定義
pg_extension	インストールされた拡張
pg_foreign_data_wrapper	外部データラップの定義
pg_foreign_server	外部サーバの定義
pg_foreign_table	追加の外部テーブル情報
pg_index	追加インデックス情報
pg_inherits	テーブル継承階層
pg_language	関数記述言語
pg_largeobject	ラージオブジェクト用のデータページ
pg_largeobject_metadata	ラージオブジェクトのメタデータ
pg_namespace	スキーマ
pg_opclass	アクセスメソッド演算子クラス
pg_operator	演算子
pg_opfamily	アクセスメソッド演算子族
pg_pltemplate	手続き言語のためのテンプレートデータ
pg_proc	関数とプロシージャ
pg_range	範囲型の情報
pg_rewrite	問い合わせ書き換えルール

pg_seclabel	データベースオブジェクト上のセキュリティラベル
pg_shdepend	共有オブジェクトの依存関係
pg_shdescription	共有オブジェクトに対するコメント
pg_shseclabel	共有データベースオブジェクトのセキュリティラベル
pg_statistic	プランナ統計情報
pg_tablespace	データベースクラスタ内のテーブル空間
pg_trigger	トリガ
pg_ts_config	全文検索設定
pg_ts_config_map	全文検索設定のトークン写像
pg_ts_dict	全文検索辞書
pg_ts_parser	全文検索パーサ
pg_ts_template	全文検索テンプレート
pg_type	データ型
pg_user_mapping	外部サーバへのユーザのマッピング

[前のページ](#)
システムカタログ

[ホーム](#)
[上に戻る](#)

[次のページ](#)
pg_aggregate

45.2. pg_aggregate

pg_aggregateカタログには集約関数の情報が格納されています。集約関数とは、値の集合（多くの場合は問い合わせ条件に該当する各行の1つの列）にある操作を行い、それらすべての値の演算の結果得られる単一の値を返します。集約関数の代表的なものはsum、countそしてmaxです。pg_aggregate内の各項目は、pg_proc内の項目の拡張です。pg_procの項目には、集約の名前、入出力データ型および通常の間数と類似したその他の情報が含まれます。

表 45-2. pg_aggregateの列

名前	型	参照先	説明
aggfnoid	regproc	pg_proc.oid	集約関数のpg_proc OID
aggtransfn	regproc	pg_proc.oid	遷移関数
aggfinalfn	regproc	pg_proc.oid	最終関数（ない時はゼロ）
aggstortop	oid	pg_operator.oid	関連するソート演算子（ない時はゼロ）
aggtranstype	oid	pg_type.oid	集約関数の内部遷移（状態）データのデータ型
agginitval	text		遷移状態の初期値。 外部文字列表現での初期値を含んだテキストフィールド。フィールドがNULL値の場合、推移状態はNULL値で始まります。

新しい集約関数は[CREATE AGGREGATE](#)コマンドで登録されます。集約関数の書き方や遷移関数の説明などの詳細は[項35.10](#)を参照してください。

45.3. pg_am

pg_amカタログにはインデックスアクセスメソッドの情報が格納されます。 システムがサポートするインデックスアクセスメソッド毎に1つの行が存在します。 このカタログの内容については[第52章](#)で詳しく説明します。

表 45-3. pg_amの列

名前	型	参照先	説明
amname	name		アクセスメソッド名
amstrategies	int2		このアクセスメソッド用の演算子戦略番号。 もしアクセスメソッドが固定された演算子戦略群を持たない場合はゼロ
amsupport	int2		このアクセスメソッド用のサポートルーチンの数
amcanorder	bool		アクセスメソッドは、インデックス列の値によってソートされた順序付けスキャンをサポートしているか？
amcanorderbyop	bool		アクセスメソッドは、インデックス列に対する演算子の結果によってソートされた順序付けスキャンをサポートしているか？
amcanbackward	bool		アクセスメソッドは後方スキャンをサポートしているか？
amcanunique	bool		アクセスメソッドが一意インデックスをサポートするかどうか？
amcanmulticol	bool		アクセスメソッドが複数列インデックスをサポートするかどうか？
amoptionalkey	bool		アクセスメソッドが最初のインデックス列に対して制約無しのスキャンをサポートするかどうか？
amsearcharray	bool		アクセスメソッドはScalarArrayOpExpr検索をサポートするかどうか？
amsearchnulls	bool		アクセスメソッドはIS NULL/NOT NULL検索をサポートしているか？
amstorage	bool		インデックス格納データの型を列のデータ型から変更できるか？
amclusterable	bool		この型のインデックスはクラスタ可能か？
ampredlocks	bool		この種類のインデックスが細粒度の述部ロックを管理するか？
amkeytype	oid	pg_type.oid	インデックス内に保存されたデータ型、もしくは固定型ではない場合はゼロ
aminsert	regproc	pg_proc.oid	"このタプルを挿入する"関数
ambeginscan	regproc	pg_proc.oid	"インデックススキャンを準備する"関数
amgettuple	regproc	pg_proc.oid	"次に有効なタプル"関数。 存在しない場合はゼロ
amgetbitmap	regproc	pg_proc.oid	"有効なタプルをすべて取り出す"関数。 存在しない場合はゼロ
amrescan	regproc	pg_proc.oid	"インデックススキャンを（再）開始する"関数
amendscan	regproc	pg_proc.oid	"インデックススキャン後の整理"関数
ammarkpos	regproc	pg_proc.oid	"現在のスキャン位置を記録する"関数
amrestrpos	regproc	pg_proc.oid	"記録したスキャン位置に復元する"関数
ambuild	regproc	pg_proc.oid	"新規インデックスを構築する"関数
ambuildempty	regproc	pg_proc.oid	"空のインデックスを構築する"関数
ambulkdelete	regproc	pg_proc.oid	一括削除関数
amvacuumcleanup	regproc	pg_proc.oid	VACUUM後の消去関数
amcanreturn	regproc	pg_proc.oid	インデックスがインデックスオンリースキャンをサポートするかどうかを検査する関数。 存在しなければゼロ
amcostestimate	regproc	pg_proc.oid	インデックススキャンのコスト推測関数
amoptions	regproc	pg_proc.oid	インデックスのreloptionsを解析、検証する関数

45.4. pg_amop

pg_amop カタログにはアクセスメソッド演算子の集合に関連付けられた演算子の情報が格納されています。演算子族のメンバである演算子毎に1つの行が存在します。演算子族のメンバは検索演算子または順序付け演算子のいずれかになることができます。演算子は1つ以上の演算子族に現れますが、演算子族の中では検索でも順序付けでも複数現れることはありません。（ほとんどありませんが、ある演算子が検索目的と順序付け目的の両方で使用されることが許されます。）

表 45-4. pg_amopの列

名前	型	参照先	説明
amopfamily	oid	pg_opfamily.oid	この項目用の演算子族
amoplefttype	oid	pg_type.oid	演算子の左辺側のデータ型
amoprightright	oid	pg_type.oid	演算子の右辺側のデータ型
amopstrategy	int2		演算子の戦略番号
amoppurpose	char		演算子の目的。検索用ならば <code>s</code> 、順序付け用であれば。
amopopr	oid	pg_operator.oid	演算子のOID
amopmethod	oid	pg_am.oid	演算子族用のインデックスアクセスメソッド
amopsortfamily	oid	pg_opfamily.oid	順序付け用の演算子の場合、この項目のソートが従うbtree演算子族。検索用演算子であればゼロ

"検索"用演算子の項目は、この演算子族のインデックスを `WHERE indexed_column operator constant` を持たすすべての行を見つけるための検索に使用できることを示します。いうまでもありませんが、こうした演算子は論理型を返さなければならず、また左辺の入力型はインデックス列のデータ型に一致しなければなりません。

"順序付け"用演算子の項目は、この演算子族のインデックスを `ORDER BY indexed_column operator constant` で表される順序で行を返すためのスキャンに使用することができることを示します。こうした演算子の左辺の入力型はインデックス列のデータ型に一致しなければならないことは同じですが、任意のソート可能なデータ型を返すことができます。ORDER BYの正確な意味は、この演算子の結果型用のbtree演算子族を参照する必要があるamopsortfamily列により指定されます。

注意: 現在、順序付け演算子のソート順は参照される演算子族のデフォルト、つまりASC NULLS LASTであると仮定されています。これは将来、ソートオプションを明示的に指定する追加列を加えることで緩和されるかもしれません。

項目のamopmethodは、項目を含む演算子族のopfmethodに一致しなくてはなりません。（ここでのamopmethodは、性能上の理由からカタログ構造を意図的に非正規化したものも含みます。）また、amoplefttypeとamoprightrightは、参照されているpg_operator項目のoprleftとoprrightに一致しなくてはなりません。

45.5. pg_amproc

pg_amproc カタログには、アクセスメソッド演算子族に関連付けられたサポートプロシージャの情報が格納されます。 演算子族に含まれるサポートプロシージャ毎に1つの行が存在します。

表 45-5. pg_amprocの列

名前	型	参照先	説明
amprocfamily	oid	pg_opfamily.oid	この項目用の演算子族
amproclefttype	oid	pg_type.oid	関連付けられた演算子の左辺のデータ型
amprocrighttype	oid	pg_type.oid	関連付けられた演算子の右辺データ型
amprocnum	int2		サポートプロシージャ番号
amproc	regproc	pg_proc.oid	プロシージャのOID

amproclefttypeとamprocrighttype属性の通常解釈は、それらが、特定のサポートプロシージャがサポートする演算子の左と右の入力型を識別することです。いくつかのアクセスメソッドに対して、これらはサポートプロシージャ自身の入力データ型に一致することもあります。 また、そうでないものもあります。 インデックスに対して"デフォルト"サポートプロシージャの概念があります。 これはamproclefttypeとamprocrighttypeの両方が、インデックス演算クラスのopcintypeに等しい、という概念です。

45.6. pg_attrdef

pg_attrdefシステムカタログは列のデフォルト値を格納します。列の主要な情報はpg_attributeに格納されています（下記を参照）。（テーブルが作成された時、もしくは列が追加された時に）デフォルト値を明示的に特定した列のみここに項目を持ちます。

表 45-6. pg_attrdefの列

名前	型	参照先	説明
adrelid	oid	pg_class.oid	この列が属するテーブル
adnum	int2	pg_attribute.attnum	列番号
adbin	pg_node_tree		列のデフォルト値の内部表現
adsrc	text		人間が見てわかるデフォルト値の表現

adsrcフィールドは歴史的なもので使用しない方が良いでしょう。この値は、デフォルト値の表現に影響を与える外部の変化を追跡しないからです。デフォルト値を表示するには、（例えばpg_get_exprを使った）adbinフィールドの逆コンパイルが、良い方法です。

45.7. pg_attribute

pg_attributeカタログにはテーブルの列情報が格納されます。 データベース内のすべてのテーブルの各列に対し必ず1つのpg_attribute行があります。 (また、インデックスとpg_classに項目を持つすべてのオブジェクトに対しての属性記述があります。)

属性という表現は列と同等の意味で、歴史的背景からそのように呼ばれています。

表 45-7. pg_attributeの列

名前	型	参照先	説明
attrelid	oid	pg_class.oid	この列が属するテーブル
attname	name		列名
atttypid	oid	pg_type.oid	この列のデータ型
attstattarget	int4		attstattargetはANALYZEによるこの列に対する蓄積された統計情報をどの程度詳しく管理するかを規定します。 値がゼロの場合は統計情報を収集しません。 負の値の場合は、システムのデフォルトの統計目標を使用すべきであるということです。 正の値が厳密に意味するところはデータ型に依存します。 スカラデータ型に対してattstattargetは収集する"最も一般的な値"の目標となる数であり、また作成する度数分布ビンの目標数でもあります。
attlen	int2		この列の型のpg_type.typelenのコピー
attnum	int2		列番号。 通常、列は1から始まる番号付けがなされます。 oidのようなシステムによる列には(任意の)負の番号が付きます。
attndims	int4		列が配列型の場合は次元数を表現し、そうでない時はゼロです。(現在配列の次元数は強制されていませんのでゼロ以外のどのような値であっても"これは配列である"ということの意味します。)
attcacheoff	int4		格納時は必ず-1となりますが、メモリ内の行記述子に読み込まれた場合は、行内での属性オフセットをキャッシュするために更新される可能性があります。
atttypmod	int4		atttypmodは、テーブル作成時に与えられた型固有のデータ(たとえばvarchar列の最大長)を記録します。 これは型固有の入力関数や長さ強制関数に渡されます。 atttypmodを必要としない型では、通常、この値は-1です。
attbyval	bool		この列の型のpg_type.tybyvalのコピー
attstorage	char		通常、この列の型のpg_type.typstorageのコピー。 TOAST可能なデータ型では、格納ポリシーを制御するために列の作成後に変更することができます。
attalign	char		この列の型のpg_type.typalignのコピー
attnotnull	bool		NOT NULL制約式。 この列を変更し制約を有効にしたり無効にしたりできます。
atthasdef	bool		この列にはデフォルト値があります。 その場合、実際に値を定義するpg_attrdefカタログ内に対応する項目があります。
attisdropped	bool		この列は既に削除されていて有効ではありません。 削除された列は物理的にはまだテーブル上に存在していますが、パーサによって無視されるためSQLでアクセスすることができません。
attislocal	bool		この列はリレーション内でローカルに定義されます。 列がローカルに定義されると同時に継承される場合もあることに注意してください。
attinhcount	int4		この列が持つ直接の祖先の数です。 祖先を持っている列の削除や名前は変更はできません。
attcollation	oid	pg_collation.oid	列で定義された照合順序。 列が照合順序の設定ができないデータ型の場合はゼロ
attacl	aclitem[]		この列に特定して付与された場合における、列レベルのアクセス権限
attoptions	text[]		"keyword=value"文字列のような、属性レベルのオプション
attfdwoptions	text[]		"keyword=value"文字列のような、外部データラップアップションの属性レベル

削除された列のpg_attribute項目では、atttypidはゼロにリセットされます。 しかしattlenとpg_typeからコピーされた他の

フィールドは、有効なままです。 この動作は、削除された列のデータ型が後になって削除されて、pg_type行が存在しないような状況の場合に必要となります。 attlenと他のフィールドは、テーブル内の行の内容を解釈するために使用されます。

45.8. pg_authid

pg_authidカタログはデータベース認証識別子（ロール）の情報を保持します。ロールは"ユーザ"と"グループ"の概念を包括しています。ユーザは本質的にrolcanloginフラグセットを持ったロールです。どのようなロール（rolcanloginを持っている、持っていないに関わらず）も他のロールをメンバとして持っても構いません。pg_auth_membersを参照してください。

このカタログはパスワードを含んでいるため、第三者が内容を読むことができないようにしなければいけません。pg_rolesは、pg_authidのビューで、パスワードのフィールドは空白となっていますので内容を読み取ることができます。

第20章でユーザと権限管理に関するより詳細について説明します。

ユーザの本人確認はクラスタ全体にわたる情報ですので、pg_authidはクラスタのすべてのデータベースで共有されます。データベース毎ではなく、クラスタ毎にたった1つだけpg_authidが存在します。

表 45-8. pg_authidの列

名前	型	説明
rolname	name	ロール名
rolsuper	bool	ロールはスーパーユーザの権限を持っています
rolinherit	bool	ロールは自動的にメンバとして属するロールの権限を継承します
rolcreatorole	bool	ロールはロールを作成することができます
rolcreatedb	bool	ロールはデータベースを作成することができます
rolcatupdate	bool	ロールはシステムカタログを直接更新することができます（スーパーユーザであってもここが真でなければ更新不可）
rolcanlogin	bool	ロールはログインすることができます。つまりロールはセッションを始める認証の識別子となることができます
rolreplication	bool	ロールはレプリケーション用のロールです。つまり、このロールはストリーミングレプリケーション（ 項25.2.5 参照）を初期化すること、および、pg_start_backupとpg_stop_backupを使用してシステムのバックアップモードを設定/解除することができます。
rolconnlimit	int4	ログイン可能なロールでは、これはロールが確立できる同時実行接続数を設定します。-1は制限無しを意味します。
rolpassword	text	（おそらく暗号化された）パスワード。無い場合はNULLです。パスワードが暗号化された場合、この列にはmd5という文字列の後に32文字の16進数表記のMD5ハッシュが含まれます。MD5ハッシュはユーザのパスワードとユーザ名を連結したものになります 例えばユーザjoeのパスワードがxyzyzyの場合、PostgreSQLはxyzyzyjoeをMD5でハッシュ化したものを格納します。この書式に従わないパスワードは暗号化されていないとみなされます。
rolvaliduntil	timestampz	パスワード有効期限（パスワード認証でのみ使用）。 NULLの場合には満了時間はありません。

45.9. pg_auth_members

pg_auth_members カタログはロール間のメンバシップ関係を示しています。 循環していなければ、どのような関係でも許可されています。

ユーザの同一性はクラスタ間で保たれる必要があるため、pg_auth_members はクラスタ間のすべてのデータベースで共有されています。 pg_auth_members のコピーはデータベースごとではなく、各クラスタにひとつだけ持っています。

表 45-9. pg_auth_members の列

名前	型	参照先	説明
roleid	oid	pg_authid.oid	メンバを持っているロールのID
member	oid	pg_authid.oid	roleid のメンバであるロールのID
grantor	oid	pg_authid.oid	このメンバシップを与えたロールのID
admin_option	bool		もしmember がroleid のメンバシップを他に与えることができる場合は真

45.10. pg_cast

pg_cast カタログにはデータ型変換パスが格納されます。 ここには、組み込みのパスとユーザ定義のパスが存在します。

pg_cast は、システムがどのように動作するかわかっているような、あらゆる型変換を表しているわけではないということに注意してください。 いくつかの一般的な規則から推測できないような型変換についてのみ表しています。 例えば、ドメインとその基本の型は明示的にpg_cast 内で表されていません。 他の重要な例外は"自動I/O変換キャスト"です。 これらのキャストは、text 型やほかの文字列型から変換したりされたりするのにデータ型自身のI/O関数を用いていますが、これらのキャストは明示的にpg_cast 内において表されていません。

表 45-10. pg_cast の列

名前	型	参照先	説明
castsouce	oid	pg_type.oid	変換元データ型のOID
casttarget	oid	pg_type.oid	対象データ型のOID
castfunc	oid	pg_proc.oid	このキャストを実行するために使用する関数のOID。キャストメソッドが関数を必要としない場合はゼロが格納されます。
castcontext	char		キャストがどの文脈で呼び出し可能かを示します。eは明示のキャストとしてのみ起動されることを意味します（CAST、::構文を使用します）。aは、対象となる列を明示的に特定するだけでなく暗黙的にも特定することを意味します。iは他の場合と同様に演算式内で暗黙的であることを意味します。
castmethod	char		どのようにキャストが実行されるかを示します。fはcastfuncフィールド内で示される関数が使用されていることを意味します。iは入出力関数が使用されていることを示します。bは型がバイナリを強制しているため、変換が必要ないことを意味します。

pg_cast 内に挙げられているキャスト関数は、第1番目の引数の型として、キャスト元の型をいつも取らなければいけません。 また、キャスト関数は、結果の型としてキャスト先の型を返します。キャスト関数は3つまで引数を持つことができます。もし存在するなら、2番目の引数はinteger 型でなくてはなりません。この引数はキャスト先の型に関連付けられた型修飾子を受け取ります。2番目の引数がない場合は、-1です。3番目の引数は、もし存在する場合は、boolean 型でなくてはなりません。この引数は、もしキャストが明示的なキャストであればtrueを受け取り、そうでない場合はfalseを受け取ります。

もし関連のある関数が複数の引数を持つ場合、キャストの元と先で型が同じであるpg_cast 項目を作成することが妥当です。このような項目は、"length coercion functions"を表現します。"length coercion functions"は型の値を特定の型の修飾子の値に適するように修正します。

pg_cast 項目が異なるキャスト元とキャスト先の型を持っていて、かつ関数が複数の引数を持つ時は、1つの型から別の型への変換し、かつ、1つの手順で長さの修正を適用することを意味します。このような項目が利用できない時は、型修飾子を使用した型の修正は2つの手順が必要です。1つはデータ型の間での変換で、2つ目は修飾子を適用することです。

45.11. pg_class

pg_classカタログはテーブルと、その他に列を持つもの、あるいはテーブルに似た全てのものを目録にしています。 その中にはインデックス (pg_indexも参照)、シーケンス、ビュー、複合型およびTOASTテーブルが含まれます。 relkindを参照してください。 これより以降、"リレーション"と記されている場合はこれらすべてのオブジェクトを意味しています。 すべての列が全てのリレーションの型に該当するとは限りません。

表 45-11. pg_classの列

名前	型	参照先	説明
relname	name		テーブル、インデックス、ビューなどの名前
relnamespace	oid	pg_namespace.oid	このリレーションを持つ名前空間のOID
reltype	oid	pg_type.oid	もし何らかの (pg_type項目を持たないインデックスではゼロ) が存在した場合このテーブルの行の型に対応するデータ型のOID
reloftype	oid	pg_type.oid	型付けされたテーブルでは背後にある複合型のOID。 その他のリレーションではゼロ。
relowner	oid	pg_authid.oid	リレーションの所有者
relam	oid	pg_am.oid	これがインデックスの場合は、それに使われている (B-tree、ハッシュなどの) アクセスメソッド
relfilenode	oid		このリレーションのディスク上のファイルの名前です。 ゼロはディスク上のファイル名が低レベルな状態で決定される"マップ付けされた"リレーションであることを意味します。
reltablespace	oid	pg_tablespace.oid	このリレーションが保存されているテーブル空間。 もしゼロならば、このデータベースのデフォルトテーブル空間を意味します。 (リレーションがディスク上のファイルになくても、問題はありません。)
relpages	int4		このテーブルのディスク上におけるページ単位 (BLCKSZ) の表現サイズ。 これはプランナで使用される単なる推測値です。 VACUUM、ANALYZEおよびCREATE INDEXコマンドなどの一部のDDLコマンドで更新されます。
reltuples	float4		テーブル内の行数。 これはプランナで使用される単なる推測値です。 VACUUM、ANALYZE、CREATE INDEXなどの一部のDDLコマンドで更新されます。
relallvisible	int4		テーブル内の可視マップ内で全て可視とマークされているページ数。 これはプランナで使用される単なる見積です。 これはVACUUM、ANALYZEさらにCREATE INDEXといったいくつかのDDLコマンドで更新されます。
reltoastrelid	oid	pg_class.oid	このテーブルに関連しているTOASTテーブルのOID。 何もない場合はゼロです。 TOASTテーブルは"行に収まらない"大きい属性を副テーブルに格納します。
reltoastidxid	oid	pg_class.oid	TOASTテーブルに対するインデックスのOID。 TOASTテーブルでない場合はゼロ。
relhasindex	bool		テーブルであり、かつ、インデックスを持つ (あるいはつい最近まで持っていた) 時は真。
relisshared	bool		クラスタ内の全てのデータベースにわたってこのテーブルが共有されている場合は真。 (pg_databaseのような) ある特定のシステムカタログのみ共有されます。
relpersistence	char		pは永続テーブル、uはログを取らないテーブル、tは一時テーブルを表します。
relkind	char		rは通常のテーブル、iはインデックス、sはシーケンス、vはビュー、cは複合型、tはTOASTテーブル、fは外部テーブルを表します。
relnatts	int2		リレーションにあるユーザ列数 (システム列は含みません)。 pg_attributeにこれに対応する数多くの項目があるはずです。 pg_attribute.attnumも参照してください。
relchecks	int2		テーブル上のCHECK制約の数。 pg_constraint カタログを参照してください。
relhasoids	bool		リレーションの行毎にOIDを生成する場合は真。

relhaspkey	bool		テーブルにプライマリキーが含まれている（あるいは過去含まれていた）場合に真。
relhasrules	bool		もしテーブルにルールがある（あるいは以前あった）場合に真。 pg_rewrite カタログを参照してください。
relhastriggers	bool		もしテーブルにトリガがある（あるいは以前あった）場合に真。 pg_trigger カタログを参照してください。
relhassubclass	bool		もしテーブルが子テーブルに継承されている（または以前に継承されていた）場合は真。
relfrozenxid	xid		この値より以前のトランザクションIDはすべて、このテーブルで永遠（"凍結"）トランザクションIDに置き換えられます。これは、このテーブルに対して、トランザクションID周回を防ぎ、かつ、 pg_clog を縮小させることを目的としたバキュームを行うかどうかを追跡するために使用されます。リレーションがテーブルではない場合は0（InvalidTransactionId）です。
relacl	aclitem[]		アクセス権限。詳細は GRANT と REVOKE の説明を参照してください。
reloptions	text[]		"keyword=value"文字列のような、アクセスメソッド特有のオプション。

`pg_class`内の複数の論理型フラグは、ゆっくりと保守されます。正しい状態にあるときに真であることが保証されていますが、その条件が真でなくなった時即座に偽に再設定されないかもしれません。例えば`relhasindex`は`CREATE INDEX`で設定されますが、`DROP INDEX`では決して初期化されません。代わりに`VACUUM`がそのテーブルにインデックスがないことを判定した場合に`relhasindex`を初期化します。この調整により競合状態を防止し、同時実行性が向上します。

45.12. pg_constraint

pg_constraintカタログはテーブル上の検査制約、プライマリキー制約、一意性制約、外部キー制約、排他制約を格納します（列制約は特別扱いされていません。全ての列制約は何らかのテーブル制約と同等です。）非NULL制約はここではなく、pg_attributeカタログで示されます。

(CREATE CONSTRAINT TRIGGERで作成される)ユーザ定義の制約トリガもこのテーブルの項目の元になります。

ドメイン上の検査制約もここに格納されます。

表 45-12. pg_constraintの列

名前	型	参照先	説明
conname	name		制約名（一意である必要はありません）。
connamespace	oid	pg_namespace.oid	この制約を含む名前空間のOID。
contype	char		c = 検査制約、 f = 外部キー制約、 p = プライマリキー制約、 u = 一意性制約、 t = 制約トリガ、 x = 排他制約
condeferrable	bool		制約は遅延可能かどうか？
condeferred	bool		制約はデフォルトで遅延可能かどうか？
convalidated	bool		制約が検証されているか？現時点では外部キーとチェック制約の場合のみ偽になる可能性があります。
conrelid	oid	pg_class.oid	この制約が存在しているテーブルです。テーブル制約でなければ0です。
contypid	oid	pg_type.oid	この制約が存在しているドメインです。ドメイン制約でなければ0です。
conindid	oid	pg_class.oid	一意性制約、プライマリキー制約、外部キー制約、排他制約の場合、この制約をサポートするインデックス。さもなければゼロ。
confrelid	oid	pg_class.oid	外部キーであれば、参照されるテーブルです。そうでなければ0です。
confupdtype	char		外部キー更新アクションコード: a = no action, r = restrict, c = cascade, n = set null, d = set default
confdeltype	char		外部キー削除アクションコード: a = no action, r = restrict, c = cascade, n = set null, d = set default
confmatchtype	char		外部キーの一致型: f = full, p = partial, u = simple (指定なし)
conislocal	bool		この制約はリレーションでローカルに定義されています。制約はローカルに定義されていて同時に継承されます。
coninhcount	int4		この制約がもつ直系の先祖の数。先祖の数がゼロではない制約は削除や改名はできません。
connoinherit	bool		この制約はリレーションのためにローカルで定義されます。これは非継承制約です。
conkey	int2[]	pg_attribute.attnum	テーブル制約(外部キーを含みますが制約トリガは含みません)であれば、その制約によって制約される列のリスト
confkey	int2[]	pg_attribute.attnum	外部キーであれば、参照される列のリスト
conpfeqop	oid[]	pg_operator.oid	外部キーであれば、PK = FKの比較のための同値演算子のリスト
conppeqop	oid[]	pg_operator.oid	外部キーであれば、PK = PKの比較のための同値演算子のリスト
conffeqop	oid[]	pg_operator.oid	外部キーであれば、FK = FKの比較のための同値演算子のリスト
conexclp	oid[]	pg_operator.oid	排他制約の場合、列単位の排他演算子のリスト。
conbin	pg_node_tree		検査制約であれば、式の内部表現形式
consrc	text		検査制約であれば、人間が見てわかる形式の式

排他制約の場合、単純な列参照である制約要素でのみconkeyが有用です。その他の場合、conkeyはゼロであり、関連するインデックスは制約される式を調査して見つけなければなりません。（したがってインデックスではconkeyはpg_index.indkeyの内容と同じものを持ちます。）

注意: は参照されているオブジェクトが変更されても更新されません。例えば列名の変更を追跡しません。
検査制約を引き出すためには、このフィールドに信頼を寄せるよりも`pg_get_constraintdef()`を使うのが最善です。

注意: `pg_class.relchecks`はそれぞれのリレーションに対してこのテーブルで検出された検査制約の項目数と一致しなければなりません。

[前のページ](#)
pg_class

[ホーム](#)
[上に戻る](#)

[次のページ](#)
pg_collation

45.13. pg_collation

pg_collationカタログは利用可能な、SQL名とオペレーティングシステムのロケール分類との基本的な対応付けを行う照合順序を記述します。 [項22.2](#)を参照してください。

表 45-13. pg_collationの列

名前	型	参照先	説明
collname	name		照合順序の名前（名前空間および符号化方式で一意）
collnamespace	oid	pg_namespace.oid	この照合順序を含む名前空間のOID
collowner	oid	pg_authid.oid	照合順序の所有者
collencoding	int4		この照合順序を適用することができる符号化方式。任意の符号化方式で動作する場合は-1
collcollate	name		この照合順序オブジェクト用のLC_COLLATE
collctype	name		この照合順序オブジェクト用のLC_CTYPE

このカタログの一意キーは(collname, collnamespace)だけではなく(collname,collencoding, collnamespace)です。 PostgreSQLは通常、 collencoding が現在のデータベースの符号化方式または-1と一致しない照合順序をすべて無視します。 また、 collencoding = -1を持つ項目と名前が一致する新しい項目の作成は許されません。 したがって照合順序を識別するためには、カタログの定義に従うと一意ではない場合であっても、修飾されたSQL名称 (schema.name) を使用することで十分です。 このようにカタログを定義した理由は、今後そのクラスタで使用される可能性があるすべての符号化方式のための項目を保持できるようにしなければなりませんので、クラスタの初期化時にinitdbがシステムで利用可能なすべてのロケール用の項目でこのカタログにデータを投入するためです。

template0データベースでは、後でtemplate0から複製されるデータベースの符号化方式に一致するかもしれませんが、データベース符号化方式に一致しない符号化方式のための照合順を作成することが有用になるかもしれません。 現在これは手作業で行う必要があります。

45.14. pg_conversion

pg_conversionカタログは符号化方式変換プロシージャを記述します。 詳細は[CREATE CONVERSION](#)を参照してください。

表 45-14. pg_conversionの列

名前	型	参照先	説明
conname	name		変換名（名前空間内で一意）
connamespace	oid	pg_namespace.oid	この変換を含む名前空間のOID。
conowner	oid	pg_authid.oid	変換の所有者
conforencoding	int4		符号化元のID
contoencoding	int4		符号化先のID
conproc	regproc	pg_proc.oid	変換プロシージャ
condefault	bool		これがデフォルト変換である場合は真

45.15. pg_database

pg_databaseカタログには使用可能なデータベースの情報が格納されます。 データベースは[CREATE DATABASE](#)コマンドで作成されます。 いくつかのパラメータの詳細については[第21章](#)を参照してください。

ほとんどのシステムカタログとは異なり、pg_databaseはクラスタにおける全てのデータベースにわたって共有されます。 データベース毎に1つではなく、クラスタ毎にたった1つだけpg_databaseのコピーが存在します。

表 45-15. pg_databaseの列

名前	型	参照先	説明
datname	name		データベース名
datdba	oid	pg_authid.oid	データベースの所有者。通常はそのデータベースの作成者
encoding	int4		このデータベースの文字符号化方式。 (pg_encoding_to_char() で、この番号から符号化方式名称に変換できます。)
datcollate	name		このデータベースのLC_COLLATE
datctype	name		このデータベースのLC_CTYPE
datistemplate	bool		真であれば、このデータベースの複製として新規データベースを作成するようにCREATE DATABASEの"TEMPLATE"句の中でこのデータベースを使用できます。
datallowconn	bool		偽の時はこのデータベースには誰も接続できません。 これはtemplate0データベースが変更されることを防ぐために使用されます。
datconnlimit	int4		このデータベースに対する同時のコネクションの最大数を設定します。 -1は無制限を意味します。
datlastsysoid	oid		データベース最終のシステムOID。 pg_dumpでは特に有用です。
datfrozenxid	xid		このデータベースの中で、この値よりも前のトランザクションIDは、永遠 ("凍結") トランザクションIDを持つように変更されています。 これは、このデータベースに対して、トランザクションID周回を防ぎ、かつ、pg_clogを縮小させることを目的としたバキュームを行うかどうかを追跡するために使用されます。 これはテーブル毎のpg_class.relfrozenxid値の最小値になります。
dattablespace	oid	pg_tablespace.oid	データベース用のデフォルトテーブル空間。 このデータベース内でpg_class.reltablespaceがゼロであるすべてのテーブルは、このテーブル空間に格納されます。 特に、共有されていないすべてのシステムカタログはこのテーブル空間にあります。
datacl	aclitem[]		アクセス権限。 GRANT と REVOKE を参照してください。

45.16. pg_db_role_setting

pg_db_role_settingカタログはロールとデータベースの組み合わせ毎に、実行時設定変数に設定されるデフォルト値を記録します。

ほとんどのカタログとは異なり、pg_db_role_settingはクラスタのすべてのデータベースにまたがって共有されます。つまりクラスタにはpg_db_role_settingのコピーは1つしかありません。 データベース毎に1つではありません。

表 45-16. pg_db_role_settingの列

名前	型	参照先	説明
setdatabase	oid	pg_database.oid	この設定が適用されるデータベースのOID。データベース固有でなければゼロです。
setrole	oid	pg_authid.oid	この設定が適用されるロールのOID。ロール固有でなければゼロです。
setconfig	text[]		実行時設定パラメータのデフォルト。

45.17. pg_default_acl

pg_default_aclカタログには、新規に作成されたオブジェクトに割り当てられた初期権限が格納されます。

表 45-17. pg_default_aclの列

名前	型	参照先	説明
defaclrole	oid	pg_authid.oid	この項目に関連するロールのOID。
defaclnamespace	oid	pg_namespace.oid	この項目に関連する名前空間のOID。何もない場合はゼロです。
defaclobjtype	char		この項目のオブジェクト種類。 <code>r</code> = リレーション(テーブル、ビュー)、 <code>s</code> = シーケンス、 <code>f</code> = 関数、 <code>T</code> = 型
defaclacl	aclitem[]		この種類のオブジェクトが作成時に保持しなければならないアクセス権限。

pg_default_aclの項目は、指示されたユーザに属するオブジェクトに割り当てられる初期権限を示します。 現在2種類の項目があります。 defaclnamespace = 0を持つ"大域的"な項目と特定のスキーマを参照する"スキーマ単位"の項目です。 大域的な項目が存在する場合、その種類のオブジェクトの通常の組み込まれたデフォルト権限を上書きします。 もしスキーマ単位の項目があれば、それは大域的な権限または組み込まれたデフォルト権限に追加される権限を表します。

他のカタログ内のACL項目がNULLの場合、その時のpg_default_acl内のものではなくそのオブジェクトの組み込まれたデフォルト権限を表すものが取られます。 pg_default_aclはオブジェクトの生成時のみに考慮されます。

45.18. pg_depend

pg_dependカタログは、データベースオブジェクト間の依存関係を記録します。この情報によってDROPコマンドが、他のどのオブジェクトをDROP CASCADEで削除する必要があるか、また、DROP RESTRICTで削除を防止するかの場合を判断します。

pg_shdependも参照してください。これはデータベースクラスタ間で共有されるオブジェクトの依存関係に対する似たような機能を持っています。

表 45-18. pg_dependの列

名前	型	参照先	説明
classid	oid	pg_class.oid	依存するオブジェクトを含んだシステムカタログのOID
objid	oid	any OID column	特定の依存するオブジェクトのOID
objsubid	int4		テーブル列の場合、これは列番号です（objidとclassidはテーブル自身を参照します）。他のすべての種類のオブジェクトでは、このフィールドはゼロです。
refclassid	oid	pg_class.oid	参照されるオブジェクトが存在するシステムカタログのOID
refobjid	oid	any OID column	特定の参照されるオブジェクトのOID
refobjsubid	int4		テーブル列の場合、これは列番号です（refobjidとrefclassidはテーブル自身を参照します）。他のすべての種類のオブジェクトでは、このフィールドはゼロです。
deptype	char		この依存関係の特定のセマンティクスを定義するコード（後述）。

すべての場合において、pg_dependエントリは依存するオブジェクトも削除しない限り、参照されるオブジェクトを削除できないことを示します。もっとも、deptypeによって指定される以下のようないくつかのオプションもあります。

DEPENDENCY_NORMAL (n)

個別に作成されたオブジェクト間の通常の関係です。依存するオブジェクトは参照されるオブジェクトに影響を与えずに削除できます。参照されるオブジェクトはCASCADEを指定することによってのみ削除することができます。この場合は依存するオブジェクトも削除されます。例：テーブルの列はそのデータ型に対して通常の依存関係を持ちます。

DEPENDENCY_AUTO (a)

依存するオブジェクトは参照されるオブジェクトから独立して削除することができます。そして、参照されるオブジェクトが削除される時は（RESTRICTもしくはCASCADEモードに関わりなく）依存するオブジェクトも自動的に削除されなければなりません。例：テーブル上の名前付き制約はテーブル上に自動設定されているため、テーブルが削除されるとなくなります。

DEPENDENCY_INTERNAL (i)

依存するオブジェクトは参照されるオブジェクトの作成時に作成されたもので、実際に内部実装の一部に過ぎません。依存するオブジェクトに対してDROPコマンドを直接的に実行できません（その代わりに、参照されるオブジェクトに対してDROPを実行するように指示されます）。参照されるオブジェクトにDROPを実行すると、CASCADEが指定されているかどうかに関わらず、依存するオブジェクトも削除されます。例：外部キー制約を強制するために作成されたトリガは内部的に制約のpg_constraint項目に依存しています。

DEPENDENCY_EXTENSION (e)

依存するオブジェクトは参照されるオブジェクトのextensionのメンバです（[pg_extension](#)参照）。依存するオブジェクトは参照されるオブジェクトに対するDROP EXTENSION経由でのみ削除することができます。機能的にはこの種類の依存関係は内部依存関係と同様に動作しますが、明確さとpg_dumpを単純化するために別々に保持されます。

DEPENDENCY_PIN (p)

依存するオブジェクトはありません。この種類の項目は、参照されるオブジェクトにシステム自体が依存し、したがってオブジェクトを絶対に削除してはならないことを示します。この種類の項目はinitdbの実行時にのみ作成されます。依存するオブジェクト列にはゼロが含まれます。

将来的に、他の依存関係のオプションが必要になる可能性があります。

45.19. pg_description

各データベースオブジェクトに対して付けられたオプションの補足説明（コメント）はpg_descriptionカタログに格納されます。 補足説明はCOMMENTコマンドで編集でき、psqlの\dコマンドで表示できます。 多くの組み込み型のシステムオブジェクトの説明は、pg_descriptionの最初の部分で提供されています。

pg_shdescriptionも参照してください。 こちらは、データベースクラスタに渡って共有されるオブジェクトに関する説明について、同様の機能を行います。

表 45-19. pg_descriptionの列

名前	型	参照先	説明
objoid	oid	any OID column	この補足説明が属するオブジェクトのOID
classoid	oid	pg_class.oid	このオブジェクトが現れるシステムカタログのOID
objsubid	int4		テーブル列についてのコメントの場合、これは列の（objoidもしくはclassoidはテーブル自身を参照します）列番号です。 他のすべての種類のオブジェクトでは、このフィールドはゼロです。
description	text		このオブジェクトの説明となる任意のテキスト

45.20. pg_enum

pg_enumカタログは、各列挙型についてその値とラベルを示す項目を含みます。 ある与えられた列挙値の内部表現は、実際にはpg_enum内の関連付けられた行のOIDです。

表 45-20. pg_enumの列

名前	型	参照先	説明
enumtypid	oid	pg_type.oid	この列挙値を所有しているpg_type項目のOID
enumsortorder	float4		列挙型におけるこの列挙値のソート位置
enumlabel	name		この列挙値のテキストラベル

pg_enum行のOIDは次のような特別な規則に従います。 列挙型のソート順序と同じ順序で並んだ偶数のOIDが保証されています。 つまり、2つの偶数のOIDが同じ列挙型に属する場合、OIDの小さい方がより小さいenumsortorder値を持たなければなりません。 奇数のOID値はソート順序に関連を持つてはなりません。 この規則により列挙の比較処理は多くの一般的な場合でカタログの検索を防ぐことができます。 列挙型の作成および変更を行う処理は、可能であれば偶数のOIDを列挙値に割り当てようとしています。

列挙型を作成する時、そのメンバには1..*n*のソート順位置が割り当てられます。 しかし後で追加されたメンバには負もしくはenumsortorderの小数值が与えられる可能性があります。 これらの値の要件は、各列挙型の中で正しく順序付けされ、かつ一意であることです。

45.21. pg_extension

pg_extensionカタログにはインストールされた拡張に関する情報が格納されます。 拡張の詳細については[項35.15](#)を参照してください。

表 45-21. pg_extensionの列

名前	型	参照先	説明
extname	name		拡張の名前
extowner	oid	pg_authid.oid	拡張の所有者
extnamespace	oid	pg_namespace.oid	拡張が提供するオブジェクトを含むスキーマ
extrelocatable	bool		拡張が他のスキーマに再配置可能である場合真
extversion	text		拡張のバージョン名
extconfig	oid[]	pg_class.oid	拡張の設定テーブルのregclass OIDの配列。なければNULL
extcondition	text[]		拡張の設定テーブル用のWHERE句フィルタ条件の配列。なければNULL

ほとんどの"名前空間"に関する列を持つカタログと異なり、extnamespaceは、拡張がそのスキーマに属することを意図したものではありません。 拡張の名前は決してスキーマで修飾されません。 extnamespaceは、拡張のオブジェクトのすべて、あるいは、ほとんどを含むスキーマを示します。 extrelocatableが真の場合、このスキーマは拡張に属するすべてのスキーマ修飾可能なオブジェクトを含まなければなりません。

45.22. pg_foreign_data_wrapper

pg_foreign_data_wrapperカタログは外部データラップの定義を保存します。 外部データラップは外部サーバにあるデータにアクセスするための機構です。

表 45-22. pg_foreign_data_wrapperの列

名前	型	参照先	説明
fdwname	name		外部データラップの名前
fdwowner	oid	pg_authid.oid	外部データラップの所有者
fdwhandler	oid	pg_proc.oid	外部データラップに対する実行関数の提供に責任を持つハンドラ関数への参照。 ハンドラ関数がない場合はゼロになります。
fdwvalidator	oid	pg_proc.oid	外部サーバや外部データラップを使用するユーザマップと同様に外部データラップに対して与えられたオプションの正当性を検査する有効性検証関数。 有効性検証関数がない場合はゼロになります。
fdwacl	aclitem[]		アクセス権限。 詳細は GRANT と REVOKE を参照してください。
fdwoptions	text[]		外部データラップの"keyword=value"のような特定のオプション。

45.23. pg_foreign_server

pg_foreign_serverカタログは外部サーバの定義を保存します。 外部サーバはリモートサーバなど外部データの源を記述します。 外部サーバは外部データラッパを介してアクセスされます。

表 45-23. pg_foreign_serverの列

名前	型	参照先	説明
srvname	name		外部サーバの名前
srvowner	oid	pg_authid .oid	外部サーバの所有者
srvfdw	oid	pg_foreign_data_wrapper .oid	外部サーバの外部データラッパのOID
srvtype	text		サーバの型 (オプション)
srvversion	text		サーバのバージョン (オプション)
srvacl	aclitem[]		アクセス権限。 詳細は GRANT と REVOKE を参照してください。
srvoptions	text[]		外部サーバの"keyword=value"のような特定のオプション。

45.24. pg_foreign_table

pg_foreign_tableカタログには、外部テーブルに関する補助情報が含まれます。 外部テーブルは主にpg_classの項目により表されます。 pg_foreign_tableの項目には、外部テーブルに属する情報のみに関する情報が含まれ、他の種類のリレーションは含まれません。

表 45-24. pg_foreign_tableの列

名前	型	参照先	説明
ftrelid	oid	pg_class.oid	この外部テーブルに対するpg_class項目のOID
ftserver	oid	pg_foreign_server.oid	この外部テーブルに対する外部サーバOID
ftoptions	text[]		"keyword=value"文字列のような、外部テーブルのオプション

45.25. pg_index

pg_indexカタログはインデックス情報の一部を保持します。 その他のほとんどの情報はpg_classにあります。

表 45-25. pg_indexの列

名前	型	参照先	説明
indexrelid	oid	pg_class.oid	このインデックスに対するpg_class項目のOID
indrelid	oid	pg_class.oid	このインデックスが使われるテーブルに対するpg_class項目のOID
indnatts	int2		インデックス内の列数 (pg_class.relnattsの複製)
indisunique	bool		真の場合は一意性インデックス
indisprimary	bool		真の場合このインデックスはテーブルのプライマリキーを表します (この値が真の時indisuniqueは常に真でなければなりません)。
indisexclusion	bool		真の場合、このインデックスは排他制約をサポートします。
indimmediate	bool		真の場合、一意性検査が挿入時即座に強制されます。(indisuniqueが真でなければ無関係です。)
indisclustered	bool		真の場合、前回このインデックスを元にテーブルはクラスタ化されました。
indisvalid	bool		このフラグとindisreadyが真の場合、現在このインデックスは問い合わせに対して有効です。 偽は、インデックスが不完全かもしれないことを意味します。 INSERT/UPDATE操作による変更が行われているはずで、問い合わせに使用するには安全ではありません。 一意性インデックスであれば、一意性も保証されません。
indcheckxmin	bool		真の場合、pg_index行のxminがTransactionXminイベント境界値を下回るまで、問い合わせはインデックスを使用してはいけません。 なぜなら、テーブルは互換性の無い行と共に破壊されたHOTチェーンを含み、それらが可視であるかもしれないからです。
indisready	bool		真の場合、インデックスは挿入に対する準備ができています。 偽の場合はインデックスはINSERT/UPDATE操作により無視されなければならないことを意味します。
indkey	int2vector	pg_attribute.attnum	このインデックスがどのテーブル列をインデックスとしているかを示すindnatts配列の値です。 例えば、1 3は1番目と3番目のテーブル列がインデックスキーとなっていることを示します。 この配列でゼロとなっているのは対応するインデックスの属性が単純な列参照ではなくテーブル列に渡った演算式であることを示します。
indcollation	oidvector	pg_collation.oid	インデックスキー内の各列に関してここにはインデックスで使用される照合順序のOIDが含まれます。
indclass	oidvector	pg_opclass.oid	インデックスキー内のそれぞれの列に対して、使用する演算子クラスのOIDを保持します。 pg_opclass を参照してください。
indoption	int2vector		列毎のフラグビットを格納するindnatts値の配列です。 ビットの意味はインデックスのアクセスメソッドによって定義されています。
indexprs	pg_node_tree		単純な列参照でないインデックス属性の (nodeToString() 表現による) 演算式ツリー。 単純な参照の場合はインデックス属性はすべてNULL。
indpred	pg_node_tree		部分インデックス属性の (nodeToString() 表現による) 演算式ツリー。 部分インデックスでなければNULL。

45.26. pg_inherits

pg_inherits カタログは継承階層の情報を記録します。 データベース内の、それぞれの直接の子テーブルに対して1つの記述があります（直接ではない継承は、記述の連鎖から決定されます）。

表 45-26. pg_inheritsの列

名前	型	参照先	説明
inhrelid	oid	pg_class.oid	子テーブルのOID。
inhparent	oid	pg_class.oid	親テーブルのOID。
inhseqno	int4		1つの子テーブルに1つ以上の親テーブルが存在する場合（複数継承）、この番号の順序に従って継承された列が配置されます。 番号は1から始まります。

45.27. pg_language

pg_languageカタログはユーザ定義関数やストアードプロシージャを作成することができる言語を登録します。言語ハンドラの詳細は[CREATE LANGUAGE](#)と[第38章](#)を参照してください。

表 45-27. pg_languageの列

名前	型	参照先	説明
lanname	name		言語名
lanowner	oid	pg_authid.oid	言語の所有者
lanispl	bool		(SQLなどの) 内部言語では偽となり、ユーザ定義言語では真となります。現在pg_dumpではダンプされる必要のある言語を決定するためにこれを使用していますが、いずれ別の処理方法に置き換えられる可能性があります。
lanpltrusted	bool		信頼できる言語である場合は真です。これは通常のSQL実行環境の外部に対してまったくアクセスを許可しないことを意味しています。スーパーユーザだけが信頼できない言語で関数を作成することができます。
lanplcallfoid	oid	pg_proc.oid	内部言語でない場合、特定の言語で作成されたすべての関数の実行に責任を持つ特別な関数である言語ハンドラを参照します。
laninline	oid	pg_proc.oid	これは"インライン"匿名コードブロック(DOブロック)の実行に責任を持つ関数を参照します。インラインブロックをサポートしない場合はゼロ。
lanvalidator	oid	pg_proc.oid	これは、新しい関数が作成された時に構文や有効性の検査を引き受ける言語有効性検査関数を参照します。有効性検査関数がない場合はゼロになります。
lanacl	aclitem[]		アクセス権限。 GRANT と REVOKE を参照してください。

45.28. pg_largeobject

pg_largeobject カタログは"ラージオブジェクト"を構築するデータを保持します。ラージオブジェクトは作成された時に割り当てられたOIDで識別されます。それぞれのラージオブジェクトはpg_largeobjectの行に都合良く格納されるのに十分に足る小さなセグメント、もしくは"ページ"に分割されます。ページごとのデータ量は（現在BLCKSZ/4あるいは典型的に2キロバイトの）LOBLKSIZEとして定義されます。

PostgreSQL 9.0より前までは、ラージオブジェクトに関連した権限構造はありませんでした。その結果pg_largeobjectは可読性が高いもので、システム内のすべてのラージオブジェクトのOIDを入手するために使用することができました。これはもはや当てはまりません。ラージオブジェクトのOIDのリストを入手するためにはpg_largeobject_metadataを使用してください。

表 45-28. pg_largeobjectの列

名前	型	参照先	説明
loid	oid	pg_largeobject_metadata.oid	このページを含んだラージオブジェクトの識別子
pageno	int4		ラージオブジェクト内の（ゼロから数えた）このページのページ番号
data	bytea		ラージオブジェクト内に保存された実データ。LOBLKSIZEバイトを絶対上回りません。たぶんそれより小さいでしょう。

pg_largeobjectのそれぞれの行はオブジェクト内のバイトオフセット（pageno * LOBLKSIZE）から始まるラージオブジェクトの1ページ分のデータを保持します。ページが見つからなかったり、たとえオブジェクトの最後のページでない場合でもLOBLKSIZEより小さくてもよいといった、あちこちに散らばって保存されてもよいような実装になっています。ラージオブジェクトの中で見つからない部分はゼロとして読み込まれます。

45.29. pg_largeobject_metadata

pg_largeobject_metadataはラージオブジェクトに関連したメタデータを保持します。 実際のラージオブジェクトデータはpg_largeobjectに格納されます。

表 45-29. pg_largeobject_metadataの列

名前	型	参照先	説明
lomowner	oid	pg_authid.oid	ラージオブジェクトの所有者
lomacl	aclitem[]		アクセス権限。 GRANT および REVOKE を参照してください。

45.30. pg_namespace

pg_namespace カタログは名前空間を保存します。 名前空間はSQLスキーマの裏にある構造です。 それぞれの名前空間は、リレーション、型などの集合を、名前が競合することなく、個別に持ちます。

表 45-30. pg_namespaceの列

名前	型	参照先	説明
nspname	name		名前空間の名前
nspowner	oid	pg_authid.oid	名前空間の所有者
nspacl	aclitem[]		アクセス権限。 GRANT と REVOKE を参照してください。

45.31. pg_opclass

pg_opclass カタログはインデックスアクセスメソッド演算子クラスを定義します。それぞれの演算子クラスは特定のデータ型のインデックス列のセマンティクスと特定のインデックスアクセスメソッドを定義します。演算子クラスは、ある特定の演算子族は特定のインデックス可能な列データの型に対して適用できる、ということを本質的に特定します。インデックス付けされた列を実際に使用可能な演算子族の演算子群は、その列のデータ型を左辺の入力として受け付けます。

演算子クラスについては[項35.14](#)に詳細に説明されています。

表 45-31. pg_opclass の列

名前	型	参照先	説明
opcmethod	oid	pg_am.oid	対象のインデックスアクセスメソッド演算子クラス
opcname	name		この演算子クラスの名前
opcnamespace	oid	pg_namespace.oid	この演算子クラスの名前空間
opcowner	oid	pg_authid.oid	演算子クラスの所有者
opcfamily	oid	pg_opfamily.oid	演算子クラスを含んでいる演算子族
opcintype	oid	pg_type.oid	演算子クラスがインデックスを作成するデータ型
opcdefault	bool		演算子クラスがopcintypeのデフォルトである場合は真
opckeytype	oid	pg_type.oid	インデックス内に格納されているデータ型。opcintypeと同じ場合はゼロ

演算子クラスのopcmethodは、演算子クラスを含んでいる演算子族のopfmethodに一致しなければいけません。また、任意のopcmethodとopcintypeの組み合わせに対してopcdefaultが真となるようなpg_opclass行が複数存在してはいけません。

45.32. pg_operator

pg_operatorカタログは演算子の情報を保存します。 [CREATE OPERATOR](#)と[項35.12](#)を参照してください。

表 45-32. pg_operatorの列

名前	型	参照先	説明
oprname	name		演算子名
oprnamespace	oid	pg_namespace.oid	この演算子を含む名前空間のOID
oprowner	oid	pg_authid.oid	演算子の所有者
oprkind	char		b = 挿入辞 ("両側")、 l = 接頭辞 ("左側")、 r = 接尾辞 ("右側")
oprcanmerge	bool		この演算子はマージ結合をサポートします。
oprcanhash	bool		この演算子はハッシュ結合をサポートします。
oprleft	oid	pg_type.oid	左辺オペランドの型
oprright	oid	pg_type.oid	右辺オペランドの型
oprresult	oid	pg_type.oid	結果の型
oprcom	oid	pg_operator.oid	もし存在すればこの演算子の交代演算子
oprnegate	oid	pg_operator.oid	もし存在すればこの演算子の否定子
oprcode	regproc	pg_proc.oid	この演算子を実装する関数
oprrest	regproc	pg_proc.oid	この演算子の制約選択評価関数
oprjoin	regproc	pg_proc.oid	この演算子の結合選択評価関数

未使用の列にはゼロが入ります。 例えば、接頭辞演算子ではoprleftはゼロとなります。

45.33. pg_opfamily

pg_opfamilyカタログは演算子族を定義します。それぞれの演算子族は、演算子とサポートルーチン(特定のインデックスアクセスメソッドのために特化されたセマンティクスを実装するような関連付けられたもの)を集めたものです。さらに、演算子族内の演算子はすべて、アクセスメソッドにより特定される方法において"互換性"があります。演算子族の概念は、データ型を跨る演算子がインデックスで使用されることを許可し、さらにアクセスメソッドのセマンティクスの知識を使用することについて理由付けすることも許可します。

演算子族については[項35.14](#)で詳しく説明します。

表 45-33. pg_opfamilyの列

名前	型	参照先	説明
opfmethod	oid	pg_am .oid	対象のインデックスアクセスメソッド演算子族
opfname	name		演算子族の名称
opfnamespace	oid	pg_namespace .oid	演算子族の名前空間
opfowner	oid	pg_authid .oid	演算子族の所有者

演算子族を定義している情報の大部分が、pg_opfamily行にあるわけではなく、[pg_amop](#)や[pg_amproc](#)や[pg_opclass](#)行にあります。

45.34. pg_pltemplate

pg_pltemplateカタログは手続き言語の"テンプレート"に関する情報を格納します。 手続き言語のテンプレートを使用することで、特定のデータベース内で単純なCREATE LANGUAGEコマンドにより言語が作成できます。 実装の詳細を指定する必要はありません。

多くのシステムカタログと異なりpg_pltemplateはクラスタの全てのデータベースに共有されています。 データベース毎ではなく、クラスタ毎にただ1つのpg_pltemplateのコピーがあります。 これにより、それぞれのデータベースは、必要とされたときにデータにアクセスできます。

表 45-34. pg_pltemplateの列

名前	型	説明
tmplname	name	このテンプレートのための言語の名前
tmpltrusted	boolean	言語が信頼されている場合は真
tmpldbcreate	boolean	言語がデータベース所有者により作成されているかもしれない場合は真
tmplhandler	text	呼び出しハンドラ関数の名前
tmplinline	text	匿名ブロックハンドラ関数の名前。存在しない場合はNULL。
tmplvalidator	text	有効性検出関数の名前。無い場合はNULL
tmpllibrary	text	言語を実装している共有ライブラリのパス
tmplacl	aclitem[]	テンプレートのアクセス権限（実際には未使用）

手続き言語のテンプレートを扱うコマンドは、今のところありません。 組み込まれた情報を変えるためにはスーパーユーザが、INSERT、DELETEもしくはUPDATEを使用してテーブルを変更する必要があります。

注意: 手続き言語に関するこの情報を対応する拡張インストール用スクリプトで保持することを優先させ、今後のPostgreSQLのリリースでpg_pltemplateは削除される予定です。

45.35. pg_proc

pg_procカタログは関数（またはプロシージャ）に関する情報を格納します。 [CREATE FUNCTION](#)と[項35.3](#)を参照してください。

このテーブルには、普通の関数のみならず集約関数も含まれます。 proisaggが真ならpg_aggregateに一致する行があるはずで

表 45-35. pg_procの列

名前	型	参照先	説明
proname	name		関数名
pronamespace	oid	pg_namespace.oid	この関数を含む名前空間のOID
proowner	oid	pg_authid.oid	関数の所有者
prolang	oid	pg_language.oid	この関数の実装言語または呼び出しインタフェース
procost	float4		推定実行コスト(cpu_operator_cost 単位です)。 proretsetの場合は、返される行毎のコストになります。
prorows	float4		結果の推定行数(proretset でなければゼロになります)
provariadic	oid	pg_type.oid	可変配列パラメータの要素のデータ型。関数が可変パラメータを持っていない場合はゼロになります。
protransform	regproc	pg_proc.oid	この関数呼び出しは他の関数により簡略化できます。 (項35.9.11 を参照してください)
proisagg	bool		関数は集約関数です。
proiswindow	bool		関数はウィンドウ関数です。
prosecdef	bool		セキュリティ定義の関数（すなわち"setuid"関数）
proleakproof	bool		この関数には副作用がありません。引数に関する情報が戻り値以外から伝わることはありません。 引数の値に依存するエラーを発生する可能性がある関数はすべてリークプルーフ関数ではありません。
proisstrict	bool		関数は呼び出し引数がNULLの場合にはNULLを返します。 その場合、関数は実際にはまったく呼び出されません。 "厳密"ではない関数はNULL値入力を取り扱えるようにしなければいけません。
proretset	bool		集合（すなわち指定されたデータ型の複数の値）を返す関数
provolatile	char		provolatileは、関数の結果が入力引数のみで決定されるか、または外部要素に影響されるかを示します。 "immutable"関数を表すiは同じ入力に対し常に同じ結果をもたらします。 "stable"関数を表すsは（固定入力に対する）結果はスキャン内で変わりません。 "volatile"関数を表すvは常に異なる結果を出す可能性があります（また、副作用を持つ関数にvを使用することで、その関数に対する呼び出しが最適化されないようにすることができます）。
pronargs	int2		入力の引数の数
pronargdefaults	int2		デフォルト値を持つ引数の数
prorettype	oid	pg_type.oid	戻り値のデータ型
proargtypes	oidvector	pg_type.oid	関数の引数のデータ型を格納した配列。 これは入力引数（INOUTとVARIADICも含みます）のみを含んでいて、関数の呼び出しシグネチャを表現します。
proallargtypes	oid[]	pg_type.oid	関数の引数のデータ型を格納した配列。 これは（OUTとINOUT引数を含んだ）全ての引数を含みます。 しかし、すべての引数がINであった場合は、この列はNULLになります。 歴史的な理由からproargtypesは0から番号が振られていますが、 添字は1から始まっていることに注意してください。
proargmodes	char[]		関数の引数のモードを格納した配列。 以下のように符号化されています。 IN引数に対してはi、 OUT引数に対してはo、 INOUT引数に対してはb、 VARIADIC引数に対してはv、 TABLE引数に対してはt。 もしすべての引数

			がIN引数であった場合は、この列はNULLとなります。 添字はproargtypesではなくproallargtypesの位置に対応していることに注意してください。
proargnames	text[]		関数の引数名を格納する配列。 名前のない引数は、配列内では空文字列で設定されます。 もしすべての引数に名前がない場合は、この列はNULLとなります。 添字はproargtypesではなくproallargtypesの位置に対応していることに注意してください。
proargdefaults	pg_node_tree		デフォルト値のための(nodeToString() 表現の)演算式ツリー。 これはpronargdefaultsの要素のリストで、最後のM個の入力引数と対応しています（つまり最後のN proargtypesの位置ということです）。 もし引数にデフォルト値がない場合は、この列はNULLになります。
prosrc	text		関数の起動方法を関数ハンドラに伝えます。 実装言語や呼び出し規約に依存して、使用する言語用の関数の実際のソースコード、リンクシンボル、ファイル名などになります。
probin	text		関数の起動方法についての追加情報。 同じように解釈は言語に依存します。
proconfig	text[]		実行時の設定変数に対する関数のローカル設定
proacl	aclitem[]		アクセス権限。 GRANT と REVOKE を参照してください。

コンパイル言語で作成された、組み込みおよび動的にロードされる関数では、prosrcは関数のC言語名（リンクシンボル）を持ちます。 他の種類の言語はすべて、prosrcは関数のソーステキストを持ちます。 probinは動的にロードされるC関数に対してその関数を保有する共有ライブラリファイルの名前を与える以外には使用されていません。

45.36. pg_range

pg_rangeカタログは、範囲型についての情報を保存します。これはpg_type内の型のエントリに追加されます。

表 45-36. pg_rangeの列

名前	型	参照先	説明
rngtypid	oid	pg_type.oid	範囲型のOID
rngsubtype	oid	pg_type.oid	この範囲型の要素型(派生元型)のOID
rngcollation	oid	pg_collation.oid	範囲比較のために使用される照合のOID。何もない場合はゼロです。
rngsubopc	oid	pg_opclass.oid	範囲比較のために使用される派生元型の演算子クラスのOID
rngcanonical	regproc	pg_proc.oid	範囲型を標準型に変換する関数のOID。何もない場合はゼロです。
rngsubdiff	regproc	pg_proc.oid	2つの要素値の間の違いをdouble precisionとして返す関数のOID、なければゼロ

rngsubopc (および、要素型が照合可能である場合はrngcollation)は 範囲型で使用されるソートの順番を決定します。rngcanonicalは要素型が離散的である場合に使用されます。rngsubdiffは省略可能ですが、範囲型に対するGiSTインデックスの性能を向上するためには提供しなければなりません。

45.37. pg_rewrite

pg_rewriteカタログはテーブルとビューに対する書き換えルールを保存します。

表 45-37. pg_rewriteの列

名前	型	参照先	説明
rulename	name		ルール名
ev_class	oid	pg_class.oid	ルールを適用するテーブル
ev_attr	int2		ルールを適用する列（現在は常にテーブル全体を表すゼロ）
ev_type	char		ルールを適用するイベントの型： 1 = SELECT、 2 = UPDATE、 3 = INSERT、 4 = DELETE
ev_enabled	char		ルールがどの session replication role モードで発行されるかを制御します。 O = ルールは"origin"および"local"モードで発行。 D = ルールは無効。 R = ルールは"replica"モードで発行。 A = ルールは常に発行。
is_instead	bool		ルールがINSTEADルールの場合は真
ev_qual	pg_node_tree		ルールの制約条件に対する（nodeToString() 表現による）演算式ツリー
ev_action	pg_node_tree		ルールのアクションに対する（nodeToString() 表現による）問い合わせツリー

注意: テーブルがこのカタログ内のルールを持つ場合、pg_class.relhasrulesは真でなければなりません。

45.38. pg_seclabel

pg_seclabelカタログにはデータベースオブジェクト上のセキュリティラベルが格納されます。セキュリティラベルは[SECURITY LABEL](#)コマンドを用いて操作することができます。セキュリティラベルを閲覧するより簡単な方法については[項45.63](#)を参照してください。

[pg_shseclabel](#)を参照してください。これは、データベースクラスタ間で共有されたデータベースオブジェクトにおけるセキュリティラベルのための類似した機能を提供します。

表 45-38. pg_seclabelの列

名前	型	参照先	説明
objoid	oid	任意のOID列	このセキュリティラベルが付属するオブジェクトのOID
classoid	oid	pg_class .oid	このオブジェクトが出現するシステムカタログのOID
objsubid	int4		テーブル列上のセキュリティラベルでは、これは列番号です（objoidおよびclassoidはテーブル自身を参照します）。他のすべての種類のオブジェクトでは、この列はゼロです。
provider	text		このラベルに関連付いたラベルプロバイダです。
label	text		このオブジェクトに適用されるセキュリティラベルです。

45.39. pg_shdepend

pg_shdependカタログは、データベースオブジェクトとロールのような共有オブジェクト間のリレーション依存関係を保持します。この情報はPostgreSQLが依存関係を削除しようとする前に、これらのオブジェクトを参照されないようにすることを保証することを許可します。

[pg_depend](#)を参照してください。これは単一のデータベースのオブジェクトに関する依存関係に対する似たような機能を持ちます。

多くのシステムカタログと異なりpg_shdependはクラスタの全てのデータベースに共有されています。データベース毎ではなく、クラスタ毎にただ1つのpg_shdependのコピーがあります。

表 45-39. pg_shdependの列

名前	型	参照先	説明
dbid	oid	pg_database.oid	依存するオブジェクトが格納されたデータベースのOID。共有オブジェクトではゼロ
classid	oid	pg_class.oid	依存するオブジェクトが格納されたシステムカタログのOID
objid	oid	any OID column	特定の依存するオブジェクトのOID
objsubid	int4		テーブル列の場合、これは列番号です(objidとclassidはテーブル自身を参照します)。他のすべての種類のオブジェクトでは、この列はゼロになります。
refclassid	oid	pg_class.oid	参照されるオブジェクトが入っているシステムカタログのOID（共有カタログである必要があります）
refobjid	oid	任意のOID列	特定の参照されるオブジェクトのOID
deptype	char		この依存関係の特定のセマンティクスを定義するコード（後述）。

すべての場合において、pg_shdepend項目は依存するオブジェクトも削除しない限り、参照されるオブジェクトを削除できないことを示します。もっとも、deptypeによって指定される以下のようないくつかのオプションもあります。

SHARED_DEPENDENCY_OWNER (o)

参照されるオブジェクト（ロールである必要があります）が依存するオブジェクトの所有者です。

SHARED_DEPENDENCY_ACL (a)

参照されたオブジェクト（ロールである必要があります）が、依存するオブジェクトのACL（アクセス制御リスト。権限リストのこと）内で述べられています。（所有者はSHARED_DEPENDENCY_OWNER項目を持つため、SHARED_DEPENDENCY_ACL項目は、オブジェクトの所有者に対して作成されません。）

SHARED_DEPENDENCY_PIN (p)

依存するオブジェクトはありません。この種類の項目は、システム自体が参照されるオブジェクトに依存している記号です。よってオブジェクトは削除されてはいけません。この種類の項目はinitdbによってのみ作成されます。依存するオブジェクトの列はゼロを含んでいます。

他の依存関係のオプションが将来必要になる可能性があります。現状の定義は、参照されるオブジェクトとしてロールのみをサポートしていることに特に注意してください。

45.40. pg_shdescription

pg_shdescriptionには共有データベースオブジェクトに対する補足説明（コメント）を格納します。補足説明は[COMMENT](#)コマンドを使用して編集でき、psqlの\dコマンドを使用して閲覧することができます。

[pg_description](#)も参照してください。こちらは、単一データベース内のオブジェクトに関する説明について、同様の機能を行います。

他のシステムカタログと異なり、pg_shdescriptionはクラスタ内のすべてのデータベースに渡って共有されます。データベース毎に存在するのではなく、1つのクラスタにpg_shdescriptionが1つのみ存在します。

表 45-40. pg_shdescriptionの列

名前	型	参照先	説明
objoid	oid	任意のOID列	補足説明が属するオブジェクトのOID
classoid	oid	pg_class .oid	このオブジェクトが存在するシステムカタログのOID
description	text		このオブジェクトの補足説明として提供される任意のテキスト

45.41. pg_shseclabel

pg_shseclabelカタログは、共有データベースオブジェクト上のセキュリティラベルを保存します。セキュリティラベルはSECURITY LABELコマンドで操作されます。セキュリティラベルを見る簡単な方法は、[項45.63](#)を参照してください。

[pg_seclabel](#)も参照してください。これは、単一データベース内のオブジェクトを含むセキュリティラベルのための類似した機能を提供します。

多くのシステムカタログと違い、pg_shseclabelはクラスタ内の全てのデータベース間で共有されます。pg_shseclabelは、データベースごとではなく、クラスタごとに1つのみ存在します。

表 45-41. pg_shseclabel の列

名前	型	参照先	説明
objoid	oid	任意のOID列	このセキュリティラベルが属しているオブジェクトのOID
classoid	oid	pg_class .oid	このオブジェクトが出現するシステムカタログのOID
provider	text		このラベルに関連付けられたラベルプロバイダ
label	text		このオブジェクトに適用されたセキュリティラベル

45.42. pg_statistic

pg_statisticカタログはデータベースの内容に関する統計データを保存します。項目はANALYZEで作成され、後に問い合わせプランナで使用されます。最新のものと思ってもすべての統計データは本質的に大雑把なものであることに注意してください。

通常は、解析されるテーブル列毎に、stainherit = falseを持つ1つの項目が存在します。テーブルが継承された子を持つ場合、stainherit = trueを持つ2つ目の項目が作成されます。この行は継承ツリー全体に渡る列の統計情報、つまり、SELECT column FROM table*で確認できるデータに対する統計情報を表します。一方でstainherit = falseの行はSELECT column FROM ONLY tableの結果を表します。

pg_statisticはインデックス式の値についての統計データも格納します。これらはあたかも値が実際のデータ列であるかのように表現されます。特にstarelidはインデックスを参照します。これは元のテーブル列の項目に対して冗長となるので、普通の式を持たないインデックス列では項目は作成されません。現在インデックス式用の項目は常にstainherit = falseを持ちます。

異なる種類のデータに対しては違った種類の統計が相応しいことからpg_statisticはどのような情報を保存するか深く推定しないように設計されています。（例えばNULLであるような）極端に一般的な統計のみpg_statisticの特定の列に入ります。その他すべてはスロット列の内の1つのコード番号でその内容が識別される関連している列のグループである"スロット"に保存されます。src/include/catalog/pg_statistic.hを参照してください。

pg_statisticはテーブル内容に関する統計情報と言えども秘密の情報とみなされますので、一般のユーザが読み取り可能であってははいけません。（給与列の最高額と最低額などは誰もが興味をそそる良い例ですね。）pg_statsは一般のユーザが読み取り可能なpg_statisticに対するビューで、既存のユーザが読んでも差し支えないテーブルの情報のみを開示しています。

表 45-42. pg_statisticの列

名前	型	参照先	説明
starelid	oid	pg_class.oid	記述された列が属するテーブルもしくはインデックス
staattnum	int2	pg_attribute.attnum	記述された列数
stainherit	bool		真の場合、統計情報には指定されたテーブルの値だけではなく、継承関係の子の列が含まれます。
stanullfrac	float4		NULL値である列項目の割合
stawidth	int4		非NULL項目の平均保存幅（バイト単位）
stadistinct	float4		列内で非NULL個別値を持つデータ数。ゼロより大きい値は実際の個別値の数です。ゼロより小さい値はテーブル内の行数に対する負の乗数です。例えば、平均して2回ほど出現する値を持つ列はstadistinct = -0.5であると表現されます。ゼロは固有値を特定できない場合です。
stakindN	int2		pg_statistic行のN番目の"スロット"に保存されている統計情報の種類を示すコード番号。
staopN	oid	pg_operator.oid	N番目の"スロット"に保存されている統計情報を引き出すために使われる演算子。例えば、度数分布スロットはデータの並び換えの順序を定義する<演算子を示します。
stanumbersN	float4[]		N番目の"スロット"に対する適切な種類の数値統計情報、もしくはスロットの種類に数値が含まれない時はNULLです。
stavaluesN	anyarray		N番目の"スロット"に対する適切な種類の列データの値、もしくはスロットの種類にデータ値が何も保存されていない場合はNULL。それぞれの配列要素の値は実際には特定された列のデータ型、もしくは配列要素の型といったような関連のある型になります。ですからanyarrayとする以外に列型を定義することはできません。

45.43. pg_tablespace

pg_tablespaceカタログは利用できるテーブル空間についての情報を格納します。 テーブルは、ディスクの配置を管理できるようにするために特定のテーブル空間に格納することができます。

システムカタログの大部分とは違って、pg_tablespaceは、すべてのクラスタのデータベース間で共有されます。（データベース毎ではなく）クラスタ毎に、pg_tablespaceのコピーが1つだけ存在します。

表 45-43. pg_tablespaceの列

名前	型	参照先	説明
spcname	name		テーブル空間名
spcowner	oid	pg_authid.oid	テーブル空間の所有者。たいていはテーブル空間を作成したユーザ
spcacl	aclitem[]		アクセス権限。 GRANT と REVOKE を参照してください。
spcoptions	text[]		"keyword=value"文字列のようなテーブル空間レベルのオプション。

45.44. pg_trigger

pg_triggerカタログはテーブルおよびビュー上のトリガを保存します。 [CREATE TRIGGER](#)を参照してください。

表 45-44. pg_triggerの列

名前	型	参照先	説明
tgrelid	oid	pg_class.oid	トリガのかかっているテーブル
tgname	name		トリガ名（同一テーブル内で一意である必要があります）
tgfoid	oid	pg_proc.oid	呼び出される関数
tgtype	int2		トリガ発行条件を指定するビットマスク
tgenabled	char		どの session_replication_role モードでトリガが発行されるかを制御します。 <code>o</code> = "起点"モードと"ローカル"モードでトリガを発行します, <code>D</code> = トリガは無効です, <code>R</code> = "replica"モードでトリガを発行します, <code>A</code> = 常にトリガを発行します。
tgisinternal	bool		トリガが(通常tgconstraintにより識別される制約を強制するために)内部的に生成される場合は真。
tgconstrrelid	oid	pg_class.oid	参照整合性制約で参照されるテーブル
tgconstrindid	oid	pg_class.oid	一意性、プライマリキー、参照整合性制約をサポートするインデックス
tgconstraint	oid	pg_constraint.oid	存在する場合は、トリガに関連するpg_constraintの項目
tgdeferrable	bool		トリガが遅延可能である場合は真
tginitdeferred	bool		トリガの初期状態が遅延可能と宣言されていれば真
tgargs	int2		トリガ関数に渡される引数の数
tgattr	int2vector	pg_attribute.attnum	トリガが列固有であれば列番号。さもなければ空の配列
tgargs	bytea		トリガに渡される引数文字列で、それぞれヌル文字で終結
tgqual	pg_node_tree		トリガのWHEN条件に関する(nodeToString() 表現による)式ツリー。 存在しなければNULL

現在、列固有のトリガ処理はUPDATEイベントのみでサポートされていますので、tgattrはこの種類のイベントにのみ関連します。 tgtypeにはこの他のイベント用のビットが含まれているかもしれませんが、これらはtgattrの値とは関係ないテーブル全体のものであると仮定されます。

注意: tgconstraintがゼロではないとき、tgconstrrelid、tgconstrindid、tgdeferrable、tginitdeferredは参照されるpg_constraint項目と共に冗長となっています。 しかし遅延不可能なトリガを遅延可能な制約に関連付けさせることが可能です。 外部キー制約では一部を遅延可能、一部を遅延不可能なトリガを持つことができます。

注意: pg_class.reltriggersは、リレーションがこのカタログ内にトリガを持っている場合は真とならなければなりません。

45.45. pg_ts_config

pg_ts_configカタログは、テキスト検索の設定を表す項目を含みます。 設定は、特定のテキスト検索パーサと、それぞれのパーサの出力トークン型のために使用される辞書の一覧を指定します。 パーサはpg_ts_config項目内に示されていますが、トークンと辞書の対応付けは、[pg_ts_config_map](#)内の補助項目内に定義されています。

PostgreSQLのテキスト検索機能については[第12章](#)で詳しく説明します。

表 45-45. pg_ts_configの列

名前	型	参照先	説明
cfgname	name		テキスト検索設定の名称
cfgnamespace	oid	pg_namespace.oid	この設定を含む名前空間のOID
cfgowner	oid	pg_authid.oid	この設定の所有者
cfgparser	oid	pg_ts_parser.oid	この設定のためのテキスト検索パーサのOID

45.46. pg_ts_config_map

pg_ts_config_mapカタログは、どのテキスト検索辞書を参照するべきかを示す項目を含みます。 さらに、それぞれのテキスト検索設定のパーサの出力トークンをどの順番で参照すべきかを示す項目を含みます。

PostgreSQLのテキスト検索機能については[第12章](#)で詳しく説明します。

表 45-46. pg_ts_config_mapの列

名前	型	参照先	説明
mapcfg	oid	pg_ts_config.oid	このマップ項目を所有するpg_ts_config項目のOID
maptokentype	integer		設定のパーサにより発行されるトークンの種類
mapseqno	integer		この項目を参照する順番(小さいmapseqnoが先となります)
mapdict	oid	pg_ts_dict.oid	参照するテキスト検索辞書のOID

45.47. pg_ts_dict

pg_ts_dict カタログは、テキスト検索辞書を定義する項目を含みます。辞書は、必要な実装関数すべてを指定するテキスト検索のテンプレートに依存します。辞書自身は、テンプレートによりサポートされている、ユーザが設定可能なパラメータ値を提供します。ここでは、辞書が特権のないユーザにより作成されることを許可します。パラメータは、dictinitoption テキスト文字列で指定されます。その書式と意味はテンプレートにより変化します。

PostgreSQLのテキスト検索機能については[第12章](#)で詳しく説明します。

表 45-47. pg_ts_dict の列

名前	型	参照先	説明
dictname	name		テキスト検索辞書の名称
dictnamespace	oid	pg_namespace.oid	この辞書を含む名前空間のOID
dictowner	oid	pg_authid.oid	辞書の所有者
dicttemplate	oid	pg_ts_template.oid	辞書のためのテキスト検索テンプレートのOID
dictinitoption	text		テンプレートのための初期化オプション文字列

45.48. pg_ts_parser

pg_ts_parserカタログはテキスト検索パーサを定義する項目を含みます。パーサは、入力テキストを字句に分割することとトークン型を字句に割り当てることに責任を持ちます。パーサはC言語レベルの関数で実装されていなくてはならないため、新規のパーサの作成はデータベースのスーパーユーザに制限されています。

PostgreSQLのテキスト検索機能については[第12章](#)で詳しく説明します。

表 45-48. pg_ts_parserの列

名前	型	参照先	説明
prsname	name		テキスト検索パーサの名称
prsnamespace	oid	pg_namespace.oid	このパーサを含む名前空間のOID
prsstart	regproc	pg_proc.oid	パーサ起動関数のOID
prstoken	regproc	pg_proc.oid	パーサの次のトークン関数のOID
prsend	regproc	pg_proc.oid	パーサの終了関数のOID
prshheadline	regproc	pg_proc.oid	パーサの見出し関数のOID
prsllextype	regproc	pg_proc.oid	パーサの字句型関数のOID

45.49. pg_ts_template

pg_ts_templateカタログはテキスト検索テンプレートを定義する項目を含みます。 テンプレートはテキスト検索辞書クラスの骨格を実装したものです。 テンプレートはC言語レベルの関数で実装されなくてはいけないため、新規のテンプレートの作成はデータベースのスーパーユーザに制限されています。

PostgreSQLのテキスト検索機能については[第12章](#)で詳しく説明します。

表 45-49. pg_ts_templateの列

名前	型	参照先	説明
tmplname	name		テキスト検索テンプレートの名称
tmplnamespace	oid	pg_namespace.oid	このテンプレートを含む名前空間のOID
tmplinit	regproc	pg_proc.oid	テンプレートの初期化関数のOID
tmpllexize	regproc	pg_proc.oid	テンプレートの字句関数のOID

45.50. pg_type

pg_type カタログはデータ型の情報を保存します。 スカラ型と列挙型（基本型）は[CREATE TYPE](#)で作成され、ドメインは[CREATE DOMAIN](#)で作成されます。 複合型がテーブルの行構成を表すためデータベースの個々のテーブルに対して自動的に作成されます。 複合型をCREATE TYPE ASで作成することもできます。

表 45-50. pg_typeの列

名前	型	参照先	説明
typname	name		データ型名
typnamespace	oid	pg_namespace.oid	この型を含む名前空間のOID
typowner	oid	pg_authid.oid	型の所有者
typlen	int2		固定長型では、typlenは型の内部表現内でのバイト数です。しかし、可変長型ではtyplenは負となります。-1は"varlena"型（最初の4バイトにデータ長を含むもの）を意味し、-2はヌル終端のC言語の文字列を示します。
typbyval	bool		typbyvalは内部関数がこの型の値を値渡しか、参照渡しかを決定します。typlenが1、2、4バイト長（もしくはDatumが8バイトのマシン上では8バイト長）以外であれば、typbyvalを偽にする必要があります。可変長型は必ず参照渡しになります。typbyvalは長さが値渡し可能でも偽になり得ることに注意してください。
typtype	char		typtypeでは、bは基本型、cは複合型（例えばテーブルの行の型）、dは派生型（ドメインなど）、eは列挙型、pは疑似型、rは範囲型となります。typrelidおよびtypbasetypeも参照してください。
typcategory	char		typcategoryは、パーサがどの暗黙のキャストが"選択"されるべきか決定するのに使用されるデータ型の任意の分類です。 表45-51 を参照してください。
typispreferred	bool		型がtypcategory内で選択されたキャスト対象である場合に真となります。
typisdefined	bool		型が定義されると真、ここが未定義型に対する予備の場所である時は偽。typisdefinedが偽の場合、型名と名前空間とOID以外は信頼すべきではありません。
typdelim	char		配列入力の構文解析をする際にこの型の2つの値を分離する文字。区切り文字は配列データ型ではなく配列要素データ型に関連付けられることに注意してください。
typrelid	oid	pg_class.oid	もしこれが複合型（typtypeを参照）であれば、この列は関連するテーブルを定義するpg_class項目を指します。（独立の複合型の場合、pg_class項目は実際にはテーブルを表しませんが、いずれにしても型のpg_attribute項目をリンクするために必要です）複合型でない場合はゼロです。
typelem	oid	pg_type.oid	typelemがゼロでない場合、これはpg_typeの別の列を特定します。現在の型は、typelem型の値を生成する配列のように、配列要素を持てるようになります。"本当の"の配列型は可変長（typlen = -1）ですが、例えばnameとpointのように、いくつかの固定長（typlen > 0）型は同時に非ゼロのtypelemを持つことができます。もし固定長型がtypelemを持つ場合、その内部表現は他のデータを持たないtypelemデータ型の数個の値でなければなりません。可変長配列型には配列サブルーチンで定義されたヘッダを持ちます。
typarray	oid	pg_type.oid	typarrayがゼロでない場合、typarrayはpg_type内のもうひとつの行を特定します。もうひとつの行は、この型を要素として持っている"本当"の配列型です。
typinput	regproc	pg_proc.oid	入力変換関数（テキスト形式）
typoutput	regproc	pg_proc.oid	出力変換関数（テキスト形式）
typreceive	regproc	pg_proc.oid	入力変換関数（バイナリ形式）、なければゼロ
typsend	regproc	pg_proc.oid	出力変換関数（バイナリ形式）、なければゼロ

typmodin	regproc	pg_proc.oid	型修飾子の入力関数。型が修飾子をサポートしていない場合はゼロ
typmodout	regproc	pg_proc.oid	型修飾子の出力関数。標準書式を使用する場合はゼロ
typanalyze	regproc	pg_proc.oid	独自のANALYZE関数。標準関数を使用する場合はゼロ
typalign	char		typalignはこの型の値を格納する際に必要な整列です。 ディスクに書き込む時やほとんどのPostgreSQL内の値を表現するために使用されます。 ディスクに完全な列を格納する時など、いくつかの値が連続して格納される際、境界を特定するためにこの型のデータの前にパッドが挿入されます。 整列参照はシーケンスの一番最初にあります。 使用可能な値は以下の通りです。 • c = char 整列（すなわち、整列は必要ありません）。 • s = short 整列（多くのマシンでは2バイトになります）。 • i = int 整列（多くのマシンでは4バイトになります）。 • d = double 整列（多くのマシンでは8バイトになりますが、必ずしもすべてがそうであるとは限りません）。 注意: システムテーブルで使用されている型については、pg_typeシステムカタログで定義されている大きさと整列は、コンパイラがテーブルの行を表現する構造体に対して位置決めを行うものと厳密に一致することを求められています。
typstorage	char		typstorageは、varlena型 (typplen = -1のもの) に対して型がTOASTに対応しているか、また、この型の属性のデフォルト戦略が何であるべきかを伝えます。 使用可能な値は以下の通りです。 • p: 値は必ずそのまま格納されなければいけません。 • e: 値は"従属的"リレーションに格納できます（リレーションに関してはpg_class.reltoastrelidを参照してください）。 • m: 値は圧縮されたインラインに格納することができます。 • x: 値は圧縮されたインラインもしくは"従属"格納領域に格納することができます。 m列も従属格納領域に移すことができますが、最後の切札としてでなければいけません（eとx列が先に移動されます）。
typnotnull	bool		typnotnullは型に対し非NULL制約を表します。 ドメインでのみ使用されます。
typbasetype	oid	pg_type.oid	もしこれがドメイン (typtypeを参照) であれば、typbasetypeはこれが基づいている型を指定します。 ドメインでない場合はゼロです。
typtypmod	int4		ドメインはtyptypmodを使用して、基本型に適用されるtypmodを記録します（基本型がtypmodを使用しない場合は-1）。 この型がドメインでない場合は-1です。
typndims	int4		typndimsは配列であるドメインの配列の次元数です（つまり、typbasetypeは配列型です。）。 配列型のドメインでない場合はゼロです。
			typcollationは型の照合順序を指定します。 型が照合順序をサポートしない場合、ゼロになります。 照合順序をサポートする基本型はここ

typcollation	oid	pg_collation.oid	でDEFAULT_COLLATION_OIDを持ちます。照合順序の設定可能な型全体のドメインは、そのドメインで照合順序が指定されていれば、他の照合順序OIDを持つことができます。
typdefaultbin	pg_node_tree		typdefaultbinがNULLでない場合、これは型のデフォルト式のnodeToString() 表現です。ドメインでのみ使用されます。
typdefault	text		関連するデフォルト値を持たない型であればtypdefaultはNULLです。typdefaultbinがNULLでない場合、typdefaultは、typdefaultbinによって表される人間が見てわかる形式のデフォルト式を含む必要があります。typdefaultbinがNULLでtypdefaultがNULLでない場合、typdefaultは型のデフォルト値の外部表現です。これは、定数を生成するために型の入力変換処理に渡されることがあります。
typacl	aclitem[]		アクセス特権。詳細は GRANT と REVOKE を参照してください。

[表45-51](#)はシステムで定義されたtypcategoryの値の一覧です。今後この一覧に追加されるものは同様に大文字のASCII文字になります。他のすべてのASCII文字はユーザ定義のカテゴリのために予約されています。

表 45-51. typcategoryのコード

コード	カテゴリ
A	配列型
B	論理値型
C	複合型
D	日付時刻型
E	列挙型
G	幾何学型
I	ネットワークアドレス型
N	数値型
P	仮想型
R	範囲型
S	文字列型
T	時間間隔型
U	ユーザ定義型
V	ビット列型
X	unknown型

45.51. pg_user_mapping

pg_user_mapping カタログはローカルのユーザから遠隔のユーザへのマッピングを保持します。 一般ユーザからのこのカタログへのアクセスは制限されています。 代わりに[pg_user_mappings](#)を使用してください。

表 45-52. pg_user_mapping の列

名前	型	参照先	説明
umuser	oid	pg_authid .oid	マッピングされているローカルのロールのOID。 ユーザマッピングが公開されている場合は0になります。
umserver	oid	pg_foreign_server .oid	マッピングを保持する外部サーバのOID
umoptions	text[]		"keyword=value"文字列のようなユーザマッピングの特定のオプション

45.52. システムビュー

システムカタログに加えPostgreSQLは数多くの組み込みビューを提供しています。 システムビューはいくつかの一般的に使用されるシステムカタログに対する問い合わせに手近にアクセスできるようにします。 他のビューはサーバ状態内部へのアクセスを提供します。

情報スキーマ ([第34章](#)) はシステムビューと重複する、もう一方のビューの集合を提供しています。 ここで説明しているビューはPostgreSQL特有のものであるのに対し、情報スキーマはSQL標準であることから、もし情報スキーマが必要とする情報をすべて提供してくれるのであれば情報スキーマを使用の方が良いでしょう。

[表45-53](#)は、ここで説明しているシステムビューの一覧です。 それぞれのビューのさらに詳細な説明は、これより後に述べられています。 統計情報の結果にアクセスするためのいくつかの追加のビューがあります。 それらは[表27-1](#)で説明されています。

注意書きがない限り、ここでのすべてのビューは読み取り専用です。

表 45-53. システムビュー

ビュー名	目的
pg_available_extensions	利用可能な拡張
pg_available_extension_versions	利用可能な拡張のバージョン
pg_cursors	開いているカーソル
pg_group	データベースのユーザのグループ
pg_indexes	インデックス
pg_locks	現在獲得されているロック
pg_prepared_statements	準備済みの文
pg_prepared_xacts	準備済みのトランザクション
pg_roles	データベースロール
pg_rules	ルール
pg_seclabels	セキュリティラベル
pg_settings	パラメータ設定
pg_shadow	データベースのユーザ
pg_stats	プランナの統計
pg_tables	テーブル
pg_timezone_abbrevs	時間帯省略形
pg_timezone_names	時間帯名
pg_user	データベースのユーザ
pg_user_mappings	ユーザマッピング
pg_views	ビュー

45.53. pg_available_extensions

pg_available_extensionsビューはインストレーションで利用可能な拡張を列挙します。 現在インストールされている拡張を表すpg_extensionカタログも参照してください。

表 45-54. pg_available_extensionsの列

名前	型	説明
name	name	拡張名
default_version	text	デフォルトのバージョン名称。何も指定がなければNULL
installed_version	text	現在インストールされている拡張のバージョン。インストールされていない場合はNULL
comment	text	拡張の制御ファイルからのコメント文字列

pg_available_extensionsビューは読み取り専用です。

45.54. pg_available_extension_versions

pg_available_extension_versionsビューはインストレーションで利用可能な特定の拡張のバージョンを列挙します。 現在インストールされている拡張を表す [pg_extension](#) カタログも参照してください。

表 45-55. pg_available_extension_versionsの列

名前	型	説明
name	name	拡張の名前
version	text	バージョン名称
installed	bool	現在このバージョンの拡張がインストールされている場合に真
superuser	bool	スーパーユーザのみがこの拡張をインストールできる場合に真
relocatable	bool	拡張が他のスキーマに再配置可能である場合に真
schema	name	拡張がインストールされなければならないスキーマの名前。 一部の再配置またはすべての再配置を行うことができる場合はNULL
requires	name[]	前もって必要な拡張の名前。 なければNULL
comment	text	拡張の制御ファイルからのコメント文字列

pg_available_extension_versionsビューは読み取り専用です。

45.55. pg_cursors

pg_cursorsビューは現在利用可能なカーソルを列挙します。 以下のようにカーソルは複数の方法で定義可能です。

- SQLから[DECLARE](#)文経由。
- [項46.2.3](#)で説明する、フロントエンド/バックエンドプロトコルからBindメッセージ経由。
- [項43.1](#)で説明する、サーバプログラミングインタフェース（SPI） 経由。

pg_cursorsビューは、上のいずれかの方法で作成されたカーソルを表示します。 カーソルは、WITH HOLDと宣言されていない限り、それを定義したトランザクション期間しか存在しません。 したがって、保持不可能なカーソルは、作成元トランザクションが終わるまでの間のみ、このビューに現れます。

注意: 手続き言語など、一部のPostgreSQLの要素を実装するために内部的にカーソルが使用されています。 したがって、pg_cursorsにはユーザが明示的に作成していないカーソルも含まれる可能性があります。

表 45-56. pg_cursorsの列

名前	型	説明
name	text	カーソルの名前
statement	text	カーソル宣言の際に投稿された逐語的問い合わせ文字列
is_holdable	boolean	保持可能カーソル（つまりカーソルを宣言したトランザクションがコミットされた後でもアクセス可能なカーソル）であればtrueです。 さもなくばfalseです。
is_binary	boolean	カーソルがBINARYで宣言されていたらtrue、さもなくばfalse。
is_scrollable	boolean	カーソルがスクロール可能（順序通り以外の方法に行を取り出すことが可能）であればtrue、さもなくばfalse。
creation_time	timestamptz	カーソルが宣言された時間。

pg_cursorsビューは読み取り専用です。

45.56. pg_group

pg_groupビューは下位互換のために存在しています。 バージョン8.1以前のPostgreSQLのカタログを模擬しています。 このビューは、rolcanloginとしてマークされていない、すべてのロールの名前とメンバを保持しています。 これはグループとして使用されているロールの集合と似ています。

表 45-57. pg_groupの列

名前	型	参照先	説明
groname	name	pg_authid.rolname	グループの名前
grosysid	oid	pg_authid.oid	グループのID
grolist	oid[]	pg_authid.oid	このグループのロールIDを含む配列

45.57. pg_indexes

pg_indexes ビューはデータベース内のそれぞれのインデックスについて有用な情報を提供します。

表 45-58. pg_indexesの列

名前	型	参照先	説明
schemaname	name	pg_namespace .nspname	テーブルとインデックスを含むスキーマの名前
tablename	name	pg_class .relname	インデックスのついているテーブルの名前
indexname	name	pg_class .relname	インデックスの名前
tablespace	name	pg_tablespace .spcname	インデックスを含むテーブル空間の名前（データベースのデフォルトはNULL）
indexdef	text		インデックス定義（再作成用CREATE INDEXコマンド）

45.58. pg_locks

pg_locksビューはデータベースサーバ内で開いているトランザクションにより獲得されたロックに関する情報へのアクセスを提供します。 ロックに関するより詳細な説明は[第13章](#)を参照してください。

pg_locksにはロック対象となる進行中のオブジェクト、要求されたロックモード、および関連するトランザクション毎に1つの行を持ちます。 ですから、もし複数のトランザクションがそのトランザクション上でロックを保持していたりロックを待機している場合には、同じロック対象オブジェクトが数多く出現することがあります。 しかし現在ロックされていないオブジェクトはまったく現れません。

ロック対象オブジェクトには異なる型がいくつか存在します。 リレーション全体（例：テーブル）、リレーションの個別のページ、リレーションの個別のタプル、トランザクションID（仮想と永続の両方のID）、一般的なデータベースオブジェクト（これはpg_descriptionやpg_dependと同様にクラスOIDとオブジェクトOIDで識別されます）。 さらに、リレーションを拡張する権利は、別のロック対象オブジェクトとして表現されます。 また"勧告的"ロックはユーザ定義の意味を持つ複数から形成されるかもしれません。

表 45-59. pg_locksの列

名前	型	参照先	説明
locktype	text		ロック対象オブジェクトの種類。 relation、 extend、 page、 tuple、 transactionid、 virtualxid、 object、 userlock、 advisory
database	oid	pg_database.oid	ロック対象が存在しているデータベースのOID。 対象が共有オブジェクトの場合はゼロ。 対象がトランザクションIDである場合はNULL。
relation	oid	pg_class.oid	ロックの対象となるリレーションのOID。 対象がリレーションではない場合かリレーションの一部である場合はNULL。
page	integer		ロックの対象となるリレーション内のページ番号。 対象がタプルもしくはリレーションページではない場合はNULL。
tuple	smallint		ページ内のロックの対象となっているタプル番号。 対象がタプルではない場合はNULL。
virtualxid	text		ロックの対象となるトランザクションの仮想ID。 対象が仮想トランザクションIDではない場合はNULL。
transactionid	xid		ロックの対象となるトランザクションのID。 対象がトランザクションIDではない場合はNULL。
classid	oid	pg_class.oid	ロックの対象を含むシステムカタログのOID。 対象が一般的なデータベースオブジェクトではない場合はNULL。
objid	oid	any OID column	システムカタログ内のロックの対象のOID。 対象が一般的なデータベースオブジェクトでない場合はNULL。
objsubid	smallint		ロック対象の列番号（classidとobjidはテーブル自身を参照します）、その他の一般的なデータベースオブジェクトではゼロ、一般的ではないデータベースオブジェクトではNULLです。
virtualtransaction	text		ロックを保持、もしくは待っている仮想トランザクションID。
pid	integer		ロックを保持、もしくは待っているサーバプロセスのプロセスID。 ただし準備されたトランザクションによりロックが保持されている場合はNULL。
mode	text		このプロセスで保持または要求するロックモードの名称。 (項13.3.1 and 項13.2.3 参照)
granted	boolean		ロックが保持されている場合は真、ロックが待ち状態の場合は偽
fastpath	boolean		ファストパス経由でロックが獲得されている場合は真、メインロックテーブル経由で獲得されている場合は偽。

指定されたトランザクションにより所有されているロックを表す行内ではgrantedは真です。 偽の場合はこのロックを獲得するため現在トランザクションが待機中であることを示しています。 つまり、同じロック対象のオブジェクトに対して何らかの他のトランザクションが競合するロックを獲得していることを意味します。 待機中のトランザクションはその別のトランザクションがロックを解放するまで活動を控えます（もしくはデッドロック状態が検出されることになります）。 単一トランザクションでは一度に多くても1つのロックを獲得するために待機します。

すべてのトランザクションはそのすべての過程完了までその仮想トランザクションID上に排他的ロックをかけます。もしある永続IDがトランザクションに割り当てられる（普通はトランザクションがデータベースの状態を変化させるときのみ）に発生しますと、トランザクションは終了するまで永続トランザクションIDに対して排他ロックを保持します。あるトランザクションが他のトランザクションを特定して待機しなければならないと判断した場合、他とみなしたトランザクションのIDに対し共有ロックを獲得するように試み、目的を達します（仮想IDであるか永続IDであるかは、その状況によります）。これは、他とみなしたトランザクションが完了し、そしてロックを解放した場合のみ成功します。

ダブルはロック対象のオブジェクト種類ですが、行レベルロックについての情報はメモリではなく、ディスクに保存されます。よって行レベルロックは通常、このビューには現れません。もしトランザクションが行レベルロックの待ち状態である場合は、その行ロックを保持している永続トランザクションIDを待つ状態で、そのトランザクションはビューに現れます。

勧告的ロックは、単一のbigint値、または、2つの整数値をキーとして獲得することができます。bigintの場合は、その上位半分がclassid列内に表示され、残りの下位半分はobjid列内に表示されます。また、objsubidは1となります。整数値キーでは、最初のキーがclassid列に、2番目のキーがobjid列に表示され、objsubidは2となります。キーの実際の意味はユーザに任されています。勧告的ロックはデータベースに対して局所的ですので、勧告的ロックではdatabase列が意味を持ちます。

pg_locksは現行のデータベースに関連するロックのみならず、データベースクラスタ内のすべてのロックに関する全体的なビューを提供します。relation列はロックされたリレーションを識別するためにpg_class.oidと結合することができますが、これは現行のデータベース内のリレーション（database列が現行のデータベースのOIDまたはゼロとなっているもの）に対してのみ正常に動作します。

ロック保持もしくは保持を待機しているセッションのさらなる情報を入手するためpg_stat_activityビューのpid列とpid列を結合することができます。また、準備されたトランザクションを使用している場合には、transactionはpg_prepared_xactsビューのtransaction列と結合でき、ロックを保持している準備されたトランザクションに関して、より多くの情報を取得できます。（準備されたトランザクションはロックを待つことはありませんが、実行時に獲得したロックを保持し続けます。）

pg_locksビューは、異なるシステムにおける、通常のロックマネージャと述部ロックマネージャの両方からのデータを表示します。さらに通常のロックマネージャではロックを通常ロックと近道ロックに細分化します。このデータが完全に一貫性があることは保証されません。ビューが問い合わせされると、近道ロック（fastpath = trueが真）は、ロックマネージャ全体の状態を凍結することなく、各バックエンドからひとつひとつ収集されます。このため情報収集期間中にロックが獲得されたり解放されたりされる可能性があります。しかし、これらのロックはその時点で存在する他のロックと競合することがないことが分かっていることに注意してください。近道ロックについてすべてのバックエンドを問い合わせた後、通常のロックマネージャの残りは1つの単位としてロックされ、残りすべてのロックの一貫性があるスナップショットを原子的な処理で収集します。ロックマネージャのロックを解除した後、述部ロックマネージャは同様にロックされ、すべての述部ロックを原子的な処理で収集します。このように、近道ロックという例外がありますが、各ロックマネージャは一貫性をもった結果セットを生成します。しかし、両方のロックマネージャを同時にロックしませんので、通常のロックマネージャを問い合わせた後と述部ロックマネージャを問い合わせる前の間にロックが獲得されたり解放されたりされる可能性があります。

このビューが頻繁にアクセスされている場合は、通常もしくは述部ロックマネージャをロックするとデータベースのパフォーマンスに影響があります。ロックマネージャからデータを取得するために、ロックは必要最低限の時間だけ保持されますが、パフォーマンスに影響がある可能性を全くないわけではありません。

[前のページ](#)
pg_indexes

[ホーム](#)
[上に戻る](#)

[次のページ](#)
pg_prepared_statements

45.59. pg_prepared_statements

pg_prepared_statementsビューは現在のセッションで利用可能な準備済み文をすべて表示します。 準備済み文についての詳細は[PREPARE](#)を参照してください。

pg_prepared_statementsには、1つの準備済み文に対して一行が存在します。 新しい準備済み文が作成されると行が追加され、準備済み文が解放される（例えば[DEALLOCATE](#)を使用）と行が削除されます。

表 45-60. pg_prepared_statementsの列

名前	型	説明
name	text	準備済み文の識別子
statement	text	この準備済み文を作成するためにクライアントが送付した問い合わせ文字列。 SQL経由で作成された準備済み文では、これはクライアントが送信したPREPARE文です。 フロントエンド/バックエンドプロトコル経由で作成された準備済み文では、これは準備済み文自身のテキストです。
prepare_time	timestampz	準備済み文が作成された時間
parameter_types	regtype[]	regtype配列形式の準備済み文が想定しているパラメータ型。 配列要素に対応するOIDは、regtypeからoidへのキャストを行うことで取り出すことができます。
from_sql	boolean	準備済み文がPREPARE SQL文経由で作成された場合はtrue、フロントエンド/バックエンドプロトコル経由で文が準備された場合はfalse

pg_prepared_statementsビューは読み取り専用です。

45.60. pg_prepared_xacts

pg_prepared_xactsビューは、現状で2相コミットのために準備されたトランザクションについての情報を表示します（詳細は[PREPARE TRANSACTION](#)を参照してください）。

pg_prepared_xactsは、準備されたトランザクション毎に1つの行を含みます。この項目はトランザクションがコミットもしくはロールバックされたときに削除されます。

表 45-61. pg_prepared_xactsの列

名前	型	参照先	説明
transaction	xid		準備されたトランザクションに対する数値のトランザクション識別子
gid	text		トランザクションに割り当てられたグローバルのトランザクション識別子
prepared	timestamp with time zone		トランザクションがコミットのために準備された時間
owner	name	pg_authid .rolname	トランザクションを実行したユーザ名
database	name	pg_database .datname	トランザクションを実行したデータベース名

pg_prepared_xactsビューにアクセスすると、内部のトランザクション管理データ構造が一時的にロックされます。そして表示用にコピーが作成されます。これは、必要以上に長く通常の操作をブロックさせずに、ビューが一貫性のある結果を生成することを保証します。このビューが頻繁にアクセスされると、データベースの性能になんらかの影響を及ぼします。

45.61. pg_roles

pg_rolesビューはデータベースのロールに関する情報を提供します。これは単に一般に公開されているpg_authidのビューですが、パスワード列が空白になっています。

このビューは、他のカタログに対して結合を行う必要があるため、明示的に基礎となっているテーブルのOID列を表示します。

表 45-62. pg_rolesの列

名前	型	参照先	説明
rolname	name		ロール名
rolsuper	bool		ロールはスーパーユーザの権限を持っています。
rolinherit	bool		ロールは自動的にメンバであるロールの権限を継承します。
rolcreatorole	bool		ロールはさらにロールを作成することができます。
rolcreatedb	bool		ロールはデータベースを作成することができます。
rolcatupdate	bool		ロールは直接システムカタログを更新することができます。（この列が真でない限り、スーパーユーザでも更新できません。）
rolcanlogin	bool		ロールはログイン可能です。このロールは初期セッションの識別子として与えることができます。
rolreplication	bool		ロールはレプリケーション用のロールです。つまり、このロールはストリーミングレプリケーションを初期化し、pg_start_backupとpg_stop_backupを用いてシステムバックアップモードを設定、解除することができます。
rolconndef	int4		ログイン可能なロールでは、このロールが確立可能な同時接続の最大数を設定します。-1は無制限を意味します。
rolpassword	text		パスワードではありません（常に*****のように読めます）
rolvaliduntil	timestampz		パスワード有効期限（パスワード認証でのみ使用）。有効期限がない場合はNULLとなります。
rolconfig	text[]		実行時設定変数に関するロール固有のデフォルト
oid	oid	pg_authid.oid	ロールのID

45.62. pg_rules

pg_rulesビューは問い合わせ書き換えルールについての有用な情報へのアクセスを提供します。

表 45-63. pg_rulesの列

名前	型	参照先	説明
schemaname	name	pg_namespace .nspname	テーブルを含むスキーマ名
tablename	name	pg_class .relname	ルールの対象のテーブル名
rulename	name	pg_rewrite .rulename	ルール名
definition	text		ルール定義（再構築された生成コマンド）

pg_rulesビューは、ビューに対するON SELECTルールを除外します。 これらはpg_viewsにあります。

45.63. pg_seclabels

pg_seclabelsビューはセキュリティラベルに関する情報を提供します。これは[pg_seclabel](#)カタログをより問い合わせし易くしたものです。

表 45-64. pg_seclabelsの列

名前	型	参照先	説明
objoid	oid	任意のOID列	このセキュリティラベルが付属するオブジェクトのOID
classoid	oid	pg_class .oid	このオブジェクトが存在するシステムカタログのOID
objsubid	int4		テーブル列上のセキュリティラベルでは、これは列番号です（objoidおよびclassoidはテーブル自身を参照します）。他のすべての種類のオブジェクトでは、この列はゼロです。
objtype	text		このラベルが適用されるオブジェクトの種類のテキスト表現
objnamespace	oid	pg_namespace .oid	もし適用可能であればこのオブジェクト用の名前空間のOID。さもなければNULL
objname	text		このラベルが適用されるオブジェクト名称のテキスト表現
provider	text	pg_seclabel .provider	このレベルに関連付いたラベルプロバイダ
label	text	pg_seclabel .label	このオブジェクトに適用されるセキュリティラベル

45.64. pg_settings

pg_settingsビューはサーバの実行時パラメータへのアクセスを提供します。 基本的にSHOWとSETコマンドの代わりとなるインタフェースです。 同時に最大・最小値などのようにSHOWコマンドでは直接入手できないそれぞれのパラメータのいくつかの実状にアクセスする機能を提供します。

表 45-65. pg_settingsの列

名前	型	説明
name	text	実行時設定パラメータ名
setting	text	パラメータの現在値
unit	text	暗黙的なパラメータの単位
category	text	パラメータの論理グループ
short_desc	text	パラメータの簡潔な説明
extra_desc	text	追加で、より詳細なパラメータについての説明
context	text	パラメータ値を設定するために必要な文脈（後述）
vartype	text	パラメータの型（bool、enum、integer、realもしくはstring）
source	text	現在のパラメータ値のソース
min_val	text	容認されている最小のパラメータ値（数値でない場合はNULL）
max_val	text	容認されている最大のパラメータ値（数値でない場合はNULL）
enumvals	text[]	許可された列挙パラメータの値(列挙型ではない場合はNULL)
boot_val	text	パラメータが設定されていなかったとした場合に仮定されるサーバ起動時のパラメータ値
reset_val	text	現状のセッションにおいてRESETによって戻されるパラメータの値
sourcefile	text	現状の値が設定されている設定ファイル(設定ファイル以外のソースから設定された値の場合とスーパーユーザ以外から検査された時はNULLとなります)。 設定ファイル内でinclude指示子を使用する時に役に立ちます。
sourceline	integer	現状の値が設定されている設定ファイル内の行番号(設定ファイル以外のソースから設定された値の場合とスーパーユーザ以外から検査された時はNULLとなります。)

contextが取り得る値は複数あります。 この設定の変更の困難さを軽くするために、以下に示します。

internal
これらの設定は直接変更できません。 これらは内部で決定された値を反映するものです。 一部は異なる設定オプションでサーバを細構築する、または、initdbに与えるオプションを変更することで調整することができます。

postmaster
これらの設定はサーバ起動時にのみ適用することができます。 このため何かを変更するためにはサーバを再起動しなければなりません。 これらの設定用の値は通常postgresql.confファイル内に格納されている、あるいは、サーバを起動する際のコマンドラインから渡されます。 当然ながら、より低い種類のcontextを持つ設定もサーバ起動時に設定することができます。

sighup
これらの設定は、サーバを再起動することなくpostgresql.conf内を変更することで行うことができます。
postgresql.confを再度読み込み、変更を適用させるためには、postmasterにSIGHUPシグナルを送信してください。
すべての子プロセスが新しい値を選択するように、postmasterは同時に子プロセスにSIGHUPシグナルを転送します。

backend
これらの設定は、サーバを再起動することなくpostgresql.conf内を変更することで行うことができます。 また、接続要求パケットの中で特定のセッション向けに設定することもできます（例えばlibpqのPGOPTIONS環境変数）。 しかし、これらの設定はセッションが開始してから、そのセッションの中で変更することはできません。 postgresql.conf内でそれらを変更した場合は、postgresql.confを再度読み込ませるために、postmasterにSIGHUPシグナルを送信してください。
新しい値はその後で始まったセッションにのみ影響を与えます。

superuser

これらの設定は`postgresql.conf`、または、セッションの中で`SET`コマンドを使用することで設定することができます。しかし`SET`経由で変更できるのはスーパーユーザのみです。`postgresql.conf`内の変更は、セッション独自の値が`SET`で設定されていない場合にのみ、既存のセッションに影響を与えます。

user

これらの設定は`postgresql.conf`、または、セッションの中で`SET`コマンドを使用することで設定することができます。任意のユーザが自身のセッション独自の値を変更することが許されています。`postgresql.conf`内の変更は、セッション独自の値が`SET`で設定されていない場合にのみ、既存のセッションに影響を与えます。

これらのパラメータを変更する各種方法に関する情報については[項18.1](#)を参照してください。

`pg_settings`ビューには挿入も削除もできませんが、更新することは可能です。`pg_settings`行に適用される`UPDATE`は名前付きのパラメータに対して`SET`コマンドを実行することと同値です。変更は現在のセッションで使用されている値にのみ有効です。もしも後に中止されるトランザクション内で`UPDATE`が発行されると、トランザクションがロールバックされた時点で`UPDATE`コマンドは効力を失います。排他制御中のトランザクションがひとたびコミットされると、その効果は他の`UPDATE`もしくは`SET`コマンドで上書きされない限りセッションの完了まで保たれます。

[前のページ](#)
pg_seclabels

[ホーム](#)
[上に戻る](#)

[次のページ](#)
pg_shadow

45.65. pg_shadow

pg_shadowビューは下位互換のために存在しています。 バージョン8.1以前のPostgreSQLに存在していたカタログを模擬します。 [pg_authid](#)内でrolcanloginのマークがついた全てのロールの属性を保持します。

名前の由来は、このテーブルがパスワードを含むため、一般的には読めないことから来ています。 [pg_user](#)は、pg_shadowのビューですが、パスワードの列が空白となっているため一般に読むことが可能です。

表 45-66. pg_shadowの列

名前	型	参照先	説明
username	name	pg_authid.rolname	ユーザ名
usesysid	oid	pg_authid.oid	ユーザID
usecreatedb	bool		ユーザはデータベースを作成可能です。
usesuper	bool		ユーザはスーパーユーザです。
usecatupd	bool		ユーザはシステムカタログを更新可能です。（スーパーユーザであってもここが真でなければ更新できません。）
userepl	bool		ユーザはストリーミングレプリケーションを開始することができ、システムをバックアップモードにしたり、戻したりできます。
passwd	text		（おそらく暗号化された）パスワード。 存在しない場合はNULLです。 暗号化されたパスワードの格納方法については pg_authid を参照してください。
valuntil	abstime		パスワード有効期限（パスワード認証でのみ使用）
useconfig	text[]		実行時設定変数のセッションデフォルト

45.66. pg_stats

pg_statsビューはpg_statisticカタログの情報にアクセスするためのビューです。このビューは、ユーザが読み込み権限を持つテーブルに一致するpg_statisticの行に対してのみアクセスを許可しています。よって、このビューに対して一般に読み込みを許可しても安全です。

pg_statsも、その基礎となっているカタログよりも、より読みやすい書式で情報を提供するように設計されています。しかし、これは、もしpg_statisticに対して新しいスロット型が定義されるたびに、スキーマが拡張されなくてはならない、という犠牲を払っています。

表 45-67. pg_statsの列

名前	型	参照先	説明
schemaname	name	pg_namespace .nspname	テーブルを持つスキーマ名
tablename	name	pg_class .relname	テーブル名
attname	name	pg_attribute .attname	この行が記述する列名
inherited	bool		真の場合、この行には指定されたテーブルの値だけではなく、継承関係の子の列が含まれます。
null_frac	real		NULLとなっている列項目の割合
avg_width	integer		列項目のバイト単位による平均幅
n_distinct	real		ゼロより大きい値は列内の個別値の推定数です。ゼロより小さければ行数で個別値を割算した数字の負数です。（テーブルが肥大するにつれ固有値の増大があり得るとANALYZEが判断した場合に負変換形式が使われます。正変換形式は列の取り得る値が固定数を持つと思われる場合に使用されます）。例えば-1は個別値の数が行数と等しいような、一意な列を表します。
most_common_vals	anyarray		列の中の最も共通した値のリストです（他の値よりもより共通している値がない場合はNULLとなります）。
most_common_freqs	real[]		最も一般的な値の出現頻度のリストで、つまり行の総数で出現数を割算した数字です（most_common_valsがNULLの時はNULLとなります）。
histogram_bounds	anyarray		列の値を満遍なく似たような数でグループに分配した値のリストです。most_common_valsの値がもし存在すればこの度数分布計算は行われません（列データ型が<演算子を所有しない場合、もしくはmost_common_valsが全体の構成要素アカウントをリストしている場合、この列はNULLとなります）。
correlation	real		物理的な[訳注：ディスク上の]行の並び順と論理的な列の値の並び順に関する統計的相関です。この値は-1から+1の範囲です。値が-1もしくは+1の近辺にある時、ディスクにランダムアクセスする必要が少なくなるためこの列に対してのインデックススキャンは0近辺にある場合に比較して安価であると推定されます（列データ型に<演算子がない場合、この列はNULLとなります）。
most_common_elems	anyarray		列の値の中で最もよく出現する非NULLの要素値のリストです。（スカラ型の場合はNULLです。）
most_common_elem_freqs	real[]		最も一般的な要素値の出現頻度のリストで、与えられた値の少なくとも1つのインスタンスを含む行の断片です。2つもしくは3つの追加の値が1つの要素ごとの出現頻度に続きます。最小で最大の要素ごとの出現頻度があります。さらにオプションとしてNULL要素の出現頻度もあります。（most_common_elemsの場合はNULLになります。）
elem_count_histogram	real[]		列の値でNULLではない要素値の個別数のヒストグラム。これは個別のNULLではない平均値が後に続きます。（スカラ型の場合はNULLです。）

配列の最大項目数はALTER TABLE SET STATISTICS コマンドで列ごとに設定されるか、もしくは[default_statistics_target](#)実行

時パラメータで包括的に設定されるかのいずれかです。

45.67. pg_tables

pg_tablesビューはデータベース内のそれぞれのテーブルに関する有用な情報へのアクセスを提供します。

表 45-68. pg_tablesの列

名前	型	参照先	説明
schemaname	name	pg_namespace .nspname	テーブルを持つスキーマ名
tablename	name	pg_class .relname	テーブル名
tableowner	name	pg_authid .rolname	テーブルの所有者
tablespace	name	pg_tablespace .spcname	テーブルを含むテーブル空間の名前（データベースのデフォルトの場合はNULL）
hasindexes	boolean	pg_class .relhasindex	テーブルがインデックスを持っている（もしくは最近まで持っていた）なら真
hasrules	boolean	pg_class .relhasrules	テーブルにルールがある（もしくは以前あった）時は真
hastriggers	boolean	pg_class .relhastriggers	テーブルにトリガがある（もしくは以前あった）時は真

45.68. pg_timezone_abbrevs

pg_timezone_abbrevsビューは、現在日付時間の入力処理で認識されている、時間帯省略形のリストを提供します。 このビューの内容は、[timezone_abbreviations](#)実行時パラメータが変更された時に変わります。

表 45-69. pg_timezone_abbrevsの列

名前	型	説明
abbrev	text	時間帯省略形
utc_offset	interval	UTCからのオフセット(正はグリニッジより西側を意味する)
is_dst	boolean	夏時間省略形の場合は真

45.69. pg_timezone_names

pg_timezone_namesは、SET TIMEZONEで認識される時間帯名称の一覧を提供します。 ここには、その関連付けされた省略形、UTCオフセット、夏時間状況などが含まれます。（PostgreSQLは技術的には、うるう秒を扱いませんので、UTCではなくUT1を使用します。） [pg_timezone_abbrevs](#)で示した省略形とは異なり、名前の多くが夏時間変換規則を意味しています。したがって、関連する情報はローカルなDST境界によって異なります。 表示される情報は、現在のCURRENT_TIMESTAMPに基づいて計算されたものです。

表 45-70. pg_timezone_namesの列

名前	型	説明
name	text	時間帯名
abbrev	text	時間帯省略形
utc_offset	interval	UTCからのオフセット(正はグリニッジより西側を意味する)
is_dst	boolean	現在夏時間である場合に真

45.70. pg_user

pg_userビューはデータベースユーザに関する情報へのアクセスを提供します。これはパスワードフィールドを隠蔽したpg_shadowを公に読めるようにしたビューです。

表 45-71. pg_userの列

名前	型	説明
username	name	ユーザ名
usesysid	oid	このユーザのID
usecreatedb	bool	ユーザはデータベースを作成可能です。
usesuper	bool	ユーザはスーパーユーザです。
usecatupd	bool	ユーザはシステムカタログを更新可能です。（スーパーユーザであってもこの列が真でなければ更新はできません。）
userepl	bool	ユーザはストリーミングレプリケーションを開始することができ、システムをバックアップモードにしたり、戻したりできます。
passwd	text	パスワードではありません（常に*****のように読めます）
valuntil	abstime	パスワードの有効期限（パスワード認証時のみ使用）
useconfig	text[]	実行時設定変数に対するセッションデフォルト

45.71. pg_user_mappings

pg_user_mappingsビューはユーザマッピングについての情報へのアクセスを提供します。これはユーザが使用する権利を持っていないオプションフィールドを取り除いた、基本的には公開されていて読み取り可能なpg_user_mappingのビューです。

表 45-72. pg_user_mappings Columns

名前	型	参照先	説明
umid	oid	pg_user_mapping .oid	ユーザマッピングのOID
srvid	oid	pg_foreign_server .oid	マッピングを含んでいる外部サーバのOID
srvname	name	pg_foreign_server .srvname	外部サーバの名称
umuser	oid	pg_authid .oid	ローカルのマッピングされているロールのOID。ユーザマッピングが公開されている場合は0となります。
username	name		マッピングされているローカルユーザの名前
umoptions	text[]		もし現状のユーザが外部サーバの所有者であれば"keyword=value"文字列のようなユーザマッピングの特定オプション。他の場合はNULLとなります。

45.72. pg_views

pg_viewsビューはデータベース内のそれぞれのビューに関する有用な情報へのアクセスを提供します。

表 45-73. pg_viewsの列

名前	型	参照先	説明
schemaname	name	pg_namespace .nspname	ビューを持つスキーマ名
viewname	name	pg_class .relname	ビュー名
viewowner	name	pg_authid .rolname	ビューの所有者
definition	text		ビュー定義（再構築されたSELECT問い合わせ）

第 46章 フロントエンド/バックエンドプロトコル

目次

46.1. [概要](#)

- 46.1.1. [メッセージ処理の概要](#)
- 46.1.2. [拡張問い合わせの概要](#)
- 46.1.3. [書式と書式コード](#)

46.2. [メッセージの流れ](#)

- 46.2.1. [開始](#)
- 46.2.2. [簡易問い合わせ](#)
- 46.2.3. [拡張問い合わせ](#)
- 46.2.4. [関数呼び出し](#)
- 46.2.5. [COPY操作](#)
- 46.2.6. [非同期操作](#)
- 46.2.7. [処理中のリクエストの取り消し](#)
- 46.2.8. [終了](#)
- 46.2.9. [SSLセッション暗号化](#)

46.3. [ストリーミングレプリケーションプロトコル](#)

46.4. [メッセージのデータ型](#)

46.5. [メッセージの書式](#)

46.6. [エラーおよび警報メッセージフィールド](#)

46.7. [プロトコル2.0からの変更点の要約](#)

PostgreSQLはフロントエンドとバックエンド（クライアントとサーバ）の通信にメッセージベースのプロトコルを使用します。このプロトコルはTCP/IPに加え、Unixドメインソケットをサポートします。ポート番号5432は、このプロトコルをサポートするサーバ用のTCPポートとしてIANAに登録されています。しかし、実際には任意の非特権ポート番号を使用することができます。

この文書はPostgreSQL 7.4以降で実装されたプロトコル3.0バージョンについて記載します。以前のプロトコルバージョンについての説明は、PostgreSQLの以前のリリースの文書を参照してください。初めの開始要求メッセージは、サーバに対し、クライアントが使用する予定のプロトコルバージョンを通知します。その後、サーバは可能であればそのプロトコルに従います。

複数のクライアントにサービスを効率的に提供するために、サーバは各クライアント毎に新規の"バックエンド"プロセスを起動します。現在の実装では、サーバに接続が届いたことを検知すると即座に新しい子プロセスが生成されます。しかし、これはプロトコルに対して透過的です。プロトコルという意味では、"バックエンド"と"サーバ"という用語は相互交換可能です。同様に"フロントエンド"と"クライアント"も相互交換可能です。

46.1. 概要

このプロトコルでは、接続開始と通常操作で段階が分かれています。接続開始段階で、フロントエンドはサーバへの接続を開き、サーバの義務を履行できるよう自身を証明します。（これは使用する認証方法に応じて、単一のメッセージになったり、複数のメッセージになったりします。）すべてうまく行けば、サーバはフロントエンドに状態情報を送信し、最終的に通常操作段階に入ります。初期の開始要求メッセージを除いて、プロトコルのこの部分はサーバによって駆動されます。

通常操作の間、フロントエンドは問い合わせやその他のコマンドをバックエンドに送信し、バックエンドは問い合わせ結果やその他の応答を返送します。（NOTIFYのように）バックエンドから依頼されずにメッセージが送信されるまれな場合がありますが、セッションのこの部分のほとんどはフロントエンドの要求によって駆動されます。

セッションの終了は通常フロントエンドが選択することですが、特定の場合はバックエンドによって強制される可能性があります。どちらの場合でも、バックエンドが接続を閉ざす時、終了前に実行中の（未完の）トランザクションをすべてロールバックします。

通常操作中は、SQLコマンドを2つのサブプロトコルのうちのいずれかによって実行することができます。"簡易問い合わせ"プロトコルでは、フロントエンドはテキストで問い合わせ文字列を単に送信し、バックエンドによって解析され、即実行されます。"拡張問い合わせ"プロトコルでは、問い合わせの処理は、解析、パラメータ値の結び付け、そして実行という複数の段階に分離されます。これは複雑性が加わりますが、柔軟性と性能という点で利点が生れます。

通常操作には、さらに、COPYのような特殊な操作向けのサブプロトコルがあります。

46.1.1. メッセージ処理の概要

すべての通信はメッセージストリームを介します。メッセージの先頭バイトはメッセージ種類を識別するもの、次の4バイトはメッセージの残りの長さを指定するものです（この長さにはメッセージ種類バイトは含まれませんが、自身を含んで数えられます）。残りのメッセージの内容は、メッセージ種類で決まります。歴史的な理由のため、一番初めにクライアントから送信されるメッセージ（開始メッセージ）にはメッセージ種類バイトはありません。

メッセージストリームの同期ずれを防ぐために、サーバとクライアントの両方は、通常、メッセージの内容を処理し始める前に、（バイト数を使用して）メッセージ全体をバッファ内に読み込みます。これにより、その内容を処理する時にエラーが検出された場合に、簡単に復旧することができます。（メッセージをバッファするために十分なメモリがない場合のような）極限状況では、受信側はメッセージの読み取りを再開する前にどれだけの量の入力を飛ばすかどうかを決定するために、バイト数を利用することができます。

反対に、サーバとクライアントの両方は、不完全なメッセージを決して送信しないように注意しなければなりません。これは通常、送信する前にバッファ内のメッセージ全体を整列させることで行われます。メッセージの送受信の途中で通信エラーが発生した場合、メッセージ境界の同期を復旧できる望みはほとんどありませんので、実用的な唯一の応答は通信を中断することです。

46.1.2. 拡張問い合わせの概要

拡張問い合わせプロトコルでは、SQLコマンドの実行は複数の段階に分割されます。段階間で保持される状態は、プリペアド文とポータル2種類のオブジェクトで表現されます。プリペアド文は、テキスト問い合わせ文字列の解析、意味解析を表現します。プリペアド文は実行準備が整ったことを示すものではありません。パラメータの特定の値が欠落しているかもしれないからです。ポータルは、すべてのパラメータ値が設定され、実行準備が整った、あるいは、既に一部実行された文を表現します。（SELECT文では、ポータルは開いているカーソルと等価です。しかし、カーソルはSELECT以外の文を扱いませんので、ここでは異なる用語を使用するよう選択しました。）

実行サイクル全体は、テキストの問い合わせ文字列からプリペアド文を生成する解析段階、プリペアド文と必要なパラメータ値によりポータルを作成するバインド段階、ポータルの問い合わせを実行する実行段階からなります。行を返す問い合わせ（SELECT、SHOWなど）の場合、操作を完了させるために複数の実行段階が必要とすることができるように、実行段階に限定された行数のみを取り出すよう指示することができます。

バックエンドは複数のプリペアド文とポータルの経過を追うことができます（しかし、1つのセッション内でのみ存在可能です。複数のセッションで共有することはできません）。存在するプリペアド文とポータルは、作成時に割り当てられた名前参照されます。さらに、"無名の"プリペアド文とポータルも存在します。これらは名前付きのオブジェクトとほとんど同じ動きをしますが、問い合わせを一回だけ実行し、その後に破棄する場合に備えて、これらに対する操作は最適化されています。一方、名前付きオブジェクトの操作は複数回の使用を想定して最適化されています。

46.1.3. 書式と書式コード

特定のデータ型のデータはいくつかの異なる書式で転送することができます。 PostgreSQL 7.4の時点でサポートしている書式は"テキスト"と"バイナリ"のみですが、プロトコルは将来の拡張に備えて準備をしています。 任意の値の必要な書式は書式コードで指定されます。 クライアントは、転送されるパラメータ値それぞれに書式コードを指定することも、問い合わせ結果の列それぞれに書式コードを指定することもできます。 テキストは書式コード0、バイナリは書式コード1です。 他の書式コードは将来の定義用に予約されています。

値のテキスト表現は、特定のデータ型の入出力変換関数で生成され、受け付けられる何らかの文字列です。 転送時の表現では、ヌル終端文字はありません。 フロントエンドでC言語文字列として処理したい場合は、必ず受信した値にヌル終端文字を追加しなければなりません。 （ついですが、テキスト書式ではヌルを埋め込むことはできません。）

整数用のバイナリ表現はネットワークバイト順（先頭に最上位バイト）を使用します。 他のデータ型のバイナリ表現については、文書もしくはソースコードを参照してください。 複雑なデータ型のバイナリ表現はサーバのバージョンによって異なる可能性があることに注意してください。 通常、テキスト書式がより移植性が高い選択肢です。

[前のページ](#)

フロントエンド/バックエンドプロトコル

[ホーム](#)
[上に戻る](#)

[次のページ](#)
メッセージの流れ

46.2. メッセージの流れ

本節では、メッセージの流れと各メッセージ種類のセマンティックスを説明します。（各メッセージの正確な表現の詳細については[項46.5](#)で説明します。） 開始、問い合わせ、関数呼び出し、`COPY`、終了といった接続状態に応じて、複数の異なるサブプロトコルがあります。 また、開始段階の後の任意の時点で発生する可能性がある、非同期操作（通知応答やコマンドのキャンセルを含む）用の特別な準備もあります。

46.2.1. 開始

セッションを開始するために、フロントエンドはサーバへの接続を開き、開始メッセージを送信します。 このメッセージには、ユーザ名と接続を希望するデータベース名が含まれます。 これはまた、使用する特定のプロトコルバージョンを識別します。（オプションとして、開始メッセージに、実行時パラメータの追加設定を含めることもできます。） サーバはその後、この情報と設定ファイル（`pg_hba.conf`など）の内容を使用して、接続が暫定的に受け付けられるかどうか、そして（もしあれば）どのような追加認証が必要かを決定します。

サーバはその後、適切な認証要求メッセージを送信します。 フロントエンドはこれに適切な認証応答メッセージ（パスワードなど）で答えなければなりません。 GSSAPIとSSPIを除くすべての認証方式では、多くても1つの要求と1つの応答が存在します。 認証方式の中には、フロントエンドからの応答をまったく必要としないものもあり、その場合、認証要求も発生しません。 GSSAPIとSSPIでは認証を完了するために複数のパケットの交換が必要となるかもしれません。

認証サイクルは、サーバによって接続要求を拒絶する（ErrorResponse）か、あるいはAuthenticationOkを送信することで終わります。

この段階でサーバから送信される可能性があるメッセージを以下に示します。

ErrorResponse

接続試行が拒絶されました。 サーバはその後即座に接続を閉じます。

AuthenticationOk

認証情報の交換が正常に完了しました。

AuthenticationKerberosV5

フロントエンドはここでサーバとのKerberos V5認証ダイアログ（ここでは説明しません。 Kerberos仕様の一部）に参加する必要があります。 これが成功すれば、サーバはAuthenticationOk応答を行います。 失敗すれば、ErrorResponse応答を行います。

AuthenticationCleartextPassword

フロントエンドはここで平文形式のパスワードを含むPasswordMessageを送信する必要があります。 これが正しいパスワードであればサーバはAuthenticationOk応答を行います。 さもなくば、ErrorResponse応答を行います。

AuthenticationMD5Password

フロントエンドはここでMD5で暗号化したパスワード（とユーザ名）を再度AuthenticationMD5Passwordメッセージで指定されたランダムな4バイトのソルトを使用して暗号化したものを含むPasswordMessageを送信する必要があります。 これが正しいパスワードであればサーバはAuthenticationOk応答を行います。 さもなくば、ErrorResponse応答を行います。 実際のPasswordMessageを`concat('md5', md5(concat(md5(concat(password, username)), random-salt)))`というSQLで計算することができます。（`md5()`関数がその結果を16進数表記で返すことに注意してください。）

AuthenticationSCMCredential

この応答はSCM資格証明メッセージをサポートするプラットフォーム上のローカルなUnixドメイン接続でのみあり得ます。 フロントエンドはSCM資格証明メッセージを発行し、その後単一のデータバイトを送信する必要があります。（データバイトの内容には意味はありません。 これはサーバが資格証明メッセージの受信にどれだけ待機すればよいのかを確実にするためだけに使用されます。） 資格証明が受け付け可能であれば、サーバはAuthenticationOk応答を行います。 さもなくば、ErrorResponse応答を行います。（この種類のメッセージは9.1より前のサーバでのみ発行されます。 最終的にはプロトコル仕様から削除されるかもしれません。）

AuthenticationGSS

ここでフロントエンドはGSSAPIの調停を開始しなければなりません。 これに対する応答におけるGSSAPIデータストリームの最初の部分で、フロントエンドはPasswordMessageを送信します。 さらにメッセージが必要となる場合、サーバはAuthenticationGSSContinueで応答します。

AuthenticationSSPI

ここでフロントエンドはSSPI調停を開始しなければなりません。 これに対する応答におけるSSPIデータストリームの最初の部分で、フロントエンドはPasswordMessageを送信します。 さらにメッセージが必要となる場合、サーバはAuthenticationGSSContinueで応答します。

AuthenticationGSSContinue

このメッセージには、GSSAPIまたはSSPI調停の直前の段階（AuthenticationGSS、AuthenticationSSPIまたは前回のAuthenticationGSSContinue）についての応答データが含まれます。 このメッセージ内のGSSAPIまたはSSPIデータが認証を完結させるため、更なる追加データが必要であることを指示している場合、フロントエンドは他のPasswordMessageとしてデータを送信しなければなりません。 このメッセージでGSSAPIまたはSSPI認証が完了すれば、次にサーバはAuthenticationOkを送信して認証が成功したことを示すか、あるいはErrorResponseを送信して失敗したことを示します。

サーバが要求した認証方式をフロントエンドがサポートしていない場合、フロントエンドは即座に接続を閉じます。

AuthenticationOkを受け取った後、フロントエンドはさらにサーバからのメッセージを待機する必要があります。 この段階で、バックエンドプロセスが起動し、このフロントエンドは単なる関心を有する第三者となります。 開始試行が失敗（ErrorResponse）する可能性がまだありますが、通常、バックエンドは何らかのParameterStatusメッセージ、BackendKeyData、そして最後にReadyForQueryを送信します。

この段階の期間中、バックエンドは開始メッセージで与えられた任意の実行時パラメータの追加設定を適用しようとします。 成功した場合は、これらの値はセッションのデフォルトになります。 エラーが発生した場合はErrorResponseを行い、終了します。

この段階でバックエンドから送信される可能性があるメッセージを以下に示します。

BackendKeyData

このメッセージは、フロントエンドがキャンセル要求を後で送信できるようにしたい場合に保存しなければならない、秘密キーデータを用意します。 フロントエンドはこのメッセージに応答してはいけませんが、ReadyForQueryメッセージの監視を続けなくてはなりません。

ParameterStatus

このメッセージは、フロントエンドに現在（初期）の`client_encoding`や`DateStyle`などのバックエンドパラメータの設定情報を通知します。 フロントエンドはこのメッセージを無視しても、将来の使用に備えてその設定を記録しても構いません。 詳細は[項46.2.6](#)を参照してください。 フロントエンドはこのメッセージに応答してはいけませんが、ReadyForQueryメッセージの監視を続けなくてはなりません。

ReadyForQuery

開始処理が完了しました。 フロントエンドはここでコマンドを発行することができます。

ErrorResponse

開始処理が失敗しました。 接続はこのメッセージの送信後に閉ざされます。

NoticeResponse

警告メッセージが発行されました。 フロントエンドはこのメッセージを表示し、ReadyForQueryもしくはErrorResponseメッセージの監視を続けなければなりません。

ReadyForQueryメッセージは各コマンドサイクルの後にバックエンドが発行するものと同じものです。 フロントエンドのコーディングにおいて必要であれば、ReadyForQueryをコマンドサイクルの開始とみなしても構いませんし、ReadyForQueryを開始段階とその後の各コマンドサイクルの終端とみなしても構いません。

46.2.2. 簡易問い合わせ

フロントエンドがQueryメッセージをバックエンドに送信することで、簡易問い合わせサイクルが開始されます。 このメッセージには、テキスト文字列で表現されたSQLコマンド（またはコマンド）が含まれます。 そうすると、バックエンドは、問い合わせコマンド文字列の内容に応じて1つ以上の応答を送信し、最終的にReadyForQueryを応答します。 ReadyForQueryは、新しいコマンドを安全に送信できることをフロントエンドに知らせます。（実際には、フロントエンドが他のコマンドを発行する前にReadyForQueryを待機することは不要です。 しかし、フロントエンドは、前のコマンドが失敗し、発行済みの後のコマンドが成功した場合に何が起きるかを了解する責任を持たなければなりません。）

バックエンドから送信される可能性がある応答メッセージを以下に示します。

CommandComplete

SQLコマンドが正常に終了しました。

CopyInResponse

バックエンドがフロントエンドからのデータをテーブルにコピーする準備ができました。 [項46.2.5](#)を参照してください。

CopyOutResponse

バックエンドがデータをテーブルからフロントエンドにコピーする準備ができました。 [項46.2.5](#)を参照してください。

RowDescription

`SELECT`や`FETCH`などの問い合わせの応答の行がまさに返されようとしていることを示します。 このメッセージには、行の列レイアウトに関する説明が含まれます。 このメッセージの後に、フロントエンドに返される各行に対するDataRowメッセージが続きます。

DataRow

`SELECT`や`FETCH`などの問い合わせで返される行の集合の1つです。

EmptyQueryResponse

空の問い合わせ文字列が検知されました。

ErrorResponse

エラーが起きました。

ReadyForQuery

問い合わせ文字列の処理が終了しました。 問い合わせ文字列は複数のSQLコマンドが含まれる場合があるため、このことを通知するために分離したメッセージが送出されます。（CommandCompleteは文字列全体ではなく、1つのSQLコマンドの処理の終了を明らかにします。） 処理が成功またはエラーで終了したかどうかにかかわらずReadyForQueryは常に送出されます。

NoticeResponse

問い合わせに関して警告メッセージが発行されました。 警告メッセージは他の応答に対する追加のメッセージです。 したがって、バックエンドはそのコマンドの処理を続行します。

`SELECT`問い合わせ（あるいは、`EXPLAIN`や`SHOW`などの行集合を返す他の問い合わせ）に対する応答は、通常、RowDescription、0個以上のDataRowメッセージ、そしてその後のCommandCompleteから構成されます。 フロントエンドへの`COPY`もしくはフロントエンドからの`COPY`は[項46.2.5](#)で説明する特別なプロトコルを呼び出します。 他の種類の問い合わせは通常CommandCompleteメッセージのみを生成します。

問い合わせ文字列には（セミコロンで区切られた）複数の問い合わせが含まれることがありますので、バックエンドが問い合わせ文字列の処理を完了する前に、こうした応答シーケンスが複数発生する可能性があります。 ReadyForQueryは、文字列全体が処理され、バックエンドが新しい問い合わせ文字列を受け付ける準備が整った時点で発行されます。

完全に空の（空白文字以外の文字がない）問い合わせ文字列を受け取った場合、その応答は、EmptyQueryResponse、続いて、ReadyForQueryとなります。

エラーが発生した場合、ErrorResponse、続いて、ReadyForQueryが発行されます。 その問い合わせ文字列に対する以降の処理は（複数の問い合わせが残っていたとしても）すべて、ErrorResponseによって中断されます。 これは、個々の問い合わせで生成されるメッセージの並びの途中で発生する可能性があることに注意してください。

簡易問い合わせモードでは、読み出される値の書式は常にテキストです。 ただし、与えられたコマンドが`BINARY`オプション付きで宣言されたカーソルからの`FETCH`であった場合は例外です。 この場合は、読み出される値はバイナリ書式になります。 RowDescriptionメッセージ内で与えられる書式コードは、どの書式が使用されているかを通知します。

他の種類のメッセージの受信を待機している時、フロントエンドは常にErrorResponseとNoticeResponseメッセージを受け取る準備ができていなければなりません。 また、外部イベントのためにバックエンドが生成する可能性があるメッセージの扱いについては[項46.2.6](#)を参照してください。

メッセージの正しい並びを前提としてコーディングするのではなく、任意のメッセージ種類を、そのメッセージが意味を持つ任意の時点で受け付ける状態マシン形式でフロントエンドのコーディングを行うことを推奨します。

46.2.3. 拡張問い合わせ

拡張問い合わせプロトコルは、上述の簡易問い合わせプロトコルを複数段階に分解します。 予備段階の結果は複数回再利用できますので、効率が上がります。 さらに、問い合わせ文字列に直接埋め込むのではなく、データ値をパラメータとして分離して提供できる機能など、利用できる追加機能があります。

拡張プロトコルでは、フロントエンドはまず、テキストの問い合わせ文字列とオプションとしてパラメータプレースホルダのデータ型情報やプリペアド文のオブジェクトの宛先名（空文字列は無名のプリペアド文を選択）を含む、Parseメッセージを送信します。 この応答はParseCompleteまたはErrorResponseです。 パラメータデータ型はそのOIDで指定することができます。 指定がなければ、パーサは型指定のないリテラル文字列定数に対する方法と同じ方法でデータ型を推定します。

注意: パラメータデータ型をゼロと設定する、または、問い合わせ文字列内で使用されているパラメータ記号（`$n`）の数より短くパラメータ型のOIDの配列を作成す

ることで、指定しないまま残すことができます。他にも、パラメータの型をvoid（つまりvoid仮想型のOID）と指定するという特別な場合があります。これは、パラメータ記号を、実際のOUTパラメータである関数パラメータとして使用することができることを意味します。通常では、voidパラメータが使用される文脈はありませんが、関数パラメータリストにこうしたパラメータ記号があると、実質的には無視されます。例えば、\$3と\$4がvoid型を持つと指定された場合、foo(\$1,\$2,\$3,\$4)といった関数呼び出しは、2つのINと2つのOUT引数を持つ関数に一致します。

注意: Parseメッセージ内の問い合わせ文字列には、複数のSQL文を含めることはできません。さもないと、構文エラーが報告されます。この制限は簡易問い合わせプロトコルにはありませんが、複数のコマンドを持つブリパード文やポータルを許すと、プロトコルが複雑になり過ぎるため、拡張プロトコルではこの制限があります。

作成に成功すると、名前付きブリパード文オブジェクトは明示的に破棄されない限り現在のセッションが終わるまで残ります。無名のブリパード文オブジェクトは、次に無名のブリパード文を宛先に指定したParse文が発行されるまでの間のみに残ります。（単なるQueryメッセージでも無名のブリパード文オブジェクトは破壊されることに注意してください。）名前付きブリパード文は、他のParseメッセージで再定義する前に明示的に閉じなければなりません。しかし、これは無名のブリパード文では必要ありません。名前付きブリパード文はまた、SQLコマンドレベルでPREPAREとEXECUTEを使用して作成しアクセスすることができます。

ブリパード文が存在すると、Bindメッセージを使用してそれを実行可能状態にすることができます。Bindメッセージは、元となるブリパード文（空文字列は無名のブリパード文を表します）の名前、宛先となるポータル（空文字列は無名ポータルを表します）の名前、およびブリパード文内のパラメータプレースホルダに使用する値を与えます。与えられたパラメータ集合はブリパード文が必要とするものと一致しなければなりません。（Parseメッセージ内でvoidパラメータを1つでも宣言した場合、BindメッセージではそれらにはNULLを渡します。）また、Bindは問い合わせで返されるデータに使用する書式を指定します。書式は全体に対して指定することも、列単位で指定することも可能です。応答はBindCompleteもしくはErrorResponseです。

注意: テキスト出力とバイナリ出力との選択は、含まれるSQLコマンドに関係なく、Bindで与えられた書式コードで決定されます。拡張問い合わせプロトコルを使用する場合、カーソル宣言のBINARY属性は役に立ちません。

典型的に問い合わせ計画はBindメッセージが処理される時に作成されます。ブリパード文がパラメータを持たない場合、または繰り返し実行される場合、サーバは作成した計画を保管し、その後の同じブリパード文に対するBindメッセージの際に再利用する可能性があります。しかし、作成できる汎用化した計画が提供された特定のパラメータ値に依存する計画より効率が下りて劣化しないことが分かった場合のみ、このように動作します。プロトコルに注目している限り、これは透過的に行われます。

作成に成功すると、名前付きポータルオブジェクトは明示的に破棄されない限り現在のセッションが終わるまで残ります。無名ポータルは、トランザクションの終わり、もしくは、次に無名ポータルを宛先に指定したParse文が発行されるまでの間のみに残ります。（単なるQueryメッセージでも無名ポータルは破壊されることに注意してください。）名前付きポータルは、他のBindメッセージで再定義する前に明示的に閉じなければなりません。しかし、これは無名ポータルでは必要ありません。名前付きポータルはまた、SQLコマンドレベルでDECLARE CURSORとFETCHを使用して作成しアクセスすることができます。

ポータルが存在すると、Executeメッセージを使用してそれを実行することができます。Executeメッセージは、ポータル名（空文字列は無名ポータルを表します）と結果行数の最大値（ゼロは"fetch all rows"を意味します）を指定します。結果行数は、ポータルが行集合を返すコマンドを含む場合のみ意味があります。その他の場合では、コマンドは常に終わりで実行され、行数は無視されます。Executeで起こり得る応答は、ExecuteではReadyForQueryやRowDescriptionが発行されない点を除き、上述の簡易問い合わせプロトコル経由で発行された問い合わせの場合と同じです。

Executeがポータルの実行を完了する前に（非ゼロの結果行数に達したために）終了した場合、PortalSuspendedを送信します。このメッセージの出現は、フロントエンドに操作を完了させるためには同一のポータルに対して、別のExecuteを発行しなければならないことを通知します。元となったSQLコマンドが完了したことを示すCommandCompleteメッセージはポータルが完了するまで送信されません。したがって、Execute段階は常にCommandComplete、EmptyQueryResponse（空の問い合わせ文字列からポータルが作成された場合）、ErrorResponse、またはPortalSuspendedの中の、正確にどれかが出現することによって常に終了します。

拡張問い合わせメッセージの一連の流れのそれぞれの終了時、フロントエンドはSyncメッセージを発行しなければなりません。このパラメータのないメッセージにより、もしBEGIN/COMMITトランザクションブロックの内部でなければ、バックエンドは現在のトランザクションを閉じます（"閉ざす"とは、エラーがなければコミット、エラーがあればロールバックすることを意味します）。そして、ReadyForQuery応答が発行されます。Syncの目的は、エラーからの復旧用の再同期を行う時点を提供することです。拡張問い合わせメッセージの処理中にエラーが検出されると、バックエンドはErrorResponseを発行し、Syncが届くまでメッセージを読み、それを破棄します。その後、ReadyForQueryを発行し、通常のメッセージ処理に戻ります。（しかし、Sync処理中にエラーが検出された場合に処理が飛ばされないことに注意してください。これにより、各Syncに対してReadyForQueryが1つのみであることを保証します。）

注意: Syncによって、BEGINで開かれたトランザクションブロックが閉ざされることはありません。ReadyForQueryメッセージにはトランザクションの状態情報が含まれていますので、この状況を検出することができます。

これらの基本的な必要操作に加え、拡張問い合わせプロトコルで 사용할 ことができる、複数の省略可能な操作があります。

Describeメッセージ（ポータルの垂種）は、既存のポータルの名前（もしくは、無名ポータル用の空文字）を指定します。応答は、実行中のポータルで返される予定の行を記述するRowDescriptionメッセージです。ポータルが行を返す問い合わせを含まない場合はNoDataメッセージです。指定したポータルが存在しない場合はErrorResponseです。

Describeメッセージ（ステートメントの垂種）は、既存のブリパード文の名前（もしくは無名のブリパード文用の空文字）を指定します。応答は、文で必要とされるパラメータを記述するParameterDescriptionメッセージ、続いて、文が実行された場合に返される予定の行を記述するRowDescriptionメッセージ（もしくは文が行を返さない場合のNoDataメッセージ）です。指定したブリパード文が存在しない場合はErrorResponseが発行されます。Bindがまだ発行されていないので、返される列の書式はまだバックエンドでは不明であることに注意してください。RowDescriptionメッセージ内の書式コードフィールドはこの場合はゼロになります。

ティップ: ほとんどの状況では、フロントエンドはExecuteを発行する前に、返ってくる結果を解釈する方法を確実に判断できるように、Describeもしくはその垂種を実行すべきです。

Closeメッセージは、既存のブリパード文、もしくはポータルを閉ざし、リソースを解放します。存在しない文やポータルに対してCloseを発行してもエラーになりません。応答は通常CloseCompleteですが、リソースの解放に何らかの問題が発生した場合はErrorResponseになります。ブリパード文を閉じると、そこから構築され、開いたポータルが暗黙的に閉ざされることに注意してください。

Flushメッセージにより特定の出力が生成されることはありません。しかし、バックエンドに対して、出力バッファ内で待機しているデータを強制的に配送させます。フロントエンドが他のコマンドを発行する前にコマンドの結果を検証したい場合に、FlushはSync以外の拡張問い合わせコマンドの後に送信される必要があります。Flushを行わないと、バックエンドで返されるメッセージは、ネットワークオーバーヘッドを最小化する、最小限のパケット数にまとめられます。

注意: 簡易問い合わせメッセージは、おおよそ、無名のブリパード文とポータルオブジェクトを使用したパラメータなしのParse、Bind、ポータル用Describe、Execute、Close、Syncの流れと同一です。違いは、問い合わせ文字列内に複数のSQL文を受け付けられ、bind/describe/executeという流れがそれぞれが成功すれば自動的に行われる点です。他の違いとして、ParseCompleteやBindComplete、CloseComplete、NoDataメッセージが返されない点があります。

46.2.4. 関数呼び出し

関数呼び出しサブプロトコルにより、クライアントはデータベースのpg_procシステムカタログに存在する任意の関数を直接呼び出す要求を行うことができます。クライアントはその関数を実行する権限を持たなければなりません。

注意: 関数呼び出しサブプロトコルは、おそらく新しく作成するコードでは使用すべきではない古い機能です。同様の結果は、SELECT function(\$1, ...)を行うブリパード文を設定することで得ることができます。そして、この関数呼び出しサイクルをBind/Executeで置き換えることができます。

関数呼び出しサイクルはフロントエンドがFunctionCallメッセージをバックエンドに送ることによって起動されます。バックエンドは1つまたは複数の応答メッセージを関数呼び出しの結果に基づいて送り、最終的にReadyForQueryメッセージを送出します。ReadyForQueryはフロントエンドに対し新規の問い合わせまたは関数呼び出しを行っても安全確実であることを通知します。

バックエンドから送信される可能性がある応答メッセージを以下に示します。

ErrorResponse

エラーが起きました。
FunctionCallResponse
関数呼び出しが完了し、メッセージで与えられた結果が返されました。（関数呼び出しプロトコルは単一のスカラ結果のみを扱うことができます。行型や結果集合を扱うことはできません。）
ReadyForQuery
関数呼び出しの処理が終了しました。処理が成功またはエラーで終了したかどうかにかかわらずReadyForQueryは常に送出されます。
NoticeResponse
関数呼び出しに関して警告メッセージが出されました。警告メッセージは他の応答に対する追加のメッセージです。したがって、バックエンドはそのコマンドの処理を続行します。

46.2.5. COPY操作

COPY コマンドにより、サーバとの間で高速な大量データ転送を行うことができます。コピーインとコピーアウト操作はそれぞれ接続を別のサブプロトコルに切り替えます。これは操作が完了するまで残ります。

コピーインモード（サーバへのデータ転送）は、バックエンドがCOPY FROM STDIN SQL文を実行した時に起動されます。バックエンドはフロントエンドにCopyInResponseメッセージを送信します。フロントエンドはその後、ゼロ個以上のCopyDataメッセージを送信し、入力データのストリームを形成します。（このメッセージの境界は行の境界に何かしら合わせる必要ありませんが、往々にしてそれが合理的な選択となります。）フロントエンドは、CopyDoneメッセージ（正常に終了させる）、もしくは、CopyFailメッセージ（COPY SQL文をエラーで失敗させる）を送信することで、コピーインモードを終了させることができます。そして、バックエンドは、COPYが始まる前の、簡易もしくは拡張問い合わせプロトコルを使用するコマンド処理モードに戻ります。そして次に、CommandComplete（成功時）またはErrorResponse（失敗時）のどちらかを送信します。

コピーインモードの期間中にバックエンドがエラー（CopyFailメッセージの受信を含む）を検知すると、バックエンドはErrorResponseメッセージを発行します。拡張問い合わせメッセージ経由でCOPYコマンドが発行された場合、バックエンドはSyncメッセージを受け取るまでフロントエンドのメッセージを破棄するようになります。Syncメッセージを受け取ると、ReadyForQueryを発行し、通常処理に戻ります。簡易問い合わせメッセージでCOPYが発行された場合、メッセージの残りは破棄され、ReadyForQueryメッセージを発行します。どちらの場合でも、その後にフロントエンドによって発行されたCopyData、CopyDone、CopyFailは単に削除されます。

バックエンドは、コピーインモード期間中、FlushとSyncメッセージを無視します。その他の種類の非コピーメッセージを受け取ると、エラーになり、上述のようにコピーイン状態を中断します（クライアントライブラリがExecuteメッセージの後に、実行したコマンドがCOPY FROM STDINかどうかを検査することなく、常にFlushまたはSyncを送信できる、という便利さのためにFlushとSyncは例外です）。

コピーアウトモード（サーバからのデータ転送）は、バックエンドがCOPY TO STDOUT SQL文を実行した時に起動します。バックエンドはCopyOutResponseメッセージをフロントエンドに送信し、その後、ゼロ個以上のCopyDataメッセージ（常に行ごとに1つ）を、そして、CopyDoneを送信します。その後、バックエンドはCOPYが始まる前のコマンド処理モードに戻り、CommandCompleteを送信します。フロントエンドは（接続の切断やCancel要求の発行は例外ですが）転送を中断することはできません。しかし、不要なCopyDataとCopyDoneメッセージを破棄することは可能です。

コピーアウトモード期間中バックエンドはエラーを検知すると、ErrorResponseメッセージを発行し、通常処理に戻ります。フロントエンドはErrorResponseの受信をコピーアウトモードの終了として扱うべきです。

NoticeResponseおよびParameterStatusメッセージがCopyDataメッセージ間に散在することがあります。フロントエンドはこのような場合も扱わなければならない。また、他の種類の非同期メッセージ（[項46.2.6](#)を参照）も同様に準備すべきです。さもなくば、CopyDataまたはCopyDone以外の種類のメッセージがコピーアウトモードの終了として扱われてしまう可能性があります。

他にも、サーバへ、およびサーバからの高速一括データ転送を行うことができるコピーボーズというコピーに関連したモードがあります。コピーボーズモードは、walsenderモードのバックエンドがSTART_REPLICATION文を実行した時に初期化されます。バックエンドはCopyBothResponseメッセージをフロントエンドに送信します。この後バックエンドとフロントエンドの両方が、接続が終了するまでの間にCopyDataメッセージを送信できるようになります。[項46.3](#)を参照してください。

CopyInResponse、CopyOutResponse、CopyBothResponseメッセージには、フロントエンドに1行当たりの列数と各列で使用する書式コードを通知するためのフィールドが含まれています。（現在の実装では、COPY操作で与えられるすべての列は同一の書式を使用します。しかし、メッセージ設計においては、これを前提としていません。）

46.2.6. 非同期操作

バックエンドが、フロントエンドのコマンドストリームで特に依頼されていないメッセージを送信する場合があります。フロントエンドは、問い合わせ作業を行っていない時であっても常に、これらのメッセージを扱う準備をしなければなりません。少なくとも、問い合わせの応答の読み込みを始める前にこれらを検査すべきです。

外部の活動によって、NoticeResponseメッセージが生成されることがあり得ます。例えば、データベース管理者が"高速"データベース停止コマンドを実行した場合、バックエンドは接続を閉ざす前にこれを通知するためにNoticeResponseを送信します。したがって、たとえ接続が名目上待機状態であったとしても、フロントエンドは常にNoticeResponseメッセージを受け付け、表示する準備をすべきです。

ParameterStatusメッセージは、任意のパラメータの実際の値が変更され、それをバックエンドがフロントエンドに通知するべきであるとみなした場合は常に生成されます。ほとんどの場合、これはフロントエンドによるSET SQLコマンド実行に対する応答の中で起こります。これは実質的には同期していますが、管理者が設定ファイルを変更し、SIGHUPシグナルをサーバに送った場合にも、パラメータ状態の変更が発生することがあります。また、SETコマンドがロールバックされた場合、現在の有効値を報告するために適切なParameterStatusメッセージが生成されます。

現時点では、ParameterStatusを生成するパラメータ群は固定されています。それらはserver_version、server_encoding、client_encoding、application_name、is_superuser、session_authorization、DateStyle、IntervalStyle、TimeZone、integer_datetimes、およびstandard_conforming_stringsです。（8.0より前までのリリースでは、server_encoding、TimeZoneおよびinteger_datetimesは送信されませんでした。8.1より前までのリリースでは、standard_conforming_stringsは送信されませんでした。8.4より前のリリースでは、IntervalStyleは送信されませんでした。9.0より前のリリースでは、application_nameは送信されませんでした。）server_version、server_encodingおよびinteger_datetimesは仮想的なパラメータであり、起動後に変更することができないことに注意してください。これは今後変更される、あるいは設定変更可能になる可能性があります。したがって、フロントエンドは未知または注目していないParameterStatusを単に無視すべきです。

フロントエンドがLISTENコマンドを発行した場合、同じチャネル名に対しNOTIFYコマンドが実行された時にバックエンドはNotificationResponseメッセージ（NoticeResponseと間違えないように！）を送出します。

注意: 現在、NotificationResponseをトランザクションの外部でのみ送信することができます。このため、これはコマンド応答の流れの途中では起こりませんが、ReadyForQueryの直前に発生する可能性があります。しかし、これを前提にフロントエンドのロジックを設計することは避けてください。プロトコル内の任意の時点でNotificationResponseを受け付けられるようにすることを勧めます。

46.2.7. 処理中のリクエストの取り消し

問い合わせの処理中に、フロントエンドが問い合わせを取り消す可能性があります。取り消し要求は、効率を高めるために、接続を開いたバックエンドに対して直接送信されません。その問い合わせを処理中のバックエンドが、フロントエンドからの新しい入力があるかどうかを定期的に確認することは好ましくありません。取り消し要求はたいていの場合、頻繁には起こらないので、通常の状態においての負担を避けるため、多少扱いにくくなっています。

取り消し要求を出す場合、フロントエンドは通常の新規接続の時に送出されるStartupMessageメッセージではなくCancelRequestメッセージをサーバに送り、新規接続を開始します。サーバはこの要求を処理し、接続を切断します。セキュリティ上の理由から、取り消し要求メッセージに対し直接の回答はありません。

CancelRequestメッセージは、接続開始段階でフロントエンドに送られたものと同一の鍵データ（PIDと秘密鍵）を含んでいない場合は無視されます。現在バックエンドが実行中の処理に対するPIDと秘密鍵が要求と一致した場合、その現在の問い合わせ処理は中断されます。（現状では、これは、その問い合わせを処理しているバックエンドプロセスに対し特別なシグナルを送ることで実装されています。）

この取り消しシグナルは何の効果も生まないことがあります。例えば、バックエンドが問い合わせの処理を完了した後に届いた場合、効果がありません。もし取り消し処理が有効であれば、エラーメッセージとともに、現在のコマンドは終了されます。

セキュリティと効率上の理由による上述の実装の結果、フロントエンドは取り消し要求が成功したかどうかを直接判断することはできません。フロントエンドはバックエンドからの問い合わせの回答を待ち続けなければいけません。取り消しを要求することは単に現在の問い合わせを早めに終わらせ、成功ではなくエラーメッセージを出して不成功とする可能性を単に高める程度のもんです。

取り消し要求は、通常のフロントエンドとバックエンドの通信接続を通してではなく新規のサーバとの接続に送られるため、取り消される問い合わせを実行したフロントエンドだけでなく、任意のプロセスによっても送信することができます。このことはマルチプロセスアプリケーションを作るに当たって柔軟性を提供します。同時に、権限のない者が問い合わせを取り消そうとするといったセキュリティ上のリスクも持ち込みます。このセキュリティ上のリスクは、取り消し要求内に動的に生成される秘密キーを供給することを必須とすることで回避されます。

46.2.8. 終了

通常の洗練された終了手順はフロントエンドがTerminateメッセージを出し、すぐに接続を閉じることです。このメッセージを受け取るとすぐにバックエンドは接続を閉じ終了します。

まれに（管理者によるデータベース停止コマンドなど）、バックエンドはフロントエンドからの要求なしに切断することがあります。こうした場合、バックエンドは、接続を閉ざす前に切断理由を通知するエラーまたは警報メッセージの送信を試みます。

他にも、どちらかの側のコアダンプ、通信リンクの消失、メッセージ境界の同期の消失など各種失敗によって終了する状況があります。フロントエンドかバックエンドいずれかが予期しない接続の中断を検知した場合、後始末を行い終了しなければいけません。フロントエンドはもし自身が終了を望まない場合、サーバに再交信し新規のバックエンドを立ち上げる選択権を持っています。解釈できないメッセージ種類を受け取った時、おそらくメッセージ境界の同期が消失したことを示しますので、接続を閉ざすことを勧めます。

通常の終了、異常な終了のどちらの場合でも、開いているトランザクションはすべてコミットされずにロールバックされます。しかし、フロントエンドがSELECT以外の問い合わせを処理中に切断した場合、バックエンドはおそらく切断に気付く前にその問い合わせを完了させてしまうでしょう。その問い合わせがトランザクションブロック（BEGIN ... COMMITの並び）外部であった場合、切断に気付く前にその結果はコミットされる可能性があります。

46.2.9. SSLセッション暗号化

PostgreSQLがSSLサポート付きで構築された場合、フロントエンドとバックエンド間の通信をSSLで暗号化することができます。攻撃者がセッショントラフィックをキャプチャできるような環境における通信を安全にすることができます。SSLを使用したPostgreSQLセッションの暗号化に関する詳細は[項17.9](#)を参照してください。

SSL暗号化接続を開始するには、フロントエンドはまず、StartupMessageではなくSSLRequestを送信します。その後サーバはそれぞれSSLの実行を行うか行わないかを示すsかnを持つ1バイトの応答を返します。フロントエンドはその応答に満足できなければ、この時点で接続を切断することができます。sの後に継続するのであれば、サーバと間でSSL起動ハンドシェーク（ここではSSLの仕様に関しては説明しません）を行います。これが成功した場合、続いて通常のStartupMessageの送信を行います。この場合、StartupMessageおよびその後のデータはSSLにより暗号化されます。nの後に、通常のStartupMessageを送信することで暗号化なしで進みます。

また、フロントエンドはサーバからのSSLRequestに対するErrorMessage応答を取り扱うための用意もすべきです。これは、PostgreSQLにSSLサポートが追加される前のサーバの場合のみで発生します。（現在ではこうしたサーバは非常に古いものといえ、ほとんど存在しません。）この場合接続を切断しなければなりませんが、フロントエンドはSSL要求なしで新しく接続を開き、処理を進めることもできます。

最初のSSLRequestはCancelRequestメッセージを送信するために開いた接続で使用することもできます。

プロトコル自体には、サーバにSSL暗号化を強制する方法は用意されていませんが、管理者は認証検査の一方法として、暗号化されていないセッションを拒否するようにサーバを設定することができます。

46.3. ストリーミングレプリケーションプロトコル

ストリーミングレプリケーションを初期化するために、フロントエンドは開始メッセージにて replication パラメータを送信します。これはバックエンドに対して、SQL 文ではなく小規模なレプリケーションコマンド群を発行できるようになる、walsender モードに入るように伝えます。walsender モードでは簡易問い合わせプロトコルのみを使用することができません。walsender モードで受け付けられるコマンドは以下の通りです。

IDENTIFY_SYSTEM

サーバに自身を識別することを要求します。サーバは以下の 3 つのフィールドを持つ単一行の結果セットをもって応答します。

systemid

クラスタを識別する一意なシステム識別子です。これを使用してスタンバイを初期化するために使用するベースバックアップが同じクラスタに由来していることを検査することができます。

timeline

現在の TimelineID です。同様にスタンバイがマスタと一貫性を持つことを検査するために使用されます。

xlogpos

現在の xlog の書き出し位置です。ストリーミングを開始できるトランザクションログの既知の位置を得る際に有用です。

START_REPLICATION xxx/xxx

サーバに対して、WAL のストリーミングを xxx/xxx WAL 時点から開始するよう指示します。サーバが、例えば、要求された WAL の断片がすでに回収されているなど、エラーを返すことがあります。成功時サーバは CopyBothResponse メッセージで応答し、フロントエンドに対する WAL のストリーミングを開始します。WAL のストリーミングは接続が壊れるまで継続され、他に受け付けられるコマンドはありません。

WAL データは CopyData メッセージ群として送信されます。（これにより他の情報を混在させることができます。具体的にはサーバはストリーム開始後に失敗が起きた場合に ErrorResponse メッセージを送信することができます。）CopyData メッセージのペイロードは以下の書式に従います。

XLogData (B)

Byte1('w')

メッセージを WAL データとして識別します。

Byte8

このメッセージ内の WAL の開始点。書式は XLogRecPtr です。

Byte8

サーバ上の現在の WAL 終了点。書式は XLogRecPtr です。

Byte8

トランザクションの時点でのサーバのシステム時刻。書式は TimestampTz です。

Byte n

WAL データストリームの断片。

単一の WAL レコードが 2 つの CopyData メッセージに分かれることはありません。しかし WAL レコードが WAL ページ境界を跨る場合、継続レコードを用いてすでに分割されていますので、ページ境界で分割することができます。言い換えると、先頭の主 WAL レコードとその継続レコードは、別の CopyData メッセージとして分かれることがあります。

WAL データ内のすべてのフィールドと上記ヘッダが送信元サーバ用の書式であることに注意してください。エンディアンやタイムスタンプの書式は受信側が送信元のシステム識別子が自身の pg_control の内容と一致するかどうかを検証しない

限り予測できません。

WAL送信プロセスが（postmasterの停止中に）正常に終了する場合、終了前にCommandCompleteメッセージを送信します。 当然ながら異常終了の場合にはこれは送信されないかもしれません。

受信側のプロセスはいつでも送信側に、以下のメッセージ書式のいずれかを使用して、応答を返送することができます。

プライマリキープアライブメッセージ(B)

Byte1('k')

このメッセージを送信元キープアライブとして識別します。

Byte8

XLogRecPtr書式で指定される、サーバ上の現在のWAL終端。

Byte8

TimestampTz書式で指定される、送信時のサーバのシステムクロック。

スタンバイ状態の更新(F)

Byte1('r')

メッセージを受信側の状態更新として識別します。

Byte8

スタンバイにおいて受信しディスクに書き込まれた最終WAL/バイト+1のXLogRecPtr書式の場所。

Byte8

スタンバイにおいてディスクに吐き出された最終WAL/バイト+1のXLogRecPtr書式の場所。

Byte8

スタンバイにおいて適用された最終WAL/バイト+1のXLogRecPtr書式の場所。

Byte8

送信時のサーバのシステムクロック（TimestampTz書式）です。

ホットスタンバイフィードバックメッセージ(F)

Byte1('h')

メッセージをホットスタンバイのフィードバックメッセージとして識別します。

Byte8

送信時のサーバのシステムクロック（TimestampTz書式）です。

Byte4

スタンバイの現在のxminです。

Byte4

スタンバイの現在のエポックです。

BASE_BACKUP [LABEL '*label*'] [PROGRESS] [FAST] [WAL] [NOWAIT]

サーバにベースバックアップのストリーミングを始めるよう指示します。 システムはバックアップが開始される前に自動的にバックアップモードになり、バックアップが完了した時に取り出されます。 以下のオプションを受け付けることができます。

LABEL '*label*'

バックアップのラベルを設定します。 指定がない場合、`base backup`というバックアップラベルが使用されます。 ラベルについての引用符付け規則は、[standard conforming strings](#)を有効にした場合の標準SQLの文字列の規

則と同じです。

PROGRESS

進行状況の報告を生成するために必要な情報を要求します。これは、ストリームが完了するまでにどのくらいかかるかを計算するために使用することができる、各テーブル空間のヘッダ内の概算容量を返送します。これは、転送を始める前のすべてのファイルサイズを1度数え上げることで計算されます。これ自体が性能に与える悪影響があるかもしれません。特に最初のデータがストリームされるまでにより多くの時間がかかる可能性があります。データベースファイルはバックアップの間変更される可能性がありますので、容量は概算に過ぎず、概算時と実ファイルを送信するまでの間に増減される可能性があります。

FAST

高速チェックポイントを要求します。

WAL

バックアップ内に必要なWALセグメントを含めます。ベースディレクトリtarファイルのpg_xlogディレクトリにある、バックアップの開始から終了までのすべてのファイルが含まれます。

NOWAIT

デフォルトでは、バックアップは必要な最終xlogセグメントがアーカイブされるまで待機します。ログアーカイブが有効でない場合は警告が発せられます。NOWAITにより、必要なログが利用できるようになったことを確認することをクライアント側の責任として、この待機や警告が無効になります。

バックアップを開始する時、サーバはまず2つの通常の結果セットを送信し、続けて1つ以上のCopyResponse結果を送信します。

最初の通常の結果セットには、単一行の単一列という形でXLogRecPtr書式のバックアップの開始位置が含まれます。

2番目の通常の結果セットには各テーブル空間につき1行を持ちます。この行のフィールドは以下の通りです。

spcoid

テーブル空間のoidです。ベースディレクトリの場合はNULLです。

spclocation

テーブル空間ディレクトリのフルパスです。ベースディレクトリの場合はNULLです。

size

進行状況報告が要求された場合は、テーブル空間の概算容量です。要求されていない場合はNULLです。

2番目の通常の結果セットの後、1つ以上のCopyResponse結果が送信されます。1つはPGDATA用、pg_default、pg_global以外の追加のテーブル空間ごとに1つ送信されます。CopyResponse結果内のデータは、テーブル空間の内容のtar形式（POSIX 1003.1-2008標準で規定された"ustar交換形式"に従う）ダンプです。ただし標準で規定された最後の2つのゼロブロックは省略されています。このtarデータが終わった後、最終の通常結果セットが送信されます。

データディレクトリと各テーブル空間のtarアーカイブには、そのディレクトリ内のファイルがPostgreSQLファイルかそのディレクトリに追加された他のファイルかに関係なく、すべて含まれます。以下に除かれるファイルを示します。

- postmaster.pid
- postmaster.opts
- サブディレクトリを含むpg_xlog。バックアップがwalファイルを含めて実行される場合、合成された版のpg_xlogが含まれます。これにはバックアップが動作するために必要なファイルのみが含まれ、残りの内容は含まれません。

サーバ上の基盤となるファイルシステムがサポートする場合、所有者、グループ、ファイルのモードが設定されます。

すべてのテーブル空間が送信された後、最終の通常の結果セットが送信されます。この結果セットには、単一行の単一列という形でXLogRecPtr書式のバックアップの終了位置が含まれます。

46.4. メッセージのデータ型

本節ではメッセージの中で使われる基本的なデータ型を説明します。

`Int $n$ ( $i$ )`

ネットワークバイト順（最上位バイトが先頭）における n ビットの整数。もし i が指定されていれば、それがそのまま使われます。さもなければ変数です。例えば`Int16`、`Int32(42)`などです。

`Int $n$ [ $k$ ]`

n ビット整数の要素数 k の配列で、それぞれはネットワークバイト順です。配列サイズ k は常にメッセージの前のフィールドで決定されます。例えば`Int16[M]`です。

`String( $s$ )`

NULL終端文字列（C様式文字列）。文字列には長さ制限の指定はありません。 s が指定されていれば、それがそのまま使われます。さもなければ値は変数です。例えば`String`、`String("user")`などです。

注意: バックエンドから返すことができる文字列の長さには事前に定義された制限はありません。フロントエンドではメモリに収まるものはすべて受け入れられるように拡張可能なバッファを使用するコーディング戦略を勧めます。これが実行できないのであれば、文字列全体を読み取り、固定長バッファに合わない後の方の文字を破棄してください。

`Byte $n$ ( $c$ )`

厳密に n バイト。フィールド幅 n が定数でない場合、メッセージの前のフィールドから決定されます。 c が指定されていれば、それがそのまま使われます。例えば `Byte2`、`Byte1('¥n')`などです。

[前のページ](#)

ストリーミングレプリケーションプロトコル

[ホーム](#)

[上に戻る](#)

[次のページ](#)

メッセージの書式

46.5. メッセージの書式

本節ではそれぞれのメッセージの詳細書式について説明します。それぞれにはフロントエンド (F)、バックエンド (B) あるいは双方 (F & B) から送出されることを示す印が付いています。各メッセージには先頭にバイト数を持っていますが、バイト数を参照しなくてもメッセージの終わりを検知できるようにメッセージ書式は定義されています。これは有効性検査を補助します。(CopyDataメッセージはデータストリームの一部を形成しますので例外です。個々のCopyDataメッセージの内容は自身でも解釈することができません。)

AuthenticationOk (B)

Byte1('R')

認証要求としてメッセージを識別します。

Int32(8)

自身を含む、メッセージ内容の長さ (バイト単位)。

Int32(0)

認証が成功したことを指定します。

AuthenticationKerberosV5 (B)

Byte1('R')

メッセージを認証要求として識別します。

Int32(8)

自身を含む、メッセージ内容の長さ (バイト単位)。

Int32(2)

Kerberos V5認証が必要であることを指定します。

AuthenticationCleartextPassword (B)

Byte1('R')

メッセージを認証要求として識別します。

Int32(8)

自身を含む、メッセージ内容の長さ (バイト単位)。

Int32(3)

平文パスワードが必要であることを指定します。

AuthenticationMD5Password (B)

Byte1('R')

メッセージが認証要求であることを識別します。

Int32(12)

自身を含む、メッセージ内容の長さ (バイト単位)。

Int32(5)

MD5暗号化パスワードが必要であることを指定します。

Byte4

パスワード暗号化用ソルトです。

AuthenticationSCMCredential (B)

Byte1('R')

メッセージが認証要求であることを識別します。

Int32(8)

自身を含む、メッセージ内容の長さ（バイト単位）。

Int32(6)

SCM資格証明メッセージが必要であることを指定します。

AuthenticationGSS (B)

Byte1('R')

メッセージが認証要求であることを識別します。

Int32(8)

自身を含む、メッセージ内容の長さ（バイト単位）。

Int32(7)

GSSAPI 認証証明メッセージが必要であることを指定します。

AuthenticationSSPI (B)

Byte1('R')

メッセージが認証要求であることを識別します。

Int32(8)

自身を含む、メッセージ内容の長さ（バイト単位）。

Int32(9)

SSPI 認証証明メッセージが必要であることを指定します。

AuthenticationGSSContinue (B)

Byte1('R')

メッセージが認証要求であることを識別します。

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

Int32(8)

このメッセージがGSSPAIまたはSSPIデータを含むことを指定します。

Byten

GSSAPIまたはSSPI認証データです。

BackendKeyData (B)

Byte1('K')

メッセージが取消用キーデータであることを識別します。もし、後でCancelRequestメッセージを発行する可能性があれば、フロントエンドはこの値を保存しなければなりません。

Int32(12)

自身を含む、メッセージ内容の長さ（バイト単位）。

Int32

このバックエンドのプロセスID。

Int32

このバックエンドの秘密鍵。

Bind (F)

Byte1('B')

このメッセージがBindコマンドであることを識別します。

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

String

宛先ポータルの名前です（空文字列で無名ポータルを選択します）。

String

元となるプリペアド文の名前です（空文字列で無名のプリペアド文を選択します）。

Int16

後続するパラメータ書式コードの数（以下では c で表します）。これは、パラメータが存在しない、あるいはすべてのパラメータがデフォルトの書式（テキスト）を使用することを示す0に、指定する書式コードをすべてのパラメータに適用することを示す1にすることができます。さもなくば、これは実際のパラメータ数と同じになります。

Int16[c]

パラメータ書式コード。現在、それぞれは0（テキスト）もしくは1（バイナリ）のいずれかでなければなりません。

Int16

後続するパラメータ値の数（ゼロの場合もあります）。これは問い合わせが必要とするパラメータ数と一致する必要があります。

次に、各パラメータに対して、以下のフィールドのペアが現れます。

Int32

パラメータ値のバイト単位の長さ（これには自身は含まれません）。ゼロにすることもできます。特別な場合として、-1はNULLというパラメータ値を示します。NULLの場合、後続の値用のバイトはありません。

Byte n

関連する書式コードで示される書式におけるパラメータの値。 n は上述の長さです。

最後のパラメータの後に、以下のフィールドが現れます。

Int16

後続する結果列書式コードの数（以下では R で表します）。これは、結果列が存在しないことを示す0、あるいはすべての結果列が（もしあれば）デフォルトの書式コード（テキスト）を使用することを示す1にすることができます。さもなくば、問い合わせの結果列の実際の数と同じになります。

Int16[R]

結果列の書式コード。現在それぞれは0（テキスト）もしくは1（バイナリ）のいずれかでなければなりません。

BindComplete (B)

Byte1('2')

メッセージがBind完了指示子であることを識別します。

Int32(4)

自身を含む、メッセージ内容の長さ（バイト単位）。

CancelRequest (F)

Int32(16)

自身を含む、メッセージ内容の長さ（バイト単位）。

Int32(80877102)

取消要求コードです。この値は、最上位16ビットに¹²³⁴が、下位16ビットに⁵⁶⁷⁸を持つように選択されます。（混乱を防ぐため、このコードはプロトコルバージョン番号と同一になってはいけません。）

Int32

対象バックエンドのプロセスIDです。

Int32

対象バックエンドの秘密鍵です。

Close (F)

Byte1('C')

メッセージがCloseコマンドであることを識別します。

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

Byte1

プリペアド文を閉ざす時は's'。ポータルを閉ざす時は'p'です。

String

閉ざすプリペアド文またはポータルの名前です（空文字列で無名のプリペアド文または無名ポータルを選択します）。

CloseComplete (B)

Byte1('3')

メッセージがClose完了指示子であることを識別します。

Int32(4)

自身を含む、メッセージ内容の長さ（バイト単位）。

CommandComplete (B)

Byte1('C')

メッセージがコマンド完了応答であることを識別します。

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

String

コマンドタグです。これは通常どのSQLコマンドが完了したかを表す単一の単語です。

INSERT コマンドの場合、タグはINSERT oid rowsです。ここでrowsは挿入された行数です。oidは、rowsが1、かつ、対象テーブルがOIDを持つ場合、挿入された行のオブジェクトIDです。さもなければ、oidは0です。

DELETE コマンドの場合、タグはDELETE rowsです。ここでrowsは削除された行数です。

UPDATE コマンドの場合、タグはUPDATE rowsです。ここでrowsは更新された行数です。

SELECTまたはCREATE TABLE ASの場合、タグはSELECT rowsとなります。ここでrowsは取り込んだ行数です。

MOVE コマンドの場合、タグはMOVE rowsです。ここでrowsは、カーソル位置が何行変更されたかを示す数です。

FETCH コマンドの場合、タグはFETCH rowsです。ここでrowsは、カーソルから何行取り出したかを示す行数です。

COPY コマンドの場合、タグはCOPY rowsです。ここでrowsは、コピーされた行数です（注意：この行数はPostgreSQL 8.2以降でのみ出力されます）。

CopyData (F & B)

Byte1('d')

メッセージがデータのCOPYであることを識別します。

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

Byte n

COPY データストリームの一部を形成するデータです。バックエンドから送信されるメッセージは、常に1つのデータ行に対応します。しかし、フロントエンドから送信されるメッセージは任意のデータストリームに分割される可能性があります。

CopyDone (F & B)

Byte1('c')

メッセージがCOPY完了指示子であることを識別します。

Int32(4)

自身を含む、メッセージ内容の長さ（バイト単位）。

CopyFail (F)

Byte1('f')

メッセージがCOPY失敗指示子であることを識別します。

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

String

失敗の原因を報告するエラーメッセージです。

CopyInResponse (B)

Byte1('G')

メッセージがStart Copy Inの応答であることを識別します。フロントエンドはここで必ずコピーインデータを送信しなければなりません（まだ準備ができていない場合はCopyFailメッセージを送信してください）。

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

Int8

0はCOPY全体の書式がテキスト（行は改行で区切られ、列は区切り文字などで区切られます）であることを示します。1は、コピー全体の書式がバイナリ（DataRowの書式と同様）であることを示します。より詳細については[COPY](#)を参照してください。

Int16

コピーされるデータ内の列数です（以下では N と表されます）。

Int16[N]

各列で使用する書式コードです。現在それぞれは0（テキスト）または1（バイナリ）でなければなりません。コピー全体の書式がテキストの場合、すべてが0でなければなりません。

CopyOutResponse (B)

Byte1('H')

メッセージがStart Copy Outの応答であることを識別します。このメッセージの後にコピーアウトデータが続きます。

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

Int8

0はCOPY全体の書式がテキスト（行は改行で区切られ、列は区切り文字などで区切られます）であることを示します。1はコピー全体の書式がバイナリ（DataRowの書式同様）であることを示します。詳細については[COPY](#)を参照してください。

Int16

コピーされるデータ内の列数です（以下では N と表されます）。

Int16[N]

各列で使用する書式コードです。現在それぞれは0（テキスト）または1（バイナリ）でなければなりません。コピー全体の書式がテキストの場合、すべてが0でなければなりません。

CopyBothResponse (B)

Byte1('W')

メッセージがStart Copy Bothの応答であることを識別します。このメッセージはストリーミングレプリケーションのみで使用されます。

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

Int8

0はCOPY全体の書式がテキスト（行は改行で区切られ、列は区切り文字などで区切られます）であることを示します。1はコピー全体の書式がバイナリ（DataRowの書式同様）であることを示します。詳細については[COPY](#)を参照してください。

Int16

コピーされるデータ内の列数です（以下では N と表されます）。

Int16[N]

各列で使用する書式コードです。現在それぞれは0（テキスト）または1（バイナリ）でなければなりません。コピー全体の書式がテキストの場合、すべてが0でなければなりません。

DataRow (B)

Byte1('D')

メッセージがデータ行であることを識別します。

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

Int16

後に続く列値の数です（ゼロの場合もあります）。

次に、各列について以下のフィールドのペアが現れます。

Int32

列値のバイト単位の長さです（これには自身は含まれません）。ゼロとすることもできます。特別な場合として、-1はNULLという列値を示します。NULLの場合、後続の値用のバイトはありません。

Byte*n*

関連する書式コードで示される書式における列の値。*n*は上述の長さです。

Describe (F)

Byte1('D')

メッセージがDescribeコマンドであることを識別します。

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

Byte1

プリペアド文の記述の場合は's'。ポータルの記述の場合は'p'です。

String

記述を求めるプリペアド文またはポータルの名前です（空文字列で無名のプリペアド文または無名ポータルを選択します）。

EmptyQueryResponse (B)

Byte1('I')

メッセージが空の問い合わせ文字列に対する応答であることを識別します（これはCommandCompleteを置き換えます）。

Int32(4)

自身を含む、メッセージ内容の長さ（バイト単位）。

ErrorResponse (B)

Byte1('E')

メッセージがエラーであることを識別します。

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

このメッセージの本体には、ゼロバイトを終端として後続する、1つ以上の識別されるフィールドが含まれます。フィールドは任意の順番で現れる可能性があります。各フィールドには以下があります。

Byte1

フィールド種類を識別するコードです。ゼロならば、メッセージの終端であり、後続する文字列がないことを表します。[項46.6](#)に、現時点でフィールド種類として定義されているものを列挙します。将来もっと多くのフィールド種類が追加される可能性がありますので、フロントエンドは、認知できない種類のフィールドに対して何もせずに無視すべきです。

String

フィールド値です。

Execute (F)

Byte1('E')

メッセージがExecuteコマンドであることを識別します。

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

String

実行するポータルの名前です。（空文字列で無名ポータルを選択します）。

Int32

ポータルが行を返す問い合わせの場合、返される行数の最大値です（他の問い合わせでは無視されます）。ゼロは"無制限"を表します。

Flush (F)

Byte1('H')

メッセージがFlushコマンドであることを識別します。

Int32(4)

自身を含む、メッセージ内容の長さ（バイト単位）。

FunctionCall (F)

Byte1('F')

メッセージが関数呼び出しであることを識別します。

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

Int32

呼び出す関数のオブジェクトIDを指定します。

Int16

後述する引数書式コード数です（以下では*c*と表します）。これは、引数が存在しない、あるいは、すべての引数がデフォルトの書式（テキスト）を使用することを示す0に、指定する書式コードをすべての引数に適用することを示す1にすることができます。さもなくば、これは実際の引数の数と同じになります。

Int16[*c*]

引数の書式コードです。それぞれは、0（テキスト）もしくは1（バイナリ）でなければなりません。

Int16

関数に提供する引数の数を指定します。

次に、各引数に対して以下のフィールドのペアが現れます。

Int32

引数の値のバイト単位の長さです（これには自身は含まれません）。0とすることもできます。特別な場合として、-1はNULLという引数の値を示します。NULLの場合、後続の値用のバイトはありません。

Byten

関連する書式コードで示される書式における引数の値。*n*は上述の長さです。

最後の引数の後に、以下のフィールドが現れます。

Int16

関数結果用の書式コードです。現在、0（テキスト）または1（バイナリ）でなければなりません。

FunctionCallResponse (B)

Byte1('V')

メッセージが関数呼び出しの結果であることを識別します。

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

Int32

関数の結果の値のバイト単位の長さです（これには自身は含まれません）。ゼロとすることもできます。特別な場合として、-1はNULLという関数の結果の値を示します。NULLの場合、後続の値用のバイトはありません。

Byten

関連する書式コードで示される書式における関数の結果の値。*n*は上述の長さです。

NoData (B)

Byte1('n')

メッセージがデータなしの指示子であることを識別します。

Int32(4)

自身を含む、メッセージ内容の長さ（バイト単位）。

NoticeResponse (B)

Byte1('N')

メッセージが警報であることを識別します。

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

このメッセージの本体には、ゼロバイトを終端として後続する、1つ以上の識別されるフィールドが含まれます。フィールドは任意の順番で現れる可能性があります。各フィールドには以下があります。

Byte1

フィールド種類を識別するコードです。ゼロならば、メッセージの終端であり、後続する文字列がないことを表します。[項46.6](#)に、現時点でフィールド種類として定義されているものを列挙します。将来もっと多くのフィールド種類が追加される可能性がありますので、フロントエンドは、認知できない種類のフィールドに対して何もせずに無視すべきです。

String

フィールドの値です。

NotificationResponse (B)

Byte1('A')

メッセージが通知応答であることを識別します。

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

Int32

通知元バックエンドのプロセスIDです。

String

通知の発生元となったチャンネル名です。

String

通知プロセスから渡される"ペイロード"文字列です。

ParameterDescription (B)

Byte1('t')

メッセージがパラメータ記述であることを識別します。

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

Int16

文で使用するパラメータ数です（ゼロとすることができます）。

そして、各パラメータに対して、以下が続きます。

Int32

パラメータのデータ型のオブジェクトIDを指定します。

ParameterStatus (B)

Byte1('S')

メッセージが実行時パラメータ状態報告であることを識別します。

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

String

報告される実行時パラメータの名前です。

String

そのパラメータの現在値です。

Parse (F)

Byte1('P')

メッセージがParseコマンドであることを識別します。

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

String

宛先のプリペアド文の名前です（空文字列で無名のプリペアド文を選択します）。

String

解析される問い合わせ文字列です。

Int16

指定されるパラメータデータ型の数です（ゼロとすることができます）。これは、問い合わせ文字列内にあるパラメータの数を示すものではないことに注意してください。 フロントエンドが型指定を希望するパラメータの数ではありません。

その後、各パラメータに対し、以下が続きます。

Int32

パラメータのデータ型のオブジェクトIDを指定します。ここにゼロを書くことは型指定を行わないことと同じです。

ParseComplete (B)

Byte1('1')

メッセージがParse完了指示子であることを識別します。

Int32(4)

自身を含む、メッセージ内容の長さ（バイト単位）。

PasswordMessage (F)

Byte1('p')

メッセージがパスワード応答であることを識別します。これがGSSAPIまたはSSPI応答メッセージでも使用されることに注意してください。（これは本当は設計エラーです。こうした場合含まれるデータはヌル終端の文字列ではなく、任意のバイナリデータを取り得るためです。）

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

String

パスワードです（必要ならば暗号化されています）。

PortalSuspended (B)

Byte1('s')

メッセージがポータル中断指示子であることを識別します。これは、Executeメッセージの行数制限に達した場合にのみ現れることに注意してください。

Int32(4)

自身を含む、メッセージ内容の長さ（バイト単位）。

Query (F)

Byte1('Q')

メッセージが簡易問い合わせであることを識別します。

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

String

問い合わせ文字列自体です。

ReadyForQuery (B)

Byte1('Z')

このメッセージ種類を識別します。バックエンドで新しい問い合わせサイクルの準備が整った時には常にReadyForQueryが送信されます。

Int32(5)

自身を含む、メッセージ内容の長さ（バイト単位）。

Byte1

現在のバックエンドのトランザクション状態指示子です。取り得る値は、待機状態（トランザクションブロックにない状態）に'I'、トランザクションブロック内の場合に'T'、失敗したトランザクションブロック（ブロックが終わるまで問い合わせは拒絶されます）内の場合に'E'です。

RowDescription (B)

Byte1('T')

メッセージが行の記述であることを識別します。

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

Int16

行内のフィールド数を指定します（ゼロとすることができます）。

その後、各フィールドに対して以下が続きます。

String

フィールド名です。

Int32

フィールドが特定のテーブルの列として識別できる場合、テーブルのオブジェクトIDです。さもなければゼロです。

Int16

フィールドが特定のテーブルの列として識別できる場合、列の属性番号です。さもなければゼロです。

Int32

フィールドのデータ型のオブジェクトIDです。

Int16

データ型の大きさ（`pg_type.typelen`を参照）です。負の値が可変長の型を表すことに注意してください。

Int32

型修飾子（`pg_attribute.atttypmod`を参照）です。修飾子の意味は型に固有です。

Int16

フィールドに使用される書式コードです。現在、0（テキスト）または1（バイナリ）のいずれかになります。RowDescriptionがステートメント用のDescribeから返された場合、書式コードはまだ不明ですので、常に0になります。

SSLRequest (F)

Int32(8)

自身を含む、メッセージ内容の長さ（バイト単位）。

Int32(80877103)

SSL要求コードです。この値は、最上位16ビットに¹²³⁴が、下位16ビットに⁵⁶⁷⁹を持つように選択されます（混乱を防ぐため、このコードはプロトコルバージョン番号と同一になってはいけません）。

StartupMessage (F)

Int32

自身を含む、メッセージ内容の長さ（バイト単位）。

Int32(196608)

プロトコルバージョン番号です。最上位16ビットは主バージョン番号（本章で説明したプロトコルでは3）です。下位16ビットは副バージョン番号（本章で説明したプロトコルでは0）です。

プロトコルバージョン番号の後に、1つ以上のパラメータ名と値文字列のペアが続きます。最後の名前と値のペアの後に終端としてゼロバイトが必要です。パラメータは任意の順番で現れることができます。userは必須です。他は省略可能

です。各パラメータは以下のように指定されます。

String

パラメータ名です。現在認識される名前を以下に示します。

user

接続するデータベースユーザ名です。必須。デフォルトはありません。

database

接続するデータベースです。デフォルトはユーザ名です。

options

バックエンド用のコマンドライン引数です（これは個々の実行時パラメータ設定のため、廃棄予定です）。

上記に加え、バックエンドの起動時に設定可能な、任意の実行時パラメータが列挙される可能性があります。こうした設定は、バックエンド起動時に（もしあればコマンドラインオプションの解析の後に）適用されます。この値はセッションのデフォルトとして動作します。

String

パラメータの値です。

Sync (F)

Byte1('S')

メッセージがSyncコマンドであることを識別します。

Int32(4)

自身を含む、メッセージ内容の長さ（バイト単位）。

Terminate (F)

Byte1('X')

メッセージが終了であることを識別します。

Int32(4)

自身を含む、メッセージ内容の長さ（バイト単位）。

[前のページ](#)

メッセージのデータ型

[ホーム](#)
[上に戻る](#)

[次のページ](#)

エラーおよび警報メッセージフィールド

46.6. エラーおよび警報メッセージフィールド

本節では、ErrorResponseおよびNoticeResponseメッセージ内で現れる可能性があるフィールドについて説明します。それぞれのフィールド種類は、単一バイトの識別子トークンを持ちます。メッセージ内に与えられる任意のフィールド種類は、多くてもメッセージ当たり1つでなければならないことに注意してください。

S

深刻度です。フィールドの内容はERROR、FATAL、PANIC（エラーメッセージ内）、WARNING、NOTICE、DEBUG、INFO、LOG（警報メッセージ内）、もしくはこれらの1つの地域化された翻訳です。常に存在します。

C

コード、そのエラー用のSQLSTATEコードです（[付録A](#)を参照）。地域化されません。常に存在します。

M

メッセージ、主に人にわかりやすいエラーメッセージです。これは正確、簡潔でなければなりません（通常は1行です）。常に存在します。

D

詳細です。問題のより詳細を説明する省略可能な二次的なエラーメッセージです。複数行にまたがる可能性があります。

H

ヒントです。その問題にどう対応するかを表す省略可能な提言です。これは、詳細と異なり、事実ではなく提案（不適切な場合もありますが）を提供することを目的としたものです。複数行にまたがる可能性があります。

P

位置です。フィールド値は、エラーカーソルの位置を示すもので、元の問い合わせ文字列へのインデックスを10進ASCIIで表した整数です。先頭の文字がインデックス1になり、位置はバイトではなく文字で数えられます。

P

内部的位置です。これはPと同じ定義ですが、カーソルの位置がクライアントによって発せられたコマンドではなく内部的に生成されたコマンドを参照する場合に使用されます。このフィールドが現れる時には常にqも現れます。

Q

内部的問い合わせ。失敗した、内部生成のコマンドテキストです。これは例えば、PL/pgSQL関数によって発行されたSQL問い合わせなどです。

W

場所です。エラーが発生したコンテキストを示します。現在ここには、実行中の手続き言語関数と内部生成問い合わせの呼び出しスタックトレースバックが含まれます。この追跡情報は、1行当たり1項目として、最も最近のものが初めに現れます。

F

ファイルです。エラーを報告した、ソースコードのファイル名です。

L

行です。エラーを報告した、ソースコードの行番号です。

R

ルーチンです。エラーを報告した、ソースコードのルーチン名です。

クライアントには、必要な情報を表示する際、整形する責任があります。具体的には、必要に応じて長い行を分割しなければなりません。エラーメッセージフィールド内にある改行文字は、改行ではなく、段落の区切りとして扱わなければなりません。

46.7. プロトコル2.0からの変更点の要約

本節では、既存のクライアントライブラリをプロトコル3.0に更新しようとする開発者向けに、変更点の簡易チェックリストを示します。

最初の開始パケットは、固定書式ではなく、柔軟な文字列のリスト書式を使用します。実行時パラメータのセッションのデフォルト値が直接開始パケット内に指定できるようになった点に注意してください。（実際、以前でも`options`フィールドを使用してこれを行うことができましたが、`options`には長さに制限があること、および値内の空白文字を引用符でくくる方法がないことから、あまり安全な技法ではありませんでした。）

すべてのメッセージが、メッセージ種類バイトの直後にバイト数を持つようになりました（種類バイトがない開始パケットは例外です）。また、`PasswordMessage`が種類バイトを持つようになったことにも注意してください。

`ErrorResponse`および`NoticeResponse`（'E'および'N'）メッセージが複数のフィールドを持つようになりました。これを使用して、クライアントコードは、必要な冗長度に合わせて、エラーメッセージを組み立てることができます。個々のフィールドが通常改行で終わらないことに注意してください。単一の文字列を送信する古いプロトコルでは、常に改行で終わっていました。

`ReadyForQuery`（'Z'）メッセージに、トランザクション状態指示子が含まれます。

`BinaryRow`と`DataRow`メッセージ種類間の区別がなくなりました。1つの`DataRow`メッセージ種類で、すべての書式で記述されたデータを返すことができます。`DataRow`のレイアウトが解析しやすいように変更されたことに注意してください。またバイナリ値の表現も変更されました。もはやサーバの内部表現に直接束縛されません。

新しい"拡張問い合わせ"サブプロトコルがあります。これにより、フロントエンドメッセージ種類に`Parse`、`Execute`、`Describe`、`Close`、`Flush`、および`Sync`が、バックエンドメッセージ種類に`ParseComplete`、`BindComplete`、`PortalSuspended`、`ParameterDescription`、`NoData`、および`CloseComplete`が追加されました。既存のクライアントは、このサブプロトコルを意識する必要はありませんが、これを使用することで、性能や機能を向上させることができます。

`COPY` データが`CopyData`と`CopyDone`メッセージにカプセル化されるようになりました。`COPY`中のエラーから復旧するための十分に定義された方法があります。特別な"`¥.`"という最後の行はもはや不要で、`COPY OUT`で送信されません。（`COPY IN`ではまだ終端として認識されます。しかし、この使用は廃止予定で、最終的には削除されます。）バイナリ`COPY`がサポートされます。`CopyInResponse`と`CopyOutResponse`メッセージは、列数と各列の書式を示すフィールドが含まれます。

`FunctionCall`と`FunctionCallResponse`メッセージのレイアウトが変更されました。`FunctionCall`は関数へのNULL引数を渡すことができるようになりました。また、テキストとバイナリ書式のどちらでもパラメータの引き渡しと結果の取り出しを扱うことができます。サーバの内部データ表現への直接アクセスを提供しなくなりましたので、`FunctionCall`を潜在的なセキュリティホールとみなす理由はもはやありません。

バックエンドは、接続開始時にクライアントライブラリが興味を持つとみなされるすべてのパラメータのために`ParameterStatus`（'S'）メッセージを送信します。その後、これらのパラメータのいずれかの実際の値が変更された時は常に、`ParameterStatus`メッセージが送信されます。

`RowDescription`（'T'）メッセージは、新規に、記述する各列に対してテーブルのOIDと列番号フィールドを伝えます。また各列の書式コードも示します。

`CursorResponse`（'P'）メッセージはもはやバックエンドで生成されません。

`NotificationResponse`（'A'）メッセージは、`NOTIFY`イベントの送信者から渡される"ペイロード"文字列を伝えることができる追加文字列フィールドを持ちます。

`EmptyQueryResponse`（'I'）メッセージは、空の文字列パラメータを含めるために使用されていました。これは削除されました。

第 47章 PostgreSQL コーディング規約

目次

- 47.1. [書式](#)
- 47.2. [サーバ内部からのエラーの報告](#)
- 47.3. [エラーメッセージのスタイルガイド](#)
 - 47.3.1. [何がどこで起こったか](#)
 - 47.3.2. [整形](#)
 - 47.3.3. [引用符](#)
 - 47.3.4. [引用符の使用](#)
 - 47.3.5. [文法と句読点](#)
 - 47.3.6. [大文字と小文字](#)
 - 47.3.7. [受動態の禁止](#)
 - 47.3.8. [現在形と過去形](#)
 - 47.3.9. [オブジェクトの種類](#)
 - 47.3.10. [角括弧](#)
 - 47.3.11. [エラーメッセージの組み立て](#)
 - 47.3.12. [エラーの理由](#)
 - 47.3.13. [関数名](#)
 - 47.3.14. [ややこしい単語の防止](#)
 - 47.3.15. [適切なスペル](#)
 - 47.3.16. [地域化](#)

47.1. 書式

ソースコードの書式では、タブを4カラムとするスペーシングを使用し、現在はタブを保存しています（つまりタブをスペースに展開しません）。各論理インデントのレベルは、さらに1つのタブストップです。

配置規則（括弧の位置など）はBSDの慣例に従います。特にif、while、switchなどの制御ブロックのための中括弧はそれ自身を一行で表します。

コードが80カラムのウィンドウで読み易くなるように1行の長さを制限してください。（これは80カラムを超えてはならないことを意味していません。例えば、任意の場所にある長いエラーメッセージ文字列をコードが80カラムに収まるように改行を含めても、おそらく可読性を向上させることはありません。）

C++形式のコメント（//コメント）は使用しないでください。厳密なANSI Cコンパイラはそれらを受け付けません。同じ理由で、中間ブロックで新規の変数を宣言するようなC++拡張を使用しないでください。

複数行に渡るコメントブロックの推奨書式は以下のようになります。

```
/*
 * コメントテキストはここから始まり
 * ここまで続きます
 */
```

列1で始まるコメントブロックはpgindentによるのと同じ通りに維持されますが、字下げされたコメントブロックを、あたかも平文テキストのように還流します。ある字下げブロックの中で改行を維持したい場合は以下のようにダッシュを追加します。

```
/*-----
 * コメントテキストはここから始まり
 * ここまで続きます
 *-----
 */
```

登録されたパッチは完全にはこの書式規則に従う必要はありませんが、そのようにすることを勧めます。登録されるコードは次のリリースの前にpgindentを通します。ですので、他の書式規則に従って作成して、見た目を良くすることに意味がありません。優れたパッチに関する原則は、"新しいコードがその前後にある既存のコードのように見える"ことです。

src/toolsディレクトリには、確実に上記の規約に従った書式になることを補助する、emacs、xemacs、vimエディタで利用できるサンプルの設定ファイルがあります。または、vi内で下記のようにコマンドを実行してください。

```
:set ts=4
```

テキスト閲覧ツールmoreとlessでは以下のようにすればタブを適切に表示させることができます。

```
more -x4
less -x4
```

47.2. サーバ内部からのエラーの報告

サーバコード内から生成されるエラー、警告、ログメッセージは、`ereport`もしくはこれに似た古い`elog`を使用して作成してください。この関数の使用はいくつか説明が必要なほど複雑です。

すべてのメッセージには、深刻度レベル（`DEBUG`から`PANIC`までの範囲）と主メッセージテキストという、2つの必須要素があります。さらに、省略可能な要素があります。その中で最もよく使用されるのは、SQL仕様のSQLSTATE規則に従うエラー識別コードです。`ereport`自身はシェル関数で、主に、メッセージ生成をCソースコード内の関数呼び出しのように行わせる、構文の便宜上存在します。`ereport`で直接受け付けられる唯一のパラメータは深刻度レベルです。主メッセージテキストと任意の省略可能なメッセージ要素は、`ereport`呼び出し内で`errmsg`などの補助関数を呼び出すことで生成されます。

典型的な`ereport`の呼び出しは以下のようなものです。

```
ereport(ERROR,
        (errmsg(ERRCODE_DIVISION_BY_ZERO),
         errmsg("division by zero")));
```

これはエラー深刻度レベル`ERROR`（普通のエラー）を指定します。`errmsg`呼び出しは、`src/include/utils/errcodes.h`で定義されたマクロを使用してSQLSTATEエラーコードを指定します。`errmsg`呼び出しは主メッセージテキストを提供します。補助関数呼び出しを囲む余計な括弧群に注意してください。これらはいらいらさせられますが、構文上必要です。

以下に、より複雑な例を示します。

```
ereport(ERROR,
        (errmsg(ERRCODE_AMBIGUOUS_FUNCTION),
         errmsg("function %s is not unique",
                func_signature_string(funcname, nargs,
                                     NIL, actual_arg_types)),
         errhint("Unable to choose a best candidate function. "
                 "You might need to add explicit typecasts.)));
```

これは、実行時の値をメッセージテキスト内に埋め込むための整形用コードの使用を示します。また、省略可能な"ヒント"メッセージも提供されています。

`ereport`で使用可能な補助ルーチンを以下に示します。

- `errmsg(sqlerrcode)` は、その条件用のSQLSTATEエラー識別コードを指定します。このルーチンが呼び出されないと、エラー識別子のデフォルトは、エラー深刻度レベルが`ERROR`以上の場合には`ERRCODE_INTERNAL_ERROR`に、エラー深刻度レベルが`WARNING`の場合には`ERRCODE_WARNING`に、さもなければ（`NOTICE`以下）`ERRCODE_SUCCESSFUL_COMPLETION`になります。これらのデフォルトはしばしば便利ですが、`errmsg()` 呼び出しを省略する前に、常に適切かどうかを検討してください。
- `errmsg(const char *msg, ...)` は、主エラーメッセージテキストを指定し、また、実行時の値をそこに挿入することもできます。挿入は、`sprintf`様式の書式コードで指定されます。`sprintf`で受け付けられる標準の書式コードに加え、`%m`書式コードを使用して、現在の`errno`の値用の`strerror`で返されるエラーメッセージを挿入することができます。[\[1\]](#) `%m`は`errmsg`のパラメータリスト内に対応する項目を必要としません。メッセージ文字列は、書式コードの処理を行う前に、地域化のために`gettext`を通ることに注意してください。
- `errmsg_internal(const char *msg, ...)` は、そのメッセージ文字列は変換されずに国際化用メッセージ辞書が含まれない点を除き、`errmsg`と同一です。これは、おそらく翻訳作業を行う価値がない"発生できなかった"場合用に使用すべきです。
- `errmsg_plural(const char *fmt_singular, const char *fmt_plural, unsigned long n, ...)` は`errmsg`のようですが、メッセージの各種の複数書式があります。`fmt_singular`は英語の単数書式、`fmt_plural`は英語の複数書式、`n`はどの複数書式が必要なのかを決定する整数値で、残りの引数は選択された書式文字列に従って書式化されます。より詳細な情報は[項48.2.2](#)を参照してください。
- `errdetail(const char *msg, ...)` は省略可能な"詳細"メッセージを提供します。これは、主メッセージ内に記述するには不適切と考えられる追加情報がある場合に使用されます。このメッセージ文字列は`errmsg`とまったく同じ方法で処理されます。
- `errdetail_internal(const char *msg, ...)` は、メッセージが翻訳されない、または、国際化メッセージ辞書内に含まれない点を除き、`errdetail`と同じです。これは、例えばほとんどのユーザに役に立つにはあまりにも技術的すぎるなど、翻訳する手間をかける価値がない詳細メッセージで有用であるはずです。
- `errdetail_plural(const char *fmt_singular, const char *fmt_plural, unsigned long n, ...)` は`errdetail`と似て

いますが、メッセージの各種の複数書式をサポートします。 より詳細な情報は[項48.2.2](#)を参照してください。

- この文字列がサーバログのみに渡り、クライアントに渡らない点を除き `errdetail_log(const char *msg, ...)` は `errdetail` と同一です。 `errdetail` (や上記の等価物の1つ) と `errdetail_log` が共に使用された場合、1つの文字列はクライアントに渡り、もう1つはログに渡ります。 クライアントに送られるレポートに含めるにはセキュリティに対して慎重を期さなければならないもの、あるいは膨大すぎるエラー詳細に対して効果があります。
- `errhint(const char *msg, ...)` は、省略可能な"ヒント"メッセージを提供します。 これは、何が悪かったのかについての事実に基づく詳細とは反対で、問題を解決させる方法に関する提言を提供するために使用されます。 このメッセージ文字列は `errmsg` とまったく同じ方法で処理されます。
- `errcontext(const char *msg, ...)` は通常 `ereport` メッセージ側から直接呼び出されません。 エラーが発生したコンテキスト、例えばPL関数の現在位置の情報を提供するために `error_context_stack` コールバック関数内で使用されます。 メッセージ文字列は `errmsg` とまったく同じ方法で処理されます。 他の補助関数とは異なり、1つの `ereport` 呼び出しで何度も呼び出すことができます。 こうして提供される文字列の並びは、改行で区切った形で連結されます。
- `errposition(int cursorpos)` は、問い合わせ文字列内でエラーが発生した位置をテキストで指定します。 現在、問い合わせ処理の字句解析および構文解析段階でエラーが検出された場合にのみ役に立ちます。
- `errcode_for_file_access()` は、ファイルアクセス関連のシステムコールでの失敗用のSQLSTATEエラー識別子を適切に選択する、便利な関数です。 保存された `errno` を使用して、どのエラーコードを生成するかを決定します。 通常、これは主エラーメッセージテキスト内で `%m` と組み合わせて使用されなければなりません。
- `errcode_for_socket_access()` は、ソケット関連のシステムコールでの失敗用のSQLSTATEエラー識別子を適切に選択する、便利な関数です。
- `errhidestmt(bool hide_stmt)` は、postmasterのログ内のメッセージにおけるSTATEMENT:部分を抑制するために呼び出すことができます。 通常これは、メッセージテキスト内にすでに現在の文が含まれている場合に適しています。

まだ頻繁に使用されている、古めの `elog` 関数があります。 以下の `elog` 呼び出しは、

```
elog(level, "format string", ...);
```

以下とまったく同じです。

```
ereport(level, (errmsg_internal("format string", ...)));
```

SQLSTATEエラーコードが常にデフォルトになること、メッセージ文字列が変換されないことに注意してください。 したがって、`elog` は、内部エラーと低レベルなデバッグ用ログにのみ使用すべきです。 一般ユーザを対象とする任意のメッセージは `ereport` を使用すべきです。 それでもなお、システム内の"発生できなかった"内部エラーの検査に `elog` がまだ多く使用されています。 これは、こうした単純な表記のメッセージに適しています。

[項47.3](#)に推奨するエラーメッセージの作成についての提言を示します。

注意

[1]

つまり、`ereport` 呼び出しに達した時点の値です。 補助報告ルーチン内で `errno` を変更しても効果はありません。 `errmsg` 内で `strerror(errno)` を明示的に記述したとしても正確なものにはなりません。 したがって、このようにはしないでください。

[前のページ](#)
書式

[ホーム](#)
[上に戻る](#)

[次のページ](#)
エラーメッセージのスタイルガイド

47.3. エラーメッセージのスタイルガイド

このスタイルガイドでは、PostgreSQLで生成されるすべてのメッセージに対する、一貫性維持、ユーザに親切なスタイルについての希望を説明します。

47.3.1. 何がどこで起こったか

主メッセージは、簡潔に事実を示すものにすべきです。特定の関数名など実装の詳細への参照は止めるべきです。"簡潔"は"ごく普通の条件下で1行に収まる"ことを意味します。主メッセージを簡潔にするために必要であれば、また、特定のシステムコールが失敗したなど実装の詳細について記載したいのであれば、詳細メッセージを使用してください。主メッセージ、詳細メッセージの両方は事実を示すものにすべきです。どうすれば問題を解決できるかに関する提言には、その提言が常に適切とは限らない場合は特に、ヒントメッセージを使用してください。

例えば、

```
IpcMemoryCreate: shmget(key=%d, size=%u, 0%o) failed: %m
(plus a long addendum that is basically a hint)
```

ではなく、以下のように記述してください。

```
Primary:      could not create shared memory segment: %m
Detail:       Failed syscall was shmget(key=%d, size=%u, 0%o).
Hint:         the addendum
```

理論的根拠：主メッセージを簡潔にすることは、要点を維持することを助けます。そして、クライアントは、エラーメッセージ用に1行分確保すれば十分であるという仮定の下で画面設計を行うことができます。詳細メッセージやヒントメッセージを冗長モードに格下げしたり、エラーの詳細を表示するウィンドウをポップアップさせることもできます。また、詳細メッセージやヒントメッセージは通常ディスク容量を節約するためにサーバログには出力されません。ユーザはどうやってもその詳細を知りませんので、実装の詳細への参照を避けることが最善です。

47.3.2. 整形

メッセージテキストの整形に関して、特定の前提を行わないでください。クライアントやサーバログでは、自身の必要性に合わせて行を改行すると想定してください。長めのメッセージでは、改行文字 (¥n) を推奨する段落の区切りを示すものとして使用することができます。メッセージの終わりに改行を付けないでください。タブや他の整形用文字を使用しないでください。（エラーの内容の表示では、関数呼び出しなどのコンテキストのレベルを区切るために、改行が自動的に追加されます。）

理論的根拠：メッセージは必ずしも端末型のディスプレイに表示されるとは限りません。GUIのディスプレイやブラウザでは、こうした書式指示はうまくいったとしても無視されます。

47.3.3. 引用符

英文では、引用が適切な場合には二重引用符を使用すべきです。他の言語でのテキストは、出版上の慣習や他のプログラムのコンピュータ出力と矛盾しない種類の引用符の1つを一貫して使用してください。

理論的根拠：二重引用符と単一引用符の選択はどちらかでもよいものですが、推奨する方がよく使われています。SQL規約（すなわち文字列には単一引用符、識別子には二重引用符）に従って、オブジェクトの種類に応じて引用符を選択することを推奨する人もいます。しかし、これは言語内部の技術的な事項であり、多くのユーザが理解できるものではありません。また、他の種類の引用手法には拡張できません。他の言語へ翻訳できません。ですので、あまり意味がありません。

47.3.4. 引用符の使用

ファイル名、ユーザ提供の識別子、複数の単語を含む変数を区切るために引用符を使用してください。複数の単語を含まない変数（例えば演算子名）には引用符を使用しないでください。

バックエンドには必要に応じて出力に二重引用符を付与する関数(例えば `format_type_be()`)があります。こうした関数の出力の前後にさらに引用符を追加しないでください。

理論的根拠：オブジェクトの名前をメッセージ内に埋め込む際に曖昧さが生じることがあります。埋め込む名前がどこから始まりどこで終わるかについての表記には一貫性を持たせてください。しかし、不必要にメッセージをまとめたり、引用符を二重にすることは止めてください。

47.3.5. 文法と句読点

この規則は、主エラーメッセージと詳細/ヒントメッセージとで異なります。

主エラーメッセージ：最初の文字を大文字にしないでください。メッセージの最後にピリオドを付けしないでください。メッセージの終わりに感嘆符を付けようとは考えないでください。

詳細メッセージとヒントメッセージ：完全な文章を使用し、終わりにピリオドを付けてください。文章の最初の単語は大文字にしてください。他の文が続く場合はピリオドの後に空白を2つ入れてください（英文の場合です。他の言語では不適切かもしれません）。

エラー文脈文字列：先頭文字を大文字にせず、また、終わりににはピリオドを付けしないでください。文脈文字列は通常完全な文章にすべきではありません。

理論的根拠：句読点の禁止により、クライアントアプリケーションでは、そのメッセージを各種文法的なコンテキストに埋め込みやすくなります。主メッセージはしばしば文法的に完全な文章になっていません。（そして、1行以上の長さになりそうであれば、主メッセージと詳細メッセージに分割すべきです。）しかし、詳細メッセージとヒントメッセージは、より長く、かつ複数の文章を持つ必要があるかもしれません。一貫性のため、これらは、たとえ文章が1つだけであっても、完全な文章形式に従うべきです。

47.3.6. 大文字と小文字

メッセージの言葉使いでは小文字を使用してください。主エラーメッセージの場合は先頭文字も含みます。SQLコマンドとキーワードがメッセージ内に出現する場合は大文字を使用してください。

理論的根拠：メッセージは完全な文章かもしれませんが、そうではないかもしれませんので、この方法は、より簡単にすべての見た目の一貫性を向上します。

47.3.7. 受動態の禁止

能動態を使用してください。能動的な主語がある（"AはBを行うことができない"）場合は完全な文章を使用してください。主語がプログラム自体である場合は、主語を付けずに電報様式を使用してください。プログラムに"I（私）"を使用しないでください。

理論的根拠：プログラムは人間ではありません。他を真似ないでください。

47.3.8. 現在形と過去形

試行が失敗したが、次は（何らかの問題を修正した後に）成功するかもしれない場合は過去形を使用してください。失敗が永続するようであれば、現在形を使用してください。

以下の2つの意味には無視できないほどの違いがあります。

```
could not open file "%s": %m
```

および

```
cannot open file "%s"
```

最初のものは、ファイルを開くことに失敗したことを意味します。メッセージには、"ディスクが一杯"や"ファイルが存在しない"といった、その理由を付けるべきです。次回はディスクに空きがあるかもしれませんが、問題のファイルが存在するかもしれませんので過去形が適切です。

2番目の形式は、そのプログラム内の指名されたファイルを開く機能が存在しない、あるいは、概念的に不可能であることを示します。この条件は永遠に続きますので現在形が適切です。

理論的根拠：仮定ですが、一般的なユーザは単なるメッセージの時制から多くの意味を引き出すことはできないでしょう。しかし、言語が文法を提供してくれますので、それを正確に使用すべきでしょう。

47.3.9. オブジェクトの種類

オブジェクトの名前を引用する時、そのオブジェクトの種類を記載してください。

理論的根拠：さもないと、"foo.bar.baz"が何なのか誰もわかりません。

47.3.10. 角括弧

角括弧は、（1）コマンドの概要にて省略可能な引数を表す、（2）配列の添字を表す、ためだけに使用されます。

理論的根拠： 広く知られる慣習に対応するものがなく、人々を混乱させることになります。

47.3.11. エラーメッセージの組み立て

メッセージに、他で生成されるテキストを含める場合、以下の様式で埋め込んでください。

```
could not open file %s: %m
```

理論的根拠： すべての起こり得るエラーコードを単一のなめらかな文章に埋め込むことを考えることは困難です。 ですので、何らかの句読点が必要とされます。 括弧の中にテキストを埋め込むこともまた推奨されていましたが、よくある埋め込むテキストがそのメッセージの最も重要となる場合は不自然です。

47.3.12. エラーの理由

メッセージは常にエラーが発生した理由を記述すべきです。 以下に例を示します。

```
BAD:      could not open file %s
BETTER:  could not open file %s (I/O failure)
```

理由がわからない場合はコードを直すべきです。

47.3.13. 関数名

エラーテキストには、それを報告したルーチンの名前を含めないでください。 必要に応じて、そのルーチンを見つける他の機構がありますが、ほとんどのユーザにとって役に立つ情報ではありません。 関数名がないと、エラーメッセージにあまり意味がないのであれば、言葉使いを変えてください。

```
BAD:      pg_atoi: error in "z": cannot parse "z"
BETTER:  invalid input syntax for integer: "z"
```

同時に呼び出した関数名の記述も止めてください。 代わりにそのコードが何をしようとしたのかを記述してください。

```
BAD:      open() failed: %m
BETTER:  could not open file %s: %m
```

もし本当に必要であれば、詳細メッセージにそのシステムコールを記載してください（詳細メッセージの情報としてシステムコールに実際に渡した値を与えることが適切な場合もあります）。

理論的根拠： ユーザはその関数が何を行うのかを知りません。

47.3.14. ややこしい単語の防止

Unable. "Unable"はほとんど受動態です。 "cannot"または"could not"の適切な方を使用してください。

Bad. "bad result"といったエラーメッセージは、知的に解釈することが非常に困難です。 結果が"bad"である理由、例えば"invalid format"を記述してください。

Illegal. "Illegal"は規則違反を表します。 他は"invalid"です。 より良くするために、なぜ無効なのかについても記述してください。

Unknown. "unknown"は極力使用しないでください。 "error: unknown response"について考えてみましょう。 どんな応答であるかわからなければ、どうやって何がエラーなのかわかるでしょうか。 "Unrecognized"を選んだ方が良い場合がしばしばあります。 また、その警告の中に値が含まれていることを確認してください。

```
BAD:      unknown node type
BETTER:  unrecognized node type: 42
```

Find vs. Exists. プログラムがリソースの場所について無視できないアルゴリズム（例えばパスの検索）を使用し、そのアルゴリズムが失敗した場合、プログラムがリソースを"find"できなかったと記述すべきでしょう。 一方、想定したリソースの場所 はわかっているが、プログラムがアクセスできなかった場合は、リソースが"exist"しなかったと記述してください。 この場合に"find"を使用すると、弱く取られ、問題が混乱します。

May, Can, Might. "May"は許可を示し、文書やエラーメッセージではあまり使われません（たとえば、熊手を借りられま

す)。 "Can"は能力を示し（たとえば、丸太を持ち上げることができます）、"might"は可能性を示します（たとえば、雨が降るかもしれません）。 意味を明確にし、翻訳を補助するために適切に使用してください。

短縮. "can't"などの短縮は避けてください。 代わりに"cannot"を使用してください。

47.3.15. 適切なスペル

単語の完全なスペルを使用してください。 例えば、以下は止めてください。

- spec
- stats
- parens
- auth
- xact

理論的根拠： これは一貫性を向上します。

47.3.16. 地域化

エラーメッセージは他の言語に翻訳される必要があることを忘れないでください。 [項48.2.2](#)のガイドラインに従い、翻訳者に苦労を強いることを防いでください。

[前のページ](#)

サーバ内部からのエラーの報告

[ホーム](#)
[上に戻る](#)

[次のページ](#)
多言語サポート

第 48 章 多言語サポート

目次

48.1. [翻訳者へ](#)

48.1.1. [必要事項](#)

48.1.2. [概念](#)

48.1.3. [メッセージカタログの作成と保守](#)

48.1.4. [PO ファイルの編集](#)

48.2. [プログラマへ](#)

48.2.1. [仕組み](#)

48.2.2. [メッセージ記述の指針](#)

48.1. 翻訳者へ

PostgreSQL プログラムは（サーバ、クライアントともに）メッセージが翻訳されていれば、使い慣れた言語でそのメッセージを出すことができます。翻訳メッセージセットの作成と保守は、自分の言語を上手に話し PostgreSQL の成果に貢献したい人々の協力が必要です。この作業にはプログラムである必要はまったくありません。本節ではお手伝いの方法を説明します。

48.1.1. 必要事項

協力者の言語の熟練度については判断しません。本節ではソフトウェアツールについて説明します。理論的には、テキストエディタのみが必要です。しかし、これは自分で作成した翻訳メッセージを試そうとはしないという、あまりあり得ない場合のみです。ソースツリーを構築する際に、`--enable-nls` オプションを使用していることを確認してください。これにより、全てのエンドユーザがとにかく必要とする、`libintl` ライブラリと `msgfmt` プログラムも同時に検査されます。作業を試す際には、インストール手順の適切な部分に従ってください。

新規に翻訳作業を始めるか、（後述の）メッセージカタログのマージを行いたい場合、GNU 版と互換性を持った実装の `xgettext` と `msgmerge` という 2 つのプログラムがそれぞれ必要です。将来はパッケージ化されたソース配布物を使用する場合に `xgettext` を必要としないように変更する予定です（Git 版で作業をしているのであれば、まだこれが必要です）。現在 GNU Gettext 0.10.36 以上を推奨します。

使用するマシンの `gettext` の実装については、文書と一緒に付いてくると思います。以下のいくつかはおそらく重複していますが、追補すべき詳細についてはその文書を参照してください。

48.1.2. 概念

英語による元のメッセージとそれを基に（場合によって）翻訳されたメッセージの組み合わせはメッセージカタログに、（関連するプログラムはメッセージカタログを共有することができますが）各プログラム、対象とする言語に対して一つずつ保持されます。メッセージカタログには 2 つのファイル形式があります。1 つは "PO" ファイル（移植可能オブジェクト-Portable Object-を意味します）で翻訳者が編集する特別な構文を持った平文ファイルです。2 番目は "MO" ファイル（マシンオブジェクト-Machine Object-を意味します）で対応する PO ファイルから生成され、国際化プログラムの実行の際に使用されるバイナリファイルです。翻訳者は、MO ファイルを扱いません。実際に扱うことは困難です。

メッセージカタログファイルの拡張子は想像していたかもしれませんが、`.po` もしくは `.mo` です。基本名（拡張子を除いた部分）は、プログラムが伴っている名前、もしくは翻訳目的とする言語の名前のいずれかで、状況によって変わります。少し混乱するかもしれませんが。例えば、`psql.po`（`psql` 用の PO ファイル）、もしくは `fr.mo`（フランス語用の MO ファイル）です。

PO ファイルの書式を以下に示します。

```
# comment

msgid "original string"
msgstr "translated string"

msgid "more original"
msgstr "another translated"
"string can be broken up like this"

...
```

`msgid` はプログラムのソースから抽出されます。（これは必要はありませんが最も一般的な方法です。）`msgstr` 行は初期状態では空であり、翻訳者によって有益な文字列が埋め込まれます。この文字列には、C 言語形式のエスケープ文字を含めることも、上に示したように複数行にまたがって続けることもできます。（継続行は必ず行の先頭から始まらなければなりません。）

`#` 文字はコメントの開始を示します。`#` 文字の直後に空白文字があった場合、それは翻訳者によって保守されるコメントです。`#` の直後に非空白文字が付く、自動的に付与されるコメントもあります。これらは、PO ファイルに対する操作を行う各種ツールによって保守され、翻訳者の補助を意図しています。

```
#. automatic comment
#: filename.c:1023
#, flags, flags
```

`#` スタイルのコメントはそのメッセージが使用されているソースファイルから抽出されます。おそらくプログラマが、翻訳者のために追加した、そのようにしてほしいと考える体裁についてなどの情報です。`#:` コメントは、ソース内でそのメッセージが使用される正確な場所を示します。翻訳者はプログラムソースを参照する必要はありませんが、翻訳の正確さに疑問がある場合にソースを参照することができます。`#,` コメントは何らかの方法でメッセージを説明するフラグです。現在、2 つのフラグがあります。そのメッセージがプログラムソースの変更によって古いものとなった可能性がある場合、`fuzzy` が設定されます。翻訳者

はこれを検証し、fuzzyフラグを削除できます。fuzzyメッセージはエンドユーザからは利用できないことに注意してください。もう1つのフラグはc-formatで、そのメッセージがprintf形式の書式テンプレートであることを示します。これは、翻訳側もプレースホルダの型と同じ番号を持った書式文字列でなければならないことを意味します。これを検証するツールがあり、それらはc-formatフラグを入力として受け取ります。

48.1.3. メッセージカタログの作成と保守

さて、"空の"メッセージカタログをどうやって作成するのでしょうか。まず、翻訳したいメッセージを持つプログラムが存在するディレクトリに移動します。nls.mkファイルがあればこのプログラムは翻訳の準備が整っています。

もし、.poファイルが数個既にあれば、誰かがある翻訳作業を行っています。そのファイルはlanguage.poと名前が付けられています。ここで、languageは[ISO 639-1の2文字言語コード \(小文字\)](#)を表します。例えば、fr.poはフランス語用です。1つの言語に複数の翻訳成果が必要である場合そのファイルの名前はlanguage_region.poのようになります。ここで、regionは[ISO 3166-1の2文字国コード \(大文字\)](#)を表します。例えば、pt_BR.poはブラジルでのポルトガル語用を示します。翻訳対象とする言語用のファイルがあれば、それを基に作業を始めることができます。

新規に翻訳を始める必要がある場合、以下のコマンドを最初に実行してください。

```
gmake init-po
```

これは、prognome.potファイルを作成します。（.potは"実用の"POファイルと区別するためのものです。tは"テンプレート"を意味します。）このファイルをlanguage.poにコピーして編集します。新規の言語が利用可能になったことを知らせるために、nls.mkを編集し、言語（もしくは言語と国）コードを以下のように追加してください。

```
AVAIL_LANGUAGES := de fr
```

（もちろん他の言語があるかもしれません。）

対象のプログラムやライブラリの変更に伴い、メッセージはプログラマによって変更、追加されます。この場合は始めからやり直す必要はありません。その代わりに以下のコマンドを実行してください。

```
gmake update-po
```

そうすると新しい空のメッセージカタログファイル（最初に使用したpotファイル）が作成され、既存のPOファイルにマージされます。このマージのアルゴリズムが特定のメッセージに関して確実でない場合、それは上で説明した"fuzzy"となります。新規POファイルは.po.newという拡張子付きで保存されます。

48.1.4. POファイルの編集

POファイルは普通のテキストエディタで編集することができます。翻訳者はmsgstrディレクティブの後の引用符の間の部分の変更、コメントの追加、fuzzyフラグの変更のみを行えばよいのです。Emacsには（予想通り）POモードがあり、非常に使いやすいものです。

POファイルを完全に埋めることは必要ありません。ソフトウェアは使用できる翻訳がない（もしくは翻訳が空の）場合自動的に元の文字列に戻します。他の人が作業を引き継ぐことができますので、ソースツリー内に不完全な翻訳を含めることにも問題はありません。しかし、マージの後のfuzzyフラグを削除することを優先に考えることが推奨されています。fuzzyエントリはインストールされないことを忘れないでください。これらは何が正しい翻訳になり得るかの参照のためにのみ提供されています。

以下に、翻訳の編集を行う際に注意すべき点を示します。

- 元の文字列の終端が改行の場合、翻訳も同様になっていることを確認してください。タブなども同様です。
- 元がprintf書式文字列の場合、翻訳も同じでなければなりません。また、翻訳は同じ書式識別子を同じ順番で持たなければなりません。言語固有の規則によっては不可能な場合や扱いづらい場合も起こります。このような場合は、以下の書式指定子を使用することができます。

```
msgstr "Die Datei %2$s hat %1$su Zeichen."
```

そして、リストの最初のプレースホルダが実際にはこのリストの2番目の引数に使用されます。digits\$は%の直後に続く必要があり、他の書式の操作の前に使用する必要があります。（この機能はprintf系の関数に本当に存在するものです。メッセージ国際化以外ではほとんど使用されませんので、聞いたことがないかもしれません。）

- 元の文字列に言語上の間違いがある場合、それを報告し（もしくはプログラムソースを直して）、普通に翻訳してください。修正された文字列は、プログラムのソースが修正された時にマージ可能になります。元の文字列が事実と異なる場合、それを報告し（もしくは自ら直して）、翻訳を行わないでください。その代わりに、POファイルのその文字列にコメントを付けてください。

- 元の文字列のスタイルや調子を維持してください。特に、(cannot open file %sなど) 文章になっていないメッセージは、(翻訳する言語が大文字小文字を区別するのであれば) おそらく最初の文字を大文字にしてはなりませんし、(翻訳する言語が句読点として使用しているのであれば) ピリオドを終わりに付けてはいけません。 [項47.3](#)を読むと参考になります。
- メッセージの意味がわからない時や、曖昧な場合は、開発者用のメーリングリストに問い合わせてください。英語を話すエンドユーザも理解できない、または曖昧であると判断することができる機会となり、メッセージの改良を行う最善のものです。

[前のページ](#)
多言語サポート

[ホーム](#)
[上に戻る](#)

[次のページ](#)
プログラマへ

48.2. プログラマへ

48.2.1. 仕組み

本節では、PostgreSQL 配布物の一部であるプログラムやライブラリにおける各国語サポートの実装方法を説明します。現在は C プログラムにのみ適用できます。

プログラムに **NLS** サポートを追加する

- 1 11 プログラムの起動処理に以下のコードを追加してください。

```
#ifdef ENABLE_NLS
#include <locale.h>
#endif

...

#ifdef ENABLE_NLS
setlocale(LC_ALL, "");
bindtextdomain("progrname", LOCALEDIR);
textdomain("progrname");
#endif
```

(*progrname* は実際には自由に選択できます。)

- 1 11 どこであろうと翻訳の候補となるメッセージが見つかったら、`gettext()` の呼び出しが追加される必要があります。例えば、

```
fprintf(stderr, "panic level %d\n", lvl);
```

は、次のように変更されます。

```
fprintf(stderr, gettext("panic level %d\n"), lvl);
```

(NLS サポートが組み込まれていない場合、`gettext` はノーオペレーション命令として定義されます。)

これは混乱しがちになります。一般的なショートカットは以下のものです。

```
#define _(x) gettext(x)
```

他の解決方法は、バックエンドにおける `ereport()` のように、そのプログラムが通信のほとんどを 1 つまたは数個の関数で行っている場合有効です。その場合、この関数の内部で全ての入力文字列に対し `gettext` を呼び出すようにすることになります。

- 1 11 プログラムのソースのあるディレクトリに `nls.mk` を追加してください。これは `makefile` として読まれます。以下の変数への代入をここで設定する必要があります。

CATALOG_NAME

`textdomain()` の呼び出しに使用されるプログラム名です。

AVAIL_LANGUAGES

用意された翻訳のリストです。初めは空です。

GETTEXT_FILES

翻訳可能文字列を含むファイルの一覧です。つまり、これらは `gettext` もしくは他の解決法として印が付けられます。結局、これはプログラムのほとんど全てのソースファイルを含むことになります。この一覧があまりに長くなる場合、最初の "file" を + とし、2 番目の単語を 1 行に対して 1 つのファイル名を持ったファイルとすることができます。

GETTEXT_TRIGGERS

翻訳者が作業を行う上で、どの関数呼び出しが翻訳可能文字列を含むかを知る必要に迫られた場合に、メッセージカタログを生成するツールです。デフォルトでは、`gettext()` 呼び出しのみを認識します。`_` や他の識別子を使用した場合、ここに記載しなければなりません。翻訳可能文字列がその最初の引数ではない場合、その項目は（例えば 2 番目の引数の場合）`func:2` という形式でなければなりません。複数形メッセージをサポートする関数がある場合、その項目は（単一形および複数形メッセージ引数を特定する）`func:1,2` のようになります。

ビルドシステムは、自動的にメッセージカタログの構築およびインストールを行います。

48.2.2. メッセージ記述の指針

メッセージの翻訳を簡単にするために以下に指針をいくつか示します。

- 以下のように実行時に文章を構築することはしないでください。

```
printf("Files were %s.%n", flag ? "copied" : "removed");
```

文章内の単語の順番は言語によって異なる可能性があります。さらに全ての断章に対して `gettext()` を呼び出すことを覚えていたとしても、断章が個別に的確に翻訳されるわけではありません。それぞれのメッセージが全て調和して翻訳されるかどうか、ちょっとしたコードの複製を用意するとよいかもしれません。番号、ファイル名、および実行時変数のみメッセージテキストに実行時に挿入するべきです。

- 同様の理由で、以下も上手くいきません。

```
printf("copied %d file%s", n, n!=1 ? "s" : "");
```

これは複数形がどのように形成されるかを決めてかかっているからです。もし、以下のようにして回避できたと考えると、

```
if (n==1)
    printf("copied 1 file");
else
    printf("copied %d files", n);
```

失望することになります。言語の中には、独特の規則によって2つ以上の形式になるものもあります。問題全体を回避するためこのメッセージを設計することが最善です。たとえば以下のようにします。

```
printf("number of copied files: %d", n);
```

適切に複数形を持つメッセージを構築したいと本当に思うのなら、これに対するサポートがありますが、多少厄介です。`ereport()` 内の主たる、または詳細なエラーメッセージを生成する場合、以下のように書くことができます。

```
errmsg_plural("copied %d file",
              "copied %d files",
              n,
              n)
```

最初の引数は英文の単数形に適切な書式文字列で、二番目は英文の複数形に適切な書式文字列、そして三番目はどの複数形を使用するのかを決定する整数制御値です。引き続き引数はいつものように書式文字列毎に書式化されます。（通常、複数化制御値は書式化されるべき値の内の1つです。）英語では n が1であるか、そうでないかのみ重要ですが、他の言語では数多くの複数書式が存在します。翻訳者にはグループとして2つの英文書式を参照し、 n の実行時の値に基づいて選択される適切な1つでもって、複数の代替文字列を供給する機会があります。

`errmsg`あるいは`errdetail`報告に直接行かない複数形メッセージが必要であれば、現行の関数、`ngettext`を使用する必要があります。`gettext`のドキュメントを参照してください。

- メッセージをどのように他の出力と合わせる予定なのかなど翻訳者と何か連絡を取り合いたい場合、`translator`で始まるコメントを最初に付けてどうなるかを知らせてください。以下のようにします。

```
/* translator: This message is not what it seems to be. */
```

これらのコメントはメッセージカタログにコピーされますので翻訳者は参照できます。

第 49 章手続き言語ハンドラの作成

現在のコンパイル言語用"Version-1"インタフェース以外のある言語で作成された関数の呼び出しはすべて、特定の言語用の呼び出しハンドラを経由します（これには、ユーザ定義手続き言語で作成された関数、SQLで作成された関数、コンパイル言語用Version-0インタフェースを使用した関数が含まれます）。提供されたソーステキストを解釈するなどによって、関数の実行を意味のある方法で行うことは、呼び出しハンドラの責任です。本章では、どのように新しい手続き言語の呼び出しハンドラを作成できるかについて概要を示します。

手続き言語用の呼び出しハンドラは"通常"の関数で、Cなどのコンパイル言語で作成し、Version-1インタフェースを使用し、引数を取らずに`language_handler`を返すものとしてPostgreSQLに登録しなければなりません。この特殊な仮想型は、その関数を呼び出しハンドラとして識別し、SQLコマンド内で直接その関数が呼び出されることを防止します。C言語の呼び出し規約と動的ロードについては[項35.9](#)を参照してください。

呼び出しハンドラは、他の関数と同じ方法で呼び出されます。引数値と呼び出された関数についての情報を含む`FunctionCallInfoData struct`のポインタを受け取り、`Datum`型の結果を返すもの（そして、SQLのNULLという結果を返そうとする場合に、`FunctionCallInfoData`構造体の`isnull`フィールドを設定するかもしれないもの）と想定されています。呼び出しハンドラと通常の呼び出される関数との違いは、`FunctionCallInfoData`構造体の`flinfo->fn_oid`に、呼び出しハンドラ自身ではなく、実際に呼び出される関数のOIDが含まれるという点です。呼び出しハンドラはこのフィールドを使用して、どの関数を呼び出すかを決定しなければなりません。また、渡された引数リストは、呼び出しハンドラの宣言ではなく、目的とする関数の宣言に従うよう設定されています。

`pg_proc`システムカタログから関数のエントリを取り出し、呼び出される関数の引数と戻り値の型を解析するまでを呼び出しハンドラが行います。関数の`CREATE FUNCTION`コマンドの`AS`句は、`pg_proc`の行の`prosrc`列にあります。これは通常、手続き言語自体で作成されたソーステキストですが、理論上はファイルへのパス名や、何らかの呼び出しハンドラに詳細に何をすべきかを通知するものとすることもできます。

1つのSQL文で同じ関数が何回も呼び出されることがよくあります。呼び出しハンドラは、`flinfo->fn_extra`フィールドを使用して、呼び出す関数に関する情報を繰り返し検索することを防ぐことができます。これは初期状態ではNULLですが、呼び出しハンドラによって呼び出す関数の情報を指すように設定することもできます。その後の呼び出しでは、`flinfo->fn_extra`が非NULLであれば、それを使用して、情報検索の段階を省略することができます。呼び出しハンドラは`flinfo->fn_extra`が少なくとも現在の問い合わせの終了まで有効なメモリを指しているかどうかを確認しなければなりません。`FmgrInfo`データ構造体は長く保持される可能性があるからです。この方法の1つとして、`flinfo->fn_mcxt`で指定されたメモリコンテキスト内に余分なデータを割り当てることです。このデータは通常`FmgrInfo`自身と同期間有効です。しかし、ハンドラはまた、長時間メモリコンテキストにあるものを使用するかどうかを選ぶこともできます。これにより関数定義情報を、問い合わせをまたいでキャッシュすることができます。

手続き言語関数がトリガとして呼び出された場合、引数は通常の方法では渡されず、`FunctionCallInfoData`の`context`フィールドが、普通の関数呼び出しのようにNULLにはならず、`TriggerData`構造体を指しています。呼び出しハンドラは、手続き言語に対しトリガ情報を取り出す機構を提供しなければなりません。

以下は、Cで作成した手続き言語ハンドラの雛型です。

```
#include "postgres.h"
#include "executor/spi.h"
#include "commands/trigger.h"
#include "fmgr.h"
#include "access/heapam.h"
#include "utils/syscache.h"
#include "catalog/pg_proc.h"
#include "catalog/pg_type.h"

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

PG_FUNCTION_INFO_V1(plsample_call_handler);

Datum
plsample_call_handler(PG_FUNCTION_ARGS)
{
    Datum          retval;

    if (CALLED_AS_TRIGGER(fcinfo))
    {
        /*
         * トリガ関数として呼び出された
         */
        TriggerData *trigdata = (TriggerData *) fcinfo->context;
```

```

        retval = ...
    }
else
{
    /*
     * 関数として呼び出された
     */

    retval = ...
}

return retval;
}

```

数千行のコードを上ドットの代わりに追加するだけで、呼び出しハンドラを完成することができます。

ハンドラ関数を動的ロード可能なモジュールにコンパイル ([項35.9.6](#)を参照してください) した後、以下のコマンドでサンプルの手続き言語を登録することができます。

```

CREATE FUNCTION plsample_call_handler() RETURNS language_handler
AS 'filename'
LANGUAGE C;
CREATE LANGUAGE plsample
HANDLER plsample_call_handler;

```

最低限の手続き言語を作成する場合には呼び出しハンドラを提供するだけで十分ですが、他にも省略可能ですが、その言語の利用をより簡便にするために提供できる2つの関数があります。これらは有効性検証関数とインラインハンドラです。有効性検証関数を提供して、[CREATE FUNCTION](#)時に言語固有の検査を行うことができます。インラインハンドラを提供して、言語に[DO](#)コマンド経由の匿名コードブロック実行をサポートさせることができます。

有効性検証関数が手続き言語により提供される場合、`oid`型の単一パラメータを取る関数として宣言しなければなりません。有効性検証関数の結果は無視されます。そのためよく`void`を返すものと宣言されます。有効性検証関数はその手続き言語で関数を作成または置換する[CREATE FUNCTION](#)の最後に呼び出されます。渡されるOIDは関数の`pg_proc`行のOIDです。有効性検証関数は通常の方法でこの行を取り出さなければならず、そして適切な検査を実行します。典型的な検査として、関数引数および結果の型がその言語でサポートされているかや関数本体がその言語において文法的に正しいかどうかを検証することが挙げられます。有効性検証関数がその関数に問題がないことを判定したら、単に戻ります。エラーがあることを判定したら、通常のエラー報告機構を使用して報告しなければなりません。エラーを返すことで、強制的にトランザクションはロールバックされ、不正な関数定義がコミットされることを防ぎます。

有効性検証関数は通常[check_function_bodies](#)パラメータを遵守しなければなりません。これが無効な場合、高価または文脈次第の検査を飛ばさなければなりません。具体的にはこのパラメータは、関数本体が他のデータベースオブジェクトに依存している可能性を考慮することなく手続き言語関数をロードできるように、`pg_dump`により無効にされます。(この仕様のため呼び出しハンドラは有効性検証関数が完全にその関数を検査したことを前提としてはいけません。有効性検証関数を持つ目的は、呼び出しハンドラが検査を省略できることではなく、明確なエラーが[CREATE FUNCTION](#)コマンド内に存在する場合、それを即座にユーザに通知することです。)

インラインハンドラが手続き言語により提供される場合、その関数は`internal`型の単一パラメータを取るものとして宣言されなければなりません。インラインハンドラの結果は無視されます。そのためよく`void`を返すものと宣言されます。インラインハンドラは特定の手続き言語で`DO`文が実行された時に呼び出されます。実際に渡されるパラメータは`InlineCodeBlock`構造体のポインタです。ここには`DO`文のパラメータ、具体的には実行される匿名コードブロックのテキスト、に関する情報が含まれています。インラインハンドラはこのコードを実行し、戻らなければなりません。

簡単な[CREATE EXTENSION](#)コマンドで言語をインストールすることが十分にできるように、これらの関数宣言と[CREATE LANGUAGE](#)コマンド自身を拡張としてまとめることを勧めます。拡張の作成方法については[項35.15](#)を参照してください。

独自の言語ハンドラを作成する際、標準配布物に含まれる手続き言語は優れたリファレンスです。ソースツリーの`src/pl`サブディレクトリを調べてください。[CREATE LANGUAGE](#)マニュアルページもまた有用な情報を含みます。

[前のページ](#)
プログラマへ

[ホーム](#)
[上に戻る](#)

[次のページ](#)
外部データラッパの作成

第 50章外部データラップの作成

目次

- 50.1. [Foreign Data Wrapper Functions](#)
- 50.2. [外部データラップのコールバックルーチン](#)
- 50.3. [外部データラップヘルパ関数](#)
- 50.4. [外部データラップのクエリプラン作成](#)

外部テーブルへの全ての操作は、コアサーバの呼び出しを処理する関数のセットで構成される、外部データラップによって扱われます。 リモートデータソースからデータを取り出し、そのデータをPostgreSQLエグゼキュータに返却する責任は外部データラップにあります。 本章では、新しい外部データラップを作成する方法の概要を示します。

独自の外部データラップを作成する際、標準配布物に含まれているものは優れたリファレンスです。 ソースツリーの `contrib/file_fdw` サブディレクトリを調べてください。 [CREATE FOREIGN DATA WRAPPER](#) マニュアルページにも有用な情報があります。

注意: 標準SQLでは外部データラップを作成するインターフェースを定義しています。 しかしながら、PostgreSQLに適応させる労力が大きく、また標準のAPIが広く採用されているわけでもないので、PostgreSQLはそのAPIを実装していません。

50.1. Foreign Data Wrapper Functions

FDWの作者は、ハンドラ関数と、オプションで検証関数を実装する必要があります。両関数とも、version-1インターフェースを使用して、Cなどのコンパイル言語で作成しなければなりません。C言語の呼び出し規約と動的ロードについては[項35.9](#)を参照してください。

ハンドラ関数は単に、プランナやエグゼキュータ、様々なメンテナンスコマンドから呼び出されるコールバック関数のポインタを保持する構造体を返します。FDWを作成するための労力のほとんどは、これらのコールバック関数を実装することに費やされます。ハンドラ関数は、引数を取らず特殊な仮想型である`fdw_handler`を返す関数としてPostgreSQLに登録しなければなりません。コールバック関数は通常のC言語関数で、SQLレベルでは参照も呼び出しもできません。コールバック関数の説明は[項50.2](#)にあります。

検証関数は、そのラップを使用する外部サーバ、ユーザマッピング、外部テーブルだけでなく、外部データラップ自身の`CREATE`や`ALTER`といったコマンドで指定されたオプションの妥当性を検証する責任があります。検証関数は、検証するオプションを含むtext配列と、オプションを関連付けるオブジェクトの種類を表すOID(そのオブジェクトが格納されるシステムカタログのOIDで`ForeignDataWrapperRelationId`または`ForeignServerRelationId`または`UserMappingRelationId`または`ForeignTableRelationId`のいずれか)という二つの引数を取るものとして登録しなければなりません。検証関数が指定されなかった場合、オブジェクト作成時やオブジェクト変更時にオプションはチェックされません。

50.2. 外部データラッパのコールバックルーチン

FDWハンドラ関数は、以下のコールバック関数へのポインタを含む、`palloc`された`FdwRoutine`構造体を返します。

```
void
GetForeignRelSize (PlannerInfo *root,
                  RelOptInfo *baserel,
                  Oid foreigntableid);
```

外部テーブルのリレーションサイズ見積もりを取得します。この関数は、ある外部テーブルを使用するクエリのプラン作成の開始時に呼び出されます。`root`はそのクエリに関するプランナのグローバル情報です。`baserel`はそのテーブルに関するプランナの情報です。そして、`foreigntableid`はその外部テーブルの`pg_class` OIDです。(`foreigntableid`はプランナデータ構造体からも取得できますが、手間を省くために明示的に渡されます。)

この関数は、制約条件によるフィルタリングも考慮に入れた、そのテーブルスキャンが返すと見込まれる件数に`baserel->rows`を更新するべきです。`baserel->rows`の初期値は固定のデフォルト見積もりなので、可能な限り置き換えられるべきです。この関数は、行の幅のよりよい見積もりを計算できるのであれば、`baserel->width`を更新することも選択出来ます。

追加情報については[項50.4](#)を参照してください。

```
void
GetForeignPaths (PlannerInfo *root,
                RelOptInfo *baserel,
                Oid foreigntableid);
```

外部テーブル対するスキャンとしてありえるアクセスパスを作成します。この関数はクエリのプラン作成中に呼び出されます。引数は、先に呼ばれている`GetForeignRelSize`と同じです。

この関数は、少なくとも一つのアクセスパス(`ForeignPath`ノード)を作成して、それぞれのパスを`baserel->pathlist`に追加するために`add_path`を呼ばなければなりません。`ForeignPath`ノードを構築するには`create_foreignscan_path`を使うことが推奨されています。この関数は、たとえばソート済みの結果を表現する有効な`pathkeys`を持つパスのような複数のアクセスパスを作成することが出来ます。それぞれのアクセスパスはコスト見積もりを含まねばならず、また意図する特定のスキャン方式を識別するのに必要なFDW固有の情報を持つことが出来ます。

追加情報については[項50.4](#)を参照してください。

```
ForeignScan *
GetForeignPlan (PlannerInfo *root,
               RelOptInfo *baserel,
               Oid foreigntableid,
               ForeignPath *best_path,
               List *tlist,
               List *scan_clauses);
```

選択された外部アクセスパスから`ForeignScan`プランノードを作成します。この関数はクエリプラン作成の最後に呼び出されます。引数は、`GetForeignRelSize`と同じものに、選択された`ForeignPath`(事前に`GetForeignPaths`によって作成されたもの)と、そのプランノードによって出力されるターゲットリスト、およびそのプランノードで強制される制限句が追加されます。

この関数は`ForeignScan`プランノードを作成して返さなければなりません。`ForeignScan`ノードを構築するには`make_foreignscan`を使うことが推奨されています。

追加情報については[項50.4](#)を参照してください。

```
void
ExplainForeignScan (ForeignScanState *node,
                  ExplainState *es);
```

外部テーブルスキャンの追加の`EXPLAIN`出力を表示します。何も表示する必要がないのであれば、単にリターンすることもできます。もしくは、`EXPLAIN`出力にフィールドを追加するために`ExplainPropertyText`や関連する関数を呼び出すべきです。`es`の中のフラグフィールドは何を表示するかを決めるのに使用できます。また、`EXPLAIN ANALYZE`の場合には、実行時統計情報を提供するために`ForeignScanState`ノードの状態を調べることができます。

```
void
BeginForeignScan (ForeignScanState *node,
                 int eflags);
```

外部テーブルスキャンの実行を開始します。この関数はエグゼキュータのスタートアップ中に呼び出されます。スキャンを開始できるようになる前に、あらゆる必要な初期化を実行するべきですが、実際のスキャンの実行を始めるべきではありません(それは最初の`IterateForeignScan`呼び出しにおいて行われるべきです)。`ForeignScanState`ノードは作成されていますが、その`fdw_state`フィールドは`NULL`のままです。スキャンするテーブルの情報は、`ForeignScanState`ノード(実際にはその先にあ

る、PlanForeignScanから返されたFdwPlan構造体へのポインタを含む、ForeignScanプランノード)を通じてアクセス可能です。

(eflags & EXEC_FLAG_EXPLAIN_ONLY) が真の場合、この関数は外部に見えるアクションを起こすべきではないことに注意してください。 ExplainForeignScanやEndForeignScan用にノード状態を有効にするのに必要とされる最小限のことだけをすべきです。

```
TupleTableSlot *
IterateForeignScan (ForeignScanState *node);
```

外部ソースから一行を取り出して、それをタプルテーブルスロットに入れて返します(この用途にはノードのScanTupleSlotを使うべきです)。 利用可能な行がない場合は、NULLを返します。 タプルテーブルスロット機構を使うと、物理タプルと仮想タプルのどちらでも返せます。 ほとんどの場合、パフォーマンスの観点から後者を選ぶのが良いでしょう。 この関数は、呼出しごとにリセットされる短命なメモリコンテキスト内で呼び出されることに注意してください。 より長命なストレージが必要な場合は、BeginForeignScanでメモリコンテキストを作成するか、ノードのEStateに含まれるes_query_cxtを使用してください。

返される行はスキャンされている外部テーブルのカラムシグネチャと一致しなければなりません。 不要なカラムを取り出さないように最適化することを選ぶなら、それらのカラム位置にNULLを入れるべきです。

PostgreSQLのエグゼキュータは返された行が外部テーブルの列に定義されたNOT NULL制約に違反しているかどうかは気にしませんが、プランナはそれに関心を持ち、NULL値を含まないと宣言された列にNULL値が現れた場合に不正なクエリ最適化をしてしまう場合があることに注意してください。 ユーザがあつてはならないと宣言したのにNULL値に遭遇した場合は(データ型が一致しなかった場合にする必要があるのと同様に)エラーを発生させるのが適切でしょう。

```
void
ReScanForeignScan (ForeignScanState *node);
```

先頭からスキャンを再開します。 スキャンが依存するいずれかのパラメータが値を変更しているかもしれないので、新しいスキャンが必ずしも厳密に同じ行を返すとは限らないことに注意してください。

```
void
EndForeignScan (ForeignScanState *node);
```

スキャンを終了しリソースを解放します。 通常、pallocされたメモリを解放することは重要ではありませんが、開いたファイルやリモートサーバへの接続などはクリーンアップするべきです。

```
bool
AnalyzeForeignTable (Relation relation,
                     AcquireSampleRowsFunc *func,
                     BlockNumber *totalpages);
```

この関数はANALYZEが外部テーブルに対して実行されたときに呼び出されます。 もしFDWがこの外部テーブルの統計情報を収集できる場合は、そのテーブルからサンプル行を集める関数のポインタとテーブルサイズのページ単位での見積もりをそれぞれfuncとtotalpagesに渡しtrueを返す必要があります。 そうでない場合は、falseを返します。 もしFDWが統計情報の収集をどのテーブルについてもサポートしない場合は、AnalyzeForeignTableポインターをNULLにすることもできます。

もし提供される場合は、サンプル収集関数はこのようなシグネチャを持つ必要があります。

```
int
AcquireSampleRowsFunc (Relation relation, int elevel,
                      HeapTuple *rows, int targrows,
                      double *totalrows,
                      double *totaldeadrows);
```

最大targrows行のランダムサンプルをテーブルから収集し、呼び出し元が提供するrows配列に格納する必要があります。 実際に収集された行の数を返す必要があります。 さらに、テーブルに含まれる有効行と不要行の合計数の見積もりを出力パラメータのtotalrowsとtotaldeadrowsに返す必要があります。(もしFDWが不要行という概念を持たない場合はtotaldeadrowsを0に設定してください。)

FdwRoutine構造体はさらなる情報を含むsrc/include/foreign/fdwapi.hで宣言されています。

FdwRoutineとFdwPlanの構造体型はsrc/include/foreign/fdwapi.hで宣言されているので、さらなる詳細はそちらを見て下さい。

[前のページ](#)

Foreign Data Wrapper Functions

[ホーム](#)

[上に戻る](#)

[次のページ](#)

外部データラップヘルパ関数

50.3. 外部データラッパヘルパ関数

FDWオプションのようなFDW関連オブジェクトの属性に外部データラッパの作者が簡単にアクセスできるように、いくつかのヘルパ関数がコアサーバからエクスポートされています。これらの関数を使用するには、ヘッダファイル`foreign/foreign.h`をあなたのソースファイルにインクルードする必要があります。このヘッダはまたこれらの関数が返す構造体も定義しています。

```
ForeignDataWrapper *  
GetForeignDataWrapper(Oid fdwid);
```

この関数は指定されたOIDを持つ外部データラッパの`ForeignDataWrapper`オブジェクトを返します。`ForeignDataWrapper`オブジェクトはFDWのプロパティを含みます(詳細は`foreign/foreign.h`を参照して下さい)。

```
ForeignServer *  
GetForeignServer(Oid serverid);
```

この関数は指定されたOIDを持つ外部サーバの`ForeignServer`オブジェクトを返します。`ForeignServer`オブジェクトは外部サーバのプロパティを含みます(詳細は`foreign/foreign.h`を参照して下さい)。

```
UserMapping *  
GetUserMapping(Oid userid, Oid serverid);
```

この関数は指定されたロールと指定された外部サーバのユーザマッピングの`UserMapping`オブジェクトを返します。(もし特定のユーザのマッピングがない場合は、`PUBLIC`のためのマッピングを返すか、それもなければエラーを発生させます。)

`UserMapping`オブジェクトはユーザマッピングのプロパティを含みます(詳細は`foreign/foreign.h`を参照して下さい)。

```
ForeignTable *  
GetForeignTable(Oid relid);
```

この関数は指定されたOIDを持つ外部テーブルの`ForeignTable`オブジェクトを返します。`ForeignTable`オブジェクトは外部テーブルのプロパティを含みます(詳細は`foreign/foreign.h`を参照して下さい)。

```
List *  
GetForeignColumnOptions(Oid relid, AttrNumber attnum);
```

この関数は指定された外部テーブルOIDと属性番号に該当する列の列単位のFDWオプションを`DefElem`のリスト形式で返します。その列がオプションを持たない場合は`NIL`が返ります。

いくつかのオブジェクト種別については、OIDベースのものに加えて名前ベースの検索関数もあります。

```
ForeignDataWrapper *  
GetForeignDataWrapperByName(const char *name, bool missing_ok);
```

この関数は指定された名前の外部データラッパの`ForeignDataWrapper`オブジェクトを返します。外部データラッパが見つからない場合は、`missing_ok`が`true`の場合は`NULL`を返し、それ以外の場合はエラーを発生させます。

```
ForeignServer *  
GetForeignServerByName(const char *name, bool missing_ok);
```

この関数は指定された名前の外部サーバの`ForeignServer`オブジェクトを返します。外部サーバが見つからない場合は、`missing_ok`が`true`の場合は`NULL`を返し、それ以外の場合はエラーを発生させます。

50.4. 外部データラッパのクエリプラン作成

FDWコールバック関数の`GetForeignRelSize`、`GetForeignPaths`、`GetForeignPlan`はPostgreSQLプランナの動作と協調しなければなりません。ここでは、これらの関数がすべき事に関するいくつかの注意事項を述べます。

`root`と`basere1`に含まれる情報は、外部テーブルから取得する必要のある情報の量(とそれによるコスト)を削減するために使用できます。`basere1->baserestrictinfo`は、取得される行をフィルタリングする制約条件(`WHERE`句)を含んでいるため、特に興味深いものです。(コアのエグゼキュータが代わりにそれらをチェックできるので、FDW自身がこれらの制約を適用しなければならないわけではありません。) `basere1->reltargetlist`はどのカラムが取得される必要があるかを決定するのに使用できます。ただし、このリストは`ForeignScan`プランノードから出力すべきカラムしか含んでおらず、条件検査には必要でもクエリからは出力されないカラムは含まないことに注意してください。

様々なプライベートフィールドがFDWのプラン作成関数で情報を格納する目的で利用できます。一般的に、プラン作成の最後に回収できるように、FDW固有フィールドに格納するものは全て`palloc`で確保すべきです。

`basere1->fdw_private`は、`void`ポインタで、FDWのプラン作成関数で特定の外部テーブルに関する情報を格納する目的で利用できます。コアプランナは、`basere1`ノードが作成されるときに`NULL`で初期化するときを除いて、このフィールドに一切に触れません。このフィールドは、`GetForeignRelSize`から`GetForeignPaths`や`GetForeignPaths`から`GetForeignPlan`といったように情報を順次伝えるの便利で、結果として再計算を省くことができます。

`GetForeignPaths`では、`ForeignPath`ノードの`fdw_private`フィールドに固有情報を格納することで、異なるアクセスパスを区別できます。`fdw_private`は`List`ポインタとして宣言されていますが、コアプランナがこのフィールドを操作することはないため、実際にはなんでも格納できます。しかし、バックエンドのデバッグサポート機能を利用できるように`nodeToString`でダンプ出来る形式を使うのが最良の手法です。

`GetForeignPlan`では、選択された`ForeignPath`ノードの`fdw_private`フィールドを調べて、`ForeignScan`プランノード内に格納されプラン実行時に利用可能な`fdw_exprs`と`fdw_private`の二つのリストを生成することができます。これらは両方とも`copyObject`がコピーできる形式でなければなりません。`fdw_private`リストにはこれ以外に制約はなく、コアバックエンドによって解釈されることはありません。`fdw_exprs`リストが`NIL`でない場合は、クエリ実行時に実行されることを意図した式ツリーが含まれていることが期待されます。これらのツリーは、完全に実行可能な状態にするためにプランナによる後処理を受けます。

`GetForeignPlan`では、一般的に渡されたターゲットリストはそのままプランノードにコピーできます。渡された`scan_clauses`リストは`basere1->baserestrictinfo`と同じ句を含みますが、実行効率のよい別の順番に並べ替えることもできます。FDWにできるのが`RestrictInfo`ノードを`scan_clauses`リストから(`extract_actual_clauses`を使って)抜き出して、全ての句をプランノードの`qual`リストに入れるだけ、といった単純なケースでは、全ての句は実行時にエグゼキュータによってチェックされます。より複雑なFDWは内部で一部の句をチェックできるかもしれませんが、そのような場合には、エグゼキュータが再チェックのために時間を無駄にしないように、それらの句はプランノードの`qual`リストから削除できます。

たとえば、ローカル側で評価された`sub_expression`の値があればリモートサーバ側で実行出来るとFDWが判断するような、`foreign_variable = sub_expression`といった形式の制限句をFDWが識別するかもしれません。パスのコスト見積もりに影響するので、そのような句の実際の識別は`GetForeignPaths`でなされるべきです。おそらく、そのパスの`fdw_private`フィールドは識別された句の`RestrictInfo`ノードをさすポインタを含むでしょう。そして、`GetForeignPlan`はその句を`scan_clauses`から取り除き、実行可能な形式にほぐされることを保障するために`sub_expression`を`fdw_exprs`に追加するでしょう。また、おそらく、実行時に何をすべきかをプラン実行関数に伝えるためにプランノードの`fdw_private`フィールドに制御情報を入れるでしょう。リモートサーバに送られたクエリは、実行時に`fdw_exprs`式ツリーを評価して得られた値をパラメータ値とする`WHERE foreign_variable = $1`のようなものを伴うでしょう。

FDWはそのテーブルの制限句のみに依存するパスを常に少なくとも一つは生成すべきです。結合クエリでは、例えば`foreign_variable = local_variable`といった結合句に依存するパス(群)を生成することもできます。そのような句は`basere1->baserestrictinfo`には見つからず、リレーションの結合リストにあるはずで、そのような句を使用するパスは"parameterized path"と呼ばれます。このようなパスでは、選択された結合句(群)で使用されているリレーション(群)を`param_info`の適合する値から特定しなければなりません;その値を計算するには`get_basere1_parampathinfo`を使用します。`GetForeignPlan`では、結合句の`local_variable`部分が`fdw_exprs`に追加され、実行時には通常の制限句と同じように動作します。

第 51 章 遺伝的問い合わせ最適化

目次

- 51.1. [複雑な最適化問題としての問い合わせ処理](#)
- 51.2. [遺伝的アルゴリズム](#)
- 51.3. [PostgreSQLの遺伝的問い合わせ最適化 \(GEQO\)](#)
 - 51.3.1. [GEQOを使用した計画候補の生成](#)
 - 51.3.2. [PostgreSQL GEQOの今後の実装作業](#)
- 51.4. [さらに深く知るには](#)

著者: このドキュメントはMartin Utesch (<utesch@aut.tu-freiberg.de>) によって、ドイツ、フライブルグにあるUniversity of Mining and TechnologyのInstitute of Automatic Controlのために書かれました。

51.1. 複雑な最適化問題としての問い合わせ処理

リレーショナル演算子の中で、処理と最適化が一番難しいのは結合です。実行可能な問い合わせ計画の数は問い合わせの中に含まれる結合の数によって指数関数的に増加します。個々の結合や、多様なインデックス（例えばPostgreSQLのB-tree、ハッシュ、GiST、GINなど）をリレーションのアクセスパスとして処理するため、様々な結合メソッド（例えばPostgreSQLの入れ子ループ、ハッシュ結合、マージ結合など）を提供することが、さらなる最適化を行わなければならない腐心の原因となっています。

通常のPostgreSQL問い合わせオプティマイザは、候補ストラテジ空間にわたってしらみつぶしに近い検索を行います。IBMのSystem Rデータベースで初めて導入された、このアルゴリズムはほぼ最適な結合順を生成しますが、問い合わせ内の結合数が増えると膨大な処理時間とメモリ空間を必要とします。このため、通常のPostgreSQL問い合わせオプティマイザは結合するテーブル数の多い問い合わせには向いていません。

ドイツ、フライブルグにあるUniversity of Mining and TechnologyのInstitute of Automatic Controlでは、送電網の保守のための意志決定知識ベースシステムのためのバックエンドとしてPostgreSQL DBMSを使おうとしたため問題が起きました。そのDBMSは知識ベースシステムの推論マシンのために、大規模な結合の問い合わせを処理する必要があったのです。こうした問い合わせに含まれる結合数を行うことは、通常の問い合わせオプティマイザでは実現不可能でした。

以下では、多数の結合を持つ問い合わせを効率的に行うことができるように、結合順問題を解決する遺伝的アルゴリズムの実装を説明します。

51.2. 遺伝的アルゴリズム

遺伝的アルゴリズム (GA) は発見的な最適化手法で、無作為の検索として働きます。最適化の問題に対する解の集合は個体群とみなされます。個体の環境への順応の度合は適応度によって指定されます。

検索空間の中で個体の同格性は、その実体が文字列の集合である染色体によって表現されます。遺伝子は最適化をしようとしている1つのパラメータの値を符号化する染色体の一部分です。遺伝子の符号化の典型的な例としてバイナリもしくは整数が挙げられます。

進化の過程のシミュレーションである、再組合せ、突然変異、淘汰を通して、祖先よりも適応度の平均が高い新世代の検索点が見つけられます。

comp.ai.geneticのFAQによると、GAが問題に対する純粋な無作為検索ではないことをどんなに強調してもし過ぎということはありません。GAは確率的なプロセスを使いますが、結果は明らかに（無作為よりもより良い）非無作為です。

図 51-1. 遺伝的アルゴリズムの構造図

$P(t)$ 時間 t における先祖の世代

$P''(t)$ 時間 t における子孫の世代

```
+=====+
|>>>>>>>>>> Algorithm GA <<<<<<<<<<<<<<|
+=====+
| INITIALIZE t := 0                               |
+=====+
| INITIALIZE P(t)                                 |
+=====+
| evaluate FITNESS of P(t)                         |
+=====+
| while not STOPPING CRITERION do                 |
|   +-----+                                     |
|   | P'(t) := RECOMBINATION{P(t)}                 |
|   +-----+                                     |
|   | P''(t) := MUTATION{P'(t)}                   |
|   +-----+                                     |
|   | P(t+1) := SELECTION{P''(t) + P(t)}           |
|   +-----+                                     |
|   | evaluate FITNESS of P''(t)                   |
|   +-----+                                     |
|   | t := t + 1                                   |
|   +-----+                                     |
+=====+
```


51.3. PostgreSQLの遺伝的問い合わせ最適化（GEQO）

GEQOのモジュールは、問い合わせ最適化問題をあたかもよく知られている巡回セールスマン問題（TSP）のように扱います。可能な問い合わせプランは、整数の文字列として符号化されます。それぞれの文字列は、問い合わせの1つのリレーションから次へと結合の順番を表します。例えば、以下の結合ツリーは整数文字列「4-1-3-2」によって符号化されています。

```

      /¥
     /¥ 2
    /¥ 3
   /¥ 1
  4 1
```

これが意味するのは、まずリレーション「4」と「1」を、次に「3」を、そして「2」を結合するということです。ここで1、2、3、4はPostgreSQLオプティマイザ内でリレーションIDを表します。

PostgreSQLにおけるGEQO実装の特有な特徴は下記の様なものです。

- 定常状態GAの使用（世代全体の置き換えではなく、個体の中で適応度の低いものだけの置き換え）は、改良された問い合わせ計画へ素早い収束を可能にします。これは、妥当な時間内での問い合わせ処理にはきわめて重要です。
- GAによるTSPの解決策の辺損失を低く抑えるため、非常に適した辺再組合せ交叉を使用します。
- TSPの合法的巡回を行うために必要な修復処理を要求しないように、遺伝的演算子の突然変異は無視しています。

GEQOモジュールの部品は D. WhitleyのGenitorアルゴリズムを適合させたものです。

GEQOモジュールにより、PostgreSQL問い合わせオプティマイザが、大きな結合問い合わせをしらみつぶし検索以外の方法で実行することが可能になります。

51.3.1. GEQOを使用した計画候補の生成

GEQOの計画作成では、個々のリレーションのスキャンに対する計画を生成するために標準のプランナが使用されます。そして、結合計画が遺伝的手法を用いて展開されます。上で示した通り、結合計画候補はそれぞれ、基本リレーションの結合順によって表現されています。初期段階では、GEQOコードは単純にランダムに取り得る結合順をいくつか生成します。考慮された結合順それぞれについて、標準プランナコードが呼び出され、その結合順を使用して問い合わせを行った場合のコストを推定します。（結合順の各段階において、全体で3つの取り得る結合戦略が考慮されます。そして、あらかじめ決められたリレーションスキャン計画もすべて利用可能です。推定コストとはこれらの可能性の中から最も安価なものです。）より低い推定コストの結合順を、より高い推定コストのものより"より高い適応度"と判断します。遺伝的アルゴリズムは適応度が低い候補を破棄します。そして、より多く合致する候補の遺伝子を組み合わせ、つまり、検討すべき新しい順序を作成するために既知の低コスト結合順をランダムに位置を選択して、新しい候補が生成されます。事前に設定された数まで結合順を検討するまで、この処理が繰り返されます。そして、この検索の間にもっとも優れたものが、最終的な計画を生成するために使用されます。

初期の群を選択する時、および、その後の最善の候補の突然変異の時に無作為な選択がなされますので、この処理は生来非決定論的なものです。選択された計画の予期せぬ変化を避けるために、GEQOアルゴリズムの各実行では乱数生成器を現在の`geqo_seed`パラメータ設定で再スタートさせます。`geqo_seed`とその他のGEQOパラメータが変更されない限り、一定の問い合わせ（と統計のようなプランナへの他の入力）に対しては同じ計画が生成されます。異なる検索パスで実験するためには、`geqo_seed`を変更してみてください。

51.3.2. PostgreSQL GEQOの今後の実装作業

遺伝的アルゴリズムのパラメータ設定を改善するためにはまだ課題が残っています。

`src/backend/optimizer/geqo/geqo_main.c`の`gimme_pool_size`と`gimme_number_generations`というルーチンでは、次の2つの相反する要求を満たす妥協点を見つけなければいけません。

- 問い合わせ計画の最適性
- 計算時間

現在の実装では、各結合順候補の適応度は標準プランナの結合選択と、一から作成したコスト推定コードを実行して推定されます。異なる候補が同様の副結合順で使用されるにつれて、多くの作業が繰り返されることとなります。これは、副結合のコスト推定を記憶することで、非常に高速になるはずですが、この状態を記憶するために要するメモリ量が非合理的に拡大することを防止することが問題です。

最も基本的なレベルでは、TSP用に設計されたGAアルゴリズムを用いた問い合わせ最適化の解法が適切かどうかは明確ではありません。

ません。TSPの場合は、部分文字列（巡回経路の一部）に関連付けられたコストは残りの巡回経路と独立していますが、これは問い合わせ最適化の場合には確実に成り立ちません。したがって、辺再組合せ交叉が最も有効な突然変異手続きかどうかは疑わしいと言えます。

[前のページ](#)
遺伝的アルゴリズム

[ホーム](#)
[上に戻る](#)

[次のページ](#)
さらに深く知るには

51.4. さらに深く知るには

次に示す資料は、さらに詳しい遺伝的アルゴリズムに関する情報が記載されています。

- [The Hitch-Hiker's Guide to Evolutionary Computation](#), (FAQ for news://comp.ai.genetic)
- [Evolutionary Computation and its application to art and design](#), Craig Reynoldsによるもの
- [Fundamentals of Database Systems](#)
- [The design and implementation of the POSTGRES query optimizer](#)

第 52章 インデックスアクセスメソッドのインタフェース定義

目次

- 52.1. [インデックス用のカタログ項目](#)
- 52.2. [インデックスアクセスメソッド関数](#)
- 52.3. [インデックススキャン](#)
- 52.4. [インデックスのロック処理に関する検討](#)
- 52.5. [インデックス一意性検査](#)
- 52.6. [インデックスコスト推定関数](#)

本章は、PostgreSQLのコアシステムと個々のインデックス種類を管理するインデックスアクセスメソッドとの間のインタフェースを定義します。コアシステムはインデックスの仕様のみを把握しています。したがって、追加コードを記述することで完全に新しいインデックス種類を開発することができます。

PostgreSQLのインデックスはすべて、技術的には補助的なインデックスとして知られるものです。つまり、インデックスは対象となるテーブルファイルとは物理的に分かれています。各インデックスは独自の物理的なリレーションとして格納され、また、`pg_class`カタログ内の項目として記述されます。インデックスの内容は完全にそのインデックスアクセスメソッドの制御下にあります。実際、すべてのインデックスアクセスメソッドは、通常の格納マネージャとバッファマネージャを使用してインデックスの内容にアクセスできるように、インデックスを標準サイズのページに分割します。（既存のすべてのインデックスアクセスメソッドではさらに、[項56.6](#)で説明する標準ページレイアウトを使用し、すべて同じ書式をインデックスタブルヘッダに使用します。しかし、これはアクセスメソッドに強制されていることではありません。）

インデックスは効率的にあるデータキー値を、インデックスの親テーブル内の行バージョン（タプル）のタプル識別子言い換えるとTIDに関連付けます。TIDは、ブロック番号、ブロック内の項目番号（[項56.6](#)を参照）から構成されます。これは、特定の行バージョンをテーブルから取り出すのに十分な情報です。MVCCでは1つの論理的な行に複数の現在のバージョンがあることを、インデックスが直接意識することはありません。インデックスでは、各タプルは、独自にインデックス項目を持たなければならない独立したオブジェクトです。したがって、行を更新すると、キーの値が変わっていなくても、その行に対してまったく新しいインデックス項目が作成されます。（HOTタプルはこの説明の例外ですが、インデックスはこれらにも関与しません。）（バキューム実行によって）不要タプル自身が回収された時に、不要タプルに対するインデックス項目は回収されます。

52.1. インデックス用のカタログ項目

各インデックスメソッドは、pg_amシステムカタログ ([項45.3](#)を参照) 内の行によって記述されます。pg_am行の主な内容は、アクセスメソッドで提供されるインデックスアクセス関数を識別するpg_proc項目への参照です。本章の後でこの関数用のAPIを定義します。pg_am行はさらに、複数列のインデックスをサポートするかなどといった、アクセスメソッドの固定属性をいくつか規定します。現時点では、pg_am項目の作成や削除に関する特別な補助はありません。新しいアクセスメソッドを作成することができる人間ならば、適切な行を挿入できる能力があると想定しています。

使い易くするために、インデックスアクセスメソッドはまた、pg_opfamily、pg_opclass、pg_amopおよびpg_amproc内で定義される、複数の演算子族と演算子クラスを持ちます。これらの項目により、プランナは、このアクセスメソッドのインデックスがどのような問い合わせ条件に対して使用できるかを決定することができます。演算子族と演算子クラスについては、[項35.14](#)で説明します。これは本章を読む上で必要となる資料です。

個々のインデックスは、インデックスを物理的なリレーションとして記述するpg_class項目と、インデックスの論理的内容、つまり、インデックスが持つインデックス列の集合とその列の意味を、関連する演算子クラスで再現されたものとして表すpg_index項目とで定義されます。インデックス列(キー値)は、背後のテーブルの単純な列、あるいは、テーブル行に対する式とすることができます。通常、インデックスアクセスメソッドはインデックスキー値が何を表すかについて考慮しません。(常に計算済みのキー値として扱われます。) しかし、pg_index内の演算子クラスの情報を深く考慮します。この両方のカタログ項目は、インデックスに対するすべての操作に渡されるRelationデータ構造の一部としてアクセスすることができます。

pg_amのフラグ列の中には、明確な意味を持たないものがあります。amcanuniqueの必要条件は[項52.5](#)で説明されています。amcanmulticolフラグはアクセスメソッドが複数列に対するインデックスをサポートすることを表し、amoptionalkeyは、インデックス可能な制限句が最初のインデックス列に指定されていないスキャンを許可することを表します。amcanmulticolが偽の場合、amoptionalkeyは基本的に、アクセスメソッドが制限句なしで完全なインデックススキャンをサポートするかどうかを表します。複数列に対するインデックスをサポートするアクセスメソッドは、最初の列以降のすべてまたは一部の列に関する制限がなくてもスキャンをサポートしなければなりません。しかし、最初のインデックス列にいくつかの制限を要求することは認められています。これは、amoptionalkeyを偽に設定することで通知されます。インデックスアクセスメソッドがamoptionalkeyを偽にする1つの理由は、NULLをインデックス付けしない場合です。ほとんどのインデックス可能な演算子は厳密で、NULL値の入力に対してTRUEを返すことができませんので、NULLに対してインデックス項目を格納しないことは一見魅力的です。これはインデックススキャンによって何も返しません。しかし、最初のインデックス列に対する制限がないインデックススキャンでは、この引数は失敗します。プランナがこうしたスキャンキーをまったく持たないインデックスを使用することを決定する可能性がありますので、実際これは、amoptionalkeyが真のインデックスはNULLインデックスを持たなければならないことを意味します。関連する制限として、プランナはこれらの列を制限しない問い合わせでインデックスを使用することができると前提とするため、複数のインデックス列をサポートするインデックスアクセスメソッドは1番目の後の列でNULL値のインデックスをサポートしなければならないということがあります。例えば、(a,b)に対するインデックスに、WHERE a = 4という条件で問い合わせを行うことを考えてみます。システムは、このインデックスをa = 4を持つ行をスキャンすることに使用できるものと仮定します。これはもし、bがNULLの場合の行をインデックスが省略する場合は間違っています。しかし、最初のインデックス列がNULLの場合に行を省略することは問題ありません。また、NULLをインデックス付けするインデックスアクセスメソッドはamsearchnullsを設定する可能性があります。これは検索条件としてIS NULLおよびIS NOT NULL句をサポートすることを示します。

52.2. インデックスアクセスメソッド関数

インデックスアクセスメソッドが提供しなければならない、インデックス構築および保守関数を以下に示します。

```
IndexBuildResult *
ambuild (Relation heapRelation,
         Relation indexRelation,
         IndexInfo *indexInfo);
```

新しいインデックスを構築します。 空のインデックスリレーションが物理的に作成されます。 これは、アクセスメソッドが必要とする何らかの固定データと、テーブル内に既存のすべてのタプルに対応する項目が書き込まれなければなりません。 通常、`ambuild`関数は`IndexBuildHeapScan()`を呼び出し、既存のタプルをテーブルからスキャンし、インデックスに挿入しなければならないキーを計算します。 この関数は、新しいインデックスに関する統計情報を含む`palloc`された構造体を返さなければなりません。

```
void
ambuildempty (Relation indexRelation);
```

空のインデックスを構築し、それを指定されたリレーションの初期フォーク (`INIT_FORKNUM`) に書き出します。 このメソッドはログを取らないテーブルに対してのみ呼び出されます。 初期フォークに書き出された空のインデックスは、サーバの再起動の度に主リレーションフォークにコピーされます。

```
bool
amininsert (Relation indexRelation,
            Datum *values,
            bool *isnull,
            ItemPointer heap_tid,
            Relation heapRelation,
            IndexUniqueCheck checkUnique);
```

既存のインデックスに新しいタプルを挿入します。 `values`配列と`isnull`配列がインデックスされるキー値を提供するもので、`heap_tid`がインデックスされるTIDです。 アクセスメソッドが一意的なインデックスをサポートする場合（その`pg_am.amcanunique`が真の場合）、`checkUnique`は実行する一意性検査の種類を示します。 これは一意性制約が遅延可能か否かによって変わります。 [項52.5](#)を参照してください。 通常アクセスメソッドは、一意性検査を行う時に`heapRelation`パラメータのみを必要とします（タプルの有効性を検証するためにヒープ内を検索しなければなりません）。

`checkUnique`が`UNIQUE_CHECK_PARTIAL`の場合、関数の論理型の結果値で十分です。 この場合、真の結果は新しい項目は一意であることが確認されたことを、一方偽の結果は一意でない可能性があること（遅延一意性検査を予定しなければならないこと）を意味します。 他の場合では、一定の偽という結果が推奨されます。

一部のインデックスではすべてのタプルをインデックス付けしない可能性があります。 タプルがインデックス付けされない場合、`amininsert`は何も行わずに戻らなければなりません。

```
IndexBulkDeleteResult *
ambulkdelete (IndexVacuumInfo *info,
              IndexBulkDeleteResult *stats,
              IndexBulkDeleteCallback callback,
              void *callback_state);
```

インデックスからタプル（複数可）を削除します。 これは"一括削除"操作を行いますが、インデックス全体をスキャンし、各項目に対して削除すべきかどうか検査を行うように実装されることが想定されています。 渡される`callback`関数は、`callback(TID, callback_state) returns bool`という形で、参照用TIDで識別されるインデックス項目を削除すべきかどうか決定するために呼び出さなければなりません。 NULLまたは`palloc`した削除操作の影響に関する統計情報を含む構造体を返さなければなりません。 `amvacuumcleanup`に渡さなければならない情報がなければ、NULLを返しても問題ありません。

`maintenance_work_mem`の制限により、多くのタプルが削除される時、`ambulkdelete`を複数回呼び出す必要があるかもしれません。 `stats`引数は、このインデックスに対する前回の呼び出し結果です。（`VACUUM`操作における最初の呼び出しではこれはNULLです。） これにより、アクセスメソッドは操作全体に跨った統計情報を計算することができます。 典型的に、渡された`stats`がNULLでない場合、`ambulkdelete`は同じ構造体を変更し、返します。

```
IndexBulkDeleteResult *
amvacuumcleanup (IndexVacuumInfo *info,
                 IndexBulkDeleteResult *stats);
```

`VACUUM`操作（0回以上の`ambulkdelete`呼び出し）後の整理を行います。 これは、インデックス統計情報を返す以上の処理を行う必要はありません。 しかし、空のインデックスページの回収などの一括整理を行う可能性があります。 `stats`は最後の`ambulkdelete`呼び出しが返したものです。 削除する必要があるタプルが存在しなかったために`ambulkdelete`が呼び出されなかった場合はNULLとなります。 結果はNULLでなければ、`palloc`された構造体でなければなりません。 含まれる統計情報は`pg_class`を更新するために使用され、また、`VERBOSE`が指定された`VACUUM`によって報告されます。 `VACUUM`操作の間にインデッ

クスがまったく変わらなかった場合はNULLを返しても問題ありません。しかし、そうでなければ正しい統計情報を返さなければなりません。

PostgreSQL 8.4の時点で、`amvacuumcleanup`もANALYZE操作の完了時点にも呼び出されます。この場合、`stats`は常にNULLで、戻り値はまったく無視されます。この事象は`info->analyze_only`を検査することで識別されます。アクセスメソッドがそのような呼び出しで挿入後の整理以外何もしないように、そしてそれは自動バキュームワーカプロセスのみであるようにすることを推奨します。

```
bool
amcanreturn (Relation indexRelation);
```

IndexTuple形式のインデックスエントリをインデックスが設定された列の値として返すことにより、そのインデックスがインデックスオンリースキャンをサポートしているかどうかを判断します。インデックスオンリースキャンがサポートされている場合は真が返され、サポートされていない場合は偽が返ります。インデックスのアクセスメソッドがインデックスオンリースキャンをサポートしていない場合(この例のインデックスはハッシュインデックスで、インデックスには元のデータではなくハッシュ値のみが格納されます)、`pg_am`の`amcanreturn`列にゼロがセットされます。

```
void
amcostestimate (PlannerInfo *root,
                IndexPath *path,
                double loop_count,
                Cost *indexStartupCost,
                Cost *indexTotalCost,
                Selectivity *indexSelectivity,
                double *indexCorrelation);
```

インデックススキャンのコストを推定します。この関数については後述の[項52.6](#)で説明します。

```
bytea *
amoptions (ArrayType *reloptions,
           bool validate);
```

インデックス用の`reloptions`の解析と検証を行います。インデックスに非NULLの`reloptions`配列が存在する場合にのみ呼び出されます。`reloptions`は、`name=value`形式の項目からなる、`text`型の配列です。この関数は`bytea`型の値を生成しなければならないが、この値はインデックスの`relcache`項目の`rd_options`フィールドにコピーされます。`bytea`型の値の内容はアクセスメソッドが独自に定義できるように開放されています。標準のアクセスメソッドのほとんどはすべて`StdRdOptions`構造体を使用します。`validate`が真の場合、何らかのオプションが認識できなかった場合や無効な値が存在した場合、この関数は適切なエラーメッセージを報告しなければなりません。`validate`が偽の場合、無効な項目は単に無視されます。(読み込みオプションが既に`pg_catalog`に格納されている場合`validate`は偽です。アクセスメソッドがそのオプション用の規則を変更した場合にのみ、無効な項目が検出されます。そして、その場合、古い項目を無視することが適切です。) デフォルトの動作を行わせたい場合はNULLを返しても問題ありません。

当然ながらインデックスの目的は、よく修飾子やスキャンキーと呼ばれる、インデックス可能なWHERE条件を満たすタブルのスキャンをサポートすることです。インデックススキャンのセマンティックスは後の[項52.3](#)でより詳しく説明します。インデックスアクセスメソッドは"単純"インデックススキャン、"ビットマップ"インデックススキャン、またはこれら双方を提供します。インデックスアクセスメソッドが提供しなければならない、もしくは提供する可能性のあるスキャン関連の関数を以下に示します。

```
IndexScanDesc
ambeginscan (Relation indexRelation,
             int nkeys,
             int norderbys);
```

インデックススキャンを準備します。`nkeys`および`norderbys`パラメータは、スキャンで使用される等価性演算子と順序付け演算子の個数を表します。これらは領域を割り当てる目的で便利かもしれませんが。スキャンキーの実値がまだ提供されていないことに注意してください。結果は`palloc`した構造体でなければなりません。実装上の理由により、インデックスアクセスメソッドは`RelationGetIndexScan()`呼び出しによってこの構造体を作成しなければなりません。ほとんどの場合、`ambeginscan`はこの呼び出しとおそらくロックの獲得の他にはほとんど何も行いません。インデックススキャンを始める際の興味深い部分は、`amrescan`にあります。

```
void
amrescan (IndexScanDesc scan,
          ScanKey keys,
          int nkeys,
          ScanKey orderbys,
          int norderbys);
```

インデックススキャンを起動または再起動します。スキャンキーを新しくすることもできます。(過去に渡されたキーを使用して再起動するには、`key`、`orderbys`、またはその両方にNULLを渡します。) `ambeginscan`に渡したキー演算子、順序付け演算子の個数より多くを使用することはできないことに注意してください。実際には、入れ子状ループ結合によって新しい外部タブルが選択され、同じスキャンキー構造体で新しいキー比較値が必要とされた場合に、この再起動機能は使用されます。

```
boolean
amgettupple (IndexScanDesc scan,
```



```
ScanDirection direction);
```

指定されたスキャン内から指定された方向（インデックス内の前方または後方）で次のタプルを取り出します。タプルを取り出した場合は真を返します。一致するタプルが残っていない場合は偽を返します。真の場合、そのタプルのTIDが`scan`に格納されます。"成功"とは、単にインデックスにスキャンキーに一致する項目があったことを意味しているだけです。タプルが必ずヒープ内に存在することや、呼び出し元のスナップショットの試験を通過したことを意味してはいません。成功の暁には、`amgettupple`は`scan->xs_recheck`を真か偽かに設定しなければなりません。偽の意味は、インデックス項目が確実にスキャンキーに一致することです。真の意味は、これが確かなことではなく、スキャンキーで表示された条件がヒープタプルを取り出された後で再検査されなければならないことです。この対策は"非可逆"インデックス演算子をサポートします。再検査はスキャン条件のみに拡大適用されることに注意してください。部分インデックス述語（もしあれば）は`amgettupple`呼び出し元で決して再検査されません。

そのインデックスがインデックスオンリースキャンをサポートしている場合(その場合は`amcanreturn`により真が返されます)、そのアクセスメソッドは`scan->xs_want_itup`も確認し、それが真の場合、`scan->xs_itupdesc`タプルディスクリプタとともに`scan->xs_itup`に格納された`IndexTuple`ポインタの形式でインデックスエントリに対する元のインデックスされたデータが返されます。(ポインタによって参照されているデータはアクセスメソッドのもとで管理されます。データは少なくとも`amgettupple`、`amrescan`または`amendscan`によってスキャンされるまでよい状態を保たなくてはなりません。)

`amgettupple`関数は、アクセスメソッドが"単純"インデックススキャンをサポートするときのみ提供される必要があります。そうでなければ、`pg_am`行の`amgettupple`フィールドはゼロに設定されなければなりません。

```
int64
amgetbitmap (IndexScanDesc scan,
             TIDBitmap *tbm);
```

指定されたスキャンから全てのタプルを取り出し、呼び出し側が提供する`TIDBitmap`にそれらを付加します（つまり、既にビットマップ内にある集合とタプルIDの集合とのORを取ります）。取り出されたタプル数が返されます（例えばいくつかのAMは重複を検出しますので、これは単なる概算です）。タプルIDをビットマップに挿入する間、`amgetbitmap`は特定のタプルIDに必要なスキャン条件の再検査を示すことが可能です。これは`amgettupple`の`xs_recheck`出力パラメータに類似しています。注意：現在の実装においてこの機能の提供はビットマップそのものの非可逆格納を提供するのに結びついていて、したがって呼び出し側はスキャン条件と部分インデックスの述部（存在すれば）を再検査可能なタプルに対して再検査します。とは言っても常に正しいとは限りません。`amgetmulti`および`amgettupple`を同じインデックススキャン内で使用することはできません。[項52.3](#)で説明した通り、`amgetbitmap`を使用する場合には他にも制限があります。

`amgetbitmap`関数はアクセスメソッドが"ビットマップ"インデックススキャンをサポートしている場合のみ必要です。そうでなければ、`pg_am`行の中の`amgetbitmap`フィールドはゼロに設定されなければなりません。

```
void
amendscan (IndexScanDesc scan);
```

スキャンを停止し、リソースを解放します。`scan`構造体自体は解放すべきではありません。アクセスメソッドで内部的に取られたロックやピンは解放しなければなりません。

```
void
ammarkpos (IndexScanDesc scan);
```

現在のスキャン位置を記録します。アクセスメソッドは1スキャン当たり1つの記録済みスキャンのみをサポートしなければなりません。

```
void
amrestrpos (IndexScanDesc scan);
```

もっとも最近に記録された位置にスキャンを戻します。

簡便性のために、インデックスアクセスメソッド関数の`pg_proc`項目は、正確な引数の数を示さなければなりません。しかし、それらはすべて`internal`型として宣言します。（引数のほとんどがSQLでは未知の型を持つため、ユーザがこうした関数を直接呼び出すことを防ぐことがこの理由です。）戻り値の型は、`void`、`internal`、`boolean`のいずれかで適切に宣言されます。唯一の例外は`amoptions`です。これは、`text[]`および`bool`を取り`bytea`を返すように正しく宣言しなければなりません。この規定により、クライアントコードは`amoptions`を実行してオプションの設定の有効性を検査することができます。

[前のページ](#)

インデックス用のカタログ項目

[ホーム](#)
[上に戻る](#)

[次のページ](#)
インデックススキャン

52.3. インデックススキャン

インデックススキャンでは、スキャンキーに一致するものと示したすべてのタプルのTIDを繰り返すことに関する責任をインデックスアクセスメソッドが持ちます。アクセスメソッドには、実際のインデックスの親テーブルからのタプルの取り出しやタプルがスキャンの時間制限試験や他の条件を通過したかどうかの決定は含まれません。

スキャンキーは、*index_key operator constant*という形式のWHERE句の内部的表現です。ここで、*index_key*は、インデックス列の1つで、*operator*はインデックス列に関連した演算子族のメンバの1つです。インデックススキャンは、暗黙的にAND演算される0個以上のスキャンキーを持ちます。返されるタプルは指定された条件を満たすものと想定されます。

アクセスメソッドはインデックスがある特定の問い合わせに対し非可逆、または再検査を要求するかどうかを報告することができます。これは、インデックススキャンがスキャンキーを満たすすべての項目と、それに加えて、満たさない可能性のある項目を返すことを意味します。コアシステムのインデックススキャン機構はヒープタプルに対し、本当に選択されるべきかどうかを検証するためにその演算子をインデックス条件に再度適用します。再検査オプションが指定されない場合、インデックススキャンは一致する項目の集合を返さなければなりません。

確実に、指定されたスキャンキーすべてに一致するもののみをすべて正しく見つけ出すことは、完全にアクセスメソッドの責任であることに注意してください。また、コアシステムは、冗長かどうかや矛盾するかどうかを決定するための意味的な解析を行わず、単にインデックスキーと演算子族に一致するWHERE句をすべて渡します。例えば、WHERE *x* > 4 AND *x* > 14があり、*x*がB-treeインデックス列であったとすると、これは、B-tree *amrescan*関数に任されて、最初のスキャンキーが冗長であり、無視することができることが認知されます。*amrescan*における前処理の必要性は、インデックスアクセスメソッドがスキャンキーを"正規化"形式にする必要があるかどうか依存します。

一部のアクセスメソッドは、他では行いませんが、十分に定義された順序でインデックス項目を返します。アクセスメソッドが出力の順序付けをサポートできるようにする方法は、実質2種類存在します。

- 常にそのデータ (btreeなど) の自然な順序で項目を返すアクセスメソッドはpg_am.amcanorderを真に設定しなければなりません。現在、こうしたアクセスメソッドは、その等価性と順序付け演算子でbtree互換の戦略番号を使用しなければなりません。
- 順序付け演算子をサポートするアクセスメソッドはpg_am.amcanorderbyopを真に設定しなければなりません。これは、インデックスがORDER BY *index_key operator constant*を満たす順序で項目を返すことができることを示します。前述の通り、この形式のスキャン修飾子をamrescanに渡すことができます。

amgettupple関数はdirection引数を持ちます。これはForwardScanDirection (通常の場合) またはBackwardScanDirectionのいずれかを取ることができます。amrescan後の最初の呼び出しがBackwardScanDirectionを指定していた場合、一致したインデックス項目は通常の前から後ろという方向ではなく、後ろから前という方向でスキャンされます。そのため、amgettuppleは通常ならばインデックス内の最初に一致したタプルを返すところですが、最後に一致したタプルを返さなければなりません。(これはamcanorderが真に設定されたアクセスメソッドでのみ発生します。) 最初の呼び出しの後、amgettuppleは、最も最近に返された項目からどちらの方向にスキャンを進めるかを準備しなければなりません。(しかしpg_am.amcanbackwardが偽であれば、引き続きすべての呼び出しは最初のものと同じ方向を持ちます。)

順序付けされたスキャンを提供するアクセスメソッドはスキャン内位置の"記録"をサポートしなければならず、また、後でその記録された位置に戻ることをサポートしなければなりません。同じ位置が複数回記録されるかもしれません。しかし、スキャン内の1つの位置のみを記録する必要があります。新しいammarkpos呼び出しにより前回記録された位置は上書きされます。順序付けされたスキャンをサポートしないアクセスメソッドであっても、pg_am内で記録用の関数と位置に戻るための関数を提供しなければなりません、呼び出されたとしてもエラーを返すだけで十分です。

スキャン位置と記録された位置 (もしあれば) の両方は、インデックス内の同時挿入や削除という観点における一貫性を保持しなければなりません。スキャンが始まった時に存在していた場合、項目を見つけたスキャンが新しく挿入された項目を返さなかったとしても問題ありません。このような場合のスキャンでは、再スキャンやバックアップによって、あたかも最初の時点で返されたものとして項目が返されます。同様に、同時実行削除によってスキャンの結果に影響が出るかもしれません。重要なことは、挿入や削除によって、その項目自体が挿入・削除されていない項目がスキャンで失われたり二重になったりすることが起こらないという点です。

インデックスが設定された列値がインデックスに格納されている(かつ、不可逆表現ではない)場合、ヒープタプルのTIDではなくインデックスに格納された実際のデータを返すindex-only scansをサポートするのに有用です。これは、可視性マップによって可視化されたページ上のTIDが判断できる場合にのみ動作します。判断できない場合はMVCCを確認するためにヒープタプルにアクセスしなくてはなりません。しかしその動作はアクセスメソッドでは考慮されていません。

amgettuppleを使用する代わりに、amgetbitmapを使用して、一回の呼び出しですべてのタプルを取り出してインデックススキャンを行うことができます。これはアクセスメソッド内でのロック/ロック解除という過程を防ぐことができますので、amgettuppleよりもかなり効率的です。実際には、amgetbitmapはamgettupple呼び出しを繰り返すことと同じ効果を持つはず

ですが、物事を単純化するために複数の制限を加えています。まず第一に、`amgetbitmap`は一回ですべてのタプルを返し、スキャン位置の記録と位置戻しをサポートしません。第二に、特定の順序付けをまったく持たないビットマップの中にタプルが返されます。これは`amgetbitmap`が`direction`引数を取らない理由です。（順序付け演算子はこのようなスキャンでは決して与えられません。）また、`amgetbitmap`によるインデックスオンリースキャンは提供されていません。なぜなら、インデックスタブルの内容を返す手段がないからです。最後に、`amgetbitmap`は返されたタプルに関し、[項52.4](#)に記載した意味でのロックを保証しません。

アクセスメソッドの内部実装がどちらか片方のAPIにそぐわない場合、`amgettupple`を実装せず`amgetbitmap`のみを実装、またはその逆も許されていることに注意してください。

52.4. インデックスのロック処理に関する検討

インデックスアクセスメソッドは、複数のプロセスによるインデックスの同時更新を取り扱えなければなりません。PostgreSQL コアシステムはインデックススキャン中にインデックスに対して `AccessShareLock` を獲得します。また、(通常の `VACUUM` を含む) インデックスの更新中に `RowExclusiveLock` を獲得します。これらの種類のロックは競合しませんので、アクセスメソッドは必要になるかもしれない粒度の細かなロック処理に関して責任を持ちます。インデックスの生成、破棄、`REINDEX` 時にインデックス全体に対する排他ロックが獲得されます。

同時更新をサポートするインデックス種類を構築することは通常、必要な動作について広範かつ微細にわたる解析が必要です。B-tree および ハッシュインデックス種類では、`src/backend/access/nbtree/README` と `src/backend/access/hash/README` にある設計に関する決定事項を読むことができます。

インデックス自身の内部的な一貫性要求の他に、同時実行更新には、親テーブル (ヒープ) とインデックス間の一貫性に関する問題が発生します。PostgreSQL はヒープへのアクセスおよび更新とインデックスへのアクセスおよび更新を分離していますので、インデックスとヒープとの間の一貫性が無くなる間隔が存在します。以下の規則でこうした問題を扱います。

- 新しいヒープ項目はインデックス項目を作成する前に作成されます。(このため、同時実行インデックススキャンはヒープエントリを確認する時によく失敗します。インデックスの読み取りは、未コミットの行を対象としませんので問題ありません。しかし、[項52.5](#)を参照してください。)
- ヒープエントリが (`VACUUM` によって) 削除される時、これに対するすべてのインデックス項目が先に削除されます。
- インデックススキャンは、最後に `amgettupple` が返した項目を保持するインデックスページ上のピンを管理しなければなりません。また、`ambulkdelete` は、他のバックエンドがピンを持つページから項目を削除することはできません。この規則の必要性については後で説明します。

3 番目の規則がないと、`VACUUM` によって削除される直前に、インデックス読み取りがインデックス項目を見つけ、そして、`VACUUM` によって削除された後に対応するヒープ項目に達する可能性があります。空の項目スロットは `heap_fetch()` で無視されますので、これは読み取りが達した時にその項目番号が未使用である場合でも大きな問題は起こりません。しかし、第三のバックエンドがすでにその項目スロットを他のものに再使用した場合はどうなるでしょうか？ そのスロット内の新しいものが、スナップショット試験を通過するには新しすぎることが確実ですので、MVCC に則ったスナップショットを使用する場合は問題ありません。しかし、MVCC に則らないスナップショット (`SnapshotNow` など) では、実際にはスキャンキーに合わない行を受け付け、返す可能性があります。すべての場合においてヒープ行に対しスキャンキーの再検査を行うことを必須とすることで、こうした状況から保護することができますが、これは高価すぎます。代わりに、読み取りがまだ一致するヒープ項目へのインデックス項目の "作業中" であることを示す代理として、インデックスページに対するピンを使用します。このピンに対して `ambulkdelete` がブロックするようにすることで、読み取りの作業が終わる前に `VACUUM` がそのヒープ項目を削除できないことを確実にします。実行時におけるこの対策のコストは小さく、実際に競合が発生するごく稀な場合にのみブロックするためのオーバーヘッドが加わります。

この対策は、インデックススキャンが "同期" していることを要求します。対応するインデックス項目のスキャンの後即座に各ヒープタプルを取り出さなければなりません。多くの理由のため、これは高価です。インデックスから多くの TID を収集し、少し後でのみヒープタプルにアクセスする "非同期" スキャンでは、必要なロック処理オーバーヘッドがかなり少なくなり、また、より効率的なヒープへのアクセスパターンを取ることができます。上の解析に従うと、MVCC に則らないスナップショットでは同期方式を使用しなければなりません、問い合わせが MVCC スナップショットを使用する場合は非同期スキャンを使用することができます。

`amgetbitmap` インデックススキャンでは、アクセスメソッドは返されるタプル上にインデックスピンをまったく保持しません。したがって、MVCC に則ったスナップショットでこうしたスキャンを使用することのみが安全です。

`ampredlocks` フラグが設定されていない場合、シリアライザブルトランザクション内でそのインデックスアクセスメソッドを使用するスキャンはいずれもインデックス全体に対するブロックしない述部ロックを獲得します。これは、同時実行のシリアライザブルトランザクションによるそのインデックスへの何らかのタプル挿入で、読み書きの競合が発生することがあります。同時実行のシリアライザブルトランザクションの集合の中で特定の読み書きの競合パターンが検知された場合、データの整合性を保護するためにこれらのトランザクションの 1 つはキャンセルされます。このフラグが設定されている場合、こうしたトランザクションのキャンセルの頻度を低減することになる、より粒度の細かな述部ロックをインデックスアクセスメソッドが実装していることを示します。

52.5. インデックス一意性検査

PostgreSQLは、SQLの一意性制約を一意性インデックスを使用して強制します。このインデックスでは、同一キーに対し複数の項目を許しません。この機能をサポートするアクセスメソッドは`pg_am.amcanunique`を真に設定します。（現時点ではb-treeのみがこれをサポートします。）

MVCCのため、インデックス内に物理的に重複した項目が存在することができるのが常に必要です。これらの項目は1つの論理的な行の連続的なバージョンを示します。実際に強制させたい動作は、MVCCスナップショットが同じインデックスキーを持つ行を2つ含めないことです。一意性インデックスに新しい行を挿入する時に検査しなければならない状況を以下のように分割することができます。

- 競合する有効な行が現在のトランザクションで削除された場合は問題ありません。（具体的には、UPDATEは常に新しいバージョンを挿入する前に古い行バージョンを削除します。これによりキーを変更することなく行をUPDATEすることができます。）
- 競合する行が未コミットのトランザクションで挿入された場合、挿入しようとしている方はトランザクションのコミットが分かるまで待機しなければなりません。ロールバックした場合は競合しません。競合する行が削除されずにコミットした場合、一意性違反となります。（具体的には、他のトランザクションの終了をただ待機し、終了後に可視性の検査を完全に再実行します。）
- 同様に、競合する有効な行が未コミットのトランザクションで削除された場合、挿入しようとしている方はトランザクションのコミットまたはアバートを待機しなければならず、その後、試験を繰り返します。

さらに、上記規則に従った一意性違反を報告する直前に、アクセスメソッドは挿入される行の有効性を再度検査しなければなりません。もし、無効なコミットであれば、違反を報告してはいけません。（現在のトランザクションによって作成された通常の行の挿入という状況では、これは発生することはありません。しかし、これは`CREATE UNIQUE INDEX CONCURRENTLY`中に発生することがあります。）

インデックスアクセスメソッドにこうした試験を自身で行うことを要求します。これは、インデックスの内容に対して重複するキーを持つことを示している任意の行のコミット状態を検査するために、ヒープまでアクセスしなければならないことを意味します。これが醜くモジュール化されないことには疑う余地はありません。しかし、余計な作業を防ぐことができます。もし分離された探査を行ったとすると、新しいインデックス項目を挿入する場所を検索する時、競合する行に対するインデックス検索がどうしても繰り返されます。さらに、競合検査がインデックス行の挿入部分で統合されて行われない限り、競合状態を防ぐ明確な方法がありません。

一意性制約が遅延可能である場合はさらに複雑になります。新しい行向けのインデックス項目を挿入可能にする必要があります。しかし一意性違反エラーは文の終わりまたはそれ以降まで遅延されます。不要なインデックス検索の繰り返しを防ぐために、インデックスアクセスメソッドは初期の挿入の間に前座の一意性検査を行わなければなりません。これが現存するタプルとまったく競合がないことを示した場合、それで終了です。さもなければ、制約を強制する時に再検査を行うようスケジュールします。再検査の時点で対象のタプルと同じキーを持つ何らかの他のタプルが存在すると、エラーを報告しなければなりません。（この目的のために"存在する"は実際には"インデックス項目のHOTチェーン内に何らかのタプルが存在する"ことを意味します。）これを実装するために、`aminert`は以下のいずれかの値を持つ`checkUnique`パラメータを渡されます。

- `UNIQUE_CHECK_NO`は、一意性検査を行うことはない(これは一意性インデックスではない)ことを示します。
- `UNIQUE_CHECK_YES`は、上述の通り遅延がない一意性インデックスであり、一意性検査を即時に行わなければならないことを示します。
- `UNIQUE_CHECK_PARTIAL`は一意性制約が遅延可能であることを示します。PostgreSQLはこのモードを使用して、各行のインデックス項目を挿入します。このアクセスメソッドはインデックス内の重複する項目を許さなければなりません。そして`aminert`から偽を返すことで重複の可能性があることを報告しなければなりません。偽が返された行それぞれに対して、遅延再検査が予定されます。

アクセスメソッドは一意性制約違反となるかもしれない行を識別しなければなりません。しかし間違っただけを報告することはエラーではありません。これにより他のトランザクションを待つことなく検査を行うことができます。ここで報告された重複はエラーとして扱われず、後で再検査されます。再検査時には重複しなくなっている可能性があります。

- `UNIQUE_CHECK_EXISTING`は、一意性違反の可能性があるとして報告された行に対する遅延再検査であることを示します。これは`aminert`を呼び出すことで実装されますが、アクセスメソッドはこの場合に新しいインデックス項目を挿入してはいけません。インデックス項目はすでに存在します。それよりも、アクセスメソッドは他に存在するインデックス項目があるか検査する必要があります。もし存在し、対象の行もまだ存在する場合エラーを報告します。

`UNIQUE_CHECK_EXISTING`呼び出しでは、アクセスメソッドはさらに対象行が実際にインデックス内に既存の項目を持つか

検証し、もしなければエラーを報告することを推奨します。 `amininsert`に渡されるインデックスタプル値が再計算されているため勧めます。 インデックス定義に実際には不変ではない関数が含まれる場合、インデックスの間違った領域を検査してしまうかもしれません。 再検査にて対象行の存在を検査することで、元の挿入で使用されたものと同じタプル値をスキャンしていることを検証します。

[前のページ](#)

インデックスのロック処理に関する検討

[ホーム](#)
[上に戻る](#)

[次のページ](#)

インデックスコスト推定関数

52.6. インデックスコスト推定関数

`amcostestimate` 関数には、インデックスと共に使用できることが決まっている WHERE 句および ORDER BY 句のリストを含む、インデックススキャンの可能性を記述する情報が与えられます。この関数はインデックスにアクセスするコストの概算と WHERE 句の選択度（つまりインデックススキャンにて抽出される行の親テーブルにおける割合）を返さなくてはなりません。単純な場合だと、ほとんどすべてのコスト概算の作業は、オプティマイザの標準ルーチン呼び出すことで行われます。`amcostestimate` 関数を持つことの意味は、標準の概算を改善することができる場合に、インデックスアクセスメソッドがインデックス型固有の知識体系を提供することができるということです。

それぞれの `amcostestimate` 関数は以下のシグネチャを持たなければいけません。

```
void
amcostestimate (PlannerInfo *root,
                IndexPath *path,
                double loop_count,
                Cost *indexStartupCost,
                Cost *indexTotalCost,
                Selectivity *indexSelectivity,
                double *indexCorrelation);
```

最初の3つのパラメータは入力です。

`root`

処理されている問い合わせに関するプランナの情報。

`path`

インデックスアクセスパスが考慮されます。コストと選択性値を除くすべてのフィールドが有効です。

`loop_count`

コスト概算の算出対象となるインデックススキャンが繰り返された回数です。これは通常、ネストループジョインの内部で利用されるスキャンの回数よりも大きい値になります。コスト概算は1回のスキャンのための値であることに注意してください。`loop_count` がより大きい場合、複数のスキャンにより得られる効果をみるには十分な値といえるでしょう。

最後の4つのパラメータは参照渡し出力です。

`*indexStartupCost`

インデックスの起動処理にかかるコストに設定されます。

`*indexTotalCost`

インデックス処理の全体のコストに設定されます。

`*indexSelectivity`

インデックスの選択度に設定されます。

`*indexCorrelation`

インデックススキャンの順番と背後のテーブルの順番間の相関係数に設定されます。

コスト概算関数は、SQLやその他の手続き言語ではなく、C言語で書かれなければいけないことに注意してください。理由はプランナ/オプティマイザの内部データ構造にアクセスしなければいけないためです。

インデックスアクセスコストは `src/backend/optimizer/path/costsize.c` で使われる、逐次的なディスクブロックの取り出しには `seq_page_cost` のコストが、順不同の取り出しには `random_page_cost` のコストが、そして、1つのインデックス行の処理には通常 `cpu_index_tuple_cost` というコストがかかる、というパラメータで計算されなければなりません。さらに、インデックス処理（特に `indexquals` 自体の評価）の間に呼び出される比較演算すべてに対して、`cpu_operator_cost` に適当な係数をかけたコストがかかります。

アクセスコストは、インデックス自身のスキャンと関係するすべてのディスクとCPUコストも含むべきですが、インデックスで識別される親テーブルの行の処理や抽出にかかるコストは含めてはいけません。

"起動用コスト"は、最初の行を取り出し始めることができるようになる前に費やされなければならない総スキャンコストの一部です。ほとんどのインデックスでは、これはゼロとすることができます。しかし、高い起動用コストを持つインデックス種類ではこれを非ゼロにすることを勧めます。

`indexSelectivity`は、インデックススキャンの間に抽出される親テーブルの行の概算された割合として設定されるべきです。非可逆問い合わせの場合はこの値が、与えられた制約条件を実際に通過する行の割合よりも高くなるがよくあります。

`indexCorrelation`は、インデックスの順番とテーブルの順番の間の（-1.0から1.0までの間の値を取る）相関として設定されるべきです。この値は、メインテーブルから行を取り出すためのコスト概算を調整するために使用されます。

`loop_count`の値が1より大きい場合、戻り値はインデックスを利用した1回のスキャンを想定した平均値であるべきです。

コスト概算

典型的なコスト概算は次のように進められます。

- 1 11 与えられた制約条件に基づいて訪れられるメインテーブルの行の割合を概算して返します。インデックス型固有の知識体系を持たない場合、標準のオプティマイザの関数である`clauselist_selectivity()`を使用してください。

```
*indexSelectivity = clauselist_selectivity(root, path->indexquals,
                                           path->indexinfo->rel->reloid,
                                           JOIN_INNER, NULL);
```

- 1 11 スキャン中に訪れられるインデックスの行数を概算します。多くのインデックス種類では、これは`indexSelectivity`とインデックスの中にある行数を掛けたものと等しいですが、それより多い場合もあります。（ページおよび行内のインデックスのサイズは`path->indexinfo`構造体から得ることができることに注意してください。）
- 1 11 スキャン中に抽出されるインデックスページ数を概算します。これは単に`indexSelectivity`にページ内のインデックスのサイズを掛けたものになるでしょう。
- 1 11 インデックスアクセスコストを計算します。汎用的な概算においては以下のように行うでしょう。

```
/*
 * Our generic assumption is that the index pages will be read
 * sequentially, so they cost seq_page_cost each, not random_page_cost.
 * Also, we charge for evaluation of the indexquals at each index row.
 * All the costs are assumed to be paid incrementally during the scan.
 */
cost_qual_eval(&index_qual_cost, path->indexquals, root);
*indexStartupCost = index_qual_cost.startup;
*indexTotalCost = seq_page_cost * numIndexPages +
    (cpu_index_tuple_cost + index_qual_cost.per_tuple) * numIndexTuples;
```

しかし、上では繰り返されるインデックススキャンにかかるインデックス読み込みについて減価償却を考慮していません。

- 1 11 インデックスの相関を概算します。1つのフィールドに対する単純な順番のインデックスでは、これは`pg_statistic`から入手することができます。相関が未知の場合、概算を用心深く考えるとゼロ（無相関）となります。

コスト概算関数の例は`src/backend/utils/adt/selfuncs.c`にあります。

[前のページ](#)

インデックス一意性検査

[ホーム](#)

[上に戻る](#)

[次のページ](#)

GiSTインデックス

第 53 章 GiST インデックス

目次

- 53.1. [序文](#)
- 53.2. [拡張性](#)
- 53.3. [実装](#)
- 53.4. [例](#)

53.1. 序文

GiSTは汎用検索ツリー（Generalized Search Tree）を表します。これは、均衡な、ツリー構造のアクセスメソッドで、任意のインデックスの枠組みを実装する基本的なテンプレートとして動作します。 B-tree、R-treeやその他多くのインデックスの枠組みをGiSTで実装することができます。

GiSTの利点の1つは、データベースの専門家ではなく、データ型分野の専門家によって、適切なアクセスメソッドで独自のデータ型を開発することができることです。

ここで示す情報の一部は、カリフォルニア大学バークレイ校のGiSTインデックスプロジェクト、[web site](#)およびMarcel Kornackerの論文、[Access Methods for Next-Generation Database Systems](#)から派生したものです。 PostgreSQLにおけるGiSTの実装は、主に、Teodor SigaevとOleg Bartunovによって保守されています。 そして、彼らの[ウェブサイト](#)にも多くの情報があります。

53.2. 拡張性

伝統的に、新しいインデックスメソッドの実装は、非常に難しい作業を意味していました。 ロックマネージャやログ先行書き込みなどデータベースの内部動作を理解する必要がありました。 GiSTインタフェースは高度に抽象化されており、アクセスメソッドの実装者には、アクセスするデータ型のセマンティックスのみの実装を要求します。 GiST層自身が同時実行性、ログ処理、ツリー構造の検索処理に関する注意を行います。

この拡張性と、他の、扱うことができるデータを対象とした標準検索ツリーの拡張性とを混同すべきではありません。 例えば、PostgreSQLは拡張可能なB-treeとハッシュインデックスをサポートしています。 これは、PostgreSQLを使用して、任意のデータ型に対するB-treeやハッシュを構築することができることを意味します。 しかし、B-treeは範囲述語 (<、=、>) のみをサポートし、ハッシュインデックスは等価性問い合わせのみをサポートします。

ですから、PostgreSQLのB-treeで例えば画像群をインデックス付けする場合、"画像xは画像yと同じか"、"画像xは画像yより小さいか"、"画像xは画像yより大きいか"といった問い合わせのみ発行することができます。 この文脈でどのように"同じか"や"より小さいか"、"より大きいか"を定義するかに依存して、これが有意なこともあるでしょう。 しかし、GiSTを基にしたインデックスを使用すれば、問題分野に特化した、おそらくは、"馬の画像を全て見つけたい"、"露出オーバーの写真をすべて見つけたい"といった質問に答えられる手段を作成することができます。

GiSTアクセスメソッドを有効にし、実行するために行なわなければならないことは、ツリーのキーの動作を定義する、複数のユーザ定義のメソッドを実装することです。 当然ながら、これらのメソッドは手の込んだ問い合わせをサポートするためかなり意匠を凝らす必要があります。 しかし、すべての標準的な問い合わせ (B-treeやR-treeなど) ではこれらは、相対的に見てごく簡単です。 まとめると、GiSTは汎用性、コード再利用、整理されたインタフェースと拡張性を兼ね備えたものです。

GiST用の演算子クラスが提供しなければならない7つのメソッドを以下に示します。 8番目は省略可能です。 インデックスの正確性は、確実にsame、consistent、unionメソッドを適切に実装することです。 一方、インデックスの効率 (容量と速度) はpenaltyとpicksplitメソッドに依存します。 残る2つの基本メソッドはcompressとdecompress ですが、これによりインデックスはインデックス付けするデータと異なるデータ型のツリーデータを内部で持つことができますようになります。 リーフはインデックス付けするデータ型となりますが、他のツリーノードは何らかのC構造体を取ることができます。 (しかしここでもPostgreSQLのデータ型規約に従わなければなりません。 容量が可変のデータに関してはvarlenaを参照してください。) ツリーの内部データ型がSQLレベルで存在する場合、CREATE OPERATOR CLASS コマンドのSTORAGEオプションを使用することができます。 省略可能な8番目のメソッドはdistanceです。 これは演算子クラスに順序付けスキャン (近傍検索) をサポートさせた場合に必要なです。

consistent

インデックス項目_pと問い合わせ値_qが与えられると、この関数はインデックス項目が問い合わせと"一貫性"があるかどうか、つまり、述語"indexed_columnindexable_operator q"が、インデックス項目で表現される行に対して真かどうかを決定します。 リーフインデックス項目では、これはインデックス付条件の試験と等価です。 一方で内部ツリーノードでは、これはツリーノードで表現されるインデックスの副ツリーを走査する必要があるかどうかを決定します。 結果がtrueならば、recheckフラグも返されなければなりません。 これは、述語が確実に真なのか一部のみに真なのかを示します。 recheck = falseならば、インデックスは述語条件を正確に試験されたことを示し、recheck = trueならば行が単に一致候補であることを示します。 この場合、システムは自動的にindexable_operatorを実際の行値に対して評価し、本当に一致するかどうか確認します。 この規則により、GiSTはインデックス構造が非可逆な場合でもある場合でもサポートすることができます。

この関数のSQL宣言は以下のようになります。

```
CREATE OR REPLACE FUNCTION my_consistent(internal, data_type, smallint, oid, internal)
RETURNS bool
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

そして、Cモジュール内の対応するコードは以下のような骨格に従うことになります。

```
Datum          my_consistent(PG_FUNCTION_ARGS);
PG_FUNCTION_INFO_V1(my_consistent);

Datum
my_consistent(PG_FUNCTION_ARGS)
{
    GISTENTRY *entry = (GISTENTRY *) PG_GETARG_POINTER(0);
    data_type *query = PG_GETARG_DATA_TYPE_P(1);
    StrategyNumber strategy = (StrategyNumber) PG_GETARG_UINT16(2);
    /* Oid subtype = PG_GETARG_OID(3); */
    bool *recheck = (bool *) PG_GETARG_POINTER(4);
    data_type *key = DatumGetDataTypes(entry->key);
    bool      retval;
```

```

/*
 * strategy、keyおよびqueryの関数として戻り値を決定してください。
 *
 * インデックスツリー内のどこで呼びだされているかを知るためGIST_LEAF(entry)を使用してください。
 * それは、例えば = 演算子をサポートする場合重宝です
 * (非リーフノードにおける空でないunion()とリーフノードにおける等価性を検査することができます)。
 */

*recheck = true;          /* もしくは検査が正確であれば偽 */

PG_RETURN_BOOL(retval);
}

```

ここで、keyはインデックス要素であり、queryはインデックスに対して検索される値です。 StrategyNumberパラメータは、演算子クラスのどの演算子が適用されるかを示します。 これはCREATE OPERATOR CLASSコマンドの演算子番号の1つに一致します。 このクラスに含めた演算子が何かに応じて、queryのデータ型は変動することがあります。 しかし、上記骨格は変動しないものと考えられます。

union

このメソッドはツリー内の情報を統合します。 項目の集合が与えられると、この関数は与えられた項目すべてを表現するインデックス項目を新しく生成します。

この関数のSQL宣言は以下のようになります。

```

CREATE OR REPLACE FUNCTION my_union(internal, internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;

```

そして、Cモジュール内の対応するコードは以下のような骨格に従うことになります。

```

Datum          my_union(PG_FUNCTION_ARGS);
PG_FUNCTION_INFO_V1(my_union);

Datum
my_union(PG_FUNCTION_ARGS)
{
    GistEntryVector *entryvec = (GistEntryVector *) PG_GETARG_POINTER(0);
    GISTENTRY *ent = entryvec->vector;
    data_type *out,
                *tmp,
                *old;
    int         numranges,
                i = 0;

    numranges = entryvec->n;
    tmp = DatumGetDataTypes(ent[0].key);
    out = tmp;

    if (numranges == 1)
    {
        out = data_type_deep_copy(tmp);

        PG_RETURN_DATA_TYPE_P(out);
    }

    for (i = 1; i < numranges; i++)
    {
        old = out;
        tmp = DatumGetDataTypes(ent[i].key);
        out = my_union_implementation(out, tmp);
    }

    PG_RETURN_DATA_TYPE_P(out);
}

```

ご覧になったように、この骨格でunion(X, Y, Z) = union(union(X, Y), Z)であるようなデータ型を処理しています。 このGiSTサポートメソッドに適切なunionアルゴリズムを実装することで、このような場合以外のデータ型をサポートすることは非常に容易です。

unionの実装関数は新たにpalloc()されたメモリへのポインタを返さなければなりません。 単に入力されたものを返すことはできません。

compress

データ項目をインデックスページ内の物理的な格納に適した形式に変換します。

この関数のSQL宣言は以下のようになります。

```

CREATE OR REPLACE FUNCTION my_compress(internal)

```

1655/2500

```

RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;

```

そして、Cモジュール内の対応するコードは以下のような骨格に従うことになります。

```

Datum          my_compress(PG_FUNCTION_ARGS);
PG_FUNCTION_INFO_V1(my_compress);

Datum
my_compress(PG_FUNCTION_ARGS)
{
    GISTENTRY *entry = (GISTENTRY *) PG_GETARG_POINTER(0);
    GISTENTRY *retval;

    if (entry->leafkey)
    {
        /* 圧縮バージョンで entry->key を差し替え */
        compressed_data_type *compressed_data = malloc(sizeof(compressed_data_type));

        /* entry->key ... から *compressed_data を補填 */
        retval = malloc(sizeof(GISTENTRY));
        gistentryinit(retval, PointerGetDatum(compressed_data),
                      entry->rel, entry->page, entry->offset, FALSE);
    }
    else
    {
        /* 通常非リーフ項目に対して行うことはない */
        retval = entry;
    }

    PG_RETURN_POINTER(retval);
}

```

当然ながら`compressed_data_type`を、リーフノードを圧縮するために変換する特定の型に適合させなければなりません。

また必要に応じて、ここで`NULL`値の圧縮に関して注意をしなければなりません。例えば`gist_circle_compress`などでは(Datum) 0を格納します。

decompress

`compress`メソッドの逆です。 データ項目のインデックス表現から、データベースで扱うことができる書式に変換します。

この関数のSQL宣言は以下のようになります。

```

CREATE OR REPLACE FUNCTION my_decompress(internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;

```

そして、Cモジュール内の対応するコードは以下のような骨格に従うことになります。

```

Datum          my_decompress(PG_FUNCTION_ARGS);
PG_FUNCTION_INFO_V1(my_decompress);

Datum
my_decompress(PG_FUNCTION_ARGS)
{
    PG_RETURN_POINTER(PG_GETARG_POINTER(0));
}

```

上記骨格は、伸長を必要としない場合に適したものです。

penalty

新しい項目をツリーの特定の分岐点に挿入するための"コスト"を示す値を返します。 項目は、ツリー内で`penalty`が最小の経路に挿入されます。 `penalty`から返される値は非負でなければなりません。 負の値が返された場合、ゼロとして扱われます。

この関数のSQL宣言は以下のようになります。

```

CREATE OR REPLACE FUNCTION my_penalty(internal, internal, internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT; -- in some cases penalty functions need not be strict

```

そして、Cモジュール内の対応するコードは以下のような骨格に従うことになります。

```

Datum          my_penalty(PG_FUNCTION_ARGS);

```

```
PG_FUNCTION_INFO_V1(my_penalty);

Datum
my_penalty(PG_FUNCTION_ARGS)
{
    GISTENTRY *origentry = (GISTENTRY *) PG_GETARG_POINTER(0);
    GISTENTRY *newentry = (GISTENTRY *) PG_GETARG_POINTER(1);
    float *penalty = (float *) PG_GETARG_POINTER(2);
    data_type *orig = DatumGetDataType(origentry->key);
    data_type *new = DatumGetDataType(newentry->key);

    *penalty = my_penalty_implementation(orig, new);
    PG_RETURN_POINTER(penalty);
}
```

penalty関数は優れた性能のインデックスではきわめて重要です。これは、挿入の段階で新しい項目をツリーに追加する場所を決定する際にどの分岐に従うかを決定するために使用されます。問い合わせの際、インデックスのバランスが良ければ、検索が速くなります。

picksplit

インデックスページ分割が必要になった時、この関数は、ページ内のどの項目を古いページに残すか、および、どれを新しいページに移動するかを決定します。

この関数のSQL宣言は以下のようになります。

```
CREATE OR REPLACE FUNCTION my_picksplit(internal, internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

そして、Cモジュール内の対応するコードは以下のような骨格に従うことになります。

```
Datum
my_picksplit(PG_FUNCTION_ARGS);
PG_FUNCTION_INFO_V1(my_picksplit);

Datum
my_picksplit(PG_FUNCTION_ARGS)
{
    GistEntryVector *entryvec = (GistEntryVector *) PG_GETARG_POINTER(0);
    OffsetNumber maxoff = entryvec->n - 1;
    GISTENTRY *ent = entryvec->vector;
    GIST_SPLITVEC *v = (GIST_SPLITVEC *) PG_GETARG_POINTER(1);
    int i,
        nbytes;
    OffsetNumber *left,
        *right;
    data_type *tmp_union;
    data_type *unionL;
    data_type *unionR;
    GISTENTRY **raw_entryvec;

    maxoff = entryvec->n - 1;
    nbytes = (maxoff + 1) * sizeof(OffsetNumber);

    v->spl_left = (OffsetNumber *) palloc(nbytes);
    left = v->spl_left;
    v->spl_nleft = 0;

    v->spl_right = (OffsetNumber *) palloc(nbytes);
    right = v->spl_right;
    v->spl_nright = 0;

    unionL = NULL;
    unionR = NULL;

    /* 項目自体のベクタの初期化 */
    raw_entryvec = (GISTENTRY **) malloc(entryvec->n * sizeof(void *));
    for (i = FirstOffsetNumber; i <= maxoff; i = OffsetNumberNext(i))
        raw_entryvec[i] = &(entryvec->vector[i]);

    for (i = FirstOffsetNumber; i <= maxoff; i = OffsetNumberNext(i))
    {
        int real_index = raw_entryvec[i] - entryvec->vector;

        tmp_union = DatumGetDataType(entryvec->vector[real_index].key);
        Assert(tmp_union != NULL);

        /*
         * インデックス項目の格納場所を決定し、それに合わせてunionLとunionRを更新
         * します。v_spl_left もしくは v_spl_right のどちらかに項目を追加します。
         * カウンタに留意してください。
         */
    }
```

```

        if (my_choice_is_left(unionL, curl, unionR, curr))
        {
            if (unionL == NULL)
                unionL = tmp_union;
            else
                unionL = my_union_implementation(unionL, tmp_union);

            *left = real_index;
            ++left;
            ++(v->spl_nleft);
        }
        else
        {
            /*
             * Same on the right
             */
        }
    }

    v->spl_ldatum = DataTypeGetDatum(unionL);
    v->spl_rdatum = DataTypeGetDatum(unionR);
    PG_RETURN_POINTER(v);
}

```

penalty同様、picksplit関数も優れた性能のインデックスのためにきわめて重要です。penaltyとpicksplitの実装を適切に設計することが、性能が良いGiSTインデックスの実装を行うことにつながります。

same

2つのインデックス項目が同一の場合に真、さもなければ偽を返します。

この関数のSQL宣言は以下のようになります。

```

CREATE OR REPLACE FUNCTION my_same(internal, internal, internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;

```

そして、Cモジュール内の対応するコードは以下のような骨格に従うことになります。

```

Datum
my_same(PG_FUNCTION_ARGS);
PG_FUNCTION_INFO_V1(my_same);

Datum
my_same(PG_FUNCTION_ARGS)
{
    prefix_range *v1 = PG_GETARG_PREFIX_RANGE_P(0);
    prefix_range *v2 = PG_GETARG_PREFIX_RANGE_P(1);
    bool *result = (bool *) PG_GETARG_POINTER(2);

    *result = my_eq(v1, v2);
    PG_RETURN_POINTER(result);
}

```

歴史的な理由により、same関数は単純に論理値の結果を返しません。その代わり、3番目の引数で指定された場所にフラグを格納しなければなりません。

distance

インデックス項目_pと問い合わせ値_qを与えると、この関数は問い合わせ値からのインデックス項目の"距離"を決定します。この関数は、演算子クラスが何らかの順序付け演算子を含む場合には提供しなければなりません。順序付け演算子を使用する問い合わせは、まず最小の"距離"を持つインデックス項目を返すことで実装されます。このためこの結果は演算子の意味と一貫性を持たなければなりません。リーフインデックスノード項目では、結果は単にインデックス項目との距離を表します。内部ツリーノードでは、結果はすべての子項目が持つ中から最も最小の距離でなければなりません。

この関数のSQL宣言は以下のようにならなければなりません。

```

CREATE OR REPLACE FUNCTION my_distance(internal, data_type, smallint, oid)
RETURNS float8
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;

```

Cモジュールにおける対応するコードは次の骨格に従うことになります。

```

Datum
my_distance(PG_FUNCTION_ARGS);
PG_FUNCTION_INFO_V1(my_distance);

Datum
my_distance(PG_FUNCTION_ARGS)
{
    GISTENTRY *entry = (GISTENTRY *) PG_GETARG_POINTER(0);

```



```

data_type *query = PG_GETARG_DATA_TYPE_P(1);
StrategyNumber strategy = (StrategyNumber) PG_GETARG_UINT16(2);
/* Oid subtype = PG_GETARG_OID(3); */
data_type *key = DatumGetDataTypes(entry->key);
double      retval;

/*
 * determine return value as a function of strategy, key and query.
 */

PG_RETURN_FLOAT8(retval);
}

```

`distance`関数の引数は、`recheck`フラグが使用されない点を除いて、`consistent`関数の引数と同一です。タプルが返された後タプルを再度順序付けする手段がありませんので、リーフインデックス項目への距離は常に正確に決定される必要があります。内部ツリーノードへの距離の決定に関しては、その結果がすべての子の実際の距離よりも大きくならない限り、多少の概算は許されます。したがって、例えば、幾何学に関するアプリケーションでは、通常境界矩形への距離で十分です。結果値は何らかの有限の`float8`になります。（無限大やマイナス無限大はNULLなどの場合を扱うために内部的に使用されます。このため`distance`関数がこれらの値を返すことは勧められません。）

すべてのGiSTサポートメソッドは通常短期間有効なメモリコンテキストで呼び出されます。つまり`CurrentMemoryContext`は各タプルが処理された後にリセットされます。そのため`palloc`したすべてを`pfree`することに注意することはあまり重要ではありません。しかし、サポートメソッドで、繰り返される呼び出しを跨がってデータをキャッシュすることが有用な場合があります。このためには、`fcinfo->flinfo->fn_mcxt`の中で超期間有効なデータを割り当て、そこへのポインタを`fcinfo->flinfo->fn_extra`の中に保持してください。こうしたデータはインデックス操作（例えば1つのGiSTインデックススキャン、インデックス構築、インデックスタプルの挿入）の間有効です。`fn_extra`値を置き換える時に以前の値を`pfree`することに注意してください。さもないと操作の間リークが蓄積されます。

[前のページ](#)
序文

[ホーム](#)
[上に戻る](#)

[次のページ](#)
実装

53.3. 実装

53.3.1. バッファ付きGiST構築

すべてのタプルを単純に挿入することによって大規模なGiSTインデックスを構築することは、インデックスタプルがインデックス全体に分散し、インデックスがキャッシュに収まらない程大規模である場合、挿入の際に大量のランダムI/Oを行わなければなりませんので、低速になりがちです。バージョン9.2からPostgreSQLはバッファ処理に基づいてGiSTインデックスを構築するより効率的な方法をサポートします。これは、順序付けされていないデータ群に対して必要なランダムI/O数を劇的に減らします。十分に順序付けされたデータ群では、一度にわずかなページ数のみが新しいタプルを受け取り、そのためインデックス全体がキャッシュに収まらなくてもこれらのページがキャッシュ内に収まりますので、利点はより小さく、または利点がなくなります。

しかしバッファ付きインデックス構築は、CPUリソースを多少多く消費する`penalty`関数をより多く呼び出さなければなりません。またバッファ付き構築で使用するバッファは、最大作成されるインデックスと同じサイズまで、一時的にディスク容量を必要とします。バッファ処理は作成されるインデックスの品質にも、良くも悪くも、影響を与えます。この影響は、入力データの分布や演算子クラスの実装等、様々な要因に依存します。

デフォルトでは、インデックスのサイズが`effective_cache_size`に達した時にGiSTインデックス構築はバッファ処理方式に切り替わります。CREATE INDEXコマンドの`BUFFERING`パラメータによって、手作業で有効または無効にすることができます。デフォルトの動作は大抵の場合良好です。しかし、入力データが順序付けされている場合、バッファ処理を無効にすることで構築が多少高速になります。

53.4. 例

PostgreSQL のソース配布物には GiST を使用したインデックスメソッドの実装のいくつかの事例が含まれています。 コアシステムは現在全文検索サポート (`tsvector` と `tsquery` のインデックス付け) や組み込みの幾何データ型の一部に対する R-Tree と等価な機能を提供します (`src/backend/access/gist/gistproc.c` を参照してください)。 以下の `contrib` モジュールも同時に GiST 演算子クラスを含みます。

`btree_gist`

いくつかのデータ型に対する B-tree 等価機能

`cube`

多次元の立方体用のインデックス

`hstore`

(キー、値) の組み合わせを格納するモジュール

`intarray`

int4 値の 1 次元配列用の RD-Tree

`ltree`

疑似ツリー構造用のインデックス

`pg_trgm`

トリグラム一致を使用したテキストの類似性

`seg`

"浮動小数点範囲" のインデックス

[前のページ](#)
実装

[ホーム](#)
[上に戻る](#)

[次のページ](#)
SP-GiST インデックス

第 54 章 SP-GiST インデックス

目次

54.1. [はじめに](#)

54.2. [拡張性](#)

54.3. [実装](#)

54.3.1. [SP-GiST の制限](#)

54.3.2. [ノードラベルのない SP-GiST](#)

54.3.3. ["All-the-same" 内部タプル](#)

54.4. [例](#)

[前のページ](#)

例

[ホーム](#)
[上に戻る](#)

[次のページ](#)
[はじめに](#)

54.1. はじめに

SP-GiSTは、空間分割された(Space-Partitioned)GiSTを短縮した語です。 SP-GiSTは分割された探索木をサポートし、四分木、kd木、接尾辞木(接尾辞トライ木)など広範にわたる様々な非平衡データ構造の開発を可能にします。 これらの構造に共通の特徴は、それらが探索空間を繰り返し小さな領域に分割し、その領域の大きさが必ずしも等しくない、ということです。 分割規則によく適合した検索は非常に高速になります。

これらのよく使われるデータ構造は、元々はメモリ内での利用のために開発されたものでした。 主記憶上では、それらは通常、ポインタにより接続され、動的に割り当てられるノードの集合として設計されます。 このようなポインタのチェーンは長くなりがちで、非常に多くのディスクアクセスが必要となるため、ディスク上に直接格納するには適しません。 これとは反対に、データベースのデータ構造は、I/Oを最小化する、大きな論理出力数を持つべきです。 SP-GiSTによって解決される困難とは、探索木のノードをディスクのページにマップするときに、多数のノードを通り抜ける場合であっても、探索ではごく少数のディスクページにしかアクセスしないですむようにすることです。

GiSTと同じく、SP-GiSTは適切なアクセス方法のある独自のデータ型の開発を可能にするためのもので、データベースのエキスパートよりもむしろ、そのデータ型の領域のエキスパートによる開発を可能にします。

ここで記述する情報の一部はPurdue大学のSP-GiSTインデックスプロジェクト[WEBサイト](#)によるものです。 PostgreSQLでのSP-GiSTの実装は、おもにTeodor SigaevとOleg Bartunovによって保守されており、詳しい情報は彼らの[WEBサイト](#)にあります。

54.2. 拡張性

SP-GiSTは高度に抽象化されたインタフェースを提供します。アクセスメソッドの開発者は特定のデータ型専用のメソッドだけを開発する必要があります。SP-GiSTのコアは効率的なディスクマッピングと木構造の探索を担当します。また、同時実行制御とログ出力も担当します。

SP-GiSTのツリーのリーフタプルは、インデックスの付けられた列と同じデータ型の値を含んでいます。ルートレベルにあるリーフタプルは、必ずインデックスが付けられた元のデータの値を含んでいます。より下のレベルのリーフタプルは、接尾辞など、圧縮された表現しか含んでいないかも知れません。この場合、演算子クラスのサポート関数が、内部タプルをリーフレベルまでたどりながら集める情報を使って元の値を再構築する必要があります。

内部タプルは、探索木の分岐点となるため、もっと複雑です。それぞれの内部タプルは1つ以上のノードの集合を含んでおり、ノードは類似のリーフ値のグループを表現します。ノードは下向きのリンクを含んでおり、これは下のレベルの別の内部タプルを指すか、あるいはすべて同じインデックスページ上に載っているリーフタプルの短いリストを指しています。それぞれのノードは、それを記述するlabelを持っています。例えば、接尾辞木では、ノードのラベルは文字列の値の次の文字にすることができます。省略可能ですが、内部タプルはそのすべてのメンバーを記述する接頭辞の値を持つことができます。接尾辞木では、これは表現される文字列に共通の接頭辞とすることができます。接頭辞の値は、必ずしも本当の接頭辞である必要はなく、演算子クラスが必要とする任意の値で良いです。例えば四分木では、その中心点を保持し、4つの象限をそこから相対的に測ることができます。そうすると、四分木の内部タプルはこの中心点の周りの象限に対応する4つのノードも含むことになるでしょう。

木構造のアルゴリズムには、現在のタプルのレベル(深さ)を知っていることが必要なものがあります。そこで、SP-GiSTのコアは、演算子クラスが木構造をたどって下がる時にレベル数の管理を可能にしています。また、必要であれば、表現される値を加算的に再構築することもサポートしています。

注意: SP-GiSTのコアのコードはnullエントリに対しても対応しています。SP-GiSTのインデックスはインデックス列がnullのエントリに対しても格納しますが、これはインデックスの演算子クラスのコードからは隠されているので、nullのインデックスエントリや検索条件が演算子クラスのメソッドに渡されることはありません。(SP-GiSTの演算子は厳格なのでNULL値について成功を返すことはできないと想定されています。)従って、ここではこれ以上、NULLについて議論しません。

SP-GiSTのインデックス演算子クラスが提供しなければならないユーザ定義メソッドが5つあります。5つのメソッドはいずれも2つのinternal引数を取り、1番目の引数はサポートメソッドの入力値を含むCの構造体へのポインタ、2番目の引数は出力値が置かれるCの構造体へのポインタという形式に従っています。メソッドのうち4つは、その結果がすべて出力構造体の中にあるので、単にvoidを返しますが、leaf_consistentは、さらにbooleanの結果を返します。メソッドは、その入力構造体のどのフィールドも変更してはいけません。どんな場合でも、出力構造体はユーザ定義メソッドを呼び出す前にゼロに初期化されます。

5つのユーザ定義メソッドは以下のとおりです。

config

接頭辞とノードラベルのデータ型のデータ型OIDを含め、インデックスの実装に関する静的情報を返します。

関数のSQL宣言は以下のようになります。

```
CREATE FUNCTION my_config(internal, internal) RETURNS void ...
```

1番目の引数はCのspgConfigIn構造体へのポインタで、関数の入力データを含みます。2番目の引数はCのspgConfigOut構造体へのポインタで、関数が結果のデータを入れます。

```
typedef struct spgConfigIn
{
    Oid      attType;          /* Data type to be indexed */
} spgConfigIn;

typedef struct spgConfigOut
{
    Oid      prefixType;       /* Data type of inner-tuple prefixes */
    Oid      labelType;        /* Data type of inner-tuple node labels */
    bool     canReturnData;    /* Opclass can reconstruct original data */
    bool     longValuesOK;     /* Opclass can cope with values > 1 page */
} spgConfigOut;
```

attTypeは多様のインデックス演算子クラスをサポートするために渡されます。通常の固定データ型の演算子クラスでは、それは常に同じ値を持っているので無視できます。

接頭辞を使わない演算子クラスでは、prefixTypeをVOIDOIDに設定することができます。同様に、ノードラベルを使わな

演算子クラスでは、`labelType`を`VOIDOID`に設定することができます。演算子クラスが、元々提供されていたインデックスの値を再構築できるときは、`canReturnData`を`true`にします。`attType`が可変長で、演算子クラスが接尾辞付けの繰り返しによって長い値を分割できるときにのみ、`longValuesOK`を`true`にします([項54.3.1](#)参照)。

choose

内部タプルに新しい値を挿入するときのメソッドを選択します。

関数のSQL宣言は以下のようになります。

```
CREATE FUNCTION my_choose(internal, internal) RETURNS void ...
```

1番目の引数はCの`spgChooseIn`構造体へのポインタで、関数の入力データを含みます。2番目の引数はCの`spgChooseOut`構造体へのポインタで、関数が結果のデータを入れます。

```
typedef struct spgChooseIn
{
    Datum        datum;           /* original datum to be indexed */
    Datum        leafDatum;       /* current datum to be stored at leaf */
    int          level;           /* current level (counting from zero) */

    /* Data from current inner tuple */
    bool         allTheSame;      /* tuple is marked all-the-same? */
    bool         hasPrefix;       /* tuple has a prefix? */
    Datum        prefixDatum;     /* if so, the prefix value */
    int          nNodes;          /* number of nodes in the inner tuple */
    Datum        *nodeLabels;     /* node label values (NULL if none) */
} spgChooseIn;

typedef enum spgChooseResultType
{
    spgMatchNode = 1,             /* descend into existing node */
    spgAddNode,                   /* add a node to the inner tuple */
    spgSplitTuple                 /* split inner tuple (change its prefix) */
} spgChooseResultType;

typedef struct spgChooseOut
{
    spgChooseResultType resultType; /* action code, see above */
    union
    {
        struct /* results for spgMatchNode */
        {
            int      nodeN;        /* descend to this node (index from 0) */
            int      levelAdd;     /* increment level by this much */
            Datum     restDatum;    /* new leaf datum */
        } matchNode;
        struct /* results for spgAddNode */
        {
            Datum     nodeLabel;    /* new node's label */
            int        nodeN;       /* where to insert it (index from 0) */
        } addNode;
        struct /* results for spgSplitTuple */
        {
            /* Info to form new inner tuple with one node */
            bool      prefixHasPrefix; /* tuple should have a prefix? */
            Datum     prefixPrefixDatum; /* if so, its value */
            Datum     nodeLabel;       /* node's label */

            /* Info to form new lower-level inner tuple with all old nodes */
            bool      postfixHasPrefix; /* tuple should have a prefix? */
            Datum     postfixPrefixDatum; /* if so, its value */
        } splitTuple;
    } result;
} spgChooseOut;
```

`datum`はインデックスに挿入される元のデータです。`leafDatum`は最初は`datum`と同じですが、`choose`あるいは`picksplit`メソッドがそれを変更すると、ツリーのより低いレベルで変更されることがあります。挿入の探索がリーフのページに到達したとき、`leafDatum`の現在値が、新しく作成されるリーフタプルに格納される値となります。`level`は、ルートレベルを0として、現在の内部タプルのレベルを示します。現在の内部タプルが複数の同等なノードを含むとして印を付けられているとき、`allTheSame`を`true`にします([項54.3.3](#)参照)。現在の内部タプルが接頭辞を含むとき、`hasPrefix`を`true`にします。このとき、`prefixDatum`がその値になります。`nNodes`は内部タプルが含む子ノードの数で、`nodeLabels`はそれらのラベル値の配列、あるいはラベルがなければNULLになります。

`choose`関数は、新しい値が既存の子ノードの1つとマッチするか、新しい子ノードを追加する必要があるか、あるいは新しい値がタプルの接頭辞と適合しないので内部タプルを分割してより制限のない接頭辞を作成する必要があるか、を決定することができます。

新しい値が既存の子ノードの1つにマッチしたときは、`resultType`を`spgMatchNode`にセットします。`nodeN`はノードの配列中のそのノードの番号(0から)にセットします。`levelAdd`は、そのノードをたどって下がる時に生じた`level`の増分

にセットします。あるいは演算子クラスがレベルを使っていなければ0のままにします。 `restDatum`は、演算子クラスがデータのあるレベルから次のレベルに変更しないのであれば、`datum`に等しくセットします。そうでなければ、次のレベルで`leafDatum`として使われる修正された値にセットします。

新しい子ノードを追加しなければならないときは、`resultType`を`spgAddNode`にセットします。 `nodeLabel`は、新しいノードで使われるラベルにセットし、`nodeN`はノードの配列中の挿入される場所のノードの番号(0から)にセットします。 ノードを追加した後で、`choose`関数を修正された内部タプルを使って再び呼び出しますが、このときは、`spgMatchNode`という結果になるはずで

新しい値がタプルの接頭辞と適合しないときは、`resultType`を`spgSplitTuple`にセットします。 このアクションは、すべての既存のノードを新しい低位の内部タプルに移動し、新しい低位の内部タプルにリンクする単一のノードを持つ新しいタプルで既存のタプルを置換します。 `prefixHasPrefix`は新しい上位のタプルが接頭辞を持つかどうかを示し、持つ場合には`prefixPrefixDatum`をその接頭辞の値にセットします。 インデックスに追加される新しい値を受け入れるため、新しい接頭辞の値は元のものよりも制限の緩いものになっている必要があり、また元の接頭辞より長くはなりません。

`nodeLabel`は、新しい低位の内部タプルを指し示すノードで使われるラベルにセットします。 `postfixHasPrefix`は、新しい低位のタプルが接頭辞を持つかどうかを示し、持つときには`postfixPrefixDatum`を接頭辞の値にセットします。 新しい低位に移動したタプルのノードのラベルを変更する機会も、子のインデックスのエントリを変更する機会もありませんから、これら2つの接頭辞と追加のラベルの組み合わせは、元の接頭辞と同じ意味を持つ必要があります。 ノードが分割された後で、`choose`を置換した内部タプルを使って再び呼び出します。 この呼び出しは、通常は`spgAddNode`という結果になります。というのは、分割のステップで追加されたノードのラベルは恐らく新しい値とマッチしないからです。 従って、その後、3回目の呼び出しでやっと`spgMatchNode`が返り、リーフレベルに下がる挿入が可能となります。

`picksplit`

リーフタプルの集合に対し、新しい内部タプルをどうやって作るかを決定します。

関数のSQL宣言は以下のようになります。

```
CREATE FUNCTION my_picksplit(internal, internal) RETURNS void ...
```

1番目の引数はCの`spgPickSplitIn`構造体へのポインタで、関数の入力データを含みます。 2番目の引数はCの`spgPickSplitOut`構造体へのポインタで、関数が結果のデータを入れます。

```
typedef struct spgPickSplitIn
{
    int          nTuples;          /* number of leaf tuples */
    Datum        *datums;          /* their datums (array of length nTuples) */
    int          level;           /* current level (counting from zero) */
} spgPickSplitIn;

typedef struct spgPickSplitOut
{
    bool          hasPrefix;       /* new inner tuple should have a prefix? */
    Datum        prefixDatum;     /* if so, its value */

    int          nNodes;          /* number of nodes for new inner tuple */
    Datum        *nodeLabels;     /* their labels (or NULL for no labels) */

    int          *mapTuplesToNodes; /* node index for each leaf tuple */
    Datum        *leafTupleDatums; /* datum to store in each new leaf tuple */
} spgPickSplitOut;
```

`nTuples`は入力されるリーフタプルの個数です。 `datums`はデータの値の配列です。 `level`はすべてのリーフタプルの現在のレベルで、これが新しい内部タプルのレベルになります。

`hasPrefix`は新しい内部タプルが接頭辞を持つかどうかを示し、持つ場合は`prefixDatum`を接頭辞の値にセットします。 `nNodes`は新しい内部タプルが含むノードの数を示し、`nodeLabels`はそのラベル値の配列にセットします。(ノードがラベルを必要としないときは、`nodeLabels`をNULLにセットします。詳細は[項54.3.2](#)を参照してください。)

`mapTuplesToNodes`は、それぞれのリーフタプルが割り当てられるノードの番号(0から)の配列にセットします。

`leafTupleDatums`は新しいリーフタプルに格納される値の配列にセットします(演算子クラスがデータのあるレベルから次のレベルに変更しなければこれらは入力の`datums`と同じになります)。

`picksplit`関数は、`nodeLabels`、`mapTuplesToNodes`、`leafTupleDatums`の配列について`palloc`しなければならないことに注意してください。

2つ以上のリーフタプルを与えた場合、`picksplit`関数はそれらを2つ以上のノードに分類すると予想されます。そうでなければ、リーフタプルを複数のページにまたがって分割するという、この操作の究極の目的を実現できないからです。

従って、`picksplit`がすべてのリーフタプルを同じノードに置くことになった場合には、SP-GiSTのコアのコードがその決定を覆して内部タプルを生成し、その中の複数の同一のラベルが付けられたノードに、リーフタプルが無作為に割り当てられます。そのようなタプルは、このことが発生したことを明示するため、`allTheSame`と印がつけられます。

`choose`関数と`inner_consistent`関数は、これらの内部タプルについて、適切な注意をして取り扱わなければならないません。詳細な情報は[項54.3.3](#)を参照してください。

config関数がlongValuesOKをtrueにセットし、1ページよりも大きな入力値を与える場合にのみ、picksplitを1つだけのリーフタプルに適用できます。この場合の操作の重要な点は、接頭辞をはがして、新しい、より短いリーフデータの値を生成することです。この呼出は、1ページに収まる短さのリーフデータが生成されるまで繰り返されます。詳細な情報は[項54.3.1](#)を参照してください。

inner_consistent

ツリーの探索でたどるべきノード(枝)の集合を返します。

関数のSQL宣言は以下のようになります。

```
CREATE FUNCTION my_inner_consistent(internal, internal) RETURNS void ...
```

1番目の引数はCのspgInnerConsistentIn構造体へのポインタで、関数の入力データを含みます。2番目の引数はCのspgInnerConsistentOut構造体へのポインタで、関数が結果のデータを入れます。

```
typedef struct spgInnerConsistentIn
{
    ScanKey      scankeys;          /* array of operators and comparison values */
    int          nkeys;             /* length of array */

    Datum        reconstructedValue; /* value reconstructed at parent */
    int          level;             /* current level (counting from zero) */
    bool         returnData;        /* original data must be returned? */

    /* Data from current inner tuple */
    bool         allTheSame;        /* tuple is marked all-the-same? */
    bool         hasPrefix;         /* tuple has a prefix? */
    Datum        prefixDatum;       /* if so, the prefix value */
    int          nNodes;            /* number of nodes in the inner tuple */
    Datum        *nodeLabels;       /* node label values (NULL if none) */
} spgInnerConsistentIn;

typedef struct spgInnerConsistentOut
{
    int          nNodes;            /* number of child nodes to be visited */
    int          *nodeNumbers;      /* their indexes in the node array */
    int          *levelAdds;        /* increment level by this much for each */
    Datum        *reconstructedValues; /* associated reconstructed values */
} spgInnerConsistentOut;
```

配列scankeysは長さがnkeysで、インデックス検索の条件を記述します。複数の条件はANDで結合されます。つまり、条件のすべてを満たすインデックスエントリのみが対象となります。(nkeysが0ならば、すべてのエントリが検索条件を満たすことになる、ということに注意してください。)通常、consistent関数では、配列のそれぞれのエントリのsk_strategyおよびsk_argumentフィールドのみが問題となります。これらのフィールドにはそれぞれインデックス付け可能な演算子と比較値が入ります。なお、比較値がNULLかどうかを確認するためにsk_flagsを検査する必要はありません。なぜならSP-GiSTのコアのコードがそのような条件を除外するからです。reconstructedValueは親タプルのために再構築された値で、ルートレベルの場合、あるいは親レベルのinner_consistent関数が値を返さなかった場合は(Datum) 0となります。levelは現在の内部タプルのレベルを、ルートレベルを0として数えたものです。returnDataは、この問い合わせで再構築されたデータが必要な場合にtrueとなりますが、これはconfig関数がcanReturnDataを確認した場合にのみ、そうなります。allTheSameは、現在の内部タプルに"all-the-same"の印が付いている場合にtrueになります。この場合、すべてのノードは(ラベルがあれば)同じラベルを持っていますから、そのすべてが問い合わせにマッチするか、いずれもマッチしないかのいずれかになります([項54.3.3](#)参照)。hasPrefixは現在の内部タプルが接頭辞を持っている場合にtrueとなり、このときprefixDatumがその値となります。nNodesは内部タプルが含む子ノードの数です。nodeLabelsはそれらのラベル値の配列で、ノードにラベルがないときはNULLになります。

nNodesは探索で訪れる必要のある子ノードの数にセットされなければなりません。また、nodeNumbersはそれらの番号の配列にセットされなければなりません。演算子クラスがレベルを監視しているときは、それぞれのノードへと下って訪れるときに必要なレベルの増分の配列をlevelAddsにセットします。(この増分はすべてのノードについて同じになることも多いですが、必ずしもそうなるとは限らないので配列が使われます。)値の再構築が必要なのは、訪れるそれぞれの子ノードについて再構築された値の配列をreconstructedValuesにセットします。再構築が必要でなければ、reconstructedValuesをNULLのままにします。inner_consistent関数は、nodeNumbers、levelAdds、reconstructedValuesの配列についてpallocしなければならないことに注意してください。

leaf_consistent

リーフタプルが問い合わせを満たす場合、trueを返します。

関数のSQL宣言は以下のようになります。

```
CREATE FUNCTION my_leaf_consistent(internal, internal) RETURNS bool ...
```

1番目の引数はCのspgLeafConsistentIn構造体へのポインタで、関数の入力データを含みます。2番目の引数はCのspgLeafConsistentOut構造体へのポインタで、関数が結果のデータを入れます。

```

typedef struct spgLeafConsistentIn
{
    ScanKey      scankeys;      /* array of operators and comparison values */
    int          nkeys;         /* length of array */

    Datum        reconstructedValue; /* value reconstructed at parent */
    int          level;          /* current level (counting from zero) */
    bool         returnData;     /* original data must be returned? */

    Datum        leafDatum;      /* datum in leaf tuple */
} spgLeafConsistentIn;

typedef struct spgLeafConsistentOut
{
    Datum        leafValue;      /* reconstructed original data, if any */
    bool         recheck;        /* set true if operator must be rechecked */
} spgLeafConsistentOut;

```

配列 `scankeys` は長さが `nkeys` で、インデックス探索の条件を記述します。複数の条件はANDで結合されます。つまり、条件のすべてを満たすインデックスエントリのみが対象となります。(`nkeys` が0ならば、すべてのエントリが検索条件を満たすことになる、ということに注意してください。) 通常、`consistent` 関数では、配列のそれぞれのエントリの `sk_strategy` および `sk_argument` フィールドのみが問題となります。これらのフィールドにはそれぞれインデックス付け可能な演算子と比較値が入ります。なお、比較値がNULLかどうかを確認するために `sk_flags` を検査する必要はありません。なぜならSP-GiSTのコアのコードがそのような条件を除外するからです。 `reconstructedValue` は親タブルのために再構築された値で、ルートレベルの場合、あるいは親レベルの `inner_consistent` 関数が値を返さなかった場合は (Datum) 0 となります。 `level` は現在のリーフタブルのレベルを、ルートレベルを0として数えたものです。 `returnData` は、この問い合わせで再構築されたデータが必要な場合に `true` となりますが、これは `config` 関数が `canReturnData` を確認した場合にのみ、そうなります。 `leafDatum` は現在のリーフタブルに格納されている鍵の値です。

この関数は、リーフタブルが問い合わせにマッチすれば `true` を返し、マッチしなければ `false` を返します。 `true` の場合、 `returnData` が `true` であれば、 `leafValue` は、このリーフタブルにインデックス付けするために元々提供された値にセットされなければなりません。また、マッチするかどうかが不確実で、マッチするかの確認のために実際のヒープタブルに演算子を再適用しなければならないときは、 `recheck` が `true` にセットされることがあります。

SP-GiSTのすべてのサポートメソッドは、通常は短期間有効なメモリコンテキスト内で呼び出されます。つまり、それぞれのタブルについて処理した後で `CurrentMemoryContext` はリセットされます。したがって、`palloc` したもののすべてについて `pfree` することを気にかけることはあまり重要ではありません。(`config` メソッドは例外で、メモリリークを避けるようにする必要があります。しかし、通常は `config` メソッドは、パラメータとして渡された構造体に定数を代入する以外、何もする必要がありません。)

インデックス付けされた列が照合可能なデータ型の場合、インデックスの照合は、標準的な `PG_GET_COLLATION()` の仕組みを使ってすべてのサポートメソッドに渡されます。

[前のページ](#)
はじめに

[ホーム](#)
[上に戻る](#)

[次のページ](#)
実装

54.3. 実装

この節では、SP-GiSTの演算子クラスを実装する人にとって知っているのと役に立つ、実装についての詳細とその他の秘訣について説明します。

54.3.1. SP-GiSTの制限

それぞれのリーフタプルおよび内部タプルは1つのインデックスページ内(デフォルトで8KB)に収まらなければなりません。従って、可変長のデータ型の値をインデックス付けするときは、長い値は接尾辞木のようなメソッドによってのみサポートされます。つまり、ツリーのそれぞれのレベルではページに収まる短さの接頭辞を含み、最後のリーフレベルでは、やはりページに収まる短さの接尾辞を含む、というようなものです。このようなことが発生する場合の対応の準備ができていない場合のみ、演算子クラスは`longValuesOK`をTRUEにセットするべきです。そうでなければ、SP-GiSTのコアは、インデックスページに収めるには大きすぎる値についてのインデックス付け要求を拒絶します。

同様に、内部タプルが大きくなりすぎてインデックスページに収まらない、ということにならないようにするのは、演算子クラスの責任です。これにより、1つの内部タプルで使うことができる子ノードの数、および接頭辞の値の最大サイズが制限されます。

内部タプルのノードがリーフタプルの集合を指しているとき、それらのタプルはすべて同じインデックスページ内になければならない、という制限もあります。(これは、シークの回数を減らし、そのようなタプルを一つにつなげるリンクに必要なスペースを減らす、という設計上の決定によるものです。)リーフタプルの集合が1ページに大きくなって1ページに収まらなくなると、分割が実行され、中間の内部タプルが挿入されます。これで問題を解決するためには、新しい内部タプルは、リーフの値の集合を2つ以上のノードのグループに分割しなければなりません。演算子クラスの`picksplit`関数がそれをするのに失敗したときは、SP-GiSTのコアは、[項54.3.3](#)に記述されている特別な手段に頼ることになります。

54.3.2. ノードラベルのないSP-GiST

木構造のアルゴリズムには、それぞれの内部タプルに対して固定された集合のノードを使うものがあります。例えば四分木では、内部タプルの中心点の周りの4つの象限に対応するちょうど4つのノードが必ずあります。このような場合、コードは典型的には数字を使ったノードで動作し、明示的なノードラベルは必要ありません。ノードラベルを使わない(そしてそれによりいくらかのスペースを節約する)ために、`picksplit`関数は`nodeLabels`配列としてNULLを返すことができます。この結果、その後の`choose`関数および`inner_consistent`関数の呼び出しにおいても`nodeLabels`はNULLになります。原則として、ノードラベルは同じインデックス中の一部の内部タプルに使い、他の内部タプルには省略する、ということができます。

ラベルのないノードを持つ内部タプルを処理するときに、`choose`が`spgAddNode`を返すのはエラーです。というのは、この場合、ノードの集合は固定されていると想定されるからです。また、`spgSplitTuple`のアクションでラベルのないノードを生成する用意はありません。というのは、`spgAddNode`のアクションも必要になると考えられるからです。

54.3.3. "All-the-same"内部タプル

`picksplit`が入力のリーフ値を少なくとも2つのノード分類に分割できなかった場合、SP-GiSTのコアは演算子クラスの`picksplit`関数の結果を無効にすることがあります。これが起きると、複数のノードを持つ新しい内部タプルが作成されます。それぞれのノードは、`picksplit`が一つのノードに付与したもの(あれば)と同じラベルを持ち、リーフ値はこれらの等価なノード間でランダムに分割されます。内部タプルには`allTheSame`のフラグがセットされ、`choose`関数および`inner_consistent`関数に対し、そのタプルが通常期待されるようなノードの集合を持っていないことを警告します。

`allTheSame`の処理において、`choose`の`spgMatchNode`という結果は、新しい値は等価なノードのどれに割り当てられても良い、という意味に解釈されます。コアのコードは入力された`nodeN`の値を無視し、(ツリーの平衡を保つために)ノードの1つにランダムに降りていきます。`choose`が`spgAddNode`を返すのはエラーです。というのは、そうするとすべてのノードが等価ではなくなるからです。挿入する値が既存のノードとマッチしない時は、`spgSplitTuple`のアクションを使わなければなりません。

`allTheSame`のタプルの処理において、すべてのノードは等価なので、`inner_consistent`関数は、インデックス検索を続けるためのターゲットとして、すべてのノードを返すか、ノードを1つも返さないかのいずれかであるべきです。このために、特殊ケースを扱うコードが必要になるかもしれませんし、必要ないかもしれません。それは、`inner_consistent`関数が、通常、ノードの意味についてどの程度のことを仮定しているかに依存します。

54.4. 例

PostgreSQLのソースコードの配布物にはSP-GiSTのインデックス演算子クラスの例がいくつか含まれています。 コアのシステムは現在、text列についての接尾辞木と点についての2種類の木(4分木とkd木)を提供しています。 コードを参照するには、`src/backend/access/spgist/`を調べて下さい。

第 55 章GIN インデックス

目次

55.1. [はじめに](#)

55.2. [拡張性](#)

55.3. [実装](#)

55.3.1. [GIN 高速更新手法](#)

55.3.2. [部分一致アルゴリズム](#)

55.4. [GIN の小技](#)

55.5. [制限](#)

55.6. [例](#)

55.1. はじめに

GINとは汎用転置インデックスを表します。GINは、インデックス対象の項目が複合型であり、インデックスにより取り扱われる問い合わせが複合型の項目内に存在する要素値に対して検索する必要がある場合を取り扱うために設計されました。例えば、項目が文書であり、問い合わせが特定の単語を含む文書を検索することがあり得ます。

ここでインデックス対象の複合型の値を参照するために項目という単語を使用します。また、キーという単語を要素値を参照するために使用します。GINは本質的に、項目の値ではなく、常にキーを格納し検索します。

GINインデックスは(キー、投稿されたリスト)の組み合わせの集合を格納します。ここで"投稿されたリスト"はキーに合う行IDの集合です。項目は1つ以上のキーを含むことができますので、同じ行IDが複数の投稿されたリストに現れることがあります。キー値はそれぞれ一度のみ格納されます。このためGINインデックスの容量は、同じキーが何度も現れる場合に非常に小さくなります。

GINインデックスは、GINアクセスメソッドが高速化対象の操作を把握する必要がないという意味で汎用化されています。その代わり、特定のデータ型に対して定義された独自の戦略を使用します。戦略は、インデックス付けされた項目と問い合わせ条件からキーを抽出する方法および問い合わせ内のいくつかのキー値を含む行が実際に問い合わせを満たすかどうかを決定する方法を定義します。

GINの利点の1つは、データベース専門家ではなくデータ型の分野における専門家により、適切なアクセスメソッドを持つ独自のデータ型を開発できるという点です。これはGiSTの使用とほぼ同じ利点です。

PostgreSQLにおけるGINの実装は、主にTeodor SigaevとOleg Bartunovにより保守されています。GINに関する情報は彼らの[webサイト](#)により多く記載されています。

55.2. 拡張性

GINインタフェースは高度に抽象化されています。アクセスメソッド実装者に要求されることは、アクセスするデータ型の意味を実装することだけです。GIN層自体が同時実行性、ログ処理、ツリー構造の検索処理に関する注意を行います。

GINアクセスメソッドを動作させるために取ることは、4つ（または5つ）のユーザ定義関数を実装することだけです。これは、ツリー内のキーの動作とキーとインデックス付けされる項目、インデックス可能な問い合わせ間の関係を定義します。すなわち、GINは、一般化、コード再利用、整理されたインタフェースによる拡張性を組み合わせます。

GIN用の演算子クラスが提供しなければならない4つのメソッドを示します。

```
int compare(Datum a, Datum b)
```

キー（インデックス付けされる項目ではありません）を比較し、0より小さい、0、または0より大きい整数を返します。それぞれ、最初のキーが2番目のキーより、小さい、等しい、または大きいことを示します。NULLキーがこの関数に渡されることはありません。

```
Datum *extractValue(Datum itemValue, int32 *nkeys, bool **nullFlags)
```

インデックス対象値に与えられる、pallocで割り当てられたキーの配列を返します。返されるキーの数は*nkeysに格納しなければなりません。キーのいずれかがNULLになるかもしれない場合、*nkeys論理値配列をpallocで割り当てそのアドレスを*nullFlagsに格納し、必要に応じてNULLフラグを設定してください。すべてのキーが非NULLであれば、*nullFlagsをNULL（初期値）のままにすることができます。項目がキーを含まない場合、戻り値はNULLになるかもしれません。

```
Datum *extractQuery(Datum query, int32 *nkeys, StrategyNumber n, bool **pmatch, Pointer **extra_data, bool **nullFlags, int32 *searchMode)
```

問い合わせ対象の値に与えられる、pallocで割り当てられたキーの配列を返します。つまり、queryはインデックス可能な演算子の右辺の値です。この左辺はインデックス対象の列です。nは演算子クラス内の演算子の戦略番号です（[項35.14.2](#)を参照）。extractQueryはしばしば、queryのデータ型とキー値を抽出するために使用しなければならないメソッドを決定するために、nを調べなければなりません。返されるキーの数を*nkeysに格納しなければなりません。キーのいずれかがNULLとなる可能性がある場合はまた、*nkeys個の論理値の配列をpallocで割り当て、*nullFlagsにそのアドレスを格納し、必要に応じてNULLフラグを設定してください。すべてのキーが非NULLならば*nullFlagsはヌル（初期値）のままにしておくことができます。queryがキーを含まない場合、戻り値をNULLにすることができます。

searchModeは出力引数です。これによりextractQueryは検索がどのように行われるかの詳細を指定することができます。*searchModeがGIN_SEARCH_MODE_DEFAULT（呼び出し前にこの値に初期化されます。）に設定された場合、返されるキーの少なくとも1つに一致する項目が合致候補とみなされます。*searchModeがGIN_SEARCH_MODE_INCLUDE_EMPTYに設定された場合、少なくとも1つの一致するキーを含む項目に加え、キーをまったく含まない項目が合致候補とみなされます。（このモードは例えば何のサブセットかを求める演算子を実装する際に有用です。）

*searchModeがGIN_SEARCH_MODE_ALLに設定された場合、返されるキーのいずれかに一致するかどうかは関係なく、インデックス内の非NULLの項目すべてが合致候とみなされます。（このモードは、基本的にインデックス全体のスキャン処理が必要ですので、他の2つの選択肢と比べてかなり低速になります。しかし境界条件を正確に実装するためにこれが必要になるかもしれません。おそらく、このモードを必要とする演算子はほとんどの場合、GIN演算子クラス向けに優れた候補ではありません。）このモードを設定するために使用する記号はaccess/gin.hで定義されています。

pmatchは部分一致が提供されている場合に使用する出力引数です。使用するには、extractQueryが*nkeys論理値の配列を割り当て、そのアドレスを*pmatchに格納しなければなりません。関連するキーが部分一致を必要とするとき、それぞれの配列要素は真に、そうでなければ偽に設定されなければなりません。*pmatchがNULLに設定されている場合、GINは部分一致が必要ないと想定します。呼び出し前に変数はNULLに初期化されますので、この引数は部分一致が提供されていない演算子クラスでは、単に無視できます。

extra_dataは、extractQueryがconsistentとcomparePartialメソッドに追加データを渡すことができるようにする出力引数です。使用するには、extractQueryが*nkeysポインタの配列を割り当て、そのアドレスを*extra_dataに格納し、そして望まれるものは何でも個別のポインタに格納しなければなりません。変数は呼び出し前にNULLに初期化されますので、追加データを必要としない演算子クラスでこの引数は単に無視できます。もし*extra_dataが設定されれば、配列全部がconsistentメソッドに、適切な要素がcomparePartialメソッドに渡されます。

```
bool consistent(bool check[], StrategyNumber n, Datum query, int32 nkeys, Pointer extra_data[], bool *recheck, Datum queryKeys[], bool nullFlags[])
```

インデックス付けられた項目が戦略番号nを持つ問い合わせ演算子を満たす（または、recheck印が返されたときはたぶん満たすかもしれない）場合に真を返します。GINは項目を明示的に格納しませんので、この関数はインデックス付けされた項目の値に直接アクセスすることができません。どちらかというと、この問い合わせから取り出される指定された問い合わせで現れるキー値に関する知識が利用できるものです。check配列は長さnkeysであり、このqueryデータに対して事

前に行われた`extractQuery`が返したキーの数と同じです。インデックス対象の項目が対応する問い合わせキーを持つ場合、`check`配列の各要素は真です。つまり、`(check[i] == TRUE)`の場合、`extractQuery`の結果配列の*i*番目のキーがインデックス対象項目内に存在します。元の`query`データは、`consistent`メソッドがそれを調査する必要がある場合に、渡されます。このため`queryKeys[]`および`nullFlags[]`は事前に`extractQuery`によって返されます。
`extra_data`は`extractQuery`により返された追加データ配列で、ない場合はNULLです。

`extractQuery`が`queryKeys[]`内でNULLキーを返す時、インデックス対象項目がNULLキーを含む場合は対応する`check[]`要素は真です、つまり、`check[]`の意味はIS NOT DISTINCT FROMのようなものです。`consistent`関数は、通常の値の合致とNULL合致との違いを通知する必要がある場合、対応する`nullFlags[]`要素を検査することができます。

成功の場合、`*recheck`はヒープタプルが問い合わせ演算子に対し再検査を必要とすれば真で、インデックス検査が的確であれば偽です。つまり、FALSEという戻り値はヒープタプルが問い合わせに合わないことを保証し、`*recheck`が付いたTRUEという戻り値はヒープタプルが問い合わせに一致する可能性があるため、それを取り出し、元のインデックス付けされた項目を直接問い合わせ演算子で評価することで再検査する必要があることを意味します。

省略可能ですが、GINに対する演算子クラスは第5のメソッドを提供します。

```
int comparePartial(Datum partial_key, Datum key, StrategyNumber n, Pointer extra_data)
```

問い合わせキーとインデックスキーの部分一致を比較します。符号が結果を示す整数が返ります。ゼロ未満はインデックスキーは問い合わせに一致しないが、インデックススキャンを続けるべきであることを示します。ゼロはインデックスキーが問い合わせに一致することを示します。ゼロより大きな値はこれ以上の一致はありえないためインデックススキャンを停止すべきであることを示します。スキャンをいつ停止するかを決めるためにセマンテックスが必要とされる場合、部分一致問い合わせを生成した演算子の戦略番号*n*が提供されます。また`extra_data`は`extractQuery`で作成される追加データ配列の対応する要素、もしなければNULLです。NULLキーがこの関数に渡されることはありません。

"部分一致"問い合わせをサポートするためには、演算子クラスは`comparePartial`メソッドを提供しなければなりません。またその`extractQuery`は、部分一致問い合わせであった時に`pmatch`パラメータを設定しなければなりません。詳細については[項55.3.2](#)を参照してください。

上記の各種Datum値の実データ型は、演算子クラスに依存して変動します。`extractValue`に渡される項目値は常に演算子クラスの入力型であり、キー値はすべてそのクラスのSTORAGE型でなければなりません。`extractQuery`および`consistent`に渡される`query`引数は、戦略番号によって識別されるクラスのメンバ演算子の右辺入力型として指定されたものになります。正しい型のキー値がそこから抽出できる限り、これは項目の型と同じである必要はありません。

[前のページ](#)
はじめに

[ホーム](#)
[上に戻る](#)

[次のページ](#)
実装

55.3. 実装

GINインデックスはキー全体に対するB-treeインデックスを持ちます。そのキーはそれぞれインデックス対象項目の要素（例えば配列のメンバ）であり、リーフページ内のタプルはそれぞれ、ヒープポインタのB-treeへのポインタ（"投稿されたツリー"）か、もしリストがキー値と共に単一インデックスタプルに合う程度十分に小さければヒープポインタの単純なリスト（"投稿されたリスト"）です。

PostgreSQL 9.1からNULLキー値をインデックスに含められるようになりました。またプレースホルダとしてのNULLが、NULLまたは`extractValue`によるとキーを含まないインデックス対象項目についてインデックスに含められます。これにより空の項目を見つけ出すための検索を行うことができます。

複数列に対するGINインデックスは複合型の値（列番号、キー値）全体について単一のB-treeを構築することで実装されます。異なる列に対するキー値は別の型となるかもしれません。

55.3.1. GIN高速更新手法

1つのヒープ行の挿入または更新によりインデックスへの挿入が多く発生するという、転置インデックスの本質的な性質のためGINインデックスの更新は低速になりがちです。（各キー用のヒープ行はインデックス付けされた項目から取り出されます。） PostgreSQL 8.4からGINは、新しいタプルを一時的なソートされていない、待機中の項目リストに挿入することにより、この作業の大部分を遅延させることができるようになりました。テーブルがバキュームされた時、または、待機中のリストが大きくなりすぎた（`work_mem`よりも大きくなった）時、初期のインデックス作成の際に使用されるものと同様の一括挿入技法を使用して、項目は主GINデータ構造に移動されます。これは、バキュームのオーバーヘッドが追加されることを考慮したとしても、GINインデックスの更新速度を著しく向上します。さらに、フォアグラウンドの問い合わせ処理ではなくバックグラウンド処理でこのオーバーヘッド作業を実行することができます。

この手法の大きな欠点は、検索時に通常のインデックス検索に加え待機中の項目リストのスキャンを行わなければならない点です。このため、待機中の項目リストが大きくなると検索が顕著に遅くなります。他の欠点は、ほとんどの更新は高速ですが、待機中の項目リストが"大きくなりすぎる"きっかけとなった更新は即時の整理処理を招くことになり、他の更新に比べ大きく低速になります。自動バキュームを適切に使用することで、これらの両方の問題を最小化することができます。

一貫した応答時間が更新速度より重要な場合、GINインデックスに対する`FASTUPDATE`格納パラメータを無効にすることにより、待機中の項目の使用を無効にすることができます。詳細は[CREATE INDEX](#)を参照してください。

55.3.2. 部分一致アルゴリズム

GINは"部分一致"問い合わせをサポートすることができます。この問い合わせは1つ以上のキーに正確に一致することは決定しませんが、キー値の合理的に狭い（`compare`サポートメソッドで決まるキーのソート順に従った）範囲内に一致する可能性があります。`extractQuery`は、正確に一致したキー値を返す代わりに、検索される範囲の下限となるキー値を返し、`pmatch`フラグを真に設定します。そしてキー範囲を`comparePartial`メソッドを使用して検索します。`comparePartial`は一致するインデックスキーではゼロを、一致しないが検索すべき範囲内にあればゼロ未満の値を、インデックスキーが一致可能な範囲を超えた場合はゼロより大きな値を返さなければなりません。

55.4. GINの小技

作成と挿入

各項目に対して多くのキーが挿入される可能性がありますので、GINインデックスへの挿入は低速になることがあります。ですので、テーブルに対する大量の挿入では、GINインデックスを削除し、大量の挿入が終わった段階で再作成することを勧めます。

PostgreSQL 8.4から遅延インデックス作成が使用されるため、この勧告は必要性が薄れました。（[項55.3.1](#)を参照してください。）しかし非常に大規模な更新では、インデックスの削除と再作成がまだ最善かもしれません。

[maintenance_work_mem](#)

GINインデックスの構築時間は`maintenance_work_mem`の設定に非常に敏感です。インデックス作成時に作業メモリをより少なく使用しようとはしません。

[work_mem](#)

既存の`FASTUPDATE`が有効なGINインデックスに対して挿入を繰り返す間、待機中の項目リストが`work_mem`より大きくなると、システムはこのリストを整理します。観測される応答時間の変動を防ぐためには、待機中リストの整理をバックグラウンド（例えば自動バキューム）で起きるようにすることが望まれます。フォアグラウンドでの整理処理は、`work_mem`を大きくすること、もしくは自動バキュームをより積極的に行うことで防ぐことができます。しかし`work_mem`を大きくすることは、フォアグラウンドで整理処理が発生した時により長い時間がかかることを意味します。

[gin_fuzzy_search_limit](#)

GINインデックス開発の主な目的は、スケーラビリティが高い全文検索のサポートをPostgreSQLで作成することでした。全文検索の結果は非常に大規模な結果セットを返します。さらに、問い合わせが非常に高頻度な単語を持つ場合、こうした状況はよく発生しますが、大規模な結果セットは有用ですらありません。ディスクから大量のタプルを読み、ソートすることは長い時間がかかりますので、実運用レベルでは受け入れられません。（インデックス検索自体は非常に高速であることに注意してください。）

こうした問い合わせの実行を簡単に制御できるように、GINは返される行数に対して設定可能なソフト上限、`gin_fuzzy_search_limit`設定パラメータを持ちます。これはデフォルトでは0です（無制限を意味します）。非0の制限が設定された場合、返されるセットは結果セット全体からランダムに選んだサブセットになります。

"ソフト"は、問い合わせとシステムの乱数ジェネレータの品質に依存して、返される結果の実際の数が指定した上限より多少異なることを意味します。

経験上、数千（例えば5000から20000）の値がうまく動作します。

55.5. 制限

GINは、インデックス付け可能な演算子は厳密であると仮定します。これは`extractValue`はNULL項目値についてはまったく呼び出されない（代わりにインデックス項目のブレースホルダが自動的に生成される）こと、および、`extractQuery`は問い合わせの値がNULLの場合に呼び出されない（代わりに問い合わせは不一致であるとみなされる）ことを意味します。しかし非NULLの複合型項目内または問い合わせ値内のNULLキー値はサポートされます。

55.6. 例

PostgreSQLソース配布物には`tsvector`とすべての整数型の1次元配列に対するGIN演算子クラスが含まれています。`tsvector`にある前置検索はGINの部分一致機能の使用で実装されています。以下の`contrib`モジュールにもGIN演算子クラスが含まれています。

`btree_gin`

さまざまなデータ型に対するB-tree等価の機能

`hstore`

(キー、値) の組み合わせを格納するモジュール

`intarray`

`int[]` に対する高度サポート

`pg_trgm`

トリグラム一致を使用したテキスト類似度

第 56 章 データベースの物理的な格納

目次

- 56.1. [データベースファイルのレイアウト](#)
- 56.2. [TOAST](#)
- 56.3. [空き領域マップ](#)
- 56.4. [可視性マップ](#)
- 56.5. [初期化フォーク](#)
- 56.6. [データベースページのレイアウト](#)

本章ではPostgreSQLデータベースで使用される物理的格納書式についての概要を説明します。

56.1. データベースファイルのレイアウト

本節ではファイルとディレクトリというレベルで格納書式について説明します。

データベースクラスタで必要となるすべてのデータは、クラスタのデータディレクトリ内に格納され、通常（このディレクトリを定義するために使用できる環境変数名にちなんで）PGDATAとして参照されます。通常のPGDATAの位置は/var/lib/pgsql/dataです。異なるサーバインスタンスによって管理することで、複数のクラスタを同一のマシン上に存在させることができます。

表56-1に示すように、PGDATAディレクトリには数個のサブディレクトリと制御ファイルがあります。これら必要な項目に加え、クラスタの設定ファイルであるpostgresql.conf、pg_hba.confおよびpg_ident.confが伝統的にPGDATA内に格納されます（ただしPostgreSQL 8.0以降では他の場所にも保存できます）。

表 56-1. PGDATAの内容

項目	説明
PG_VERSION	PostgreSQLの主バージョン番号を保有するファイル
base	データベースごとのサブディレクトリを保有するサブディレクトリ
global	pg_databaseのようなクラスタで共有するテーブルを保有するサブディレクトリ
pg_clog	トランザクションのコミット状態のデータを保有するサブディレクトリ
pg_multixact	マルチトランザクションの状態のデータを保有するサブディレクトリ（共有行ロックで使用されます）
pg_notify	LISTEN/NOTIFY状態データを保有するサブディレクトリ
pg_serial	コミットされたシリアライズابلトランザクションに関する情報を保有するサブディレクトリ
pg_snapshots	エクスポートされたスナップショットを保有するサブディレクトリ
pg_stat_tmp	統計用サブシステム用の一時ファイルを保有するサブディレクトリ
pg_subtrans	サブトランザクションの状態のデータを保有するサブディレクトリ
pg_tblspc	テーブル空間へのシンボリックリンクを保有するサブディレクトリ
pg_twophase	プリペアドトランザクション用の状態ファイルを保有するサブディレクトリ
pg_xlog	WAL（ログ先行書き込み）ファイルを保有するサブディレクトリ
postmaster.opts	最後にサーバを起動した時のコマンドラインオプションを記録するファイル
postmaster.pid	現在のpostmasterプロセスID（PID）、クラスタのデータディレクトリパス、postmaster起動時のタイムスタンプ、ポート番号、Unixドメインソケットのディレクトリパス（Windowsでは空）、有効な監視アドレスの一番目（IPアドレスまたは*、TCPを監視していない場合は空）および共有メモリのセグメントIDを記録するロックファイル（サーバが停止した後は存在しません）

クラスタ内の各データベースに対して、PGDATA/base内にサブディレクトリが存在し、サブディレクトリ名はpg_database内のデータベースOIDとなります。このサブディレクトリはデータベースファイルのデフォルトの位置であり、特にシステムカタログがそこに格納されます。

各テーブルおよびインデックスは別個のファイルに格納されます。通常のリレーションでは、これらのファイル名はテーブルまたはインデックスのファイルノード番号となります。ファイルノード番号はpg_class.relfilenode内で見つけられます。しかし一時的なリレーションでは、ファイル名はtBBB_FFFという形になります。ここでBBBはファイルを生成したバックエンドのバックエンドID、FFFはファイルノード番号です。どちらの場合でも、主ファイル（いわゆる主フォーク）に加え、それぞれのテーブルとインデックスはリレーションに利用できる空き領域についての情報を格納する空き領域マップ（項56.3参照）を持ちます。空き領域マップはファイルノード番号に接尾辞_fsmがついた名前のファイルに格納されます。テーブルは同時に、どのページが不要なタプルを持っていないと判断できるように追跡する可視性マップを持ち、フォークに接尾辞_vmを付けたファイルに格納します。可視性マップは項56.4でより詳しく解説します。ログを取らないテーブルとインデックスは、初期化フォークという第3のフォークを持ち、フォークに接尾辞_initを付けたファイルに格納します（項56.5参照）。

注意
テーブルにおけるファイルノード番号とOIDは多くの場合一致しますが、常に一致するとは限らないことに注意してください。TRUNCATE、REINDEX、CLUSTER等のいくつかの操作、およびALTER TABLEにおけるいくつかの構文は、OIDを保持したままファイルノード番号を変更できます。ファイルノード番号とテーブルOIDが同一であると仮定しないでください。またpg_class自身を含む特定のシステムカタログにおいて、pg_class.relfilenodeはゼロを持ちます。これらのカタログの実際

のファイルノード番号は低レベルなデータ構造内に保管されており、`pg_relation_filenode()` 関数を使用して入手できます。

テーブルまたはインデックスが1ギガバイトを超えると、ギガバイト単位のセグメントに分割されます。最初のセグメントのファイル名はファイルノード番号と同一であり、それ以降は、ファイルノード番号.1、ファイルノード番号.2等の名称になります。この配置法によってファイル容量に制限のあるプラットフォームにおける問題を回避します。（実際、1ギガバイトは単なるデフォルトのセグメント容量です。セグメント容量はPostgreSQLを構築する際、`--with-segsize` 設定オプションを使用して調整することができます。）原理上、空き領域マップと可視性マップのフォークにおいても複数のセグメントも必要とする可能性があります。実際のところは起こりそうにありません。

項目が大きくなりそうな列を持ったテーブルは、連携したTOASTテーブルを保有する可能性があります。TOASTテーブルは、テーブル行の中には大き過ぎて適切に保持できないフィールド値を行の外部に格納するために使用されます。TOASTテーブルが存在する時、`pg_class.reltoastrelid`は元のテーブルとTOASTテーブルを結びつけます。[項56.2](#)を参照してください。

テーブルおよびインデックスの内容は、[項56.6](#)においてさらに考察されています。

テーブル空間は状況をさらに複雑にします。ユーザが定義したテーブル空間はそれぞれ、`PGDATA/pg_tblspc` ディレクトリ内に物理的なテーブル空間ディレクトリ（つまりそのテーブル空間の`CREATE TABLESPACE` コマンドで指定された場所）を指し示す、シンボリックリンクを持ちます。シンボリックリンクの名称はテーブル空間のOIDとなります。物理的テーブル空間ディレクトリの内部では、`PG_9.0_201008051`などのPostgreSQLサーバのバージョンに依存した名前のサブディレクトリが存在します。（このサブディレクトリを使用する理由は、競合することなく`CREATE TABLESPACE` で指定する場所と同じものを将来のバージョンのデータベースでも使用できるようにするためです。）このバージョン固有のサブディレクトリの内部では、テーブル空間に要素を持つデータベースごとに、データベースOIDをディレクトリ名としたサブディレクトリが存在します。テーブルとインデックスは、ファイルノードの命名の規定に従って、そのディレクトリ内に格納されます。`pg_default` テーブル空間は`pg_tblspc`を通してアクセスされるのではなく、`PGDATA/base`と連携します。同様に、`pg_global` テーブル空間は`pg_tblspc`を通してアクセスされるのではなく、`PGDATA/global`と連携します。

`pg_relation_filepath()` 関数は任意のリレーションの(`PGDATA`から相対的な)パス全体を示します。これは上の規則の多くを記憶する必要がありませんので、しばしば有用です。しかし、この関数がリレーションの主フォークの最初のセグメントの名前だけを返すことに注意して下さい。リレーションに関したすべてのファイルを見つけるためにセグメント番号や`_fsm`や`_vm`を追加する必要があるかもしれません。

一時ファイル（メモリ内に収まりきらないデータのソートなどの操作）は`PGDATA/base/pgsql_tmp`内、または、`pg_default`以外のテーブル空間が指定されていた場合はテーブル空間ディレクトリ下の`pgsql_tmp`サブディレクトリ内に作成されます。一時ファイルの名前は`pgsql_tmpPPP.NNN`という形式です。ここで、`PPP`は所有するバックエンドのPIDであり、`NNN`で同一バックエンドで作成された別の一時ファイルと区別します。

[前のページ](#)

データベースの物理的な格納

[ホーム](#)

[上に戻る](#)

[次のページ](#)

TOAST

56.2. TOAST

本節ではTOAST（過大属性格納技法：The Oversized-Attribute Storage Technique）の概要について説明します。

PostgreSQLは固定長のページサイズ（通常8キロバイト）を使用し、複数ページにまたがるタプルを許しません。そのため、大規模なフィールド値を直接格納できません。大規模なフィールド値を圧縮したり、複数の物理的な行に分割したりすることで、この限界はなくなりました。これはユーザからは透過的に発生し、また、バックエンドのコード全体には小さな影響しか与えませんでした。この技法はTOAST（または"パンをスライスして以来最善のもの"）という愛称で呼ばれます。[訳注："パンをスライスして以来最善のもの（the best thing since sliced bread）"は素晴らしいものを意味します。]

一部のデータ型のみがTOASTをサポートします。大規模なフィールド値を生成することがないデータ型にオーバーヘッドを負わせる必要はありません。TOASTをサポートするためには、データ型は可変長（*varlena*）表現を持たなければなりません。格納する値の最初の32ビットワードにはバイト単位の値の（このワード自体を含む）長さが含まれます。TOASTは残りの表現について制限しません。TOAST可能なデータ型をサポートするC言語関数はすべて、TOAST化された入力値を注意して扱わなければなりません（通常これは、入力に対して何か作業をする前にPG_DETOAST_DATUMを呼び出すことで行われますが、もっと効率的な方法が可能な場合もあります）。

TOASTはvarlenaの長さワードの2ビット（ビッグエンディアンのマシンでは上位ビット、リトルエンディアンのマシンでは下位ビット）を勝手に使用します。そのため、すべてのTOAST可能なデータ型の値の論理サイズは1ギガバイト（ 2^{30} - 1バイト）までになります。両ビットが0の場合、値はそのデータ型の普通のTOAST化されていない値となり、長さワードの残りのビットはデータの（長さワードを含む）総サイズ（バイト単位）となります。上位側または下位側のどちらか片方のビットが設定された場合、値は通常の4バイトのヘッダを持たず1バイトのヘッダを持ちます。また、長さワードの残りビットはデータの（長さワードを含む）総サイズ（バイト単位）となります。特殊な状況として、長さワードの残りビットがすべて0（自身の長さを含みますのでありえません）の場合、その値は別のTOASTテーブルに保存される行外データへのポインタです。（TOASTポインタのサイズはデータの第2バイト内で与えられます。）単一バイトヘッダを持つ値は特定の境界に整列されません。最後に上位側または下位側のビットが0で隣のビットが設定されている場合、データの内容は圧縮され、使用前に伸長しなければなりません。この場合、長さワードの残りビットは元データのサイズではなく圧縮したデータの総サイズになります。圧縮が行外データでも起こりえますが、varlenaヘッダには圧縮されているかどうかについての情報がないことに注意してください。その代わりTOASTポインタの内容にこの情報が含まれています。

テーブルの任意の列がTOAST可能な場合、そのテーブルは連携するTOASTテーブルを持ちます。TOASTテーブルのOIDはテーブルのpg_class.reltoastrelid項目に格納されます。詳細は後で説明しますが、行外のTOAST化された値はTOASTテーブル内に保持されます。

使用される圧縮技術は、LZ系の圧縮技術の1つで単純かつ非常に高速なものです。詳細はsrc/backend/utils/adit/pg_lzcompress.cを参照してください。

行外の値は（圧縮される場合は圧縮後に）最大TOAST_MAX_CHUNK_SIZEバイトの塊に分割されます（デフォルトではこの値は4チャンク行が1ページに収まり、およそ2000バイトになるように選ばれます）。各塊は、データを持つテーブルと連携するTOASTテーブル内に個別の行として格納されます。すべてのTOASTテーブルはchunk_id列（特定のTOAST化された値を識別するOID）、chunk_seq列（値の塊に対する連番）、chunk_data（塊の実際のデータ）列を持ちます。chunk_idとchunk_seqに対する一意性インデックスは値の抽出を高速化します。したがって、行外のTOAST化された値を示すポインタデータには、検索先となるTOASTテーブルのOIDと指定した値のOID(chunk_id)を格納しなければなりません。簡便性のために、ポインタデータには論理データサイズ（元々の非圧縮のデータ長）と実際の格納サイズ（圧縮時には異なります）も格納されます。varlenaヘッダバイトに収納するためにTOASTポインタデータの総サイズは、表現される値の実サイズに関係なく、18バイトになります。

TOASTのコードは、テーブル内に格納される値がTOAST_TUPLE_THRESHOLD（通常2キロバイト）を超える時にのみ実行されます。TOASTコードは、行の値がより小さくなるかそれ以上の縮小ができなくなるまで、フィールド値の圧縮や行外への移動を行います。更新操作中、変更されない値は通常そのまま残ります。行外の値を持つ行の更新では、行外の値の変更がなければTOASTするコストはかかりません。

TOASTコードでは、以下のTOAST可能な列を格納するための4つの異なる戦略を認めます。

- **PLAIN**は圧縮や行外の格納を防止します。さらにvarlena型での単一バイトヘッダの使用を無効にします。これはTOAST化不可能のデータ型の列に対してのみ取り得る戦略です。
- **EXTENDED**では、圧縮と行外の格納を許します。これはほとんどのTOAST可能なデータ型のデフォルトです。圧縮がまず行われ、それでも行が大き過ぎるのであれば行外に格納します。
- **EXTERNAL**は非圧縮の行外格納を許します。EXTERNALを使用すると、textとbytea列全体に対する部分文字列操作が高速化されます。こうした操作は非圧縮の行外の値から必要な部分を取り出す時に最適化されるためです（格納領域が増加するという欠点があります）。

- `MAIN`は圧縮を許しますが、行外の格納はできません（実際にはこうした列についても行外の格納は行われます。しかし、他に行を縮小させページに合わせる方法がない場合の最後の手段としてのみです）。

TOAST可能なデータ型はそれぞれ、そのデータ型の列用のデフォルトの戦略を指定します。しかし`ALTER TABLE SET STORAGE`を使用して、あるテーブル列の戦略を変更することができます。

この機構には、ページをまたがる行の値を許可するといった素直な手法に比べて多くの利点があります。通常問い合わせは比較的小さなキー値に対する比較で条件付けされるものと仮定すると、エグゼキュータの仕事のほとんどは主だった行の項目を使用して行われることになります。TOAST化属性の大規模な値は、（それが選択されている時）結果集合をクライアントに戻す時に引き出されるだけです。このため、主テーブルは行外の格納を使用しない場合に比べて、かなり小さくなり、その行は共有バッファキャッシュにより合うようになります。ソート集合もまた小さくなり、ソートが完全にメモリ内で行われる頻度が高くなります。小規模な試験結果ですが、典型的なHTMLページとそのURLを持つテーブルでは、TOASTテーブルを含め、元々のデータサイズのおよそ半分で格納でき、さらに、主テーブルには全体のデータのおよそ10%のみ（URLと一部の小さなHTMLページ）が格納されました。TOAST化されないようにすべてのHTMLページを7キロバイト程度に切り詰めたTOASTかつ圧縮化しないテーブルと比べ、実行時に違いはありませんでした。

[前のページ](#)

データベースファイルのレイアウト

[ホーム](#)
[上に戻る](#)

[次のページ](#)
空き領域マップ

56.3. 空き領域マップ

ヒープとハッシュインデックス以外のインデックスリレーションはそれぞれ、そのリレーション内で利用可能な領域を継続して追跡するために、空き領域マップ (FSM) を持ちます。これは、個々のリレーションのフォーク内の主リレーションデータに沿って、リレーションのファイルノード番号に `_fsm` という接尾辞を付けた名前のファイルに格納されます。例えばリレーションのファイルノードが 12345 であれば、FSM は主リレーションファイルと同じディレクトリ内の `12345_fsm` という名前のファイルに格納されます。

空き領域マップは FSM ページのツリーとして編成されます。最下位レベルの FSM ページはすべてのヒープ（またはインデックス）ページで利用可能な空き領域を、各ページ毎に 1 バイト使用して格納します。上位レベルは下位レベルからの情報を集約します。

各 FSM ページの内部はノード当たり 1 バイトを持つ配列内に格納されたバイナリツリーです。各リーフノードはヒープページ、または下位レベルの FSM ページを表現します。各非リーフノード内には、子の値より大きな値が格納されます。したがってリーフノード内の最大値がルートに格納されます。

FSM がどのように構築されるか、そしてどのように更新、検索されるかに関する詳細は `src/backend/storage/freespace/README` を参照してください。 [pg_freespacemap](#) モジュールを使用して、空き領域マップに格納された情報を調べることができます。

56.4. 可視性マップ

各ヒープリレーションは、どのページがすべての実行中のトランザクションから可視であることが分かっているタプルのみを持つことを追跡する、可視性マップ (VM) を持ちます。これは、リレーションのファイルノード番号に `_vm` という接尾辞を付与した名前の別のリレーションフォーク内に主リレーションデータと並行して格納されます。例えばリレーションのファイルノードが12345の場合、VMは主リレーションファイルと同じディレクトリ内の `12345_vm` というファイル内に格納されます。インデックスはVMを持たないことに注意してください。

可視性マップは単にヒープページ当たり1ビットを保持します。セットされたビットは、ページ内のすべてのタプルがすべてのトランザクションから可視であることが分かっていることを意味します。これは、そのページにはバキュームが必要なタプルをまったく含んでいないことを意味します。またこの情報は、インデックスタプルのみを使用して問い合わせに答えるためにインデックスオンリースキャンによっても使用されます。

このマップは、ビットがセットされている時は常にこの条件が真であることを確実に把握できるという点で保守的ですが、ビットがセットされていない場合は、真かもしれませんが偽かもしれません。可視性マップのビットはバキュームによってのみで設定されます。しかしページに対する任意のデータ編集操作によってクリアされます。

56.5. 初期化フォーク

ログを取らないテーブルと、ログを取らないテーブルに対するインデックスは、それぞれ初期化フォークを持ちます。初期化フォークとは適切な種類の空テーブルと空インデックスです。ログを取らないテーブルをクラッシュのために再度空にしなければならない場合、初期化フォークで主フォーク全体をコピーし、その他のフォークは消去されます（これらは必要に応じて自動的に再作成されます）。

56.6. データベースページのレイアウト

本節ではPostgreSQLのテーブルおよびインデックスで使われるページ書式の概略について説明します。[1] TOASTのテーブルとシーケンスは、通常のテーブルと同様に整形されています。

以下の説明では1バイトは8ビットからなることを前提としています。さらに、アイテムという単語は、ページに格納される個別のデータ値のことを指しています。テーブル内ではアイテムは行であり、インデックス内ではアイテムはインデックスのエントリです。

テーブルとインデックスはすべて、固定サイズ（通常8キロバイト。サーバのコンパイル時に異なるサイズを設定可能）のページの集まりとして格納されます。テーブルでは、すべてのページは論理上等価です。したがって、あるアイテム（行）はどのページにでも格納することができます。インデックスでは、初めのページは通常、制御用の情報を保持するメタページとして予約されます。また、インデックスではインデックスアクセスメソッドに依存した様々なページ種類があります。

[表56-2](#)はページの全体的なレイアウトを示しています。各ページには5つの部分があります。

表 56-2. ページレイアウト全体

アイテム	説明
PageHeaderData	長さは24バイト。空き領域ポインタを含む、ページについての一般情報です。
ItemIdData	実際のアイテムを指す（オフセットと長さの）ペアの配列です。1アイテムにつき4バイトです。
空き領域	割り当てられていない空間です。新規のアイテムポインタはこの領域の先頭から、新規のアイテムは最後から割り当てられます。
アイテム	実際のアイテムそのものです。
特別な空間	インデックスアクセスメソッド特有のデータです。異なるメソッドは異なるデータを格納します。通常のテーブルでは空です。

それぞれのページの最初の24バイトはページヘッダ（PageHeaderData）から構成されています。その書式を[表56-3](#)にて説明します。最初の2つのフィールドは、このページに関連する最も最近のWAL項目を表しています。次にフラグビットを含む2バイトのフィールドがあります。その後2バイトの整数フィールドが3つ続きます（pd_lower、pd_upper、pd_special）。これらには、割り当てられていない空間の始まり、割り当てられていない空間の終わり、そして特別な空間の始まりのバイトオフセットが格納されています。ページヘッダの次の2バイトであるpd_pagesize_versionは、ページサイズとバージョン指示子の両方を格納します。PostgreSQL 8.3のバージョン番号は4、PostgreSQL 8.1と8.2のバージョン番号は3、PostgreSQL 8.0のバージョン番号は2、PostgreSQL 7.3と7.4のバージョン番号は1です。それより前のリリースのバージョン番号は0です（ほとんどのバージョン間で基本的なページレイアウトやヘッダの書式は変更されていませんが、ヒープ行ヘッダのレイアウトが変更されました）。ページサイズは基本的に照合用としてのみ存在しています。同一インスタレーションでの複数のページサイズはサポートされていません。最後のフィールドはそのページの切り詰めが有益かどうかを示すヒントです。これはページ上で切り詰められていないもっとも古いXMAXが追跡するものです。

表 56-3. PageHeaderDataのレイアウト

フィールド	型	長さ	説明
pd_lsn	XLogRecPtr	8バイト	LSN: このページへの最終変更に対応するxlogレコードの最後のバイトの次のバイト
pd_tli	uint16	2バイト	最終変更のTimeLineID(その下位側16ビットのみ)
pd_flags	uint16	2バイト	フラグビット
pd_lower	LocationIndex	2バイト	空き領域の始まりに対するオフセット
pd_upper	LocationIndex	2バイト	空き領域の終わりに対するオフセット
pd_special	LocationIndex	2バイト	特別な空間の始まりに対するオフセット
pd_pagesize_version	uint16	2バイト	ページサイズおよびレイアウトのバージョン番号の情報

		ト	
pd_prune_xid	TransactionId	4バイト ト	ページ上でもっとも古い切り詰められていないXMAX。存在しなければゼロ。

詳細情報についてはsrc/include/storage/bufpage.hを参照してください。

ページヘッダに続くのはアイテム識別子 (ItemIdData) です。 識別子ごとに4バイトを必要とします。 アイテム識別子は、アイテムが開始されるバイトオフセット、バイト単位の長さ、そしてその解釈に影響する属性ビット群を持っています。 新しいアイテム識別子は必要に応じて、未割当て空間の最初から割り当てられます。 アイテム識別子の数は、新しい識別子を割り当てるために増加されるpd_lowerを見ることで決定できます。 アイテム識別子は解放されるまで動かされることがないので、アイテム自体が空き領域をまとめるためにページ上で移動される場合でも、そのインデックスはアイテムを参照するために長期にわたって使うことができます。 実際、PostgreSQLが作る、アイテムへのポインタ (ItemPointer、CTIDとも言います) はページ番号とアイテム識別子のインデックスによって構成されています。

アイテム自体は、未割り当て空間の最後から順番に割り当てられた空間に格納されます。 正確な構造は、テーブルに何を含めるかによって異なります。 テーブルとシーケンスの両方が、以下で説明するHeapTupleHeaderDataという構造を使用します。

最後のセクションは、アクセスメソッドが格納しようとするものを何でも含めることのできる"特別なセクション"です。 例えば、B-treeインデックスは、そのページの両隣のページへのリンク、ならびに、インデックス構造体に関連したその他の何らかのデータを持ちます。 通常のテーブルではこれはまったく使用されません (ページサイズを同じにするためにpd_specialを設定することで示されます) 。

テーブル行はすべて同じ方法で構成されています。 固定サイズのヘッダ (ほとんどのマシンで23バイトを占有します) があり、その後にオプションのNULLビットマップ、オプションのオブジェクトIDフィールド、およびユーザデータが続きます。 ヘッダについては表56-4で説明します。 実際のユーザデータ (行内の列) は、常にプラットフォームのMAXALIGN距離の倍数であるt_hoffで示されるオフセットから始まります。 NULLビットマップはHEAP_HASNULLビットがt_infomaskで設定されている場合にのみ存在します。 存在する場合は、固定ヘッダのすぐ後ろから始まり、データ列ごとに1ビットとするのに十分なバイト数を占有します (合計すると、t_nattsのビット数となります) 。 このビットのリスト内では、1ビットは非NULLを、0ビットはNULLを示します。 このビットマップが存在しない場合、すべての列が非NULLとみなされます。 オブジェクトIDはHEAP_HASOIDビットがt_infomaskで設定されている場合にのみ存在します。 存在する場合、これはt_hoff境界の直前に現れます。 t_hoffをMAXALIGNの倍数とするために必要なパッドは全て、NULLビットマップとオブジェクトIDの間に現れます (このことにより、オブジェクトIDの位置揃えが確実に適切になります) 。

表 56-4. HeapTupleHeaderDataのレイアウト

フィールド	型	長さ	説明
t_xmin	TransactionId	4バイト	挿入XIDスタンプ
t_xmax	TransactionId	4バイト	削除XIDスタンプ
t_cid	CommandId	4バイト	挿入、削除の両方または片方のCIDスタンプ (t_xvacと共有)
t_xvac	TransactionId	4バイト	行バージョンを移すVACUUM操作用XID
t_ctid	ItemPointerData	6バイト	この行または最新バージョンの行の現在のTID
t_infomask2	uint16	2バイト	属性の数と各種フラグビット
t_infomask	uint16	2バイト	様々なフラグビット
t_hoff	uint8	1バイト	ユーザデータに対するオフセット

詳細情報についてはsrc/include/access/htup.hを参照してください。

実際のデータの解釈は、他のテーブル、ほとんどの場合、pg_attributeから取得された情報でのみ行うことができます。 フィールド位置を識別するために必要なキー値は、attlenおよびattalignです。 フィールドの幅が固定されていてNULL値が存在しない場合を除き、特定の属性を直接取得する方法はありません。 この仕組みはすべて、heap_getattr、fastgetattrおよびheap_getsysattr関数にラップされています。

データを読むためには、それぞれの属性を順番に検査する必要があります。 まず、NULLビットマップに従ってフィールドがNULLかどうかを検査します。 もしNULLであれば、次に進みます。 次に、位置揃えが正しいことを確認してください。 フィールドの幅が固定されていれば、すべてのバイトが単純に配置されます。 可変長のフィールド (attlen == -1) の場合はもう少し複雑です。 可変長のデータ型はすべて、格納する値の長さといくつかのフラグビットを持つstruct varlenaという共通ヘッダ構造体を共有します。 フラグによって、データは行内、または別のテーブル (TOAST) のいずれかとなったり、圧縮済みとなったりします (項56.2を参照してください) 。

注意

[1]

実際にはインデックスアクセスメソッドはこのページ書式を使用する必要はありません。既存のすべてのインデックスメソッドがこの基本書式を使用しています。しかし、インデックスメタページに保持されるデータは通常、アイテムレイアウト規則に正確には従っていません。

[前のページ](#)

初期化フォーク

[ホーム](#)
[上に戻る](#)

[次のページ](#)

BKIバックエンドインタフェース

第 57 章 BKI バックエンドインタフェース

目次

- 57.1. [BKI ファイル形式](#)
- 57.2. [BKI コマンド](#)
- 57.3. [BKI ファイルのブートストラップの構成](#)
- 57.4. [例](#)

バックエンドインタフェース (BKI) ファイルは、"bootstrap" モードで稼働中の PostgreSQL で解釈される特殊な言語で記述されたスクリプトです。普通の SQL コマンドを実行するにはシステムカタログが事前に存在していなければなりません。が、bootstrap モードではシステムカタログを一から作成することができます。ですから BKI ファイルはデータベースシステムを初めて作成する時に使用します。（そしてそれ以外の時にはあまり有効ではありません。）

initdb は新しいデータベースクラスタを作成する際に、その作業の一部を担わすため BKI ファイルを使います。initdb が使う入力ファイルは PostgreSQL の構築時に `genbki.pl` というプログラムによって、ソースツリー内の `src/include/catalog/` にある特別な書式の C のヘッダファイルから作成されます。作成された BKI ファイルは `postgres.bki` と呼ばれ、通常インストールツリーの `share` サブディレクトリにインストールされます。

関連情報は initdb のドキュメントから見付けられます。

57.1. BKIファイル形式

本節ではPostgreSQLのバックエンドがどのようにしてBKIファイルを解釈するのかを説明します。例として`postgres.bki`ファイルが手元にあると、説明が一層理解しやすくなるでしょう。

BKIの入力は一連のコマンドで構成されます。コマンドはいくつかのトークンから構成されていて、コマンドの構文に依存しています。トークンは通常空白で分離されていますが、どちらとも解釈されるような曖昧性がなければ必要性ありません。特別なコマンド区切り文字はありません。したがって、構文上その前のコマンドに属することができない次のトークンは新たなコマンドとなります（通常、わかりやすくするために、新しいコマンドは新しい行に記述します）。トークンはある一定のキーワードや特別な文字（括弧やカンマなど）、数字、二重引用符で囲まれた文字列などが使用できます。大文字/小文字は全て区別されます。

#で始まる行は無視されます。

57.2. BKIコマンド

```
create tablename tableoid [bootstrap] [shared_relation] [without_oids] [rowtype_oid oid] (name1 = type1 [, name2 = type2, ...])
```

括弧で与えられた列と、OID *tableoid*を持つ*tablename*というテーブルを作成します。

次の列型はbootstrap.cで直接サポートされます。bool、bytea、char (1バイト)、name、int2、int4、regproc、regclass、regtype、text、oid、tid、xid、cid、int2vector、oidvector、_int4 (配列)、_text (配列)、_oid (配列)、_char (配列)、_aclitem (配列)。その他の型を持つテーブルを作成することはできませんが、pg_typeが完了し適切な項目で埋められるまで完了させることができません。(これらの列型のみブートストラップされたテーブルで使用されますが、非ブートストラップカタログは如何なる組み込み型も含む事があるという事を実際に意味しています。)

bootstrapが指定された場合、テーブルはディスク上に作成されるだけで、pg_classやpg_attributeなどにその項目は登録されません。したがって、これらの項目が (insert コマンドで) 固定化されるまで、普通のSQL操作でこのテーブルにアクセスできません。このオプションはpg_classなど自身を作成するために使用されます。

shared_relationが指定された場合、テーブルは作成されます。without_oidsが指定されていない限りテーブルはOIDを持ちます。テーブルの行型OID (pg_type OID) はrowtype_oid句で指定できます。指定されなければ、OIDは自動的に生成されます。(bootstrap が指定されていれば、rowtype_oid句は役に立ちません。しかし、文書化のためにともかく指定はできます。)

```
open tablename
```

データを挿入するために*tablename*と名前が付けられたテーブルを開きます。現在開いているテーブルは閉じられます。

```
close [tablename]
```

開いているテーブルを閉じます。テーブル名は様々な方法で検査可能ですが、必要性はありません

```
insert [OID = oid_value] ( value1 value2 ... )
```

value1やvalue2などを列の値として、oid_valueをOIDとし、開いているテーブルに行を挿入します。もしoid_valueがゼロ (0)、もしくは句が省略され、テーブルがOIDを所有していれば、次の有効なOIDが割り当てられます。

NULL値は特別なキーワード、_null_によって指定できます。空白を含む値は二重引用符で囲まなければなりません。

```
declare [unique] index indexname indexoid on tablename using amname ( opclass1 name1 [, ...] )
```

amnameアクセスメソッドを使用して、tablenameと名付けられたテーブル上に、OID indexoidを所有する、indexnameという名前のインデックスを作成します。インデックスが付けられるフィールドは、name1、name2など、そして使用される演算子クラスはopclass1、opclass2などとそれぞれ呼ばれます。このインデックスファイルは作成され、適切なカタログ項目が作成されますが、このコマンドではインデックスの内容の初期化を行いません。

```
declare toast toasttableoid toastindexoid on tablename
```

tablenameという名前のテーブル用のTOASTテーブルを作成します。このTOASTテーブルはOIDとしてtoasttableoidが割り当てられ、そのインデックスはOIDとしてtoastindexoidが割り当てられます。declare indexと同様、インデックスの作成は遅延されます。

```
build indices
```

前に宣言されたインデックスを作成します。

57.3. BKIファイルのブートストラップの構成

openコマンドは、テーブルが、使用するテーブルが存在し、開かれるテーブルに対しエントリを所有するまで使用できません。（これら最小限度のテーブルは、pg_class、pg_attribute、pg_proc、およびpg_typeです。）これらのテーブル自体が充填されるようにするには、bootstrapオプションを伴ったcreateが明示的にデータの挿入のために作成されたテーブルを開きます。

また、必要とするシステムカタログが作成され、値が設定されるまで、declare indexおよびdeclare toast コマンドは使用できません。

従い、postgres.bkiの構造は以下でなければなりません。

```
1 11 1つの重要なテーブルをcreate bootstrap
1 11 少なくとも重要なテーブルを記述するデータをinsert
1 11 close
1 11 その他の重要テーブルに対して反復。
1 11 重要でないテーブルを（bootstrap無しで）create
1 11 open
1 11 求められるデータのinsert
1 11 close
1 11 その他の重要でないテーブルに対して反復。
1 11 インデックスおよびTOASTテーブルの定義。
1 11 build indices
```

他にも確かに、ドキュメント化されていない順序に関する依存性があります。

57.4. 例

次の一連のコマンドは、それぞれ`int4`型と`text`型の2つの列、`cola`と`colb`を持ち、OID 420 が付いた`test_table`テーブルを作成し、そして2つの行をテーブルに挿入します。

```
create test_table 420 (cola = int4, colb = text)
open test_table
insert OID=421 ( 1 "value1" )
insert OID=422 ( 2 _null_ )
close test_table
```

第 58 章 プランナが統計情報をどのように使用するか

本章は、[項14.1](#)と[項14.2](#)で扱われている題材を基にしている、問い合わせの各段階において返される行数を推定するために、プランナがシステムの統計情報をどのように使用するかについて更なる詳細をいくつか説明します。これは計画作成処理において重要な部分で、コスト計算用の多くの情報を提供します。

本章の目的はコードを詳しく文書化することではありません。どのように動作するのかに関する概要を表すことが目的です。これによりおそらく、後にコードを参照するユーザの習得速度が向上するでしょう。

[前のページ](#)

例

[ホーム](#)[上に戻る](#)[次のページ](#)

行推定の例

58.1. 行推定の例

以下の例はPostgreSQLリグレーション試験データベース内のテーブルを使用します。表示される出力はバージョン8.3で取得しました。以前の（または以降の）バージョンとは動作が変わっているかもしれません。また、`ANALYZE`は統計情報を生成する時にランダムなサンプリングを行いますので、結果は`ANALYZE`を新しく行った後に多少変わることにご注意ください。

非常に簡単な問い合わせから始めましょう。

```
EXPLAIN SELECT * FROM tenk1;

               QUERY PLAN
-----
Seq Scan on tenk1  (cost=0.00..458.00 rows=10000 width=244)
```

プランナがどのように`tenk1`の濃度を決定するかについては[項14.2](#)で説明しました。しかし、ここで完全になるように繰り返し説明します。ページ数および行数は`pg_class`から検索されます。

```
SELECT relpages, reltuples FROM pg_class WHERE relname = 'tenk1';

 relpages | reltuples
-----+-----
      358 |      10000
```

これらの値は最後にそのテーブルを`VACUUM`または`ANALYZE`を行った時点のものです。プランナはその後、テーブル内の実際のページ数を取り出します（これはテーブルスキャンを行わない安価な操作です）。`relpages`と異なる場合、`reltuples`は現在の推定行数に達するまで増大されます。この場合、`relpages`の値は最新のものですので、推定行は`reltuples`と同じです。

次に`WHERE`句に範囲条件を持つ例に進みましょう。

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 1000;

               QUERY PLAN
-----
Bitmap Heap Scan on tenk1  (cost=24.06..394.64 rows=1007 width=244)
  Recheck Cond: (unique1 < 1000)
    -> Bitmap Index Scan on tenk1_unique1  (cost=0.00..23.80 rows=1007 width=0)
        Index Cond: (unique1 < 1000)
```

プランナは`WHERE`句の条件を検査し、`pg_operator`内の<演算子用の選択度関数を検索します。これは`oprrest`列に保持されます。今回の例ではこの項は`scalarlttsel`です。`scalarlttsel`関数は、`pg_statistics`から`unique1`の度数分布を取り出します。手作業で問い合わせる場合は、より単純な`pg_stats`ビューを検索した方が簡単です。

```
SELECT histogram_bounds FROM pg_stats
WHERE tablename='tenk1' AND attname='unique1';

 histogram_bounds
-----
{0,993,1997,3050,4040,5036,5957,7057,8029,9016,9995}
```

次に、"`< 1000`"で占められる度数分布率を取り出します。これが選択度です。この度数分布は、範囲を等頻度のバケットに分割します。ですので、行わなければならないことは、値が入るバケットを見つけ、その部分と、その前にあるバケット全体を数えることです。1000という値は明らかに2番目のバケット（993 - 1997）にあります。従って、値が各バケットの中で線形に分布していると仮定すると、選択度を以下のように計算することができます。

```
selectivity = (1 + (1000 - bucket[2].min)/(bucket[2].max - bucket[2].min))/num_buckets
             = (1 + (1000 - 993)/(1997 - 993))/10
             = 0.100697
```

つまり、1つのバケット全体に、2番目のバケットとの線形比率を加えたものを、バケット数で割ったものとなります。ここで、行の推定値は、選択度と`tenk1`の濃度を掛け合わせたものとして計算されます。

```
rows = rel_cardinality * selectivity
      = 10000 * 0.100697
      = 1007 (rounding off)
```

次に、`WHERE`句に等価条件を持つ例を検討してみましょう。

```
EXPLAIN SELECT * FROM tenk1 WHERE stringul = 'CRAAAA';

               QUERY PLAN
-----
Seq Scan on tenk1  (cost=0.00..483.00 rows=30 width=244)
  Filter: (stringul = 'CRAAAA'::name)
```

繰り返しますが、プランナはWHERE句の条件を検査し、=用の選択度関数、この場合はeqselを検索します。等価性の推定では、度数分布は役に立ちません。代わりに、選択度の決定には最頻値（MCV）のリストが使用されます。MCVを見てみましょう。後で有用になる列がいくつかあります。

```
SELECT null_frac, n_distinct, most_common_vals, most_common_freqs FROM pg_stats
WHERE tablename='tenk1' AND attname='stringul';

null_frac      | 0
n_distinct     | 676
most_common_vals | {EJAAAA, BBAAAA, CRAAAA, FCAAAA, FEAAAA, GSAAAA, JOAAAA, MCAAAA, NAAAAA, WGAAAA}
most_common_freqs | {0.00333333, 0.003, 0.003, 0.003, 0.003, 0.003, 0.003, 0.003, 0.003, 0.003}

CRAAAAがMCVのリスト内にありますので、選択度は単に最頻度（MCF）のリスト内の対応する項目になります。
```

```
selectivity = mcf[3]
            = 0.003
```

前と同様、推定される行数は単に前回同様、この値とtenk1の濃度との積です。

```
rows = 10000 * 0.003
      = 30
```

ここで、同じ問い合わせを見てみます。ただし、今回は定数がMCV内にありません。

```
EXPLAIN SELECT * FROM tenk1 WHERE stringul = 'xxx';

               QUERY PLAN
-----
Seq Scan on tenk1  (cost=0.00..483.00 rows=15 width=244)
  Filter: (stringul = 'xxx'::name)
```

値がMCVの一覧にない場合、選択度をどのように推定するかは大きく異なります。値が一覧にない場合に使用される方法は、MCVすべての頻度に関する知識を組み合わせたものです。

```
selectivity = (1 - sum(mvf))/(num_distinct - num_mcv)
            = (1 - (0.00333333 + 0.003 + 0.003 + 0.003 + 0.003 + 0.003 +
                    0.003 + 0.003 + 0.003 + 0.003))/(676 - 10)
            = 0.0014559
```

つまり、MCVの頻度をすべて足し合わせたものを1から差し引き、そして、この他の個別値の数で除算します。これは、MCV以外の列の割合は、この他の個別値すべてに渡って一様に分布していることを前提としていることになります。NULL値が存在しないため、これを考慮する必要がないことに注意してください。（さもなくば、分子から同様にNULLの割合を差し引くことになります。）推定行数は以下のように普通に計算されます。

```
rows = 10000 * 0.0014559
      = 15 (rounding off)
```

前述のunique1 < 1000を使用した例はscalarltselが本当は何を行うかについて、単純化しすぎたものでした。ここまでで、MCVを使用した例を見てきましたので、多少詳細に補てんすることができます。unique1は一意的な列であるため、MCVが存在しません（ある値が他の値と同じとなることがないことは明確です）ので、例は計算自体は正確なものでした。一意ではない列では、通常度数分布とMCVリストの両方が存在します。そして、度数分布は、MCVで表される列母集団の位置を含みません。より正確な推定を行うことができるため、この方法を行います。この状況では、scalarltselは直接条件（例えば"< 1000"）をMCVリストの各値に適用し、条件を満たすMCVの頻度を足し合わせます。これがMCVのテーブル部分における正確な推定選択度です。その後度数分布が上と同様に使われ、MCV以外のテーブル部分における選択度を推定します。そしてこの2つの値を組み合わせて、全体の選択度を推定します。例えば、以下を検討します。

```
EXPLAIN SELECT * FROM tenk1 WHERE stringul < 'IAAAAA';

               QUERY PLAN
-----
Seq Scan on tenk1  (cost=0.00..483.00 rows=3077 width=244)
  Filter: (stringul < 'IAAAAA'::name)
```

すでにstringulのMCV情報は確認していますので、ここでは度数分布を見てみます。

```
SELECT histogram_bounds FROM pg_stats
WHERE tablename='tenk1' AND attname='stringul';

               histogram_bounds
-----
{AAAAAA, CQAAAA, FRAAAA, IBAAAA, KRAAAA, NFAAAA, PSAAAA, SGAAAA, VAAAAA, XLAAAA, ZAAAAA}
```

MCVリストを検査すると、stringul < 'IAAAAA'条件は先頭の6項目で満たされ、最後の4項目で満たされることがわかります。ですので、母集団のMCV部分における選択度は以下ようになります。

```
selectivity = sum(relevant mvfs)
            = 0.00333333 + 0.003 + 0.003 + 0.003 + 0.003 + 0.003
            = 0.01833333
```

MCFの総和はまた、MCVで表される母集団の合計割合が0.03033333であり、したがって度数分布で表される割合が0.96966667であることがわかります。（繰り返しますが、NULLは存在しません。もし存在する場合はここで除外しなければなりません。）IAAAAAAという値は3番目のバケットの終端近辺になることを確認することができます。異なる文字の頻度について多少安っぽい仮定を使用すると、プランナはIAAAAAより小さい母集団の度数分布の部分の推定値は0.298387になります。そしてMCVと非MCV母集団についての推定値を組み合わせます。

```
selectivity = mcv_selectivity + histogram_selectivity * histogram_fraction
              = 0.01833333 + 0.298387 * 0.96966667
              = 0.307669

rows        = 10000 * 0.307669
              = 3077 (rounding off)
```

列の分布がかなり平坦ですので、この特定の例におけるMCVリストによる訂正はかなり小さなものです。（これらの特定の値が他より頻出するものと示す統計情報はほとんどサンプリングエラーによります。）より一般的な、一部の値が他より非常に多く頻出する場合では、最頻値に対する選択度が正確に検出されますので、この複雑な処理により精度が改良されます。

次にWHERE句に複数の条件を持つ場合を検討しましょう。

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 1000 AND stringu1 = 'xxx';

               QUERY PLAN
-----
Bitmap Heap Scan on tenk1 (cost=23.80..396.91 rows=1 width=244)
  Recheck Cond: (unique1 < 1000)
  Filter: (stringu1 = 'xxx'::name)
-> Bitmap Index Scan on tenk1 unique1 (cost=0.00..23.80 rows=1007 width=0)
    Index Cond: (unique1 < 1000)
```

プランナは2つの条件が独立していると仮定します。このため、個々の句の選択度が掛け合わされます。

```
selectivity = selectivity(unique1 < 1000) * selectivity(stringu1 = 'xxx')
              = 0.100697 * 0.0014559
              = 0.0001466

rows        = 10000 * 0.0001466
              = 1 (rounding off)
```

ビットマップインデックススキャンにより返されるものと推定される行数は、インデックスで使用される条件のみを反映することに注意してください。後続のヒープ取り出しのコスト推定に影響しますので、これは重要です。

最後に、結合を含む問い合わせを見てみましょう。

```
EXPLAIN SELECT * FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 50 AND t1.unique2 = t2.unique2;

               QUERY PLAN
-----
Nested Loop (cost=4.64..456.23 rows=50 width=488)
-> Bitmap Heap Scan on tenk1 t1 (cost=4.64..142.17 rows=50 width=244)
    Recheck Cond: (unique1 < 50)
-> Bitmap Index Scan on tenk1 unique1 (cost=0.00..4.63 rows=50 width=0)
    Index Cond: (unique1 < 50)
-> Index Scan using tenk2 unique2 on tenk2 t2 (cost=0.00..6.27 rows=1 width=244)
    Index Cond: (unique2 = t1.unique2)
```

tenk1 "unique1 < 50"に関する制限が、入れ子状ループ結合の前に評価されます。これは、前の範囲に関する例と同様に扱われます。しかし、今回の値50はunique1度数分布の最初のバケットにありますので、以下のようになります。

```
selectivity = (0 + (50 - bucket[1].min)/(bucket[1].max - bucket[1].min))/num_buckets
              = (0 + (50 - 0)/(993 - 0))/10
              = 0.005035

rows        = 10000 * 0.005035
              = 50 (rounding off)
```

結合の制限はt2.unique2 = t1.unique2です。演算子はよく使用する単なる=ですが、選択度関数はpg_operatorのoprjoin列から入手され、eqjoinselとなります。eqjoinselはtenk2およびtenk1の両方の統計情報を検索します。

```
SELECT tablename, null_frac,n_distinct, most_common_vals FROM pg_stats
WHERE tablename IN ('tenk1', 'tenk2') AND attname='unique2';

tablename | null_frac | n_distinct | most_common_vals
-----+-----+-----+-----
tenk1     |          0 |          -1 |
tenk2     |          0 |          -1 |
```

今回の場合、すべての値が一意であるため、unique2に関するMCV情報がありません。ですので、両リレーシヨンの個別値数とNULL値の部分のみに依存したアルゴリズムを使用することができます。

```
selectivity = (1 - null_frac1) * (1 - null_frac2) * min(1/num_distinct1, 1/num_distinct2)
             = (1 - 0) * (1 - 0) / max(10000, 10000)
             = 0.0001
```

これは、各リレーションにおいて、1からNULL部分を差し引き、個別値数の最大値で割った値です。この結合が生成するはずの行数は、2つの入力のデカルト積の濃度に、この選択度を掛けたものとして計算されます。

```
rows = (outer_cardinality * inner_cardinality) * selectivity
       = (50 * 10000) * 0.0001
       = 50
```

2つの列に対するMCVリストがありますので、`eqjoinselect`はMCVで表される列母集団部分の結合選択度を決めるために、MCVリストを直接比較します。残りの母集団に対する推定はここで示した同じ手法に従います。

`inner_cardinality`を10000、つまり`tenk2`の変更がないサイズと示していることに注意してください。`EXPLAIN`の出力を検査すると、結合行の推定が $50 * 1$ 、つまり、外側の行数と`tenk2`上の内側のインデックススキャン毎に得られる推定行数を掛けた数から来ていると思うかもしれませんが。しかし、実際はそうではありません。結合リレーションサイズは、具体的な結合計画が検討される前に推定されます。もしすべてがうまくいけば、結合サイズを推定する2つの方法は同じ答えを導きます。しかし、四捨五入誤差などの要因により多少異なる場合があります。

詳細に興味を持った方向けに、テーブル（すべての`WHERE`句の前にあるもの）のサイズ推定は`src/backend/optimizer/util/plancat.c`で行われます。句の選択度に関する一般的なロジックについては`src/backend/optimizer/path/clausesel.c`にあります。演算子固有の選択度関数についてはたいてい`src/backend/utils/adt/selfuncs.c`内にあります。

[前のページ](#)

プランナが統計情報をどのように使用するか

[ホーム](#)

[上に戻る](#)

[次のページ](#)

付録

VIII. 付録

目次

A. [PostgreSQLエラーコード](#)

B. [日付/時刻のサポート](#)

- B.1. [日付/時刻入力の解釈](#)
- B.2. [日付/時刻キーワード](#)
- B.3. [日付/時刻設定ファイル](#)
- B.4. [単位の歴史](#)

C. [SQLキーワード](#)

D. [SQLへの準拠](#)

- D.1. [サポートされている機能](#)
- D.2. [サポートされていない機能](#)

E. [リリースノート](#)

- E.1. [リリース9.2.4](#)
- E.2. [リリース9.2.3](#)
- E.3. [リリース9.2.2](#)
- E.4. [リリース9.2.1](#)
- E.5. [リリース9.2](#)
- E.6. [リリース9.1.9](#)
- E.7. [リリース9.1.8](#)
- E.8. [リリース9.1.7](#)
- E.9. [リリース9.1.6](#)
- E.10. [リリース9.1.5](#)
- E.11. [リリース9.1.4](#)
- E.12. [リリース9.1.3](#)
- E.13. [リリース9.1.2](#)
- E.14. [リリース9.1.1](#)
- E.15. [リリース9.1](#)
- E.16. [リリース9.0.13](#)
- E.17. [リリース9.0.12](#)
- E.18. [リリース9.0.11](#)
- E.19. [リリース9.0.10](#)
- E.20. [リリース9.0.9](#)
- E.21. [リリース9.0.8](#)
- E.22. [リリース9.0.7](#)
- E.23. [リリース9.0.6](#)
- E.24. [リリース9.0.5](#)
- E.25. [リリース9.0.4](#)
- E.26. [リリース9.0.3](#)
- E.27. [リリース9.0.2](#)
- E.28. [リリース9.0.1](#)
- E.29. [リリース9.0](#)
- E.30. [リリース8.4.17](#)
- E.31. [リリース8.4.16](#)
- E.32. [リリース8.4.15](#)
- E.33. [リリース8.4.14](#)
- E.34. [リリース8.4.13](#)
- E.35. [リリース8.4.12](#)
- E.36. [リリース8.4.11](#)
- E.37. [リリース8.4.10](#)
- E.38. [リリース8.4.9](#)
- E.39. [リリース8.4.8](#)
- E.40. [リリース8.4.7](#)
- E.41. [リリース8.4.6](#)
- E.42. [リリース 8.4.5](#)
- E.43. [リリース 8.4.4](#)

E.44. [リリース 8.4.3](#)
E.45. [リリース 8.4.2](#)
E.46. [リリース8.4.1](#)
E.47. [リリース8.4](#)
E.48. [リリース8.3.23](#)
E.49. [リリース8.3.22](#)
E.50. [リリース8.3.21](#)
E.51. [リリース8.3.20](#)
E.52. [リリース8.3.19](#)
E.53. [リリース8.3.18](#)
E.54. [リリース8.3.17](#)
E.55. [リリース8.3.16](#)
E.56. [リリース8.3.15](#)
E.57. [リリース8.3.14](#)
E.58. [リリース8.3.13](#)
E.59. [リリース8.3.12](#)
E.60. [リリース8.3.11](#)
E.61. [リリース8.3.10](#)
E.62. [リリース8.3.9](#)
E.63. [リリース8.3.8](#)
E.64. [リリース8.3.7](#)
E.65. [リリース8.3.6](#)
E.66. [リリース8.3.5](#)
E.67. [リリース8.3.4](#)
E.68. [リリース8.3.3](#)
E.69. [リリース8.3.2](#)
E.70. [リリース8.3.1](#)
E.71. [リリース 8.3](#)
E.72. [リリース8.2.23](#)
E.73. [リリース8.2.22](#)
E.74. [リリース8.2.21](#)
E.75. [リリース8.2.20](#)
E.76. [リリース8.2.19](#)
E.77. [リリース8.2.18](#)
E.78. [リリース8.2.17](#)
E.79. [リリース8.2.16](#)
E.80. [リリース8.2.15](#)
E.81. [リリース8.2.14](#)
E.82. [リリース8.2.13](#)
E.83. [リリース8.2.12](#)
E.84. [リリース8.2.11](#)
E.85. [リリース8.2.10](#)
E.86. [リリース8.2.9](#)
E.87. [リリース8.2.8](#)
E.88. [リリース8.2.7](#)
E.89. [リリース8.2.6](#)
E.90. [リリース8.2.5](#)
E.91. [リリース8.2.4](#)
E.92. [リリース8.2.3](#)
E.93. [リリース 8.2.2](#)
E.94. [リリース 8.2.1](#)
E.95. [リリース 8.2](#)
E.96. [リリース8.1.23](#)
E.97. [リリース8.1.22](#)
E.98. [リリース8.1.21](#)
E.99. [リリース8.1.20](#)
E.100. [リリース8.1.19](#)
E.101. [リリース8.1.18](#)
E.102. [リリース8.1.17](#)
E.103. [リリース8.1.16](#)
E.104. [リリース8.1.15](#)
E.105. [リリース8.1.14](#)
E.106. [リリース8.1.13](#)
E.107. [リリース8.1.12](#)
E.108. [リリース8.1.11](#)

E.109. [リリース8.1.10](#)
E.110. [リリース8.1.9](#)
E.111. [リリース8.1.8](#)
E.112. [リリース8.1.7](#)
E.113. [リリース 8.1.6](#)
E.114. [リリース8.1.5](#)
E.115. [リリース8.1.4](#)
E.116. [リリース8.1.3](#)
E.117. [リリース8.1.2](#)
E.118. [リリース8.1.1](#)
E.119. [リリース8.1](#)
E.120. [リリース8.0.26](#)
E.121. [リリース8.0.25](#)
E.122. [リリース8.0.24](#)
E.123. [リリース8.0.23](#)
E.124. [リリース8.0.22](#)
E.125. [リリース8.0.21](#)
E.126. [リリース8.0.20](#)
E.127. [リリース8.0.19](#)
E.128. [リリース8.0.18](#)
E.129. [リリース8.0.17](#)
E.130. [リリース8.0.16](#)
E.131. [リリース8.0.15](#)
E.132. [リリース8.0.14](#)
E.133. [リリース8.0.13](#)
E.134. [リリース 8.0.12](#)
E.135. [リリース 8.0.11](#)
E.136. [リリース 8.0.10](#)
E.137. [リリース8.0.9](#)
E.138. [リリース8.0.8](#)
E.139. [リリース8.0.7](#)
E.140. [リリース8.0.6](#)
E.141. [リリース8.0.5](#)
E.142. [リリース 8.0.4](#)
E.143. [リリース8.0.3](#)
E.144. [リリース8.0.2](#)
E.145. [リリース8.0.1](#)
E.146. [リリース8.0](#)
E.147. [リリース7.4.30](#)
E.148. [リリース7.4.29](#)
E.149. [リリース7.4.28](#)
E.150. [リリース7.4.27](#)
E.151. [リリース7.4.26](#)
E.152. [リリース7.4.25](#)
E.153. [リリース7.4.24](#)
E.154. [リリース7.4.23](#)
E.155. [リリース7.4.22](#)
E.156. [リリース7.4.21](#)
E.157. [リリース7.4.20](#)
E.158. [リリース7.4.19](#)
E.159. [リリース7.4.18](#)
E.160. [リリース7.4.17](#)
E.161. [リリース 7.4.16](#)
E.162. [リリース 7.4.15](#)
E.163. [リリース7.4.14](#)
E.164. [リリース7.4.13](#)
E.165. [リリース7.4.12](#)
E.166. [リリース7.4.11](#)
E.167. [リリース7.4.10](#)
E.168. [リリース 7.4.9](#)
E.169. [リリース7.4.8](#)
E.170. [リリース7.4.7](#)
E.171. [リリース7.4.6](#)
E.172. [リリース 7.4.5](#)
E.173. [リリース 7.4.4](#)

E.174. [リリース7.4.3](#)
E.175. [リリース7.4.2](#)
E.176. [リリース7.4.1](#)
E.177. [リリース 7.4](#)
E.178. [リリース7.3.21](#)
E.179. [リリース7.3.20](#)
E.180. [リリース7.3.19](#)
E.181. [リリース 7.3.18](#)
E.182. [リリース 7.3.17](#)
E.183. [リリース7.3.16](#)
E.184. [リリース7.3.15](#)
E.185. [リリース7.3.14](#)
E.186. [リリース7.3.13](#)
E.187. [リリース7.3.12](#)
E.188. [リリース 7.3.11](#)
E.189. [リリース7.3.10](#)
E.190. [リリース7.3.9](#)
E.191. [リリース7.3.8](#)
E.192. [リリース7.3.7](#)
E.193. [リリース7.3.6](#)
E.194. [リリース 7.3.5](#)
E.195. [リリース 7.3.4](#)
E.196. [リリース 7.3.3](#)
E.197. [リリース 7.3.2](#)
E.198. [リリース 7.3.1](#)
E.199. [リリース 7.3](#)
E.200. [リリース7.2.8](#)
E.201. [リリース7.2.7](#)
E.202. [リリース7.2.6](#)
E.203. [リリース7.2.5](#)
E.204. [リリース7.2.4](#)
E.205. [リリース 7.2.3](#)
E.206. [リリース 7.2.2](#)
E.207. [リリース 7.2.1](#)
E.208. [リリース 7.2](#)
E.209. [リリース 7.1.3](#)
E.210. [リリース 7.1.2](#)
E.211. [リリース 7.1.2](#)
E.212. [リリース 7.1](#)
E.213. [リリース 7.0.3](#)
E.214. [リリース 7.0.2](#)
E.215. [リリース 7.0.1](#)
E.216. [リリース 7.0](#)
E.217. [リリース 6.5.3](#)
E.218. [リリース 6.5.2](#)
E.219. [リリース 6.5.1](#)
E.220. [リリース 6.5](#)
E.221. [リリース 6.4.2](#)
E.222. [リリース 6.4.1](#)
E.223. [リリース 6.4](#)
E.224. [リリース 6.3.2](#)
E.225. [リリース 6.3.1](#)
E.226. [リリース 6.3](#)
E.227. [リリース 6.2.1](#)
E.228. [リリース 6.2](#)
E.229. [リリース 6.1.1](#)
E.230. [リリース 6.1](#)
E.231. [リリース 6.0](#)
E.232. [リリース 1.09](#)
E.233. [リリース 1.02](#)
E.234. [リリース 1.01](#)
E.235. [リリース 1.0](#)
E.236. [Postgres95 リリース 0.03](#)
E.237. [Postgres95 リリース 0.02](#)
E.238. [Postgres95 リリース 0.01](#)

F. [追加で提供されるモジュール](#)

- F.1. [adminpack](#)
- F.2. [auth_delay](#)
- F.3. [auto_explain](#)
- F.4. [btree_gin](#)
- F.5. [btree_gist](#)
- F.6. [chpasswd](#)
- F.7. [citext](#)
- F.8. [cube](#)
- F.9. [dblink](#)
- F.10. [dict_int](#)
- F.11. [dict_xsyn](#)
- F.12. [dummy_seclabel](#)
- F.13. [earthdistance](#)
- F.14. [file_fdw](#)
- F.15. [fuzzystrmatch](#)
- F.16. [hstore](#)
- F.17. [intagg](#)
- F.18. [intarray](#)
- F.19. [isn](#)
- F.20. [lo](#)
- F.21. [ltree](#)
- F.22. [pageinspect](#)
- F.23. [パスワード検査](#)
- F.24. [pg_buffercache](#)
- F.25. [pgcrypto](#)
- F.26. [pg_freespacemap](#)
- F.27. [pgrowlocks](#)
- F.28. [pg_stat_statements](#)
- F.29. [pgstattuple](#)
- F.30. [pg_trgm](#)
- F.31. [seg](#)
- F.32. [sepgsql](#)
- F.33. [spi](#)
- F.34. [sslinfo](#)
- F.35. [tablefunc](#)
- F.36. [tcn](#)
- F.37. [test_parser](#)
- F.38. [tsearch2](#)
- F.39. [unaccent](#)
- F.40. [uuid-oss](#)
- F.41. [xml2](#)

G. [追加で提供されるプログラム](#)

- G.1. [クライアントアプリケーション](#)
- G.2. [サーバアプリケーション](#)

H. [外部プロジェクト](#)

- H.1. [クライアントインタフェース](#)
- H.2. [管理ツール](#)
- H.3. [手続き言語](#)
- H.4. [拡張](#)

I. [ソースコードリポジトリ](#)

- I.1. [Gitを使ってソースを入手する](#)

J. [ドキュメント作成](#)

- J.1. [DocBook](#)
- J.2. [ツールセット](#)
- J.3. [文書の構築](#)

- J.4. [文書の起草](#)
- J.5. [スタイルガイド](#)

K. [頭字語](#)

[前のページ](#)
行推定の例

[ホーム](#)

[次のページ](#)
PostgreSQLエラーコード

付録 A. PostgreSQLエラーコード

PostgreSQLサーバによって発行されるメッセージは全て、標準SQLにおける"SQLSTATE"コードの記述方法に従った、5文字のエラーコードが割り当てられています。どのようなエラー条件が発生したかを把握しなければならないアプリケーションは、通常テキスト形式のエラーメッセージを確認するのではなく、このエラーコードを検査すべきです。このエラーコードは、PostgreSQLリリースの違いによって変更することはあまりありません。また、エラーメッセージの各国言語化による変更にも影響されません。PostgreSQLで発行されるエラーコードのいくつか（全部ではありません）は、標準SQLで定義されていることに注意してください。標準SQLで定義されていない追加のエラーコードは、独自のものであったり他のデータベースから取り入れたものです。

標準に従い、エラーコードの最初の2文字はエラーのクラスを表し、残り3文字がそのクラス内の特定条件を表します。したがって、特定のエラーコードを認識できないアプリケーションであっても、エラークラスから何をすべきか推定することができます。

[表A-1](#)は、PostgreSQL 9.2.4で定義されたエラーコードを全て一覧で示しています（標準SQLでは定義されていますが、現在実際に使用されていないものもあります）。エラークラスも示しています。各エラークラスには、残りの3文字が000となる"標準"エラーコードが存在します。あるクラスの範囲内で発生したが、より特定のコードが割り当てられていないエラー条件のためだけに、このコードが使用されます。

"Condition Name"列に示されているシンボルはPL/pgSQLで使用している条件名です。条件名は大文字もしくは小文字でも記述されます。（PL/pgSQLは、エラーの場合と異なり、警告の場合にはその状態名を認識しません。これらはクラス00と01と02です）。

表 A-1. PostgreSQLエラーコード

エラーコード	条件名
クラス 00 — 正常終了	
00000	successful_completion
クラス 01 — 警告	
01000	warning
0100C	dynamic_result_sets_returned
01008	implicit_zero_bit_padding
01003	null_value_eliminated_in_set_function
01007	privilege_not_granted
01006	privilege_not_revoked
01004	string_data_right_truncation
01P01	deprecated_feature
クラス 02 — データがない。(SQL標準ではこれは警告クラス)	
02000	no_data
02001	no_additional_dynamic_result_sets_returned
クラス 03 — SQL文の未完了	
03000	sql_statement_not_yet_complete
クラス 08 — 接続の例外	
08000	connection_exception
08003	connection_does_not_exist
08006	connection_failure
08001	sqlclient_unable_to_establish_sqlconnection
08004	sqlserver_rejected_establishment_of_sqlconnection
08007	transaction_resolution_unknown
08P01	protocol_violation
クラス 09 — トリガによるアクションの例外	
09000	triggered_action_exception
Class 0A — サポートされない機能	
0A000	feature_not_supported
Class 0B — 無効なトランザクションの初期	
0B000	invalid_transaction_initiation
Class 0F — ロケータの例外	

0F000	locator_exception
0F001	invalid_locator_specification
Class 0L — 無効な権限付与	
0L000	invalid_grantor
0LP01	invalid_grant_operation
Class 0P — 無効なロールの指定	
0P000	invalid_role_specification
Class 0Z — Diagnostics例外	
0Z000	diagnostics_exception
0Z002	stacked_diagnostics_accessed_without_active_handler
Class 20 — Caseが存在しない	
20000	case_not_found
Class 21 — 次数違反	
21000	cardinality_violation
Class 22 — データ例外	
22000	data_exception
2202E	array_subscript_error
22021	character_not_in_repertoire
22008	datetime_field_overflow
22012	division_by_zero
22005	error_in_assignment
2200B	escape_character_conflict
22022	indicator_overflow
22015	interval_field_overflow
2201E	invalid_argument_for_logarithm
22014	invalid_argument_for_ntile_function
22016	invalid_argument_for_nth_value_function
2201F	invalid_argument_for_power_function
2201G	invalid_argument_for_width_bucket_function
22018	invalid_character_value_for_cast
22007	invalid_datetime_format
22019	invalid_escape_character
2200D	invalid_escape_octet
22025	invalid_escape_sequence
22P06	nonstandard_use_of_escape_character
22010	invalid_indicator_parameter_value
22023	invalid_parameter_value
2201B	invalid_regular_expression
2201W	invalid_row_count_in_limit_clause
2201X	invalid_row_count_in_result_offset_clause
22009	invalid_time_zone_displacement_value
2200C	invalid_use_of_escape_character
2200G	most_specific_type_mismatch
22004	null_value_not_allowed
22002	null_value_no_indicator_parameter
22003	numeric_value_out_of_range
22026	string_data_length_mismatch
22001	string_data_right_truncation
22011	substring_error
22027	trim_error
22024	unterminated_c_string
2200F	zero_length_character_string
22P01	floating_point_exception
22P02	invalid text representation
22P03	invalid binary representation
22P04	bad_copy_file_format
22P05	untranslatable_character
2200L	not_an_xml_document
2200M	invalid xml document
2200N	invalid xml content
2200S	invalid xml comment
2200T	invalid_xml_processing_instruction

Class 23 — 整合性制約違反	
23000	integrity_constraint_violation
23001	restrict_violation
23502	not_null_violation
23503	foreign_key_violation
23505	unique_violation
23514	check_violation
23P01	exclusion_violation
Class 24 — 無効なカーソル状態	
24000	invalid_cursor_state
Class 25 — 無効なトランザクション状態	
25000	invalid transaction state
25001	active_sql_transaction
25002	branch transaction already active
25008	held cursor requires same isolation level
25003	inappropriate access mode for branch transaction
25004	inappropriate isolation level for branch transaction
25005	no active sql transaction for branch transaction
25006	read only sql transaction
25007	schema and data statement mixing not supported
25P01	no active sql transaction
25P02	in failed sql transaction
Class 26 — 無効なSQL文の名前	
26000	invalid sql statement name
Class 27 — トリガによるデータ変更違反	
27000	triggered_data_change_violation
Class 28 — 無効な認証指定	
28000	invalid_authorization_specification
28P01	invalid_password
Class 2B — 依存する権限記述子がまだ存在する	
2B000	dependent_privilege_descriptors_still_exist
2BP01	dependent_objects_still_exist
Class 2D — 無効なトランザクションの終了	
2D000	invalid_transaction_termination
Class 2F — SQL関数例外	
2F000	sql_routine_exception
2F005	function_executed_no_return_statement
2F002	modifying_sql_data_not_permitted
2F003	prohibited_sql_statement_attempted
2F004	reading_sql_data_not_permitted
Class 34 — 無効なカーソル名称	
34000	invalid_cursor_name
Class 38 — 外部関数例外	
38000	external_routine_exception
38001	containing_sql_not_permitted
38002	modifying_sql_data_not_permitted
38003	prohibited_sql_statement_attempted
38004	reading_sql_data_not_permitted
Class 39 — 外部関数呼び出し例外	
39000	external_routine_invocation_exception
39001	invalid_sqlstate_returned
39004	値は許されていない
39P01	trigger_protocol_violated
39P02	srf_protocol_violated
Class 3B — セーブポイント例外	
3B000	savepoint_exception
3B001	invalid_savepoint_specification
Class 3D — 無効なカタログ名称	
3D000	invalid_catalog_name

Class 3F — 無効なスキーマ名称	
3F000	invalid_schema_name
Class 40 — トランザクションロールバック	
40000	transaction_rollback
40002	transaction_integrity_constraint_violation
40001	serialization_failure
40003	statement_completion_unknown
40P01	deadlock_detected
Class 42 — 構文エラー、もしくはアクセスロール違反	
42000	syntax_error_or_access_rule_violation
42601	syntax_error
42501	insufficient_privilege
42846	cannot_coerce
42803	grouping_error
42P20	windowing_error
42P19	invalid_recursion
42830	invalid_foreign_key
42602	invalid_name
42622	name_too_long
42939	reserved_name
42804	datatype_mismatch
42P18	indeterminate_datatype
42P21	collation_mismatch
42P22	indeterminate_collation
42809	wrong_object_type
42703	undefined_column
42883	undefined_function
42P01	undefined_table
42P02	undefined_parameter
42704	undefined_object
42701	duplicate_column
42P03	duplicate_cursor
42P04	duplicate_database
42723	duplicate_function
42P05	duplicate_prepared_statement
42P06	duplicate_schema
42P07	duplicate_table
42712	duplicate_alias
42710	duplicate_object
42702	ambiguous_column
42725	ambiguous_function
42P08	ambiguous_parameter
42P09	ambiguous_alias
42P10	invalid_column_reference
42611	invalid_column_definition
42P11	invalid_cursor_definition
42P12	invalid_database_definition
42P13	invalid_function_definition
42P14	invalid_prepared_statement_definition
42P15	invalid_schema_definition
42P16	invalid_table_definition
42P17	invalid_object_definition
Class 44 — 検査オプションに伴う違反	
44000	with_check_option_violation
Class 53 — リソース不足	
53000	insufficient_resources
53100	disk_full
53200	out_of_memory
53300	too_many_connections
53400	configuration_limit_exceeded
Class 54 — Program Limit Exceeded	
54000	program_limit_exceeded
54001	statement_too_complex

54011	too_many_columns
54023	too_many_arguments
Class 55 — 必要条件を満たさないオブジェクト	
55000	object_not_in_prerequisite_state
55006	object_in_use
55P02	cant_change_runtime_param
55P03	lock_not_available
Class 57 — 操作の介入	
57000	operator_intervention
57014	query_canceled
57P01	admin_shutdown
57P02	crash_shutdown
57P03	cannot_connect_now
57P04	database_dropped
Class 58 — システムエラー（外部原因によるPostgreSQL自体のエラー）	
58000	system_error
58030	io_error
58P01	undefined_file
58P02	duplicate_file
Class F0 — 設定ファイルエラー	
F0000	config_file_error
F0001	lock_file_exists
Class HV — 外部データラッパーエラー (SQL/MED)	
HV000	fdw_error
HV005	fdw_column_name_not_found
HV002	fdw_dynamic_parameter_value_needed
HV010	fdw_function_sequence_error
HV021	fdw_inconsistent_descriptor_information
HV024	fdw_invalid_attribute_value
HV007	fdw_invalid_column_name
HV008	fdw_invalid_column_number
HV004	fdw_invalid_data_type
HV006	fdw_invalid_data_type_descriptors
HV091	fdw_invalid_descriptor_field_identifier
HV00B	fdw_invalid_handle
HV00C	fdw_invalid_option_index
HV00D	fdw_invalid_option_name
HV090	fdw_invalid_string_length_or_buffer_length
HV00A	fdw_invalid_string_format
HV009	fdw_invalid_use_of_null_pointer
HV014	fdw_too_many_handles
HV001	fdw_out_of_memory
HV00P	fdw_no_schemas
HV00J	fdw_option_name_not_found
HV00K	fdw_reply_handle
HV00Q	fdw_schema_not_found
HV00R	fdw_table_not_found
HV00L	fdw_unable_to_create_execution
HV00M	fdw_unable_to_create_reply
HV00N	fdw_unable_to_establish_connection
Class P0 — PL/pgSQLエラー	
P0000	plpgsql_error
P0001	raise_exception
P0002	no_data_found
P0003	too_many_rows
Class XX — Internal Error	
XX000	internal_error
XX001	data_corrupted
XX002	index_corrupted

付録 B. 日付/時刻のサポート

目次

- B.1. [日付/時刻入力の解釈](#)
- B.2. [日付/時刻キーワード](#)
- B.3. [日付/時刻設定ファイル](#)
- B.4. [単位の歴史](#)

PostgreSQLは、全ての日付/時刻入力のサポートにおいて、内蔵しているヒューリスティックなパーサを使用します。日付と時刻は文字列で入力され、そのフィールドにはどのような種類の情報が入るのが事前に決められている別個のフィールドに分割されます。それぞれのフィールドは解釈された後、数値を割り当てられたり、無視されたり、あるいははねられたりします。構文解析に際し、月、曜日、および時間帯を含む、全てのテキストフィールドに対する内部参照テーブルがあります。

この付録ではこれらの参照テーブルの内容についての情報と構文解析で日付と時刻を解読する手順を説明します。

[前のページ](#)

PostgreSQLエラーコード

[ホーム](#)
[上に戻る](#)[次のページ](#)

日付/時刻入力の解釈

B.1. 日付/時刻入力の解釈

日付/時刻データ型は全て以下の手順で解釈されます。

- 1 11 入力文字列をトークンに分割し、そしてそれぞれのトークンを文字列、時刻、時間帯、または数値というように分類します。
 - 1 11 数値トークンにコロン (:) が含まれている場合は、時刻文字列です。そこに続く全ての数字とコロンを含みます。
 - 1 11 数値トークンにハイフン (-)、スラッシュ (/)、または2つ以上のドット (.) が含まれている場合は、テキストの月名がある日付文字列です。日付トークンがすでに現れている場合は代わりに、時間帯名として解釈されます (例えばAmerica/New_York)。
 - 1 11 トークンが数値だけの場合、それは単項、もしくはISO 8601の連結形式の日付 (例: 1999年1月13日を示す19990113)、あるいは時刻 (例: 14:15:16を示す141516) のいずれかです。
 - 1 11 トークンがプラス記号 (+) あるいはマイナス記号 (-) で始まっている場合は、数値形式の時間帯フィールドか特殊なフィールドです。
- 1 11 もしトークンがテキスト文字列の場合、以下のように可能性のある文字列と照合されます。
 - 1 11 時間帯短縮形としてトークンをテーブルからバイナリ検索します。
 - 1 11 探索できなかった場合は同様にそのトークンに対し特殊文字 (たとえばtoday)、日 (たとえばThursday)、月 (たとえばJanuary)、ノイズ (たとえばat、on) に一致するかどうか、参照テーブルにバイナリ検索をかけます。
 - 1 11 それでも探し出せなかった場合、エラーを返します。
- 1 11 トークンが数値あるいは数値フィールドの場合を以下に示します。
 - 1 11 トークンが8桁または6桁、かつ、以前に他のどのような日付フィールドも読まれていない場合は、"連結された日付" (例えば、19990118または990118) として解釈されます。その解釈方法はYYYYMMDDまたはYYMMDDです。
 - 1 11 もしトークンが3桁で年が既に読み込まれている場合は年内の経過日数と解釈されます。
 - 1 11 4桁または6桁の場合で年が既に読み込まれている時は時刻 (HHMMまたはHHMMSS) と解釈されます。
 - 1 11 3桁以上の場合、かつ、どの日付フィールドもまだ見つからない場合は年と解釈されます (この場合、残る日付フィールドの順序は強制的にyy-mm-ddと解釈されます)。
 - 1 11 さもないと、日付フィールドの順序は、DateStyleの設定mm-dd-yy、dd-mm-yy、yy-mm-ddに従うものと仮定されます。月や月内の日のフィールドが範囲外であれば、エラーになります。
- 1 11 もしBCが指定された場合は内部格納用に年を負の数にして1を加えます (グレゴリオ暦にはゼロ年がないので、数値的には1BC (紀元前1年) がゼロ年になります)。
- 1 11 BCが指定されず年フィールドの長さが2桁の場合、年は4桁になるよう調整されます。そのフィールドが70未満の場合は2000が加えられますが、その他の場合には1900が加えられます。

ティップ: (例えば、西暦99年を0099のように) グレゴリオ暦の西暦元年から99年までは、ゼロを前に付加して4桁で入力することができます。

B.2. 日付/時刻キーワード

[表B-1](#)に月名として認識されるトークンを示します。

表 B-1. 月名

月	簡略形
January	Jan
February	Feb
March	Mar
April	Apr
May	
June	Jun
July	Jul
August	Aug
September	Sep、Sept
October	Oct
November	Nov
December	Dec

[表B-2](#)に、曜日名として認識されるトークンを示します。

表 B-2. 曜日名

曜日	簡略形
Sunday	Sun
Monday	Mon
Tuesday	Tue、Tues
Wednesday	Wed、Weds
Thursday	Thu、Thur、Thurs
Friday	Fri
Saturday	Sat

[表B-3](#)に、様々な修飾子の目的を持つトークンを示します。

表 B-3. 日付/時刻フィールドの修飾子

識別子	説明
AM	12:00以前の時刻
AT	このキーワードは無視されます
JULIAN, JD, J	次のフィールドはユリウス日
ON	このキーワードは無視されます
PM	12:00と12:00以降の時刻
T	次のフィールドは時刻

B.3. 日付/時刻設定ファイル

時間帯省略形は十分に標準化されていないので、PostgreSQLでは、サーバで受け付け可能な省略形群をカスタマイズできる仕組みを提供します。 [timezone_abbreviations](#) 実行時パラメータは有効な省略形群を決定します。 このパラメータはすべてのデータベースユーザで変更可能ですが、取り得る値はデータベース管理者により制御されます。 これらは実際にはインストレーションディレクトリの `.../share/timezonesets/` 内の設定ファイル名です。 代替ファイルをこのディレクトリに追加することにより、管理者は時間帯省略形に対するローカルポリシーを設定することができます。

`timezone_abbreviations` は、 `.../share/timezonesets/` に存在する、名前のすべてがアルファベットである任意のファイルの名前を指定することができます。 (`timezone_abbreviations` 内の非アルファベット文字の禁止により、意図したディレクトリ以外のファイル読み取りを防ぐことができます。 また、バックアップファイルやその他のファイルの読み取りを防ぐこともできます。)

時間帯省略形ファイルには空白行や `#` から始まるコメントを含めることができます。 コメント以外は以下の書式を持たなければなりません。

```
time_zone_name offset
time_zone_name offset D
@INCLUDE file_name
@OVERRIDE
```

`time_zone_name` は単なる定義された省略形です。 `offset` は時間帯のUTCからのオフセットを秒数で表したものです。 グリニッジより東にあれば正、西にあれば負の値となります。 たとえば、`-18000` はグリニッジより西に5時間、すなわち、北アメリカ東海岸の標準時間を示します。 `D` は、時間帯名が標準時間ではなくローカル時間での夏時間を表すことを示します。 既知のすべての時間帯オフセットは15分刻みですので、秒数は900の倍数となります。

`@INCLUDE` 構文により、 `.../share/timezonesets/` ディレクトリ内の他のファイルを含むことができます。 深さに制限がありますが、入れ子に含むことができます。

`@OVERRIDE` 構文は、ファイル内の続く項目が既存の項目（つまり、インクルードされたファイルから得られた項目）を上書きできることを示します。 これがないと、同一時間帯省略形の定義が競合した場合にエラーとみなされます。

未変更のインストレーションでは、`Default` ファイルに、ほとんど全世界の競合しない時間帯省略形をすべて記載しています。 さらに `Australia` および `India` ファイルがこれらの地区向けに提供されています。 これらのファイルは `Default` ファイルの先頭で含有されています。 必要に応じて時間帯の追加・変更を行ってください。

参考のため、標準のインストレーションには `Africa.txt`、`America.txt` など含まれています。 これらには `zoneinfo` 時間帯データベースに従って使用されている時間帯省略形に関する情報がすべて含まれています。 これらのファイル内にある時間帯定義を必要に応じてコピーペーストして独自の設定ファイルを編集することができます。 これらのファイル名にドットが入っているので、`timezone_abbreviations` 設定から直接参照できないことに注意してください。

注意: 時間帯データを読み込む時にエラーが発生した場合、新しい値は適用されず、古い値がそのまま残ります。 データベースの起動時にエラーが起きた場合は、起動に失敗します。

注意

設定ファイル内で定義される時間帯省略形は、PostgreSQLに組み込み済みの時間帯以外の意味も変更します。 たとえば、`Australia` 設定ファイルでは `SAT`（南オーストラリア標準時間）を定義しています。 このファイルが有効な場合、`SAT` は土曜の省略形として認識されなくなります。

注意

`.../share/timezonesets/` を変更する時にバックアップを責任を持って行ってください。 このディレクトリは通常のデータベースダンプに含まれません。

B.4. 単位の歴史

標準SQLでは、「"日付時刻リテラル"定義の中で、"日付時刻の値"はグレゴリオ暦に従った日付と時間の自然法則に則る」と明記されています。PostgreSQLは標準SQLの指針に従い、グレゴリオ暦が使われる以前の年に対してもグレゴリオ暦で日付を数えます。この規則は先発グレゴリオ暦として知られています。

ユリウス暦は、紀元前45年にユリウス・カエサル (Julius Caesar) によって広められたものです。西欧でグレゴリオ暦への移行が開始された1582年まで一般的に使用されていました。ユリウス暦では、太陽年は $365\text{日} + 1/4\text{日} = 365.25\text{日}$ と概算されます。この暦では、128年で約1日のずれが生じます。

ローマ教皇グレゴリウス13世 (Gregory XIII) はトレントの公会議 (Council of Trent) の勧告に従って累積していた暦のずれを修正しました。グレゴリオ暦では、太陽年は $365 + 97/400\text{日} = 365.2425\text{日}$ と近似されます。したがって、グレゴリオ暦で太陽年が1日ずれるにはおよそ3,300年を要します。

$365 + 97/400$ という近似は、下記の規則に従って400年間に97回のうるう年を設けることによって得られています。

4で割り切れる年を、うるう年にする。

ただし、100で割り切れる年は、うるう年にしない。

ただし、400で割り切れる年は、結局うるう年とする。

したがって、1700、1800、1900、2100、2200はうるう年ではありませんが、2000、2400はうるう年です。それに比べ、古いユリウス暦では4で割り切れる年のみがうるう年でした。

1582年2月の教皇勅書は、1582年の10月から10日間除外することを命じ、したがって10月4日の翌日を10月15日としました。この慣行はイタリア、ポーランド、ポルトガル、スペインで遵守されました。他のカトリックの国々もすぐ後に追従しましたが、プロテスタントの国々は変更を嫌がり、ギリシャ正教を信奉する国々は20世紀の初めまで変更を行いませんでした。1752年に大英帝国とその自治領 (現在のアメリカ合衆国を含む) でもその改革は行われました。したがって、1752年9月2日の次は1752年9月14日となっています。このような理由から、Unixシステムで`cal`プログラムを実行すると、下記のような結果になります。

```
$ cal 9 1752
    September 1752
   S  M  Tu  W  Th  F  S
        1   2  14  15  16
  17  18  19  20  21  22  23
  24  25  26  27  28  29  30
```

しかし、当然、この暦は大英帝国とその自治領でのみ有効なものであり、他の場所ではそうではありません。様々な場所で様々な時代に使われていた実際の暦の追いかけることは難しく、また、混乱することになるでしょうから、PostgreSQLは追いかけることはせず、この方法が歴史的に正確でなくても日付すべてでグレゴリオ暦の規則に従います。

グレゴリオ暦が開発されるより前から、様々な暦が世界の多くの地域で開発されました。例えば、中国暦のルーツは紀元前14世紀まで遡ることができますし、伝説では、黄帝が紀元前2637年にこの暦を発明したとされています。中華人民共和国では、公的な目的ではグレゴリオ暦を使用していますが、祝祭日の決定には中国暦を使用します。

ユリウス日は別の種類の暦であり、名前が似ていて混乱しますが、ユリウス暦とは関係ありません。ユリウス日は、フランスの学者Joseph Justus Scaliger (1540-1609) によって発明され、おそらくこの語源は彼の父であるイタリアの学者、Julius Caesar Scaliger (1484-1558) からの引用と考えられます。ユリウス日システムでは、JD 0 (よくいわれるユリウス日と呼ばれます) から始まる日は連番です。JD 0はユリウス暦の紀元前4713年1月1日、またはグレゴリオ暦の紀元前4714年11月24日に対応します。ユリウス日の数え方は、天文学者が夜間の観測にラベルを付けるためにより良く使用されました。このため、深夜0時から深夜0時までではなく、UTC正午から次のUTC正午までが1日でした。JD 0は紀元前4714年11月24日のUTC正午から紀元前4714年11月25日のUTC正午までの24時間であると明示されています。

PostgreSQLは日付の入出力においてユリウス日記法を (および、一部の日付時間間隔の計算においてユリウス日を) サポートしますが、正午から正午までという微妙な日付の数え方を守っていません。PostgreSQLは深夜0時から深夜0時までとしてユリウス日を扱います。

付録 C. SQLキーワード

[表C-1](#)に標準SQLとPostgreSQL 9.2.4においてキーワードとされているすべてのトークンを示します。この背景となる情報は[項4.1.1](#)にあります。（スペースの都合上、標準SQLのより新しい2つのバージョンとと歴史的な比較のためのSQL-92のみを含めています。これらと他の標準の中間のバージョンの違いはわずかです。）

SQLは予約と未予約キーワードを区別します。標準に従うと予約キーワードのみが真のキーワードとなり、その予約キーワードは識別子として使用することはできません。未予約キーワードはある特定の文脈で特別な意味を持ち、また、その他の文脈では識別子として使用することができます。ほとんどの未予約キーワードは、SQLで規定された組み込みのテーブル名と関数名です。未予約キーワードの本質的な存在意義は、ある文脈においては前もって定義された意味があることを宣言することだけです。

PostgreSQLの構文解析の過程では、少々複雑になります。複数の異なったトークンのクラスがあり、それらは識別子としてまったく使用できないものから、普通の識別子と比較して、パーサ内で特別なステータスを持たないものまであります。（後者は一般的にはSQLで規定された関数です。）PostgreSQLでは予約キーワードは完全に予約されたものではなく、列ラベルとして使用することができます（例：CHECKは予約キーワードですが、SELECT 55 AS CHECKのようにすることができます）。

[表C-1](#)のPostgreSQLの列では、パーサがはっきりと理解しているキーワードではあるが、列名やテーブル名としても使用できるものを"未予約"としています。キーワードの中には、どちらかといえば未予約であるが、関数名や型名として使用できないものもあり、そのように注記しています。（このようなキーワードのほとんどは、特殊な構文をもった組み込み済みの関数やデータ型を表しています。この関数または型は使用することができますが、ユーザによって定義し直すことはできません。）一方、"予約"とされるものは列名やテーブル名として使用できないトークンです。予約キーワードの中には関数名やデータ型として使用できるものもあります。この情報も以下の表に示しています。何も注記がなければ、予約キーワードは"AS"の列ラベル名としてしか使用することができません。

一般的な規則として、以下に示すキーワードのいずれかを識別子として含むコマンドで、おかしなパーサエラーが発生した場合、その識別子を引用符でくくって問題が解決するかどうか確認してください。

[表C-1](#)を見る前に、PostgreSQLにおいて予約されていないキーワードが、その単語に関連する機能を実装していないということを意味していないことを理解しておいてください。逆に、キーワードがあるということも機能が存在することを意味していません。

表 C-1. SQLキーワード

キーワード	PostgreSQL	SQL:2011	SQL:2008	SQL-92
A		未予約	未予約	
ABORT	未予約			
ABS		予約	予約	
ABSENT		未予約	未予約	
ABSOLUTE	未予約	未予約	未予約	予約
ACCESS	未予約			
ACCORDING		未予約	未予約	
ACTION	未予約	未予約	未予約	予約
ADA		未予約	未予約	未予約
ADD	未予約	未予約	未予約	予約
ADMIN	未予約	未予約	未予約	
AFTER	未予約	未予約	未予約	
AGGREGATE	未予約			
ALL	予約	予約	予約	予約
ALLOCATE		予約	予約	予約
ALSO	未予約			
ALTER	未予約	予約	予約	予約
ALWAYS	未予約	未予約	未予約	
ANALYSE	予約			

ANALYZE	予約			
AND	予約	予約	予約	予約
ANY	予約	予約	予約	予約
ARE	予約	予約	予約	予約
ARRAY	予約	予約	予約	
ARRAY_AGG		予約	予約	
ARRAY_MAX_CARDINALITY		予約		
AS	予約	予約	予約	予約
ASC	予約	未予約	未予約	予約
ASENSITIVE		予約	予約	
ASSERTION	未予約	未予約	未予約	予約
ASSIGNMENT	未予約	未予約	未予約	
ASYMMETRIC	予約	予約	予約	
AT	未予約	予約	予約	予約
ATOMIC		予約	予約	
ATTRIBUTE	未予約	未予約	未予約	
ATTRIBUTES		未予約	未予約	
AUTHORIZATION	予約（関数または型として使用可）	予約	予約	予約
AVG		予約	予約	予約
BACKWARD	未予約			
BASE64		未予約	未予約	
BEFORE	未予約	未予約	未予約	
BEGIN	未予約	予約	予約	予約
BEGIN_FRAME		予約		
BEGIN_PARTITION		予約		
BERNOULLI		未予約	未予約	
BETWEEN	未予約(関数または型として使用不可)	予約	予約	予約
BIGINT	未予約（関数または型として使用不可）	予約	予約	
BINARY	予約（関数または型として使用可）	予約	予約	
BIT	未予約（関数または型として使用不可）			予約
BIT_LENGTH				予約
BLOB		予約	予約	
BLOCKED		未予約	未予約	
BOM		未予約	未予約	
BOOLEAN	未予約（関数または型として使用不可）	予約	予約	
BOTH	予約	予約	予約	予約
BREADTH		未予約	未予約	
BY	未予約	予約	予約	予約
C		未予約	未予約	未予約
CACHE	未予約			
CALL		予約	予約	
CALLED	未予約	予約	予約	
CARDINALITY		予約	予約	
CASCADE	未予約	未予約	未予約	予約
CASCADEED	未予約	予約	予約	予約
CASE	予約	予約	予約	予約
CAST	予約	予約	予約	予約

CATALOG	未予約	未予約	未予約	予約
CATALOG_NAME		未予約	未予約	未予約
CEIL		予約	予約	
CEILING		予約	予約	
CHAIN	未予約	未予約	未予約	
CHAR	未予約（関数または型として使用不可）	予約	予約	予約
CHARACTER	未予約（関数または型として使用不可）	予約	予約	予約
CHARACTERISTICS	未予約	未予約	未予約	
CHARACTERS		未予約	未予約	
CHARACTER_LENGTH		予約	予約	予約
CHARACTER_SET_CATALOG		未予約	未予約	未予約
CHARACTER_SET_NAME		未予約	未予約	未予約
CHARACTER_SET_SCHEMA		未予約	未予約	未予約
CHAR_LENGTH		予約	予約	予約
CHECK	予約	予約	予約	予約
CHECKPOINT	未予約			
CLASS	未予約			
CLASS_ORIGIN		未予約	未予約	未予約
CLOB		予約	予約	
CLOSE	未予約	予約	予約	予約
CLUSTER	未予約			
COALESCE	未予約（関数または型として使用不可）	予約	予約	予約
COBOL		未予約	未予約	未予約
COLLATE	予約	予約	予約	予約
COLLATION	予約（関数または型として使用可）	未予約	未予約	予約
COLLATION_CATALOG		未予約	未予約	未予約
COLLATION_NAME		未予約	未予約	未予約
COLLATION_SCHEMA		未予約	未予約	未予約
COLLECT		予約	予約	
COLUMN	予約	予約	予約	予約
COLUMNS		未予約	未予約	
COLUMN_NAME		未予約	未予約	未予約
COMMAND_FUNCTION		未予約	未予約	未予約
COMMAND_FUNCTION_CODE		未予約	未予約	
COMMENT	未予約			
COMMENTS	未予約			
COMMIT	未予約	予約	予約	予約
COMMITTED	未予約	未予約	未予約	未予約
CONCURRENTLY	予約（関数または型として使用）			
CONDITION		予約	予約	
CONDITION_NUMBER		未予約	未予約	未予約
CONFIGURATION	未予約			
CONNECT		予約	予約	予約
CONNECTION	未予約	未予約	未予約	予約
CONNECTION_NAME		未予約	未予約	未予約
CONSTRAINT	予約	予約	予約	予約
CONSTRAINTS	未予約	未予約	未予約	予約

CONSTRAINT_CATALOG		未予約	未予約	未予約
CONSTRAINT_NAME		未予約	未予約	未予約
CONSTRAINT_SCHEMA		未予約	未予約	未予約
CONSTRUCTOR		未予約	未予約	
CONTAINS		予約	未予約	
CONTENT	未予約	未予約	未予約	
CONTINUE	未予約	未予約	未予約	予約
CONTROL		未予約	未予約	
CONVERSION	未予約			
CONVERT		予約	予約	予約
COPY	未予約			
CORR		予約	予約	
CORRESPONDING		予約	予約	予約
COST	未予約			
COUNT		予約	予約	予約
COVAR_POP		予約	予約	
COVAR_SAMP		予約	予約	
CREATE	予約	予約	予約	予約
CROSS	予約（関数または型として使用可）	予約	予約	予約
CSV	未予約			
CUBE		予約	予約	
CUME_DIST		予約	予約	
CURRENT	未予約	予約	予約	予約
CURRENT_CATALOG	予約	予約	予約	
CURRENT_DATE	予約	予約	予約	予約
CURRENT_DEFAULT_TRANSFORM_GROUP		予約	予約	
CURRENT_PATH		予約	予約	
CURRENT_ROLE	予約	予約	予約	
CURRENT_ROW		予約		
CURRENT_SCHEMA	予約（関数または型として使用可）	予約	予約	
CURRENT_TIME	予約	予約	予約	予約
CURRENT_TIMESTAMP	予約	予約	予約	予約
CURRENT_TRANSFORM_GROUP_FOR_TYPE		予約	予約	
CURRENT_USER	予約	予約	予約	予約
CURSOR	未予約	予約	予約	予約
CURSOR_NAME		未予約	未予約	未予約
CYCLE	未予約	予約	予約	
DATA	未予約	未予約	未予約	未予約
DATABASE	未予約			
DATALINK		予約	予約	
DATE		予約	予約	予約
DATETIME_INTERVAL_CODE		未予約	未予約	未予約
DATETIME_INTERVAL_PRECISION		未予約	未予約	未予約
DAY	未予約	予約	予約	予約
DB		未予約	未予約	
DEALLOCATE	未予約	予約	予約	予約
DEC	未予約（関数または型として使用不可）	予約	予約	予約

DECIMAL	未予約（関数または型として使用不可）	予約	予約	予約
DECLARE	未予約	予約	予約	予約
DEFAULT	予約	予約	予約	予約
DEFAULTS	未予約	未予約	未予約	
DEFERRABLE	予約	未予約	未予約	予約
DEFERRED	未予約	未予約	未予約	予約
DEFINED		未予約	未予約	
DEFINER	未予約	未予約	未予約	
DEGREE		未予約	未予約	
DELETE	未予約	予約	予約	予約
DELIMITER	未予約			
DELIMITERS	未予約			
DENSE_RANK		予約	予約	
DEPTH		未予約	未予約	
DEREF		予約	予約	
DERIVED		未予約	未予約	
DESC	予約	未予約	未予約	予約
DESCRIBE		予約	予約	予約
DESCRIPTOR		未予約	未予約	予約
DETERMINISTIC		予約	予約	
DIAGNOSTICS		未予約	未予約	予約
DICTIONARY	未予約			
DISABLE	未予約			
DISCARD	未予約			
DISCONNECT		予約	予約	予約
DISPATCH		未予約	未予約	
DISTINCT	予約	予約	予約	予約
DLNEWCOPY		予約	予約	
DLPREVIOUSCOPY		予約	予約	
DLURLCOMPLETE		予約	予約	
DLURLCOMPLETEONLY		予約	予約	
DLURLCOMPLETEWRITE		予約	予約	
DLURLPATH		予約	予約	
DLURLPATHONLY		予約	予約	
DLURLPATHWRITE		予約	予約	
DLURLSCHEME		予約	予約	
DLURLSERVER		予約	予約	
DLVALUE		予約	予約	
DO	予約			
DOCUMENT	未予約	未予約	未予約	
DOMAIN	未予約	未予約	未予約	予約
DOUBLE	未予約	予約	予約	予約
DROP	未予約	予約	予約	予約
DYNAMIC		予約	予約	
DYNAMIC_FUNCTION		未予約	未予約	未予約
DYNAMIC_FUNCTION_CODE		未予約	未予約	
EACH	未予約	予約	予約	

ELEMENT		予約	予約	
ELSE	予約	予約	予約	予約
EMPTY		未予約	未予約	
ENABLE	未予約			
ENCODING	未予約	未予約	未予約	
ENCRYPTED	未予約			
END	予約	予約	予約	予約
END-EXEC		予約	予約	予約
END_FRAME		予約		
END_PARTITION		予約		
ENFORCED		未予約		
ENUM	未予約			
EQUALS		予約	未予約	
ESCAPE	未予約	予約	予約	予約
EVERY		予約	予約	
EXCEPT	予約	予約	予約	予約
EXCEPTION				予約
EXCLUDE	未予約	未予約	未予約	
EXCLUDING	未予約	未予約	未予約	
EXCLUSIVE	未予約			
EXEC		予約	予約	予約
EXECUTE	未予約	予約	予約	予約
EXISTS	未予約（関数または型として使用不可）	予約	予約	予約
EXP		予約	予約	
EXPLAIN	未予約			
EXPRESSION		未予約		
EXTENSION	未予約			
EXTERNAL	未予約	予約	予約	予約
EXTRACT	未予約（関数または型として使用不可）	予約	予約	予約
FALSE	予約	予約	予約	予約
FAMILY	未予約			
FETCH	予約	予約	予約	予約
FILE		未予約	未予約	
FILTER		予約	予約	
FINAL		未予約	未予約	
FIRST	未予約	未予約	未予約	予約
FIRST_VALUE		予約	予約	
FLAG		未予約	未予約	
FLOAT	未予約（関数または型として使用不可）	予約	予約	予約
FLOOR		予約	予約	
FOLLOWING	未予約	未予約	未予約	
FOR	予約	予約	予約	予約
FORCE	未予約			
FOREIGN	予約	予約	予約	予約
FORTRAN		未予約	未予約	未予約
FORWARD	未予約			
FOUND		未予約	未予約	予約

FRAME_ROW		予約		
FREE		予約	予約	
FREEZE	予約（関数または型として使用可）			
FROM	予約	予約	予約	予約
FS		未予約	未予約	
FULL	予約（関数または型として使用可）	予約	予約	予約
FUNCTION	未予約	予約	予約	
FUNCTIONS	未予約			
FUSION		予約	予約	
G		未予約	未予約	
GENERAL		未予約	未予約	
GENERATED		未予約	未予約	
GET		予約	予約	予約
GLOBAL	未予約	予約	予約	予約
GO		未予約	未予約	予約
GOTO		未予約	未予約	予約
GRANT	予約	予約	予約	予約
GRANTED	未予約	未予約	未予約	
GREATEST	未予約（関数または型として使用不可）			
GROUP	予約	予約	予約	予約
GROUPING		予約	予約	
GROUPS		予約		
HANDLER	未予約			
HAVING	予約	予約	予約	予約
HEADER	未予約			
HEX		未予約	未予約	
HIERARCHY		未予約	未予約	
HOLD	未予約	予約	予約	
HOURL	未予約	予約	予約	予約
ID		未予約	未予約	
IDENTITY	未予約	予約	予約	予約
IF	未予約			
IGNORE		未予約	未予約	
ILIKE	予約（関数または型として使用可）			
IMMEDIATE	未予約	未予約	未予約	予約
IMMEDIATELY		未予約		
IMMUTABLE	未予約			
IMPLEMENTATION		未予約	未予約	
IMPLICIT	未予約			
IMPORT		予約	予約	
IN	予約	予約	予約	予約
INCLUDING	未予約	未予約	未予約	
INCREMENT	未予約	未予約	未予約	
INDENT		未予約	未予約	
INDEX	未予約			
INDEXES	未予約			
INDICATOR		予約	予約	予約

INHERIT	未予約			
INHERITS	未予約			
INITIALLY	予約	未予約	未予約	予約
INLINE	未予約			
INNER	予約（関数または型として使用可）	予約	予約	予約
INOUT	未予約（関数または型として使用不可）	予約	予約	
INPUT	未予約	未予約	未予約	予約
INSENSITIVE	未予約	予約	予約	予約
INSERT	未予約	予約	予約	予約
INSTANCE		未予約	未予約	
INSTANTIABLE		未予約	未予約	
INSTEAD	未予約	未予約	未予約	
INT	未予約（関数または型として使用不可）	予約	予約	予約
INTEGER	未予約（関数または型として使用不可）	予約	予約	予約
INTEGRITY		未予約	未予約	
INTERSECT	予約	予約	予約	予約
INTERSECTION		予約	予約	
INTERVAL	未予約（関数または型として使用不可）	予約	予約	予約
INTO	予約	予約	予約	予約
INVOKER	未予約	未予約	未予約	
IS	予約（関数または型として使用可）	予約	予約	予約
ISNULL	予約（関数または型として使用可）			
ISOLATION	未予約	未予約	未予約	予約
JOIN	予約（関数または型として使用可）	予約	予約	予約
K		未予約	未予約	
KEY	未予約	未予約	未予約	予約
KEY_MEMBER		未予約	未予約	
KEY_TYPE		未予約	未予約	
LABEL	未予約			
LAG		予約	予約	
LANGUAGE	未予約	予約	予約	予約
LARGE	未予約	予約	予約	
LAST	未予約	未予約	未予約	予約
LAST_VALUE		予約	予約	
LATERAL		予約	予約	
LC_COLLATE	未予約			
LC_CTYPE	未予約			
LEAD		予約	予約	
LEADING	予約	予約	予約	予約
LEAKPROOF	未予約			
LEAST	未予約（関数または型として使用不可）			
LEFT	予約（関数または型として使用可）	予約	予約	予約
LENGTH		未予約	未予約	未予約
LEVEL	未予約	未予約	未予約	予約
LIBRARY		未予約	未予約	
LIKE	予約（関数または型として使用可）	予約	予約	予約
LIKE_REGEX		予約	予約	

LIMIT	予約	未予約	未予約	
LINK		未予約	未予約	
LISTEN	未予約			
LN		予約	予約	
LOAD	未予約			
LOCAL	未予約	予約	予約	予約
LOCALTIME	予約	予約	予約	
LOCALTIMESTAMP	予約	予約	予約	
LOCATION	未予約	未予約	未予約	
LOCATOR		未予約	未予約	
LOCK	未予約			
LOWER		予約	予約	予約
M		未予約	未予約	
MAP		未予約	未予約	
MAPPING	未予約	未予約	未予約	
MATCH	未予約	予約	予約	予約
MATCHED		未予約	未予約	
MAX		予約	予約	予約
MAXVALUE	未予約	未予約	未予約	
MAX_CARDINALITY			予約	
MEMBER		予約	予約	
MERGE		予約	予約	
MESSAGE_LENGTH		未予約	未予約	未予約
MESSAGE_OCTET_LENGTH		未予約	未予約	未予約
MESSAGE_TEXT		未予約	未予約	未予約
METHOD		予約	予約	
MIN		予約	予約	予約
MINUTE	未予約	予約	予約	予約
MINVALUE	未予約	未予約	未予約	
MOD		予約	予約	
MODE	未予約			
MODIFIES		予約	予約	
MODULE		予約	予約	予約
MONTH	未予約	予約	予約	予約
MORE		未予約	未予約	未予約
MOVE	未予約			
MULTISET		予約	予約	
MUMPS		未予約	未予約	未予約
NAME	未予約	未予約	未予約	未予約
NAMES	未予約	未予約	未予約	予約
NAMESPACE		未予約	未予約	
NATIONAL	未予約（関数または型として使用不可）	予約	予約	予約
NATURAL	予約（関数または型として使用可）	予約	予約	予約
NCHAR	未予約（関数または型として使用不可）	予約	予約	予約
NCLOB		予約	予約	
NESTING		未予約	未予約	
NEW		予約	予約	

NEXT	未予約	未予約	未予約	予約
NFC		未予約	未予約	
NFD		未予約	未予約	
NFKC		未予約	未予約	
NFKD		未予約	未予約	
NIL		未予約	未予約	
NO	未予約	予約	予約	予約
NONE	未予約（関数または型として使用不可）	予約	予約	
NORMALIZE		予約	予約	
NORMALIZED		未予約	未予約	
NOT	予約	予約	予約	予約
NOTHING	未予約			
NOTIFY	未予約			
NOTNULL	予約（関数または型として使用可）			
NOWAIT	未予約			
NTH_VALUE		予約	予約	
NTILE		予約	予約	
NULL	予約	予約	予約	予約
NULLABLE		未予約	未予約	未予約
NULLIF	未予約（関数または型として使用不可）	予約	予約	予約
NULLS	未予約	未予約	未予約	
NUMBER		未予約	未予約	未予約
NUMERIC	未予約（関数または型として使用不可）	予約	予約	予約
OBJECT	未予約	未予約	未予約	
OCCURRENCES_REGEX		予約	予約	
OCTETS		未予約	未予約	
OCTET_LENGTH		予約	予約	予約
OF	未予約	予約	予約	予約
OFF	未予約	未予約	未予約	
OFFSET	予約	予約	予約	
OIDS	未予約			
OLD		予約	予約	
ON	予約	予約	予約	予約
ONLY	予約	予約	予約	予約
OPEN		予約	予約	予約
OPERATOR	未予約			
OPTION	未予約	未予約	未予約	予約
OPTIONS	未予約	未予約	未予約	
OR	予約	予約	予約	予約
ORDER	予約	予約	予約	予約
ORDERING		未予約	未予約	
ORDINALITY		未予約	未予約	
OTHERS		未予約	未予約	
OUT	未予約（関数または型として使用不可）	予約	予約	
OUTER	予約（関数または型として使用可）	予約	予約	予約
OUTPUT		未予約	未予約	予約
OVER	予約（関数または型として使用可）	予約	予約	

OVERLAPS	予約（関数または型として使用可）	予約	予約	予約
OVERLAY	未予約（関数または型として使用不可）	予約	予約	
OVERRIDING		未予約	未予約	
OWNED	未予約			
OWNER	未予約			
P		未予約	未予約	
PAD		未予約	未予約	予約
PARAMETER		予約	予約	
PARAMETER_MODE		未予約	未予約	
PARAMETER_NAME		未予約	未予約	
PARAMETER_ORDINAL_POSITION		未予約	未予約	
PARAMETER_SPECIFIC_CATALOG		未予約	未予約	
PARAMETER_SPECIFIC_NAME		未予約	未予約	
PARAMETER_SPECIFIC_SCHEMA		未予約	未予約	
PARSER	未予約			
PARTIAL	未予約	未予約	未予約	予約
PARTITION	未予約	予約	予約	
PASCAL		未予約	未予約	未予約
PASSING	未予約	未予約	未予約	
PASSTHROUGH		未予約	未予約	
PASSWORD	未予約			
PATH		未予約	未予約	
PERCENT		予約		
PERCENTILE_CONT		予約	予約	
PERCENTILE_DISC		予約	予約	
PERCENT_RANK		予約	予約	
PERIOD		予約		
PERMISSION		未予約	未予約	
PLACING	予約	未予約	未予約	
PLANS	未予約			
PLI		未予約	未予約	未予約
PORTION		予約		
POSITION	未予約（関数または型として使用不可）	予約	予約	予約
POSITION_REGEX		予約	予約	
POWER		予約	予約	
PRECEDES		予約		
PRECEDING	未予約	未予約	未予約	
PRECISION	未予約（関数または型として使用不可）	予約	予約	予約
PREPARE	未予約	予約	予約	予約
PREPARED	未予約			
PRESERVE	未予約	未予約	未予約	予約
PRIMARY	予約	予約	予約	予約
PRIOR	未予約	未予約	未予約	予約
PRIVILEGES	未予約	未予約	未予約	予約
PROCEDURAL	未予約			
PROCEDURE	未予約	予約	予約	予約
PUBLIC		未予約	未予約	予約

QUOTE	未予約			
RANGE	未予約	予約	予約	
RANK		予約	予約	
READ	未予約	未予約	未予約	予約
READS		予約	予約	
REAL	未予約（関数または型として使用不可）	予約	予約	予約
REASSIGN	未予約			
RECHECK	未予約			
RECOVERY		未予約	未予約	
RECURSIVE	未予約	予約	予約	
REF	未予約	予約	予約	
REFERENCES	予約	予約	予約	予約
REFERENCING		予約	予約	
REGR_AVGX		予約	予約	
REGR_AVGY		予約	予約	
REGR_COUNT		予約	予約	
REGR_INTERCEPT		予約	予約	
REGR_R2		予約	予約	
REGR_SLOPE		予約	予約	
REGR_SXX		予約	予約	
REGR_SXY		予約	予約	
REGR_SYY		予約	予約	
REINDEX	未予約			
RELATIVE	未予約	未予約	未予約	予約
RELEASE	未予約	予約	予約	
RENAME	未予約			
REPEATABLE	未予約	未予約	未予約	未予約
REPLACE	未予約			
REPLICA	未予約			
REQUIRING		未予約	未予約	
RESET	未予約			
RESPECT		未予約	未予約	
RESTART	未予約	未予約	未予約	
RESTORE		未予約	未予約	
RESTRICT	未予約	未予約	未予約	予約
RESULT		予約	予約	
RETURN		予約	予約	
RETURNED_CARDINALITY		未予約	未予約	
RETURNED_LENGTH		未予約	未予約	未予約
RETURNED_OCTET_LENGTH		未予約	未予約	未予約
RETURNED_SQLSTATE		未予約	未予約	未予約
RETURNING	予約	未予約	未予約	
RETURNS	未予約	予約	予約	
REVOKE	未予約	予約	予約	予約
RIGHT	予約（関数または型として使用可）	予約	予約	予約
ROLE	未予約	未予約	未予約	
ROLLBACK	未予約	予約	予約	予約

ROLLUP		予約	予約	
ROUTINE		未予約	未予約	
ROUTINE_CATALOG		未予約	未予約	
ROUTINE_NAME		未予約	未予約	
ROUTINE_SCHEMA		未予約	未予約	
ROW	未予約（関数または型として使用不可）	予約	予約	
ROWS	未予約	予約	予約	予約
ROW_COUNT		未予約	未予約	未予約
ROW_NUMBER		予約	予約	
RULE	未予約			
SAVEPOINT	未予約	予約	予約	
SCALE		未予約	未予約	未予約
SCHEMA	未予約	未予約	未予約	予約
SCHEMA_NAME		未予約	未予約	未予約
SCOPE		予約	予約	
SCOPE_CATALOG		未予約	未予約	
SCOPE_NAME		未予約	未予約	
SCOPE_SCHEMA		未予約	未予約	
SCROLL	未予約	予約	予約	予約
SEARCH	未予約	予約	予約	
SECOND	未予約	予約	予約	予約
SECTION		未予約	未予約	予約
SECURITY	未予約	未予約	未予約	
SELECT	予約	予約	予約	予約
SELECTIVE		未予約	未予約	
SELF		未予約	未予約	
SENSITIVE		予約	予約	
SEQUENCE	未予約	未予約	未予約	
SEQUENCES	未予約			
SERIALIZABLE	未予約	未予約	未予約	未予約
SERVER	未予約	未予約	未予約	
SERVER_NAME		未予約	未予約	未予約
SESSION	未予約	未予約	未予約	予約
SESSION_USER	予約	予約	予約	予約
SET	未予約	予約	予約	予約
SETOF	未予約（関数または型として使用不可）			
SETS		未予約	未予約	
SHARE	未予約			
SHOW	未予約			
SIMILAR	予約（関数または型として使用可）	予約	予約	
SIMPLE	未予約	未予約	未予約	
SIZE		未予約	未予約	予約
SMALLINT	未予約（関数または型として使用不可）	予約	予約	予約
SNAPSHOT	未予約			
SOME	予約	予約	予約	予約
SOURCE		未予約	未予約	
SPACE		未予約	未予約	予約

SPECIFIC		予約	予約	
SPECIFICTYPE		予約	予約	
SPECIFIC_NAME		未予約	未予約	
SQL		予約	予約	予約
SQLCODE				予約
SQLERROR				予約
SQLEXCEPTION		予約	予約	
SQLSTATE		予約	予約	予約
SQLWARNING		予約	予約	
SQRT		予約	予約	
STABLE	未予約			
STANDALONE	未予約	未予約	未予約	
START	未予約	予約	予約	
STATE		未予約	未予約	
STATEMENT	未予約	未予約	未予約	
STATIC		予約	予約	
STATISTICS	未予約			
STDDEV_POP		予約	予約	
STDDEV_SAMP		予約	予約	
STDIN	未予約			
STDOUT	未予約			
STORAGE	未予約			
STRICT	未予約			
STRIP	未予約	未予約	未予約	
STRUCTURE		未予約	未予約	
STYLE		未予約	未予約	
SUBCLASS_ORIGIN		未予約	未予約	未予約
SUBMULTISET		予約	予約	
SUBSTRING	未予約（関数または型として使用不可）	予約	予約	予約
SUBSTRING_REGEX		予約	予約	
SUCCEEDS		予約		
SUM		予約	予約	予約
SYMMETRIC	予約	予約	予約	
SYSID	未予約			
SYSTEM	未予約	予約	予約	
SYSTEM_TIME		予約		
SYSTEM_USER		予約	予約	予約
T		未予約	未予約	
TABLE	予約	予約	予約	予約
TABLES	未予約			
TABLESAMPLE		予約	予約	
TABLESPACE	未予約			
TABLE_NAME		未予約	未予約	未予約
TEMP	未予約			
TEMPLATE	未予約			
TEMPORARY	未予約	未予約	未予約	予約
TEXT	未予約			

THEN	予約	予約	予約	予約
TIES		未予約	未予約	
TIME	未予約（関数または型として使用不可）	予約	予約	予約
TIMESTAMP	未予約（関数または型として使用不可）	予約	予約	予約
TIMEZONE_HOUR		予約	予約	予約
TIMEZONE_MINUTE		予約	予約	予約
TO	予約	予約	予約	予約
TOKEN		未予約	未予約	
TOP_LEVEL_COUNT		未予約	未予約	
TRAILING	予約	予約	予約	予約
TRANSACTION	未予約	未予約	未予約	予約
TRANSACTIONS_COMMITTED		未予約	未予約	
TRANSACTIONS_ROLLED_BACK		未予約	未予約	
TRANSACTION_ACTIVE		未予約	未予約	
TRANSFORM		未予約	未予約	
TRANSFORMS		未予約	未予約	
TRANSLATE		予約	予約	予約
TRANSLATE_REGEX		予約	予約	
TRANSLATION		予約	予約	予約
TREAT	未予約（関数または型として使用不可）	予約	予約	
TRIGGER	未予約	予約	予約	
TRIGGER_CATALOG		未予約	未予約	
TRIGGER_NAME		未予約	未予約	
TRIGGER_SCHEMA		未予約	未予約	
TRIM	未予約（関数または型として使用不可）	予約	予約	予約
TRIM_ARRAY		予約	予約	
TRUE	予約	予約	予約	予約
TRUNCATE	未予約	予約	予約	
TRUSTED	未予約			
TYPE	未予約	未予約	未予約	未予約
TYPES	未予約			
UESCAPE		予約	予約	
UNBOUNDED	未予約	未予約	未予約	
UNCOMMITTED	未予約	未予約	未予約	未予約
UNDER		未予約	未予約	
UNENCRYPTED	未予約			
UNION	予約	予約	予約	予約
UNIQUE	予約	予約	予約	予約
UNKNOWN	未予約	予約	予約	予約
UNLINK		未予約	未予約	
UNLISTEN	未予約			
UNLOGGED	未予約			
UNNAMED		未予約	未予約	未予約
UNNEST		予約	予約	
UNTIL	未予約			
UNTYPED		未予約	未予約	
UPDATE	未予約	予約	予約	予約

UPPER		予約	予約	予約
URI		未予約	未予約	
USAGE		未予約	未予約	予約
USER	予約	予約	予約	予約
USER_DEFINED_TYPE_CATALOG		未予約	未予約	
USER_DEFINED_TYPE_CODE		未予約	未予約	
USER_DEFINED_TYPE_NAME		未予約	未予約	
USER_DEFINED_TYPE_SCHEMA		未予約	未予約	
USING	予約	予約	予約	予約
VACUUM	未予約			
VALID	未予約	未予約	未予約	
VALIDATE	未予約			
VALIDATOR	未予約			
VALUE	未予約	予約	予約	予約
VALUES	未予約（関数または型として使用不可）	予約	予約	予約
VALUE_OF		予約		
VARBINARY		予約	予約	
VARCHAR	未予約（関数または型として使用不可）	予約	予約	予約
VARIADIC	予約			
VARYING	未予約	予約	予約	予約
VAR_POP		予約	予約	
VAR_SAMP		予約	予約	
VERBOSE	予約（関数または型として使用可）			
VERSION	未予約	未予約	未予約	
VERSIONING		予約		
VIEW	未予約	未予約	未予約	予約
VOLATILE	未予約			
WHEN	予約	予約	予約	予約
WHENEVER		予約	予約	予約
WHERE	予約	予約	予約	予約
WHITESPACE	未予約	未予約	未予約	
WIDTH_BUCKET		予約	予約	
WINDOW	予約	予約	予約	
WITH	予約	予約	予約	予約
WITHIN		予約	予約	
WITHOUT	未予約	予約	予約	
WORK	未予約	未予約	未予約	予約
WRAPPER	未予約	未予約	未予約	
WRITE	未予約	未予約	未予約	予約
XML	未予約	予約	予約	
XMLAGG		予約	予約	
XMLATTRIBUTES	未予約（関数または型として使用不可）	予約	予約	
XMLBINARY		予約	予約	
XMLCAST		予約	予約	
XMLCOMMENT		予約	予約	
XMLCONCAT	未予約（関数または型として使用不可）	予約	予約	
XMLDECLARATION		未予約	未予約	

XMLDOCUMENT		予約	予約	
XMLELEMENT	未予約（関数または型として使用不可）	予約	予約	
XML EXISTS	未予約（関数または型として使用不可）	予約	予約	
XMLFOREST	未予約（関数または型として使用不可）	予約	予約	
XMLITERATE		予約	予約	
XMLNAMESPACES		予約	予約	
XMLPARSE	未予約（関数または型として使用不可）	予約	予約	
XMLPI	未予約（関数または型として使用不可）	予約	予約	
XMLQUERY		予約	予約	
XMLROOT	未予約（関数または型として使用不可）			
XMLSCHEMA		未予約	未予約	
XMLSERIALIZE	未予約（関数または型として使用不可）	予約	予約	
XMLTABLE		予約	予約	
XMLTEXT		予約	予約	
XMLVALIDATE		予約	予約	
YEAR	未予約	予約	予約	予約
YES	未予約	未予約	未予約	
ZONE	未予約	未予約	未予約	予約

[前のページ](#)
単位の歴史

[ホーム](#)
[上に戻る](#)

[次のページ](#)
SQLへの準拠

付録 D. SQLへの準拠

目次

- D.1. [サポートされている機能](#)
- D.2. [サポートされていない機能](#)

本節では、PostgreSQLがどの程度現在の標準SQLに準拠しているかについて、概要を説明します。以下の情報は互換性についての完全な説明ではありません。しかし、ユーザにとって十分適切かつ有用な詳細を主な話題としてここで示しています。

標準SQLの公式な名称は、ISO/IEC 9075 "Database Language SQL"です。この標準SQLの改訂バージョンは不定期にリリースされています。最も最近の改訂は2011年に行われました。この2011年版はISO/IEC 9075:2011、もしくは単にSQL:2011と呼ばれています。この前のバージョンはSQL:2008、SQL:2003、SQL:1999とSQL-92です。それぞれ前のバージョンを置き換えたバージョンですので、より昔のバージョンへの適合性についての主張には公的な利点がありません。PostgreSQLの開発では、伝統的な機能もしくは共通の考えと矛盾しないように、標準SQLの最新の公式バージョンに準拠させようとしています。標準SQLで必須とされた機能の多くは、多少構文や機能に違いはあるものの、サポートされています。さらに適合性を高めることが将来のリリースで期待されています。

SQL-92では、適合性について、Entry、Intermediate、Fullという3つの機能セットを定義しました。標準SQLの準拠をうたっているデータベース管理システムのほとんどは、Entryレベルのみに適合しています。IntermediateもしくはFullレベルの機能の全体的なセットは、非常に膨大になり過ぎるか、もしくは旧来の動作と競合するからです。

SQL:1999から、標準SQLは、SQL-92で見受けられた3レベルに機能を非効率的に分散させるのではなく、個々の機能を大規模な集合として定義されるようになりました。こうした機能の大規模なサブセットを、全てのSQL準拠の実装が提供しなければならない"コア"機能として表しています。いくつかの省略可能な機能は"パッケージ"という形にまとめられ、SQLの実装はこれに対して互換性をうたうことができるようになりました。つまり特定機能のグループへの準拠が可能です。

SQL:2003版以降の標準はまた、数個の部品に分かれています。それぞれ省略した名前を持ちます。この部品に連続した番号が付いていないことに注意してください。

- ISO/IEC 9075-1 Framework (SQL/Framework)
- ISO/IEC 9075-2 Foundation (SQL/Foundation)
- ISO/IEC 9075-3 Call Level Interface (SQL/CLI)
- ISO/IEC 9075-4 Persistent Stored Modules (SQL/PSM)
- ISO/IEC 9075-9 Management of External Data (SQL/MED)
- ISO/IEC 9075-10 Object Language Bindings (SQL/OLB)
- ISO/IEC 9075-11 Information and Definition Schemas (SQL/Schemata)
- ISO/IEC 9075-13 Routines and Types using the Java Language (SQL/JRT)
- ISO/IEC 9075-14 XML-related specifications (SQL/XML)

PostgreSQLのコア部分は 1、2、9、11、および14番の部分に対応しています。第3部分はODBCドライバを、そして第13部分はPL/Javaプラグインを網羅していますが、これらコンポーネントに対する正確な適合性は検証されていません。第4と第10部分は現時点でPostgreSQLに実装されていません。

PostgreSQLはSQL:2011の主な機能のほとんどをサポートします。完全なコアの互換性に必要な179の必須機能の内、PostgreSQLは少なくとも160個に適合します。さらに省略可能な機能を数多くサポートします。本書の執筆時点で、SQL:2011のコアに完全に適合したデータベース管理システムのバージョンはないということを、ここで言及しておくのは価値のあることかもしれません。

以下の2つの節では、PostgreSQLでサポートされているこれらの機能をリストし、その後にPostgreSQLでまだサポートされていないSQL:2011で定義された機能をリストしています。これら2つのリストはおおよそのものです。サポート対象であるとリストされている機能が些細な部分で準拠していない可能性があり、サポートされていないある機能の大部分が、実際には実装されている可能性があります。本書の主要な部分では、動作するものとししないものについての最も正確な情報を常に伝えます。

注意: ハイフンを含む機能コードはサブ機能です。したがって、特定のサブ機能がサポートされない場合、他のサブ機能がサポートされていてもそのメイン機能はサポートされない機能としてリストされています。

D.1. サポートされている機能

識別子	パッケージ	説明	コメント
B012		埋め込みC	
B021		直接SQL	
E011	コア	数値データ型	
E011-01	コア	INTEGERおよびSMALLINTデータ型	
E011-02	コア	REAL、DOUBLE PRECISION、FLOATデータ型	
E011-03	コア	DECIMALおよびNUMERICデータ型	
E011-04	コア	算術演算子	
E011-05	コア	数値比較	
E011-06	コア	数値データ型間の暗黙キャスト	
E021	コア	文字データ型	
E021-01	コア	CHARACTERデータ型	
E021-02	コア	CHARACTER VARYINGデータ型	
E021-03	コア	文字リテラル	
E021-04	コア	CHARACTER_LENGTH関数	数える前にCHARACTER値の最後の空白を除去します。
E021-05	コア	OCTET_LENGTH関数	
E021-06	コア	SUBSTRING関数	
E021-07	コア	文字の連結	
E021-08	コア	UPPERおよびLOWER関数	
E021-09	コア	TRIM関数	
E021-10	コア	文字列型間の暗黙的キャスト	
E021-11	コア	POSITION関数	
E021-12	コア	文字の比較	
E031	コア	識別子	
E031-01	コア	制限付き識別子	
E031-02	コア	小文字の識別子	
E031-03	コア	末尾のアンダースコア	

E051	コア	基本問い合わせ仕様	
E051-01	コア	SELECT DISTINCT	
E051-02	コア	GROUP BY句	
E051-04	コア	選択リストにない句を持つGROUP BYが可能	
E051-05	コア	選択リスト項目の名前変更が可能	
E051-06	コア	HAVING句	
E051-07	コア	選択リスト内の修飾付き*	
E051-08	コア	FROM句内の相関名	
E051-09	コア	FROM句内の列名の変更	
E061	コア	基本述部および検索条件	
E061-01	コア	比較述部	
E061-02	コア	BETWEEN 述部	
E061-03	コア	値のリストが付いたIN述部	
E061-04	コア	LIKE 述部	
E061-05	コア	ESCAPE句付きのLIKE 述部	
E061-06	コア	NULL 述部	
E061-07	コア	修飾付き比較述部	
E061-08	コア	EXISTS述部	
E061-09	コア	比較術部内の副問い合わせ	
E061-11	コア	IN述部内の副問い合わせ	
E061-12	コア	修飾付き比較述部内の副問い合わせ	
E061-13	コア	相関副問い合わせ	
E061-14	コア	検索条件	
E071	コア	基本問い合わせ式	
E071-01	コア	UNION DISTINCTテーブル演算子	
E071-02	コア	UNION ALL テーブル演算子	
E071-03	コア	EXCEPT DISTINCT テーブル演算子	
E071-05	コア	正確に同一のデータ型を持つ必要がないテーブル演算子経由の列の結合	
E071-	コア	副問い合わせ内のテーブル演算子	

06			
E081	コア	基本権限	
E081-01	コア	SELECT権限	
E081-02	コア	DELETE権限	
E081-03	コア	テーブルレベルのINSERT権限	
E081-04	コア	テーブルレベルのUPDATE権限	
E081-05	コア	列レベルのUPDATE権限	
E081-06	コア	テーブルレベルのREFERENCES権限	
E081-07	コア	列レベルのREFERENCES権限	
E081-08	コア	WITH GRANT OPTION	
E081-09	コア	USAGE権限	
E081-10	コア	EXECUTE権限	
E091	コア	集合関数	
E091-01	コア	AVG	
E091-02	コア	COUNT	
E091-03	コア	MAX	
E091-04	コア	MIN	
E091-05	コア	SUM	
E091-06	コア	ALL修飾子	
E091-07	コア	DISTINCT修飾子	
E101	コア	基本データ操作	
E101-01	コア	INSERT 文	
E101-03	コア	Searched UPDATE 文	
E101-04	コア	Searched DELETE 文	
E111	コア	単一行SELECT文	
E121	コア	基本カーソルサポート	
E121-01	コア	DECLARE CURSOR	
E121-02	コア	ORDER BY の列は選択リスト内に不要	
E121-03	コア	ORDER BY句内の値式	
E121-04	コア	OPEN文	

E121-06	コア	位置指定UPDATE 文	
E121-07	コア	位置指定DELETE 文	
E121-08	コア	CLOSE 文	
E121-10	コア	FETCH 文、暗黙的なNEXT	
E121-17	コア	WITH HOLDカーソル	
E131	コア	NULL値のサポート(値の変わりとなるNULL)	
E141	コア	基本整合性制約	
E141-01	コア	NOT NULL制約	
E141-02	コア	NOT NULL句用のUNIQUE制約	
E141-03	コア	PRIMARY KEY制約	
E141-04	コア	参照削除動作、参照更新動作の両方でNO ACTIONをデフォルトで持つ基本FOREIGN KEY制約	
E141-06	コア	CHECK制約	
E141-07	コア	列デフォルト	
E141-08	コア	PRIMARY KEYはNOT NULLを意味する	
E141-10	コア	外部キー名の指定順序は任意	
E151	コア	トランザクションサポート	
E151-01	コア	COMMIT 文	
E151-02	コア	ROLLBACK 文	
E152	コア	基本SET TRANSACTION文	
E152-01	コア	SET TRANSACTION 文: ISOLATION LEVEL SERIALIZABLE句	
E152-02	コア	SET TRANSACTION 文: READ ONLY and READ WRITE句	
E161	コア	二重のマイナス記号から始まるSQLコメント	
E171	コア	SQLSTATEサポート	
F021	コア	基本情報スキーマ	
F021-01	コア	COLUMNS ビュー	
F021-02	コア	TABLES ビュー	
F021-03	コア	VIEWS ビュー	
F021-04	コア	TABLE_CONSTRAINTS ビュー	
F021-05	コア	REFERENTIAL_CONSTRAINTS ビュー	
F021-06	コア	CHECK_CONSTRAINTS ビュー	

F031	コア	基本スキーマ操作	
F031-01	コア	永続基礎テーブルを作成するCREATE TABLE文	
F031-02	コア	CREATE VIEW文	
F031-03	コア	GRANT文	
F031-04	コア	ALTER TABLE文: ADD COLUMN句	
F031-13	コア	DROP TABLE 文: RESTRICT句	
F031-16	コア	DROP VIEW 文: RESTRICT句	
F031-19	コア	REVOKE 文: RESTRICT句	
F032		CASCADE削除動作	
F033		ALTER TABLE文: DROP COLUMN句	
F034		拡張REVOKE文	
F034-01		スキーマオブジェクトの所有者以外が実行するREVOKE文	
F034-02		REVOKE 文: GRANT OPTION FOR句	
F034-03		被譲与者がWITH GRANT OPTIONを持つ権限を削除するREVOKE 文	
F041	コア	基本結合テーブル	
F041-01	コア	内部結合(INNERキーワードは不要)	
F041-02	コア	INNERキーワード	
F041-03	コア	LEFT OUTER JOIN	
F041-04	コア	RIGHT OUTER JOIN	
F041-05	コア	入れ子にできる外部結合	
F041-07	コア	左または右外部結合の内部テーブルは内部結合内でも使用可能	
F041-08	コア	全ての比較演算子のサポート(単なる=は除く)	
F051	コア	基本日付時刻	
F051-01	コア	DATE データ型 (DATEリテラルのサポートを含む)	
F051-02	コア	少なくとも0の秒精度を持つTIME データ型 (TIMEリテラルのサポートを含む)	
F051-03	コア	少なくとも0と6の秒精度を持つTIMESTAMP データ型 (TIMESTAMPリテラルのサポートを含む)	
F051-04	コア	DATE、TIME、TIMESTAMPデータ型に対する比較述部	
F051-05	コア	日付時刻型と文字列型間の明示的なキャスト	
F051-06	コア	CURRENT_DATE	
F051-07	コア	LOCALTIME	

F051-08	コア	LOCALTIMESTAMP	
F052	拡張日付時刻機能	時間間隔および日付時刻計算	
F053		OVERLAPS 述部	
F081	コア	ビューのUNIONおよびEXCEPT	
F111		SERIALIZABLE以外の隔離レベル	
F111-01		READ UNCOMMITTED隔離レベル	
F111-02		READ COMMITTED隔離レベル	
F111-03		REPEATABLE READ隔離レベル	
F131	コア	グループ化操作	
F131-01	コア	グループ化されたビューを使用する問い合わせにおけるWHERE、GROUP BY、HAVING句のサポート	
F131-02	コア	グループ化されたビューを使用する問い合わせにおける複数テーブルのサポート	
F131-03	コア	グループ化されたビューを使用する問い合わせにおける集合関数のサポート	
F131-04	コア	GROUP BY、HAVING句、グループ化されたビューを持つ副問い合わせ	
F131-05	コア	GROUP BY、HAVING句、グループ化されたビューを持つ単一行SELECT	
F171		ユーザ毎の複数スキーマ	
F191	拡張整合性管理	参照削除動作	
F200		TRUNCATE TABLE文	
F201	コア	CAST関数	
F221	コア	明示的なデフォルト	
F222		INSERT文: DEFAULT VALUES句	
F231		権限テーブル	
F231-01		TABLE_PRIVILEGESビュー	
F231-02		COLUMN_PRIVILEGESビュー	
F231-03		USAGE_PRIVILEGESビュー	
F251		ドメインサポート	
F261	コア	CASE式	
F261-01	コア	単純なCASE	
F261-02	コア	検索付きCASE	
F261-03	コア	NULLIF	
F261-04	コア	COALESCE	
F262		拡張CASE式	
F271		複合文字リテラル	
F281		高度なLIKE	
F302		INTERSECTテーブル演算子	
F302-01		INTERSECT DISTINCTテーブル演算子	

F302-02		INTERSECT ALLテーブル演算子	
F304		EXCEPT ALLテーブル演算子	
F311-01	コア	CREATE SCHEMA	
F311-02	コア	永続基礎テーブル用のCREATE TABLE	
F311-03	コア	CREATE VIEW	
F311-05	コア	GRANT 文	
F321		ユーザ認証	
F361		副プログラムのサポート	
F381		拡張スキーマ操作	
F381-01		ALTER TABLE 文: ALTER COLUMN句	
F381-02		ALTER TABLE 文: ADD CONSTRAINT句	
F381-03		ALTER TABLE 文: DROP CONSTRAINT句	
F382		列データ型変更	
F383		Set column not null句	
F391		長い識別子	
F392		識別子内のUnicodeエスケープ	
F393		リテラル内のUnicodeエスケープ	
F401		拡張結合テーブル	
F401-01		NATURAL JOIN	
F401-02		FULL OUTER JOIN	
F401-04		CROSS JOIN	
F402		LOB、配列、複数セットと結合する名前付きの列	
F411	拡張日付時刻機能	時間帯指定	リテラル解釈に関する違いあり
F421		各国言語	
F431		読み取りのみのスクロール可能なカーソル	
F431-01		明示的なNEXTを持つFETCH	
F431-02		FETCH FIRST	
F431-03		FETCH LAST	
F431-04		FETCH PRIOR	
F431-05		FETCH ABSOLUTE	
F431-06		FETCH RELATIVE	
F441		拡張集合関数のサポート	
F442		set関数内の混在する列参照	
F471	コア	スカラ副問い合わせ値	

F481	コア	拡張NULL述部	
F491	拡張整合性管理	制約管理	
F501	コア	機能と準拠ビュー	
F501-01	コア	SQL_FEATURESビュー	
F501-02	コア	SQL_SIZINGビュー	
F501-03	コア	SQL_LANGUAGESビュー	
F502		高度ドキュメントテーブル	
F502-01		SQL_SIZING_PROFILESビュー	
F502-02		SQL_IMPLEMENTATION_INFOビュー	
F502-03		SQL_PACKAGESビュー	
F531		一時テーブル	
F555	拡張日付時刻機能	高度な秒精度	
F561		完全な値式	
F571		真値試験	
F591		派生テーブル	
F611		指示子データ型	
F641		行、テーブルコンストラクタ	
F651		カタログ名修飾子	
F661		単純テーブル	
F672		遡及的検査制約	
F690		照合サポート	しかし文字セットサポートはありません
F692		高度照合サポート	
F701	拡張整合性管理	参照更新動作	
F711		ドメインのALTER	
F731		列のINSERT権限	
F761		セッション管理	
F762		CURRENT_CATALOG	
F763		CURRENT_SCHEMA	
F771		接続管理	
F781		自己参照操作	
F791		非sensitiveカーソル	
F801		完全な集合関数	
F850		<問い合わせ式>における最上位<order by句>	
F851		副問い合わせにおける<order by 句>	
F852		ビューにおける最上位<order by句>	
F855		<問い合わせ式>における入れ子状の<order by句>	
F856		<問い合わせ式>における入れ子状の<fetch first句>	
F857		<問い合わせ式>における最上位<fetch first句>	
F858		副問い合わせにおける<fetch first句>	
F859		ビューにおける最上位<fetch first句>	
F860		<fetch first 句>における<fetch first row count>	
F861		<問い合わせ式>における最上位<result offset句>	

F862		副問い合わせにおける<result offset句>	
F863		<問い合わせ式>における入れ子状の<result offset句>	
F864		ビューにおける最上位<result offset句>	
F865		<result offset 句>における<offset row count>	
S071	拡張オブジェクトサ ポート	関数内のSQLパスおよび型名の解決	
S092		ユーザデータ型の配列	
S095		問い合わせによる配列の構築	
S096		省略可能な配列境界	
S098		ARRAY_AGG	
S111	拡張オブジェクトサ ポート	問い合わせ式内のONLY	
S201		配列についてのSQLから呼び出すルーチン	
S201- 01		配列パラメータ	
S201- 02		関数の結果型としての配列	
S211	拡張オブジェクトサ ポート	ユーザ定義キャスト関数	
T031		BOOLEANデータ型	
T071		BIGINTデータ型	
T121		問い合わせ式内のWITH(RECURSIVEを除く)	
T122		副問い合わせにおけるWITH (RECURSIVEを除く)	
T131		再帰問い合わせ	
T132		副問い合わせにおける再帰問い合わせ	
T141		SIMILAR述部	
T151		DISTINCT述部	
T152		否定付きDISTINCT述部	
T171		テーブル定義内のLIKE句	
T172		テーブル定義内のAS副問い合わせ句	
T173		テーブル定義内の拡張LIKE句	
T191	拡張整合性管理	参照動作のRESTRICT	
T201	拡張整合性管理	参照制約向けの比較可能データ型	
T211- 01	動的データベース, 拡張整合性管理	基礎テーブルのUPDATE、INSERT、DELETEで駆動されるトリガ	
T211- 02	動的データベース, 拡張整合性管理	BEFOREトリガ	
T211- 03	動的データベース, 拡張整合性管理	AFTERトリガ	
T211- 04	動的データベース, 拡張整合性管理	FOR EACH ROWトリガ	
T211- 05	動的データベース, 拡張整合性管理	トリガ呼び出し前に真になる検索条件の指定機能	
T211- 07	動的データベース, 拡張整合性管理	TRIGGER権限	
T212	拡張整合性管理	高度トリガ機能	
T213		INSTEAD OFトリガ	
T231		Sensitiveカーソル	
T241		START TRANSACTION文	
T271		セーブポイント	

T281		列粒度のSELECT権限	
T312		OVERLAY関数	
T321-01	コア	オーバーロードがないユーザ定義関数	
T321-03	コア	関数呼び出し	
T321-06	コア	ROUTINESビュー	
T321-07	コア	PARAMETERSビュー	
T323		外部ルーチンに対する明示的なセキュリティ	
T331		基本ロール	
T341		SQLから呼び出される関数、プロシージャのオーバーロード	
T351		はさまれたSQLコメント(/*...*/ コメント)	
T441		ABSおよびMOD関数	
T461		対称BETWEEN述部	
T501		高度なEXISTS述部	
T551		デフォルト構文用の省略可能なキーワード	
T581		正規表現部分文字列関数	
T591		NULL列を含むUNIQUE制約	
T614		NTILE関数	
T615		LEADおよびLAG関数	
T617		FIRST_VALUEおよびLAST_VALUE関数	
T621		高度数値関数	
T631	Core	リスト要素内のIN述部	
T651		SQLルーチン内のSQLスキーマ文	
T655		周期に依存するルーチン	
X010		XML型	
X011		XML型の配列	
X016		永続XML値	
X020		XMLConcat	
X031		XMLElement	
X032		XMLForest	
X034		XMLAgg	
X035		XMLAgg: ORDER BYオプション	
X036		XMLComment	
X037		XMLPI	
X040		基本テーブルマッピング	
X041		基本テーブルマッピング: NULLがない	
X042		基本テーブルマッピング: NULLをnilとして扱う	
X043		基本テーブルマッピング: テーブルをフォレストとして扱う	
X044		基本テーブルマッピング: テーブルを要素として扱う	
X045		基本テーブルマッピング: 対象ネームスペースを持つ	
X046		基本テーブルマッピング: データマッピング	
X047		基本テーブルマッピング: メタデータのマッピング	
X048		基本テーブルマッピング: バイナリ列のBASE64符号化	
X049		基本テーブルマッピング: バイナリ列のHEX符号化	
X050		高度テーブルマッピング	

X051		高度テーブルマッピング: NULLがない	
X052		高度テーブルマッピング: NULLをnilとして扱う	
X053		高度テーブルマッピング: テーブルをフォレストとして扱う	
X054		高度テーブルマッピング: テーブルを要素として扱う	
X055		高度テーブルマッピング: 対象ネームスペース	
X056		高度テーブルマッピング: データマッピング	
X057		高度テーブルマッピング: メタデータのマッピング	
X058		高度テーブルマッピング: バイナリ列のBASE64符号化	
X059		高度テーブルマッピング: バイナリ列のHEX符号化	
X060		XMLParse: 文字列入力およびCONTENTオプション	
X061		XMLParse: 文字列入力およびDOCUMENTオプション	
X070		XMLSerialize: 文字列シリアライゼーションおよびCONTENTオプション	
X071		XMLSerialize: 文字列シリアライゼーションおよびDOCUMENTオプション	
X072		XMLSerialize: 文字列シリアライゼーション	
X090		XML文書述語	
X120		SQL関数におけるXMLパラメータ	
X121		外部関数におけるXMLパラメータ	
X400		名前と識別子のマッピング	
X410		列のデータ型の変更: XML型	

[前のページ](#)
SQLへの準拠

[ホーム](#)
[上に戻る](#)

[次のページ](#)
サポートされていない機能

D.2. サポートされていない機能

以下のSQL:2011で定義されている機能は本リリースのPostgreSQLでは実装されていません。 たまに同等の機能が実装されていることがあります。

識別子	パッケージ	説明	コメント
B011		埋め込みAda	
B013		埋め込みCOBOL	
B014		埋め込みFortran	
B015		埋め込みMUMPS	
B016		埋め込みPascal	
B017		埋め込みPL/I	
B031		基本動的SQL	
B032		拡張動的SQL	
B032-01		<describe input statement>	
B033		型のないSQLから呼び出す関数の引数	
B034		カーソル属性の動的指定	
B035		非拡張記述子名	
B041		埋め込みSQL例外宣言への拡張	
B051		高度な実行権	
B111		モジュール言語 Ada	
B112		モジュール言語 C	
B113		モジュール言語 COBOL	
B114		モジュール言語 Fortran	
B115		モジュール言語 MUMPS	
B116		モジュール言語 Pascal	
B117		モジュール言語 PL/I	
B121		ルーチン言語 Ada	
B122		ルーチン言語 C	
B123		ルーチン言語 COBOL	
B124		ルーチン言語 Fortran	
B125		ルーチン言語 MUMPS	
B126		ルーチン言語 Pascal	
B127		ルーチン言語 PL/I	
B128		ルーチン言語 SQL	
B211		モジュール言語 Ada: VARCHARおよびNUMERICのサポート	
B221		ルーチン言語 Ada: VARCHARおよびNUMERICのサポート	
E153	コア	副問い合わせを持つ更新	
E182	コア	モジュール言語	
F054		DATE型の述語リストにおけるTIMESTAMP	
F121		基本診断管理	
F121-01		GET DIAGNOSTICS 文	
F121-02		SET TRANSACTION 文: DIAGNOSTICS SIZE句	

F122		高度診断管理	
F123		すべての診断	
F181	コア	複数モジュールサポート	
F202		TRUNCATE TABLE: identity column restartオプション	
F263		単純CASE式におけるカンマ区切り述部	
F291		UNIQUE述部	
F301		問い合わせ式内のCORRESPONDING	
F311	コア	スキーマ定義文	
F311-04	コア	CREATE VIEW: WITH CHECK OPTION	
F312		MERGE文	
F313		高度MERGE文	
F314		DELETE分岐を持つMERGE文	
F341		使用方法テーブル	ROUTINE_*_USAGEテーブルはありません
F384		Drop identity property句	
F385		Drop column generation expression句	
F386		Set identity column generation句	
F394		標準フォーム指定（省略可能）	
F403		部分結合されたテーブル	
F451		文字セット定義	
F461		名前付き文字セット	
F492		省略可能なテーブル制約の強制	
F521	拡張整合性管理	表明	
F671	拡張整合性管理	CHECK内の副問い合わせ	意図的な省略
F693		SQLセッション、クライアントモジュールの照合	
F695		翻訳サポート	
F696		追加用翻訳ドキュメント	
F721		遅延可能制約	外部キーおよび一意キーのみ
F741		参照MATCH型	部分一致は未実装
F751		高度なビューのCHECK	
F812	コア	基本フラギング	
F813		拡張フラギング	
F821		ローカルなテーブル参照	
F831		完全なカーソル更新	
F831-01		更新可能、スクロール可能なカーソル	
F831-02		更新可能、順序付きカーソル	
F841		LIKE_REGEX述語	
F842		OCCURENCES_REGEX関数	
F843		POSITION_REGEX関数	
F844		SUBSTRING_REGEX関数	
F845		TRANSLATE_REGEX関数	
F846		正規表現演算子における8進形式サポート	
F847		非固定正規表現	
F866		FETCH FIRST句: PERCENTオプション	

F867		FETCH FIRST句: WITH TIESオプション	
S011	コア	Distinctデータ型	
S011-01	コア	USER_DEFINED_TYPESビュー	
S023	基本オブジェクトサポート	基本構造化型	
S024	拡張オブジェクトサポート	拡張構造化型	
S025		最終構造化型	
S026		自己参照構造化型	
S027		メソッド名を指定したメソッドの作成	
S028		交換可能UDTオプションリスト	
S041	基本オブジェクトサポート	基本参照型	
S043	拡張オブジェクトサポート	拡張参照型	
S051	基本オブジェクトサポート	型のテーブルの作成	一部サポート
S081	拡張オブジェクトサポート	副テーブル	
S091		基本配列サポート	一部サポート
S091-01		組み込みデータ型の配列	
S091-02		個々の型の配列	
S091-03		配列式	
S094		参照型の配列	
S097		配列要素の代入	
S151	基本オブジェクトサポート	型述部	
S161	拡張オブジェクトサポート	副型の扱い	
S162		参照用の副型の取り扱い	
S202		複数セットについてのSQLから呼び出すルーチン	
S231	拡張オブジェクトサポート	構造化型のロケータ	
S232		配列のロケータ	
S233		複数セットのロケータ	
S241		変形関数	
S242		変形文の変更	
S251		ユーザ定義の順序	
S261		特定の型メソッド	
S271		基本複数セットサポート	
S272		ユーザ定義型の複数セット	
S274		参照型の複数セット	
S275		高度複数セットサポート	
S281		入れ子の照合型	
S291		行全体に対する一意性制約	
S301		高度UNNEST	
S401		配列型に基づく個別型	

S402		個別型に基づく個別型	
S403		MAX_CARDINALITY	
S404		TRIM_ARRAY	
T011		情報スキーマ内のタイムスタンプ	
T021		BINARYおよびVARBINARYデータ型	
T022		BINARYおよびVARBINARYデータ型の高度サポート	
T023		コンパウンドバイナリリテラル	
T024		バイナリリテラルにおける空白	
T041	基本オブジェクトサ ポート	基本LOBデータ型サポート	
T041- 01	基本オブジェクトサ ポート	BLOB データ型	
T041- 02	基本オブジェクトサ ポート	CLOB データ型	
T041- 03	基本オブジェクトサ ポート	LOBデータ型用 のPOSITION、LENGTH、LOWER、TRIM、UPPER、SUBSTRING 関数	
T041- 04	基本オブジェクトサ ポート	LOBデータ型の連結	
T041- 05	基本オブジェクトサ ポート	LOBロケータ: 保持不可能	
T042		拡張LOBデータ型サポート	
T043		Multiplier T	
T044		Multiplier P	
T051		行型	
T052		行型用のMAXとMIN	
T053		全てのフィールドを参照するための明示的な別名	
T061		UCSサポート	
T101		高度なnullabilityの決定	
T111		更新可能な結合、和集合、列	
T174		列の識別	
T175		生成される列	
T176		連番生成サポート	
T177		連番生成サポート(簡単なリスタートオプション)	
T178		Identity列 (簡単なリスタートオプション)	
T180		システムバージョン付きテーブル	
T181		Application-time periodテーブル	
T211	動的データベース, 拡張整合性管理	基本トリガ機能	
T211- 06	動的データベース, 拡張整合性管理	トリガと制約間の相互作用についての実行時規則のサポート	
T211- 08	動的データベース, 拡張整合性管理	同一イベントに対する複数のトリガは、カタログ内で作成された順 番で実行される	意図的な省略
T251		SET TRANSACTION文: LOCALオプション	
T261		連鎖トランザクション	
T272		高度セーブポイント管理	
T285		高度継承された列名	
T301		機能の依存性	一部サポート
T321	コア	基本SQL呼び出しルーチン	
T321-			

02	コア	オーバーロードがないユーザ定義ストアドプロシージャ	
T321-04	コア	CALL 文	
T321-05	コア	RETURN 文	
T322	PSM	宣言されたデータ型属性	
T324		SQLルーチンに対する明示的なセキュリティ	
T325		修飾付きSQLパラメータ参照	
T326		テーブル関数	
T332		拡張ロール	ほとんどサポート
T431	OLAP	拡張グループ化機能	
T432		入れ子、連結されたGROUPING SETS	
T433		複数引数のGROUPING関数	
T434		GROUP BY DISTINCT	
T471		結果集合戻り値	
T472		DESCRIBE CURSOR	
T491		LATERAL派生テーブル	
T495		データ変更と取り出しの組み合わせ	異なる構文
T502		Period述部	
T511		トランザクションカウント	
T521		CALL文における名前付き引数	
T522		SQLから呼び出されるプロシージャのINパラメータに対するデフォルト値	
T541		更新可能なテーブル参照	
T561		保持可能なロケータ	
T571		配列を返すSQLが呼び出す外部関数	
T572		複数セットを返す、SQLから呼び出す外部関数	
T601		ローカルなカーソル参照	
T611	OLAP	基本OLAP操作	ほとんどの構文をサポート
T612		高度OLAP操作	ほとんどの構文をサポート
T613		サンプリング	
T616		LEADおよびLAG関数用のNull処理オプション	
T618		NTH_VALUE関数	関数自体はありますが、一部のオプションがありません
T619		入れ子状ウィンドウ関数	
T620		WINDOW句: GROUPSオプション	
T641		複数列の代入	構文の一部のみサポート
T652		SQLルーチン内のSQL動的文	
T653		外部ルーチン内のSQLスキーマ文	
T654		外部ルーチン内のSQL動的文	
M001		データリンク	
M002		SQL/CLI経由のデータリンク	
M003		埋め込みSQL経由のデータリンク	
M004		外部データサポート	一部サポート
M005		外部スキーマサポート	
M006		GetSQLString処理	
M007		TransmitRequest	

M009		GetOptsおよびGetStatistics処理	
M010		外部データラッパサポート	
M011		Ada経由のデータリンク	
M012		C経由のデータリンク	
M013		COBOL経由のデータリンク	
M014		Fortran経由のデータリンク	
M015		M経由のデータリンク	
M016		Pascal経由のデータリンク	
M017		PL/I経由のデータリンク	
M018		Adaにおける外部データラッパインタフェース処理	
M019		Cにおける外部データラッパインタフェース処理	
M020		COBOLにおける外部データラッパインタフェース処理	
M021		Fortranにおける外部データラッパインタフェース処理	
M022		MUMPSにおける外部データラッパインタフェース処理	
M023		Pascalにおける外部データラッパインタフェース処理	
M024		PL/Iにおける外部データラッパインタフェース処理	
M030		SQLサーバ外部データサポート	
M031		外部データラッパ汎用処理	
X012		XML型の集合	
X013		XML型の固有型	
X014		XML型の属性	
X015		XML型のフィールド	
X025		XMLCast	
X030		XMLDocument	
X038		XMLText	
X065		XMLParse: BLOB入力およびCONTENTオプション	
X066		XMLParse: BLOB入力およびDOCUMENTオプション	
X068		XMLSerialize: BOM	
X069		XMLSerialize: INDENT	
X073		XMLSerialize: BLOBシリアライゼーションおよびCONTENTオプション	
X074		XMLSerialize: BLOBシリアライゼーションおよびDOCUMENTオプション	
X075		XMLSerialize: BLOBシリアライゼーション	
X076		XMLSerialize: VERSION	
X077		XMLSerialize: 明示的ENCODINGオプション	
X078		XMLSerialize: 明示的XML宣言	
X080		XML公開におけるネームスペース	
X081		問い合わせレベルのXMLネームスペース宣言	
X082		DMLにおけるXMLネームスペース宣言	
X083		DDLにおけるXMLネームスペース宣言	
X084		複合文におけるXMLネームスペース宣言	
X085		事前定義のネームスペース接頭辞	
X086		XMLTableにおけるXMLネームスペース宣言	
X091		XMLコンテキスト述語	
X096		XMLExists	
X100		XML用のホスト言語サポート: CONTENTオプション	
X101		XML用のホスト言語サポート: DOCUMENTオプション	

X110		XML用のホスト言語サポート: VARCHARマッピング	
X111		XML用のホスト言語サポート: CLOBマッピング	
X112		XML用のホスト言語サポート: BLOBマッピング	
X113		XML用のホスト言語サポート: STRIP WHITESPACEオプション	
X114		XML用のホスト言語サポート: PRESERVE WHITESPACEオプション	
X131		問い合わせレベルのXMLBINARY句	
X132		DMLにおけるXMLBINARY句	
X133		DDLにおけるXMLBINARY句	
X134		複合型におけるXMLBINARY句	
X135		副問い合わせにおけるXMLBINARY句	
X141		IS VALID述語: データ駆動ケース	
X142		IS VALID述語: ACCORDING TO句	
X143		IS VALID述語: ELEMENT句	
X144		IS VALID 述語: スキーマ位置	
X145		検査制約外のIS VALID 述語	
X151		DOCUMENTオプション付きIS VALID 述語	
X152		CONTENTオプション付きIS VALID 述語	
X153		SEQUENCEオプション付きIS VALID 述語	
X155		IS VALID 述語: ELEMENT句のないNAMESPACE	
X157		IS VALID 述語: ELEMENT句付きのNO NAMESPACE	
X160		登録XMLスキーマに対する基本情報スキーマ	
X161		登録XMLスキーマに対する高度情報スキーマ	
X170		XMLにおけるnull取り扱いオプション	
X171		NIL ON NO CONTENTオプション	
X181		XML(DOCUMENT(UNTYPED))型	
X182		XML(DOCUMENT(ANY))型	
X190		XML(SEQUENCE)型	
X191		XML(DOCUMENT(XMLSCHEMA))型	
X192		XML(CONTENT(XMLSCHEMA))型	
X200		XMLQuery	
X201		XMLQuery: RETURNING CONTENT	
X202		XMLQuery: RETURNING SEQUENCE	
X203		XMLQuery: コンテキスト項目を渡す	
X204		XMLQuery: XQuery変数の初期化	
X205		XMLQuery: EMPTY ON EMPTYオプション	
X206		XMLQuery: NULL ON EMPTYオプション	
X211		XML 1.1サポート	
X221		値によるXMLを渡す機構	
X222		参照によるXMLを渡す機構	
X231		XML(CONTENT(UNTYPED))型	
X232		XML(CONTENT(ANY))型	
X241		XML公開におけるRETURNING CONTENT	
X242		XML公開におけるRETURNING SEQUENCE	
X251		XML(DOCUMENT(UNTYPED))型の永続的なXML値	
X252		XML(DOCUMENT(ANY))型の永続的なXML値	
X253		XML(CONTENT(UNTYPED))型の永続的なXML値	

X254		XML(CONTENT(ANY))型の永続的なXML値	
X255		XML(SEQUENCE)型の永続的なXML値	
X256		XML(DOCUMENT(XMLSCHEMA))型の永続的なXML値	
X257		XML(CONTENT(XMLSCHEMA))型の永続的なXML値	
X260		XML型: ELEMENT句	
X261		XML型: ELEMENT句のないNAMESPACE	
X263		XML型: ELEMENT句付きNO NAMESPACE	
X264		XML型: スキーマ位置	
X271		XMLValidate: データ駆動ケース	
X272		XMLValidate: ACCORDING TO句	
X273		XMLValidate: ELEMENT句	
X274		XMLValidate: スキーマ位置	
X281		XMLValidate: DOCUMENTオプション付き	
X282		XMLValidate CONTENTオプション付き	
X283		XMLValidate SEQUENCEオプション付き	
X284		XMLValidate ELEMENT句のないNAMESPACE	
X286		XMLValidate: ELEMENT句付きNO NAMESPACE	
X300		XMLTable	
X301		XMLTable: 派生列リストオプション	
X302		XMLTable: 序数列オプション	
X303		XMLTable: 列デフォルトオプション	
X304		XMLTable: コンテキスト項目を渡す	
X305		XMLTable: XQuery変数の初期化	

[前のページ](#)

サポートされている機能

[ホーム](#)

[上に戻る](#)

[次のページ](#)

リリースノート

付録 E. リリースノート

目次

- E.1. [リリース9.2.4](#)
- E.2. [リリース9.2.3](#)
- E.3. [リリース9.2.2](#)
- E.4. [リリース9.2.1](#)
- E.5. [リリース9.2](#)
- E.6. [リリース9.1.9](#)
- E.7. [リリース9.1.8](#)
- E.8. [リリース9.1.7](#)
- E.9. [リリース9.1.6](#)
- E.10. [リリース9.1.5](#)
- E.11. [リリース9.1.4](#)
- E.12. [リリース9.1.3](#)
- E.13. [リリース9.1.2](#)
- E.14. [リリース9.1.1](#)
- E.15. [リリース9.1](#)
- E.16. [リリース9.0.13](#)
- E.17. [リリース9.0.12](#)
- E.18. [リリース9.0.11](#)
- E.19. [リリース9.0.10](#)
- E.20. [リリース9.0.9](#)
- E.21. [リリース9.0.8](#)
- E.22. [リリース9.0.7](#)
- E.23. [リリース9.0.6](#)
- E.24. [リリース9.0.5](#)
- E.25. [リリース9.0.4](#)
- E.26. [リリース9.0.3](#)
- E.27. [リリース9.0.2](#)
- E.28. [リリース9.0.1](#)
- E.29. [リリース9.0](#)
- E.30. [リリース8.4.17](#)
- E.31. [リリース8.4.16](#)
- E.32. [リリース8.4.15](#)
- E.33. [リリース8.4.14](#)
- E.34. [リリース8.4.13](#)
- E.35. [リリース8.4.12](#)
- E.36. [リリース8.4.11](#)
- E.37. [リリース8.4.10](#)
- E.38. [リリース8.4.9](#)
- E.39. [リリース8.4.8](#)
- E.40. [リリース8.4.7](#)
- E.41. [リリース8.4.6](#)
- E.42. [リリース 8.4.5](#)
- E.43. [リリース 8.4.4](#)
- E.44. [リリース 8.4.3](#)
- E.45. [リリース 8.4.2](#)
- E.46. [リリース8.4.1](#)
- E.47. [リリース8.4](#)
- E.48. [リリース8.3.23](#)
- E.49. [リリース8.3.22](#)
- E.50. [リリース8.3.21](#)
- E.51. [リリース8.3.20](#)
- E.52. [リリース8.3.19](#)
- E.53. [リリース8.3.18](#)
- E.54. [リリース8.3.17](#)
- E.55. [リリース8.3.16](#)
- E.56. [リリース8.3.15](#)
- E.57. [リリース8.3.14](#)
- E.58. [リリース8.3.13](#)

E.59. [リリース8.3.12](#)
E.60. [リリース8.3.11](#)
E.61. [リリース8.3.10](#)
E.62. [リリース8.3.9](#)
E.63. [リリース8.3.8](#)
E.64. [リリース8.3.7](#)
E.65. [リリース8.3.6](#)
E.66. [リリース8.3.5](#)
E.67. [リリース8.3.4](#)
E.68. [リリース8.3.3](#)
E.69. [リリース8.3.2](#)
E.70. [リリース8.3.1](#)
E.71. [リリース 8.3](#)
E.72. [リリース8.2.23](#)
E.73. [リリース8.2.22](#)
E.74. [リリース8.2.21](#)
E.75. [リリース8.2.20](#)
E.76. [リリース8.2.19](#)
E.77. [リリース8.2.18](#)
E.78. [リリース8.2.17](#)
E.79. [リリース8.2.16](#)
E.80. [リリース8.2.15](#)
E.81. [リリース8.2.14](#)
E.82. [リリース8.2.13](#)
E.83. [リリース8.2.12](#)
E.84. [リリース8.2.11](#)
E.85. [リリース8.2.10](#)
E.86. [リリース8.2.9](#)
E.87. [リリース8.2.8](#)
E.88. [リリース8.2.7](#)
E.89. [リリース8.2.6](#)
E.90. [リリース8.2.5](#)
E.91. [リリース8.2.4](#)
E.92. [リリース8.2.3](#)
E.93. [リリース 8.2.2](#)
E.94. [リリース 8.2.1](#)
E.95. [リリース 8.2](#)
E.96. [リリース8.1.23](#)
E.97. [リリース8.1.22](#)
E.98. [リリース8.1.21](#)
E.99. [リリース8.1.20](#)
E.100. [リリース8.1.19](#)
E.101. [リリース8.1.18](#)
E.102. [リリース8.1.17](#)
E.103. [リリース8.1.16](#)
E.104. [リリース8.1.15](#)
E.105. [リリース8.1.14](#)
E.106. [リリース8.1.13](#)
E.107. [リリース8.1.12](#)
E.108. [リリース8.1.11](#)
E.109. [リリース8.1.10](#)
E.110. [リリース8.1.9](#)
E.111. [リリース8.1.8](#)
E.112. [リリース8.1.7](#)
E.113. [リリース 8.1.6](#)
E.114. [リリース8.1.5](#)
E.115. [リリース8.1.4](#)
E.116. [リリース8.1.3](#)
E.117. [リリース8.1.2](#)
E.118. [リリース8.1.1](#)
E.119. [リリース8.1](#)
E.120. [リリース8.0.26](#)
E.121. [リリース8.0.25](#)
E.122. [リリース8.0.24](#)
E.123. [リリース8.0.23](#)

E.124. [リリース8.0.22](#)
E.125. [リリース8.0.21](#)
E.126. [リリース8.0.20](#)
E.127. [リリース8.0.19](#)
E.128. [リリース8.0.18](#)
E.129. [リリース8.0.17](#)
E.130. [リリース8.0.16](#)
E.131. [リリース8.0.15](#)
E.132. [リリース8.0.14](#)
E.133. [リリース8.0.13](#)
E.134. [リリース 8.0.12](#)
E.135. [リリース 8.0.11](#)
E.136. [リリース 8.0.10](#)
E.137. [リリース8.0.9](#)
E.138. [リリース8.0.8](#)
E.139. [リリース8.0.7](#)
E.140. [リリース8.0.6](#)
E.141. [リリース8.0.5](#)
E.142. [リリース 8.0.4](#)
E.143. [リリース8.0.3](#)
E.144. [リリース8.0.2](#)
E.145. [リリース8.0.1](#)
E.146. [リリース8.0](#)
E.147. [リリース7.4.30](#)
E.148. [リリース7.4.29](#)
E.149. [リリース7.4.28](#)
E.150. [リリース7.4.27](#)
E.151. [リリース7.4.26](#)
E.152. [リリース7.4.25](#)
E.153. [リリース7.4.24](#)
E.154. [リリース7.4.23](#)
E.155. [リリース7.4.22](#)
E.156. [リリース7.4.21](#)
E.157. [リリース7.4.20](#)
E.158. [リリース7.4.19](#)
E.159. [リリース7.4.18](#)
E.160. [リリース7.4.17](#)
E.161. [リリース 7.4.16](#)
E.162. [リリース 7.4.15](#)
E.163. [リリース7.4.14](#)
E.164. [リリース7.4.13](#)
E.165. [リリース7.4.12](#)
E.166. [リリース7.4.11](#)
E.167. [リリース7.4.10](#)
E.168. [リリース 7.4.9](#)
E.169. [リリース7.4.8](#)
E.170. [リリース7.4.7](#)
E.171. [リリース7.4.6](#)
E.172. [リリース 7.4.5](#)
E.173. [リリース 7.4.4](#)
E.174. [リリース7.4.3](#)
E.175. [リリース7.4.2](#)
E.176. [リリース7.4.1](#)
E.177. [リリース 7.4](#)
E.178. [リリース7.3.21](#)
E.179. [リリース7.3.20](#)
E.180. [リリース7.3.19](#)
E.181. [リリース 7.3.18](#)
E.182. [リリース 7.3.17](#)
E.183. [リリース7.3.16](#)
E.184. [リリース7.3.15](#)
E.185. [リリース7.3.14](#)
E.186. [リリース7.3.13](#)
E.187. [リリース7.3.12](#)
E.188. [リリース 7.3.11](#)

E.189. [リリース7.3.10](#)
E.190. [リリース7.3.9](#)
E.191. [リリース7.3.8](#)
E.192. [リリース7.3.7](#)
E.193. [リリース7.3.6](#)
E.194. [リリース 7.3.5](#)
E.195. [リリース 7.3.4](#)
E.196. [リリース 7.3.3](#)
E.197. [リリース 7.3.2](#)
E.198. [リリース 7.3.1](#)
E.199. [リリース 7.3](#)
E.200. [リリース7.2.8](#)
E.201. [リリース7.2.7](#)
E.202. [リリース7.2.6](#)
E.203. [リリース7.2.5](#)
E.204. [リリース7.2.4](#)
E.205. [リリース 7.2.3](#)
E.206. [リリース 7.2.2](#)
E.207. [リリース 7.2.1](#)
E.208. [リリース 7.2](#)
E.209. [リリース 7.1.3](#)
E.210. [リリース 7.1.2](#)
E.211. [リリース 7.1.2](#)
E.212. [リリース 7.1](#)
E.213. [リリース 7.0.3](#)
E.214. [リリース 7.0.2](#)
E.215. [リリース 7.0.1](#)
E.216. [リリース 7.0](#)
E.217. [リリース 6.5.3](#)
E.218. [リリース 6.5.2](#)
E.219. [リリース 6.5.1](#)
E.220. [リリース 6.5](#)
E.221. [リリース 6.4.2](#)
E.222. [リリース 6.4.1](#)
E.223. [リリース 6.4](#)
E.224. [リリース 6.3.2](#)
E.225. [リリース 6.3.1](#)
E.226. [リリース 6.3](#)
E.227. [リリース 6.2.1](#)
E.228. [リリース 6.2](#)
E.229. [リリース 6.1.1](#)
E.230. [リリース 6.1](#)
E.231. [リリース 6.0](#)
E.232. [リリース 1.09](#)
E.233. [リリース 1.02](#)
E.234. [リリース 1.01](#)
E.235. [リリース 1.0](#)
E.236. [Postgres95 リリース 0.03](#)
E.237. [Postgres95 リリース 0.02](#)
E.238. [Postgres95 リリース 0.01](#)

このリリースノートには、PostgreSQLの各リリースでなされた重要な変更点が記載されています。また、主要機能や移行に関する問題点も最初に記載されています。リリースノートにはごく一部のユーザにしか影響しない変更点や内部的なものであるためユーザには隠れている変更点は記載されていません。例えば、オブティマイザはほぼすべてのリリースで改良されていますが、この改良はユーザにとっては通常、ただ問い合わせが高速になったと感じるものです。

各リリースにおける変更点の全一覧は、各リリースの[Gitログ](#)を参照することで入手できます。 [pgsql-committers](#)[メーリングリスト](#)にもすべてのソースコードの変更点が記録されています。また、特定のファイルに対する変更点を表示する[Webインタフェース](#)が存在します。

各項目の後に記述した名前は、その項目の主な開発者の名前です。もちろんすべての変更はコミュニティによる議論やパッチレビューが行われていますので、各項目は本当はコミュニティによるものと言えます。

E.1. リリース9.2.4

リリース日: 2013-04-04

このリリースは9.2.3に対し、各種不具合を修正したものです。9.2メジャーリリースにおける新機能については、[項E.5](#)を参照してください。

E.1.1. バージョン9.2.4への移行

9.2.Xからの移行ではダンプ/リストアは不要です。

しかし、このリリースではGiSTインデックスの管理についていくつかの不具合を修正しています。このアップデートをインストールした後、以下に記述する条件の1つ以上に該当するGiSTインデックスすべてについて、`REINDEX`を実行することを推奨します。

また、9.2.2よりも前のバージョンからアップグレードする場合は、9.2.2のリリースノートを参照してください。

E.1.2. 変更点

- サーバのコマンド行スイッチについての安全でない解析を修正しました。(Mitsumasa Kondo, Kyotaro Horiguchi)

"-"で始まるデータベース名を含む接続要求に細工をすることで、例えその要求が最終的に拒絶された場合でも、サーバのデータディレクトリ内のファイルを損傷させる、あるいは破壊することが可能でした。(CVE-2013-1899)

- OpenSSLの乱数の状態を、postmasterのそれぞれの子プロセスでリセットするようにしました。(Marko Kreen)

`contrib/pgcrypto`関数によって生成された乱数が、他のデータベースユーザによって比較的推定しやすいという問題が、これにより回避されます。この危険はpostmasterが`ssl = on`で構成されている時にのみ問題となりますが、ほとんどの接続ではSSL暗号化を利用していません。(CVE-2013-1900)

- REPLICATION権限の検査で、認証されたユーザでなく、現在のユーザについて調べるようにしました。(Noah Misch)

この誤りを利用すると、権限のないデータベースユーザが`pg_start_backup()`や`pg_stop_backup()`を呼び出し、それにより定期的なバックアップの作成を妨害することが可能でした。(CVE-2013-1901)

- GiSTインデックスで、"fuzzy"な幾何比較を使うのが適切でないときは、使わないようにしました。(Alexander Korotkov)

幾何データ型のコアでは"fuzzy"な同値性を利用して比較を行ないますが、`gist_box_same`では正確な比較を行う必要があり、そうしなければ、それを利用するGiSTインデックスは一貫しないものになります。このアップデートをインストールした後、`box`、`polygon`、`circle`、`point`型の列のすべてのGiSTインデックスについて`REINDEX`を実行してください。これらはいずれも`gist_box_same`を使っています。

- 可変長のデータ型、つまり`text`、`bytea`、`bit`、`numeric`型の列で`contrib/btree_gist`を使うGiSTインデックスについて誤った範囲の結合と不正なロジックを修正しました。(Tom Lane)

この誤りにより、存在するキーが検索で見つからないような一貫性のないインデックスが作成されたり、インデックスが無駄に大きくなることがありました。このアップデートをインストールした後、そのようなインデックスについて`REINDEX`を実行することを推奨します。

- 複数列についてのインデックスで、GiSTページを分割するコードの不具合を修正しました。(Tom Lane)

この誤りにより、存在するキーが検索で見つからないような一貫性のないインデックスが作られることや、検索が非常に非効率になるインデックスが作られることがありました。このアップデートをインストールした後、複数列についてのGiSTインデックスについて`REINDEX`を実行することを推奨します。

- `gist_point_consistent`で曖昧さを一貫性のある方法で処理するようにしました。(Alexander Korotkov)

`point`型の列のGiSTインデックスについてのインデックススキャンは、シーケンシャルスキャンとは異なる結果になる場合があります。これは、`gist_point_consistent`と演算子のコードで、厳密な比較をするか、曖昧な比較をするかについて不一致があったからです。

- WAL再生におけるバッファのリークを修正しました。(Heikki Linnakangas)

この不具合により、再生中に"incorrect local pin count"のエラーが発生し、リカバリができなくなる場合があります。

- データベースが正常に停止しておらず、かつ`recovery.conf`ファイルが存在する場合は、アーカイブリカバリに入る前に、確実にクラッシュリカバリを実行するようにしました。(Heikki Linnakangas, Kyotaro Horiguchi, Mitsumasa Kondo)

この修正は、実行中のサーバのファイルシステムスナップショットでスタンバイサーバを初期化する、というようなケースでデータベースの一貫性を保証するために必要です。

- クラッシュリカバリの最中に、まだアーカイブされていないWALファイルが削除されないようにしました。(Heikki Linnakangas, Fujii Masao)
- `DELETE RETURNING`での競合条件を修正しました。(Tom Lane)

特定の条件下で、`DELETE RETURNING`は現在のプロセスがもはや確保していない共有バッファからデータをフェッチすることがありました。その間に他のプロセスがそのバッファを変更した場合、ゴミの値が`RETURNING`の出力となったり、あるいはクラッシュすることもありました。

- 正規表現のコンパイルで無限ループになる可能性があるのを修正しました。(Tom Lane, Don Porter)
- 正規表現のコンパイル時にNULLポインタから参照先を取得する可能性があるのを修正しました。(Tom Lane)
- `to_char()`で、それが適切な場合、ASCIIのみの大文字変換規則を利用するように修正しました。(Tom Lane)

これにより、ロケールに依存すべきでないいくつかのテンプレートのパターンにおける誤った動作が修正されます。トルコ語のロケールで"`ı`"と"`i`"が正しく処理されていませんでした。

- 1999-12-31 24:00:00というタイムスタンプが理由もなく拒絶されていたのを修正しました。(Tom Lane)
- SQL言語の関数が範囲型をサポートする関数として安全に使えるよう修正しました。(Tom Lane)
- 単一のトランザクションが`UNLISTEN`を実行し、その後`LISTEN`を実行した場合のロジックの誤りを修正しました。(Tom Lane)

この場合、セッションは通知イベントを監視すべきですが、実際には全く監視しない状態になっていました。

- 他のビューから参照されているビューに列が追加された後、プランナがクラッシュする可能性があるのを修正しました。(Tom Lane)
- `EXPLAIN (ANALYZE, TIMING OFF)`のパフォーマンス問題を修正しました。(Pavel Stehule)
- "picksplit doesn't support secondary split"という不必要なログメッセージを削除しました。(Josh Hansen, Tom Lane)

このメッセージは何らかのコードを期待して追加されたように思われますが、そのコードは書かれませんでした。また、GiSTによる二次的分割のデフォルト処理は実際には非常に良いので、そのコードは今後も書かれないでしょう。そのため、エンドユーザにブツブツ言うのをやめることにしました。

- `gist_box_picksplit()`にあった二次的分割のサポートの痕跡を削除しました。(Tom Lane)

二次的分割の実装はデフォルトの実装に比べて良くないどころか、実際にはそれより悪いものでした。そこでそれを削除し、デフォルトのコードに処理させるようにしました。

- セッションの最後のいくつかのコミットとアボートの件数を統計情報コレクタに送信するのに失敗する可能性があるのを修正しました。(Tom Lane)
- PL/Perlの`spi_prepare()`関数のメモリリークを修正しました。(Alex Hunsaker, Tom Lane)
- `pg_dumpall`でデータベース名に"`=`"が含まれるものを正しく処理できるよう修正しました。(Heikki Linnakangas)
- `pg_dump`が不正な接続文字列が渡された時にクラッシュしないようにしました。(Heikki Linnakangas)
- `pg_dump`と`pg_upgrade`で不正なインデックスを無視するようにしました。(Michael Paquier, Bruce Momjian)

不正なインデックスをダンプするとリストア時に問題が発生する可能性があります。例えば、ユニーク条件を満たさないテーブルのデータに対してユニーク条件を強制したためにインデックス作成に失敗したような場合です。また、インデックスの作成が実際にはまだ進行中の場合もありますが、このときはDDLによる更新がまだコミットされていないと考えるのが適当でしょうから、`pg_dump`がダンプする必要はありません。`pg_upgrade`は不正なインデックスでエラーを起こ

す代わりに、それをスキップするようになります。

- `pg_basebackup`でテーブル空間をバックアップするとき、現在のサーババージョンのサブディレクトリだけを含めるようにしました。(Heikki Linnakangas)
- `pg_basebackup`および`pg_receivexlog`でサーバのバージョン確認をするようにしました。これにより動作しないバージョンの組み合わせに対して、正しく失敗するようになります。(Heikki Linnakangas)
- `contrib/dblink`について`DateStyle`あるいは`IntervalStyle`の一貫しない設定を安全に処理するよう修正しました。(Daniel Farina, Tom Lane)

以前は、リモートのサーバがこれらのパラメータについて異なる設定を指定した場合、曖昧な日付が誤って読み取られることがありました。この修正により、`dblink`の問い合わせによりフェッチされる日時および期間の列が正しく解釈されることが保証されます。しかし、一貫しない設定はそれでも危険であることに注意してください。SQLコマンド中のリテラル値がリモートのサーバに送られた場合、ローカルで処理されるのと異なる解釈をされる可能性があります。

- `contrib/pg_trgm`の`similarity()`関数が、トライグラムのない文字列について0を返すよう修正しました。(Tom Lane)

以前は内部でのゼロ割りによりNaNを返していました。

- Microsoft Visual Studio 2012でPostgreSQLをビルドできるようにしました。(Brar Piening, Noah Misch)
- 時間帯データファイルを`tzdata`リリース2013bに更新しました。チリ、ハイチ、モロッコ、パラグアイ、およびロシアのいくつかの地域での夏時間の変更が含まれます。また、数多くの場所での歴史的な地域データの修正が含まれます。

また、時間帯の略称について、ロシアおよび他の地域での以下の最近の変更を反映しました。

`CHOT`、`GET`、`IRKT`、`KGT`、`KRAT`、`MAGT`、`MAWT`、`MSK`、`NOVT`、`OMST`、`TKT`、`VLAT`、`WST`、`YAKT`、`YEKT`については、それらの現在の意味に従います。`VOLT` (Europe/Volgograd (ヨーロッパ/ヴォルゴグラード))および`MIST` (Antarctica/Macquarie (南極/マッコリー島))をデフォルトの短縮名リストに追加しました。

[前のページ](#)

リリースノート

[ホーム](#)
[上に戻る](#)

[次のページ](#)
リリース9.2.3

E.2. リリース9.2.3

リリース日: 2013-02-07

このリリースは9.2.2に対し、各種不具合を修正したものです。9.2メジャーリリースにおける新機能については、[項E.5](#)を参照してください。

E.2.1. バージョン9.2.3への移行

9.2.Xからの移行ではダンプ/リストアは不要です。

しかし、9.2.2よりも前のバージョンからアップグレードする場合は、9.2.2のリリースノートを参照してください。

E.2.2. 変更点

- SQLから`enum_recv`を実行できないようにしました。(Tom Lane)

この関数は誤った宣言をされており、単純なSQLコマンドでサーバをクラッシュさせることが可能でした。実際のところ、サーバのメモリの内容を検査するために攻撃者がこれを利用することが可能になっていました。この問題を(Secunia SVCRP経由で)報告してくれたSumit Soniに感謝します。(CVE-2013-0255)

- WALの再生時にデータベースが一貫性のある状態に到達したことを検知するときの複数の問題を修正しました。(Fujii Masao, Heikki Linnakangas, Simon Riggs, Andres Freund)
- 実際の再実行作業が必要でない場合における、バックアップの終了点の検出について修正しました。(Heikki Linnakangas)

この誤りにより、間違って"WAL ends before end of online backup"(オンラインバックアップの終了前にWALが終了)のエラーを起こすことがありました。

- リレーションのファイルを切り詰めるときに、最小復旧地点を更新するようにしました。(Heikki Linnakangas)
データが捨てられた後は、タイムライン上のそれより前の地点で復旧を止めることは、もはや安全ではありません。
- 復旧対象のタイムラインを変更した後のWALセグメントの再利用について修正しました。(Heikki Linnakangas)
- カスケードしたスタンバイサーバ上でタイムラインの履歴ファイルをアーカイブから適切にリストアするようにしました。(Heikki Linnakangas)
- ホットスタンバイのサーバ上でのロック競合の検出について修正しました。(Andres Freund, Robert Haas)
- ホットスタンバイモードで問い合わせの取り消しに失敗するのを修正しました。(Noah Misch, Simon Riggs)
ホットスタンバイでの競合する問い合わせを取り消す要求が失敗することがあり、これらの問い合わせで一貫性のないデータを見ることがありました。
- 復旧停止機能について、ユーザが接続できるようになる前には停止しないようにしました。(Tom Lane)
- 副SELECTの結果から添字付けあるいは列選択できるようSQLの文法を修正しました。(Tom Lane)
- 負荷が大きい時の自動バキュームの切り詰めの性能問題を修正しました。(Jan Wieck)

テーブルの最後にある空のページを切り詰めるときに排他的ロックが必要ですが、競合するロック要求があるとき、自動バキュームのプログラムはそれに失敗してテーブルロックを解放するようになっていました。負荷が重いと切り詰めが全く起らず、テーブルが大きくなるということが容易に起こり得ました。部分的な切り詰めを実行、ロックを解放、その後ロックの再取得と処理の継続を試みるように修正しました。この修正により、競合するロック要求がある場合に、自動バキュームがロックを解放するまでの平均時間が大きく短縮されます。

- `SPI_execute`および関連する関数の性能を改善しました。これにより、PL/pgSQLの`EXECUTE`が改善されます。(Heikki Linnakangas, Tom Lane)

計画をキャッシュする機能の見直しの結果として9.2で追加されたデータコピーのオーバーヘッドを削除しました。これにより9.1と比較した性能の劣化がなくなり、また特に実行される問い合わせ文字列が多数のSQL文を含むときにメモリ

を節約します。

副次的効果は、複数の文からなる問い合わせの文字列が完全に逐次的に処理されることです。つまり、前の文の実行が終わってから、次の文の解析と計画をするようになります。これにより、以前からあった問題が解決され、後の文の動作に影響を与えるDDLは期待通りに動作するようになります。

- インデックスの利用について、9.2以前のコスト推定に戻しました。(Tom Lane)

誤差についての適切でない変更により、非常に大きなインデックスを利用する場合について、コストの推定が高くなりすぎるがありました。

- `DROP INDEX CONCURRENTLY`で時々クラッシュするのを修正しました。(Tom Lane)
- `CREATE/DROP INDEX CONCURRENTLY`の実行中に、共有メモリのロック表が破損する可能性があるのを修正しました。(Tom Lane)
- 1つのタプルが(ページサイズ・フィルファクタ)より大きい場合について、`COPY`で複数タプルを挿入するコードを修正しました。(Heikki Linnakangas)

以前のコードでは無限ループに陥る可能性があります。

- `pg_tablespace`をスキャンするときに競合問題が発生しないようにしました。(Stephen Frost, Tom Lane)

`pg_tablespace`のエントリに複数の同時更新があるとき、`CREATE DATABASE`および`DROP DATABASE`が正しく動作しない場合があります。

- `DROP OWNED`がデータベース全体あるいはテーブル空間を削除しようとしなかったようにしました。(Álvaro Herrera)

安全のため、これらのオブジェクトは削除するのではなく、所有権を変更すべきです。

- `vacuum_freeze_table_age`の実装の誤りを修正しました。(Andres Freund)

`vacuum_freeze_min_age`の値よりも多くのトランザクションを実行したインストール環境では、この誤りにより自動バキュームが部分テーブルスキャンを実行できなくなり、その結果、必ずテーブル全体のスキャンが発生します。

- `RowExpr`あるいは`XmlExpr`が2度、解析された時に、誤った動作をしないようにしました。(Andres Freund, Tom Lane)

この誤りは`CREATE TABLE LIKE INCLUDING INDEXES`などの状況においてユーザに明らかになることができました。

- ハッシュテーブルのサイズ計算で整数のオーバーフローが起きないようにする仕組みを改善しました。(Jeff Davis)
- データ型の権限に関連するいくつかのバグを修正しました。(Tom Lane)

型のデフォルト権限にいくつかの問題があり、`pg_dump`はそのような権限を全くダンプ出力できませんでした。

- サーバがクラッシュした後、残されていた一時テーブルを無視しないという問題を修正しました。(Tom Lane)
- Windowsにおいて、`postmaster`のログファイルをサイズの問題で再利用しないという問題を修正しました。(Jeff Janes, Heikki Linnakangas)
- `to_date()`が範囲外の日付を拒絶するようにしました。(Hitoshi Harada)

- `pg_extension_config_dump()`が拡張のアップデートについて適切に処理するよう修正しました。(Tom Lane)

この関数は、対象のテーブルのすべての既存のエントリを置換するようになりました。これにより、拡張をアップデートするスクリプトから利用可能になります。

- PL/pgSQLが単純な式について計画時エラーを出す場合があるのを修正しました。(Tom Lane)

以前のコードでは、このエラーにより、`CONTEXT`トレースバックの1行目を無視することがありました。

- 関数が複数のテーブルのトリガーとして使われる場合について、PL/Pythonの処理を修正しました。(Andres Freund)
- Windowsで非ASCIIのプロンプト文字列が、確実に正しいコードページに変換されるようにしました。(Alexander Law, Noah Misch)

このバグは`psql`および他のいくつかのクライアントプログラムに影響がありました。

- データベースに接続していないとき、`psql`の`\?`コマンドがクラッシュする可能性があったのを修正しました。(Meng Qingzhong)

- `pg_basebackup`の実行中にリレーシヨンのファイルが削除された時にエラーが起きる可能性があるのを修正しました。(Heikki Linnakangas)
- `pg_basebackup -X fetch`でスタンバイサーバのバックアップをしているときに、タイムラインの切り替えをできるようにしました。(Heikki Linnakangas)
- `pg_dump`をホットスタンバイのサーバで実行するとき、ログを取らないテーブルのデータを除外するようにしました。(Magnus Hagander)

スタンバイのサーバにはログを取らないテーブルのデータがありませんから、これはいずれにせよ失敗します。従って、自動的に`--no-unlogged-table-data`と見なすのが最も良い方法であると思われます。

- `pg_upgrade`が不正なインデックスを安全に扱うよう修正しました。(Bruce Momjian)
- `pg_upgrade`の`-O/-o`オプションを修正しました。(Marti Raudsepp)
- `libpq`の`PQprintTuples`でバッファを1バイト行き過ぎてしまう問題を修正しました。(Xi Wang)

この古い関数はPostgreSQL本体によっては使われていませんが、クライアントのプログラムでは今でも使われているかもしれません。

- `ecpglib`で翻訳されたメッセージを適切に使うようにしました。(Chen Huajun)
- MSVC上で`ecpg_compat`および`pgtypes`のライブラリを適切にインストールするようにしました。(Jiang Guiqing)
- `isinf()`をシステムが提供していなければ、我々のバージョンを`libecpg`に含めるようにしました。(Jiang Guiqing)
- `libedit/libreadline`からの誤ったエクスポートに騙されないよう、`configure`による提供関数の確認の順序を調整しました。(Christoph Berg)
- Windowsのビルド番号が時とともに確実に大きくなるようにしました。(Magnus Hagander)
- Windows用にクロスコンパイルしたときに、`pgxs`が正しく`.exe`の拡張子の付いた実行プログラムを生成するようにしました。(Zoltan Boszormenyi)
- 新しい時間帯の略号`FET`を追加しました。(Tom Lane)

これは一部の東ヨーロッパの時間帯で使われています。

E.3. リリース9.2.2

リリース日: 2012-12-06

このリリースは9.2.1に対し、各種不具合を修正したものです。9.2メジャーリリースにおける新機能については、[項E.5](#)を参照してください。

E.3.1. バージョン9.2.2への移行

9.2.Xからの移行ではダンプ/リストアは不要です。

しかし、以下の変更履歴の最初の項目に記述されている通り、CONCURRENTLYで作成されたインデックスの問題を修正するためにREINDEXの操作を行う必要があるかもしれません。

また、バージョン9.2.0からアップグレードする場合は、9.2.1のリリースノートを参照してください。

E.3.2. 変更点

- CREATE/DROP INDEX CONCURRENTLYに関連する複数の不具合を修正しました。(Andres Freund, Tom Lane, Simon Riggs, Pavan Deolasee)

DROP INDEX CONCURRENTLYの追加によって引き起こされたエラーにより、CREATE INDEX CONCURRENTLYの最初の段階で索引作成について誤った決定がされる場合があります。そのため、このコマンドによって作成されたインデックスが破損していることがあります。9.2.XにおいてCREATE INDEX CONCURRENTLYで作成されたインデックスについては、このアップデートを適用した後で再作成することが推奨されます。

さらに、CREATE/DROP INDEX CONCURRENTLYがインデックスのpg_indexの行の状態を変更するときに、その場にある更新を使うように修正しました。これにより、同時実行中のセッションが対象のインデックスを更新できず、それによりCONCURRENTLYで作成されるインデックスを破損させることになる競合条件を防ぐことができます。

また、CREATE INDEX CONCURRENTLY コマンドのエラーの結果として作られる不正なインデックスについて、他の様々な操作が確実に無視するように修正しました。この中で最も重要なのはVACUUMです。なぜなら、この不正なインデックスを修正あるいは削除するための修正の処理が実行される前に、自動バキュームが実行されることがあり得るからです。

また、DROP INDEX CONCURRENTLYについて、対象のインデックスを利用するすべての問い合わせが終了しなくても、挿入が可能になるように修正しました。

また、DROP INDEX CONCURRENTLYが取り消された時の不正な動作を修正しました。以前のプログラムでは、削除できないインデックスが残る可能性がありました。

- DROP INDEX CONCURRENTLYの述部ロックを修正しました。(Kevin Grittner)

これまでは、SSI述部ロックは間違ったタイミングで処理されており、シリアライズブルのトランザクションがDROPと並行して実行されるという誤った動作をする可能性がありました。

- WAL再生中のバッファロックを修正しました。(Tom Lane)

複数ページにまたがるWALレコードを再生する場合のバッファロックについて、WAL再生のプログラムに完全でない部分がありました。これにより、ホットスタンバイの問い合わせが一時的に一貫性のない状態になることがあり、その結果、誤った結果や不正なエラーを起こすことができました。

- GINインデックスについてのWAL生成のロジックの誤りを修正しました。(Tom Lane)

分裂ページのエラーが発生すると、これによりインデックスが破損することがありました。

- SP-GiSTインデックスについてのWAL再生のロジックの誤りを修正しました。(Tom Lane)

クラッシュの後、あるいはスタンバイサーバにおいて、これによりインデックスが破損することがありました。

- WAL再生中にベースバックアップの終了位置を誤検知するのを修正しました。(Heikki Linnakangas)

この誤りにより、データベースが一貫性のある状態になるより前にホットスタンバイモードが起動してしまうことがありました。

- ホットスタンバイのサーバを通常の実行にプロモートする時、起動プロセスの仮想XIDロックを正しく削除するようにしました。(Simon Riggs)

この誤りにより、`CREATE INDEX CONCURRENTLY`などいくつかの操作をこの後で実行できない場合があります。

- スタンバイモードにおいて偽物の"out-of-sequence timeline ID"エラーが出ないようにしました。(Heikki Linnakangas)
- postmasterがシャットダウンのシグナルを受け取った後は、新しい子プロセスを起動しないようにしました。(Tom Lane)

この誤りにより、シャットダウンが必要以上に長く時間がかかる、あるいはユーザによる追加の操作がなければ終了しない、ということがありました。

- `log_rotation_age`が2³¹ミリ秒(約25日)を超えていても、sysloggerがエラーを起こさないように修正しました。(Tom Lane)
- `WaitLatch()`について、要求されたタイムアウトの期限が切れたら、すぐに戻るように修正しました。(Jeff Janes, Tom Lane)

以前のプログラムでは、ウェイトを終了させない割り込みが定常的に発生すると、`WaitLatch()`からの戻りが無期限に遅れることがありました。これはautovacuum launcherプロセスにおいて問題であることがわかりました。他の部分でも同じように問題となる可能性があります。

- メモリが不足した時でも、内部ハッシュテーブルが破損しないようにしました。(Hitoshi Harada)
- 削除したテーブルのファイルデスクリプタをトランザクションが終了した後もオープンのまま保持し続ける問題を修正しました。(Tom Lane)

これにより、削除してから時間が経っているテーブルがディスクスペースを占め続けるという問題の発生が減るはずで

- 新しい子プロセスがラッチへのパイプを作成できないとき、データベース全体がクラッシュして再起動する問題を修正しました。(Tom Lane)

新しいプロセスはエラーにするしかありませんが、データベース全体を強制的に再起動させるべき理由はありませんから、再起動しないようにしました。これによりカーネルのファイルデスクリプタが足りなくなりそうな場合の堅牢性が向上します。

- 平坦化されていない副問い合わせとの結合でプランナがクラッシュする問題を修正しました。(Tom Lane)
- 外部結合上の厳格でない等価な句についての計画作成を修正しました。(Tom Lane)

厳格でない構造を他のものと等号で結ぶ句について、プランナが誤った制約を導出する場合があります。例えば、`WHERE COALESCE(foo, 0) = 0`で`foo`が外部結合のNULLになるかもしれない側のもののときです。9.2では以前のリリースに比べてこの種のエラーが起こる場合が多いことがわかりましたが、基本的な不具合はずっと以前からありました。

- 継承ツリー上でインデックスにより最適化されるMIN/MAXがある場合の`SELECT DISTINCT`について修正しました。(Tom Lane)

これらの要素の組み合わせに対して、プランナは"failed to re-find MinMaxAggInfo record"というエラーを起こしていました。

- それらに実際に意味的な違いがあるという少数の場合を除き、プランナはすべての目的において、暗示的なキャストと明示的なキャストを等価なものとして扱うようにしました。(Tom Lane)
- 問い合わせに対して部分インデックスが使えるかどうかを考慮する際に、JOIN句を含めるようにしました。(Tom Lane)

例えば、`x IS NOT NULL`という述部を確立するには、厳格なJOIN句があれば十分です。これにより、9.2でのプランナの機能低下が修正されます、というのは、以前のバージョンでは同等の導出が可能でした。

- 同一のインデックスについて多数のインデックス利用可能なJOIN句がある場合について、計画作成の時間の増大を制限しました。(Tom Lane)
- プランナについて、等価なクラスから排他制約を証明する能力を向上しました。(Tom Lane)
- ハッシュされた副計画の部分行のマッチングについて、型をまたがる場合を正しく処理できるよう修正しました。(Tom Lane)

これはWHERE (a, b) NOT IN (SELECT x, y FROM ...)のような複数の列についてのNOT INの副計画で、例えばbとyがそれぞれint4型とint8型である、というような場合に影響します。この処理の誤りにより、そこで使われているデータ型によって、誤った結果が得られたり、あるいはクラッシュする、ということが起きていました。

- Bツリーのマーク/リストアの機能で、配列のキーを正しく処理できるように修正しました。(Tom Lane)

この誤りにより、マージ結合の内側がindexed_column = ANY(array)の条件を使うインデックススキャンのときに、誤った結果を返すことができました。

- スナップショットの回数を減らすパッチを元に戻しました。(Tom Lane)

問い合わせ実行中のスナップショットの回数を減らすという9.2の変更により、以前のリリースでは見られなかったいくつかの異常な動作が発生しました。これは、問い合わせで使われる表をロックするより前に取得したスナップショットを使って問い合わせの実行が進められるからです。このため、例えば、ある問い合わせより前に実行されているトランザクションが排他ロックを取得している場合であっても、その問い合わせは、トランザクションによる更新を参照できるとは限らなくなっていました。この件については将来のリリースで再検討することになると思いますが、当面は9.2以前と同じ状態に戻すことにします。

- AFTER ROW UPDATE/DELETEトリガーで古いタプルを再フェッチする時、バッファロックを取得するようにしました。(Andres Freund)

この誤りにより、非常に特異な状況において、トリガーのWHEN条件、あるいは外部キーを強制するトリガーの事前チェックのロジックに対して誤ったデータを渡すことができました。その結果として、クラッシュを起こしたり、あるいはトリガーを実行するかどうかについて誤った判断をすることがありました。

- ALTER COLUMN TYPEについて、継承されたチェック制約を正しく処理できるように修正しました。(Pavan Deolasee)

これは8.4より前のリリースでは正しく動作していました。この修正により8.4およびそれ以降のリリースでも正しく動作するようになりました。

- ALTER EXTENSION SET SCHEMAについて、いくつかの付随するオブジェクトを新しいスキーマに移動するのに失敗することがあるのを修正しました。(Álvaro Herrera, Dimitri Fontaine)

- 拡張問い合わせプロトコルでCREATE TABLE AS EXECUTEを正しく処理できるようにしました。(Tom Lane)

- DROP RULE IF NOT EXISTSおよびDROP TRIGGER IF NOT EXISTSにおいて、入力 of 構文解析ツリーを変更しないようにしました。(Tom Lane)

この誤りにより、これらの型の一つのキャッシュされた命令が再実行されると、エラーが起こることがありました。

- REASSIGN OWNEDについて、表領域のGRANTが正しく処理できるように修正しました。(Álvaro Herrera)

- ビューのシステム列についてのpg_attributeの誤ったエントリを無視するようにしました。(Tom Lane)

ビューにはシステム列はありません。しかし、表をビューに変換するとき、そのようなエントリを取り除くことを忘れていました。これは9.3およびそれ以降のリリースでは適切に修正されますが、それ以前のブランチにおいても、誤って変換されたビューが存在しないようにする必要があります。

- ルールの出力について、INSERT INTO table DEFAULT VALUESを正しくダンプできるよう修正しました。(Tom Lane)

- 問い合わせが非常に多くのUNION/INTERSECT/EXCEPTを含むときでもスタックオーバーフローが発生しないようにしました。(Tom Lane)

- もっとも小さな整数値を-1で割り算した時に、プラットフォームによってはエラーとなるのを修正しました。(Xi Wang, Tom Lane)

- 日付文字列の解析で、文字列の終わりを越えてアクセスすることがあるのを修正しました。(Hitoshi Harada)

- チェックポイント中にXID周回が発生し、かつwal_levelがhot_standbyである場合にXIDエポックを進めるのに失敗するのを修正しました。(Tom Lane, Andres Freund)

この誤りはPostgreSQL自体には特に影響はありませんでしたが、TXID値が後戻りしているように見えるため、txid_current()およびそれに関連した関数に依存するアプリケーションにとっては問題でした。

- pg_terminate_backend()とpg_cancel_backend()について、対象のプロセスが存在しない場合に、エラーを発生させないようにしました。(Josh Kupersmidt)

以前からスーパーユーザーによって呼び出された場合は意図した通りの動作をしていましたが、一般ユーザに呼び出され

たときはそうではありませんでした。

- ページ境界における`pg_stat_replication.sync_state`の表示を修正しました。(Kyotaro Horiguchi)
- Unixドメインソケットのパス名がプラットフォーム依存の最大長を超えていた場合に、理解可能なエラーメッセージを生成するようにしました。(Tom Lane, Andrew Dunstan)

以前は"Non-recoverable failure in name resolution"(名前解決で解決不能なエラーが発生しました)といった非常に不親切なエラーを起こしていました。

- 複合列の値をクライアントに送るときのメモリリークを修正しました。(Tom Lane)
- コミット時に副トランザクションのロックを検索しないようにすることで、いくつかのサイクルを節約しました。(Simon Riggs)

多くの排他ロックを保持するトランザクションにおいて、この不要な動作は極めて高価なものとなる場合があります。

- `postmaster.pid`ファイルの読み込みについて、`pg_ctl`をより堅牢にしました。(Heikki Linnakangas)

これは競合条件、およびファイルデスク립タのリークを解決します。

- `psql`において、不正にエンコードされたデータが入力され、かつ`client_encoding`の設定がSJISのようにクライアントでしか利用できない文字セットである場合にクラッシュする場合があるのを修正しました。(Jiang Guiqing)
- `pg_dump`について`SEQUENCE SET`の要素をアーカイブ内のデータ部(データの前ではない)にダンプするようにしました。(Tom Lane)

これにより`--data-only`と`--section=data`の意味に一貫性がないという望ましくない問題が解決されます。また、拡張設定テーブルとして印付けられたシーケンスのダンプについても修正されます。

- `pg_dump`の`--clean`モードにおける`DROP DATABASE`コマンドの処理を修正しました。(Guillaume Lelarge)

9.2.0から`pg_dump --clean`は`DROP DATABASE`コマンドを発行するようになりましたが、これは使い方によって不要であるかあるいは危険であるかのどちらかでした。以後はこれをしなくなります。この修正により、`--clean`と`--create`を組み合わせた場合も適切に動作するようになります。つまり`DROP DATABASE`と`CREATE DATABASE`を実行し、その後で対象のデータベースに接続します。

- 循環的な依存があり、かつリレーションのオプションがないビューについて、`pg_dump`を修正しました。(Tom Lane)

循環的な依存のあるビューについてリレーションのオプションをダンプするという以前の修正では、ビューにオプションがないという場合に正しく動作せず、`ALTER VIEW foo SET ()`という不正な構文を出力していました。

- `pg_dump`が`tar`出力形式で生成する`restore.sql`スクリプトの不具合を修正しました。(Tom Lane)

このスクリプトはテーブル名が大文字を含む場合に全く動作しませんでした。また、このスクリプトで通常のCOPYモードだけでなく`--inserts`モードの場合もデータをリストアできるようにしました。

- `pg_restore`でPOSIX準拠の`tar`ファイルを処理できるよう修正しました。(Brian Weaver, Tom Lane)

`pg_dump`の`tar`出力モードの元々のプログラムは、POSIX標準に完全には準拠していないファイルを生成していました。この問題はバージョン9.3で修正されます。このパッチは9.3より前のブランチを修正するもので、正しくないフォーマットと正しいフォーマットの両方を処理できるようにします。これにより、9.3がリリースされた時に互換性の問題を防ぐことができます。

- `pg_basebackup`が生成する`tar`ファイルがPOSIX準拠になるように修正しました。(Brian Weaver, Tom Lane)

- `pg_resetxlog`について、データディレクトリへの相対パスが与えられた時に、`postmaster.pid`の場所を正しく特定できるよう修正しました。(Tom Lane)

この誤りにより、データディレクトリを使用している動作中の`postmaster`があることに`pg_resetxlog`が気が付かないことがありました。

- `libpq`の`lo_import()`関数および`lo_export()`関数が、ファイルI/Oのエラーを適切に報告するよう修正しました。(Tom Lane)
- `ecpg`について、ネストした構造体のポインタ変数の処理を修正しました。(Muhammad Usama)
- `ecpg`の`ecpg_get_data`関数について、配列を正しく処理するよう修正しました。(Michael Meskes)

pg_upgradeがシステムカタログについてTOASTテーブルの処理をしようとしないようにしました。(Bruce Momjian)

これにより、`information_schema`が削除され、再作成された場合に発生していたエラーが修正されます。これ以外のエラーが起こる場合もありました。

- 新しいクラスタにおいて`synchronous_commit`を`off`にすることで、pg_upgradeのパフォーマンスを向上しました。(Bruce Momjian)
- `contrib/pageinspect`のBツリーページ検査関数で、ページの検査中はバッファロックを取得するようにしました。(Tom Lane)
- `malloc(0)` および `realloc(NULL, 0)` の移植性のない動作について回避策を適用しました。(Tom Lane)

これらの呼び出しが`NULL`を返すプラットフォームにおいて、これが誤ってout-of-memory(メモリ不足)であると解釈されるようなコードがいくつかありました。既知のケースでは、ユーザ定義の集約が含まれないデータベースについてpg_dumpが正しく動作しないというものがありました。他にも正しく動作しないケースがあったかもしれません。

- 拡張について`make install`を実行した時に`extension`というインストールディレクトリが確実に作成されるようにしました。(Cédric Villemain)

以前は、拡張のMakefileで`MODULEDIR`が定義されていると、このステップが実行されませんでした。

- AIX上でのロードモジュールのビルドについてpgxsのサポートを修正しました。(Tom Lane)

元々のソースツリーの外側でモジュールをビルドしてもAIXでは正しく動作しませんでした。

- 時間帯データファイルをtzdataリリース2012jに更新しました。キューバ、イスラエル、ヨルダン、リビア、パレスチナ、西サモア、およびブラジルの一部での夏時間の変更が含まれます。

[前のページ](#)
リリース9.2.3

[ホーム](#)
[上に戻る](#)

[次のページ](#)
リリース9.2.1

E.4. リリース9.2.1

リリース日: 2012-09-24

このリリースは9.2.0に対し、各種不具合を修正したものです。9.2メジャーリリースにおける新機能については[項E.5](#)を参照してください。

E.4.1. バージョン9.2.1への移行

9.2.Xからの移行ではダンプ/リストアは不要です。

しかし、以下の変更履歴の最初に記述されているデータ破損の不具合の影響から復旧させるために、`REINDEX`および、あるいは`VACUUM`の操作を実行する必要があるかもしれません。

E.4.2. 変更点

- WAL再生時の共有バッファの永続性の印付けを修正しました。(Jeff Davis)

この間違いにより、チェックポイント時にバッファが書き出されなかったことがありました。その結果、その後もそのバッファを書き出すことなくサーバがクラッシュすると、データ破損が発生することになります。データ破損は、クラッシュからの復旧の後にはどのサーバでも発生する可能性があります。特にスタンバイのスレーブサーバではWAL再生を実行する回数がずっと多いですから、この問題が発生する可能性もずっと高くなります。btreeおよびGINインデックスが破損する可能性は低いです。テーブルの"可視性マップ"が破損する可能性はずっと高く、これによりインデックスオンリースキャンが誤った結果を返す可能性があります。テーブルデータの内容はこの不具合によって破損することはありません。

この不具合のためにインデックスが破損したという既知の事例はありませんが、念のため、稼働用のインストレーションでは、9.2.1にアップグレードした後、都合の良い時に、すべてのbtreeおよびGINインデックスについて`REINDEX`を実行することを勧めます。

また、`vacuum_freeze_table_age`を0に設定して、すべてのテーブルについて`VACUUM`を実行することを勧めます。これにより、すべての不正な可視性マップのデータが修正されます。`vacuum_cost_delay`を調整して、バキュームがパフォーマンスに与える影響を小さくすることができます。ただし、それによりバキュームが終了するまでの時間は長くなります。

- `WHERE indexed_column IN (list_of_values)`を含む問い合わせの結果のソートが正しくない場合があるのを修正しました。(Tom Lane)
- `GROUP BY`でウィンドウ関数および集計関数を含む式がある問い合わせに対して、プランナがエラーになることがあるのを修正しました。(Tom Lane)
- プランナによるエグゼキュータのパラメータの代入について修正しました。(Tom Lane)
- この間違いにより、同じ`WITH`の副問い合わせを複数回スキャンする問い合わせについて、間違った結果を返すことがありました。
- インデックススキャンの結合条件について、プランナの処理を改善しました。(Tom Lane)
- 接頭辞、つまり`word:*`のパターンを含むテキスト検索の問い合わせについて、選択性の推定を改善しました。(Tom Lane)
- 権限の変更についての遅延した認知を修正しました。(Tom Lane)

自分のトランザクションで既に保持しているロックだけしか必要としないコマンドは、それと同時の`GRANT`または`REVOKE`で、そのトランザクションの開始よりも後に実行されたものに気付かない場合があります。

- 列が配列型のドメインである場合に、`ANALYZE`がエラーとならないよう修正しました。(Tom Lane)
- PL/Perlの再帰関数が実行時に再定義される場合に、PL/Perlがクラッシュしないようにしました。(Tom Lane)
- PL/Perlで正しく最適化されない場合があるのを回避しました。(Tom Lane)

一部のLinuxディストリビューションでは、誤ったバージョンの`pthread.h`がインストールされており、PL/Perlのコンパイルされたコードが誤ったものになります。この結果、PL/Perlの関数がエラーを発生させる他の関数を呼び出すと、クラッシュすることがありました。

- pg_upgradeのpg_configへの依存は、不要なので取り除きました。(Peter Eisentraut)
- 時間帯データファイルをtzdataリリース2012fに更新しました。 フィジーでの夏時間の変更が含まれます。

[前のページ](#)

リリース9.2.2

[ホーム](#)
[上に戻る](#)

[次のページ](#)

リリース9.2

E.5. リリース9.2

リリース日: 2012-09-10

E.5.1. 概要

このリリースでは、新しいSQLの機能もありますが、主にパフォーマンスの向上に注力してきました。レプリケーションのサポートにおいても開発が継続されています。主な機能強化には以下のものがあります。

- 問い合わせがヒープにアクセスせず、インデックスのみからデータを検索できるようにしました(インデックスオンリースキャン)。
- ブリペアド文が使われる場合であっても、特定のパラメータ値のための特別な計画をプランナが生成できるようにしました。
- 入れ子ループで内部インデックススキャンを使えるようプランナを改善しました。
- ストリーミングレプリケーションのスレーブが、他のスレーブにデータを転送できるようにしました([カスケードレプリケーション](#))。
- [pg_basebackup](#)で、スタンバイサーバからバックアップを作れるようにしました。
- WALファイルが書き込まれた時に、その更新をアーカイブするツール[pg_receivexlog](#)を追加しました。
- [SP-GiST](#) (Space-Partitioned GiST、空間分割されたGiST)というインデックスアクセス方法を追加しました。
- [範囲データ型](#)のサポートを追加しました。
- [JSON](#) データ型を追加しました。
- ビューに[security_barrier](#)オプションを追加しました。
- libpqの接続文字列を、[URI](#)の形式にできるようにしました。
- libpqが大きな結果集合をより良く処理できるように、[1行処理モード](#)を追加しました。

上記の項目については、以下でより詳細に説明します。

E.5.2. バージョン9.2への移行

以前のリリースからデータを移行したい時は、どのリリースについても、pg_dumpを利用したダンプとリストア、あるいはpg_upgradeの利用が必要です。

バージョン9.2には、以前のバージョンとの互換性に影響する多くの変更点が含まれています。以下の非互換性に注意してください。

E.5.2.1. システムカタログ

- [pg_tablespace](#)からspclocationフィールドを削除しました。(Magnus Hagander)

このフィールドは、テーブル空間の場所を実際に定義するシンボリックリンクと同じでしたので、テーブル空間を移動するときにそれがなくなってしまうエラーを起こす危険性がありました。この変更により、サーバが停止している間に、シンボリックリンクを手作業で操作することにより、テーブル空間のディレクトリを移動することが可能になります。このフィールドを置き換え、このシンボリックリンクについて問い合わせができるように、[pg_tablespace_location\(\)](#)を追加しました。

- tsvectorのmost-common-element(最も共通な要素)統計を[pg_stats](#)の新しい列に移動しました。(Alexander Korotkov)

以前、tsvectorの列についてmost_common_valsおよびmost_common_freqsで利用可能だったデータについては、most_common_elemsおよびmost_common_elem_freqsを参照してください。

E.5.2.2. Functions

- [hstore](#)の=>演算子を削除しました。(Robert Haas)

代わりにhstore(text, text)を使ってください。=>は標準SQLで他の目的のために予約しているので、PostgreSQL 9.0からは=>という名前の演算子が作られると警告メッセージを出力していました。

- [xpath\(\)](#)が文字列中の特殊文字を確実にエスケープするようにしました。(Florian Pflug)

これがないと、結果が正当なXMLにならない場合があります。

- オブジェクトが存在しないとき、[pg_relation_size\(\)](#)およびその同類の関数がNULLを返すようにしました。(Phil Sorber)

これにより、同時にDROPをした直後にこれらの関数を呼び出してもエラーを返さなくなります。

- [EXTRACT\(EPOCH FROM timestamp without time zone\)](#)がUTCの0時ではなく、現地時間の0時からの時間を返すようにしました。(Tom Lane)

この変更は、リリース7.3でなされた、誤った考えによる変更を元に戻すものです。UTCの0時から時間を計るのは一貫性がありませんでした。その結果が[timezone](#)の設定に依存することになりますが、timestamp without time zoneの計算はそうになるべきではないからです。入力値をtimestamp with time zoneにキャストすれば、以前と同じ動作が得られます。

- 時間文字列の終わりにyesterday、todayあるいはtomorrowが付いている場合について、適切に処理するようにしました。(Dean Rasheed)

以前は、SELECT '04:00:00 yesterday'::timestampを0時に実行すると、前日の日付を返していました。

- [to_date\(\)](#)および[to_timestamp\(\)](#)に不完全な日付を渡したとき、2020年に向けて丸めるように修正しました。(Bruce Momjian)

以前は、3桁以下の年、および年のマスクを渡したときの丸め方が一貫していませんでした。

E.5.2.3. オブジェクトの操作

- [ALTER DOMAIN](#)がドメインでない型に対して動作しないようにしました。(Peter Eisentraut)

以前は、ドメインでない型に対して所有者とスキーマを変更することができました。

- [CREATE FUNCTION](#)で、手続き言語の名前を強制的に小文字に変換しないようにしました。(Robert Haas)

引用符のついていない言語識別子は引き続き小文字変換されますが、文字列および引用符付きの識別子に対して強制的な小文字変換をしないようにしました。従って、例えば、`CREATE FUNCTION ... LANGUAGE 'C'` は、動作しなくなります。これは'`c`'とする必要がありますが、引用符を付けないのがもっと良いでしょう。

- 外部キーの強制トリガーについてシステムが生成する名前を変更しました。(Tom Lane)

この変更により、自己参照型の外部キー制約のあるいくつかの境界条件においても、トリガーが正しい順番で実行されることが確実にになります。

E.5.2.4. コマンドラインツール

- [psql](#)のメタコマンドの引数における、逆引用符、変数展開、引用符付きの部分文字列の処理を一貫したものにしました。(Tom Lane)

以前は、そのような参照が隣接する文字列と空白で分けられていなかった場合、奇妙な取り扱いをされていました。例えば、'`FOO'BAR`'は`FOO BAR`と出力されていました(想定されない空白を挿入していた)。また、'`FOO'BAR'BAZ`'は変更なしにそのまま出力されていました(大勢の期待と異なり、引用符を削除しなかった)。

- [clusterdb](#)のテーブル名を二重引用符付きとして処理しないようにしました。また、[reindexdb](#)のテーブル名およびインデックス名を二重引用符付きとして処理しないようにしました。(Bruce Momjian)

引用符が必要な時は、コマンドの引数に二重引用符を含める必要があります。

- [createuser](#)は、デフォルトではオプション設定のためのプロンプトを表示しないようになりました。(Peter Eisentraut)

以前と同じ動作をさせるには`--interactive`を使ってください。

- `--interactive`が指定されていないければ、[dropuser](#)はユーザ名のプロンプトを表示しないようにしました。(Peter Eisentraut)

E.5.2.5. サーバの設定

- [サーバ側のSSLファイルの場所](#)を指定するためのサーバパラメータを追加しました。(Peter Eisentraut)

これにより、以前はデータディレクトリ内の`server.crt`、`server.key`、`root.crt`、`root.crl`としてハードコードされていたファイルの名前と場所を変更することができます。サーバは`root.crt`や`root.crl`をデフォルトでは検査しないようになりました。これらのファイルをロードするには、関連するパラメータをデフォルトでない値に設定する必要があります。

- `silent_mode`パラメータを削除しました。(Heikki Linnakangas)

同様の動作は、`pg_ctl start -l postmaster.log`とすることで実現できます。

- `wal_sender_delay`パラメータは、必要とされていないので削除しました。(Tom Lane)

- `custom_variable_classes`パラメータを削除しました。(Tom Lane)

この設定によって提供される検査は信頼できないものでした。これからは、任意の設定の前に任意のクラス名を置くことができます。

E.5.2.6. 監視

- [pg_stat_activity](#).`procpid`の名前を`pid`に変更し、他のシステムテーブルと同じようにしました。(Magnus Hagander)

- [pg_stat_activity](#)にプロセスの状態を表示する新しい列を作りました。(Scott Mead, Magnus Hagander)

アイドル状態のセッションについては、直前の`query`と`query_start`の値が利用可能となるので、高度な分析が可能になります。

- [pg_stat_activity](#).`current_query`は問い合わせが完了しても値がクリアされないで、名前を`query`に変更しました。(Magnus Hagander)

- SQLレベルの時間に関する統計の全ての値を、ミリ秒で測定される`float8`の列に変更しました。(Tom Lane)

この変更により、値がミリ秒まで正確だがそれ以上にはできないという設計上の前提が取り除かれました(`float8`は小数にできるからです)。これにより影響を受ける列は、[pg_stat_user_functions](#).`total_time`、[pg_stat_user_functions](#).`self_time`、[pg_stat_xact_user_functions](#).`total_time`、[pg_stat_xact_user_functions](#).`self_time`です。これらの列の基となっている統計関数も、`bigint`のミリ秒ではなく、`float8`のミリ秒を返すようになります。[contrib/pg_stat_statements](#)の`total_time`列も、ミリ秒で測定されるようになります。

E.5.3. 変更点

以下に、PostgreSQL 9.2とそれ以前のメジャーリリースとの違いに関する詳細な説明をします。

E.5.3.1. サーバ

E.5.3.1.1. パフォーマンス

- 問い合わせがインデックスのみからデータを検索し、ヒープにアクセスしなくても済むことを可能にしました。(Robert Haas, Ibrar Ahmed, Heikki Linnakangas, Tom Lane)

この機能はよくインデックスオンリースキャン(*index-only scan*)と呼ばれます。可視性マップの報告により、すべてのセッションに対して可視であるタプルだけが含まれるヒープページについては、ヒープのアクセスをスキップできます。従って、この利益は主に、専ら静的であるデータに当てはまります。この機能を実装するのに必要な部分の一つとして、可視性マップはクラッシュに対して安全にされました。

- [SP-GiST](#)(Space-Partitioned GiST、空間分割したGiST)インデックスアクセス方式を追加しました。(Teodor Sigaev, Oleg Bartunov, Tom Lane)

SP-GiSTは、柔軟性の点でGiSTに匹敵するものですが、均衡なツリーではなく不均衡に分割された検索構造をサポートします。これが適合する問題では、インデックスの作成時間、検索時間のいずれにおいても、SP-GiSTはGiSTよりも速く動作します。

- 高負荷の状況においてグループコミットが効率的に動作するようにしました。(Peter Geoghegan, imon Riggs, Heikki Linnakangas)

以前は、内部ロックの競合により書込負荷のある状況でコミットをまとめて実行するのは非効率的でした。

- 新しいファストパス(fast-path)ロック機構を使うことで、競合しないロックが管理できるようにしました。(Robert Haas)

- 仮想的なトランザクションIDロックを作成することのオーバーヘッドを削減しました。(Robert Haas)

- ・ シリアライザブル隔離レベルのロックのオーバーヘッドを削減しました。(Dan Ports)
- ・ PowerPCとItaniumでのスピンロックのパフォーマンスを向上しました。(Manabu Ori, Robert Haas, Tom Lane)
- ・ 共有無効キャッシュのメッセージのオーバーヘッドを削減しました。(Robert Haas)
- ・ PGPROC共有メモリ配列のよくアクセスされるメンバを別の配列に移動しました。(Pavan Deolasee, Heikki Linnakangas, Robert Haas)
- ・ ヒープにタブルをバッチで追加することにより、COPYの性能を向上しました。(Heikki Linnakangas)
- ・ より良いツリーをより少ないメモリ割り当てのオーバーヘッドで生成することにより、幾何データ型についてのGiSTインデックスのパフォーマンスを向上しました。(Alexander Korotkov)
- ・ GiSTインデックスの構築時間を向上しました。(Alexander Korotkov, Heikki Linnakangas)
- ・ 一時テーブルおよびログを取らないテーブルについて、ヒントビットをより早くセットできるようにしました。(Robert Haas)
- ・ インラインの、SQLから呼び出し可能でない比較関数によってソートを実行できるようにしました。(Peter Geoghegan, Robert Haas, Tom Lane)
- ・ CLOGバッファの数がshared_buffersに基づいて変わるようにしました。(Robert Haas, Simon Riggs, Tom Lane)
- ・ テーブルまたはデータベースの削除で発生するバッファブールスキャンの性能を向上しました。(Jeff Janes, Simon Riggs)
- ・ 多くのテーブルが削除される、あるいは空にされるときの、チェックポイントによるfsync要求のキューの性能を向上しました。(Tom Lane)
- ・ Windowsの子プロセスに、安全な個数のファイルディスクリプタを渡すようにしました。(Heikki Linnakangas)

これによりWindowsのセッションが以前より多くのオープン状態のファイルディスクリプタを使うことができます。

E.5.3.1.2. プロセス管理

- ・ チェックポイントを実行する専用のバックグラウンドプロセスを作りました。(Simon Riggs)

以前は、バックグラウンドライタがダーティページの書き込みとチェックポイントの両方を行っていました。これを2つのプロセスに分けることにより、それぞれがより予測可能な状態で目的を達することができます。

- ・ walwriterプロセスをより早く再開することで、非同期コミットの動作を改善しました。(Simon Riggs)

以前は、wal_writer_delayが、WALをディスクにフラッシュする唯一のトリガーでした。WALバッファが一杯になったときも、WAL書き込みが起きます。

- ・ 処理が何も行われていない間、bgwriter、walwriter、checkpointer、statistics collector、log collector、archiverのバックグラウンドプロセスが、より効率的に休止できるようにしました。(Peter Geoghegan, Tom Lane)

この一連の変更により、何もすることがないときのプロセスの再開の頻度が減り、アイドル状態のサーバの電力消費が大幅に削減されます。

E.5.3.1.3. オプティマイザ

- ・ プリペアド文が使われる場合であっても、特定のパラメータ値のための特別な計画をプランナが生成できるようにしました。(Tom Lane)

以前は、プリペアド文は必ず1つだけの"一般的な"計画があり、それがすべてのパラメータ値に対して使われていました。これは、明示的な定数値を持つプリペアドでない文で使われる計画に比べ、大幅に劣ることが時々ありました。これからは、プランナは、特定のパラメータ値に対して特別な計画を生成しようとします。特別な計画による利点がないことが繰り返し証明された後になって初めて、一般的な計画が使われます。この変更により、以前からあった、(PL/pgSQLの動的でない文を含め)プリペアド文を使うことによるパフォーマンスの不利を排除できます。

- ・ 入れ子ループで内部インデックススキャンを使えるようプランナを改善しました。(Tom Lane)

新しい"パラメータ化されたパス"の機構により、内部インデックススキャンが、そのスキャンより2つ以上結合レベルが上のリレーションの値を使うことができるようになります。これにより、(外部結合などの)意味的な制約により、可能な結合順序が制限される場合にパフォーマンスが大きく向上します。

- ・ 外部データラッパの計画のAPIを改善しました。(Etsuro Fujita, Shigeru Hanada, Tom Lane)

ラッパがそのテーブルに対して複数のアクセス"パス"を提供できるようになったので、結合の計画がより柔軟になります。

- ・ テーブルでないリレーションについて、自己矛盾する制約句を認知します。(Tom Lane)

この検査はconstraint_exclusionがonに設定されている場合にのみ実行されます。

- ・ 単純なインデックススキャン、およびインデックスオンリースキャンでindexed_col op ANY(ARRAY[...])の条件が使われるようにしました。(Tom Lane)

以前は、このような条件は、ビットマップインデックススキャンでのみ使うことができました。

- ・ boolean型の列でMIN/MAXの索引の最適化をサポートしました。(Marti Raudsepp)

- ・ 行数の推定を設定するとき、SELECTのターゲットリスト内の集合を返す関数について説明するようにしました。(Tom Lane)

- ・ 複製された列を持つインデックスをより信頼できる方法で処理するようプランナを修正しました。(Tom Lane)

- ・ 配列について、要素頻度の統計情報を収集し、利用するようにしました。(Alexander Korotkov, Tom Lane)

この変更により、配列についての<@、@&、@>演算子(配列の含有および重なり)の選択性評価が改善されます。

- ・ 外部テーブルの統計情報を収集できるようにしました。(Etsuro Fujita)

- ・ 部分インデックスの利用について、コスト推定を改善しました。(Tom Lane)

- ・ 副問い合わせで参照される列の統計情報を利用できるようプランナの機能を改善しました。(Tom Lane)

- ・ DISTINCTを使う副問い合わせについて、統計の推定を改善しました。(Tom Lane)

E.5.3.1.4. 認証

- ・ pg_hba.confで指定されるロール名とsameroleが自動的にスーパーユーザを含むとは扱わないようにしました。(Andrew Dunstan)

これにより、グループロールによるrejectの行を使うのが簡単になります。

- pg_hba.confによるトークン解析の処理がより一貫したものになるよう調整しました。(Brendan Jurd, Álvaro Herrera)

- 空のpg_hba.confファイルを禁止しました。(Tom Lane)

これは、間違った構成についてより早く検知するためです。

- スーパーユーザ権限はreplication権限を暗示するものとしました。(Noah Misch)

これにより、そのような権限を明示的に割り当てる必要がなくなります。

E.5.3.1.5. 監視

- バックエンドがクラッシュする間に、現在の問い合わせの文字列のログ出力を試みるようにしました。(Marti Raudsepp)

- 自動バキュームのI/O処理のログ出力を、より冗長にしました。(Greg Smith, Noah Misch)

このログ出力はlog_autovacuum_min_durationをトリガとして実行されます。

- WAL再生の失敗をより早く報告するようにしました。(Fujii Masao)

サーバーがマスターモードになるまで失敗が報告されない場合があります。

- WALの場所の比較を単純にするため、pg_xlog_location_diff()を追加しました。(Euler Taveira de Oliveira)

これはレプリケーションの遅延を計算するのに便利です。

- Windowsでイベントログのアプリケーション名を設定可能にしました。(MauMau, Magnus Hagander)

event_source サーバパラメータを設定することで、異なるインスタンスが異なる識別子を持つイベントログ使うことができるようになります。これは、syslog_identの動作と似ています。

- 実行中のトランザクションがある場合を除き、"unexpected EOF"のメッセージをDEBUG1レベルに変更しました。(Magnus Hagander)

- この変更により、データベース接続を正しく閉じないアプリケーションによって起こされるログファイルへの雑多な書き込みが減少します。

E.5.3.1.6. 統計情報ビュー

- 一時ファイルのサイズとファイル数をpg_stat_databaseシステムビューで追跡するようにしました。(Tomas Vondra)

- デッドロックのカウンタをpg_stat_databaseシステムビューに追加しました。(Magnus Hagander)

- サーバパラメータtrack_io_timingを追加し、I/Oの時間を追跡できるようにしました。(Ants Aasma, Robert Haas)

- チェックポイントの時間の情報をpg_stat_bgwriterで報告するようにしました。(Greg Smith, Peter Geoghegan)

E.5.3.1.7. サーバの設定

- search_pathで指定したスキーマが存在しないとき、黙って無視するようにしました。(Tom Lane)

- これにより、すべてのデータベースには存在しないスキーマ名を含むような一般的なパス設定の利用が、より便利になります。

- スーパーユーザが、deadlock_timeoutについて、クスタ単位だけでなく、セッション単位でも設定できるようにしました。(Noah Misch)

これにより、デッドロックに関連しそうなトランザクションについてdeadlock_timeoutを小さくできるので、より速く検出できるようになります。また、この値を大きくすることで、デッドロックにより、あるセッションが取消対象として選択される可能性を小さくするために使うこともできます。

- セッションあたりの一時ファイルの容量を制限するため、サーバパラメータtemp_file_limitを追加しました。(Mark Kirkwood)

- スーパーユーザだけが設定できる拡張の独自変数について、関連する拡張をロードする前にスーパーユーザがSETできるようにしました。(Tom Lane)

SETがスーパーユーザによって実行されたかどうか、システムは記憶するようになります。これにより、拡張をロードするときに、適切な権限の検査が可能になります。

- 設定パラメータについて問い合わせをするpostmasterの-cオプションを追加しました。(Bruce Momjian)

これにより、PGDATAあるいは-Dが、設定ファイルだけがインストールされたディレクトリを指している場合について、pg_ctlがより良い処理をできるようになります。

- CREATE DATABASEで、空のロケール名を暗示された値で置換するようにしました。(Tom Lane)

これにより、サーバの再起動後に、pg_database.datcollateあるいはdatatypeが異なる解釈をされることを防ぎます。

E.5.3.1.7.1. postgresql.conf

- postgresql.confで、最初のエラーだけでなく、複数のエラーが報告されるようにしました。(Alexey Klyukin, Tom Lane)

- postgresql.confの再読み込みで、特定のセッションにとって不正な設定が含まれる場合でも、すべてのセッションによって処理されるようにしました。(Alexey Klyukin)

以前は、そのようなセッション内で有効でない値があると、そのセッションは全ての設定変更を無視していました。

- 設定ファイルにinclude_if_existsの機能を追加しました。(Greg Smith)

これは、ファイルが見つからなくてもエラーを発生させない、という点を除き、includeと同じ動作をします。

- initdbの実行中にサーバの時間帯を調べ、postgresql.confのtimezoneおよびlog_timezoneエントリを適切に設定するようにしました。(Tom Lane)

これにより、時間のかかる時間帯の調査をサーバの起動時に実施しないですみます。

- Windowsでpostgresql.confの行番号が報告されるように、pg_settingsを修正しました。(Tom Lane)

E.5.3.2. レプリケーションとリカバリ

- ストリーミングレプリケーションのスレーブが、他のスレーブにデータを転送できるようにしました(カスケードレプリケーション)。(Fujii Masao)

以前は、スタンバイサーバにストリーミングレプリケーションのログファイルを提供できるのはマスターサーバだけでした。

- [synchronous_commit](#)のモードに新しく[remote_write](#)を追加しました。(Fujii Masao, Simon Riggs)

このモードでは、スタンバイサーバがトランザクションデータをスタンバイのオペレーティングシステム上で書き込むのを待ちますが、スタンバイのディスクにデータがフラッシュされるのは待ちません。

- [pg_receivexlog](#)ツールを追加しました。これは、完全なWALファイルを待つ代わりに、WALファイルの変更を、書き込まれたときにアーカイブします。(Magnus Hagander)
- [pg_basebackup](#)でスタンバイサーバからバックアップを作れるようにしました。(Jun Ishizuka, Fujii Masao)
この機能により、新しいベースバックアップを作る作業について、プライマリサーバの負荷を減らすことができます。
- [pg_basebackup](#)がバックアップを実行している間も、WALファイルをストリーム転送できるようにしました。(Magnus Hagander)
これにより、WALファイルがプライマリ上で捨てられる前に、スタンバイに転送することができます。

E.5.3.3. 問い合わせ

- クライアントが切断されたら、実行中の問い合わせを取り消すようにしました。(Florian Pflug)
- 問い合わせの実行中にクライアントの接続が失われたことをバックエンドが検知したら、問い合わせを完了しようとせずにキャンセルします。
- 行式において実行時に列名を保持するようにしました。(Andrew Dunstan, Tom Lane)

この変更により、行の値が`hstore`あるいは`json`型に変換される場合により良い結果が得られます。つまり、返される値のフィールドには期待される名前がつけられます。

- [副SELECT](#)の結果に使われる列のラベルを改善しました。(Marti Raudsepp)

以前は、`?column?`という一般的なラベルが使われていました。

- 不明の値の型を決定する経験則を改善しました。(Tom Lane)

不明の定数は、それを使う演算子のもう一方の側の値と同じ型を持つかもしれないという長年使われていた規則を、単純な演算子の場合だけでなく、多様演算子を考慮する場合も適用するようにしました。

- ドメイン型へのキャスト、およびドメイン型からのキャストの作成について警告を出すようにしました。(Robert Haas)

そのようなキャストには何の効果もありません。

- 行が[CHECK](#)制約、あるいは[NOT NULL](#)制約に違反したとき、行の内容をエラー詳細として表示するようにしました。(Jan Kundrát)

これにより、insertやupdateが多数の行を処理しているときに、どの行に問題があるか特定するのが容易になるはずです。

E.5.3.4. オブジェクトの操作

- DDLの同時実行中に、より信頼できる操作を提供しました。(Robert Haas, Noah Misch)

この変更ではロックを追加し、それにより、多くの状況で"cache lookup failed"(キャッシュの検索に失敗)のエラーを起こさなくなります。また、同時削除中のスキーマにリレーションを追加することはできなくなりました。以前はこれによりシステムカタログの内容が不整合を起こすことがありました。

- [DROP INDEX](#)に[CONCURRENTLY](#)オプションを追加しました。(Simon Riggs)

これにより、他のセッションを妨害することなくインデックスを削除できます。

- 外部データラッパに列ごとのオプションが付けられるようにしました。(Shigeru Hanada)

- ビューの定義の見やすい表示(pretty-print)を改善しました。(Andrew Dunstan)

E.5.3.4.1. 制約

- [CHECK](#)制約を[NOT VALID](#)として宣言できるようにしました。(Álvaro Herrera)

[NOT VALID](#)の制約を追加しても、既存の行が制約を満たすかどうかを検査するテーブルスキャンは実行されません。その後に、新規に追加あるいは更新する行は検査されます。すべての行が制約を満たすかどうか確かではありませんから、プランナが[constraint_exclusion](#)を考慮する際、そのような制約は無視されます。

新しい[ALTER TABLE VALIDATE](#)コマンドは[NOT VALID](#)の制約を既存の行について検査することができ、その後で通常の制約に変換します。

- [CHECK](#)制約を[NO INHERIT](#)として宣言できるようにしました。(Nikhil Sontakke, Alex Hunsaker, Álvaro Herrera)

こうすると、制約は親テーブルでのみ有効で、子テーブルでは無効になります。

- 制約の[名前を変更](#)できる機能を追加しました。(Peter Eisentraut)

E.5.3.4.2. ALTER

- いくつかの[ALTER TABLE ... ALTER COLUMN TYPE](#)の操作において、テーブルとインデックスの再作成の必要性を削減しました。(Noah Misch)

[varchar](#)あるいは[varbit](#)の列の長さの制限を大きくする、あるいは制限をなくしてしまう場合には、テーブルの再書き込みは必要なくなりました。[numeric](#)列の精度を大きくする場合、あるいは列を制限のある[numeric](#)から制限のない[numeric](#)に変更する場合は、テーブルの再書き込みは必要なくなりました。[interval](#)型、[timestamp](#)型、[timestampz](#)型についての同様なケースでもテーブルの再書き込みは行われません。

- [ALTER TABLE](#)の外部キーの再検証について、不要な場合には実行しないようにしました。(Noah Misch)

- いくつかの[ALTER](#)コマンドに[IF EXISTS](#)オプションを追加しました。(Pavel Stehule)

例えば、[ALTER FOREIGN TABLE IF EXISTS foo RENAME TO bar](#)のように使います。

- [ALTER FOREIGN DATA WRAPPER](#)および[ALTER SERVER](#)に[RENAME](#)を追加しました。(Peter Eisentraut)

- [ALTER DOMAIN](#)に[RENAME](#)を追加しました。(Peter Eisentraut)

以前も[ALTER TYPE](#)を使えばドメインの名前の変更は可能でした。

- `ALTER DOMAIN`の`DROP CONSTRAINT`で、存在しない制約を指定したらエラーを発生させるようにしました。(Peter Eisentraut)
- `IF EXISTS`オプションが追加されたので、以前と同じ動作をさせることは可能です。

E.5.3.4.3. `CREATE TABLE`

- 外部テーブル、ビューおよび複合型からの`CREATE TABLE (LIKE ...)`を可能にしました。(Peter Eisentraut)
- 例えば、テーブルのスキーマがビューにマッチするようなテーブルを作成することも可能になります。
- `CREATE TABLE (LIKE ...)`でインデックスのコメントをコピーするときに、インデックス名の衝突が起きないように修正しました。(Tom Lane)
- `CREATE TABLE ... AS EXECUTE`で、`WITH NO DATA`と列名の指定を処理できるように修正しました。(Tom Lane)

E.5.3.4.4. オブジェクトの権限

- ビューに`security_barrier`オプションを追加しました。(KaiGai Kohei, Robert Haas)
- このオプションは、ビューが保護するデータがユーザに開示されるような最適化を防ぎます。例えば、安全でない関数を含む句をビューの`WHERE`句に入れるようなことです。このようなビューは、通常のビューに比べパフォーマンスが悪くなることが予想されます。
- 新しい関数の属性`LEAKPROOF`を追加しました。これは、`security_barrier`のビューで安全に利用できる関数に印を付けるものです。(KaiGai Kohei)
- データ型についての権限の機能を追加しました。(Peter Eisentraut)
- これにより、型とドメインについて、SQL準拠の`USAGE`権限がサポートされます。その目的は、どのユーザが型の依存性を作成できるか制限することです。というのは、そのような依存性は型の所有者による型定義の変更の能力を制限することになるからです。
- `SELECT INTO`および`CREATE TABLE AS`で、`INSERT`権限について確認するようにしました。(KaiGai Kohei)
- `SELECT INTO`や`CREATE TABLE AS`ではオブジェクトが作られるので、作成者は通常は`insert`権限を持っています。しかし、`ALTER DEFAULT PRIVILEGES`によりそのような権限を取り消した場合など、そうならない例外的な場合もあります。

E.5.3.5. ユーティリティ操作

- `VACUUM`でページをロックできない時は、そのページをより安易にスキップするようにしました。(Simon Riggs, Robert Haas)
- この変更により、`VACUUM`が"詰まって"しまう事故が大幅に減るでしょう。
- `EXPLAIN (BUFFERS)`がダーティーにされたブロック数、書き込まれたブロックを数えるようにしました。(Robert Haas)
- `EXPLAIN ANALYZE`がフィルタの段階で除外した行数を報告するようにしました。(Marko Tiikkaja)
- `EXPLAIN ANALYZE`で時間の値が必要とされていないとき、時間測定のおーバーヘッドがなくなるようにしました。(Tomas Vondra)
- これは新しい`TIMING`オプションを`FALSE`に設定することで実現できます。

E.5.3.6. データ型

- `範囲データ型`の機能を追加しました。(Jeff Davis, Tom Lane, Alexander Korotkov)
- 範囲データ型は、その基本のデータ型に属する下限値と上限値を記録します。含有、重なり、共通部分などの演算をサポートします。
- `JSON`データ型を追加しました。(Robert Haas)
- この型はJSON(JavaScript Object Notation)データを適切に検証して保存します。
- `array_to_json()`および`row_to_json()`を追加しました。(Andrew Dunstan)
- `SMALLSERIAL`データ型を追加しました。(Mike Pultz)
- これは`SERIAL`と似ていますが、シーケンスを2バイトの整数の列(`int2`)に格納します。
- `ドメイン`を`NOT VALID`として宣言できるようにしました。(Álvaro Herrera)
- このオプションはドメインの作成時にも指定できますし、また`ALTER DOMAIN > ... ADD CONSTRAINT ... NOT VALID`としても設定できます。`ALTER DOMAIN ... VALIDATE CONSTRAINT`とすると、制約を完全に検証します。
- `money`データ型で、サポートするロケール独自の書式オプションの数を増やしました。(Tom Lane)
- 具体的には、金額の出力における値の順序、符号、通貨記号についてすべてのPOSIXのオプションを尊重するようにしました。また、3桁毎の区切り文字も、POSIXの要請に従い、小数点の左側にだけ挿入されることを確実にしました。
- `macaddr`データ型について、ビット演算の`"and"`、`"or"`、`"not"`演算子をサポートしました。(Brendan Jurd)
- `xpath()`にスカラー値が渡されたとき、1つだけの要素を持つXML配列を返すようにしました。(Florian Pflug)
- 以前は空の配列を返していました。この変更に伴い、そのような式について、`xpath_exists()`は偽ではなく真を返すようになります。
- XMLのエラー処理がより堅牢になるよう改善しました。(Florian Pflug)

E.5.3.7. 関数

- スーパーユーザでないユーザが、同じユーザの他のセッションに対して`pg_cancel_backend()`および`pg_terminate_backend()`を使うことができるようにしました。(Magnus Hagander, Josh Kopershmidt, Dan Farina)
- 以前は、スーパーユーザだけがこれらの関数を使うことができました。
- トランザクションスナップショットのインポートとエクスポートができるようにしました。(Joachim Wieland, Tom Lane)
- これにより、複数のトランザクションがデータベースの状態に関する同一のビューを共有することができるようになります。スナップショットのエクスポートは`pg_export_snapshot()`によって、インポートは`SET TRANSACTION SNAPSHOT`によって行います。現在実行中のトランザクションのスナップショットだけがインポート

可能です。

- 式で[COLLATION_FOR](#)をサポートしました。(Peter Eisentraut)

これは、式の照合順序を表す文字列を返します。

- [pg_opfamily_is_visible\(\)](#)を追加しました。(Josh Kuperushmidt)
- [pg_size_pretty\(\)](#)を[pg_xlog_location_diff\(\)](#)で使えるように、`numeric`型の亜種を追加しました。(Fujii Masao)
- [pg_trigger_depth\(\)](#)関数を追加しました。(Kevin Grittner)

この関数は、現在のトリガの呼び出し階層数を返します。

- [string_agg\(\)](#)で`bytea`型の値を処理できるようにしました。(Pavel Stehule)
- 正規表現で、より大きな量指定された部分表現の中で後方参照が使われる場合について修正しました。(Tom Lane)

例えば、`^(¥w+)( ¥1)+$`です。 以前のリリースでは、後方参照が実際に最初の出現とマッチしていることを検査していませんでした。

E.5.3.8. 情報スキーマ

- 情報スキーマにビュー[role_udt_grants](#)、[udt_privileges](#)、[user_defined_types](#)を追加しました。(Peter Eisentraut)
 - 情報スキーマのビュー[element_types](#)に、複合型の属性を追加しました。(Peter Eisentraut)
 - 情報スキーマに[interval_type](#)列を実装しました。(Peter Eisentraut)
- 以前はこの列はNULLとして読み出されていました。
- 情報スキーマのビュー[attributes](#)、[columns](#)、[domains](#)、[element_types](#)で、照合順序に関連した列を実装しました。(Peter Eisentraut)
 - 情報スキーマのビュー[table_privileges](#)で[with_hierarchy](#)列を実装しました。(Peter Eisentraut)
 - 情報スキーマでシーケンスの[USAGE](#)権限の表示を追加しました。(Peter Eisentraut)
 - 情報スキーマがデフォルト権限を表示するようにしました。(Peter Eisentraut)

以前は空でないデフォルト権限はビューに表示されませんでした。

E.5.3.9. サーバサイド言語

E.5.3.9.1. [PL/pgSQL](#)サーバサイド言語

- PL/pgSQLのカーソルの[OPEN](#)コマンドで、パラメータを名前指定できるようにしました。(Yeb Havinga)
- PL/pgSQLで例外情報を取得する[GET STACKED DIAGNOSTICS](#)コマンドを追加しました。(Pavel Stehule)
- 型情報をキャッシュすることにより、PL/pgSQLでの配列の代入を高速化しました。(Pavel Stehule)
- PL/pgSQLで[ELIF](#)句が長く続いているときのパフォーマンスとメモリ消費を改善しました。(Tom Lane)
- PL/pgSQLのエラーメッセージで、関数の名前だけでなくシングネチャも出力するようにしました。(Pavel Stehule)

E.5.3.9.2. [PL/Python](#)サーバサイド言語

- PL/PythonにSPIカーソルのサポートを追加しました。(Jan Urbanski)
- これにより、PL/Pythonが結果集合の一部だけを読むことができるようになります。
- PL/Pythonに結果メタデータ関数を追加しました。(Peter Eisentraut)
- 具体的には、結果オブジェクト関数[colnames](#)、[coltypes](#)、[coltypmods](#)が追加されました。
- Python 2.2をサポートから外しました。(Peter Eisentraut)

E.5.3.9.3. [SQL](#)サーバサイド言語

- SQL言語による関数で、パラメータを名前参照できるようにしました。(Matthew Draper)
- これを使うには、関数の引数を名前にして、その引数名をSQL関数の本体から参照するだけで良いです。

E.5.3.10. クライアントアプリケーション

- [initdb](#)のオプションに、`--auth-local`と`--auth-host`を追加しました。(Peter Eisentraut)
- これにより、[pg_hba.conf](#)の認証設定で、`local`と`host`を別々に設定することができます。 `--auth`は引き続き、両方の設定を行います。
- [createuser](#)で、レプリケーションの権限を設定するために、`--replication/--no-replication`のフラグを追加しました。(Fujii Masao)
 - [dropdb](#)と[dropuser](#)に`--if-exists`オプションを追加しました。(Josh Kuperushmidt)
 - コマンドラインツールが接続先のデータベースを指定でき、また[postgres](#)データベースへの接続に失敗した時は、`template1`に接続するようにしました。(Robert Haas)

E.5.3.10.1. [psql](#)

- 端末の幅に合わせて、出力を自動的に拡張する表示モードを追加しました。(Peter Eisentraut)
- `¥x`コマンドに[auto](#)オプションを追加しました。 通常の出力が画面の幅より広いとき、拡張モードに切り替えます。
- 起動に使われたファイルのディレクトリからの相対パス名でスクリプトファイルを指定できるようにしました。(Gurjeet Singh)

これは新しいコマンド`¥ir`で実行できます。

- `psql`の変数名で、非ASCII文字の使用をサポートしました。(Tom Lane)
- 特定メジャーバージョン用の`.psqlrc`ファイルをサポートしました。(Bruce Momjian)
`psql`は以前から、特定マイナーバージョン用の`.psqlrc`ファイルはサポートしていました。
- `psql`の履歴ファイル、起動ファイルの位置について、環境変数で優先指定できるようにしました。(Andrew Dunstan)
`PSQL_HISTORY`と`PSQLRC`が設定されていれば、そこで指定されたファイル名が使われます。
- 子プロセスに渡される環境変数を変更できる`¥setenv`コマンドを追加しました。(Andrew Dunstan)
- `psql`のエディタの一時ファイルで`.sql`を拡張子に使うようにしました。(Peter Eisentraut)
拡張子によって動作が変わるエディタは、これにより正しいモードを選択できるようになります。
- `psql`で、0バイトのフィールド区切り文字およびレコード区切り文字を使えるようにしました。(Peter Eisentraut)
様々なシェルツール(例: `find`)で、0バイトの区切り文字(NUL)を使うことができます。
- `¥timing`オプションで、失敗した問い合わせについても時間を表示するようにしました。(Magnus Hagander)
以前は、成功した問い合わせについてのみ時間が表示されていました。
- `psql`による`¥copy`とSQLの`COPY`の処理を統一し、また厳格にしました。(Noah Misch)
この修正により、失敗した時の動作が、より予測できるものになり、また、`¥set ON_ERROR_ROLLBACK`を尊重したのになります。

E.5.3.10.2. 情報表示コマンド

- シーケンスについての`¥d`で、シーケンスを所有するテーブルと列の名前を表示するようにしました。(Magnus Hagander)
- `¥d+`で列の統計対象を表示するようにしました。(Magnus Hagander)
- `¥du`でロールのパスワードの有効期限日を表示するようにしました。(Fabrício de Royes Mello)
- キャスト、変換、ドメイン、言語についてコメントを表示するようにしました。(Josh Kopershmidt)
これは、それぞれ`¥dc+`、`¥dc+`、`¥dD+`、`¥dL`の出力に含まれます。
- SQL/MEDオブジェクトのコメントを表示するようにしました。(Josh Kopershmidt)
これは、外部サーバ、外部テーブル、外部データラップのそれぞれについて、`¥des+`、`¥det+`、`¥dew+`の出力に含まれます。
- `¥dd`を変更し、その種のオブジェクト型専用のバックスラッシュコマンドを持たないものについてのみコメントを表示するようにしました。(Josh Kopershmidt)

E.5.3.10.3. タブ補完

- `psql`のタブ補完で、SQLのキーワードについては、新しい`COMP_KEYWORD_CASE`の設定に基づいて、大文字あるいは小文字で補完するようにしました。(Peter Eisentraut)
- `EXECUTE`についてタブ補完ができるようにしました。(Andreas Karlsson)
- `GRANT/REVOKE`でのロールの参照についてタブ補完ができるようにしました。(Peter Eisentraut)
- ファイル名のタブ補完で、必要な引用符を補完できるようにしました。(Noah Misch)
- `TABLE`についてのタブ補完で、ビューも含むように変更しました。(Magnus Hagander)

E.5.3.10.4. `pg_dump`

- `pg_dump`に`--exclude-table-data`オプションを追加しました。(Andrew Dunstan)
これにより、テーブルの定義はダンプするがそのデータは出力しない、ということをテーブル毎に指定できます。
- `pg_dump`と`pg_restore`に`--section`オプションを追加しました。(Andrew Dunstan)
有効な値は`pre-data`、`data`、`post-data`です。このオプションは複数回指定することで、2つ以上のセクションを選択することができます。
- `pg_dumpall`で、まずすべてのロールについてダンプし、それからロールのすべての構成設定についてダンプするようにしました。(Phil Sorber)
これにより、ロールの構成設定が他のロールを参照する場合でもエラーを発生しなくなります。
- 新しいクラスタに`postgres`データベースが存在しない場合でも、`pg_dumpall`がエラーを起こさないようにしました。(Robert Haas)
- 外部サーバのユーザマッピングをユーザ名の順にダンプするようにしました。(Peter Eisentraut)
これにより、予測可能なダンプファイルが生成されます。
- 演算子を予測可能な順序でダンプするようにしました。(Peter Eisentraut)
- 拡張設定テーブルが`pg_dump`でダンプされる時の規則を厳格にしました。(Tom Lane)
- `pg_dump`が、より役に立つ依存性の情報を吐き出すようにしました。(Tom Lane)
アーカイブ形式のダンプに含まれる依存性のリンクは、ダンプ内のどこにも現れないオブジェクトを参照することが多く、以前は非常に限られた使い方ができませんでした。これからはダンプされるオブジェクトの間の実際の依存性(間接的なこともある)を出力します。
- `pg_dump`が多くのデータベースオブジェクトをダンプするときの性能を向上しました。(Tom Lane)

E.5.3.11. `libpq`

- `libpq`の接続文字列について`URI`の形式にすることを可能にしました。(Alexander Shulgin)

その構文は`postgres://`で始まります。これにより、データベース接続を表現するURIについて、アプリケーションが独自のパーサを実装しなくても済むようになります。

- SSL圧縮を使用不可にする[接続オプション](#)を追加しました。(Laurenz Albe)
高速なネットワークでSSL圧縮によるオーバーヘッドをなくすために使うことができます。
- 大きな結果集合をより良く処理できるように、[1行処理モード](#)を追加しました。(Kyotaro Horiguchi, Marko Kreen)
以前は、`libpq`は必ず問い合わせ結果の全体をメモリに取り込んでから、アプリケーションに返していました。
- 関数`PQconnectdbParams`、`PQconnectStartParams`、`PQpingParams`の宣言に`const`修飾子を追加しました。(Lionel Elie Mamane)
- `.pgpass`ファイルで、パスワードフィールドにエスケープ文字を含められるようにしました。(Robert Haas)
- プロセスを終了させる必要があるとき、ライブラリ関数が`exit()`ではなく、`abort()`を使うようにしました。(Peter Eisentraut)
この選択は、プログラムが使う通常の終了コードと競合せず、また呼び出し側が捕捉できるシグナルを生成します。

E.5.3.12. ソースコード

- 使われていないポートを削除しました。(Peter Eisentraut)
以下のプラットフォームはサポートされなくなりました。 `dgux`、`nextstep`、`sunos4`、`svr4`、`ultrix4`、`univel`、`bsd`。
- [MS Visual Studio 2010](#)でのビルドをサポートしました。(Brar Piening)
- MinGW-w64の32ビットコンパイラでのコンパイルを可能にしました。(Lars Kanis)
- インストール時に`pqsql.h`を`include/server`にインストールするようにしました。(Heikki Linnakangas)
- ラッチ機構に`postmaster`の終了の検知を含めるよう改善しました。(Peter Geoghegan, Heikki Linnakangas, Tom Lane)
これにより、バックグラウンドプロセスが以前はイベントのポーリングのために再開しなければならなかった主要な理由の1つが取り除かれます。
- Cの柔軟な配列のメンバーを(サポートされていれば)使うようにしました。(Peter Eisentraut)
- 同時トランザクションのリグレーションテスト(`isolationtester`)を改善しました。(Noah Misch)
- `thread_test`を修正し、テストファイルを`/tmp`でなく、カレントディレクトリに作成するようにしました。(Bruce Momjian)
- `flex`と`bison`の警告およびエラー報告を改善しました。(Tom Lane)
- メモリバリアのサポートを追加しました。(Robert Haas)
これは、現在は使用されていません。
- `pgindent`を修正し、`typedef`ファイルを使うようにしました。(Bruce Momjian)
- サーバのログに送信されることになっているメッセージを処理するフックを追加しました。(Martin Pihlak)
- `DROP`コマンドに、オブジェクトアクセスのフックを追加しました。(KaiGai Kohei)
- いくつかのオブジェクト型について`DROP`の処理を一つにまとめました。(KaiGai Kohei)
- `pg_upgrade`テストスイートを追加しました。(Peter Eisentraut)
- 正規表現のコードをTCL 8.5.11と同期し、内部処理を改善しました。(Tom Lane)
- CRCテーブルを`libpgport`に移し、それを別のインクルードファイルで提供するようにしました。(Daniel Farina)
- メジャーバージョンのリリースノートの作成で使えるように、`git_changelog`のオプションを追加しました。(Bruce Momjian)
- Linuxの`/proc/self/oom_score_adj` APIをサポートしました。(Tom Lane)

E.5.3.13. 追加モジュール

- `libpq`の新しい1行処理モードを使うことで、[dblink](#)の性能を向上しました。(Kyotaro Horiguchi, Marko Kreen)
この改善は、`dblink_send_query()`/`dblink_get_result()`には適用されません。
- [file_fdw](#)で`force_not_null`オプションをサポートしました。(Shigeru Hanada)
- [pg_archivecleanup](#)の試運転モードを実装しました。(Gabriele Bartolini)
これは、削除されることになるファイルの名前を出力するだけです。
- [pgbench](#)に新しいスイッチ`--unlogged-tables`、`--tablespace`、`--index-tablespace`を追加しました。(Robert Haas)
- [pg_test_fsync](#)が、一定回数のサイクル内ではなく、一定の時間内についてテストするように変更しました。(Bruce Momjian)
`-o`/サイクル数、のオプションを削除し、`-s`/秒数、のオプションを追加しました。
- 時計の単調性と時間測定のオーバーヘッドを測定するユーティリティ、[pg_test_timing](#)を追加しました。(Ants Aasma, Greg Smith)
- [tcn](#)(Triggered Change Notification、トリガによる変更の通知)モジュールを追加しました。これはテーブルの変更について、`NOTIFY`イベントを生成します。(Kevin Grittner)

E.5.3.13.1. [pg_upgrade](#)

- `pg_upgrade`の環境変数を変更しました。(Bruce Momjian)
`data`、`bin`、`port`の環境変数名を`PG`で始まるものに変更しました。 `PGPORT`の代わりに`PGPORTOLD`/`PGPORTNEW`を使うようにしました。

- pg_upgradeのログ出力とエラー報告について大幅に見直しました。(Bruce Momjian)

追記書き込みのみのファイルを4つ作り、成功したときはそれらを削除するようにしました。これらのファイルを無条件に残す、`-r/--retain`オプションを追加しました。pg_upgradeの`-g/-G/-1`オプションは不要なので削除しました。また、ログファイルのパーミッションを厳しくしました。

- pg_upgradeで、より正確なオプティマイザ用統計情報を少しずつ生成するスクリプトを作成するようにしました。(Bruce Momjian)

これにより、アップグレード後に最小限のクラスタの統計情報を生成するのに要する時間を削減できます。

- pg_upgradeで、`postgres`データベースのない古いクラスタをアップグレードできるようにしました。(Bruce Momjian)

- pg_upgradeで、一部の古い、あるいは新しいデータベースが見つからなくても、それらが空であれば処理できるようにしました。(Bruce Momjian)

- pg_upgradeで、設定ファイルだけがディレクトリにインストールされている場合を処理できるようにしました。(Bruce Momjian)

- pg_upgradeでサーバーにパラメータを渡す`-o/-O`オプションを追加しました。(Bruce Momjian)

これは、設定ファイルだけがディレクトリにインストールされている場合に便利です。

- pg_upgradeがデフォルトでポート50432を使うように変更しました。(Bruce Momjian)

これにより、アップグレード中にクライアントが意図せず接続してしまうのを防ぐことができます。

- pg_upgradeにおけるクラスタのロックを削減しました。(Bruce Momjian)

具体的には、リンクモードが利用されている場合に古いクラスタをロックするだけで、それをスキーマがリストアされた直後に実行します。

E.5.3.13.2. [pg_stat_statements](#)

- pg_stat_statementsで、SQLテキストの正規化により、類似した問い合わせを1つにまとめられるようにしました。(Peter Geoghegan, Tom Lane)

パラメータ化されていないSQLを使うアプリケーションのユーザは、ログファイルの詳細な分析をしなくても、問い合わせのパフォーマンスを監視できるようになります。

- ダーティーにされたブロック数、書き込まれたブロック数、読み書きの回数をpg_stat_statementsに追加しました。(Robert Haas, Ants Aasma)

- pg_stat_statementsが`PREPARE` および `EXECUTE` コマンドを二重に数えないようにしました。(Tom Lane)

E.5.3.13.3. [sepgsql](#)

- グローバルオブジェクトについて`SECURITY LABEL`をサポートしました。(KaiGai Kohei, Robert Haas)

具体的には、データベース、テーブル空間、ロールについてセキュリティラベルを追加しました。

- sepgsqlがデータベースのラベルを尊重するようにしました。(KaiGai Kohei)

- 様々なオブジェクトの作成にあたり、sepgsqlのパーミッションの検査を行うようにしました。(KaiGai Kohei)

- `sepgsql_setcon()` およびそれに関連した関数を追加しました。これらはsepgsqlのセキュリティドメインを制御します。(KaiGai Kohei)

- パフォーマンスを向上するため、sepgsqlにユーザ空間のアクセスのキャッシュを追加しました。(KaiGai Kohei)

E.5.3.14. 文書

- オプションで、Webサイトのスタイルシートを使ってHTML文書をビルドするルールを追加しました。(Magnus Hagander)

`gmake STYLE=website draft` を使ってください。

- `EXPLAIN`の文書を改善しました。(Tom Lane)

- `vacuumdb`のようなコマンドラインツールを使うときに、ユーザ名やデータベース名が二重引用符付きとして維持されることを文書に記述しました。(Bruce Momjian)

- MD5認証でクライアントによって返される実際の文字列について文書に記述しました。(Cyan Ogilvie)

- `CREATE TEMP TABLE` で`GLOBAL`と`LOCAL`の利用を勧められないものとししました。(Noah Misch)

PostgreSQLでは長い間、これらのキーワードを意味のないものとして扱って来ましたが、今後もそのようにし続けます。しかし将来、それらは標準SQLが意図しているものになるかも知れませんが、アプリケーションはそれらを使わないようにすべきです。

E.6. リリース9.1.9

リリース日: 2013-04-04

このリリースは9.1.8に対し、各種不具合を修正したものです。9.1メジャーリリースにおける新機能については、[項E.15](#)を参照してください。

E.6.1. バージョン9.1.9への移行

9.1.Xからの移行ではダンプ/リストアは不要です。

しかし、このリリースではGiSTインデックスの管理についていくつかの不具合を修正しています。このアップデートをインストールした後、以下に記述する条件の1つ以上に該当するGiSTインデックスすべてについて、`REINDEX`を実行することを推奨します。

また、9.1.6よりも前のバージョンからアップグレードする場合は、9.1.6のリリースノートを参照してください。

E.6.2. 変更点

- サーバのコマンド行スイッチについての安全でない解析を修正しました。(Mitsumasa Kondo, Kyotaro Horiguchi)

"-"で始まるデータベース名を含む接続要求に細工をすることで、例えその要求が最終的に拒絶された場合でも、サーバのデータディレクトリ内のファイルを損傷させる、あるいは破壊することが可能でした。(CVE-2013-1899)

- OpenSSLの乱数の状態を、postmasterのそれぞれの子プロセスでリセットするようにしました。(Marko Kreen)

`contrib/pgcrypto`関数によって生成された乱数が、他のデータベースユーザによって比較的推定しやすいという問題が、これにより回避されます。この危険はpostmasterが`ssl = on`で構成されている時にのみ問題となりますが、ほとんどの接続ではSSL暗号化を利用していません。(CVE-2013-1900)

- REPLICATION権限の検査で、認証されたユーザでなく、現在のユーザについて調べるようにしました。(Noah Misch)

この誤りを利用すると、権限のないデータベースユーザが`pg_start_backup()`や`pg_stop_backup()`を呼び出し、それにより定期的なバックアップの作成を妨害することが可能でした。(CVE-2013-1901)

- GiSTインデックスで、"fuzzy"な幾何比較を使うのが適切でないときは、使わないようにしました。(Alexander Korotkov)

幾何データ型のコアでは"fuzzy"な同値性を利用して比較を行ないますが、`gist_box_same`では正確な比較を行う必要があります、そうしなければ、それを利用するGiSTインデックスは一貫しないものになります。このアップデートをインストールした後、`box`、`polygon`、`circle`、`point`型の列のすべてのGiSTインデックスについて`REINDEX`を実行してください。これらはいずれも`gist_box_same`を使っています。

- 可変長のデータ型、つまり`text`、`bytea`、`bit`、`numeric`型の列で`contrib/btree_gist`を使うGiSTインデックスについて誤った範囲の結合と不正なロジックを修正しました。(Tom Lane)

この誤りにより、存在するキーが検索で見つからないような一貫性のないインデックスが作成されたり、インデックスが無駄に大きくなることがありました。このアップデートをインストールした後、そのようなインデックスについて`REINDEX`を実行することを推奨します。

- 複数列についてのインデックスで、GiSTページを分割するコードの不具合を修正しました。(Tom Lane)

この誤りにより、存在するキーが検索で見つからないような一貫性のないインデックスが作られることや、検索が非常に非効率になるインデックスが作られることがありました。このアップデートをインストールした後、複数列についてのGiSTインデックスについて`REINDEX`を実行することを推奨します。

- `gist_point_consistent`で曖昧さを一貫性のある方法で処理するようにしました。(Alexander Korotkov)

`point`型の列のGiSTインデックスについてのインデックススキャンは、シーケンシャルスキャンとは異なる結果になる場合があります。これは、`gist_point_consistent`と演算子のコードで、厳密な比較をするか、曖昧な比較をするかについて不一致があったからです。

- WAL再生におけるバッファのリークを修正しました。(Heikki Linnakangas)

この不具合により、再生中に"incorrect local pin count"のエラーが発生し、リカバリができなくなる場合があります。

- `DELETE RETURNING`での競合条件を修正しました。(Tom Lane)

特定の条件下で、`DELETE RETURNING`は現在のプロセスがもはや確保していない共有バッファからデータをフェッチすることがありました。その間に他のプロセスがそのバッファを変更した場合、ゴミの値が`RETURNING`の出力となったり、あるいはクラッシュすることもありました。

- 正規表現のコンパイルで無限ループになる可能性があるのを修正しました。(Tom Lane, Don Porter)
- 正規表現のコンパイル時にNULLポインタから参照先を取得する可能性があるのを修正しました。(Tom Lane)
- `to_char()`で、それが適切な場合、ASCIIのみの大文字変換規則を利用するように修正しました。(Tom Lane)

これにより、ロケールに依存すべきでないいくつかのテンプレートのパターンにおける誤った動作が修正されます。トルコ語のロケールで"`ı`"と"`i`"が正しく処理されていませんでした。

- 1999-12-31 24:00:00というタイムスタンプが理由もなく拒絶されていたのを修正しました。(Tom Lane)
- 単一のトランザクションが`UNLISTEN`を実行し、その後`LISTEN`を実行した場合のロジックの誤りを修正しました。(Tom Lane)

この場合、セッションは通知イベントを監視すべきですが、実際には全く監視しない状態になっていました。

- 他のビューから参照されているビューに列が追加された後、プランナがクラッシュする可能性があるのを修正しました。(Tom Lane)
- "picksplit doesn't support secondary split"という不必要なログメッセージを削除しました。(Josh Hansen, Tom Lane)

このメッセージは何らかのコードを期待して追加されたように思われますが、そのコードは書かれませんでした。また、GiSTによる二次的分割のデフォルト処理は実際には非常に良いので、そのコードは今後も書かれないでしょう。そのため、エンドユーザにブツブツ言うのをやめることにしました。

- セッションの最後のいくつかのコミットとアボートの件数を統計情報コレクタに送信するのに失敗する可能性があるのを修正しました。(Tom Lane)
- PL/Perlの`spi_prepare()`関数のメモリリークを修正しました。(Alex Hunsaker, Tom Lane)
- `pg_dumpall`でデータベース名に"="が含まれるものを正しく処理できるよう修正しました。(Heikki Linnakangas)
- `pg_dump`が不正な接続文字列が渡された時にクラッシュしないようにしました。(Heikki Linnakangas)
- `pg_dump`と`pg_upgrade`で不正なインデックスを無視するようにしました。(Michael Paquier, Bruce Momjian)

不正なインデックスをダンプするとリストア時に問題が発生する可能性があります。例えば、ユニーク条件を満たさないテーブルのデータに対してユニーク条件を強制したためにインデックス作成に失敗したような場合です。また、インデックスの作成が実際にはまだ進行中の場合もありますが、このときはDDLによる更新がまだコミットされていないと考えるのが適当でしょうから、`pg_dump`がダンプする必要はありません。`pg_upgrade`は不正なインデックスでエラーを起こす代わりに、それをスキップするようになります。

- `pg_basebackup`でテーブル空間をバックアップするとき、現在のサーババージョンのサブディレクトリだけを含めるようにしました。(Heikki Linnakangas)
- `pg_basebackup`および`pg_receivexlog`でサーバのバージョン確認をするようにしました。これにより動作しないバージョンの組み合わせに対して、正しく失敗するようになります。(Heikki Linnakangas)
- `contrib/pg_trgm`の`similarity()`関数が、トライグラムのない文字列について0を返すよう修正しました。(Tom Lane)

以前は内部でのゼロ割りによりNaNを返していました。

- 時間帯データファイルを`tzdata`リリース2013bに更新しました。チリ、ハイチ、モロッコ、パラグアイ、およびロシアのいくつかの地域での夏時間の変更が含まれます。また、数多くの場所での歴史的な地域データの修正が含まれます。

また、時間帯の略称について、ロシアおよび他の地域での以下の最近の変更を反映しました。

`CHOT`、`GET`、`IRKT`、`KGT`、`KRAT`、`MAGT`、`MAWT`、`MSK`、`NOVT`、`OMST`、`TKT`、`VLAT`、`WST`、`YAKT`、`YEKT`については、それらの現在の意味に従います。`VOLT` (Europe/Volgograd (ヨーロッパ/ヴォルゴグラード))および`MIST` (Antarctica/Macquarie (南極/マッコリー島))をデフォルトの短縮名リストに追加しました。

E.7. リリース9.1.8

リリース日: 2013-02-07

このリリースは9.1.7に対し、各種不具合を修正したものです。9.1メジャーリリースにおける新機能については、[項E.15](#)を参照してください。

E.7.1. バージョン9.1.8への移行

9.1.Xからの移行ではダンプ/リストアは不要です。

しかし、9.1.6よりも前のバージョンからアップグレードする場合は、9.1.6のリリースノートを参照してください。

E.7.2. 変更点

- SQLから`enum_recv`を実行できないようにしました。(Tom Lane)

この関数は誤った宣言をされており、単純なSQLコマンドでサーバをクラッシュさせることが可能でした。実際のところ、サーバのメモリの内容を検査するために攻撃者がこれを利用することが可能になっていました。この問題を(Secunia SVCRP経由で)報告してくれたSumit Soniに感謝します。(CVE-2013-0255)

- WALの再生時にデータベースが一貫性のある状態に到達したことを検知するときの複数の問題を修正しました。(Fujii Masao, Heikki Linnakangas, Simon Riggs, Andres Freund)

- リレーションのファイルを切り詰めるときに、最小復旧地点を更新するようにしました。(Heikki Linnakangas)

データが捨てられた後は、タイムライン上のそれより前の地点で復旧を止めることは、もはや安全ではありません。

- 復旧対象のタイムラインを変更した後のWALセグメントの再利用について修正しました。(Heikki Linnakangas)

- ホットスタンバイモードで問い合わせの取り消しに失敗するのを修正しました。(Noah Misch, Simon Riggs)

ホットスタンバイでの競合する問い合わせを取り消す要求が失敗することがあり、これらの問い合わせで一貫性のないデータを見ることがありました。

- 復旧停止機能について、ユーザが接続できるようになる前には停止しないようにしました。(Tom Lane)

- 副SELECTの結果から添字付けあるいは列選択できるようSQLの文法を修正しました。(Tom Lane)

- 負荷が大きい時の自動バキュームの切り詰めの性能問題を修正しました。(Jan Wieck)

テーブルの最後にある空のページを切り詰めるときに排他的ロックが必要ですが、競合するロック要求があるとき、自動バキュームのプログラムはそれに失敗してテーブルロックを解放するようになっていました。負荷が重いと切り詰めが全く起こらず、テーブルが大きくなるということが容易に起こり得ました。部分的な切り詰めを実行、ロックを解放、その後ロックの再取得と処理の継続を試みるように修正しました。この修正により、競合するロック要求がある場合に、自動バキュームがロックを解放するまでの平均時間が大きく短縮されます。

- `pg_tablespace`をスキャンするときに競合問題が発生しないようにしました。(Stephen Frost, Tom Lane)

`pg_tablespace`のエントリに複数の同時更新があるとき、`CREATE DATABASE`および`DROP DATABASE`が正しく動作しない場合があります。

- `DROP OWNED`がデータベース全体あるいはテーブル空間を削除しようとしなかったようにしました。(Álvaro Herrera)

安全のため、これらのオブジェクトは削除するのではなく、所有権を変更すべきです。

- `vacuum_freeze_table_age`の実装の誤りを修正しました。(Andres Freund)

`vacuum_freeze_min_age`の値よりも多くのトランザクションを実行したインストール環境では、この誤りにより自動バキュームが部分テーブルスキャンを実行できなくなり、その結果、必ずテーブル全体のスキャンが発生します。

- `RowExpr`あるいは`XmlExpr`が2度、解析された時に、誤った動作をしないようにしました。(Andres Freund, Tom Lane)

この誤りは`CREATE TABLE LIKE INCLUDING INDEXES`などの状況においてユーザに明らかになることがありました。

- ハッシュテーブルのサイズ計算で整数のオーバーフローが起きないようにする仕組みを改善しました。(Jeff Davis)
- サーバがクラッシュした後、残されていた一時テーブルを無視しないという問題を修正しました。(Tom Lane)
- `to_date()` が範囲外の日付を拒絶するようにしました。(Hitoshi Harada)
- `pg_extension_config_dump()` が拡張のアップデートについて適切に処理するよう修正しました。(Tom Lane)

この関数は、対象のテーブルのすべての既存のエントリを置換するようになりました。これにより、拡張をアップデートするスクリプトから利用可能になります。

- 関数が複数のテーブルのトリガーとして使われる場合について、PL/Pythonの処理を修正しました。(Andres Freund)
- Windowsで非ASCIIのプロンプト文字列が、確実に正しいコードページに変換されるようにしました。(Alexander Law, Noah Misch)

このバグはpsqlおよび他のいくつかのクライアントプログラムに影響がありました。

- データベースに接続していないとき、psqlの`\?`コマンドがクラッシュする可能性があったのを修正しました。(Meng Qingzhong)
- `pg_basebackup`の実行中にリレーションのファイルが削除された時にエラーが起きる可能性があるのを修正しました。(Heikki Linnakangas)
- `pg_dump`をホットスタンバイのサーバで実行するとき、ログを取らないテーブルのデータを除外するようにしました。(Magnus Hagander)

スタンバイのサーバにはログを取らないテーブルのデータがありませんから、これはいずれにせよ失敗します。従って、自動的に`--no-unlogged-table-data`と見なすのが最も良い方法であると思われます。

- `pg_upgrade`が不正なインデックスを安全に扱うよう修正しました。(Bruce Momjian)
- `libpq`の`PQprintTuples`でバッファを1バイト行き過ぎてしまう問題を修正しました。(Xi Wang)

この古い関数はPostgreSQL本体によっては使われていませんが、クライアントのプログラムでは今でも使われているかもしれません。

- `ecpglib`で翻訳されたメッセージを適切に使うようにしました。(Chen Huajun)
- MSVC上で`ecpg_compat`および`pgtypes`のライブラリを適切にインストールするようにしました。(Jiang Guiqing)
- `isinf()` をシステムが提供していなければ、我々のバージョンを`libecpg`に含めるようにしました。(Jiang Guiqing)
- `libedit/libreadline`からの誤ったエクスポートに騙されないよう、`configure`による提供関数の確認の順序を調整しました。(Christoph Berg)
- Windowsのビルド番号が時とともに確実に大きくなるようにしました。(Magnus Hagander)
- Windows用にクロスコンパイルしたときに、`pgxs`が正しく`.exe`の拡張子の付いた実行プログラムを生成するようにしました。(Zoltan Boszormenyi)
- 新しい時間帯の略号`FET`を追加しました。(Tom Lane)

これは一部の東ヨーロッパの時間帯で使われています。

E.8. リリース9.1.7

リリース日: 2012-12-06

このリリースは9.1.6に対し、各種不具合を修正したものです。9.1メジャーリリースにおける新機能については、[項E.15](#)を参照してください。

E.8.1. バージョン9.1.7への移行

9.1.Xからの移行ではダンプ/リストアは不要です。

しかし9.1.6より前のバージョンからアップグレードする場合は、9.1.6のリリースノートを参照してください。

E.8.2. 変更点

- `CREATE INDEX CONCURRENTLY`に関連する複数の不具合を修正しました。(Andres Freund, Tom Lane)

`CREATE INDEX CONCURRENTLY`がインデックスの`pg_index`の行の状態を変更するときに、その場にある更新を使うように修正しました。これにより、同時実行中のセッションが対象のインデックスを更新できず、それにより`CONCURRENTLY`で作成されるインデックスを破損させることになる競合条件を防ぐことができます。

また、`CREATE INDEX CONCURRENTLY`コマンドのエラーの結果として作られる不正なインデックスについて、他の様々な操作が確実に無視するように修正しました。この中で最も重要なのは`VACUUM`です。なぜなら、この不正なインデックスを修正あるいは削除するための修正の処理が実行される前に、自動バキュームが実行されることがあり得るからです。

- WAL再生中のバッファロックを修正しました。(Tom Lane)

複数ページにまたがるWALレコードを再生する場合のバッファロックについて、WAL再生のプログラムに完全でない部分がありました。これにより、ホットスタンバイの問い合わせが一時的に一貫性のない状態になることがあり、その結果、誤った結果や不正なエラーを起こすことができました。

- GINインデックスについてのWAL生成のロジックの誤りを修正しました。(Tom Lane)

分裂ページのエラーが発生すると、これによりインデックスが破損することがありました。

- ホットスタンバイのサーバを通常の実行にプロモートする時、起動プロセスの仮想XIDロックを正しく削除するようにしました。(Simon Riggs)

この誤りにより、`CREATE INDEX CONCURRENTLY`などいくつかの操作をこの後で実行できない場合があります。

- スタンバイモードにおいて偽物の"out-of-sequence timeline ID"エラーが出ないようにしました。(Heikki Linnakangas)

- postmasterがシャットダウンのシグナルを受け取った後は、新しい子プロセスを起動しないようにしました。(Tom Lane)

この誤りにより、シャットダウンが必要以上に長く時間がかかる、あるいはユーザによる追加の操作がなければ終了しない、ということがありました。

- メモリが不足した時でも、内部ハッシュテーブルが破損しないようにしました。(Hitoshi Harada)

- 削除したテーブルのファイルディスクリプタをトランザクションが終了した後もオープンのまま保持し続ける問題を修正しました。(Tom Lane)

これにより、削除してから時間が経っているテーブルがディスクスペースを占め続けるという問題の発生が減るはずで

す。

- 新しい子プロセスがラッチへのパイプを作成できないとき、データベース全体がクラッシュして再起動する問題を修正しました。(Tom Lane)

新しいプロセスはエラーにするしかありませんが、データベース全体を強制的に再起動させるべき理由はありませんから、再起動しないようにしました。これによりカーネルのファイルディスクリプタが足りなくなりそうな場合の堅牢性が向上します。

- 外部結合上の厳格でない等価な句についての計画作成を修正しました。(Tom Lane)

厳格でない構造を他のものと等号で結ぶ句について、プランナが誤った制約を導出する場合があります。例えば、`WHERE COALESCE(foo, 0) = 0`で`foo`が外部結合のNULLになるかもしれない側のもののときです。

- 継承ツリー上でインデックスにより最適化されるMIN/MAXがある場合のSELECT DISTINCTについて修正しました。(Tom Lane)

これらの要素の組み合わせに対して、プランナは"failed to re-find MinMaxAggInfo record"というエラーを起こしていました。

- プランナについて、等価なクラスから排他制約を証明する能力を向上しました。(Tom Lane)
- ハッシュされた副計画の部分行のマッチングについて、型をまたがる場合を正しく処理できるよう修正しました。(Tom Lane)

これは`WHERE (a, b) NOT IN (SELECT x, y FROM ...)`のような複数の列についてのNOT INの副計画で、例えば`b`と`y`がそれぞれ`int4`型と`int8`型である、というような場合に影響します。この処理の誤りにより、そこで使われているデータ型によって、誤った結果が得られたり、あるいはクラッシュする、ということが起きていました。

- AFTER ROW UPDATE/DELETEトリガーで古いタプルを再フェッチする時、バッファロックを取得するようにしました。(Andres Freund)

この誤りにより、非常に特異な状況において、トリガーのWHEN条件、あるいは外部キーを強制するトリガーの事前チェックのロジックに対して誤ったデータを渡すことができました。その結果として、クラッシュを起こしたり、あるいはトリガーを実行するかどうかについて誤った判断をすることがありました。

- ALTER COLUMN TYPEについて、継承されたチェック制約を正しく処理できるように修正しました。(Pavan Deolasee)

これは8.4より前のリリースでは正しく動作していました。この修正により8.4およびそれ以降のリリースでも正しく動作するようになりました。

- ALTER EXTENSION SET SCHEMAについて、いくつかの付随するオブジェクトを新しいスキーマに移動するのに失敗することがあるのを修正しました。(Álvaro Herrera, Dimitri Fontaine)
- REASSIGN OWNEDについて、表領域のGRANTが正しく処理できるように修正しました。(Álvaro Herrera)
- ビューのシステム列についてのpg_attributeの誤ったエントリを無視するようにしました。(Tom Lane)

ビューにはシステム列はありません。しかし、表をビューに変換するとき、そのようなエントリを取り除くことを忘れていました。これは9.3およびそれ以降のリリースでは適切に修正されますが、それ以前のブランチにおいても、誤って変換されたビューが存在しないようにする必要があります。

- ルールの出力について、INSERT INTO table DEFAULT VALUESを正しくダンプできるよう修正しました。(Tom Lane)
- 問い合わせが非常に多くのUNION/INTERSECT/EXCEPTを含むときでもスタックオーバーフローが発生しないようにしました。(Tom Lane)
- もっとも小さな整数値を-1で割り算した時に、プラットフォームによってはエラーとなるのを修正しました。(Xi Wang, Tom Lane)
- 日付文字列の解析で、文字列の終わりを越えてアクセスすることがあるのを修正しました。(Hitoshi Harada)
- チェックポイント中にXID周回が発生し、かつwal_levelがhot_standbyである場合にXIDエポックを進めるのに失敗するのを修正しました。(Tom Lane, Andres Freund)

この誤りはPostgreSQL自体には特に影響はありませんでしたが、TXID値が後戻りしているように見えるため、txid_current()およびそれに関連した関数に依存するアプリケーションにとっては問題でした。

- ページ境界におけるpg_stat_replication.sync_stateの表示を修正しました。(Kyotaro Horiguchi)
- Unixドメインソケットのパス名がプラットフォーム依存の最大長を超えていた場合に、理解可能なエラーメッセージを生成するようにしました。(Tom Lane, Andrew Dunstan)

以前は"Non-recoverable failure in name resolution"(名前解決で解決不能なエラーが発生しました)といった非常に不親切なエラーを起こしていました。

- 複合列の値をクライアントに送るときのメモリリークを修正しました。(Tom Lane)

- `postmaster.pid` ファイルの読み込みについて、`pg_ctl` をより堅牢にしました。(Heikki Linnakangas)

競合条件、およびファイルデスクリプタのリークを解決しました。

- `psql` において、不正にエンコードされたデータが入力され、かつ `client_encoding` の設定が SJIS のようにクライアントでしか利用できない文字セットである場合にクラッシュする場合があるのを修正しました。(Jiang Guiqing)
- `pg_dump` について `SEQUENCE SET` の要素をアーカイブ内のデータ部(データの前ではない)にダンプするようにしました。(Tom Lane)

この変更により、拡張設定テーブルとして印付けられたシーケンスのダンプが修正されます。

- `pg_dump` が `tar` 出力形式で生成する `restore.sql` スクリプトの不具合を修正しました。(Tom Lane)

このスクリプトはテーブル名が大文字を含む場合に全く動作しませんでした。また、このスクリプトで通常の COPY モードだけでなく `--inserts` モードの場合もデータをリストアできるようにしました。

- `pg_restore` で POSIX 準拠の `tar` ファイルを処理できるよう修正しました。(Brian Weaver, Tom Lane)

`pg_dump` の `tar` 出力モードの元々のプログラムは、POSIX 標準に完全には準拠していないファイルを生成していました。この問題はバージョン 9.3 で修正されます。このパッチは 9.3 より前のブランチを修正するもので、正しくないフォーマットと正しいフォーマットの両方を処理できるようにします。これにより、9.3 がリリースされた時に互換性の問題を防ぐことができます。

- `pg_basebackup` が生成する `tar` ファイルが POSIX 準拠になるように修正しました。(Brian Weaver, Tom Lane)

- `pg_resetxlog` について、データディレクトリへの相対パスが与えられた時に、`postmaster.pid` の場所を正しく特定できるよう修正しました。(Tom Lane)

この誤りにより、データディレクトリを使用している動作中の `postmaster` があることに `pg_resetxlog` が気が付かないことがありました。

- `libpq` の `lo_import()` 関数および `lo_export()` 関数が、ファイル I/O のエラーを適切に報告するよう修正しました。(Tom Lane)

- `ecpg` について、ネストした構造体のポインタ変数の処理を修正しました。(Muhammad Usama)

- `ecpg` の `ecpg_get_data` 関数について、配列を正しく処理するよう修正しました。(Michael Meskes)

- `contrib/pageinspect` の B ツリーページ検査関数で、ページの検査中はバッファロックを取得するようになりました。(Tom Lane)

- 拡張について `make install` を実行した時に `extension` というインストールディレクトリが確実に作成されるようにしました。(Cédric Villemain)

以前は、拡張の Makefile で `MODULEDIR` が定義されていると、このステップが実行されませんでした。

- AIX 上でのロードモジュールのビルドについて `pgxs` のサポートを修正しました。(Tom Lane)

元々のソースツリーの外側でモジュールをビルドしても AIX では正しく動作しませんでした。

- 時間帯データファイルを `tzdata` リリース 2012j に更新しました。キューバ、イスラエル、ヨルダン、リビア、パレスチナ、西サモア、およびブラジルの一部での夏時間の変更が含まれます。

E.9. リリース9.1.6

リリース日: 2012-09-24

このリリースは9.1.5に対し、各種不具合を修正したものです。9.1メジャーリリースにおける新機能については[項E.15](#)を参照してください。

E.9.1. バージョン9.1.6への移行

9.1.Xからの移行ではダンプ/リストアは不要です。

しかし、以下の変更履歴の最初に記述されているデータ破損の不具合の影響から復旧させるために、`REINDEX`の操作を実行する必要があるかもしれません。

また9.1.4より前のバージョンからアップグレードする場合は9.1.4のリリースノートを参照してください。

E.9.2. 変更点

- WAL再生時の共有バッファの永続性の印付けを修正しました。(Jeff Davis)

この間違いにより、チェックポイント時にバッファが書き出されなかったことがありました。その結果、その後もそのバッファを書き出すことなくサーバがクラッシュすると、データ破損が発生することになります。データ破損は、クラッシュからの復旧の後にはどのサーバでも発生する可能性があります。特にスタンバイのスレーブサーバではWAL再生を実行する回数がずっと多いですから、この問題が発生する可能性もずっと高くなります。btreeおよびGINインデックスが破損する可能性は低いです。テーブルの"可視性マップ"が破損する可能性はずっと高いです。幸いにして、9.1では可視性マップは重要なデータではなく、9.1のインストール環境でこの破損が引き起こす最悪の結果はバキューム処理が一時的に非効率になることです。テーブルデータの内容がこの不具合によって破損することはありません。

この不具合のためにインデックスが破損したという既知の事例はありませんが、念のため、稼働用のインストレーションでは、9.1.6にアップグレードした後、都合の良い時に、すべてのbtreeおよびGINインデックスについて`REINDEX`を実行することを勧めます。

また9.2.Xへのその場でのアップグレードを実施する予定なら、その前に`vacuum_freeze_table_age`を0に設定して、すべてのテーブルについて`VACUUM`を実行する>ことを勧めます。これにより、可視性マップに残っている誤ったデータについて、9.2.Xがそれを参照するより前に修正されることが確実にになります。`vacuum_cost_delay`を調整して、バキュームがパフォーマンスに与える影響を小さくすることができます。ただし、それによりバキュームが終了するまでの時間は長くなります。

- プランナによるエグゼキュータのパラメータの代入について修正しました。また、CTEプランノードについてエグゼキュータの再スキャンの条件を修正しました。(Tom Lane)

これらの間違いにより、同じ`WITH`の副問い合わせを複数回スキャンする問い合わせについて、間違った結果を返すことがありました。

- `default_transaction_isolation`が`serializable`に設定されているときの誤った動作について修正しました。(Kevin Grittner, Tom Lane, Heikki Linnakangas)

Windowsでのプロセス起動時のクラッシュや、ホットスタンバイの動作でのクラッシュなどの現象が含まれます。

- 接頭辞、つまり`word:*`のパターンを含むテキスト検索の問い合わせについて、選択性の推定を改善しました。(Tom Lane)

- GiSTインデックスにおけるページ分割の決定を改善しました。(Alexander Korotkov, Robert Haas, Tom Lane)

複数列のGiSTインデックスで、この間違いにより、予期しない膨張が起きることがありました。

- 権限が取り消されても、その権限を引き続き保持している場合は、取り消しが連鎖しないようにしました。(Tom Lane)

あるロールxからグラントオプションを取り消したが、xが他の誰かから付与されたことによってそのオプションを引き続き保持している、という場合は、ロールyがxによって付与されたその権限を再帰的に取り消すべきではありません。

- 拡張が、自分が割り当てられたスキーマを含むことを禁止しました。(Thom Brown)

これにより循環的な依存関係が作られ、pg_dumpなどが混乱することがありました。人間もこれによって混乱させられるので禁止しました。

- ホットスタンバイ構成に誤りがあるときのエラーメッセージを改善しました。(Gurjeet Singh)
- configureが`mbstowcs_l`を探すようにしました。(Tom Lane)

これにより、AIXの一部のバージョンでのビルドエラーが解決します。

- PL/Perlが使われている時のSIGFPEの処理について修正しました。(Andres Freund)

PerlはプロセスのSIGFPEハンドラをSIG_IGNにリセットするため、後になってクラッシュが起きる場合があります。PL/Perlを初期化した後で、Postgresの通常のシグナルハンドラをリストアするようにしました。

- PL/Perlの再帰関数が実行時に再定義される場合に、PL/Perlがクラッシュしないようにしました。(Tom Lane)
- PL/Perlで正しく最適化されない場合があるのを回避しました。(Tom Lane)

一部のLinuxディストリビューションでは、誤ったバージョンの`pthread.h`がインストールされており、PL/Perlのコンパイルされたコードが誤ったものになります。この結果、PL/Perlの関数がエラーを発生させる他の関数を呼び出すと、クラッシュすることがありました。

- contrib/pg_trgmのLIKEパターンの分析のプログラムの不具合を修正しました。(Fujii Masao)

トリグラムインデックスを使ったLIKEの問い合わせで、パターンがLIKEのエスケープ文字を含む場合に誤った結果が生成されることがありました。

- pg_upgradeのWindowsにおける行末の処理を修正しました。(Andrew Dunstan)

以前は、pg_upgradeは関数の実体などの場所で、復帰文字(CR)を追加あるいは削除することがありました。

- Windows上でpg_upgradeが生成するスクリプトで、パス名の区切り文字にバックスラッシュを使うようにしました。(Andrew Dunstan)
- pg_upgradeのpg_configへの依存は、不要なので取り除きました。(Peter Eisentraut)
- 時間帯データファイルをtzdataリリース2012fに更新しました。フィジーでの夏時間の変更が含まれます。

E.10. リリース9.1.5

リリース日: 2012-08-17

このリリースは9.1.4に対し、各種不具合を修正したものです。9.1メジャーリリースにおける新機能については[項E.15](#)を参照してください。

E.10.1. バージョン9.1.5への移行

9.1.Xからの移行ではダンプ/リストアは不要です。

しかし9.1.4より前のバージョンからアップグレードする場合は9.1.4のリリースノートを参照してください。

E.10.2. 変更点

- XML実体参照を介した外部ファイル/URLへのアクセスを防止します。(Noah Misch, Tom Lane)

`xml_parse()` は、XML値で参照されるDTDと実体参照を解決するために必要な外部ファイルやURLの入手を試みます。これにより、権限を持たないデータベースユーザがデータベースサーバの権限でデータを入手することができてしまいました。この外部データは直接ユーザには返されませんが、データが有効なXMLとして解析できなかった場合その一部がエラーメッセージの中に漏洩することがあり得ます。またどのような場合であっても、ファイルが存在するかどうか検査できるだけでも攻撃者にとって有用になり得ます。(CVE-2012-3489)

- `contrib/xml2`の`xsslt_process()`を介した外部ファイル/URLへのアクセスを防止します。(Peter Eisentraut)

`libxslt`はスタイルシートコマンドを通してファイルとURLの両方を読み書きする機能を提供します。これにより、権限を持たないデータベースユーザがデータベースサーバの権限でデータを読み書きすることができてしまいました。`libxslt`のセキュリティオプションを適切に使用することでこれを無効にします。(CVE-2012-3488)

また`xsslt_process()`の外部ファイル/URLからドキュメントとスタイルシートを取り込む機能を取り除きました。これは文書化された"機能"でしたが、推奨されないものと長くみなされていました。CVE-2012-3489に対する修正のためこの能力が壊れましたが、その修正を行うことは止め、単に取り除きました。

- 早すぎるbtreeインデックスページの回収を防止します。(Noah Misch)

読み取り専用トランザクションによるXIDの割り当てを省略できるようにした時、読み取り専用トランザクションがまだ参照している時に削除されたbtreeページを回収されてしまう可能性をもたらしました。これは間違ったインデックス検索結果という結果になります。時間に関する必要条件のため、こうしたエラーが実際に発生する可能性は非常に低いものでしたが、修正しなければなりません。

- 新しく作成された、またはリセットされたシーケンスに関するクラッシュに対する安全性を修正しました。(Tom Lane)

`ALTER SEQUENCE`が新しく作成されたシーケンスまたはリセットされたシーケンスに対して実行された場合、その後`nextval()`呼び出しが一度だけ行われ、そして、サーバがクラッシュすると、WAL再生はシーケンスを`nextval()`が行われていないように見える状態にリストアします。このため次の`nextval()`呼び出しにより最初のシーケンス値が再び返されることになります。シリアル列のシーケンス作成には`ALTER SEQUENCE OWNED BY`処理が含まれていますので、特にこれは`serial`列で現れます。

- `enum`列値の比較における競合条件を修正しました。(Robert Haas, Tom Lane)

現在の問い合わせが始まってから追加された列挙値と合う場合に比較が失敗することがありました。

- ホットスタンバイではない時に正確なエポックを報告するように`txid_current()`を修正しました。(Heikki Linnakangas)

これは以前のマイナーリリースで導入されたリグレッションを修正します。

- 同期スタンバイとして不適切なレプリケーション接続を選択しないようにします。(Fujii Masao)

マスタが同期スタンバイとして`pg_receivexlog`や`pg_basebackup`などの仮想サーバを不適切に選択し、無限に待機してしまう可能性があります。

- マスタのトランザクションが多くの副トランザクションを持つ時のホットスタンバイの起動処理の不具合を修正しまし

た。(Andres Freund)

この間違いは"out-of-order XID insertion in KnownAssignedXids"と報告される失敗をもたらします。

- `pg_start_backup()`の後確実に`backup_label`ファイルを`fsync`します。(Dave Kerr)
- WAL送信処理プロセスにおけるタイムアウト処理を修正しました。(Tom Lane)

WAL送信処理バックグラウンドプロセスでSIGALRMハンドラの設定を忘れていました。この結果タイムアウトが発生する何らかの境界条件で永遠に待機してしまいます。

- WAL書き込み処理によるそれぞれのバックグラウンド吐き出しの後にWAL送信処理を再開します。(Andres Freund、Simon Riggs)

これは、作業が非同期にコミットされるトランザクションのみを含む時のレプリケーション遅延を大きく減少します。

- ディスク容量不足等I/O問題をよりよく処理するように`LISTEN/NOTIFY`を修正しました。(Tom Lane)

書き込みの失敗の後、以降すべての`NOTIFY`メッセージの送信試行は"Could not read from file "pg_notify/nnnn" at offset nnnnn: Success"のようなメッセージで失敗しました。

- 自動バキュームは直接ブロックされたプロセスによってのみ自動キャンセルさせることができます。

元々のコードでは一部の場合に一貫性がない動作を許していました。具体的には`deadlock_timeout`猶予期間より短時間で自動バキュームはキャンセルされました。

- 自動バキュームのキャンセルについてのログ処理を改良しました。(Robert Haas)
- サーバ起動後一番最初のログローテーションの間`log_truncate_on_rotation`が動作するようにログコレクタを修正しました。(Tom Lane)

- 入れ子状の集合操作 (`UNION/INTERSECT/EXCEPT`) に付与される`WITH`を修正しました。(Tom Lane)
- 副問い合わせへの行全体の参照が余計な`GROUP BY`や`ORDER BY`列を含まないことを確実にしました。(Tom Lane)
- `ALTER TABLE ... ADD CONSTRAINT USING INDEX`時に生成される依存関係を修正しました。(Tom Lane)

このコマンドはインデックス用の冗長な`pg_depend`項目を残しました。これは後の操作、特にインデックス対象の列のいずれかに対する`ALTER TABLE ... ALTER COLUMN TYPE`を混乱させるかもしれませんでした。

- `REASSIGN OWNED`の拡張に対する動作を修正しました。(Alvaro Herrera)
- `CREATE TABLE`時の`CHECK`制約およびインデックス定義における行全体の参照の複製を許しません。(Tom Lane)

この状況は`LIKE`または`INHERITS`を持つ`CREATE TABLE`で発生することがあり得ます。複製される行全体の変数は、新しいテーブルではなく元のテーブルの行型で正しくラベル付けされませんでした。`LIKE`では、後で行型がよく分岐しますので、このような状況を拒絶する方が合理的のように思います。`INHERITS`では理想的には、親のテーブルの行型への暗黙的な変換によって許可すべきですが、安全に以前のバージョンにパッチを充てるにはより多くの作業が必要です。

- `ARRAY(SELECT ...)`副問い合わせにおけるメモリリークを修正しました。(Heikki Linnakangas、Tom Lane)
- 演算子の選択性推定処理に正しい照合順序を渡すようにプランナを修正しました。(Tom Lane)

これまではコアの選択性推定関数では必要とされていませんでしたが、サードパーティ製のコードで必要とするかもしれません。

- 正規表現からの共通接頭辞の抽出を修正しました。(Tom Lane)

<!-- The code could get confused by quantified parenthesized subexpressions, such as `^(foo)?bar`. This would lead to incorrect index optimization of searches for such patterns. --> このコードでは`^(foo)?bar`などの修飾された括弧付きの副式で混乱してすることがあり得ました。こうしたパターンの検索に対して不正確なインデックス最適化をもたらしました。

- `interval`定数における、符号付き`hh:mm`、`hh:mm:ss`フィールドの解析に関する不具合を修正しました。(Amit Kapila、Tom Lane)
- 部分的な`GROUP BY`リストを持つビューをより良く扱うように`pg_dump`を修正しました。(Tom Lane)

`GROUP BY`内にプライマリキー列のみが列挙されているが、あたかもグループ化されているかのように他のテーブル列を使用するビューは、プライマリキーに依存するものと印付けられます。`pg_dump`におけるこうしたプライマリキーへの依

存に関する不適切な扱いの結果、順序の並びが悪い、良くてもリストア効率が悪い、最悪並行pg_restore実行が完全に失敗する、ダンプになりました。

- PL/Perlにおいて、SQL_ASCII符号化方式の時にUTF8フラグの設定を防ぎます。(Alex Hunsaker、Kyotaro Horiguchi、Alvaro Herrera)
- PL/Pythonにおいて、PythonのUnicode文字列からサーバ符号化方式に変換する時にPythonの関数ではなくPostgresの符号化方式変換関数を使用します。(Jan Urbanski)

これは、特にPythonがPostgresがサポートするすべての符号化方式をサポートしていない場合の何らかの境界値問題を防止します。 注意すべき機能上の変更は、サーバ符号化方式がSQL_ASCIIの場合、文字列のUTF-8表現を得ることです。 これまでは文字列内のASCII以外の文字はエラーになりました。

- PL/PythonにおいてPostgreSQL符号化方式とPython符号化方式との間の対応付けを修正しました。(Jan Urbanski)
- contrib/xml2の`xslt_process()`においてエラーを適切に報告します。(Tom Lane)
- 時間帯データファイルをtzdataリリース2012eに更新しました。 モロッコおよびトケラウにおける夏時間の変更が含まれます。

[前のページ](#)
リリース9.1.6

[ホーム](#)
[上に戻る](#)

[次のページ](#)
リリース9.1.4

E.11. リリース9.1.4

リリース日: 2012-06-04

このリリースは9.1.3に対し、各種不具合を修正したものです。9.1メジャーリリースにおける新機能については[項E.15](#)を参照してください。

E.11.1. バージョン9.1.4への移行

9.1.Xからの移行ではダンプ/リストアは不要です。

しかし、`citext` データ型を使用し、かつ、`pg_upgrade`を実行して過去のメジャーリリースからアップグレードした場合、`CREATE EXTENSION citext FROM unpackaged`を実行して`citext`操作における照合順序に関連した失敗を防止しなければなりません。`citext` データ型のインスタンスを含む、9.1より前のダンプからリストアする場合も、同じことが必要です。もし9.1.4にアップグレードする前にすでに`CREATE EXTENSION`を実行していた場合は、代わりに変更点の3番目で説明する通りに手作業でカタログを更新しなければなりません。

また9.1.2より前のバージョンからアップグレードする場合は9.1.2のリリースノートを参照してください。

E.11.2. 変更点

- `contrib/pgcrypto`の`DES_crypt()`関数における不正なパスワード変換を修正しました。(Solar Designer)

パスワード文字列に`0x80`というバイト値が含まれる場合、パスワードの残りが無視され、見かけよりも非常に弱いパスワードになってしまいました。この修正によって、文字列の残りが適切にDESハッシュ内に含まれるようになります。したがって、この不具合の影響を受ける、格納済みのパスワード値はすべて一致なくなりますので、格納済みの値を更新しなければなりません。(CVE-2012-2143)

- 手続き言語の呼び出しハンドラにおいて`SECURITY DEFINER`および`SET`属性を無視します。(Tom Lane)

こうした属性を呼び出しハンドラに与えるとサーバがクラッシュするかもしれません。(CVE-2012-2655)

- `contrib/citext`の更新スクリプトが`citext`配列とそのドメインの照合順序を修正するようにしました。(Tom Lane)

9.1.2リリースで、9.1より前のインストレーションからアップグレードまたはリロードしたデータベースにおける`citext`列とインデックスの照合順序の修正を行いましたが、その修正が完全ではありませんでした。`citext`の配列とドメインの取り扱いを無視していました。このリリースではこのモジュールの更新スクリプトを拡張し、これらの場合も取り扱うようにしました。事前に更新スクリプトを実行していた場合は、代わりに手作業で照合順序更新コマンドを実行しなければなりません。この作業に関する詳細な情報については9.1.2のリリースノートを参照してください。

- 数字による時間帯オフセットによりUTCから16時間まで離れた`timestamp`の入力を行うことができます。(Tom Lane)

一部の歴史的な時間帯はこれまでの上限であった15時間よりも離れたオフセットを持ちます。このためリロード時にダンプされたデータ値が拒絶される結果になることがありました。

- 与えられた時刻が現在の時間帯の最後のDST遷移時刻と正確に一致する場合をうまく処理するようにタイムスタンプ変換を修正しました。(Tom Lane)

長い間これは見落とされていました。ほとんどのDSTを使用する時間帯では、将来のDST変更が無限に続くものと仮定していましたので、これまで気付きませんでした。

- `text`から`name`へのキャストおよび`char`から`name`へのキャストがマルチバイト符号化方式で正しく文字列の切り詰めを行うように修正しました。(Karl Schnaitter)
- `to_tsquery()`のメモリコピーにおける不具合を修正しました。(Heikki Linnakangas)
- `txid_current()`がホットスタンバイにおいて実行された場合に、正確なエポックを確実に報告します。(Simon Riggs)
- 副問い合わせ内の外部プレースホルダに関するプランナの扱いを修正しました。(Tom Lane)

この不具合は、上位の問い合わせの外部結合の内NULLを含む方から由来する変数を参照する副SELECTに影響します。9.1では、この不具合によって影響を受ける問い合わせは"ERROR: Upper-level PlaceholderVar found where not expected"というエラーで失敗します。しかし9.0と8.4では、副問い合わせに渡る値がNULLであるべき時にNULLにな

りませんので、警告なく間違っているかもしれない回答が得られます。

- 単純な変数以外の出力列を持つ `UNION ALL` 副問い合わせの計画作成を修正しました。(Tom Lane)

9.1において、"MergeAppend child's targetlist doesn't match MergeAppend"エラーに対する的外れな修正の結果、こうした状況での計画作成が、著しく悪化しました。この修正を戻し、他の方法を取りました。

- `pg_attribute` が非常に大きい場合の低速なセッション起動を修正しました。(Tom Lane)

`pg_attribute` が `shared_buffers` の1/4を超える場合、セッション起動時に時々必要となるキャッシュ再構築コードが同期スキャンロジックを開始し、通常より長い期間かかるようになりました。この問題は、特に多くの新しいセッションが一度に開始する場合に深刻です。

- 適切な頻度で発生する問い合わせキャンセルに対する連続的なスキャン検査を確実に行います。(Merlin Moncurie)

有効なタプルを含まない連続するページを多くを巡るスキャンは、その間、中断に応答しません。

- Windowsの `PGSemaphoreLock()` の実装は、戻る前に `ImmediateInterruptOK` を確実に消去します。(Tom Lane)

この見落としは、同じ問い合わせの中で後で受け取った問い合わせキャンセル中断が、安全ではない時に受け付けられ、予期しない、悪化した結果をもたらす可能性があったことを意味します。

- ビューまたはルールを出力する時に行全体の変数を安全に表示します。(Abbas Butt, Tom Lane)

曖昧な名前を含む（つまりその名前がテーブル名か問い合わせの列名かいずれかになる）境界条件では、曖昧な方法で出力し、ビューまたはルールがダンプとリストアの間で別の解釈がされる危険性がありました。操作を行わないキャストを付与することで曖昧な状況を防ぎます。

- 無効な符号化方式に対応するヌルマーカ文字列を正しく扱うように `COPY FROM` を修正しました。(Tom Lane)

`E'¥¥0'` などのヌルマーカ文字列は動作しなければなりませんし、以前は動作していました。この状況は8.4で壊れました。

- `RETURNING` 句を含む書き出し可能なCTEに対する `EXPLAIN VERBOSE` を修正しました。(Tom Lane)

- 勧告的ロックが存在する場合に正しく動くように `PREPARE TRANSACTION` を修正しました。(Tom Lane)

歴史的に `PREPARE TRANSACTION` は、セッションが保持するセッションレベルの勧告的ロックを単に無視していました。しかしこの状態は9.1で意図せずに壊れました。

- ログを取らないテーブルの切り詰めを修正しました。(Robert Haas)

- 対話式以外の `search_path` への代入において存在しないスキーマを無視します。(Tom Lane)

これは9.1の動作を過去のブランチに再度合わせました。これまでの9.1では `ALTER DATABASE SET` などの場所から得られた `search_path` 設定で指定される、存在しないスキーマに対してエラーとしました。

- 拡張スクリプト内で使用される一時テーブルまたは短期間のテーブルに関する不具合を修正しました。(Tom Lane)

これには、表面には現れない短期間のテーブルを使用しますので、拡張更新スクリプトの中の `ALTER TABLE` の書き換えなども含まれます。

- 自動バキュームのワーカプロセスは確実にスタック長の検査を適切に行います。(Heikki Linnakangas)

これまでは、自動 `ANALYZE` によって呼び出される関数の中の無限再帰がワーカプロセスをクラッシュさせる可能性がありました。

- 高負荷時にログの干渉性が失われないようにログコレクタを修正しました。(Andrew Dunstan)

これまでは、コレクタは非常に忙しい場合、大規模メッセージの再構成に失敗する可能性がありました。

- `SIGHUP` を受信した後にファイルのローテーションを確実に再開するようにログコレクタを修正しました。(Tom Lane)

- GiSTインデックスにおける "too many LWLocks taken" 失敗を修正しました。(Heikki Linnakangas)

- GINインデックスに対するWAL再生ロジックを修正し、インデックスがその後に削除された場合でも失敗しないようにしました。(Tom Lane)

- クラッシュ後のプリペアドトランザクションのSSI競合を正しく検出します。(Dan Ports)

- 一時テーブルのみを変更したトランザクションがコミットされた時の同期レプリケーションの遅延を防止します。(Heikki Linnakangas)
 こうした場合、トランザクションのコミットレコードはスタンバイサーバに吐き出す必要はありません。しかし一部のコードでこれが分からずに、とにかく発生を待機していました。
- pg_basebackupのエラー処理を修正しました。(Thomas Ogrisegg、Fujii Masao)
- 接続が終了した場合にビジーープに入らないようにwalsenderを修正しました。(Fujii Masao)
- PL/pgSQLのRETURN NEXT コマンドにおけるメモリリークを修正しました。(Joe Conway)
- 対象が関数の第一変数であった場合のPL/pgSQLのGET DIAGNOSTICS コマンドを修正しました。(Tom Lane)
- PL/Perlは確実にTD変数をパッケージで修飾します。(Alex Hunsaker)
 この不具合により、現在のパッケージを変更する関数呼び出しで入れ子状になった時に、トリガの呼び出しに失敗しました。
- 文字列としてその結果値を受け付ける複合型を返すPL/Python関数を修正しました。(Jan Urbanski)
 この状態は、複合型の結果値を辞書など他の書式に与えることができるという9.1の追加により意図せず壊れました。
- psqlの拡張表示 (¥x) モードにおいてメモリの最後にアクセスできない可能性を修正しました。(Peter Eisentraut)
- データベースに多くのオブジェクトが含まれている場合のpg_dumpの複数の性能問題を修正しました。(Jeff Janes、Tom Lane)
 データベースに多くのスキーマが含まれる場合、多くのオブジェクトが依存関係のあるループにある場合、所有されたシーケンスが多く存在する場合に、pg_dumpが非常に低速になる可能性があります。
- pg_restoreにおける、ディレクトリ形式のアーカイブを読み取る際のメモリリークとファイル記述子のリークを修正しました。
- デフォルト以外のテーブル空間内に格納されたデータベースがクラスタのデフォルトのテーブル空間内のテーブルを含む状況に対するpg_upgradeを修正しました。(Bruce Momjian)
- ecpgにおいて、まれに起こるメモリリークとsqlca_t構造体の後の1バイトの上書きを修正しました。(Peter Eisentraut)
- エラー時に一時データベース接続をリークしないように、contrib/dblinkのdblink_exec() を修正しました。(Tom Lane)
- エラーメッセージ内で正しい接続名を報告するようにcontrib/dblinkを修正しました。(Kyotaro Horiguchi)
- 多くのラージオブジェクトを削除する時に複数のトランザクションを使用するようにcontrib/vacuumloを修正しました。(Tim Lewis、Robert Haas、Tom Lane)

この変更により、多くのオブジェクトを削除しなければならない時にmax_locks_per_transactionを超えることを防ぎます。この動作は新しい-1 (制限) オプションで調整することができます。

- 時間帯データファイルをtzdataリリース2012cに更新しました。南極、アルメリア、チリ、キューバ、フォークランド諸島、ガザ、ハイチ、ヘブロン、モロッコ、シリア、トケラウ諸島における夏時間の変更、カナダに関する歴史的な変更が含まれます。

E.12. リリース9.1.3

リリース日: 2012-02-27

このリリースは9.1.2に対し、各種不具合を修正したものです。9.1メジャーリリースにおける新機能については[項E.15](#)を参照してください。

E.12.1. バージョン9.1.3への移行

9.1.Xからの移行ではダンプ/リストアは不要です。

しかし、9.1.2より前のバージョンからアップグレードする場合は9.1.2のリリースノートを参照してください。

E.12.2. 変更点

- `CREATE TRIGGER`においてトリガ関数に対する実行権限が必要になりました。(Robert Haas)

この検査が無かったため、別のユーザが自身が所有するテーブルにその関数をインストールすることで、偽造した入力データでトリガ関数を実行することができました。`SECURITY DEFINER`が付いたトリガ関数でのみ重大です。その他のトリガ関数ではとにかくテーブル所有者として実行されるためです。(CVE-2012-0866)

- SSL証明書内のコモンネームにおける独断的な長さ制限を取り除きました。(Heikki Linnakangas)

`libpq`とサーバの両方ともSSL証明書から取り出したコモンネームを32バイトで切り詰めていました。通常想定外の検証エラー以上のことを引き起こしますが、ある証明書の所有者が他者になりすますことができるという、多少怪しいシナリオがあります。被害を受けるにはコモンネームが正確に32バイト長でなければならない、また、攻撃者は信頼されたCAに対してコモンネームが接頭辞としてその文字列を持つ証明書を発行するように説得しなければなりません。またサーバになりすますには、さらにクライアントからの接続を中継するための攻撃が必要です。(CVE-2012-0867)

- `pg_dump`のコメントに記述された名前の中の改行を空白に変換します。(Robert Haas)

`pg_dump`は、その出力スクリプトにおけるSQLコメント内で発行されるオブジェクト名のサニタイズに関して注意を払っていませんでした。改行を含む名前は少なくともそのスクリプトを構文的に不正にさせます。悪意をもって組み立てられたオブジェクト名によって、スクリプトがリロードする時にSQLインジェクションの危険性があり得ました。(CVE-2012-0868)

- バキューム処理と同時に挿入を行った場合の**btree**インデックス破損を修正しました。(Tom Lane)

挿入によって発生するインデックスページ分割によって、同時実行中の**VACUUM**が削除すべきインデックス項目を削除し損なうことが時々発生することがありました。対応するテーブル行が削除された後、対応先が無いインデックス項目によってエラー ("could not read block N in file ..."など)、最悪は、解放されたテーブル位置に再挿入された関係がない行にちなんだ、警告のない間違った問い合わせ結果が引き起こされます。この不具合は8.2から存在していましたが、あまり頻発に発生しませんでしたので、これまで究明されませんでした。使用中のデータベースで発生していたのではと疑わしければ、対象のインデックスを再インデックス付けすることで修正されます。

- WAL再生中の一時的な共有バッファのゼロ化を修正しました。(Tom Lane)

再生ロジックでは時々共有バッファをゼロ化し再度書き込んでいました。このためその内容は一時的に無効になります。ホットスタンバイモードではこの結果、平行して実行される問い合わせがゴミデータを参照することになり得ました。いろいろな兆候が現れますが、もっとも良くある兆候は"invalid memory alloc request size"です。

- `READ COMMITTED`再検査において、データを変更する**WITH**副計画の取り扱いを修正しました。(Tom Lane)

`INSERT/UPDATE/DELETE`を含む**WITH**は、`READ COMMITTED`モードにおける同時更新のために、親の**UPDATE**または**DELETE**コマンドが1つ以上の行を再評価しなければならない場合に、クラッシュしました。

- SSIトランザクションの整理における境界問題を修正しました。(Dan Ports)

読み書きのあるシリアライズブルトランザクションを完了させるときに、残存する有効なシリアライズブルトランザクションがすべて読み取り専用であった場合にクラッシュすることがありました。

- ホットスタンバイのクラッシュの後、`postmaster`の再起動を試みるように修正しました。(Tom Lane)

ホットスタンバイモードで操作している時に任意のバックエンドプロセスがクラッシュした場合、ロジックエラーのため、クラスタを再起動させようとはせずにpostmasterを終了させました。

- 最近更新された行によって所有されるTOAST値に対する`CLUSTER/VACUUM FULL`の取り扱いを修正しました。(Tom Lane)
この見落としにより、これらのコマンドのいずれかを実行している時に、TOASTテーブルのインデックスに対して"duplicate key value violates unique constraint"エラーが報告されることがあり得ました。
- テーブル所有者を変更する時に、テーブル単位の権限だけではなく列単位の権限も更新します。(Tom Lane)
この失敗は、事前に付与された列権限が古い所有者により付与されたものとして表示されることを意味します。つまり新しい所有者またはスーパーユーザであってもテーブル所有者を追跡できなくなった権限を取り除くことができないことを意味します。
- `REASSIGN OWNED`において外部データラップおよび外部サーバをサポートします。(Alvaro Herrera)
これらのオブジェクトの所有権を変更しなければならなかった場合に、このコマンドは"unexpected classid"エラーで失敗しました。
- `ALTER USER/DATABASE SET`における一部の設定で存在しない値を許可します。(Heikki Linakangas)
`default_text_search_config`、`default_tablespace`、`temp_tablespaces`を未知の名前に設定することができます。これらが実際に使用されている別のデータベースでは既知であるかもしれませんが、またテーブル空間の場合はまだテーブル空間が作成されていないかもしれないからです。過去`search_path`においても同じ問題がありましたが、これらの設定も同様に動作するようにしました。
- `INSERT`式内の`COLLATE`により引き起こる"unsupported node type"エラーを修正しました。(Tom Lane)
- コミット後のテーブルファイルの削除に問題があった場合のクラッシュを防止します。(Tom Lane)
テーブル削除は、そのトランザクションがコミットした後にのみ背後のディスク上のファイルを削除しなければなりません。(例えばファイルの権限の誤設定のため)失敗した場合、トランザクションをアボートするには遅すぎますので、コードは単に警告メッセージを出力し継続するものと考えられます。リリース8.4においてこのロジックが壊れ、こうした状況でPANICが発生し、再起動できないデータベースとなりました。
- `DROP TABLESPACE`のWAL再生中に発生したエラーから修復します。(Tom Lane)
再生はテーブル空間ディレクトリの削除を試みますが、(対象ディレクトリの所有者や権限が不正であったなど)失敗する可能性は多々あります。これまでは再生コードはパニックし、手作業を介さなければ再起動できないデータベースになりました。ディレクトリの削除に失敗したところで多少ディスク領域が無駄に使われるだけですので、問題を記録し、継続する方がより良いと考えました。
- ホットスタンバイのAccessExclusiveLocksのログ記録における競合問題を修正しました。(Simon Riggs)
ロックが"transaction zero"によって保持されていると記録されることが時々ありました。少なくともスレーブサーバでアサーションエラーが発生することが分かっていますが、もっと深刻な問題が起こる可能性があります。
- WAL再生中にOIDカウンタを、たとえ周回していたとしても、正しく追跡します。(Tom Lane)
これまでは、OIDカウンタはシステムが再生モードを抜けるまで高値のままになりました。実際にはほとんどnilという結果になりますが、マスタに昇格されるスタンバイサーバでは、値が必要になってから、OIDカウンタを合理的な値まで進めるために長時間かかる可能性があります。
- クラッシュリカバリの最初の、誤解を招く"consistent recovery state reached"ログメッセージの発行を抑制します。(Heikki Linnakangas)
- `pg_stat_replication.replay_location`の初期値を修正しました。(Fujii Masao)
これまでは、少なくとも1つのWALレコードが再生されるまで、表示される値は間違っていました。
- *が付いた正規表現の後方参照を修正しました。(Tom Lane)
コードでは、正確な文字列一致を強制せずに、実質後方参照シンボルで参照されるパターン副式を満たす任意の文字列を受け付けました。
同様の問題はまだ、直接量指定子のサブジェクトとならない、より大きく量化された式に埋め込まれた後方参照でも残っています。こちらは将来のリリースのPostgreSQLで対応予定です。
- `inet/cidr`値の処理に最近入ったメモリリークを修正しました。(Heikki Linnakangas)

2011年12月のPostgreSQLリリース内のパッチによって、これらの操作にメモリリークが発生しました。これらの列に対するbtreeインデックス等で重大な問題になる可能性があります。

- `UNION ALL`を通してインデックス式制限をプッシュダウンするプランナの機能を修正しました。(Tom Lane)

9.1.2において別の問題を修正する際に、この種の最適化が意図せず無効になりました。

- 継承テーブルに対する`UPDATE/DELETE`において参照される`WITH`句の計画作成を修正しました。(Tom Lane)

この不具合により"could not find plan for CTE"失敗が発生しました。

- `column IN (...)` インデックス条件を扱うためのGINコスト推定を修正しました。(Marti Raudsepp)

この見落としにより、こうした条件がGINインデックスを用いて使用される場合によくクラッシュするようになりました。

- 既存のセッションが開いたままの失敗トランザクションを持つ場合のアサーション失敗を防止します。

この不具合は、アサートが有効ではない通常の構築では影響はありません。

- SQL言語関数内の`CREATE TABLE AS/SELECT INTO`の後に対応先がなくなったポインタを修正しました。(Tom Lane)

ほとんどの場合、これはアサートが有効な構築におけるアサーション失敗という結果になるだけですが、もっと悪い結果になる可能性があります。

- Windowsのsysloggerにおけるファイルハンドルの二重クローズを防ぎます。(MauMau)

通常はこのエラーは表面化しませんが、Windowsのデバッグ版を実行している場合は例外が引き起こります。

- `plpgsql`におけるI/O変換関連のメモリリークを修正しました。(Andres Freund、Jan Urbanski、Tom Lane)

現在の関数が終わるまで、特定の操作がメモリリークしました。

- `perl`の`SvPVutf8()`関数の不具合を回避しました。(Andrew Dunstan)

この関数は`typeglob`または特定の読み取り専用のオブジェクトが渡された場合にクラッシュします。 `plperl`でこれらを渡さないようにしました。

- `pg_dump`において、拡張自体がダンプ対象でない場合に拡張の設定テーブルの内容をダンプしません。(Tom Lane)

- 継承されたテーブル列に対する`pg_dump`の取り扱いを改良しました。(Tom Lane)

`pg_dump`は、子の列が親の列と異なるデフォルト式を持つという状況を間違って扱いました。デフォルトがテキストとして親のデフォルトと同一であるが、実際は同一ではない場合（例えば、スキーマ検索パスの違いのため）、異なるものとして認識せず、そのため、ダンプしリストアした後、子は親のデフォルトを継承することができました。子の列が`NOT NULL`であり親がそうではなかった場合も、微妙に間違っリストアされました。

- `INSERT`形式のテーブルデータに対する`pg_restore`の直接データベースにリストアするモードを修正しました。(Tom Lane)

他の問題に対する修正における見落としの結果、2011年9月または12月付けのリリースに含まれる`pg_restore`では、`--inserts`または`--column-inserts` オプションを付けて作成されたアーカイブファイルを直接データベースにリストアすることができませんでした。アーカイブファイル自体には失敗はなく、テキストモード出力では問題ありませんでした。

- `plpython`の共有ライブラリの名称変更を扱うよう`pg_upgrade`を変更しました。(Bruce Momjian)

この見落としのため、`plpython`を含む9.1より前のデータベースをアップグレードすることができませんでした。

- `pg_upgrade`が`regclass`列を含むテーブルを処理できるようにしました。(Bruce Momjian)

`pg_upgrade`が`pg_class` OIDの保護に注意を払うようになりましたので、この制限を続ける理由はなくなりました。

- SSLクライアント証明書ファイルを検索するときに、`libpq`が`ENOTDIR`エラーを無視するようにしました。(Magnus Hagander)

これによりユーザのホームディレクトリが`/dev/null`のようなものに設定されていたとしても、証明書なしですが、SSL接続を確立できます。

- `ecpg`のSQLDA領域におけるフィールド整列問題を修正しました。(Zoltan Boszormenyi)

- `ecpg`の`DEALLOCATE`文において`AT`オプションを可能にしました。(Michael Meskes)
これをサポートする基盤は少し前から存在しましたが、見落としのため、エラー検査でこの状況を拒絶していました。
- `ecpg`において`varchar`構造体を定義するときに変数名を使用しません。(Michael Meskes)
- `contrib/auto_explain`のJSON出力モードが有効なJSONを生成するように修正しました。(Andrew Dunstan)
この出力では、本来括弧を使用しなければならない最上位で、角括弧を使用していました。
- `contrib/intarray`の`int[] & int[]`演算子のエラーを修正しました。(Guillaume Lelarge)
2つの入力配列が共通して持つ最小の整数が1であり、どちらかの配列により小さな値があった場合、1が結果から間違っ
て省かれました。
- `contrib/pgcrypto`の`encrypt_iv()` および `decrypt_iv()` の誤判定を修正しました。(Marko Kreen)
これらの関数は特定の種類の無効入力エラーの通知に失敗し、不正な入力に対してランダムなゴミの値を代わりに返しま
した。
- `contrib/test_parser`における1バイトのバッファオーバーランを修正しました。(Paul Guyot)
コードでは必要より1バイト多く読み取ろうとし、境界条件ではクラッシュします。 `contrib/test_parser`は単なるサン
プルコードであり、これ自体はセキュリティ問題ではありませんが、例のコードとしては良くありません。
- ARMにおいて可能ならばスピロックに `__sync_lock_test_and_set()` を使用します。(Martin Pitt)
この関数は、以前の廃止予定でARMv6以降では使用できなくなった`SWPB`命令の使用を置き換えるものです。最近
のARMボードでも古いコードは明白な方法で失敗することはありませんが、単に同時アクセスのインターロックを行わ
ず、マルチプロセス操作において奇妙な失敗をもたらすと報告されています。
- 受け付け可能なバージョンの`gcc`を用いて構築する場合に`-fexcess-precision=standard`オプションを使用しま
す。(Andrew Dunstan)
最近のバージョンの`gcc`は独創的な結果を生成するというさまざまなシナリオを防ぎます。
- FreeBSDにおいてスレッド化されたPythonを使用できるようにしました。(Chris Rees)
`configure`スクリプトはこれまで、この組み合わせは動作しないと前提していました。しかしFreeBSDではこの問題が修
正されたので、エラー検査を取り除きました。
- MinGWが標準的な名前のOpenSSLライブラリを使用して構築できるようにしました。(Tomasz Ostrowski)

E.13. リリース9.1.2

リリース日: 2011-12-05

このリリースは9.1.1に対し、各種不具合を修正したものです。9.1メジャーリリースにおける新機能については[項E.15](#)を参照してください。

E.13.1. バージョン9.1.2への移行

9.1.Xからの移行ではダンプ/リストアは不要です。

しかし、`information_schema.referential_constraints`ビュー定義において長期に渡って潜在したエラーが見つかりました。このビューからの正確な結果に依存している場合、後述の変更点の最初で説明する通りその定義を置き換えなければなりません。

また`citext`データ型を使用し、かつ、`pg_upgrade`を実行して過去のメジャーリリースからアップグレードした場合、`CREATE EXTENSION citext FROM unpackaged`を実行して、`citext`操作における照合順序に関連した失敗を防止しなければなりません。`citext`データ型のインスタンスを含む、9.1より前のダンプからリストアする場合も、同じことが必要です。もし9.1.2にアップグレードする前にすでに`CREATE EXTENSION`を実行していた場合は、代わりに変更点の2番目で説明する通りに手作業でカタログを更新しなければなりません。

E.13.2. 変更点

- `information_schema.referential_constraints`ビューの不具合を修正しました。(Tom Lane)

このビューでは、外部キーと依存するプライマリキーまたは一意キー制約との対応付けについて十分な注意を行っていませんでした。このため、外部キー制約がまったく表示されない、何回も表示される、あるいは、実際の依存する制約とは異なる制約に依存するものと言い張るといった結果になることがありました。

このビュー定義は`initdb`によってインストールされますので、ただアップグレードしただけでは問題は解消されません。既存のインストールでこの問題を解消させなければならない場合、（スーパーユーザとして）`information_schema`スキーマを削除して、`SHAREDIR/information_schema.sql`を元に再作成することができます。（`SHAREDIR`の場所が分からない場合は`pg_config --sharedir`を実行してください。）これは修正対象のデータベースそれぞれで繰り返さなければなりません。

- `contrib/citext`のアップグレードスクリプトで`citext`列とインデックスの照合順序を修正するようにしました。(Tom Lane)

9.1以前のサーバから`pg_upgrade`を行う時および`citext`型を含む9.1より前のダンプを9.1サーバにロードする時、既存の`citext`列とインデックスは正しく照合順序の設定可能なデータ型として印付けられません。このためこれらに対する操作が"could not determine which collation to use for string comparison"などのエラーで失敗してしまいます。この変更により、`CREATE EXTENSION citext FROM unpackaged`時に`citext`モジュールを適切な9.1の拡張としてアップグレードするものと同じスクリプトでこれらの問題は修正されます。

この問題の被害を受けるデータベースにすでにアップグレードしており、かつ、すでに`CREATE EXTENSION`コマンドを実行している場合は、`SHAREDIR/extension/citext--unpackaged--1.0.sql`の最後にある`UPDATE`コマンドを（スーパーユーザとして）手作業で実行することができます。（`SHAREDIR`の場所が分からない場合は`pg_config --sharedir`を実行してください。）確かに行ったかどうか不明な場合、これを繰り返し行っても危険はありません。

- スカラを返す関数の出力を結合する`UPDATE`または`DELETE`実行時にクラッシュする可能性を修正しました。(Tom Lane)

クラッシュは対象の行が同時に更新された場合にのみ起こり得ました。このためこの問題は間欠的にのみ表面化しました。

- GINインデックス更新に関するWALレコードの不正確な再生を修正しました。(Tom Lane)

このためクラッシュ後またはホットスタンバイサーバにおいて、インデックス項目の検索に過渡的に失敗する結果になり得ました。しかし、この問題はインデックスに対する次の`VACUUM`で修復されます。

- `CREATE TABLE dest AS SELECT * FROM src`または`INSERT INTO dest SELECT * FROM src`時のTOAST関連のデータの破損を修正しました。(Tom Lane)

テーブルがALTER TABLE ADD COLUMNによって変更された場合、データを逐語的に別のテーブルにコピーしようとした時に特定の境界条件で破損した結果になることがありました。この問題は、8.4以降において、この構文においてのみ明確に発生し得たものですが、同じ不具合を引き起こす他のコードパスがある場合に備えて、より過去のバージョンにおいてもパッチを適用しました。

- ホットスタンバイの起動時に失敗する可能性を修正しました。(Simon Riggs)
- 初期スナップショットが不完全である場合のホットスタンバイの起動を高速にしました。(Simon Riggs)
- 無効なsyscache項目からTOASTテーブルにアクセスする時の競合条件を修正しました。(Tom Lane)

典型的な兆候は、"missing chunk number 0 for toast value NNNNN in pg_toast_2619"のような一時的なエラーです。ここで挙げられるTOASTテーブルは常にシステムカタログに属します。

- パラメータのデフォルト式で使用する項目について関数の依存関係を追跡します。(Tom Lane)

これまでは、参照されるオブジェクトが、参照する関数を削除または変更することなく、削除されることがあり得ました。このため関数を使用される時に誤動作することになりました。単にこの更新をインストールするだけでは依存関係が消失した項目が修正されないことに注意してください。修正するためには、その後にこうした関数それぞれにCREATE OR REPLACEを行う必要があります。デフォルト値が組み込み以外のオブジェクトに依存する関数があれば、これを行うことを勧めます。

- 入れ子状ループ結合におけるプレースホルダ変数の不正確な管理を修正しました。(Tom Lane)

この不具合は"variable not found in subplan target list"というプランナエラーをもたらすことが分かっています。さらに、外部結合が含まれる場合に間違った問い合わせ結果になる可能性もあります。

- 集約を含む式でソートされたウィンドウ関数を修正しました。(Tom Lane)

これまでは"could not find pathkey item to sort"というプランナエラーで失敗することがあり得ました。

- "MergeAppend child's targetlist doesn't match MergeAppend"というプランナエラーを修正しました。(Tom Lane)
- 照合順序設定可能な入力と不可能な入力の両方を持つ演算子に対するインデックス対応付けを修正しました。(Tom Lane)

9.1.0では、照合順序の設定ができない入力型の左辺と設定ができる入力型の右辺を持つインデックス付け可能な演算子では、左辺の列のインデックスに対応しないものと認識されました。例えばhstore ? textです。

- 複数のOUTパラメータを持つ、集合を返すSQL関数をインライン化することができます。(Tom Lane)
- 結合除去において遅延された一意性制約を信用しません。(Tom Lane、Marti Raudsepp)

遅延された一意性制約はトランザクション内部で保持されないかもしれません。このため、間違った問い合わせ結果をもたらすかもしれないとみなします。

- DatumGetInetP() が1バイトヘッダを持つinetデータを展開するようにしました。さらに展開を行わない、新しいマクロDatumGetInetPP()を追加しました。(Heikki Linnakangas)

この変更がコアコードに影響することはありませんが、DatumGetInetP()が通常の規約の通りに展開したデータを生成することを想定しているアドオンのコードにおけるクラッシュを防止できるかもしれません。

- money型の入出力におけるロケールのサポートを改良しました。(Tom Lane)

すべての標準lc_monetary書式付けオプションをサポートしていないという点以外にも、入力関数と出力関数に一貫性がなかったため、ダンプされたmoney値では再度読み取ることができないロケールがありました。

- transform_null_equalsがCASE foo WHEN NULL ...式に影響を与えないようにしました。(Heikki Linnakangas)

transform_null_equalsは、CASE構文によって内部的に生成された等価な検査ではなく、直接ユーザによって書かれたfoo = NULLに影響を与えることのみを前提としています。

- 自己参照外部キーをより良くサポートするために、外部キートリガの生成順序を変更しました。(Tom Lane)

カスケードしている外部キーが自身のテーブルを参照する場合、1行の更新が1つのイベントでON UPDATEトリガとCHECKトリガの両方を発行します。ON UPDATEトリガを最初に実行しなければなりません。そうしないと、CHECKは最終的ではない行の状態を検査することになり、不適切なエラーが発生する可能性があります。しかしこれらのトリガの発行順序はその名前によって決まります。トリガは"RI_ConstraintTrigger_NNNN"という規約に従った名前前で自動生成さ

れますので、一般的には作成順序によって決まります。適切な修正にはこの規約の変更が必要です。9.2で変更する予定ですが、既存のリリースで変更することには危険があるように思われます。このため、このパッチはトリガの作成順序を変更するだけです。この種のエラーが起きたユーザは、トリガを正しい順序にするために、外部キー制約を削除し再作成しなければなりません。

- DROP OPERATOR FAMILY内でIF EXISTSが正しく動作するように修正しました。(Robert Haas)

- 拡張自身のスクリプトの中から拡張を削除することを禁じました。(Tom Lane)

これにより、拡張依存関係が不正確に管理された場合のおかしな動作を防止します。

- 自動生成される型を拡張のメンバと印付けません。(Robert Haas)

リレーション行型と自動生成される配列型をpg_depend内の拡張自身のメンバ関係項目に持たせる必要はありません。またこうした項目を作成すると、拡張の更新が複雑になります。

- CREATE EXTENSION時の、事前に存在する無効なsearch_path設定をうまく対処します。(Tom Lane)

- バッファ割当率を追跡する際の浮動小数アンダーフローを防止します。(Greg Matthews)

これ自体は害がないものですが、特定のプラットフォームではこれにより煩わしいカーネルログメッセージが現れます。

- 自動バキュームトランザクションがシリアライズブルモードで実行しないようにしました。(Tom Lane)

これまでの自動バキュームはクラスタ全体のデフォルトトランザクション隔離レベルを使用していましたが、READ COMMITTED以上のレベルを使用する必要はありません。またSERIALIZABLEを使用すると他のプロセスで不要な遅延が発生し得ました。

- walsenderプロセスがSIGTERMに適切に反応することを確実にしました。(Magnus Hagander)

- ベースバックアップからpostmaster.optsを除外しました。(Magnus Hagander)

- Windowsにおける子プロセス起動時設定ファイルの名前と行番号を保持します。(Tom Lane)

これまでこれらはpg_settingsビューで正しく表示されませんでした。

- ecpgのSQLDA領域における不正確なフィールドの整列を修正しました。(Zoltan Borszormenyi)

- psqlのコマンド履歴内で空行を含むコマンドを保持します。(Robert Haas)

これまでの動作では、例えば文字列リテラル内の空行が削除された場合に問題が起きました。

- pg_dumpにおける、プラットフォーム固有の無限ループを防止します。(Steve Singer)

- pg_dumpにおける平文テキスト出力書式の圧縮を修正しました。(Adrian Klaver、Tom Lane)

pg_dumpは歴史的に、-Fがない-zを平文テキスト出力をgzipで圧縮した形で出力しなければならないことを意味すると解釈していました。この動作に戻しました。

- テーブルの行型など自動生成された型の間でユーザが定義したキャストをダンプするようにpg_dumpを修正しました。(Tom Lane)

- pg_dumpにおいて外部サーバ名の引用符付けの失敗を修正しました。(Tom Lane)

- pg_upgradeについて様々な修正を行いました。(Bruce Momjian)

排他制約の正確な取り扱い、Windowsにおける失敗の防止、8.4データベースにおける対応しないTOASTテーブル名に関する警告の抑制。

- PL/pgSQLにおいて行型を定義するための外部テーブルを許可します。(Alexander Soudakov)

- PL/Perl関数の結果の変換を取り決めました。(Alex Hunsaker、Tom Lane)

voidを返すPL/Perl関数とその最後のPerl文の結果値を無視するという9.1より前の動作に戻しました。9.1.0では、最後の文が参照を返す場合にエラーになりました。また、複合型に対して文字列値を返す場合も、文字列が型の入力書式に合う限り、確実に動作するようにしました。さらに、関数の宣言された結果型が配列または複合型ではない場合、それぞれPerlの配列またはハッシュを返そうとするとエラーになります。(こうした場合、9.1より前のバージョンでは、ほとんど役に立たないARRAY(0x221a9a0)やHASH(0x221aa90)といった文字列が返されました。)

- PL/Perlの文字列が常に正しくUTF8で符号化されることを確実にしました。(Amit Khandekar、Alex Hunsaker)

- PL/Perlの構築時に好みのバージョンのxsubppを使用します。 オペレーティングシステムのメインコピーである必要はありません。(David Wheeler、Alex Hunsaker)
- PL/Pythonの例外においてSQLSTATEを正しく伝搬します。(Mika Eloranta、Jan Urbanski)
- 構築対象メジャーバージョン以外のバージョンのPython用のPL/Python拡張ファイルをインストールしません。(Peter Eisentraut)
- スクリプトファイルがpsqlに渡される場合に有用なエラーメッセージを出力するように、すべてのcontrib拡張スクリプトファイルを変更しました。(Andrew Dunstan、Tom Lane)

これは、これらのファイルをロードするためにCREATE EXTENSIONという新しい方法をユーザに知らせる助けになるはずで
す。ほとんどの場合、スクリプトファイルを直接読み込んでも失敗しますが、解釈し難いメッセージが出力されました。

- contrib/dict_intおよびcontrib/dict_xsyn内の間違ったコードを修正しました。(Tom Lane)

一部の関数で、palloc() で返されるメモリはゼロ埋めされていることが保証されていると間違っただけを前提をしていました。

- 通常のリグレッション試験からcontrib/sepgsqlを取り除きました。(Tom Lane)

これらの試験の設定にはroot権限が必要ですので、自動的に実行させるのは実用的ではありません。代わりに手作業による方法に切替え、その試験を補助する試験スクリプトを用意しました。

- contrib/unaccentの設定ファイル解析において様々なエラーを修正しました。(Tom Lane)
- pgstatindex() において問い合わせキャンセル割込みを適切に受け付けます。(Robert Haas)
- Mac OS X起動スクリプトにおける間違ったログファイル名の引用符付けを修正しました。(Sidar Lopez)
- 意図せずに有効になっていたWAL_DEBUGを元に戻しました。(Robert Haas)

幸いなことにデバッグ用ツールが作動する時でもこれは非常に安価なものです。しかしデフォルトで有効にすることは意図していませんので、元に戻しました。

- VPATH構築ですべてのサーバヘッダファイルが適切にインストールされることを確実にしました。(Peter Eisentraut)
- 冗長エラーメッセージ内で報告されるファイル名を短くしました。(Peter Eisentraut)

通常構築では常にCファイルの名前だけがエラーメッセージ呼び出しの中で報告されました。しかしこれまでVPATH構築では絶対パス名が報告されました。

- 中央アメリカ用のWindowsの時間帯名の解釈を修正しました。(Tom Lane)

通常中央アメリカではまったく夏時間が認められませんので、"Central America Standard Time"をCST6CDTではなくCST6に対応付けしました。

- 時間帯データファイルをtzdata release 2011nに更新しました。ブラジル、キューバ、フィジィ、パレスチナ、ロシア、サモアにおける夏時間規則の変更、アラスカ、イギリス領東アフリカの歴史的な修正が含まれています。

E.14. リリース9.1.1

リリース日: 2011-09-26

このリリースは9.1.0に対し、わずかな不具合を修正したものです。 9.1メジャーリリースにおける新機能については[項E.15](#)を参照してください。

E.14.1. バージョン9.1.1への移行

9.1.Xからの移行ではダンプ/リストアは不要です。

E.14.2. 変更点

- pg_options_to_tableを値を持たないオプションに対してNULLを返すようにしました。(Tom Lane)

これまではこのような場合にサーバがクラッシュしてしまいました。

- GiSTインデックススキャン終了時のメモリリークを修正しました。(Tom Lane)

すでに多くの行を持つテーブルに対して新たなGiSTを元にした排他制約を作成する際の検証など、多くの分割GiSTインデックススキャンを行うコマンドが一時的にこのリークの原因となる大容量のメモリを求めることがあり得ました。

- CREATE TEMPORARY TABLEにおけるpg_tempスキーマへの明示的な参照を修正しました。(Robert Haas)

これは使用できるものでしたが、9.1.0では失敗しました。

E.15. リリース9.1

リリース日: 2011-09-12

E.15.1. 概要

このリリースは、PostgreSQLが伝統的なリレーショナルデータベース機能群を超え、PostgreSQL独特の新しく革新的な機能を持つものになったことを表しています。 リリース9.0で導入されたストリーミングレプリケーション機能は、同期レプリケーションオプション、ストリームを使用したバックアップ、監視の改良を追加することにより非常に強化されました。 以下に多くの改良を示します。

- [同期レプリケーション](#)が可能です。
- [外部テーブル](#)のサポートを追加しました。
- 列単位の[照合順序](#)のサポートを追加しました。
- PostgreSQLの追加パッケージを簡略化する[拡張](#)を追加しました。
- 本物の[シリアライズابل隔離レベル](#)を追加しました。
- [CREATE TABLE](#)内のUNLOGGEDオプションを使用する、ログを取らないテーブルをサポートします。
- [WITH](#)句内でデータ変更コマンド（INSERT/UPDATE/DELETE）が可能です。
- [GiSTインデックス](#)に（演算子で順序付けされた）近傍検索を追加しました。
- [SECURITY LABEL](#) コマンドと[SELinux権限制御](#)のサポートを追加しました。
- [PL/Python](#)サーバサイド言語を更新しました。

上記の項目については後述の節で詳しく説明します。

E.15.2. バージョン9.1への移行

全ての以前のリリースからデータの移行を行うためには、pg_dumpを使用したダンプ/リストア、または、pg_upgradeの使用が必要です。

バージョン9.1には、以前のリリースとの互換性に影響する多くの変更が含まれます。 以下の非互換性に注意してください。

E.15.2.1. 文字列

- [standard_conforming_strings](#)のデフォルト値が有効に変わりました。(Robert Haas)

デフォルトで文字列リテラル内のバックスラッシュが通常の文字になります。 この変更により長く存在した標準SQLとの非互換性がなくなります。 長い間[escape_string_warning](#)がこの使用に関する警告を発生していました。 E''文字列が文字列内にバックスラッシュを埋め込む適切な方法であり、この変更による影響を受けません。

警告

この変更は、これを想定せずに古い規則にしたがった独自の文字列エスケープ処理を行うアプリケーションを破壊する可能性があります。 この結果SQLインジェクションセキュリティホールをもたらし、ひどいものです。 信頼できない入力を外部にさらすアプリケーションを試験し、テキスト文字列内のバックスラッシュと単一引用符を正しく扱っていることを確認してください。

E.15.2.2. キャスト

- 複合型に対する、関数形式の型キャストおよび属性形式の型キャストを許しません。(Tom Lane)

例えば`composite_value.text`と`text(composite_value)`は許されません。 意図しないこの構文の使用により頻繁に不具合

報告がなされました。これは不具合ではありませんが、こうした表現を拒絶するように戻す方が良いように考えました。複合型全体のキャストを本当に意図している場合には`CAST`および`::`構文がまだ使用できます。

- 配列に基づくドメインに対するキャスト検査を厳しくしました。(Tom Lane)

ドメインが配列型に基づく場合、要素を取り出したり代入したりするためにドメイン値に添え字を付けるなど、配列要素にアクセスするためにドメイン型に"目を通す"ことができます。例えば`UPDATE ... SET domaincol[5] = ...`などこうしたドメイン値の要素への代入はドメイン型の制約で再検査されるようになりました。これまではこの検査は飛ばされていました。

E.15.2.3. 配列

- `string_to_array()` が長さゼロの文字列に対して空の配列を返すように変更しました。(Pavel Stehule)

これまではNULL値を返しました。

- NULLセパレータが文字列を文字に分割するように`string_to_array()`を変更しました。(Pavel Stehule)

これまではNULL値を返しました。

E.15.2.4. オブジェクトの操作

- BEFORE/AFTERトリガに対する不適切な検査を修正しました。(Tom Lane)

トリガが次の3つの場合で発行できるようになりました。BEFORE、AFTER、何らかの作業のINSTEAD OF（代わり）。トリガ関数の作者は、3つの場合すべてにおいてロジックが健全であることを検証しなければなりません。

- ロールにコメントを設定するためにはスーパーユーザ権限またはCREATEROLE権限が必要です。(Tom Lane)

E.15.2.5. サーバの設定

- 後方に移動することがないように`pg_last_xlog_receive_location()`を変更しました。(Fujii Masao)

これまでは、ストリーミングレプリケーションが再開する時に`pg_last_xlog_receive_location()`の値が後方に移ることがあり得ました。

- `log_connections`にしたがったレプリケーション接続のログが取られます。(Magnus Hagander)

これまでは、レプリケーション接続のログは常に取りられていました。

E.15.2.6. PL/pgSQLサーバサイド言語

- 付与された例外ブロックで例外が捕捉できるようにパラメータを持たないPL/pgSQLのRAISEコマンドを変更しました。(Piyush Newe)

これまではコードブロック内のRAISEは常に付与された例外ブロックのスコープと同じになりました。このため同じスコープのものは捕捉できませんでした。

- 他の手続き言語と一貫性を持つようにPL/pgSQLのエラー行のコードの番号付けを調整しました。(Pavel Stehule)

これまでのPL/pgSQLでは、関数本体の先頭にある空行を無視していました(数えませんでした)。これは他のすべての言語と一貫していませんでしたので、この特別な扱いを取りやめました。

- INパラメータとOUTパラメータの名前に競合がある場合にPL/pgSQLが警告を出すようにしました。(Tom Lane)

これまでは重複を検出しませんでした。その名前は警告なく単にOUTパラメータのみを参照していました。

- PL/pgSQL変数の型修飾子がSQLパーサから可視になりました。(Tom Lane)

PL/pgSQL変数に付与された(varchar型の長さ制限などの)型修飾子はこれまで代入時に強制されていましたが、他の目的のためにはすべて無視されていました。こうした変数が、より同じ修飾子を付けて宣言されたテーブル列のように動作するようになりました。ほとんどの場合で目に見えるような違いが発生するとは思われませんが、PL/pgSQL関数で発行された同じSQLコマンドで微妙な違いが発生するかもしれません。

E.15.2.7. Contrib

- すべてのcontribモジュールは、独自のSQLスクリプトを手作業で呼び出すのではなく、`CREATE EXTENSION`を用いてイン

ストールされます。(Dimitri Fontaine、Tom Lane)

バージョン9.0のcontribモジュールを含む既存のデータベースを更新する場合は、既存のcontribモジュールのオブジェクトを拡張としてまとめるために`CREATE EXTENSION ... FROM unpackaged`を使用してください。9.0より前のバージョンから更新する場合は、古いアンインストールスクリプトを使用してcontribモジュールのオブジェクトを削除してから`CREATE EXTENSION`を使用してください。

E.15.2.8. その他の非互換性

- [pg_stat_reset\(\)](#)がすべてのデータベースレベルの統計情報を再初期化するようになりました。(Tomas Vondra)
一部のpg_stat_databaseカウンタが再初期化されていませんでした。
- [information_schema.triggers](#)の一部の列名を新しい標準SQLの名前に合致するように修正しました。(Dean Rasheed)
- ECPGのカーソル名を大文字小文字を区別せずに扱います。(Zoltan Boszormenyi)

E.15.3. 変更点

PostgreSQL 9.1と前回のメジャーリリースとの間の変更について詳細を以下に記します。

E.15.3.1. サーバ

E.15.3.1.1. 性能

- [CREATE TABLE](#)内のUNLOGGEDオプションを使用する、ログを取らないテーブルをサポートします。(Robert Haas)
こうしたテーブルは通常のテーブルよりも優れた更新性能を提供しますが、クラッシュ時に安全ではありません。サーバがクラッシュした場合、その内容は自動的に消去されます。またその内容はレプリケーションのスレーブには伝播されません。
- FULL OUTER JOINをハッシュ結合として実現することができます。また、LEFT OUTER JOINまたはRIGHT OUTER JOINの片側をハッシュ化することができます。(Tom Lane)
これまでのFULL OUTER JOINではマージ結合としてのみ実現することができました。またLEFT OUTER JOINとRIGHT OUTER JOINは結合のNULLを持つ側のみをハッシュ化することができました。これらの変更により、さらに問い合わせ最適化の可能性が加わりました。
- fsync要求をまとめます。(Robert Haas、Greg Smith)
これは非常に書き込み負荷が高い状況での性能を大きく改良します。
- [commit_siblings](#)の性能を向上しました。(Greg Smith)
これにより、より少ないオーバーヘッドでcommit_siblingsを使用することができます。
- 大規模なispell辞書で必要とされるメモリ量を減少しました。(Pavel Stehule、Tom Lane)
- "blind writes"後データファイルが開いたままになることを防止します。(Alvaro Herrera)
これは、バックエンドが削除された後も長く間開いた状態のファイルを保持するかもしれず、カーネルによるディスク領域の回収を妨げるという状況を修正します。

E.15.3.1.2. オプティマイザ

- 継承テーブルのスキャンが有意にソートされた結果を返すことができます。(Greg Stark、Hans-Jurgen Schonig、Robert Haas、Tom Lane)
これにより、継承テーブルを持ち、ORDER BY、LIMIT、MIN/MAXを使用する問い合わせをより良く最適化することができます。
- GINインデックススキャンのコスト推定を改良しました。(Teodor Sigaev)
- 集約およびウィンドウ関数のコスト推定を改良しました。(Tom Lane)

E.15.3.1.3. 認証

- [pg_hba.conf](#)において、ホスト名およびホスト接尾辞（.example.comなど）をサポートします。(Peter Eisentraut)
これまではホストのIPアドレスとCIDR値のみがサポートされていました。
- [pg_hba.conf](#)のホスト列においてallキーワードをサポートします。(Peter Eisentraut)
これまでは、このために0.0.0.0/0または::/0を使用していました。
- Unixソケット接続をサポートしないプラットフォームにおいて、[pg_hba.conf](#)内のlocal行を拒絶します。(Magnus Hagander)
これまではこうした行は警告なく無視されましたが、驚くような結果を招くかもしれませんでした。これにより他の未サポートの場合とより同じように動作するようになりました。
- [GSSAPI](#)を[SSPI](#)を介したサーバ認証に使用できるようになりました。(Christian Ullrich)
特にこれによりUnixベースのGSSAPIクライアントがWindowsサーバでSSPI認証させることができます。
- ローカルソケット経由の[ident](#)認証を[peer](#)と名付けました。(Magnus Hagander)
古い用語はまだ後方互換性のために受け入れられます。しかし2つの方式は根本的に異なるものですので、別の名前を採用する方が良いのかもしれませんが。
- 資格制御メッセージの使用を防止するために、[peer](#)認証を書き直しました。(Tom Lane)
この変更によりピア認証コードがより単純に、より良い性能がでるようになりました。しかし、[getpeereid](#)関数が同等のソケット操作を提供するプラットフォームが必要です。分かっている限りでは、これまでピア認証が動作し、このバージョンで動作しなくなったプラットフォームは5.0より前のNetBSDのみです。

E.15.3.1.4. 監視

- [log_checkpoints](#)により制御される、リスタートポイントおよびチェックポイントのログ処理に詳細を追加しました。(Fujii Masao, Greg Smith)
WALファイルの詳細や同期処理に関する詳細が含まれるようになりました。
- ログ処理コントローラにより作成されるログファイルの権限を制御する、[log_file_mode](#)が追加されました。(Martin Pihlak)
- syslogログ処理用の最大行長を接頭辞分と900バイトをたし合わせた値に減らしました。(Noah Misch)
これにより、一般的な2kBではなく1kBの長さ制限を持つsyslog実装における長大な行の切り詰めを防止します。

E.15.3.1.5. 統計情報ビュー

- [client_hostname](#)列を[pg_stat_activity](#)に追加しました。(Peter Eisentraut)
これまではクライアントのアドレスのみが報告されていました。
- [pg_stat_xact_*](#)統計情報関数およびビューを追加しました。(Joel Jacobson)
これらはデータベース全体の統計情報カウンタビューと似ていますが、現在のトランザクションのみを計数に反映します。
- データベースレベルおよびバックグラウンドライタの統計情報ビューに最終再初期化時間を追加しました。(Tomas Vondra)
- [pg_stat_*_tables](#)ビューにバキューム操作および解析操作の回数を示す列を追加しました。(Magnus Hagander)
- [buffers_backend_fsync](#)列を[pg_stat_bgwriter](#)に追加しました。(Greg Smith)
この新しい列は、バッファのバックグラウンドfsyncの回数を数えます。

E.15.3.1.6. サーバの設定

- [wal_buffers](#)の自動調整を提供します。(Greg Smith)
[wal_buffers](#)の値はデフォルトで[shared_buffers](#)の容量に応じて自動的に調整されるようになりました。

`deadlock_timeout`、`log_min_duration_statement`、`log_autovacuum_min_duration`の最大値を増やしました。(Peter Eisentraut)

これまでのこれらのパラメータの最大値はそれぞれおよそ35分でしかありませんでした。もっと大きな値が許されるようになりました。

E.15.3.2. レプリケーションとリカバリ

E.15.3.2.1. ストリーミングレプリケーションと継続的アーカイブ

- [同期レプリケーション](#)が可能です。(Simon Riggs, Fujii Masao)

これによりプライマリサーバは、コミットを認める前にスタンバイがトランザクション情報をディスクに書き出すまで待機することができます。一度に1つのスタンバイのみが、[synchronous_standby_names](#)設定で制御される同期スタンバイの役割を果たすことができます。同期レプリケーションは、[synchronous_commit](#)設定を用いてトランザクション単位で有効にしたり無効にしたりすることができます。

- ストリーミングレプリケーション接続を使用してファイルシステムバックアップをスタンバイサーバに送信するためのプロトコルサポートを追加しました。(Magnus Hagander, Heikki Linnakangas)

これによりスタンバイサーバを構築すると気にファイルシステムバックアップを手作業で転送する必要がなくなります。

- `replication_timeout`設定を追加しました。(Fujii Masao, Heikki Linnakangas)

`replication_timeout`間隔より長く待機状態であったレプリケーション接続は自動的に終了します。これまでは失敗している接続は、多くの状況で不便な長さであるTCPタイムアウトに達するまで通常検出されませんでした。

- 新しいスタンバイサーバの作成やデータベースバックアップのための[pg_basebackup](#)コマンドラインツールを追加しました。(Magnus Hagander)

- ロール用に[replication権限](#)を追加しました。(Magnus Hagander)

これはストリーミングレプリケーションで使用される読み取り専用の権限です。これによりスーパーユーザ以外のロールがレプリケーション用接続を使用することができます。以前はスーパーユーザのみがレプリケーション用接続を初期化することができました。デフォルトでスーパーユーザは引き続きこの権限を持ちます。

E.15.3.2.2. レプリケーションの監視

- WAL送信プロセスの活動を表示する[pg_stat_replication](#)システムビューを追加しました。(Itagaki Takahiro, Simon Riggs)

これは接続されたスタンバイサーバすべての状態を報告します。

- [pg_last_xact_replay_timestamp\(\)](#)監視用関数を追加しました。(Fujii Masao)

これは、もっとも最近プライマリでコミットまたはアボートしたレコードがスタンバイに適用された時間を報告します。

E.15.3.2.3. ホットスタンバイ

- スタンバイがプライマリの古い行バージョンの整理を延期させることを有効にする[hot_standby_feedback](#)設定パラメータを追加しました。(Simon Riggs)

これはスタンバイ上で長時間実行している問い合わせがキャンセルされることを防ぐために役立ちます。

- キャンセルされた問い合わせとその理由を示す[pg_stat_database_conflicts](#)システムビューを追加しました。(Magnus Hagander)

キャンセルは、削除されたテーブル空間、ロックタイムアウト、古いスナップショット、ピン付けされたバッファ、デッドロックが原因で起こり得ます。

- [pg_stat_database](#)に[conflicts](#)カウントを追加しました。(Magnus Hagander)

これはデータベース内で発生した競合回数です。

- [max_standby_archive_delay](#)および[max_standby_streaming_delay](#)の最大値を増やしました。

これまでのこれらのパラメータの最大値はそれぞれおよそ35分でしかありませんでした。もっと大きな値が許されるよ

うになりました。

- 削除されたデータベースが原因のリカバリの競合を報告する[ERRCODE_T_R_DATABASE_DROPPED](#)エラーコードを追加しました。(Tatsuo Ishii)

これは接続プールソフトウェアで有用です。

E.15.3.2.4. リカバリの制御

- ストリーミングレプリケーションの再生を制御する関数を追加しました。(Simon Riggs)

新しい関数は、[pg_xlog_replay_pause\(\)](#)、[pg_xlog_replay_resume\(\)](#)、および、[pg_is_xlog_replay_paused\(\)](#) 状態関数です。

- 指定対象でリカバ리를 休止させるために[pause_at_recovery_target](#) 設定を `recovery.conf` に追加しました。(Simon Riggs)

これによりリカバリ中のサーバに問い合わせ、リカバリポイントが希望のものかどうかを検査することができます。

- [pg_create_restore_point\(\)](#) を使用して、リストアポイントを名前を付けて作成する機能を追加しました。(Jaime Casanova)

`recovery.conf` の新しい [recovery_target_name](#) 設定を使用して、これらの名前付きリストアポイントをリカバリ対象として指定することができます。

- スタンバイリカバリが新しい時系列に自動的に切り替えることができます。(Heikki Linnakangas)

スタンバイサーバは新しい時系列用のアーカイブディレクトリを定期的にスキャンします。

- バックエンドのクラッシュ後の自動サーバ再起動を無効にする [restart_after_crash](#) 設定を追加しました。(Robert Haas)

これにより外部のクラスタ管理ソフトウェアがサーバを再起動すべきかどうかを制御できるようになります。

- `recovery.conf` で `postgresql.conf` と同じ引用符付けを使用できるようになりました。(Dimitri Fontaine)

これまではすべての値を引用符付けしなければなりませんでした。

E.15.3.3. 問い合わせ

- 本物の [シリアライズブル隔離レベル](#) を追加しました。

これまでは、シリアライズブルを要求した場合に、そのトランザクション全体で単一MVCCスナップショットが使用されることのみが保証され、文書化された特定の例外が許容されていました。これまでのスナップショット隔離動作は、[REPEATABLE READ](#) 隔離レベルを要求することでまだ使用することができます。

- [WITH](#) 句内でデータ変更コマンド (INSERT/UPDATE/DELETE) が可能です。(Marko Tiikkaja, Hitoshi Harada)

これらのコマンドは `RETURNING` を使用して含有する問い合わせにデータを渡すことができます。

- [WITH](#) 句を INSERT、UPDATE、DELETE 文に付与することができます。(Marko Tiikkaja, Hitoshi Harada)

- プライマリキーが `GROUP BY` 句で指定されていた場合、問い合わせの対象リスト内で [GROUP BY](#) されていない列が許されず。(Peter Eisentraut)

標準SQLではすでにこの動作が許されています。またプライマリキーですので結果にあいまいさはありません。

- [UNION/INTERSECT/EXCEPT](#) 句内で `DISTINCT` キーワードを使用することができます。(Tom Lane)

`DISTINCT` はデフォルトの動作ですので、このキーワードは冗長です。しかし標準SQLでは許されています。

- `EXPLAIN ANALYZE` の時、ルールを持つ普通の問い合わせが同じスナップショットを使用するように修正しました。(Marko Tiikkaja)

これまでの `EXPLAIN ANALYZE` は、ルールを持つ問い合わせに対して多少時期が異なるスナップショットを使用していました。 `EXPLAIN ANALYZE` の動作はより論理的になるべきと評価されました。

E.15.3.3.1. 文字列

- 列単位の[照合順序](#)のサポートを追加しました。(Peter Eisentraut, Tom Lane)

これまでの照合順序（テキスト文字列のソート順序）はデータベース作成時のみで設定することができました。照合順序は、標準SQLのCOLLATE句を用いて、列、ドメイン、インデックス、式単位で設定することができるようになりました。

E.15.3.4. オブジェクトの操作

- PostgreSQLの追加パッケージを簡略化する[拡張](#)を追加しました。(Dimitri Fontaine, Tom Lane)

拡張は新しい[CREATE/ALTER/DROP EXTENSION](#)コマンドで制御されます。これは、PostgreSQLインストレーションにまつまったオブジェクトを追加するためのその場しのぎの方法を置き換えます。

- [外部テーブル](#)のサポートを追加しました。(Shigeru Hanada, Robert Haas, Jan Urbanski, Heikki Linnakangas)

これによりデータベースの外部に格納されたデータをPostgreSQLに格納されたデータそのものであるかのように使用することができます。しかし現時点では外部テーブルは読み取り専用です。

- [ALTER TYPE](#)を用いて既存の列挙型に新しい値を追加することができます。(Andrew Dunstan)

- [ALTER TYPE ... ADD/DROP/ALTER/RENAME ATTRIBUTE](#)を追加しました。(Peter Eisentraut)

これにより複合型の変更ができます。

E.15.3.4.1. オブジェクトに対するALTER

- 型付けされたテーブルに対する[ALTER TYPE](#)操作に[RESTRICT/CASCADE](#)を追加しました。(Peter Eisentraut)

これはADD/DROP/ALTER/RENAME ATTRIBUTEのカスケード動作を制御します。

- `ALTER TABLE name {OF | NOT OF} type`をサポートします。(Noah Misch)

この構文により、スタンドアローンのテーブルを型付けされたテーブルにすること、型付けされたテーブルをスタンドアローンにすることができます。

- `ALTER ... SET SCHEMA`コマンド内でより多くのオブジェクトに対するサポートを追加しました。(Dimitri Fontaine)

このコマンドは、変換、演算子、演算子クラス、演算子族、全文検索設定、全文検索辞書、全文検索パーサ、全文検索テンプレートをサポートするようになりました。

E.15.3.4.2. [CREATE/ALTER TABLE](#)

- `ALTER TABLE ... ADD UNIQUE/PRIMARY KEY USING INDEX`を追加しました。(Gurjeet Singh)

これによりプライマリキーまたは一意性制約を、同時に作成された一意性インデックスを含む、既存の一意性インデックスを用いて定義することができます。

- `ALTER TABLE`で検証なく外部キーを追加することができます。(Simon Riggs)

新しいオプションはNOT VALIDという名前です。この制約の状態を後でVALIDATEDに変更することで有効性検査を行うことができます。これらを用いて、読み書き操作に対して最小限の影響で外部キーを追加することができます。

- 適切な状況においてテーブル書き換えを行わない[ALTER TABLE ... SET DATA TYPE](#)が可能です。(Noah Misch, Robert Haas)

例えば、`varchar`列を`text`に変換する場合にテーブルの書き換えが必要なくなりました。しかし`varchar`列の長さ制限を増やす場合にはまだテーブルの書き換えが必要です。

- [CREATE TABLE IF NOT EXISTS](#)構文を追加しました。(Robert Haas)

これにより、テーブルがすでに存在する場合でもエラーを発生することなく、テーブル作成が許されます。

- 2つのバックエンドが同時に1つのテーブルに継承した子テーブルを追加しようとした時に潜在する"tuple concurrently updated"エラーを修正しました。(Robert Haas)

セッションが親テーブルを同時に更新できなくするために、[ALTER TABLE](#)が親テーブルに対してより強いロックを取得するようになりました。

E.15.3.4.3. オブジェクトの権限

- [SECURITY LABEL](#) コマンドを追加しました。(KaiGai Kohei)

これによりオブジェクトにセキュリティラベルを付与することができます。

E.15.3.5. ユーティリティ操作

- トランザクションレベルの[勧告的ロック](#)を追加しました。(Marko Tiikkaja)

これは既存のセッションレベルの勧告的ロックと似ていますが、こうしたロックはトランザクション終了時に自動的に解放されます。

- [TRUNCATE ... RESTART IDENTITY](#)はトランザクション的にシーケンスを再開させます。(Steve Singer)

これまでのカウンタは、バックエンドがコミット中のトランザクション処理とコミット完了の間にクラッシュした場合、同期されないままとなる可能性がありました。

E.15.3.5.1. [COPY](#)

- `ENCODING` オプションを[COPY TO/FROM](#)に追加しました。(Hitoshi Harada、Itagaki Takahiro)

これによりCOPYファイルの符号化方式をクライアント符号化方式とは別に指定することができます。

- 双方向[COPY](#)プロトコルのサポートを追加しました。(Fujii Masao)

現時点ではストリーミングレプリケーションによってのみ使用されています。

E.15.3.5.2. [EXPLAIN](#)

- `EXPLAIN VERBOSE`が、FunctionScanノード内の関数呼び出し式を示すようにしました。(Tom Lane)

E.15.3.5.3. [VACUUM](#)

- [VACUUM FULL VERBOSE](#)および[CLUSTER VERBOSE](#)の出力にさらに詳細を追加しました。(Itagaki Takahiro)

新しい情報には、有効タプル数、無効タプル数、`CLUSTER`が再構築するためにインデックスを使用するかどうかが含まれます。

- テーブルロックを獲得できなかった場合に[自動バキューム](#)が待機することを防ぎます。(Robert Haas)

後でそのテーブルのバキュームを試みます。

E.15.3.5.4. [CLUSTER](#)

- `CLUSTER`がインデックスをスキャンするよりも安価であるように見える場合にインデックスのスキャンではなく、テーブルをソートすることができます。

E.15.3.5.5. インデックス

- [GiSTインデックス](#)に（演算子で順序付けされた）近傍検索を追加しました。(Teodor Sigaev、Tom Lane)

これによりGiSTインデックスが`LIMIT`を持つ問い合わせにおいて N 個の近傍値を素早く返すことができます。例えば、

```
SELECT * FROM places ORDER BY location <-> point '(101,456)' LIMIT 10;
```

は指定された対象位置に最も近い10点を見つけ出します。

- [GINインデックス](#)がNULL値と空値をインデックス付けすることができます。(Tom Lane)

これにより完全なGINインデックススキャンができ、GINスキャンが失敗していた各種境界条件を修正することができます。

- [GINインデックス](#)がより良く重複する検索項目を認識することができます。(Tom Lane)

これは、特に不要な完全なインデックススキャンを防止できる場合に、インデックススキャンのコストを軽減します。

- [GiSTインデックス](#)が完全にクラッシュに対して安全になるように修正しました。(Heikki Linnakangas)

これまでは`REINDEX`が必要となる（通知される）まれな場合があります。

E.15.3.6. データ型

- `numeric`をより小さな容量で使用できます。通常は2バイトのヘッダです。(Robert Haas)

これまではすべての`numeric`値は4バイトのヘッダを持ちました。この変更によりディスク容量が少なく済みます。

- `money`による`money`の除算のサポートを追加しました。(Andy Balholm)
- `void`型に対するバイナリI/Oができます。(Radoslaw Smogura)
- 幾何演算における斜辺計算を改良しました。(Paul Matthews)

これは不要なオーバーフローを防止します。またより正確になる可能性があります。

- 配列値のハッシュ化をサポートします。(Tom Lane)

これはさらに問い合わせ最適化の選択肢が提供します。

- すべての列の型がソート可能でない限り、複合型をソート可能として扱いません。(Tom Lane)

これは、ソート処理なく問い合わせを実現できる場合、実行時に潜在する"could not identify a comparison function"失敗を防止します。また`ANALYZE`は、こうした複合型の列に対する不適切な統計情報収集メソッドの使用を試みません。

E.15.3.6.1. キャスト

- `money`と`numeric`間のキャストのサポートを追加しました。(Andy Balholm)
- `int4`および`int8`から`money`へのキャストのサポートを追加しました。(Joey Adams)
- 型付けされたテーブルの場合、テーブルの行型からテーブルのスーパー型へのキャストが可能です。(Peter Eisentraut)

これは、行型からスーパーテーブルの行型へのキャストが可能であるという既存の機能に類似したものです。

E.15.3.6.2. XML

- `XML EXISTS`および`xpath_exists()` XML関数を追加しました。(Mike Fowler)

これらはXPathマッチに使用されます。

- `xml_is_well_formed()`、`xml_is_well_formed_document()`、`xml_is_well_formed_content()` XML関数を追加しました。(Mike Fowler)

これらは、入力が適切に整形されたXMLかどうかを検査します。これらは、これまで廃止予定の`contrib/xml2`モジュールでのみ利用できた機能を提供するものです。

E.15.3.7. 関数

- Cの`printf()`と類似の動作を行う`format(text, ...)` SQL関数を追加しました。(Pavel Stehule、Robert Haas)
- 現時点では文字列、SQLリテラル、SQL識別子の書式付けをサポートします。

- `concat()`、`concat_ws()`、`left()`、`right()`、`reverse()` 文字列関数を追加しました。(Pavel Stehule)

これらは他のデータベース製品との互換性を高めます。

- バイナリファイルを読み取る`pg_read_binary_file()`関数を追加しました。(Dimitri Fontaine、Itagaki Takahiro)
- ファイル全体を読み取る`pg_read_file()`関数の単一パラメータ版を追加しました。(Dimitri Fontaine、Itagaki Takahiro)
- NULL値の処理を制御するために、3つのパラメータを持つ`array_to_string()`、`string_to_array()`の構文を追加しました。(Pavel Stehule)

E.15.3.7.1. オブジェクト情報関数

- `pg_describe_object()`関数を追加しました。(Alvaro Herrera)

この関数は、`pg_class` OID、オブジェクトOID、副オブジェクトIDに基づいてオブジェクトを説明する可読性が高い文

字列を入手するために使用されます。これを[pg_depend](#)の内容を解釈する補助として使用することができます。

- 組み込み演算子とその基になる関数についてのコメントを更新しました。(Tom Lane)

関連する演算子を介して使用されることを意図した関数についてその旨をコメント付けしました。

- `EXPLAIN`内および[pg_get_viewdef\(\)](#)のようなシステムカタログ関数内の識別子すべてを強制的に引用符付けする[quote_all_identifiers](#)変数を追加しました。(Robert Haas)

これにより、スキーマを引用符付けに関して異なる規則を持つツールや他のデータベースにエクスポートすることがより簡単になります。

- [information_schema.sequences](#)システムビューに列を追加しました。(Peter Eisentraut)

これまでもビューは存在していましたが、シーケンスパラメータに関する列は実装されていませんでした。

- [has_table_privilege\(\)](#)および関連する関数内で仮想ロール名として[public](#)が許されます。

これにより[public](#)権限に対する検査を行うことができます。

E.15.3.7.2. 関数およびトリガの作成

- ビューに対する[INSTEAD OF](#)トリガをサポートします。(Dean Rasheed)

この機能を更新可能なビューを完全に実装するために使用することができます。

E.15.3.8. サーバサイド言語

E.15.3.8.1. [PL/pgSQL](#)サーバサイド言語

- [FOREACH_IN_ARRAY](#)をPL/pgSQLに追加しました。(Pavel Stehule)

これは、配列値の要素を使用する反復についてのこれまでの方法より効率と可読性を向上します。

- パラメータを持たない[RAISE](#)を[RAISE ERROR](#)を捕捉できる同じ場所で捕捉できるようになりました。(Piyush Newe)

これまでのコードでは、実行中の例外ハンドラを含むブロックからこのエラーを発生させました。新しい動作は他のDBMS製品とより一貫性があります。

E.15.3.8.2. [PL/Perl](#)サーバサイド言語

- PL/Perl関数で一般的なレコード引数が許されます。(Andrew Dunstan)

PL/Perl関数を[record](#)型を受け付けるように宣言することができるようになりました。この動作は任意の名前付き複合型に対するものと同じです。

- PL/Perlの配列引数をPerl配列に変換します。(Alexey Klyukin, Alex Hunsaker)

文字列表現もまだ可能です。

- PL/Perlの複合型引数をPerlのハッシュに変換します。(Alexey Klyukin, Alex Hunsaker)

文字列表現もまだ可能です。

E.15.3.8.3. [PL/Python](#)サーバサイド言語

- PL/Pythonにテーブル関数のサポートを追加しました。(Jan Urbanski)

PL/Pythonが複数のOUTパラメータとレコードセットを返すことができるようになりました。

- PL/Pythonに検証関数を追加しました。(Jan Urbanski)

これによりPL/Python関数は関数生成時に構文検査を行うことができます。

- PL/Python内でSQL問い合わせに対する例外を許します。(Jan Urbanski)

これによりSQLが生成する例外エラーコードをPL/Python例外ブロックからアクセスすることができます。

- PL/Pythonに明示的な副トランザクションを追加しました。(Jan Urbanski)

- 文字列の引用符付けを行うPL/Python関数を追加しました。(Jan Urbanski)

これらの関数は[plpy.quote_ident](#)、[plpy.quote_literal](#)、[plpy.quote_nullable](#)です。

- PL/Pythonエラーにトレースバック情報を追加しました。(Jan Urbanski)
- `PLy_elog`を用いて反復子からのPL/Pythonエラーを報告します。(Jan Urbanski)
- Python 3での例外処理を修正しました。(Jan Urbanski)

これまではPython 3下の`plpy`では例外クラスを利用することができませんでした。

E.15.3.9. クライアントアプリケーション

- [createlang](#)および[droplang](#)を廃止予定とし、拡張コマンドを呼び出すだけのものにしました。(Tom Lane)

E.15.3.9.1. [psql](#)

- 現在の接続情報を示す`\conninfo` `psql`コマンドを追加しました。(David Christensen)
 - 関数定義を示す`\sf` `psql`コマンドを追加しました。(Pavel Stehule)
 - 言語を列挙する`\dl` `psql`コマンドを追加しました。(Fernando Ike)
 - `psql`の`\dn`(スキーマ列挙)コマンドに`s` ("システム") オプションを追加しました。(Tom Lane)
- `s`のない`\dn`ではシステムスキーマを表示しないようになりました。
- `psql`の`\e`および`\ef`コマンドが、エディタにおけるカーソル位置を設定するために使用される行番号を受け付けることができます。(Pavel Stehule)

これは`PSQL_EDITOR_LINENUMBER_ARG`環境変数にしたがってエディタに渡されます。

- `psql`がデフォルトでオペレーティングシステムのロケールからクライアント符号化方式を設定するようにしました。(Heikki Linnakangas)

これは`PGCLIENTENCODING`環境変数が設定されていない場合に起こります。

- `\d`が一意性インデックスと一意性制約を区別するようにしました。(Josh Kupersmidt)
- 9.0以降のサーバと接続している場合、`\dt+`が`pg_relation_size`ではなく`pg_table_size`を報告するようにしました。(Bernd Helmle)

これはテーブルサイズの測定にはより有用です。しかしこれまで同じ表示で報告された値と同じではないことに注意してください。

- タブ補完サポートを追加しました。(Itagaki Takahiro、Pavel Stehule、Andrey Popp、Christoph Berg、David Fetter、Josh Kupersmidt)

E.15.3.9.2. [pg_dump](#)

- すべての識別子を強制的に引用符付けする`--quote-all-identifiers`オプションを[pg_dump](#)および[pg_dumpall](#)に追加しました。(Robert Haas)

- `directory`形式を[pg_dump](#)に追加しました。(Joachim Wieland、Heikki Linnakangas)

これは内部的には`tar` `pg_dump`形式と似ています。

E.15.3.9.3. [pg_ctl](#)

- サーバが稼動していないと間違った報告をしないように[pg_ctl](#)を修正しました。(Bruce Momjian)

これまではサーバが稼動していても[pg_ctl](#)が認証させられなかった場合にこれが起こることがありました。

- `pg_ctl start`の"待機" (`-w`) オプションを改良しました。(Bruce Momjian、Tom Lane)

待機モードは非常に堅牢になりました。デフォルト以外のポート番号、デフォルト以外のUnixソケット位置、権限問題、古いpostmasterのロックファイルにより混乱することがなくなります。

- スタンバイサーバをプライマリに切り替える `promote` オプションを `pg_ctl` に追加しました。(Fujii Masao)

E.15.3.10. 開発ツール

E.15.3.10.1. `libpq`

- `PGCLIENTENCODING` 環境変数と同様に振舞う `client_encoding` `libpq` 接続オプションを追加しました。(Heikki Linnakangas)

`auto` という値はオペレーティングシステムのロケールに基づいてクライアント符号化方式を設定します。

- `libpq` ライブラリのバージョンを返す `PQlibVersion()` を追加しました。(Magnus Hagander)

`libpq` はサーババージョンを返す `PQserverVersion()` をすでに有していました。

- Unix ドメインソケット経由で接続している場合に、`libpq` データベースクライアントは新しい接続オプション `requirepeer` を使用してサーバプロセスのユーザ名を検査することができます。(Peter Eisentraut)

PostgreSQL ではすでにサーバに Unix ドメインソケット経由で接続するクライアントユーザ名を検査させることが可能でした。

- `libpq` に `PQping()` および `PQpingParams()` を追加しました。(Bruce Momjian、Tom Lane)

これらの関数により、新しくセッションを開こうとしなくても、サーバの状態を検出することができます。

E.15.3.10.2. `ECPG`

- `ECPG` は `WHERE CURRENT OF` 句の中であっても動的カーソル名を受け付けることができます。(Zoltan Boszormenyi)
- `ecpglib` が `double` 値をこれまでの 14 桁ではなく 15 桁の精度で書き込むようにしました。(Akira Kurosawa)

E.15.3.11. 構築オプション

- HP-UX C コンパイラで受け付けられる `+Olibmerrno` コンパイルオプションを使用します。(Ibrar Ahmed)

これは最近の HP プラットフォームで起こり得る数学ライブラリ呼び出し誤動作を防止します。

E.15.3.11.1. `Makefiles`

- 並行 `make` のサポートを改良しました。(Peter Eisentraut)

これによりコンパイルがより高速になります。また `make -k` がより一貫性をもって動作します。

- GNU `make` 3.80 以降が必要です。(Peter Eisentraut)

並行 `make` の改良のためこれが必要です。

- `make maintainer-check` ターゲットを追加しました。(Peter Eisentraut)

このターゲットは、構築やリグレッションテストでは不適切な、様々なソースコード検査を行います。現時点では、`duplicate_oids`、SGML の構文およびタブの検査、NLS 構文検査があります。

- `contrib` 内の `make check` をサポートします。(Peter Eisentraut)

従来は `make installcheck` のみが動作していましたが、一時的なインストール内での試験をサポートするようになりました。最上位レベルの `make check-world` ターゲットに、この方法による `contrib` の試験が含まれるようになりました。

E.15.3.11.2. `Windows`

- `Windows` において、`pg_ctl` がサービスを自動または手動として登録することができます。(Quan Zongliang)

- `Windows` において `クラッシュダンプ` の収集サポートを追加しました。(Craig Ringer、Magnus Hagander)

`minidumps` がデバッグ用ではない `Windows` バイナリで生成でき、標準のデバッグツールで解析できるようになりました。

- MinGW64 コンパイラを使用する構築を有効にしました。(Andrew Dunstan)

これにより、Windowsプラットフォーム以外であってもクロスコンパイルすることにより64ビットWindowsバイナリを構築することができます。

E.15.3.12. ソースコード

- GUC変数の代入フック用のAPIを改良しました。(Tom Lane)

これまでの代入フック機能が、検査フックと代入フックに分離されました。前者は失敗しても構いませんが、後者は失敗してはなりません。この変更は、独自のGUCパラメータを定義する追加モジュールに影響を与えます。

- イベントの待機をサポートするためにソースコードにラッチを加えました。(Heikki Linnakangas)
- データ変更の権限検査ロジックをまとめました。(KaiGai Kohei)
- 一貫性を目的に、なくなった`get_object_oid()`関数を追加しました。(Robert Haas)
- 競合するキーワードを除去することにより、[追加モジュールのコンパイル](#)でのC++コンパイラを使用する機能を改良しました。(Tom Lane)

- DragonFly BSDに対するサポートを追加しました。(Rumko)
- バックエンドで使用するために`quote_literal_cstr()`を公開しました。(Robert Haas)
- デフォルトの符号化方式で[リグレッション試験](#)を実行します。(Peter Eisentraut)

これまでのリグレッション試験では常に`SQL_ASCII`符号化方式で実行していました。

- `cvs2cl`と`pgcvslog`を置き換える`src/tools/git_changelog`を追加しました。(Robert Haas、Tom Lane)
- `git-external-diff`スクリプトを`src/tools`に追加しました。(Bruce Momjian)

これは`git`からコンテキストdiffを生成するために使用されます。

- Clangを用いた構築のサポートを改良しました。(Peter Eisentraut)

E.15.3.12.1. サーバフック

- 権限を編さするソースコードフックを追加しました。(Robert Haas、Stephen Frost)
- セキュリティフレームワークで使用するための`post-object-creation`関数フックを追加しました。(KaiGai Kohei)
- クライアント認証フックを追加しました。(KaiGai Kohei)

E.15.3.13. Contrib

- 新しい[拡張](#)機構を用いて`contrib`モジュールと手続き言語をインストールするように変更しました。(Tom Lane、Dimitri Fontaine)

- [contrib/file_fdw](#)外部データラッパを追加しました。(Shigeru Hanada)

この外部データラッパを使用する外部テーブルは、`COPY`と非常によく似た方法で通常のファイルを読み取ることができます。

- [contrib/pg_trgm](#)および[contrib/btree_gist](#)に近傍検索サポートを追加しました。(Teodor Sigaev)
- 等価以外の検索に対する[contrib/btree_gist](#)サポートを追加しました。(Jeff Davis)
- マルチバイト文字を扱うように[contrib/fuzzystrmatch](#)の`levenshtein()`関数を修正しました。(Alexander Korotkov)
- [contrib/sslinfo](#)に`ssl_cipher()`および`ssl_version()`関数を追加しました。(Robert Haas)
- インデックス付けされた空配列に関して一貫した結果になるように[contrib/intarray](#)および[contrib/hstore](#)を修正しました。(Tom Lane)

これまでは、インデックスを使用する空配列の問い合わせは、シーケンシャルスキャンを使用した場合の結果と異なる結果が返る可能性があります。

- [contrib/intarray](#)が多次元の配列に対して適切に動作するようにしました。(Tom Lane)

- [contrib/intarray](#)において、実際にはNULL値が無い場合でもNULLが存在するとしてエラーになることを防止しました。(Tom Lane)
- [contrib/intarray](#)において、含有演算子が空配列を考慮するように修正しました。(Tom Lane)
空配列は他の任意の配列に含まれるものと、正しくみなされるようになりました。
- [contrib/xml2](#)において、`xslt_process()`で取り扱うことができる、`parameter=value`の組み合わせ数についての独断的な制限を除去しました。(Pavel Stehule)
以前の制限は10でした。
- [contrib/pageinspect](#)において、32ビット値としてinfomaskを返すようにheap_page_itemを修正しました。(Alvaro Herrera)
これは混乱を招く負の値が返されることを防ぎます。基になる値は16ビット符号なし整数です。

E.15.3.13.1. セキュリティ

- SELinuxと権限検査をやりとりする[contrib/sepgsql](#)を追加しました。(KaiGai Kohei)
これは新しい[SECURITY LABEL](#)機能を使用します。
- [auth_delay](#) contribモジュールを追加しました。(KaiGai Kohei)
これは、認証失敗を返す前にサーバを休止します。ブルートフォースパスワード攻撃をより困難にするために設計されました。
- [dummy_seclabel](#) contribモジュールを追加しました。(KaiGai Kohei)
これは権限に関するリグレーション試験で使用されます。

E.15.3.13.2. 性能

- [contrib/pg_trgm](#)にLIKEおよびILIKE インデックス検索のサポートを追加しました。(Alexander Korotkov)
- [contrib/fuzzystrmatch](#)に小さな距離を最適化する`levenshtein_less_equal()`関数を追加しました。(Alexander Korotkov)
- [contrib/seg](#)列のインデックス検索の性能を向上しました。(Alexander Korotkov)
- 多くのリレーションを持つデータベースにおいての[pg_upgrade](#)の性能を向上しました。(Bruce Momjian)
- 文単位の遅延を報告するフラグを[contrib/pgbench](#)に追加しました。(Florian Pflug)

E.15.3.13.3. Fsyncの試験

- `src/tools/test_fsync`を[contrib/pg_test_fsync](#)に移動しました。(Bruce Momjian、Tom Lane)
- [contrib/pg_test_fsync](#)にO_DIRECTサポートを追加しました。(Bruce Momjian)
これは、[wal_sync_method](#)によるO_DIRECTの使用に対応します。
- [contrib/pg_test_fsync](#)に新しい試験を追加しました。(Bruce Momjian)

E.15.3.14. 文書

- [ECPG](#)の文書を大きく改良しました。(Satoshi Nagayasu)
- 大量の校正と文書の改良を行いました。(Thom Brown、Josh Kupersmidt、Susanne Ebrecht)
- [exit_on_error](#)についての文書を追加しました。(Robert Haas)
このパラメータは任意のエラーでセッションを終了させます。
- [pg_options_to_table\(\)](#)についての文書を追加しました。(Josh Berkus)
この関数は可読性の高い書式でテーブル格納オプションを表示します。

[\(compositeval\).*](#)構文を使用することですべての複合型のフィールドにアクセスできることを文書化しました。(Peter Eisentraut)

- [translate\(\)](#)が、対応するto文字を持たないfrom内の文字を除去することを文書化しました。(Josh Kopershmidt)
- CREATE CONSTRAINT TRIGGERと[CREATE TRIGGER](#)の文書をまとめました。(Alvaro Herrera)
- [permission](#)と[upgrade](#)の文書を中心に集めました。(Bruce Momjian)
- Solaris10についての[カーネルチューニング文書](#)を追加しました。(Josh Berkus)

これまではSolaris9のみについてカーネルチューニングが文書化されていました。

- HISTORY ファイル内の非ASCII文字を一貫して扱います。(Peter Eisentraut)

HISTORY ファイルは英語で記述されていますが、貢献者の名前を非ASCII文字で扱う必要がありました。文字セットに関して前提を置くことなく合理的に判読できるように書き直されました。

[前のページ](#)

リリース9.1.1

[ホーム](#)
[上に戻る](#)

[次のページ](#)

リリース9.0.13

E.16. リリース9.0.13

リリース日: 2013-04-04

このリリースは9.0.12に対し、各種不具合を修正したものです。9.0メジャーリリースにおける新機能については、[項E.29](#)を参照してください。

E.16.1. バージョン9.0.13への移行

9.0.Xからの移行ではダンプ/リストアは不要です。

しかし、このリリースではGiSTインデックスの管理についていくつかの不具合を修正しています。このアップデートをインストールした後、以下に記述する条件の1つ以上に該当するGiSTインデックスすべてについて、`REINDEX`を実行することを推奨します。

また、9.0.6よりも前のバージョンからアップグレードする場合は、9.0.6のリリースノートを参照してください。

E.16.2. 変更点

- サーバのコマンド行スイッチについての安全でない解析を修正しました。(Mitsumasa Kondo, Kyotaro Horiguchi)

"-"で始まるデータベース名を含む接続要求に細工をすることで、例えその要求が最終的に拒絶された場合でも、サーバのデータディレクトリ内のファイルを損傷させる、あるいは破壊することが可能でした。(CVE-2013-1899)

- OpenSSLの乱数の状態を、postmasterのそれぞれの子プロセスでリセットするようにしました。(Marko Kreen)

`contrib/pgcrypto`関数によって生成された乱数が、他のデータベースユーザによって比較的推定しやすいという問題が、これにより回避されます。この危険はpostmasterが`ssl = on`で構成されている時にのみ問題となりますが、ほとんどの接続ではSSL暗号化を利用していません。(CVE-2013-1900)

- GiSTインデックスで、"fuzzy"な幾何比較を使うのが適切でないときは、使わないようにしました。(Alexander Korotkov)

幾何データ型のコアでは"fuzzy"な同値性を利用して比較を行ないますが、`gist_box_same`では正確な比較を行う必要があり、そうしなければ、それを利用するGiSTインデックスは一貫しないものになります。このアップデートをインストールした後、`box`、`polygon`、`circle`、`point`型の列のすべてのGiSTインデックスについて`REINDEX`を実行してください。これらはいずれも`gist_box_same`を使っています。

- 可変長のデータ型、つまり`text`、`bytea`、`bit`、`numeric`型の列で`contrib/btree_gist`を使うGiSTインデックスについて誤った範囲の結合と不正なロジックを修正しました。(Tom Lane)

この誤りにより、存在するキーが検索で見つからないような一貫性のないインデックスが作成されたり、インデックスが無駄に大きくなることがありました。このアップデートをインストールした後、そのようなインデックスについて`REINDEX`を実行することを推奨します。

- 複数列についてのインデックスで、GiSTページを分割するコードの不具合を修正しました。(Tom Lane)

この誤りにより、存在するキーが検索で見つからないような一貫性のないインデックスが作られることや、検索が非常に非効率になるインデックスが作られることがありました。このアップデートをインストールした後、複数列についてのGiSTインデックスについて`REINDEX`を実行することを推奨します。

- `gist_point_consistent`で曖昧さを一貫性のある方法で処理するようにしました。(Alexander Korotkov)

`point`型の列のGiSTインデックスについてのインデックススキャンは、シーケンシャルスキャンとは異なる結果になる場合があります。これは、`gist_point_consistent`と演算子のコードで、厳密な比較をするか、曖昧な比較をするかについて不一致があったからです。

- WAL再生におけるバッファのリークを修正しました。(Heikki Linnakangas)

この不具合により、再生中に"incorrect local pin count"のエラーが発生し、リカバリができなくなる場合があります。

- `DELETE RETURNING`での競合条件を修正しました。(Tom Lane)

特定の条件下で、`DELETE RETURNING`は現在のプロセスがもはや確保していない共有バッファからデータをフェッチすることがありました。その間に他のプロセスがそのバッファを変更した場合、ゴミの値が`RETURNING`の出力となったり、あるいはクラッシュすることもありました。

- 正規表現のコンパイルで無限ループになる可能性があるのを修正しました。(Tom Lane, Don Porter)
- 正規表現のコンパイル時にNULLポインタから参照先を取得する可能性があるのを修正しました。(Tom Lane)
- `to_char()` で、それが適切な場合、ASCIIのみの大文字変換規則を利用するように修正しました。(Tom Lane)

これにより、ロケールに依存すべきでないいくつかのテンプレートのパターンにおける誤った動作が修正されます。トルコ語のロケールで"`i`"と"`ı`"が正しく処理されていませんでした。

- 1999-12-31 24:00:00というタイムスタンプが理由もなく拒絶されていたのを修正しました。(Tom Lane)
- 単一のトランザクションが`UNLISTEN`を実行し、その後`LISTEN`を実行した場合のロジックの誤りを修正しました。(Tom Lane)

この場合、セッションは通知イベントを監視すべきですが、実際には全く監視しない状態になっていました。

- "picksplit doesn't support secondary split"という不必要なログメッセージを削除しました。(Josh Hansen, Tom Lane)

このメッセージは何らかのコードを期待して追加されたように思われますが、そのコードは書かれませんでした。また、GiSTによる二次的分割のデフォルト処理は実際には非常に良いので、そのコードは今後も書かれないでしょう。そのため、エンドユーザにブツブツ言うのをやめることにしました。

- セッションの最後のいくつかのコミットとアボートの件数を統計情報コレクタに送信するのに失敗する可能性があるのを修正しました。(Tom Lane)
- PL/Perlの`spi_prepare()`関数のメモリリークを修正しました。(Alex Hunsaker, Tom Lane)
- `pg_dumpall`でデータベース名に"`=`"が含まれるものを正しく処理できるよう修正しました。(Heikki Linnakangas)
- `pg_dump`が不正な接続文字列が渡された時にクラッシュしないようにしました。(Heikki Linnakangas)
- `pg_dump`と`pg_upgrade`で不正なインデックスを無視するようにしました。(Michael Paquier, Bruce Momjian)

不正なインデックスをダンプするとリストア時に問題が発生する可能性があります。例えば、ユニーク条件を満たさないテーブルのデータに対してユニーク条件を強制したためにインデックス作成に失敗したような場合です。また、インデックスの作成が実際にはまだ進行中の場合もありますが、このときはDDLによる更新がまだコミットされていないと考えるのが適当でしょうから、`pg_dump`がダンプする必要はありません。`pg_upgrade`は不正なインデックスでエラーを起こす代わりに、それをスキップするようになります。

- `contrib/pg_trgm`の`similarity()`関数が、トライグラムのない文字列について0を返すよう修正しました。(Tom Lane)

以前は内部でのゼロ割りによりNaNを返していました。

- 時間帯データファイルを`tzdata`リリース2013bに更新しました。チリ、ハイチ、モロッコ、パラグアイ、およびロシアのいくつかの地域での夏時間の変更が含まれます。また、数多くの場所での歴史的な地域データの修正が含まれます。

また、時間帯の略称について、ロシアおよび他の地域での以下の最近の変更を反映しました。

`CHOT`、`GET`、`IRKT`、`KGT`、`KRAT`、`MAGT`、`MAWT`、`MSK`、`NOVT`、`OMST`、`TKT`、`VLAT`、`WST`、`YAKT`、`YEKT`については、それらの現在の意味に従います。`VOLT` (Europe/Volgograd (ヨーロッパ/ヴォルゴグラード))および`MIST` (Antarctica/Macquarie (南極/マッコリー島))をデフォルトの短縮名リストに追加しました。

E.17. リリース9.0.12

リリース日: 2013-02-07

このリリースは9.0.11に対し、各種不具合を修正したものです。9.0メジャーリリースにおける新機能については、[項E.29](#)を参照してください。

E.17.1. バージョン9.0.12への移行

9.0.Xからの移行ではダンプ/リストアは不要です。

しかし、9.0.6よりも前のバージョンからアップグレードする場合は、9.0.6のリリースノートを参照してください。

E.17.2. 変更点

- SQLから`enum_recv`を実行できないようにしました。(Tom Lane)

この関数は誤った宣言をされており、単純なSQLコマンドでサーバをクラッシュさせることが可能でした。実際のところ、サーバのメモリの内容を検査するために攻撃者がこれを利用することが可能になっていました。この問題を(Secunia SVCRP経由で)報告してくれたSumit Soniに感謝します。(CVE-2013-0255)

- WALの再生時にデータベースが一貫性のある状態に到達したことを検知するときの複数の問題を修正しました。(Fujii Masao, Heikki Linnakangas, Simon Riggs, Andres Freund)

- リレーションのファイルを切り詰めるときに、最小復旧地点を更新するようにしました。(Heikki Linnakangas)

データが捨てられた後は、タイムライン上のそれより前の地点で復旧を止めることは、もはや安全ではありません。

- ホットスタンバイモードで問い合わせの取り消しに失敗するのを修正しました。(Noah Misch, Simon Riggs)

ホットスタンバイでの競合する問い合わせを取り消す要求が失敗することがあり、これらの問い合わせで一貫性のないデータを見ることがありました。

- 副SELECTの結果から添字付けあるいは列選択できるようSQLの文法を修正しました。(Tom Lane)

- 負荷が大きい時の自動バキュームの切り詰めの性能問題を修正しました。(Jan Wieck)

テーブルの最後にある空のページを切り詰めるときに排他的ロックが必要ですが、競合するロック要求があるとき、自動バキュームのプログラムはそれに失敗してテーブルロックを解放するようになっていました。負荷が重いと切り詰めが全く起こらず、テーブルが大きくなるということが容易に起こり得ました。部分的な切り詰めを実行、ロックを解放、その後ロックの再取得と処理の継続を試みるように修正しました。この修正により、競合するロック要求がある場合に、自動バキュームがロックを解放するまでの平均時間が大きく短縮されます。

- `pg_tablespace`をスキャンするときに競合問題が発生しないようにしました。(Stephen Frost, Tom Lane)

`pg_tablespace`のエントリに複数の同時更新があるとき、`CREATE DATABASE`および`DROP DATABASE`が正しく動作しない場合があります。

- `DROP OWNED`がデータベース全体あるいはテーブル空間を削除しようとしなかったようにしました。(Álvaro Herrera)

安全のため、これらのオブジェクトは削除するのではなく、所有権を変更すべきです。

- `vacuum_freeze_table_age`の実装の誤りを修正しました。(Andres Freund)

`vacuum_freeze_min_age`の値よりも多くのトランザクションを実行したインストール環境では、この誤りにより自動バキュームが部分テーブルスキャンを実行できなくなり、その結果、必ずテーブル全体のスキャンが発生します。

- `RowExpr`あるいは`XmlExpr`が2度、解析された時に、誤った動作をしないようにしました。(Andres Freund, Tom Lane)

この誤りは`CREATE TABLE LIKE INCLUDING INDEXES`などの状況においてユーザに明らかになることがありました。

- ハッシュテーブルのサイズ計算で整数のオーバーフローが起きないようにする仕組みを改善しました。(Jeff Davis)

- `to_date()` が範囲外の日付を拒絶するようにしました。(Hitoshi Harada)

- Windowsで非ASCIIのプロンプト文字列が、確実に正しいコードページに変換されるようにしました。(Alexander Law, Noah Misch)

このバグはpsqlおよび他のいくつかのクライアントプログラムに影響がありました。

- データベースに接続していないとき、psqlの`\?`コマンドがクラッシュする可能性があったのを修正しました。(Meng Qingzhong)
- pg_upgradeが不正なインデックスを安全に扱うよう修正しました。(Bruce Momjian)
- libpqのPQprintTuplesでバッファを1バイト行き過ぎてしまう問題を修正しました。(Xi Wang)

この古い関数はPostgreSQL本体によっては使われていませんが、クライアントのプログラムでは今でも使われているかもしれません。

- ecpglibで翻訳されたメッセージを適切に使うようにしました。(Chen Huajun)
- MSVC上でecpg_compatおよびpgtypesのライブラリを適切にインストールするようにしました。(Jiang Guiqing)
- `isinf()` をシステムが提供していなければ、我々のバージョンをlibecpggに含めるようにしました。(Jiang Guiqing)
- libedit/libreadlineからの誤ったエクスポートに騙されないよう、configureによる提供関数の確認の順序を調整しました。(Christoph Berg)
- Windowsのビルド番号が時とともに確実に大きくなるようにしました。(Magnus Hagander)
- Windows用にクロスコンパイルしたときに、pgxsが正しく`.exe`の拡張子の付いた実行プログラムを生成するようにしました。(Zoltan Borszormenyi)
- 新しい時間帯の略号`FET`を追加しました。(Tom Lane)

これは一部の東ヨーロッパの時間帯で使われています。

[前のページ](#)

リリース9.0.13

[ホーム](#)
[上に戻る](#)

[次のページ](#)

リリース9.0.11

E.18. リリース9.0.11

リリース日: 2012-12-06

このリリースは9.0.10に対し、各種不具合を修正したものです。9.0メジャーリリースにおける新機能については、[項E.29](#)を参照してください。

E.18.1. バージョン9.0.11への移行

9.0.Xからの移行ではダンプ/リストアは不要です。

しかし9.0.6より前のバージョンからアップグレードする場合は、9.0.6のリリースノートを参照してください。

E.18.2. 変更点

- `CREATE INDEX CONCURRENTLY`に関連する複数の不具合を修正しました。(Andres Freund, Tom Lane)

`CREATE INDEX CONCURRENTLY`がインデックスの`pg_index`の行の状態を変更するときに、その場にある更新を使うように修正しました。これにより、同時実行中のセッションが対象のインデックスを更新できず、それにより`CONCURRENTLY`で作成されるインデックスを破損させることになる競合条件を防ぐことができます。

また、`CREATE INDEX CONCURRENTLY`コマンドのエラーの結果として作られる不正なインデックスについて、他の様々な操作が確実に無視するように修正しました。この中で最も重要なのは`VACUUM`です。なぜなら、この不正なインデックスを修正あるいは削除するための修正の処理が実行される前に、自動バキュームが実行されることがあり得るからです。

- WAL再生中のバッファロックを修正しました。(Tom Lane)

複数ページにまたがるWALレコードを再生する場合のバッファロックについて、WAL再生のプログラムに完全でない部分がありました。これにより、ホットスタンバイの問い合わせが一時的に一貫性のない状態になることがあり、その結果、誤った結果や不正なエラーを起こすことがありました。

- GINインデックスについてのWAL生成のロジックの誤りを修正しました。(Tom Lane)

分裂ページのエラーが発生すると、これによりインデックスが破損することがありました。

- ホットスタンバイのサーバを通常の実行にプロモートする時、起動プロセスの仮想XIDロックを正しく削除するようにしました。(Simon Riggs)

この誤りにより、`CREATE INDEX CONCURRENTLY`などいくつかの操作をこの後で実行できない場合があります。

- スタンバイモードにおいて偽物の"out-of-sequence timeline ID"エラーが出ないようにしました。(Heikki Linnakangas)

- postmasterがシャットダウンのシグナルを受け取った後は、新しい子プロセスを起動しないようにしました。(Tom Lane)

この誤りにより、シャットダウンが必要以上に長く時間がかかる、あるいはユーザによる追加の操作がなければ終了しない、ということがありました。

- メモリが不足した時でも、内部ハッシュテーブルが破損しないようにしました。(Hitoshi Harada)

- 外部結合上の厳格でない等価な句についての計画作成を修正しました。(Tom Lane)

厳格でない構造を他のものと等号で結ぶ句について、プランナが誤った制約を導出する場合があります。例えば、`WHERE COALESCE(foo, 0) = 0`で`foo`が外部結合のNULLになるかもしれない側のもののときです。

- プランナについて、等価なクラスから排他制約を証明する能力を向上しました。(Tom Lane)

- ハッシュされた副計画の部分行のマッチングについて、型をまたがる場合を正しく処理できるよう修正しました。(Tom Lane)

これは`WHERE (a, b) NOT IN (SELECT x, y FROM ...)`のような複数の列についての`NOT IN`の副計画で、例えば`b`と`y`がそれぞれ`int4`型と`int8`型である、というような場合に影響します。この処理の誤りにより、そこで使われているデータ型に

よって、誤った結果が得られたり、あるいはクラッシュする、ということが起きていました。

- `AFTER ROW UPDATE/DELETE` トリガーで古いタプルを再フェッチする時、バッファロックを取得するようにしました。(Andres Freund)

この誤りにより、非常に特異な状況において、外部キーを強制するトリガーの事前チェックのロジックに対して誤ったデータを渡すことがありました。その結果として、クラッシュを起こしたり、あるいはトリガーを実行するかどうかについて誤った判断をすることがありました。

- `ALTER COLUMN TYPE` について、継承されたチェック制約を正しく処理できるように修正しました。(Pavan Deolasee)

これは8.4より前のリリースでは正しく動作していました。この修正により8.4およびそれ以降のリリースでも正しく動作するようになりました。

- `REASSIGN OWNED` について、表領域のGRANTが正しく処理できるように修正しました。(Álvaro Herrera)

- ビューのシステム列についての `pg_attribute` の誤ったエントリを無視するようにしました。(Tom Lane)

ビューにはシステム列はありません。しかし、表をビューに変換するとき、そのようなエントリを取り除くことを忘れていました。これは9.3およびそれ以降のリリースでは適切に修正されますが、それ以前のブランチにおいても、誤って変換されたビューが存在しないようにする必要があります。

- ルールの出力について、`INSERT INTO table DEFAULT VALUES` を正しくダンプできるよう修正しました。(Tom Lane)

- 問い合わせが非常に多くの `UNION/INTERSECT/EXCEPT` を含むときでもスタックオーバーフローが発生しないようにしました。(Tom Lane)

- もっとも小さな整数値を-1で割り算した時に、プラットフォームによってはエラーとなるのを修正しました。(Xi Wang, Tom Lane)

- 日付文字列の解析で、文字列の終わりを越えてアクセスすることがあるのを修正しました。(Hitoshi Harada)

- チェックポイント中にXID周回が発生し、かつ `wal_level` が `hot_standby` である場合にXIDエポックを進めるのに失敗するのを修正しました。(Tom Lane, Andres Freund)

この誤りはPostgreSQL自体には特に影響はありませんでしたが、TXID値が後戻りしているように見えるため、`txid_current()` およびそれに関連した関数に依存するアプリケーションにとっては問題でした。

- Unixドメインソケットのパス名がプラットフォーム依存の最大長を超えていた場合に、理解可能なエラーメッセージを生成するようにしました。(Tom Lane, Andrew Dunstan)

以前は"Non-recoverable failure in name resolution"(名前解決で解決不能なエラーが発生しました)といった非常に不親切なエラーを起こしていました。

- 複合列の値をクライアントに送るときのメモリリークを修正しました。(Tom Lane)

- `postmaster.pid` ファイルの読み込みについて、`pg_ctl` をより堅牢にしました。(Heikki Linnakangas)

競合条件、およびファイルデスクリプタのリークを解決しました。

- `psql` において、不正にエンコードされたデータが入力され、かつ `client_encoding` の設定がSJISのようにクライアントでしか利用できない文字セットである場合にクラッシュする場合があるのを修正しました。(Jiang Guiqing)

- `pg_dump` が `tar` 出力形式で生成する `restore.sql` スクリプトの不具合を修正しました。(Tom Lane)

このスクリプトはテーブル名が大文字を含む場合に全く動作しませんでした。また、このスクリプトで通常のCOPYモードだけでなく `--inserts` モードの場合もデータをリストアできるようにしました。

- `pg_restore` でPOSIX準拠の `tar` ファイルを処理できるよう修正しました。(Brian Weaver, Tom Lane)

`pg_dump` の `tar` 出力モードの元々のプログラムは、POSIX標準に完全には準拠していないファイルを生成していました。この問題はバージョン9.3で修正されます。このパッチは9.3より前のブランチを修正するもので、正しくないフォーマットと正しいフォーマットの両方を処理できるようにします。これにより、9.3がリリースされた時に互換性の問題を防ぐことができます。

- `pg_resetxlog` について、データディレクトリへの相対パスが与えられた時に、`postmaster.pid` の場所を正しく特定できるよう修正しました。(Tom Lane)

この誤りにより、データディレクトリを使用している動作中の `postmaster` があることに `pg_resetxlog` が気が付かないこ

とがありました。

- libpqの`lo_import()`関数および`lo_export()`関数が、ファイルI/Oのエラーを適切に報告するよう修正しました。(Tom Lane)
- ecpgについて、ネストした構造体のポインタ変数の処理を修正しました。(Muhammad Usama)
- ecpgの`ecpg_get_data`関数について、配列を正しく処理するよう修正しました。(Michael Meskes)
- contrib/pageinspectのBツリーページ検査関数で、ページの検査中はバッファロックを取得するようにしました。(Tom Lane)
- AIX上でのロードモジュールのビルドについてpgxsのサポートを修正しました。(Tom Lane)
元々のソースツリーの外側でモジュールをビルドしてもAIXでは正しく動作しませんでした。
- 時間帯データファイルをtzdataリリース2012jに更新しました。キューバ、イスラエル、ヨルダン、リビア、パレスチナ、西サモア、およびブラジルの一部での夏時間の変更が含まれます。

[前のページ](#)

リリース9.0.12

[ホーム](#)
[上に戻る](#)

[次のページ](#)

リリース9.0.10

E.19. リリース9.0.10

リリース日: 2012-09-24

このリリースは9.0.9に対し、各種不具合を修正したものです。9.0メジャーリリースにおける新機能については[項E.29](#)を参照してください。

E.19.1. バージョン9.0.10への移行

9.0.Xからの移行ではダンプ/リストアは不要です。

しかし9.0.6より前のバージョンからアップグレードする場合は9.0.6のリリースノートを参照してください。

E.19.2. 変更点

- プランナによるエグゼキュータのパラメータの代入について修正しました。また、CTEプランノードについてエグゼキュータの再スキャンの条件を修正しました。(Tom Lane)

これらの間違いにより、同じWITHの副問い合わせを複数回スキャンする問い合わせについて、間違った結果を返すことがありました。

- GiSTインデックスにおけるページ分割の決定を改善しました。(Alexander Korotkov, Robert Haas, Tom Lane)

複数列のGiSTインデックスで、この間違いにより、予期しない膨張が起きることがありました。

- 権限が取り消されても、その権限を引き続き保持している場合は、取り消しが連鎖しないようにしました。(Tom Lane)

あるロールxからグラントオプションを取り消したが、xが他の誰かから付与されたことによってそのオプションを引き続き保持している、という場合は、ロールyがxによって付与されたその権限を再帰的に取り消すべきではありません。

- ホットスタンバイ構成に誤りがあるときのエラーメッセージを改善しました。(Gurjeet Singh)

- PL/Perlが使われている時のSIGFPEの処理について修正しました。(Andres Freund)

PerlはプロセスのSIGFPEハンドラをSIG_IGNにリセットするため、後になってクラッシュが起きる場合があります。PL/Perlを初期化した後で、Postgresの通常のシグナルハンドラをリストアするようにしました。

- PL/Perlの再帰関数が実行時に再定義される場合に、PL/Perlがクラッシュしないようにしました。(Tom Lane)

- PL/Perlで正しく最適化されない場合があるのを回避しました。(Tom Lane)

一部のLinuxディストリビューションでは、誤ったバージョンのpthread.hがインストールされており、PL/Perlのコンパイルされたコードが誤ったものになります。この結果、PL/Perlの関数がエラーを発生させる他の関数を呼び出すと、クラッシュすることがありました。

- pg_upgradeのWindowsにおける行末の処理を修正しました。(Andrew Dunstan)

以前は、pg_upgradeは関数の実体などの場所で、復帰文字(CR)を追加あるいは削除することがありました。

- Windows上でpg_upgradeが生成するスクリプトで、パス名の区切り文字にバックスラッシュを使うようにしました。(Andrew Dunstan)

- 時間帯データファイルをtzdataリリース2012fに更新しました。フィジーでの夏時間の変更が含まれます。

E.20. リリース9.0.9

リリース日: 2012-08-17

このリリースは9.0.8に対し、各種の不具合を修正したものです。9.0メジャーリリースにおける新機能については[項E.29](#)を参照してください。

E.20.1. バージョン9.0.9への移行

9.0.Xからの移行ではダンプ/リストアは不要です。

しかし9.0.6より前のバージョンからアップグレードする場合は9.0.6のリリースノートを参照してください。

E.20.2. 変更点

- XML実体参照を介した外部ファイル/URLへのアクセスを防止します。(Noah Misch, Tom Lane)

`xml_parse()` は、XML値で参照されるDTDと実体参照を解決するために必要な外部ファイルやURLの入手を試みます。これにより、権限を持たないデータベースユーザがデータベースサーバの権限でデータを入手することができてしまいました。この外部データは直接ユーザには返されませんが、データが有効なXMLとして解析できなかった場合その一部がエラーメッセージの中に漏洩することがあり得ます。またどのような場合であっても、ファイルが存在するかどうか検査できるだけでも攻撃者にとって有用になり得ます。(CVE-2012-3489)

- `contrib/xml2`の`xsslt_process()`を介した外部ファイル/URLへのアクセスを防止します。(Peter Eisentraut)

`libxslt`はスタイルシートコマンドを通してファイルとURLの両方を読み書きする機能を提供します。これにより、権限を持たないデータベースユーザがデータベースサーバの権限でデータを読み書きすることができてしまいました。`libxslt`のセキュリティオプションを適切に使用することでこれを無効にします。(CVE-2012-3488)

また`xsslt_process()`の外部ファイル/URLからドキュメントとスタイルシートを取り込む機能を取り除きました。これは文書化された"機能"でしたが、推奨されないものと長くみなされていました。CVE-2012-3489に対する修正のためこの能力が壊れましたが、その修正を行うことは止め、単に取り除きました。

- 早すぎるbtreeインデックスページの回収を防止します。(Noah Misch)

読み取り専用トランザクションによるXIDの割り当てを省略できるようにした時、読み取り専用トランザクションがまだ参照している時に削除されたbtreeページを回収されてしまう可能性をもたらしました。これは間違ったインデックス検索結果という結果になります。時間に関する必要条件のため、こうしたエラーが実際に発生する可能性は非常に低いものですが、修正しなければなりません。

- 新しく作成された、またはリセットされたシーケンスに関するクラッシュに対する安全性を修正しました。(Tom Lane)

`ALTER SEQUENCE`が新しく作成されたシーケンスまたはリセットされたシーケンスに対して実行された場合、その後`nextval()`呼び出しが一度だけ行われ、そして、サーバがクラッシュすると、WAL再生はシーケンスを`nextval()`が行われていないように見える状態にリストアします。このため次の`nextval()`呼び出しにより最初のシーケンス値が再び返されることになります。シリアル列のシーケンス作成には`ALTER SEQUENCE OWNED BY`処理が含まれていますので、特にこれは`serial`列で現れます。

- ホットスタンバイではない時に正確なエポックを報告するように`txid_current()`を修正しました。(Heikki Linnakangas)

これは以前のマイナーリリースで導入されたリグレッションを修正します。

- マスタのトランザクションが多くの副トランザクションを持つ時のホットスタンバイの起動処理の不具合を修正しました。(Andres Freund)

この間違いは"out-of-order XID insertion in KnownAssignedXids"と報告される失敗がもたらします。

- `pg_start_backup()`の後確実に`backup_label`ファイルを`fsync`します。(Dave Kerr)
- WAL送信処理プロセスにおけるタイムアウト処理を修正しました。(Tom Lane)

WAL送信処理バックグラウンドプロセスではSIGALRMハンドラの確立を忘れていました。この結果タイムアウトが発生

する何らかの境界条件で永遠に待機してしまいます。

- 9.1からのバックパッチはfsync要求キューの圧縮を向上しました。(Robert Haas)

チェックポイント期間の性能を向上します。9.1の変更は安全に過去のバージョンにパッチを充てられると判断できる程度に実地試験が行われたようです。

- ディスク容量不足等I/O問題をよりよく処理するようにLISTEN/NOTIFYを修正しました。(Tom Lane)

書き込みの失敗の後、以降すべてのNOTIFYメッセージの送信試行は"Could not read from file "pg_notify/nnnn" at offset nnnnn: Success"のようなメッセージで失敗しました。

- 自動バキュームは直接ブロックされたプロセスによってのみ自動キャンセルさせることができます。(Tom Lane)

元々のコードでは一部の場合に一貫性がない動作を許していました。具体的にはdeadlock_timeout猶予期間より短時間で自動バキュームはキャンセルされました。

- 自動バキュームのキャンセルについてのログ処理を改良しました。(Robert Haas)

- サーバ起動後一番最初のログローテーションの間log_truncate_on_rotationが動作するようにログコレクタを修正しました。(Tom Lane)

- 入れ子状の集合操作 (UNION/INTERSECT/EXCEPT) に付与されるWITHを修正しました。(Tom Lane)

- 副問い合わせへの行全体の参照が余計なGROUP BYやORDER BY列を含まないことを確実にしました。(Tom Lane)

- CREATE TABLE時のCHECK制約およびインデックス定義における行全体の参照の複製を許しません。(Tom Lane)

この状況はLIKEまたはINHERITSを持つCREATE TABLEで発生することがあり得ます。複製される行全体の変数は、新しいテーブルではなく元のテーブルの行型で正しくラベル付けされませんでした。LIKEでは、後で行型がよく分岐しますので、このような状況を拒絶する方が合理的のように思います。INHERITSでは理想的には、親のテーブルの行型への暗黙的な変換によって許可すべきですが、安全に以前のバージョンにパッチを充てるにはより多くの作業が必要です。

- ARRAY(SELECT ...) 副問い合わせにおけるメモリリークを修正しました。(Heikki Linnakangas、Tom Lane)

- 正規表現からの共通接頭辞の抽出を修正しました。(Tom Lane)

このコードでは^(foo)?barなどの修飾された括弧付きの副式で混乱してしまったりすることがあり得ました。こうしたパターンの検索に対して不正確なインデックス最適化をもたらしました。

- interval定数における、符号付きhh:mm、hh:mm:ssフィールドの解析に関する不具合を修正しました。(Amit Kapila、Tom Lane)

- PL/Pythonにおいて、PythonのUnicode文字列からサーバ符号化方式に変換する時にPythonの関数ではなくPostgresの符号化方式変換関数を使用します。(Jan Urbanski)

これは、特にPythonがPostgresがサポートするすべての符号化方式をサポートしていない場合の何らかの境界値問題を防止します。注意すべき機能上の変更は、サーバ符号化方式がSQL_ASCIIの場合、文字列のUTF-8表現を得ることです。これまでは文字列内のASCII以外の文字はエラーになりました。

- PL/PythonにおいてPostgreSQL符号化方式とPython符号化方式との間の対応付けを修正しました。(Jan Urbanski)

- contrib/xml2のxslt_process() においてエラーを適切に報告します。(Tom Lane)

- 時間帯データファイルをtzdataリリース2012eに更新しました。モロッコおよびトケラウにおける夏時間の変更が含まれます。

E.21. リリース9.0.8

リリース日: 2012-06-04

このリリースは9.0.7に対し、各種の不具合を修正したものです。9.0メジャーリリースにおける新機能については[項E.29](#)を参照してください。

E.21.1. バージョン9.0.8への移行

9.0.Xからの移行ではダンプ/リストアは不要です。

しかし9.0.6より前のバージョンからアップグレードする場合は9.0.6のリリースノートを参照してください。

E.21.2. 変更点

- contrib/pgcryptoのDES `crypt()` 関数における不正なパスワード変換を修正しました。(Solar Designer)

パスワード文字列に`0x80`というバイト値が含まれる場合、パスワードの残りが無視され、見かけよりも非常に弱いパスワードになってしまいました。この修正によって、文字列の残りが適切にDESハッシュ内に含まれるようになります。したがって、この不具合の影響を受ける、格納済みのパスワード値はすべて一致なくなりますので、格納済みの値を更新しなければなりません。(CVE-2012-2143)

- 手続き言語の呼び出しハンドラにおいてSECURITY DEFINERおよびSET属性を無視します。(Tom Lane)

こうした属性を呼び出しハンドラに与えるとサーバがクラッシュするかもしれません。(CVE-2012-2655)

- 数字による時間帯オフセットによりUTCから16時間まで離れたtimestampの入力を行うことができます。(Tom Lane)

一部の歴史的な時間帯はこれまでの上限であった15時間よりも離れたオフセットを持ちます。このためリロード時にダンプされたデータ値が拒絶される結果になることがありました。

- 与えられた時刻が現在の時間帯の最後のDST遷移時刻と正確に一致する場合をうまく処理するようにタイムスタンプ変換を修正しました。(Tom Lane)

長い間これは見落とされていました。ほとんどのDSTを使用する時間帯では、将来のDST変更が無限に続くものと仮定していましたので、これまで気付きませんでした。

- textからnameへのキャストおよびcharからnameへのキャストがマルチバイト符号化方式で正しく文字列の切り詰めを行うように修正しました。(Karl Schnaitter)

- to_tsquery()のメモリコピーにおける不具合を修正しました。(Heikki Linnakangas)

- txid_current()がホットスタンバイにおいて実行された場合に、正確なエポックを確実に報告します。(Simon Riggs)

- 副問い合わせ内の外部プレースホルダに関するプランナの扱いを修正しました。(Tom Lane)

この不具合は、上位の問い合わせの外部結合の内NULLを含む方から由来する変数を参照する副SELECTに影響します。9.1では、この不具合によって影響を受ける問い合わせは"ERROR: Upper-level PlaceholderVar found where not expected"というエラーで失敗します。しかし9.0と8.4では、副問い合わせに渡る値がNULLであるべき時にNULLになりませんので、警告なく間違っているかもしれない回答が得られます。

- pg_attributeが非常に大きい場合の低速なセッション起動を修正しました。(Tom Lane)

pg_attributeがshared_buffersの1/4を超える場合、セッション起動時に時々必要となるキャッシュ再構築コードが同期スキャンロジックを開始し、通常より長い期間かかるようになりました。この問題は、特に多くの新しいセッションが一度に開始する場合に深刻です。

- 適切な頻度で発生する問い合わせキャンセルに対する連続的なスキャン検査を確実に行います。(Merlin Moncurie)

有効なタプルを含まない連続するページを多くを巡るスキャンは、その間、中断に応答しません。

- WindowsのPGSemaphoreLock()の実装は、戻る前にImmediateInterruptOKを確実に消去します。(Tom Lane)

この見落としは、同じ問い合わせの中で後で受け取った問い合わせキャンセル中断が、安全ではない時に受け付けられ、予期しない、悪化した結果をもたらす可能性があったことを意味します。

- ビューまたはルールを出力する時に行全体の変数を安全に表示します。(Abbas Butt, Tom Lane)

曖昧な名前を含む（つまりその名前がテーブル名か問い合わせの列名かいずれかになる）境界条件では、曖昧な方法で出力し、ビューまたはルールがダンプとリストアの間で別の解釈がされる危険性がありました。操作を行わないキャストを付与することで曖昧な状況を防ぎます。

- 無効な符号化方式に対応するヌルマーカ文字列を正しく扱うように`COPY FROM`を修正しました。(Tom Lane)

`E'¥¥0'`などのヌルマーカ文字列は動作しなければなりませんし、以前は動作していました。この状況は8.4で壊れました。

- 自動バキュームのワーカプロセスは確実にスタック長の検査を適切に行います。(Heikki Linnakangas)

これまでは、自動`ANALYZE`によって呼び出される関数の中の無限再帰がワーカプロセスをクラッシュさせる可能性がありました。

- 高負荷時にログの干渉性が失われるようにログコレクタを修正しました。(Andrew Dunstan)

これまでは、コレクタは非常に忙しい場合、大規模メッセージの再構成に失敗する可能性がありました。

- `SIGHUP`を受信した後にファイルのローテーションを確実に再開するようにログコレクタを修正しました。(Tom Lane)

- `GIN`インデックスに対する`WAL`再生ロジックを修正し、インデックスがその後に削除された場合でも失敗しないようにしました。(Tom Lane)

- `PL/pgSQL`の`RETURN NEXT`コマンドにおけるメモリリークを修正しました。(Joe Conway)

- 対象が関数の第一変数であった場合の`PL/pgSQL`の`GET DIAGNOSTICS`コマンドを修正しました。(Tom Lane)

- `psql`の拡張表示（`¥x`）モードにおいてメモリの最後にアクセスできない可能性を修正しました。(Peter Eisentraut)

- データベースに多くのオブジェクトが含まれている場合の`pg_dump`の複数の性能問題を修正しました。(Jeff Janes, Tom Lane)

データベースに多くのスキーマが含まれる場合、多くのオブジェクトが依存関係のあるループにある場合、所有されたシーケンスが多く存在する場合に、`pg_dump`が非常に低速になる可能性がありました。

- デフォルト以外のテーブル空間内に格納されたデータベースがクラスタのデフォルトのテーブル空間内のテーブルを含む状況に対する`pg_upgrade`を修正しました。(Bruce Momjian)

- `ecpg`において、まれに起こるメモリリークと`sqlca_t`構造体の後の1バイトの上書きを修正しました。(Peter Eisentraut)

- エラー時に一時データベース接続をリークしないように、`contrib/dblink`の`dblink_exec()`を修正しました。(Tom Lane)

- エラーメッセージ内で正しい接続名を報告するように`contrib/dblink`を修正しました。(Kyotaro Horiguchi)

- 多くのラージオブジェクトを削除する時に複数のトランザクションを使用するように`contrib/vacuumlo`を修正しました。(Tim Lewis, Robert Haas, Tom Lane)

この変更により、多くのオブジェクトを削除しなければならない時に`max_locks_per_transaction`を超えることを防ぎます。この動作は新しい`-1`（制限）オプションで調整することができます。

- 時間帯データファイルを`tzdata`リリース2012cに更新しました。南極、アルメリア、チリ、キューバ、フォークランド諸島、ガザ、ハイチ、ヘブロン、モロッコ、シリア、トケラウ諸島における夏時間の変更、カナダに関する歴史的な変更が含まれます。

E.22. リリース9.0.7

リリース日: 2012-02-27

このリリースは9.0.6に対し、各種の不具合を修正したものです。9.0メジャーリリースにおける新機能については[項E.29](#)を参照してください。

E.22.1. バージョン9.0.7への移行

9.0.Xからの移行ではダンプ/リストアは不要です。

しかし9.0.6より前のバージョンからアップグレードする場合は9.0.6のリリースノートを参照してください。

E.22.2. 変更点

- `CREATE TRIGGER`においてトリガ関数に対する実行権限が必要になりました。(Robert Haas)

この検査が無かったため、別のユーザが自身が所有するテーブルにその関数をインストールすることで、偽造した入力データでトリガ関数を実行することができました。 `SECURITY DEFINER`が付いたトリガ関数でのみ重大です。その他のトリガ関数ではとにかくテーブル所有者として実行されるためです。(CVE-2012-0866)

- SSL証明書内のコモンネームにおける独断的な長さ制限を取り除きました。(Heikki Linnakangas)

`libpq`とサーバの両方ともSSL証明書から取り出したコモンネームを32バイトで切り詰めていました。通常想定外の検証エラー以上のことを引き起こしますが、ある証明書の所有者が他者になりすますことができるという、多少怪しいシナリオがあります。被害を受けるにはコモンネームが正確に32バイト長でなければならない、また、攻撃者は信頼されたCAに対してコモンネームが接頭辞としてその文字列を持つ証明書を発行するように説得しなければなりません。またサーバになりすますには、さらにクライアントからの接続を中継するための攻撃が必要です。(CVE-2012-0867)

- `pg_dump`のコメントに記述された名前の中の改行を空白に変換します。(Robert Haas)

`pg_dump`は、その出力スクリプトにおけるSQLコメント内で発行されるオブジェクト名のサニタイズに関して注意を払っていませんでした。改行を含む名前は少なくともそのスクリプトを構文的に不正にさせます。悪意をもって組み立てられたオブジェクト名によって、スクリプトがリロードする時にSQLインジェクションの危険性があり得ました。(CVE-2012-0868)

- バキューム処理と同時に挿入を行った場合の**btree**インデックス破損を修正しました。(Tom Lane)

挿入によって発生するインデックスページ分割によって、同時実行中の**VACUUM**が削除すべきインデックス項目を削除し損なうことが時々発生することがありました。対応するテーブル行が削除された後、対応先が無いインデックス項目によってエラー ("could not read block N in file ..."など)、最悪は、解放されたテーブル位置に再挿入された関係がない行にちなんだ、警告のない間違った問い合わせ結果が引き起こされます。この不具合は8.2から存在していましたが、あまり頻発に発生しませんでしたので、これまで究明されませんでした。使用中のデータベースで発生していたのではと疑わしければ、対象のインデックスを再インデックス付けすることで修正されます。

- WAL再生中の一時的な共有バッファのゼロ化を修正しました。(Tom Lane)

再生ロジックでは時々共有バッファをゼロ化し再度書き込んでいました。このためその内容は一時的に無効になります。ホットスタンバイモードではこの結果、平行して実行される問い合わせがゴミデータを参照することになり得ました。いろいろな兆候が現れますが、もっとも良くある兆候は"invalid memory alloc request size"です。

- ホットスタンバイのクラッシュの後、`postmaster`の再起動を試みるように修正しました。(Tom Lane)

ホットスタンバイモードで操作している時に任意のバックエンドプロセスがクラッシュした場合、ロジックエラーのため、クラスタを再起動させようとはせず**postmaster**を終了させました。

- 最近更新された行によって所有される**TOAST**値に対する**CLUSTER/VACUUM FULL**の取り扱いを修正しました。(Tom Lane)

この見落としにより、これらのコマンドのいずれかを実行している時に、**TOAST**テーブルのインデックスに対して"duplicate key value violates unique constraint"エラーが報告されることがあり得ました。

- テーブル所有者を変更する時に、テーブル単位の権限だけではなく列単位の権限も更新します。(Tom Lane)

この失敗は、事前に付与された列権限が古い所有者により付与されたものとして表示されることを意味します。つまり新しい所有者またはスーパーユーザであってもテーブル所有者を追跡できなくなった権限を取り除くことができないことを意味します。

- `REASSIGN OWNED`において外部データラッパおよび外部サーバをサポートします。(Alvaro Herrera)

これらのオブジェクトの所有権を変更しなければならなかった場合に、このコマンドは"unexpected classid"エラーで失敗しました。

- `ALTER USER/DATABASE SET`における一部の設定で存在しない値を許可します。(Heikki Linakangas)

`default_text_search_config`、`default_tablespace`、`temp_tablespaces`を未知の名前に設定することができます。これらが実際に使用されている別のデータベースでは既知であるかもしれませんが、またテーブル空間の場合はまだテーブル空間が作成されていないかもしれないからです。過去`search_path`においても同じ問題がありましたが、これらの設定も同様に動作するようにしました。

- コミット後のテーブルファイルの削除に問題があった場合のクラッシュを防止します。(Tom Lane)

テーブル削除は、そのトランザクションがコミットした後にのみ背後のディスク上のファイルを削除しなければなりません。(例えばファイルの権限の誤設定のため)失敗した場合、トランザクションをアボートするには遅すぎますので、コードは単に警告メッセージを出力し継続するものと考えられます。リリース8.4においてこのロジックが壊れ、こうした状況でPANICが発生し、再起動できないデータベースとなりました。

- `DROP TABLESPACE`のWAL再生中に発生したエラーから修復します。(Tom Lane)

再生はテーブル空間ディレクトリの削除を試みますが、(対象ディレクトリの所有者や権限が不正であったなど)失敗する可能性は多々あります。これまでは再生コードはパニックし、手作業を介さなければ再起動できないデータベースになりました。ディレクトリの削除に失敗したところで多少ディスク領域が無駄に使われるだけですので、問題を記録し、継続する方がより良いと考えました。

- ホットスタンバイのAccessExclusiveLocksのログ記録における競合問題を修正しました。(Simon Riggs)

ロックが"transaction zero"によって保持されていると記録されることが時々ありました。少なくともスレーブサーバでアサーションエラーが発生することが分かっていますが、もっと深刻な問題が起こる可能性があります。

- WAL再生中にOIDカウンタを、たとえ周回していたとしても、正しく追跡します。(Tom Lane)

これまでは、OIDカウンタはシステムが再生モードを抜けるまで高値のままになりました。実際にはほとんどnilという結果になりますが、マスタに昇格されるスタンバイサーバでは、値が必要になってから、OIDカウンタを合理的な値まで進めるために長時間かかる可能性があります。

- クラッシュリカバリの最初の、誤解を招く"consistent recovery state reached"ログメッセージの発行を抑制します。(Heikki Linnakangas)

- `pg_stat_replication.replay_location`の初期値を修正しました。(Fujii Masao)

これまでは、少なくとも1つのWALレコードが再生されるまで、表示される値は間違っていました。

- *が付いた正規表現の後方参照を修正しました。(Tom Lane)

コードでは、正確な文字列一致を強制せずに、実質後方参照シンボルで参照されるパターン副式を満たす任意の文字列を受け付けました。

同様の問題はまだ、直接量指定子のサブジェクトとならない、より大きく量化された式に埋め込まれた後方参照でも残っています。こちらは将来のリリースのPostgreSQLで対応予定です。

- `inet/cidr`値の処理に最近入ったメモリリークを修正しました。(Heikki Linnakangas)

2011年12月のPostgreSQLリリース内のパッチによって、これらの操作にメモリリークが発生しました。これらの列に対するbtreeインデックス等で重大な問題になる可能性があります。

- SQL言語関数内の`CREATE TABLE AS/SELECT INTO`の後に対応先がなくなったポインタを修正しました。(Tom Lane)

ほとんどの場合、これはアサートが有効な構築におけるアサーション失敗という結果になるだけですが、もっと悪い結果になる可能性があります。

- Windowsのsysloggerにおけるファイルハンドルの二重クローズを防ぎます。(MauMau)

通常はこのエラーは表面化しませんが、Windowsのデバッグ版を実行している場合は例外が引き起こります。

- plpgsqlにおけるI/O変換関連のメモリリークを修正しました。(Andres Freund、Jan Urbanski、Tom Lane)
現在の関数が終わるまで、特定の操作がメモリリークしました。

- 継承されたテーブル列に対するpg_dumpの取り扱いを改良しました。(Tom Lane)

pg_dumpは、子の列が親の列と異なるデフォルト式を持つという状況を間違っ**て**扱いました。デフォルトがテキストとして親のデフォルトと同一であるが、実際は同一ではない場合（例えば、スキーマ検索パスの違いのため）、異なるものとして認識せず、そのため、ダンプしリストアした後、子は親のデフォルトを継承することができました。子の列がNOT NULLであり親がそうではなかった場合も、微妙に間違っ**て**リストアされました。

- INSERT形式のテーブルデータに対するpg_restoreの直接データベースにリストアするモードを修正しました。(Tom Lane)

他の問題に対する修正における見落としの結果、2011年9月または12月付けのリリースに含まれるpg_restoreでは、--insertsまたは--column-insertsオプションを付けて作成されたアーカイブファイルを直接データベースにリストアすることができませんでした。アーカイブファイル自体には失敗はなく、テキストモード出力では問題ありませんでした。

- pg_upgradeがregclass列を含むテーブルを処理できるようにしました。(Bruce Momjian)

pg_upgradeがpg_class OIDの保護に注意を払うようになりましたので、この制限を続ける理由はなくなりました。

- SSLクライアント証明書ファイルを検索するときに、libpqがENOTDIRエラーを無視するようにしました。(Magnus Hagander)

これによりユーザのホームディレクトリが/dev/nullのようなものに設定されていたとしても、証明書なしですが、SSL接続を確立できます。

- ecpgのSQLDA領域におけるフィールド整列問題を修正しました。(Zoltan Boszormenyi)

- ecpgのDEALLOCATE文においてATオプションを可能にしました。(Michael Meskes)

これをサポートする基盤は少し前から存在しましたが、見落としのため、エラー検査でこの状況を拒絶していました。

- ecpgにおいてvarchar構造体を定義するときに変数名を使用しません。(Michael Meskes)

- contrib/auto_explainのJSON出力モードが有効なJSONを生成するように修正しました。(Andrew Dunstan)

この出力では、本来括弧を使用しなければならない最上位で、角括弧を使用していました。

- contrib/intarrayのint[] & int[] 演算子のエラーを修正しました。(Guillaume Lelarge)

2つの入力配列が共通して持つ最小の整数が1であり、どちらかの配列により小さな値があった場合、1が結果から間違っ**て**省かれました。

- contrib/pgcryptoのencrypt_iv() およびdecrypt_iv() の誤判定を修正しました。(Marko Kreen)

これらの関数は特定の種類の無効入力エラーの通知に失敗し、不正な入力に対してランダムなゴミの値を代わりに返しました。

- contrib/test_parserにおける1バイトのバッファオーバーランを修正しました。(Paul Guyot)

コードでは必要より1バイト多く読み取ろうとし、境界条件ではクラッシュします。contrib/test_parserは単なるサンプルコードであり、これ自体はセキュリティ問題ではありませんが、例のコードとしては良くありません。

- ARMにおいて可能ならばスピンのロックに__sync_lock_test_and_set()を使用します。(Martin Pitt)

この関数は、以前の廃止予定でARMv6以降では使用できなくなったSWPB命令の使用を置き換えるものです。最近のARMボードでも古いコードは明白な方法で失敗することはありませんが、単に同時アクセスのインターロックを行わず、マルチプロセス操作において奇妙な失敗をもたらすと報告されています。

- 受け付け可能なバージョンのgccを用いて構築する場合に-fexcess-precision=standardオプションを使用します。(Andrew Dunstan)

最近のバージョンのgccは独創的な結果を生成するというさまざまなシナリオを防ぎます。

- FreeBSDにおいてスレッド化されたPythonを使用できるようにしました。(Chris Rees)

configureスクリプトはこれまで、この組み合わせは動作しないと前提していました。しかしFreeBSDではこの問題が修

正されましたので、エラー検査を取り除きました。

[前のページ](#)

リリース9.0.8

[ホーム](#)
[上に戻る](#)

[次のページ](#)

リリース9.0.6

E.23. リリース9.0.6

リリース日: 2011-12-05

このリリースは9.0.5に対し、各種の不具合を修正したものです。9.0メジャーリリースにおける新機能については[項E.29](#)を参照してください。

E.23.1. バージョン9.0.6への移行

9.0.Xからの移行ではダンプ/リストアは不要です。

しかし、`information_schema.referential_constraints`ビュー定義において長期に渡って潜在したエラーが見つかりました。このビューからの正確な結果に依存している場合、後述の変更点の最初で説明する通りその定義を置き換えなければなりません。

また9.0.4より前のバージョンからアップグレードする場合は、9.0.4のリリースノートを参照してください。

E.23.2. 変更点

- `information_schema.referential_constraints`ビューの不具合を修正しました。(Tom Lane)

このビューでは、外部キーと依存するプライマリキーまたは一意キー制約との対応付けについて十分な注意を行っていませんでした。このため、外部キー制約がまったく表示されない、何回も表示される、あるいは、実際の依存する制約とは異なる制約に依存するものと言い張るといった結果になることがありました。

このビュー定義はinitdbによってインストールされますので、ただアップグレードしただけでは問題は解消されません。既存のインストールでこの問題を解消させなければならない場合、（スーパーユーザとして）`information_schema`スキーマを削除して、`SHAREDIR/information_schema.sql`を元に再作成することができます。（`SHAREDIR`の場所が分からない場合は`pg_config --sharedir`を実行してください。）これは修正対象のデータベースそれぞれで繰り返さなければなりません。

- スカラを返す関数の出力を結合するUPDATEまたはDELETE実行時にクラッシュする可能性を修正しました。(Tom Lane)

クラッシュは対象の行が同時に更新された場合にのみ起こり得ました。このためこの問題は間欠的にのみ表面化しました。

- GINインデックス更新に関するWALレコードの不正確な再生を修正しました。(Tom Lane)

このためクラッシュ後またはホットスタンバイサーバにおいて、インデックス項目の検索に過渡的に失敗する結果になり得ました。しかし、この問題はインデックスに対する次のVACUUMで修復されます。

- `CREATE TABLE dest AS SELECT * FROM src`または`INSERT INTO dest SELECT * FROM src`時のTOAST関連のデータの破損を修正しました。(Tom Lane)

テーブルが`ALTER TABLE ADD COLUMN`によって変更された場合、データを逐語的に別のテーブルにコピーしようとした時に特定の境界条件で破損した結果になることがありました。この問題は、8.4以降において、この構文においてのみ明確に発生し得たものですが、同じ不具合を引き起こす他のコードパスがある場合に備えて、より過去のバージョンにおいてもパッチを適用しました。

- ホットスタンバイの起動時に失敗する可能性を修正しました。(Simon Riggs)
- 初期スナップショットが不完全である場合のホットスタンバイの起動を高速にしました。(Simon Riggs)
- 無効なsyscache項目からTOASTテーブルにアクセスする時の競合条件を修正しました。(Tom Lane)

典型的な兆候は、"missing chunk number 0 for toast value NNNNN in pg_toast_2619"のような一時的なエラーです。ここで挙げられるTOASTテーブルは常にシステムカタログに属します。

- パラメータのデフォルト式で使用する項目について関数の依存関係を追跡します。(Tom Lane)

これまでは、参照されるオブジェクトが、参照する関数を削除または変更することなく、削除されることがあり得ました。このため関数が使用される時に誤動作することになりました。単にこの更新をインストールするだけでは依存関係が消失した項目が修正されないことに注意してください。修正するためには、その後こうした関数それぞれにCREATE

OR REPLACE を行う必要があります。デフォルト値が組み込み以外のオブジェクトに依存する関数があれば、これを行うことを勧めます。

- 複数のOUTパラメータを持つ、集合を返すSQL関数をインライン化することができます。(Tom Lane)
- 結合除去において遅延された一意性制約を信用しません。(Tom Lane、Marti Raudsepp)

遅延された一意性制約はトランザクション内部で保持されないかもしれません。このため、間違った問い合わせ結果をもたらすかもしれないとみなします。

- DatumGetInetP() が1バイトヘッダを持つinetデータを展開するようにしました。さらに展開を行わない、新しいマクロDatumGetInetPP() を追加しました。(Heikki Linnakangas)

この変更がコアコードに影響することはありませんが、DatumGetInetP() が通常の規約の通りに展開したデータを生成することを想定しているアドオンのコードにおけるクラッシュを防止できるかもしれません。

- money 型の入出力におけるロケールのサポートを改良しました。(Tom Lane)

すべての標準 `lc_monetary` 書式付けオプションをサポートしていないという点以外にも、入力関数と出力関数に一貫性がなかったため、ダンプされたmoney値では再度読み取ることができないロケールがありました。

- `transform_null_equals` が `CASE foo WHEN NULL ...` 式に影響を与えないようにしました。(Heikki Linnakangas)

`transform_null_equals` は、CASE 構文によって内部的に生成された等価な検査ではなく、直接ユーザによって書かれた `foo = NULL` に影響を与えることのみを前提としています。

- 自己参照外部キーをより良くサポートするために、外部キートリガの生成順序を変更しました。(Tom Lane)

カスケードしている外部キーが自身のテーブルを参照する場合、1行の更新が1つのイベントでON UPDATE トリガとCHECK トリガの両方を発行します。ON UPDATE トリガを最初に実行しなければなりません。そうしないと、CHECK は最終的ではない行の状態を検査することになり、不適切なエラーが発生する可能性があります。しかしこれらのトリガの発行順序はその名前によって決まります。トリガは"RI_ConstraintTrigger_NNNN"という規約に従った名前で自動生成されますので、一般的には作成順序によって決まります。適切な修正にはこの規約の変更が必要です。9.2で変更する予定ですが、既存のリリースで変更することには危険があるように思われます。このため、このパッチはトリガの作成順序を変更するだけです。この種のエラーが起きたユーザは、トリガを正しい順序にするために、外部キー制約を削除し再作成しなければなりません。

- バッファ割当率を追跡する際の浮動小数アンダーフローを防止します。(Greg Matthews)

これ自体は害がないものですが、特定のプラットフォームではこれにより煩わしいカーネルログメッセージが現れます。

- Windowsにおける子プロセス起動時設定ファイルの名前と行番号を保持します。(Tom Lane)

これまでこれらはpg_settingsビューで正しく表示されませんでした。

- ecpgのSQLDA領域における不正確なフィールドの整列を修正しました。(Zoltan Boszormenyi)

- psqlのコマンド履歴内で空行を含むコマンドを保持します。(Robert Haas)

これまでの動作では、例えば文字列リテラル内の空行が削除された場合に問題が起きました。

- テーブルの行型など自動生成された型の間でユーザが定義したキャストをダンプするようにpg_dumpを修正しました。(Tom Lane)

- pg_upgradeについて様々な修正を行いました。(Bruce Momjian)

排他制約の正確な取り扱い、Windowsにおける失敗の防止、8.4データベースにおける対応しないTOASTテーブル名に関する警告の抑制。

- PL/Perlの構築時に好みのバージョンのxsubppを使用します。オペレーティングシステムのメインコピーである必要はありません。(David Wheeler、Alex Hunsaker)

- contrib/dict_intおよびcontrib/dict_xsyn内の間違ったコードを修正しました。(Tom Lane)

一部の関数で、palloc() で返されるメモリはゼロ埋めされていることが保証されていると間違った前提をしていました。

- contrib/unaccentの設定ファイル解析において様々なエラーを修正しました。(Tom Lane)

- pgstatindex() において問い合わせキャンセル割込みを適切に受け付けます。(Robert Haas)

- Mac OS X起動スクリプトにおける間違ったログファイル名の引用符付けを修正しました。(Sidar Lopez)
- VPATH構築ですべてのサーバヘッダファイルが適切にインストールされることを確実にしました。(Peter Eisentraut)
- 冗長エラーメッセージ内で報告されるファイル名を短くしました。(Peter Eisentraut)

通常の構築では常にCファイルの名前だけがエラーメッセージ呼び出しの中で報告されました。しかしこれまでVPATH構築では絶対パス名が報告されました。

- 中央アメリカ用のWindowsの時間帯名の解釈を修正しました。(Tom Lane)

通常中央アメリカではまったく夏時間が認められませんので、"Central America Standard Time"をCST6CDTではなくCST6に対応付けしました。

- 時間帯データファイルをtzdata release 2011nに更新しました。ブラジル、キューバ、フィジィ、パレスチナ、ロシア、サモアにおける夏時間規則の変更、アラスカ、イギリス領東アフリカの歴史的な修正が含まれています。

[前のページ](#)

リリース9.0.7

[ホーム](#)
[上に戻る](#)

[次のページ](#)

リリース9.0.5

E.24. リリース9.0.5

リリース日: 2011-09-26

このリリースは9.0.4に対し、各種の不具合を修正したものです。9.0メジャーリリースにおける新機能については[項E.29](#)を参照してください。

E.24.1. バージョン9.0.5への移行

9.0.Xからの移行ではダンプ/リストアは不要です。

しかし9.0.4より前のバージョンからアップグレードする場合は、9.0.4のリリースノートを参照してください。

E.24.2. 変更

- システムカタログに対する`VACUUM FULL`または`CLUSTER`後のカタログキャッシュの無効化を修正しました。(Tom Lane)

システムカタログ行の位置が他の場所に変わった場合に、同時実行のサーバプロセスから認識できない場合があり、その行を更新しようとした時にカタログが破損する可能性があります。最悪の場合、テーブル全体が完全に喪失するほど悪い結果になります。

- `sinval`リセット処理期間の操作順を修正し、システムカタログ内で`TOAST` OIDが確実に予約されるようにしました。(Tom Lane)

この間違いによりシステムカタログに対する`VACUUM FULL`または`CLUSTER`後に一時的な失敗が発生する可能性があります。

- HOT更新されたか疑わしいタブルのインデックス処理における不具合を修正しました。(Tom Lane)

この不具合のために、システムカタログの再インデックス処理の後インデックスが破損する可能性があります。ユーザインデックスには影響しないはずです。

- GiSTインデックスページ分割処理における複数の不具合を修正しました。(Heikki Linnakangas)

発生する可能性は小さいのですが、インデックスの破損を招く可能性があります。

- `tsvector_concat()` におけるバッファオーバーランの可能性を修正しました。(Tom Lane)

この関数は必要なメモリを少なく見積もり、その結果サーバクラッシュを招く可能性があります。

- `xml_recv`において"standalone"パラメータ処理時のクラッシュを修正しました。(Tom Lane)

- `pg_options_to_table`を値を持たないオプションに対してNULLを返すようにしました。(Tom Lane)

これまではこのような場合にサーバクラッシュしてしまいました。

- `ANALYZE`およびSJIS-2004符号化変換においてメモリの最後を超えてアクセスする可能性を修正しました。(Noah Misch)

これは非常に可能性が低いサーバクラッシュ状況を修正します。

- NULL入力から`pg_stat_reset_shared()`を保護しました。(Magnus Hagander)

- リカバリの競合デッドロックがサブトランザクション内で検知された場合に失敗する可能性を修正しました。(Tom Lane)

- ホットスタンバイにおいて、btreeインデックスページ回収時の偽の競合を防止します。(Noah Misch、Simon Riggs)

- リカバリが終わっても動きつづけるWAL受信処理を停止します。(Heikki Linnakangas)

これまでのpostmasterはこうした状況ではパニックになりました。しかしこの状況は実際には正当なものでした。

- `relcache`初期ファイルの無効化における競合状態を修正しました。(Tom Lane)

新しいバックエンドプロセスが無効な初期ファイルを読み取ろうとするが、データが無効であることを通知するinvalメッセージを取り損なう期間がありました。この結果カタログアクセスにおいて、起動処理の後の"could not read block 0 in file ..."といった奇妙な失敗が起こります。

- GiSTインデックススキャン終了時のメモリリークを修正しました。(Tom Lane)

すでに多くの行を持つテーブルに対して新たなGiSTを基にした排他制約を作成する際の検証など、多くの分割GiSTインデックススキャンを行うコマンドが一時的にこのリークの原因となる大容量のメモリを求めることがあり得ました。

- 入力コマンド文字列に対する符号化方式の変換が必要かつLISTENが有効である時のメモリリークを修正しました。(Tom Lane)
- 保持可能カーソルとplpgsqlのRETURN NEXTコマンドをサポートするタプルストアにおける(メモリ膨張をもたらすかもしれない)不正なメモリ計上を修正しました。(Tom Lane)
- BEFOREおよびAFTERトリガの両方が存在する場合のトリガのWHEN条件を修正しました。(Tom Lane)

同一の更新に対してBEFORE ROWトリガが発行されていた場合、AFTER ROW UPDATEのWHEN条件の評価がクラッシュする可能性があります。

- 大規模かつ非可逆なビットマップを構築する時の性能問題を修正しました。(Tom Lane)
- 一意列の結合選択性推定を修正しました。(Tom Lane)

結合結果のサイズ推定を悪化させる可能性がある、間違っただランナの発見的手法を修正します。

- 副選択対象リスト内でのみ出現する、入れ子状のPlaceholderVar式を修正しました。(Tom Lane)

この間違いの結果、外側の結合の出力がNULLとして不正に現れます。

- プランナは、空の親テーブルが実際に空であるという仮定をすることができます。(Tom Lane)

通常、計画作成のために空テーブルは何らかの最小サイズを持つものと仮定されます。しかしこの発見的手法は、よく永遠に空になる、継承階層の親テーブルでは良いどころか不都合であるようです。

- 入れ子状のEXISTS問い合わせを適切に最適化できます。(Tom Lane)
- 配列作成、経路作成関数のパッド用バイトを確実にゼロにするように修正しました。(Tom Lane)
- inner-indexscan副計画内のResultノードの開閉を取り扱うようEXPLAINを修正しました。(Tom Lane)

この見落としのよくある兆候は"bogus varno"エラーです。

- indexedcol IS NULL条件のB-Tree前処理を修正しました。(Dean Rasheed)

同じインデックス列に別の種類のB-Treeインデックス付け可能な条件が組み合わされている場合、こうした条件は満たされません。この状態は9.0.0以降で不正に処理されており、存在しないはずの問い合わせ出力をもたらしました。

- WAL再生を壊すgcc 4.5.0の不具合を回避します。(Tom Lane)

サーバクラッシュ後にコミットされたトランザクションの損失をもたらす可能性があります。

- ビュー内のVALUESについてのダンプ不具合を修正しました。(Tom Lane)
- シーケンスに対するSELECT FOR UPDATE/SHAREを許しません。(Tom Lane)

この操作は想定通りに動作せず、また、失敗をもたらすことがあり得ました。

- つねにpg_class.reltuples/relpagesを更新するようにVACUUMを修正しました。(Tom Lane)

自動バキュームによる、いつテーブルをバキュームするかについての決定をだんだんと悪化させる、いくつかの状況を修正します。

- ハッシュテーブルのサイズ計算時の整数オーバーフローから保護します。(Tom Lane)
- CLUSTERがすでに削除されたTOASTデータへアクセスしようとする状況を修正しました。(Tom Lane)
- 初期認証トランザクション中の予期しないタイムアウト失敗を修正しました。(Tom Lane)

- "peer" 認証に関する証明書制御メッセージの使用における移植性不具合を修正しました。(Tom Lane)
- 複数の往復が必要な場合のSSPIログインを修正しました。(Ahmed Shinwari、Magnus Hagander)
この問題の典型的な兆候はSSPIログイン期間の"The function requested is not supported"エラーです。
- 独自変数クラスの変数を新しく `postgresql.conf` に追加する時の失敗を修正しました。(Tom Lane)
- `pg_hba.conf` に `hostssl` が含まれているがSSLが無効である場合にエラーを発生します。(Tom Lane)
以前の警告なくこうした行を無視するという動作より、この方がよりユーザフレンドリであると判断しました。
- `DROP OWNED BY` がシーケンス上のデフォルト権限を削除しようとした時の失敗を修正しました。(Shigeru Hanada)
- `pg_srand48` シード初期化における記述ミスを修正しました。(Andres Freund)
このため、提供されたシードの全ビットを使用すると失敗する可能性があります。この関数はほとんどのプラットフォームで使用されていません (`srandom` を持たないプラットフォームのみです) ので、想定よりもランダム性が小さいシードによるセキュリティ露見の可能性はとにかく最小だったと思われます。
- `LIMIT` と `OFFSET` 値の合計が 2^{63} を超えた場合の整数オーバーフローを防止します。(Heikki Linnakangas)
- `generate_series()` の `int4` 版と `int8` 版にオーバーフロー検査を追加しました。(Robert Haas)
- `to_char()` における末尾のゼロ除去を修正しました。(Marti Raudsepp)
小数点の後の桁位置がない `FM` 付きの書式において、小数点より左のゼロが正しく除去できていませんでした。
- 2^{63} 近辺の入力に対するオーバーフローを防ぐように `pg_size_pretty()` を修正しました。(Tom Lane)
- `plpgsql` のレコード値における `typmod` マッチ検査を弱めました。(Tom Lane)
過度に積極的な検査によって、保持しなければならない長さ修飾子が破棄されることがあり得ました。
- `initdb` の間、ロケール名内の引用符を正しく扱います。(Heikki Linnakangas)
この問題は、"People's Republic of China"などの、いくつかのWindowsのロケールにて起こり得ました。
- `pg_upgrade` において、孤児になった一時テーブルをダンプしません。(Bruce Momjian)
テーブルOIDの割り当てが、新旧のインストレーション間で同期できなくなる状況を防ぎます。
- 8.3からのアップグレードにおいて、`TOAST` テーブルの `relfrozenxids` を維持するように `pg_upgrade` を修正しました。(Bruce Momjian)
これに失敗すると、アップグレードの後 `pg_clog` ファイルがあまりに早く削除される可能性があります。
- `pg_upgrade` において、Windowsで `-1` (ログ) オプションが動作するように修正しました。(Bruce Momjian)
- `pg_ctl` において、Windowsのサービス登録用の静寂モードをサポートしました。(MauMau)
- 異なるファイルから `COPY` している時の、`psql` のスクリプトファイル行番号の計数処理を修正しました。(Tom Lane)
- `standard_conforming_strings` に合わせて、`pg_restore` の直接データベースモードを修正しました。(Tom Lane)
`pg_restore` が、`standard_conforming_strings` が `on` に設定されたアーカイブファイルからデータベースサーバに直接リストアする時に間違ったコマンドを発行する可能性があります。
- 並行 `pg_restore` をサポートしない状況をよりユーザフレンドリにしました。(Tom Lane)
この変更により、何らかのリストア作業が発生する前にこうした状況を検知し報告することが確実にになりました。
- `libpq` のLDAPサービス検索コードにおける `write-past-buffer-end` とメモリリークを修正しました。(Albe Laurenz)
- `libpq` において、非ブロッキングI/OとSSL接続を使用する場合の失敗を防止します。(Martin Pihlak、Tom Lane)
- `libpq` の接続開始期間のエラーの取扱いを改良しました。(Tom Lane)
具体的には、SSL接続開始期間の `fork()` 失敗についてのサーバ報告への応答がより健全になりました。
- SSL失敗に関する `libpq` のエラー報告を改良しました。(Tom Lane)

- 元々サーバ問い合わせから得られたPGresultに新しいタプルを追加する時に、クラッシュする可能性を避けるようにPQsetvalue()を修正しました。(Andrew Chernow)
- ecpglibがdouble値を15桁の精度で書き込むようにしました。(Akira Kurosawa)
- ecpglibにおいて、エラー後確実にLC_NUMERICの設定を元に戻します。(Michael Meskes)
- blowfishの符号付き文字に関する不具合(CVE-2011-2483)に対する上流の修正を適用しました。(Tom Lane)

contrib/pg_cryptoのblowfish暗号化コードは、charが符号付きであるプラットフォーム(ほとんどのプラットフォーム)において間違った結果を生成し、暗号化されたパスワードが本来より脆弱になりました。
- contrib/segにおけるメモリリークを修正しました。(Heikki Linnakangas)
- 空のインデックスに対して一貫性を持った結果を生成するようにpgstatindex()を修正しました。(Tom Lane)
- perl 5.14を用いたビルドを可能にしました。(Alex Hunsaker)
- 空白を含むファイルパスでのビルドとインストールに関連した問題を修正しました。(Tom Lane)
- 時間帯データファイルをtzdata release 2011iに更新しました。 カナダ、エジプト、ロシア、サモア、南スーダンにおける夏時間規則の変更が含まれています。

[前のページ](#)
リリース9.0.6

[ホーム](#)
[上に戻る](#)

[次のページ](#)
リリース9.0.4

E.25. リリース9.0.4

リリース日: 2011-04-18

このリリースは9.0.3に対し、各種の不具合を修正したものです。9.0メジャーリリースにおける新機能については[項E.29](#)を参照してください。

E.25.1. バージョン9.0.4への移行

9.0.Xからの移行ではダンプ/リストアは不要です。

しかし、使用しているインストレーションが以前のバージョンからpg_upgradeを用いてアップグレードしたものであれば、pg_upgradeにおいて今回修正された不具合のためにデータを失う可能性を防止するための操作を行わなければなりません。推奨する解消方法はすべてのTOASTテーブルに対してVACUUM FREEZEを行うことです。詳しくはhttp://wiki.postgresql.org/wiki/20110408pg_upgrade_fixを参照してください。

E.25.2. 変更点

- pg_upgradeのTOASTテーブルの取り扱いを修正しました。(Bruce Momjian)

pg_upgradeの間、TOASTテーブルに対するpg_class.relfrozenxid値が正しく新しいインストレーションにコピーされていませんでした。このため実行後、TOASTテーブル内のタプルを有効にするために必要であるのに、pg_clogファイルが破棄されてしまう結果となり、そのため"could not access status of transaction"エラーをもたらすことがあり得ました。

このエラーは、pg_upgradeを用いてアップグレードしたインストレーションにおいて、データ損失という重大な危険を引き起こします。このパッチは、今後pg_upgradeの利用した場合で問題が起こらないようにするものです。不具合のあるバージョンのpg_upgradeで処理されたインストレーションにおける問題を治すものではありません。

- 間違った"PD_ALL_VISIBLE flag was incorrectly set"という警告を抑制します。(Heikki Linnakangas)

VACUUMが時々実際のところ有効な場合でもこの警告を発していました。

- ホットスタンバイの競合状態に対してより良いSQLSTATEエラーコードを使用します。(Tatsuo Ishii, Simon Riggs)

再試行可能な競合エラーはすべて、再試行が可能であることを示すエラーコードを持つようになりました。また、マスター側でデータベースが削除されたことが原因でセッションが閉じた場合、接続プールの処理が正しくこの状況を扱うことができるように、ERRCODE_ADMIN_SHUTDOWNではなくERRCODE_DATABASE_DROPPEDとして報告されるようになりました。

- 起動処理とbgwriter処理との相互作用において断続的なハングを防止します。(Simon Riggs)

これはホットスタンバイ以外の場合のリカバリに影響しました。

- 複合型自身を複合型に含めることを許可しません。(Tom Lane)

これは、複合型を処理する際にサーバが無限再帰をしてしまう状況を防止します。こうした構造の使い道は一部ありますが、常に安全に動作することを確実にするために必要な作業量を正当化するほどの説得力はないと思われます。

- カタログキャッシュの初期化中のデッドロックの可能性を取り除きました。(Nikhil Sontakke)

一部のキャッシュを読み取るコードが、インデックスのカタログをロックする前にシステムインデックスに対して共有ロックを獲得する場合があります。これは他でより標準的な順序で排他ロックを獲得しようとする処理とデッドロックする可能性があります。

- 対象タプルに対する同時更新があった時のBEFORE ROW UPDATEトリガ処理における、不正な領域を指し示すポインタ問題を修正しました。(Tom Lane)

この不具合は、UPDATE RETURNING ctidを実行しようとした時に、断続的に"cannot extract system attribute from virtual tuple"で失敗するという結果で気付きました。更新したタプルに対して不正なインデックス項目が生成されてしまうなど、より深刻なエラーとなる可能性がごくわずかながら存在します。

- 遅延トリガイベントを待たせているテーブルに対するDROP TABLEを許可しません。(Tom Lane)

以前はDROPが進み、トリガが最終的に発行された時に"could not open relation with OID nnn"エラーをもたらしました。

- pg_hba.conf内で"replication"をユーザ名として許可します。(Andrew Dunstan)

"replication"はデータベース名の列では特別なものですが、ユーザ名の列でも間違っても同様に特別扱いされていました。

- GEQO最適化中、常に偽のWHERE条件によって引き起こるクラッシュを防止します。(Tom Lane)
- 半結合とアンチ結合の場合のプランナの取り扱いを改良しました。(Tom Lane)
- 副問い合わせ内のSELECT FOR UPDATEの扱いを修正しました。(Tom Lane)

この不具合はよく"cannot extract system attribute from virtual tuple"エラーを引き起こしました。

- 全文検索の選択性推定においてNULLを考慮するように修正しました。(Jesper Krogh)
- インデックスアドバイザープラグインによって注入された仮想インデックスをサポートするようにget_actual_variable_range()を修正しました。(Gurjeet Singh)
- PL/Pythonの部分配列化に関するメモリリークを修正しました。(Daniel Popowich)
- ユーザのホームディレクトリが利用できない場合でもlibpqのSSL初期化ができるようになりました。(Tom Lane)

SSLモードがルート証明書ファイルを必要としない場合、失敗させる必要はありません。この変更により9.0より前のリリースの振舞いに戻りました。

- conninfo_array_parseでエラーを検知した時に有益なエラーメッセージを返すようにlibpqを修正しました。(Joseph Adams)

タイポによりライブラリがエラーメッセージを含むPGconn構造体ではなくヌルをアプリケーションに返していました。

- ecpgプリプロセッサの浮動小数点定数の扱いを修正しました。(Heikki Linnakangas)
- 並列pg_restoreがPOST_DATA項目上のコメントを正しく扱うように修正しました。(Arnd Hannemann)
- TOCファイル内の(1KB以上の)長い行に対処するようにpg_restoreを修正しました。(Tom Lane)
- コンパイラの過度に積極的な最適化によるゼロ除算が原因のクラッシュに対して、より多くの対策を取りました。(Aurelien Jarno)
- MIPS上のFreeBSDとOpenBSDでdlopen()の使用をサポートしました。(Tom Lane)

これらのシステムではこのシステム関数はMIPSハードウェアで利用できないという仮定を直接書き込んだコードがありました。より最近のバージョンでは利用できますので、コンパイル時の試験を使用するように変更しました。

- HP-UXにおけるコンパイル失敗を修正しました。(Heikki Linnakangas)
- プロセスの起動時Windowsコンソールへの書き出しを非常に早い段階で試行する時にクラッシュしないようにしました。(Rushabh Lathia)
- WindowsにおいてMinGW 64ビットコンパイラによる構築をサポートしました。(Andrew Dunstan)
- Windowsにおけるlibintlのバージョン非互換問題を修正しました。(Hiroshi Inoue)
- Windowsのビルドスクリプト内のxcopyの使用がWindows 7で正しく動作するように修正しました。(Andrew Dunstan)

ビルドスクリプトのみに影響し、インストールや使用には影響しません。

- Cygwinにおいてpg_regressで使用するパス区切り文字を修正しました。(Andrew Dunstan)
- 時間帯データファイルをtzdata release 2011fに更新しました。チリ、キューバ、フォークランド諸島、モロッコ、サモア、トルコにおける夏時間規則の変更、南オーストラリア、アラスカ、ハワイの歴史的な修正が含まれています。

E.26. リリース9.0.3

リリース日: 2011-01-31

このリリースは9.0.2に対し、各種の不具合を修正したものです。9.0メジャーリリースにおける新機能については[項E.29](#)を参照してください。

E.26.1. バージョン9.0.3への移行

9.0.Xからの移行ではダンプ/リストアは不要です。

E.26.2. 変更点

- walreceiverの終了前に、受信したすべてのWALをディスクにfsyncされることを確実にしました。(Heikki Linnakangas)

こうしないとスタンバイサーバは一部の同期されていないWALを再生することがあり得、もしシステムがまさにその時点でクラッシュした場合おそらくデータ破損につながります。

- walreceiverにおける余分なfsync実行を防止しました。(Heikki Linnakangas)
- 必要な場合にALTER TABLEが一意性制約および排他制約を再検証するようにしました。(Noah Misch)

VACUUM FULLおよびCLUSTER実行中の再検証を抑制することを目的とした変更が、意図せずALTER TABLEにも影響したため、9.0でおかしくなりました。

- 継承ツリーのテーブルがすべて同一でない場合、その継承ツリーのUPDATEに対するEvalPlanQualを修正しました。(Tom Lane)

テーブルの行型の（一部の子テーブルにのみ削除された列が存在するなど）何らかの変化はEvalPlanQualコードを混乱させ、誤動作、最悪はクラッシュを導きました。EvalPlanQualは同一行に対する同時更新の間だけ実行されますので、この問題は散発的にしか発生しませんでした。

- EXPLAINがCASE式の簡易構文を表示しようとした時の失敗を防止します。(Tom Lane)

CASEのテスト式が定数の場合、プランナはCASEを式表示用コードを混乱させる形式に単純化してしまいました。その結果"unexpected CASE WHEN clause"というエラーになりました。

- 過去に存在した添字範囲に対する部分配列代入を修正しました。(Tom Lane)

新しく追加される添字と過去に存在した添字の先頭との間に隙間があった場合、コードは古い配列のヌルビットマップからコピーしなければならない項目数を誤計算してしまい、データ破損またはクラッシュを導く可能性があります。

- プランナにおける、非常に時間が離れた日付値に対する想定外の変換オーバーフローを防止します。(Tom Lane)

date型はtimestamp型で表現可能な範囲よりもより広い日付範囲をサポートします。しかしプランナは常に問題なくdateからtimestampへの変換が可能であることを仮定していました。

- 配列にNULL項目が含まれる場合のPL/Pythonのクラッシュを修正しました。(Alex Hunsaker)
- 配列の次元を定義する定数についてのecpgの固定の長さ制限を取り除きました。(Michael Meskes)
- ... & !(subexpression) | ...を含むtsquery値の間違った解析を修正しました。(Tom Lane)

こうした演算子の組み合わせを含む問い合わせは正しく実行されませんでした。同じエラーがcontrib/intarrayのquery_int型とcontrib/ltreeのltxtquery型に存在しました。

- query_int型に対するcontrib/intarray入力関数におけるバッファオーバーランを修正しました。(Apple)

この関数の返すアドレスが上書きされる可能性があるという点で、この不具合はセキュリティ問題です。この問題を報告し修正版を提供していただいたApple Incのセキュリティチームに感謝します。(CVE-2010-4015)

- contrib/segのGiST picksplitアルゴリズムにおける不具合を修正しました。(Alexander Korotkov)

これにより、`seg` 列上のGiSTインデックスにおいて実際に不正な結果になることはありませんが、非常に非効率的な結果になることがあり得ました。こうしたインデックスがある場合、この更新版をインストールした後に`REINDEX` 処理を行うことを検討してください。（これは過去の更新の`contrib/cube` で修正された不具合と同じです。）

[前のページ](#)

リリース9.0.4

[ホーム](#)

[上に戻る](#)

[次のページ](#)

リリース9.0.2

E.27. リリース9.0.2

リリース日: 2010-12-16

このリリースは9.0.1に対し、各種の不具合を修正したものです。 9.0メジャーリリースにおける新機能については[項E.29](#)を参照してください。

E.27.1. バージョン9.0.2への移行

9.0.Xからの移行ではダンプ/リストアは不要です。

E.27.2. 変更点

- Linuxにて`wal_sync_method`のデフォルトが強制的に`fdatasync`になるようにしました。(Tom Lane, Marti Raudsepp)

長い間、Linuxでのデフォルトは実質`fdatasync`でした。しかし最近のカーネルの変更により、PostgreSQLは代わりに`open_datasync`を選択するようになりました。この選択は性能向上をまったくもたらず、特定のシステム、特に`data=journal`マウントオプションを使用した`ext4`では、完全に失敗してしまいます。

- ホットスタンバイの再生中の"too many KnownAssignedXids"エラーを修正しました。(Heikki Linnakangas)
- ホットスタンバイ中のロック獲得における競合条件を修正しました。(Simon Riggs)
- ホットスタンバイ中の不要な競合を防止しました。(Simon Riggs)

これは、本当は競合していないが、再生がスタンバイ問い合わせと競合するとみなしてしまう（そのため再生遅延や問い合わせキャンセルの可能性が発生する）一部の状況を修正します。

- GINインデックスに対するWAL再生ロジックにおける分類不具合を修正しました。(Tom Lane)

これは"bad buffer id: 0"という失敗、あるいはレプリケーション中のインデックス内容の破損という結果になり得ました。

- 開始チェックポイントWALレコードが再実行時点と同じWALセグメント内に存在しない場合のベースバックアップからのリカバリを修正しました。(Jeff Davis)
- マスターデータベースクラスタの作成直後にストリーミングレプリケーションが有効になった時の境界値不具合を修正しました。(Heikki Linnakangas)
- 複数のワーカが有効なまま長時間残った場合の自動バキュームの永続的な速度低下を修正しました。(Tom Lane)

多くのテーブルを処理している場合、自動バキュームワーカの`vacuum_cost_limit`の効果は、極端な速度低下をもたらすほぼゼロまで落ちることがあり得ました。

- 自動バキュームランチャの長期のメモリリークを修正しました。(Alvaro Herrera)
- トランザクションの外部から差し迫るトランザクション周回条件の通知試行の際の失敗を防止します。(Tom Lane)

このミスによりデータベース起動処理が失敗するため、トランザクションの周回が差し迫った後のリカバリを妨害していました。

- IA64においてレジスタスタックのオーバーラン検知サポートを追加しました。(Tom Lane)

IA64アーキテクチャには2つのハードウェアスタックがあります。スタックオーバーラン失敗を完全に防ぐには両方を検査しなければなりません。

- `copyObject()`におけるスタックオーバーフロー検査を追加しました。(Tom Lane)

非常に複雑な問い合わせが与えられた場合のスタックオーバーフローのため、特定のコードパスがクラッシュする可能性があります。

- 一時GiSTインデックスにおけるページ分割検知を修正しました。(Heikki Linnakangas)

例えば挿入が行われた時に一時インデックスをスキャンする開いているカーソルがあると、そのインデックスで"同時実

行"ページ分割を持つことができます。 GiSTではこの状況の検知に失敗していました。 このためGiSTはそのカーソルの実行が続く間、間違った結果をもたらすことができました。

- 接続処理の初期段階における検査エラーを修正しました。(Tom Lane)

子プロセスがあまりに多くある場合のこの検査が一部の状況で飛ばされていました。 このため固定長配列に新しい子プロセスを追加しようとする時にpostmasterのクラッシュする可能性があります。

- ウィンドウ関数の効率を向上しました。(Tom Lane)

多数のタプルを前もって読み取る必要があるが、work_memがそれらすべてをメモリ内に保持できるほど大きいという、特定の状況では予期しないほど低速になりました。 特にpercent_rank()、cume_dist()、ntile()ではこの問題が当てはまります。

- 複雑なインデックス式をANALYZEする時のメモリリークを防ぎます。(Tom Lane)
- 行全体のVarを使用するインデックスがそのテーブルに依存し続けることを確実にしました。(Tom Lane)

create index i on t (foo(t.*))のように宣言されたインデックスはそのテーブルが削除された時に自動的に削除されませんでした。

- DROP OWNED BYにおいて、あるユーザに属している外部データラップ/サーバ権限について失われていたサポートを追加しました。(Heikki Linnakangas)
- 複数のOUTパラメータを持つSQL関数を"インライン"化しません。(Tom Lane)

結果の想定行型に関する情報が存在しないためにクラッシュする可能性を防ぎます。

- 引数リストにインライン化可能なユーザ関数への参照を含む、集合を返す関数をインライン化する時のクラッシュを修正しました。(Tom Lane)
- ORDER BY、LIMIT、FOR UPDATE、WITHがINSERT ... VALUESのVALUES部に付いた場合の動作を正しくしました。(Tom Lane)
- OFFキーワードを未予約にしました。(Heikki Linnakangas)

これは、PL/pgSQLにおいてoffを変数名として使用する場合の問題を防ぎます。 これは9.0より前では動作していましたが、PL/pgSQLがコアの予約語を予約済みとして扱うようになったため、動作しなくなっていました。

- COALESCE() 式の定数折りたたみを修正しました。(Tom Lane)

プランナは時々、実際には決して達しない副式の評価を行い、想定外のエラーをもたらす可能性があります。

- 行全体のVarの比較において"could not find pathkey item to sort"というプランナ失敗を修正しました。(Tom Lane)
- GSSAPIサポート付きでコンパイルされたpostmasterで、接続受け入れ (accept() またはその後すぐになされる呼び出しのいずれか) が失敗した時のpostmasterのクラッシュを修正しました。(Alexander Chernikov)
- RADIUS認証サーバからの無効な応答パケットを受け取った後に再試行します。(Magnus Hagander)

これは危険性が低いサービス不能状態の可能性を修正します。

- log_temp_filesが有効である時の一時ファイルの削除失敗を修正しました。(Tom Lane)

ログメッセージ出力試行時にエラーが起きた場合、削除が行われませんでした。 このため一時ファイルが累積してしまいました。

- InhRelationノードに関する機能性出力を追加しました。(Tom Lane)

これによりdebug_print_parseが有効な場合に特定種類の問い合わせが行われた場合の失敗を防止します。

- 点から水平線分への距離計算を正しく修正しました。(Tom Lane)

この不具合は様々な複数の幾何距離測定演算子に影響します。

- ecpgにおけるトランザクション状態の計算を正しく修正しました。(Itagaki Takahiro)
- psqlのUnicodeエスケープサポートのエラーを修正しました。(Tom Lane)

- アーカイブに多くのラージオブジェクトがある場合の並行pg_restoreを高速化しました。(Tom Lane)
- 再帰もしくはエラー修復状態にならない"単純な"式についてのPL/pgSQLの扱いを修正しました。(Tom Lane)
- 存在しない列に関するPL/pgSQLのエラー報告処理を修正しました。(Tom Lane)
9.0では、"record foo has no field bar"の方が適切である場合に時々"missing FROM-clause entry for table foo"と報告していました。
- PL/Pythonがタプルフィールドに代入する際にtypmod（長さや精度に関する制限）を尊重するように修正しました。(Tom Lane)
これは8.4からの退行を修正します。
- PL/Pythonの集合を返す関数の扱いを修正しました。(Jan Urbanski)
集合結果を生成するイテレータ内でSPI関数呼び出しが失敗しました。
- contrib/cubeのGiST picksplitアルゴリズムの不具合を修正しました。(Alexander Korotkov)
cube列上のGiSTインデックスにおいて、実際に不正確な結果になることはありませんが、このため相当効率性が劣化する可能性があります。こうしたインデックスがある場合、この更新をインストールした後にREINDEXすることを検討してください。
- contrib/dblink内で、新しい接続を作成する時を除き、"identifier will be truncated"注意を出力しないようにしました。(Itagaki Takahiro)
- contrib/pgcryptoにおいて公開キーがない場合にコアダンプする可能性を修正しました。(Marti Raudsepp)
- contrib/pg_upgradeにおけるバッファオーバーランを修正しました。(Hernan Gonzalez)
- contrib/xml2のXPath問い合わせ関数におけるメモリリークを修正しました。(Tom Lane)
- 時間帯データファイルをtzdata release 2010oに更新しました。フィジィ、サモアにおける夏時間規則の変更、香港の歴史的な修正が含まれています。

E.28. リリース9.0.1

リリース日: 2010-10-04

このリリースは9.0.0に対し、各種の不具合を修正したものです。9.0メジャーリリースにおける新機能については[項E.29](#)を参照してください。

E.28.1. バージョン9.0.1への移行

9.0.Xからの移行ではダンプ/リストアは不要です。

E.28.2. 変更点

- PL/PerlおよびPL/Tclにおいて、呼び出し元のSQLユーザIDごとに別のインタプリタを使用します。(Tom Lane)

この変更は、同じセッション内でその後に異なるSQLユーザIDにより実行されるPerlまたはTclコード（例えばSECURITY DEFINERの内部）を破壊することで発生する可能性があるというセキュリティ問題を防止するものです。ほとんどのスクリプト用言語は、対象とする関数で呼び出される標準関数や演算子の再定義など、これを行うことができる多くの方法を提供します。この変更がないと、PerlまたはTcl言語の使用権限を持つ任意のSQLユーザは基本的に、対象の関数の所有者が持つSQL権限でできることを何でもできるようになります。

この変更の欠点はPerlおよびTcl関数間の内部通信がより難しくなるという点です。逃げ道を提供するために、PL/PerlUおよびPL/TclU関数ではまだセッション当たり1つのインタプリタのみを使用します。これらのすべての関数は従来よりデータベーススーパーユーザの信頼レベルで実行されますので、これはセキュリティ問題とはみなされません。

信頼された実行を提供することを主張するサードパーティ製の手続き言語は似たようなセキュリティ問題を持つ可能性があります。セキュリティが非常に重要な目的のために依存している手続き言語があれば、その作者に連絡を取ることを勧めます。

この問題を指摘いただいたTim Bunceに感謝します(CVE-2010-3433)。

- pg_get_expr() 関数を副選択の出力に対して使用し続けられるようにpg_get_expr() のセキュリティ修正を改良しました。(Tom Lane)
- 不正なプレースホルダ位置の評価を修正しました。(Tom Lane)

この不具合のため外部結合の内側が出力リスト内に制限がない式を持つ副選択であった場合など、問い合わせ出力がNULLであるべき時に非NULLの結果になるかもしれませんでした。

- 結合除去におけるプレースホルダ式の扱いを修正しました。(Tom Lane)
- UNION ALLのメンバーリレーションの二重スキャンの可能性を修正しました。(Tom Lane)
- 監視中止後のProcessIncomingNotify()における無限ループを防止します。(Jeff Davis)
- 自動バキューム処理中のクラッシュからshow_session_authorization()を防止します。(Tom Lane)
- 0001-01-01 AD以前のユリウス日の入力を再度許します。(Tom Lane)

8.4より前では'J1000000'::dateなどの入力が動作しました。しかし、エラー検査を加えたことで意図せず壊れていました。

- 自動コミットモードが無効な場合、DISCARD ALLをトランザクションブロック内に包むべきものではないコマンドとしてpsqlが認識するようにしました。(Itagaki Takahiro)
- 構築手順や文書を更新して、ソースコードリポジトリがCVSからGitに変更されたことを反映しました。(Magnus Hagander他)

E.29. リリース9.0

リリース日: 2010-09-20

E.29.1. 概要

このリリースのPostgreSQLで数年に渡り要望があった、簡単に利用できるレプリケーション、一括権限管理機能、匿名ブロックといった機能を追加しました。これまでのメジャーリリースではその範囲は保守的なものでしたが、このリリースでは、これまでPostgreSQLを使用してきたユーザや新しく使い始めるユーザが喜ぶ機能を提供することに大胆かつ新しい期待を持っています。これはすべて非互換性がほぼないようになされました。主な改良点を以下に示します。

- 組込みのログ SHIPPINGによるレプリケーション。この進歩は次の2つの機能から構成されます。ストリーミングレプリケーションにより、継続的なアーカイブ(WAL)ファイルをスタンバイサーバに対してネットワーク上にストリーミングさせることができます。ホットスタンバイにより、継続的アーカイブを行うスタンバイデータベースサーバは読み取り専用の問い合わせを受け付けることができます。これらを合わせた結果、シングルマスタと複数の読み取り専用のスレーブサーバをサポートします。
- より簡単なデータベースオブジェクトの権限管理。 [GRANT/REVOKE IN SCHEMA](#)は既存のオブジェクトに対する一括権限変更をサポートし、そして、 [ALTER DEFAULT PRIVILEGES](#)は将来作成されるオブジェクトの権限を制御します。ラージオブジェクト (BLOB) も同様に権限管理をサポートするようになりました。
- ストアドプロシージャサポートの大きな改良。 [DO](#)文により、その場限り、または、"匿名"のコードブロックをサポートします。関数は名前付きパラメータを使用して呼び出すことができるようになりました。 [PL/pgSQL](#)はデフォルトでインストールされるようになり、また、 [PL/Perl](#)および [PL/Python](#)は、Python3のサポートを含む多くの点で改良されました。
- [64ビットWindows](#)を完全にサポートします。
- 追加されたウィンドウ処理オプション (`PRECEDING` and `FOLLOWING`) と集約関数に渡される値の順序を制御する機能による、より先端的な報告問い合わせ。
- SQL互換の [列単位のトリガ](#)と条件付きのトリガ実行など、新しいトリガ機能。
- [遅延可能な一意性制約](#)。工夫することなく一意キーの大量更新ができるようになります。
- [排他制約](#)。これらは、複雑な条件を強制させることができる、汎用的な一意性制約を提供します。
- 新しく高度なセキュリティ機能。RADIUS認証、LDAP認証の改良、パスワード強度を検査する新しい [passwordcheck](#) オプションモジュールなどがあります。
- 新しい高性能な [LISTEN/NOTIFY](#) 機能の実装。待ち状態のイベントがテーブルではなく、メモリアベースのキューに格納されるようになりました。また、以前のようにイベント名だけを送信するのではなく、"ペイロード"文字列を各イベントといっしょに送信できるようになりました。
- 新しい [VACUUM FULL](#) の実装。このコマンドは、容量を抑えるために行を一行一行移動するのではなく、テーブルおよびインデックス全体を書き換えるようになりました。ほとんどの状況でかなり高速になり、また、インデックスが膨張することがなくなりました。
- 8.3または8.4から9.0へのその場での更新をサポートする [pg_upgrade](#) を追加しました。
- 不要な結合を除去するなど、特定の種類の問い合わせに対する複数の性能向上。オブジェクト/リレーションマップ(ORM)で生成されるなどの自動生成される問い合わせの一部を最適化する際に役に立ちます。
- [EXPLAIN](#) の改良。この出力がJSON、XML、YAML形式で利用可能になりました。これまでは利用できなかったバッファ使用状況などその他のデータが含まれるようになりました。
- 新しい関数やより多くのデータ能力など、 [hstore](#) の改良。

上述の項目については、以下でより詳しく説明します。

E.29.2. バージョン9.0への移行

全ての以前のリリースからデータの移行を行うためには、`pg_dump`を使用したダンプ/リストア、または、`pg_upgrade`の使用が必要です。

バージョン9.0には、新機能のサポートおよびコード品質の向上のために、後方互換性を一部壊してしまう多くの変更が含まれます。具体的には `PL/pgSQL`、ポイントインタイムリカバリ (PITR) およびウォームスタンバイを多く使用しているユーザは、これらにユーザに影響する変更が多くありますので、自身のアプリケーションを試験しなければなりません。以下の非互換性に注意してください。

E.29.2.1. サーバの設定

- `add_missing_from` サーバパラメータがなくなりました。これは長い間デフォルトで無効でした。(Tom Lane)
- `regex_flavor` サーバパラメータがなくなりました。これは長い間デフォルトで `advanced` でした。(Tom Lane)
- `archive_mode` は `archive_command` のみに影響するようになりました。新しい `wal_level` 設定は先行書き込みログの内容に影響を与えます。
- `log_temp_files` がデフォルトのファイルサイズの単位としてキロバイトを使用するようになりました。(Robert Haas)

E.29.2.2. 問い合わせ

- [親テーブル](#)を問い合わせる場合、問い合わせの一部としてスキャンされる子テーブルに対する異なる権限検査がまったく行われません。(Peter Eisentraut)

標準SQLではこの挙動を規定しています。また、実際の所、親と同様に子それぞれに権限を検査するという以前の振舞いよりも非常に便利です。

E.29.2.3. データ型

- [bytea](#)の出力はデフォルトで16進数書式になりました。(Peter Eisentraut)

互換性が必要ならば[bytea_output](#)サーバパラメータを使用して伝統的な出力書式を選択することができます。

- 配列入力において通常のASCII空白文字のみを無視可能なものとみなすようになりました。一部のロケールに従うと空白文字になるものであっても、ASCII文字以外を無視することはありません。

これは、配列の値がサーバのロケール設定に依存して異なった解釈をされる場合の一部の境界問題を防止します。

- [SIMILAR TO](#)パターンとSQL形式の[substring\(\)](#)パターンの標準互換性を高めました。(Tom Lane)

これには、パターンのメタ文字としての`?`および`{...}`の扱いが含まれます。これらの文字は以前は単なるリテラル文字でしたが、SQL:2008で追加された新しい機能に対応しました。また`^`および`$`は単なるリテラル文字として扱われるようになりました。これまで、パターンがSQL規則ではなくPOSIXに従うかのようにメタ文字として扱われていました。また、標準SQLの[substring\(\)](#)では入れ子のための括弧が部分文字列の取得に影響を与えることがなくなりました。同時に大括弧式(文字クラス)の処理の標準に対する互換性が高まりました。

- 標準SQLに従い、[substring\(\)](#)の第3パラメータでは負の長さ値を拒絶します。(Tom Lane)

- [date_trunc](#)を秒の端数に対する精度を減らす時に四捨五入するのではなく切り捨てるようにしました。(Tom Lane)

整数を基にした日付時刻では常にこのようにこのコードは動作していました。浮動小数点を基にした日付時刻も同様に動作するようになりました。

E.29.2.4. オブジェクトの改名

- 子テーブルが複数の関連がない親から同じ列を継承する場合、[RENAME](#)中の列名の一貫性をより厳しく強制します。(KaiGai Kohei)

- 背後にあるテーブル列が改名された場合にインデックス名とインデックス列名は自動的に改名されなくなりました。(Tom Lane)

管理者はまだこうしたインデックスや列を手作業で改名することができます。この変更は、改名後も一意性インデックスが正しく認識されるように、JDBCドライバとおそらく他のドライバの更新を要求します。

- [CREATE OR REPLACE FUNCTION](#)で宣言された関数パラメータの名前を変更できなくなりました。(Pavel Stehule)

名前付きパラメータ呼び出しにおけるあいまい性を防ぐために、既存関数の宣言における入力変数に対する別名を変更できないようになりました。(しかし過去に名前を付けていないパラメータには名前を割り当てることができます。) これを行うためには関数を[DROP](#)し再作成しなければなりません。

E.29.2.5. PL/pgSQL

- 変数名が問い合わせで使用される列名と競合する場合にPL/pgSQLはエラーを発生するようになりました。(Tom Lane)

これまでの動作は問い合わせの列名より優先してあいまいな名前をPL/pgSQL変数にバインドすることでした。これはよく驚くほどの誤動作をもたらしていました。エラーを発生させることにより、曖昧な状況を簡単に検知できるようになります。こうしたエラーを発生させる関数を修正し、競合を解消させることを推奨しますが、[plpgsql.variable_conflict](#)サーバ変数または関数毎の[#variable_conflict](#)オプションにより従来の動作に戻すことができます。

- PL/pgSQLは特定のSQL予約語と一致する変数名を許さなくなりました。(Tom Lane)

これはPL/pgSQLのパースをよりコアのSQLパースに近づけるように調整した結果です。必要ならば、変数名を二重引用符でくくることでこの制限を回避することができます。

- PL/pgSQLは、基本型と同様に想定する型修飾子と一致する複合型結果の列を必要とするようになりました。(Pavel Stehule, Tom Lane)

例えば、結果の列の型が[NUMERIC\(30,2\)](#)と宣言されていた場合、その列に多少異なる精度の[NUMERIC](#)を返すことを受け付けなくなりました。これまでのバージョンでは修飾子の検査は省略され、その結果、実際には宣言された制限に従わない結果行が許されていました。

- PL/pgSQLが複合型のフィールドへの選択をより一貫性を持って扱うようになりました。(Tom Lane)

これまででは、[SELECT ... INTO rec.fld FROM ...](#)のような文は[fld](#)レコードフィールドが複合型であったとしてもスカラの代入として扱われました。これが、[INTO](#)の対象が通常の複合型の変数であった場合と同様に、レコードの代入として扱われるようになりました。このため、このフィールドのサブフィールドに代入される値は、これまでのバージョンにおける[ROW\(...\)](#)式ではなく、[SELECT](#)リストの別々の列として記述されなければなりません。

9.0と過去のリリースの両方で動作する方法で行う必要があれば、[rec.fld := ROW\(...\) FROM ...](#)のように記述することができます。

- PL/pgSQLの[RENAME](#)宣言を削除しました。(Tom Lane)

`RENAME`の代わりに[ALIAS](#)を使用してください。これは、以前のような（例えば`$1`のような）ドル記号付きの変数だけではなく任意の変数に別名を作成することができるようになります。

E.29.2.6. その他の非互換性

- 演算子名としての`=>`を使用を廃止する予定です。(Robert Haas)

名前付きパラメータに関する標準SQL記法をサポートするために、将来のバージョンのPostgreSQLではおそらくこの演算子名を完全に拒絶します。現時点ではまだ許されますが、このような演算子が定義された場合に警告が現れます。

- 64ビット整数データ型が動作しないプラットフォームのサポートを取りやめました。(Tom Lane)
まだサポートされているプラットフォームはすべて64ビット整数データ型が動作するはずです。

E.29.3. 変更点

バージョン9.0には予想しないほど多くの主要機能が追加され、また、増強、改良、新しいコマンド、新しい関数、その他の変更が200以上あります。

E.29.3.1. サーバ

E.29.3.1.1. 継続的アーカイブ処理とストリーミングレプリケーション

PostgreSQLに存在するスタンバイサーバ機能は、スタンバイサーバにおいて読み取り専用の問い合わせをサポートすること、および、マスタサーバとスタンバイサーバとの間の遅延を大きく削減することができるように拡張されました。多くのユーザにとって、これは有用かつ、高可用性、スケールアウト性を目的としたレプリケーションを構成する管理作業を低減します。

- スタンバイサーバで読み取り専用の問い合わせを受け付けることができます。(Simon Riggs, Heikki Linnakangas)

この機能はホットスタンバイと呼ばれます。この機能を制御するための新しい設定が`postgresql.conf`と`recovery.conf`にあります。さらに広大な[文書](#)も存在します。

- 先行書き込みログ(WAL)データをスタンバイサーバに流し込むことができます。(Fujii Masao, Heikki Linnakangas)

これはストリーミングレプリケーションと呼ばれます。これまでWALデータはWALファイル全体（通常それぞれ16メガバイト）を単位としてのみスタンバイサーバに送付することができました。ストリーミングレプリケーションはこの非効率性を減らし、マスタの変更をほとんど遅延なくスタンバイサーバに伝播できるようになります。この機能を制御するための新しい設定が`postgresql.conf`と`recovery.conf`にあります。さらに広範囲の[文書](#)も存在します。

- スタンバイサーバのWAL状況の監視に使用することができる、[pg_last_xlog_receive_location\(\)](#)および`pg_last_xlog_replay_location()`を追加しました。(Simon Riggs, Fujii Masao, Heikki Linnakangas)

E.29.3.1.2. 性能

- [ALTER TABLESPACE ... SET/RESET](#)を使用して、テーブル空間単位の値をシーケンシャルおよびランダムページコスト推定(`seq_page_cost/random_page_cost`)に設定できます。(Robert Haas)

- 結合問い合わせにおけるEvalPlanQual再検査の性能と信頼性を改善しました。(Tom Lane)

最近更新された行に対する、結合を含むUPDATE、DELETEおよびSELECT FOR UPDATE/SHARE問い合わせが非常によく動作するようになりました。

- 同じトランザクションにおいてテーブル生成またはテーブルデータの消去が以前に行われていた場合の、[TRUNCATE](#)の性能を改良しました。(Tom Lane)
- 継承された子テーブルの検索性能を改良しました。(Tom Lane)

E.29.3.1.3. オプティマイザ

- 不要な[外部結合](#)を除きました。(Robert Haas)

内側が一意かつ結合の上位で参照されない場合の外部結合は不要ですので、削除されるようになりました。これは、オブジェクト/リレーションマップ（ORM）で作成される問い合わせなど、多くの自動生成される問い合わせを高速化します。

- `IS NOT NULL`制限でインデックスを使用できます。(Tom Lane)

これは特に、インデックス内に多くのNULL値を含む場合の`MAX()`/`MIN()`値の検索に有用です。

- いつマテリアライズされたノードを使用するか、および、`DISTINCT`についてソート処理を使用するかハッシュ処理を使用するか、に関するオプティマイザの選択を改良しました。(Tom Lane)
- `boolean <>`演算子を含む式におけるオプティマイザの等価検知を改良しました。(Tom Lane)

E.29.3.1.4. GEQO

- GEQOが問い合わせ計画を作成する時毎回、同じランダムシードを使用します。(Andres Freund)

遺伝的問い合わせオプティマイザ(GEQO)は無作為な計画を選択し続けますが、同一の問い合わせに対して同じ無作為な計画を選択するように

なりました。これによりより一貫した性能が得られます。他の計画を試すために[geqo_seed](#)を変更することができます。

- GEQOの計画選択を改良しました。(Tom Lane)

まれに発生する"failed to make a valid plan"エラーを防止します。また計画作成速度も改良されたはずです。

E.29.3.1.5. オプティマイザの統計情報

- 継承ツリー統計情報をサポートする[ANALYZE](#)を改良しました。(Tom Lane)

具体的にはこれはパーティション分けされたテーブルで有効です。しかし子テーブルが変更された時に、自動バキュームはまだ自動的に親テーブルを再解析しません。

- 再解析が必要か否かに関する[autovacuum](#)の判定を改良しました。(Tom Lane)
- より大きい/より小さいの比較についてオプティマイザの推定を改良しました。(Tom Lane)

より大きい/より小さいの比較についての統計情報を検索する時、比較用の値がヒストグラムバケットの先頭または末尾にあれば、現在の列の最小と最大を得るために（利用可能であれば）インデックスを使用します。これはデータ範囲の終端に近い比較値に対する推定精度を、特に新しいデータの追加によりデータ範囲が定期的に変更する場合に、大きく改善します。

- [ALTER TABLE](#)を使用することで、個別値の個数に関する統計情報の設定が可能です。(Robert Haas)

これにより、列の個別値の数もしくは割合をユーザが上書きすることができます。この統計情報は通常[ANALYZE](#)で計算されますが、特に非常に多くの行があるテーブルでは推定値が悪化することがあります。

E.29.3.1.6. 認証

- [RADIUS](#) (リモート認証ダイヤルインユーザサービス)認証サポートを追加しました。(Magnus Hagander)
- [LDAP](#)(軽量ディレクトリアクセスプロトコル)認証において"search/bind"モードで操作できます。(Robert Fleming, Magnus Hagander)
これにより、まずユーザを検索し、その後そのユーザ用に返されるDN(識別名)をシステムが使用できるようになります。
- [samehost](#)および[samenet](#)指定を[pg_hba.conf](#)に追加しました。(Stef Walter)
これらはそれぞれサーバのIPおよびサブネットアドレスに一致します。
- クライアントが適切なクライアント証明書を返すことができるように、クライアントに信頼できるSSLルート証明書の名前を渡します。(Craig Ringer)

E.29.3.1.7. 監視

- クライアントが[pg_stat_activity](#)で表示される、[アプリケーション名](#)を設定できる機能を追加しました。(Dave Page)
これにより管理者は元となるアプリケーションによってデータベースの流量を特徴付けし、また、問題への対応を行うことができます。
- [log_line_prefix](#)にSQLSTATEオプション(%e)を追加しました。(Guillaume Smet)
これによりユーザはエラーコード番号によりエラーとメッセージに関する統計情報を収集することができます。
- UTF16符号化方式でWindowsのイベントログに書き出します。(Itagaki Takahiro)
これでWindowsのPostgreSQLのログメッセージに対する多言語サポートが実現しました。

E.29.3.1.8. 統計情報カウンタ

- クラスタ全体で共有するバックグラウンドライタの統計情報をリセットするための[pg_stat_reset_shared\('bgwriter'\)](#)を追加しました。(Greg Smith)
- 個々のテーブルおよび関数用の統計情報カウンタのリセットを可能とする[pg_stat_reset_single_table_counters\(\)](#)および[pg_stat_reset_single_function_counters\(\)](#)を追加しました。(Magnus Hagander)

E.29.3.1.9. サーバ設定

- [データベースまたはロールの組み合わせ](#)に基づく設定パラメータを設定することができます。(Alvaro Herrera)
これまではデータベース単位およびロール単位の設定のみが可能で、組み合わせることができませんでした。すべてのロールおよびデータベースの設定が新しい[pg_db_role_setting](#)システムカタログに格納されるようになりました。新しいpsqlの`\drds`コマンドはこれらの設定を表示します。古い[pg_roles](#)、[pg_shadow](#)および[pg_user](#)システムビューでは組み合わせた情報を表示しません。このためこれらはユーザまたはデータベース用の設定を完全に表さなくなりました。
- [bonjour](#) サーバパラメータを追加しました。これはBonjourが有効なサーバが自身をBonjour経由で広告するかどうかを制御します。(Tom Lane)
デフォルトは無効で、広告を行わないことを意味します。これによりパッケージ作成者は個々のユーザがこの機能を欲しないかどうか気にすることなく、Bonjourを有効として構築したものを配布することができます。
- オプティマイザ内でノードマテリアライゼーションの使用を制御する、[enable_material](#) サーバパラメータを追加しました。(Robert Haas)

デフォルトは有効です。無効の場合、オプティマイザは純粋に性能上の理由のためにマテリアライズされたノードを追加しません。ですが正確性のために必要な場合は使用されます。

- デフォルトのファイルサイズの単位としてキロバイトを使用するように[log_temp_files](#)サーバパラメータを変更しました。(Robert Haas)
これまでは単位指定がない場合、この設定はバイト単位として解釈されました。

- `postgresql.conf`が再読み込みされた場合、パラメータ値の変更記録が残ります。(Peter Eisentraut)

これにより管理者およびセキュリティ担当者はデータベースが変更された時点を監査することができるようになります。また[postgresql.conf](#)の変更が反映されたかどうか検査することもでき、非常に便利です。

- 独自のサーバパラメータに対して、適切にスーパーユーザ権限を強制します。(Tom Lane)

スーパーユーザ以外のユーザが、その時にサーバが認識していないパラメータに対する[ALTER ROLE/DATABASE SET](#)を発行することができなくなりました。これによりサーバは、スーパーユーザのみのパラメータがスーパーユーザのみにより設定されることを正しく検査することができます。これまではセッション開始時に[SET](#)は受け付けられますが、無視されていました。このためスーパーユーザのみの独自パラメータが本来の有用性を持っていませんでした。

E.29.3.2. 問い合わせ

- 返される行数を常に予測できるように、[LIMIT](#)の適用後に[SELECT FOR UPDATE/SHARE](#)処理を行います。(Tom Lane)

これまでは、同時実行のトランザクションによりなされた変更により、[SELECT FOR UPDATE](#)がその[LIMIT](#)で指定した行より少ない行が返ってしまうことが発生しました。[ORDER BY](#)を付けた[FOR UPDATE](#)は今でも予想外の結果を生成することがありますが、これは副問い合わせで[FOR UPDATE](#)を使用することで正しくすることができます。

- [LIMIT/OFFSET](#)構文に関して、伝統的な方法と標準SQLの方法を混在できるようにしました。(Tom Lane)
- [ウィンドウ関数](#)でサポートされるフレームオプションを拡張しました。(Hitoshi Harada)

フレームは[CURRENT ROW](#)から始められるようになり、また、[ROWS n PRECEDING/FOLLOWING](#)オプションがサポートされるようになりました。

- [SELECT INTO](#)および[CREATE TABLE AS](#)はそのコマンドタグ内でクライアントに行数を返します。(Boszormenyi Zoltan)

これにより、追加の[COUNT](#)問い合わせを行うことなく結果数やページ数を計算することができますので、クライアントとの往復作業を完全になくすることができます。

E.29.3.2.1. Unicode文字列

- Unicodeサロゲートペア(二重の16ビット表現)を[U&](#)文字列および識別子でサポートします。(Peter Eisentraut)
- [E'...'](#)文字列内のUnicodeエスケープをサポートします。(Marko Kreen)

E.29.3.3. オブジェクトの操作

- ディスクへの吐き出しを遅延させることにより[CREATE DATABASE](#)を高速化しました。(Andres Freund, Greg Stark)
- テーブル、ビュー、複合型の列に対してのみに[コメント](#)を許します。インデックスやTOASTテーブルのような他の種類のリレーションでは許されません。(Tom Lane)
- 値を持たない[列挙型](#)の作成が可能です。(Bruce Momjian)
- 行をページに合わせることができない場合を除き、[MAIN](#)格納型を持つ列の値を主ヒープページに保持させます。(Kevin Grittner)
これまでは、[MAIN](#)値は行の容量がページ容量の1/4になるまで、強制的にTOASTテーブルとなりました。

E.29.3.3.1. ALTER TABLE

- [DROP COLUMN](#)および[DROP CONSTRAINT](#)に[IF EXISTS](#)を実装しました。(Andres Freund)
- テーブルを書き換える[ALTER TABLE](#)コマンドでWALロギングを飛ばすことができます。(Itagaki Takahiro)

こうした操作は新しいテーブルのコピーが作成されるかロールバックされるかのいずれかです。このため継続的アーカイブモードで実行している場合を除き、WALアーカイブ処理を飛ばすことができます。これによりI/Oオーバーヘッドが減少し、性能が向上します。

- テーブル所有者以外により実行された時の[ALTER TABLE table ADD COLUMN col serial](#)の失敗を修正しました。(Tom Lane)

E.29.3.3.2. CREATE TABLE

- [CREATE TABLE ... LIKE](#)コマンドにおいて、[COMMENTS](#)および[STORAGE](#)設定の複製サポートを追加しました。(Itagaki Takahiro)
- [CREATE TABLE ... LIKE](#)コマンドに全属性コピーの省略形を追加しました。(Itagaki Takahiro)
- 標準SQLの[CREATE TABLE ... OF type](#)コマンドを追加しました。(Peter Eisentraut)

これにより、既存の複合型に一致するテーブルを作成することができます。コマンド内で制約やデフォルトを追加で指定することができます。

E.29.3.3.3. 制約

- [遅延可能な一意性制約](#)を追加しました。(Dean Rasheed)

これにより、列が一意的インデックスを持つ場合でもプライマリキーと印付けされている場合でもUPDATE tab SET col = col + 1などの大量更新が確実に動作できるようになります。制約がDEFERRABLEであると指定されている場合、各行が更新された後ではなく、文の最後で検査されるようになります。また制約検査を現在のトランザクションの最後まで遅延させ、複数のSQLコマンドをまたがって更新を行うこともできます。

- [排他制約](#)を追加しました。(Jeff Davis)

排他制約は等価だけではなく任意の比較演算を許すことにより、一意性制約を汎用化します。これはCREATE TABLE CONSTRAINT ... EXCLUDE句により作成されます。排他制約のもっともよくある使用法は、単に一致しないではなく、重複を許さない列項目を指定することです。時間間隔やその他配列などの範囲に対して有効です。この機能は、多くの予定帳、時間管理、科学アプリケーションにおけるデータ整合性検査を改良します。

- 失敗の原因となる値を報告する、一意性制約違反エラーメッセージを改良しました。(Itagaki Takahiro)

例えば、一意性制約違反ではKey (x)=(2) already exists(キー(x)=(2)はすでに存在します)のように報告されます。

E.29.3.3.4. オブジェクトの権限

- 新しい[GRANT/REVOKE IN SCHEMA](#)句を使用してスキーマ全体にまたがった大量な権限変更を行う機能を追加しました。(Petr Jelinek)

オブジェクト権限管理を簡略化すると共に、アプリケーションのデータセキュリティのためにデータベースロールを使用することをより簡単にします。

- 今後作成されるオブジェクトの権限を制御するための[ALTER DEFAULT PRIVILEGES](#)を追加しました。(Petr Jelinek)

これは複雑なデータベースアプリケーションにおけるオブジェクト権限の割り当てを大きく簡略化します。テーブル、ビュー、シーケンス、関数に対してデフォルトの権限を設定することができます。デフォルトはスキーマ単位またはデータベース全体に対して割り当てることができます。

- GRANT/REVOKEによりラージオブジェクト (BLOB) の権限を制御する機能を追加しました。(KaiGai Kohei)

これまでの、すべてのデータベースユーザは任意のラージオブジェクトを読み取り、変更することができました。読み取り権限、書き込み権限をラージオブジェクト毎に付与したり取り上げたりできるようになりました。また、ラージオブジェクトの所有権も追跡されます。

E.29.3.4. ユーティリティ操作

- [LISTEN/NOTIFY](#)をシステムテーブルではなく、メモリキュー内に待ちイベントを格納するようにしました。(Joachim Wieland)

既存のトランザクションのサポートと配送保証機能を保ちつつ、これは性能を大きく改良しました。

- [NOTIFY](#)はリスナに省略可能な"ペイロード"文字列を渡すことができます。(Joachim Wieland)

これにより汎用目的のイベントキューシステムとしてのLISTEN/NOTIFYの有用性を大きく改良します。

- すべてのデータベース単位のシステムカタログに[CLUSTER](#)を行うことができます。(Tom Lane)

共有カタログはまだクラスタ化できません。

E.29.3.4.1. [COPY](#)

- COPY ... CSV FORCE QUOTE *を受け付けます。(Itagaki Takahiro)

FORCE QUOTE句において*が"すべての列"を表す省略形として使用できるようになりました。

- オプションを括弧内で指定する新しいCOPY構文を追加しました。(Robert Haas, Emmanuel Cecchet)

これにより今後のCOPYオプションの柔軟性が改良されます。過去に存在したオプションのためだけに、古い構文もまだサポートされます。

E.29.3.4.2. [EXPLAIN](#)

- EXPLAINがXML、JSON、YAML形式で出力することができます。(Robert Haas, Greg Sabino Mullane)

新しい出力形式はマシンによる可読性が簡単になり、EXPLAIN出力の解析用の新しいツールの開発を補助します。

- EXPLAIN ANALYZE時に問い合わせバッファの活動を報告する、新しいBUFFERSオプションを追加しました。(Itagaki Takahiro)

これにより個々の問い合わせに対してより良い問い合わせプロファイリングを行うことができます。[log_statement_stats](#)と関連する設定の出力では、バッファ使用量は報告されなくなりました。

- EXPLAINの出力にハッシュ使用状況に関する情報を追加しました。(Robert Haas)

- 括弧内でオプションの指定を行うことができる、新しいEXPLAIN構文を追加しました。(Robert Haas)

これにより将来のEXPLAINオプションに対する柔軟性が大きく高まりました。過去に存在したオプションのためだけに、古い構文もまだサポートされます。

E.29.3.4.3. [VACUUM](#)

- 容量を抑えるために行を一行一行移動するのではなく、テーブル全体を書き換え、インデックスを再構築するように `VACUUM FULL` を変更しました。(Itagaki Takahiro, Tom Lane)
これまでの方式では通常低速であり、インデックスの膨張を引き起こしました。新しい方式では `VACUUM FULL` 実行時に一時的により多くのディスク領域を使用することに注意してください。通常テーブルとインデックスにより占められる領域の2倍程度使用する可能性があります。
- オプションを括弧内に指定することができる、新しい `VACUUM` 構文を追加しました。(Itagaki Takahiro)
これにより将来の `VACUUM` オプションに対する柔軟性が大きく高まりました。過去に存在したオプションのためだけに、古い構文もまだサポートされます。

E.29.3.4.4. インデックス

- [CREATE INDEX](#) においてインデックス名を省略した場合、インデックス名が自動的に付与できるようにしました。(Tom Lane)
- デフォルトで、複数列に対するインデックスが対象列すべてにちなんだ名前が付けられるようになり、また、式インデックス列はその式にちなんだ名前が付けられるようになりました。
- 共有システムカタログのインデックス再作成が完全にトランザクションを守るとなり、かつ、クラッシュに対して安全になりました。(Tom Lane)
これまでは共有インデックスに対するインデックス再作成はスタンダードローンモードでのみ許されていました。またこの操作中のクラッシュによりインデックスは操作前よりも悪化した状態になり得ました。
- GiST に `point_ops` 演算子クラスを追加しました。(Teodor Sigaev)
この機能により `point` 列に対する GiST インデックス付けが可能です。このインデックスは `point <@ polygon` (多角形内の点) などのいろいろな種類の問い合わせに使用することができます。これにより多くの PostGIS 問い合わせが高速になるはずです。
- GIN インデックス生成において赤黒バイナリツリーを使用します。(Teodor Sigaev)
赤黒ツリーは自身でバランスを保ちます。これは入力順序がランダムではない場合に起こる低速化を防ぎます。

E.29.3.5. データ型

- [bytea](#) 値を16進数記法で記述できます。(Peter Eisentraut)
[bytea_output](#) サーバパラメータは `bytea` 出力において16進数が伝統的な書式のどちらを使用するかを制御します。`libpq` の `PQescapeByteaConn()` 関数は、PostgreSQL 9.0以降のサーバに接続する場合、自動的に16進数書式を使用します。しかし9.0より前のバージョンの `libpq` は新しいサーバからの16進数書式を正しく処理しません。
新しい16進数書式は、バイナリデータを使用するより多くのアプリケーションと直接的な互換性を持つことになり、余計な変換を行うことなく格納、抽出を行うことができるようになります。また伝統的な書式よりもかなり高速に読み書きすることができます。
- [extra_float_digits](#) サーバパラメータを3に増やすことができます。(Tom Lane)
これまでの最大の `extra_float_digits` は2でした。 `float4` 値を正確にダンプ/リストアするために3桁が必要な場合があります。`pg_dump` は、3桁を許すサーバからのダンプを行う時に3という設定を使用するようになります。
- `int2vector` 入力に対する入力検査を厳しくしました。(Caleb Welton)

E.29.3.5.1. [全文検索](#)

- `synonym` 辞書において接頭辞サポートを追加しました。(Teodor Sigaev)
- フィルタ処理辞書を追加しました。(Teodor Sigaev)
フィルタ処理用辞書により、トークンを変更した後に後続の辞書に渡すことができます。
- メールアドレストークンにおいてアンダースコアを認めます。(Teodor Sigaev)
- URLトークンの解析処理においてより標準互換の規則を使用します。(Tom Lane)

E.29.3.6. 関数

- 関数呼び出しにおいて、パラメータ名を提供し、それらに関数定義における名前付きパラメータに一致させることができます。(Pavel Stehule)
例えば、関数が `a` および `b` というパラメータを取るものと定義されている場合、その関数を `func(a := 7, b := 12)` や `func(b := 12, a := 7)` のように呼び出すことができます。
- UTF-8サーバ符号化方式で処理する、ロケール固有 [正規表現](#) をサポートしました。(Tom Lane)
ロケール固有の正規表現機能には、大文字小文字を区別しない一致およびロケール固有の文字クラスが含まれます。これまでは、これらの機能は (LATIN1 など) シングルバイトサーバ符号化方式を使用している場合、ASCII 以外の文字でのみ正しく動作していました。これらはまだ他の UTF-8 以外のマルチバイト符号化方式では適切ではありません。

- [to_char\(\)](#) において科学技術表記法をサポートしました。 ((['EEEE'規定](#)) (Pavel Stehule, Brendan Jurd))
- [to_char\(\)](#) がY、YYおよびYYY指定において[FM](#)(字詰めモード)を尊重するようにしました。(Bruce Momjian, Tom Lane)
YYYYではすでに尊重されていました。
- Windowsにおいて翻訳された数値および貨幣文字を正しい符号化方式で出力するように[to_char\(\)](#)を修正しました。(Hiroshi Inoue, Itagaki Takahir, Bruce Momjian)
- 多角形をまたがる["重複"](#)および"含有"演算の計算を正しくしました。(Teodor Sigaev)
多角形の`&&` (重複) 演算子はこれまで、2つの多角形を囲む矩形が重複するかどうかのみを検査していました。これがより正しく検査するようになりました。多角形の`&>`および`<&` (含有する/含有される) 演算子はこれまで、多角形の頂点がすべてもう片方に含まれているかどうかを検査していました。これは一部の非凸多角形で間違って"真"を返す可能性がありました。多角形のすべての線分がもう片方に含まれているかどうかを検査するようになりました。

E.29.3.6.1. 集約

- 集約関数で[ORDER BY](#)を使用することができます。(Andrew Gierth)
例えば[array_agg\(a ORDER BY b\)](#)がサポートされるようになりました。これは入力値の順序が重要な集約で有用であり、順序を決定するための非標準的な副問い合わせを使用する必要がなくなります。
- 複数引数を持つ集約関数が[DISTINCT](#)を使用できるようになりました。(Andrew Gierth)
- 値を単一の文字列にまとめる[string_agg\(\)](#)集約関数を追加しました。(Pavel Stehule)
- 集約遷移関数が[STRICT](#)と印付けされていない場合、[DISTINCT](#)付きで呼び出される集約関数がNULL値を渡すようになりました。(Andrew Gierth)
例えば[agg\(DISTINCT x\)](#)はNULLというx値を[agg\(\)](#)に渡す可能性があります。これは[DISTINCT](#)指定がない場合の動作との一貫性を高めます。

E.29.3.6.2. ビット列

- [bit](#)列に対する[get_bit\(\)](#)および[set_bit\(\)](#)を追加しました。これらは[bytea](#)に対するものを射影したものです。(Leonardo F)
- [bit](#)列および[bytea](#)に対する[OVERLAY\(\)](#)を実装(置き換え)しました。(Leonardo F)

E.29.3.6.3. オブジェクト情報関数

- [pg_relation_size\(\)](#) 関数よりもよりユーザ向けのインタフェースを提供する[pg_table_size\(\)](#)および[pg_indexes_size\(\)](#)を追加しました。(Bernd Helmle)
- シーケンスの権限検査のための[has_sequence_privilege\(\)](#)を追加しました。(Abhijit Menon-Sen)
- [information_schema](#)ビューをSQL:2008に従うように更新しました。(Peter Eisentraut)
- [information_schema](#)ビューが[char](#)および[varchar](#)列の最大オクテット長を適切に表示するようにしました。(peter Eisentraut)
- [information_schema](#)の権限ビューを高速化しました。(Joachim Wieland)

E.29.3.6.4. 関数およびトリガの作成

- [DO](#)文を使用した匿名コードブロックの実行をサポートしました。(Petr Jelinek, Joshua Tolley, Hannu Valtonen)
これにより一時的な関数の作成と削除を行う必要なく、サーバサイドのコードを実行できるようになりました。ユーザが関数を定義する権限を持つ、任意の言語でコードを実行することができます。
- 標準SQL互換の[列単位のトリガ](#)を実装しました。(Itagaki Takahiro)
例えばUPDATEのSETリスト内にあるなど、問い合わせにより指定された列が影響を受けた時にのみ、こうしたトリガは発行されます。情報スキーマにもこうした情報が表示されるようになりました。
- トリガの発行を行うか否かを制御できるように、[CREATE TRIGGER](#)に[WHEN](#)句を追加しました。(Itagaki Takahiro)
トリガの中で同様の検査を常に行うことができますが、外部の[WHEN](#)句で検査を行う方が性能が上がります。

E.29.3.7. サーバサイド言語

- [CREATE LANGUAGE](#)に[OR REPLACE](#)句を追加しました。(Tom Lane)
これはまだ存在しない省略可能な言語をインストールする際に役に立ちます。また、PL/pgSQLがデフォルトでインストールされるようになったことに部分的に役に立ちます。

E.29.3.7.1. PL/pgSQL サーバサイド言語

- デフォルトでPL/pgSQLをインストールします。(Bruce Momjian)
この言語を利用可能とすることに関して、セキュリティや性能を問題とする管理者は、この言語を特定のデータベースから削除することができ

ます。

- 関数内の問い合わせにおいて識別子と競合するPL/pgSQL変数名が使用される状況の取り扱いを改良しました。(Tom Lane)

デフォルトでは予想外の動作を防ぐために競合があるとエラーになるようになりました。これは`plpgsql.variable_conflict`設定パラメータや関数単位の`#variable_conflict`オプションを使用して、変数が問い合わせが提供する列を使用させるように、変更することができます。どちらの場合でもPL/pgSQLは文法的に有効でないところで変数置換を行わないようになりました。

- PL/pgSQLが独自版ではなく主レクサを使用するようにしました。(Tom Lane)

これにより確実に文字列のエスケープ処理など主システムの詳細な振舞いを正確に追従します。PL/pgSQL内で予約されたとみなされるキーワード群など、ユーザから見える詳細も当然変更されました。

- 無効なレコード参照に対する不要なエラーを発生させないようにしました。(Tom Lane)

周りの式に達する度ではなく、参照が実際に取り出された時のみにエラーが発生するようになりました。例えば、多くの方がトリガで以下を行おうとしています。

```
if TG_OP = 'INSERT' and NEW.coll = ... then
```

これは想定通りに動作しなくなりました。

- 削除された列を持つ行型を扱うPL/pgSQLの機能を改良しました。(Pavel Stehule)

- PL/pgSQL関数において入力パラメータを変数に代入することができます。(Steve Prentice)

これまでは、入力パラメータは`CONST`と宣言されたものと扱われていました。このため関数のコードではこれらを変更することができませんでした。同じ制限を持たせない他のDBMSからの関数移植を容易にするために、この制限は削除されました。入力パラメータは渡された値で初期化された局所変数のように動作するようになりました。

- PL/pgSQLにおいてエラー発生箇所の報告を改良しました。(Tom Lane)

- PL/pgSQLの`MOVE FORWARD/BACKWARD`に`count`および`ALL`オプションを追加しました。(Pavel Stehule)

- PL/pgSQLの`WHERE CURRENT OF`でカーソル変数を使用することができます。(Tom Lane)

- PL/pgSQLの`OPEN cursor FOR EXECUTE`はパラメータの使用できます。(Pavel Stehule, Itagaki Takahiro)

これは新しい`USING`句を用いて実現しました。

E.29.3.7.2. [PL/Perl](#)サーバサイド言語

- 次のPL/Perl関数を新規に追加しました。

`quote_literal()`、`quote_nullable()`、`quote_ident()`、`encode_bytea()`、`decode_bytea()`、`looks_like_number()`、`encode_array_literal()`、`encode_array_constructor()` (Tim Bunce)

- PL/Perlの初期化関数を指定する、`plperl.on_init`サーバパラメータを追加しました。(Tim Bunce)

また`plperl.on_plperl_init`および`plperl.on_plperl_u_init`を使用して、それぞれ信頼できる/信頼できない場合に特化した初期化を行うことができます。

- PL/Perlにおいて`END`ブロックをサポートします。(Tim Bunce)

現在`END`ブロックではデータベースにアクセスすることはできません。

- PL/Perlにおいて`use strict`を使用できます。(Tim Bunce)

またPerlの`strict`検査は新しい`plperl.use_strict`サーバパラメータで大域的に有効にすることができます。

- PL/Perlにおいて`require`を使用できます。(Tim Bunce)

これは基本的に、そのモジュールがロードされているかどうかを確認することを試験します。ロードされなければエラーとなります。管理者が初期化パラメータを通して前もってロードしていないモジュールをロードすることはできません。

- Perlバージョン5.10以降が使用されている場合、PL/Perlにおいて`use feature`を使用できます。(Tim Bunce)

- PL/Perlの戻り値がサーバ符号化方式において有効かどうかを検証します。(Andrew Dunstan)

E.29.3.7.3. [PL/Python](#)サーバサイド言語

- PL/PythonにUnicodeサポートを追加しました。(Peter Eisentraut)

文字列は必要に応じて、サーバ符号化方式から、サーバ符号化方式へ自動的に変換されます。

- PL/Pythonの`bytea`サポートを改良しました。(Caleb Welton)

PL/Pythonに渡される`Bytea`値は、PostgreSQLの`bytea`テキスト書式ではなく、バイナリとして表現されるようになりました。またPL/Pythonからヌルバイトを含む`Bytea`値が適切に出力されるようになりました。PL/Pythonに渡す論理値、整数値、浮動小数点値も改良されました。

- PL/Pythonにおいてパラメータおよび戻り値として`配列`がサポートされるようになりました。(Peter Eisentraut)

- SQLドメインとPythonの型との対応付けを改良しました。(Peter Eisentraut)
- PL/PythonにPython 3のサポートを追加しました。(Peter Eisentraut)

この新しいサーバサイド言語は`plpython3u`と呼ばれます。これは同じセッションでPython 2サーバサイド言語と一緒に使用することはできません。

- PL/Pythonにおけるエラー発生箇所および例外の報告を改良しました。(Peter Eisentraut)

E.29.3.8. クライアントアプリケーション

- `vacuumdb`にバキュームを行わずに解析を行う`--analyze-only`オプションを追加しました。(Bruce Momjian)

E.29.3.8.1. `psql`

- SQL文字列や識別子と同様に`psql`の**変数**の値の引用符付け処理とエスケープ処理のサポートを追加しました。(Pavel Stehule, Robert Haas)
例えば、`:'var'`は`var`の値を引用符付けしリテラル文字列としてエスケープされます。一方、`:"var"`はその値を引用符付けしSQL識別子としてエスケープされます。
- `psql`によって読み取られるスクリプトファイル内の先頭のUTF-8で符号化されたUnicodeバイトオーダー印を無視します。(Itagaki Takahiro)
これはクライアント符号化方式がUTF-8の場合に有効です。これは、大抵はWindowsですが、こうした印の挿入を要求する特定のエディタの互換性を改良します。
- `--single-transaction`を適切に尊重するように`psql --file -`を修正しました。(Bruce Momjian)
- 2つの`psql`のセッションが同時に実行している場合に`psql`のコマンドライン履歴の上書きを防止します。(Tom Lane)
- `psql`のタブ補完サポートを改良しました。(Itagaki Takahiro)
- 有効な場合、`"quiet"`モードか否かに関係なく`%timing`出力を表示します。(Peter Eisentraut)

E.29.3.8.1.1. `psql`表示

- `psql`における改行を含む列の表示を改良しました。(Roger Leigh)
この動作がデフォルトになりました。 `%pset linestyle old-ascii`を使用することで、これまでの書式を利用することができます。
- `psql`で`%pset linestyle unicode`を使用してUnicodeの装飾描画文字を使用できます。(Roger Leigh)

E.29.3.8.1.2. `psql %d`コマンド

- `%d`が指定された親テーブルから継承された子テーブルを表示するようにしました。(Damien Clochard)
`%d`では子テーブルの個数のみを表示し、`%d+`はすべての子テーブルの名前を表示します。
- `%d index_name`でインデックス列の定義を表示します。(Khee Chin)
この定義は式インデックスで有用です。
- `psql`において、`%d`ではなく`%d+`のみでビューを定義した問い合わせを表示します。(Peter Eisentraut)
常に問い合わせを含めることはあまりにも冗長であると思われました。

E.29.3.8.2. `pg_dump`

- `pg_dump/pg_restore --clean`が同時にラジオブジェクトを削除するようにしました。(Itagaki Takahiro)
- `standard_conforming_strings`が有効な場合に適切にラジオブジェクトをダンプするように`pg_dump`を修正しました。(Tom Lane)
これまでのコードでは、アーカイブファイルにダンプし`pg_restore`からスクリプト出力を生成する時に失敗しました。
- `pg_restore`がスクリプト出力を生成する時に16進書式でラジオブジェクトを出力するようになりました。
スクリプトがその後9.0より前のサーバにロードされる場合に互換性に関する問題を引き起こすかもしれません。これを回避するには代わりにサーバに直接リストアしてください。
- `pg_dump`は複合型の列に割り当てられたコメントをダンプすることができます。(Taro Minowa (Higepon))
- `pg_dump --verbose`がテキスト出力モードにおいて`pg_dump`とサーバのバージョンを出力するようにしました。(Jim Cox, Tom Lane)
これらはカスタム出力モードではすでに提供されていました。
- スイッチと省略可能なファイル名の後にコマンドライン引数が残っていた場合に、`pg_restore`が注意するようになりました。(Tom Lane)
これまでではこうした引数すべては警告なく無視されました。

E.29.3.8.3. `pg_ctl`

ブート時にpostmasterを起動するためにpg_ctlを安全に使用することができます。(Tom Lane)

これまでは、無効なpostmasterロックファイルに基づき、pg_ctl親プロセスが間違ってpostmasterが起動中であると認識することがありました。その結果データベースを起動させる際に一時的な失敗が起こりました。

- pg_ctlに(initdbを呼び出す)データベースの初期化機能を付与しました。(Zdenek Kotala)

E.29.3.9. 開発ツール

E.29.3.9.1. libpq

- 新しいlibpq関数、[PQconnectdbParams\(\)](#)および[PQconnectStartParams\(\)](#)を追加しました。(Guillaume Lelarge)

これらの関数は[PQconnectdb\(\)](#)と[PQconnectStart\(\)](#)と似たものですが、1つの文字列として必要とするすべてのオプションを提供するのではなく、接続オプションのヌル終端の配列を受け付けます。

- libpq関数、[PQescapeLiteral\(\)](#)および[PQescapeIdentifier\(\)](#)を追加しました。(Robert Haas)

これらの関数は適切に引用符付け、およびエスケープされたSQLリテラル文字列および識別子を返します。[PQescapeStringConn\(\)](#)では必要でしたが、呼び出し元で前もって文字列結果を割り当てておく必要はありません。

- ユーザ単位のサービスファイル([.pg_service.conf](#))のサポートを追加しました。これはサイト全体のサービスファイルの前に検査されます。(Peter Eisentraut)

- 指定されたlibpqサービスが見つからない場合に適切にエラーを報告します。(Peter Eisentraut)

- libpqに[TCPキープアライブに関する設定](#)を追加しました。(Tollef Fog Heen, Fujii Masao, Robert Haas)

キープアライブに関する設定はTCP接続のサーバ側ですでにサポートされていました。

- 代替方法があるプラットフォームでは、libpqにおけるSIGPIPEをブロックおよびブロック解除するための余計なシステムコールを防止します。(Jeremy Kerr)

- [.pgpass](#)が提供するパスワードが失敗した場合、エラーメッセージにそのパスワードの源を記載します。(Bruce Momjian)

- クライアント証明書ファイル内で与えられるすべてのSSL証明書をロードします。(Tom Lane)

これは間接的に署名されたSSLクライアント証明書のサポートを改良します。

E.29.3.9.2. ecpg

- ecpgに[SQLDA](#)(SQL記述子領域)サポートを追加しました。(Boszormenyi Zoltan)

- ecpgに[DESCRIBE \[OUTPUT\]](#)文を追加しました。(Boszormenyi Zoltan)

- 現在のトランザクション状態を返す[ECPGtransactionStatus](#)を追加しました。(Bernd Helmle)

- ecpgのinformix互換モードにstringデータ型を追加しました。(Boszormenyi Zoltan)

- ecpgでnewおよびold変数名を制限なく使用できます。(Michael Meskes)

- ecpgにおいてfree()内の変数を使用することができます。(Michael Meskes)

- SQL3以外のデータ型に対してecpg_dynamic_type()がゼロを返すようにしました。(Michael Meskes)

これまではOIDデータ型の否定を返しました。しかしこれは有効なOID型と混乱を招くことができました。

- 64ビットのlong型をすでに持っていたプラットフォームにおいてlong long型をサポートします。(Michael Meskes)

E.29.3.9.2.1. ecpgのカーソル

- ecpg本来のモードにてスコープ外のカーソルサポートを追加しました。(Boszormenyi Zoltan)

これによりDECLAREでOPENを呼び出した時にスコープ範囲外にある変数を使用することができます。この機能はすでにecpgのInformix互換モードに存在していました。

- ecpgにおいて動的なカーソル名が可能です。(Boszormenyi Zoltan)

- ecpgはFETCHおよびMOVEでノイズの単語FROMとINを使用することができます。(Boszormenyi Zoltan)

E.29.3.10. 構築オプション

- デフォルトでクライアントスレッドセーフを有効にします。(Bruce Momjian)

スレッドセーフに関するオプションは[configure](#)の--disable-thread-safetyにより無効にすることができます。

- Linuxのout-of-memory killerの制御のサポートを追加しました。(Alex Hunsaker, Tom Lane)

/proc/self/oom_adjによりLinuxのout-of-memory (OOM) killerを無効にすることができますが、これはpostmasterに対するOOMによる停止を無効にする際に推奨されるものです。postmasterの子プロセスに対するOOMによる停止を再度有効にすることが望まれることがあ

ります。新しい[LINUX_OOM_ADJ](#)コンパイルオプションにより子プロセスに対するkillerを再度有効にすることができます。

E.29.3.10.1. Makefile

- [world](#)、`install-world`および`installcheck-world`というMakefileのターゲットを追加しました。(Andrew Dunstan)
これらは既存の`all`、`install`、`installcheck`と似ていますが、HTML文書の構築、`contrib`の構築及び試験、サーバサイド言語と`ecpg`の試験も行います。
- データおよび文書のインストール場所の制御をPGXS Makefileに追加しました。(Mark Cave-Ayland)
- 1つのHTMLファイルもしくは1つのテキストファイルとしてPostgreSQL文書を構築するMakefileのルールを追加しました。(Peter Eisentraut, Bruce Momjian)

E.29.3.10.2. Windows

- [64ビットWindows](#)でのコンパイルと64ビットモードでの実行サポートを追加しました。(Tsutomu Yamada, Magnus Hagander)
これによりWindows上で大容量の共有メモリをとることができます。
- [Visual Studio 2008](#)を使用してサーバを構築することをサポートします。(Magnus Hagander)

E.29.3.11. ソースコード

- 配布用のtarballに含まれるtarアーカイブファイルとしてではなく、前もって構築された文書をサブディレクトリツリーに配備します。(Peter Eisentraut)
例えば、前もって構築されたHTML文書は`doc/src/sgml/html`に配備されます。 マニュアルページも同様にまとめられます。
- サーバのレクサを再入可能にしました。(Tom Lane)
PL/pgSQLによるレクサの使用において必要でした。
- メモリ割り当て速度を改良しました。(Tom Lane, Greg Stark)
- ユーザ定義の制約トリガが`pg_trigger`同様に`pg_constraint`内に項目を持つようになりました。(Tom Lane)
この変更のため、`pg_constraint.pgconstrname`が冗長になりましたので、削除されました。
- 制約の強制用のインデックスの使用をより良く提供するために`pg_constraint.conindid`および`pg_trigger.tgconstrindid`システムカタログ列を追加しました。(Tom Lane)
- 一つのオペレーティングシステムのシグナルを使用して、複数条件でバックエンドと通信できるようにしました。(Fujii Masao)
多くのシグナル条件にあるプラットフォーム固有の制約を持たずに、新しい機能を追加できるようになります。
- ソースコード試験の適用範囲を改良し、`contrib`、PL/Python、PL/Perlを含むようになりました。(Peter Eisentraut, Andrew Dunstan)
- システムテーブルの起動処理用のフラットファイルの使用を削除しました。(Tom Lane, Alvaro Herrera)
これは多くのロールとデータベースを使用する場合の性能を改良し、一部の失敗条件を取り除きます。
- "bootstrapped"カタログ用の`pg_attribute`の初期内容を自動的に生成します。(John Naylor)
これはこれらのカタログの変更を大きく簡略化します。
- `execMain.c`の外部にINSERT/UPDATE/DELETE操作処理を分けました。(Marko Tiikkaja)
更新が別のModifyTableノードで実行されるようになりました。 この変更は今後の改良のための基盤として必要とされます。
- psqlのSQLヘルプテキストの翻訳機能を簡易化しました。(Peter Eisentraut)
- 配布用tarball内のすべてファイルパスが100文字未満になるように一部のファイル名を短くしました。(Tom Lane)
一部の伸長プログラムで長いファイル名に関する問題があります。
- 新しい[ERRCODE_INVALID_PASSWORD](#) SQLSTATEエラーコードを追加しました。(Bruce Momjian)
- 作者の許可を得て、わずかに残っていた個人的なソースコード内の著作権注記を適切に削除しました。(Bruce Momjian)
個人的な著作権注記は無意味になりましたが、コミュニティはそれに対する問い合わせに時折回答しなければなりませんでした。
- PostgreSQLを永続性を持たせないモードで実行させる場合の性能向上に関する[節](#)を文書に追加しました。(Bruce Momjian)
- 依存性検査を正しく動作させ、不要な再構築を防ぐように、HTML文書構築用のMakefileルールを再構成しました。(Peter Eisentraut)
- マニュアルページを作成する際にDocbook2XではなくDocBook XSLを使用します。(Peter Eisentraut)
これはマニュアルページの構築に必要なツール群を変更します。
- PL/Perlのコード構成を改良しました。(Tim Bunce)

- PL/Perlにおけるエラー文脈報告を改良しました。(Alexey Klyukin)

E.29.3.11.1. 構築に関する新しい要求

これらの要求が配布用のtarballから構築する場合には適用されないことに注意してください。構築するために以下のプログラムが使用されたファイルがtarballには含まれているためです。

- configureを構築する場合にAutoconf 2.63が必要です。(Peter Eisentraut)
- CVSから取り出したソースから構築する場合にFlex 2.5.31以降が必要です。(Tom Lane)
- CVSから取り出したソースから構築するためにはPerlバージョン5.8以降が必要です。(John Naylor, Andrew Dunstan)

E.29.3.11.2. 移植性

- Bonjourに関してより近代的なAPIを使用します。(Tom Lane)
BonjourのサポートはOS X 10.3以降を必要とするようになりました。より古いAPIはAppleにより廃止されました。
- SuperHアーキテクチャ用のスピントックサポートが追加されました。(Nobuhiro Iwamatsu)
- GCC以外のコンパイラがインライン関数をサポートする場合にそれを使用できるようにしました。(Kurt Harriman)
- 64ビット整数のデータ型が動作しないプラットフォームのサポートを削除しました。(Tom Lane)
- プラットフォームを跨る一貫性を高めるためにLD_FLAGSの使用を再構成しました。(Tom Lane)

LD_FLAGSは実行形式ファイルと共有ライブラリの2つをリンクする場合に使用されるようになりました。実行形式ファイルをリンクする場合のLD_FLAGS_EXと共有ライブラリをリンクする場合のLD_FLAGS_SLを追加しました。

E.29.3.11.3. サーバのプログラミング

- バックエンドヘッダファイルを安全にC++でインクルードできるようにしました。(Kurt Harriman, Peter Eisentraut)
これらの変更はバックエンドのコードにおいてこれまでC++の使用を困難にしていたキーワードの競合を除去しました。しかしまだバックエンド関数でC++を使用する場合には多少複雑さが残っています。適切なところで`extern "C" { }`がまだ必要です。またメモリ管理とエラー処理はまだ解決が困難です。
- C関数を集約として呼び出されるかどうかの確認に使用される[AggCheckCallContext\(\)](#)を追加しました。(Hitoshi Harada)
- キャッシュキーの最大数をコードに直接書き込むことを防ぐためにSearchSysCache()と関連する関数の呼び出し規約を変更しました。(Robert Haas)
現時点では既存の呼び出しもまだ動作しますが、新しい形式に変更しなければ9.1以降で動作しなくなることが想定されます。
- 第4引数で非NULLを提供するためにはfastgetattr()およびheap_getattr()バックエンドマクロの呼び出しが必要です。(Robert Haas)
- 独自の型解析関数は、渡されるデータの型を決定するためにVacAttrStats.attrに依存してはいけません。
これは、背後にある列のデータ型と異なる格納方式のインデックス列上の統計情報を収集できるように変更されました。解析すべき実際のデータ型を通知する新しいフィールドが存在します。

E.29.3.11.4. サーバフック

- ColumnRefノードとParamRefノードを処理するためのパーサフックを追加しました。(Tom Lane)
- ロード可能なモジュールがユーティリティコマンドを制御できるようにProcessUtilityフックを追加しました。(Itagaki Takahiro)

E.29.3.11.5. バイナリ更新のサポート

- その場での更新をサポートする[contrib/pg_upgrade](#)を追加しました。(Bruce Momjian)
これによりPostgreSQLの新しいメジャーリリースに更新する時にデータベースのダンプ/リストア処理の必要性がなくなり、停止時間が桁違いに少なくなります。これはPostgreSQL 8.3および8.4から9.0への更新をサポートします。
- バイナリ更新中に[relfilenodes](#)リレーションの値の保護サポートを追加しました。(Bruce Momjian)
- バイナリ更新中にpg_typeとpg_enumのOIDの保護サポートを追加しました。(Bruce Momjian)
- テーブル空間内のデータファイルをそのPostgreSQLバージョンに特化したサブディレクトリに移します。(Bruce Momjian)
これはバイナリ更新を簡単にします。

E.29.3.12. Contrib

- [contrib/pgbench](#)にマルチスレッドオプション(-j)を追加しました。(Itagaki Takahiro)
これにより複数のCPUを使用したpgbench試験ができ、pgbench自体が試験のボトルネックとなる危険が軽減します。

[contrib/pgbench](#)に`Yshell`および`Ysetshell`メタコマンドを追加しました。(Michael Paquier)

- [contrib/dict_xsyn](#)に機能を追加しました。(Sergey Karpov)

新しいオプションは`matchorig`、`matchsynonyms`、`keepsynonyms`です。

- 全文検索辞書[contrib/unaccent](#)を追加しました。(Teodor Sigaev)

このフィルタ処理辞書は文字から発音記号を取り除きます。 複数言語に対する全文検索がより簡単になります。

- [contrib/dblink](#)に[dblink_get_notify\(\)](#)を追加しました。(Marcus Kempe)

これによりdblinkにおいて非同期通知を行うことができます。

- [contrib/dblink](#)の削除された列の取り扱いを改良しました。(Tom Lane)

これは[dblink_build_sql_insert\(\)](#)と関連する関数に影響します。 これらの関数は物理的な列番号ではなく論理的な列番号に従って列を番号付けするようになりました。

- [contrib/hstore](#)のデータ長に関する制限を大きく増加しました。 また、`hstore`列に対する`GROUP BY`および`DISTINCT`操作ができるようにB-treeおよびハッシュ機能を追加しました。(Andrew Gierth)

新しい関数や演算子も追加されました。 これらの改良により`hstore`はPostgreSQLに組み込まれた、すべての機能を持つキー/バリューストアになりました。

- パスワード強度に関するサイト固有のポリシーをサポートする[contrib/passwordcheck](#)を追加しました。(Laurenz Albe)

このモジュールのソースコードはサイト独自のパスワードポリシーに合わせるように変更しなければなりません。

- [contrib/pg_archivecleanup](#)ツールを追加しました。(Simon Riggs)

これは、不要になったアーカイブファイルを削除するための`archive_cleanup_command`サーバパラメータで利用されるために設計されました。

- [contrib/auto_explain](#)の出力に問い合わせテキストを追加しました。(Andrew Dunstan)

- [contrib/pg_stat_statements](#)にバッファアクセスカウンタを追加しました。(Itagaki Takahiro)

- `/proc/self/oom_adj`を使用して[Linux out-of-memory](#)(OOM) killerを無効にできるように[contrib/start-scripts/linux](#)を更新しました。(Alex Hunsaker, Tom Lane)

E.30. リリース8.4.17

リリース日: 2013-04-04

このリリースは8.4.16に対し、各種不具合を修正したものです。8.4メジャーリリースにおける新機能については、[項E.47](#)を参照してください。

E.30.1. バージョン8.4.17への移行

8.4.Xからの移行ではダンプ/リストアは不要です。

しかし、このリリースではGiSTインデックスの管理についていくつかの不具合を修正しています。このアップデートをインストールした後、以下に記述する条件の1つ以上に該当するGiSTインデックスすべてについて、`REINDEX`を実行することを推奨します。

また、8.4.10よりも前のバージョンからアップグレードする場合は、8.4.10のリリースノートを参照してください。

E.30.2. 変更点

- OpenSSLの乱数の状態を、postmasterのそれぞれの子プロセスでリセットするようにしました。(Marko Kreen)

`contrib/pgcrypto`関数によって生成された乱数が、他のデータベースユーザによって比較的推定しやすいという問題が、これにより回避されます。この危険はpostmasterが`ssl = on`で構成されている時にのみ問題となりますが、ほとんどの接続ではSSL暗号化を利用していません。(CVE-2013-1900)

- GiSTインデックスで、"fuzzy"な幾何比較を使うのが適切でないときは、使わないようにしました。(Alexander Korotkov)

幾何データ型のコアでは"fuzzy"な同値性を利用して比較を行ないますが、`gist_box_same`では正確な比較を行う必要があり、そうしなければ、それを利用するGiSTインデックスは一貫しないものになります。このアップデートをインストールした後、`box`、`polygon`、`circle`、`point`型の列のすべてのGiSTインデックスについて`REINDEX`を実行してください。これらはいずれも`gist_box_same`を使っています。

- 可変長のデータ型、つまり`text`、`bytea`、`bit`、`numeric`型の列で`contrib/btree_gist`を使うGiSTインデックスについて誤った範囲の結合と不正なロジックを修正しました。(Tom Lane)

この誤りにより、存在するキーが検索で見つからないような一貫性のないインデックスが作成されたり、インデックスが無駄に大きくなることがありました。このアップデートをインストールした後、そのようなインデックスについて`REINDEX`を実行することを推奨します。

- 複数列についてのインデックスで、GiSTページを分割するコードの不具合を修正しました。(Tom Lane)

この誤りにより、存在するキーが検索で見つからないような一貫性のないインデックスが作られることや、検索が非常に非効率になるインデックスが作られることがありました。このアップデートをインストールした後、複数列についてのGiSTインデックスについて`REINDEX`を実行することを推奨します。

- 正規表現のコンパイルで無限ループになる可能性があるのを修正しました。(Tom Lane, Don Porter)
- 正規表現のコンパイル時にNULLポインタから参照先を取得する可能性があるのを修正しました。(Tom Lane)
- `to_char()`で、それが適切な場合、ASCIIのみの大文字変換規則を利用するように修正しました。(Tom Lane)

これにより、ロケールに依存すべきでないいくつかのテンプレートのパターンにおける誤った動作が修正されます。トルコ語のロケールで"`ı`"と"`i`"が正しく処理されていませんでした。

- 1999-12-31 24:00:00というタイムスタンプが理由もなく拒絶されていたのを修正しました。(Tom Lane)
- "picksplit doesn't support secondary split"という不必要なログメッセージを削除しました。(Josh Hansen, Tom Lane)

このメッセージは何らかのコードを期待して追加されたように思われますが、そのコードは書かれませんでした。また、GiSTによる二次的分割のデフォルト処理は実際には非常に良いので、そのコードは今後も書かれないう。そのため、エンドユーザにブツブツ言うのをやめることにしました。

- セッションの最後のいくつかのコミットとアボートの件数を統計情報コレクタに送信するのに失敗する可能性があるのを修正しました。(Tom Lane)
- PL/Perlの`spi_prepare()`関数のメモリリークを修正しました。(Alex Hunsaker, Tom Lane)
- `pg_dumpall`でデータベース名に"="が含まれるものを正しく処理できるよう修正しました。(Heikki Linnakangas)
- `pg_dump`が不正な接続文字列が渡された時にクラッシュしないようにしました。(Heikki Linnakangas)
- `pg_dump`で不正なインデックスを無視するようにしました。(Michael Paquier)

不正なインデックスをダンプするとリストア時に問題が発生する可能性があります。例えば、ユニーク条件を満たさないテーブルのデータに対してユニーク条件を強制したためにインデックス作成に失敗したような場合です。また、インデックスの作成が実際にはまだ進行中の場合もありますが、このときはDDLによる更新がまだコミットされていないと考えるのが適当でしょうから、`pg_dump`がダンプする必要はありません。

- `contrib/pg_trgm`の`similarity()`関数が、トライグラムのない文字列について0を返すよう修正しました。(Tom Lane)

以前は内部でのゼロ割りによりNaNを返していました。

- 時間帯データファイルを`tzdata`リリース2013bに更新しました。チリ、ハイチ、モロッコ、パラグアイ、およびロシアのいくつかの地域での夏時間の変更が含まれます。また、数多くの場所での歴史的な地域データの修正が含まれます。

また、時間帯の略称について、ロシアおよび他の地域での以下の最近の変更を反映しました。

CHOT、GET、IRKT、KGT、KRAT、MAGT、MAWT、MSK、NOVT、OMST、TKT、VLAT、WST、YAKT、YEKTについては、それらの現在の意味に従います。VOLT (Europe/Volgograd (ヨーロッパ/ヴォルゴグラード))およびMIST (Antarctica/Macquarie (南極/マッコリー島))をデフォルトの短縮名リストに追加しました。

E.31. リリース8.4.16

リリース日: 2013-02-07

このリリースは8.4.15に対し、各種不具合を修正したものです。8.4メジャーリリースにおける新機能については、[項E.47](#)を参照してください。

E.31.1. バージョン8.4.16への移行

8.4.Xからの移行ではダンプ/リストアは不要です。

しかし、8.4.10よりも前のバージョンからアップグレードする場合は、8.4.10のリリースノートを参照してください。

E.31.2. 変更点

- SQLから`enum_recv`を実行できないようにしました。(Tom Lane)

この関数は誤った宣言をされており、単純なSQLコマンドでサーバをクラッシュさせることが可能でした。実際のところ、サーバのメモリの内容を検査するために攻撃者がこれを利用することが可能になっていました。この問題を(Secunia SVCRP経由で)報告してくれたSumit Soniに感謝します。(CVE-2013-0255)

- リレーションのファイルを切り詰めるときに、最小復旧地点を更新するようにしました。(Heikki Linnakangas)

データが捨てられた後は、タイムライン上のそれより前の地点で復旧を止めることは、もはや安全ではありません。

- 副SELECTの結果から添字付けあるいは列選択できるようSQLの文法を修正しました。(Tom Lane)

- `pg_tablespace`をスキャンするときに競合問題が発生しないようにしました。(Stephen Frost, Tom Lane)

`pg_tablespace`のエントリに複数の同時更新があるとき、`CREATE DATABASE`および`DROP DATABASE`が正しく動作しない場合があります。

- `DROP OWNED`がデータベース全体あるいはテーブル空間を削除しようとしなかったようにしました。(Álvaro Herrera)

安全のため、これらのオブジェクトは削除するのではなく、所有権を変更すべきです。

- [vacuum_freeze_table_age](#)の実装の誤りを修正しました。(Andres Freund)

[vacuum_freeze_min_age](#)の値よりも多くのトランザクションを実行したインストール環境では、この誤りにより自動バキュームが部分テーブルスキャンを実行できなくなり、その結果、必ずテーブル全体のスキャンが発生します。

- `RowExpr`あるいは`XmlExpr`が2度、解析された時に、誤った動作をしないようにしました。(Andres Freund, Tom Lane)

この誤りは`CREATE TABLE LIKE INCLUDING INDEXES`などの状況においてユーザに明らかになることがありました。

- ハッシュテーブルのサイズ計算で整数のオーバーフローが起きないようにする仕組みを改善しました。(Jeff Davis)

- `to_date()`が範囲外の日付を拒絶するようにしました。(Hitoshi Harada)

- Windowsで非ASCIIのプロンプト文字列が、確実に正しいコードページに変換されるようにしました。(Alexander Law, Noah Misch)

このバグはpsqlおよび他のいくつかのクライアントプログラムに影響がありました。

- データベースに接続していないとき、psqlの`\?`コマンドがクラッシュする可能性があったのを修正しました。(Meng Qingzhong)

- libpqの`PQprintTuples`でバッファを1バイト行き過ぎてしまう問題を修正しました。(Xi Wang)

この古い関数はPostgreSQL本体によっては使われていませんが、クライアントのプログラムでは今でも使われているかもしれません。

- ecpglibで翻訳されたメッセージを適切に使うようにしました。(Chen Huajun)

- MSVC上でecpg_compatおよびpgtypesのライブラリを適切にインストールするようにしました。(Jiang Guiqing)
- libedit/libreadlineからの誤ったエクスポートに騙されないよう、configureによる提供関数の確認の順序を調整しました。(Christoph Berg)
- Windowsのビルド番号が時とともに確実に大きくなるようにしました。(Magnus Hagander)
- Windows用にクロスコンパイルしたときに、pgxsが正しく.exeの拡張子の付いた実行プログラムを生成するようにしました。(Zoltan Borszormenyi)
- 新しい時間帯の略号`FET`を追加しました。(Tom Lane)

これは一部の東ヨーロッパの時間帯で使われています。

[前のページ](#)

リリース8.4.17

[ホーム](#)
[上に戻る](#)

[次のページ](#)

リリース8.4.15

E.32. リリース8.4.15

リリース日: 2012-12-06

このリリースは8.4.14に対し、各種不具合を修正したものです。8.4メジャーリリースにおける新機能については、[項E.47](#)を参照してください。

E.32.1. バージョン8.4.15への移行

8.4.Xからの移行ではダンプ/リストアは不要です。

しかし8.4.10より前のバージョンからアップグレードする場合は、8.4.10のリリースノートを参照してください。

E.32.2. 変更点

- `CREATE INDEX CONCURRENTLY`に関連する複数の不具合を修正しました。(Andres Freund, Tom Lane)

`CREATE INDEX CONCURRENTLY`がインデックスの`pg_index`の行の状態を変更するときに、その場にある更新を使うように修正しました。これにより、同時実行中のセッションが対象のインデックスを更新できず、それにより`CONCURRENTLY`で作成されるインデックスを破損させることになる競合条件を防ぐことができます。

また、`CREATE INDEX CONCURRENTLY`コマンドのエラーの結果として作られる不正なインデックスについて、他の様々な操作が確実に無視するように修正しました。この中で最も重要なのは`VACUUM`です。なぜなら、この不正なインデックスを修正あるいは削除するための修正の処理が実行される前に、自動バキュームが実行されることがあり得るからです。

- メモリが不足した時でも、内部ハッシュテーブルが破損しないようにしました。(Hitoshi Harada)
- 外部結合上の厳格でない等価な句についての計画作成を修正しました。(Tom Lane)

厳格でない構造を他のものと等号で結ぶ句について、プランナが誤った制約を導出する場合があります。例えば、`WHERE COALESCE(foo, 0) = 0`で`foo`が外部結合のNULLになるかもしれない側のもののときです。

- プランナについて、等価なクラスから排他制約を証明する能力を向上しました。(Tom Lane)
- ハッシュされた副計画の部分行のマッチングについて、型をまたがる場合を正しく処理できるよう修正しました。(Tom Lane)

これは`WHERE (a, b) NOT IN (SELECT x, y FROM ...)`のような複数の列についての`NOT IN`の副計画で、例えば`b`と`y`がそれぞれ`int4`型と`int8`型である、というような場合に影響します。この処理の誤りにより、そこで使われているデータ型によって、誤った結果が得られたり、あるいはクラッシュする、ということが起きていました。

- `AFTER ROW UPDATE/DELETE`トリガーで古いタブルを再フェッチする時、バッファロックを取得するようにしました。(Andres Freund)

この誤りにより、非常に特異な状況において、外部キーを強制するトリガーの事前チェックのロジックに対して誤ったデータを渡すことができました。その結果として、クラッシュを起こしたり、あるいはトリガーを実行するかどうかについて誤った判断をすることがありました。

- `ALTER COLUMN TYPE`について、継承されたチェック制約を正しく処理できるように修正しました。(Pavan Deolasee)

これは8.4より前のリリースでは正しく動作していました。この修正により8.4およびそれ以降のリリースでも正しく動作するようになりました。

- `REASSIGN OWNED`について、表領域の`GRANT`が正しく処理できるように修正しました。(Álvaro Herrera)
- ビューのシステム列についての`pg_attribute`の誤ったエントリを無視するようにしました。(Tom Lane)

ビューにはシステム列はありません。しかし、表をビューに変換するとき、そのようなエントリを取り除くことを忘れていました。これは9.3およびそれ以降のリリースでは適切に修正されますが、それ以前のブランチにおいても、誤って変換されたビューが存在しないようにする必要があります。

- ルールの出力について、`INSERT INTO table DEFAULT VALUES`を正しくダンプできるよう修正しました。(Tom Lane)

問い合わせが非常に多くのUNION/INTERSECT/EXCEPTを含むときでもスタックオーバーフローが発生しないようにしました。(Tom Lane)

- もっとも小さな整数値を-1で割り算した時に、プラットフォームによってはエラーとなるのを修正しました。(Xi Wang, Tom Lane)
- 日付文字列の解析で、文字列の終わりを越えてアクセスすることがあるのを修正しました。(Hitoshi Harada)
- Unixドメインソケットのパス名がプラットフォーム依存の最大長を超えていた場合に、理解可能なエラーメッセージを生成するようにしました。(Tom Lane, Andrew Dunstan)

以前は"Non-recoverable failure in name resolution"(名前解決で解決不能なエラーが発生しました)といった非常に不親切なエラーを起こしていました。

- 複合列の値をクライアントに送るときのメモリリークを修正しました。(Tom Lane)
- `postmaster.pid`ファイルの読み込みについて、`pg_ctl`をより堅牢にしました。(Heikki Linnakangas)

競合条件、およびファイルデスクリプタのリークを解決しました。

- `psql`において、不正にエンコードされたデータが入力され、かつ`client_encoding`の設定がSJISのようにクライアントでしか利用できない文字セットである場合にクラッシュする場合があるのを修正しました。(Jiang Guiqing)
- `pg_dump`が`tar`出力形式で生成する`restore.sql`スクリプトの不具合を修正しました。(Tom Lane)

このスクリプトはテーブル名が大文字を含む場合に全く動作しませんでした。また、このスクリプトで通常のCOPYモードだけでなく`--inserts`モードの場合もデータをリストアできるようにしました。

- `pg_restore`でPOSIX準拠の`tar`ファイル进行处理できるよう修正しました。(Brian Weaver, Tom Lane)

`pg_dump`の`tar`出力モードの元々のプログラムは、POSIX標準に完全には準拠していないファイルを生成していました。この問題はバージョン9.3で修正されます。このパッチは9.3より前のブランチを修正するもので、正しくないフォーマットと正しいフォーマットの両方を処理できるようにします。これにより、9.3がリリースされた時に互換性の問題を防ぐことができます。

- `pg_resetxlog`について、データディレクトリへの相対パスが与えられた時に、`postmaster.pid`の場所を正しく特定できるよう修正しました。(Tom Lane)

この誤りにより、データディレクトリを使用している動作中の`postmaster`があることに`pg_resetxlog`が気が付かないことがありました。

- `libpq`の`lo_import()`関数および`lo_export()`関数が、ファイルI/Oのエラーを適切に報告するよう修正しました。(Tom Lane)
 - `ecpg`について、ネストした構造体のポインタ変数の処理を修正しました。(Muhammad Usama)
 - `contrib/pageinspect`のBツリーページ検査関数で、ページの検査中はバッファロックを取得するようにしました。(Tom Lane)
 - AIX上でのロードモジュールのビルドについて`pgxs`のサポートを修正しました。(Tom Lane)
- 元々のソースツリーの外側でモジュールをビルドしてもAIXでは正しく動作しませんでした。
- 時間帯データファイルを`tzdata`リリース2012jに更新しました。キューバ、イスラエル、ヨルダン、リビア、パレスチナ、西サモア、およびブラジルの一部での夏時間の変更が含まれます。

E.33. リリース8.4.14

リリース日: 2012-09-24

このリリースは8.4.13に対し、各種不具合を修正したものです。8.4メジャーリリースにおける新機能については[項E.47](#)を参照してください。

E.33.1. バージョン8.4.14への移行

8.4.Xからの移行ではダンプ/リストアは不要です。

しかし8.4.10より前のバージョンからアップグレードする場合は8.4.10のリリースノートを参照してください。

E.33.2. 変更点

- プランナによるエグゼキュータのパラメータの代入について修正しました。また、CTEプランノードについてエグゼキュータの再スキャンの条件を修正しました。(Tom Lane)

これらの間違いにより、同じWITHの副問い合わせを複数回スキャンする問い合わせについて、間違った結果を返すことがありました。

- GiSTインデックスにおけるページ分割の決定を改善しました。(Alexander Korotkov, Robert Haas, Tom Lane)

複数列のGiSTインデックスで、この間違いにより、予期しない膨張が起きることがありました。

- 権限が取り消されても、その権限を引き続き保持している場合は、取り消しが連鎖しないようにしました。(Tom Lane)

あるロールxからグラントオプションを取り消したが、xが他の誰かから付与されたことによってそのオプションを引き続き保持している、という場合は、ロールyがxによって付与されたその権限を再帰的に取り消すべきではありません。

- PL/Perlが使われている時のSIGFPEの処理について修正しました。(Andres Freund)

PerlはプロセスのSIGFPEハンドラをSIG_IGNにリセットするため、後になってクラッシュが起きる場合があります。PL/Perlを初期化した後で、Postgresの通常のシグナルハンドラをリストアするようにしました。

- PL/Perlの再帰関数が実行時に再定義される場合に、PL/Perlがクラッシュしないようにしました。(Tom Lane)

- PL/Perlで正しく最適化されない場合があるのを回避しました。(Tom Lane)

一部のLinuxディストリビューションでは、誤ったバージョンのpthread.hがインストールされており、PL/Perlのコンパイルされたコードが誤ったものになります。この結果、PL/Perlの関数がエラーを発生させる他の関数を呼び出すと、クラッシュすることがありました。

- 時間帯データファイルをtzdataリリース2012fに更新しました。フィジーでの夏時間の変更が含まれます。

E.34. リリース8.4.13

リリース日: 2012-08-17

このリリースは8.4.12に対し、各種の不具合を修正したものです。 8.4メジャーリリースにおける新機能については[項E.47](#)を参照してください。

E.34.1. バージョン8.4.13への移行

8.4.Xからの移行ではダンプ/リストアは不要です。

しかし8.4.10より前のバージョンからアップグレードする場合には8.4.10のリリースノートを参照してください。

E.34.2. 変更点

- XML実体参照を介した外部ファイル/URLへのアクセスを防止します。(Noah Misch, Tom Lane)

`xml_parse()` は、XML値で参照されるDTDと実体参照を解決するために必要な外部ファイルやURLの入手を試みます。これにより、権限を持たないデータベースユーザがデータベースサーバの権限でデータを入手することができてしまいました。この外部データは直接ユーザには返されませんが、データが有効なXMLとして解析できなかった場合その一部がエラーメッセージの中に漏洩することがあり得ます。またどのような場合であっても、ファイルが存在するかどうか検査できるだけでも攻撃者にとって有用になり得ます。(CVE-2012-3489)

- `contrib/xml2`の`xsslt_process()`を介した外部ファイル/URLへのアクセスを防止します。(Peter Eisentraut)

`libxslt`はスタイルシートコマンドを通してファイルとURLの両方を読み書きする機能を提供します。これにより、権限を持たないデータベースユーザがデータベースサーバの権限でデータを読み書きすることができてしまいました。`libxslt`のセキュリティオプションを適切に使用することでこれを無効にします。(CVE-2012-3488)

また`xsslt_process()`の外部ファイル/URLからドキュメントとスタイルシートを取り込む機能を取り除きました。これは文書化された"機能"でしたが、推奨されないものと長くみなされていました。CVE-2012-3489に対する修正のためこの能力が壊れましたが、その修正を行うことは止め、単に取り除きました。

- 早すぎるbtreeインデックスページの回収を防止します。(Noah Misch)

読み取り専用トランザクションによるXIDの割り当てを省略できるようにした時、読み取り専用トランザクションがまだ参照している時に削除されたbtreeページを回収されてしまう可能性をもたらしました。これは間違ったインデックス検索結果という結果になります。時間に関する必要条件のため、こうしたエラーが実際に発生する可能性は非常に低いものでしたが、修正しなければなりません。

- 新しく作成された、またはリセットされたシーケンスに関するクラッシュに対する安全性を修正しました。(Tom Lane)

`ALTER SEQUENCE`が新しく作成されたシーケンスまたはリセットされたシーケンスに対して実行された場合、その後`nextval()`呼び出しが一度だけ行われ、そして、サーバがクラッシュすると、WAL再生はシーケンスを`nextval()`が行われていないように見える状態にリストアします。このため次の`nextval()`呼び出しにより最初のシーケンス値が再び返されることになります。シリアル列のシーケンス作成には`ALTER SEQUENCE OWNED BY`処理が含まれていますので、特にこれは`serial`列で現れます。

- `pg_start_backup()`の後確実に`backup_label`ファイルを`fsync`します。(Dave Kerr)

- 9.1からのバックパッチは`fsync`要求キューの圧縮を向上しました。(Robert Haas)

チェックポイント期間の性能を向上します。9.1の変更は安全に過去のバージョンにパッチを充てられると判断できる程度に実地試験が行われたようです。

- 自動バキュームは直接ブロックされたプロセスによってのみ自動キャンセルさせることができます。(Tom Lane)

元々のコードでは一部の場合に一貫性がない動作を許していました。具体的には`deadlock_timeout`猶予期間より短時間で自動バキュームはキャンセルされました。

- 自動バキュームのキャンセルについてのログ処理を改良しました。(Robert Haas)

- サーバ起動後一番最初のログローテーションの間`log_truncate_on_rotation`が動作するようにログコレクタを修正しまし

た。(Tom Lane)

- 入れ子状の集合操作 (`UNION/INTERSECT/EXCEPT`) に付与される `WITH` を修正しました。(Tom Lane)
- 副問い合わせへの行全体の参照が余計な `GROUP BY` や `ORDER BY` 列を含まないことを確実にしました。(Tom Lane)
- `CREATE TABLE` 時の `CHECK` 制約およびインデックス定義における行全体の参照の複製を許しません。(Tom Lane)

この状況は `LIKE` または `INHERITS` を持つ `CREATE TABLE` で発生することがあり得ます。複製される行全体の変数は、新しいテーブルではなく元のテーブルの行型で正しくラベル付けされませんでした。 `LIKE` では、後で行型がよく分岐しますので、このような状況を拒絶する方が合理的のように思います。 `INHERITS` では理想的には、親のテーブルの行型への暗黙的な変換によって許可すべきですが、安全に以前のバージョンにパッチを充てるにはより多くの作業が必要です。

- `ARRAY(SELECT ...)` 副問い合わせにおけるメモリリークを修正しました。(Heikki Linnakangas, Tom Lane)
- 正規表現からの共通接頭辞の抽出を修正しました。(Tom Lane)

このコードでは `^(foo)?bar` などの修飾された括弧付きの副式で混乱してしまふことがあり得ました。こうしたパターンの検索に対して不正確なインデックス最適化をもたらしました。

- `interval` 定数における、符号付き `hh:mm`、`hh:mm:ss` フィールドの解析に関する不具合を修正しました。(Amit Kapila, Tom Lane)
- `contrib/xml2` の `xslt_process()` においてエラーを適切に報告します。(Tom Lane)
- 時間帯データファイルを `tzdata` リリース 2012e に更新しました。モロッコおよびトケラウにおける夏時間の変更が含まれます。

[前のページ](#)
リリース 8.4.14

[ホーム](#)
[上に戻る](#)

[次のページ](#)
リリース 8.4.12

E.35. リリース8.4.12

リリース日: 2012-06-04

このリリースは8.4.11に対し、各種の不具合を修正したものです。 8.4メジャーリリースにおける新機能については[項E.47](#)を参照してください。

E.35.1. バージョン8.4.12への移行

8.4.Xからの移行ではダンプ/リストアは不要です。

しかし8.4.10より前のバージョンからアップグレードする場合には8.4.10のリリースノートを参照してください。

E.35.2. 変更点

- contrib/pgcryptoのDES `crypt()` 関数における不正なパスワード変換を修正しました。(Solar Designer)

パスワード文字列に0x80というバイト値が含まれる場合、パスワードの残りが無視され、見かけよりも非常に弱いパスワードになってしまいました。この修正によって、文字列の残りが適切にDESハッシュ内に含まれるようになります。したがって、この不具合の影響を受ける、格納済みのパスワード値はすべて一致なくなりますので、格納済みの値を更新しなければなりません。(CVE-2012-2143)

- 手続き言語の呼び出しハンドラにおいてSECURITY DEFINERおよびSET属性を無視します。(Tom Lane)

こうした属性を呼び出しハンドラに与えるとサーバがクラッシュするかもしれません。(CVE-2012-2655)

- 数字による時間帯オフセットによりUTCから16時間まで離れたtimestampの入力を行うことができます。(Tom Lane)

一部の歴史的な時間帯はこれまでの上限であった15時間よりも離れたオフセットを持ちます。このためリロード時にダンプされたデータ値が拒絶される結果になることがありました。

- 与えられた時刻が現在の時間帯の最後のDST遷移時刻と正確に一致する場合をうまく処理するようにタイムスタンプ変換を修正しました。(Tom Lane)

長い間これは見落とされていました。ほとんどのDSTを使用する時間帯では、将来のDST変更が無限に続くものと仮定していましたので、これまで気付きませんでした。

- textからnameへのキャストおよびcharからnameへのキャストがマルチバイト符号化方式で正しく文字列の切り詰めを行うように修正しました。(Karl Schnaitter)

- to_tsquery()のメモリコピーにおける不具合を修正しました。(Heikki Linnakangas)

- 副問い合わせ内の外部プレースホルダに関するプランナの扱いを修正しました。(Tom Lane)

この不具合は、上位の問い合わせの外部結合の内NULLを含む方から由来する変数を参照する副SELECTに影響します。9.1では、この不具合によって影響を受ける問い合わせは"ERROR: Upper-level PlaceholderVar found where not expected"というエラーで失敗します。しかし9.0と8.4では、副問い合わせに渡る値がNULLであるべき時にNULLになりませんので、警告なく間違っているかもしれない回答が得られます。

- pg_attributeが非常に大きい場合の低速なセッション起動を修正しました。(Tom Lane)

pg_attributeがshared_buffersの1/4を超える場合、セッション起動時に時々必要となるキャッシュ再構築コードが同期スキャンロジックを開始し、通常より長い期間かかるようになりました。この問題は、特に多くの新しいセッションが一度に開始する場合に深刻です。

- 適切な頻度で発生する問い合わせキャンセルに対する連続的なスキャン検査を確実に行います。(Merlin Moncurie)

有効なタプルを含まない連続するページを多くを巡るスキャンは、その間、中断に応答しません。

- WindowsのPGSemaphoreLock()の実装は、戻る前にImmediateInterruptOKを確実に消去します。(Tom Lane)

この見落としは、同じ問い合わせの中で後で受け取った問い合わせキャンセル中断が、安全ではない時に受け付けられ、予期しない、悪化した結果をもたらす可能性があったことを意味します。

- ビューまたはルールを出力する時に行全体の変数を安全に表示します。(Abbas Butt, Tom Lane)

曖昧な名前を含む（つまりその名前がテーブル名か問い合わせの列名かいずれかになる）境界条件では、曖昧な方法で出力し、ビューまたはルールがダンプとリストアの間で別の解釈がされる危険性がありました。操作を行わないキャストを付与することで曖昧な状況を防ぎます。

- 無効な符号化方式に対応するヌルマーカ文字列を正しく扱うように`COPY FROM`を修正しました。(Tom Lane)

`E'¥¥0'`などのヌルマーカ文字列は動作しなければなりませんし、以前は動作していました。この状況は8.4で壊れました。

- 自動バキュームのワーカプロセスは確実にスタック長の検査を適切に行います。(Heikki Linnakangas)

これまでは、自動`ANALYZE`によって呼び出される関数の中の無限再帰がワーカプロセスをクラッシュさせる可能性がありました。

- 高負荷時にログの干渉性が失われないようにログコレクタを修正しました。(Andrew Dunstan)

これまでは、コレクタは非常に忙しい場合、大規模メッセージの再構成に失敗する可能性がありました。

- `SIGHUP`を受信した後にファイルのローテーションを確実に再開するようにログコレクタを修正しました。(Tom Lane)

- GINインデックスに対するWAL再生ロジックを修正し、インデックスがその後に削除された場合でも失敗しないようにしました。(Tom Lane)

- PL/pgSQLの`RETURN NEXT`コマンドにおけるメモリリークを修正しました。(Joe Conway)

- 対象が関数の第一変数であった場合のPL/pgSQLの`GET DIAGNOSTICS`コマンドを修正しました。(Tom Lane)

- `psql`の拡張表示（`¥x`）モードにおいてメモリの最後にアクセスできない可能性を修正しました。(Peter Eisentraut)

- データベースに多くのオブジェクトが含まれている場合の`pg_dump`の複数の性能問題を修正しました。(Jeff Janes, Tom Lane)

データベースに多くのスキーマが含まれる場合、多くのオブジェクトが依存関係のあるループにある場合、所有されたシーケンスが多く存在する場合に、`pg_dump`が非常に低速になる可能性がありました。

- エラー時に一時データベース接続をリークしないように、`contrib/dblink`の`dblink_exec()`を修正しました。(Tom Lane)

- エラーメッセージ内で正しい接続名を報告するように`contrib/dblink`を修正しました。(Kyotaro Horiguchi)

- 時間帯データファイルを`tzdata`リリース2012cに更新しました。南極、アルメリア、チリ、キューバ、フォークランド諸島、ガザ、ハイチ、ヘブロン、モロッコ、シリア、トケラウ諸島における夏時間の変更、カナダに関する歴史的な変更が含まれます。

E.36. リリース8.4.11

リリース日: 2012-02-27

このリリースは8.4.10に対し、各種の不具合を修正したものです。8.4メジャーリリースにおける新機能については[項E.47](#)を参照してください。

E.36.1. バージョン8.4.11への移行

8.4.Xからの移行ではダンプ/リストアは不要です。

しかし8.4.10より前のバージョンからアップグレードする場合には8.4.10のリリースノートを参照してください。

E.36.2. 変更点

- `CREATE TRIGGER`においてトリガ関数に対する実行権限が必要になりました。(Robert Haas)

この検査が無かったため、別のユーザが自身が所有するテーブルにその関数をインストールすることで、偽造した入力データでトリガ関数を実行することができました。`SECURITY DEFINER`が付いたトリガ関数でのみ重大です。その他のトリガ関数ではとにかくテーブル所有者として実行されるためです。(CVE-2012-0866)

- SSL証明書内のコモンネームにおける独断的な長さ制限を取り除きました。(Heikki Linnakangas)

`libpq`とサーバの両方ともSSL証明書から取り出したコモンネームを32バイトで切り詰めていました。通常想定外の検証エラー以上のことを引き起こしませんが、ある証明書の所有者が他者になりすますことができるという、多少怪しいシナリオがあります。被害を受けるにはコモンネームが正確に32バイト長でなければならない、また、攻撃者は信頼されたCAに対してコモンネームが接頭辞としてその文字列を持つ証明書を発行するように説得しなければなりません。またサーバになりすますには、さらにクライアントからの接続を中継するための攻撃が必要です。(CVE-2012-0867)

- `pg_dump`のコメントに記述された名前の中の改行を空白に変換します。(Robert Haas)

`pg_dump`は、その出力スクリプトにおけるSQLコメント内で発行されるオブジェクト名のサニタイズに関して注意を払っていませんでした。改行を含む名前は少なくともそのスクリプトを構文的に不正にさせます。悪意をもって組み立てられたオブジェクト名によって、スクリプトがリロードする時にSQLインジェクションの危険性があり得ました。(CVE-2012-0868)

- バキューム処理と同時に挿入を行った場合の**btree**インデックス破損を修正しました。(Tom Lane)

挿入によって発生するインデックスページ分割によって、同時実行中の**VACUUM**が削除すべきインデックス項目を削除し損なうことが時々発生することがありました。対応するテーブル行が削除された後、対応先が無いインデックス項目によってエラー ("could not read block N in file ..."など)、最悪は、解放されたテーブル位置に再挿入された関係がない行にちなんだ、警告のない間違った問い合わせ結果が引き起こされます。この不具合は8.2から存在していましたが、あまり頻発に発生しませんでしたので、これまで究明されませんでした。使用中のデータベースで発生していたのではと疑わしければ、対象のインデックスを再インデックス付けすることで修正されます。

- テーブル所有者を変更する時に、テーブル単位の権限だけではなく列単位の権限も更新します。(Tom Lane)

この失敗は、事前に付与された列権限が古い所有者により付与されたものとして表示されることを意味します。つまり新しい所有者またはスーパーユーザであってもテーブル所有者を追跡できなくなった権限を取り除くことができないことを意味します。

- `ALTER USER/DATABASE SET`における一部の設定で存在しない値を許可します。(Heikki Linnakangas)

`default_text_search_config`、`default_tablespace`、`temp_tablespace`を未知の名前に設定することができます。これらが実際に使用されている別のデータベースでは既知であるかもしれませんが、またテーブル空間の場合はまだテーブル空間が作成されていないかもしれないからです。過去`search_path`においても同じ問題がありましたが、これらの設定も同様に動作するようにしました。

- コミット後のテーブルファイルの削除に問題があった場合のクラッシュを防止します。(Tom Lane)

テーブル削除は、そのトランザクションがコミットした後にのみ背後のディスク上のファイルを削除しなければなりません。(例えばファイルの権限の誤設定のため)失敗した場合、トランザクションをアボートするには遅すぎますので、

コードは単に警告メッセージを出力し継続するものと考えられます。リリース8.4においてこのロジックが壊れ、こうした状況でPANICが発生し、再起動できないデータベースとなりました。

- WAL再生中にOIDカウンタを、たとえ周回していたとしても、正しく追跡します。(Tom Lane)

これまでは、OIDカウンタはシステムが再生モードを抜けるまで高値のままになりました。実際にはほとんどnilという結果になりますが、マスタに昇格されるスタンバイサーバでは、値が必要になってから、OIDカウンタを合理的な値まで進めるために長時間かかる可能性がありました。

- *が付いた正規表現の後方参照を修正しました。(Tom Lane)

コードでは、正確な文字列一致を強制せずに、実質後方参照シンボルで参照されるパターン副式を満たす任意の文字列を受け付けました。

同様の問題はまだ、直接量指定子のサブジェクトとならない、より大きく量化された式に埋め込まれた後方参照でも残っています。こちらは将来のリリースのPostgreSQLで対応予定です。

- `inet/cidr` 値の処理に最近入ったメモリリークを修正しました。(Heikki Linnakangas)

2011年12月のPostgreSQLリリース内のパッチによって、これらの操作にメモリリークが発生しました。これらの列に対するbtreeインデックス等で重大な問題になる可能性がありました。

- SQL言語関数内の`CREATE TABLE AS/SELECT INTO`の後に対応先がなくなったポインタを修正しました。(Tom Lane)

ほとんどの場合、これはアサートが有効な構築におけるアサーション失敗という結果になるだけですが、もっと悪い結果になる可能性がありました。

- Windowsのsysloggerにおけるファイルハンドルの二重クローズを防ぎます。(MauMau)

通常はこのエラーは表面化しませんが、Windowsのデバッグ版を実行している場合は例外が引き起こります。

- `plpgsql`におけるI/O変換関連のメモリリークを修正しました。(Andres Freund、Jan Urbanski、Tom Lane)

現在の関数が終わるまで、特定の操作がメモリリークしました。

- 継承されたテーブル列に対する`pg_dump`の取り扱いを改良しました。(Tom Lane)

`pg_dump`は、子の列が親の列と異なるデフォルト式を持つという状況を間違って扱いました。デフォルトがテキストとして親のデフォルトと同一であるが、実際は同一ではない場合（例えば、スキーマ検索パスの違いのため）、異なるものとして認識せず、そのため、ダンプしリストアした後、子は親のデフォルトを継承することができました。子の列が`NOT NULL`であり親がそうではなかった場合も、微妙に間違っリストアされました。

- `INSERT`形式のテーブルデータに対する`pg_restore`の直接データベースにリストアするモードを修正しました。(Tom Lane)

他の問題に対する修正における見落としの結果、2011年9月または12月付けのリリースに含まれる`pg_restore`では、`--inserts`または`--column-inserts` オプションを付けて作成されたアーカイブファイルを直接データベースにリストアすることができませんでした。アーカイブファイル自体には失敗はなく、テキストモード出力では問題ありませんでした。

- `ecpg`の`DEALLOCATE`文において`AT`オプションを可能にしました。(Michael Meskes)

これをサポートする基盤は少し前から存在しましたが、見落としのため、エラー検査でこの状況を拒絶していました。

- `contrib/intarray`の`int[] & int[]` 演算子のエラーを修正しました。(Guillaume Lelarge)

2つの入力配列が共通して持つ最小の整数が1であり、どちらかの配列により小さな値があった場合、1が結果から間違っ省かれました。

- `contrib/pgcrypto`の`encrypt_iv()` および`decrypt_iv()` の誤判定を修正しました。(Marko Kreen)

これらの関数は特定の種類の無効入力エラーの通知に失敗し、不正な入力に対してランダムなゴミの値を代わりに返しました。

- `contrib/test_parser`における1バイトのバッファオーバーランを修正しました。(Paul Guyot)

コードでは必要より1バイト多く読み取ろうとし、境界条件ではクラッシュします。`contrib/test_parser`は単なるサンプルコードであり、これ自体はセキュリティ問題ではありませんが、例のコードとしては良くありません。

- ARMにおいて可能ならばスピロックに`__sync_lock_test_and_set()`を使用します。(Martin Pitt)

この関数は、以前の廃止予定でARMv6以降では使用できなくなったSWPB命令の使用を置き換えるものです。最近のARMボードでも古いコードは明白な方法で失敗することはありませんが、単に同時アクセスのインターロックを行わず、マルチプロセス操作において奇妙な失敗をもたらすと報告されています。

- 受け付け可能なバージョンのgccを用いて構築する場合に`-fexcess-precision=standard`オプションを使用します。(Andrew Dunstan)

最近のバージョンのgccは独創的な結果を生成するというさまざまなシナリオを防ぎます。

- FreeBSDにおいてスレッド化されたPythonを使用できるようにしました。(Chris Rees)

configureスクリプトはこれまで、この組み合わせは動作しないと前提していました。しかしFreeBSDではこの問題が修正されたので、エラー検査を取り除きました。

[前のページ](#)

リリース8.4.12

[ホーム](#)
[上に戻る](#)

[次のページ](#)

リリース8.4.10

E.37. リリース8.4.10

リリース日: 2011-12-05

このリリースは8.4.9に対し、各種の不具合を修正したものです。8.4メジャーリリースにおける新機能については[項E.47](#)を参照してください。

E.37.1. バージョン8.4.10への移行

8.4.Xからの移行ではダンプ/リストアは不要です。

しかし、`information_schema.referential_constraints`ビュー定義において長期に渡って潜在したエラーが見つかりました。このビューからの正確な結果に依存している場合、後述の変更点の最初で説明する通りその定義を置き換えなければなりません。

また8.4.8より前のバージョンからアップグレードする場合は、8.4.8のリリースノートを参照してください。

E.37.2. 変更点

- `information_schema.referential_constraints`ビューの不具合を修正しました。(Tom Lane)

このビューでは、外部キーと依存するプライマリキーまたは一意キー制約との対応付けについて十分な注意を行っていませんでした。このため、外部キー制約がまったく表示されない、何回も表示される、あるいは、実際の依存する制約とは異なる制約に依存するものと言い張るという結果になることがありました。

このビュー定義はinitdbによってインストールされますので、ただアップグレードしただけでは問題は解消されません。既存のインストールでこの問題を解消させなければならない場合、（スーパーユーザとして）`information_schema`スキーマを削除して、`SHAREDIR/information_schema.sql`を元に再作成することができます。（`SHAREDIR`の場所が分からない場合は`pg_config --sharedir`を実行してください。）これは修正対象のデータベースそれぞれで繰り返さなければなりません。

- GINインデックス更新に関するWALレコードの不正確な再生を修正しました。(Tom Lane)

このためクラッシュ後またはホットスタンバイサーバにおいて、インデックス項目の検索に過渡的に失敗する結果になりました。しかし、この問題はインデックスに対する次の`VACUUM`で修復されます。

- `CREATE TABLE dest AS SELECT * FROM src`または`INSERT INTO dest SELECT * FROM src`時のTOAST関連のデータの破損を修正しました。(Tom Lane)

テーブルが`ALTER TABLE ADD COLUMN`によって変更された場合、データを逐語的に別のテーブルにコピーしようとした時に特定の境界条件で破損した結果になることがありました。この問題は、8.4以降において、この構文においてのみ明確に発生し得たものですが、同じ不具合を引き起こす他のコードパスがある場合に備えて、より過去のバージョンにおいてもパッチを適用しました。

- 無効なsyscache項目からTOASTテーブルにアクセスする時の競合条件を修正しました。(Tom Lane)

典型的な兆候は、"missing chunk number 0 for toast value NNNNN in pg_toast_2619"のような一時的なエラーです。ここで挙げられるTOASTテーブルは常にシステムカタログに属します。

- パラメータのデフォルト式で使用する項目について関数の依存関係を追跡します。(Tom Lane)

これまでは、参照されるオブジェクトが、参照する関数を削除または変更することなく、削除されることがあり得ました。このため関数を使用される時に誤動作することになりました。単にこの更新をインストールするだけでは依存関係が消失した項目が修正されないことに注意してください。修正するためには、その後こうした関数それぞれに`CREATE OR REPLACE`を行う必要があります。デフォルト値が組み込み以外のオブジェクトに依存する関数があれば、これを行うことを勧めます。

- 複数のOUTパラメータを持つ、集合を返すSQL関数をインライン化することができます。(Tom Lane)

- `DatumGetInetP()` が1バイトヘッダを持つinetデータを展開するようにしました。さらに展開を行わない、新しいマクロ`DatumGetInetPP()`を追加しました。(Heikki Linnakangas)

この変更がコアコードに影響することはありませんが、`DatumGetInetP()` が通常の規約の通りに展開したデータを生成す

ることを想定しているアドオンのコードにおけるクラッシュを防止できるかもしれません。

- `money` 型の入出力におけるロケールのサポートを改良しました。(Tom Lane)

すべての標準 `lc_monetary` 書式付けオプションをサポートしていないという点以外にも、入力関数と出力関数に一貫性がなかったため、ダンプされた `money` 値では再度読み取ることができないロケールがありました。

- `transform_null_equals` が `CASE foo WHEN NULL ...` 式に影響を与えないようにしました。(Heikki Linnakangas)

`transform_null_equals` は、`CASE` 構文によって内部的に生成された等価な検査ではなく、直接ユーザによって書かれた `foo = NULL` に影響を与えることのみを前提としています。

- 自己参照外部キーをより良くサポートするために、外部キートリガの生成順序を変更しました。(Tom Lane)

カスケードしている外部キーが自身のテーブルを参照する場合、1 行の更新が 1 つのイベントで `ON UPDATE` トリガと `CHECK` トリガの両方を発行します。 `ON UPDATE` トリガを最初に行う必要があります。 そうしないと、`CHECK` は最終的ではない行の状態を検査することになり、不適切なエラーが発生する可能性があります。 しかしこれらのトリガの発行順序はその名前によって決まります。 トリガは `"RI_ConstraintTrigger_NNNN"` という規約に従った名前が自動生成されますので、一般的には作成順序によって決まります。 適切な修正にはこの規約の変更が必要です。 9.2 で変更する予定ですが、既存のリリースで変更することには危険があるように思われます。 このため、このパッチはトリガの作成順序を変更するだけです。 この種のエラーが起きたユーザは、トリガを正しい順序にするために、外部キー制約を削除し再作成しなければなりません。

- バッファ割当率を追跡する際の浮動小数アンダーフローを防止します。(Greg Matthews)

これ自体は害がないものですが、特定のプラットフォームではこれにより煩わしいカーネルログメッセージが現れます。

- Windows における子プロセス起動時設定ファイルの名前と行番号を保持します。(Tom Lane)

これまでこれらは `pg_settings` ビューで正しく表示されませんでした。

- `psql` のコマンド履歴内で空行を含むコマンドを保持します。(Robert Haas)

これまでの動作では、例えば文字列リテラル内の空行が削除された場合に問題が起きました。

- テーブルの行型など自動生成された型の間でユーザが定義したキャストをダンプするように `pg_dump` を修正しました。(Tom Lane)

- PL/Perl の構築時に好みのバージョンの `xsubpp` を使用します。 オペレーティングシステムのメインコピーである必要はありません。(David Wheeler, Alex Hunsaker)

- `contrib/dict_int` および `contrib/dict_xsyn` 内の間違ったコードを修正しました。(Tom Lane)

一部の関数で、`palloc()` で返されるメモリはゼロ埋めされていることが保証されていると間違った前提をしていました。

- `pgstatindex()` において問い合わせキャンセル割込みを適切に受け付けます。(Robert Haas)

- `VPATH` 構築ですべてのサーバヘッダファイルが適切にインストールされることを確実にしました。(Peter Eisentraut)

- 冗長エラーメッセージ内で報告されるファイル名を短くしました。(Peter Eisentraut)

通常の構築では常に C ファイルの名前だけがエラーメッセージ呼び出しの中で報告されました。 しかしこれまで `VPATH` 構築では絶対パス名が報告されました。

- 中央アメリカ用の Windows の時間帯名の解釈を修正しました。(Tom Lane)

通常中央アメリカではまったく夏時間が認められませんので、`"Central America Standard Time"` を `CST6CDT` ではなく `CST6` に対応付けしました。

- 時間帯データファイルを `tzdata release 2011n` に更新しました。 ブラジル、キューバ、フィジィ、パレスチナ、ロシア、サモアにおける夏時間規則の変更、アラスカ、イギリス領東アフリカの歴史的な修正が含まれています。

E.38. リリース8.4.9

リリース日: 2011-09-26

このリリースは8.4.8に対し、各種の不具合を修正したものです。 8.4メジャーリリースにおける新機能については[項E.47](#)を参照してください。

E.38.1. バージョン8.4.9への移行

8.4.Xからの移行ではダンプ/リストアは不要です。

しかし8.4.8より前のバージョンからアップグレードする場合は、8.4.8のリリースノートを参照してください。

E.38.2. 変更点

- HOT更新されたか疑わしいタプルのインデックス処理における不具合を修正しました。(Tom Lane)

この不具合のために、システムカタログの再インデックス処理の後インデックスが破損する可能性があります。 ユーザインデックスには影響しないはずです。

- GiSTインデックスページ分割処理における複数の不具合を修正しました。(Heikki Linnakangas)

発生する可能性は小さいのですが、インデックスの破損を招く可能性があります。

- `tsvector_concat()` におけるバッファオーバーランの可能性を修正しました。(Tom Lane)

この関数は必要なメモリを少なく見積もり、その結果サーバクラッシュを招く可能性があります。

- `xml_recv` において"standalone"パラメータ処理時のクラッシュを修正しました。(Tom Lane)

- `pg_options_to_table` を値を持たないオプションに対してNULLを返すようにしました。(Tom Lane)

これまではこのような場合にサーバクラッシュしてしまいました。

- `ANALYZE` およびSJIS-2004符号化変換においてメモリの最後を超えてアクセスする可能性を修正しました。(Noah Misch)

これは非常に可能性が低いサーバクラッシュ状況を修正します。

- 起動プロセスとbgwriterプロセスの相互作用における断続的なハングを防止します。(Simon Riggs)

これはホットスタンバイ以外の状況でのリカバリに影響します。

- `relcache`初期ファイルの無効化における競合状態を修正しました。(Tom Lane)

新しいバックエンドプロセスが無効な初期ファイルを読み取ろうとするが、データが無効であることを通知する`inval`メッセージを取り損なう期間がありました。 この結果カタログアクセスにおいて、起動処理の後の"could not read block 0 in file ..."といった奇妙な失敗が起こります。

- GiSTインデックススキャン終了時のメモリリークを修正しました。(Tom Lane)

すでに多くの行を持つテーブルに対して新たなGiSTを基にした排他制約を作成する際の検証など、多くの分割GiSTインデックススキャンを行うコマンドが一時的にこのリークの原因となる大容量のメモリを求めることがあり得ました。

- 保持可能カーソルと`plpgsql`の`RETURN NEXT`コマンドをサポートするタプルストアにおける(メモリ膨張をもたらすかもしれない)不正なメモリ計上を修正しました。(Tom Lane)

- 大規模かつ非可逆なビットマップを構築する時の性能問題を修正しました。(Tom Lane)

- 一意列の結合選択性推定を修正しました。(Tom Lane)

結合結果のサイズ推定を悪化させる可能性がある、間違ったプランナの発見的手法を修正します。

- 副選択対象リスト内でのみ出現する、入れ子状の`PlaceholderVar`式を修正しました。(Tom Lane)

この間違いの結果、外側の結合の出力がNULLとして不正に現れます。

- 入れ子状のEXISTS問い合わせを適切に最適化できます。(Tom Lane)

- 配列作成、経路作成関数のパッド用バイトを確実にゼロにするように修正しました。(Tom Lane)

これにより、プランナが意味的に等しい定数を等しくないと思なし、最適化を悪化させる一部の状況を防ぎます。

- inner-indexscan副計画内のResultノードの開閉を取り扱うようEXPLAINを修正しました。(Tom Lane)

この見落としのよくある兆候は"bogus varno"エラーです。

- WAL再生を壊すgcc 4.5.0の不具合を回避します。(Tom Lane)

サーバクラッシュ後にコミットされたトランザクションの損失をもたらす可能性がありました。

- ビュー内のVALUESについてのダンプ不具合を修正しました。(Tom Lane)

- シーケンスに対するSELECT FOR UPDATE/SHAREを許しません。(Tom Lane)

この操作は想定通りに動作せず、また、失敗をもたらすことがあり得ました。

- つねにpg_class.reltuples/reldpagesを更新するようにVACUUMを修正しました。(Tom Lane)

自動バキュームによる、いつテーブルをバキュームするかについての決定をだんだんと悪化させる、いくつかの状況を修正します。

- ハッシュテーブルのサイズ計算時の整数オーバーフローから保護します。(Tom Lane)

- CLUSTERがすでに削除されたTOASTデータへアクセスしようとする状況を修正しました。(Tom Lane)

- "peer"認証に関する証明書制御メッセージの使用における移植性不具合を修正しました。(Tom Lane)

- 複数の往復が必要な場合のSSPIログインを修正しました。(Ahmed Shinwari, Magnus Hagander)

この問題の典型的な兆候はSSPIログイン期間の"The function requested is not supported"エラーです。

- pg_hba.confにhostsslが含まれているがSSLが無効である場合にエラーを発生します。(Tom Lane)

以前の警告なくこうした行を無視するという動作より、この方がよりユーザフレンドリであると判断しました。

- pg_srand48 シード初期化における記述ミスを修正しました。(Andres Freund)

このため、提供されたシードの全ビットを使用すると失敗する可能性がありました。この関数はほとんどのプラットフォームで使用されていません(srandomを持たないプラットフォームのみです)ので、想定よりもランダム性が小さいシードによるセキュリティ露見の可能性はとにかく最小だったと思われます。

- LIMITとOFFSET値の合計が2^63を超えた場合の整数オーバーフローを防止します。(Heikki Linnakangas)

- generate_series()のint4版とint8版にオーバーフロー検査を追加しました。(Robert Haas)

- to_char()における末尾のゼロ除去を修正しました。(Marti Raudsepp)

小数点の後の桁位置がないFM付きの書式において、小数点より左のゼロが正しく除去できていませんでした。

- 2^63近辺の入力に対するオーバーフローを防ぐようにpg_size_pretty()を修正しました。(Tom Lane)

- plpgsqlのレコード値におけるtypmodマッチ検査を弱めました。(Tom Lane)

過度に積極的な検査によって、保持しなければならない長さ修飾子が破棄されることがあり得ました。

- initdbの間、ロケール名内の引用符を正しく扱います。(Heikki Linnakangas)

この問題は、"People's Republic of China"などの、いくつかのWindowsのロケールにて起こり得ました。

- 8.3からのアップグレードにおいて、TOASTテーブルのrelfrozenxidsを維持するようにpg_upgradeを修正しました。(Bruce Momjian)

これに失敗すると、アップグレードの後pg_clogファイルがあまりに早く削除される可能性がありました。

pg_ctlにおいて、Windowsのサービス登録用の静寂モードをサポートしました。(MauMau)

- 異なるファイルからCOPYしている時の、psqlのスクリプトファイル行番号の計数処理を修正しました。(Tom Lane)
- standard_conforming_stringsに合わせて、pg_restoreの直接データベースモードを修正しました。(Tom Lane)

pg_restoreが、standard_conforming_stringsがonに設定されたアーカイブファイルからデータベースサーバに直接リストアする時に間違ったコマンドを発行する可能性があります。

- 並行pg_restoreをサポートしない状況をよりユーザフレンドリにしました。(Tom Lane)

この変更により、何らかのリストア作業が発生する前にこうした状況を検知し報告することが確実にになりました。

- libpqのLDAPサービス検索コードにおけるwrite-past-buffer-endとメモリリークを修正しました。(Albe Laurenz)
- libpqにおいて、非ブロッキングI/OとSSL接続を使用する場合の失敗を防止します。(Martin Pihlak、Tom Lane)
- libpqの接続開始期間のエラーの取扱いを改良しました。(Tom Lane)

具体的には、SSL接続開始期間のfork() 失敗についてのサーバ報告への応答がより健全になりました。

- SSL失敗に関するlibpqのエラー報告を改良しました。(Tom Lane)
- 元々サーバ問い合わせから得られたPGresultに新しいタプルを追加する時に、クラッシュする可能性を避けるようにPQsetvalue() を修正しました。(Andrew Chernow)

- ecpglibがdouble値を15桁の精度で書き込むようにしました。(Akira Kurosawa)

- ecpglibにおいて、エラー後確実にLC_NUMERICの設定を元に戻します。(Michael Meskes)

- blowfishの符号付き文字に関する不具合(CVE-2011-2483)に対する上流の修正を適用しました。(Tom Lane)

contrib/pg_cryptoのblowfish暗号化コードは、charが符号付きであるプラットフォーム(ほとんどのプラットフォーム)において間違った結果を生成し、暗号化されたパスワードが本来より脆弱になりました。

- contrib/segにおけるメモリリークを修正しました。(Heikki Linnakangas)
- 空のインデックスに対して一貫性を持った結果を生成するようにpgstatindex() を修正しました。(Tom Lane)
- perl 5.14を用いたビルドを可能にしました。(Alex Hunsaker)
- システム関数の存在を検出するためのconfigureスクリプトの方法を更新しました。(Tom Lane)

8.3と8.2で使用しているautoconfのバージョンは、リンク時の最適化を行うコンパイラによってだまされることがあり得ました。

- 空白を含むファイルパスでのビルドとインストールに関連した問題を修正しました。(Tom Lane)
- 時間帯データファイルをtzdata release 2011iに更新しました。 カナダ、エジプト、ロシア、サモア、南スーダンにおける夏時間規則の変更が含まれています。

E.39. リリース8.4.8

リリース日: 2011-04-18

このリリースは8.4.7に対し、各種の不具合を修正したものです。8.4メジャーリリースにおける新機能については[項E.47](#)を参照してください。

E.39.1. バージョン8.4.8への移行

8.4.Xからの移行ではダンプ/リストアは不要です。

しかし、使用しているインストレーションが以前のバージョンからpg_upgradeを用いてアップグレードしたものであれば、pg_upgradeにおいて今回修正された不具合のためにデータを失う可能性を防止するための操作を行わなければなりません。推奨する解消方法はすべてのTOASTテーブルに対してVACUUM FREEZEを行うことです。詳しくはhttp://wiki.postgresql.org/wiki/20110408pg_upgrade_fixを参照してください。

また8.4.2以前のバージョンから更新する場合は8.4.2のリリースノートを参照してください。

E.39.2. 変更点

- pg_upgradeのTOASTテーブルの取り扱いを修正しました。(Bruce Momjian)

pg_upgradeの間、TOASTテーブルに対するpg_class.relfrozenxid値が正しく新しいインストレーションにコピーされていませんでした。このため実行後、TOASTテーブル内のタプルを有効にするために必要であるのに、pg_clogファイルが破棄されてしまう結果となり、そのため"could not access status of transaction"エラーをもたらすことがあり得ました。

このエラーは、pg_upgradeを用いてアップグレードしたインストレーションにおいて、データ損失という重大な危険を引き起こします。このパッチは、今後pg_upgradeの利用した場合で問題が起こらないようにするものです。不具合のあるバージョンのpg_upgradeで処理されたインストレーションにおける問題を治すものではありません。

- 間違った"PD_ALL_VISIBLE flag was incorrectly set"という警告を抑制します。(Heikki Linnakangas)

VACUUMが時々実際のところ有効な場合でもこの警告を発していました。

- 複合型自身を複合型に含めることを許可しません。(Tom Lane)

これは、複合型を処理する際にサーバが無限再帰をしてしまう状況を防止します。こうした構造の使い道は一部ありますが、常に安全に動作することを確実にするために必要な作業量を正当化するほどの説得力はないと思われます。

- カタログキャッシュの初期化中のデッドロックの可能性を取り除きました。(Nikhil Sontakke)

一部のキャッシュを読み取るコードが、インデックスのカタログをロックする前にシステムインデックスに対して共有ロックを獲得する場合があります。これは他でより標準的な順序で排他ロックを獲得しようとする処理とデッドロックする可能性があります。

- 対象テーブルに対する同時更新があった時のBEFORE ROW UPDATEトリガ処理における、不正な領域を指し示すポインタ問題を修正しました。(Tom Lane)

この不具合は、UPDATE RETURNING ctidを実行しようとした時に、断続的に"cannot extract system attribute from virtual tuple"で失敗するという結果で気付きました。更新したテーブルに対して不正なインデックス項目が生成されてしまうなど、より深刻なエラーとなる可能性がごくわずかながら存在します。

- 遅延トリガイベントを待たせているテーブルに対するDROP TABLEを許可しません。(Tom Lane)

以前はDROPが進み、トリガが最終的に発行された時に"could not open relation with OID nnn"エラーをもたらしました。

- GEQO最適化中、常に偽のWHERE条件によって引き起こるクラッシュを防止します。(Tom Lane)

- 半結合とアンチ結合の場合のプランナの取り扱いを改良しました。(Tom Lane)

- 全文検索の選択性推定においてNULLを考慮するように修正しました。(Jesper Krogh)

- 削除された列を持つ行型を扱うPL/pgSQLの機能を改良しました。(Pavel Stehule)

これは以前9.0でなされた修正を過去のバージョンに反映したものです。

- PL/Pythonの部分配列化に関するメモリリークを修正しました。(Daniel Popowich)
- TOCファイル内の（1KB以上の）長い行に対処するようにpg_restoreを修正しました。(Tom Lane)
- コンパイラの過度に積極的な最適化によるゼロ除算が原因のクラッシュに対して、より多くの対策を取りました。(Aurelien Jarno)
- MIPS上のFreeBSDとOpenBSDでdlopen()の使用をサポートしました。(Tom Lane)

これらのシステムではこのシステム関数はMIPSハードウェアで利用できないという仮定を直接書き込んだコードがありました。より最近のバージョンでは利用できますので、コンパイル時の試験を使用するように変更しました。

- HP-UXにおけるコンパイル失敗を修正しました。(Heikki Linnakangas)
- Windowsにおけるlibintlのバージョン非互換問題を修正しました。(Hiroshi Inoue)
- Windowsのビルドスクリプト内のxcopyの使用がWindows 7で正しく動作するように修正しました。(Andrew Dunstan)

ビルドスクリプトのみに影響し、インストールや使用には影響しません。

- Cygwinにおいてpg_regressで使用されるパス区切り文字を修正しました。(Andrew Dunstan)
- 時間帯データファイルをtzdata release 2011fに更新しました。チリ、キューバ、フォークランド諸島、モロッコ、サモア、トルコにおける夏時間規則の変更、南オーストラリア、アラスカ、ハワイの歴史的な修正が含まれています。

[前のページ](#)
リリース8.4.9

[ホーム](#)
[上に戻る](#)

[次のページ](#)
リリース8.4.7

E.40. リリース8.4.7

リリース日: 2011-01-31

このリリースは8.4.6に対し、各種の不具合を修正したものです。 8.4メジャーリリースにおける新機能については[項E.47](#)を参照してください。

E.40.1. バージョン8.4.7への移行

8.4.Xからの移行ではダンプ/リストアは不要です。 しかし8.4.2以前のバージョンから更新する場合は8.4.2のリリースノートを参照してください。

E.40.2. 変更点

- EXPLAINがCASE式の簡易構文を表示しようとした時の失敗を防止します。(Tom Lane)

CASEのテスト式が定数の場合、プランナはCASEを式表示用コードを混乱させる形式に単純化してしまいました。 その結果"unexpected CASE WHEN clause"というエラーになりました。

- 過去に存在した添字範囲に対する部分配列代入を修正しました。(Tom Lane)

新しく追加される添字と過去に存在した添字の先頭との間に隙間があった場合、コードは古い配列のヌルビットマップからコピーしなければならない項目数を誤計算してしまい、データ破損またはクラッシュを導く可能性がありました。

- プランナにおける、非常に時間が離れた日付値に対する想定外の変換オーバーフローを防止します。(Tom Lane)

date型はtimestamp型で表現可能な範囲よりもより広い日付範囲をサポートします。 しかしプランナは常に問題なくdateからtimestampへの変換が可能であることを仮定していました。

- standard_conforming_stringsが有効な場合のラージオブジェクト (BLOB) 用のpg_restoreのテキスト出力を修正しました。(Tom Lane)

データベースへ直接リストアすることは正しく動作しましたが、pg_restoreがSQLテキスト出力を求め、かつ、standard_conforming_stringsが元のデータベースで有効だった場合、文字列エスケープは正しくありませんでした。

- ... & !(subexpression) | ...を含むtsquery値の間違った解析を修正しました。(Tom Lane)

こうした演算子の組み合わせを含む問い合わせは正しく実行されませんでした。 同じエラーがcontrib/intarrayのquery_int型とcontrib/ltreeのltxtquery型に存在しました。

- query_int型に対するcontrib/intarray入力関数におけるバッファオーバーランを修正しました。(Apple)

この関数の返すアドレスが上書きされる可能性があるという点で、この不具合はセキュリティ問題です。 この問題を報告し修正版を提供していただいたApple Incのセキュリティチームに感謝します。(CVE-2010-4015)

- contrib/segのGiST picksplitアルゴリズムにおける不具合を修正しました。(Alexander Korotkov)

これにより、seg列上のGiSTインデックスにおいて実際に不正な結果になることはありませんが、非常に非効率的な結果になることがあり得ました。 こうしたインデックスがある場合、この更新版をインストールした後にREINDEX処理を行うことを検討してください。(これは過去の更新のcontrib/cubeで修正された不具合と同じです。)

E.41. リリース8.4.6

リリース日: 2010-12-16

このリリースは8.4.5に対し、各種の不具合を修正したものです。8.4メジャーリリースにおける新機能については[項E.47](#)を参照してください。

E.41.1. バージョン8.4.6への移行

8.4.Xからの移行ではダンプ/リストアは不要です。しかし8.4.2以前のバージョンから更新する場合は8.4.2のリリースノートを参照してください。

E.41.2. 変更点

- Linuxにて`wal_sync_method`のデフォルトが強制的に`fdatasync`になるようにしました。(Tom Lane, Marti Raudsepp)
長い間、Linuxでのデフォルトは実質`fdatasync`でした。しかし最近のカーネルの変更により、PostgreSQLは代わりに`open_datasync`を選択するようになりました。この選択は性能向上をまったくもたらず、特定のシステム、特に`data=journal`マウントオプションを使用した`ext4`では、完全に失敗してしまいます。
- GINインデックスに対するWAL再生ロジックにおける分類不具合を修正しました。(Tom Lane)
これは"bad buffer id: 0"という失敗、あるいはレプリケーション中のインデックス内容の破損という結果になり得ました。
- 開始チェックポイントWALレコードが再実行時点と同じWALセグメント内に存在しない場合のベースバックアップからのリカバリを修正しました。(Jeff Davis)
- 複数のワーカが有効なまま長時間残った場合の自動バキュームの永続的な速度低下を修正しました。(Tom Lane)
多くのテーブルを処理している場合、自動バキュームワーカの`vacuum_cost_limit`の効果は、極端な速度低下をもたらすほぼゼロまで落ちることがあり得ました。
- IA64においてレジスタスタックのオーバーラン検知サポートを追加しました。(Tom Lane)
IA64アーキテクチャには2つのハードウェアスタックがあります。スタックオーバーラン失敗を完全に防ぐには両方を検査しなければなりません。
- `copyObject()`におけるスタックオーバーフロー検査を追加しました。(Tom Lane)
非常に複雑な問い合わせが与えられた場合のスタックオーバーフローのため、特定のコードパスがクラッシュする可能性があります。
- 一時GiSTインデックスにおけるページ分割検知を修正しました。(Heikki Linnakangas)
例えば挿入が行われた時に一時インデックスをスキャンする開いているカーソルがあると、そのインデックスで"同時実行"ページ分割を持つことができます。GiSTではこの状況の検知に失敗していました。このためGiSTはそのカーソルの実行が続く間、間違った結果をもたらすことがありました。
- 接続処理の初期段階における検査エラーを修正しました。(Tom Lane)
子プロセスがあまりに多くある場合のこの検査が一部の状況で飛ばされていました。このため固定長配列に新しい子プロセスを追加しようとする時にpostmasterのクラッシュする可能性があります。
- ウィンドウ関数の効率を向上しました。(Tom Lane)
多数のタプルを前もって読み取る必要があるが、`work_mem`がそれらすべてをメモリ内に保持できるほど大きいという、特定の状況では予期しないほど低速になりました。特に`percent_rank()`、`cume_dist()`、`ntile()`ではこの問題が当てはまります。
- 複雑なインデックス式をANALYZEする時のメモリリークを防ぎます。(Tom Lane)
- 行全体のVarを使用するインデックスがそのテーブルに依存し続けることを確実にしました。(Tom Lane)

`create index i on t (foo(t.*))`のように宣言されたインデックスはそのテーブルが削除された時に自動的に削除されませんでした。

- 複数のOUTパラメータを持つSQL関数を"インライン"化しません。(Tom Lane)

結果の想定行型に関する情報が存在しないためにクラッシュする可能性を防ぎます。

- ORDER BY、LIMIT、FOR UPDATE、WITHがINSERT ... VALUESのVALUES部に付いた場合の動作を正しくしました。(Tom Lane)

- COALESCE() 式の定数折りたたみを修正しました。(Tom Lane)

プランナは時々、実際には決して達しない副式の評価を行い、想定外のエラーをもたらす可能性がありました。

- GSSAPIサポート付きでコンパイルされたpostmasterで、接続受け入れ (accept()) またはその後すぐになされる呼び出しのいずれかが失敗した時のpostmasterのクラッシュを修正しました。(Alexander Chernikov)

- log_temp_filesが有効である時の一時ファイルの削除失敗を修正しました。(Tom Lane)

ログメッセージ出力試行時にエラーが起きた場合、削除が行われませんでした。このため一時ファイルが累積してしまいました。

- InhRelationノードに関する機能性出力を追加しました。(Tom Lane)

これによりdebug_print_parseが有効な場合に特定種類の問い合わせが行われた場合の失敗を防止します。

- 点から水平線分への距離計算を正しく修正しました。(Tom Lane)

この不具合は様々な複数の幾何距離測定演算子に影響します。

- ecpgにおけるトランザクション状態の計算を正しく修正しました。(Itagaki Takahiro)

- 再帰もしくはエラー修復状態にならない"単純な"式についてのPL/pgSQLの扱いを修正しました。(Tom Lane)

- PL/Pythonの集合を返す関数の扱いを修正しました。(Jan Urbanski)

集合結果を生成するイテレータ内でSPI関数呼び出しが失敗しました。

- contrib/cubeのGiST picksplitアルゴリズムの不具合を修正しました。(Alexander Korotkov)

cube列上のGiSTインデックスにおいて、実際に不正確な結果になることはありませんが、このため相当効率性が劣化する可能性があります。こうしたインデックスがある場合、この更新をインストールした後にREINDEXすることを検討してください。

- contrib/dblink内で、新しい接続を作成する時を除き、"identifier will be truncated"注意を出力しないようにしました。(Itagaki Takahiro)

- contrib/pgcryptoにおいて公開キーがない場合にコアダンプする可能性を修正しました。(Marti Raudsepp)

- contrib/xml2のXPath問い合わせ関数におけるメモリリークを修正しました。(Tom Lane)

- 時間帯データファイルをtzdata release 2010oに更新しました。フィジィ、サモアにおける夏時間規則の変更、香港の歴史的な修正が含まれています。

E.42. リリース 8.4.5

リリース日: 2010-10-04

このリリースは8.4.4に対し、各種の不具合を修正したものです。 8.4メジャーリリースにおける新機能については[項E.47](#)を参照してください。

E.42.1. バージョン8.4.5への移行

8.4.Xからの移行ではダンプ/リストアは不要です。 しかし8.4.2以前のバージョンから更新する場合は8.4.2のリリースノートを参照してください。

E.42.2. 変更点

- PL/PerlおよびPL/Tclにおいて、呼び出し元のSQLユーザIDごとに別のインタプリタを使用します。(Tom Lane)

この変更は、同じセッション内でその後に異なるSQLユーザIDにより実行されるPerlまたはTclコード（例えばSECURITY DEFINERの内部）を破壊することで発生する可能性があるというセキュリティ問題を防止するものです。 ほとんどのスクリプト用言語は、対象とする関数で呼び出される標準関数や演算子の再定義など、これを行うことができる多くの方法を提供します。 この変更がないと、PerlまたはTcl言語の使用権限を持つ任意のSQLユーザは基本的に、対象の関数の所有者が持つSQL権限でできることを何でもできるようになります。

この変更の欠点はPerlおよびTcl関数間の内部通信がより難しくなるという点です。 逃げ道を提供するために、PL/PerlUおよびPL/TclU関数ではまだセッション当たり1つのインタプリタのみを使用します。 これらのすべての関数は従来よりデータベーススーパーユーザの信頼レベルで実行されますので、これはセキュリティ問題とはみなされません。

信頼された実行を提供することを主張するサードパーティ製の手続き言語は似たようなセキュリティ問題を持つ可能性が大きくあります。 セキュリティが非常に重要な目的のために依存している手続き言語があれば、その作者に連絡を取ることを勧めます。

この問題を指摘いただいたTim Bunceに感謝します(CVE-2010-3433)。

- pg_get_expr() の用途として意図されたシステムカタログ列の一つではない引数で呼び出されることを拒絶することでpg_get_expr() 内のクラッシュする可能性を防止します。(Heikki Linnakangas, Tom Lane)
- Windowsにおいて終了コード128 (ERROR_WAIT_NO_CHILDREN) を致命的でないものと扱います。(Magnus Hagander)

高負荷状況下で、Windowsプロセスはこのエラーコードで起動に失敗することが時々あります。 以前はpostmasterはパニック状態としてこれを扱い、データベース全体を再起動しました。 しかしこれは過度は反応のように思われます。

- 不正なプレースホルダ位置の評価を修正しました。(Tom Lane)

この不具合のため外部結合の内側が出力リスト内に制限がない式を持つ副選択であった場合など、問い合わせ出力がNULLであるべき時に非NULLの結果になるかもしれませんでした。

- UNION ALLのメンバーレシーションの二重スキンの可能性を修正しました。(Tom Lane)
- "cannot handle unplanned sub-select"エラーを修正しました。(Tom Lane)

これは、副選択が他の副選択を含む式に展開される結合の別名参照を含む場合に発生しました。

- ビューまたは副選択への参照を持ち、かつ、入れ子の副選択内に現れる、行全体の変数の誤操作を修正しました。(Tom Lane)
- 型を跨るIN比較の誤操作を修正しました。(Tom Lane)

プランナがsort-then-unique-then-plain-join計画でIN結合を実装しようとした場合に、この失敗になり得ました。

- tsvector列に対するANALYZE統計情報の計算を修正しました。(Jan Urbanski)

元々のコードでは不正な統計情報を生成し、後で性能が悪い計画を選択するようになり得ました。

`array_agg()`、`string_agg()`、その他同様の集約関数で使用するメモリのプランナの推定を改良しました。(Hitoshi Harada)

これまでの極端な下方推定により、不適切なハッシュ集約計画が選択されるため、メモリ不足エラーが起こり得ました。

- キャッシュされた計画を一時的なものと印付けする失敗を修正しました。(Tom Lane)

`CREATE INDEX CONCURRENTLY`が参照されるテーブルの1つに対して進行している時に計画が準備された場合、インデックスの使用準備ができた後に再度計画を作成するものと仮定されています。これが確実に起こりませんでした。

- まれに報告されたB-tree失敗状況の一部をPANICからERRORを減らしました。また出力されるエラーメッセージに追加の詳細を加えました。(Tom Lane)

これにより、破損したインデックスに対するシステムの堅牢性が向上したはずです。

- GINインデックスを用いた部分一致問い合わせにおける間違った検索ロジックを修正しました。(Tom Lane)

複数のGINインデックス条件をAND/ORの組み合わせた場合、常に正しい答えを出力しませんでした。また時々必要以上に低速になりました。

- 自動バキューム処理中のクラッシュから`show_session_authorization()`を防止します。(Tom Lane)
- 返される行のすべてが実際に同一の行型でない、レコードの集合を返す関数を防御します。(Tom Lane)
- 副トランザクションのロールバック中に待ち状態のトリガイベントリストが破損する可能性を修正しました。(Tom Lane)

これはクラッシュまたは不正なトリガ発行という結果をもたらしました。

- 参照渡し関数の結果をハッシュ化する時に失敗する可能性を修正しました。(Tao Ma, Tom Lane)
- マージ結合における、結合列内のNULLの処理を改良しました。(Tom Lane)

NULLがソートで先になるようなソート順の場合、最初のNULLに達した時にマージ結合が完全に停止することができるようになりました。

- ロックファイル (`postmaster.pid`およびソケット用のロックファイル) を書き出す時に、注意してその内容のfsyncします。(Tom Lane)

これがないと、`postmaster`の起動後しばらくしてマシンがクラッシュした場合に、ロックファイルの内容が破損することがあり得ました。これが起きると、このロックファイルを手作業で削除するまで、その後の`postmaster`の起動試行が成功できなくなります。

- 過度に入れ子になった副トランザクションに対してXIDを割り当てる際の再帰を防止します。(Andres Freund, Robert Haas)

元々のコードでは、制限されたスタック領域であった場合にクラッシュすることがあり得ました。

- `walwriter`プロセス内で古いWALセグメントを開いたまま保持しないようにしました。(Magnus Hagander, Heikki Linnakangas)

これまでのコードでは、不要となったセグメントの削除を防止しました。

- バックエンド起動の初期にゴミを生成することがあった`log_line_prefix`の%iエスケープを修正しました。(Tom Lane)
- TOASTテーブルに対して部分的に指定されたりレーションオプションの誤解釈を防止します。(Itagaki Takahiro)

特に、他の`reloption`がテーブルに対して設定されていた場合`fillfactor`はゼロとして読み取られ、深刻な膨張をもたらしました。

- `ALTER TABLE ... ADD CONSTRAINT`における継承数の追跡を修正しました。(Robert Haas)
- アーカイブ処理が有効な場合、`ALTER TABLE ... SET TABLESPACE`におけるデータ破損の可能性を修正しました。(Jeff Davis)
- `CREATE DATABASE`と`ALTER DATABASE ... SET TABLESPACE`を問い合わせキャンセルにより中断させることができます。(Guillaume Lelarge)
- 指定されたインデックス式が不変かどうかに関する`CREATE INDEX`の検査を改良しました。(Tom Lane)

- 演算子クラスと演算子族を扱う `REASSIGN OWNED` を修正しました。(Asko Tiidumaa)
- 2つの空の `tsquery` 値を比較する際にコアダンプする可能性を修正しました。(Tom Lane)
- `%` に続く `_` を持つパターンに対する `LIKE` の処理を修正しました。(Tom Lane)
以前にもこれを修正しましたが、正しく処理されない場所が一部残っていました。
- 0001-01-01 AD以前のユリウス日の入力を再度許します。(Tom Lane)
8.4より前では `'J100000'::date` などの入力が動作しました。しかし、エラー検査を加えたことで意図せず壊れていました。
- カーソルに対して繰り返す `FOR` ループ内でそのカーソルが閉ざされた場合に、PL/pgSQLがクラッシュせずにエラーを発生させるように修正しました。(Heikki Linnakangas)
- PL/Pythonにおいて、`PyObject_AsVoidPtr` および `PyObject_FromVoidPtr` からのヌルポインタ結果を防止します。(Peter Eisentraut)
- libpqにおいて、`host` と `hostaddr` の両方が指定された場合の完全なSSL証明書検証を修正しました。(Tom Lane)
- 自動コミットモードが無効な場合、`DISCARD ALL` をトランザクションブロック内に包むべきものではないコマンドとしてpsqlが認識するようにしました。(Itagaki Takahiro)
- SQL/MEDオブジェクトに対するpg_dump処理におけるいくつかの問題を修正しました。(Tom Lane)
意図されていないスーパーユーザ以外により実行された場合、pg_dumpは常に失敗することに注意してください。
- pg_dumpおよびpg_restoreにおけるシークできないアーカイブファイルの処理を改良しました。(Tom Lane, Robert Haas)
これは並行リストアを適切に動作させるために重要です。
- 選択的リストアを対処する、並行pg_restore機能(`-L` オプション)を改良しました。(Tom Lane)
元々のコードでは、`-L` ファイルがデフォルト以外のリストア順序を指定していた場合に失敗しがちでした。
- `RETURNING` 句からのデータを正しく処理するようにecpgを修正しました。(Michael Meskes)
- ecpgにおけるメモリリークをいくつか修正しました。(Zoltan Boszormenyi)
- 削除された列を含むテーブルに対するcontrib/dblinkの処理を改良しました。(Tom Lane)
- contrib/dblinkにおいて"duplicate connection name"エラー後の接続リークを修正しました。(Itagaki Takahiro)
- 62バイトより長い接続名に関するcontrib/dblinkの処理を正しく修正しました。(Itagaki Takahiro)
- `hstore(text, text)` 関数をcontrib/hstoreに追加しました。(Robert Haas)
この関数は、廃止予定となった`=>`演算子に対する推奨の代替です。将来を見据えたコードが古めのバージョンのサーバでも使用できるようにするために過去にもパッチが適用されました。このパッチは、contrib/hstoreのインストール後もしくは特定のデータベースに再インストールされた後のみ効果を持つことに注意してください。代わりに手作業で`CREATE FUNCTION` コマンドを実行することを選ぶユーザがいるかもしれません。
- 構築手順や文書を更新して、ソースコードリポジトリがCVSからGitに変更されたことを反映しました。(Magnus Hagander他)
- 時間帯データファイルをtzdata release 2010Iに更新しました。エジプト、パレスチナにおける夏時間規則の変更、フィンランドの歴史的な修正が含まれています。
またこの変更では、2つのミクロネシアの時間帯を追加しています。Pacific/ChuukはPacific/Trukより好まれるようになります(好まれる省略形はTRUTではなくCHUT)、Pacific/PohnpeiはPacific/Ponapeより好まれるようになります。
- Windowsの"N. Central Asia Standard Time"時間帯をAsia/AlmatyではなくAsia/Novosibirskに対応付けるようにしました。
MicrosoftはKB976098の時間帯更新からこの時間帯の夏時間の振舞いを変更しました。Asia/Novosibirskはこの新しい振舞いにより合致しています。

E.43. リリース 8.4.4

リリース日: 2010-05-17

このリリースは8.4.3に対し、各種の不具合を修正したものです。 8.4メジャーリリースにおける新機能については[項E.47](#)を参照してください。

E.43.1. バージョン8.4.4への移行

8.4.Xからの移行ではダンプ/リストアは不要です。 しかし8.4.2以前のバージョンから更新する場合は8.4.2のリリースノートを参照してください。

E.43.2. 変更点

- `Safe.pm`を使用するのではなく、インタプリタ全体に適用される`opmask`を使用して`plperl`を強制的に制限します。(Tim Bunce, Andrew Dunstan)

最近の開発により`plperl`を信頼できるとする根拠とするには`Safe.pm`が非常に安全ではないことを知らされました。 この変更により、`Safe.pm`の使用がなくなり、常に適用されるopcodeマスクを持つ別のインタプリタを使用するようになります。 この変更の喜ばしい副作用として、`plperl`内で普通の方法でPerlの`strict` プラグマが使用できるようになりました。 また、Perlの`$a`と`$b`がソート関数で想定される動作を行うようになり、関数のコンパイルが非常に高速になりました。(CVE-2010-1169)

- PL/Tclにおいて`pltcl_modules`経由で信頼できないコードが実行されることを防止しました。(Tom)

PL/TclのデータベーステーブルからTclコードを自動読み込みする機能はトロイの木馬攻撃によりつけこまれる可能性があります。 そのテーブルの作成や挿入できるユーザに対する制限がなかったためです。 この変更により、`pltcl_modules`がスーパーユーザにより所有されていない限り、この機能は無効になります。(しかし、このテーブルに対する権限は検査されませんので、安全性が低いモジュールを本当に必要とするインストレーションでは、スーパーユーザ以外に適切な権限を付与することができます。) また、`pltclu`関数を本当に実行しようとしていない限り、制限がない"通常の"Tclにコードが読み込まれることを防止しました。(CVE-2010-1170)

- `ALTER ... SET TABLESPACE`をWAL再生する時のデータ破損を修正しました。(Tom)

`archive_mode`が有効な場合、`ALTER ... SET TABLESPACE`はWAL記録を生成しますが、その再生ロジックが正しくありませんでした。 これにより間違った場所にデータを書き出すことがあり、おそらく修復できないデータ破損が発生する可能性があります。 データ破損はスタンバイのスレーブ側で観測されますが、`ALTER`をコミットしてから次のチェックポイントに達する前にデータベースクラッシュが発生し、リカバリが発生するとマスタ側でも発生してしまいます。

- `relcache`項目の再構築中にキャッシュ再設定メッセージを受け取った場合にクラッシュする可能性を修正しました。(Heikki)

8.4.3で関連する失敗を修正する時に、このエラーが入ってしまいました。

- 関数の言語検証関数を実行する時に関数毎のGUC設定を適用します。(Itagaki Takahiro)

これは、その設定がないと関数のコードが無効になる場合の失敗を防ぎます。 例えば`search_path`が正しくないとSQL関数は解析できません。

- `constraint_exclusion = partition`の時、継承された`UPDATE`および`DELETE`の対象テーブルに対して制約による除外を実行します。(Tom)

見落としのためこれまでは、`SELECT`コマンド内で検査される制約による除外のみがこの設定で実行されました。

- 非特権ユーザがスーパーユーザのみのパラメータ設定を再設定することを許しません。(Alvaro)

これまで、非特権ユーザが自身に対する`ALTER USER ... RESET ALL`または自身が所有するデータベースに対する`ALTER DATABASE ... RESET ALL`を行った場合、そのユーザまたはデータベースに対する特別なパラメータ設定は、たとえばスーパーユーザのみにより変更可能なものとして提供されるものであっても、すべて削除されました。 `ALTER`はそのユーザが変更権限を持つパラメータのみを削除するようになりました。

- `CONTEXT`追加がログ項目になされる時にシャットダウンが発生した場合に、バックエンドのシャットダウン時にクラッシュ

する可能性を防ぎました。(Tom)

現在のトランザクションがログメッセージを出力する時点に達した時にすでにロールバックされているために文脈出力関数が失敗することがありました。

- `recovery_end_command`内の`%r`に対する取り扱いについてのエラーを修正しました。(Heikki)
値が常にゼロになっていました。
- アーカイブ処理ができる限り高速に`archive_command`の変更に対応することを確実にしました。(Tom)
- `case`式が行を返さない問い合わせであった場合に失敗しないように、`pl/pgsql`の`CASE`文を修正しました。(Tom)
- `pl/perl`の`ppport.h`を最近のPerlバージョンに更新しました。(Andrew)
- `pl/python`内の雑多なメモリリークを修正しました。(Andreas Freund, Tom)
- `ecpg`において空文字列の接続パラメータ属性を扱います。(Michael)
- 変数展開が自身を参照している際の`psql`における無限再帰を防止します。(Tom)
- `%copy (select ...)`内のドットの前後に空白を追加しないように、`psql`の`%copy`を修正しました。(Tom)
数値リテラルにおいて10進数の小数点の前後に空白が追加された場合、構文エラーが発生します。
- `client_encoding`と一致しないロケール文脈内で実行している時の`psql`における書式付けの失敗を防ぎます。(Tom)
- `contrib/intarray`演算子を使用する満足されない問い合わせにおいて、不要な"GIN indexes do not support whole-index scans"エラーを修正しました。(Tom)
- `contrib/pgstattuple`関数がキャンセル割り込みに即座に応答することを確実にしました。(Tatsuhito Kasahara)
- サーバ起動時に、既存の共有メモリセグメントに対して`shmget()`が`EINVAL`を返す状況を適切に扱うようにしました。(Tom)
この振舞いはOS Xを含むBSD派生のカーネルにおいて観測されました。共有メモリ要求サイズが大きすぎると警告の、完全に間違った起動失敗という結果になりました。
- Windowsにおけるsysloggerプロセス内のクラッシュする可能性を防ぎます。(Heikki)
- Windowsレジストリにおいて不完全な時間帯情報をより堅牢に扱います。(Magnus)
- Windowsにおいて既知の時間帯名称の集合を更新しました。(Magnus)
- 時間帯データファイルを`tzdata release 2010j`に更新しました。アルゼンチン、南極におけるオーストラリア、バングラデシュ、メキシコ、モロッコ、パキスタン、パレスチナ、ロシア、シリア、チュニジアにおける夏時間規則の変更、および、台湾の歴史的な修正が含まれます。
また、デフォルトの時間帯省略形の集合に`PKST`(パキスタン夏時間)を追加しました。

E.44. リリース 8.4.3

リリース日: 2010-03-15

このリリースは8.4.2に対し、各種の不具合を修正したものです。8.4メジャーリリースにおける新機能については[項E.47](#)を参照してください。

E.44.1. バージョン8.4.3への移行

8.4.Xからの移行ではダンプ/リストアは不要です。しかし8.4.2以前のバージョンから更新する場合は8.4.2のリリースノートを参照してください。

E.44.2. 変更点

- SSL接続においてセッションキーの再調停をどの程度の頻度で行うかを制御する`ssl_renegotiation_limit`設定パラメータを新しく追加しました。(Magnus)

これをゼロに設定することで、再調停を完全に無効にすることができます。これは壊れたSSLライブラリを使用している場合に必要になる可能性があります。具体的には一部のベンダは再調停の試行を失敗させるCVE-2009-3555に対する一時しのぎのパッチを提供しています。

- バックエンド起動時のデッドロックの可能性を修正しました。(Tom)
- `relcache`の再読み込みがきれいに行われている間にエラーを扱わないことが原因でクラッシュする可能性を修正しました。(Tom)
- 保管された計画への宙ぶらりんのポインタを使用することが原因でクラッシュする可能性を修正しました。(Tatsuo)
- `ROLLBACK`に関して、保管された計画の熱心すぎる無効化が原因でクラッシュする可能性を修正しました。(Tom)
- 副トランザクション起動時における失敗からのリカバリを試行する時にクラッシュする可能性を修正しました。(Tom)
- セーブポイントの使用およびサーバの符号化方式と異なるクライアント符号化方式に関連したサーバのメモリリークを修正しました。(Tom)
- GiSTインデックスページ分割のリカバリ終了時の整理において生成される不正なWALデータを修正しました。(Yoichi Hirai)

これはインデックスの破損という結果となり、WAL再生中にエラーが発生します。もし運が悪ければ、未完全のGiST挿入に成功した後にリカバリ終了時の整理が行われている時にクラッシュしてしまいます。

- GINインデックスに対するWAL redoの整理方式における不具合を修正しました。(Heikki)
- GINインデックス検索における走査キーの不正な比較を修正しました。(Teodor)
- `bit`型の`substring()`では、いかなる負の長さも"文字列の残りすべて"を意味するようにしました。(Tom)

これまでのコードでは-1のみを上のように扱い、他の負の値についてはクラッシュする可能性がある無効な結果値を生成しました。(CVE-2010-0442)

- 出力ビット幅が指定された整数よりも広く、それが8ビットの倍数以外である場合、先頭の端数のバイトを正確に扱うように整数からビット列への変換を修正しました。(Tom)
- 正規表現マッチが異常なまでに低速になる一部の状況を修正しました。(Tom)
- 削除された列を含む複合型の集合を返すSQL関数をインライン化する際に発生する不具合を修正しました。(Tom)
- 複合型の配列の列の要素フィールドを更新する場合の不具合を修正しました。(Tom)
- `EXPLAIN`が`FieldStore`または`ArrayRef`の代入式を表示する必要がある時の失敗を防止します。(Tom)

こうした状況は現在、`EXPLAIN VERBOSE`が計画ノードの対象リストを表示しようとした場合に発生します。

- `UNION/INTERSECT/EXCEPT`内の副問い合わせで未修飾のリテラル文字列が現れる、一部の状況における不要な強制失敗を防

止します。(Tom)

これは、8.4より前までで動作していた一部の退行を修正します。

- 行全体の変数が削除された列を含む行型を持つ一部の状況において、望まない行型互換性検査の失敗を防止します。(Tom)
- 終了位置が正確にセグメント境界であった場合、次のWALセグメント名を報告するようにバックアップ履歴ファイル内の `STOP WAL LOCATION` を修正しました。(Itagaki Takahiro)
- 常にカタログIDを `CREATE FOREIGN DATA WRAPPER` で指定されたオプション検証関数に渡します。(Martin Pihlak)
- 一時ファイルが漏洩する一部の状況を修正しました。(Heikki)

これは前のマイナーリリースでもたらされた問題を修正します。失敗する状況の1つは集合を返す `plpgsql` 関数が他の関数の例外ハンドラ内で呼び出された場合です。

- `FULL JOIN ON FALSE` のサポートを追加しました。(Tom)

これは、定数対偽の結合条件に簡略化できるようになった一部の問い合わせについての8.4より前のリリースからの退行を防止します。

- 論理値変数の場合の制約による排他処理を改良しました。具体的には、`"bool_column = false"` 制約を持つパーティションを除外することができるようになりました。(Tom)
- `INOUT` キャストをバイナリ互換を表すものとして扱うことを防止します。(Heikki)
- 列レベル権限の付与または取り下げができない場合の警告メッセージに列名を含めます。(Stephen Frost)

これにより、以前よりもより有用になり、`REVOKE` が複数のメッセージを生成する場合の混乱防止を助けます。以前は重複して出現しました。

- `pg_hba.conf` とその関連ファイルを読み取る時、`@` が引用符記号の内部にある場合に `@something` をファイル包含要求と扱わないようにしました。また、`@` 自身をファイル包含要求として扱いません。(Tom)

ロールまたはデータベース名が `@` から始まる場合の一定しない動作を防止します。空白文字を含むパス名のファイルを含む必要がある場合、まだ実現させることはできませんが、式全体を引用符でくくるのではなく `@"/path to/file"` と記述しなければなりません。

- ディレクトリが `pg_hba.conf` とその関連ファイル内で含有対象と指定されていた場合の一部のプラットフォームにおける無限ループを防止します。(Tom)
- `SSL_read` または `SSL_write` が `errno` を設定することなく失敗した場合に無限ループする可能性を修正しました。(Tom)

一部の報告では `openssl` の一部のWindows版で発生する可能性があります。

- 正しく動作させるためにホスト名が必要ですので、ローカル接続ではGSSAPI認証を許しません。(Magnus)
- 想定外の文字列の解放を行うアプリケーションから `ecpg` を保護します。(Michael)
- 接続が消失した場合 `ecpg` に適切な `SQLSTATE` を報告させるようにしました。(Michael)
- `psql` の `\d` 出力におけるセル内容の変換を修正しました。(Heikki)
- `latex` および `troff` 出力書式では行うべきではない文字列の書式付けを行わないように `psql` の `numericlocale` オプションを修正しました。(Heikki)
- `psql` における問い合わせ毎の小さなメモリリークを修正しました。(Tom)
- `ON_ERROR_STOP` と `--single-transaction` の両方が指定され、かつ、暗黙的な `COMMIT` 時にエラーが起きた場合に、`psql` が正しい終了コード(3)を返すようにしました。(Bruce)
- `pg_dump` における外部サーバの権限出力を修正しました。(Heikki)
- 範囲外の依存性IDにより並列 `pg_restore` がクラッシュする可能性を修正しました。(Tom)
- 複合列が `NULL` に設定される場合に `plpgsql` が失敗することを修正しました。(Tom)
- PL/PerlUからのPL/Perl関数の呼び出し、およびその逆の場合において失敗する可能性を修正しました。(Tim Bunce)

- コンパイラ固有の誤動作が起こる可能性を防ぐためにPL/Python内に`volatile`記号を追加しました。(Zdenek Kotala)
- PL/Tclは確実にTclインタプリタを完全に初期化します。(Tom)

このミスの既知の兆候は、Tcl 8.5以降を使用している場合のTclの`clock`コマンドの誤動作だけです。

- `ExecutorEnd`が失敗したトランザクションまたは副トランザクションで作成されたポータルで実行されることを防止します。(Tom)

`contrib/auto_explain`を使用する場合に問題になることが分かっています。

- `dblink_build_sql_*`関数であまりに多くのキー列が指定された場合の`contrib/dblink`のクラッシュを防止します。(Rushabh Lathia, Joe Conway)

- `contrib/ltree`操作においてゼロ次元の配列を許可します。(Tom)

この状況は以前はエラーとして拒絶されました。しかしこれを要素ゼロの配列と同様に扱う方がより簡便です。特にこれは、`ltree`演算子が`ARRAY(SELECT ...)`の結果に適用され、副問い合わせが行を返さない場合の不要な失敗を防止します。

- ずさんなメモリ管理によって引き起こる`contrib/xml2`の雑多なクラッシュを修正しました。(Tom)
- Windowsにおいて`contrib/xml2`の構築をより堅牢にしました。(Andrew)
- Widowsのシグナル処理における競合状態を修正しました。(Radu Ilie)

この不具合の兆候として既知のものは、負荷が高い状況において`pg_listener`内の行が削除されることです。

- Cコンパイラが64ビット整数データ型の動作を提供しない場合に`configure`スクリプトが失敗を報告するようにしました。(Tom)

この状況は既に打破されており、もはやサポートする価値がないようです。ですので代わりに`configure`時に単に拒絶します。

- 時間帯データファイルを`tzdata release 2010e`に更新しました。バングラデシュ、チリ、フィジー、メキシコ、パラグアイ、サモアにおける夏時間規則の変更が含まれます。

[前のページ](#)

リリース 8.4.4

[ホーム](#)
[上に戻る](#)

[次のページ](#)

リリース 8.4.2

E.45. リリース 8.4.2

リリース日: 2009-12-14

このリリースは8.4.1に対し、各種の不具合を修正したものです。8.4メジャーリリースにおける新機能については[項E.47](#)を参照してください。

E.45.1. バージョン8.4.2への移行

8.4.Xからの移行ではダンプ/リストアは不要です。しかしハッシュインデックスが存在する場合、8.4.2に移行した後に、破損している可能性があるためそれを修復するためにREINDEXを行わなければなりません。

E.45.2. 変更点

- セッション固有の状態を変更するインデックス関数により引き起こされる間接的なセキュリティ脅威から保護します。(Gurjeet Singh, Tom)
この変更により不変インデックス関数がスーパーユーザのセッションを破壊できることを防ぎます。(CVE-2009-4136)

- ヌルバイトが埋め込まれたコモンネーム (CN) フィールドをもつSSL証明書を拒絶します。(Magnus)
これは、SSL検証においてサーバまたはクライアント名に対する証明書が不正に一致しないようにします。(CVE-2009-4034)

- ハッシュインデックス破損を修正しました。(Tom)

8.4では、ハッシュインデックスはハッシュ値でソートして項目を保持するように変更されましたが、順序を保持するためにバケット分割および圧縮処理の更新に失敗します。このためこうした操作を行うアプリケーションは、永続的なインデックスの破損をもたらす可能性があります。これをうまく扱うためには、更新をインストールした後にハッシュインデックスすべてをREINDEXすることを推奨します。

- バックエンド起動時のキャッシュ初期化の間でクラッシュする可能性を修正しました。(Tom)

- 空のシンソラス辞書におけるクラッシュを防止しました。(Tom)

- 安全ではない時点でのVACUUMを中断するシグナルを防止します。(Alvaro)

この修正により、VACUUM FULLがすでにタプルの移動をコミットした後にキャンセルされた場合のパニックや、通常のVACUUMがテーブルを切り詰めた後に割り込みを受けた場合の一時的なエラーを防止します。

- ハッシュテーブルサイズの計算における整数オーバーフローによるクラッシュの可能性を修正しました。(Tom)

これは、ハッシュ結合の結果のプランナの推定サイズが極端に大きい場合に発生することがありました。

- 内部的な依存関係をもつオブジェクトに対する DROPが試行された場合のクラッシュを修正しました。(Tom)

- inet/cidrの比較における非常にまれなクラッシュを修正しました。(Chris Mikkelsen)

- 準備されたトランザクションで保持されるタプルレベルの共有ロックを確実に無視しないようにしました。(Heikki)

- 副トランザクション内でアクセスされるカーソルで使用される一時ファイルの早すぎる削除を修正しました。(Heikki)

- 新しいCSVログファイルに変更する時のsysloggerプロセスのメモリリークを修正しました。(Tom)

- pg_hba.confの再解析時のpostmasterのメモリリークを修正しました。(Tom)

- Windowsにおける権限ダウングレードロジックを修正しました。(Jesse Morris)

これは、Windowsにおいて"could not locate matching postgres executable"といった誤解を招くエラーメッセージでよくデータベースの起動が失敗する状況の一部を修正します。

- 主問い合わせにおけるFOR UPDATE/SHAREがWITH問い合わせに伝搬されないようにしました。(Tom)

たとえば

```
WITH w AS (SELECT * FROM foo) SELECT * FROM w, bar ... FOR UPDATE
```

では、FOR UPDATEはbarに影響し、fooには影響しなくなります。これは、以前の8.4の動作より有用かつ一貫性があります。以前はFOR UPDATEをWITHまで伝搬させようとし、常に雑多な実装上の制限により失敗していました。また、WITH問い合わせは主問い合わせと独立して実行されるという設計規則にも従います。

- そのすぐ内側に別のWITH RECURSIVEを持つWITH RECURSIVE問い合わせの不具合を修正しました。(Tom)

- ハッシュインデックスにおける同時実行性に関する不具合を修正しました。(Tom)

同時実行の挿入によりインデックス走査が一時的に間違った結果を報告することがありました。

- GiSTインデックスページ分割において、分割がインデックスの最初の列以外に依存する場合の不正なロジックを修正しました。(Paul Ramsey)

- fastupdateが有効な場合、複数列に対するGINインデックスの検索結果が不正となる点を修正しました。(Teodor)

- GINインデックスにおけるWAL項目の生成における不具合を修正しました。(Tom)

これらの不具合は、`full_page_writes`が有効な場合は現れることはありませんが、無効にした場合、次のチェックポイント以前にクラッシュが起きた場合、WAL再生は確実に失敗しました。

- チェックポイントの最後で古いWALファイルの回収や削除が失敗した場合にエラーを出力しません。(Heikki)

致命的ではないものと扱い、チェックポイントを完了させることができるようにする方が優れています。将来のチェックポイントでは削除を再試行します。こうした問題は通常の操作では想定されませんが、うまく設計されていないWindowsのアンチウイルスソフトウェアやバックアップソフトウェアにより現れることがあります。

- WindowsにおいてWALファイルが繰り返しアーカイブされないようにしました。(Heikki)

別の症状として、一部の他のプロセスが不要となったファイルの削除に干渉する場合にも発生します。

- PAMパスワード処理をより堅牢に修正しました。(Tom)

以前のコードでは、Linuxの`pam_krb5` PAMモジュールとドメインコントローラとしてのMicrosoft Active Directoryを組み合わせると失敗することが知られていました。PAMスタックに渡される引数に関する正当ではない仮定を作成することであるため、どこかで問題になるかもしれません。

- GSSAPIおよびSSPI認証方式における認証トークン(Kerberosチケット)の最大サイズを上げました。(Ian Turner)

以前の2000バイトという制限はUnix Kerberosの実装では十分でしたが、Windowsドメインコントローラで発行されるチケットではより大きくなることがあります。

- ドメインが配列型を管理する場合、ドメイン制約が`ARRAY[...]::domain`のような式として確実に強制されるようにしました。(Heikki)

- 複合型の列を外部キーとして含む一部の場合における外部キーのロジックを修正しました。(Tom)

- カーソルのスナップショットが生成後、確実に変更されないようにしました。(Alvaro)

これは、同じトランザクション内で後の操作が、そのカーソルが返すものと想定するデータを変更する場合、カーソルが間違った結果をもたらすようになります。

- `CREATE TABLE`が別の継承する親テーブル由来のデフォルト式を適切にマージするように修正しました。(Tom)

これは動作していましたが、8.4では壊れていました。

- シーケンスのアクセス統計情報の収集を再度有効にしました。(Akira Kurosawa)

これは動作していましたが、8.3では壊れていました。

- `CREATE OR REPLACE FUNCTION`における所有権限の依存関係の処理を修正しました。(Tom)

- `WHERE x=x`条件の不正な処理を修正しました。(Tom)

一部の場合、これらは冗長なものとして無視されることがありましたが、そうではありません。 — これらは`x IS NOT NULL`と等価です。

- テキスト等価性に関する揮発式においてハッシュ集約を使用して`DISTINCT`を実装する場合における不正な計画生成を修正しました。(Tom)

- 揮発`SELECT DISTINCT ON`式におけるアサート失敗を修正しました。(Tom)

- 空の`tsvector`値に対して`ts_stat()`が失敗ないように改修しました。(Tom)

- 全文検索パーサがXML属性内のアンダースコアを受け付けるようにしました。(Peter)

- `xml`バイナリ入力における符号化方式の扱いを修正しました。(Heikki)

XMLヘッダにて符号化方式の指定がない場合、デフォルトでUTF-8を仮定するようになります。以前の扱いには一貫性がありませんでした。

- `plperl`から`plperl1`の呼び出し、またはその逆における不具合を修正しました。(Tom)

内部の関数からのエラー終了が、外側の関数用のPerlインタプリタを正しく再選択できずにクラッシュをもたらす可能性がありました。

- PL/Perl関数が再定義された時のセッション有効期間のメモリリークを修正しました。(Tom)

- 集合を返すPL/Perl関数により返される場合Perl配列が適切にPostgreSQL配列に返還されることを確実にしました。(Andrew Dunstan, Abhijit Menon-Sen)

集合以外を返す関数ですでに正しく動作しました。

- PL/Pythonの例外処理におけるまれなクラッシュを修正しました。(Peter)

- `DECLARE CURSOR`文におけるコメントに関する`ecpg`問題を修正しました。(Michael)

- `ecpg`が最近追加されたキーワードを予約語と扱わない問題を修正しました。(Tom)

これは、`CALLED`、`CATALOG`、`DEFINER`、`ENUM`、`FOLLOWING`、`INVOKER`、`OPTIONS`、`PARTITION`、`PRECEDING`、`RANGE`、`SECURITY`、`SERVER`、`UNBOUNDED`、`WRAPPER`キーワードに影響します。

- `psql`の`\df`関数名パラメータにおいて正規表現特殊文字を再度許します。(Tom)

- `contrib/fuzzystrmatch`において、デフォルト以外のコストによる`levenshtein`距離の計算を正しくしました。(Marcin Mank)

- contrib/pg_standbyにて、Windowsのシグナルを使用したフェイルオーバーの通知を無効にしました。(Fujii Masao)

WindowsではUnixのようなシグナルを持ちませんので、これはまったく役に立っていませんでした。 さらに最近の変更により実際にクラッシュしてしまいます。

- FREEZEおよびVERBOSE オプションをcontrib/vacuumdbが生成するVACUUM コマンドの右側につけました。(Heikki)
- contrib/dblinkがエラーになった場合の接続が漏えいする可能性を修正しました。(Tatsuhito Kasahara)
- postgresqlのflexモジュールが正しいシステムヘッダファイルを使用してコンパイルされることを確実にしました。(Tom)

これは、--enable-largefileが生成されたコード内に非互換の変更を引き起こすプラットフォームにおける構築失敗を修正します。

- 今後のバージョンのlibpqとの互換性を高めるために、postmasterが接続要求パケットにおけるapplication_nameパラメータをすべて無視するようにしました。(Tom)
- 最新の現実合うように時間帯省略形ファイルを更新しました。(Joachim Wieland)

デフォルト時間帯群にIDTの追加が含まれます。

- 時間帯データファイルをtzdata release 2009sまで更新しました。 南極、アルゼンチン、バングラデシュ、フィジー、ノヴォクズネツク、パキスタン、パレスチナ、サモア、シリアにおける夏時間規則の変更、および香港に関する歴史的な修正が含まれます。

[前のページ](#)

リリース 8.4.3

[ホーム](#)

[上に戻る](#)

[次のページ](#)

リリース8.4.1

E.46. リリース8.4.1

リリース日: 2009-09-09

このリリースは8.4に対し、各種の不具合を修正したものです。 8.4メジャーリリースにおける新機能については[項E.47](#)を参照してください。

E.46.1. バージョン8.4.1への移行

8.4.Xからの移行ではダンプ/リストアは不要です。

E.46.2. 変更点

- アーカイブリカバリの最後におけるWALページヘッダの初期化を修正しました。(Heikki)
その後に行うアーカイブリカバリにおいてWAL処理が失敗する可能性があります。
- "cannot make new WAL entries during recovery (リカバリ中に新しいWAL項目を作成できません)"というエラーを修正しました。(Tom)
- クラッシュ後に無効になった行が参照可能になる問題を修正しました。(Tom)
この不具合には、サーバクラッシュ後にページステータスビットが正しく設定されない可能性が含まれていました。
- security-definer関数内部のRESET ROLEおよびRESET SESSION AUTHORIZATIONを不許可としました。(Tom, Heikki)
これは、security-definer関数内部におけるSET ROLEとSET SESSION AUTHORIZATIONを不許可にする過去のパッチ (CVE-2007-6600参照) で抜けていたものを補うものです。
- 既にロードされたロード可能モジュールに対するLOADを何も行わないようにしました。(Tom)
これまでは、LOADはそのモジュールをアンロードし、再ロードしようとしていました。 ですが、これは安全ではなく、かつ、まったく意味がないものでした。
- ウィンドウ関数PARTITION BYおよびORDER BYの項目を、常に単純な式として扱うようにしました。(Tom)
8.4.0では、これらのリストは最上位のGROUP BYとORDER BYリストで使用される規則にしたがって解析されました。 しかし標準SQLに対して正しくなく、また、巡回する可能性をもたらすものでした。
- セミ結合の計画作成における複数のエラーを修正しました。(Tom)
これはINまたはEXISTSが他の結合といっしょに使用される一部の状況において、間違った検索結果を導きました。
- 外部結合内の副問い合わせに対する行全体参照の扱いを修正しました。(Tom)
例えばSELECT COUNT(ss.*) FROM ... LEFT JOIN (SELECT ...) ss ON ...です。 ここでss.*は、NULL拡張された結合行ではROW(NULL, NULL, ...)として扱われました。 これは単純なNULLと同じではありません。 これを単純なNULLとして扱うようにしました。
- Windowsの共有メモリ割り当てコードを修正しました。(Tsutomu Yamada, Magnus)
この不具合がよく報告される"could not reattach to shared memory (共有メモリに再接続できませんでした)"というエラーメッセージを導いていました。
- plperlにおけるロケール処理を修正しました。(Heikki)
この不具合により、plperl関数が呼び出された時にサーバのロケール設定が変更され、データ破損が発生してしまうことがありえました。
- reloptions処理を修正し、確実に、あるオプション設定により他のオプションがデフォルト値に強制されないようにしました。(Itagaki Takahiro)
- たとえ"スマートシャットダウン"が進行中であったとしても、"高速シャットダウン"要求が実行中のセッションを強制的に終了させることを確実にしました。(Fujii Masao)

- `GROUP BY`問い合わせにおける`array_agg()`のメモリリークを防止します。(Tom)
- `to_char(..., 'TH')`を'`HH`'/'`HH12`'の大文字の序数表現と扱います。(Heikki)
以前は'`th`' (小文字) として扱われました。
- `time`および`time with time zone`入力に対して、`EXTRACT(second)` および`EXTRACT(milliseconds)`の結果に端数部分を含めました。(Tom)
これは常に浮動小数点型の日付時刻設定では行われていました。しかし整数型の日付時刻に関するコードは壊れていました。
- `INTERVAL 'x ms'`において`x`が200万を超え、かつ、整数型の日付時刻が使用された場合のオーバーフローを修正しました。(Alex Hunsaker)
- インデックススキャンにおけるトースト化された値の処理性能を改良しました。(Tom)
これは特に[PostGIS](#)で有用です。
- `commit_delay`を無効にする際の打ち間違いを修正しました。(Jeff Janes)
- サーバがサイレントモードで起動した場合、起動初期のメッセージを`postmaster.log`に出力します。(Tom)
これまではこうしたエラーメッセージは破棄され、デバッグが困難になりました。
- FAQの翻訳版を削除しました。(Peter)
現在これは[wiki](#)に存在します。英語版のFAQはすでにwikiに移動していました。
- `postgresql.conf`が空の場合に、`pg_ctl`が無限ループに陥らないように修正しました。(Jeff Davis)
- `pg_dump`の`--binary-upgrade`モードにおける複数のエラーを修正しました。(Bruce, Tom)
`pg_dump --binary-upgrade`は`pg_migrator`で使用されます。
- `contrib/xml2`の`xslt_process()`がパラメータの最大値(20)を正しく扱うように修正しました。(Tom)
- `COPY FROM STDIN`中のエラーからの復旧するための`libpq`のコードについて、堅牢性を高めました。(Tom)
- `readline`と`editline`ライブラリの両方がインストールされている場合、それらの競合するヘッダファイルが`include`されないようにしました。(Zdenek Kotala)
- 一部のプラットフォームにおける、"division by zero (0除算)"ではなく"floating-point exception (浮動小数点例外)"を引き起こすgccの不具合を回避しました。(Tom)
- バングラデシュ、エジプト、モーリシャスにおける夏時間規則の変更のため、時間帯データファイルを`tzdata`リリース2009Iに更新しました。

E.47. リリース8.4

リリース日: 2009-07-01

E.47.1. 概要

長年の開発の結果PostgreSQLは多くの領域においてその機能が完全なものになりました。このリリースでは新機能（認証、監視、領域再利用など）に関しては的を絞った手法を取り、また最近の標準SQLで定義された機能を追加しています。改良された主な点を以下に示します。

- ウィンドウ関数
- 共通テーブル式および再帰問い合わせ
- 関数のデフォルトパラメータおよび可変長パラメータ
- 並行リストア
- 列権限
- データベース単位のロケール設定
- ハッシュインデックスの改良
- `EXISTS`および`NOT EXISTS`問い合わせにおける、結合性能の改良
- ウォームスタンバイの使用の簡易化
- 空き領域マップの自動サイズ調整
- 可視性マップ（変更頻度が少ないテーブルにおけるバキュームオーバーヘッドを大きく低減します）
- バージョンを問わないpsql（古めのサーバに対してもバックスラッシュコマンドが動作します）
- ユーザ認証におけるSSL証明書のサポート
- 関数単位の実行時統計情報
- psqlにおける関数編集の簡易化
- 新しいcontribモジュール:pg_stat_statements、auto_explain、citext、btree_gin

上記項目は以下の節で詳細に説明します。

E.47.2. バージョン8.4への移行

すべての過去のリリースからデータを移行する場合、pg_dumpを使用したダンプ/リストアが必要です。

既知の非互換性を以下に示します。

E.47.2.1. 一般

- デフォルトでは64ビット整数型の日付時間型を使用します。(Neil Conway)

これまでは、configureの`--enable-integer-datetimes`オプションでこれを選択していました。過去の動作を維持するためには`--disable-integer-datetimes`付きで構築してください。

- ipccleanユーティリティコマンドを削除しました。(Bruce)

このユーティリティは一部のプラットフォームでしか動作しませんでした。ユーザは代わりに使用するオペレーティングシステムのツールを使用すべきです。

E.47.2.2. サーバの設定

- ログファイルのサイズを小さくするため`log_min_messages`のデフォルト設定を`warning`に変更しました（これまででは`notice`でした）。(Tom)
- `max_prepared_transactions`のデフォルト設定をゼロに変更しました（これまででは5でした）。(Tom)
- LOGメッセージレベルで`debug_print_parse`、`debug_print_rewritten`および`debug_print_plan`の出力が行われるようにしました。 これまでは`DEBUG1`レベルでした。(Tom)
- `debug_pretty_print`のデフォルトを`on`にしました。(Tom)
- `explain_pretty_print`パラメータを削除しました（必要がなくなりました）。(Tom)
- 他のログ処理用のオプションと同様に、スーパーユーザのみが`log_temp_files`を設定できるようにしました。(Simon Riggs)
- `log_filename`に`%`エスケープが存在しない場合、エポックタイムスタンプが自動的につかないようにしました。(Robert Haas)

外部のログローテーションツールで使えるようにログファイル名を固定化したいユーザが一部にいましたので、この変更がなされました。

- `recovery.conf`から`log_restartpoints`を削除しました。 代わりに`log_checkpoints`を使用してください。(Simon)
- `krb_realm`および`krb_server_hostname`を削除しました。 これらは`pg_hba.conf`内で設定されるようになりました。(Magnus)
- また、後述のように`pg_hba.conf`には大きな変更点があります。

E.47.2.3. 問い合わせ

- `TRUNCATE`および`LOCK`が指定したテーブルの子テーブルにも適用されるよう変更しました。(Peter)

これらのコマンドは子テーブルの処理を抑制できるように`ONLY`オプションを受け付けるようになりました。 これまでの動作が必要な場合このオプションを使用しなければなりません。

- `SELECT DISTINCT`および`UNION/INTERSECT/EXCEPT`はソートされた出力を常に生成しないようになりました。(Tom)

これまででは、こうした種類の問い合わせは常に、ソート・一意性処理（つまりソート後隣接する重複を除去）により重複行を取り除きました。 これがハッシュ処理により実装できるようになり、その結果ソートされた出力が生成されなくなりました。 出力順がソートされていることに依存するアプリケーションで推奨する修正方法は`ORDER BY`句を付与することです。 短期的な回避方法として、`enable_hashagg`を無効にすることで過去の動作に戻すことができます。 しかしこれは性能を大きく劣化させる修正方法です。 しかし`SELECT DISTINCT ON`ではハッシュ処理を使用しませんので、この動作には変更ありません。

- 子テーブルは親テーブルの`CHECK`制約を強制的に継承します。(Alex Hunsaker, Nikhil Sontakke, Tom)

これまでではこうした制約を子テーブルで削除することができ、その結果、親テーブルの走査時に制約に反する行が現れることがありました。 これは非一貫性、かつ標準SQLに反するものと考えられました。

- 負の`LIMIT`または負の`OFFSET`値を、ゼロとして扱うことなく、許さなくなりました。(Simon)
- トランザクションブロック外での`LOCK TABLE`を許さなくなりました。(Tom)

ロックは即座に解放されますので、こうした操作は無意味です。

- シーケンスが追加の`start_value`列を持つようになりました。(Zoltan Boszormenyi)

これが`ALTER SEQUENCE ... RESTART`をサポートします。

E.47.2.4. 関数および演算子

- `numeric`型の0の累乗がエラーとならずに、0を返すようになりました。 また`numeric`型の0の0乗もエラーとならずに、1を返すようになりました。(Bruce)

これは、従来の`float8`型における動作と一致します。

- 浮動小数点における単項のマイナスがマイナスゼロを生成できるようになりました。(Tom)

変更後の動作はIEEE標準との互換性がより高いものです。

- エスケープ文字がLIKEパターンの最後の文字であった場合（つまりエスケープすべきものが存在しない場合）、エラーになります。(Tom)

これまでは、こうしたエスケープ文字は警告なく無視されました。 このためアプリケーションのロジックエラーを隠してしまう可能性があります。

- これまでLIKEインデックス比較に使用されていた演算子`~=`および`<>`を削除しました。(Tom)

パターンインデックスは通常の等価演算子を使用するようになりました。

- `xpath()` がその引数を変更することなくlibxmlに渡すようになりました。(Andrew)

これは、XML引数が整形済みのXML文書でなければならないことを意味します。 これまでのコードではXML断片も許可するようにもくろんでいましたが、うまく動作しませんでした。

- `xmlelement()` が要素値と同様に属性値を整形するようになりました(Peter)

これまでは、属性値は通常のSQL出力の動作に従って成形されていました。 これはXMLの規約に時として合致しません。

- libxmlを使用する関数向けのメモリ管理を作成し直しました。(Tom)

この変更により、PL/Perlや他の追加コードにおけるlibxmlを使用した場合とのいくつかの互換性に関する問題が避けられるはずです。

- ハッシュ関数用により高速なアルゴリズムを採用しました。(Bob Jenkinsの作業を元としたKenneth Marshall)

組み込みのハッシュ関数の多くで、リトルエンディアンマシンとビッグエンディアンマシンとで異なる結果になるようになりました。

E.47.2.4.1. 時間に関する関数および演算子

- `DateStyle`がinterval型の出力書式を制御しないようになりました。 代替りとなる新しい`IntervalStyle`変数が存在します。(Ron Mayer)

- `timestamp`および`interval`の出力における秒の小数部の扱いに関する一貫性を向上しました。(Ron Mayer)

これにより、小数部の桁が以前と異なる表示になったり、切り捨てではなく四捨五入になったりする可能性があります。

- `to_char()` の翻訳された月名、日名が`LC_MESSAGES`ではなく`LC_TIME`に依存するようになりました。(Euler Taveira de Oliveira)

- `to_date()` および`to_timestamp()` が無効な入力に対してより一貫したエラー報告を行うようにしました。(Brendan Jurd)

これまでのバージョンでは、書式文字列に一致しない入力を無視したり、警告なく誤解釈したりすることがよくありました。 こうした状況でエラーとなるようになりました。

- 午前/午後 (`AM/PM`) および紀元 (`BC/AD`) の書式指定に関して大文字小文字の厳密な違いを必要としないように`to_timestamp()` を修正しました。(Brendan Jurd)

例えば`ad`という入力値は`AD`書式文字列に一致するようになりました。

E.47.3. 変更点

PostgreSQL 8.4とこれまでのメジャーリリースとの間の変更点についての詳細を以下に示します。

E.47.3.1. 性能

- オプティマイザの統計情報計算を改良しました。(Jan Urbanski, Tom)

特に全文検索演算子に関する推定が大きく改良されました。

- ハッシュ処理を用いた`SELECT DISTINCT`および`UNION/INTERSECT/EXCEPT`が可能になりました。(Tom)

これは、こうした種類の問い合わせが自動的にソートされた出力を生成しなくなったことを意味します。

- 半結合および反結合に関して明確な概念を作成しました。(Tom)

この作業により、`IN (SELECT ...)` 句に関するこれまでのとってつけたような扱いを形式化しました。さらにこれを `EXISTS` および `NOT EXISTS` 句にも拡張しました。これにより `EXISTS` および `NOT EXISTS` 問い合わせの計画作成が非常に改善されるはずです。一般的には、論理的には同一である `IN` と `EXISTS` 句が、同程度の性能を持つようになりました。これまではよく `IN` の方が勝っていました。

- 外部結語が背後にある副問い合わせの最適化を改良しました。(Tom)

これまでは、副問い合わせまたはビューが外部結合の `NULL` を持ち得る側にそれが現れる場合や厳密でない式（例えば定数）がその結果リストに含まれる場合、非常によく最適化することができませんでした。

- Boyer-Moore-Horspool 検索を使用することで、`text_position()` と関連する関数の性能を改良しました。(David Rowley)

これは特に長い検索パターンで有用です。

- 要求があった時にのみファイルに書き込むことにより、統計情報集計ファイルの書き込みによる I/O 負荷が低減しました。(Martin Pihlak)

- 一括挿入の性能を改良しました。(Robert Haas, Simon)

- `default_statistics_target` のデフォルト値を 10 から 100 に増やしました。(Greg Sabino Mullane, Tom)

また最大値も 1000 から 10000 に増やしました。

- 継承または `UNION ALL` を含む問い合わせにおいて、デフォルトで `constraint_exclusion` 検査を行います。(Tom)

`constraint_exclusion` の新しい設定 `partition` がこの動作を指定するために追加されました。

- ビットマップインデックス走査において I/O の先読みが可能になりました。(Greg Stark)

先読み量は `effective_io_concurrency` により制御されます。この機能はカーネルが `posix_fadvise()` サポートを有する場合のみ利用可能です。

- `FROM` 句内の単純な集合を返す SQL 関数をインライン化しました。(Richard Rowell)

- 外側のリレーション内で特に一般的な値が結合キー値となる特別な状況を考慮することにより、複数バッチのハッシュ結合の性能を向上しました。(Bryce Cutt, Ramon Lawrence)

- "physical tlist" を抑制することにより複数バッチのハッシュ結合における一時データ量を軽減しました。(Michael Henderson, Ramon Lawrence)

- `CREATE INDEX CONCURRENTLY` 中のトランザクション内待機状態セッションを待機しないようにしました。(Simon)

- 共有キャッシュの無効化の性能を向上しました。(Tom)

E.47.3.2. サーバ

E.47.3.2.1. 設定

- `pg_settings` が有効な値を表示できるようにするために、多くの `postgresql.conf` 設定が列挙型の値に変換されます。(Magnus)
- プランナが取り出すものと仮定するカーソル行数を制御するための `cursor_tuple_fraction` パラメータを追加しました。(Robert Hell)
- `postgresql.conf` におけるカスタム変数クラスの名前にアンダースコアが使用できるようになりました。(Tom)

E.47.3.2.2. 認証とセキュリティ

- (安全ではない) `crypt` 認証方式サポートを削除しました。(Magnus)

使用することができる平文ではないパスワード方式はまったくありませんでしたので、これは実質的に PostgreSQL 7.2 クライアントライブラリで廃止されていました。

- `pg_ident.conf` 内の正規表現をサポートします。(Magnus)
- Kerberos/GSSAPI パラメータが `postmaster` を再起動させることなく変更できるようになりました。(Magnus)
- サーバ証明書ファイル内の SSL 証明書チェーンをサポートします。(Andrew Gierth)

完全な証明書チェーンを含めることにより、クライアントは、商用CAを使用する場合によくある、ローカルに保持した中間CAの証明書をすべてに持たせることなく証明書を検証できるようになります。

- `db_user_namespace` を有効にし、MD5 認証を組み合わせで使用した場合に、適切なエラーメッセージを報告するようになります。(Bruce)

E.47.3.2.3. `pg_hba.conf`

- すべての認証オプションが `name=value` 構文を使用するように変更しました。(Magnus)

これにより `ldap`、`pam`、`ident` 認証方式において互換性がない変更がなされました。これらの方式を持つ `pg_hba.conf` はすべて新しい書式を使う形に書き換えなければなりません。

- `ident sameuser` オプションを削除しました。代わりにこれをユーザマップの指定がない場合のデフォルトの動作としました。(Magnus)

- すべての外部認証方式においてユーザマップパラメータを許可します。(Magnus)

これまではユーザマップは `ident` 認証でのみサポートされていました。

- クライアント証明要求を制御するための `clientcert` オプションを追加しました。(Magnus)

これまではサーバのデータディレクトリ内のルート証明書の有無により制御されていました。

- SSL 証明書経由のユーザ認証を可能にする `cert` 認証方式を追加しました。(Magnus)

これまでの SSL 証明書はクライアントが証明書へのアクセスを持つかどうか検証するのみで、ユーザの認証は行いませんでした。

- `krb5`、`gssapi`、`sspi` のレルムおよび `krb5` のホスト設定が `pg_hba.conf` 内で指定できるようになりました。(Magnus)

これらは `postgresql.conf` 内の設定を上書きします。

- `krb5`、`gssapi`、`sspi` 方式向けの `include_realm` パラメータを追加しました。(Magnus)

これにより異なるレルムからの同一ユーザがユーザマップを使用する別のデータベースユーザとして認証できるようになります。

- 即座にエラーを報告するために、ロード時に `pg_hba.conf` を完全に解析します。(Magnus)

これまでは、ファイル内のほとんどのエラーは、クライアントが接続しようとするまで検知されませんでした。このためエラーのあるファイルによりシステムを使用できないようにしてしまいます。新しい動作では、エラーが再読み込み時に検知されると、おかしいファイルは拒絶され、`postmaster` は過去の値を用いて継続します。

- `pg_hba.conf` 内の最初のエラーを検知した後中断するのではなく、すべての解析エラーを表示します。(Selena Deckelmann)

- Solaris において Unix ドメインソケット経由での `ident` 認証をサポートします。(Garick Hamlin)

E.47.3.2.4. 継続的なアーカイブ処理

- チェックポイントをできる限り早い終了を強制するためのオプションを `pg_start_backup()` に提供します。(Tom)

デフォルトの動作では過度な I/O 消費を抑えますが、同時実行の問い合わせ活動がなければ意味がありません。

- 変更された WAL ファイルがアーカイブされるまで `pg_stop_backup()` を待機させます。(Simon)

これは `pg_stop_backup()` 完了時にバックアップが有効になったことを保証します。

- アーカイブが有効な場合、すべてのトランザクションを即座にアーカイブすることができるよう、シャットダウン時に最終 WAL セグメントをローテートします。(Guillaume Smet, Heikki)

- 継続的なアーカイブ処理のベースバックアップが実行している間、"smart" シャットダウンを遅延します。(Laurenz Albe)

- "fast" シャットダウンが要求された場合は継続的なアーカイブ処理のベースバックアップをキャンセルします。(Laurenz Albe)

の論理型の変数が
た。(Bruce)

の論理型変数と同じ範囲の文字列を取ることができるようになりました

E.47.3.2.5. 監視

- PostgreSQL設定ファイルが最後に読み込まれた日時を報告する`pg_conf_load_time()`を追加しました。(George Gensure)
- 安全にバックエンドを停止させる`pg_terminate_backend()`を追加しました (`SIGTERM`シグナルも動作します)。(Tom, Bruce)

常に単一バックエンドに対して`SIGTERM`を行うことは可能でしたが、これまでは未サポートとされていました。この状況を試験したところいくつか不具合があり、それらを修正しました。

- ユーザ定義関数呼び出し数と実行時間を追跡する機能を追加しました。(Martin Pihlak)

関数統計情報は新しいシステムビュー`pg_stat_user_functions`に出力されます。追跡は新しいパラメータ`track_functions`により制御されます。

- `pg_stat_activity`内の最大問い合わせ文字列サイズの指定が`track_activity_query_size`パラメータでできるようになりました。(Thomas Lee)
- 性能改善を目的として、`syslog`に送信する最大行数を増やしました。(Tom)
- 読み取りのみの設定変数`segment_size`、`wal_block_size`、`wal_segment_size`を追加しました。(Bernd Helmle)
- デッドロックの報告時に、サーバログにデッドロックに関わるすべての問い合わせテキストを記録します。(Itagaki Takahiro)
- 特定のプロセスIDに関する情報を返す`pg_stat_get_activity(pid)`関数を追加しました。(Magnus)
- サーバの統計情報ファイルの場所を`stats_temp_directory`経由で指定できるようにしました。(Magnus)

これにより統計情報ファイルをRAM上のディレクトリに格納させて、I/O要求を低減させることができます。再起動に渡って保持できるように、起動時または終了時にファイルは通常場所 (`$PGDATA/global/`) にコピーされます。

E.47.3.3. 問い合わせ

- `WINDOW`関数のサポートを追加しました。(Hitoshi Harada)
- `WITH`句 (CTE) のサポートを追加しました。これには`WITH RECURSIVE`が含まれます。(Yoshiyuki Asaba, Tatsuo Ishii, Tom)
- `TABLE`コマンドを追加しました。(Peter)

`TABLE tablename`は、`SELECT * FROM tablename`の標準SQLにおける省略形です。

- `SELECT` (または`RETURNING`) の列出力ラベルを指定する時`AS`を省略可能にしました。(Hiroshi Saito)

これは列ラベルがPostgreSQLキーワードではない場合のみ動作します。その他の場合は`AS`はまだ必要です。

- `SELECT`結果リスト内において集合を返す関数を、タブルストアを介して結果を返す関数であったとしてもサポートします。(Tom)

具体的には、これはPL/pgSQLや他のPL言語で作成された関数をこの方法で呼び出すことができるようになったことを意味します。

- 集約問い合わせやグループ化問い合わせの出力において集合を返す関数をサポートします。(Tom)

- `SELECT FOR UPDATE/SHARE`が継承ツリー上に動作できるようになりました。(Tom)

- SQL/MED用の基盤を追加しました。(Martin Pihlak, Peter)

リモートまたは外部SQL/MED機能はまだありません。しかしこの変更は、`dblink`や`plproxy`のようなモジュール向けの接続情報の管理のための標準化と陳腐化されないシステムを提供します。

- 参照されるスキーマ、関数、演算子、演算子クラスが変更された時にキャッシュされた計画を無効化します。(Martin Pihlak, Tom)

これは、DDLの変更にその場で応答するシステム機能を改良します。

- 複合型の比較を可能とし、かつ、匿名複合型の配列を可能としました。(Tom)

これにより、`row(1, 1.1) = any (array[row(7, 7.7), row(1, 1.0)])`のような式が可能になります。これは特に再帰問い合わせで有用です。

- Uniode文字列リテラルと、たとえば`U&'d¥0061t¥+000061'`といったコードポイントを使用した識別子指定のサポートを追加しました。(Peter)
- 文字列リテラルおよびCOPYデータ内の`¥000`を拒絶します。(Tom)

これまでは、これは受け付けられましたが、文字列定数の終端という作用がありました。

- エラー位置を報告するパーサ機能を改良しました。(Tom)

エラー位置がデータ型の不一致などの多くの意味的なエラーに対して報告されるようになりました。これまでは位置づけることができませんでした。

E.47.3.3.1. TRUNCATE

- 文レベルのON TRUNCATEトリガをサポートします。(Simon)
 - TRUNCATE TABLEにRESTART/CONTINUE IDENTITYオプションを追加しました。(Zoltan Boszormenyi)
- シーケンスの開始値をALTER SEQUENCE START WITHで変更できます。
- TRUNCATE tab1, tab1が成功するようになりました。(Bruce)
 - 独立したTRUNCATE権限を追加しました。(Robert Haas)

E.47.3.3.2. EXPLAIN

- EXPLAIN VERBOSEが計画ノード毎に出力列を表示するようにしました。(Tom)

これまでのEXPLAIN VERBOSEでは問い合わせ計画の内部表現を出力しました。(この動作はdebug_print_planを介して利用できるようになりました。)

- EXPLAINが個々のテーブルの副計画と初期計画とを識別するようにしました。(Tom)
- EXPLAINがdebug_print_planを順守するようにしました。(Tom)
- CREATE TABLE ASに対するEXPLAINが可能です。(Peter)

E.47.3.3.3. LIMIT/OFFSET

- LIMITおよびOFFSET内の副問い合わせが可能です。(Tom)
- LIMIT/OFFSET機能に標準SQL構文を追加しました。(Peter)

すなわち`OFFSET num {ROW|ROWS} FETCH {FIRST|NEXT} [num] {ROW|ROWS} ONLY`です。

E.47.3.4. オブジェクト操作

- 列レベル権限のサポートを追加しました。(Stephen Frost, KaiGai Kohei)
- CASCADEの必要性を軽減させるために複数オブジェクトのDROP操作を再作成しました。(Alex Hunsaker)

例えば、テーブルBがテーブルAに依存している場合、コマンド`DROP TABLE A, B`がCASCADEオプションを必要としないようになりました。

- オブジェクトの依存関係を削除を開始する前にロックを確実に取得することにより、同時実行のDROPコマンドに関する各種問題を修正しました。(Tom)
- DROPコマンド中の依存関係の報告を改良しました。(Tom)
- 標準SQLに従い、CREATE TABLE ASにWITH [NO] DATA句を追加しました。(Peter, Tom)
- ユーザ定義のI/O変換キャストのサポートを追加しました。(Heikki)

- `CREATE AGGREGATE`が`internal`遷移データ型を使用できるようにしました。(Tom)
- `CREATE TYPE`に`LIKE`句を追加しました。(Tom)
これにより既存の型と同じ内部表現を使用するデータ型の作成が簡易になりました。
- ユーザ定義の基本型に対する型カテゴリと"好ましい"状態指定を可能としました。(Tom)
これにより、ユーザ定義型の強制動作についてより多くの制御が可能になります。
- `CREATE OR REPLACE VIEW`がビューの最後に列を追加できるようにしました。(Robert Haas)

E.47.3.4.1. `ALTER`

- `ALTER TYPE RENAME`を追加しました。(Petr Jelinek)
- シーケンスが初期値をリセットできるように`ALTER SEQUENCE ... RESTART` (パラメータなし) を追加しました。(Zoltan Boszormenyi)
- テーブル、インデックス、シーケンス、ビューを合理的に組み合わせられるように`ALTER TABLE`の構文を変更しました。(Tom)

この変更により以下の新しい構文を使用することができます。

- `ALTER SEQUENCE OWNER TO`
- `ALTER VIEW ALTER COLUMN SET/DROP DEFAULT`
- `ALTER VIEW OWNER TO`
- `ALTER VIEW SET SCHEMA`

実際ここでは新しい機能はありませんが、これまではこれらを行うためには`ALTER TABLE`と記述する必要があり、混乱を招きました。

- `ALTER TABLE ... ALTER COLUMN ... SET DATA TYPE`構文サポートを追加しました。(Peter)
これはすでにサポートされていた機能に関する標準SQLの構文です。
- 物理的にOID値を削除するために`ALTER TABLE SET WITHOUT OIDS`がテーブルを書き換えるようにしました。(Tom)
また、OIDを追加できるようにテーブルを書き換えるための`ALTER TABLE SET WITH OIDS`を追加しました。

E.47.3.4.2. データベース操作

- 未コミットな準備済みトランザクションが原因で`CREATE/DROP/RENAME DATABASE`が失敗した場合の報告を改良しました。(Tom)
- `LC_COLLATE`および`LC_CTYPE`をデータベース単位の設定にしました。(Radek Strnad, Heikki)
これはこれまでデータベース単位の設定であった照合順を符号化方式に似せました。
- データベース符号化方式、照合順 (`LC_COLLATE`)、文字クラス (`LC_CTYPE`) が一致するかどうかの検査を改良しました。(Heikki, Tom)

新しいデータベースの符号化方式とロケール設定が`template0`からコピーした場合のみ変更できることに特に注意してください。これは設定が一致しないデータを複製する可能性を防止します。

- データベースを新しいテーブル空間に移動させる`ALTER DATABASE SET TABLESPACE`を追加しました。(Guillaume Lelarge, Bernd Helmle)

E.47.3.5. ユーティリティ操作

- `CLUSTER`コマンドと`clusterdb`に`VERBOSE`オプションを追加しました。(Jim Cox)
- 待機中のトリガイベントの記録に必要なメモリを減らしました。(Tom)

E.47.3.5.1. インデックス

- ハッシュインデックスの構築速度とアクセス速度を大幅に改良しました。(Tom Raney, Shreya Bhargava)
これによりハッシュインデックスがB-Treeインデックスより高速になる場合もあり得ます。しかしハッシュインデックスはまだクラッシュに対して安全ではありません。
- ハッシュインデックスがインデックス対象の列の完全な値ではなく、ハッシュコードのみを保管するようにしました。(Xiao Meng)
これによりインデックス対象の列が長い場合ハッシュインデックスのサイズが大きく減少し、性能が向上しました。
- GINインデックスにおける高速更新オプションを実装しました。(Teodor, Oleg)
このオプションは大きく更新速度を向上しますが、検索速度に小さな影響が現れます。
- `xxx_pattern_ops` インデックスがLIKEだけではなく、単純な等価演算子でも使用できるようになりました。(Tom)

E.47.3.5.2. 全文検索

- GIN重みづけ検索を全文検索インデックスに対して行う際、`@@`を使用する必要性をなくしました。(Tom, Teodor)
通常の`@@`テキスト検索演算子を代わりに使用することができます。
- `@@`テキスト検索操作のオプティマイザの選択性関数を追加しました。(Jan Urbanski)
- 全文検索において前方一致が可能になりました。(Teodor Sigaev, Oleg Bartunov)
- 複数列に対するGINインデックスをサポートします。(Teodor Sigaev)
- ネパール語とヒンディーアルファベットのサポートを改良しました。(Teodor)

E.47.3.5.3. VACUUM

- 独立したリレーション単位の"フォーク"ファイル内で空き領域を追跡します。(Heikki)
`VACUUM`で見つかった空き領域は、固定長の共有メモリ領域ではなく`*_fsm`ファイルに記録されるようになりました。
`max_fsm_pages`と`max_fsm_relations`設定は削除され、空き領域管理の管理作業を大きく簡易化しました。
- バキューム処理を必要としないページを追跡する可視性マップを追加しました。(Heikki)
これにより、テーブルのごく一部のみがバキューム処理を必要とする場合に、`VACUUM`がテーブルすべてを走査することを防ぐことができます。可視性マップはリレーション単位の"フォーク"ファイル内に保管されます。
- `VACUUM`が可視性マップを無視し、そしてタプルを凍結するためにテーブル全体に対して行われなければならないタイミングを制御する`vacuum_freeze_table_age`パラメータを追加しました。(Heikki)
- トランザクションスナップショットをより入念に追跡します。(Alvaro)
これは、長期のトランザクションが存在する領域を回収する、`VACUUM`能力を向上します。
- リレーション単位の自動バキュームの指定する機能と`CREATE TABLE`内で`TOAST`パラメータを指定する機能を追加しました。(Alvaro, Euler Taveira de Oliveira)
自動バキュームオプションはシステムテーブル内に保管するために使用されます。
- `vacuumdb`に`--freeze`オプションを追加しました。(Bruce)

E.47.3.6. データ型

- 全文検索類義語辞書用に`CaseSensitive`オプションを追加しました。(Simon)
- `NUMERIC`除算の精度を改良しました。(Tom)
- `int2`に`int8`を使用した基本算術演算子を追加しました。(Tom)
これは、一部の状況における明示的なキャストの必要性を除去します。
- `UUID`の入力にて4桁ごとに省略可能なハイフンが受け付けられるようになりました。(Robert Haas)
- 論理データ型の入力として`on/off`を可能としました。(Itagaki Takahiro)

- `numeric` 型の入力文字列において `NaN` の前後の空白を許可します。(Sam Mason)

E.47.3.6.1. 時間に関するデータ型

- `0 BC` や `000`、`0000` という年を拒絶します。(Tom)

これまでは、これらは `1 BC` として解釈されました。(注意: `0` および `00` という年はまだ2000年であると仮定されます。)

- デフォルトの既知の時間帯省略形リストに `SGT` (シンガポール時間) を含めました。(Tom)
- `date` 型の値として `infinity` および `-infinity` をサポートします。(Tom)
- `interval` リテラルの解析をより標準に近づけました。(Tom, Ron Mayer)

例えば `INTERVAL '1' YEAR` は想定通りになりました。

- 標準SQLに従うために、`interval` の秒の端数に関する精度を `second` キーワードの後に指定できるようにしました。(Tom)

これまでは、この精度は `interval` キーワードの後に指定する必要がありました。(後方互換性のためこの構文はまだサポートされますが、廃止予定です。) データ型の定義は標準の書式を使用して出力されるようになりました。

- ISO 8601 `interval` 構文をサポートします。(Ron Mayer, Kevin Grittner)

例えば、`INTERVAL '1Y2M3DT4H5M6.7S'` がサポートされるようになりました。

- `interval` 値の出力方法を制御する `IntervalStyle` パラメータを追加しました。(Ron Mayer)

有効な値は、`postgres`、`postgres_verbose`、`sql_standard`、`iso_8601` です。この設定はまた、一部のフィールドのみが正または負の指定を持つ場合の、負の `interval` の取り扱い方法を制御します。

- `timestamp` および `interval` 出力における秒の端数の取扱いの一貫性を改良しました。(Ron Mayer)

E.47.3.6.2. 配列

- `ARRAY[...]::integer[]` など `ARRAY[]` 式に適用されるキャストの扱いを改良しました。(Brendan Jurd)

これまでのPostgreSQLは、その後に続くキャストを参照することなく、`ARRAY[]` 式のデータ型を決定しようと試みしました。これは多くの場合、特に `ARRAY[]` 式が空または `NULL` などの曖昧な項目しか含まないような場合で不要に失敗することがあり得ました。配列要素が必要とする型を決定する時にキャストが考慮されるようになりました。

- 標準SQLに一致させるため、`ARRAY` 次元のSQL構文を省略可能にしました。(Peter)
- 配列次元数を返す `array_ndims()` を追加しました。(Robert Haas)
- 配列の指定次元の長さを返す `array_length()` を追加しました。(Jim Nasby, Robert Haas, Peter Eisentraut)
- すべての集約した値を単一の配列として返す `array_agg()` 集約関数を追加しました。(Robert Haas, Jeff Davis, Peter)
- 配列を個々の行値に変換する `unnest()` を追加しました。(Tom)

これは `array_agg()` の反対です。

- 初期値を持つ配列を作成する `array_fill()` を追加しました。(Pavel Stehule)
- 配列の添え字範囲を簡単に生成する `generate_subscripts()` を追加しました。(Pavel Stehule)

E.47.3.6.3. 大規模値の格納 (TOAST)

- 値に対するTOAST圧縮は32バイト程度とみなします(これまでは256バイト)。(Greg Stark)
- TOAST圧縮を使用する前に空き領域が最低25%必要です。(これまでは小規模値では20%、大規模値ではその規模程度が必要でした。)(Greg)
- TOAST可能なフィールドが大規模フィールドと小規模フィールドとが混在する行に対して、大規模値を優先して行外に押し出し、小規模な値を不要に圧縮することを防ぐことができるように、TOASTの発見的処理過程を改良しました。

E.47.3.7. 関数

- `setseed()` が-1から1までの値 (0から1ではありません) を受け付けられることを文書化し、そしてその有効範囲を強制するようにしました。(Kris Jurka)
- `lo_import(filename, oid)` サーバサイド関数を追加しました。(Tatsuo)
- `quote_nullable()` を追加しました。 この動作は`quote_literal()` と似ていますが、引数NULLに対して文字列NULLを返します。(Brendan Jurd)
- 複数のテキスト断片を抽出できるように全文検索`headline()` 関数を改良しました。(Sushant Sinha)
- データの変更がない更新に対するオーバーヘッドを防止するための`suppress_redundant_updates_trigger()` トリガ関数を追加しました。(Andrew)
- `numeric` 除算を丸めなく実行する`div(numeric, numeric)` を追加しました。(Tom)
- `generate_series()` の`timestamp` 版と`timestampz` 版を追加しました。(Hitoshi Harada)

E.47.3.7.1. オブジェクト情報関数

- 現在実行中の問い合わせを知る必要がある関数による使用を目的とした`current_query()` を実装しました。(Tomas Doran)
- パーサのキーワードのリストを返す`pg_get_keywords()` を追加しました。(Dave Page)
- 関数定義を見るための`pg_get_functiondef()` を追加しました。(Abhijit Menon-Sen)
- 変数を含まない式のデパース時に`pg_get_expr()` 2番目の引数をゼロにすることができます。
- `pg_relation_size()` が`regclass` を使用するように変更しました。(Heikki)
- `pg_relation_size(data_type_name)` は動作しなくなりました。
- `pg_settings` の出力に`boot_val` 列と`reset_val` 列を追加しました。(Greg Smith)
- 設定ファイル内の変数集合のために、`pg_settings` の出力にソースファイル名と行番号列を追加しました。(Magnus, Alvaro)

セキュリティ上の理由により、これらの列はスーパーユーザのみが参照することができます。

- `CURRENT_CATALOG`、`CURRENT_SCHEMA`、`SET CATALOG`、`SET SCHEMA` のサポートを追加しました。(Peter)
- これらは既存の機能に対して標準SQL構文を提供します。
- 任意の値のデータ型を返す`pg_typeof()` を追加しました。(Brendan Jurd)
- `version()` は、サーバが32ビットバイナリか64ビットバイナリかについての情報を返すようにしました。(Bruce)
- 情報スキーマの列`is_insertable_into` と`is_updatable` が一貫性を持つ動作をするよう修正しました。(Peter)
- 情報スキーマの`datetime_precision` 列の動作を改良しました。(Peter)
- これらの列は`date` 列ではゼロを、`time`、`timestamp`、精度宣言がない`interval` では6 (デフォルト精度) を表示するようになりました。 これまではNULLを表示していました。
- 残りの集合を返す組み込み関数がOUTパラメータを使用するように変換しました。(Jaime Casanova)
- これにより次の関数を列リストの指定を行わずに呼び出すことができるようになります。
`pg_show_all_settings()`、`pg_lock_status()`、`pg_prepared_xact()`、`pg_prepared_statement()`、`pg_cursor()`
- `pg_*_is_visible()` および`has_*_privilege()` 関数が無効なOIDに対して、エラーとならずにNULLを返すようにしました。(Tom)
- 1度の呼び出しで複数の権限の論理和を調査できるように`has_*_privilege()` 関数を拡張しました。(Stephen Frost, Tom)
- `has_column_privilege()` および`has_any_column_privilege()` 関数を追加しました。(Stephen Frost, Tom)

E.47.3.7.2. 関数作成

- 可変長引数の関数 (引数の数が変動する関数) をサポートします。(Pavel Stehule)

終端までの引数のみを省略可能とすることができます。 またこれらはすべて同じデータ型でなければなりません。

- 関数引数のデフォルト値をサポートします。(Pavel Stehule)
- `CREATE FUNCTION ... RETURNS TABLE`句を追加しました。(Pavel Stehule)
- SQL言語関数が`INSERT/UPDATE/DELETE RETURNING`句の出力を返すことができます。(Tom)

E.47.3.7.3. PL/pgSQLサーバサイド言語

- より簡単に動的問い合わせ文字列にデータを挿入するための`EXECUTE USING`をサポートします。(Pavel Stehule)
- `FOR`ループを使用するカーソルの結果全体を繰り返すことが可能になりました。(Pavel Stehule)
- `RETURN QUERY EXECUTE`をサポートします。(Pavel Stehule)
- `RAISE`コマンドを改良しました。(Pavel Stehule)
 - `DETAIL`および`HINT`フィールドをサポートします。
 - `SQLSTATE`エラーコードの指定をサポートします。
 - 例外名パラメータをサポートします。
 - そこで起こったエラーを再発行できるように例外ブロックにおいてパラメータ無しの`RAISE`を可能としました。
- `EXCEPTION`リスト内で`SQLSTATE`コードの指定を可能としました。(Pavel Stehule)

これは独自の`SQLSTATE`コードを扱う際に有用です。

- `CASE`文をサポートします。(Pavel Stehule)
 - `RETURN QUERY`が特殊な`FOUND`と`GET DIAGNOSTICS ROW_COUNT`変数を設定するようにしました。(Pavel Stehule)
 - `FETCH`および`MOVE`が`GET DIAGNOSTICS ROW_COUNT`変数を設定するようにしました。(Andrew Gierth)
 - ラベル無しの`EXIT`が常にもっとも内側のループを抜けるようにしました。(Tom)
- これまでは、他のループよりも入れ子に最も近い`BEGIN`ブロックが存在すると、そのブロックが代わりに終了します。 新しい動作はOracle(TM)と一致するもので、かつ、これまで文書で説明していたものです。
- 文字列リテラルと入れ子状ブロックのコメントの処理が主SQLパーサの処理と一致するようにしました。(Tom)

具体的には、`RAISE`内の書式文字列が、`standard_conforming_strings`に従うようになることを含め、他の文字列リテラルと同じように動作するようになりました。 またこの変更は、`standard_conforming_strings`が有効な場合に有効なコマンドが失敗するといった他の問題も修正します。

- 入れ子状の例外ブロックの深さが異なる場所で、同一関数が呼び出された時のメモリリークを防止します。(Tom)

E.47.3.8. クライアントアプリケーション

- `pg_ctl restart`がコマンドライン引数を保持するよう修正しました。(Bruce)
 - `-W/--password`オプションを持つすべてのユーティリティにおいて、パスワードプロンプト表示を防止する`-w/--no-password`オプションを追加しました。(Peter)
 - `createdb`、`createuser`、`dropdb`、`dropuser`の`-q`（メッセージ無し）オプションを削除しました。(Peter)
- これらのオプションはPostgreSQL 8.3から効果がないものでした。

E.47.3.8.1. psql

- 起動時の冗長なバナーを除去しました。単に`help`を勧めるだけのものになりました。(Joshua Drake)
- `help`が一般的なバックスラッシュコマンドを表示するようにしました。(Greg Sabino Mullane)
- 画面幅に合わせて出力を、または、`¥pset columns`が設定された場合はファイルもしくはパイプへの出力も改行する、`¥pset format wrapped`モードを追加しました。(Bryce Nesbitt)

- `¥pset`内で`on`および`off`だけでなく、サポートされるすべての論理型の値の綴りが利用できるようにしました。(Bruce)
 これまでは、"off"以外の文字列は警告なく`true`を意味するものと解釈されていました。 `psql`は認識できない綴りについて警告を出すようになりました（ただしまだ`true`と解釈されます）。
- 幅が広い出力の際にページャを使用します。(Bruce)
- 一文字のバックスラッシュコマンドと最初の引数との間に空白を要求します。(Bernd Helmle)
 これにより歴史的なあいまいさの根源がなくなりました。
- スキーマ修飾および引用符付きの識別子に対するタブ補間サポートを改良しました。(Greg Sabino Mullane)
- `¥timing`に省略可能な`on/off`引数を追加しました。(David Fetter)
- 複数行に渡ってアクセス制御権限を表示します。(Brendan Jurd, Andreas Scherbaum)
- `¥l`がデータベースアクセス権限を表示するようにしました。(Andrew Gilligan)
- 権限がある場合`¥l+`がデータベース容量を表示するようにしました。(Andrew Gilligan)
- 関数定義を編集するための`¥ef`コマンドを追加しました。(Abhijit Menon-Sen)

E.47.3.8.2. `psql ¥d*`コマンド

- `s`修飾子が指定された場合のみ、パターン引数を持たない`¥d*`コマンドがシステムオブジェクトを表示するようにしました。(Greg Sabino Mullane, Bruce)
 これまでの動作では`¥d`の亜種の間での一貫性がありませんでした。 また、ほとんどの場合ユーザオブジェクトのみを参照する簡単な方法がありませんでした。
- 現行バージョンのサーバだけではなく、古めの（7.4までの）バージョンのPostgreSQLサーバで`¥d*`コマンドが動作するように改良しました。(Guillaume Lelarge)
- `¥d`が対象テーブルを参照する外部キー制約を表示するようにしました。(Kenneth D'Souza)
- シーケンスに対する`¥d`がその列値を表示するようにしました。(Euler Taveira de Oliveira)
- `¥d+`の表示に列の格納種類や関連するその他のオプションを追加しました。(Gregory Stark, Euler Taveira de Oliveira)
- `¥dt+`出力にリレーション容量を表示します。(Dickson S. Guedes)
- `¥dT+`において`enum`型で取り得る値を表示します。(David Fetter)
- キャスト内に存在するデータ型に一致するワイルドカードパターンを、`¥dc`が受け付けることを可能としました。(Tom)
- `¥df`の出力に関数種類列を追加しました。 また、選択した種類の関数のみを列挙するオプションを追加しました。(David Fetter)
- `¥df`が`cstring`型を取る、または、返す関数を隠さないようにしました。(Tom)
 これまでは、そのほとんどが興味を引くとは思われないデータ型のI/O関数であったため、こうした関数は隠れていました。 デフォルトのシステム関数の隠ぺいに関する新しいポリシーではこの欠点を不要にします。

E.47.3.8.3. `pg_dump`

- ダンプをテーブル空間の構造が異なるクラスタにリストアできるように、`pg_dump/pg_dumpall/pg_restore`に`--no-tablespaces`オプションを追加しました。(Gavin Roy)
- `pg_dump`と`pg_dumpall`から`-d`および`-D`オプションを削除しました。(Tom)
 これらのオプションは、他のPostgreSQLクライアントアプリケーションにおけるデータベース名を選択するためのオプションと、あまりに頻繁に混乱されていました。 この機能はまだ利用可能ですが、長めのオプション名`--inserts`または`-column-inserts`と記述する必要があります。
- `pg_dump`と`pg_dumpall`から`-i`および`--ignore-version`オプションを削除しました。(Tom)
 このオプションを使用してもエラーにはなりませんが、何も効果はありません。 安全のためバージョンチェックが必須となり

ましたので、このオプションは削除されました。

- ダンプまたはリストア中の `statement_timeout` を無効にしました。(Joshua Drake)
- `pg_dump/pg_dumpall` に `--lock-wait-timeout` オプションを追加しました。(David Gould)

これにより、指定期間中に共有ロックを獲得できなかった場合に、ダンプが失敗できるようになりました。

- 参照テーブルより前に外部キーにより参照されるテーブルをダンプするために `pg_dump --data-only` の出力順序を変更します。(Tom)
 - これにより、外部キーがすでに存在する時にデータロードができるようになりました。巡回参照により安全な順序付けができない場合、`NOTICE` が発行されます。
 - `pg_dump`、`pg_dumpall`、`pg_restore` が指定したロールを使用することができるようになりました。(Benedek László)
 - `pg_restore` がリストアするために複数の同時接続を使用できるようになりました。(Andrew)
- 同時接続数は `--jobs` オプションにより制御されます。これは独自書式のアーカイブでのみサポートされます。

E.47.3.9. プログラミングツール

E.47.3.9.1. libpq

- 新しい `lo_import_with_oid()` 関数を使用して、ラージオブジェクトのインポートの際に `OID` を指定できるようになりました。(Tatsuo)
- "イベント"サポートを追加しました。(Andrew Chernow, Merlin Moncure)
- これは、`PGconn` および `PGresult` オブジェクトに関連付けされた非公開データを管理するためのコールバックを登録する機能を追加したものです。
- 複数のエラーメッセージを複数行のエラー報告として返すことができるようにエラー処理を改良しました。(Magnus)
- `PQexecParams()` と関連する関数が、空の問い合わせに対して `PGRES_EMPTY_QUERY` を返すようにしました。(Tom)
- これまでは `PGRES_COMMAND_OK` を返していました。
- Windows における `WSACleanup()` のオーバーヘッドを防止する方法を文書化しました。(Andrew Chernow)
- デフォルトのデータベースユーザ名を決定する際に Kerberos チケットに依存しません。(Magnus)

これまでは、Kerberos 機能を有効に構築した libpq では、Kerberos 認証を使用しない接続であったとしても、利用可能な Kerberos チケットのプリンシパル名をデフォルトのデータベースユーザ名として使用しました。これは一貫性がなく混乱を招くものと思われます。デフォルトのユーザ名は Kerberos の有無に関係なく同じ方法で決定されるようになりました。しかし、Kerberos 認証を使用する場合、データベースユーザ名はチケットと一致しなければならないことに注意してください。

E.47.3.9.2. libpq SSL (セキュアソケットレイア) サポート

- SSL 接続の証明書検証を修正しました。(Magnus)
- libpq は、SSL 接続を確立する際に、証明書とサーバ名の検証をサポートするようになりました。検証においてルート証明書が利用できない場合、SSL 接続は失敗します。`sslmode` パラメータは証明書検証を有効にするため、および、その検査レベルを設定するために使用されます。デフォルトではまだまったく検証を行わず、クライアント上でルート証明書を必要とすることなく SSL を有効にしたサーバと接続することができます。
- ワイルドカードサーバ証明書をサポートします。(Magnus)
- 証明書の CN が * で始まる場合、ホスト名との比較を行う際にワイルドカードとして扱われ、複数のサーバで同じ証明書を使用できるようになります。
- クライアント証明書ファイルの位置を指定できるようになりました。(Mark Woodward, Alvaro, Magnus)
- OpenSSL/libcrypto の初期化について多くを制御できるように `PQinitOpenSSL` 関数を追加しました。(Andrew Chernow)
- データベース接続が開いたままの場合、libpq が OpenSSL コールバックの登録を取り消すようにしました。(Bruce, Magnus, Russell Smith)

これはlibpqライブラリをアンロードするアプリケーションで必要とされます。 こうしないと無効なOpenSSLコールバックが残ってしまいます。

E.47.3.9.3. ecpg

- 翻訳メッセージのサポートを追加しました。(Euler Taveira de Oliveira)
- ecpgパーサが自動的にサーバパーサから生成されるようになりました。(Michael)

これまではecpgパーサは手作業で保守されていました。

E.47.3.9.4. サーバプログラミングインタフェース (SPI)

- 行外のパラメータを持つ、一回しか使用されない計画をサポートするようになりました。(Tom)
- `SPI_execute()` に新しい戻り値コード `SPI_OK_REWRITTEN` を追加しました。(Heikki)

これは、コマンドが別の種類のコマンドに書き換えられた時に使用されます。

- `executor/spi.h` から不要なインクルードを削除しました。(Tom)

これらをインクルードするために `spi.h` に依存している場合、SPIを使用するモジュールにいくつか `#include` 行の追加が必要となる場合があります。

E.47.3.10. 構築オプション

- 構築システムをAutoconf 2.61を使用するように更新しました。(Peter)
- ソースコードから構築する場合GNU bisonが必要です。(Peter)

実質ここ数年で、これは必要とされていましたが、他のパーサツールをサポートしようという要望がなくなりました。

- `pg_config --htmldir` オプションを追加しました。(Peter)
- サーバ内部で `float4` を値渡しします。(Zoltan Boszormenyi)

過去の動作を使用するために `--disable-float4-byval` オプションを `configure` に追加しました。 この変更により、旧スタイル (バージョン0) の呼び出し規約を使用し、 `float4` 値を渡したり返したりする外部C関数は動作しなくなります。 このため、こうした関数があり、それを変更したくない場合は、この `configure` オプションが必要になるでしょう。

- 64ビットプラットフォーム上では、サーバ内部で `float8`、`int8`、これらに関連するデータ型を値渡しします。(Zoltan Boszormenyi)

過去の動作を使用するために `--disable-float8-byval` オプションを `configure` に追加しました。 上記のとおり、この変更は旧スタイルの外部C関数の動作に影響を与えます。

- `--with-segsize`、`--with-blocksize`、`--with-wal-blocksize`、`--with-wal-segsize` オプションを `configure` に追加しました。(Zdenek Kotala, Tom)

これまでは `pg_config_manual.h` を編集して変更するしかなかった、複数の定数を簡単に構築時に制御することができるようになります。

- Solaris 2.5でスレッド版の構築ができるようになりました。(Bruce)
- Solaris上でシステムの `getopt_long()` を使用します。(Zdenek Kotala, Tom)

これにより、Solarisユーザが期待するものに、より矛盾がないオプション処理ができるようになりました。

- LinuxにおけるSun Studioコンパイラのサポートを追加しました。(Julius Stroffek)
- バックエンドのgettextドメインにメジャーバージョン番号を追加しました。 また、ライブラリのgettextドメインに `soname` メジャーバージョン番号を追加しました。(Peter)

これは複数のバージョンの並行インストールを簡易化します。

- gcovを使用したコードカバレッジ試験のサポートを追加しました。(Michelle Caisse)
- MingwおよびCygwinにおけるout-of-tree構築が可能です。(Richard Evans)

- ソースのクロスコンパイルを行うプラットフォームとしてのMingwの使用を修正しました。(Peter)

E.47.3.11. ソースコード

- 64ビットの時間帯データファイルをサポートします。(Heikki)

これは、2038年以降の夏時間（DST）計算のサポートを追加します。

- プラットフォームの`time_t`データ型の使用を廃止しました。(Tom)

一部のプラットフォームでは64ビットの`time_t`に移行されましたが、一部はまだです。またWindowsではまだどうするか決められていません。しかし、`time_t`と同じ意味を持つ`pg_time_t`定義は（プラットフォームが64ビット整数型を持たない限り）常に64ビットです。そして、すべてのモジュールのAPIとディスク上のデータ書式においてこの型を使用します。

- クロスコンパイル時の時間帯データベース処理における不具合を修正しました。(Richard Evans)
 - 複数ではなく単一処理でバックエンドのオブジェクトファイルをリンクします。(Peter)
 - 複数形のより良い翻訳を可能にするためにgettextサポートを改良しました。(Peter)
 - PL言語にメッセージ翻訳サポートを追加しました。(Alvaro, Peter)
 - より多くのDTraceプローブを追加しました。(Robert Lor)
 - Mac OS X LeopardやSolaris以外のプラットフォーム上のDTraceサポートを有効にしました。(Robert Lor)
 - C文字列と`text`データの間の変換を、この目的のために共通化された関数を提供することで、簡易化および標準化しました。(Brendan Jurd, Tom)
 - フロントエンドプログラムが`postgres.h`をインクルードすることなくインクルードすることができるように、`include/catalog/`ヘッダファイルを整理しました。(Zdenek Kotala)
 - `name`を`char`と同じ整列とし、インデックス内の`name`項目のゼロ埋めを抑制します。(Tom)
 - 動的にロードされたコードが`exit()`を実行した場合の復旧を改良しました。(Tom)
 - プラグインにエグゼキュータを監視させるためのフックを追加しました。(Itagaki Takahiro)
 - プランナの統計情報検索動作を上書きできるようにするフックを追加しました。(Simon Riggs)
 - 独自の共有メモリ要求のために`shmem_startup_hook()`を追加しました。(Tom)
 - インデックスアクセスメソッド`amgetmulti`のエントリポイントを`amgetbitmap`に置き換えました。また、実行時の演算子損失の決定をサポートするために`amgettupple`のAPIを拡張しました。(Heikki, Tom, Teodor)
- GINおよびGiST演算子クラス`consistent`関数のAPIも同様に拡張しました。
- GINインデックスにおいて部分一致検索のサポートを追加しました。(Teodor Sigaev, Oleg Bartunov)
 - `pg_class`の`reltriggers`列を論理値型の`relhastriggers`に置き換えました。(Simon)
- また`pg_class`の使用されない、`relukeys`、`relfkeys`、`relrefs`列を削除しました。
- 一時テーブルの識別を簡単にするために`pg_class`に`relistemp`列を追加しました。(Tom)
 - プラットフォーム向けのFAQを主文書に移しました。(Peter)
 - パーサ入力ファイルが競合状態で構築されることを防ぎます。(Peter)
 - KOI8U（ウクライナ語）符号化方式のサポートを追加しました。(Peter)
 - 日本語翻訳メッセージを追加しました。(Japan PostgreSQL Users Group)
- これは別のプロジェクトとして保守されたものを使用しました。
- MSVC構築システムにおける`LC_MESSAGES`を設定した場合の問題を修正しました。(Hiroshi Inoue, Hiroshi Saito, Magnus)

E.47.3.12. Contrib

- 指定期間を超える問い合わせに対して自動的にEXPLAINを行うcontrib/auto_explainを追加しました。(Itagaki Takahiro, Tom)
- GINインデックスがより多くのデータ型を扱うことを可能にするcontrib/btree_ginを追加しました。(Oleg, Teodor)
- 大文字小文字を区別しない、マルチバイトに対応したテキストデータ型を提供するcontrib/citextを追加しました。(David Wheeler)
- サーバ全体で文の実行統計情報を追跡するcontrib/pg_stat_statementsを追加しました。(Itagaki Takahiro)
- contrib/pgbenchに期間オプションと問い合わせモードオプションを追加しました。(Itagaki Takahiro)
- contrib/pgbenchが、単なるaccounts、branches、history、tellersではなく、pgbench_accounts、pgbench_branches、pgbench_history、pgbench_tellersという名前のテーブルを使用するようにしました。(Tom)

これはpgbenchの実行により、実データが偶然破壊されてしまう危険性を低減させるためのものです。

- 20億ページを超えるテーブルとインデックスを扱うようにcontrib/pgstattupleを修正しました。(Tatsuhito Kasahara)
- contrib/fuzzystrmatchにおいて、ユーザが挿入・削除・置換のコストを指定できる、レーベンシュタイン文字列距離関数を追加しました。(Volkan Yazici)
- contrib/ltreeがマルチバイト符号化方式をサポートするようにしました。(laser)
- contrib/dblinkがSQL/MEDカタログに格納された接続情報を使用できるようにしました。(Joe Conway)
- リモートサーバからのエラーに対するcontrib/dblinkの報告を改良しました。(Joe Conway)
- ローカルなデータベース符号化方式と一致させるためにcontrib/dblinkがclient_encodingを設定するようにしました。(Joe Conway)

これにより、異なる符号化方式を使用するリモートサーバとの通信時の符号化方式に関する問題を防ぎます。

- 偶発的にサーバの.pgpass ファイルが使用されることなく、contrib/dblinkがユーザが提供したパスワードを確実に使用するようにしました。(Joe Conway)

これは小規模なセキュリティの向上です。

- contrib/pageinspectにfsm_page_contents()を追加しました。(Heikki)
- 空き領域マップファイル(*_fsm)をサポートするようにget_raw_page()を変更しました。またcontrib/pg_freespacemapも更新しました。
- contrib/pg_trgmにマルチバイト符号化方式のサポートを追加しました。(Teodor)
- 新しいarray_agg() およびunnest()関数を使用するようにcontrib/intaggを作成し直しました。(Tom)
- contrib/pg_standbyがフェイルオーバー前にすべての利用可能なWALを復旧するようにしました。(Fujii Masao, Simon, Heikki)

この作業を安全にするために、recovery.conf内で新しいrecovery_end_commandオプションを設定して、フェイルオーバー後のトリガファイルを整理する必要があります。pg_standbyは自身でトリガファイルを削除しないようになりました。

- contrib/pg_standbyの-lオプションを無効にしました。シンボリックリンクの使用が安全ではないためです。(Simon)

E.48. リリース8.3.23

リリース日: 2013-02-07

このリリースは8.3.22に対し、各種不具合を修正したものです。8.3メジャーリリースにおける新機能については、[項E.71](#)を参照してください。

本リリースはPostgreSQLの8.3.Xシリーズの最後のリリースとなる予定です。早めに新しいリリースのブランチに更新することを推奨します。

E.48.1. バージョン8.3.23への移行

8.3.Xからの移行ではダンプ/リストアは不要です。

しかし、8.3.17よりも前のバージョンからアップグレードする場合は、8.3.17のリリースノートを参照してください。

E.48.2. 変更点

- SQLから`enum_recv`を実行できないようにしました。(Tom Lane)

この関数は誤った宣言をされており、単純なSQLコマンドでサーバをクラッシュさせることが可能でした。実際のところ、サーバのメモリの内容を検査するために攻撃者がこれを利用することが可能になっていました。この問題を(Secunia SVCRP経由で)報告してくれたSumit Soniに感謝します。(CVE-2013-0255)

- 副SELECTの結果から添字付けあるいは列選択できるようSQLの文法を修正しました。(Tom Lane)
- `pg_tablespace`をスキャンするときに競合問題が発生しないようにしました。(Stephen Frost, Tom Lane)

`pg_tablespace`のエントリに複数の同時更新があるとき、`CREATE DATABASE`および`DROP DATABASE`が正しく動作しない場合があります。

- `DROP OWNED`がデータベース全体あるいはテーブル空間を削除しようとしなかったようにしました。(Álvaro Herrera)
安全のため、これらのオブジェクトは削除するのではなく、所有権を変更すべきです。
- `RowExpr`あるいは`XmlExpr`が2度、解析された時に、誤った動作をしないようにしました。(Andres Freund, Tom Lane)
この誤りは`CREATE TABLE LIKE INCLUDING INDEXES`などの状況においてユーザに明らかになることがありました。
- ハッシュテーブルのサイズ計算で整数のオーバーフローが起きないようにする仕組みを改善しました。(Jeff Davis)
- Windowsで非ASCIIのプロンプト文字列が、確実に正しいコードページに変換されるようにしました。(Alexander Law, Noah Misch)

このバグはpsqlおよび他のいくつかのクライアントプログラムに影響がありました。

- データベースに接続していないとき、psqlの`\?`コマンドがクラッシュする可能性があったのを修正しました。(Meng Qingzhong)
- libpqの`PQprintTuples`でバッファを1バイト行き過ぎてしまう問題を修正しました。(Xi Wang)

この古い関数はPostgreSQL本体によっては使われていませんが、クライアントのプログラムでは今でも使われているかもしれません。

- libedit/libreadlineからの誤ったエクスポートに騙されないよう、configureによる提供関数の確認の順序を調整しました。(Christoph Berg)
- Windowsのビルド番号が時とともに確実に大きくなるようにしました。(Magnus Hagander)
- Windows用にクロスコンパイルしたときに、pgxsが正しく`.exe`の拡張子の付いた実行プログラムを生成するようにしました。(Zoltan Boszormenyi)
- 新しい時間帯の略号`FET`を追加しました。(Tom Lane)

これは一部の東ヨーロッパの時間帯で使われています。

[前のページ](#)
リリース8.4

[ホーム](#)
[上に戻る](#)

[次のページ](#)
リリース8.3.22

E.49. リリース8.3.22

リリース日: 2012-12-06

このリリースは8.3.21に対し、各種不具合を修正したものです。8.3メジャーリリースにおける新機能については、[項E.71](#)を参照してください。

PostgreSQLコミュニティは2013年2月に8.3.Xリリースの更新の公開を停止します。早めにより新しいリリースのブランチに更新することを推奨します。

E.49.1. バージョン8.3.22への移行

8.3.Xからの移行ではダンプ/リストアは不要です。

しかし8.3.17より前のバージョンからアップグレードする場合は、8.3.17のリリースノートを参照してください。

E.49.2. 変更点

- `CREATE INDEX CONCURRENTLY`に関連する複数の不具合を修正しました。(Andres Freund, Tom Lane)

`CREATE INDEX CONCURRENTLY`がインデックスの`pg_index`の行の状態を変更するときに、その場にある更新を使うように修正しました。これにより、同時実行中のセッションが対象のインデックスを更新できず、それにより`CONCURRENTLY`で作成されるインデックスを破損させることになる競合条件を防ぐことができます。

また、`CREATE INDEX CONCURRENTLY` コマンドのエラーの結果として作られる不正なインデックスについて、他の様々な操作が確実に無視するように修正しました。この中で最も重要なのは`VACUUM`です。なぜなら、この不正なインデックスを修正あるいは削除するための修正の処理が実行される前に、自動バキュームが実行されることがあり得るからです。

- メモリが不足した時でも、内部ハッシュテーブルが破損しないようにしました。(Hitoshi Harada)
- 外部結合上の厳格でない等価な句についての計画作成を修正しました。(Tom Lane)

厳格でない構造を他のものと等号で結ぶ句について、プランナが誤った制約を導出する場合があります。例えば、`WHERE COALESCE(foo, 0) = 0`で`foo`が外部結合のNULLになるかもしれない側のもののときです。

- プランナについて、等価なクラスから排他制約を証明する能力を向上しました。(Tom Lane)
- ハッシュされた副計画の部分行のマッチングについて、型をまたがる場合を正しく処理できるよう修正しました。(Tom Lane)

これは`WHERE (a, b) NOT IN (SELECT x, y FROM ...)`のような複数の列についての`NOT IN`の副計画で、例えば`b`と`y`がそれぞれ`int4`型と`int8`型である、というような場合に影響します。この処理の誤りにより、そこで使われているデータ型によって、誤った結果が得られたり、あるいはクラッシュする、ということが起きていました。

- `AFTER ROW UPDATE/DELETE`トリガーで古いタプルを再フェッチする時、バッファロックを取得するようにしました。(Andres Freund)

この誤りにより、非常に特異な状況において、外部キーを強制するトリガーの事前チェックのロジックに対して誤ったデータを渡すことがありました。その結果として、クラッシュを起こしたり、あるいはトリガーを実行するかどうかについて誤った判断をすることがありました。

- `REASSIGN OWNED`について、表領域の`GRANT`が正しく処理できるように修正しました。(Álvaro Herrera)
- ビューのシステム列についての`pg_attribute`の誤ったエントリを無視するようにしました。(Tom Lane)

ビューにはシステム列はありません。しかし、表をビューに変換するとき、そのようなエントリを取り除くことを忘れていました。これは9.3およびそれ以降のリリースでは適切に修正されますが、それ以前のブランチにおいても、誤って変換されたビューが存在しないようにする必要があります。

- ルールの出力について、`INSERT INTO table DEFAULT VALUES`を正しくダンプできるよう修正しました。(Tom Lane)
- 問い合わせが非常に多くの`UNION/INTERSECT/EXCEPT`を含むときでもスタックオーバーフローが発生しないようにしました。(Tom Lane)

- もっとも小さな整数値を-1で割り算した時に、プラットフォームによってはエラーとなるのを修正しました。(Xi Wang, Tom Lane)
- 日付文字列の解析で、文字列の終わりを越えてアクセスすることがあるのを修正しました。(Hitoshi Harada)
- Unixドメインソケットのパス名がプラットフォーム依存の最大長を超えていた場合に、理解可能なエラーメッセージを生成するようにしました。(Tom Lane, Andrew Dunstan)

以前は"Non-recoverable failure in name resolution"(名前解決で解決不能なエラーが発生しました)といった非常に不親切なエラーを起こしていました。

- 複合列の値をクライアントに送るときのメモリリークを修正しました。(Tom Lane)
- `postmaster.pid`ファイルの読み込みについて、`pg_ctl`をより堅牢にしました。(Heikki Linnakangas)

競合条件、およびファイルデスク립タのリークを解決しました。

- `psql`において、不正にエンコードされたデータが入力され、かつ`client_encoding`の設定がSJISのようにクライアントでしか利用できない文字セットである場合にクラッシュする場合があるのを修正しました。(Jiang Guiqing)
- `pg_dump`が`tar`出力形式で生成する`restore.sql`スクリプトの不具合を修正しました。(Tom Lane)

このスクリプトはテーブル名が大文字を含む場合に全く動作しませんでした。また、このスクリプトで通常のCOPYモードだけでなく`--inserts`モードの場合もデータをリストアできるようにしました。

- `pg_restore`でPOSIX準拠の`tar`ファイル进行处理できるよう修正しました。(Brian Weaver, Tom Lane)

`pg_dump`の`tar`出力モードの元々のプログラムは、POSIX標準に完全には準拠していないファイルを生成していました。この問題はバージョン9.3で修正されます。このパッチは9.3より前のブランチを修正するもので、正しくないフォーマットと正しいフォーマットの両方を処理できるようにします。これにより、9.3がリリースされた時に互換性の問題を防ぐことができます。

- `pg_resetxlog`について、データディレクトリへの相対パスが与えられた時に、`postmaster.pid`の場所を正しく特定できるよう修正しました。(Tom Lane)

この誤りにより、データディレクトリを使用している動作中の`postmaster`があることに`pg_resetxlog`が気が付かないことがありました。

- `libpq`の`lo_import()`関数および`lo_export()`関数が、ファイルI/Oのエラーを適切に報告するよう修正しました。(Tom Lane)
 - `ecpg`について、ネストした構造体のポインタ変数の処理を修正しました。(Muhammad Usama)
 - `contrib/pageinspect`のBツリーページ検査関数で、ページの検査中はバッファロックを取得するようにしました。(Tom Lane)
 - AIX上でのロードモジュールのビルドについて`pgxs`のサポートを修正しました。(Tom Lane)
- 元々のソースツリーの外側でモジュールをビルドしてもAIXでは正しく動作しませんでした。
- 時間帯データファイルを`tzdata`リリース2012jに更新しました。キューバ、イスラエル、ヨルダン、リビア、パレスチナ、西サモア、およびブラジルの一部での夏時間の変更が含まれます。

E.50. リリース8.3.21

リリース日: 2012-09-24

このリリースは8.3.20に対し、各種不具合を修正したものです。 8.3メジャーリリースにおける新機能については[項E.71](#)を参照してください。

PostgreSQLコミュニティは2013年2月に8.3.Xリリースの更新の公開を停止します。 早めにより新しいリリースのブランチに更新することを推奨します。

E.50.1. バージョン8.3.21への移行

8.3.Xからの移行ではダンプ/リストアは不要です。

しかし8.3.17より前のバージョンからアップグレードする場合は8.3.17のリリースノートを参照してください。

E.50.2. 変更点

- GiSTインデックスにおけるページ分割の決定を改善しました。(Alexander Korotkov, Robert Haas, Tom Lane)
複数列のGiSTインデックスで、この間違いにより、予期しない膨張が起きることがありました。
- 権限が取り消されても、その権限を引き続き保持している場合は、取り消しが連鎖しないようにしました。(Tom Lane)
あるロールxからグラントオプションを取り消したが、xが他の誰かから付与されたことによってそのオプションを引き続き保持している、という場合は、ロールyがxによって付与されたその権限を再帰的に取り消すべきではありません。
- PL/Perlが使われている時のSIGFPEの処理について修正しました。(Andres Freund)
PerlはプロセスのSIGFPEハンドラをSIG_IGNにリセットするため、後になってクラッシュが起きる場合があります。
PL/Perlを初期化した後で、Postgresの通常のシグナルハンドラをリストアするようにしました。
- PL/Perlの再帰関数が実行時に再定義される場合に、PL/Perlがクラッシュしないようにしました。(Tom Lane)
- PL/Perlで正しく最適化されない場合があるのを回避しました。(Tom Lane)
一部のLinuxディストリビューションでは、誤ったバージョンのpthread.hがインストールされており、PL/Perlのコンパイルされたコードが誤ったものになります。 この結果、PL/Perlの関数がエラーを発生させる他の関数を呼び出すと、クラッシュすることがありました。
- 時間帯データファイルをtzdataリリース2012fに更新しました。 フィジーでの夏時間の変更が含まれます。

E.51. リリース8.3.20

リリース日: 2012-08-17

このリリースは8.3.19に対し、各種の不具合を修正したものです。 8.3メジャーリリースにおける新機能については[項E.71](#)を参照してください。

PostgreSQLコミュニティは2013年2月に8.3.x系リリースに対する更新を停止します。 ユーザにはより新しいリリース系列への更新を勧めます。

E.51.1. バージョン8.3.20への移行

8.3.Xからの移行ではダンプ/リストアは不要です。

しかし8.3.17より前のバージョンからアップグレードする場合は8.3.17のリリースノートを参照してください。

E.51.2. 変更点

- XML実体参照を介した外部ファイル/URLへのアクセスを防止します。(Noah Misch, Tom Lane)

`xml_parse()` は、XML値で参照されるDTDと実体参照を解決するために必要な外部ファイルやURLの入手を試みます。 これにより、権限を持たないデータベースユーザがデータベースサーバの権限でデータを入手することができてしまいました。 この外部データは直接ユーザには返されませんが、データが有効なXMLとして解析できなかった場合その一部がエラーメッセージの中に漏洩することがあり得ます。 またどのような場合であっても、ファイルが存在するかどうか検査できるだけでも攻撃者にとって有用になり得ます。(CVE-2012-3489)

- `contrib/xml2`の`xslt_process()`を介した外部ファイル/URLへのアクセスを防止します。(Peter Eisentraut)

`libxslt`はスタイルシートコマンドを通してファイルとURLの両方を読み書きする機能を提供します。 これにより、権限を持たないデータベースユーザがデータベースサーバの権限でデータを読み書きすることができてしまいました。 `libxslt`のセキュリティオプションを適切に使用することでこれを無効にします。(CVE-2012-3488)

また`xslt_process()`の外部ファイル/URLからドキュメントとスタイルシートを取り込む機能を取り除きました。 これは文書化された"機能"でしたが、推奨されないものと長くみなされていました。 CVE-2012-3489に対する修正のためこの能力が壊れましたが、その修正を行うことは止め、単に取り除きました。

- 早すぎるbtreeインデックスページの回収を防止します。(Noah Misch)

読み取り専用トランザクションによるXIDの割り当てを省略できるようにした時、読み取り専用トランザクションがまだ参照している時に削除されたbtreeページを回収されてしまう可能性をもたらしました。 これは間違ったインデックス検索結果という結果になります。 時間に関する必要条件のため、こうしたエラーが実際に発生する可能性は非常に低いものでしたが、修正しなければなりません。

- 新しく作成された、またはリセットされたシーケンスに関するクラッシュに対する安全性を修正しました。(Tom Lane)

`ALTER SEQUENCE`が新しく作成されたシーケンスまたはリセットされたシーケンスに対して実行された場合、その後`nextval()`呼び出しが一度だけ行われ、そして、サーバがクラッシュすると、WAL再生はシーケンスを`nextval()`が行われていないように見える状態にリストアします。 このため次の`nextval()`呼び出しにより最初のシーケンス値が再び返されることになります。 シリアル列のシーケンス作成には`ALTER SEQUENCE OWNED BY`処理が含まれていますので、特にこれは`serial`列で現れます。

- `pg_start_backup()`の後確実に`backup_label`ファイルを`fsync`します。(Dave Kerr)

- 9.1からのバックパッチは`fsync`要求キューの圧縮を向上しました。(Robert Haas)

チェックポイント期間の性能を向上します。 9.1の変更は安全に過去のバージョンにパッチを充てられると判断できる程度に実地試験が行われたようです。

- 自動バキュームは直接ブロックされたプロセスによってのみ自動キャンセルさせることができます。(Tom Lane)

元々のコードでは一部の場合に一貫性がない動作を許していました。 具体的には`deadlock_timeout`猶予期間より短時間で自動バキュームはキャンセルされました。

- 自動バキュームのキャンセルについてのログ処理を改良しました。(Robert Haas)
- サーバ起動後一番最初のログローテーションの間`log_truncate_on_rotation`が動作するようにログコレクタを修正しました。(Tom Lane)
- 副問い合わせへの行全体の参照が余計な`GROUP BY`や`ORDER BY`列を含まないことを確実にしました。(Tom Lane)
- `CREATE TABLE`時の`CHECK`制約およびインデックス定義における行全体の参照の複製を許しません。(Tom Lane)

この状況は`LIKE`または`INHERITS`を持つ`CREATE TABLE`で発生することがあり得ます。複製される行全体の変数は、新しいテーブルではなく元のテーブルの行型で正しくラベル付けされませんでした。`LIKE`では、後で行型がよく分岐しますので、このような状況を拒絶する方が合理的のように思います。`INHERITS`では理想的には、親のテーブルの行型への暗黙的な変換によって許可すべきですが、安全に以前のバージョンにパッチを充てるにはより多くの作業が必要です。

- `ARRAY(SELECT ...)` 副問い合わせにおけるメモリリークを修正しました。(Heikki Linnakangas, Tom Lane)
- 正規表現からの共通接頭辞の抽出を修正しました。(Tom Lane)

このコードでは`^(foo)?bar`などの修飾された括弧付きの副式で混乱してしまふことがあり得ました。こうしたパターンの検索に対して不正確なインデックス最適化をもたらしました。

- `contrib/xml2`の`xslt_process()`においてエラーを適切に報告します。(Tom Lane)
- 時間帯データファイルを`tzdata`リリース2012eに更新しました。モロッコおよびトケラウにおける夏時間の変更が含まれます。

[前のページ](#)
リリース8.3.21

[ホーム](#)
[上に戻る](#)

[次のページ](#)
リリース8.3.19

E.52. リリース8.3.19

リリース日: 2012-06-04

このリリースは8.3.18に対し、各種の不具合を修正したものです。 8.3メジャーリリースにおける新機能については[項E.71](#)を参照してください。

E.52.1. バージョン8.3.19への移行

8.3.Xからの移行ではダンプ/リストアは不要です。

しかし8.3.17より前のバージョンからアップグレードする場合は8.3.17のリリースノートを参照してください。

E.52.2. 変更点

- contrib/pgcryptoのDES `crypt()` 関数における不正なパスワード変換を修正しました。(Solar Designer)

パスワード文字列に`0x80`というバイト値が含まれる場合、パスワードの残りが無視され、見かけよりも非常に弱いパスワードになってしまいました。この修正によって、文字列の残りが適切にDESハッシュ内に含まれるようになります。したがって、この不具合の影響を受ける、格納済みのパスワード値はすべて一致なくなりますので、格納済みの値を更新しなければなりません。(CVE-2012-2143)

- 手続き言語の呼び出しハンドラにおいてSECURITY DEFINERおよびSET属性を無視します。(Tom Lane)

こうした属性を呼び出しハンドラに与えるとサーバがクラッシュするかもしれません。(CVE-2012-2655)

- 数字による時間帯オフセットによりUTCから16時間まで離れたtimestampの入力を行うことができます。(Tom Lane)

一部の歴史的な時間帯はこれまでの上限であった15時間よりも離れたオフセットを持ちます。このためリロード時にダンプされたデータ値が拒絶される結果になることがありました。

- 与えられた時刻が現在の時間帯の最後のDST遷移時刻と正確に一致する場合をうまく処理するようにタイムスタンプ変換を修正しました。(Tom Lane)

長い間これは見落とされていました。ほとんどのDSTを使用する時間帯では、将来のDST変更が無限に続くものと仮定していましたので、これまで気付きませんでした。

- textからnameへのキャストおよびcharからnameへのキャストがマルチバイト符号化方式で正しく文字列の切り詰めを行うように修正しました。(Karl Schnaitter)

- to_tsquery()のメモリコピーにおける不具合を修正しました。(Heikki Linnakangas)

- pg_attributeが非常に大きい場合の低速なセッション起動を修正しました。(Tom Lane)

pg_attributeがshared_buffersの1/4を超える場合、セッション起動時に時々必要となるキャッシュ再構築コードが同期スキャンロジックを開始し、通常より長い期間かかるようになりました。この問題は、特に多くの新しいセッションが一度に開始する場合に深刻です。

- 適切な頻度で発生する問い合わせキャンセルに対する連続的なスキャン検査を確実に行います。(Merlin Moncure)

有効なタプルを含まない連続するページを多くを巡るスキャンは、その間、中断に応答しません。

- WindowsのPGSemaphoreLock()の実装は、戻る前にImmediateInterruptOKを確実に消去します。(Tom Lane)

この見落としは、同じ問い合わせの中で後で受け取った問い合わせキャンセル中断が、安全ではない時に受け付けられ、予期しない、悪化した結果をもたらす可能性があったことを意味します。

- ビューまたはルールを出力する時に行全体の変数を安全に表示します。(Abbas Butt, Tom Lane)

曖昧な名前を含む(つまりその名前がテーブル名か問い合わせの列名かいずれかになる)境界条件では、曖昧な方法で出力し、ビューまたはルールがダンプとリストアの間で別の解釈がされる危険性がありました。操作を行わないキャストを付与することで曖昧な状況を防ぎます。

自動バキュームのワーカプロセスは確実にスタック長の検査を適切に行います。(Heikki Linnakangas)

これまでは、自動ANALYZEによって呼び出される関数の中の無限再帰がワーカプロセスをクラッシュさせる可能性があります。

- 高負荷時にログの干渉性が失われるようにログコレクタを修正しました。(Andrew Dunstan)

これまでは、コレクタは非常に忙しい場合、大規模メッセージの再構成に失敗する可能性があります。

- SIGHUPを受信した後にファイルのローテーションを確実に再開するようにログコレクタを修正しました。(Tom Lane)
- 対象が関数の第一変数であった場合のPL/pgSQLのGET DIAGNOSTICS コマンドを修正しました。(Tom Lane)
- データベースに多くのオブジェクトが含まれている場合のpg_dumpの複数の性能問題を修正しました。(Jeff Janes、Tom Lane)

データベースに多くのスキーマが含まれる場合、多くのオブジェクトが依存関係のあるループにある場合、所有されたシーケンスが多く存在する場合に、pg_dumpが非常に低速になる可能性があります。

- エラー時に一時データベース接続をリークしないように、contrib/dblinkのdblink_exec()を修正しました。(Tom Lane)
- 時間帯データファイルをtzdataリリース2012cに更新しました。南極、アルメリア、チリ、キューバ、フォークランド諸島、ガザ、ハイチ、ヘブロン、モロッコ、シリア、トケラウ諸島における夏時間の変更、カナダに関する歴史的な変更が含まれます。

[前のページ](#)

リリース8.3.20

[ホーム](#)
[上に戻る](#)

[次のページ](#)

リリース8.3.18

E.53. リリース8.3.18

リリース日: 2012-02-27

このリリースは8.3.17に対し、各種の不具合を修正したものです。 8.3メジャーリリースにおける新機能については[項E.71](#)を参照してください。

E.53.1. バージョン8.3.18への移行

8.3.Xからの移行ではダンプ/リストアは不要です。

しかし8.3.17より前のバージョンからアップグレードする場合は8.3.17のリリースノートを参照してください。

E.53.2. 変更点

- `CREATE TRIGGER`においてトリガ関数に対する実行権限が必要になりました。(Robert Haas)

この検査が無かったため、別のユーザが自身が所有するテーブルにその関数をインストールすることで、偽造した入力データでトリガ関数を実行することができました。 `SECURITY DEFINER`が付いたトリガ関数でのみ重大です。 その他のトリガ関数ではとにかくテーブル所有者として実行されるためです。(CVE-2012-0866)

- `pg_dump`のコメントに記述された名前の中の改行を空白に変換します。(Robert Haas)

`pg_dump`は、その出力スクリプトにおけるSQLコメント内で発行されるオブジェクト名のサニタイズに関して注意を払っていませんでした。 改行を含む名前は少なくともそのスクリプトを構文的に不正にさせます。 悪意をもって組み立てられたオブジェクト名によって、スクリプトがリロードする時にSQLインジェクションの危険性があり得ました。(CVE-2012-0868)

- バキューム処理と同時に挿入を行った場合のbtreeインデックス破損を修正しました。(Tom Lane)

挿入によって発生するインデックスページ分割によって、同時実行中の**VACUUM**が削除すべきインデックス項目を削除し損なうことが時々発生することがありました。 対応するテーブル行が削除された後、対応先が無いインデックス項目によってエラー ("could not read block N in file ..."など)、最悪は、解放されたテーブル位置に再挿入された関係がない行にちなんだ、警告のない間違った問い合わせ結果が引き起こされます。 この不具合は8.2から存在していましたが、あまり頻発に発生しませんでしたので、これまで究明されませんでした。 使用中のデータベースで発生していたのではと疑わしければ、対象のインデックスを再インデックス付けすることで修正されます。

- `ALTER USER/DATABASE SET`における一部の設定で存在しない値を許可します。(Heikki Linakangas)

`default_text_search_config`、`default_tablespace`、`temp_tablespaces`を未知の名前に設定することができます。 これらが実際に使用されている別のデータベースでは既知であるかもしれませんが、またテーブル空間の場合はまだテーブル空間が作成されていないかもしれないからです。 過去`search_path`においても同じ問題がありましたが、これらの設定も同様に動作するようにしました。

- WAL再生中にOIDカウンタを、たとえ周回していたとしても、正しく追跡します。(Tom Lane)

これまでは、OIDカウンタはシステムが再生モードを抜けるまで高値のままになりました。 実際にはほとんどnilという結果になりますが、マスタに昇格されるスタンバイサーバでは、値が必要になってから、OIDカウンタを合理的な値まで進めるために長時間かかる可能性がありました。

- *が付いた正規表現の後方参照を修正しました。(Tom Lane)

コードでは、正確な文字列一致を強制せずに、実質後方参照シンボルで参照されるパターン副式を満たす任意の文字列を受け付けました。

同様の問題はまだ、直接量指定子のサブジェクトとならない、より大きく量化された式に埋め込まれた後方参照でも残っています。 こちらは将来のリリースのPostgreSQLで対応予定です。

- `inet/cidr`値の処理に最近入ったメモリリークを修正しました。(Heikki Linnakangas)

2011年12月のPostgreSQLリリース内のパッチによって、これらの操作にメモリリークが発生しました。 これらの列に対するbtreeインデックス等で重大な問題になる可能性がありました。

- Windowsのsysloggerにおけるファイルハンドルの二重クローズを防ぎます。(MauMau)

通常はこのエラーは表面化しませんが、Windowsのデバッグ版を実行している場合は例外が引き起こります。

- plpgsqlにおけるI/O変換関連のメモリリークを修正しました。(Andres Freund、Jan Urbanski、Tom Lane)

現在の関数が終わるまで、特定の操作がメモリリークしました。

- 継承されたテーブル列に対するpg_dumpの取り扱いを改良しました。(Tom Lane)

pg_dumpは、子の列が親の列と異なるデフォルト式を持つという状況を間違っ扱いしました。デフォルトがテキストとして親のデフォルトと同一であるが、実際は同一ではない場合（例えば、スキーマ検索パスの違いのため）、異なるものとして認識せず、そのため、ダンプしリストアした後、子は親のデフォルトを継承することができました。子の列がNOT NULLであり親がそうではなかった場合も、微妙に間違っリストアされました。

- INSERT形式のテーブルデータに対するpg_restoreの直接データベースにリストアするモードを修正しました。(Tom Lane)

他の問題に対する修正における見落としの結果、2011年9月または12月付けのリリースに含まれるpg_restoreでは、--insertsまたは--column-insertsオプションを付けて作成されたアーカイブファイルを直接データベースにリストアすることができませんでした。アーカイブファイル自体には失敗はなく、テキストモード出力では問題ありませんでした。

- contrib/intarrayのint[] & int[] 演算子のエラーを修正しました。(Guillaume Lelarge)

2つの入力配列が共通して持つ最小の整数が1であり、どちらかの配列により小さな値があった場合、1が結果から間違っ省かれました。

- contrib/pgcryptoのencrypt_iv() およびdecrypt_iv() の誤判定を修正しました。(Marko Kreen)

これらの関数は特定の種類の無効入力エラーの通知に失敗し、不正な入力に対してランダムなゴミの値を代わりに返しました。

- contrib/test_parserにおける1バイトのバッファオーバーランを修正しました。(Paul Guyot)

コードでは必要より1バイト多く読み取ろうとし、境界条件ではクラッシュします。contrib/test_parserは単なるサンプルコードであり、これ自体はセキュリティ問題ではありませんが、例のコードとしては良くありません。

- ARMにおいて可能ならばスピンロックに__sync_lock_test_and_set()を使用します。(Martin Pitt)

この関数は、以前の廃止予定でARMv6以降では使用できなくなったSWPB命令の使用を置き換えるものです。最近のARMボードでも古いコードは明白な方法で失敗することはありませんが、単に同時アクセスのインターロックを行わず、マルチプロセス操作において奇妙な失敗をもたらすと報告されています。

- 受け付け可能なバージョンのgccを用いて構築する場合に-fexcess-precision=standardオプションを使用します。(Andrew Dunstan)

最近のバージョンのgccは独創的な結果を生成するというさまざまなシナリオを防ぎます。

- FreeBSDにおいてスレッド化されたPythonを使用できるようにしました。(Chris Rees)

configureスクリプトはこれまで、この組み合わせは動作しないと前提していました。しかしFreeBSDではこの問題が修正されたので、エラー検査を取り除きました。

E.54. リリース8.3.17

リリース日: 2011-12-05

このリリースは8.3.16に対し、各種の不具合を修正したものです。 8.3メジャーリリースにおける新機能については[項E.71](#)を参照してください。

E.54.1. バージョン8.3.17への移行

8.3.Xからの移行ではダンプ/リストアは不要です。

しかし、`information_schema.referential_constraints`ビュー定義において長期に渡って潜在したエラーが見つかりました。このビューからの正確な結果に依存している場合、後述の変更点の最初で説明する通りその定義を置き換えなければなりません。

また8.3.8以前のバージョンからアップデートする場合は8.3.8のリリースノートを参照してください。

E.54.2. 変更点

- `information_schema.referential_constraints`ビューの不具合を修正しました。(Tom Lane)

このビューでは、外部キーと依存するプライマリキーまたは一意キー制約との対応付けについて十分な注意を行っていませんでした。このため、外部キー制約がまったく表示されない、何回も表示される、あるいは、実際の依存する制約とは異なる制約に依存するものと言い張るという結果になることがありました。

このビュー定義はinitdbによってインストールされますので、ただアップグレードしただけでは問題は解消されません。既存のインストレーションでこの問題を解消させなければならない場合、（スーパーユーザとして）`information_schema`スキーマを削除して、`SHAREDIR/information_schema.sql`を元に再作成することができます。（`SHAREDIR`の場所が分からない場合は`pg_config --sharedir`を実行してください。）これは修正対象のデータベースそれぞれで繰り返さなければなりません。

- `<!= - Fix TOAST-related data corruption during CREATE TABLE dest AS SELECT * FROM src or INSERT INTO dest SELECT * FROM src` (Tom Lane) --> `CREATE TABLE dest AS SELECT * FROM src`または`INSERT INTO dest SELECT * FROM src`時のTOAST関連のデータの破損を修正しました。(Tom Lane)

テーブルが`ALTER TABLE ADD COLUMN`によって変更された場合、データを逐語的に別のテーブルにコピーしようとした時に特定の境界条件で破損した結果になることがありました。この問題は、8.4以降において、この構文においてのみ明確に発生し得たものですが、同じ不具合を引き起こす他のコードパスがある場合に備えて、より過去のバージョンにおいてもパッチを適用しました。

- 無効なsyscache項目からTOASTテーブルにアクセスする時の競合条件を修正しました。(Tom Lane)

典型的な兆候は、"missing chunk number 0 for toast value NNNNN in pg_toast_2619"のような一時的なエラーです。ここで挙げられるTOASTテーブルは常にシステムカタログに属します。

- `DatumGetInetP()` が1バイトヘッダを持つinetデータを展開するようにしました。さらに展開を行わない、新しいマクロ`DatumGetInetPP()`を追加しました。(Heikki Linnakangas)

この変更がコアコードに影響することはありませんが、`DatumGetInetP()`が通常の規約の通りに展開したデータを生成することを想定しているアドオンのコードにおけるクラッシュを防止できるかもしれません。

- `money`型の入出力におけるロケールのサポートを改良しました。(Tom Lane)

すべての標準`lc_monetary`書式付けオプションをサポートしていないという点以外にも、入力関数と出力関数に一貫性がなかったため、ダンプされた`money`値では再度読み取ることができないロケールがありました。

- `transform_null_equals`が`CASE foo WHEN NULL ...`式に影響を与えないようにしました。(Heikki Linnakangas)

`transform null equals`は、`CASE`構文によって内部的に生成された等価な検査ではなく、直接ユーザによって書かれた`foo = NULL`に影響を与えることのみを前提としています。

- 自己参照外部キーをより良くサポートするために、外部キートリガの生成順序を変更しました。(Tom Lane)

カスケードしている外部キーが自身のテーブルを参照する場合、1行の更新が1つのイベントでON UPDATEトリガとCHECKトリガの両方を発行します。ON UPDATEトリガを最初に実行しなければなりません。そうしないと、CHECKは最終的ではない行の状態を検査することになり、不適切なエラーが発生する可能性があります。しかしこれらのトリガの発行順序はその名前によって決まります。トリガは"RI_ConstraintTrigger_NNNN"という規約に従った名前で自動生成されますので、一般的には作成順序によって決まります。適切な修正にはこの規約の変更が必要です。9.2で変更する予定ですが、既存のリリースで変更することには危険があるように思われます。このため、このパッチはトリガの作成順序を変更するだけです。この種のエラーが起きたユーザは、トリガを正しい順序にするために、外部キー制約を削除し再作成しなければなりません。

- バッファ割当率を追跡する際の浮動小数アンダーフローを防止します。(Greg Matthews)

これ自体は害がないものですが、特定のプラットフォームではこれにより煩わしいカーネルログメッセージが現れます。

- psqlのコマンド履歴内で空行を含むコマンドを保持します。(Robert Haas)

これまでの動作では、例えば文字列リテラル内の空行が削除された場合に問題が起きました。

- テーブルの行型など自動生成された型の間でユーザが定義したキャストをダンプするようにpg_dumpを修正しました。(Tom Lane)
- PL/Perlの構築時に好みのバージョンのxsubppを使用します。オペレーティングシステムのメインコピーである必要はありません。(David Wheeler、Alex Hunsaker)

- contrib/dict_intおよびcontrib/dict_xsyn内の間違ったコードを修正しました。(Tom Lane)

一部の関数で、palloc()で返されるメモリはゼロ埋めされていることが保証されていると間違っただけを前提をしていました。

- pgstatindex()において問い合わせキャンセル割込みを適切に受け付けます。(Robert Haas)
- VPATH構築ですべてのサーバヘッダファイルが適切にインストールされることを確実にしました。(Peter Eisentraut)
- 冗長エラーメッセージ内で報告されるファイル名を短くしました。(Peter Eisentraut)

通常の構築では常にCファイルの名前だけがエラーメッセージ呼び出しの中で報告されました。しかしこれまでVPATH構築では絶対パス名が報告されました。

- 中央アメリカ用のWindowsの時間帯名の解釈を修正しました。(Tom Lane)

通常中央アメリカではまったく夏時間が認められませんので、"Central America Standard Time"をCST6CDTではなくCST6に対応付けしました。

- 時間帯データファイルをtzdata release 2011nに更新しました。ブラジル、キューバ、フィジー、パレスチナ、ロシア、サモアにおける夏時間規則の変更、アラスカ、イギリス領東アフリカの歴史的な修正が含まれています。

[前のページ](#)

リリース8.3.18

[ホーム](#)
[上に戻る](#)

[次のページ](#)

リリース8.3.16

E.55. リリース8.3.16

リリース日: 2011-09-26

このリリースは8.3.15に対し、各種の不具合を修正したものです。 8.3メジャーリリースにおける新機能については[項E.71](#)を参照してください。

E.55.1. バージョン8.3.16への移行

8.3.Xからの移行ではダンプ/リストアは不要です。 しかし8.3.8以前のバージョンからアップデートする場合は8.3.8のリリースノートを参照してください。

E.55.2. 変更点

- HOT更新されたか疑わしいタブルのインデックス処理における不具合を修正しました。(Tom Lane)

この不具合のために、システムカタログの再インデックス処理の後インデックスが破損する可能性がありました。 ユーザインデックスには影響しないはずです。

- GiSTインデックスページ分割処理における複数の不具合を修正しました。(Heikki Linnakangas)

発生する可能性は小さいのですが、インデックスの破損を招く可能性がありました。

- `tsvector_concat()` におけるバッファオーバーランの可能性を修正しました。(Tom Lane)

この関数は必要なメモリを少なく見積もり、その結果サーバクラッシュを招く可能性がありました。

- `xml_recv`において"standalone"パラメータ処理時のクラッシュを修正しました。(Tom Lane)

- `ANALYZE`およびSJIS-2004符号化変換においてメモリの最後を超えてアクセスする可能性を修正しました。(Noah Misch)

これは非常に可能性が低いサーバクラッシュ状況を修正します。

- `relcache`初期ファイルの無効化における競合状態を修正しました。(Tom Lane)

新しいバックエンドプロセスが無効な初期ファイルを読み取ろうとするが、データが無効であることを通知する`inval`メッセージを取り損なう期間がありました。 この結果カタログアクセスにおいて、起動処理の後の"could not read block 0 in file ..."といった奇妙な失敗が起こります。

- GiSTインデックススキャン終了時のメモリリークを修正しました。(Tom Lane)

すでに多くの行を持つテーブルに対して新たなGiSTを基にした排他制約を作成する際の検証など、多くの分割GiSTインデックススキャンを行うコマンドが一時的にこのリークの原因となる大容量のメモリを求めることがあり得ました。

- 大規模かつ非可逆なビットマップを構築する時の性能問題を修正しました。(Tom Lane)

- 配列作成、経路作成関数のパッド用バイトを確実にゼロにするように修正しました。(Tom Lane)

これにより、プランナが意味的に等しい定数を等しくないとみなし、最適化を悪化させる一部の状況を防ぎます。

- WAL再生を壊すgcc 4.5.0の不具合を回避します。(Tom Lane)

サーバクラッシュ後にコミットされたトランザクションの損失をもたらす可能性がありました。

- ビュー内の`VALUES`についてのダンプ不具合を修正しました。(Tom Lane)

- シーケンスに対する`SELECT FOR UPDATE/SHARE`を許しません。(Tom Lane)

この操作は想定通りに動作せず、また、失敗をもたらすことがあり得ました。

- ハッシュテーブルのサイズ計算時の整数オーバーフローから保護します。(Tom Lane)

- `CLUSTER`がすでに削除されたTOASTデータへアクセスしようとする状況を修正しました。(Tom Lane)

- "peer" 認証に関する証明書制御メッセージの使用における移植性不具合を修正しました。(Tom Lane)
- 複数の往復が必要な場合のSSPIログインを修正しました。(Ahmed Shinwari、Magnus Hagander)
この問題の典型的な兆候はSSPIログイン期間の"The function requested is not supported"エラーです。
- `pg_srand48` シード初期化における記述ミスを修正しました。(Andres Freund)
このため、提供されたシードの全ビットを使用すると失敗する可能性があります。この関数はほとんどのプラットフォームで使用されていません(`srandom`を持たないプラットフォームのみです)ので、想定よりもランダム性が小さいシードによるセキュリティ露見の可能性はとにかく最小だったと思われます。
- `LIMIT` と `OFFSET` 値の合計が 2^{63} を超えた場合の整数オーバーフローを防止します。(Heikki Linnakangas)
- `generate_series()` の `int4` 版と `int8` 版にオーバーフロー検査を追加しました。(Robert Haas)
- `to_char()` における末尾のゼロ除去を修正しました。(Marti Raudsepp)
小数点の後の桁位置がない `FM` 付きの書式において、小数点より左のゼロが正しく除去できていませんでした。
- 2^{63} 近辺の入力に対するオーバーフローを防ぐように `pg_size_pretty()` を修正しました。(Tom Lane)
- `pg_ctl` において、Windowsのサービス登録用の静寂モードをサポートしました。(MauMau)
- 異なるファイルから `COPY` している時の、`psql` のスクリプトファイル行番号の計数処理を修正しました。(Tom Lane)
- `standard_conforming_strings` に合わせて、`pg_restore` の直接データベースモードを修正しました。(Tom Lane)
`pg_restore` が、`standard_conforming_strings` が `on` に設定されたアーカイブファイルからデータベースサーバに直接リストアする時に間違ったコマンドを発行する可能性があります。
- `libpq` のLDAPサービス検索コードにおけるwrite-past-buffer-endとメモリリークを修正しました。(Albe Laurenz)
- `libpq` において、非ブロッキングI/OとSSL接続を使用する場合の失敗を防止します。(Martin Pihlak、Tom Lane)
- `libpq` の接続開始期間のエラーの取扱いを改良しました。(Tom Lane)
具体的には、SSL接続開始期間の `fork()` 失敗についてのサーバ報告への応答がより健全になりました。
- SSL失敗に関する `libpq` のエラー報告を改良しました。(Tom Lane)
- `ecpglib` が `double` 値を15桁の精度で書き込むようにしました。(Akira Kurosawa)
- `ecpglib` において、エラー後確実に `LC_NUMERIC` の設定を元に戻します。(Michael Meskes)
- `blowfish` の符号付き文字に関する不具合(CVE-2011-2483)に対する上流の修正を適用しました。(Tom Lane)
`contrib/pg_crypto` の `blowfish` 暗号化コードは、`char` が符号付きであるプラットフォーム(ほとんどのプラットフォーム)において間違った結果を生成し、暗号化されたパスワードが本来より脆弱になりました。
- `contrib/seg` におけるメモリリークを修正しました。(Heikki Linnakangas)
- 空のインデックスに対して一貫性を持った結果を生成するように `pgstatindex()` を修正しました。(Tom Lane)
- `perl 5.14` を用いたビルドを可能にしました。(Alex Hunsaker)
- システム関数の存在を検出するための `configure` スクリプトの方法を更新しました。(Tom Lane)
`8.3` と `8.2` で使用している `autoconf` のバージョンは、リンク時の最適化を行うコンパイラによってだまされることがあり得ました。
- 空白を含むファイルパスでのビルドとインストールに関連した問題を修正しました。(Tom Lane)
- 時間帯データファイルを `tzdata release 2011i` に更新しました。カナダ、エジプト、ロシア、サモア、南スーダンにおける夏時間規則の変更が含まれています。

E.56. リリース8.3.15

リリース日: 2011-04-18

このリリースは8.3.14に対し、各種の不具合を修正したものです。 8.3メジャーリリースにおける新機能については[項E.71](#)を参照してください。

E.56.1. バージョン8.3.15への移行

8.3.Xからの移行ではダンプ/リストアは不要です。 しかし8.3.8以前のバージョンからアップデートする場合は8.3.8のリリースノートを参照してください。

E.56.2. 変更点

- 複合型自身を複合型に含めることを許可しません。(Tom Lane)

これは、複合型を処理する際にサーバが無限再帰をしてしまう状況を防止します。 こうした構造の使い道は一部ありますが、常に安全に動作することを確実にするために必要な作業量を正当化するほどの説得力はないと思われます。

- カタログキャッシュの初期化中のデッドロックの可能性を取り除きました。(Nikhil Sontakke)

一部のキャッシュを読み取るコードが、インデックスのカタログをロックする前にシステムインデックスに対して共有ロックを獲得する場合があります。 これは他でより標準的な順序で排他ロックを獲得しようとする処理とデッドロックする可能性があります。

- 対象タプルに対する同時更新があった時のBEFORE ROW UPDATEトリガ処理における、不正な領域を指し示すポインタ問題を修正しました。(Tom Lane)

この不具合は、UPDATE RETURNING ctidを実行しようとした時に、断続的に"cannot extract system attribute from virtual tuple"で失敗するという結果で気付きました。 更新したタプルに対して不正なインデックス項目が生成されてしまうなど、より深刻なエラーとなる可能性がごくわずかながら存在します。

- 遅延トリガイベントを待たせているテーブルに対するDROP TABLEを許可しません。(Tom Lane)

以前はDROPが進み、トリガが最終的に発行された時に"could not open relation with OID nnn"エラーをもたらしました。

- PL/Pythonの部分配列化に関するメモリリークを修正しました。(Daniel Popowich)
- TOCファイル内の(1KB以上の)長い行に対処するようにpg_restoreを修正しました。(Tom Lane)

- コンパイラの過度に積極的な最適化によるゼロ除算が原因のクラッシュに対して、より多くの対策を取りました。(Aurelien Jarno)

- MIPS上のFreeBSDとOpenBSDでdlopen()の使用をサポートしました。(Tom Lane)

これらのシステムではこのシステム関数はMIPSハードウェアで利用できないという仮定を直接書き込んだコードがありました。 より最近のバージョンでは利用できますので、コンパイル時の試験を使用するように変更しました。

- HP-UXにおけるコンパイル失敗を修正しました。(Heikki Linnakangas)
- Windowsにおけるlibintlのバージョン非互換問題を修正しました。(Hiroshi Inoue)
- Windowsのビルドスクリプト内のxcopyの使用がWindows 7で正しく動作するように修正しました。(Andrew Dunstan)

ビルドスクリプトのみに影響し、インストールや使用には影響しません。

- Cygwinにおいてpg_regressで 사용되는パス区切り文字を修正しました。(Andrew Dunstan)
- 時間帯データファイルをtzdata release 2011fに更新しました。 チリ、キューバ、フォークランド諸島、モロッコ、サモア、トルコにおける夏時間規則の変更、南オーストラリア、アラスカ、ハワイの歴史的な修正が含まれています。

E.57. リリース8.3.14

リリース日: 2011-01-31

このリリースは8.3.13に対し、各種の不具合を修正したものです。 8.3メジャーリリースにおける新機能については[項E.71](#)を参照してください。

E.57.1. バージョン8.3.14への移行

8.3.Xからの移行ではダンプ/リストアは不要です。 しかし8.3.8以前のバージョンからアップデートする場合は8.3.8のリリースノートを参照してください。

E.57.2. 変更点

- EXPLAINがCASE式の簡易構文を表示しようとした時の失敗を防止します。(Tom Lane)

CASEのテスト式が定数の場合、プランナはCASEを式表示用コードを混乱させる形式に単純化してしまいました。 その結果"unexpected CASE WHEN clause"というエラーになりました。

- 過去に存在した添字範囲に対する部分配列代入を修正しました。(Tom Lane)

新しく追加される添字と過去に存在した添字の先頭との間に隙間があった場合、コードは古い配列のヌルビットマップからコピーしなければならない項目数を誤計算してしまい、データ破損またはクラッシュを導く可能性がありました。

- プランナにおける、非常に時間が離れた日付値に対する想定外の変換オーバーフローを防止します。(Tom Lane)

date型はtimestamp型で表現可能な範囲よりもより広い日付範囲をサポートします。 しかしプランナは常に問題なくdateからtimestampへの変換が可能であることを仮定していました。

- standard_conforming_stringsが有効な場合のラージオブジェクト (BLOB) 用のpg_restoreのテキスト出力を修正しました。(Tom Lane)

データベースへ直接リストアすることは正しく動作しましたが、pg_restoreがSQLテキスト出力を求め、かつ、standard_conforming_stringsが元のデータベースで有効だった場合、文字列エスケープは正しくありませんでした。

- ... & !(subexpression) | ...を含むtsquery値の間違った解析を修正しました。(Tom Lane)

こうした演算子の組み合わせを含む問い合わせは正しく実行されませんでした。 同じエラーがcontrib/intarrayのquery_int型とcontrib/ltreeのltxtquery型に存在しました。

- query_int型に対するcontrib/intarray入力関数におけるバッファオーバーランを修正しました。(Apple)

この関数の返すアドレスが上書きされる可能性があるという点で、この不具合はセキュリティ問題です。 この問題を報告し修正版を提供していただいたApple Incのセキュリティチームに感謝します。(CVE-2010-4015)

- contrib/segのGiST picksplitアルゴリズムにおける不具合を修正しました。(Alexander Korotkov)

これにより、seg列上のGiSTインデックスにおいて実際に不正な結果になることはありませんが、非常に非効率的な結果になることがあります。 こうしたインデックスがある場合、この更新版をインストールした後にREINDEX処理を行うことを検討してください。(これは過去の更新のcontrib/cubeで修正された不具合と同じです。)

E.58. リリース8.3.13

リリース日: 2010-12-16

このリリースは8.3.12に対し、各種の不具合を修正したものです。 8.3メジャーリリースにおける新機能については[項E.71](#)を参照してください。

E.58.1. バージョン8.3.13への移行

8.3.Xからの移行ではダンプ/リストアは不要です。 しかし8.3.8以前のバージョンからアップデートする場合は8.3.8のリリースノートを参照してください。

E.58.2. 変更点

- Linuxにて`wal_sync_method`のデフォルトが強制的に`fdatasync`になるようにしました。(Tom Lane, Marti Raudsepp)
長い間、Linuxでのデフォルトは実質`fdatasync`でした。 しかし最近のカーネルの変更により、PostgreSQLは代わりに`open_datasync`を選択するようになりました。 この選択は性能向上をまったくもたらず、特定のシステム、特に`data=journal`マウントオプションを使用した`ext4`では、完全に失敗してしまいます。
- GINインデックスに対するWAL再生ロジックにおける分類不具合を修正しました。(Tom Lane)
これは"bad buffer id: 0"という失敗、あるいはレプリケーション中のインデックス内容の破損という結果になり得ました。
- 開始チェックポイントWALレコードが再実行時点と同じWALセグメント内に存在しない場合のベースバックアップからのリカバリを修正しました。(Jeff Davis)
- 複数のワーカが有効なまま長時間残った場合の自動バキュームの永続的な速度低下を修正しました。(Tom Lane)
多くのテーブルを処理している場合、自動バキュームワーカの`vacuum_cost_limit`の効果は、極端な速度低下をもたらすほぼゼロまで落ちることがあり得ました。
- IA64においてレジスタスタックのオーバーラン検知サポートを追加しました。(Tom Lane)
IA64アーキテクチャには2つのハードウェアスタックがあります。 スタックオーバーラン失敗を完全に防ぐには両方を検査しなければなりません。
- `copyObject()`におけるスタックオーバーフロー検査を追加しました。(Tom Lane)
非常に複雑な問い合わせが与えられた場合のスタックオーバーフローのため、特定のコードパスがクラッシュする可能性があります。
- 一時GiSTインデックスにおけるページ分割検知を修正しました。(Heikki Linnakangas)
例えば挿入が行われた時に一時インデックスをスキャンする開いているカーソルがあると、そのインデックスで"同時実行"ページ分割を持つことができます。 GiSTではこの状況の検知に失敗していました。 このためGiSTはそのカーソルの実行が続く間、間違った結果をもたらすことがありました。
- 複雑なインデックス式を`ANALYZE`する時のメモリリークを防ぎます。(Tom Lane)
- 行全体のVarを使用するインデックスがそのテーブルに依存し続けることを確実にしました。(Tom Lane)
`create index i on t (foo(t.*))`のように宣言されたインデックスはそのテーブルが削除された時に自動的に削除されませんでした。
- 複数のOUTパラメータを持つSQL関数を"インライン"化しません。(Tom Lane)
結果の想定行型に関する情報が存在しないためにクラッシュする可能性を防ぎます。
- ORDER BY、LIMIT、FOR UPDATE、WITHがINSERT ... VALUESのVALUES部に付いた場合の動作を正しくしました。(Tom Lane)

`COALESCE()` 式の定数折りたたみを修正しました。(Tom Lane)

プランナは時々、実際には決して達しない副式の評価を行い、想定外のエラーをもたらす可能性があります。

- GSSAPIサポート付きでコンパイルされたpostmasterで、接続受け入れ (`accept()` またはその後すぐになされる呼び出しのいずれか) が失敗した時のpostmasterのクラッシュを修正しました。(Alexander Chernikov)
- `log_temp_files` が有効である時の一時ファイルの削除失敗を修正しました。(Tom Lane)

ログメッセージ出力試行時にエラーが起きた場合、削除が行われませんでした。このため一時ファイルが累積してしまいました。

- `InhRelation` ノードに関する機能性出力を追加しました。(Tom Lane)

これにより `debug_print_parse` が有効な場合に特定種類の問い合わせが行われた場合の失敗を防止します。

- 点から水平線分への距離計算を正しく修正しました。(Tom Lane)

この不具合は様々な複数の幾何距離測定演算子に影響します。

- 再帰もしくはエラー修復状態にならない"単純な"式についてのPL/pgSQLの扱いを修正しました。(Tom Lane)
- PL/Pythonの集合を返す関数の扱いを修正しました。(Jan Urbanski)

集合結果を生成するイテレータ内でSPI関数呼び出しが失敗しました。

- `contrib/cube` のGiST picksplitアルゴリズムの不具合を修正しました。(Alexander Korotkov)

`cube` 列上のGiSTインデックスにおいて、実際に不正確な結果になることはありませんが、このため相当効率性が劣化する可能性があります。こうしたインデックスがある場合、この更新をインストールした後に `REINDEX` することを検討してください。

- `contrib/dblink` 内で、新しい接続を作成する時を除き、"identifier will be truncated" 注意を出力しないようにしました。(Itagaki Takahiro)
- `contrib/pgcrypto` において公開キーがない場合にコアダンプする可能性を修正しました。(Marti Raudsepp)
- `contrib/xml2` のXPath問い合わせ関数におけるメモリリークを修正しました。(Tom Lane)
- 時間帯データファイルを `tzdata release 2010o` に更新しました。フィジィ、サモアにおける夏時間規則の変更、香港の歴史的な修正が含まれています。

E.59. リリース8.3.12

リリース日: 2010-10-04

このリリースは8.3.11に対し、各種の不具合を修正したものです。 8.3メジャーリリースにおける新機能については[項E.71](#)を参照してください。

E.59.1. バージョン8.3.12への移行

8.3.Xからの移行ではダンプ/リストアは不要です。 しかし8.3.8以前のバージョンからアップデートする場合は8.3.8のリリースノートを参照してください。

E.59.2. 変更点

- PL/PerlおよびPL/Tclにおいて、呼び出し元のSQLユーザIDごとに別のインタプリタを使用します。(Tom Lane)

この変更は、同じセッション内でその後に異なるSQLユーザIDにより実行されるPerlまたはTclコード（例えばSECURITY DEFINERの内部）を破壊することで発生する可能性があるというセキュリティ問題を防止するものです。 ほとんどのスクリプト用言語は、対象とする関数で呼び出される標準関数や演算子の再定義など、これを行うことができる多くの方法を提供します。 この変更がないと、PerlまたはTcl言語の使用権限を持つ任意のSQLユーザは基本的に、対象の関数の所有者が持つSQL権限でできることを何でもできるようになります。

この変更の欠点はPerlおよびTcl関数間の内部通信がより難しくなるという点です。 逃げ道を提供するために、PL/PerlUおよびPL/TclU関数ではまだセッション当たり1つのインタプリタのみを使用します。 これらのすべての関数は従来よりデータベーススーパーユーザの信頼レベルで実行されますので、これはセキュリティ問題とはみなされません。

信頼された実行を提供することを主張するサードパーティ製の手続き言語は似たようなセキュリティ問題を持つ可能性が大きくあります。 セキュリティが非常に重要な目的のために依存している手続き言語があれば、その作者に連絡を取ることを勧めます。

この問題を指摘いただいたTim Bunceに感謝します(CVE-2010-3433)。

- `pg_get_expr()` の用途として意図されたシステムカタログ列の一つではない引数で呼び出されることを拒絶することで`pg_get_expr()` 内のクラッシュする可能性を防止します。(Heikki Linnakangas, Tom Lane)
- Windowsにおいて終了コード128 (`ERROR_WAIT_NO_CHILDREN`) を致命的でないものと扱います。(Magnus Hagander)

高負荷状況下で、Windowsプロセスはこのエラーコードで起動に失敗することが時々あります。 以前はpostmasterはパニック状態としてこれを扱い、データベース全体を再起動しました。 しかしこれは過度は反応のように思われます。

- 追加のインデックススキャンにおける厳密ではないOR結合句の不正な使用を修正しました。(Tom Lane)

これは8.3系列にはなかった8.4の修正を過去にパッチ適用したものです。 これは8.3.8で持ち込まれた、内側のリレーションが継承ツリーまたはUNION ALL副問い合わせの場合に不正な外部結合結果となるというエラーを修正するものです。

- UNION ALLのメンバーリレーションの二重スキャンの可能性を修正しました。(Tom Lane)

- "cannot handle unplanned sub-select"エラーを修正しました。(Tom Lane)

これは、副選択が他の副選択を含む式に展開される結合の別名参照を含む場合に発生しました。

- キャッシュされた計画を一時的なものと印付けする失敗を修正しました。(Tom Lane)

CREATE INDEX CONCURRENTLYが参照されるテーブルの1つに対して進行している時に計画が準備された場合、インデックスの使用準備ができた後に再度計画を作成するものと仮定されています。 これが確実に起こりませんでした。

- まれに報告されたB-tree失敗状況の一部をPANICからERRORを減らしました。 また出力されるエラーメッセージに追加の詳細を加えました。(Tom Lane)

これにより、破損したインデックスに対するシステムの堅牢性が向上したはずです。

- 自動バキューム処理中のクラッシュからshow_session_authorization()を防止します。(Tom Lane)
- 返される行のすべてが実際に同一の行型でない、レコードの集合を返す関数を防御します。(Tom Lane)
- 参照渡し関数の結果をハッシュ化する時に失敗する可能性を修正しました。(Tao Ma, Tom Lane)
- マージ結合における、結合列内のNULLの処理を改良しました。(Tom Lane)

NULLがソートで先になるようなソート順の場合、最初のNULLに達した時にマージ結合が完全に停止することができるようになりました。

- ロックファイル (postmaster.pidおよびソケット用のロックファイル) を書き出す時に、注意してその内容のfsyncします。(Tom Lane)

これがないと、postmasterの起動後しばらくしてマシンがクラッシュした場合に、ロックファイルの内容が破損することがあり得ました。これが起きると、このロックファイルを手作業で削除するまで、その後のpostmasterの起動試行が成功できなくなります。

- 過度に入れ子になった副トランザクションに対してXIDを割り当てる際の再帰を防止します。(Andres Freund, Robert Haas)

元々のコードでは、制限されたスタック領域であった場合にクラッシュすることがあり得ました。

- walwriterプロセス内で古いWALセグメントを開いたまま保持しないようにしました。(Magnus Hagander, Heikki Linnakangas)

これまでのコードでは、不要となったセグメントの削除を防止しました。

- バックエンド起動の初期にゴミを生成することがあったlog_line_prefixの%iエスケープを修正しました。(Tom Lane)
- アーカイブ処理が有効な場合、ALTER TABLE ... SET TABLESPACEにおけるデータ破損の可能性を修正しました。(Jeff Davis)

- CREATE DATABASEとALTER DATABASE ... SET TABLESPACEを問い合わせキャンセルにより中断させることができます。(Guillaume Lelarge)

- 演算子クラスと演算子族を扱うREASSIGN OWNEDを修正しました。(Asko Tiidumaa)

- 2つの空のtsquery値を比較する際にコアダンプする可能性を修正しました。(Tom Lane)

- %に続く_を持つパターンに対するLIKEの処理を修正しました。(Tom Lane)

以前にもこれを修正しましたが、正しく処理されない場所が一部残っていました。

- PL/Pythonにおいて、PyCObject_AsVoidPtrおよびPyCObject_FromVoidPtrからのヌルポインタ結果を防止します。(Peter Eisentraut)

- 自動コミットモードが無効な場合、DISCARD ALLをトランザクションブロック内に包むべきものではないコマンドとしてpsqlが認識するようにしました。(Itagaki Takahiro)

- RETURNING句からのデータを正しく処理するようにecpgを修正しました。(Michael Meskes)

- 削除された列を含むテーブルに対するcontrib/dblinkの処理を改良しました。(Tom Lane)

- contrib/dblinkにおいて"duplicate connection name"エラー後の接続リークを修正しました。(Itagaki Takahiro)

- 62バイトより長い接続名に関するcontrib/dblinkの処理を正しく修正しました。(Itagaki Takahiro)

- hstore(text, text) 関数をcontrib/hstoreに追加しました。(Robert Haas)

この関数は、廃止予定となった=>演算子に対する推奨の代替です。 将来を見据えたコードが古めのバージョンのサーバでも使用できるようにするために過去にもパッチが適用されました。 このパッチは、contrib/hstoreのインストール後もしくは特定のデータベースに再インストールされた後にのみ効果を持つことに注意してください。 代わりに手作業でCREATE FUNCTIONコマンドを実行することを選ぶユーザがいるかもしれません。

- 構築手順や文書を更新して、ソースコードリポジトリがCVSからGitに変更されたことを反映しました。(Magnus Hagander他)

- 時間帯データファイルをtzdata release 2010Iに更新しました。 エジプト、パレスチナにおける夏時間規則の変更、フィンランドの歴史的な修正が含まれています。

またこの変更では、2つのミクロネシアの時間帯を追加しています。 Pacific/ChuukはPacific/Trukより好まれるようになり（好まれる省略形はTRUTではなくCHUT）、Pacific/PohnpeiはPacific/Ponapeより好まれるようになります。

- Windowsの"N. Central Asia Standard Time"時間帯をAsia/AlmatyではなくAsia/Novosibirskに対応付けるようにしました。

MicrosoftはKB976098の時間帯更新からこの時間帯の夏時間の振舞いを変更しました。 Asia/Novosibirskはこの新しい振舞いにより合致しています。

[前のページ](#)

リリース8.3.13

[ホーム](#)
[上に戻る](#)

[次のページ](#)

リリース8.3.11

E.60. リリース8.3.11

リリース日: 2010-05-17

このリリースは8.3.10に対し、各種の不具合を修正したものです。 8.3メジャーリリースにおける新機能については[項E.71](#)を参照してください。

E.60.1. バージョン8.3.11への移行

8.3.Xからの移行ではダンプ/リストアは不要です。 しかし8.3.8以前のバージョンからアップデートする場合は8.3.8のリリースノートを参照してください。

E.60.2. 変更点

- Safe.pmを使用するのではなく、インタプリタ全体に適用されるopmaskを使用してplperlを強制的に制限します。(Tim Bunce, Andrew Dunstan)

最近の開発によりplperlを信頼できるとする根拠とするにはSafe.pmが非常に安全ではないことを知らされました。 この変更により、Safe.pmの使用がなくなり、常に適用されるopcodeマスクを持つ別のインタプリタを使用するようになります。 この変更の喜ばしい副作用として、plperl内で普通の方法でPerlのstrictプラグマが使用できるようになりました。 また、Perlの\$aと\$bがソート関数で想定される動作を行うようになり、関数のコンパイルが非常に高速になりました。(CVE-2010-1169)

- PL/Tclにおいてpltcl_modules経由で信頼できないコードが実行されることを防止しました。(Tom)

PL/TclのデータベーステーブルからTclコードを自動読み込みする機能はトロイの木馬攻撃によりつけこまれる可能性があります。 そのテーブルの作成や挿入できるユーザに対する制限がなかったためです。 この変更により、pltcl_modulesがスーパーユーザにより所有されていない限り、この機能は無効になります。(しかし、このテーブルに対する権限は検査されませんので、安全性が低いモジュールを本当に必要とするインストレーションでは、スーパーユーザ以外に適切な権限を付与することができます。) また、 pltclu関数を本当に実行しようとしていない限り、制限がない"通常の"Tclにコードが読み込まれることを防止しました。(CVE-2010-1170)

- relcache項目の再構築中にキャッシュ再設定メッセージを受け取った場合にクラッシュする可能性を修正しました。(Heikki)

8.3.10で関連する失敗を修正する時に、このエラーが入ってしまいました。

- 関数の言語検証関数を実行する時に関数毎のGUC設定を適用します。(Itagaki Takahiro)

これは、その設定がないと関数のコードが無効になる場合の失敗を防ぎます。 例えばsearch_pathが正しくないとSQL関数は解析できません。

- 非特権ユーザがスーパーユーザのみのパラメータ設定を再設定することを許しません。(Alvaro)

これまで、非特権ユーザが自身に対するALTER USER ... RESET ALLまたは自身が所有するデータベースに対するALTER DATABASE ... RESET ALLを行った場合、そのユーザまたはデータベースに対する特別なパラメータ設定は、たとえスーパーユーザのみにより変更可能なものとして提供されるものであっても、すべて削除されました。 ALTERはそのユーザが変更権限を持つパラメータのみを削除するようになりました。

- CONTEXT追加がログ項目になされる時にシャットダウンが発生した場合に、バックエンドのシャットダウン時にクラッシュする可能性を防ぎました。(Tom)

現在のトランザクションがログメッセージを出力する時点に達した時にすでにロールバックされているために文脈出力関数が失敗することがありました。

- アーカイブ処理ができる限り高速にarchive_commandの変更に対応することを確実にしました。(Tom)
- pl/perlのppport.hを最近のPerlバージョンに更新しました。(Andrew)
- pl/python内の雑多なメモリリークを修正しました。(Andreas Freund, Tom)
- 変数展開が自身を参照している際のpsqlにおける無限再帰を防止します。(Tom)

- `¥copy (select ...)` 内のドットの前後に空白を追加しないように、psqlの`¥copy`を修正しました。(Tom)

数値リテラルにおいて10進数の小数点の前後に空白が追加された場合、構文エラーが発生します。

- `contrib/intarray` 演算子を使用する満足されない問い合わせにおいて、不要な"GIN indexes do not support whole-index scans"エラーを修正しました。(Tom)
- `contrib/pgstattuple` 関数がキャンセル割り込みに即座に応答することを確実にしました。(Tatsuhito Kasahara)
- サーバ起動時に、既存の共有メモリセグメントに対して `shmget()` が `EINVAL` を返す状況を適切に扱うようにしました。(Tom)

この振舞いはOS Xを含むBSD派生のカーネルにおいて観測されました。共有メモリ要求サイズが大きすぎるという警告の、完全に間違った起動失敗という結果になりました。

- Windowsにおけるsysloggerプロセス内のクラッシュする可能性を防ぎます。(Heikki)
- Windowsレジストリにおいて不完全な時間帯情報をより堅牢に扱います。(Magnus)
- Windowsにおいて既知の時間帯名称の集合を更新しました。(Magnus)
- 時間帯データファイルをtzdata release 2010jに更新しました。アルゼンチン、南極におけるオーストラリア、バンラデシュ、メキシコ、モロッコ、パキスタン、パレスチナ、ロシア、シリア、チュニジアにおける夏時間規則の変更、および、台湾の歴史的な修正が含まれます。

また、デフォルトの時間帯省略形の集合に `PKST` (パキスタン夏時間) を追加しました。

[前のページ](#)

リリース8.3.12

[ホーム](#)
[上に戻る](#)

[次のページ](#)

リリース8.3.10

E.61. リリース8.3.10

リリース日: 2010-03-15

このリリースは8.3.9に対し、各種の不具合を修正したものです。8.3メジャーリリースにおける新機能については[項E.71](#)を参照してください。

E.61.1. バージョン8.3.10への移行

8.3.Xからの移行ではダンプ/リストアは不要です。しかし8.3.8以前のバージョンからアップデートする場合は8.3.8のリリースノートを参照してください。

E.61.2. 変更点

- SSL接続においてセッションキーの再調停をどの程度の頻度で行うかを制御する`ssl_renegotiation_limit`設定パラメータを新しく追加しました。(Magnus)

これをゼロに設定することで、再調停を完全に無効にすることができます。これは壊れたSSLライブラリを使用している場合に必要になる可能性があります。具体的には一部のベンダは再調停の試行を失敗させるCVE-2009-3555に対する一時しのぎのパッチを提供しています。

- バックエンド起動時のデッドロックの可能性を修正しました。(Tom)
- `relcache`の再読み込みがきれいに行われている間にエラーを扱わないことが原因でクラッシュする可能性を修正しました。(Tom)
- 保管された計画への宙ぶらりんのポインタを使用することが原因でクラッシュする可能性を修正しました。(Tatsuo)
- 副トランザクション起動時における失敗からのリカバリを試行する時にクラッシュする可能性を修正しました。(Tom)
- セーブポイントの使用およびサーバの符号化方式と異なるクライアント符号化方式に関連したサーバのメモリリークを修正しました。(Tom)
- GiSTインデックスページ分割のリカバリ終了時の整理において生成される不正なWALデータを修正しました。(Yoichi Hirai)

これはインデックスの破損という結果となり、WAL再生中にエラーが発生します。もし運が悪ければ、不完全のGiST挿入に成功した後にリカバリ終了時の整理が行われている時にクラッシュしてしまいます。

- `bit`型の`substring()`では、いかなる負の長さも"文字列の残りすべて"を意味するようにしました。(Tom)

これまでのコードでは-1のみを上のように扱い、他の負の値についてはクラッシュする可能性がある無効な結果値を生成しました。(CVE-2010-0442)

- 出力ビット幅が指定された整数よりも広く、それが8ビットの倍数以外である場合、先頭の端数のバイトを正確に扱うように整数からビット列への変換を修正しました。(Tom)
- 正規表現マッチが異常なまでに低速になる一部の状況を修正しました。(Tom)
- ずさんなメモリ管理によって引き起こる`xml`の雑多なクラッシュを修正しました。(Tom)

これは先に8.4に適用された変更の後戻りのパッチです。8.3のコードは不具合が多いことが知られていましたが、何らかの実地試験を受けるまで後戻りのパッチをしようとは思わないほど新しいコードはかなり変更されていました。

- 複合型の配列の列の要素フィールドを更新する場合の不具合を修正しました。(Tom)
- 終了位置が正確にセグメント境界であった場合、次のWALセグメント名を報告するようにバックアップ履歴ファイル内の`STOP WAL LOCATION`を修正しました。(Itagaki Takahiro)
- 一時ファイルが漏洩する一部の状況を修正しました。(Heikki)

これは前のマイナーリリースでもたらされた問題を修正します。失敗する状況の1つは集合を返す`plpgsql`関数が他の関数の例外ハンドラ内で呼び出された場合です。

- 論理値変数の場合の制約による排他処理を改良しました。具体的には、"bool_column = false"制約を持つパーティションを除外することができるようになりました。(Tom)
- pg_hba.confとその関連ファイルを読み取る時、@が引用符記号の内部にある場合に@somethingをファイル包含要求と扱わないようにしました。また、@自身をファイル含有要求として扱いません。(Tom)

ロールまたはデータベース名が@から始まる場合の一定しない動作を防止します。空白文字を含むパス名のファイルを含む必要がある場合、まだ実現させることはできますが、式全体を引用符でくくるのではなく@"path to/file"と記述しなければなりません。

- ディレクトリがpg_hba.confとその関連ファイル内で含有対象と指定されていた場合の一部のプラットフォームにおける無限ループを防止します。(Tom)
- SSL_readまたはSSL_writeがerrnoを設定することなく失敗した場合に無限ループする可能性を修正しました。(Tom)
一部の報告ではopensslの一部のWindows版で発生する可能性があります。
- 正しく動作させるためにホスト名が必要ですので、ローカル接続ではGSSAPI認証を許しません。(Magnus)
- 接続が消失した場合ecpgに適切なSQLSTATEを報告させるようにしました。(Michael)
- latexおよびtroff出力書式では行うべきではない文字列の書式付けを行わないようにpsqlのnumericlocaleオプションを修正しました。(Heikki)
- ON_ERROR_STOPと--single-transactionの両方が指定され、かつ、暗黙的なCOMMIT時にエラーが起きた場合に、psqlが正しい終了コード(3)を返すようにしました。(Bruce)
- 複合列がNULLに設定される場合にplpgsqlが失敗することを修正しました。(Tom)
- PL/PerlUからのPL/Perl関数の呼び出し、およびその逆の場合において失敗する可能性を修正しました。(Tim Bunce)
- コンパイラ固有の誤動作が起こる可能性を防ぐためにPL/Python内にvolatile記号を追加しました。(Zdenek Kotala)
- PL/Tclは確実にTclインタプリタを完全に初期化します。(Tom)

このミスの既知の兆候は、Tcl 8.5以降を使用している場合のTclのclockコマンドの誤動作だけです。

- dblink_build_sql_*関数であまりに多くのキー列が指定された場合のcontrib/dblinkのクラッシュを防止します。(Rushabh Lathia, Joe Conway)
- contrib/ltree操作においてゼロ次元の配列を許可します。(Tom)

この状況は以前はエラーとして拒絶されました。しかしこれを要素ゼロの配列と同様に扱う方がより簡便です。特にこれは、ltree演算子がARRAY(SELECT ...)の結果に適用され、副問い合わせが行を返さない場合の不要な失敗を防止します。

- ずさんなメモリ管理によって引き起こるcontrib/xml2の雑多なクラッシュを修正しました。(Tom)
- Windowsにおいてcontrib/xml2の構築をより堅牢にしました。(Andrew)
- Widowsのシグナル処理における競合状態を修正しました。(Radu Ilie)

この不具合の兆候として既知のものは、負荷が高い状況においてpg_listener内の行が削除されることです。

- 時間帯データファイルをtzdata release 2010eに更新しました。バングラデシュ、チリ、フィジー、メキシコ、パラグアイ、サモアにおける夏時間規則の変更が含まれます。

E.62. リリース8.3.9

リリース日: 2009-12-14

このリリースは8.3.8に対し、各種の不具合を修正したものです。8.3メジャーリリースにおける新機能については[項E.71](#)を参照してください。

E.62.1. バージョン8.3.9への移行

8.3.Xからの移行ではダンプ/リストアは不要です。しかし8.3.8以前のバージョンからアップデートする場合は8.3.8のリリースノートを参照してください。

E.62.2. 変更点

- セッション固有の状態を変更するインデックス関数により引き起こされる間接的なセキュリティ脅威から保護します。(Gurjeet Singh, Tom)

この変更により不変インデックス関数がスーパーユーザのセッションを破壊できることを防ぎます。(CVE-2009-4136)

- ヌルバイトが埋め込まれたコモンネーム (CN) フィールドをもつSSL証明書を拒絶します。(Magnus)

これは、SSL検証においてサーバまたはクライアント名に対する証明書が不正に一致しないようにします。(CVE-2009-4034)

- バックエンド起動時のキャッシュ初期化の間でクラッシュする可能性を修正しました。(Tom)
- 空のシソーラス辞書におけるクラッシュを防止しました。(Tom)
- 安全ではない時点でのVACUUMを中断するシグナルを防止します。(Alvaro)

この修正により、VACUUM FULLがすでにタブルの移動をコミットした後にキャンセルされた場合のパニックや、通常のVACUUMがテーブルを切り詰めた後に割り込みを受けた場合の一時的なエラーを防止します。

- ハッシュテーブルサイズの計算における整数オーバーフローによるクラッシュの可能性を修正しました。(Tom)

これは、ハッシュ結合の結果のプランナの推定サイズが極端に大きい場合に発生することがありました。

- inet/cidrの比較における非常にまれなクラッシュを修正しました。(Chris Mikkelsen)
- 準備されたトランザクションで保持されるタブルレベルの共有ロックを確実に無視しないようにしました。(Heikki)
- 副トランザクション内でアクセスされるカーソルで使われる一時ファイルの早すぎる削除を修正しました。(Heikki)
- 新しいCSVログファイルに変更する時のsysloggerプロセスのメモリリークを修正しました。(Tom)
- Windowsにおける権限ダウングレードロジックを修正しました。(Jesse Morris)

これは、Windowsにおいて"could not locate matching postgres executable"といった誤解を招くエラーメッセージでよくデータベースの起動が失敗する状況の一部を修正します。

- GiSTインデックスページ分割において、分割がインデックスの最初の列以外に依存する場合の不正なロジックを修正しました。(Paul Ramsey)
- チェックポイントの最後で古いWALファイルの回収や削除が失敗した場合にエラーを出力しません。(Heikki)

致命的ではないものと扱い、チェックポイントを完了させることができるようにする方が優れています。将来のチェックポイントでは削除を再試行します。こうした問題は通常の操作では想定されませんが、うまく設計されていないWindowsのアンチウィルスソフトウェアやバックアップソフトウェアにより現れることがあります。

- WindowsにおいてWALファイルが繰り返しアーカイブされないようにしました。(Heikki)

別の症状として、一部の他のプロセスが不要となったファイルの削除に干渉する場合にも発生します。

- PAMパスワード処理をより堅牢に修正しました。(Tom)

以前のコードでは、Linuxの`pam_krb5` PAMモジュールとドメインコントローラとしてのMicrosoft Active Directoryを組み合わせると失敗することが知られていました。PAMスタックに渡される引数に関する正当ではない仮定を作成することであるため、どこかで問題になるかもしれません。

- GSSAPIおよびSSPI認証方式における認証トークン（Kerberosチケット）の最大サイズを上げました。（Ian Turner）

以前の2000バイトという制限はUnix Kerberosの実装では十分でしたが、Windowsドメインコントローラで発行されるチケットではより大きくなる可能性があります。

- シーケンスのアクセス統計情報の収集を再度有効にしました。（Akira Kurosawa）

これは動作していましたが、8.3では壊れていました。

- `CREATE OR REPLACE FUNCTION`における所有権限の依存関係の処理を修正しました。（Tom）

- `WHERE x=x`条件の不正な処理を修正しました。（Tom）

一部の場合、これらは冗長なものとして無視されることがありましたが、そうではありません。 — これらは`x IS NOT NULL`と等価です。

- 全文検索パーサがXML属性内のアンダースコアを受け付けるようにしました。（Peter）

- `xml`バイナリ入力における符号化方式の扱いを修正しました。（Heikki）

XMLヘッダにて符号化方式の指定がない場合、デフォルトでUTF-8を仮定するようになります。以前の扱いには一貫性がありませんでした。

- `plperl`から`plperl1`の呼び出し、またはその逆における不具合を修正しました。（Tom）

内部の関数からのエラー終了が、外側の関数用のPerlインタプリタを正しく再選択できずにクラッシュをもたらす可能性があります。

- PL/Perl関数が再定義された時のセッション有効期間のメモリリークを修正しました。（Tom）

- 集合を返すPL/Perl関数により返される場合Perl配列が適切にPostgreSQL配列に返還されることを確実にしました。（Andrew Dunstan, Abhijit Menon-Sen）

集合以外を返す関数ではすでに正しく動作しました。

- PL/Pythonの例外処理におけるまれなクラッシュを修正しました。（Peter）

- `contrib/pg_standby`にて、Windowsのシグナルを使用したフェイルオーバーの通知を無効にしました。（Fujii Masao）

WindowsではUnixのようなシグナルを持ちませんので、これはまったく役に立っていませんでした。さらに最近の変更により実際にクラッシュしてしまいます。

- `psql`の`flex`モジュールが正しいシステムヘッダファイルを使用してコンパイルされることを確実にしました。（Tom）

これは、`--enable-largefile`が生成されたコード内に非互換の変更を引き起こすプラットフォームにおける構築失敗を修正します。

- 今後のバージョンの`libpq`との互換性を高めるために、`postmaster`が接続要求パケットにおける`application_name`パラメータをすべて無視するようにしました。（Tom）

- 最新の現実合うように時間帯省略形ファイルを更新しました。（Joachim Wieland）

デフォルト時間帯群に`IDT`の追加が含まれます。

- 時間帯データファイルを`tzdata release 2009s`まで更新しました。南極、アルゼンチン、バングラデシュ、フィジー、ノヴォクズネツク、パキスタン、パレスチナ、サモア、シリアにおける夏時間規則の変更、および香港に関する歴史的な修正が含まれます。

E.63. リリース8.3.8

リリース日: 2009-09-09

このリリースは8.3.7に対し、各種の不具合を修正したものです。8.3メジャーリリースにおける新機能については[項E.71](#)を参照してください。

E.63.1. バージョン8.3.8への移行

8.3.Xからの移行ではダンプ/リストアは不要です。しかし、`interval`列に対するハッシュインデックスを持つ場合は8.3.8にアップデートした後にそれらを`REINDEX`しなければなりません。また、8.3.5より前のバージョンからアップグレードする場合は、8.3.5のリリースノートを参照してください。

E.63.2. 変更点

- Windowsの共有メモリ割り当てコードを修正しました。(Tutomu Yamada, Magnus)

この不具合がよく報告される"could not reattach to shared memory (共有メモリに再接続できませんでした)"というエラーメッセージを導いていました。

- `pg_start_backup()` 実行中にWALセグメントスイッチを強制します。(Heikki)

これによりベースバックアップが使用不可となる可能性がある境界状態を防止します。

- `security-definer`関数内部の`RESET ROLE`および`RESET SESSION AUTHORIZATION`を不許可としました。(Tom, Heikki)

これは、`security-definer`関数内部における`SET ROLE`と`SET SESSION AUTHORIZATION`を不許可にする過去のパッチ (CVE-2007-6600参照) で抜けていたものを補うものです。

- 既にロードされたロード可能モジュールに対する`LOAD`を何も行わないようにしました。(Tom)

これまでは、`LOAD`はそのモジュールをアンロードし、再ロードしようとしていました。ですが、これは安全ではなく、かつ、まったく意味がないものでした。

- LDAP認証における空のパスワードを許可しません。(Magnus)

- 外側の集約関数の引数に現れる副問い合わせの扱いを修正しました。(Tom)

- ソートまたはマテリアライズ計画ノードの出力から行全体の値の抽出する処理に関連した不具合を修正しました。(Tom)

- `synchronize_seqscans`がスクロール可能な`WITH HOLD`カーソルの結果を変更することを防止します。(Tom)

- 100句を超えるANDまたはORリストが存在した場合に部分インデックスと制約による除外の最適化を無効にするプランナの変更を元に戻しました。(Tom)

- `interval`データ型のハッシュ計算を修正しました。(Tom)

これは、時間間隔値に対するハッシュ結合が間違った結果を生成しないように修正します。また、時間間隔型の列に対するハッシュインデックスの内容も変更します。もしこうしたインデックスがあれば、アップデートの後にそれらを`REINDEX`する必要があります。

- `to_char(..., 'TH')`を'`HH`'/'`HH12`'の大文字の序数表現と扱います。(Heikki)

以前は'`th`' (小文字) として扱われました。

- `INTERVAL 'x ms'`において`x`が200万を超え、かつ、整数型の日付時刻が使用された場合のオーバーフローを修正しました。(Alex Hunsaker)

- 点と線分間の距離計算を修正しました。(Tom)

これにより幾何演算子の一部が間違った結果を返していました。

- 通貨が端数桁を持たないロケール、たとえば日本、において`money`データ型が動作するように修正しました。(Itagaki Takahiro)

E.64. リリース8.3.7

リリース日: 2009-03-16

このリリースは8.3.6に対し、各種の不具合を修正したものです。8.3メジャーリリースにおける新機能については[項E.71](#)を参照してください。

E.64.1. バージョン8.3.7への移行

8.3.Xからの移行ではダンプ/リストアは不要です。しかし、8.3.5より前のバージョンからアップグレードする場合は、8.3.5のリリースノートを参照してください。

E.64.2. 変更点

- 符号化方式の変換に失敗した時、再帰的なエラーによるクラッシュを防止しました。(Tom)

この変更は、最近の2つのマイナーリリースにて関連する失敗の状況に対してなされた改修を拡張したものです。前回の修正は元の問題報告に特化したものでしたが、符号化方式変換関数で発生するすべてのエラーがそのエラーを報告しようとして、無限に再帰される可能性があることが分かりました。したがって、再帰的なエラー報告を行う状況になったことがわかった時の解決策は、変換と符号化方式を無効にし、通常のASCII形式のエラーメッセージで報告することです。(CVE-2009-0922)

- 特定の変換関数に対する、間違っ符号化方式を用いたCREATE CONVERSIONを許可しません。(Heikki)

これにより、符号化方式に関する失敗における、あり得る状況を防止します。前回の変更は、同じ問題における別の種類の失敗に対する防止策でした。

- 不要なパス表現の変更を行わないようにxpath()を修正しました。また、必要な時により正しく試行するように修正しました。(Andrew)

標準SQLでは、xpathは文書断片となるデータに対して動作しなければならないと提言していますが、libxmlはこれをサポートしておらず、実際のところXpath標準に従って動作しているかどうか不明です。xpathは、データとパス表現の両方を変更することでこの不適合を回避しようとしていました。しかしこの変更の不具合があり、有効な検索が失敗することが発生する可能性があります。xpathは、整形式のXML文書かどうか検査し、そして、整形式の場合にデータやパス表現を変更せずにlibxmlを呼び出すようにしました。さもなくば、失敗する可能性を少なくした別の変更方法が使用されます。

注意: 新しい変更方法はまだ100%満足するものではありません。また、完全な解決はできないように考えられます。したがってこのパッチは、既存のアプリケーションが不必要に障害を起こさないようにするための応急処置とみなさなければなりません。PostgreSQL 8.4では単純に、整形式の文書ではないデータに対するxpathの利用を拒絶するようになるはずです。

- to_char()に引数のデータに対して不適切な整形用のコードが渡された時のコアダンプを修正しました。(Tom)
- Cロケールがマルチバイト符号化方式で使用された場合の全文検索が失敗する可能性を修正しました。(Teodor)

wchar_tがintよりもビット数が小さい時にクラッシュする可能性があります。具体的にはWindowsです。

- Eメールのような複数の@文字を含む文字列に対する、非常に非効率的な全文検索のパースの扱いを修正しました。(Heikki)
- 大規模な副問い合わせの出力リストにおける副SELECTに関するプランナの問題を修正しました。(Tom)

このバグに関する既知の兆候は、含まれるデータ型に依存して発生する"failed to locate grouping columns"というエラーです。しかし他の問題も発生していたかもしれません。

- 暗黙的な強制を使用したCASE WHENの逆コンパイルを修正しました。(Tom)

この間違いにより、アサートを有効にして構築した場合にアサート失敗が発生する可能性があります。また、他の構築状況でもビューの検証やダンプを行う際に"unexpected CASE WHEN clause"というエラーメッセージが発生する可能性があります。

- TOASTテーブルの行型に対する所有者を間違っ割り当てる可能性を修正しました。(Tom)

`CLUSTER`または`ALTER TABLE`の書き換え構文がテーブル所有者以外のユーザにより実行された場合、テーブルのTOASTテーブル向けの`pg_type`項目が実行したユーザが所有するものとして記録されてしまいました。TOASTの行型に対する権限は通常のデータベース操作ではまったく検証されませんので、これによりすぐに問題が発生することはありません。しかし、後でコマンドを発行したロールを削除しようとした場合に想定外の失敗（8.1または8.2）や削除後に`pg_dump`が"owner of data type appears to be invalid"という警告が発生する（8.3）可能性がありました。

- 現在のセッションがまったく`LISTEN`コマンドを実行していない場合に`UNLISTEN`がすぐに終了するように変更しました。（Tom）

ほとんどの状況では、これは特に有用な最適化ではありません。しかし、`DISCARD ALL`は`UNLISTEN`を呼び出しますので、これまでのコードでは、`DISCARD ALL`を非常に多く使用するアプリケーションでは、顕著な性能問題が発生していました。

- PL/pgSQLが、`INSERT`後の`INTO`を文字列の起点だけではなく、任意の位置における`INTO`変数句として扱わないように修正しました。具体的には、`CREATE RULE`内の`INSERT INTO`にて失敗しないようにしました。（Tom）
- ブロックの終了時に、PL/pgSQLのエラー状態変数を完全に消去します。（Ashesh Vashi、Dave Page）

これはPL/pgSQL自体の問題ではありませんが、これを行わないと、PL/pgSQLデバグが関数の状態を検証する際にクラッシュする可能性がありました。

- Windowsにおいて`CallNamedPipe()`呼び出しが失敗した時に再試行します。（Steve Marshall、Magnus）

この関数は一時的に失敗することが時々あるようです。これまでは重大なエラーとしてすべての失敗を扱っていましたが、`LISTEN`、`NOTIFY`やその他の操作が混乱する可能性がありました。

- デフォルトの既知の時間帯省略形のリストに`MUST`（モーリシャス島夏時間）を追加しました。（Xavier Bugaud）

[前のページ](#)

リリース8.3.8

[ホーム](#)
[上に戻る](#)

[次のページ](#)

リリース8.3.6

E.65. リリース8.3.6

リリース日: 2009-02-02

このリリースは8.3.5に対し、各種の不具合を修正したものです。8.3メジャーリリースにおける新機能については[項E.71](#)を参照してください。

E.65.1. バージョン8.3.6への移行

8.3.Xからの移行ではダンプ/リストアは不要です。しかし、8.3.5より前のバージョンからアップグレードする場合は、8.3.5のリリースノートを参照してください。

E.65.2. 変更点

- `DISCARD ALL`を従来行っていたすべてに追加して勧告的ロックを解放するようにしました。(Tom)
これはもっとも適切な動作になるように決定されました。しかし、既存のアプリケーションに影響を与える可能性があります。
- GiSTインデックス全スキャンが正常に動作するよう修正しました。(Teodor)
このエラーにより、GiSTインデックス上でクラスタ化されたテーブルでは、行の損失が発生する可能性があります。
- `xmlconcat (NULL)`のクラッシュを修正しました。(Peter)
- ビットとして高位ビットをセットする文字が使用された場合、`ispell`辞書がクラッシュする可能性を修正しました(Teodor)。
よく使用されるあるノルウェー辞書で発生することが分かっています。他でも同じ状況が存在するかもしれません。
- 複合型における`pg_dump`出力の誤った順序を修正しました。(Tom)
もっともよくある問題は、ユーザ定義の演算子クラスが、それらを必要とするインデックスやビューの後にダンプされるというものです。
- `headline()`関数におけるURLの扱いを改良しました。(Teodor)
- `headline()`関数における延長見出しの扱いを改良しました。(Teodor)
- 符号化方式の変換が符号化方式の組み合わせの指定を間違えた変換関数で作成された場合にあり得る、アサートの失敗や誤変換を防止します。(Tom, Heikki)
- 例えば`INSERT`が`UPDATE`に書き換えられるといった、PL/pgSQLで実行された文が他の種類の文に書き換えられた場合にあり得るアサートの失敗を修正しました。(Heikki)
- データ型入力関数でスナップショットが利用できることを確実にしました。(Tom)
これは主に、ユーザ定義の安定または不変関数を含む`CHECK`制約付きで宣言されたドメインに影響します。こうした関数は通常、スナップショットが設定されていない場合失敗します。
- データ型入出力、特に、ドメインの検査制約で使用するSPIを使用する関数をより安全にしました。(Tom)
- `VACUUM`において小規模なテーブルに対する不要なロックを防止しました。(Heikki)
- `ALTER TABLE ENABLE/DISABLE RULE`が活動中のセッションで認識され続けることがあるという問題を修正しました。(Tom)
- `UPDATE RETURNING tableoid`が現在のOIDではなくゼロをかえす問題を修正しました。(Tom)
- `ANYARRAY`を取るものと宣言された関数が、その型の`pg_statistic`列で動作できるようにしました。
これはこれまで動作していましたが、残念ながら8.3で壊れました。
- 推移的な等価性が外部結合句に適用される場合のプランナの選択度の誤推定を修正しました。(Tom)

これにより... `from a left join b on a.a1 = b.b1 where a.a1 = 42` ...のような問い合わせで劣化した計画が作成されました。

- オプティマイザの長大なINリストの扱いを改良しました。

この変更により、制約による除外が有効な場合にこうしたリストが、長く時間を浪費することを防止します。

- GINインデックス構築時の同期スキャンを防止します。(Tom)

GINはTID順序を増やすことでタプルの挿入に最適化されていますので、同期スキャンを使用することを選択すると、3倍以上構築が低速になる可能性があります。

- 保持可能なカーソルの内容がTOASTテーブルの内容に依存しないことを確実にしました。(Tom)

これまでカーソル内の大規模なフィールドはTOASTポインタとして表現される可能性があります。これは、参照先のテーブルがカーソルを読み取る前に削除された場合や、大規模な値が削除、そしてバキュームされてしまった場合に失敗しました。これは通常のカーソルでは発生することはあり得ませんが、トランザクションの生成以降保持されるカーソルではあり得ます。

- 集合を返す関数が結果全体を読み取らずに終了した時のメモリリークを修正しました。(Tom)
- データベース符号化方式がUTF-8でない場合に、XML関数における符号化方式変換の問題を修正しました。(Tom)

- `contrib/dblink`の`dblink_get_result(text,bool)`関数を修正しました。(Joe)

- `contrib/sslinfo`関数から不要な出力が行われる可能性を修正しました。(Tom)

- あるコマンドで一度に複数のトリガが発生した時の`contrib/tsearch2`互換性トリガの不正な動作を修正しました。(Teodor)

- 自動バキュームにおける間違ったシグナルが発生する可能性を修正しました。(Heikki)

- Windows 7 betaにおけるサービスとしての実行をサポートします。(Dave and Magnus)

- `ecpg`の可変長文字構造体の扱いを修正しました。(Michael)

- `configure`スクリプトがPL/Perl用のリンク情報の入手ができなかった場合に失敗を適切に報告するように修正しました。(Andrew)

- すべての文書において、現在活動していない`pgsql-ports`と`pgsql-patches`メーリングリストの代わりに、`pgsql-bugs`または`pgsql-hackers`を参照するようにしました。

- 時間帯データファイルを`tzdata`リリース2009a（カトマンズ、およびスイス、キューバにおける歴史的なDSTの訂正）に更新しました。

E.66. リリース8.3.5

リリース日: 2008-11-03

このリリースは8.3.4に対し、各種の不具合を修正したものです。8.3メジャーリリースにおける新機能については[項E.71](#)を参照してください。

E.66.1. バージョン8.3.5への移行

8.3.Xからの移行ではダンプ/リストアは不要です。しかし、8.3.1より前のバージョンからアップグレードする場合は、8.3.1のリリースノートを参照してください。またこれまでの8.3.Xリリースを実行している場合は、アップグレード後にすべてのGiSTインデックスを`REINDEX`することを勧めます。

E.66.2. 変更点

- 削除後に間違ったインデックス項目を"dead"と印付けたことによるGiSTインデックスの破損を修正しました。(Teodor)
これによりインデックス検索は存在するはずの行の検索に失敗します。破損したインデックスは`REINDEX`を使用して修正することができます。
- クライアントの符号化方式が翻訳されたエラーメッセージを表すことができない時のバックエンドのクラッシュを修正しました。(Tom)
以前にも同様の問題に対応しましたが、"等価な文字を持たない"メッセージ自体が変換されない場合にまだ失敗しました。今回の修正では、こうした状況を検知した時、翻訳を無効にし、通常のASCIIエラーメッセージを送信します。
- `bytea`からXML対応付けにおけるクラッシュの可能性を修正しました。(Michael McMaster)
- トリガから深い入れ子状態の関数が呼び出された時にクラッシュする可能性を修正しました。(Tom)
- `expression IN (expression-list)`問い合わせの最適化を改良しました。(Tom. Robert Haasの発案に基づいた。)
8.2.xおよび8.3.xでは、右辺に問い合わせ変数が存在する状況を取扱う効率が過去のバージョンより落ちていました。今回の修正では、こうした状況における振舞いを8.1に戻しました。
- 副`SELECT`が`FROM`、複数行の`VALUES`リスト、`RETURNING`リスト内の関数呼び出し内に存在する場合の、間違ったルール問い合わせの展開を修正しました。(Tom)
この問題のよくある徴候は、"unrecognized node type"というエラーです。
- GiSTインデックスの`IS NULL`検索の再スキャン中のアサート失敗を修正しました。(Teodor)
- ハッシュされた集約計画の再スキャン中のメモリリークを修正しました。(Neil)
- 新しく定義されたPL/pgSQLトリガ関数が通常の関数として呼び出された場合に報告されるエラーを確実なものにしました。(Tom)
- `CREATE DATABASE`がファイルコピーを始める前に強制的にチェックポイントを行います。(Heikki)
これにより、元のデータベースにおいてファイルが最近削除された場合に失敗する可能性を防止します。
- `ALTER SET TABLESPACE`を使ってテーブルを別のテーブル空間に移動する時に`relfilenode`番号が衝突する可能性を防止します。(Heikki)
このコマンドは、先のディレクトリで未使用であることが確認されたファイル名を選ばずに、既存のファイル名を再使用しようとしていました。
- 1つの問い合わせ項目がテキストの先頭の単語に一致した場合、テキスト検索の間違った見出し生成を修正しました。(Sushant Sinha)
- `--enable-integer-datetimes`で構築された場合、ISO以外の日付書式を使用した時の時間間隔値における不適切な秒端数表示を修正しました。(Ron Mayer)

エスケープされていた場合であっても`ILIKE`が大文字小文字を区別せずに文字を比較するようにしました。(Andrew)

- 文のログ処理によって`DISCARD`が適切に取り扱われることを確実にしました。(Tom)
- PITR復旧中の前回完了したトランザクション時刻のログ処理の間違いを修正しました。(Tom)
- 渡されたタプルとタプル記述子が異なる列数を持っていた場合に`SPI_getvalue`および`SPI_getbinval`が適切に動作することを確実にしました。(Tom)

テーブルに列が追加または削除された場合、こうした状況は正常です。しかしこの2つの関数は適切に扱っていませんでした。起こり得る唯一の結果は間違ったエラーが示されることです。

- `SessionReplicationRole`を`PGDLLIMPORT`として印をつけました。このためWindows上のSlonyで使用することができます。(Magnus)
- `libpq`の`gsslib`パラメータを使用している時の小さなメモリリークを修正しました。(Magnus)

パラメータ文字列で使用された領域が接続が閉じた時に開放されてませんでした

- 必要に応じて`libgssapi`が`libpq`にリンクされることを確実にしました。(Markus Schaaf)
 - `ecpg`の`CREATE ROLE`解析を修正しました。(Michael)
 - `pg_ctl restart`の最近の破損を修正しました。(Tom)
 - `pg_control`がバイナリモードで開かれることを確実にしました。(Itagaki Takahiro)
- `pg_controldata`および`pg_resetxlog`はこれを正しく行っていないでました。このためWindowsで失敗しました。
- 時間帯データファイルを`tzdata`リリース2008i（アルゼンチン、ブラジル、モーリシャス、シリアにおける夏時間規則の変更）まで更新しました。

E.67. リリース8.3.4

リリース日: 2008-09-22

このリリースは8.3.3に対し、各種の不具合を修正したものです。8.3メジャーリリースにおける新機能については[項E.71](#)を参照してください。

E.67.1. バージョン8.3.4への移行

8.3.Xからの移行ではダンプ/リストアは不要です。しかし、8.3.1より前のバージョンからアップグレードする場合は、8.3.1のリリースノートを参照してください。

E.67.2. 変更点

- B-tree WALリカバリコード内の不具合を修正しました。(Heikki)

ページ分割操作中にWALが途中で止まった場合にリカバリが失敗しました。

- HOTページの切り詰めの際に間違ったカットオフXIDを使用する可能性を修正しました。(Alvaro)

このエラーにより、`VACUUM`によって生じるシステムカタログの破壊の危険性が生まれました。不要タプルの削除が早すぎる可能性があります。カタログの検証中にシステムがMVCC規則に従わないため、実行中のデータベース捜査における影響は最小です。しかし、`pg_dump`や他のクライアントプログラムからの出力が過渡的に間違ったものになるかもしれません。

- `datfrozensid`の誤計算の可能性を修正しました。(Alvaro)

このエラーにより古い`pg_clog`データの削除に失敗するというこ最近の報告が説明できます。

- `pg_class`が再インデックス付けされた後不正確なHOT更新を修正しました。(Tom)

`REINDEX TABLE pg_class`が`ALTER TABLE RENAME`または`ALTER TABLE SET SCHEMA`コマンドによるセッションと同じセッションで続けられた場合`pg_class`の破損が発生する可能性があります。

- 失った"コンボ cid"状況を修正しました。(Karl Schnaitter)

複数の副トランザクションで削除されたが、すべての副トランザクションがアボートされた場合、このエラーによりトランザクションから不正に不可視の行が生じました。

- 自動バキュームにより検査中のテーブルが運悪く削除された場合に自動バキュームがクラッシュしないようにしました。(Alvaro)

- 局所ロックのカウンタを32ビットから64ビットに広げました。(Tom)

これは、かなり長いトランザクションでカウンタがオーバーフローし、想定外の"すでに保持済のロックである"というエラーになるという報告に対応したものです。

- GiSTインデックススキャン中に出力タプルが重複する可能性を修正しました。(Teodor)

- どちらかのテーブルが変更された場合の外部キー検査問い合わせを初めから作り直しました。(Tom)

これまで8.3は問い合わせの計画を再作成しようとしていましたが、これまでに生成された問い合わせテキストから作業していました。これは、テーブルや列の名前が変わった場合に失敗することになります。

- ビューが単純な`UNION ALL`式を含む場合、行われなかった権限検査を修正しました。(Heikki)

被参照テーブルの権限は正しく検査されていましたが、ビュー自体の権限はそうではありませんでした。

- `INSERT`または`UPDATE`で生成されるタプルが対象のテーブルの現在の行型に一致するか確認するために、エグゼキュータ起動の際の検査を追加しました。(Tom)

このような状況は8.3では起こりえないと考えていますが、過去のリリースでは発生することがあります。ですので、この検査は用心深いものといえます。

- `DROP OWNED`中の削除が繰り返される可能性を修正しました。(Tom)

これによりよく"リレーションNNNのキャッシュ検索に失敗した"などのおかしなエラーが発生しました。

- XML操作における複数のメモリリークを修正しました。(Kris Jurka, Tom)
- 対象のデータ型で受け入れられない場合、正しくエラーになるように`xmlserialize()`を修正しました。(Tom)
- テキスト検索設定ファイルの解析においてマルチバイト文字を間違って扱う数か所を修正しました。(Tom)

設定ファイル内に存在する特定の文字により常に"符号化方式において無効なバイトシーケンス"という失敗が発生しました。

- テキスト検索設定ファイル内で報告されるすべてのエラーに対してファイル名と行番号を含めました。(Tom)
- まず時間帯引数を時間帯省略形として解釈し、これまでのように他の方法を取らずに、それが失敗した場合にのみ完全な時間帯名称として解釈するように`AT TIME ZONE`を修正しました。(Tom)

タイムスタンプ入力関数は常にこの順序であいまいな時間帯名の解決を行っていました。`AT TIME ZONE`も同様にすることで一貫性が向上し、そして、8.1で混入された互換性に関する不具合を修正します。過去のバージョンの`AT TIME ZONE`では省略形のみを受け付けていましたので、あいまいな場合に8.0以前と同様な動作をするようになりました。

- 64ビットプラットフォームで稼働している場合、日付時刻入力関数が正しく整数オーバーフローを検知できるよう修正しました。(Tom)
- 単位指定がある設定パラメータを表示する際に、単位変換による整数オーバーフローを防ぎます。(Tom)
- 非常に長いログメッセージをsyslogに書き出す際の性能を向上しました。(Tom)
- `pg_hba.conf`内のLDAP URLの接尾辞部分に空白文字を許します。(Tom)
- `SELECT DISTINCT ON`問い合わせに対するカーソルの後方スキャンにおける不具合を修正しました。(Tom)
- 間違っ外部結合の下に`IS NULL`検査を押し下げるというプランナの不具合を修正しました。

これは、上位の`OR`句のすべての辺において同一のリレーションに対する`IS NULL`検査が存在する場合に発生しました。

- 入れ子状の副問い合わせ式がある場合のプランナの不具合を修正しました。(Tom)

外側の副問い合わせが親問い合わせに直接の依存性がなく、内側の副問い合わせが依存性がある場合、外側の値が新しい親問い合わせの行に対して再計算されなくなる可能性があります。

- プランナが論理型の結果を生成する`GROUP BY`式を、式の内容に関係なく、常に2つのグループになると推定することを修正しました。(Tom)

これにより、`col IS NULL`などの特定の論理試験に対して、通常の`GROUP BY`推定より非常により良く精度が上がります。

- `FOR`ループの対象値が複合型のフィールドを含むレコードである場合に失敗しないようにPL/pgSQLを修正しました。(Tom)
- Tcl 8.5で正しく動作するようPL/Tclを修正しました。また、Tcl間でやりとりされるデータの符号化方式についてより注意を払うように修正しました。(Tom)
- `PQescapeBytea()`の性能を向上しました。(Rudolf Leitgeb)
- Windowsにおいて、libpqがシステムコール当たり64kBを超えて送信しないようにすることで、Microsoftの不具合を回避しました。(Magnus)
- `SET`コマンドで変数を正しく扱うよう`ecpg`を修正しました。(Michael)
- `pg_dump`および`pg_restore`がSQLコマンドの送信に失敗した後に報告するエラーを改良しました。(Tom)
- `pg_ctl`が`restart`の間正しく`postmaster`のコマンドライン引数を保持するように修正しました。(Bruce)
- `pg_standby`の間違ったWALファイルのカットオフポイント計算を修正しました。(Simon)
- 時間帯データファイルをtzdataリリース2008f（アルゼンチン、バハマ、ブラジル、モーリシャス、モロッコ、パキスタン、パレスチナ、パラグアイにおける夏時間規則の変更）まで更新しました。

E.68. リリース8.3.3

リリース日: 2008-06-12

このリリースは8.3.2に対し、重大な1つの不具合と軽微な1つの不具合を修正したものです。 8.3メジャーリリースにおける新機能については[項E.71](#)を参照してください。

E.68.1. バージョン8.3.3への移行

8.3.Xからの移行ではダンプ/リストアは不要です。 しかし、8.3.1より前のバージョンからアップグレードする場合は、8.3.1のリリースノートを参照してください。

E.68.2. 変更点

- `pg_get_ruledef()` で負の定数を括弧でくるようにしました。(Tom)

この修正前では、ビューまたはルールにおける負の定数がそのまま、例えば`-42::integer`のようにダンプされる可能性があります。 これは若干正しくありません。 演算子の前置規則に従うと、`(-42)::integer`となるべきです。 通常は大した違いが起こることはありませんが、最近の別のパッチと相互作用して、PostgreSQLが有効な`SELECT DISTINCT`ビューの問い合わせを拒絶することになりました。 `pg_dump`の出力のリロード失敗という結果になることもあり得たため、これは高優先度の修正として扱われました。 実際にダンプ出力が正しくなかったバージョンは8.3.1と8.2.7だけです。

- `ALTER AGGREGATE ... OWNER TO`が`pg_shdepend`を更新するようにしました。(Tom)

この見落としにより、集約が後で`DROP OWNED`または`REASSIGN OWNED`操作で使用された場合に問題が発生するおそれがありました。

E.69. リリース8.3.2

リリース日: リリースされませんでした

このリリースには8.3.1からの各種修正が含まれています。 8.3メジャーリリースにおける新機能については[項E.71](#)を参照してください。

E.69.1. バージョン8.3.2への移行

8.3.Xからの移行ではダンプ/リストアは不要です。 しかし、8.3.1より前のバージョンからアップグレードする場合は、8.3.1のリリースノートを参照してください。

E.69.2. 変更点

- WindowsでUTF-8データベース符号化方式と異なるクライアント符号化方式が使用された場合に発生する`ERRORDATA_STACK_SIZE exceeded`のクラッシュを修正しました。(Tom)
- `recovery_command`パラメータの`%r`マクロにおける間違ったアーカイブ消去ポイント計算を修正しました。(Simon)
ウォームスタンバイスクリプトがいつWALセグメントファイルを捨てるかを`%r`に基づいて決める場合にデータ損失が発生する可能性があります。
- 新しい列がすべて非NULLに初期化されていることが正しく検査されるように`ALTER TABLE ADD COLUMN ... PRIMARY KEY`を修正しました。(Brendan Jurd)
これまでのバージョンでは、この必要条件の検査をまったく行っていませんでした。
- 手続き言語にも動作できるように`REASSIGN OWNED`を修正しました。(Alvaro)
- `SELECT`以外の最上位操作を行う問い合わせの副問い合わせとして`SELECT FOR UPDATE/SHARE`が存在した場合の問題を修正しました。(Tom)
- "同じ"制約を、共通の祖先からその制約を継承する複数の親リレーションから継承する場合に、`CREATE TABLE`が失敗する可能性を修正しました。(Tom)
- `UPDATE`または`DELETE`の対象テーブルに別名が付与されていたならば、`pg_get_ruledef()`がその別名を表示するように修正しました。(Tom)
- TidScan計画で使用されるTID内の範囲外ブロック番号が警告なく行にまったく一致しないという結果になる、8.3以前の動作に戻しました。
8.3.0と8.3.1では代わりにエラーになりました。
- `too many LWLocks taken`という結果をもたらす、GINの不具合を修正しました。(Teodor)
- `tsquery`における壊れたGiST比較関数を修正しました。(Teodor)
- 動作することが見込まれる型についてのドメインを受け付けるように`tsvector_update_trigger()`と`ts_stat()`を修正しました。(Tom)
- 列挙データ型を外部キーとしてサポートするエラーを修正しました。(Tom)
- 破損したデータを伸長する時に潜在的クラッシュを防止します。(Zdenek Kotala)
- 遅延削除と`DROP DATABASE`間の競合条件を修正しました。(Heikki)

最悪の場合、新しいデータベースに新しく作成したテーブルが最近削除されたテーブルと同じOIDを持つことが起こったら、そのテーブルを削除する結果になりました。 しかし、もちろんこれは非常に可能性が低いシナリオです。

- バックエンドが共有メモリを破損したままSIGTERMによる終了する2か所を修正しました。(Tom)

SIGTERMがデータベースクラスタ全体を停止するために使用された場合、どちらも非常に重要ではありません。 しかし、個々のバックエンドにSIGTERMを送信することを試みる場合が問題でした。

- x と y が異なるデータ型を取る場合に`x IN (SELECT y FROM ...)`句に対して生成される不正確な計画による潜在的クラッシュを修正しました。そして、 y 型から x 型への変換に損失がある場合に、意味的な正確に動作することを確認しました。(Tom)
 - プランナが既知のParam値が定数であった場合にそれを代用しないという見落としを修正しました。(Tom)
- 8.3.0と8.3.1では、このミスは部分的に無名の拡張問い合わせ文の最適化を無効にしました。特に、LIKEパターンがパラメータとして渡される場合、LIKE対インデックススキャン最適化は決して適用されませんでした。また、パラメータ値に依存した制約による除外も動作しませんでした。
- インデックス可能なMINまたはMAX集約がDISTINCTまたはORDER BYで使用された場合のプランナの失敗を修正しました。(Tom)
 - プランナがSortノードを生み出す計画ノードで"physical tlist"を使用しないことを確実にしました。(Tom)
- これによりソートされたデータ内に未使用の列値が含まれるため、本来必要なデータより多くのデータをソートで処理しなければならませんでした。
- 不要な問い合わせ文字列の複製をやめました。(Tom)
- この修正は、8.3.0でもたらされた、多数のコマンドが単一問い合わせ文字列として受け付けた時の性能問題を修正します。
- 子トランザクションのXIDを検査する際に、`TransactionIdIsCurrentTransactionId()` がリニア検索ではなく二分検索を使用するようにしました。(Heikki)
- これは、8.3.0が以前のリリースに比べ非常に低速になる一部の状況を修正します。
- キリル語の"Yo"文字 (2つのドットを持つёとѐ) を扱うISO-8859-5と他の符号化方式との間の変換を修正しました。(Sergey Burladyan)
 - 複数のデータ型入力関数、特に、結果に初期化されず予測できない値を含む未使用のバイトを許す`array_in()`を修正しました。(Tom)
- これは、2つの明らかに同じリテラル値が等しいとみなされず、結果パーサは一致しないORDER BYおよびDISTINCT式に関して不平をいうといったエラーをもたらします。
- 正規表現副文字列一致 (`substring(string from pattern)`) における境界状態を修正しました。(Tom)
- この問題は、パターン全体は一致するが、ユーザが括弧付きの副式を指定し、その副式が一致しなかった時に発生します。例えば、`substring('foo' from 'foo(bar)?')` です。(bar) は一致しませんので、これはNULLを返すべきです。しかし、間違ってパターン全体が一致 (つまりfoo) という結果を返していました。
- XID周回を防ぐために起動された自動バキュームのキャンセルを防ぎます。(Alvaro)
 - 統計情報コレクタに報告する数がより正確になるように、ANALYZEの疑わしいタプル (まだコミットされていないトランザクションにより挿入または削除されたタプル) の扱いを改良しました。(Pavan Deolasee)
 - `--xlogdir (-X)` オプションにおける相対パスを拒絶するようにinitdbを修正しました。(Tom)
 - psqlがタブ文字を、8.3.0と8.3.1で使用された`¥x09`ではなく、適切な数の空白文字で表示するようにしました。(Bruce)
 - 時間帯データファイルをtzdataリリース2008c (モナコ、イラク、チョイバルサン、パキスタン、シリア、キューバ、アルゼンチン/サンルイスにおける昼時間規則の変更) まで更新しました。
 - ecpglibに`ECPGget_PGconn()` 関数を追加しました。(Michael)
 - ecpgの`PGTYPEStimestamp_sub()` 関数からの不正確な結果を修正しました。(Michael)
 - ecpgにおける継続行印の扱いを修正しました。(Michael)
 - `contrib/cube` 関数に潜在したクラッシュを修正しました。(Tom)
 - 入力問い合わせがNULL値を返す場合の`contrib/xml2`の`xpath_table()` 関数のコアダンプを修正しました。(Tom)
 - `contrib/xml2`のmakefileがCFLAGSを上書きしないように修正しました。また、libxsltの存在を適切に自動検出するようにしました。(Tom)

E.70. リリース8.3.1

リリース日: 2008-03-17

このリリースは8.3.0の各種不具合を修正したものです。 8.3メジャーリリースにおける新機能については[項E.71](#)を参照してください。

E.70.1. バージョン8.3.1への移行

8.3.Xからの移行ではダンプ/リストアは不要です。 しかし、後述のWindowsロケール問題の影響がある場合、移行後にテキスト型の列上のインデックスに対しREINDEXを行わなければならないかもしれません。

E.70.2. 変更点

- 異なる文字の組み合わせを等価とみなすWindowsロケールにおける文字列比較を修正しました。(Tom)

この修正はWindowsのみ、かつ、UTF-8データベース符号化方式を使用している場合にのみ当てはまります。 他の状況に対する同じ修正が2年前になされましたが、UTF-8を使用するWindowsでは変更されなかった経路のコードを使用しました。 同一ではない一部の文字列を等価とみなすロケールを使用している場合、既存のテキスト型列上のインデックスを修正するためにREINDEXを行う必要があるかもしれません。

- VACUUM FULLにおける境界不具合を修正しました。(Tom)

異なるシステムカタログに対する同時実行VACUUM FULLの間でのデッドロックの可能性が8.2で発生していました。 これを修正しました。 デッドロックが致命的なコード部分で発生し得たため、8.3ではこれが悪化し、ただのERROR条件ではなくPANICとなりました。

また、システムカタログのバキューム中に途中で失敗したVACUUM FULLにより、同時実行中のデータベースセッションのキャッシュを破壊する結果になります。

他にも8.3で発生したVACUUM FULLの不具合により、有効なタプルを含まないページの処理中にクラッシュまたはメモリ不足報告が発生する結果になります。

- characterまたはbit列を含む外部キー検査の誤動作を修正しました。(Tom)

参照列が異なる型だが型互換がある場合（例えばvarchar）、制約が間違っって強制されました。

- 不要となった外部キー検査における不必要なデッドロック失敗を防ぎます。(Stephan Szabo, Tom)
- 準備済文の再計画時のコアダンプの可能性を修正しました。(Tom)

この不具合は、SQL PREPAREではなく、プロトコルレベルの準備操作にのみ影響しました。 このため、準備済文を非常に多く使用するJDBC、DBIなどクライアントサイドのドライバでのみ現れる傾向がありました。

- SPIを使用する関数を呼び出す問い合わせの再作成時に可能性がある失敗を修正しました。(Tom)
- 異なるデータ型を含む行全体の比較における失敗を修正しました。(Tom)
- 長く存在したLISTEN/NOTIFY競合条件を修正しました。(Tom)

まれな場合ですが、NOTIFYを実行した同時実行中のトランザクションが後でコミットされたことが確認できたため、受け取ることが期待できる場合であっても、LISTENを実行したばかりのセッションが注意を受け取らない可能性があります。

この修正の副作用として、まだコミットされていないLISTENコマンドを実行したトランザクションが、LISTENで検索するために選ばなければならないpg_listener内の行にまったく現れなくなりました。 これまでは現れていました。 この動作についてはどちらの方式も文書化されていませんでしたが、過去の動作に依存するアプリケーションがあるかもしれません。

- 準備されたトランザクションではLISTENとUNLISTENは使用できません。(Tom)

これまでは許されていましたが、これを行うと様々な悪影響が発生しました。 特に構成したバックエンドはUNLISTENが未コミットである間終了することができませんでした。

- 準備されたトランザクションでは一時テーブルの削除を行うことはできません。(Heikki)
これは正確には8.1で禁止されていたのですが、8.2と8.3では不注意にこの検査が壊れていました。
- ハッシュインデックスを使用する問い合わせの間にエラーが発生した場合のまれなクラッシュを修正しました。(Heikki)
- `tsquery`値の間違った比較を修正しました。(Teodor)
- 単一バイト符号化方式における非ASCII文字のLIKEの間違った動作を修正しました。(Rolf Jentsch)
- `xmlvalidate`を無効にしました。(Tom)
この関数は8.3リリース以前に削除すべきでしたが、不注意にソースコード内に残っていました。非特権ユーザがサーバでアクセス可能な任意のファイルの先頭数文字を読み取ることができますので、これは小さなセキュリティ上の危険性があります。
- ある方法で集合を返す関数を使用した場合のメモリリークを修正しました。(Neil)
- `encode(bytea, 'escape')`がすべての上位ビット群を¥nnn8進エスケープ列に変換するようにしました。(Tom)
データベース符号化方式がマルチバイトの場合に符号化問題を防ぐために必要です。`encode`の特定の結果を想定するアプリケーションでは、この変更は互換性に関する問題が発生します。
- 紀元前の2月29日に関する日付時刻値の入力を修正しました。(Tom)
これまでのコードでは、うるう年の判定に間違いがありました。
- `ALTER OWNER`の一部の構文で発生した"未知のノード型"エラーを修正しました。(Tom)
- `CREATE TABLE LIKE INCLUDING INDEXES`におけるテーブル空間権限エラーを防止しました。(Tom)
- ロック待ちが中断した時に`pg_stat_activity.waiting`フラグを確実に初期化します。(Tom)
- Windows Vistaにおいてプロセス処理の取扱いを修正しました。(Dave, Magnus)
具体的には、この修正により管理者ユーザとしてサーバを起動できるようになります。
- 時間帯データファイルを`tzdata`リリース2008aに更新しました(具体的には、最近のチリの変更です)。時間帯省略形VET (ベネズエラ)がUTC-4:00ではなくUTC-4:30を意味するように調整しました。(Tom)
- 配列に関する`ecpg`の問題を修正しました。(Michael)
- `pg_ctl`が正しくコマンドラインオプションから`postmaster`のポート番号を取り出すように修正しました。(Itagaki Takahiro, Tom)
これまでは、`pg_ctl start -w`は間違ったポート上の`postmaster`に接続しようとし、おかしな起動失敗の報告をもたらしていました。
- 最近のバージョンのgccにおける、最適化ミスの可能性を防止するために`-fwrapv`を使用します。(Tom)
これはgcc 4.3以降でPostgreSQLを構築する場合に必須であることは分かっていました。
- MSVCを使用した`contrib/uuid-oss`の構築が可能になりました。(Hiroshi Saito)

E.71. リリース 8.3

リリース日: 2008-02-04

E.71.1. 概要

重大な新機能と性能向上を伴うこのリリースは、PostgreSQLを大きく飛躍させたものと言えます。これは開発速度が劇的に速くなったコミュニティの成長によりなされました。このリリースは以下の主要機能を追加します。

- 全文検索がコアデータベースシステムに統合されました。
- SQL/XML標準をサポートしました。これに伴い、新しい演算子とXMLデータ型が含められました。
- 列挙データ型 (ENUM)
- 複合型の配列
- 汎用一意識別子 (UUID) データ型
- NULLのソートを先にするか後にするかに関する制御を追加
- 更新可能カーソル
- サーバ設定パラメータが関数単位で設定可能になりました。
- ユーザ定義型が型修飾子を持つことができるようになりました。
- テーブル定義の変更時または統計情報の更新時における、キャッシュされた問い合わせの自動計画再作成
- ログ処理および統計情報収集における多くの改良
- Windowsにおける認証のためのセキュリティサービスプロバイダインタフェース (SSPI) のサポート
- 複数の同時実行autovacuumプロセスのサポート。およびその他のautovacuumの改良
- Microsoft Visual C++を使用したPostgreSQL配布物全体のコンパイルを可能にしました。

主要な性能に関する改良を以下に列挙します。これらの改善のほとんどは自動的なものですので、ユーザによる変更や調整は不要です。

- 非同期コミットは、トランザクションコミット時のWALへの書き出しを遅延します。
- 各チェックポイントにおけるI/Oのピークを抑えるために、チェックポイントの書き出しをより長い時間に分散させることができます。
- ヒープオンリータブル (HOT) は、ほとんどのUPDATEとDELETEにおける領域の再利用を促進します。
- ジャストインタイムのバックグラウンドライタ戦略はディスク書き出し効率を改善します。
- 読み取りのみのトランザクションに対して非永続的なトランザクションIDを使用することで、オーバーヘッドとVACUUMの必要性を削減します。
- フィールド単位および行単位の格納にかかるオーバーヘッドが削減されました。
- 大規模なシーケンススキャンがよく使用されるキャッシュされたページを押し出さないようになりました。
- 同時に行われる大規模なシーケンススキャンがディスク読み取りを共有できるようになりました。
- ORDER BY ... LIMITがソートなしで行うことができるようになりました。

上記項目は後述の節でより詳細に説明します。

E.71.2. バージョン8.3への移行

すべての過去のリリースからデータを移行する場合、pg_dumpを使用したダンプ/リストアが必要です。

既知の非互換性を以下に示します。

E.71.2.1. 一般

- 文字以外のデータ型がTEXTへの自動キャストが行われなくなりました。(Peter, Tom)

これまでの、文字以外の値がtext入力を必要とする演算子や関数に渡された場合、ほとんど（すべてではありません）の組み込みデータ型では自動的にtextにキャストされました。これが行われなくなりました。すべての文字以外の型でtextへの明示的なキャストが必要になります。例えば以下の式は以前は動作しました。

```
substr(current_date, 1, 4)
23 LIKE '2%'
```

しかし、今回からそれぞれ"function does not exist"および"operator does not exist"というエラーメッセージが引き出されるようになります。代わりに以下のように明示的なキャストを使用してください。

```
substr(current_date::text, 1, 4)
23::text LIKE '2%'
```

（当然ですが、より冗長なCAST()構文を使用することも可能です。）変更理由は、こうした自動キャストがよく思いもよらぬ動作を引き起こすためです。過去のリリースにおけるこうした例を次に示します。この式は受け付けられますが、想定した動作を行いません。

```
current_date < 2017-11-17
```

これは実際には日付と整数とを比較します。これは拒絶すべきものです（このリリースでは拒絶するようになります）。しかし、自動キャストの存在により、他に<演算子に一致するものがなければtext < text演算子がこの式に一致することができますので、両辺はtextにキャストされ、テキストの比較が行われます。

char(n)およびvarchar(n)型はまだtextに自動キャストされます。また、textへの自動キャストはまだ連結演算子(||)では、少なくとも1つの入力が文字型であれば、動作します。

- contrib/tsearch2の全文検索機能がコアサーバに移りましたが、いくつか小さな構文変更があります。

contrib/tsearch2は互換性を保持するためのインタフェースを持つようになりました。

- ARRAY(SELECT ...)においてSELECTが行を返さない場合、NULLではなく空の配列を返すようになりました。(Tom)
- 基本データ型に対する配列型の名前が常にアンダースコアを基本型名称の前につけたものになりました。

古い命名規約は可能ならばまだ尊重されますが、アプリケーションのコードはこれに依存すべきではありません。指定した型に関連付いた配列データ型を識別する際は、代わりに新しいpg_type.typarray列を使用してください。

- ORDER BY ... USING operatorがB-tree演算子クラスで定義される小なり演算子または大なり演算子を使用しなければならなくなりました。

この制限は一貫性がない結果を防ぐために加えられました。

- SET LOCALがロールバックされない限り、最も外側のトランザクションの終わりまで永続するようになりました。(Tom)

以前はSET LOCALの効果は副トランザクションのコミット (RELEASE SAVEPOINTまたはPL/pgSQL例外ブロックからの終了) 後失われました。

- トランザクションブロック内で拒絶されたコマンドが、複数文からなる問い合わせ文字列内でも拒絶されるようになりました。(Tom)

例えば"BEGIN; DROP DATABASE; COMMIT"は単一の問い合わせメッセージとして送信されたとしても拒絶されるようになります。

- トランザクションブロック外のROLLBACKがWARNINGではなくNOTICEを発行するようになりました。(Bruce)
- NOTIFY/LISTEN/UNLISTENがスキーマ修飾された名前を受付ないようにしました。(Bruce)

これまでのこれらのコマンドはschema.relationを受け付けていましたが、スキーマ部分を見捨てていたため、混乱を招きました。

- ALTER SEQUENCEがシーケンスのcurrval()の状態に影響を与えないようになりました。(Tom)
- 外部キーはデータ型をまたがる参照においてインデックス付け可能な状態に一致しなければならなくなりました。(Tom)

これは意味的な一貫性を向上させ、また、性能に関する問題を防止する役に立ちます。

- オブジェクトサイズに関する情報を閲覧する合理的な権限を持つユーザに、オブジェクトサイズに関する関数を限定します。(Tom)

例えばpg_database_size()は、デフォルトで全員に与えられているCONNECT権限を要求します。pg_tablespace_size()はテーブル空間におけるCREATE権限を要求します。また、テーブル空間がデータベースのデフォルトのテーブル空間である場合にも許可されます。

- 文書化されていない!= (含まれていない) 演算子を削除しました。(Tom)

こうした操作を行うにはNOT IN (SELECT ...)が適しています。

- 内部用ハッシュ関数がより均一に分布するようになりました。(Tom)

アプリケーションコードがPostgreSQL内部用のハッシュ関数を使用してハッシュ値を計算し格納する場合、ハッシュ値は再生成されなければなりません。

- 可変長データ値の取扱いに関するCコード規約を変更しました。(Greg Stark, Tom)

新しい`SET_VARSIZE()` マクロは、生成された`varlena`型の長さを設定するために使用しなければなりません。また、より多くの状況で入力値の伸長("TOAST解除")を行う必要があるかもしれません。

- `DEBUG` レベルを使用しない限り、連続アーカイブ処理がアーカイブ操作の成功の度にサーバログに報告しないようになりました。

E.71.2.2. 設定パラメータ

- サーバ管理用のパラメータで多くの変更があります。

`bgwriter_lru_percent`、`bgwriter_all_percent`、`bgwriter_all_maxpages`、`stats_start_collector`、`stats_reset_on_server_start` が削除されました。 `redirect_stderr` の名前が `logging_collector` になりました。 `stats_command_string` の名前が `track_activities` になりました。 `stats_block_level` と `stats_row_level` は `track_counts` にまとめられました。 `archive_mode` という新しい論理型の設定パラメータはアーカイブ処理を制御します。 `Autovacuum` のデフォルト設定が変わりました。

- `stats_start_collector` パラメータが削除されました。(Tom)

UDPソケットの作成に失敗しない限りコレクタプロセスを常に起動するようになりました。

- `stats_reset_on_server_start` パラメータが削除されました。(Tom)

この目的に `pg_stat_reset()` を使用することができますので、このパラメータは削除されました。

- `postgresql.conf` でパラメータをコメントにすることで、そのデフォルト値に戻るようにしました。(Joachim Wieland)

これまではコメント化された項目は、次にサーバを再起動するまで、その値は変わらずに残りました。

E.71.2.3. 文字符号化方式

- 符号化が無効なデータに対して、より多くの検査を追加しました。(Andrew)

この変更は、バックスラッシュでエスケープされた文字列リテラル処理と `COPY` のエスケープ処理に存在したいくつかの抜け穴を塞ぐものです。これで、結果が無効なマルチバイト文字を作成したかどうか、エスケープを解除した文字列を再度検査するようになりました。

- サーバのロケール設定と矛盾するデータベース符号化方式を不許可としました。(Tom)

多くのプラットフォームでは、`C` ロケールがすべてのデータベース符号化方式で動作する唯一のロケールです。他のロケール設定は特定の符号化方式を意味することになり、データベース符号化方式と多少異なると、誤動作します。(典型的な症例として、間違ったテキストソート順や `upper()` もしくは `lower()` の誤動作があります。) サーバは互換性が無い符号化方式でデータベースを作成することを拒絶するようになりました。

- `chr()` が無効な符号化方式の値を生成できないことを確実にしました。(Andrew)

UTF8で符号化されたデータベースでは、`chr()` の引数はUnicodeコードポイントとして扱われるようになりました。他のマルチバイト符号化方式では、`chr()` の引数は7ビットASCII文字を指名しなければなりません。ゼロは受け付けられません。 `ascii()` も併せて調整されました。

- `convert()` の動作を有効な符号化方式に確実にするように調整しました。(Andrew)

2つの引数を取る `convert()` 構文は削除されました。3つの引数を取る構文は、最初の引数として `bytea` を取り、`bytea` を返すようになりました。機能の欠落を補うために、新しい関数を3つ追加しました。

- `convert_from(bytea, name)` returns text — 最初の引数を指定された符号化方式からデータベースの符号化方式に変換します。
- `convert_to(text, name)` returns bytea — 最初の引数をデータベース符号化方式から指定された符号化方式に変換します。
- `length(bytea, name)` returns integer — 指定した符号化方式における最初の引数の文字長を返します。

- `convert(argument USING conversion_name)` を削除しました。(Andrew)

この動作は標準SQLに合いませんでした。

- JOHAB符号化方式をクライアント側のみにしました。(Tatsuo)

サーバ側の符号化方式としてJOHABは安全ではありません。

E.71.3. 変更点

PostgreSQL 8.3と以前のメジャーリリースとの間の違いについて、以下に詳しく説明します。

E.71.3.1. 性能

- 非同期コミットは、トランザクションコミット時のWALへの書き出しを遅延します。(Simon)

この機能により、短期のデータ変更を行うトランザクションの性能が大きく向上しました。欠点はディスクへの書き出しが遅れるため、データをディスクへの書き出す前にデータベースシステムまたはオペレーティングシステムがクラッシュした場合コミットしたデータが失われる点です。この機能は多少のデータ損失を受け入れることができるアプリケーションでは役に立ちます。`fsync`を無効にする場合と異なり、非同期コミットを使用する場合にはデータベースの一貫性に関する危険はありません。最悪の場合でも、クラッシュの後、コミットされたと報告されている最後の数個のトランザクションが完全にコミットされていない可能性があるだけです。この機能は`synchronous_commit`を無効にすることで有効になります（一部のトランザクションが極めて重要であり、その他は重要でないような場合、これはセッション単位やトランザクション単位で行うことができます。）。`wal_writer_delay`を調整して、トランザクションが実際にディスクに書き出されるまでの最大遅延を制御することができます。

- 各チェックポイントにおけるI/Oのピークを抑えるために、チェックポイントの書き出しをより長い時間に分散させることができます。(Itagaki Takahiro and Heikki Linnakangas)

これまでは、すべての変更済みバッファはチェックポイント時可能な限り高速にディスクに書き出すことが強制されていました。これによりI/Oが急激に増加するため、サーバの性能が劣化しました。この新しい方式はチェックポイント時のディスク書き出しを分散させ、I/O使用のピークを抑えます。（ユーザが要求したチェックポイントおよび停止時のチェックポイントではまだできる限り高速に書き出されます。）

- ヒープオンリータブル（HOT）は、ほとんどの`UPDATE`と`DELETE`における領域の再利用を促進します。(Pavan Deolaseeおよび他の多くからのアイデアによる)

`UPDATE`および`DELETE`は、失敗した`INSERT`と同様、不要なタブルを背後に残します。これまでは`VACUUM`のみが不要タブルにより占められた領域を回収することができました。HOTにより、インデックス付けされた列に変更がなされていなければ、不要タブル領域を`INSERT`または`UPDATE`と同時に、自動的に回収できるようになりました。これによりより一貫した性能が得られます。また、HOTは重複したインデックス項目が追加されることを防止します。

- ジャストインタイムのバックグラウンドライタ戦略はディスク書き出し効率を改善します。(Greg Smith, Itagaki Takahiro)

これはバックグラウンドライタの手作業による調整作業の必要性を大きく減らします。

- フィールド単位および行単位の格納にかかるオーバーヘッドが削減されました。(Greg Stark, Heikki Linnakangas)

128バイト未満の長さのデータ値を持つ可変長データ型において、その格納が3から6バイト減少することがわかります。例えば、隣接する2つの`char(1)`型のフィールドでは16バイトではなく4バイト使用するようになりました。また、行ヘッダは以前に比べ4バイト短くなりました。

- 読み取りのみのトランザクションにて非永続トランザクションIDを使用することでオーバーヘッドと`VACUUM`の必要性が減少します。(Florian Pflug)

非永続トランザクションIDは大域的なトランザクションカウンタを増加させません。このため、これは`pg_clog`への負荷を低減し、トランザクションID周回を防ぐための強制的なバキュームの実行間隔を長くします。同時実行性を向上させる性能向上も行われました。

- 読み取りのみのコマンドの後にコマンドカウンタの増加を行いません。(Tom)

これまでは1トランザクション当たり 2^{32} (40億)コマンドという固定の限界がありました。今回実際にデータベースを変更するコマンドのみが数えられるようになりました。このため制限自体はまだ存在しますが、悩まされることはかなり減少したはずです。

- バックエンドから作業を分けるために専用のWALライタプロセスを作成しました。(Simon)

- `CLUSTER`および`COPY`で不要なWAL書き出しを行わないようにしました。(Simon)

WALアーカイブ処理が有効でない限り、システムは`CLUSTER`でのWAL書き出しを行わず、コマンドの終りでテーブルを`fsync()`するだけになりました。また、テーブルが同じトランザクションで作成された場合の`COPY`でも同じことを行います。

- 大規模なシーケンススキャンがよく使用されるキャッシュされたページを押し出さないようになりました。(Simon, Heikki, Tom)

- 同時に行われる大規模なシーケンススキャンがディスク読み取りを共有できるようになりました。

これは、（他のシーケンススキャンがすでに実行中の）テーブルの途中から新しいシーケンススキャンを開始すること、および、その開始から終わりまでを包み隠すことによりなされました。これは`ORDER BY`の指定がない問い合わせで返される行の順序に影響を与えます。必要ならば`synchronize_seqscans`設定パラメータを使用して無効にすることができます。

- `ORDER BY ... LIMIT`がソートなしで行うことができるようになりました。(Greg Stark)

これは、テーブル全体を完全にソートする代わりに、テーブルのスキャンを断続的に行い、"上位N個の"候補行のみを追跡することで行われます。これは対応するインデックスがない場合やLIMITが大きな値でない場合に有効です。

- バックエンドから統計情報コレクタに送信されるメッセージの頻度に制限を加えました。(Tom)

これは短めのトランザクションにおけるオーバーヘッドを低減します。しかし、統計情報が計算される前の遅延が増加する場合があります。

- 多くのNULLを持つ場合のハッシュ結合の性能を向上しました。(Tom)
- 正確にデータ型が一致しない場合の演算子検索が高速になりました。(Tom)

E.71.3.2. サーバ

- 自動バキュームがデフォルトで有効になりました。(Alvaro)

デフォルトに変更することを正当化するために、自動バキュームを有効にさせた場合の欠点を除去するための変更が多くなされました。その他の自動バキュームに関するパラメータも複数変更されました。

- 複数の自動バキュームプロセスの同時実行をサポートします。(Alvaro, Itagaki Takahiro)

これにより複数のバキュームを同時に実行できるようになります。これは、大規模テーブルのバキュームによりより小さなテーブルのバキュームが遅延されることを防ぎます。

- テーブル定義の変更時または統計情報の更新時における、キャッシュされた問い合わせ計画を自動的に再作成します。(Tom)

これまで一時テーブルを参照するPL/pgSQL関数はEXECUTEが使用されていない限り、その関数を呼び出し中に一時テーブルが削除、再作成された場合、失敗しました。この改良により、この問題および関連する多くの問題が解消されます。

- 一時テーブルおよび一時ファイル用のテーブル空間を制御するtemp_tablespacesパラメータが追加されました。(Jaime Casanova, Albert Cervera, Bernd Helmle)

このパラメータは使用されるテーブル空間のリストを定義します。これにより複数のテーブル空間をまたがったI/O負荷分散が可能です。一時オブジェクトが作成される度にランダムなテーブル空間が選択されます。一時ファイルはデータベース単位のpgsql_tmp/ディレクトリに格納されず、テーブル空間単位のディレクトリに格納されるようになりました。

- 一時テーブル用のTOASTテーブルを特別なpg_toast_temp_nnnという名前のスキーマに格納します。(Tom)

これにより低レベルなコードでこれらのテーブルを一時的なものと認識できるようになります。WALに記録されない変更や共有バッファではなくアクセスにローカルを使用するなど、各種最適化が可能です。またこれは、バックエンドが予期せず一時TOASTテーブルへのファイル参照を開き続ける際の不具合を解消します。

- 新しい接続要求の固定的な処理フローが、postmasterの停止やクラッシュからの再起動を不定期間遅延させられるという問題を解消します。(Tom)
- 次のチェックポイントの後まで削除されたテーブルのrelfilenodeの再利用を防止することで、非常に可能性が低いデータが損失する状況を保護します。(Heikki)
- 古い形式の外部キートリガ定義を通常の外部キー制約に変換するようにCREATE CONSTRAINT TRIGGERを修正しました。(Tom)

これは、contrib/adddependを使用して返還されていない、7.3以前のデータベースで持ち込まれた外部キー制約の移行を簡単にします。

- 継承されたデフォルトを上書きするようDEFAULT NULLを修正しました。(Tom)

これまでDEFAULT NULLはノイズ語句とみなされていました。しかし、存在しなかった場合に親テーブルまたはドメインから継承される非NULLのデフォルトを上書きすべきでした（今回から上書きします）。

- 新しい符号化方式EUC_JIS_2004とSHIFT_JIS_2004を追加しました。(Tatsuo)

これらの符号化方式とUTF-8は相互に変換可能です。

- サーバ起動時のログメッセージを "database system is ready" から "database system is ready to accept connections" へ変更しました。また、その時期も調整しました。

このメッセージは本当に接続を受け付ける準備が整った時にのみ現れるようになりました。

E.71.3.3. 監視

- 自動バキュームの活動ログ処理の設定をサポートするためのlog_autovacuum_min_durationを追加しました。(Simon, Alvaro)
- ロック待機をログに記録するためのlog_lock_waitsパラメータを追加しました。(Simon)

一時ファイルの使用状況をログに記録するための`log_temp_files`パラメータを追加しました。(Bill Moran)

- チェックポイントのログ処理を改良するための`log_checkpoints`パラメータを追加しました。(Greg Smith, Heikki)
- `log_line_prefix`がすべてのプロセス内で`%s`および`%c`エスケープをサポートするようになりました。(Andrew)

これまでは、これらのエスケープはユーザセッションでのみ動作し、バックグラウンドデータベースプロセスでは動作しませんでした。

- ポイントインタイムリカバリの再起動点のログ処理を制御するための`log_restartpoints`を追加しました。(Simon)
 - リカバリの終わり、および、記録された再起動点で最終トランザクションの終了時刻がログに記録されるようになりました。(Simon)
 - 自動バキュームが`pg_stat_activity`内にその活動開始時刻を報告するようになりました。(Tom)
 - サーバログをカンマ区切り値 (CSV) 書式で出力できるようになりました。(Arul Shaji, Greg Smith, Andrew Dunstan)
- 後の解析のために、CSV書式のログファイルを簡単にデータベーステーブルにロードすることができます。

- PostgreSQLが提供する時間帯サポートをサーバログで表示されるタイムスタンプの書式に使用します。(Tom)

間違った符号化方式のローカルの時間帯を使用した場合のWindows固有の問題を防止します。 クライアント側で可視の`timezone`パラメータとは独立した、ログメッセージで使用する時間帯を制御するための`log_timezone`パラメータが追加されました。

- 新しい`pg_stat_bgwriter`システムビューはバックグラウンドライタの活動に関する統計情報を表します。
- データベース全体のタブル統計情報のための列を`pg_stat_database`に新しく追加しました。(Magnus)
- `xact_start` (トランザクションの開始時刻) 列を`pg_stat_activity`に追加しました。(Neil)

これで、長期間実行中のトランザクションの識別が容易になります。

- `n_live_tuples`および`n_dead_tuples`列を`pg_stat_all_tables`と関連するビューに追加しました。(Glen Parker)
- `stats_block_level`と`stats_row_level`パラメータを1つの`track_counts`パラメータにまとめました。 これは、統計情報コレクタプロセスにすべてのメッセージを送信するかどうかを制御します。(Tom)
- `stats_command_string`パラメータの名前を`track_activities`に変更しました。(Tom)
- コミットされたトランザクションとアボートしたトランザクションが異なる影響を持つことを認識するために、有効タプルと無効タプルの統計的な数え方を修正しました。(Tom)

E.71.3.4. 認証

- Windowsにおける認証用のセキュリティサービスプロバイダインタフェース (SSPI) をサポートします。(Magnus)
 - GSSAPI認証をサポートします。(Henry Hotz, Magnus)
- GSSAPIは業界標準ですので、Kerberos認証そのものより好まれるはずです。
- グローバルなSSL設定ファイルをサポートします。(Victor Wagner)
 - 受け付けるSSL暗号を制御するための`ssl_ciphers`パラメータを追加しました。(Victor Wagner)
 - Kerberosレルムパラメータ`krb_realm`を追加しました。(Magnus)

E.71.3.5. 先行書き込みログ (WAL) と連続的なアーカイブ処理

- トランザクションWAL記録に記録されるタイムスタンプが`time_t`からTimestampTz表現に変わりました。(Tom)
- これは0.5秒という精度をWALに提供します。 これはポイントインタイムリカバリで役に立ちます。
- ウォームスタンバイサーバで必要とするWALディスク容量を減少させました。(Simon)

この変更によりウォームスタンバイサーバは、不要になったWALファイルの自動削除を行いつつ、もっとも初期かつまだ必要なWALファイルの名前をリカバリスクリプトに渡すことができます。 これは`recovery.conf`の `restore_command`において`%r`を使用することで行われます。

- 新しい論理型の設定パラメータ`archive_mode`はアーカイブ処理を制御します。(Simon)

これまでは、`archive_command`を空文字列に設定することでアーカイブ処理を無効にしました。 今回からは`archive_command`と独立した`archive_mode`がアーカイブ処理の有効無効を切替えます。 これは一時的にアーカイブ処理を停止する際に役に立ちます。

E.71.3.6. 問い合わせ

- 全文検索がコアデータベースシステムに統合されました。(Teodor, Oleg)

テキスト検索が改良され、コアのコードに移り、デフォルトでインストールされるようになりました。 contrib/tsearch2は互換性を維持するためのインタフェースを含むようになりました。

- NULLのソートを先にするか後にするかに関する制御を追加しました。(Teodor, Tom)

ORDER BY ... NULLS FIRST/LASTという構文です。

- インデックスにおいて、列単位の昇順/降順 (ASC/DESC) 順序付けオプションが可能です。(Teodor, Tom)

これまでは、ASC/DESC指示子が混在したORDER BYを使用する問い合わせでは、インデックスを十分使用することはできませんでした。今回から、ASC/DESC指示子に一致するようにインデックスが作成された場合、こうした状況でもインデックスを十分に使用できるようになります。インデックス内のNULLのソート順も制御できるようになりました。

- col IS NULLがインデックスを使用できるようになりました。(Teodor)

- 更新可能なカーソル(Arul Shaji, Tom)

これは、カーソルで返される行をUPDATEまたはDELETEするためにプライマリーキーを参照する必要をなくします。 UPDATE/DELETE WHERE CURRENT OFという構文です。

- カーソルでFOR UPDATEが使用可能です。(Arul Shaji, Tom)

- データ型のI/O関数を呼び出すことで、すべてのデータ型と標準文字列型 (TEXT、VARCHAR、CHAR) との間の双方向のキャストをサポートする汎用機構を作成しました。(Tom)

これまでは、こうしたキャストはこの目的に特化した関数を持つ型でのみ利用可能でした。こうした新しいキャストは文字列へキャストする場合は代入のみ、逆は明示的のみです。このため予期しない動作を行わないはずです。

- すべての入力ドメイン型の時、UNIONとそれに関連する式がドメイン型を返すことができます。

これまでは、出力はドメインの基本型になるものとみなされていました。

- 2つの異なるデータ型を使用する時限定されたハッシュ処理が可能です。(Tom)

これにより、データ型を跨った比較を行う状況においてデータ型のハッシュ関数が互換性がある場合、ハッシュ結合、ハッシュインデックス、ハッシュ化副計画、ハッシュ集約を使用することができます。現在、データ型を跨るハッシュ処理のサポートは、smallint/integer/bigintに対するもの、およびfloat4/float8に対するものが存在します。

- WHERE句内に同じ変数があることを検知する、オプティマイザのロジックを改良しました。(Tom)

これにより、マージ結合が降順ソート順で動作するようになります。また、冗長なソート列の認識も改良されました。

- ほとんどのテーブルが制約による除外を行う大規模な継承構造の計画を作成する性能が向上しました。(Tom)

E.71.3.7. オブジェクトの操作

- 複合型の配列。(David Fetter, Andrew, Tom)

明示的に宣言された複合型の配列の他、システムカタログ、シーケンス、TOASTテーブルの行型を除く、通常のテーブルやビューの行型の配列もサポートされます。

- サーバ設定パラメータを関数単位で設定できるようになりました。(Tom)

例えば関数は、実行時に異なるsearch_pathが存在する場合の誤動作を防ぐために、独自のsearch_pathを設定することができます。 SECURITY DEFINER関数はセキュリティの抜け穴を防ぐためにsearch_pathを設定すべきです。

- CREATE/ALTER FUNCTIONがCOSTおよびROWSオプションをサポートするようになりました。(Tom)

COSTにより関数呼び出しにかかるコストを指定できます。ROWSにより集合を返す関数で返される平均行数を指定できます。これらの値は最善の計画を選択する際にオプティマイザにより使用されます。

- CREATE TABLE LIKE ... INCLUDING INDEXESを実装しました。(Trevor Hardcastle, Nikhil Sontakke, Neil)

- CREATE INDEX CONCURRENTLYは他のデータベース内のトランザクションを無視することができます。(Simon)

- ALTER VIEW ... RENAME TOおよびALTER SEQUENCE ... RENAME TOを追加しました。(David Fetter, Neil)

これまでは、ALTER TABLE ... RENAME TO経由でしか行うことができませんでした。

- CREATE/DROP/RENAME DATABASEが失敗する前に、競合するバックエンドの終了を短時間待機するようにしました。(Tom)

これにより、これらのコマンドが成功する可能性が増えました。

- リプリケーションを目的とした、設定パラメータを使用してトリガとルールをまとめて無効化することができます。(Jan)

これによりリプリケーションシステムは、システムカタログを直接変更することなく、トリガとルールをグループとして無効にすることができます。この動作は`ALTER TABLE`と新しい`session_replication_role`パラメータにより制御されます。

- ユーザ定義型が型修飾子を持つことができるようになりました。(Teodor, Tom)

これによりユーザ定義型は`ssnum(7)`のような修飾子を取ることができます。以前は組み込み型のみが修飾子を持つことができました。

E.71.3.8. ユーティリティコマンド

- スーパーユーザ以外のデータベース所有者が、デフォルトで所有するデータベースに信頼できる手続き言語を追加できるようになりました。(Jeremy Drake)

これはかなり安全ですが、一部の管理者はこの権限を取り上げたいと考えるかもしれません。これは`pg_pltemplate`、`tmpldbacreate`で制御されます。

- セッションの現在のパラメータ設定を今後のセッションにおけるデフォルトとして使用できます。(Tom)

これは`CREATE/ALTER FUNCTION`、`ALTER DATABASE`、`ALTER ROLE`で`SET ... FROM CURRENT`を付けることで行われます。

- 新しいコマンド`DISCARD ALL`、`DISCARD PLANS`、`DISCARD TEMPORARY`、`CLOSE ALL`、`DEALLOCATE ALL`を実装しました。(Marko Kreen, Neil)

これらのコマンドは単にデータベースセッションをその初期状態に再設定します。特に接続プールソフトウェアで有用です。

- `CLUSTER`をMVCCセーフにしました。(Heikki Linnakangas)

これまでは`CLUSTER`は、不要と確定したすべてのタブルを、たとえMVCCの可視性規則の下でそのタブルを参照することができるはずのトランザクションが他にあったとしても破棄しました。

- `CLUSTER`に新しい構文、`CLUSTER table USING index`を追加しました。(Holger Schurig)

古い`CLUSTER`構文もまだサポートされますが、新しい構文の方が論理的であるとみなされています。

- 複雑な計画をより正確に表示できるように`EXPLAIN`を修正しました。(Tom)

副計画出力への参照が、複雑な場合`?columnN?`を使わずに、常に正しく表示されるようになりました。

- ユーザが削除された時に報告される情報量を制限しました。(Alvaro)

これまでは、多くのオブジェクトを所有するユーザを削除した（または削除しようとした）時、そのオブジェクトすべてを列挙する、多くの`NOTICE`または`ERROR`メッセージが生成されました。これは一部のクライアントアプリケーションで問題になりました。メッセージ長は制限されるようになりましたが、サーバログにはまだすべてのリストが送信されます。

E.71.3.9. データ型

- SQL/XML標準をサポートしました。これに伴い、新しい演算子とXMLデータ型が含まれました。(Nikolay Samokhvalov, Pavel Stehule, Peter)

- 列挙データ型 (`ENUM`)。(Tom Dunstan)

この機能は、小さく、固定の集合のみの値を許すフィールドに対して簡便なサポートを提供します。`ENUM`型の作成例は`CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy')`です。

- 汎用一意識別子 (`UUID`) データ型。(Gevik Babakhani, Neil)

これはRFC 4122にほぼ合致します。

- `MONEY` データ型を64ビットに広げました。(D'Arcy Cain)

これは `MONEY` 型の値がサポートする範囲を大きく増やします。

- 一貫してInfinity and NAN (Not A Number)を扱うことができるように`float4/float8`を修正しました。(Bruce)

以前のコードではInfinity とオーバーフロー条件の区別に一貫性がありませんでした。

- `boolean` 値の入力の際、前後の空白文字が許されます。(Neil)

- 区切り文字として数字と小文字を使用することを`COPY`は防止するようになりました。(Tom)

E.71.3.10. 関数

- 新しい正規表現関数 `regexp_matches()`、`regexp_split_to_array()`、`regexp_split_to_table()` を追加しました。(Jeremy Drake, Neil)
これらの関数は正規表現副式の抽出を提供し、POSIX正規表現を使用した文字列の分割を行うことができます。
- ラージオブジェクトの切り詰めを行う `lo_truncate()` を追加しました。(Kris Jurka)
- `float8` データ型に対する `width_bucket()` を実装しました。(Neil)
- 現在のトランザクションで収集された統計情報のスナップショットを破棄する `pg_stat_clear_snapshot()` を追加しました。(Tom)
トランザクションにおける最初の統計情報の要求はトランザクション期間で変更されない統計情報のスナップショットを取ることで、この関数によりこのスナップショットを破棄し、次の統計情報問い合わせで新しいスナップショットを読み込むことができます。これは特に、単一トランザクション内に制限されるPL/pgSQL関数で有用です。
- `EXTRACT()` および `date_part()` に `isodow` オプションを追加しました。(Bruce)
これは日曜を7とした週内日数を返します (`dow` は日曜を0とします)。
- `to_char()`、`to_date()`、`to_timestamp()` に `ID` (ISO週内日数) および `IDDD` (ISO年内日数) 書式コードを追加しました。(Brendan Jurd)
- `to_timestamp()` および `to_date()` が、長さが可変となる可能性があるフィールドに対して `TM` (trim) オプションを前提とするようにしました。
これはOracleの動作と合致します。
- `to_date()/to_timestamp()` の `D` (非ISO週内日数) の変換における一つ違いのエラーを修正しました。(Bruce)
- `setseed()` を、無意味な整数値ではなく `void` を返すようにしました。(Neil)
- `NUMERIC` 用のハッシュ関数を追加しました。(Neil)
これにより `NUMERIC` 列でハッシュインデックスとハッシュを基にした計画が利用できるようになりました。
- `LIKE/ILIKE` の効率を、特にUTF-8のようなマルチバイト文字セットで、向上させました。(Andrew, Itagaki Takahiro)
- `currtid()` が対象テーブルについての `SELECT` を必要とするようにしました。(Tom)
- 活動中のトランザクションIDを問い合わせるための `txid_*` 関数を複数追加しました。(Jan)
これは各種リプリケーションソリューションで有用です。

E.71.3.11. PL/pgSQLサーバサイド言語

- `FETCH` での方向指定を含む、スクロール可能なカーソルのサポートを追加しました。(Pavel Stehule)
- バックエンドの `FETCH` コマンドと合わせ、PL/pgSQLの `FETCH` にて `FROM` の代わりに `IN` を使用可能にしました。(Pavel Stehule)
- PL/pgSQLに `MOVE` を追加しました。(Magnus, Pavel Stehule, Neil)
- `RETURN QUERY` を実装しました。(Pavel Stehule, Neil)
これは、PL/pgSQLの集合を返す関数で問い合わせの結果を返したい場合向けに簡便な構文を追加します。 `RETURN QUERY` は、`RETURN NEXT` をループさせるよりも簡単で効率的です。
- 関数のパラメータ名を関数名で修飾できるようにしました。(Tom)
例えば、`myfunc.myvar` です。これは特に、列名と名前が一致する可能性がある変数を問い合わせ内で指定する場合に役に立ちます。
- ブロックレベルの変数修飾が適切に動作するようにしました。(Tom)
これまでは、より外側のブロックラベルがより内側のレコードや行参照の認識に意図せず干渉していました。
- `FOR` ループの `STEP` 値の必要条件を厳密にしました。(Tom)
非正の `STEP` 値を許しません。また、ループのオーバーフローを取り扱います。
- 構文エラーで報告する発生位置の精度を高めました。(Tom)

E.71.3.12. 他のサーバサイド言語

- PL/Perl `spi_prepare()` への型名引数が `pg_type` 内の名前の他にデータ型の別名を取ることができるようになりました。(Andrew)
- PL/Python `plpy.prepare()` への型名引数が `pg_type` 内の名前の他にデータ型の別名を取ることができるようになりました。(Andrew)
- PL/Tcl `spi_prepare` への型名引数が `pg_type` 内の名前の他にデータ型の別名を取ることができるようになりました。(Andrew)
- PL/PythonUがPython 2.5でコンパイルできるようになりました。(Marko Kreen)
- Python版と互換性のある、本当のPL/Python論理型をサポートします。(Marko Kreen)
- スレッドが有効な `libtcl` を使用した場合にバックエンド内で複数のスレッドが生成されるPL/Tclの問題を修正しました。(Steve Marshall, Paul Bayer, Doug Knight)

これはまったく不愉快な事象を引き起こしました。

E.71.3.13. [psql](#)

- `%d` の出力で無効なトリガを別途列挙します。(Brendan Jurd)
- パターン指定の `%d` で、常に `%` そのものに一致します。(Tom)
- `%da` 出力にて集約の戻り値型を示します。(Greg Sabino Mullane)
- `%df+` の出力に関数の揮発性状態を追加しました。(Neil)
- `%prompt` 機能を追加しました。(Chad Wagner)
- `%pset`、`%t`、`%x` が単に切替えるだけでなく `on` または `off` を指定できるようになりました。(Chad Wagner)
- `%sleep` 機能を追加しました。(Jan)
- `%copy` で `%timing` 出力を有効にしました。(Andrew)
- Windowsにおいて `%timing` の解像度を向上させました。(Itagaki Takahiro)
- 各バックスラッシュコマンドの後 `%o` 出力をフラッシュします。(Tom)
- `-f` で指定された入力ファイルの読み取り時に正確にエラーを検知し報告します。(Peter)
- `-u` オプションを削除しました (これはかなり前から廃止予定でした)。(Tom)

E.71.3.14. [pg_dump](#)

- `pg_dumpall` に `--tablespaces-only` および `--roles-only` オプションを追加しました。(Dave Page)
 - `pg_dumpall` に出力ファイルオプションを追加しました。(Dave Page)
- これは主に、`pg_dump` 子プロセスの出力リダイレクションが動作しない、Windowsで有用です。
- `pg_dumpall` が初期接続用にデフォルトの `template1` 以外のデータベース名を受け付けられるようになりました。(Dave Page)
 - `-n` および `-t` スイッチにて、常に `%` そのものに一致します。(Tom)
 - データベースが数千のオブジェクトを持つ場合の性能を向上しました。(Tom)
 - `-u` オプションを削除しました (これはかなり前から廃止予定でした)。(Tom)

E.71.3.15. 他のクライアントアプリケーション

- `initdb` で、`pg_xlog` ディレクトリの場所を指定できるようになりました。(Euler Taveira de Oliveira)
 - サポートされるオペレーティングシステムでは、`pg_regress` 内でサーバコアファイルの生成が有効になりました。(Andrew)
 - `pg_ctl` に `-t` (時間切れ) パラメータを追加しました。(Bruce)
- これは、サーバ起動時または停止時に `pg_ctl` が待機する時間を制御します。これまでの待機時間は固定で60秒でした。
- サーバコアダンプの生成を制御するために `pg_ctl` のオプションを追加しました。(Andrew)
 - Control-Cで `clusterdb`、`reindexdb`、`vacuumdb` をキャンセルできるようにしました。(Itagaki Takahiro, Magnus)
 - `createdb`、`createuser`、`dropdb`、`dropuser` でコマンドタグの出力を抑制します。(Peter)

--quiet オプションは無視され、8.4で削除される予定です。全データベースを実行している場合の進行メッセージは、実際のエラーではありませんので、標準エラー出力ではなく標準出力に出力されるようになりました。

E.71.3.16. libpq

- PQsetdbLogin() の dbName パラメータが等号記号を含む時 conninfo 文字列として解釈されます。(Andrew)
まだ PQsetdbLogin() を使用しているクライアントプログラムで conninfo 文字列を使用できるようになります。
- 大域的なSSL設定ファイルをサポートします。(Victor Wagner)
- SSLハードウェアキーを制御する PGSSLKEY 環境変数を追加しました。(Victor Wagner)
- ラージオブジェクトを切り詰めるための lo_truncate() を追加しました。(Kris Jurka)
- サーバがパスワードを必要としているが、何も与えられていない場合に真を返す PQconnectionNeedsPassword() を追加しました。(Joe Conway, Tom)
接続試行に失敗した後に真を返した場合、クライアントアプリケーションはユーザにパスワードの入力を促すべきです。これまでのアプリケーションでは、パスワードが必要かどうかを決めるために、特定エラーメッセージの検査を行う必要がありました。この方法は推奨されなくなりました。
- 与えたパスワードが実際に使用された場合に真を返す PQconnectionUsedPassword() を追加しました。(Joe Conway, Tom)
ユーザが与えたパスワードが実際に有効だったかどうかの確認が重要となる一部のセキュリティコンテキストで、これは役に立ちます。

E.71.3.17. ecpg

- V3フロントエンド/バックエンドプロトコルを使用します。(Michael)
これは、サーバサイドの準備済み文のサポートを追加します。
- Windowsでpthreadではなくネイティブのスレッドを使用します。(Magnus)
- ecpglibのスレッド安全性を向上しました。(Itagaki Takahiro)
- ecpgライブラリが必要なAPIシンボルのみをエクスポートするようにしました。(Michael)

E.71.3.18. Windowsへの移植

- Microsoft Visual C++を使用したPostgreSQL配布物全体のコンパイルを可能にしました。(Magnus and others)
これによりWindowsを基に開発する開発者が慣れた開発ツール、デバッグツールを使用できるようになりました。また、Visual C++で作成されたWindows実行形式ファイルは、他のツールセットで作成されたファイルより優れた安定性と性能を持ちます。クライアントのみを構築するVisual C++のスク립トは削除されました。
- 多くの子プロセスを持つ時のpostmasterのメモリ使用量を劇的に減少させました。(Magnus)
- リグレーション試験が管理者ユーザで起動できるようになりました。(Magnus)
- ネイティブな共有メモリの実装を追加しました。(Magnus)

E.71.3.19. サーバプログラミングインタフェース(SPI)

- SPIにカーソル関連の機能を追加しました。(Pavel Stehule)
カーソル関連の計画作成オプションにアクセスすることができます。また、FETCH/MOVE処理を追加しました。
- SPI_execute経由でカーソルコマンドを実行することができます。(Tom)
SPI_ERROR_CURSORはまだ存在しますが、何も返しません。
- SPI計画ポインタはvoid *ではなく SPIPlanPtrとして宣言されます。(Tom)
これはアプリケーションコードを壊しませんが、単純なプログラムミスを見付けやすくするために変更することを勧めます。

E.71.3.20. 構築オプション

- コードプロファイリング (gccでのみ動作) を有効にするためにconfigureのオプションに--enable-profilingを追加しました。(Korry Douglas and Nikhil Sontakke)

オペレーティングシステムの時間帯データベースを使用するためにconfigureのオプションに`--with-system-tzdata`を追加しました。(Peter)

- pg_configプログラムがPATHの先頭がないPostgreSQLインストレーションに対して拡張を構築できるようにPGXSを修正しました。(Tom)
 - SGML文書の構築時の`gmake draft`をサポートします。(Bruce)
- `draft`を使わない場合、インデックス更新を確実にするために必要ならば、文書の構築が2回行われます。

E.71.3.21. ソースコード

- サードパーティにより`DDLIMPORT`が宣言されたインクルードファイル(Tclのような)との競合を防ぐために、`DDLIMPORT`マクロ名を`PGDDLIMPORT`に変更しました。(Magnus)
- データ型を跨った比較を含む問い合わせの計画作成を改良するために"演算子族"を作成しました。(Tom)
- 問い合わせを満たすことができるものが存在しなかったことを通知できるようにGINの`extractQuery()` APIを更新しました。(Teodor)
- `NAMEDATALEN`定義を`postgres_ext.h`から`pg_config_manual.h`に移動しました。(Peter)
- すべてのプラットフォームで`strncpy()` および `strlcat()` を提供します。 また、エラーが起こりがちの `strncpy()`、`strncat()` などの使用を置き換えました。(Peter)
- 外部のプラグインによりプランナを監視(置き換えることすら可能)させる、および、仮定の状況に対する計画を作成するためのフックを作成しました。(Gurjeet Singh, Tom)
- プラグインにプランナの結合検索順の位置を上書きさせるための`join_search_hook`関数変数を作成しました。(Julius Stroffek)
- ルネサスM32Rプロセッサ向けの`tas()` サポートを追加しました。(Kazuhiro Inaoka)
- `quote_identifier()` および `pg_dump` は、文法に従い保持されないキーワードに引用符を付けません。(Tom)
- `sign_dscale`ワードが重みより前に来るように、`NUMERIC` データ型のディスク上の表現を変更しました。
- Darwin ≥ 6.0 、つまりOS X 10.2以上でPOSIXではなくSYSVセマフォを使用します。(Chris Marcellino)
- [acronym](#) および [NFS](#) に関する節を文書に追加しました。(Bruce)
- "PostgreSQL"の別名として受け入れられるものとして"Postgres"が文書化されました。(Peter)
- データベースサーバが停止している場合のサーバなりすまし防止に関する文書を追加しました。(Bruce)

E.71.3.22. Contrib

- `contrib README`の内容を主PostgreSQL文書に移しました。(Albert Cervera i Areny)
 - 低レベルなページ検査を行う`contrib/pageinspect`モジュールを追加しました。(Simon, Heikki)
 - ウォームスタンバイ操作を制御する`contrib/pg_standby`モジュールを追加しました。(Simon)
 - OSSP UUIDライブラリを使用してUUID値を生成する`contrib/uuid-oss`モジュールを追加しました。(Peter)
- `configure --with-oss-uuid`を使用して有効にします。これは新しく組込まれたUUID型を使用します。
- テキスト検索辞書テンプレートとパーサを追加するためのサンプルを提供する`contrib/dict_int`、`contrib/dict_xsyn`、`contrib/test_parser`モジュールを追加しました。(Sergey Karpov)
 - `contrib/pgbench`でフィルファクタを設定できるようにしました。(Pavan Deolasee)
 - `contrib/pgbench -1`にタイムスタンプを追加しました。(Greg Smith)
 - `contrib/pgbuffercache`に使用カウントに関する統計情報を追加しました。(Greg Smith)
 - `contrib/hstore`にGINサポートを追加しました。(Teodor)
 - `contrib/pg_trgm`にGINサポートを追加しました。(Guillaume Smet, Teodor)
 - `contrib/start-scripts`のOS/X起動スクリプトを更新しました。(Mark Cotner, David Fetter)
 - `pgrowlocks()` と `dblink_get_pkey()` を対象テーブルにSELECT権限を持つユーザに制限しました。(Tom)
 - `contrib/pgstattuple`関数をスーパーユーザに制限しました。(Tom)

- contrib/xml2は推奨しません。8.4で削除される予定です。

PostgreSQLのコアにある新しいXMLサポートがこのモジュールに取って代わりました。

[前のページ](#)

リリース8.3.1

[ホーム](#)
[上に戻る](#)

[次のページ](#)

リリース8.2.23

E.72. リリース8.2.23

リリース日: 2011-12-05

このリリースは8.2.22に対し、各種の不具合を修正したものです。 8.2メジャーリリースにおける新機能については[項E.95](#)を参照してください。

これが8.2.x系列における最後のPostgreSQLリリースとなる予定です。 早めにより新しい系列のリリースに更新することを勧めます。

E.72.1. バージョン8.2.23への移行

8.2.Xからの移行ではダンプ/リストアは不要です。

しかし、`information_schema.referential_constraints`ビュー定義において長期に渡って潜在したエラーが見つかりました。このビューからの正確な結果に依存している場合、後述の変更点の最初で説明する通りその定義を置き換えなければなりません。

また8.2.14以前のバージョンからアップデートする場合は8.2.14のリリースノートを参照してください。

E.72.2. 変更点

- `information_schema.referential_constraints`ビューの不具合を修正しました。(Tom Lane)

このビューでは、外部キーと依存するプライマリキーまたは一意キー制約との対応付けについて十分な注意を行っていませんでした。 このため、外部キー制約がまったく表示されない、何回も表示される、あるいは、実際の依存する制約とは異なる制約に依存するものと言い張るという結果になることがありました。

このビュー定義は`initdb`によってインストールされますので、ただアップグレードしただけでは問題は解消されません。既存のインストールでこの問題を解消させなければならない場合、（スーパーユーザとして）`information_schema`スキーマを削除して、`SHAREDIR/information_schema.sql`を元に再作成することができます。（`SHAREDIR`の場所が分からない場合は`pg_config --sharedir`を実行してください。） これは修正対象のデータベースそれぞれで繰り返さなければなりません。

- `CREATE TABLE dest AS SELECT * FROM src`または`INSERT INTO dest SELECT * FROM src`時のTOAST関連のデータの破損を修正しました。(Tom Lane)

テーブルが`ALTER TABLE ADD COLUMN`によって変更された場合、データを逐語的に別のテーブルにコピーしようとした時に特定の境界条件で破損した結果になることがありました。 この問題は、8.4以降において、この構文においてのみ明確に発生し得たものですが、同じ不具合を引き起こす他のコードパスがある場合に備えて、より過去のバージョンにおいてもパッチを適用しました。

- 無効な`syscache`項目からTOASTテーブルにアクセスする時の競合条件を修正しました。(Tom Lane)

典型的な兆候は、`"missing chunk number 0 for toast value NNNNN in pg_toast_2619"`のような一時的なエラーです。 ここで挙げられるTOASTテーブルは常にシステムカタログに属します。

- `money`型の入出力におけるロケールのサポートを改良しました。(Tom Lane)

すべての標準`lc_monetary`書式付けオプションをサポートしていないという点以外にも、入力関数と出力関数に一貫性がなかったため、ダンプされた`money`値では再度読み取ることができないロケールがありました。

- `transform_null_equals`が`CASE foo WHEN NULL ...`式に影響を与えないようにしました。(Heikki Linnakangas)

`transform_null_equals`は、`CASE`構文によって内部的に生成された等価な検査ではなく、直接ユーザによって書かれた`foo = NULL`に影響を与えることのみを前提としています。

- 自己参照外部キーをより良くサポートするために、外部キートリガの生成順序を変更しました。(Tom Lane)

カスケードしている外部キーが自身のテーブルを参照する場合、1行の更新が1つのイベントで`ON UPDATE`トリガと`CHECK`トリガの両方を発行します。 `ON UPDATE`トリガを最初に行うなければなりません。 そうしないと、`CHECK`は最終的ではない行の状態を検査することになり、不適切なエラーが発生する可能性があります。 しかしこれらのトリガの発行順序はその名前によって決まります。 トリガは`"RI_ConstraintTrigger_NNNN"`という規約に従った名前前で自動生成さ

れますので、一般的には作成順序によって決まります。適切な修正にはこの規約の変更が必要です。9.2で変更する予定ですが、既存のリリースで変更することには危険があるように思われます。このため、このパッチはトリガの作成順序を変更するだけです。この種のエラーが起きたユーザは、トリガを正しい順序にするために、外部キー制約を削除し再作成しなければなりません。

- `psql`のコマンド履歴内で空行を含むコマンドを保持します。(Robert Haas)

これまでの動作では、例えば文字列リテラル内の空行が削除された場合に問題が起きました。

- PL/Perlの構築時に好みのバージョンの`xsubpp`を使用します。オペレーティングシステムのメインコピーである必要はありません。(David Wheeler、Alex Hunsaker)
- `pgstattindex()`において問い合わせキャンセル割込みを適切に受け付けます。(Robert Haas)
- VPATH構築ですべてのサーバヘッダファイルが適切にインストールされることを確実にしました。(Peter Eisentraut)
- 冗長エラーメッセージ内で報告されるファイル名を短くしました。(Peter Eisentraut)

通常の構築では常にCファイルの名前だけがエラーメッセージ呼び出しの中で報告されました。しかしこれまでVPATH構築では絶対パス名が報告されました。

- 中央アメリカ用のWindowsの時間帯名の解釈を修正しました。(Tom Lane)

通常中央アメリカではまったく夏時間が認められませんので、"Central America Standard Time"を`CST6CDT`ではなく`CST6`に対応付けしました。

- 時間帯データファイルを`tzdata release 2011n`に更新しました。ブラジル、キューバ、フィジィ、パレスチナ、ロシア、サモアにおける夏時間規則の変更、アラスカ、イギリス領東アフリカの歴史的な修正が含まれています。

E.73. リリース8.2.22

リリース日: 2011-09-26

このリリースは8.2.21に対し、各種の不具合を修正したものです。 8.2メジャーリリースにおける新機能については[項E.95](#)を参照してください。

PostgreSQLコミュニティは2011年12月に8.2.Xリリースの更新の公開を停止します。 早めにより新しいブランチへの更新を推奨します。

E.73.1. バージョン8.2.22への移行

8.2.Xからの移行ではダンプ/リストアは不要です。 しかし8.2.14以前のバージョンからアップデートする場合は8.2.14のリリースノートを参照してください。

E.73.2. 変更点

- GiSTインデックスページ分割処理における複数の不具合を修正しました。(Heikki Linnakangas)
発生する可能性は小さいのですが、インデックスの破損を招く可能性がありました。
- `ANALYZE` およびSJIS-2004符号化変換においてメモリの最後を超えてアクセスする可能性を修正しました。(Noah Misch)
これは非常に可能性が低いサーバクラッシュ状況を修正します。
- `relcache`初期ファイルの無効化における競合状態を修正しました。(Tom Lane)
新しいバックエンドプロセスが無効な初期ファイルを読み取ろうとするが、データが無効であることを通知する`inval`メッセージを取り損なう期間がありました。 この結果カタログアクセスにおいて、起動処理の後の"could not read block 0 in file ..."といった奇妙な失敗が起こります。
- GiSTインデックススキャン終了時のメモリリークを修正しました。(Tom Lane)
すでに多くの行を持つテーブルに対して新たなGiSTを基にした排他制約を作成する際の検証など、多くの分割GiSTインデックススキャンを行うコマンドが一時的にこのリークの原因となる大容量のメモリを求めることがあり得ました。
- 大規模かつ非可逆なビットマップを構築する時の性能問題を修正しました。(Tom Lane)
- 配列作成、経路作成関数のパッド用バイトを確実にゼロにするように修正しました。(Tom Lane)
これにより、プランナが意味的に等しい定数を等しくないとみなし、最適化を悪化させる一部の状況を防ぎます。
- WAL再生を壊すgcc 4.5.0の不具合を回避します。(Tom Lane)
サーバクラッシュ後にコミットされたトランザクションの損失をもたらす可能性がありました。
- ビュー内の`VALUES`についてのダンプ不具合を修正しました。(Tom Lane)
- シーケンスに対する`SELECT FOR UPDATE/SHARE`を許しません。(Tom Lane)
この操作は想定通りに動作せず、また、失敗をもたらすことがあり得ました。
- ハッシュテーブルのサイズ計算時の整数オーバーフローから保護します。(Tom Lane)
- "peer" 認証に関する証明書制御メッセージの使用における移植性不具合を修正しました。(Tom Lane)
- `pg_srand48` シード初期化における記述ミスを修正しました。(Andres Freund)
このため、提供されたシードの全ビットを使用すると失敗する可能性がありました。 この関数はほとんどのプラットフォームで使用されていません (`srandom` を持たないプラットフォームのみです) ので、想定よりもランダム性が小さいシードによるセキュリティ露見の可能性はとにかく最小だったと思われます。
- `LIMIT` と `OFFSET` 値の合計が 2^{63} を超えた場合の整数オーバーフローを防止します。(Heikki Linnakangas)

- `generate_series()` の `int4` 版と `int8` 版にオーバーフロー検査を追加しました。(Robert Haas)
- `to_char()` における末尾のゼロ除去を修正しました。(Marti Raudsepp)
小数点の後の桁位置がない `FM` 付きの書式において、小数点より左のゼロが正しく除去できていませんでした。
- 2^{63} 近辺の入力に対するオーバーフローを防ぐように `pg_size_pretty()` を修正しました。(Tom Lane)
- 異なるファイルから `COPY` している時の、`psql` のスクリプトファイル行番号の計数処理を修正しました。(Tom Lane)
- `standard_conforming_strings` に合わせて、`pg_restore` の直接データベースモードを修正しました。(Tom Lane)
`pg_restore` が、`standard_conforming_strings` が `on` に設定されたアーカイブファイルからデータベースサーバに直接リストアする時に間違ったコマンドを発行する可能性があります。
- `libpq` の LDAP サービス検索コードにおける `write-past-buffer-end` とメモリリークを修正しました。(Albe Laurenz)
- `libpq` において、非ブロッキング I/O と SSL 接続を使用する場合の失敗を防止します。(Martin Pihlak, Tom Lane)
- `libpq` の接続開始期間のエラーの取扱いを改良しました。(Tom Lane)
具体的には、SSL 接続開始期間の `fork()` 失敗についてのサーバ報告への応答がより健全になりました。
- `ecpglib` が `double` 値を 15 桁の精度で書き込むようにしました。(Akira Kurosawa)
- `blowfish` の符号付き文字に関する不具合 (CVE-2011-2483) に対する上流の修正を適用しました。(Tom Lane)
`contrib/pg_crypto` の `blowfish` 暗号化コードは、`char` が符号付きであるプラットフォーム(ほとんどのプラットフォーム)において間違った結果を生成し、暗号化されたパスワードが本来より脆弱になりました。
- `contrib/seg` におけるメモリリークを修正しました。(Heikki Linnakangas)
- 空のインデックスに対して一貫性を持った結果を生成するように `pgstatindex()` を修正しました。(Tom Lane)
- `perl 5.14` を用いたビルドを可能にしました。(Alex Hunsaker)
- システム関数の存在を検出するための `configure` スクリプトの方法を更新しました。(Tom Lane)
8.3 と 8.2 で使用している `autoconf` のバージョンは、リンク時の最適化を行うコンパイラによってだまされることがあり得ました。
- 空白を含むファイルパスでのビルドとインストールに関連した問題を修正しました。(Tom Lane)
- 時間帯データファイルを `tzdata release 2011i` に更新しました。カナダ、エジプト、ロシア、サモア、南スーダンにおける夏時間規則の変更が含まれています。

E.74. リリース8.2.21

リリース日: 2011-04-18

このリリースは8.2.20に対し、各種の不具合を修正したものです。 8.2メジャーリリースにおける新機能については[項E.95](#)を参照してください。

E.74.1. バージョン8.2.21への移行

8.2.Xからの移行ではダンプ/リストアは不要です。 しかし8.2.14以前のバージョンからアップデートする場合は8.2.14のリリースノートを参照してください。

E.74.2. 変更点

- カタログキャッシュの初期化中のデッドロックの可能性を取り除きました。(Nikhil Sontakke)

一部のキャッシュを読み取るコードが、インデックスのカタログをロックする前にシステムインデックスに対して共有ロックを獲得する場合があります。 これは他でより標準的な順序で排他ロックを獲得しようとする処理とデッドロックする可能性があります。

- 対象タプルに対する同時更新があった時のBEFORE ROW UPDATEトリガ処理における、不正な領域を指し示すポインタ問題を修正しました。(Tom Lane)

この不具合は、UPDATE RETURNING ctidを実行しようとした時に、断続的に"cannot extract system attribute from virtual tuple"で失敗するという結果で気付きました。 更新したタプルに対して不正なインデックス項目が生成されてしまうなど、より深刻なエラーとなる可能性がごくわずかながら存在します。

- 遅延トリガイベントを待たせているテーブルに対するDROP TABLEを許可しません。(Tom Lane)

以前はDROPが進み、トリガが最終的に発行された時に"could not open relation with OID nnn"エラーをもたらしました。

- PL/Pythonの部分配列化に関するメモリリークを修正しました。(Daniel Popowich)
- TOCファイル内の(1KB以上の)長い行に対処するようにpg_restoreを修正しました。(Tom Lane)
- コンパイラの過度に積極的な最適化によるゼロ除算が原因のクラッシュに対して、より多くの対策を取りました。(Aurelien Jarno)
- MIPS上のFreeBSDとOpenBSDでdlopen()の使用をサポートしました。(Tom Lane)

これらのシステムではこのシステム関数はMIPSハードウェアで利用できないという仮定を直接書き込んだコードがありました。 より最近のバージョンでは利用できますので、コンパイル時の試験を使用するように変更しました。

- HP-UXにおけるコンパイル失敗を修正しました。(Heikki Linnakangas)
- Cygwinにおいてpg_regressで使用するパス区切り文字を修正しました。(Andrew Dunstan)
- 時間帯データファイルをtzdata release 2011fに更新しました。 チリ、キューバ、フォークランド諸島、モロッコ、サモア、トルコにおける夏時間規則の変更、南オーストラリア、アラスカ、ハワイの歴史的な修正が含まれています。

E.75. リリース8.2.20

リリース日: 2011-01-31

このリリースは8.2.19に対し、各種の不具合を修正したものです。 8.2メジャーリリースにおける新機能については[項E.95](#)を参照してください。

E.75.1. バージョン8.2.20への移行

8.2.Xからの移行ではダンプ/リストアは不要です。 しかし8.2.14以前のバージョンからアップデートする場合は8.2.14のリリースノートを参照してください。

E.75.2. 変更点

- EXPLAINがCASE式の簡易構文を表示しようとした時の失敗を防止します。(Tom Lane)

CASEのテスト式が定数の場合、プランナはCASEを式表示用コードを混乱させる形式に単純化してしまいました。 その結果"unexpected CASE WHEN clause"というエラーになりました。

- 過去に存在した添字範囲に対する部分配列代入を修正しました。(Tom Lane)

新しく追加される添字と過去に存在した添字の先頭との間に隙間があった場合、コードは古い配列のヌルビットマップからコピーしなければならない項目数を誤計算してしまい、データ破損またはクラッシュを導く可能性がありました。

- プランナにおける、非常に時間が離れた日付値に対する想定外の変換オーバーフローを防止します。(Tom Lane)

date型はtimestamp型で表現可能な範囲よりもより広い日付範囲をサポートします。 しかしプランナは常に問題なくdateからtimestampへの変換が可能であることを仮定していました。

- standard_conforming_stringsが有効な場合のラージオブジェクト (BLOB) 用のpg_restoreのテキスト出力を修正しました。(Tom Lane)

データベースへ直接リストアすることは正しく動作しましたが、pg_restoreがSQLテキスト出力を求め、かつ、standard_conforming_stringsが元のデータベースで有効だった場合、文字列エスケープは正しくありませんでした。

- ... & !(subexpression) | ...を含むtsquery値の間違った解析を修正しました。(Tom Lane)

こうした演算子の組み合わせを含む問い合わせは正しく実行されませんでした。 同じエラーがcontrib/intarrayのquery_int型とcontrib/ltreeのltxtquery型に存在しました。

- query_int型に対するcontrib/intarray入力関数におけるバッファオーバーランを修正しました。(Apple)

この関数の返すアドレスが上書きされる可能性があるという点で、この不具合はセキュリティ問題です。 この問題を報告し修正版を提供していただいたApple Incのセキュリティチームに感謝します。(CVE-2010-4015)

- contrib/segのGiST picksplitアルゴリズムにおける不具合を修正しました。(Alexander Korotkov)

これにより、seg列上のGiSTインデックスにおいて実際に不正な結果になることはありませんが、非常に非効率的な結果になることがあり得ました。 こうしたインデックスがある場合、この更新版をインストールした後にREINDEX処理を行うことを検討してください。(これは過去の更新のcontrib/cubeで修正された不具合と同じです。)

E.76. リリース8.2.19

リリース日: 2010-12-16

このリリースは8.2.18に対し、各種の不具合を修正したものです。 8.2メジャーリリースにおける新機能については[項E.95](#)を参照してください。

E.76.1. バージョン8.2.19への移行

8.2.Xからの移行ではダンプ/リストアは不要です。 しかし8.2.14以前のバージョンからアップデートする場合は8.2.14のリリースノートを参照してください。

E.76.2. 変更点

- Linuxにて`wal_sync_method`のデフォルトが強制的に`fdatasync`になるようにしました。(Tom Lane, Marti Raudsepp)
長い間、Linuxでのデフォルトは実質`fdatasync`でした。 しかし最近のカーネルの変更により、PostgreSQLは代わりに`open_datasync`を選択するようになりました。 この選択は性能向上をまったくもたらず、特定のシステム、特に`data=journal` マウントオプションを使用した`ext4`では、完全に失敗してしまいます。
- GINインデックスに対するWAL再生ロジックにおける分類不具合を修正しました。(Tom Lane)
これは"bad buffer id: 0"という失敗、あるいはレプリケーション中のインデックス内容の破損という結果になり得ました。
- 開始チェックポイントWALレコードが再実行時点と同じWALセグメント内に存在しない場合のベースバックアップからのリカバリを修正しました。(Jeff Davis)
- IA64においてレジスタスタックのオーバーラン検知サポートを追加しました。(Tom Lane)
IA64アーキテクチャには2つのハードウェアスタックがあります。 スタックオーバーラン失敗を完全に防ぐには両方を検査しなければなりません。
- `copyObject()`におけるスタックオーバーフロー検査を追加しました。(Tom Lane)
非常に複雑な問い合わせが与えられた場合のスタックオーバーフローのため、特定のコードパスがクラッシュする可能性があります。
- 一時GiSTインデックスにおけるページ分割検知を修正しました。(Heikki Linnakangas)
例えば挿入が行われた時に一時インデックスをスキャンする開いているカーソルがあると、そのインデックスで"同時実行"ページ分割を持つことができます。 GiSTではこの状況の検知に失敗していました。 このためGiSTはそのカーソルの実行が続く間、間違った結果をもたらすことがありました。
- 複雑なインデックス式を`ANALYZE`する時のメモリリークを防ぎます。(Tom Lane)
- 行全体のVarを使用するインデックスがそのテーブルに依存し続けることを確実にしました。(Tom Lane)
`create index i on t (foo(t.*))`のように宣言されたインデックスはそのテーブルが削除された時に自動的に削除されませんでした。
- 複数の`OUT`パラメータを持つSQL関数を"インライン"化しません。(Tom Lane)
結果の想定行型に関する情報が存在しないためにクラッシュする可能性を防ぎます。
- `ORDER BY`、`LIMIT`、`FOR UPDATE`、`WITH`が`INSERT ... VALUES`の`VALUES`部に付いた場合の動作を正しくしました。(Tom Lane)
- `COALESCE()` 式の定数折りたたみを修正しました。(Tom Lane)
プランナは時々、実際には決して達しない副式の評価を行い、想定外のエラーをもたらす可能性がありました。
- `InhRelation`ノードに関する機能性出力を追加しました。(Tom Lane)

これにより `debug_print_parse` が有効な場合に特定種類の問い合わせが行われた場合の失敗を防止します。

- 点から水平線分への距離計算を正しく修正しました。(Tom Lane)

この不具合は様々な複数の幾何距離測定演算子に影響します。

- 再帰もしくはエラー修復状態にならない"単純な"式についてのPL/pgSQLの扱いを修正しました。(Tom Lane)
- PL/Pythonの集合を返す関数の扱いを修正しました。(Jan Urbanski)

集合結果を生成するイテレータ内でSPI関数呼び出しが失敗しました。

- `contrib/cube` のGiST picksplitアルゴリズムの不具合を修正しました。(Alexander Korotkov)

`cube` 列上のGiSTインデックスにおいて、実際に不正確な結果になることはありませんが、このため相当効率性が劣化する可能性があります。こうしたインデックスがある場合、この更新をインストールした後に `REINDEX` することを検討してください。

- `contrib/dblink` 内で、新しい接続を作成する時を除き、"identifier will be truncated" 注意を出力しないようにしました。(Itagaki Takahiro)
- `contrib/pgcrypto` において公開キーがない場合にコアダンプする可能性を修正しました。(Marti Raudsepp)
- `contrib/xml2` のXPath問い合わせ関数におけるメモリリークを修正しました。(Tom Lane)
- 時間帯データファイルを `tzdata release 2010o` に更新しました。フィジィ、サモアにおける夏時間規則の変更、香港の歴史的な修正が含まれています。

[前のページ](#)
リリース8.2.20

[ホーム](#)
[上に戻る](#)

[次のページ](#)
リリース8.2.18

E.77. リリース8.2.18

リリース日: 2010-10-04

このリリースは8.2.17に対し、各種の不具合を修正したものです。 8.2メジャーリリースにおける新機能については[項E.95](#)を参照してください。

E.77.1. バージョン8.2.18への移行

8.2.Xからの移行ではダンプ/リストアは不要です。 しかし8.2.14以前のバージョンからアップデートする場合は8.2.14のリリースノートを参照してください。

E.77.2. 変更点

- PL/PerlおよびPL/Tclにおいて、呼び出し元のSQLユーザIDごとに別のインタプリタを使用します。(Tom Lane)

この変更は、同じセッション内でその後に異なるSQLユーザIDにより実行されるPerlまたはTclコード（例えばSECURITY DEFINERの内部）を破壊することで発生する可能性があるというセキュリティ問題を防止するものです。 ほとんどのスクリプト用言語は、対象とする関数で呼び出される標準関数や演算子の再定義など、これを行うことができる多くの方法を提供します。 この変更がないと、PerlまたはTcl言語の使用権限を持つ任意のSQLユーザは基本的に、対象の関数の所有者が持つSQL権限でできることを何でもできるようになります。

この変更の欠点はPerlおよびTcl関数間の内部通信がより難しくなるという点です。 逃げ道を提供するために、PL/PerlUおよびPL/TclU関数ではまだセッション当たり1つのインタプリタのみを使用します。 これらのすべての関数は従来よりデータベーススーパーユーザの信頼レベルで実行されますので、これはセキュリティ問題とはみなされません。

信頼された実行を提供することを主張するサードパーティ製の手続き言語は似たようなセキュリティ問題を持つ可能性が大きくあります。 セキュリティが非常に重要な目的のために依存している手続き言語があれば、その作者に連絡を取ることを勧めます。

この問題を指摘いただいたTim Bunceに感謝します(CVE-2010-3433)。

- pg_get_expr() の用途として意図されたシステムカタログ列の一つではない引数で呼び出されることを拒絶することでpg_get_expr() 内のクラッシュする可能性を防止します。(Heikki Linnakangas, Tom Lane)
- Windowsの共有メモリ割り当てコードを修正しました。(Tutomu Yamada, Magnus Hagander)

この不具合により、よく報告された"could not reattach to shared memory"というエラーメッセージがもたらされました。 これは、すこし前により新しいブランチに適用された改修を過去のバージョンに適用したものです。

- Windowsにおいて終了コード128 (ERROR_WAIT_NO_CHILDREN) を致命的でないものと扱います。(Magnus Hagander)

高負荷状況下で、Windowsプロセスはこのエラーコードで起動に失敗することが時々あります。 以前はpostmasterはパニック状態としてこれを扱い、データベース全体を再起動しました。 しかしこれは過度は反応のように思われます。

- UNION ALLのメンバーレシーションの二重スキンの可能性を修正しました。(Tom Lane)
- "cannot handle unplanned sub-select"エラーを修正しました。(Tom Lane)

これは、副選択が他の副選択を含む式に展開される結合の別名参照を含む場合に発生しました。

- まれに報告されたB-tree失敗状況の一部をPANICからERRORを減らしました。 また出力されるエラーメッセージに追加の詳細を加えました。(Tom Lane)

これにより、破損したインデックスに対するシステムの堅牢性が向上したはずです。

- 自動バキューム処理中のクラッシュからshow_session_authorization()を防止します。(Tom Lane)
- 返される行のすべてが実際に同一の行型でない、レコードの集合を返す関数を防御します。(Tom Lane)
- 参照渡し関数の結果をハッシュ化する時に失敗する可能性を修正しました。(Tao Ma, Tom Lane)

ロックファイル (`postmaster.pid`およびソケット用のロックファイル) を書き出す時に、注意してその内容のfsyncします。(Tom Lane)

これがないと、`postmaster`の起動後しばらくしてマシンがクラッシュした場合に、ロックファイルの内容が破損することがあり得ました。これが起きると、このロックファイルを手作業で削除するまで、その後の`postmaster`の起動試行が成功できなくなります。

- 過度に入れ子になった副トランザクションに対してXIDを割り当てる際の再帰を防止します。(Andres Freund, Robert Haas)

元々のコードでは、制限されたスタック領域であった場合にクラッシュすることがあり得ました。

- バックエンド起動の初期にゴミを生成することがあった`log_line_prefix`の%iエスケープを修正しました。(Tom Lane)
- アーカイブ処理が有効な場合、`ALTER TABLE ... SET TABLESPACE`におけるデータ破損の可能性を修正しました。(Jeff Davis)
- `CREATE DATABASE`と`ALTER DATABASE ... SET TABLESPACE`を問い合わせキャンセルにより中断させることができます。(Guillaume Lelarge)
- PL/Pythonにおいて、`PyObject_AsVoidPtr`および`PyObject_FromVoidPtr`からのヌルポインタ結果を防止します。(Peter Eisentraut)
- 削除された列を含むテーブルに対する`contrib/dblink`の処理を改良しました。(Tom Lane)
- `contrib/dblink`において"duplicate connection name"エラー後の接続リークを修正しました。(Itagaki Takahiro)
- 62バイトより長い接続名に関する`contrib/dblink`の処理を正しく修正しました。(Itagaki Takahiro)
- `hstore(text, text)`関数を`contrib/hstore`に追加しました。(Robert Haas)

この関数は、廃止予定となった`=>`演算子に対する推奨の代替です。 将来を見据えたコードが古めのバージョンのサーバでも使用できるようにするために過去にもパッチが適用されました。 このパッチは、`contrib/hstore`のインストール後もしくは特定のデータベースに再インストールされた後にのみ効果を持つことに注意してください。 代わりに手作業で`CREATE FUNCTION`コマンドを実行することを選ぶユーザがいるかもしれません。

- 構築手順や文書を更新して、ソースコードリポジトリがCVSからGitに変更されたことを反映しました。(Magnus Hagander他)
- 時間帯データファイルを`tzdata release 2010l`に更新しました。 エジプト、パレスチナにおける夏時間規則の変更、フィンランドの歴史的な修正が含まれています。

またこの変更では、2つのミクロネシアの時間帯を追加しています。 `Pacific/Chuuk`は`Pacific/Truk`より好まれるようになり（好まれる省略形は`TRUT`ではなく`CHUT`）、`Pacific/Pohnpei`は`Pacific/Ponape`より好まれるようになります。

- Windowsの"N. Central Asia Standard Time"時間帯を`Asia/Almaty`ではなく`Asia/Novosibirsk`に対応付けるようにしました。

MicrosoftはKB976098の時間帯更新からこの時間帯の夏時間の振舞いを変更しました。 `Asia/Novosibirsk`はこの新しい振舞いにより合致しています。

E.78. リリース8.2.17

リリース日: 2010-05-17

このリリースは8.2.16に対し、各種の不具合を修正したものです。 8.2メジャーリリースにおける新機能については[項E.95](#)を参照してください。

E.78.1. バージョン8.2.17への移行

8.2.Xからの移行ではダンプ/リストアは不要です。 しかし8.2.14以前のバージョンからアップデートする場合は8.2.14のリリースノートを参照してください。

E.78.2. 変更点

- `Safe.pm`を使用するのではなく、インタプリタ全体に適用される`opmask`を使用して`plperl`を強制的に制限します。(Tim Bunce, Andrew Dunstan)

最近の開発により`plperl`を信頼できるとする根拠とするには`Safe.pm`が非常に安全ではないことを知らされました。 この変更により、`Safe.pm`の使用がなくなり、常に適用されるopcodeマスクを持つ別のインタプリタを使用するようになります。 この変更の喜ばしい副作用として、`plperl`内で普通の方法でPerlの`strict`プラグマが使用できるようになりました。 また、Perlの`$a`と`$b`がソート関数で想定される動作を行うようになり、関数のコンパイルが非常に高速になりました。(CVE-2010-1169)

- PL/Tclにおいて`pltcl_modules`経由で信頼できないコードが実行されることを防止しました。(Tom)

PL/TclのデータベーステーブルからTclコードを自動読み込みする機能はトロイの木馬攻撃によりつけこまれる可能性があります。 そのテーブルの作成や挿入できるユーザに対する制限がなかったためです。 この変更により、`pltcl_modules`がスーパーユーザにより所有されていない限り、この機能は無効になります。(しかし、このテーブルに対する権限は検査されませんので、安全性が低いモジュールを本当に必要とするインストレーションでは、スーパーユーザ以外に適切な権限を付与することができます。) また、`pltclu`関数を本当に実行しようとしていない限り、制限がない"通常の"Tclにコードが読み込まれることを防止しました。(CVE-2010-1170)

- `relcache`項目の再構築中にキャッシュ再設定メッセージを受け取った場合にクラッシュする可能性を修正しました。(Heikki)

8.2.16で関連する失敗を修正する時に、このエラーが入ってしまいました。

- 非特権ユーザがスーパーユーザのみのパラメータ設定を再設定することを許しません。(Alvaro)

これまで、非特権ユーザが自身に対する`ALTER USER ... RESET ALL`または自身が所有するデータベースに対する`ALTER DATABASE ... RESET ALL`を行った場合、そのユーザまたはデータベースに対する特別なパラメータ設定は、たとえスーパーユーザのみにより変更可能なものとして提供されるものであっても、すべて削除されました。 `ALTER`はそのユーザが変更権限を持つパラメータのみを削除するようになりました。

- `CONTEXT`追加がログ項目になされる時にシャットダウンが発生した場合に、バックエンドのシャットダウン時にクラッシュする可能性を防ぎました。(Tom)

現在のトランザクションがログメッセージを出力する時点に達した時にすでにロールバックされているために文脈出力関数が失敗することがありました。

- `pl/perl`の`ppport.h`を最近のPerlバージョンに更新しました。(Andrew)
- `pl/python`内の雑多なメモリリークを修正しました。(Andreas Freund, Tom)
- 変数展開が自身を参照している際の`psql`における無限再帰を防止します。(Tom)
- `%copy (select ...)`内のドットの前後に空白を追加しないように、`psql`の`%copy`を修正しました。(Tom)

数値リテラルにおいて10進数の小数点の前後に空白が追加された場合、構文エラーが発生します。

- `contrib/pgstattuple`関数がキャンセル割り込みに即座に応答することを確実にしました。(Tatsuhito Kasahara)
- サーバ起動時に、既存の共有メモリセグメントに対して`shmget()`が`EINVAL`を返す状況を適切に扱うようにし

た。(Tom)

この振舞いはOS Xを含むBSD派生のカーネルにおいて観測されました。共有メモリ要求サイズが大きすぎるという警告の、完全に間違った起動失敗という結果になりました。

- Windowsにおけるsysloggerプロセス内のクラッシュする可能性を防ぎます。(Heikki)
- Windowsレジストリにおいて不完全な時間帯情報をより堅牢に扱います。(Magnus)
- Windowsにおいて既知の時間帯名称の集合を更新しました。(Magnus)
- 時間帯データファイルをtzdata release 2010jに更新しました。アルゼンチン、南極におけるオーストラリア、バンラデシュ、メキシコ、モロッコ、パキスタン、パレスチナ、ロシア、シリア、チュニジアにおける夏時間規則の変更、および、台湾の歴史的な修正が含まれます。

また、デフォルトの時間帯省略形の集合にPKST(パキスタン夏時間)を追加しました。

[前のページ](#)

リリース8.2.18

[ホーム](#)

[上に戻る](#)

[次のページ](#)

リリース8.2.16

E.79. リリース8.2.16

リリース日: 2010-03-15

このリリースは8.2.15に対し、各種の不具合を修正したものです。 8.2メジャーリリースにおける新機能については[項E.95](#)を参照してください。

E.79.1. バージョン8.2.16への移行

8.2.Xからの移行ではダンプ/リストアは不要です。 しかし8.2.14以前のバージョンからアップデートする場合は8.2.14のリリースノートを参照してください。

E.79.2. 変更点

- SSL接続においてセッションキーの再調停をどの程度の頻度で行うかを制御する`ssl_renegotiation_limit`設定パラメータを新しく追加しました。(Magnus)

これをゼロに設定することで、再調停を完全に無効にすることができます。 これは壊れたSSLライブラリを使用している場合に必要になる可能性があります。 具体的には一部のベンダは再調停の試行を失敗させるCVE-2009-3555に対する一時しのぎのパッチを提供しています。

- バックエンド起動時のデッドロックの可能性を修正しました。(Tom)
- `relcache`の再読み込みがきれいに行われている間にエラーを扱わないことが原因でクラッシュする可能性を修正しました。(Tom)
- 副トランザクション起動時における失敗からのリカバリを試行する時にクラッシュする可能性を修正しました。(Tom)
- セーブポイントの使用およびサーバの符号化方式と異なるクライアント符号化方式に関連したサーバのメモリリークを修正しました。(Tom)
- GiSTインデックスページ分割のリカバリ終了時の整理において生成される不正なWALデータを修正しました。(Yoichi Hirai)

これはインデックスの破損という結果となり、WAL再生中にエラーが発生します。 もし運が悪ければ、不完全のGiST挿入に成功した後にリカバリ終了時の整理が行われている時にクラッシュしてしまいます。

- `bit`型の`substring()`では、いかなる負の長さも"文字列の残りすべて"を意味するようにしました。(Tom)
- これまでのコードでは-1のみを上のように扱い、他の負の値についてはクラッシュする可能性がある無効な結果値を生成しました。(CVE-2010-0442)
- 出力ビット幅が指定された整数よりも広く、それが8ビットの倍数以外である場合、先頭の端数のバイトを正確に扱うように整数からビット列への変換を修正しました。(Tom)
- 正規表現マッチが異常なまでに低速になる一部の状況を修正しました。(Tom)
- 終了位置が正確にセグメント境界であった場合、次のWALセグメント名を報告するようにバックアップ履歴ファイル内の`STOP WAL LOCATION`を修正しました。(Itagaki Takahiro)
- 一時ファイルが漏洩する一部の状況を修正しました。(Heikki)

これは前のマイナーリリースでもたらされた問題を修正します。 失敗する状況の1つは集合を返す`plpgsql`関数が他の関数の例外ハンドラ内で呼び出された場合です。

- 論理値変数の場合の制約による排他処理を改良しました。 具体的には、"`bool_column = false`"制約を持つパーティションを除外することができるようになりました。(Tom)
- `pg_hba.conf`とその関連ファイルを読み取る時、`@`が引用符記号の内部にある場合に`@something`をファイル包含要求と扱わないようにしました。 また、`@`自身をファイル包含要求として扱いません。(Tom)

ロールまたはデータベース名が`@`から始まる場合の一定しない動作を防止します。 空白文字を含むパス名のファイルを包含する必要がある場合、まだ実現させることはできませんが、式全体を引用符でくくるのではなく`@"/path to/file"`と記述

しなければなりません。

- ディレクトリが`pg_hba.conf`とその関連ファイル内で含有対象と指定されていた場合の一部のプラットフォームにおける無限ループを防止します。(Tom)
- `SSL_read`または`SSL_write`が`errno`を設定することなく失敗した場合に無限ループする可能性を修正しました。(Tom)
一部の報告ではopensslの一部のWindows版で発生する可能性があります。
- `latex`および`troff`出力書式では行うべきではない文字列の書式付けを行わないようにpsqlの`numericlocale`オプションを修正しました。(Heikki)
- `ON_ERROR_STOP`と`--single-transaction`の両方が指定され、かつ、暗黙的な`COMMIT`時にエラーが起きた場合に、psqlが正しい終了コード(3)を返すようにしました。(Bruce)
- 複合列がNULLに設定される場合にplpgsqlが失敗することを修正しました。(Tom)
- PL/PerlUからのPL/Perl関数の呼び出し、およびその逆の場合において失敗する可能性を修正しました。(Tim Bunce)
- コンパイラ固有の誤動作が起こる可能性を防ぐためにPL/Python内に`volatile`記号を追加しました。(Zdenek Kotala)
- PL/Tclは確実にTclインタプリタを完全に初期化します。(Tom)

このミスの既知の兆候は、Tcl 8.5以降を使用している場合のTclの`clock`コマンドの誤動作だけです。

- `dblink_build_sql_*`関数であまりに多くのキー列が指定された場合の`contrib/dblink`のクラッシュを防止します。(Rushabh Lathia, Joe Conway)
- ずさんなメモリ管理によって引き起こる`contrib/xml2`の雑多なクラッシュを修正しました。(Tom)
- Windowsにおいて`contrib/xml2`の構築をより堅牢にしました。(Andrew)
- Widowsのシグナル処理における競合状態を修正しました。(Radu Ilie)

この不具合の兆候として既知のものは、負荷が高い状況において`pg_listener`内の行が削除されることです。

- 時間帯データファイルを`tzdata release 2010e`に更新しました。 バングラディッシュ、チリ、フィジー、メキシコ、パラグアイ、サモアにおける夏時間規則の変更が含まれます。

E.80. リリース8.2.15

リリース日: 2009-12-14

このリリースは8.2.14に対し、各種の不具合を修正したものです。 8.2メジャーリリースにおける新機能については[項E.95](#)を参照してください。

E.80.1. バージョン8.2.15への移行

8.2.Xからの移行ではダンプ/リストアは不要です。 しかし8.2.14以前のバージョンからアップデートする場合は8.2.14のリリースノートを参照してください。

E.80.2. 変更点

- セッション固有の状態を変更するインデックス関数により引き起こされる間接的なセキュリティ脅威から保護します。(Gurjeet Singh, Tom)

この変更により不変インデックス関数がスーパーユーザのセッションを破壊できることを防ぎます。(CVE-2009-4136)

- ヌルバイトが埋め込まれたコモンネーム (CN) フィールドをもつSSL証明書を拒絶します。(Magnus)

これは、SSL検証においてサーバまたはクライアント名に対する証明書が不正に一致しないようにします。(CVE-2009-4034)

- バックエンド起動時のキャッシュ初期化の間でクラッシュする可能性を修正しました。(Tom)
- 安全ではない時点でのVACUUMを中断するシグナルを防止します。(Alvaro)

この修正により、VACUUM FULLがすでにタプルの移動をコミットした後にキャンセルされた場合のパニックや、通常のVACUUMがテーブルを切り詰めた後に割り込みを受けた場合の一時的なエラーを防止します。

- ハッシュテーブルサイズの計算における整数オーバーフローによるクラッシュの可能性を修正しました。(Tom)

これは、ハッシュ結合の結果のプランナの推定サイズが極端に大きい場合に発生することがありました。

- inet/cidrの比較における非常にまれなクラッシュを修正しました。(Chris Mikkelsen)
- 準備されたトランザクションで保持されるタプルレベルの共有ロックを確実に無視しないようにしました。(Heikki)
- 副トランザクション内でアクセスされるカーソルで使用する一時ファイルの早すぎる削除を修正しました。(Heikki)
- GiSTインデックスページ分割において、分割がインデックスの最初の列以外に依存する場合の不正なロジックを修正しました。(Paul Ramsey)
- チェックポイントの最後で古いWALファイルの回収や削除が失敗した場合にエラーを出力しません。(Heikki)

致命的ではないものと扱い、チェックポイントを完了させることができるようにする方が優れています。 将来のチェックポイントでは削除を再試行します。 こうした問題は通常の操作では想定されませんが、うまく設計されていないWindowsのアンチウィルスソフトウェアやバックアップソフトウェアにより現れることがあります。

- WindowsにおいてWALファイルが繰り返しアーカイブされないようにしました。(Heikki)

別の症状として、一部の他のプロセスが不要となったファイルの削除に干渉する場合にも発生します。

- PAMパスワード処理をより堅牢に修正しました。(Tom)

以前のコードでは、Linuxのpam_krb5 PAMモジュールとドメインコントローラとしてのMicrosoft Active Directoryを組み合わせると失敗することが知られていました。 PAMスタックに渡される引数に関する正当ではない仮定を作成することであるため、どこかで問題になるかもしれません。

- CREATE OR REPLACE FUNCTIONにおける所有権限の依存関係の処理を修正しました。(Tom)
- plperlからplperl1の呼び出し、またはその逆における不具合を修正しました。(Tom)

内部の関数からのエラー終了が、外側の関数用のPerlインタプリタを正しく再選択できずにクラッシュをもたらす可能性があります。

- PL/Perl関数が再定義された時のセッション有効期間のメモリリークを修正しました。(Tom)
- 集合を返すPL/Perl関数により返される場合Perl配列が適切にPostgreSQL配列に返還されることを確実にしました。(Andrew Dunstan, Abhijit Menon-Sen)

集合以外を返す関数ではすでに正しく動作しました。

- PL/Pythonの例外処理におけるまれなクラッシュを修正しました。(Peter)
- psqlのflexモジュールが正しいシステムヘッダファイルを使用してコンパイルされることを確実にしました。(Tom)

これは、`--enable-largefile`が生成されたコード内に非互換の変更を引き起こすプラットフォームにおける構築失敗を修正します。

- 今後のバージョンのlibpqとの互換性を高めるために、postmasterが接続要求パケットにおける`application_name`パラメータをすべて無視するようにしました。(Tom)
- 最新の現実に合わせて時間帯省略形ファイルを更新しました。(Joachim Wieland)

デフォルト時間帯群にIDTの追加が含まれます。

- 時間帯データファイルをtzdata release 2009sまで更新しました。南極、アルゼンチン、バングラデシュ、フィジー、ノヴォクズネツク、パキスタン、パレスチナ、サモア、シリアにおける夏時間規則の変更、および香港に関する歴史的な修正が含まれます。

[前のページ](#)

リリース8.2.16

[ホーム](#)
[上に戻る](#)

[次のページ](#)

リリース8.2.14

E.81. リリース8.2.14

リリース日: 2009-09-09

このリリースは8.2.13に対し、各種の不具合を修正したものです。 8.2メジャーリリースにおける新機能については[項E.95](#)を参照してください。

E.81.1. バージョン8.2.14への移行

8.2.Xからの移行ではダンプ/リストアは不要です。しかし、`interval`列に対するハッシュインデックスを持つ場合は8.2.14にアップデートした後にそれらを`REINDEX`しなければなりません。また、8.2.11より前のバージョンからアップグレードする場合は、8.2.11のリリースノートを参照してください。

E.81.2. 変更点

- pg start backup() 実行中にWALセグメントスイッチを強制します。(Heikki)

これによりベースバックアップが使用不可となる可能性がある境界状態を防止します。

- security-definer関数内部のRESET ROLEおよびRESET SESSION AUTHORIZATIONを不許可としました。(Tom, Heikki)

これは、security-definer関数内部におけるSET ROLEとSET SESSION AUTHORIZATIONを不許可にする過去のパッチ（CVE-2007-6600参照）で抜けていたものを補うものです。

- 既にロードされたロード可能モジュールに対するLOADを何も行わないようにしました。(Tom)

これまでは、LOADはそのモジュールをアンロードし、再ロードしようとしていました。ですが、これは安全ではなく、かつ、まったく意味がないものでした。

- LDAP認証における空のパスワードを許可しません。(Magnus)

- 外側の集約関数の引数に現れる服問い合わせの扱いを修正しました。(Tom)

- ソートまたはマテリアライズ計画ノードの出力から行全体の値の抽出する処理に関連した不具合を修正しました。(Tom)

- 100句を超えるANDまたはORリストが存在した場合に部分インデックスと制約による除外の最適化を無効にするプランナの変更を元に戻しました。(Tom)

- interval データ型のハッシュ計算を修正しました。(Tom)

これは、時間間隔値に対するハッシュ結合が間違っただけの結果を生成しないように修正します。また、時間間隔型の列に対するハッシュインデックスの内容も変更します。もしこうしたインデックスがあれば、アップデートの後にそれらをREINDEXする必要があります。

- `to char(..., 'TH')` を 'HH'/'HH12' の大文字の序数表現と扱います。(Heikki)

以前は'th'（小文字）として扱われました。

- INTERVAL 'x ms'においてxが200万を超え、かつ、整数型の日付時刻が使用された場合のオーバーフローを修正しました。(Alex Hunsaker)

- 点と線分間の距離計算を修正しました。(Tom)

これにより幾何演算子の一部が間違った結果を返していました。

- 通貨が端数桁を持たないロケール、たとえば日本、において`money`データ型が動作するように修正しました。(Itagaki Takahiro)

- [illegible]

- GiST R-tree演算子クラスにおけるページ分割点の選択を改良しました。(Teodor)

- 入力値が（ほぼ）ソート済の順序であった場合におけるGINインデックスへの一括挿入の性能劣化を防止します。（Tom）

- PL/pgSQLの一部の文脈における、NOT NULLドメイン制約を正しく強制します。(Tom)
- plperlの初期化における移植性の問題を修正しました。(Andrew Dunstan)
- `postgresql.conf`が空の場合に、`pg_ctl`が無限ループに陥らないように修正しました。(Jeff Davis)
- キーまたは値がデータ構造に合わないほど長い場合、`contrib/hstore`が警告なく切り詰めるのではなく、エラーとなるようにしました。(Andrew Gierth)
- `contrib/xml2`の`xslt_process()`がパラメータの最大値(20)を正しく扱うように修正しました。(Tom)
- `COPY FROM STDIN`中のエラーからの復旧するためのlibpqのコードについて、堅牢性を高めました。(Tom)
- `readline`と`editline`ライブラリの両方がインストールされている場合、それらの競合するヘッダファイルがincludeされないようにしました。(Zdenek Kotala)
- バングラデシュ、エジプト、ヨルダン、パキスタン、アルゼンチン/サンルイス、キューバ、ヨルダン（歴史的な修正のみ）、モーリシャス、モロッコ、パレスチナ、シリア、チュニジアにおける夏時間規則の変更のため、時間帯データファイルをtzdataリリース2009Iに更新しました。

[前のページ](#)
リリース8.2.15

[ホーム](#)
[上に戻る](#)

[次のページ](#)
リリース8.2.13

E.82. リリース8.2.13

リリース日: 2009-03-16

このリリースは8.2.12に対し、各種の不具合を修正したものです。 8.2メジャーリリースにおける新機能については[項E.95](#)を参照してください。

E.82.1. バージョン8.2.13への移行

8.2.Xからの移行ではダンプ/リストアは不要です。 しかし、8.2.11より前のバージョンからアップグレードする場合は、8.2.11のリリースノートを参照してください。

E.82.2. 変更点

- 符号化方式の変換に失敗した時、再帰的なエラーによるクラッシュを防止しました。(Tom)

この変更は、最近の2つのマイナーリリースにて関連する失敗の状況に対してなされた改修を拡張したものです。 前回の修正は元の問題報告に特化したものでしたが、符号化方式変換関数で発生するすべてのエラーがそのエラーを報告しようとして、無限に再帰される可能性があることが分かりました。 したがって、再帰的なエラー報告を行う状況になったことがわかった時の解決策は、変換と符号化方式を無効にし、通常のASCII形式のエラーメッセージで報告することです。(CVE-2009-0922)

- 特定の変換関数に対する、間違っ符号化方式を用いたCREATE CONVERSIONを許可しません。(Heikki)

これにより、符号化方式に関する失敗における、あり得る状況を防止します。 前回の変更は、同じ問題における別の種類の失敗に対する防止策でした。

- to_char() に引数のデータに対して不適切な整形用のコードが渡された時のコアダンプを修正しました。(Tom)
- Cロケールがマルチバイト符号化方式で使用された場合のcontrib/tsearch2が失敗する可能性を修正しました。(Teodor)

wchar_tがintよりもビット数が小さい時にクラッシュする可能性がありました。 具体的にはWindowsです。

- Eメールのような複数の@文字を含む文字列に対する、非常に非効率的なcontrib/tsearch2のパーサの扱いを修正しました。(Heikki)
- 暗黙的な強制を使用したCASE WHENの逆コンパイルを修正しました。(Tom)

この間違いにより、アサートを有効にして構築した場合にアサート失敗が発生する可能性がありました。 また、他の構築状況でもビューの検証やダンプを行う際に"unexpected CASE WHEN clause"というエラーメッセージが発生する可能性がありました。

- TOASTテーブルの行型に対する所有者を間違って割り当てる可能性を修正しました。(Tom)

CLUSTERまたはALTER TABLEの書き換え構文がテーブル所有者以外のユーザにより実行された場合、テーブルのTOASTテーブル向けのpg_type項目が実行したユーザが所有するものとして記録されてしまいました。 TOASTの行型に対する権限は通常のデータベース操作ではまったく検証されませんので、これによりすぐに問題が発生することはありません。 しかし、後でコマンドを発行したロールを削除しようとした場合に想定外の失敗(8.1または8.2)や削除後にpg_dumpが"owner of data type appears to be invalid"という警告が発生する(8.3)可能性がありました。

- PL/pgSQLが、INSERT後のINTOを文字列の起点だけでなく、任意の位置におけるINTO変数句として扱わないように修正しました。 具体的には、CREATE RULE内のINSERT INTOにて失敗しないようにしました。(Tom)
- ブロックの終了時に、PL/pgSQLのエラー状態変数を完全に消去します。(Ashesh Vashi、Dave Page)

これはPL/pgSQL自体の問題ではありませんが、これを行わないと、PL/pgSQLデバグが関数の状態を検証する際にクラッシュする可能性がありました。

- WindowsにおいてCallNamedPipe() 呼び出しが失敗した時に再試行します。(Steve Marshall、Magnus)

この関数は一時的に失敗することが時々あるようです。 これまでは重大なエラーとしてすべての失敗を扱っていましたが、LISTEN、NOTIFYやその他の操作が混乱する可能性がありました。

- デフォルトの既知の時間帯省略形のリストにMUST（モーリシャス島夏時間）を追加しました。(Xavier Bugaud)

[前のページ](#)

リリース8.2.14

[ホーム](#)
[上に戻る](#)

[次のページ](#)

リリース8.2.12

E.83. リリース8.2.12

リリース日: 2009-02-02

このリリースは8.2.11に対し、各種の不具合を修正したものです。 8.2メジャーリリースにおける新機能については[項E.95](#)を参照してください。

E.83.1. バージョン8.2.12への移行

8.2.Xからの移行ではダンプ/リストアは不要です。しかし、8.2.11より前のバージョンからアップグレードする場合は、8.2.11のリリースノートを参照してください。

E.83.2. 変更点

- `headline()` 関数におけるURLの扱いを改良しました。(Teodor)
- `headline()` 関数における延長見出しの扱いを改良しました。(Teodor)
- 符号化方式の変換が符号化方式の組み合わせの指定を間違えた変換関数で作成された場合にあり得る、アサートの失敗や誤変換を防止します。(Tom, Heikki)
- 例えば`INSERT`が`UPDATE`に書き換えられるといった、PL/pgSQLで実行された文が他の種類の文に書き換えられた場合にあり得るアサートの失敗を修正しました。(Heikki)
- データ型入力関数でスナップショットが利用できることを確実にしました。(Tom)

これは主に、ユーザ定義の安定または不変関数を含む`CHECK`制約付きで宣言されたドメインに影響します。こうした関数は通常、スナップショットが設定されていない場合失敗します。

- データ型入出力、特に、ドメインの検査制約で使用するSPIを使用する関数をより安全にしました。(Tom)
- `VACUUM`において小規模なテーブルに対する不要なロックを防止しました。(Heikki)
- `UPDATE RETURNING tableoid`が現在のOIDではなくゼロをかえす問題を修正しました。(Tom)
- 推移的な等価性が外部結合句に適用される場合のプランナの選択度の誤推定を修正しました。(Tom)

これにより... `from a left join b on a.a1 = b.b1 where a.a1 = 42 ...`のような問い合わせで劣化した計画が作成されました。

- オプティマイザの長大な`IN`リストの扱いを改良しました。

この変更により、制約による除外が有効な場合にこうしたリストが、長く時間を浪費することを防止します。

- 保持可能なカーソルの内容がTOASTテーブルの内容に依存しないことを確実にしました。(Tom)

これまでカーソル内の大規模なフィールドはTOASTポインタとして表現される可能性がありました。これは、参照先のテーブルがカーソルを読み取る前に削除された場合や、大規模な値が削除、そしてバキュームされてしまった場合に失敗しました。これは通常のカーソルでは発生することはあり得ませんが、トランザクションの生成以降保持されるカーソルではあり得ます。

- 集合を返す関数が結果全体を読み取らずに終了した時のメモリリークを修正しました。(Tom)
- `contrib/dblink`の`dblink_get_result(text,bool)`関数を修正しました。(Joe)
- `contrib/sslinfo`関数から不要な出力が行われる可能性を修正しました。(Tom)
- `configure`スクリプトがPL/Perl用のリンク情報の入手ができなかった場合に失敗を適切に報告するように修正しました。(Andrew)
- すべての文書において、現在活動していない`pgsql-ports`と`pgsql-patches`メーリングリストの代わりに、`pgsql-bugs`または`pgsql-hackers`を参照するようにしました。
- 時間帯データファイルを`tzdata`リリース2009a (カトマンズ、およびスイス、キューバにおける歴史的なDSTの訂正) に

更新しました。

[前のページ](#)

リリース8.2.13

[ホーム](#)

[上に戻る](#)

[次のページ](#)

リリース8.2.11

E.84. リリース8.2.11

リリース日: 2008-11-03

このリリースは8.2.10に対し、各種の不具合を修正したものです。 8.2メジャーリリースにおける新機能については[項E.95](#)を参照してください。

E.84.1. バージョン8.2.11への移行

8.2.Xからの移行ではダンプ/リストアは不要です。しかし、8.2.7より前のバージョンからアップグレードする場合は、8.2.7のリリースノートを参照してください。また、これまでの8.2.Xリリースを使用していた場合、アップグレード後にすべてのGiSTインデックスを`REINDEX`することを勧めます。

E.84.2. 変更点

- 削除後に間違ったインデックス項目を"dead"と印付けたことによるGiSTインデックスの破損を修正しました。(Teodor)
これによりインデックス検索は存在するはずの行の検索に失敗します。破損したインデックスは`REINDEX`を使用して修復させることができます。

- クライアントの符号化方式が翻訳されたエラーメッセージを表すことができない時のバックエンドのクラッシュを修正しました。(Tom)

以前にも同様の問題に対応しましたが、"等価な文字を持たない"メッセージ自体が変換されない場合にまだ失敗しました。今回の修正では、こうした状況を検知した時、翻訳を無効にし、通常のASCIIエラーメッセージを送信します。

- トリガから深い入れ子状態の関数が呼び出された時にクラッシュする可能性を修正しました。(Tom)
- `expression IN (expression-list)`問い合わせの最適化を改良しました。(Tom。Robert Haasの発案に基づいた。)

8.2.xおよび8.3.xでは、右辺に問い合わせ変数が存在する状況を取扱う効率が過去のバージョンより落ちていました。今回の修正では、こうした状況における振舞いを8.1に戻しました。

- 副`SELECT`が`FROM`、複数行の`VALUES`リスト、`RETURNING`リスト内の関数呼び出し内に存在する場合の、間違ったルール問い合わせの展開を修正しました。(Tom)

この問題のよくある徴候は、"unrecognized node type"というエラーです。

- ハッシュされた集約計画の再スキャン中のメモリリークを修正しました。(Neil)
- 新しく定義されたPL/pgSQLトリガ関数が通常の関数として呼び出された場合に報告されるエラーを確実なものにしました。(Tom)
- `ALTER SET TABLESPACE`を使ってテーブルを別のテーブル空間に移動する時に`relfilenode`番号が衝突する可能性を防止します。(Heikki)

このコマンドは、先のディレクトリで未使用であることが確認されたファイル名を選ばずに、既存のファイル名を再使用しようとしていました。

- 1つの問い合わせ項目がテキストの先頭の単語に一致した場合、テキスト検索の間違った見出し生成を修正しました。(Sushant Sinha)
- `--enable-integer-datetimes`で構築された場合、ISO以外の日付書式を使用した時の時間間隔値における不適切な秒端数表示を修正しました。(Ron Mayer)
- 渡されたタプルとタプル記述子が異なる列数を持っていた場合に`SPI_getvalue`および`SPI_getbinval`が適切に動作することを確実にしました。(Tom)

テーブルに列が追加または削除された場合、こうした状況は正常です。しかしこの2つの関数は適切に扱っていませんでした。起こり得る唯一の結果は間違ったエラーが示されることです。

- `ecpg`の`CREATE ROLE`解析を修正しました。(Michael)

`pg_ctl restart`の最近の破損を修正しました。(Tom)

- `pg_control`がバイナリモードで開かれることを確実にしました。(Itagaki Takahiro)

`pg_controldata`および`pg_resetxlog`はこれを正しく行っていませんでした。このためWindowsで失敗しました。

- 時間帯データファイルをtzdataリリース2008i（アルゼンチン、ブラジル、モーリシャス、シリアにおける夏時間規則の変更）まで更新しました。

[前のページ](#)

リリース8.2.12

[ホーム](#)
[上に戻る](#)

[次のページ](#)

リリース8.2.10

E.85. リリース8.2.10

リリース日: 2008-09-22

このリリースは8.2.9に対し、各種の不具合を修正したものです。8.2メジャーリリースにおける新機能については[項E.95](#)を参照してください。

E.85.1. バージョン8.2.10への移行

8.2.Xからの移行ではダンプ/リストアは不要です。しかし、8.2.7より前のバージョンからアップグレードする場合は、8.2.7のリリースノートを参照してください。

E.85.2. 変更点

- B-tree WALリカバリコード内の不具合を修正しました。(Heikki)

ページ分割操作中にWALが途中で止まった場合にリカバリが失敗しました。

- `datfrozenxid`の誤計算の可能性を修正しました。(Alvaro)

このエラーにより古い`pg_clog`データの削除に失敗するというこ最近の報告が説明できます。

- 局所ロックのカウンタを32ビットから64ビットに広げました。(Tom)

これは、かなり長いトランザクションでカウンタがオーバーフローし、想定外の"すでに保持済のロックである"というエラーになるという報告に対応したものです。

- GiSTインデックススキャン中に出力タプルが重複する可能性を修正しました。(Teodor)

- ビューが単純な`UNION ALL`式を含む場合、行われなかった権限検査を修正しました。(Heikki)

被参照テーブルの権限は正しく検査されていましたが、ビュー自体の権限はそうではありませんでした。

- `INSERT`または`UPDATE`で生成されるタプルが対象のテーブルの現在の行型に一致するか確認するために、エグゼキュータ起動の際の検査を追加しました。(Tom)

`ALTER COLUMN TYPE`の後にこれまでキャッシュしていた計画を再利用した場合、こうした状況が起こります。データ破壊やその結果発生するクラッシュをこの検査により防ぎます。

- `DROP OWNED`中の削除が繰り返される可能性を修正しました。(Tom)

これによりよく"リレーションNNNのキャッシュ検索に失敗した"などのおかしなエラーが発生しました。

- まず時間帯引数を時間帯省略形として解釈し、これまでのように他の方法を取らずに、それが失敗した場合にのみ完全な時間帯名称として解釈するように`AT TIME ZONE`を修正しました。(Tom)

タイムスタンプ入力関数は常にこの順序であいまいな時間帯名の解決を行っていました。`AT TIME ZONE`も同様にすることで一貫性が向上し、そして、8.1で混入された互換性に関する不具合を修正します。過去のバージョンの`AT TIME ZONE`では省略形のみを受け付けていましたので、あいまいな場合に8.0以前と同様な動作をするようになりました。

- 64ビットプラットフォームで稼働している場合、日付時刻入力関数が正しく整数オーバーフローを検知できるよう修正しました。(Tom)

- 単位指定がある設定パラメータを表示する際に、単位変換による整数オーバーフローを防ぎます。(Tom)

- 非常に長いログメッセージをsyslogに書き出す際の性能を向上しました。(Tom)

- `pg_hba.conf`内のLDAP URLの接尾辞部分に空白文字を許します。(Tom)

- `SELECT DISTINCT ON`問い合わせに対するカーソルの後方スキャンにおける不具合を修正しました。(Tom)

- 入れ子状の副問い合わせ式がある場合のプランナの不具合を修正しました。(Tom)

外側の副問い合わせが親問い合わせに直接の依存性がなく、内側の副問い合わせが依存性がある場合、外側の値が新しい

親間い合わせの行に対して再計算されなくなる可能性があります。

- プランナが論理型の結果を生成する`GROUP BY`式を、式の内容に関係なく、常に2つのグループになると推定することを修正しました。(Tom)

これにより、`col IS NULL`などの特定の論理試験に対して、通常の`GROUP BY`推定より非常により良く精度が上がります。

- `FOR`ループの対象値が複合型のフィールドを含むレコードである場合に失敗しないようにPL/pgSQLを修正しました。(Tom)
- Tcl 8.5で正しく動作するようPL/Tclを修正しました。 また、Tcl間でやりとりされるデータの符号化方式についてより注意を払うように修正しました。(Tom)
- Windowsにおいて、libpqがシステムコール当たり64kBを超えて送信しないようにすることで、Microsoftの不具合を回避しました。(Magnus)
- `pg_dump`および`pg_restore`がSQLコマンドの送信に失敗した後に報告するエラーを改良しました。(Tom)
- `pg_ctl`が`restart`の間正しくpostmasterのコマンドライン引数を保持するように修正しました。(Bruce)
- 時間帯データファイルをtzdataリリース2008f（アルゼンチン、バハマ、ブラジル、モーリシャス、モロッコ、パキスタン、パレスチナ、パラグアイにおける夏時間規則の変更）まで更新しました。

[前のページ](#)

リリース8.2.11

[ホーム](#)

[上に戻る](#)

[次のページ](#)

リリース8.2.9

E.86. リリース8.2.9

リリース日: 2008-06-12

このリリースは8.2.8に対し、重大な1つの不具合と軽微な1つの不具合を修正したものです。8.2メジャーリリースにおける新機能については[項E.95](#)を参照してください。

E.86.1. バージョン8.2.9への移行

8.2.Xからの移行ではダンプ/リストアは不要です。しかし、8.2.7より前のバージョンからアップグレードする場合は、8.2.7のリリースノートを参照してください。

E.86.2. 変更点

- `pg_get_ruledef()` で負の定数を括弧でくるようにしました。(Tom)

この修正前では、ビューまたはルールにおける負の定数がそのまま、例えば`-42::integer`のようにダンプされる可能性があります。これは若干正しくありません。演算子の前置規則に従うと、`(-42)::integer`となるべきです。通常は大した違いが起こることはありませんが、最近の別のパッチと相互作用して、PostgreSQLが有効な`SELECT DISTINCT`ビューの問い合わせを拒絶することになりました。pg_dumpの出力のリロード失敗という結果になることもあり得たため、これは高優先度の修正として扱われました。実際にダンプ出力が正しくなかったバージョンは8.3.1と8.2.7だけです。

- `ALTER AGGREGATE ... OWNER TO`がpg_shdependを更新するようにしました。(Tom)

この見落としにより、集約が後で`DROP OWNED`または`REASSIGN OWNED`操作で使用された場合に問題が発生するおそれがありました。

E.87. リリース8.2.8

リリース日: リリースされませんでした

このリリースには8.2.7からの各種修正が含まれています。 8.2メジャーリリースにおける新機能については[項E.95](#)を参照してください。

E.87.1. バージョン8.2.8への移行

8.2.Xからの移行ではダンプ/リストアは不要です。 しかし、8.2.7より前のバージョンからアップグレードする場合は、8.2.7のリリースノートを参照してください。

E.87.2. 変更点

- WindowsでUTF-8データベース符号化方式と異なるクライアント符号化方式が使用された場合に発生するERRORDATA_STACK_SIZE exceededのクラッシュを修正しました。(Tom)
- 新しい列がすべて非NULLに初期化されていることが正しく検査されるようにALTER TABLE ADD COLUMN ... PRIMARY KEYを修正しました。(Brendan Jurd)

これまでのバージョンでは、この必要条件の検査をまったく行っていませんでした。

- "同じ"制約を、共通の祖先からその制約を継承する複数の親リレーションから継承する場合に、CREATE TABLEが失敗する可能性を修正しました。(Tom)
- UPDATEまたはDELETEの対象テーブルに別名が付与されていたならば、pg_get_ruledef() がその別名を表示するように修正しました。(Tom)
- too many LWLocks takenという結果をもたらす、GINの不具合を修正しました。(Teodor)
- 破損したデータを伸長する時に潜在するクラッシュを防止します。(Zdenek Kotala)
- バックエンドが共有メモリを破損したままSIGTERMによる終了する2か所を修正しました。(Tom)
- SIGTERMがデータベースクラスタ全体を停止するために使用された場合、どちらも非常に重要ではありません。 しかし、個々のバックエンドにSIGTERMを送信することを試みる場合が問題でした。
- キリル語の"Yo"文字 (2つのドットを持つёとѐ) を扱うISO-8859-5と他の符号化方式との間の変換を修正しました。(Sergey Burladyan)
- 複数のデータ型入力関数、特に、結果に初期化されず予測できない値を含む未使用のバイトを許すarray_in() を修正しました。(Tom)

これは、2つの明らかに同じリテラル値が等しいとみなされず、結果パーサは一致しないORDER BYおよびDISTINCT式に関して不平をいうといったエラーをもたらします。

- 正規表現副文字列一致 (substring(string from pattern)) における境界状態を修正しました。(Tom)

この問題は、パターン全体は一致するが、ユーザが括弧付きの副式を指定し、その副式が一致しなかった時に発生します。例えば、substring('foo' from 'foo(bar)?') です。(bar) は一致しませんので、これはNULLを返すべきです。しかし、間違ってパターン全体が一致 (つまりfoo) という結果を返していました。

- 時間帯データファイルをtzdataリリース2008c (モナコ、イラク、チョイバルサン、パキスタン、シリア、キューバ、アルゼンチン/サンルイスにおける昼時間規則の変更) まで更新しました。
- ecpgのPGTYPEStimestamp_sub() 関数からの不正確な結果を修正しました。(Michael)
- contrib/tsearch2のtsquery型用の壊れたGiST比較関数を修正しました。(Teodor)
- contrib/cube関数に潜在したクラッシュを修正しました。(Tom)
- 入力問い合わせがNULL値を返す場合のcontrib/xml2のxpath_table() 関数のコアダンプを修正しました。(Tom)

contrib/xml2のmakefileがCFLAGSを上書きしないように修正しました。(Tom)

- DatumGetBool マクロがgcc 4.3でエラーにならないよう修正しました。(Tom)

この問題は、論理値を返す"古い形式" (V0)のC関数に影響します。これは8.3ではすでに修正されていましたが、当時なかったバックパッチが必要になりました。

[前のページ](#)

リリース8.2.9

[ホーム](#)
[上に戻る](#)

[次のページ](#)

リリース8.2.7

E.88. リリース8.2.7

リリース日: 2008-03-17

このリリースは8.2.6の各種不具合を修正したものです。 8.2メジャーリリースにおける新機能については[項E.95](#)を参照してください。

E.88.1. バージョン8.2.7への移行

8.2.Xからの移行ではダンプ/リストアは不要です。 しかし、後述のWindowsロケール問題の影響がある場合、移行後にテキスト型の列上のインデックスに対しREINDEXを行わなければならないかもしれません。

E.88.2. 変更点

- 異なる文字の組み合わせを等価とみなすWindowsロケールにおける文字列比較を修正しました。(Tom)

この修正はWindowsのみ、かつ、UTF-8データベース符号化方式を使用している場合にのみ当てはまります。 他の状況に対する同じ修正が2年前になされましたが、UTF-8を使用するWindowsでは変更されなかった経路のコードを使用しました。 同一ではない一部の文字列を等価とみなすロケールを使用している場合、既存のテキスト型列上のインデックスを修正するためにREINDEXを行う必要があるかもしれません。

- 異なるシステムカタログに対する同時実行VACUUM FULLの間でのデッドロックの可能性を修正しました。(Tom)
- 長く存在したLISTEN/NOTIFY競合条件を修正しました。(Tom)

まれな場合ですが、NOTIFYを実行した同時実行中のトランザクションが後でコミットされたことが確認できたため、受け取ることが期待できる場合であっても、LISTENを実行したばかりのセッションが注意を受け取らない可能性があります。

この修正の副作用として、まだコミットされていないLISTENコマンドを実行したトランザクションが、LISTENで検索するために選ばなければならないpg_listener内の行にまったく現れなくなりました。 これまでは現れていました。 この動作についてはどちらの方式も文書化されていませんでしたが、過去の動作に依存するアプリケーションがあるかもしれません。

- 準備されたトランザクションではLISTENとUNLISTENは使用できません。(Tom)

これまでは許されていましたが、これを行うと様々な悪影響が発生しました。 特に構成したバックエンドはUNLISTENが未コミットである間終了することができませんでした。

- 準備されたトランザクションでは一時テーブルの削除を行うことはできません。(Heikki)

これは正確には8.1で禁止されていたのですが、8.2と8.3では不注意にこの検査が壊れていました。

- ハッシュインデックスを使用する問い合わせの間にエラーが発生した場合のまれなクラッシュを修正しました。(Heikki)
- ある方法で集合を返す関数を使用した場合のメモリリークを修正しました。(Neil)
- 紀元前の2月29日に関する日付時刻値の入力を修正しました。(Tom)

これまでのコードでは、うるう年の判定に間違いがありました。

- ALTER OWNERの一部の構文で発生した"未知のノード型"エラーを修正しました。(Tom)
- ロック待ちが中断した時にpg_stat_activity.waitingフラグを確実に初期化します。(Tom)
- Windows Vistaにおいてプロセス処理の取扱いを修正しました。(Dave, Magnus)

具体的には、この修正により管理者ユーザとしてサーバを起動できるようになります。

- 時間帯データファイルをtzdataリリース2008aに更新しました（具体的には、最近のチリの変更です）。 時間帯省略形VET（ベネズエラ）がUTC-4:00ではなくUTC-4:30を意味するように調整しました。(Tom)
- pg_ctlが正しくコマンドラインオプションからpostmasterのポート番号を取り出すように修正しました。(Itagaki)

Takahiro, Tom)

これまでは、`pg_ctl start -w`は間違ったポート上のpostmasterに接続しようとし、おかしい起動失敗の報告をもたらしていました。

- 最近のバージョンのgccにおける、最適化ミスの可能性を防止するために`-fwrapv`を使用します。(Tom)

これはgcc 4.3以降でPostgreSQLを構築する場合に必須であることは分かっていました。

- `INT_MAX` マイクロ秒 (約35分) より大きい`statement_timeout`値を正しく強制します。(Tom)

この不具合は`--enable-integer-datetimes`で構築した場合にのみ影響します。

- 一定の折りたたみが副問い合わせを単純化する時の"想定外のPARAM_SUBLINK ID"というプランナエラーを修正しました。(Tom)

- `IS NULL`および`NOT`式の、制約による除外の取扱いにおける論理エラーを修正しました。(Tom)

プランナは時々、NULL結果の可能性のために除外すべきでないパーティションを除外しました。

- "failed to build any N-way joins"というプランナエラーの別の原因を修正しました。(Tom)

これは、結合句が見つかる前に句のない結合を強制しなければならない場合に発生しました。

- 外部結合の計画作成における間違った定数の伝搬を修正しました。(Tom)

プランナは時々、間違った問い合わせ結果をもたらす、変数が定数と同じであると制約されていると間違った判断を行いました。

- `ORDER BY`および`GROUP BY`における定数式の表示を修正しました。(Tom)

明示的にキャストされた定数が間違って表示されていました。これは例えば、ダンプ/再ロード時のビュー定義の破壊をもたらします。

- `COPY OUT`時にNOTICEメッセージを正しく扱うようにlibpqを修正しました。(Tom)

この失敗は、ユーザ定義データ型の出力関数がNOTICEを発行する時にのみ発生することが観察されました。しかし、他の原因により発生しないことは保証しません。

E.89. リリース8.2.6

リリース日: 2008-01-07

このリリースは、重大なセキュリティ問題の修正を含む、8.2.5の各種不具合を修正したものです。8.2メジャーリリースにおける新機能については[項E.95](#)を参照してください。

E.89.1. バージョン8.2.6への移行

8.2.Xからの移行ではダンプ/リストアは不要です。

E.89.2. 変更点

- インデックス内の関数がVACUUM、ANALYZEなどを実行するユーザの権限で実行されることを防止します。(Tom)

インデックス式内で使用される関数および部分インデックスの述語は、新しいテーブル項目が作成されると評価されます。これが、もし信頼できないユーザが所有するテーブルを変更したとすると、トロイの木馬となるコードの実行という危険性を引き起こすことは以前から解っていました。(トリガ、デフォルト、検査制約なども同種の危険性を引き起こします。)しかしインデックス内の関数は、VACUUM FULLなどの日常の保守作業で実行される可能性があるため、さらに危険です。こうした保守作業は通常自動的にスーパーユーザ権限で行われます。例えば、極悪なユーザがトロイの木馬となるインデックス定義を設定し、次の日常のバキューム処理を待つことで、スーパーユーザ権限でコードを実行することが可能です。この修正は、SECURITY DEFINER関数で使用されていた権限変更機構を使用して、標準的な保守作業(VACUUM、ANALYZE、REINDEX、CLUSTERを含む)が呼び出したユーザではなくテーブル所有者として実行するように調整しました。こうしたセキュリティ制限の迂回を防ぐために、SET SESSION AUTHORIZATIONおよびSET ROLEがSECURITY DEFINERコンテキストで禁止されるようになりました。(CVE-2007-6600)

- 正規表現パッケージ内のさまざまな不具合を修正しました。(Tom, Will Drewry)

より巧妙な正規表現パターンにより、信頼できないソースから正規表現検索パターンを受け付けるアプリケーションでは、クラッシュ、無限または無限に近いループ、大量のメモリの浪費、DoS攻撃を引き起こすすべてが発生する可能性があります。(CVE-2007-4769、CVE-2007-4772、CVE-2007-6067)

- /contrib/dblinkを使用して、パスワード認証のみをセキュリティ制限として使用するには非スーパーユーザでなければなりません。

8.2.5で行った修正は不十分で、一部のdblink関数に対してのみ問題を解消していました。(CVE-2007-6601、CVE-2007-3278)

- GINインデックスに対するWAL再生における不具合を修正しました。(Teodor)
- maintenance_work_memが4GB以上の場合にGINインデックスの構築が適切に動作するように修正しました。(Tom)
- 時間帯ファイルをtzdataリリース2007kに更新しました(具体的には最近のアルゼンチンの変更)。(Tom)
- 非CロケールにおけるプランナのLIKE/正規表現の推定を改良しました。(Tom)
- 深い階層の外部結合における計画処理速度を修正しました。同様に結合順の非効率な選択を行う可能性も修正しました。(Tom)
- 一部のWHERE false AND var IN (SELECT ...)におけるプランナの失敗を修正しました。(Tom)
- CREATE TABLE ... SERIALとALTER SEQUENCE ... OWNED BYがシーケンスのcurrval()状態を変更しないようにしました。(Tom)
- ALTER TABLE ... ALTER COLUMN TYPEで再構築されるインデックスのテーブル空間と格納パラメータを保持します。(Tom)
- アーカイブリカバリが、リカバリの停止時間が使用された時だけでなく、常に新しいWAL時系列から始まるようにしました。

これは、まれに起こる可能性のある既存の最後のWALセグメントのアーカイブされたコピーを上書きしようとする危険を防止します。また元の定義より単純、明確になったと考えられます。

- テーブルが小さすぎて有用でない場合、`VACUUM`がすべての`maintenance_work_mem`を使用しないようにしました。
- マルチバイトデータベース符号化方式を使用している時に潜在する、`translate()`のクラッシュを修正しました。
- `corr()`が負の相関値に対して正しい結果を返すようにしました。(Neil)
- 68年を超える間隔で`extract(epoch from interval)`がオーバーフローすることを修正しました。(Tom)
- 信頼された関数内でUTF-8正規表現が使用された場合にPL/Perlが失敗しないように修正しました。(Andrew)
- プラットフォームのPerlが`bool`型を`char`ではなく`int`として定義した場合に対応するようにPL/Perlを修正しました。(Tom)

これは理論上どこでも発生する可能性がありましたが、標準的に構築されたPerlではこうなりませんでした。Mac OS X 10.5までは...

- 64ビットマシン上のPython 2.5で正常に動作するようPL/Pythonを修正しました。(Marko Kreen)
- 長い例外メッセージでクラッシュしないようPL/Pythonを修正しました。(Alvaro)
- 親テーブルと異なるデフォルト式を持つ、継承子テーブルを正しく扱うよう`pg_dump`を修正しました。(Tom)
- `PGPASSFILE`が普通のファイル以外のファイルを参照する場合に`libpq`がクラッシュすることを修正しました。
- `ecpg`パーサを修正しました。(Michael)
- 128ビットより長いキーで失敗するOpenSSLライブラリに対して`contrib/pgcrypto`が防衛するようになりました。これは少なくとも一部のバージョンのSolarisで発生します。(Marko Kreen)
- `contrib/tablefunc`の`crosstab()`が、クラッシュするのではなく、独自の意味でカテゴリとしてNULL行識別子を扱うようにしました。(Joe)
- `tsvector`および`tsquery`出力関数が正しくバックスラッシュをエスケープするように修正しました。(Teodor, Bruce)
- 入力文字列が大きな場合に`to_tsvector()`がクラッシュすることを修正しました。(Teodor)
- `configure`スクリプトを再生成する場合、特定のバージョンのAutoconfを使用することが必要になりました。(Peter)

これは開発者とパッケージ作成者のみに影響します。この変更は、AutoconfとPostgreSQLの未確認のバージョンの組み合わせを使用する事故を防ぐためになされました。その結果がうまくいくかどうかについて責任を負う限り、異なるバージョンのAutoconfを本当に使用したい場合は、このバージョン検査を取り除くことができます。

- PostgreSQLがMinGWの新しいバージョンで構築できるように`gettimeofday`の構築用検査を更新しました。

E.90. リリース8.2.5

リリース日: 2007-09-17

このリリースは8.2.4の各種不具合を修正したものです。 8.2メジャーリリースにおける新機能については[項E.95](#)を参照してください。

E.90.1. バージョン8.2.5への移行

8.2.Xからの移行ではダンプ/リストアは不要です。

E.90.2. 変更点

- トランザクションが、行を挿入した後に、同一テーブルに対する同時実行中のVACUUMの終了時刻とほぼ同じ時点でアポートした場合のインデックス破損を防止します。(Tom)
- ドメイン越しのドメインを含む場合のALTER DOMAIN ADD CONSTRAINTを修正しました。(Tom)
- CREATE DOMAIN ... DEFAULT NULLが適切に動作するようにしました。(Tom)
- 外部結合に関するプランナの問題をいくつか、特に、t1 LEFT JOIN t2 WHERE t2.col IS NULLに対するおかしいサイズ推定を修正しました。
- intervalデータ型が、ミリ秒のみ、または、マイクロ秒のみから構成された入力を受け付けられるようになりました。(Neil)
- timestamp型の値で、年の前に時間帯名称を現せられるようになりました。(Tom)
- /contrib/tsearch2で使用されるGINインデックスを修正しました。(Teodor)
- r-treeインデックスの挿入を高速化しました。(Teodor)
- 過度なSSLエラーメッセージのログ処理を修正しました。(Tom)
- syslogerプロセスを使用する時にログメッセージが差し込まれないようにログ処理を修正しました。(Andrew)
- log_min_error_statementログ処理がメモリ不足になった時のクラッシュを修正しました。(Tom)
- 一部の外部キーの境界の不正な扱いを修正しました。(Tom)
- stddev_pop(numeric) と var_pop(numeric) を修正しました。(Tom)
- 他のセッションによる一時テーブル処理を防ぐために起こるREINDEXとCLUSTERの失敗を防ぎます。(Alvaro)
- 時間帯データベース規則を更新しました。具体的にはニュージーランドで予定している変更です。(Tom)
- Windowsソケットとセマフォを改良しました。(Magnus)
- pg_ctl -wがWin32サービスモードで適切に動作するようになりました。(Dave Page)
- WindowsでMIT Kerberosを使用する場合のメモリ割り当てに関する不具合を修正しました。(Magnus)
- 符号化方式が不整合となる可能性があるため、Windowsにてログのタイムスタンプに時間帯名(%Z)を出力しないようにしました。(Tom)
- セキュリティ強化のため、パスワード認証のみを使用する/contrib/dblinkを使用するユーザが非特権ユーザであることを要求します。(Joe)
- セキュリティ上の理由で、/contrib/pgstattuple関数をスーパーユーザに制限しました。(Tom)
- /contrib/intarrayがGIN演算子クラスをデフォルトにしないようにしました（ダンプ/リストア時に問題が発生します）。(Tom)

E.91. リリース8.2.4

リリース日: 2007-04-23

このリリースは8.2.3の各種不具合を修正したもので、セキュリティ問題の修正を1つ含みます。8.2メジャーリリースにおける新機能については[項E.95](#)を参照してください。

E.91.1. バージョン8.2.4への移行

8.2.Xからの移行ではダンプ/リストアは不要です。

E.91.2. 変更点

- `search_path`における一時テーブルスキーマの明示的な指定をサポートしました。また、関数や演算子ではその検索を無効にしました(Tom)。
これは、SECURITY DEFINER関数が`search_path`の値を本当に安全に設定するために必要です。こうしないと、一時オブジェクトを使用して権限のないSQLユーザがSECURITY DEFINER関数の権限でコードを実行することができてしまうためです (CVE-2007-2138)。詳細はCREATE FUNCTIONを参照してください。
- Windowsにおいて、各バックエンドで再ロードを強制的に行うことで`shared_preload_libraries`を修正しました (Korry Douglas)。
- 翻訳された日や月名を適切に大文字、小文字に変換できるように、`to_char()`を修正しました。
- `/contrib/tsearch2`のクラッシュを修正しました(Teodor)。
- COMMIT PREPAREDは、そのトランザクションを準備したデータベースと同じデータベースで実行する必要があります。(Heikki)
- Windowsにおいて、2ギガバイトを越えるバイナリバックアップを`pg_dump`が行えるようになりました(Magnus)。
- 伝統的中國（台灣）語のFAQを更新しました(Zhou Daojing)。
- 統計情報コレクタがあまりに頻繁にディスクに書き込まないようにしました(Tom)。
- VACUUM FULLのUPDATE連鎖方法において潜在したデータ破損不具合を修正しました(Tom, Pavan Deolasee)。
- 配列型を使用するドメインにおける不具合を修正しました(Tom)。
- 元となるテーブルのダンプを行わない場合でも、`-t`を使用して連番型の列のシーケンスをダンプできるよう`pg_dump`を修正しました。(Tom)
- 外部結合の向上やビットマップスキンの選択ロジックなど、プランナを修正しました(Tom)。
- PL/pgSQL関数がEXCEPTIONブロック内でRETURNを試行する場合に潜在する誤作動やクラッシュの可能性を修正しました。(Tom)
- ハッシュインデックスの拡張時のパニックを修正しました(Tom)。
- POSIX書式の時間帯指定が新しいUSA DST規則に従うよう修正しました(Tom)。

E.92. リリース8.2.3

リリース日: 2007-02-07

このリリースには、8.2.2に対する2つの修正が含まれています。 8.2メジャーリリースにおける新機能については[項E.95](#)を参照してください。

E.92.1. バージョン8.2.3への移行

8.2.Xからの移行ではダンプ/リストアは不要です。

E.92.2. 変更点

- 制約および関数インデックスにおける型の長さに対する重複/限定検査を削除しました。(Tom)
- 副問い合わせにおけるMIN/MAXがインデックスを再度利用できるよう最適化を修正しました。(Tom)

E.93. リリース 8.2.2

リリース日: 2007-02-05

このリリースは8.2.1の各種不具合を修正したもので、セキュリティ問題の修正も含みます。 8.2メジャーリリースにおける新機能については[項E.95](#)を参照してください。

E.93.1. バージョン8.2.2への移行

8.2.Xからの移行ではダンプ/リストアは不要です。

E.93.2. 変更点

- 接続ユーザがバックエンドのメモリを読み取ることができるセキュリティ脆弱性を取り除きました。(Tom)

この脆弱性には、SQL関数が宣言されたデータ型を返すかどうか、そのテーブル列のデータ型が変更されたかどうかに関する通常の検査の抑制も含まれています (CVE-2007-0555、CVE-2007-0556)。これらのエラーを悪用して簡単にバックエンドをクラッシュさせることができます。 また原理的には、アクセスを許していないはずのユーザがデータベースの内容を読み取ることができてしまいます。

- 実行不可能な分割点の選択によりB-Treeインデックスページの分割が失敗する可能性がある、稀にも起こらないような不具合を修正しました。(Heikki Linnakangas)

- Borland Cコンパイルスクリプトを修正しました。(L Bayuk)

- 00で終わる年に対する`to_char('CC')`の扱いが適切になりました。(Tom)

2000年は21世紀ではなく、20世紀です。

- `/contrib/tsearch2`の各種言語対応を改良しました。(Tatsuo, Teodor)

- `information_schema.key_column_usage`ビューにおける不正確な権限検査を修正しました。(Tom)

この兆候は"OID nnnnnを持つリレーションが存在しない"というエラーです。 `initdb`を行わずにこの修正を適用するには、`CREATE OR REPLACE VIEW`を使用して、`share/information_schema.sql`内にある正しい定義をインストールすることです。 各データベースでこれを行わなければならないことに注意してください。

- 多くのテーブルを持つデータベースに対する`VACUUM`の性能を向上しました。(Tom)

- `UNION`をきっかけとした、稀に発生する`Assert()`クラッシュを修正しました。(Tom)

- `ROW`不等条件を使用するインデックス検索が不正確になる潜在的な不具合を修正しました。(Tom)

- 3バイト長を越えるUTF8シーケンスに関する、複数バイト文字処理のセキュリティを強化しました。(Tom)

- 削除済みのファイルに対する`fsync`試行による、Windowsにおいて発生するにせの"権限がない"というエラーを修正しました。(Magnus, Tom)

- Windowsにおいて統計情報コレクタがハングする不具合を修正しました。(Magnus)

この不具合に伴い自動バキュームも動作不能になります。

- 使用中のPL/pgSQL関数を更新する時にクラッシュする可能性を修正しました。(Tom)

- PL/pgSQLにおけるドメイン型の扱いを改良しました。(Sergiy Vyshnevetskiy, Tom)

- PL/pgSQL例外ブロックの処理でエラーが起きる可能性を修正しました。(Tom)

E.94. リリース 8.2.1

リリース日: 2007-01-08

このリリースは8.2の各種不具合を修正したものです。 8.2メジャーリリースにおける新機能については[項E.95](#)を参照してください。

E.94.1. バージョン8.2.1への移行

8.2からの移行ではダンプ/リストアは不要です。

E.94.2. 変更点

- `SELECT ... LIMIT ALL` (または`LIMIT NULL`) 時のクラッシュを修正しました。(Tom)
- `/contrib/tsearch2`に対する複数の修正(Teodor)
- Windowsにおいて、オペレーティングシステムからのログメッセージをASCII符号化方式に変更しました。(Hiroshi Saito)

この修正は、オペレーティングシステムとデータベースサーバの間で符号化方式に違いがある場合の変換問題を解消するものです。

- Windowsにおける`win32.mak`を使用した`pg_dump`のリンクを修正しました。(Hiroshi Saito)
- 外部結合問い合わせに対するプランナの間違いを修正しました。(Tom)
- 副問い合わせを含む問い合わせにおける複数の問題を修正しました。(Tom)
- SPIにおける副トランザクションのアボート時のクラッシュの可能性を修正しました。(Tom)
- PLはすべてSPIを使用しますので、これはすべてのPL関数に影響します。
- PDF文書作成を高速化しました。(Peter)
- JST (日本)時間帯省略形を再度追加しました。(Tom)
- インデックススキャンに関連した最適化方針の決定を改良しました。(Tom)
- `psql`において、複数バイトを組み合わせた文字を`¥u`として出力せずに、以前のように出力するようにしました。(Tom)
- 括弧を使用した正規表現を使用するインデックスを改良しました。(Tom)

この改良により、`psql ¥d`の性能も向上しました。

- 8.2以前のサーバからダンプする際に、`pg_dumpall`は、そのデータベースがPUBLICに`CONNECT`権限を持たせていることを前提とするようにしました。(Tom)

これは、`pg_hba.conf`で許可されていれば、誰でもデータベースに接続できるという以前の振る舞いを維持します。

E.95. リリース 8.2

リリース日: 2006-12-05

E.95.1. 概要

本リリースでは、以下のような多くの機能追加やユーザから要望があった性能改善がなされています。

- INSERT/UPDATE/DELETE RETURNINGや複数行のVALUESリスト、UPDATE/DELETEにおける省略可能な対象テーブル別名など問い合わせ言語の改良。
- 同時に行われるINSERT/UPDATE/DELETE操作をブロックしないインデックス作成。
- 外部結合の再順序付けのサポートなどの、多くの問い合わせ最適化の改良。
- ソート処理性能の向上、および、使用するメモリ量の低減。
- より効率的なロックによる同時実行性の向上。
- より効率的なバキューム処理。
- ウォームスタンバイサーバ管理の簡略化。
- テーブルおよびインデックスにおけるFILLFACTOR サポート（新機能）。
- 監視機能、ログ機能、性能チューニングの追加。
- オブジェクトの生成、削除に関するより細かな制御。
- 既存テーブルに対するテーブル継承関係の定義および削除が可能になりました。
- COPY TOが任意のSELECT分の出力をコピーできるようになりました。
- 配列内のNULLなど、配列の改良。
- 複数入力集約やSQL:2003の統計処理関数など、集約関数の改良。
- 多くのcontrib/の改良。

E.95.2. バージョン8.2への移行

すべての過去のリリースからデータを移行する場合、pg_dumpを使用したダンプ/リストアが必要です。

以下に既知の非互換な点を示します。

- `escape_string_warning`をデフォルトでonにしました。(Bruce)
これは、バックスラッシュエスケープが非エスケープ(Eの付かない)文字列内で使用された場合に警告を発します。
- リスト要素foo.*が、以前のような入れ子状の行型フィールドを生成するのではなく、そのメンバフィールドのリストに展開されるように、[行コンストラクタ構文](#) (ROW(...))を変更しました。(Tom)
実質、この新しい動作は例えば、トリガでIF row(new.*) IS DISTINCT FROM row(old.*)を使用してデータ変更を検査することができるなど、大変便利です。 .*を省略することで古い動作を使用することもできます。
- [行の比較](#)を標準SQL構文に従うようにしました。 また、インデックススキャンで使用されるようになりました。(Tom)
以前は、行の等号および <> 比較は標準に従っていましたが、< <= > >= は従っていませんでした。 新しい比較では、行値に一致する複数列インデックス用のインデックス制約として使用されるようになりました。
- [row IS \[NOT\] NULL](#) 検査が標準SQL文法に従うようにしました。(Tom)
以前の動作では、IS NULLという単純な場合は標準に従っていましたが、IS NOT NULLは行のフィールドのいずれかが非NULLの場合に真を返しました。 標準では、全てのフィールドが非NULLの場合にのみ真を返すものと規定しています。
- [SET CONSTRAINT](#) が1つの制約にのみ適用されるようにしました。(Kris Jurka)
以前のリリースでは、SET CONSTRAINTは一致する名前を持つすべての制約を変更しました。 本リリースではスキーマ検索パスを使用し、最初に一致する制約のみを変更するようになりました。 スキーマの指定もサポートされます。 これにより、より標準SQLに従うようになりました。
- セキュリティのためテーブルのRULE権限を削除しました。(Tom)
本リリースから、テーブルの所有者のみがテーブルのルールを作成と変更を行うことができるようになります。 後方互換性のためGRANT/REVOKE RULEはまだ受け付けられますが、何も行われません。
- 配列比較の改良。(Tom)
配列次元も比較されるようになりました。
- [配列の結合](#)を文書に合わせるように変更しました。(Tom)
この変更により、結合が配列の下限値を変更するような場合、過去の動作と異なるようになります。
- postmasterとpostgres コマンドラインオプションを同一にしました。(Peter)
これにより、postmasterが、-oを使用せずに、各バックエンドに引数を渡すことができるようになりました。 単一文字のオプションと競合するため、一部のオプションは長めのオプションとしてのみ利用できるようになったことに注意してください。
- postmasterシンボリックリンクの使用の廃止予定(Peter)
postmaster及びpostgresコマンドが同じように動作するようになりました。 その動作はコマンドラインオプションによって定まります。 postmasterというシンボリックリンクが互換性のために残されていますが、実際のところ必要性はありません。
- 問い合わせが出力されていなくても[log_duration](#)が出力するように変更しました。(Tom)

以前のリリースでは、問い合わせが過去にログに現れている場合にのみ`log_duration`が出力されました。

- `to_char(time)`および`to_char(interval)`がHHとHH12を12時間周期で扱うようになりました。

ほとんどのアプリケーションでは12時間周期の表示を求めない限り、HH24を使用すべきです。

- 変換におけるマスクされないゼロビットがINETからCIDRになりました。(Tom)

これにより変換された値が確実にCIDRとして有効になるようになりました。

- `australian_timezones`設定変数を削除しました。(Joachim Wieland)

この変数は、より一般的な時間帯省略形の設定機構により上書きされていました。

- 入れ子状ループインデックススキャン用のコスト推定が改良されました。(Tom)

これは、非現実的な小さい`random_page_cost`値に設定する必要性を無視するかもしれません。非常に小さな`random_page_cost`を使用している場合は、試験をして確認してください。

- `pg_dump`の`-n`オプションと`-t`オプションの動作を変更しました。(Greg Sabino Mullane)

詳細は`pg_dump`のマニュアルページを参照してください。

- `libpqPQdsplen()`が有効な値を返すように変更しました。(Martijn van Oosterhout)

- `libpqPQgetssl()`をSSL *ではなくvoid *を返すように宣言しました。(Martijn van Oosterhout)

これによりアプリケーションは、OpenSSLヘッダをインクルードすることなくこの関数を使用できるようになります。

- バージョン互換性検査のために、C言語による動的ロード可能なモジュールが`PG_MODULE_MAGIC`マクロ呼び出しをインクルードしなければならなくなりました。(Martijn van Oosterhout)

- セキュリティ上の理由により、PL/PerlU関数によって使用されるモジュールはPL/Perl関数から使用できなくなりました。(Andrew)

注意: これはまた、PL/Perl関数とPL/PerlU関数との間でデータを共有できなくなったことも意味します。一部のPerlインストレーションでは、単一プロセス内で複数のインタプリタが存在することを許すための正しいフラグをつけてコンパイルされていませんでした。こうした状況では、PL/PerlとPL/PerlUを単一のバックエンドで使用することはできません。解決方法は、複数のインタプリタをサポートするPerlインストレーションを入手することです。

- `contrib/xml2/`で、`xml_valid()`が`xml_is_well_formed()`に名前を変更しました。(Tom)

後方互換性のために`xml_valid()`も残っていますが、今後のリリースでスキーマ検査を行うように変更されます。

- `contrib/ora2pg/`が削除されました。 <http://www.samse.fr/GPL/ora2pg>が新設されました。

- PgFoundryに移行されたcontribモジュール`adddepend`、`dbase`、`dbmirror`、`fulltextindex`、`mac`、`userlock`が削除されました。

- 見捨てられたcontribモジュール`mSQL-interface`、`tips`が削除されました。

- QNXおよびBEOSへの移植が削除されました。(Bruce)

これらの移植に関して活動できる保守者がいなくなりました。

E.95.3. 変更点

以下に、PostgreSQL 8.2と以前のメジャーリリースとの間の違いについて詳しく記します。

E.95.3.1. 性能向上

- ある環境において、プランナが外部結合の再順序付けすることが可能になりました。(Tom)

以前のリリースでは、外部結合は常に問い合わせに記載された順序で評価されていました。この変更により、結合の順序を変更しても問い合わせの意味が変わらないことが決定できる場合、問い合わせオプティマイザが外部結合の順序を考慮できるようになりました。複数の結合が含まれる場合や、内部結合と外部結合が混在する場合に、目に見える程の性能が向上します。

- `IN (式のリスト)`の効率を向上しました。(Tom)

- ソート速度を向上し、メモリ消費量を低減しました。(Simon, Tom)

- 副トランザクションの性能を向上しました。(Alvaro, Itagaki Takahiro, Tom)

- `table`および`index`の生成に`FILLFACTOR`を追加しました。(ITAGAKI Takahiro)

これは、データベースの容量が増えた時の性能を向上させることができるように、各テーブルページやインデックスページに追加の空き領域を確保します。これは特に、クラスト処理を管理する時に有効です。

- `shared_buffers`および`max_fsm_pages`のデフォルト値を増やしました。(Andrew)

- ロック管理テーブルをセクションに分割することでロック処理の性能を向上しました。(Tom)

これによりロック処理の粒度が細くなり、競合が減少します。

- シーケンシャルスキャンにおけるロックの必要性を減少しました。(Qingqing Zhou)

- データベース生成または削除時のロックの必要性を減少しました。(Tom)

- `LIKE`、`ILIKE`、`regular expression`操作における、オプティマイザの選択性推定を改良しました。(Tom)

- 継承されたテーブルの結合や`UNION ALL`ビューの計画作成を改良しました。(Tom)

- 制約による除外が継承された`UPDATE`および`DELETE`問い合わせに適用できるようになりました。(Tom)

既に`SELECT`は制約による除外に従います。

- 条件が外側の問い合わせを継承する変数にのみに依存する条件など、一定の`WHERE`句に対する計画作成を改良しました。(Tom)

- プロトコルレベルの無名準備済み文が、`BIND`値の集合毎に計画作成されるようになりました。(Tom)

正確なパラメータが計画で使用されるようになりますので、性能が向上します。

- B-Treeインデックスのパキューム処理の速度が向上しました。(Heikki Linnakangas, Tom)
- [VACUUM](#)期間中のインデックスを持たないテーブルに対する余計なスキャンを防止します。(Greg Stark)
- 複数列に対する[GiST](#)インデックス作成を改良しました。(Oleg, Teodor)
- B-Treeページの分割を行う前に不要なインデックス項目を削除します。(Junji Teramoto)

E.95.3.2. サーバの変更

- 新しいトランザクションログファイルに強制的に切り替えることができました。(Simon, Tom)

これは、マスタと同期するウォームスタンバイサーバーを維持するために有益なものです。 トランザクションログファイルの切り替えはまた、[pg_stop_backup\(\)](#)時に発生します。 これにより、確実に、復旧に必要なすべてのトランザクションログファイルが即座にアーカイブされるようになります。

- WAL情報関数を追加しました。(Simon)

現在のトランザクションログファイルの挿入箇所を問い合わせる関数、[pg_stop_backup\(\)](#)や関連する関数で表示される16進数のWALの場所からWALファイル名を決定する関数を追加しました。

- WAL再生時のクラッシュからの復旧を改良しました。(Simon)

サーバはWALリカバリ時の定期的なチェックポイントを行うようになりました。 そのため、クラッシュした場合、その後のWALの修復は省かれます。 また、これにより、ウォームスタンバイサーバがクラッシュしたとしても、ベースバックアップからログ全体を再生する必要性もなくなります。

- 長期に渡るWAL再生の信頼性を向上しました。(Heikki, Simon, Tom)

これまでは、2億トランザクション以上をロールフォーワードしようとした場合、XID周回のために正常に動作しませんでした。 これは、ウォームスタンバイサーバが、新しいベースバックアップから定期的にリロードしなおす必要があることを意味します。

- 指定間隔でトランザクションログファイルの切り替えを強制できるように[archive_timeout](#)を追加しました。(Simon)

これは、ウォームスタンバイサーバ用の最大複製遅延を強制します。

- ネイティブな[LDAP](#)認証を追加しました。(Magnus Hagander)

これは特に、WindowsなどPAMをサポートしないプラットフォームで便利です。

- [GRANT CONNECT ON DATABASE](#)を追加しました。(Gevik Babakhani)

これにより、データベースアクセス全体をSQLレベルで制御できます。 これは、既存の[pg_hba.conf](#)による制御の上位に追加されるフィルタとして動作します。

- [SSL証明書失効リスト](#) (CRL)ファイルのサポートを追加しました。(Libor Hohoš)

サーバおよびlibpqがCRLファイルを認識できるようになりました。

- [GiST](#)インデックスがクラスタ可能になりました。(Teodor)

- 定常的な自動パキュームのサーバログ項目を削除しました。(Bruce)

[pg_stat_activity](#)が自動パキュームの状況を表示するようになりました。

- データベース全体ではなく、個々のテーブルで最大XID年代を追跡するようにしました。(Alvaro)

これにより不要なVACUUMを防ぐことができ、トランザクションID周回を防止するためのオーバーヘッドを減少させることができます。

- 前回パキュームした時刻、解析した時刻を統計コレクタの列に追加しました。(Larry Rosenman)

これらの変数は[pg_stat_*_tables](#)システムビューに表示されます。

- 統計処理監視、特にstats_command_stringの性能を向上しました。(Tom, Bruce)

このリリースではデフォルトでstats_command_stringが有効です。 そのオーバーヘッドが最小になっています。 これは[pg_stat_activity](#)がデフォルトですべての有効な問い合わせを表示するようになったことを意味します。

- waiting列を[pg_stat_activity](#)に追加しました。(Tom)

これにより、pg_stat_activityはps表示で含まれるすべての情報を表示できるようになります。

- psの表示を各コマンドで更新するかどうかを制御する[update_process_title](#)設定パラメータを追加しました。(Bruce)

ps表示の更新処理が高価なプラットフォームでは、これを無効にし、ステータス情報をpg_stat_activityに任せの方が良いかもしれません。

- 設定時に単位を指定できるようになりました。(Peter)

例えば、[shared_buffers](#)を、暗算でサイズ変換することなく、32MBと指定できるようになりました。

- postgresql.conf内で[include指示子](#)のサポートを追加しました。(Joachim Wieland)

- プロトコルレベルのprepare/bind/executeメッセージのログ処理を改良しました。(Bruce, Tom)

ログ処理などで、文の名前、バインドパラメータ値、実行される問い合わせテキストを表示するようになりました。 また、log_min_error_statementが有効な場合、問い合わせテキストには、エラーメッセージが適切に含まれます。

- [max_stack_depth](#)が危険な値に設定されることを防止します。

実際のカーネルスタック長の上限が決定できるプラットフォーム（ほとんどの場合）では、max_stack_depthの初期デフォルト値が安全になりました。 また、危険なほど大きな値が設定されたとしても拒絶します。

- 多くの場合で、問い合わせ内のエラー箇所のハイライトが可能になりました。(Tom)

これまでのような基本的な構文エラーだけでなく、サーバが一部の意味的なエラー（不明な列名など）に対しエラー箇所を指定して報告できるようになりました。

- VACUUMにおける"failed to re-find parent key"エラーを修正しました。

- サーバ再起動時、pg_internal.initキャッシュファイルを初期化するようにしました。(Simon)

これは、PITRリカバリの後にキャッシュファイルが無効なデータを含む危険性を防止します。

VACUUMによる1ギガバイト境界を越える大規模リレーションの消去における競合状態を修正しました。(Tom)

- 行レベルロックにおいて不要なデッドロックエラーを発生する不具合を修正しました。(Tom)
- 数ギガバイトのハッシュインデックスに影響する不具合を修正しました。(Tom)
- 各バックエンドプロセスが自身のプロセスグループリーダーとなります。(Tom)

これにより、問い合わせのキャンセルがバックエンドプロセスやアーカイブ/リカバリプロセスから呼び出された子プロセスを中断できるようになりました。

E.95.3.3. 問い合わせの変更

- [INSERT/UPDATE/DELETE](#) RETURNINGを追加しました。(Jonah Harris, Tom)

これにより、これらのコマンドが、新しい行に対して計算された連番キーなどの値を返すことができます。UPDATEの場合、更新されたバージョンの行の値が返されます。

- 標準SQLに従った、複数行VALUES句のサポートを追加しました。(Joe, Tom)

これにより、INSERTは複数の定数行を挿入することができます。また、定数を使用した結果セットを生成する問い合わせを行うこともできます。例えば、INSERT ... VALUES (...), (...), ...やSELECT * FROM (VALUES (...), (...), ...) AS alias(f1, ...)です。

- UPDATEおよびDELETEが対象テーブルの別名を使用できるようになりました。(Atsushi Ogawa)

標準SQLでは、これらのコマンドで別名を許していません。しかし、多くのデータベースシステムでは、便利な記法として許しています。

- UPDATEが値リストで複数列を指定できるようになりました。(Susanne Ebrecht)

基本的にこれは、列と値の組み合わせ割り当てに関する簡略形です。構文は、UPDATE tab SET (column, ...) = (val, ...)です。

- 標準に従った行の比較ができるようになりました。(Tom)

<、<=、>、>=という形式で行を辞書的に比較します。つまり、最初の要素を比較し、もし同じならば、2番目の要素をという順に比較します。以前は、これらは全要素に対するAND条件として展開されました。標準に従っているわけでもなく、特に便利であるわけでもありませんでした。

- TRUNCATEに[CASCADE](#)オプションを追加しました。(Joachim Wieland)

これにより、TRUNCATEは自動的に指定したテーブルを外部キー経由で参照するすべてのテーブルを含めるようになります。便利ではありますが、危険なものです。注意して使用してください。

- 同一SELECTコマンドでFOR UPDATEとFOR SHAREをサポートします。(Tom)

- [IS NOT DISTINCT FROM](#)を追加しました。(Pavel Stehule)

この演算子は等価演算子(=)と似ています。しかし、NULLを返すのではなく、右辺と左辺がNULLの場合に真と、片方だけがNULLの場合に偽と評価します。

- [UNION/INTERSECT/EXCEPT](#)で使用する長さ出力を改良しました。(Tom)

すべての対応する列が同じ長さで定義された場合、一般的な長さではなく、その定義された長さが結果として使用されます。

- [ILIKE](#)がマルチバイト符号化方式で動作できるようになりました。(Tom)

内部的には、ILIKEがlower()を呼び出した後にLIKEを使用するようになりました。こうした符号化方式では、ロケール固有の正規表現パターンはまだ正常に動作しません。

- [standard_conforming_strings](#)をonにできるようになりました。(Kevin Grittner)

これにより、文字列内のバックスラッシュエスケープは無効になり、PostgreSQLがより標準に準拠するようになりました。後方互換性のためデフォルトはoffですが、将来はonがデフォルトになる予定です。

- 対象リストにvolatile関数が含まれている場合に副問い合わせを平坦化しません。(Jaime Casanova)

これは、volatile関数(random()やnextval()など)の複数回の評価による想定外の振る舞いを防止します。不必要にvolatileと宣言された関数があると、性能劣化をもたらす可能性があります。

- 準備済み文と開いているカーソルを示す[pg_prepared_statements](#)および[pg_cursors](#)システムビューを追加しました。(Joachim Wieland, Neil)

接続プールを設定している場合に非常に有用です。

- [EXPLAIN](#)および[EXECUTE](#)においてポータルパラメータをサポートします。(Tom)

これにより、例えばJDBCの?パラメータでこれらのコマンドを動作させることができます。

- SQLレベルで[PREPARE](#)パラメータが指定されない場合、問い合わせの文脈から型を推定します。(Neil)

プロトコルレベルのPREPAREでは既の実現していました。

- LIMITおよびOFFSETが200億以上を使用できるようになりました。(Dhanaraj M)

E.95.3.4. オブジェクト操作における変更

- [CREATE TABLE AS](#)にTABLESPACE句を追加しました。(Neil)

これにより新しいテーブルを作成するときにテーブル空間を指定できます。

- [CREATE TABLE AS](#)にON COMMIT句を追加しました。(Neil)

これにより、一時テーブルをトランザクションのコミット時に消去したり削除したりすることができます。デフォルトの動作では、セッションが終了するまでテーブルは残ります。

- [CREATE TABLE LIKE](#)にINCLUDING CONSTRAINTSを追加しました。(Greg Stark)

これにより、新しいテーブルに簡単にCHECK制約をコピーすることができます。

- ブレースホルダ(シェル)型を生成できます。(Martijn van Oosterhout)

シェル型の宣言は、型の仕様を指定せずに型の名前を生成します。シェル型の作成は、型を"実際に"定義する前に存在しなければならない、型の入出力関数の宣言を整理することができる点で有用です。構文は、CREATE TYPE typenameです。

- 集約関数が複数の入力パラメータをサポートするようになりました。(Sergey Kopolov, Tom)

- 新しい集約関数作成[構文](#)を追加しました。(Tom)

新しい構文は`CREATE AGGREGATE aggrname (input_type) (parameter_list)`です。これはより自然に、新しい複数パラメータ集約機能をサポートします。これまでの構文もまだサポートします。

- 過去に設定したロールパスワードを削除する[ALTER ROLE PASSWORD NULL](#)を追加しました。(Peter)

- 多くのオブジェクト種類について、`IF EXISTS`付きの[DROP](#)操作を追加しました。

これにより、オブジェクトが存在しなくてもエラーとなることなく、[DROP](#)操作を行うことができます。

- ロールが所有するすべてのオブジェクトを削除する[DROP OWNED](#)を追加しました。(Alvaro)

- あるロールが所有するすべてのオブジェクトの所有権を再割り当てする[REASSIGN OWNED](#)を追加しました。(Alvaro)

これと上述の[DROP OWNED](#)によりロールの削除が簡単になります。

- [GRANT ON SEQUENCE](#)構文を追加しました。(Bruce)

シーケンス固有の権限設定のために追加されました。シーケンスに対する[GRANT ON TABLE](#)も後方互換のためにまだサポートされています。

- シーケンスに対し[currval\(\)](#)および[nextval\(\)](#)のみを許可し、[setval\(\)](#)を許可しない[USAGE](#)権限を追加しました。(Bruce)

[USAGE](#)権限により、シーケンスに対するアクセス制御を細かく制御できるようになります。[USAGE](#)を付与することで、シーケンスを増やすことはできるが、[setval\(\)](#)を使用して任意の値に設定することはできないユーザを作成することができます。

- [ALTER TABLE \[NO \] INHERIT](#)を追加しました。(Greg Stark)

これにより、テーブルの生成と削除時だけではなく、動的に継承を調整できるようになります。テーブル分割を行うために継承を使用する時に非常に有用です。

- グローバルオブジェクトに対する[コメント](#)をグローバルに保管することができます。(Kris Jurka)

これまでは、データベースに対するコメントは個々のデータベースに格納され、無効なものとしていました。また、ロールやテーブル空間に対するコメントを付与することはまったくできませんでした。この変更では、新しい共有カタログ[pg_shdescription](#)を追加し、そこにデータベース、ロール、テーブル空間に対するコメントを保管します。

E.95.3.5. ユーティリティコマンドの変更点

- テーブルへの同時書き込みをブロックせずにインデックスを作成できるオプションを追加しました。(Greg Stark, Tom)

新しい構文は[CREATE INDEX CONCURRENTLY](#)です。デフォルトでは、まだインデックス作成中のテーブルの変更はブロックされます。

- [勧告的ロック](#)機能を提供します。(Abhijit Menon-Sen, Tom)

これは、`/contrib/userlock`で使用されていたものを置き換えるために設計された、新しいロック処理APIです。`userlock`のコードはpgfoundryに移しました。

- [COPY](#)が[SELECT](#)問い合わせをダンプできるようになりました。(Zoltan Boszormenyi, Karel Zak)

これにより[COPY](#)は任意のSQL問い合わせをダンプできるようになりました。構文は[COPY \(SELECT ...\) TO](#)です。

- [COPY](#)がコピーされた行数を含むコマンドタグを返すようになりました。(Volkan YAZICI)

- [VACUUM](#)が同時実行中の他の[VACUUM](#)操作の影響を受けることなく行をエキスパイアできるようになりました。(Hannu Krossing, Alvaro, Tom)

- [initdb](#)がオペレーティングシステムのロケールを検出し、それに合う[DateStyle](#)のデフォルトを設定するようになりました。(Peter)

これにより、インストールされる[postgresql.conf](#)内の[DateStyle](#)がより好ましいものになるようになりました。

- [initdb](#)が表示する進行メッセージ数を減らしました。(Tom)

E.95.3.6. 日付時刻に関する変更点

- [timestamp](#)の入力値に完全な時間帯名称を指定できます。(Joachim Wieland)

例えば、`'2006-05-24 21:11 America/New_York':::timestamptz`です。

- 設定変更可能な時間帯省略形をサポートします。(Joachim Wieland)

使用したい時間帯省略形の集合を[timezone_abbreviations](#)設定/パラメータで選択できます。

- サポートされる時間帯を示す、[pg_timezone_abbrevs](#)および[pg_timezone_names](#)ビューを追加しました。(Magnus Hagander)

- [clock_timestamp\(\)](#)、[statement_timestamp\(\)](#)、[transaction_timestamp\(\)](#)を追加しました。(Bruce)

[clock_timestamp\(\)](#)は実際の現在時刻を、[statement_timestamp\(\)](#)は現在の問い合わせがサーバに届いた時刻を表します。[transaction_timestamp\(\)](#)は[now\(\)](#)の別名です。

- [to_char\(\)](#)が翻訳された月日名称を表すことができるようになりました。(Euler Taveira de Oliveira)

- [to_char\(time\)](#)および[to_char\(interval\)](#)でAM/PM指定の出力ができるようになりました。(Bruce)

時間間隔および時間が24時間周期で扱われます。例えば、`25 hours`はAMとみなされます。

- 時間間隔の単位を調整する[justify_interval\(\)](#)関数を追加しました。(Mark Dilger)

- GMTから14:59離れたオフセットを持つ時間帯を持つことができます。

キリバス共和国ではGMT+14が使用します。これを受け付けられるようにしました。

- 時間間隔計算を改良しました。(Michael Glaesemann, Bruce)

E.95.3.7. その他、データ型および関数の変更

- 配列がNULL要素を含むことができます。(Tom)

- 連続して項目が存在しない配列要素への代入が可能です。(Tom)

空いている配列位置はNULLで埋められます。これは標準SQLに従います。

- 部分配列の比較に関して、新規の組み込み[演算子](#) (`@>`、`<@`、`&@`)を追加しました。(Teodor, Tom)

これらの演算子は任意のデータ型に対して、GiSTまたはGINインデックスを使用してインデックス付けできます。

- INET/CIDR値に対して便利な算術操作を追加しました。

新しい演算子は& (論理積)、|(論理和)、~ (否定)、inet + int8、inet - int8、inet - inetです。

- SQL:2003の集約関数を新規に追加しました。

新しい関数は、var_pop()、var_samp()、stddev_pop()、stddev_samp()です。var_samp()およびstddev_samp()は、既存の集約関数variance()とstddev()の名前を変更したものに過ぎません。後方互換のために、後者の名前も残っています。

- SQL:2003の統計処理集約を追加しました。(Sergey Kopolov)

新しい関数

は、regr_intercept()、regr_slope()、regr_r2()、corr()、covar_samp()、covar_pop()、regr_avgx()、regr_avgy()、regr_sxy()、regr_sxx()、regr_syy()、regr_count()です。

- 他のドメインに基づいたドメインが可能になりました。(Tom)

- すべての所で適切にドメインCHECK制約が適切に強制されるようになりました。(Neil, Tom)

例えば、ドメイン型を返すものと宣言されたユーザ定義関数の結果がドメインの制約に対して検査されるようになりました。これはドメイン実装における重大な抜け穴をなくしました。

- 名前が変更されたSERIAL列のダンプ処理の問題を解消しました。(Tom)

この修正は、明示的にDEFAULTが指定されたSERIAL列と、そのシーケンス要素をダンプし、新しいALTER SEQUENCE OWNED BYコマンドを使用してリロード時にSERIAL列を再構成するためのものです。また、これによりSERIAL列の仕様が削除できるようになりました。

- サーバサイドのスリープ関数pg_sleep()を追加しました。(Joachim Wieland)

- tid (タプルID)データ型の比較演算子をすべて追加しました。(Mark Kirkwood, Greg Stark, Tom)

E.95.3.8. PL/pgSQLサーバサイド言語の変更点

- TG_table_nameおよびTG_table_schemaをトリガパラメータに追加しました。(Andrew)

TG_relnameは廃止予定です。トリガパラメータにおける同様の変更が他のPLでもなされています。

- FOR文を使用して、レコード型や行型同様にスカラ値を返すことが可能です。(Pavel Stehule)

- 繰り返しの増分を制御できるようにFORループにBY句を追加しました。(Jaime Casanova)

- STRICTをSELECT INTOに追加しました。(Matt Miller)

Oracle PL/SQLとの互換性のために、STRICTで、SELECTによって返される行が正確に1行でない場合に例外を返すようになりました。

E.95.3.9. PL/Perlサーバサイド言語における変更点

- table_nameおよびtable_schemaをトリガパラメータに追加しました。(Adam Sjøgren)

- 準備済み問い合わせを追加しました。(Dmitry Karasik)

- \$_TDトリガデータをグローバル変数にしました。(Andrew)

以前はレキシカル変数でした。このため想定外の共有違反が起きました。

- セキュリティ対策として、PL/PerlとPL/PerlUが別のインタプリタで動作するようになりました。(Andrew)

このため、共有データやロード済みモジュールを両者で共有することができません。また、Perlが複数のインタプリタが可能となるように必要なフラグをつけてコンパイルされていない場合、2つの内1つしかあるバックエンドプロセスで使用することができません。

E.95.3.10. PL/Pythonサーバサイド言語における変更点

- 名前付きパラメータが、args[]配列同様、普通の変数として渡されます。(Sven Suursoho)

- table_nameおよびtable_schemaをトリガパラメータに追加しました。(Andrew)

- 複合型および結果セットを返すことが可能です。(Sven Suursoho)

- list、iterator、generatorとして結果セットを返します。(Sven Suursoho)

- 関数はvoidを返すことができます。(Neil)

- Python 2.5をサポートしました。(Tom)

E.95.3.11. psqlにおける変更点

- クライアント側のパスワード暗号化を使用してロールのパスワードを変更する\passwordコマンドを新規に追加しました。(Peter)

- \cにより、新規にホスト、ポート番号を指定して接続することができます。(David, Volkan YAZICI)

- テーブル空間の表示をする\+を追加しました。(Philip Yarra)

- 関数の引数名とモード(OUTまたはINOUT)を含めるように\dfスラッシュコマンドを改良しました。(David Fetter)

- バイナリCOPYをサポートしました。(Andreas Pflug)

- 単一トランザクション内でセッション全体を実行させるオプションを追加しました。(Simon)

-1または--single-transactionを使用してください。

- カーソルを使用したバッチ内で自動的にSELECTの結果を取り出せるようにしました。(Chris Mair)

これは\set FETCH_COUNT nを使用して実現します。この機能により、メモリ内に結果セット全体を保存することなく、大規模な結果セットをpsql内で取り出すことができます。

複数行に渡る値を適切な列数で整列します。(Martijn van Oosterhout)

改行を含むフィールド値がより読みやすく表示されます。

- 複数行に渡る文を、1行ごとではなく、単一の文と同様に保存します。(Sergey E. Kopolov)

カーソル行による問い合わせの再呼び出しをより簡単に行うことができます。(これはWindowsでは不可能です。このプラットフォームでは、オペレーティングシステムに組み込まれたコマンドライン編集機能を使用しているためです。)

- 行カウンタを64ビットにしました。このため200億行以上のファイルを扱うことが可能です。(David Fetter)

- `INSERT/UPDATE/DELETE RETURNING` で返されるデータとコマンドステータスタグの両方を表示します。(Tom)

E.95.3.12. [pg_dump](#)における変更点

- `pg_dump`による含有または除外するオブジェクトの選択ができるようになりました。(Greg Sabino Mullane)

`pg_dump`は、複数の`-n` (スキーマ)と複数の`-t` (テーブル) オプションをサポートするようになりました。また、オブジェクトを除外する`-N`および`-T`オプションも追加されました。また、これらのスイッチの引数に、単一のオブジェクト名ではなく、例えば、`-t 'foo*'`といったワイルドカード表現を指定できるようになりました。更に、スキーマを`-t`や`-T`スイッチの一部、例えば`-t schema1.table1`として指定できるようになりました。

- テーブル作成に失敗(つまり、テーブルが既に存在する)した場合にデータのロードを中断する`--no-data-for-failed-tables`オプションを[pg_restore](#)に追加しました。(Martin Pitt)

- 単一トランザクションでセッション全体を実行させるオプションを[pg_restore](#)に追加しました。(Simon)

このためには`-1`または`--single-transaction`を使用してください。

E.95.3.13. [libpq](#)における変更点

- パスワード暗号化のために[PQencryptPassword\(\)](#)を追加しました。(Tom)

これによりパスワードを事前に暗号化して`ALTER ROLE ... PASSWORD`などのコマンドに渡すことができます。

- [PQisthreadsafe\(\)](#)関数を追加しました。(Bruce)

これによりアプリケーションはライブラリのスレッド安全性を問い合わせることができます。

- 準備済み文や開いているカーソルに関する情報を返す、[PQdescribePrepared\(\)](#)、[PQdescribePortal\(\)](#)、その他関連する関数を追加しました。(Volkan YAZICI)

- `pg_service.conf`から[LDAP](#)検索が可能です。(Laurenz Albe)

- `~/pgpass`内のホスト名をデフォルトのソケットディレクトリと一致できるようにしました。(Bruce)

ホスト名が空の場合、続きがUnixソケット接続と比較されます。この追加によりマシン上の複数のサーバプロセスの1つを特定する項目を持つことができます。

E.95.3.14. [ecpg](#)における変更点

- `SHOW`はその結果を変数に格納することができます。(Joachim Wieland)

- `COPY TO STDOUT`を追加しました。(Joachim Wieland)

- リグレーション試験を追加しました。(Joachim Wieland, Michael)

- ソースコードの多くを整理しました。(Joachim Wieland, Michael)

E.95.3.15. Windows移植

- MSVCでPostgreSQLサーバをコンパイルできるようにしました。(Magnus, Hiroshi Saito)

- ユーティリティコマンドや[pg_dump](#)のMSVCサポートを追加しました。(Hiroshi Saito)

- Windowsコードページ1253、1254、1255、1257のサポートを追加しました。(Kris Jurka)

- サーバが管理アカウントから起動できるように起動時の権限をなくしました。(Magnus)

- 安定性を向上させました。(Qingqing Zhou, Magnus)

- ネイティブなセマフォを実装しました。(Qingqing Zhou)

これまでのコードはSysVセマフォをまねたものでした。

E.95.3.16. ソースコードにおける変更点

- [GIN](#) (汎用転置インデックス) インデックスアクセスメソッドを追加しました。(Teodor, Oleg)

- R-treeインデックスを削除しました。(Tom)

R-treeは[GiST](#)を使用して再実装されました。とりわけ、これは、R-treeインデックスで先行書き込みログ(WAL)を使用したクラッシュからの復旧が可能になったことを意味します。

- バックエンドでリンクが必要となるライブラリを減少しました。(Martijn van Oosterhout, Tom)

- GNU readlineよりもlibeditを優先させるフラグをconfigureに追加しました。(Bruce)

`--with-libedit-preferred`を使用してconfigureを実行してください。

- 空白文字を含むディレクトリにインストールできるようにしました。(Peter)

- インストールディレクトリの再割り当て機能を改良しました。(Tom)

- Solarisコンパイラを使用したSolaris x86_64のサポートを追加しました。(Pierre Girard, Theo Schlossnagle, Bruce)

- DTraceサポートを追加しました。(Robert Lor)

- サードパーティアプリケーションが、C言語で>、<比較を使用してバックエンドのバージョンを検査できるように[PG_VERSION_NUM](#)を追加しました。(Bruce)

- BLCKSZと独立して、XLOG_BLCKSZを追加しました。(Mark Wong)
- ロック状況を報告するためのLWLOCK_STATS定義を追加しました。(Tom)
- 不明なconfigureオプションに関する警告を出力します。(Martijn van Oosterhout)
- デバッグや性能測定などの追加作業に使用することができる"plugin"ライブラリのサーバサポートを追加しました。(Korry Douglas)

これは2つの機能からなります。1つは、"ランデブー変数"のテーブルであり、これにより別々にロードされた共有ライブラリ間で通信を行うことができます。2つ目は、新しい`local_preload_libraries`設定パラメータであり、これにより、クライアントアプリケーションから明示的な協力をすることなく、ライブラリを指定したセッションにロードさせることができます。これにより、PL/pgSQLデバッグなど、外部アドオンを実装機能を実装することができます。

- 既存の`preload_libraries`設定パラメータを`shared_preload_libraries`に名前を変更しました。(Tom)

これは、`local_preload_libraries`との違いを明確にするために行いました。

- 新しい`server_version_num`設定パラメータを追加しました。(Greg Sabino Mullane)

これは`server_version`と似ていますが、例えば80200といった整数値です。これによりアプリケーションはバージョン検査をより簡単に行うことができます。

- `seq_page_cost`設定パラメータを追加しました。(Tom)
- Cプログラムとして`リグレッション試験`スクリプトを再実装しました。(Magnus, Tom)
- ロード可能モジュールが共有メモリと軽量ロックを割り当てできるようにしました。(Marc Munro)
- 動的にロードしたライブラリの自動的な初期化処理と最終処理を追加しました。(Ralf Engelschall, Tom)

新しい関数 `PG_init()` と `PG_fini()` が、これらのシンボルを定義したライブラリで、呼び出されます。したがって、`shared_preload_libraries`内で初期化関数を指定する必要がなくなりました。こうしたライブラリでは代わりに`_PG_init()`規約が使用されるものと仮定することができます。

- すべての共有オブジェクトファイルに`PG_MODULE_MAGIC`ヘッダブロックを追加しました。(Martijn van Oosterhout)

このマジックブロックは、ロード可能オブジェクトファイルとサーバとの間のバージョン不整合を防止するものです。

- AIXでの共有ライブラリサポートを追加しました。(Laurenz Albe)
- 新しいXML文書セクションを追加しました。(Bruce)

E.95.3.17. Contrib における変更点

- tsearch2が大きく改良されました。(Oleg, Teodor)
 - UTF8を含むマルチバイト符号化方式のサポート。
 - 問い合わせ書き換えのサポート。
 - ランク付け関数の改良。
 - シソーラス辞書のサポート
 - Ispell辞書が、OpenOfficeで使用するMySpell書式を認識するようになりました。
 - GINのサポート
- Pgadmin管理関数を含むadminpackモジュールを追加しました。

これらの関数は、デフォルトのPostgreSQLサーバには存在しない、追加のファイルシステムアクセス処理を提供します。
- sslinfoモジュールを追加しました。(Victor Wagner)

現在の接続におけるSSL証明に関する情報を表示します。
- pgrowlocksモジュールを追加しました。

これは、指定したテーブルに関する行ロック情報を表示します。
- hstoreモジュールを追加しました。(Oleg, Teodor)
- isbn_issnを置き換えて、isnモジュールを追加しました。(Jeremy Kronuz)

この新しい実装では、EAN13、UPC、ISBN (書籍)、ISMN (音楽)、ISSN (シリアル番号)をサポートします。
- pgstattupleにインデックス情報関数を追加しました。(ITAGAKI Takahiro, Satoshi Nagayasu)
- 空き領域マップ情報を表示する`pg_freemap`を追加しました。(Mark Kirkwood)
- pgcryptoが予定していたすべての機能を持つようになりました。(Marko Kreen)
 - 公開キー暗号関数が常に利用できるようにiMathライブラリをpgcryptoに含めました。
 - OpenBSD用のコードで存在しなかったSHA224アルゴリズムを追加しました。
 - 古めのOpenSSLにおけるSHA224/256/384/512ハッシュ用の組み込みコードを有効化し、常にこれらのアルゴリズムが利用できるようになりました。
 - 暗号化に使用される乱数の強度を返す`gen_random_bytes()`関数を追加しました。暗号化キーの生成に便利です。
 - `digest_exists()`、`hmac_exists()`、`cipher_exists()`関数を削除しました。
- cubeモジュールを改良しました。(Joshua Reich)

`cube(float[])`、`cube(float[], float[])`、`cube_subset(cube, int4[])`関数を追加しました。
- dblinkに非同期問い合わせ機能を追加しました。(Kai Londenberg, Joe Conway)
- 部分配列比較用に新しい演算子 (`@>`、`<@`、`@@`) を追加しました。(Tom)

各種contribパッケージで、独自のデータ型用に既にこれらの演算子を持っていましたが、命名に一貫性がありませんでした。コアコードと、こうした機能を持つcontribパッケージすべてに、一貫性を持った名前の部分配列の比較演算子を追加しました。

- インストールスクリプトを持つすべてのcontribパッケージに対してアンインストール用のスクリプトを追加しました。(David, Josh Drake)

E.96. リリース8.1.23

リリース日: 2010-12-16

このリリースは8.1.22に対し、各種の不具合を修正したものです。 8.1メジャーリリースにおける新機能については[項E.119](#)を参照してください。

これはPostgreSQL 8.1.X系列の最終リリースとなるはずです。 ユーザにはより新しい系列のリリースへの更新を強く勧めます。

E.96.1. バージョン8.1.23への移行

8.1.Xからの移行ではダンプ/リストアは不要です。 しかし8.1.18以前のバージョンからアップデートする場合は8.1.18のリリースノートを参照してください。

E.96.2. 変更点

- Linuxにて[wal_sync_method](#)のデフォルトが強制的に[fdatasync](#)になるようにしました。(Tom Lane, Marti Raudsepp)

長い間、Linuxでのデフォルトは実質[fdatasync](#)でした。 しかし最近のカーネルの変更により、PostgreSQLは代わりに[open_datasync](#)を選択するようになりました。 この選択は性能向上をまったくもたらず、特定のシステム、特に[data=journal](#) マウントオプションを使用した[ext4](#)では、完全に失敗してしまいます。

- 開始チェックポイントWALレコードが再実行時点と同じWALセグメント内に存在しない場合のベースバックアップからのリカバリを修正しました。(Jeff Davis)

- IA64においてレジスタスタックのオーバーラン検知サポートを追加しました。(Tom Lane)

IA64アーキテクチャには2つのハードウェアスタックがあります。 スタックオーバーラン失敗を完全に防ぐには両方を検査しなければなりません。

- [copyObject\(\)](#) におけるスタックオーバーフロー検査を追加しました。(Tom Lane)

非常に複雑な問い合わせが与えられた場合のスタックオーバーフローのため、特定のコードパスがクラッシュする可能性があります。

- 一時GiSTインデックスにおけるページ分割検知を修正しました。(Heikki Linnakangas)

例えば挿入が行われた時に一時インデックスをスキャンする開いているカーソルがあると、そのインデックスで"同時実行"ページ分割を持つことができます。 GiSTではこの状況の検知に失敗していました。 このためGiSTはそのカーソルの実行が続く間、間違った結果をもたらすことがありました。

- 複雑なインデックス式を[ANALYZE](#) する時のメモリリークを防ぎます。(Tom Lane)

- 行全体のVarを使用するインデックスがそのテーブルに依存し続けることを確実にしました。(Tom Lane)

`create index i on t (foo(t.*))` のように宣言されたインデックスはそのテーブルが削除された時に自動的に削除されませんでした。

- 複数のOUTパラメータを持つSQL関数を"インライン"化しません。(Tom Lane)

結果の想定行型に関する情報が存在しないためにクラッシュする可能性を防ぎます。

- [COALESCE\(\)](#) 式の定数折りたたみを修正しました。(Tom Lane)

プランナは時々、実際には決して達しない副式の評価を行い、想定外のエラーをもたらす可能性がありました。

- [InhRelation](#) ノードに関する機能性出力を追加しました。(Tom Lane)

これにより[debug_print_parse](#)が有効な場合に特定種類の問い合わせが行われた場合の失敗を防止します。

- 点から水平線分への距離計算を正しく修正しました。(Tom Lane)

この不具合は様々な複数の幾何距離測定演算子に影響します。

- 再帰もしくはエラー修復状態にならない"単純な"式についてのPL/pgSQLの扱いを修正しました。(Tom Lane)
- contrib/cubeのGiST picksplitアルゴリズムの不具合を修正しました。(Alexander Korotkov)

cube列上のGiSTインデックスにおいて、実際に不正確な結果になることはありませんが、このため相当効率性が劣化する可能性があります。こうしたインデックスがある場合、この更新をインストールした後にREINDEXすることを検討してください。

- contrib/dblink内で、新しい接続を作成する時を除き、"identifier will be truncated"注意を出力しないようにしました。(Itagaki Takahiro)
- contrib/pgcryptoにおいて公開キーがない場合にコアダンプする可能性を修正しました。(Marti Raudsepp)
- contrib/xml2のXPath問い合わせ関数におけるメモリリークを修正しました。(Tom Lane)
- 時間帯データファイルをtzdata release 2010oに更新しました。フィジィ、サモアにおける夏時間規則の変更、香港の歴史的な修正が含まれています。

[前のページ](#)
リリース 8.2

[ホーム](#)
[上に戻る](#)

[次のページ](#)
リリース8.1.22

E.97. リリース8.1.22

リリース日: 2010-10-04

このリリースは8.1.21に対し、各種の不具合を修正したものです。 8.1メジャーリリースにおける新機能については[項E.119](#)を参照してください。

PostgreSQLコミュニティは2010年11月に8.1.X系リリースの更新を停止する予定です。 ユーザには早期により新しい系列のリリースへの更新を強く勧めます。

E.97.1. バージョン8.1.22への移行

8.1.Xからの移行ではダンプ/リストアは不要です。 しかし8.1.18以前のバージョンからアップデートする場合は8.1.18のリリースノートを参照してください。

E.97.2. 変更点

- PL/PerlおよびPL/Tclにおいて、呼び出し元のSQLユーザIDごとに別のインタプリタを使用します。(Tom Lane)

この変更は、同じセッション内でその後に異なるSQLユーザIDにより実行されるPerlまたはTclコード（例えばSECURITY DEFINERの内部）を破壊することで発生する可能性があるというセキュリティ問題を防止するものです。 ほとんどのスクリプト用言語は、対象とする関数で呼び出される標準関数や演算子の再定義など、これを行うことができる多くの方法を提供します。 この変更がないと、PerlまたはTcl言語の使用権限を持つ任意のSQLユーザは基本的に、対象の関数の所有者が持つSQL権限でできることを何でもできるようになります。

この変更の欠点はPerlおよびTcl関数間の内部通信がより難しくなるという点です。 逃げ道を提供するために、PL/PerlUおよびPL/TclU関数ではまだセッション当たり1つのインタプリタのみを使用します。 これらのすべての関数は従来よりデータベーススーパーユーザの信頼レベルで実行されますので、これはセキュリティ問題とはみなされません。

信頼された実行を提供することを主張するサードパーティ製の手続き言語は似たようなセキュリティ問題を持つ可能性が大きくあります。 セキュリティが非常に重要な目的のために依存している手続き言語があれば、その作者に連絡を取ることを勧めます。

この問題を指摘いただいたTim Bunceに感謝します(CVE-2010-3433)。

- pg_get_expr() の用途として意図されたシステムカタログ列の一つではない引数で呼び出されることを拒絶することでpg_get_expr() 内のクラッシュする可能性を防止します。(Heikki Linnakangas, Tom Lane)
- "cannot handle unplanned sub-select"エラーを修正しました。(Tom Lane)

これは、副選択が他の副選択を含む式に展開される結合の別名参照を含む場合に発生しました。

- 自動バキューム処理中のクラッシュからshow_session_authorization()を防止します。(Tom Lane)
- 返される行のすべてが実際に同一の行型でない、レコードの集合を返す関数を防御します。(Tom Lane)
- 参照渡し関数の結果をハッシュ化する時に失敗する可能性を修正しました。(Tao Ma, Tom Lane)
- ロックファイル (postmaster.pidおよびソケット用のロックファイル) を書き出す時に、注意してその内容のfsyncします。(Tom Lane)

これがないと、postmasterの起動後しばらくしてマシンがクラッシュした場合に、ロックファイルの内容が破損することがあり得ました。 これが起きると、このロックファイルを手作業で削除するまで、その後のpostmasterの起動試行が成功できなくなります。

- 過度に入れ子になった副トランザクションに対してXIDを割り当てる際の再帰を防止します。(Andres Freund, Robert Haas)

元々のコードでは、制限されたスタック領域であった場合にクラッシュすることがあり得ました。

- バックエンド起動の初期にゴミを生成することがあったlog_line_prefixの%iエスケープを修正しました。(Tom Lane)

- アーカイブ処理が有効な場合、`ALTER TABLE ... SET TABLESPACE`におけるデータ破損の可能性を修正しました。(Jeff Davis)
- `CREATE DATABASE`と`ALTER DATABASE ... SET TABLESPACE`を問い合わせキャンセルにより中断させることができます。(Guillaume Lelarge)
- PL/Pythonにおいて、`PyObject_AsVoidPtr`および`PyObject_FromVoidPtr`からのヌルポインタ結果を防止します。(Peter Eisentraut)
- 削除された列を含むテーブルに対する`contrib/dblink`の処理を改良しました。(Tom Lane)
- `contrib/dblink`において"duplicate connection name"エラー後の接続リークを修正しました。(Itagaki Takahiro)
- 62バイトより長い接続名に関する`contrib/dblink`の処理を正しく修正しました。(Itagaki Takahiro)
- 構築手順や文書を更新して、ソースコードリポジトリがCVSからGitに変更されたことを反映しました。(Magnus Hagander他)
- 時間帯データファイルをtzdata release 2010Iに更新しました。 エジプト、パレスチナにおける夏時間規則の変更、フィンランドの歴史的な修正が含まれています。

またこの変更では、2つのミクロネシアの時間帯を追加しています。 Pacific/ChuukはPacific/Trukより好まれるようになります(好まれる省略形はTRUTではなくCHUT)、Pacific/PohnpeiはPacific/Ponapeより好まれるようになります。

[前のページ](#)

リリース8.1.23

[ホーム](#)

[上に戻る](#)

[次のページ](#)

リリース8.1.21

E.98. リリース8.1.21

リリース日: 2010-05-17

このリリースは8.1.20に対し、各種の不具合を修正したものです。 8.1メジャーリリースにおける新機能については[項E.119](#)を参照してください。

E.98.1. バージョン8.1.21への移行

8.1.Xからの移行ではダンプ/リストアは不要です。 しかし8.1.18以前のバージョンからアップデートする場合は8.1.18のリリースノートを参照してください。

E.98.2. 変更点

- `Safe.pm`を使用するのではなく、インタプリタ全体に適用される`opmask`を使用して`plperl`を強制的に制限します。(Tim Bunce, Andrew Dunstan)

最近の開発により`plperl`を信頼できるとする根拠とするには`Safe.pm`が非常に安全ではないことを知らされました。 この変更により、`Safe.pm`の使用がなくなり、常に適用されるopcodeマスクを持つ別のインタプリタを使用するようになります。 この変更の喜ばしい副作用として、`plperl`内で普通の方法でPerlの`strict`プラグマが使用できるようになりました。 また、Perlの`$a`と`$b`がソート関数で想定される動作を行うようになり、関数のコンパイルが非常に高速になりました。(CVE-2010-1169)
- PL/Tclにおいて`pltcl_modules`経由で信頼できないコードが実行されることを防止しました。(Tom)

PL/TclのデータベーステーブルからTclコードを自動読み込みする機能はトロイの木馬攻撃によりつけこまれる可能性があります。 そのテーブルの作成や挿入できるユーザに対する制限がなかったためです。 この変更により、`pltcl_modules`がスーパーユーザにより所有されていない限り、この機能は無効になります。(しかし、このテーブルに対する権限は検査されませんので、安全性が低いモジュールを本当に必要とするインストレーションでは、スーパーユーザ以外に適切な権限を付与することができます。) また、`pltclu`関数を本当に実行しようとしていない限り、制限がない"通常の"Tclにコードが読み込まれることを防止しました。(CVE-2010-1170)
- 非特権ユーザがスーパーユーザのみのパラメータ設定を再設定することを許しません。(Alvaro)

これまで、非特権ユーザが自身に対する`ALTER USER ... RESET ALL`または自身が所有するデータベースに対する`ALTER DATABASE ... RESET ALL`を行った場合、そのユーザまたはデータベースに対する特別なパラメータ設定は、たとえスーパーユーザのみにより変更可能なものとして提供されるものであっても、すべて削除されました。 `ALTER`はそのユーザが変更権限を持つパラメータのみを削除するようになりました。
- `CONTEXT`追加がログ項目になされる時にシャットダウンが発生した場合に、バックエンドのシャットダウン時にクラッシュする可能性を防ぎました。(Tom)

現在のトランザクションがログメッセージを出力する時点に達した時にすでにロールバックされているために文脈出力関数が失敗することがありました。
- `pl/perl`の`ppport.h`を最近のPerlバージョンに更新しました。(Andrew)
- `pl/python`内の雑多なメモリリークを修正しました。(Andreas Freund, Tom)
- 変数展開が自身を参照している際の`psql`における無限再帰を防止します。(Tom)
- `contrib/pgstattuple`関数がキャンセル割り込みに即座に応答することを確実にしました。(Tatsuhito Kasahara)
- サーバ起動時に、既存の共有メモリセグメントに対して`shmget()`が`EINVAL`を返す状況を適切に扱うようにしました。(Tom)

この振舞いはOS Xを含むBSD派生のカーネルにおいて観測されました。 共有メモリ要求サイズが大きすぎるという警告の、完全に間違った起動失敗という結果になりました。
- 時間帯データファイルを`tzdata release 2010j`に更新しました。 アルゼンチン、南極におけるオーストラリア、バンラデシュ、メキシコ、モロッコ、パキスタン、パレスチナ、ロシア、シリア、チュニジアにおける夏時間規則の変更、および、台湾の歴史的な修正が含まれます。

E.99. リリース8.1.20

リリース日: 2010-03-15

このリリースは8.1.19に対し、各種の不具合を修正したものです。 8.1メジャーリリースにおける新機能については[項E.119](#)を参照してください。

E.99.1. バージョン8.1.20への移行

8.1.Xからの移行ではダンプ/リストアは不要です。 しかし8.1.18以前のバージョンからアップデートする場合は8.1.18のリリースノートを参照してください。

E.99.2. 変更点

- SSL接続においてセッションキーの再調停をどの程度の頻度で行うかを制御する`ssl_renegotiation_limit`設定パラメータを新しく追加しました。(Magnus)

これをゼロに設定することで、再調停を完全に無効にすることができます。 これは壊れたSSLライブラリを使用している場合に必要になる可能性があります。 具体的には一部のベンダは再調停の試行を失敗させるCVE-2009-3555に対する一時しのぎのパッチを提供しています。

- 副トランザクション起動時における失敗からのリカバリを試行する時にクラッシュする可能性を修正しました。(Tom)
- セーブポイントの使用およびサーバの符号化方式と異なるクライアント符号化方式に関連したサーバのメモリリークを修正しました。(Tom)
- `bit`型の`substring()`では、いかなる負の長さも"文字列の残りすべて"を意味するようにしました。(Tom)
これまでのコードでは-1のみを上のように扱い、他の負の値についてはクラッシュする可能性がある無効な結果値を生成しました。(CVE-2010-0442)
- 出力ビット幅が指定された整数よりも広く、それが8ビットの倍数以外である場合、先頭の端数のバイトを正確に扱うように整数からビット列への変換を修正しました。(Tom)
- 正規表現マッチが異常なまでに低速になる一部の状況を修正しました。(Tom)
- 終了位置が正確にセグメント境界であった場合、次のWALセグメント名を報告するようにバックアップ履歴ファイル内の`STOP WAL LOCATION`を修正しました。(Itagaki Takahiro)
- 一時ファイルが漏洩する一部の状況を修正しました。(Heikki)

これは前のマイナーリリースでもたらされた問題を修正します。 失敗する状況の1つは集合を返す`plpgsql`関数が他の関数の例外ハンドラ内で呼び出された場合です。

- `pg_hba.conf`とその関連ファイルを読み取る時、`@`が引用符記号の内部にある場合に`@something`をファイル包含要求と扱わないようにしました。 また、`@`自身をファイル包含要求として扱いません。(Tom)

ロールまたはデータベース名が`@`から始まる場合の一定しない動作を防止します。 空白文字を含むパス名のファイルを包含する必要がある場合、まだ実現させることはできますが、式全体を引用符でくくるのではなく`@"/path to/file"`と記述しなければなりません。

- ディレクトリが`pg_hba.conf`とその関連ファイル内で含有対象と指定されていた場合の一部のプラットフォームにおける無限ループを防止します。(Tom)
- `latex`および`troff`出力書式では行うべきではない文字列の書式付けを行わないように`psql`の`numericlocale`オプションを修正しました。(Heikki)
- 複合列がNULLに設定される場合に`plpgsql`が失敗することを修正しました。(Tom)
- コンパイラ固有の誤動作が起こる可能性を防ぐためにPL/Python内に`volatile`記号を追加しました。(Zdenek Kotala)
- PL/Tclは確実にTclインタプリタを完全に初期化します。(Tom)

このミスの既知の兆候は、Tcl 8.5以降を使用している場合のTclのclockコマンドの誤動作だけです。

- dblink_build_sql_*関数であまりに多くのキー列が指定された場合のcontrib/dblinkのクラッシュを防止します。(Rushabh Lathia, Joe Conway)
- ずさんなメモリ管理によって引き起こるcontrib/xml2の雑多なクラッシュを修正しました。(Tom)
- 時間帯データファイルをtzdata release 2010eに更新しました。 バングラデシュ、チリ、フィジー、メキシコ、パラグアイ、サモアにおける夏時間規則の変更が含まれます。

[前のページ](#)

リリース8.1.21

[ホーム](#)
[上に戻る](#)

[次のページ](#)

リリース8.1.19

E.100. リリース8.1.19

リリース日: 2009-12-14

このリリースは8.1.18に対し、各種の不具合を修正したものです。 8.1メジャーリリースにおける新機能については[項E.119](#)を参照してください。

E.100.1. バージョン8.1.19への移行

8.1.Xからの移行ではダンプ/リストアは不要です。 しかし8.1.18以前のバージョンからアップデートする場合は8.1.18のリリースノートを参照してください。

E.100.2. 変更点

- セッション固有の状態を変更するインデックス関数により引き起こされる間接的なセキュリティ脅威から保護します。(Gurjeet Singh, Tom)

この変更により不変インデックス関数がスーパーユーザのセッションを破壊できることを防ぎます。(CVE-2009-4136)

- ヌルバイトが埋め込まれたコモンネーム (CN) フィールドをもつSSL証明書を拒絶します。(Magnus)

これは、SSL検証においてサーバまたはクライアント名に対する証明書が不正に一致しないようにします。(CVE-2009-4034)

- バックエンド起動時のキャッシュ初期化の間でクラッシュする可能性を修正しました。(Tom)

- 安全ではない時点でのVACUUMを中断するシグナルを防止します。(Alvaro)

この修正により、VACUUM FULLがすでにタプルの移動をコミットした後にキャンセルされた場合のパニックや、通常のVACUUMがテーブルを切り詰めた後に割り込みを受けた場合の一時的なエラーを防止します。

- ハッシュテーブルサイズの計算における整数オーバーフローによるクラッシュの可能性を修正しました。(Tom)

これは、ハッシュ結合の結果のプランナの推定サイズが極端に大きい場合に発生することがありました。

- inet/cidrの比較における非常にまれなクラッシュを修正しました。(Chris Mikkelsen)

- 準備されたトランザクションで保持されるタプルレベルの共有ロックを確実に無視しないようにしました。(Heikki)

- 副トランザクション内でアクセスされるカーソルで使用する一時ファイルの早すぎる削除を修正しました。(Heikki)

- PAMパスワード処理をより堅牢に修正しました。(Tom)

以前のコードでは、Linuxのpam_krb5 PAMモジュールとドメインコントローラとしてのMicrosoft Active Directoryを組み合わせると失敗することが知られていました。 PAMスタックに渡される引数に関する正当ではない仮定を作成することであるため、どこかで問題になるかもしれません。

- CREATE OR REPLACE FUNCTIONにおける所有権限の依存関係の処理を修正しました。(Tom)

- 集合を返すPL/Perl関数により返される場合Perl配列が適切にPostgreSQL配列に返還されることを確実にしました。(Andrew Dunstan, Abhijit Menon-Sen)

集合以外を返す関数ではすでに正しく動作しました。

- PL/Pythonの例外処理におけるまれなクラッシュを修正しました。(Peter)

- psqlのflexモジュールが正しいシステムヘッダファイルを使用してコンパイルされることを確実にしました。(Tom)

これは、--enable-largefileが生成されたコード内に非互換の変更を引き起こすプラットフォームにおける構築失敗を修正します。

- 今後のバージョンのlibpqとの互換性を高めるために、postmasterが接続要求パケットにおけるapplication_nameパラメータをすべて無視するようにしました。(Tom)

- 時間帯データファイルをtzdata release 2009sまで更新しました。 南極、アルゼンチン、バングラデシュ、フィジー、ノヴォクズネツク、パキスタン、パレスチナ、サモア、シリアにおける夏時間規則の変更、および香港に関する歴史的な修正が含まれます。

[前のページ](#)

リリース8.1.20

[ホーム](#)
[上に戻る](#)

[次のページ](#)

リリース8.1.18

E.102. リリース8.1.17

リリース日: 2009-03-16

このリリースは8.1.16に対し、各種の不具合を修正したものです。 8.1メジャーリリースにおける新機能については[項E.119](#)を参照してください。

E.102.1. バージョン8.1.17への移行

8.1.Xからの移行ではダンプ/リストアは不要です。 しかし、8.1.15より前のバージョンからアップグレードする場合は、8.1.15のリリースノートを参照してください。

E.102.2. 変更点

- 符号化方式の変換に失敗した時、再帰的なエラーによるクラッシュを防止しました。(Tom)

この変更は、最近の2つのマイナーリリースにて関連する失敗の状況に対してなされた改修を拡張したものです。 前回の修正は元の問題報告に特化したものでしたが、符号化方式変換関数で発生するすべてのエラーがそのエラーを報告しようとして、無限に再帰される可能性があることが分かりました。 したがって、再帰的なエラー報告を行う状況になったことがわかった時の解決策は、変換と符号化方式を無効にし、通常のASCII形式のエラーメッセージで報告することです。(CVE-2009-0922)

- 特定の変換関数に対する、間違った符号化方式を用いた`CREATE CONVERSION`を許可しません。(Heikki)

これにより、符号化方式に関する失敗における、あり得る状況を防止します。 前回の変更は、同じ問題における別の種類の失敗に対する防止策でした。

- `to_char()` に引数のデータに対して不適切な整形用のコードが渡された時のコアダンプを修正しました。(Tom)
- 暗黙的な強制を使用した`CASE WHEN`の逆コンパイルを修正しました。(Tom)

この間違いにより、アサートを有効にして構築した場合にアサート失敗が発生する可能性があります。 また、他の構築状況でもビューの検証やダンプを行う際に"unexpected CASE WHEN clause"というエラーメッセージが発生する可能性があります。

- TOASTテーブルの行型に対する所有者を間違って割り当てる可能性を修正しました。(Tom)

`CLUSTER` または `ALTER TABLE` の書き換え構文がテーブル所有者以外のユーザにより実行された場合、テーブルのTOASTテーブル向けの`pg_type`項目が実行したユーザが所有するものとして記録されてしまいました。 TOASTの行型に対する権限は通常のデータベース操作ではまったく検証されませんので、これによりすぐに問題が発生することはありません。 しかし、後でコマンドを発行したロールを削除しようとした場合に想定外の失敗(8.1または8.2)や削除後に`pg_dump`が"owner of data type appears to be invalid"という警告が発生する(8.3)可能性があります。

- ブロックの終了時に、PL/pgSQLのエラー状態変数を完全に消去します。(Ashesh Vashi, Dave Page)

これはPL/pgSQL自体の問題ではありませんが、これを行わないと、PL/pgSQLデバグが関数の状態を検証する際にクラッシュする可能性があります。

- デフォルトの既知の時間帯省略形のリストに`MUST` (モーリシャス島夏時間)を追加しました。(Xavier Bugaud)

E.103. リリース8.1.16

リリース日: 2009-02-02

このリリースは8.1.15に対し、各種の不具合を修正したものです。 8.1メジャーリリースにおける新機能については[項E.119](#)を参照してください。

E.103.1. バージョン8.1.16への移行

8.1.Xからの移行ではダンプ/リストアは不要です。 しかし、8.1.15より前のバージョンからアップグレードする場合は、8.1.15のリリースノートを参照してください。

E.103.2. 変更点

- 自動バキュームのクラッシュを修正しました。(Alvaro)

データベース全体をトランザクション周回問題を回避させることを目的にバキュームした後にのみクラッシュが発生します。 つまり非周期的に発生するため、調査が困難でした。

- `headline()` 関数におけるURLの扱いを改良しました。(Teodor)
- `headline()` 関数における延長見出しの扱いを改良しました。(Teodor)
- 符号化方式の変換が符号化方式の組み合わせの指定を間違えた変換関数で作成された場合にあり得る、アサートの失敗や誤変換を防止します。(Tom, Heikki)
- `VACUUM`において小規模なテーブルに対する不要なロックを防止しました。(Heikki)
- 保持可能なカーソルの内容がTOASTテーブルの内容に依存しないことを確実にしました。(Tom)

これまでカーソル内の大規模なフィールドはTOASTポインタとして表現される可能性がありました。 これは、参照先のテーブルがカーソルを読み取る前に削除された場合や、大規模な値が削除、そしてバキュームされてしまった場合に失敗しました。 これは通常のカーソルでは発生することはあり得ませんが、トランザクションの生成以降保持されるカーソルではあり得ます。

- `contrib/tsearch2`の`get_covers()` 関数における未初期化変数を修正しました。(Teodor)
- `configure`スクリプトがPL/Perl用のリンク情報の入手ができなかった場合に失敗を適切に報告するように修正しました。(Andrew)
- すべての文書において、現在活動していない`pgsql-ports`と`pgsql-patches`メーリングリストの代わりに、`pgsql-bugs`または`pgsql-hackers`を参照するようにしました。
- 時間帯データファイルをtzdataリリース2009a（カトマンズ、およびスイス、キューバにおける歴史的なDSTの訂正）に更新しました。

E.104. リリース8.1.15

リリース日: 2008-11-03

このリリースは8.1.14に対し、各種の不具合を修正したものです。 8.1メジャーリリースにおける新機能については[項E.119](#)を参照してください。

E.104.1. バージョン8.1.15への移行

8.1.Xからの移行ではダンプ/リストアは不要です。しかし、8.1.2より前のバージョンからアップグレードする場合は、8.1.2のリリースノートを参照してください。また、これまでの8.1.Xリリースを使用している場合、アップグレード後にすべてのGiSTインデックスを `REINDEX` することを勧めます。

E.104.2. 変更点

- 削除後に間違ったインデックス項目を"dead"と印付けたことによるGiSTインデックスの破損を修正しました。(Teodor)
これによりインデックス検索は存在するはずの行の検索に失敗します。破損したインデックスは`REINDEX`を使用して修正することができます。

- クライアントの符号化方式が翻訳されたエラーメッセージを表すことができない時のバックエンドのクラッシュを修正しました。(Tom)

以前にも同様の問題に対応しましたが、"等価な文字を持たない"メッセージ自体が変換されない場合にまだ失敗しました。今回の修正では、こうした状況を検知した時、翻訳を無効にし、通常のASCIIエラーメッセージを送信します。

- トリガから深い入れ子状態の関数が呼び出された時にクラッシュする可能性を修正しました。(Tom)
- 副`SELECT`が`FROM`、複数行の`VALUES` リスト、`RETURNING` リスト内の関数呼び出し内に存在する場合の、間違ったルール問い合わせの展開を修正しました。(Tom)

この問題のよくある徴候は、"unrecognized node type"というエラーです。

- 新しく定義されたPL/pgSQLトリガ関数が通常の関数として呼び出された場合に報告されるエラーを確実なものにしました。(Tom)
- `ALTER SET TABLESPACE`を使ってテーブルを別のテーブル空間に移動する時に`relfilenode`番号が衝突する可能性を防止します。(Heikki)

このコマンドは、先のディレクトリで未使用であることが確認されたファイル名を選ばずに、既存のファイル名を再使用しようとしていました。

- 1つの問い合わせ項目がテキストの先頭の単語に一致した場合、テキスト検索の間違った見出し生成を修正しました。(Sushant Sinha)
- `--enable-integer-datetimes` で構築された場合、ISO以外の日付書式を使用した時の時間間隔値における不適切な秒端数表示を修正しました。(Ron Mayer)
- 渡されたタプルとタプル記述子が異なる列数を持っていた場合に`SPI_getvalue`および`SPI_getbinval`が適切に動作することを確実にしました。(Tom)

テーブルに列が追加または削除された場合、こうした状況は正常です。しかしこの2つの関数は適切に扱っていませんでした。起こり得る唯一の結果は間違ったエラーが示されることです。

- ecpgの`CREATE ROLE`解析を修正しました。(Michael)
- `pg_ctl restart`の最近の破損を修正しました。(Tom)
- 時間帯データファイルをtzdataリリース2008i（アルゼンチン、ブラジル、モーリシャス、シリアにおける夏時間規則の変更）まで更新しました。

E.105. リリース8.1.14

リリース日: 2008-09-22

このリリースは8.1.13に対し、各種の不具合を修正したものです。 8.1メジャーリリースにおける新機能については[項E.119](#)を参照してください。

E.105.1. バージョン8.1.14への移行

8.1.Xからの移行ではダンプ/リストアは不要です。 しかし、8.1.2より前のバージョンからアップグレードする場合は、8.1.2のリリースノートを参照してください。

E.105.2. 変更点

- 局所ロックのカウンタを32ビットから64ビットに広げました。(Tom)

これは、かなり長いトランザクションでカウンタがオーバーフローし、想定外の"すでに保持済のロックである"というエラーになるという報告に対応したものです。

- GiSTインデックススキャン中に出力タプルが重複する可能性を修正しました。(Teodor)
- `INSERT` または `UPDATE` で生成されるタプルが対象のテーブルの現在の行型に一致するか確認するために、エグゼキュータ起動の際の検査を追加しました。(Tom)

`ALTER COLUMN TYPE`の後にこれまでキャッシュしていた計画を再利用した場合、こうした状況が起こります。 データ破壊やその結果発生するクラッシュをこの検査により防ぎます。

- まず時間帯引数を時間帯省略形として解釈し、これまでのように他の方法を取らずに、それが失敗した場合にのみ完全な時間帯名称として解釈するように`AT TIME ZONE`を修正しました。(Tom)

タイムスタンプ入力関数は常にこの順序であいまいな時間帯名の解決を行っていました。 `AT TIME ZONE`も同様にすることで一貫性が向上し、そして、8.1で混入された互換性に関する不具合を修正します。 過去のバージョンの`AT TIME ZONE`では省略形のみを受け付けていましたので、あいまいな場合に8.0以前と同様な動作をするようになりました。

- 64ビットプラットフォームで稼働している場合、日付時刻入力関数が正しく整数オーバーフローを検知できるよう修正しました。(Tom)
- 非常に長いログメッセージをsyslogに書き出す際の性能を向上しました。(Tom)
- `SELECT DISTINCT ON`問い合わせに対するカーソルの後方スキャンにおける不具合を修正しました。(Tom)
- 入れ子状の副問い合わせ式がある場合のプランナの不具合を修正しました。(Tom)

外側の副問い合わせが親問い合わせに直接の依存性がなく、内側の副問い合わせが依存性がある場合、外側の値が新しい親問い合わせの行に対して再計算されなくなる可能性があります。

- プランナが論理型の結果を生成する`GROUP BY`式を、式の内容に関係なく、常に2つのグループになると推定することを修正しました。(Tom)

これにより、`col IS NULL`などの特定の論理試験に対して、通常の`GROUP BY`推定より非常により良く精度が上がります。

- `FOR`ループの対象値が複合型のフィールドを含むレコードである場合に失敗しないようにPL/pgSQLを修正しました。(Tom)
- Tcl 8.5で正しく動作するようPL/Tclを修正しました。 また、Tcl間でやりとりされるデータの符号化方式についてより注意を払うように修正しました。(Tom)
- Python 2.5で動作するようPL/Pythonを修正しました。

これは、8.2の開発期間で行われた修正を後方に反映したものです。

- `pg_dump`および`pg_restore`がSQLコマンドの送信に失敗した後に報告するエラーを改良しました。(Tom)

pg_ctlが`restart`の間正しくpostmasterのコマンドライン引数を保持するように修正しました。(Bruce)

- 時間帯データファイルをtzdataリリース2008f（アルゼンチン、バハマ、ブラジル、モーリシャス、モロッコ、パキスタン、パレスチナ、パラグアイにおける夏時間規則の変更）まで更新しました。

[前のページ](#)

リリース8.1.15

[ホーム](#)
[上に戻る](#)

[次のページ](#)

リリース8.1.13

E.106. リリース8.1.13

リリース日: 2008-06-12

このリリースは8.1.12に対し、重大な1つの不具合と軽微な1つの不具合を修正したものです。 8.1メジャーリリースにおける新機能については[項E.119](#)を参照してください。

E.106.1. バージョン8.1.13への移行

8.1.Xからの移行ではダンプ/リストアは不要です。 しかし、8.1.2より前のバージョンからアップグレードする場合は、8.1.2のリリースノートを参照してください。

E.106.2. 変更点

- `pg_get_ruledef()` で負の定数を括弧でくるようにしました。(Tom)

この修正前では、ビューまたはルールにおける負の定数がそのまま、例えば`-42::integer`のようにダンプされる可能性があります。 これは若干正しくありません。 演算子の前置規則に従うと、`(-42)::integer`となるべきです。 通常は大した違いが起こることはありませんが、最近の別のパッチと相互作用して、PostgreSQLが有効な`SELECT DISTINCT`ビューの問い合わせを拒絶することになりました。 `pg_dump`の出力のリロード失敗という結果になることもあり得たため、これは高優先度の修正として扱われました。 実際にダンプ出力が正しくなかったバージョンは8.3.1と8.2.7だけです。

- `ALTER AGGREGATE ... OWNER TO`が`pg_shdepend`を更新するようにしました。(Tom)

この見落としにより、集約が後で`DROP OWNED`または`REASSIGN OWNED`操作で使用された場合に問題が発生するおそれがありました。

E.107. リリース8.1.12

リリース日: リリースされませんでした

このリリースには8.1.11からの各種修正が含まれています。 8.1メジャーリリースにおける新機能については[項E.119](#)を参照してください。

E.107.1. バージョン8.1.12への移行

8.1.Xからの移行ではダンプ/リストアは不要です。 しかし、8.1.2より前のバージョンからアップグレードする場合は、8.1.2のリリースノートを参照してください。

E.107.2. 変更点

- 新しい列がすべて非NULLに初期化されていることが正しく検査されるようにALTER TABLE ADD COLUMN ... PRIMARY KEYを修正しました。(Brendan Jurd)

これまでのバージョンでは、この必要条件の検査をまったく行っていませんでした。

- "同じ"制約を、共通の祖先からその制約を継承する複数の親リレーションから継承する場合に、CREATE TABLEが失敗する可能性を修正しました。(Tom)
- キリル語の"Yo"文字 (2つのドットを持つ_еと_Е) を扱うISO-8859-5と他の符号化方式との間の変換を修正しました。(Sergey Burladyan)
- 複数のデータ型入力関数、特に、結果に初期化されず予測できない値を含む未使用のバイトを許すarray_in()を修正しました。(Tom)

これは、2つの明らかに同じリテラル値が等しいとみなされず、結果パーサは一致しないORDER BYおよびDISTINCT式に関して不平をいうといったエラーをもたらします。

- 正規表現副文字列一致 (substring(string from pattern)) における境界状態を修正しました。(Tom)

この問題は、パターン全体は一致するが、ユーザが括弧付きの副式を指定し、その副式が一致しなかった時に発生します。例えば、substring('foo' from 'foo(bar)?') です。(bar) は一致しませんので、これはNULLを返すべきです。しかし、間違ってパターン全体が一致 (つまりfoo) という結果を返していました。

- 時間帯データファイルをtzdataリリース2008c (モナコ、イラク、チョイバルサン、パキスタン、シリア、キューバ、アルゼンチン/サンルイスにおける昼時間規則の変更) まで更新しました。
- ecpgのPGTYPEtimestamp_sub() 関数からの不正確な結果を修正しました。(Michael)
- 入力問い合わせがNULL値を返す場合のcontrib/xml2のxpath_table() 関数のコアダンプを修正しました。(Tom)
- contrib/xml2のmakefileがCFLAGSを上書きしないように修正しました。(Tom)
- DatumGetBool マクロがgcc 4.3でエラーにならないよう修正しました。(Tom)

この問題は、論理値を返す"古い形式" (V0)のC関数に影響します。これは8.3ですでに修正されていましたが、当時なかったバックパッチが必要になりました。

- 長い間存在したLISTEN/NOTIFYの競合条件を修正しました。(Tom)

まれな状況において、NOTIFYを実行する同時実行のトランザクションが後にコミットされたことが観測されたために想定されたとしても、LISTENだけを実行するセッションが注意を受け取らない可能性があります。

この修正の副作用として、LISTENコマンドをまだコミットしていないトランザクションではLISTEN用のpg_listener内の行を参照しなくなります。これは検索するために選択すべきもので、これまではできていました。この動作はできるともできないとも文書化されていませんが、一部のアプリケーションで古い動作に依存している可能性があります。

- 準備済みトランザクション内でのLISTENとUNLISTENを許しません。(Tom)

これは以前までは許されていましたが、実行しようとしても様々な望まない結果、特に発生元のバックエンド

がUNLISTENが未コミットである限り終了しないという結果が発生しました。

- ハッシュインデックスを使用する問い合わせ中でエラーが発生した場合の、稀なクラッシュを修正しました。(Heikki)
- 紀元前の2月29日に関する日付時刻値の入力を修正しました。(Tom)

これまでのコードにはうるう年の決定に間違いがありました。

- ALTER OWNERの一部の構文における"unrecognized node type"エラーを修正しました。(Tom)
- コマンドラインオプションからpostmasterのポート番号を正しく取り出せるようpg_ctlを修正しました。(Itagaki Takahiro, Tom)

これまでのpg_ctl start -wでは間違ったポート上のpostmasterに接続しようとし、起動失敗に関するおかしい報告を引き起こしました。

- 最近のバージョンのgccにおける間違った最適化の可能性を抑えるために-fwrapvを使用します。(Tom)

これはPostgreSQLをgcc 4.3以降で構築する時に必要であることは分かっていました。

- ORDER BYおよびGROUP BY内の定数式の表示を修正しました。(Tom)

明示的にキャストされた定数が不正確に表示されました。これは例えば、ダンプ/リストア時にビュー定義を壊す可能性があります。

- COPY OUT時にNOTICEメッセージを正しく扱うようlibpqを修正しました。(Tom)

このエラーは、ユーザ定義のデータ型出力ルーチンがNOTICEを発行する時に起こることのみが観測されていました。しかし、他が原因で発生していないという保証はありません。

E.108. リリース8.1.11

リリース日: 2008-01-07

このリリースは、重大なセキュリティ問題の修正を含む、8.1.10の各種不具合を修正したものです。8.1メジャーリリースにおける新機能については[項E.119](#)を参照してください。

これが、PostgreSQLコミュニティがWindows用のバイナリパッケージを作成する、最後の8.1.xリリースです。8.2.Xでは、過去のバージョンに移植することが現実的ではない、Windows固有の修正が行われていますので、Windowsユーザは8.2.X以降への移行が推奨されています。他のプラットフォームでは8.1.Xはまだサポートされます。

E.108.1. バージョン8.1.11への移行

8.1.Xからの移行ではダンプ/リストアは不要です。しかし、8.1.2より前のバージョンからアップグレードする場合は、8.1.2のリリースノートを参照してください。

E.108.2. 変更点

- インデックス内の関数がVACUUM、ANALYZEなどを実行するユーザの権限で実行されることを防止します。(Tom)

インデックス式内で使用される関数および部分インデックスの述語は、新しいテーブル項目が作成されると評価されます。これが、もし信頼できないユーザが所有するテーブルを変更したとすると、トロイの木馬となるコードの実行という危険性を引き起こすことは以前から解っていました。(トリガ、デフォルト、検査制約なども同種の危険性を引き起こします。)しかしインデックス内の関数は、VACUUM FULLなどの日常の保守作業で実行される可能性があるため、さらに危険です。こうした保守作業は通常自動的にスーパーユーザ権限で行われます。例えば、極悪なユーザがトロイの木馬となるインデックス定義を設定し、次の日常のバキューム処理を待つことで、スーパーユーザ権限でコードを実行することが可能です。この修正は、SECURITY DEFINER関数で使用されていた権限変更機構を使用して、標準的な保守作業(VACUUM、ANALYZE、REINDEX、CLUSTERを含む)が呼び出したユーザではなくテーブル所有者として実行するように調整しました。こうしたセキュリティ制限の迂回を防ぐために、SET SESSION AUTHORIZATIONおよびSET ROLEがSECURITY DEFINER コンテキストで禁止されるようになりました。(CVE-2007-6600)

- 正規表現パッケージ内のさまざまな不具合を修正しました。(Tom, Will Drewry)

より巧妙な正規表現パターンにより、信頼できないソースから正規表現検索パターンを受け付けるアプリケーションでは、クラッシュ、無限または無限に近いループ、大量のメモリの浪費、DoS攻撃を引き起こすすべてが発生する可能性があります。(CVE-2007-4769、CVE-2007-4772、CVE-2007-6067)

- /contrib/dblinkを使用して、パスワード認証のみをセキュリティ制限として使用するには非スーパーユーザでなければなりません。

8.1.10で行った修正は不十分で、一部のdblink関数に対してのみ問題を解消していました。(CVE-2007-6601、CVE-2007-3278)

- 時間帯ファイルをtzdataリリース2007kに更新しました(具体的には最近のアルゼンチンの変更)。(Tom)

- 非CロケールにおけるプランナのLIKE/正規表現の推定を改良しました。(Tom)

- 一部のWHERE false AND var IN (SELECT ...)におけるプランナの失敗を修正しました。(Tom)

- ALTER TABLE ... ALTER COLUMN TYPEで再構築されるインデックスのテーブル空間と格納パラメータを保持します。(Tom)

- アーカイブリカバリが、リカバリの停止時間が使用された時だけではなく、常に新しいWAL時系列から始まるようにしました。

これは、まれに起こる可能性のある既存の最後のWALセグメントのアーカイブされたコピーを上書きしようとする危険を防止します。また元の定義より単純、明確になったと考えられます。

- テーブルが小さすぎて有用でない場合、VACUUMがすべてのmaintenance_work_memを使用しないようにしました。
- マルチバイトデータベース符号化方式を使用している時に潜在する、translate()のクラッシュを修正しました。
- 68年を超える間隔でextract(epoch from interval)がオーバーフローすることを修正しました。(Tom)

- 信頼された関数内でUTF-8正規表現が使用された場合にPL/Perlが失敗しないように修正しました。(Andrew)
- プラットフォームのPerlが`bool`型を`char`ではなく`int`として定義した場合に対応するようにPL/Perlを修正しました。(Tom)

これは理論上どこでも発生する可能性がありましたが、標準的に構築されたPerlではこうなりませんでした。Mac OS X 10.5までは...

- 長い例外メッセージでクラッシュしないようPL/Pythonを修正しました。(Alvaro)
- 親テーブルと異なるデフォルト式を持つ、継承子テーブルを正しく扱うよう`pg_dump`を修正しました。(Tom)
- `PGPASSFILE`が普通のファイル以外のファイルを参照する場合に`libpq`がクラッシュすることを修正しました。
- `ecpg`パーサを修正しました。(Michael)
- 128ビットより長いキーで失敗するOpenSSLライブラリに対して`contrib/pgcrypto`が防衛するようにしました。これは少なくとも一部のバージョンのSolarisで発生します。(Marko Kreen)
- `contrib/tablefunc`の`crosstab()`が、クラッシュするのではなく、独自の意味でカテゴリとしてNULL行識別子を扱うようにしました。(Joe)
- `tsvector`および`tsquery`出力関数が正しくバックスラッシュをエスケープするように修正しました。(Teodor, Bruce)
- 入力文字列が大きな場合に`to_tsvector()`がクラッシュすることを修正しました。(Teodor)
- `configure`スクリプトを再生成する場合、特定のバージョンのAutoconfを使用することが必要になりました。(Peter)

これは開発者とパッケージ作成者のみに影響します。この変更は、AutoconfとPostgreSQLの未確認のバージョンの組み合わせを使用する事故を防ぐためになされました。その結果がうまくいくかどうかについて責任を負う限り、異なるバージョンのAutoconfを本当に使用したい場合は、このバージョン検査を取り除くことができます。

[前のページ](#)

リリース8.1.12

[ホーム](#)
[上に戻る](#)

[次のページ](#)

リリース8.1.10

E.109. リリース8.1.10

リリース日: 2007-09-17

このリリースは8.1.9の各種不具合を修正したものです。 8.1メジャーリリースにおける新機能については[項E.119](#)を参照してください。

E.109.1. バージョン8.1.10への移行

8.1.Xからの移行ではダンプ/リストアは不要です。 しかし、8.1.2より前のバージョンからアップグレードする場合は、8.1.2のリリースノートを参照してください。

E.109.2. 変更点

- トランザクションが、行を挿入した後に、同一テーブルに対する同時実行中のVACUUMの終了時刻とほぼ同じ時点でアポートした場合のインデックス破損を防止します。(Tom)
- CREATE DOMAIN ... DEFAULT NULLが適切に動作するようにしました。(Tom)
- intervalデータ型が、ミリ秒のみ、または、マイクロ秒のみから構成される入力を受け付けられるようになりました。(Neil)
- r-treeインデックス挿入を高速化しました。(Teodor)
- 過度なSSLエラーメッセージのログ処理を修正しました。(Tom)
- sysloggerプロセスを使用する時にログメッセージが差し込まれないようにログ処理を修正しました。(Andrew)
- log_min_error_statementログ処理がメモリ不足になった時のクラッシュを修正しました。(Tom)
- 一部の外部キーの境界の不正な扱いを修正しました。(Tom)
- 他のセッションによる一時テーブル処理を防ぐために起こるREINDEXとCLUSTERの失敗を防ぎます。(Alvaro)
- 時間帯データベース規則を更新しました。具体的にはニュージーランドで予定している変更です。(Tom)
- Windowsソケットを改良しました。(Magnus)
- 符号化方式が不整合となる可能性があるため、Windowsにてログのタイムスタンプに時間帯名(%Z)を出力しないようにしました。(Tom)
- セキュリティ強化のため、パスワード認証のみを使用する/contrib/dblinkを使用するユーザが非特権ユーザであることを要求します。(Joe)

E.110. リリース8.1.9

リリース日: 2007-04-23

このリリースは8.1.8の各種不具合を修正したもので、セキュリティ問題の修正を1つ含みます。8.1メジャーリリースにおける新機能については[項E.119](#)を参照してください。

E.110.1. バージョン8.1.9への移行

8.1.Xからの移行ではダンプ/リストアは不要です。しかし、8.1.2より前のバージョンからアップグレードする場合は、8.1.2のリリースノートを参照してください。

E.110.2. 変更点

- `search_path`における一時テーブルスキーマの明示的な指定をサポートしました。また、関数や演算子ではその検索を無効にしました(Tom)。

これは、`SECURITY DEFINER`関数が`search_path`の値を本当に安全に設定するために必要です。こうしないと、一時オブジェクトを使用して権限のないSQLユーザが`SECURITY DEFINER`関数の権限でコードを実行することができてしまうためです (CVE-2007-2138)。詳細は`CREATE FUNCTION`を参照してください。

- `/contrib/tsearch2`のクラッシュを修正しました(Teodor)。
- `COMMIT PREPARED`は、そのトランザクションを準備したデータベースと同じデータベースで実行する必要があります。(Heikki)
- `VACUUM FULL`の`UPDATE`連鎖方法において潜在したデータ破損不具合を修正しました(Tom, Pavan Deolasee)。
- 外部結合の向上やビットマップスキャンの選択ロジックなど、プランナを修正しました(Tom)。
- ハッシュインデックスの拡張時のパニックを修正しました(8.1.6で入ってしまった不具合です)(Tom)。
- POSIX書式の時間帯指定が新しいUSA DST規則に従うよう修正しました(Tom)。

E.111. リリース8.1.8

リリース日: 2007-02-07

このリリースには、8.1.7に対する1つの修正が含まれています。 8.1メジャーリリースにおける新機能については[項E.119](#)を参照してください。

E.111.1. バージョン8.1.8への移行

8.1.Xからの移行ではダンプ/リストアは不要です。 しかし、8.1.2より前のバージョンからアップグレードする場合は、8.1.2のリリースノートを参照してください。

E.111.2. 変更点

- 制約および関数インデックスにおける型の長さに対する重複/限定検査を削除しました。(Tom)

E.112. リリース8.1.7

リリース日: 2007-02-05

このリリースは8.1.6の各種不具合を修正したもので、セキュリティ問題の修正も含みます。 8.1メジャーリリースにおける新機能については[項E.119](#)を参照してください。

E.112.1. バージョン8.1.7への移行

8.1.Xからの移行ではダンプ/リストアは不要です。しかし、8.1.2より前のバージョンからアップグレードする場合は、8.1.2のリリースノートを参照してください。

E.112.2. 変更点

- 接続ユーザがバックエンドのメモリを読み取ることができるセキュリティ脆弱性を取り除きました。(Tom)
この脆弱性には、SQL関数が宣言されたデータ型を返すかどうか、そのテーブル列のデータ型が変更されたかどうかに関する通常の検査の抑制も含まれています (CVE-2007-0555、CVE-2007-0556)。これらのエラーを悪用して簡単にバックエンドをクラッシュさせることができます。また原理的には、アクセスを許していないはずのユーザがデータベースの内容を読み取ることができてしまいます。
- 実行不可能な分割点の選択によりB-Treeインデックスページの分割が失敗する可能性がある、稀に起こる不具合を修正しました。(Heikki Linnakangas)
- 多くのテーブルを持つデータベースに対するVACUUMの性能を向上しました。(Tom)
- 接続できないデータベースに永続的ではないトランザクションIDを残さないように自動バキュームを修正しました。(Alvaro)

この不具合は8.1系列にのみ影響します。

- UNIONをきっかけとした、稀に発生するAssert()クラッシュを修正しました。(Tom)
- 3バイト長を越えるUTF8シーケンスに関する、複数バイト文字処理のセキュリティを強化しました。(Tom)
- 削除済みのファイルに対するfsync試行による、Windowsにおいて発生するにせの"権限がない"というエラーを修正しました。(Magnus, Tom)
- 使用中のPL/pgSQL関数を更新する時にクラッシュする可能性を修正しました。(Tom)

E.113. リリース 8.1.6

リリース日: 2007-01-08

このリリースは8.1.5の各種不具合を修正したものです。 8.1メジャーリリースにおける新機能については[項E.119](#)を参照してください。

E.113.1. バージョン8.1.6への移行

8.1.Xからの移行ではダンプ/リストアは不要です。 しかし、8.1.2より前のバージョンからアップグレードする場合は、8.1.2のリリースノートを参照してください。

E.113.2. 変更点

- AIXにおける`getaddrinfo()`の扱いを改良しました。(Tom)
これは、特に統計情報コレクタの起動に関する問題を修正するものです。
- コメント付きのラージオブジェクト (blobs) を含むtar書式のバックアップを扱うよう`pg_restore`を修正しました。(Tom)
- `VACUUM`における"親キーの再検索に失敗する"エラーを修正しました。(Tom)
- サーバ再起動時に`pg_internal.init`キャッシュファイルを初期化するようにしました。(Simon)
PITR復旧後にキャッシュファイル内におかしなデータが存在する危険を防ぎます。
- `VACUUM`によるギガバイト境界を跨る大規模なリレーションの切り詰めにおける競合状態を修正しました。(Tom)
- 行レベルロックにおける不要なデッドロックエラーが発生する不具合を修正しました。(Tom)
- 数ギガバイトのハッシュインデックスに影響する不具合を修正しました。(Tom)
- Windowsのシグナル処理におけるデッドロックの可能性を修正しました。(Teodor)
- 複数の空要素からなる`ARRAY[]`生成時の不具合を修正しました。(Tom)
- ECPGにおける接続時のメモリリークを修正しました。(Michael)
- Darwin (OS X)のコンパイルを修正しました。(Tom)
- 新しく`initdb`を行ったインストレーションでは、`to_number()`と`to_char(numeric)`が`IMMUTABLE`ではなく`STABLE`になりました。(Tom)
これは、`lc_numeric`がこれらの関数の出力を変更する可能性があるためです。
- 括弧を使用した正規表現を使用するインデックスを改良しました。(Tom)
この改良により、`psql %d`の性能も向上しました。
- 時間帯データベースを更新しました。
具体的にはこれは、オーストラリアおよびカナダの夏時間規則に影響します。

E.114. リリース8.1.5

リリース日: 2006-10-16

このリリースは8.1.4からの各種の修正を含んでいます。 8.1メジャーリリースにおける新機能については[項E.119](#)を参照してください。

E.114.1. 8.1.5バージョンへの移行

8.1.Xを稼働している場合dump/restoreは必要ありません。とは言っても、8.1.2以前のバージョンからの更新をするのであれば、8.1.2のリリースノートを参照してください。

E.114.2. Changes

- 副問い合わせ内を除いてUPDATEコマンドでの集約関数を認めなくしました。
このような集約の振舞は事前予測できないことと、8.1.Xにてクラッシュが起きたため、機能しないようにしました。SQL標準も同じように許可していません。
- 型指定無しのリテラルをANYARRAYとされた場合のコアダンプを修正しました。
- COMMIT、もしくはROLLBACKが実行された時、拡張問い合わせプロトコルに対するログ取得期間のコアダンプを修正しました。
- 複数行を返すSQL関数を含む問い合わせにおいて、AFTERトリガーの間違った取り扱いを修正しました。
- USING句に対するNOT NULLを再検査するALTER TABLE ... TYPEを修正しました。
- 分離文字列に対するオーバラップマッチを処理するstring_to_array() を修正しました。
例えば: string_to_array('123xx456xxx789', 'xx')
- AM/PM書式のto_timestamp() を修正しました。
- ANALYZEが必要か否かを判定する自動VACUUM計算を修正しました。
- psql's \d コマンドに対するパターンマッチにおけるちょっとした部分を修正しました。
- /contrib/ltreeの索引破壊バグを修正しました。
- ecpgをより強固にするためいくつかの修正をしました。
- /contrib/dbmirrorで逆スラッシュエスケープを修正しました。
- contrib/dblink および /contrib/tsearch2 に多少の修正を加えました。
- ハッシュテーブルとビットマップインデックススキャンに効率的な改良が加えられました。
- Win32上での統計情報収集の不安定性を修正しました。
- Win32上で適切なユニットを使用するstatement_timeout を修正しました。
以前のWin32 8.1.X バージョンでは、要素100で遅延が効きませんでした。
- MSVC および Borland C++ コンパイラを修正しました。
- AIX および Intelコンパイラを修正しました。
- 連続したアーカイブ処理における稀な不具合を修正しました。(Tom)

E.115. リリース8.1.4

リリース日: 2006-05-23

このリリースは8.1.3の各種不具合を修正したもので、非常に深刻なセキュリティ問題を解消するパッチを含みます。 8.1メジャーリリースにおける新機能については[項E.119](#)を参照してください。

E.115.1. バージョン8.1.4への移行

8.1.Xからの移行ではダンプ/リストアは不要です。しかし、8.1.2より前のバージョンからアップグレードする場合は、8.1.2のリリースノートを参照してください。

CVE-2006-2313およびCVE-2006-2314に示されたSQLインジェクション攻撃を完全に防ぐためには、アプリケーション側のコードの変更が必要となる場合があります。SQLコマンド内に信頼できない文字列を埋め込むアプリケーションでは、できる限り早く、その文字列が推奨するエスケープ技法を使用していることを確実に検証しなければなりません。ほとんどの場合、アプリケーションは文字列のエスケープ処理に、その場しのぎのコードではなく、ライブラリやドライバが提供する (libpqのPQescapeStringConn()) のような関数を使用しなければなりません。

E.115.2. 変更点

- すべての場合において、符号化が無効なマルチバイト文字を拒絶するようにサーバを変更しました。(Tatsuo, Tom)

PostgreSQLは少し前からこのような方向に移行していましたが、この検査がすべての符号化方式とすべてのテキスト入力に対して統一的に適用されるようになりました。さらに、単なる警告ではなく常にエラーとなるようになりました。この変更はCVE-2006-2313で示されるような種類のSQLインジェクション攻撃から保護します。

- 文字列リテラル内の`'\'`の安全ではない使用を拒絶します。

CVE-2006-2314で示されるような種類のSQLインジェクション攻撃からのサーバ側の保護として、サーバは、SQL文字列リテラル内のASCII単一引用符の表現として、`'`のみを受け付け、`'\'`を受け付けなくなりました。デフォルトでは、SQLインジェクションが可能となる状況である、`client_encoding`がクライアント側のみの符号化方式 (SJIS、BIG5、GBK、GB18030、UHC) に設定された場合にのみ`'\'`は拒絶されます。新しい設定パラメータ`backslash_quote`により、必要な場合にこの動作を調整できます。CVE-2006-2314に対して完全に保護するには、クライアント側を変更する必要があるかもしれないことに注意してください。安全ではないクライアントを安全ではないものとして明らかにすることが、`backslash_quote`の目的の一つです。

- libpqの文字列エスケープルーチンを、符号化方式と`standard_conforming_strings`を考慮するように変更しました。

これは、CVE-2006-2313およびCVE-2006-2314で示されるセキュリティ問題に対し、libpqを使用したアプリケーションを修正します。また、将来予定されるSQL標準文字列リテラル構文への移行に対しても保護しています。同時に複数のPostgreSQL接続を使用するアプリケーションは、各データベース接続において使用される設定に合わせてエスケープ処理が正しく行われるように、`PQescapeStringConn()`と`PQescapeByteaConn()`に移行しなければなりません。"独自に"文字列エスケープ処理を行うアプリケーションはライブラリルーチンを使用するように変更しなければなりません。

- pgcryptoのキー選択の弱さを修正しました。(Marko Kreen)

偶発的なPRNGの種再作成ロジック内のエラーにより、`pgp_sym_encrypt()`で選択されるセッションキーが予測可能になりました。これはOpenSSLを使用せずに構築した場合にのみ影響します。

- 一部の不正な符号化方式変換関数を修正しました。

`win1251_to_iso`、`win866_to_iso`、`euc_tw_to_big5`、`euc_tw_to_mic`、`mic_to_euc_tw`はすべて可変拡張に関して正しくありませんでした。

- 文字列における偶然残る`'\'`の使用を整理しました。(Bruce, Jan)
- autovacuumを`pg_stat_activity`で確認できるようにしました。(Alvaro)
- `full_page_writes`を無効にしました。(Tom)

`full_page_writes`を無効にするとクラッシュリカバリが失敗する特定の状況があります。8.2で適切に修正する予定ですが、当面は単に無効にします。

- プランナ、特にビットマップインデックススキャンとMIN/MAX最適化に関してさまざまな修正を行いました。(Tom)
- マージ結合における不正な最適化を修正しました。(Tom)
外部結合で一致しなかった行の複数のコピーを省くことができました。
- 同一トランザクション内でplpgsql関数を使用し、変更する場合のクラッシュを修正しました。
- B-Treeインデックスが消去された場合のWAL再生を修正しました。
- |を含むパターンに対するSIMILAR TOを修正しました。(Tom)
- SELECT INTOおよびCREATE TABLE ASが基本ディレクトリではなくデフォルトのテーブル空間にテーブルを作成するように変更しました。(Kris Jurka)
- 独自のDH SSLパラメータを正しく使用するようにサーバを修正しました。(Michael Fuhr)
- qsortの性能を向上しました。(Dann Corbit)
現時点ではSolarisでのみこのコードが使用されます。
- Intel MacでのBonjourを修正しました。(Ashley Clark)
- さまざまな小規模のメモリリークを修正しました。
- 一部のWin32システムにおけるパスワードプロンプトに関する問題を修正しました。(Robert Kinberg)
- pg_dumpのドメインに関するデフォルト値の取り扱いを改良しました。
- 同一名称のユーザとグループを適切に扱うようpg_dumpallを修正しました。(8.1より前のサーバからのダンプを行う場合のみです。)(Tom)
ユーザとグループはLOGIN権限を持つ一つのロールにまとめられます。これまではまとめられたロールはLOGIN権限を持たなかったため、ユーザとして使用することができなくなっていました。
- pg_restore -nが文書通りに動くよう修正しました。(Tom)

E.116. リリース8.1.3

リリース日: 2006-02-14

このリリースは8.1.2の各種不具合を修正したもので、非常に深刻なセキュリティ問題を1つ含みます。 8.1メジャーリリースにおける新機能については[項E.119](#)を参照してください。

E.116.1. バージョン8.1.3への移行

8.1.Xからの移行ではダンプ/リストアは不要です。しかし、8.1.2より前のバージョンからアップグレードする場合は、8.1.2のリリースノートを参照してください。

E.116.2. 変更点

- 任意のログインしたユーザが他の任意のデータベースユーザIDにSET ROLEすることができる不具合を修正しました。(CVE-2006-0553)

有効検査が不適切であったため、ユーザは通常SET ROLEがエラーの後に以前のロール設定に戻すために使用する特殊な状況を不正利用することができました。これにより例えば、一般ユーザはスーパーユーザ状態を持つことができます。権限の拡大という危険は8.1.0から8.1.2までのみに存在します。しかし、7.3までさかのぼるすべてのリリースにおいて、アサートを有効にしてコンパイルした場合（これはデフォルトではありません）、非特権ユーザがサーバをクラッシュできるという、関連した不具合がSET SESSION AUTHORIZATIONにあります。この問題を報告してくれたAkio Ishidaに感謝します。

- 自身が挿入した行に関する行の可視性ロジックについての不具合を修正しました。(Tom)

非常に稀な状況で、現在のコマンドが挿入した行が、本来無効であるべき時に既に有効になっているものとして扱われることがありました。リリース8.0.4、7.4.9、7.3.11で組み込まれた不具合を修正しました。

- pg_clogおよびpg_subtransファイル生成時に"ファイルが既に存在する"というエラーを引き起こす競合条件を修正しました。(Tom)
- 準備済み文内のUNKNOWNパラメータ用のDOMAIN制約を適切に検査します。(Neil)
- ALTER COLUMN TYPEがFOREIGN KEY、UNIQUE、PRIMARY KEY制約を確実に適切な順番で処理するようにしました。(Nakano Yoshihisa)
- 独自の演算子や演算子クラスを参照するクロススキーマを持つダンプをリストアできるように修正しました。(Tom)
- COPYに失敗した後、pg_restoreが適切に処理を継続できるようにしました。これまではCOPYデータの残りをSQLコマンドとして扱おうとしました。(Stephen Frost)
- データディレクトリを指定しない場合にpg_ctl unregisterがクラッシュすることを修正しました。(Magnus)
- libpq PQprint HTMLタグを修正しました。(Christoph Zwerschke)
- AMD64およびPPC上のecpgのクラッシュを修正しました。(Neil)
- 関数の戻り値型宣言において、SETOFと*TYPEの両方を使用できるようにしました。
- PL/pythonにおいて、引数を渡す段階でエラーが発生した場合に適切に復旧するようにしました。(Neil)
- plperl_return_next内のメモリリークを修正しました。(Neil)
- Win32において、PL/perlのロケールの扱いをバックエンドに合わせるように修正しました。(Andrew)
- オプティマイザのさまざまな修正。(Tom)
- Win32においてpostgresql.confでlog_min_messagesがDEBUG3以上に設定された時のクラッシュを修正しました。(Bruce)
- Win32、Cygwin、OS X、AIX用のpgxs -Lライブラリ検索パス指定を修正しました。(Bruce)

Win32の管理者権限を検査する時にSIDが有効か検査します。(Magnus)

- 範囲外の日付入力を適切に拒否します。(Kris Jurka)
- configure時の`finite`および`isinf`の存在検査に関する移植性を修正しました。(Tom)
- データ行単位のカーネルコールを止め、libpq経由の`COPY IN`の速度を向上しました。(Alon Goldshuv)
- `/contrib/tsearch2`のインデックス作成速度を向上しました。(Tom)

[前のページ](#)

リリース8.1.4

[ホーム](#)
[上に戻る](#)

[次のページ](#)

リリース8.1.2

E.117. リリース8.1.2

リリース日: 2006-01-09

このリリースは8.1.1の各種不具合を修正したものです。 8.1メジャーリリースにおける新機能については[項E.119](#)を参照してください。

E.117.1. バージョン8.1.2への移行

8.1.Xからの移行ではダンプ/リストアは不要です。 しかし、後述のロケールやplperlの問題の影響がある場合、移行後にテキストの列に対するインデックスをREINDEXする必要があるかもしれません。

E.117.2. 変更点

- ShmemBackendArray内に空きがない場合にpostmasterが終了することなく続行できるように、Windows用のコードを修正しました。(Magnus)

これまでの動作では、過大な要求が連続して届いた場合にサービス不能状態になる可能性がありました。 これはWindowsへの移植でのみで起こり得ました。

- 使用済みのReadBufferが新しいものとして返されることが可能となるという、8.0で混入された不具合を修正しました。これにより、最近コミットされたデータが失われる可能性がありました。(Tom)
- トランザクション外または失敗したトランザクション内から発行される、プロトコルレベルのDescribeメッセージを修正しました。(Tom)
- 韓国語など、異なる文字の組み合わせを同一のものとみなすロケールにおける文字列比較を修正しました。(Tom)

既存のテキスト列に対するインデックスを修正するにはREINDEXが必要になります。

- plperlが後でロケールを変更できないように、postmasterの起動時にロケール用の環境変数を設定します。

これは、initdb実行時に通知したロケールと異なるロケールを表す環境変数の下でpostmasterが起動された場合に発生する問題を修正します。 このような条件では、plperlの使用によりインデックスが破損される可能性がありました。 こうした問題が発生していた場合、既存のテキスト列に対するインデックスのREINDEXが必要になります。

- インストールディレクトリに関する、より柔軟な位置の変更が可能になりました。(Tom)

これまでのリリースでは、すべてのインストールディレクトリパスが最後の要素を除いて同一である場合にのみ、位置の変更をサポートしていました。

- ISO-8859-5符号化方式およびISO-8859-9符号化方式の使用によるクラッシュを防止しました。(Tatsuo)
- 長期間存在した、ごく一部の特定のアジア向けマルチバイト文字セットを取り扱う場合のstrpos()と正規表現の不具合を修正しました。(Tatsuo)
- COPY CSVモードがすべての¥. をコピーデータの終端とみなしてしまう不具合を修正しました。

新しいコードでは、ドキュメントの記載通り、1行に¥. のみがあることを必要とします。

- データ終端記号と解釈されないように、COPY CSVモードでリテラルデータ値内の¥. を引用符で括弧するようにしました。(Bruce)
- RECORDを返す関数に関するさまざまな修正を行いました。(Tom)
- postgresql.confの最終行に改行コードがない場合でも適切に処理できるように修正しました。(Tom)
- MD5およびXDESアルゴリズム用のソルト空間をすべて利用できないという、/contrib/pgcryptoのgen_saltの不具合を修正しました。(Marko Kreen, Solar Designer)

Blowfishおよび標準DES用のソルトには影響ありません。

- 式インデックスの処理におけるautovacuumのクラッシュを修正しました。

- 指定した列数が実際に問い合わせで返される列数と異なる場合にクラッシュせずにエラーを返すように `/contrib/dblink` を修正しました。(Joe)

[前のページ](#)

リリース8.1.3

[ホーム](#)
[上に戻る](#)

[次のページ](#)

リリース8.1.1

E.118. リリース8.1.1

リリース日: 2005-12-12

このリリースは8.1.0の各種不具合を修正したものです。 8.1メジャーリリースにおける新機能については[項E.119](#)を参照してください。

E.118.1. バージョン8.1.1への移行

8.1.Xからの移行ではダンプ/リストアは不要です。

E.118.2. 変更点

- 外部結合条件の不正確な最適化を修正しました。(Tom)
- オプティマイザにより平坦化された副問い合わせを含む場合に列名が間違っ報告される問題を修正しました。(Tom)
- CHECK制約、TOAST化された列を含む状況における更新失敗を修正しました。(Tom)
- エラー修復後のbgwriter問題を修正しました。(Tom)

書き込みエラーの後、バックグラウンドライタがあふれたバッファピンを見つけ出していました。これ自体は致命的ではありませんが、これによりその後のVACUUMコマンドが不可解なブロックを行う可能性があります。

- 現在のトランザクションがすでにアボートしている時にクライアントがBindプロトコルメッセージを送信した場合の失敗を防ぎます。
- `/contrib/tsearch2`および`/contrib/ltree`を修正しました。(Teodor)
- トルコ語など単語の順番を変更する必要がある言語において、翻訳されたエラーメッセージに関する問題を修正しました。また、想定外の出力文字列の消去問題およびbigint値の最小値の間違った表示問題を修正しました。(Andrew, Tom)

これらの問題は、`port/snprintf.c`コードを使用するプラットフォームでのみ発生しました。こうしたプラットフォームには`--enable-nls`が指定された場合のBSD系なども含まれます。さらに、翻訳されたエラーメッセージに関するその他の問題が、使用する`libintl`のバージョンによってWindowsでも発生しました。

- `to_char(time)`および`to_char(interval)`で、AM/PM、HH、HH12、D書式指定を再度許可しました。
(`to_char(interval)`ではおそらくHH24を使用しなければなりません。)(Bruce)
- AIX、HPUX、MSVCにおけるコンパイルを修正しました。(Tom, Hiroshi Saito)
- オプティマイザを改良しました。(Tom)
- WindowsのNO_SYSTEM_RESOURCESエラーが発生した後に、ファイル読み書きを再試行します。(Qingqing Zhou)
- 式インデックスのANALYZE中のクラッシュではautovacuumを防止します。(Alvaro)
- 一時テーブルに対するON COMMIT DELETE ROWSに関する問題を修正しました。
- トリガがSELECT DISTINCT問い合わせの出力を変更する場合の問題を修正しました。
- 無効なUTF-8バイト列の移行方法に関する項目を8.1.0のリリースノートに追加しました。(Paul Lindner)

E.119. リリース8.1

リリース日: 2005-11-08

E.119.1. 概要

このリリースの主な変更点を以下に示します。

共有バッファキャッシュへの同時アクセスの改良(Tom)

共有バッファキャッシュへのアクセスが、特に、マルチCPUシステムにおいて深刻なスケーラビリティ問題であることは分かっていました。本リリースでは、バッファマネージャ内で行われていたロックの方法を見直し、ロックの競合を軽減し、スケーラビリティを向上しました。また、バッファマネージャは"clock sweep"置換ポリシーを使用するように変更されました。

インデックススキャンが途中でメモリ内ビットマップを使用することができるようになりました。

過去のリリースでは、テーブル検索の実行において単一インデックスのみを使用することができました。この機能により、`WHERE tab.col1 = 4 and tab.col2 = 9`を持つ問い合わせで、`col1`と`col2`に対する複数列インデックスが存在しない場合、`col1`に対するインデックスと`col2`に対するインデックスがあれば、両方のインデックスを検索し、結果をメモリ内で組み合わせ、`col1`と`col2`に関する両方の制約に一致する行のみをヒープから取り出すことができるようになります。これは、問い合わせが構造化されておらず、すべての起こり得るアクセス条件に一致するインデックスを前もって作成できないような状況では、非常に有用です。必要なランダムアクセスの総数を減少させるため、ビットマップスキャンは単一のインデックスの場合でも有用です。ビットマップインデックススキャンはテーブル全体で多くの割合を占める部分を取り出す際に効率的です。この場合、通常のインデックススキャンでは非効率的です。

二相コミットの追加(Heikki Linnakangas, Alvaro, Tom)

二相コミットを使用して、トランザクションを複数のコンピュータで"準備"することができます。すべてのコンピュータでトランザクションの準備を（失敗することなく）行った後、すべてのトランザクションをコミットすることができます。準備した後でマシンがクラッシュしたとしても、マシンの再起動後に準備したトランザクションをコミットすることができます。 `PREPARE TRANSACTION`および`COMMIT/ROLLBACK PREPARED`という新しい構文があります。また、新しい`pg_prepared_xacts`システムビューも追加されました。

ユーザとグループを置き換える、新しいロールシステムの作成(Stephen Frost)

ロールとは、ユーザとグループを組み合わせたものです。ユーザのようにロールはログイン機能を持つことができますし、グループのようにロールは他のロールをメンバとして持つこともできます。基本的にロールは、ユーザとグループの区別を取り除きました。例えば、ロールは以下のことを行うことができます。

- ログイン機能を有す（省略可能）
- オブジェクトを所有する
- データベースオブジェクトに対するアクセス権限を保持する
- 属している他のロールの権限を継承する

ユーザがあるロールにログインした後、そのユーザはログインロールの能力の他、継承したロールの能力も持つことになります。また、`SET ROLE`を使用して、属している他のロールに変わることもできます。この機能は、標準SQLのロールの概念を汎用化したものです。またこの変更により、`pg_shadow`と`pg_group`は、新しいロール機能の付いたカタログ`pg_authid`と`pg_auth_members`に置き換わりました。古いテーブルは、新しいロールテーブルに対する読み取りのみのビューとして再定義されています。

`MIN()` と `MAX()` におけるインデックスの自動使用 (Tom)

過去のリリースでは、`MIN()` または `MAX()` でインデックスを使用する方法は、問い合わせを `SELECT col FROM tab ORDER BY col LIMIT 1` に書き換える方法しかありませんでした。インデックスが自動的に使用されるようになりました。

`/contrib/pg_autovacuum` を主サーバに移動しました。(Alvaro)

`autovacuum` をサーバに統合することで、データベースサーバと同期して自動的に `autovacuum` が起動、停止できるようになりました。また、`postgresql.conf` で `autovacuum` の設定を行うことができるようになりました。

`SELECT ... FOR SHARE`を使用した共有行レベルロックの追加(Alvaro)

PostgreSQLのMVCCロックにより`SELECT`は書き込みによってブロックされることはありません。そのため、通常の操作では共有行レベルロックは不要ですが、共有行ロックを必要とするアプリケーションでは共有ロックは有用です。特に、これにより参照性合成検査で発生するロック要求が減少します。

共有オブジェクト、特にロールへの依存性を追加(Alvaro)

依存性機構の拡張により、データベースオブジェクトを所有するロールが削除されることを防ぎます。これまでは、所有者の削除により"親のいない"オブジェクトができてしまう可能性がありました。これは復旧させることができたが、厄介で好まれないものでした。

テーブルパーティションによる性能向上(Simon)

新しい`constraint_exclusion`設定パラメータは、子テーブルに一致する行が存在しないことを制約が示している場合、子テーブルの検索を防ぎます。

これにより、テーブルパーティションの基本的な種類を実現できます。子テーブルが別々のキー範囲を格納し、これを適切な`CHECK`制約で強制している場合、オプティマイザは、この制約により子テーブル内に一致することがないことが保証されている時にその子テーブルへのアクセスを飛ばします。

E.119.2. バージョン8.1への移行

全ての以前のリリースからデータの移行を行うためには、`pg_dump`を使用したダンプ、リストアが必要です。

8.0リリースでは、8.1で`interval`に対する`to_char()`を削除するとアナウンスしましたが、より優れたAPIの提案がありませんでしたので、`to_char(interval)`は強化され、サーバに残りました。

以下の非互換性に注意してください。

- `add_missing_from`のデフォルトが`false`になりました(Neil)

`FROM`参照を持たない問い合わせでテーブルが使用された場合、デフォルトでエラーとなるようになりました。まだ過去の動作を使用することができますが、そのためにはこのパラメータを`'true'`にしなければなりません。

ダンプに暗黙的な`FROM`構文を使用して作成したビューやルールがある場合、既存のダンプファイルをロードするために、`add_missing_from`を真に設定しなければならないかもしれません。これに悩まされるのは一回だけのはずです。PostgreSQL 8.1は、こうしたビューやルールを標準の明示的な`FROM`構文に変換するためです。したがって、今後のダンプではこの問題は起こりません。

- `float4/float8/oid`に長さが0の文字列('')が入力された場合、ゼロとして扱わずエラーとなるようになりました(Neil)

この変更により、現行の整数に対する長さが0の文字列の扱いと矛盾がなくなりました。この変更の時期については8.0でアナウンスしていました。

- `default_with_oids`のデフォルトが`false`になりました(Neil)

このオプションを`false`に設定した場合、`CREATE TABLE`で`WITH OIDS`を指定しない限り、ユーザが作成したテーブルはOID列を持たなくなります。すべてのPostgreSQLリリースでOIDは存在していましたが、4バイト長しかなく、インストールされたデータベース全体で共有アクセスされていることから、OIDの利用は制限されてきました。一意に行を識別するには、シーケンスまたは`SERIAL`型を使用することを勧めます。これはPostgreSQL 6.4からサポートされています。

- `E''`構文が追加されました。そのため、普通の文字列でバックスラッシュ文字を扱うことができます。(Bruce)

現在のPostgreSQLは文字列リテラル内のバックスラッシュを、特別なエスケープシーケンスの始まりとして扱います。例えば、`¥n`や`¥010`です。これで簡単に特別な値を埋め込むことができますが、非標準であり、他のデータベースからのアプリケーションの移植を困難にしています。このため、PostgreSQLプロジェクトでは、文字列内のバックスラッシュが特別な意味を持たないようにすることを計画しています。後方互換性と、特別なバックスラッシュ処理を使用したいユーザのために、新しい文字列構文を作成しました。この新しい文字列構文は、例えば`E'hi¥n'`のように、`E`、直後に単一引用符、そして文字列を始めるという形です。このリリースでは文字列内のバックスラッシュの扱いに変更はありませんが、今後のリリースに向けたアプリケーションの移行を補助するために、以下の設定パラメータが追加されました。

- `standard_conforming_strings`。このリリースで普通の文字列内のバックスラッシュをそのまま扱うかどうかを指定します。
- `escape_string_warning`。普通の文字列（非E）内でバックスラッシュが存在した場合に警告する。

`standard_conforming_strings`の値は読み取りのみです。アプリケーションでは、この値を読み取ることで、どのようにバックスラッシュが解釈されるかが分かります。（また、このパラメータが存在するかどうかで、`E''`文字列構文がサポートされているかが分かります。）今後のリリースでは、`standard_conforming_strings`は真になる予定です。つまり、非E文字列でのバックスラッシュは文字そのものとして扱われます。この変更に合わせて、バックスラッシュの特別処理が必要な場所に`E''`を使用し、かつ、`E''`を使用するように変換しなければならない文字列を更に検出するために`escape_string_warning`を有効にしてください。また、文字列内に単一引用符そのものを埋め込む場合には、PostgreSQLがサポートするバックスラッシュ+単一引用符（`¥'`）という構文ではなく、2つの単一引用符（`''`）を使用してください。後者は標準に準拠しており、また、`E''`文字列構文を使用する必要がありません。また、バックスラッシュを特別に扱わない、`$$`文字列構文も使用することができます。

- `REINDEX DATABASE`がデータベース内のすべてのインデックスの再作成を行うようになりました。(Tom)

これまでは、`REINDEX DATABASE`はシステムテーブルのみのインデックスを再作成しました。この新しい動作の方がより直感的です。新しい`REINDEX SYSTEM`コマンドが、システムテーブルのみのインデックス再作成という以前の機能を提供します。

- 読みとりのみのラジオブジェクト記述子がMVCCスナップショットセマンティックスに従うようになりました。

ラジオブジェクトが`INV_READ` (かつ`INV_WRITE`なし)で開かれた場合、その記述子から読みとられるデータが、`lo_open()`を呼び出した問い合わせで使用するトランザクションスナップショットの時点のラジオブジェクトの状態"スナップショット"を反映するようになりました。常に最新のコミットされたデータを返すという以前の動作に戻すには、`lo_open()`のモードフラグに`INV_WRITE`を付与して下さい。

- シーケンス関数の引数に対して、適切な依存性を追加しました。(Tom)

以前のリリースでは、`nextval()`、`currval()`、`setval()`に渡されるシーケンス名は、単なるテキスト文字列として保持されていました。つまり、`DEFAULT`句で使用中のシーケンスの名称変更や削除により、`DEFAULT`句が無効となりました。このリリースでは、すべて新しく作成した、内部用OIDを引数として持つシーケンス関数を保持します。これにより、シーケンス名の変更を追跡できるようになり、また、不適切なシーケンスの削除を防ぐために依存情報も追加されました。また、これにより、スキーマ名称の変更や検索パスの変更から、こうした`DEFAULT`句を保護できるようになりました。

アプリケーションの中には、以前の実行時にシーケンス名を検索する動作に依存しているものがあるかもしれません。これは、`nextval('myseq'::text)`のように、明示的に引数を`text`にキャストすることで実現することができます。

8.1より前のデータベースダンプを8.1にロードした場合、古いテキストを基とした表現が使用されますので、OIDを引数として保持させる機能は持ってません。しかし、テキスト版の`DEFAULT`句を持つデータベースを更新することができます。まず、以下の問い合わせを`fixseq.sql`などという名前のファイルに保存してください。

```
SELECT  'ALTER TABLE ' ||
        pg_catalog.quote_ident(n.nspname) || '.' ||
        pg_catalog.quote_ident(c.relname) ||
        ' ALTER COLUMN ' || pg_catalog.quote_ident(a.attname) ||
        ' SET DEFAULT ' ||
        regexp_replace(d.adsrc,
                        $$val¥¥('^[^']*')::text¥)::regclass$$,
                        $$val¥¥1$$,
                        'g') ||
        ';'
FROM    pg_namespace n, pg_class c, pg_attribute a, pg_attrdef d
WHERE   n.oid = c.relnamespace AND
        c.oid = a.attrelid AND
        a.attrelid = d.adrelid AND
        a.attnum = d.adnum AND
        d.adsrc ~ $$val¥¥('^[^']*')::text¥)::regclass$;
```

次に、調整が必要であることが分かったデータベースに対してこの問い合わせを実行してください。以下では`db1`というデータベースに対して実行しています。

```
psql -t -f fixseq.sql db1
```

これにより、データベースを新しいOID版の表現に変換するための`ALTER TABLE`コマンドが表示されます。このコマンドが問題ないことを確認した後に、以下を実行してデータベースを更新してください。

```
psql -t -f fixseq.sql db1 | psql -e db1
```

この手続きを更新すべきデータベースそれぞれに対して繰り返し実行しなければなりません。

- `psql`において、引用符で括られていない`¥{digit}+`という並びを8進数として扱うようになりました。(Bruce)

以前のリリースでは、`¥{digit}+`という並びは10進数として扱われ、`¥0{digit}+`のみが8進数として扱われました。この変更は一貫性のために行われました。

- `%`および`^`演算子の接頭辞と接尾辞の文法積を削除しました。

これは文書化されておらず、また、負値の剰余演算子 (%)の使用を複雑にしていました。

- 多角形用の<&と>をボックス型の"over"演算子と矛盾がないようにしました。(Tom)
- `CREATE LANGUAGE`は、`pg_pltemplate`の情報によって指定された引数を見捨てるすることができます。(Tom)

新しい`pg_pltemplate`システムカタログが、手続き言語の好みの定義（検証関数をもつかどうかなど）についての情報を保持するために定義されました。作成しようとする言語に関する項目がこのカタログに存在する場合、`CREATE LANGUAGE`は言語名以外のパラメータをすべて無視し、カタログの情報を使用します。古いダンプファイルからロードされる古い言語定義に伴う問題が増大したため、この方式が取られました。8.1では、ロード時にテンプレート項目が存在することに依存し、`pg_dump`は手続き言語定義を単なる `CREATE LANGUAGE name`としてダンプします。これがより将来を見込んだ表現になることを期待しています。

- `pg_cancel_backend(int)` が `integer` ではなく `boolean` を返すようになりました。(Neil)
- UTF-8のデータを8.1.Xにロードする場合に問題が起きる可能性があります。これは、以前のバージョンでは無効なUTF-8のバイト列をデータベースに投入することができましたが、今回のリリースでは正しく有効なUTF-8列のみを受け付けるようになったことが原因です。ダンプファイルを修正する方法の1つとして、`iconv -c -f UTF-8 -t UTF-8 -o cleanfile.sql dumpfile.sql`を実行することがあります。`-c`オプションにより無効な文字列が削除されます。この2つのファイルの違いを見ることで、無効な列を表示することができます。`iconv`は入力ファイル全体をメモリに読み込みますので、この処理のためにダンプを複数の小さなファイルに分割する必要があるかもしれません。

E.119.3. その他の変更点

PostgreSQL8.1と以前のメジャーリリースとの間でなされた、その他の変更点について詳細を以下に示します。

E.119.3.1. 性能向上

- GiSTとR-treeインデックスの性能を向上しました。(Neil)
 - ハッシュ結合の自動サイズ変更を含む、オプティマイザの改良を行いました。(Tom)
 - 様々な所で内部用のAPIを見直しました。
 - WALレコードのCRCを64ビットから32ビットに変更しました。(Tom)
- 64ビットのCRCを計算するコストが過大で、信頼性に関する長所は無視できるものであると考えました。
- WALページに巨大な空のギャップを書き出さないように変更しました。(Tom)
 - SMPマシン、特にOpteronマシンにおけるスピンロック動作を改良しました。(Tom)
 - 連続していないインデックス列を複数列に対するインデックスで使えるようにしました。(Tom)

例えば、これによりa,b,c列に対するインデックスを`WHERE a = 4 and c = 10`といった問い合わせで使うことができます。

- `CREATE TABLE AS/SELECT INTO`に対するWALロギングを飛ばします。(Simon)

`CREATE TABLE AS`実行中のクラッシュにより復旧中にテーブルが削除されてしまいますので、そのテーブルをロードするためにWALに記録する意味がありません。（しかし、WALアーカイブが有効な場合はロギングを行います。）

- GiSTインデックスへの同時アクセスを可能にしました。(Teodor, Oleg)
- `full_page_writes`設定パラメータを追加し、WALへページ全体を書き出すかどうか制御できるようになりました。(Bruce)

破損したデータベースから部分的なディスク書き出しを防ぐために、チェックポイントの後に変更されたディスクページを最初に取り出す時に、PostgreSQLは各データベースディスクページ全体のコピーをWALに書き出します。このオプションにより、この機能を無効にすることができ、より高速になります。部分的なページ書き出しが起こることがない、バッテリでバックアップされたディスクキャッシュを持つ場合、これは安全です。

- `wal_sync_method`で`O_SYNC`を使用している場合、利用できるのであれば`O_DIRECT`を使用します。(Itagaki Takahiro)

`O_DIRECT`によりディスクへの書き出しがカーネルキャッシュを介さずに行われます。そのため、WALへの書き出しが高速になり、性能が向上します。

- `COPY FROM`の性能を向上しました。(Alon Goldshuv)
`COPY`入力を一文字ずつではなくより巨大なまとまり内に読み込むことで実現しました。
- `COUNT()`、`SUM`、`AVG()`、`STDDEV()`、`VARIANCE()`を改良しました。(Neil, Tom)

E.119.3.2. サーバの変更

- トランザクションID(XID)周回問題を防止しました。(Tom)
 トランザクションカウンタが周回点に近づいた時にサーバは警告を発します。カウンタが周回点に近づき過ぎている場合、サーバは問い合わせの受付を停止します。これにより、必要なバキューム処理の実行を行う前にデータが損失されることは確実になくなります。
- OIDカウンタが周回した後に生じる、既存のシステムオブジェクトとオブジェクトID (OID) が競合する問題を修正しました。
- `VACUUM`時に`max_fsm_relations`と`max_fsm_pages`を増大させる必要がある場合の警告を追加しました。
- `temp_buffers`設定パラメータを追加し、一時テーブルアクセス用のローカルバッファのサイズをユーザが決定できるようになりました。
- `pg_stat_activity`にセッションの開始時刻とクライアントIPアドレスを追加しました。(Magnus)
- ビットマップスキャンのために`pg_stat`ビューを調整しました。(Tom)
 一部のフィールドの意味が少し変わりました。
- `pg_locks`ビューを改良しました。(Tom)
- クライアント側における`PREPARE`および`EXECUTE`のログを記録するようにしました。(Simon)
- Kerberos名とユーザ名に関する大文字小文字の区別を`postgresql.conf`で指定できるようにしました。(Magnus)
- `krb_server_hostname`設定パラメータを追加し、サーバプリンシパルの一部としてサーバのホスト名を指定できるようになりました。(Todd Kover)
 設定されていない場合、`keytab`内の一致するサービスプリンシパルが使用されます。これは本リリースにおける新しいKerberosの検索動作です。
- `log_line_prefix`オプションにミリ秒(`%m`)とリモートホスト(`%h`)を追加しました。(Ed L.)
- GiSTインデックスに対するWALロギングを追加しました。(Teodor, Oleg)
 GiSTインデックスはクラッシュに対してもポイントインタイムリカバリに対しても安全になりました。
- `pg_stop_backup()`を実行した時に古い`*.backup`ファイルを削除するようにしました。(Bruce)
 これにより、多くの`*.backup`ファイルが`pg_xlog`に残らないようになります。
- TCP/IPキープアライブ時間について待機、間隔、個数を制御する設定パラメータを追加しました。(Oliver Jowett)
 これらの値を変更することで、クライアント接続の喪失を素早く検知できるようになります。
- ユーザ毎およびデータベース毎の接続上限を追加しました。(Petr Jelinek)
`ALTER USER`および`ALTER DATABASE`を使用して、特定ユーザが同時に接続できる、もしくは特定のデータベースへ同時に接続できる、最大セッション数に制限をかけることができるようになりました。この制限をゼロにすると、ユーザからの接続またはデータベースへの接続が無効になります。
- 64ビットマシンにおいて、2ギガバイト以上の共有メモリとバックエンド単位の作業メモリを使用できるようになりました。(Koichi Suzuki)
- 新しい`pg_pltemplate`によりダンプファイル内の廃止された手続き言語定義を上書きできるようになりました。(Tom)

E.119.3.3. 問い合わせの変更

- 一時ビューを追加しました。(Koji Iijima, Neil)

- 集約関数やGROUP BYを持たないHAVINGを、単一グループを返すように修正しました。(Tom)

これまでこうした状況では、HAVING句をWHERE句と同様に扱っていました。これは仕様に従っていませんでした。

- DELETEにテーブルを追加指定できるようにUSING句を追加しました。(Euler Taveira de Oliveira, Neil)

以前のリリースでは、DELETE文内で結合に使用するテーブルを追加指定する明確な方法はありませんでした。UPDATEには、すでにこの目的のためのFROMがあります。

- バックエンドおよびecpg文字列における¥x 16進数エスケープのサポートを追加しました。(Bruce)

これは、標準Cの¥xエスケープ構文そのものです。8進数エスケープはサポート済みです。

- BETWEEN SYMMETRIC問い合わせ構文を追加しました。(Pavel Stehule)

この機能によりBETWEEN比較で、1番目の値が2番目より小さいことを必要としなくなりました。例えば、2 BETWEEN [ASYMMETRIC] 3 AND 1は偽を返しますが、2 BETWEEN SYMMETRIC 3 AND 1は真を返します。BETWEEN ASYMMETRICはすでにサポート済みです。

- SELECT ... FOR UPDATE/SHAREのオプションにNOWAITを追加しました。(Hans-Juergen Schoenig)

statement_timeout設定パラメータにより、一定以上の時間がかかる問い合わせをキャンセルすることができますが、このNOWAITオプションにより、SELECT ... FOR UPDATE/SHAREが即座に行ロックを獲得できなかった時、すぐに問い合わせをキャンセルすることができます。

E.119.3.4. オブジェクト操作の変更

- 共有オブジェクトの依存関係の追跡(Alvaro)

PostgreSQLでは、グローバルテーブル（ユーザ、データベース、テーブル空間）の情報を複数のデータベースから参照することができます。今回の変更でグローバルテーブルに関する依存性情報が追加されました。従って、例えば、ユーザの所有権をデータベースに跨って確認できますので、他のデータベースで何かを所有しているユーザを削除できないようにすることができます。データベースに局所的なオブジェクトに関する依存関係の追跡はすでに存在します。

- オブジェクト所有者が制限付きのALTER OWNERコマンドを実行できるようになりました。(Stephen Frost)

以前のリリースでは、スーパーユーザのみがオブジェクトの所有権を変更できました。今では、コマンドを実行したユーザがオブジェクトを所有し、かつ、新しい所有者として作成できる（つまり、そのユーザが新しい所有者のロールのメンバであり、そのロールが新規にオブジェクトを作成する時に必要なCREATE権限を持つ場合）場合、所有権を移すことができます。

- 一部のオブジェクト型（テーブル、関数、型）に対するALTER object SET SCHEMA機能を追加しました。(Bernd Helmle)

これによりオブジェクトを異なるスキーマに移すことができるようになります。

- トリガを無効にするALTER TABLE ENABLE/DISABLE TRIGGERを追加しました。(Satoshi Nagayasu)

E.119.3.5. ユーティリティコマンドの変更

- 単一のTRUNCATEコマンドで複数のテーブルの消去ができるようになりました。(Alvaro)

参照整合性検査のため、参照整合性制約の一部となるテーブルを消去することはできません。この新しい機能を使用することで、参照整合性制約に含まれる両テーブルが単一のTRUNCATEコマンドで消去される場合、TRUNCATEを使用してこうしたテーブルを消去できるようになりました。

- COPY CSVモードにおける復帰コード、改行コードの処理が適切になりました。(Andrew)

8.0リリースでは、CSV COPY TOにおける復帰コードと改行コードは一貫性のある方法で処理されていませんでした。（これはTODOリストに記載されていました。）

- COPYにて、先頭行にヘッダ行を記載できるようにCOPY WITH CSV HEADERを追加しました。(Andrew)

これにより、データファイルの先頭行に列名を記載するという一般的なCSVの使用方法を扱うことができます。COPY TOでは先頭行には列名が記載され、COPY FROMでは先頭行は無視されます。

- Windowsにおいて、EXPLAIN ANALYZEの表示を秒以下の精度にしました。(Magnus)

- EXPLAIN ANALYZEの表示にトリガ期間を追加しました。(Tom)

以前のリリースではトリガの実行時間は総実行時間に含まれていました。各トリガで費やされる時間がどのくらいかわかるようになりました。

- `COPY`にて、`¥x16`進数エスケープのサポートを追加しました。(Sergey Ten)

以前のリリースでは8進数エスケープのみをサポートしていました。

- `SHOW ALL`が変数の説明を含むようにしました。(Matthias Schmidt)

`SHOW varname` はまだ変数の値のみを表示し、説明を表示しません。

- `initdb`が`postgres`という新しい標準データベースを作成するようになりました。このためユーティリティが標準データベースを検索する時に`template1`ではなく`postgres`を使用するように変更されました。(Dave)

以前のリリースでは、`createuser`などのユーティリティのデフォルト接続用、および、新規データベースのテンプレート用として、`template1`が使用されていました。テンプレートデータベースにだれかが接続しているために新しいデータベースを作成することができず、`CREATE DATABASE`の失敗の原因となることがありました。この変更により、デフォルトの接続データベースが`postgres`になりました。つまり、`CREATE DATABASE`時にだれかが`template1`を使用している可能性がかなり減少しました。

- `/contrib/reindexdb`をサーバへ移動し、新しく`reindexdb`コマンドラインユーティリティを作成しました。(Euler Taveira de Oliveira)

E.119.3.6. データ型および関数の変更

- 配列型に対する`MAX()` および`MIN()` 集約を追加しました。(Koju Iijima)

- `CC`、`YY`、またはこの両方のフィールドが使用されている場合に合理的な動作を行うように`to_date()` および`to_timestamp()` を修正しました。(Karel Zak)

`CC` が書式指定にあり、`YYY`以上の長さが年指定にあると、`CC` を無視します。年指定の長さが`YY`以下の場合、`CC`を直前の世紀であると解釈します。

- `md5(bytea)` を追加しました。(Abhijit Menon-Sen)

`md5(text)` はすでに存在していました。

- `power(numeric, numeric)`を基にした`numeric ^ numeric`のサポートを追加しました。

関数はすでに存在していましたが、対応する演算子がありませんでした。

- 計算処理中に適切に商を除去するように`NUMERIC`の剰余を修正しました。

以前のリリースでは、巨大な値に対する剰余が商の丸めにより負の値を返すことがありました。

- `lastval()` 関数を追加しました。(Dennis Björklund)

`lastval()` は`currval()`を簡単にしたものです。これは、現在のセッションで実行されたもっとも最近の`nextval()` または`setval()`に基づいて自動的に適切なシーケンス名を決定します。

- `to_timestamp(DOUBLE PRECISION)` を追加しました。(Michael Glaesemann)

1970年からのUnix秒を`TIMESTAMP WITH TIMEZONE`に変換します。

- `pg_postmaster_start_time()` 関数を追加しました。(Euler Taveira de Oliveira, Matthias Schmidt)

- `AT TIME ZONE`で完全なゾーン名を使用できるようになりました。以前は省略形のみが使用できました。(Magnus)

以前は、定義済みのタイムゾーン名の一覧のみが`AT TIME ZONE`でサポートされていました。現在では、サポート済みのタイムゾーン名すべてを使用することができます。以下に例を示します。

```
SELECT CURRENT_TIMESTAMP AT TIME ZONE 'Europe/London';
```

上の問い合わせでは、タイムゾーンは指定した日付に影響する夏時間規則に従って調整されます。

- `GREATEST()` および`LEAST()` 可変長引数関数を追加しました。(Pavel Stehule)

これらの関数は可変数の引数を取り、引数の中で最大もしくは最小の値を返します。

- `pg_column_size()` を追加しました。(Mark Kirkwood)

これは列の格納サイズを返します。圧縮が使用されている場合もあります。

- `regexp_replace()` を追加しました。(Atsushi Ogawa)

これによりsedのように正規表現による置換を行うことができます。省略可能なフラグ引数により、全領域の選択（全置換）や大文字小文字の区別モードを指定することができます。

- `interval`の除算、乗算を修正しました。(Bruce)

以前のバージョンでは、`'4 months'::interval / 5`が`'1 mon -6 days'`といった不揃いの結果を返すことがありました。

- `timestamp`、`time`、`interval`出力における端数処理を修正しました。(Tom)

上位部分を増加させずに秒部分が60と表示される問題を修正したものです。

- `INTERVAL`に別途日フィールドを追加し、1日という間隔と24時間という間隔を区別できるようになりました。(Michael Glaesemann)

夏時間による時間調整を含む日にちは24時間ではなく、よく23時間や25時間となります。この変更により、"何日間"という時間間隔と"何時間"という時間間隔の間に概念的な違いが生じます。`timestamp`に`1 day`を加えると、その間に夏時間による時間調整があったとしても、同じローカル時間における次の日になるようになります。一方で`24 hours`を加えると、時間調整があると異なるローカル時間となります。US DSTの例を以下に示します。

```
'2005-04-03 00:00:00-05' + '1 day' = '2005-04-04 00:00:00-04'
'2005-04-03 00:00:00-05' + '24 hours' = '2005-04-04 01:00:00-04'
```

- `justify_days()` および `justify_hours()` を追加しました。(Michael Glaesemann)

これらの関数はそれぞれ、日数を適切な月数と日数に、時間数を適切な日数と時間数に調整します。

- `/contrib/dbsize` をバックエンドに移し、一部の関数の名前を変更しました。(Dave Page, Andreas Pflug)

- `pg_tablespace_size()`
- `pg_database_size()`
- `pg_relation_size()`
- `pg_total_relation_size()`
- `pg_size_pretty()`

`pg_total_relation_size()` にはインデックスとTOASTテーブルが含まれます。

- クラスタディレクトリへの読み取りのみのファイルアクセス用の関数を追加しました。(Dave Page, Andreas Pflug)

- `pg_stat_file()`
- `pg_read_file()`
- `pg_ls_dir()`

- 設定ファイルを強制的に再読み込みする `pg_reload_conf()` を追加しました。(Dave Page, Andreas Pflug)

- サーバログファイルを強制的にローテートする `pg_rotate_logfile()` を追加しました。(Dave Page, Andreas Pflug)

- TOASTテーブルを含めるように `pg_stat_*` ビューを変更しました。(Tom)

E.119.3.7. 符号化方式およびロケールの変更

- より一貫性を高め、かつ、国際的な標準に従うように一部の符号化方式の名前を変更しました。

- `UNICODE` は `UTF8` になりました。
- `ALT` は `WIN866` になりました。
- `WIN` は `WIN1251` になりました。
- `TCVN` は `WIN1258` になりました。

以前の名前もまだ使用できます。

- WIN1252 符号化方式のサポートを追加しました。(Roland Volkmann)
- 4バイトのUTF8 文字のサポートを追加しました。(John Hansen)

これまでは1バイト、2バイトおよび3バイトのUTF8 文字がサポートされていました。これは特に一部の中国語文字セットをサポートするために重要です。

- EUC_JP、SJIS 間の直接変換を可能とし、性能を向上しました。(Atsushi Ogawa)
- WindowsにおいてUTF8符号化方式が動作できるようになりました。(Magnus)

これはUTF8をWindows独自のUTF16実装に対応付けすることでなされました。

E.119.3.8. サーバサイド言語の一般的な変更

- ALTER LANGUAGE RENAME を修正しました。(Sergey Yatskevich)
- 厳密性、揮発性などの関数特性をALTER FUNCTION で変更できるようにしました。(Neil)
- 関数引数の最大数を100まで増加しました。(Tom)
- SQL関数およびPL/pgSQL関数でOUT およびINOUT パラメータを使用できるようにしました。(Tom)

OUT は、関数の戻り値をあらわす別の方法です。RETURN の代わりに使用して、OUT またはINOUT で宣言されたパラメータに割り当て、値を返します。これにより、特に複数の値を返さなければならない場合、記述がより簡単になる場合があります。以前のリリースでも関数から複数の値を返すことは可能でしたが、これによりその手続きが非常に簡単になりました（今後のリリースでこの機能は他のサーバサイド言語にも拡張される予定です）。

- 言語ハンドラ関数をpg_catalog スキーマに移動しました。
これによりpublicスキーマの削除が簡単にできるようになりました。
- SPI_getnspname() をSPIに追加しました。(Neil)

E.119.3.9. PL/pgSQLサーバサイド言語の変更

- PL/pgSQL関数のメモリ管理を全面的に見直しました。(Neil)

各関数の解析ツリーが異なるメモリコンテキストに保存されるようになりました。これにより、不要になった時にメモリの回収が簡単にできるようになりました。

- 実行時ではなく、CREATE FUNCTION 時に関数の構文を検査します。(Neil)

以前は、構文エラーのほとんどが関数を実行した時にのみ報告されていました。

- OPEN により、EXPLAIN やSHOW などのSELECT 以外の問い合わせを開くことができるようになりました。(Tom)
- 関数でRETURN 文を実行する必要がなくなりました。(Tom)

これはOUT およびINOUT 機能の新規追加に伴う副産物です。関数の戻り値を提供する必要がない場合RETURN を省略することができます。

- PL/pgSQLのEXECUTE 文における省略可能なINTO 句サポートを追加しました。(Pavel Stehule, Neil)
- CREATE TABLE AS がROW_COUNT を設定するようになりました。(Tom)
- 現在の例外に関するSQLSTATE とエラーメッセージを返すSQLSTATE とSQLERRM を定義しました。(Pavel Stehule, Neil)
これらの変数は、例外ブロック内でのみ定義されます。
- RAISE 文のパラメータを式とすることが可能になりました。(Pavel Stehule, Neil)
- ループのCONTINUE 文を追加しました。(Pavel Stehule, Neil)
- ブロックラベルとループラベルを追加しました。(Pavel Stehule)

E.119.3.10. PL/Perlサーバサイド言語の変更

- 大規模な結果セットを効率的に返すことが可能になりました。(Abhijit Menon-Sen)
これにより関数で、結果セット全体をメモリ内に構築することなく `return_next()` が使用できるようになりました。
- 一度に一行という形で、問い合わせ結果の取り出しが可能になりました。(Abhijit Menon-Sen)
これにより関数で、結果セット全体をメモリ内に保存することなく `spi_query()` と `spi_fetchrow()` が使用できるようになりました。
- サーバの符号化方式がUTF8の場合、PL/Perlは文字列を強制的にUTF8として扱うようになりました。(David Kamholz)
- PL/Perlで検証関数を追加しました。(Andrew)
これにより、構文エラーを実行時ではなく定義時に報告できるようになりました。
- 関数が配列型を返す場合にPL/PerlがPerlの配列を返すことができるようになりました。(Andrew)
これは基本的にはPostgreSQL配列をPerl配列に対応付けします。
- Perlの非致命的警告でNOTICEメッセージを生成できるようになりました。(Andrew)
- Perlのstrictモードを有効にできるようになりました。(Andrew)

E.119.3.11. psqlの変更

- トランザクション内の文がトランザクションの残りに影響することなくエラーになることができるように、`%set ON_ERROR_ROLLBACK`を追加しました。(Greg Sabino Mullane)
これは、基本的に、すべての文を副トランザクション内で行うことで実装しています。
- psql変数内で`%x`16進数文字列のサポートを追加しました。(Bruce)
8進数エスケープはすでにサポートされていました。
- `troff -ms`出力書式のサポートを追加しました。(Roger Leigh)
- 履歴ファイルの格納場所をHISTFILEで制御可能にしました。(Andreas Seltenreich)
これによりデータベースごとに履歴ファイルの格納設定が可能になります。
- `%x` (拡張モード)が`%d tablename`の結果に影響しないようにしました。(Neil)
- セッションのログを記録するための-Lオプションをpsqlに追加しました。(Lorne Sunley)
オペレーティングシステムの中には簡単にコマンドライン操作をログ記録する機能がないものがありましたので、このオプションが追加されました。
- `%d`がインデックスのテーブル空間を表示できるようにしました。(Qingqing Zhou)
- psqlのヘルプ (`%h`) が適切なヘルプ情報について最善の推測を行うようになりました。(Greg Sabino Mullane)
これによりユーザは単に構文エラーとなった問い合わせの前に`%h`を付けるだけで、サポートしている構文のヘルプ情報を得ることができるようになりました。これまでは、`%h`を使用する前にコマンド名以外の問い合わせテキストを削除しなければなりませんでした。
- ロケールを考慮した書式で数値を出力できるように`%pset numericlocale`を追加しました。(Eugen Nedelcu)
例えば、cロケールでは100000は100,000.0として、Europeanロケールでは 100.000,0として出力されます。
- 起動時のバナーでサーバのバージョン番号とpsqlのバージョン番号が異なる場合に両方とも表示するようにしました。(Bruce)
また、サーバとpsqlとの間でメジャーリリース番号が異なる場合に警告を発します。

E.119.3.12. pg_dumpの変更

- pg_restoreに-nおよび--schemaスイッチを追加しました。(Richard van den Berg)

これにより指定したスキーマのオブジェクトのみをリストアすることができます。

- テキストモードでもpg_dumpがラージオブジェクトをダンプできるようになりました。(Tom)

この変更により、ラージオブジェクトが常にダンプできるようになりました。 以前の-bスイッチは何も行いません。

- pg_dumpが一貫性を持ったラージオブジェクトのスナップショットをダンプできるようになりました。(Tom)
- ラージオブジェクトのコメントをダンプします。(Tom)
- pg_dump に--encoding を追加しました。(Magnus Hagander)

これにより、サーバの符号化方式と異なる符号化方式でデータベースをダンプできるようになりました。 これは、異なる符号化方式のマシンでダンプを転送する時有効です。

- 手続き言語のpg_pltemplateに依存します。(Tom)

手続き言語用の呼び出しハンドラがpg_catalogスキーマ内に存在する場合、pg_dumpはそのハンドラをダンプしません。 代わりに、ロード時に言語の生成パラメータを提供するpg_pltemplateカタログに依存した、単なるCREATE LANGUAGE nameを使用してその言語をダンプします。

E.119.3.13. libpqの変更

- パスワードファイルのファイル名を指定するPGPASSFILE環境変数を追加しました。(Andrew)
- lo_create()を追加しました。 これはlo_creat()と同じですが、ラージオブジェクトのOIDを指定することができます。(Tom)
- libpqがmalloc()失敗に関するエラーを一貫性を持ってクライアントアプリケーションに返すようにしました。(Neil)

E.119.3.14. ソースコードの変更

- 場所を変更したインストレーションに対して構築をサポートするようにpgxsを修正しました。
- Intelコンパイラを使用するItaniumプロセッサをサポートするスピロックを追加しました。(Vikram Kalsi)
- Windows用のKerberos 5サポートを追加しました。(Magnus)
- 中国語版FAQを追加しました。(laser@pgsqldb.com)
- OS/Xの機能名称変更に合わせて、RendezvousをBonjourに名前を変更しました。(Bruce)
- Darwinにおけるfsync_writethroughのサポートを追加しました。(Chris Campbell)
- サーバ、オプティマイザ、ロックシステム内の情報の渡し方を合理化しました。(Tom)
- pg_configがMSVCを使用してコンパイルできるようになりました。(Andrew)

これには、MSVCを使用した DBD::Pgの構築が必要です。

- Kerberos V4のサポートをやめました。(Magnus)
Kerberos 4 にはセキュリティ上の欠陥があり、保守もされていません。
- コードを整理しました。(EnterpriseDBを使用してカバレッジに関する静的解析を行いました。)
- postgresql.confのドキュメント上のデフォルトを、true/falseではなくon/offに変更しました。(Bruce)
- pg_configを改良し、より多くの構築時の値を報告できるようにしました。(Tom)
- Windowsにおいてスレッドセーフなlibpqを構築できるようにしました。(Dave Page)
- WindowsにおいてIPv6接続が利用できるようになりました。(Andrew)
- I/Oサブシステムの信頼性に関するサーバ管理用文書を追加しました。(Bruce)
- 秘匿可能な宣言をgist.hからgist_private.hに移しました。(Neil)

以前のリリースでは、gist.hには公開GiST API (GiSTインデックス実装者向け) とGiST自体を実装するために使用する

秘匿可能な宣言が混在していました。後者が`gist_private.h`という別のファイルに移りました。ほとんどのGiSTインデックス実装には影響がないはずです。

- GiSTメモリ管理を再作成しました。(Neil)

GiSTのメソッドが常に短命のメモリコンテキスト内で呼び出されるようになりました。そのため、`palloc()`で割り当てられたメモリは自動的に回収されます。これにより、GiSTインデックス実装では`pfree()`を使用して割り当てられたメモリを解放する必要がなくなりました。

E.119.3.15. Contribの変更

- `/contrib/pg_buffercache` contribモジュールを追加しました。(Mark Kirkwood)

これはデバッグや性能の調整を目的としたもので、バッファキャッシュの内容を表示します。

- 古くなりましたので`/contrib/array`が削除されました。(Tom)

- `/contrib/lo`モジュールを整理しました。(Tom)

- `/contrib/findoidjoins`を`/src/tools`に移動しました。(Tom)

- `/contrib/cube`から`<<`、`>>`、`&<`、`&>` 演算子を削除しました。

これらの演算子が有用ではないからです。

- `/contrib/btree_gist`を改良しました。(Janko Richter)

- `/contrib/pgbench`を改良しました。(Tomoaki Sato, Tatsuo)

固定のコマンド群だけでなく、ユーザが用意したSQLコマンドスクリプトで試験を行なう機能が追加されました。

- `/contrib/pgcrypto`を改良しました。(Marko Kreen)

- OpenPGP対称キーと公開キー暗号の実装。

RSAおよびElgamal公開キーアルゴリズムの両方をサポートします。

- 単体での構築。SHA256/384/512ハッシュ、Fortuna PRNGを含みます。

- OpenSSLの構築。3DESのサポート。OpenSSL < 0.9.7での内部AESの利用。

- `configure`の結果を使用した構築パラメータ (OpenSSL、zlib) の取得。

`Makefile`を編集する必要がまったくなくなりました。

- `libmhash`および`libmcrypt`サポートの削除。

E.120. リリース8.0.26

リリース日: 2010-10-04

このリリースは8.0.25に対し、各種の不具合を修正したものです。8.0メジャーリリースにおける新機能については[項E.146](#)を参照してください。

これはPostgreSQL 8.0.X系列の最終リリースとなるはずですが、ユーザにはより新しい系列のリリースへすぐに更新することを勧めます。

E.120.1. バージョン8.0.26への移行

8.0.Xからの移行ではダンプ/リストアは不要です。しかし8.0.22以前のバージョンからアップデートする場合は8.0.22のリリースノートを参照してください。

E.120.2. 変更点

- PL/PerlおよびPL/Tclにおいて、呼び出し元のSQLユーザIDごとに別のインタプリタを使用します。(Tom Lane)

この変更は、同じセッション内でその後に異なるSQLユーザIDにより実行されるPerlまたはTclコード（例えばSECURITY DEFINERの内部）を破壊することで発生する可能性があるというセキュリティ問題を防止するものです。ほとんどのスクリプト用言語は、対象とする関数で呼び出される標準関数や演算子の再定義など、これを行うことができる多くの方法を提供します。この変更がないと、PerlまたはTcl言語の使用権限を持つ任意のSQLユーザは基本的に、対象の関数の所有者が持つSQL権限でできることを何でもできるようになります。

この変更の欠点はPerlおよびTcl関数間の内部通信がより難しくなるという点です。逃げ道を提供するために、PL/PerlUおよびPL/TclU関数ではまだセッション当たり1つのインタプリタのみを使用します。これらのすべての関数は従来よりデータベーススーパーユーザの信頼レベルで実行されますので、これはセキュリティ問題とはみなされません。

信頼された実行を提供することを主張するサードパーティ製の手続き言語は似たようなセキュリティ問題を持つ可能性が大きくあります。セキュリティが非常に重要な目的のために依存している手続き言語があれば、その作者に連絡を取ることを勧めます。

この問題(CVE-2010-3433)を指摘いただいたTim Bunceに感謝します。

- pg_get_expr()の用途として意図されたシステムカタログ列の一つではない引数で呼び出されることを拒絶することでpg_get_expr()内のクラッシュする可能性を防止します。(Heikki Linnakangas, Tom Lane)
- "cannot handle unplanned sub-select"エラーを修正しました。(Tom Lane)

これは、副選択が他の副選択を含む式に展開される結合の別名参照を含む場合に発生しました。

- 返される行のすべてが実際に同一の行型でない、レコードの集合を返す関数を防御します。(Tom Lane)
- ロックファイル (postmaster.pidおよびソケット用のロックファイル) を書き出す時に、注意してその内容のfsyncします。(Tom Lane)

これがないと、postmasterの起動後しばらくしてマシンがクラッシュした場合に、ロックファイルの内容が破損することがあり得ました。これが起きると、このロックファイルを手作業で削除するまで、その後のpostmasterの起動試行が成功できなくなります。

- 過度に入れ子になった副トランザクションに対してXIDを割り当てる際の再帰を防止します。(Andres Freund, Robert Haas)

元々のコードでは、制限されたスタック領域であった場合にクラッシュすることがあり得ました。

- バックエンド起動の初期にゴミを生成することがあったlog_line_prefixの%iエスケープを修正しました。(Tom Lane)
- アーカイブ処理が有効な場合、ALTER TABLE ... SET TABLESPACEにおけるデータ破損の可能性を修正しました。(Jeff Davis)
- CREATE DATABASEとALTER DATABASE ... SET TABLESPACEを問い合わせキャンセルにより中断させることができま

す。(Guillaume Lelarge)

- PL/Pythonにおいて、`PyObject_AsVoidPtr`および`PyObject_FromVoidPtr`からのヌルポインタ結果を防止します。(Peter Eisentraut)
- 削除された列を含むテーブルに対する`contrib/dblink`の処理を改良しました。(Tom Lane)
- `contrib/dblink`において"duplicate connection name"エラー後の接続リークを修正しました。(Itagaki Takahiro)
- 62バイトより長い接続名に関する`contrib/dblink`の処理を正しく修正しました。(Itagaki Takahiro)
- 構築手順や文書を更新して、ソースコードリポジトリがCVSからGitに変更されたことを反映しました。(Magnus Hagander他)
- 時間帯データファイルをtzdata release 2010Iに更新しました。 エジプト、パレスチナにおける夏時間規則の変更、フィンランドの歴史的な修正が含まれています。

またこの変更では、2つのミクロネシアの時間帯を追加しています。 Pacific/ChuukはPacific/Trukより好まれるようになり（好まれる省略形はTRUTではなくCHUT）、Pacific/PohnpeiはPacific/Ponapeより好まれるようになります。

[前のページ](#)

リリース8.1

[ホーム](#)

[上に戻る](#)

[次のページ](#)

リリース8.0.25

E.121. リリース8.0.25

リリース日: 2010-05-17

このリリースは8.0.24に対し、各種の不具合を修正したものです。8.0メジャーリリースにおける新機能については[項E.146](#)を参照してください。

PostgreSQLコミュニティは2010年7月に8.0.X系のリリースの更新版の公開を停止します。ユーザにはより新しい系列のリリースへすぐに更新することを奨励します。

E.121.1. バージョン8.0.25への移行

8.0.Xからの移行ではダンプ/リストアは不要です。しかし8.0.22以前のバージョンからアップデートする場合は8.0.22のリリースノートを参照してください。

E.121.2. 変更点

- Safe.pmを使用するのではなく、インタプリタ全体に適用されるopmaskを使用してplperlを強制的に制限します。(Tim Bunce, Andrew Dunstan)

最近の開発によりplperlを信頼できるとする根拠とするにはSafe.pmが非常に安全ではないことを知らされました。この変更により、Safe.pmの使用がなくなり、常に適用されるopcodeマスクを持つ別のインタプリタを使用するようになります。この変更の喜ばしい副作用として、plperl内で普通の方法でPerlのstrict プラグマが使用できるようになりました。また、Perlの\$aと\$bがソート関数で想定される動作を行うようになり、関数のコンパイルが非常に高速になりました。(CVE-2010-1169)

- PL/Tclにおいてpltcl_modules経由で信頼できないコードが実行されることを防止しました。(Tom)

PL/TclのデータベーステーブルからTclコードを自動読み込みする機能はトロイの木馬攻撃によりつけこまれる可能性があります。そのテーブルの作成や挿入のできるユーザに対する制限がなかったためです。この変更により、pltcl_modulesがスーパーユーザにより所有されていない限り、この機能は無効になります。(しかし、このテーブルに対する権限は検査されませんので、安全性が低いモジュールを本当に必要とするインストレーションでは、信頼された非スーパーユーザに適切な権限を付与することができます。)また、pltclu関数を本当に実行しようとしていない限り、制限がない"通常"のTclにコードが読み込まれることを防止しました。(CVE-2010-1170)

- 非特権ユーザがスーパーユーザのみのパラメータ設定を再設定することを許しません。(Alvaro)

これまで、非特権ユーザが自身に対するALTER USER ... RESET ALLまたは自身が所有するデータベースに対するALTER DATABASE ... RESET ALLを行った場合、そのユーザまたはデータベースに対する特別なパラメータ設定は、たとえスーパーユーザのみにより変更可能なものとして提供されるものであっても、すべて削除されました。ALTERはそのユーザが変更権限を持つパラメータのみを削除するようになりました。

- CONTEXT追加がログ項目になされる時にシャットダウンが発生した場合に、バックエンドのシャットダウン時にクラッシュする可能性を防ぎました。(Tom)

現在のトランザクションがログメッセージを出力する時点に達した時にすでにロールバックされているために文脈出力関数が失敗することがありました。

- pl/perlのppport.hを最近のPerlバージョンに更新しました。(Andrew)
- pl/python内の雑多なメモリリークを修正しました。(Andreas Freund, Tom)
- 変数展開が自身を参照している際のpsqlにおける無限再帰を防止します。(Tom)
- contrib/pgstattuple関数がキャンセル割り込みに即座に応答することを確実にしました。(Tatsuhito Kasahara)
- サーバ起動時に、既存の共有メモリセグメントに対してshmget()がEINVALを返す状況を適切に扱うようにしました。(Tom)

この振舞いはOS Xを含むBSD派生のカーネルにおいて観測されました。共有メモリ要求サイズが大きすぎると警告の、完全に間違った起動失敗という結果になりました。

- 時間帯データファイルをtzdata release 2010jに更新しました。アルゼンチン、南極におけるオーストラリア、バンダ

ラデシュ、メキシコ、モロッコ、パキスタン、パレスチナ、ロシア、シリア、チュニジアにおける夏時間規則の変更、および、台湾の歴史的な修正が含まれます。

[前のページ](#)
リリース8.0.26

[ホーム](#)
[上に戻る](#)

[次のページ](#)
リリース8.0.24

E.122. リリース8.0.24

リリース日: 2010-03-15

このリリースは8.0.23に対し、各種の不具合を修正したものです。8.0メジャーリリースにおける新機能については[項E.146](#)を参照してください。

PostgreSQLコミュニティは2010年7月に8.0.X系のリリースの更新版の公開を停止します。ユーザにはより新しい系列のリリースへすぐに更新することを奨励します。

E.122.1. バージョン8.0.24への移行

8.0.Xからの移行ではダンプ/リストアは不要です。しかし8.0.22以前のバージョンからアップデートする場合は8.0.22のリリースノートを参照してください。

E.122.2. 変更点

- SSL接続においてセッションキーの再調停をどの程度の頻度で行うかを制御する`ssl_renegotiation_limit`設定パラメータを新しく追加しました。(Magnus)

これをゼロに設定することで、再調停を完全に無効にすることができます。これは壊れたSSLライブラリを使用している場合に必要になる可能性があります。具体的には一部のベンダは再調停の試行を失敗させるCVE-2009-3555に対する一時しのぎのパッチを提供しています。

- 副トランザクション起動時における失敗からのリカバリを試行する時にクラッシュする可能性を修正しました。(Tom)
- セーブポイントの使用およびサーバの符号化方式と異なるクライアント符号化方式に関連したサーバのメモリリークを修正しました。(Tom)
- `bit`型の`substring()`では、いかなる負の長さも"文字列の残りすべて"を意味するようにしました。(Tom)
これまでのコードでは-1のみを上のように扱い、他の負の値についてはクラッシュする可能性がある無効な結果値を生成しました。(CVE-2010-0442)
- 出力ビット幅が指定された整数よりも広く、それが8ビットの倍数以外である場合、先頭の端数のバイトを正確に扱うように整数からビット列への変換を修正しました。(Tom)
- 正規表現マッチが異常なまでに低速になる一部の状況を修正しました。(Tom)
- 終了位置が正確にセグメント境界であった場合、次のWALセグメント名を報告するようにバックアップ履歴ファイル内の`STOP WAL LOCATION`を修正しました。(Itagaki Takahiro)
- `pg_hba.conf`とその関連ファイルを読み取る時、`@`が引用符記号の内部にある場合に`@something`をファイル包含要求と扱わないようにしました。また、`@`自身をファイル包含要求として扱いません。(Tom)

ロールまたはデータベース名が`@`から始まる場合の一定しない動作を防止します。空白文字を含むパス名のファイルを含む必要がある場合、まだ実現させることはできますが、式全体を引用符でくくるのではなく`@"/path to/file"`と記述しなければなりません。

- ディレクトリが`pg_hba.conf`とその関連ファイル内で含有対象と指定されていた場合の一部のプラットフォームにおける無限ループを防止します。(Tom)
- 複合列がNULLに設定される場合に`plpgsql`が失敗することを修正しました。(Tom)
- コンパイラ固有の誤動作が起こる可能性を防ぐためにPL/Python内に`volatile`記号を追加しました。(Zdenek Kotala)
- PL/Tclは確実にTclインタプリタを完全に初期化します。(Tom)

このミスの既知の兆候は、Tcl 8.5以降を使用している場合のTclの`clock`コマンドの誤動作だけです。

- `dblink_build_sql_*`関数であまりに多くのキー列が指定された場合の`contrib/dblink`のクラッシュを防止します。(Rushabh Lathia, Joe Conway)

ずさんなメモリ管理によって引き起こるcontrib/xml2の雑多なクラッシュを修正しました。(Tom)

- 時間帯データファイルをtzdata release 2010eに更新しました。 バングラデシュ、チリ、フィジー、メキシコ、パラグアイ、サモアにおける夏時間規則の変更が含まれます。

[前のページ](#)

リリース8.0.25

[ホーム](#)
[上に戻る](#)

[次のページ](#)

リリース8.0.23

E.123. リリース8.0.23

リリース日: 2009-12-14

このリリースは8.0.22に対し、各種の不具合を修正したものです。 8.0メジャーリリースにおける新機能については[項E.146](#)を参照してください。

E.123.1. バージョン8.0.23への移行

8.0.Xからの移行ではダンプ/リストアは不要です。 しかし8.0.22以前のバージョンからアップデートする場合は8.0.22のリリースノートを参照してください。

E.123.2. 変更点

- セッション固有の状態を変更するインデックス関数により引き起こされる間接的なセキュリティ脅威から保護します。(Gurjeet Singh, Tom)

この変更により不変インデックス関数がスーパーユーザのセッションを破壊する可能性(CVE-2009-4136)を防ぎます。

- ヌルバイトが埋め込まれたコモンネーム (CN) フィールドをもつSSL証明書を拒絶します。(Magnus)

これは、SSL検証においてサーバまたはクライアント名に対する証明書が不正に一致しないようにします。(CVE-2009-4034)

- バックエンド起動時のキャッシュ初期化の間でクラッシュする可能性を修正しました。(Tom)

- 安全ではない時点でのVACUUMを中断するシグナルを防止します。(Alvaro)

この修正により、VACUUM FULLがすでにタプルの移動をコミットした後にキャンセルされた場合のパニックや、通常のVACUUMがテーブルを切り詰めた後に割り込みを受けた場合の一時的なエラーを防止します。

- ハッシュテーブルサイズの計算における整数オーバーフローによるクラッシュの可能性を修正しました。(Tom)

これは、ハッシュ結合の結果のプランナの推定サイズが極端に大きい場合に発生することがありました。

- inet/cidrの比較における非常にまれなクラッシュを修正しました。(Chris Mikkelsen)

- 副トランザクション内でアクセスされるカーソルで使用する一時ファイルの早すぎる削除を修正しました。(Heikki)

- PAMパスワード処理をより堅牢に修正しました。(Tom)

以前のコードでは、Linuxのpam_krb5 PAMモジュールとドメインコントローラとしてのMicrosoft Active Directoryを組み合わせると失敗することが知られていました。 PAMスタックに渡される引数に関する正当ではない仮定を作成することであるため、どこかで問題になるかもしれません。

- PL/Pythonの例外処理におけるまれなクラッシュを修正しました。(Peter)

- psqlのflexモジュールが正しいシステムヘッダファイルを使用してコンパイルされることを確実にしました。(Tom)

これは、--enable-largefileが生成されたコード内に非互換の変更を引き起こすプラットフォームにおける構築失敗を修正します。

- 今後のバージョンのlibpqとの互換性を高めるために、postmasterが接続要求パケットにおけるapplication_nameパラメータをすべて無視するようにしました。(Tom)

- 時間帯データファイルをtzdata release 2009sまで更新しました。 南極、アルゼンチン、バングラデシュ、フィジー、ノヴォクズネツク、パキスタン、パレスチナ、サモア、シリアにおける夏時間規則の変更、および香港に関する歴史的な修正が含まれます。

E.125. リリース8.0.21

リリース日: 2009-03-16

このリリースは8.0.20に対し、各種の不具合を修正したものです。 8.0メジャーリリースにおける新機能については[項E.146](#)を参照してください。

E.125.1. バージョン8.0.21への移行

8.0.Xからの移行ではダンプ/リストアは不要です。 しかし、8.0.6より前のバージョンからアップグレードする場合は、8.0.6のリリースノートを参照してください。

E.125.2. 変更点

- 符号化方式の変換に失敗した時、再帰的なエラーによるクラッシュを防止しました。(Tom)

この変更は、最近の2つのマイナーリリースにて関連する失敗の状況に対してなされた改修を拡張したものです。 前回の修正は元の問題報告に特化したものでしたが、符号化方式変換関数で発生するすべてのエラーがそのエラーを報告しようとして、無限に再帰される可能性があることが分かりました。 したがって、再帰的なエラー報告を行う状況になったことがわかった時の解決策は、変換と符号化方式を無効にし、通常のASCII形式のエラーメッセージで報告することです。(CVE-2009-0922)

- 特定の変換関数に対する、間違った符号化方式を用いた`CREATE CONVERSION`を許可しません。(Heikki)

これにより、符号化方式に関する失敗における、あり得る状況を防止します。 前回の変更は、同じ問題における別の種類の失敗に対する防止策でした。

- `to_char()` に引数のデータに対して不適切な整形用のコードが渡された時のコアダンプを修正しました。(Tom)
- デフォルトの既知の時間帯省略形のリストに`MUST` (モーリシャス島夏時間) を追加しました。(Xavier Bugaud)

E.126. リリース8.0.20

リリース日: 2009-02-02

このリリースは8.0.19に対し、各種の不具合を修正したものです。 8.0メジャーリリースにおける新機能については[項E.146](#)を参照してください。

E.126.1. バージョン8.0.20への移行

8.0.Xからの移行ではダンプ/リストアは不要です。 しかし、8.0.6より前のバージョンからアップグレードする場合は、8.0.6のリリースノートを参照してください。

E.126.2. 変更点

- `headline()` 関数におけるURLの扱いを改良しました。(Teodor)
- `headline()` 関数における延長見出しの扱いを改良しました。(Teodor)
- 符号化方式の変換が符号化方式の組み合わせの指定を間違えた変換関数で作成された場合にあり得る、アサートの失敗や誤変換を防止します。(Tom, Heikki)
- `VACUUM`において小規模なテーブルに対する不要なロックを防止しました。(Heikki)
- `contrib/tsearch2`の`get_covers()` 関数における未初期化変数を修正しました。(Teodor)
- すべての文書において、現在活動していない`pgsql-ports`と`pgsql-patches`メーリングリストの代わりに、`pgsql-bugs`または`pgsql-hackers`を参照するようにしました。(Tom)
- 時間帯データファイルをtzdataリリース2009a（カトマンズ、およびスイス、キューバにおける歴史的なDSTの訂正）に更新しました。

E.127. リリース8.0.19

リリース日: 2008-11-03

このリリースは8.0.18に対し、各種の不具合を修正したものです。 8.0メジャーリリースにおける新機能については[項E.146](#)を参照してください。

E.127.1. バージョン8.0.19への移行

8.0.Xからの移行ではダンプ/リストアは不要です。 しかし、8.0.6より前のバージョンからアップグレードする場合は、8.0.6のリリースノートを参照してください。

E.127.2. 変更点

- クライアントの符号化方式が翻訳されたエラーメッセージを表すことができない時のバックエンドのクラッシュを修正しました。(Tom)
以前にも同様の問題に対応しましたが、"等価な文字を持たない"メッセージ自体が変換されない場合にまだ失敗しました。 今回の修正では、こうした状況を検知した時、翻訳を無効にし、通常のASCIIエラーメッセージを送信します。
- トリガから深い入れ子状態の関数が呼び出された時にクラッシュする可能性を修正しました。(Tom)
- 新しく定義されたPL/pgSQLトリガ関数が通常関数として呼び出された場合に報告されるエラーを確実なものにしました。(Tom)
- 1つの問い合わせ項目がテキストの先頭の単語に一致した場合、テキスト検索の間違った見出し生成を修正しました。(Sushant Sinha)
- `--enable-integer-datetimes`で構築された場合、ISO以外の日付書式を使用した時の時間間隔値における不適切な秒端数表示を修正しました。(Ron Mayer)
- 渡されたタプルとタプル記述子が異なる列数を持っていた場合にSPI_getvalueおよびSPI_getbinvalが適切に動作することを確実にしました。(Tom)
テーブルに列が追加または削除された場合、こうした状況は正常です。 しかしこの2つの関数は適切に扱っていませんでした。 起こり得る唯一の結果は間違ったエラーが示されることです。
- ecpgのCREATE USER解析を修正しました。(Michael)
- pg_ctl restartの最近の破損を修正しました。(Tom)
- 時間帯データファイルをtzdataリリース2008i（アルゼンチン、ブラジル、モーリシャス、シリアにおける夏時間規則の変更）まで更新しました。

E.128. リリース8.0.18

リリース日: 2008-09-22

このリリースは8.0.17に対し、各種の不具合を修正したものです。 8.0メジャーリリースにおける新機能については[項E.146](#)を参照してください。

E.128.1. バージョン8.0.18への移行

8.0.Xからの移行ではダンプ/リストアは不要です。 しかし、8.0.6より前のバージョンからアップグレードする場合は、8.0.6のリリースノートを参照してください。

E.128.2. 変更点

- 局所ロックのカウンタを32ビットから64ビットに広げました。(Tom)

これは、かなり長いトランザクションでカウンタがオーバーフローし、想定外の"すでに保持済のロックである"というエラーになるという報告に対応したものです。

- `INSERT` または `UPDATE` で生成されるタプルが対象のテーブルの現在の行型に一致するか確認するために、エグゼキュータ起動の際の検査を追加しました。(Tom)

`ALTER COLUMN TYPE` の後にこれまでキャッシュしていた計画を再利用した場合、こうした状況が起こります。 データ破壊やその結果発生するクラッシュをこの検査により防ぎます。

- 64ビットプラットフォームで稼働している場合、日付時刻入力関数が正しく整数オーバーフローを検知できるよう修正しました。(Tom)
- 非常に長いログメッセージをsyslogに書き出す際の性能を向上しました。(Tom)
- `SELECT DISTINCT ON` 問い合わせに対するカーソルの後方スキャンにおける不具合を修正しました。(Tom)
- プランナが論理型の結果を生成する `GROUP BY` 式を、式の内容に関係なく、常に2つのグループになると推定することを修正しました。(Tom)

これにより、`col IS NULL` などの特定の論理試験に対して、通常の `GROUP BY` 推定より非常により良く精度が上がります。

- Tcl 8.5で正しく動作するようPL/Tclを修正しました。 また、Tcl間でやりとりされるデータの符号化方式についてより注意を払うように修正しました。(Tom)
- Python 2.5で動作するようPL/Pythonを修正しました。

これは、8.2の開発期間で行われた修正を後方に反映したものです。

- `pg_dump` および `pg_restore` がSQLコマンドの送信に失敗した後に報告するエラーを改良しました。(Tom)
- `pg_ctl` が `restart` の間正しく `postmaster` のコマンドライン引数を保持するように修正しました。(Bruce)
- 時間帯データファイルをtzdataリリース2008f（アルゼンチン、バハマ、ブラジル、モーリシャス、モロッコ、パキスタン、パレスチナ、パラグアイにおける夏時間規則の変更）まで更新しました。

E.129. リリース8.0.17

リリース日: 2008-06-12

このリリースは8.0.16に対し、重大な1つの不具合と軽微な1つの不具合を修正したものです。 8.0メジャーリリースにおける新機能については[項E.146](#)を参照してください。

E.129.1. バージョン8.0.17への移行

8.0.Xからの移行ではダンプ/リストアは不要です。 しかし、8.0.6より前のバージョンからアップグレードする場合は、8.0.6のリリースノートを参照してください。

E.129.2. 変更点

- `pg_get_ruledef()` で負の定数を括弧でくるようにしました。(Tom)

この修正前では、ビューまたはルールにおける負の定数がそのまま、例えば`-42::integer`のようにダンプされる可能性があります。 これは若干正しくありません。 演算子の前置規則に従うと、`(-42)::integer`となるべきです。 通常は大した違いが起こることはありませんが、最近の別のパッチと相互作用して、PostgreSQLが有効な`SELECT DISTINCT`ビューの問い合わせを拒絶することになりました。 `pg_dump`の出力のリロード失敗という結果になることもあり得たため、これは高優先度の修正として扱われました。 実際にダンプ出力が正しくなかったバージョンは8.3.1と8.2.7だけです。

E.130. リリース8.0.16

リリース日: リリースされませんでした

このリリースには8.0.15からの各種修正が含まれています。 8.0メジャーリリースにおける新機能については[項E.146](#)を参照してください。

E.130.1. バージョン8.0.16への移行

8.0.Xからの移行ではダンプ/リストアは不要です。 しかし、8.0.6より前のバージョンからアップグレードする場合は、8.0.6のリリースノートを参照してください。

E.130.2. 変更点

- 新しい列がすべて非NULLに初期化されていることが正しく検査されるようにALTER TABLE ADD COLUMN ... PRIMARY KEYを修正しました。(Brendan Jurd)

これまでのバージョンでは、この必要条件の検査をまったく行っていませんでした。

- "同じ"制約を、共通の祖先からその制約を継承する複数の親リレーションから継承する場合に、CREATE TABLEが失敗する可能性を修正しました。(Tom)
- キリル語の"Yo"文字 (2つのドットを持つёとЕ) を扱うISO-8859-5と他の符号化方式との間の変換を修正しました。(Sergey Burladyan)
- 複数のデータ型入力関数、特に、結果に初期化されず予測できない値を含む未使用のバイトを許すarray_in()を修正しました。(Tom)

これは、2つの明らかに同じリテラル値が等しいとみなされず、結果パーサは一致しないORDER BYおよびDISTINCT式に関して不平をいうといったエラーをもたらします。

- 正規表現副文字列一致 (substring(string from pattern)) における境界状態を修正しました。(Tom)

この問題は、パターン全体は一致するが、ユーザが括弧付きの副式を指定し、その副式が一致しなかった時に発生します。例えば、substring('foo' from 'foo(bar)?') です。(bar) は一致しませんので、これはNULLを返すべきです。しかし、間違っパターン全体が一致 (つまりfoo) という結果を返していました。

- 時間帯データファイルをtzdataリリース2008c (モナコ、イラク、チョイバルサン、パキスタン、シリア、キューバ、アルゼンチン/サンルイスにおける昼時間規則の変更) まで更新しました。
- ecpgのPGTYPEtimestamp_sub() 関数からの不正確な結果を修正しました。(Michael)
- 入力問い合わせがNULL値を返す場合のcontrib/xml2のxpath_table() 関数のコアダンプを修正しました。(Tom)
- contrib/xml2のmakefileがCFLAGSを上書きしないように修正しました。(Tom)
- DatumGetBool マクロがgcc 4.3でエラーにならないよう修正しました。(Tom)

この問題は、論理値を返す"古い形式" (V0)のC関数に影響します。これは8.3ではすでに修正されていましたが、当時なかったバックパッチが必要になりました。

- 長い間存在したLISTEN/NOTIFYの競合条件を修正しました。(Tom)

まれな状況において、NOTIFYを実行する同時実行のトランザクションが後にコミットされたことが観測されたために想定されたとしても、LISTENだけを実行するセッションが注意を受け取らない可能性があります。

この修正の副作用として、LISTENコマンドをまだコミットしていないトランザクションではLISTEN用のpg_listener内の行を参照しなくなります。これは検索するために選択すべきもので、これまでではできていました。この動作はできるともできないとも文書化されていませんが、一部のアプリケーションで古い動作に依存している可能性があります。

- ハッシュインデックスを使用する問い合わせ中でエラーが発生した場合の、稀なクラッシュを修正しました。(Heikki)
- 紀元前の2月29日に関する日付時刻値の入力を修正しました。(Tom)

これまでのコードにはうるう年の決定に間違いがありました。

- `ALTER OWNER`の一部の構文における"unrecognized node type"エラーを修正しました。(Tom)
- コマンドラインオプションからpostmasterのポート番号を正しく取り出せるようpg_ctlを修正しました。(Itagaki Takahiro, Tom)

これまでのpg_ctl start -wでは間違ったポート上のpostmasterに接続しようとし、起動失敗に関するおかしい報告を引き起こしました。

- 最近のバージョンのgccにおける間違った最適化の可能性を抑えるために-fwrapvを使用します。(Tom)

これはPostgreSQLをgcc 4.3以降で構築する時に必要であることは分かっていました。

- ORDER BYおよびGROUP BY内の定数式の表示を修正しました。(Tom)

明示的にキャストされた定数が不正確に表示されました。これは例えば、ダンプ/リストア時にビュー定義を壊す可能性があります。

- COPY OUT時にNOTICEメッセージを正しく扱うようlibpqを修正しました。(Tom)

このエラーは、ユーザ定義のデータ型出力ルーチンがNOTICEを発行する時に起こることのみが観測されていました。しかし、他が原因で発生していないという保証はありません。

[前のページ](#)

リリース8.0.17

[ホーム](#)
[上に戻る](#)

[次のページ](#)

リリース8.0.15

E.131. リリース8.0.15

リリース日: 2008-01-07

このリリースは、重大なセキュリティ問題の修正を含む、8.0.14の各種不具合を修正したものです。8.0メジャーリリースにおける新機能については[項E.146](#)を参照してください。

これが、PostgreSQLコミュニティがWindows用のバイナリパッケージを作成する、最後の8.0.Xリリースです。8.2.Xでは、過去のバージョンに移植することが現実的ではない、Windows固有の修正が行われていますので、Windowsユーザは8.2.X以降への移行が推奨されています。他のプラットフォームでは8.0.Xはまだサポートされます。

E.131.1. バージョン8.0.15への移行

8.0.Xからの移行ではダンプ/リストアは不要です。しかし、8.0.6より前のバージョンからアップグレードする場合は、8.0.6のリリースノートを参照してください。

E.131.2. 変更点

- インデックス内の関数がVACUUM、ANALYZEなどを実行するユーザの権限で実行されることを防止します。(Tom)

インデックス式内で使用される関数および部分インデックスの述語は、新しいテーブル項目が作成されると評価されます。これが、もし信頼できないユーザが所有するテーブルを変更したとすると、トロイの木馬となるコードの実行という危険性を引き起こすことは以前から解っていました。(トリガ、デフォルト、検査制約なども同種の危険性を引き起こします。)しかしインデックス内の関数は、VACUUM FULLなどの日常の保守作業で実行される可能性があるため、さらに危険です。こうした保守作業は通常自動的にスーパーユーザ権限で行われます。例えば、極悪なユーザがトロイの木馬となるインデックス定義を設定し、次の日常のバキューム処理を待つことで、スーパーユーザ権限でコードを実行することが可能です。この修正は、SECURITY DEFINER関数で使用されていた権限変更機構を使用して、標準的な保守作業(VACUUM、ANALYZE、REINDEX、CLUSTERを含む)が呼び出したユーザではなくテーブル所有者として実行するように調整しました。こうしたセキュリティ制限の迂回を防ぐために、SET SESSION AUTHORIZATIONおよびSET ROLEがSECURITY DEFINER コンテキストで禁止されるようになりました。(CVE-2007-6600)

- 正規表現パッケージ内のさまざまな不具合を修正しました。(Tom, Will Drewry)

より巧妙な正規表現パターンにより、信頼できないソースから正規表現検索パターンを受け付けるアプリケーションでは、クラッシュ、無限または無限に近いループ、大量のメモリの浪費、DoS攻撃を引き起こすすべてが発生する可能性があります。(CVE-2007-4769、CVE-2007-4772、CVE-2007-6067)

- /contrib/dblinkを使用して、パスワード認証のみをセキュリティ制限として使用するには非スーパーユーザでなければなりません。

8.0.14で行った修正は不十分で、一部のdblink関数に対してのみ問題を解消していました。(CVE-2007-6601、CVE-2007-3278)

- 時間帯ファイルをtzdataリリース2007kに更新しました(具体的には最近のアルゼンチンの変更)。(Tom)

- 一部のWHERE false AND var IN (SELECT ...)におけるプランナの失敗を修正しました。(Tom)

- ALTER TABLE ... ALTER COLUMN TYPEで再構築されるインデックスのテーブル空間と格納パラメータを保持します。(Tom)

- アーカイブリカバリが、リカバリの停止時間が使用された時だけではなく、常に新しいWAL時系列から始まるようにしました。(Simon)

これは、まれに起こる可能性のある既存の最後のWALセグメントのアーカイブされたコピーを上書きしようとする危険を防止します。また元の定義より単純、明確になったと考えられます。

- テーブルが小さすぎて有用でない場合、VACUUMがすべてのmaintenance_work_memを使用しないようにしました。(Alvaro)
- マルチバイトデータベース符号化方式を使用している時に潜在する、translate()のクラッシュを修正しました。(Tom)
- プラットフォームのPerlがbool型をcharではなくintとして定義した場合に対応するようにPL/Perlを修正しました。(Tom)

これは理論上どこでも発生する可能性がありましたが、標準的に構築されたPerlではこうなりませんでした。Mac OS X 10.5までは...

- 長い例外メッセージでクラッシュしないようPL/Pythonを修正しました。(Alvaro)
- 親テーブルと異なるデフォルト式を持つ、継承子テーブルを正しく扱うようpg_dumpを修正しました。(Tom)
- ecpgパーサを修正しました。(Michael)
- contrib/tablefuncのcrosstab()が、クラッシュするのではなく、独自の意味でカテゴリとしてNULL行識別子を扱うようにしました。(Joe)
- tsvectorおよびtsquery出力関数が正しくバックスラッシュをエスケープするように修正しました。(Teodor, Bruce)
- 入力文字列が大きな場合にto_tsvector()がクラッシュすることを修正しました。(Teodor)
- configureスクリプトを再生成する場合、特定のバージョンのAutoconfを使用することが必要になりました。(Peter)

これは開発者とパッケージ作成者のみに影響します。この変更は、AutoconfとPostgreSQLの未確認のバージョンの組み合わせを使用する事故を防ぐためになされました。その結果がうまくいくかどうかについて責任を負う限り、異なるバージョンのAutoconfを本当に使用したい場合は、このバージョン検査を取り除くことができます。

[前のページ](#)

リリース8.0.16

[ホーム](#)
[上に戻る](#)

[次のページ](#)

リリース8.0.14

E.132. リリース8.0.14

リリース日: 2007-09-17

このリリースは8.0.13の各種不具合を修正したものです。8.0メジャーリリースにおける新機能については[項E.146](#)を参照してください。

E.132.1. バージョン8.0.14への移行

8.0.Xからの移行ではダンプ/リストアは不要です。しかし、8.0.6より前のバージョンからアップグレードする場合は、8.0.6のリリースノートを参照してください。

E.132.2. 変更点

- トランザクションが、行を挿入した後に、同一テーブルに対する同時実行中のVACUUMの終了時刻とほぼ同じ時点でアポートした場合のインデックス破損を防止します。(Tom)
- CREATE DOMAIN ... DEFAULT NULLが適切に動作するようにしました。(Tom)
- 過度なSSLエラーメッセージのログ処理を修正しました。(Tom)
- sysloggerプロセスを使用する時にログメッセージが差し込まれないようにログ処理を修正しました。(Andrew)
- log_min_error_statementログ処理がメモリ不足になった時のクラッシュを修正しました。(Tom)
- 一部の外部キーの境界の不正な扱いを修正しました。(Tom)
- 他のセッションによる一時テーブル処理を防ぐために起こるCLUSTERの失敗を防ぎます。(Alvaro)
- 時間帯データベース規則を更新しました。具体的にはニュージーランドで予定している変更です。(Tom)
- Windowsソケットを改良しました。(Magnus)
- 符号化方式が不整合となる可能性があるため、Windowsにてログのタイムスタンプに時間帯名(%Z)を出力しないようにしました。(Tom)
- セキュリティ強化のため、パスワード認証のみを使用する/contrib/dblinkを使用するユーザが非特権ユーザであることを要求します。(Joe)

E.133. リリース8.0.13

リリース日: 2007-04-23

このリリースは8.0.12の各種不具合を修正したもので、セキュリティ問題の修正を1つ含みます。 8.0メジャーリリースにおける新機能については[項E.146](#)を参照してください。

E.133.1. バージョン8.0.13への移行

8.0.Xからの移行ではダンプ/リストアは不要です。 しかし、8.0.6より前のバージョンからアップグレードする場合は、8.0.6のリリースノートを参照してください。

E.133.2. 変更点

- `search_path`における一時テーブルスキーマの明示的な指定をサポートしました。 また、関数や演算子ではその検索を無効にしました(Tom)。

これは、`SECURITY DEFINER`関数が`search_path`の値を本当に安全に設定するために必要です。 こうしないと、一時オブジェクトを使用して権限のないSQLユーザが`SECURITY DEFINER`関数の権限でコードを実行することができてしまうためです (CVE-2007-2138)。 詳細は`CREATE FUNCTION`を参照してください。

- `/contrib/tsearch2`のクラッシュを修正しました(Teodor)。
- `VACUUM FULL`の`UPDATE`連鎖方法において潜在したデータ破損不具合を修正しました(Tom, Pavan Deolasee)。
- ハッシュインデックスの拡張時のパニックを修正しました(8.0.10で入ってしまった不具合です)(Tom)。
- POSIX書式の時間帯指定が新しいUSA DST規則に従うよう修正しました(Tom)。

E.134. リリース 8.0.12

リリース日: 2007-02-07

このリリースには、8.0.11に対する1つの修正が含まれています。 8.0メジャーリリースにおける新機能については[項E.146](#)を参照してください。

E.134.1. バージョン8.0.12への移行

8.0.Xからの移行ではダンプ/リストアは不要です。 しかし、8.0.6より前のバージョンからアップグレードする場合は、8.0.6のリリースノートを参照してください。

E.134.2. 変更点

- 制約および関数インデックスにおける型の長さに対する重複/限定検査を削除しました。(Tom)

E.135. リリース 8.0.11

リリース日: 2007-02-05

このリリースは8.0.10の各種不具合を修正したもので、セキュリティ問題の修正も含まれます。 8.0メジャーリリースにおける新機能については[項E.146](#)を参照してください。

E.135.1. バージョン8.0.11への移行

8.0.Xからの移行ではダンプ/リストアは不要です。しかし、8.0.6より前のバージョンからアップグレードする場合は、8.0.6のリリースノートを参照してください。

E.135.2. 変更点

- 接続ユーザがバックエンドのメモリを読み取ることができるセキュリティ脆弱性を取り除きました。(Tom)

この脆弱性には、SQL関数が宣言されたデータ型を返すかどうか、そのテーブル列のデータ型が変更されたかどうかに関する通常の検査の抑制も含まれています (CVE-2007-0555、CVE-2007-0556)。これらのエラーを悪用して簡単にバックエンドをクラッシュさせることができます。また原理的には、アクセスを許していないはずのユーザがデータベースの内容を読み取ることができてしまいます。

- 実行不可能な分割点の選択によりB-Treeインデックスページの分割が失敗する可能性がある、稀に起こる不具合を修正しました。(Heikki Linnakangas)
- UNION をきっかけとした、稀に発生するAssert()クラッシュを修正しました。(Tom)
- 3バイト長を越えるUTF8シーケンスに関する、複数バイト文字処理のセキュリティを強化しました。(Tom)

E.136. リリース 8.0.10

リリース日: 2007-01-08

このリリースは8.0.9の各種不具合を修正したものです。 8.0メジャーリリースにおける新機能については[項E.146](#)を参照してください。

E.136.1. バージョン8.0.10への移行

8.0.Xからの移行ではダンプ/リストアは不要です。 しかし、8.0.6より前のバージョンからアップグレードする場合は、8.0.6のリリースノートを参照してください。

E.136.2. 変更点

- AIXにおける`getaddrinfo()`の扱いを改良しました。(Tom)
これは、特に統計情報コレクタの起動に関する問題を修正するものです。
- `VACUUM`における"親キーの再検索に失敗する"エラーを修正しました。(Tom)
- `VACUUM`によるギガバイト境界を跨る大規模なリレーシヨンの切り詰めにおける競合状態を修正しました。(Tom)
- 数ギガバイトのハッシュインデックスに影響する不具合を修正しました。(Tom)
- Windowsのシグナル処理におけるデッドロックの可能性を修正しました。(Teodor)
- 複数の空要素からなる`ARRAY[]`生成時の不具合を修正しました。(Tom)
- ECPGにおける接続時のメモリリークを修正しました。(Michael)
- 新しくinitdbを行ったインストレーションでは、`to_number()`と`to_char(numeric)`が`IMMUTABLE`ではなく`STABLE`になりました。(Tom)
これは、`lc_numeric`がこれらの関数の出力を変更する可能性があるためです。
- 括弧を使用した正規表現を使用するインデックスを改良しました。(Tom)
この改良により、`psql \d`の性能も向上しました。
- 時間帯データベースを更新しました。
具体的にはこれは、オーストラリアおよびカナダの夏時間規則に影響します。

E.137. リリース8.0.9

リリース日: 2006-10-16

このリリースは8.0.8の各種不具合を修正したものです。 8.0メジャーリリースにおける新機能については[項E.146](#)を参照してください。

E.137.1. バージョン8.0.8への移行

8.0.Xからの移行ではダンプ/リストアは不要です。 しかし、8.0.6より前のバージョンからアップグレードする場合は、8.0.6のリリースノートを参照してください。

E.137.2. 変更点

- ルールのWHERE式でNEW行の値を参照する時のクラッシュを修正しました。(Tom)
- 型指定がないリテラルがANYARRAYと解釈される場合のコアダンプを修正しました。
- 複数行を返すSQL関数を含む問い合わせにおけるAFTERトリガの間違った扱いを修正しました。(Tom)
- `ALTER TABLE ... TYPE`が`USING` 句に対する`NOT NULL`を再検査するように修正しました。(Tom)
- 区切り文字と重なって一致する文字を扱うよう`string_to_array()`を修正しました。

例えば`string_to_array('123xx456xxx789', 'xx')`です。

- `psql`の`\d`コマンドに対するパターンマッチにおけるちょっとした部分を修正しました。
- `/contrib/ltree`におけるインデックス破損不具合を修正しました。(Teodor)
- `ecpg`をより強固にするためいくつかの修正をしました。
- `/contrib/dbmirror`におけるバックスラッシュエスケープ処理を修正しました。
- Win32における統計情報収集の不安定性を修正しました。(Tom, Andrew)
- AIXおよびIntelコンパイラに対して修正を行いました。(Tom)

E.138. リリース8.0.8

リリース日: 2006-05-23

このリリースは8.0.7の各種不具合を修正したもので、非常に深刻なセキュリティ問題を解消するパッチを含みます。 8.0メジャーリリースにおける新機能については[項E.146](#)を参照してください。

E.138.1. バージョン8.0.8への移行

8.0.Xからの移行ではダンプ/リストアは不要です。しかし、8.0.6より前のバージョンからアップグレードする場合は、8.0.6のリリースノートを参照してください。

CVE-2006-2313およびCVE-2006-2314に示されたSQLインジェクション攻撃を完全に防ぐためには、アプリケーション側のコードの変更が必要となる場合があります。SQLコマンド内に信頼できない文字列を埋め込むアプリケーションでは、できる限り早く、その文字列が推奨するエスケープ技法を使用していることを確実に検証しなければなりません。ほとんどの場合、アプリケーションは文字列のエスケープ処理に、その場しのぎのコードではなく、ライブラリやドライバが提供する (libpqのPQescapeStringConn()) のような関数を使用しなければなりません。

E.138.2. 変更点

- すべての場合において、符号化が無効なマルチバイト文字を拒絶するようにサーバを変更しました。(Tatsuo, Tom)

PostgreSQLは少し前からこのような方向に移行していましたが、この検査がすべての符号化方式とすべてのテキスト入力に対して統一的に適用されるようになりました。さらに、単なる警告ではなく常にエラーとなるようになりました。この変更はCVE-2006-2313で示されるような種類のSQLインジェクション攻撃から保護します。

- 文字列リテラル内の`¥`の安全ではない使用を拒絶します。

CVE-2006-2314で示されるような種類のSQLインジェクション攻撃からのサーバ側の保護として、サーバは、SQL文字列リテラル内のASCII単一引用符の表現として、`'`のみを受け付け、`¥`を受け付けなくなりました。デフォルトでは、SQLインジェクションが可能となる状況である、`client_encoding`がクライアント側のみの符号化方式 (SJIS、BIG5、GBK、GB18030、UHC) に設定された場合にのみ`¥`は拒絶されます。新しい設定パラメータ`backslash_quote`により、必要な場合にこの動作を調整できます。CVE-2006-2314に対して完全に保護するには、クライアント側を変更する必要があるかもしれないことに注意してください。安全ではないクライアントを安全ではないものとして明らかにすることが、`backslash_quote`の目的の一つです。

- libpqの文字列エスケープルーチンを、符号化方式と`standard_conforming_strings`を考慮するように変更しました。

これは、CVE-2006-2313およびCVE-2006-2314で示されるセキュリティ問題に対し、libpqを使用したアプリケーションを修正します。また、将来予定されるSQL標準文字列リテラル構文への移行に対しても保護しています。同時に複数のPostgreSQL接続を使用するアプリケーションは、各データベース接続において使用される設定に合わせてエスケープ処理が正しく行われるように、`PQescapeStringConn()`と`PQescapeByteaConn()`に移行しなければなりません。"独自に"文字列エスケープ処理を行うアプリケーションはライブラリルーチンを使用するように変更しなければなりません。

- 一部の不正な符号化方式変換関数を修正しました。

`win1251_to_iso`、`win866_to_iso`、`euc_tw_to_big5`、`euc_tw_to_mic`、`mic_to_euc_tw`はすべて可変拡張に関して正しくありませんでした。

- 文字列における偶然残る`¥`の使用を整理しました。(Bruce, Jan)
- インデックススキャンの論理和が本来返すべき行を失う場合がある不具合を修正しました。
- Btreeインデックスが消去された場合のWAL再生を修正しました。
- `|`を含むパターンに対する`SIMILAR TO`を修正しました。(Tom)
- `SELECT INTO`および`CREATE TABLE AS`が基本ディレクトリではなくデフォルトのテーブル空間にテーブルを作成するように変更しました。(Kris Jurka)
- 独自のDH SSLパラメータを正しく使用するようにサーバを修正しました。(Michael Fuhr)
- Intel MacでのBonjourを修正しました。(Ashley Clark)

- さまざまな小規模のメモリーリークを修正しました。
- 一部のWin32システムにおけるパスワードプロンプトに関する問題を修正しました。(Robert Kinberg)

[前のページ](#)
リリース8.0.9

[ホーム](#)
[上に戻る](#)

[次のページ](#)
リリース8.0.7

E.139. リリース8.0.7

リリース日: 2006-02-14

このリリースは8.0.6の各種不具合を修正したものです。 8.0メジャーリリースにおける新機能については[項E.146](#)を参照してください。

E.139.1. バージョン8.0.7への移行

8.0.Xからの移行ではダンプ/リストアは不要です。 しかし、8.0.6より前のバージョンからアップグレードする場合は、8.0.6のリリースノートを参照してください。

E.139.2. 変更点

- `SET SESSION AUTHORIZATION`によるクラッシュの可能性を修正しました。(CVE-2006-0553)

アサートを有効にしてコンパイルした場合（これはデフォルトではありません）、非特権ユーザがサーバをクラッシュでき、その結果、他のユーザへのサービスが一時的に不能になります。 この問題を報告してくれたAkio Ishidaに感謝します。

- 自身が挿入した行に関する行の可視性ロジックについての不具合を修正しました。(Tom)

非常に稀な状況で、現在のコマンドが挿入した行が、本来無効であるべき時に既に有効になっているものとして扱われることがありました。 リリース8.0.4、7.4.9、7.3.11で組み込まれた不具合を修正しました。

- `pg_clog`および`pg_subtrans`ファイル生成時に"ファイルが既に存在する"というエラーを引き起こす競合条件を修正しました。(Tom)
- キャッシュ無効化メッセージがおかしな時刻に届いた場合にクラッシュを引き起こす可能性を修正しました。(Tom)
- 準備済み文内の`UNKNOWN`パラメータ用の`DOMAIN`制約を適切に検査します。(Neil)
- `ALTER COLUMN TYPE`が`FOREIGN KEY`、`UNIQUE`、`PRIMARY KEY`制約を確実に適切な順番で処理するようにしました。(Nakano Yoshihisa)
- 独自の演算子や演算子クラスを参照するクロススキーマを持つダンプをリストアできるように修正しました。(Tom)
- `COPY`に失敗した後、`pg_restore`が適切に処理を継続できるようにしました。 これまでは`COPY`データの残りをSQLコマンドとして扱おうとしました。(Stephen Frost)
- データディレクトリを指定しない場合に`pg_ctl unregister`がクラッシュすることを修正しました。(Magnus)
- `libpq PQprint` HTMLタグを修正しました。(Neil)
- PL/pythonにおいて、引数を渡す段階でエラーが発生した場合に適切に復旧するようにしました。(Neil)
- Win32において、PL/perlのロケールの扱いをバックエンドに合わせるように修正しました。(Andrew)
- Win32において`postgresql.conf`で`log_min_messages`が`DEBUG3`以上に設定された時のクラッシュを修正しました。(Bruce)
- Win32、Cygwin、OS X、AIX用の`pgxs -L`ライブラリ検索パス指定を修正しました。(Bruce)
- Win32の管理者権限を検査する時にSIDが有効か検査します。(Magnus)
- 範囲外の日付入力を適切に拒否します。(Kris Jurka)
- `configure`時の`finite`および`isinf`の存在検査に関する移植性を修正しました。(Tom)

E.140. リリース8.0.6

リリース日: 2006-01-09

このリリースは8.0.5の各種不具合を修正したものです。 8.0メジャーリリースにおける新機能については[項E.146](#)を参照してください。

E.140.1. バージョン8.0.6への移行

8.0.Xからの移行ではダンプ/リストアは不要です。 しかし、8.0.3より以前のバージョンからアップグレードする場合は、8.0.3のリリースノートを参照してください。 また、後述のロケールやplperlの問題の影響がある場合、移行後にテキストの列に対するインデックスをREINDEXする必要があるかもしれません。

E.140.2. 変更点

- ShmemBackendArray内に空きがない場合にpostmasterが終了することなく続行できるように、Windows用のコードを修正しました。(Magnus)

これまでの動作では、過大な要求が連続して届いた場合にサービス不能状態になる可能性がありました。 これはWindowsへの移植でのみで起こり得ました。
- 使用済みのReadBufferが新しいものとして返されることが可能となるという、8.0で混入された不具合を修正しました。これにより、最近コミットされたデータが失われる可能性がありました。(Tom)
- トランザクション外または失敗したトランザクション内から発行される、プロトコルレベルのDescribeメッセージを修正しました。(Tom)
- 韓国語など、異なる文字の組み合わせを同一のものとみなすロケールにおける文字列比較を修正しました。(Tom)

既存のテキスト列に対するインデックスを修正するにはREINDEXが必要になります。

- plperlが後でロケールを変更できないように、postmasterの起動時にロケール用の環境変数を設定します。

これは、initdb実行時に通知したロケールと異なるロケールを表す環境変数の下でpostmasterが起動された場合に発生する問題を修正します。 このような条件では、plperlの使用によりインデックスが破損される可能性がありました。 こうした問題が発生していた場合、既存のテキスト列に対するインデックスのREINDEXが必要になります。
- インストールディレクトリに関する、より柔軟な位置の変更が可能になりました。(Tom)

これまでのリリースでは、すべてのインストールディレクトリパスが最後の要素を除いて同一である場合にのみ、位置の変更をサポートしていました。
- 長期間存在した、ごく一部の特定のアジア向けマルチバイト文字セットを取り扱う場合のstrpos()と正規表現の不具合を修正しました。(Tatsuo)
- RECORDを返す関数に関するさまざまな修正を行いました。(Tom)
- MD5およびXDESアルゴリズム用のソルト空間をすべて利用できないという、/contrib/pgcryptoのgen_saltの不具合を修正しました。(Marko Kreen, Solar Designer)

Blowfishおよび標準DES用のソルトには影響ありません。
- 指定した列数が実際に問い合わせで返される列数と異なる場合にクラッシュせずにエラーを返すように/contrib/dblinkを修正しました。(Joe)

E.141. リリース8.0.5

リリース日: 2005-12-12

このリリースは8.0.4の各種不具合を修正したものです。 8.0メジャーリリースにおける新機能については[項E.146](#)を参照してください。

E.141.1. バージョン8.0.5への移行

8.0.Xからの移行ではダンプ/リストアは不要です。 しかし、8.0.3より以前のバージョンからアップグレードする場合は、8.0.3のリリースノートを参照してください。

E.141.2. 変更点

- トランザクションログ管理における競合条件を修正しました。

I/O操作により間違ったページが初期化されてしまうわずかな状態がありました。 これは、アサート失敗またはデータ破壊をもたらします。

- エラー修復後のbgwriter問題を修正しました。(Tom)

書き込みエラーの後、バックグラウンドライタがあふれたバッファピンを見つけ出していました。 これ自体は致命的ではありませんが、これによりその後のVACUUMコマンドが不可解なブロックを行う可能性があります。

- 現在のトランザクションがすでにアボートしている時にクライアントがBindプロトコルメッセージを送信した場合の失敗を防ぎます。

- `/contrib/ltree`を修正しました。(Teodor)

- AIX、HPUXにおけるコンパイルを修正しました。(Tom)

- WindowsのNO_SYSTEM_RESOURCESエラーが発生した後に、ファイル読み書きを再試行します。(Qingqing Zhou)

- `log_line_prefix`が%iを含む場合の间歇エラーを防ぎます。

- Windowsにおける長いスクリプトに関するpsqlの性能問題を修正しました。(Merlin Moncure)

- `pg_group`通常ファイルの更新喪失を修正しました。

- 長期間残っていた、外部結合の実行計画作成エラーを修正しました。

この不具合により、"マージ結合可能な結合条件のみをサポートするRIGHT JOIN"という不正なエラーが発生することがありました。

- `postmaster.pid`の生成後まで時間帯の初期化を延ばしました。

これにより、PIDファイルが先に存在することを想定する起動スクリプトの混乱を防止できます。

- テーブルが削除された場合の`pg_autovacuum`のコアダンプを防止します。

- 副問い合わせ結果全体の行を参照する (`foo.*`) 場合の問題を修正しました。

E.142. リリース 8.0.4

リリース日: 2005-10-04

このリリースは8.0.3の各種不具合を修正したものです。 8.0メジャーリリースにおける新機能については[項E.146](#)を参照してください。

E.142.1. バージョン8.0.4への移行

8.0.Xからの移行ではダンプ/リストアは不要です。 しかし、8.0.3より以前のバージョンからアップグレードする場合は、8.0.3のリリースノートを参照してください。

E.142.2. 変更点

- `VACUUM`によって`ctid`チェーンがすぐに削除されてしまうエラーを修正し、`ctid`リンクを追跡するコードにおいてより多くの検査を行うようにしました。

これにより、長い間存在した、非常に稀な環境におけるクラッシュという問題が修正されました。

- マルチバイト文字セットを使用する場合に、`CHAR()`の空白文字埋め込みが適切に指定した長さになるように修正しました。(Yoshiyuki Asaba)

以前のリリースでは、`CHAR()`の空白文字埋め込みが正しくありませんでした。 格納する文字数を考慮せずに、単に指定されたバイト数で空白文字を埋め込んでいただけでした。

- `CREATE DATABASE`をコミットする前にチェックポイントを強制実行します。

これにより、`CREATE DATABASE`の実行後すぐにクラッシュしてしまった場合に起きる"index is not a btree"という失敗が修正されたはずです。

- `COPY`における読みとりのみのトランザクションに関する試験の意味を修正しました。

このコードで`COPY TO`を禁止していましたが、本来ならば`COPY FROM`を禁止しなければなりません。

- CSVモード入力の`COPY`において、連続して埋め込まれた改行を取り扱うようになりました。

- 年末近くの日付に対する`date_trunc('week')`を修正しました。

- リレーションの内部のみを参照する、外部結合のON句に関する計画作成の問題を修正しました。

- `x FULL JOIN y ON true`の境界に関して更に改修しました。

- `x IN (SELECT DISTINCT ...)`と関連する状態に対する過度な最適化を修正しました。

- "あいまいな"コスト比較の考慮が十分でなかったため、小さな`LIMIT`値の問い合わせに対する計画作成の間違いを修正しました。

- `array_in`と`array_recv`のOIDパラメータの検証をより偏執的に行うようにしました。

- `a`列にGiSTインデックスを持つ場合の`UPDATE a=... WHERE a=...`といった問い合わせにおいて行が見つからない点を修正しました。

- 日付時刻の解析の堅牢性を向上しました。

- 部分的に書き込まれたWALページに対する検査を改良しました。

- SSL有効時のシグナルの扱いに対する堅牢性を向上しました。

- MIPSおよびM68Kのスピンロックコードを改良しました。

- postmasterの起動時に、`max_files_per_process`以上のファイルを開かないようにしました。

- さまざまなメモリリークを修正しました。

- 移植性の向上をいろいろと行いました。
- タイムゾーンデータファイルを更新しました。
- WindowsにおけるDLLロード失敗の扱いを改良しました。
- Windowsにおける乱数発生を改良しました。
- `psql -f filename`において、指定したファイルを開くことができなかった場合の終了コードを非0にしました。
- 継承された検査制約をより正確に扱うように`pg_dump`を変更しました。
- Windowsにおいて、`pg_restore`のパスワード入力待ちを修正しました。
- PL/pgSQLにおいて、変数が参照渡し型の場合の`var := var`の扱いを修正しました。
- PL/Perlの`%_SHARED`を修正し、本当に共有できるようになりました。
- 2000秒以上の間隔で待機できるように`contrib/pg_autovacuum`を修正しました。
- 正確なSnowballコードを使用するように`contrib/tsearch2`を更新しました。

[前のページ](#)
リリース8.0.5

[ホーム](#)
[上に戻る](#)

[次のページ](#)
リリース8.0.3

E.143. リリース8.0.3

リリース日: 2005-05-09

このリリースは、複数のセキュリティ関連の問題を含む、8.0.2の各種不具合を修正したものです。 8.0メジャーリリースにおける新機能については[項E.146](#)を参照してください。

E.143.1. バージョン8.0.3への移行

8.0.Xからの移行ではダンプ/リストアは不要です。しかし、8.0.xのシステムカタログで見つかった2つの重大なセキュリティ問題を突かれる可能性があります。 ダンプ、8.0.3のinitdbを使用したinitdb、リロードを行うことで、自動的にこれらの問題を修正します。

より重大なセキュリティ問題は、組み込みの文字セット符号化変換関数により、権限を持たないユーザがSQLコマンドを呼び出すことができるという点です。このような用途のためにこれらの関数を設計していませんでしたが、悪意のある引数の設定に対する安全性がありませんでした。この修正により、これらの関数の宣言されたパラメータリストがSQLコマンドから呼び出されないように変更されました。（通常の符号化変換機構の使用には影響はありません。）

もうひとつの問題は、contrib/tsearch2モジュールが、internal型の引数を取らない場合に間違ってinternalを返すものと宣言された関数を複数作成することです。これによりinternal型の引数を使用するすべての関数について、型の安全性が損なわれます。

initdb、もしくは、後述の手作業による修正手順に従って、すべてのインストレーションにおいてこれらのエラーを修正することを強く勧めます。これらのエラーにより、少なくとも、権限を持たないデータベースユーザがサーバプロセスをクラッシュさせることができます。また、権限を持たないユーザがデータベーススーパーユーザ権限を手に入れることができる可能性もあります。

initdbを行いたくないのであれば、[7.4.8リリースノート](#)で示した手作業による修正手順を行ってください。

E.143.2. 変更点

- 誤使用を防ぐために符号化関数のシグネチャを変更しました。
- INTERNAL結果の関数の危険な使用を防止するように、contrib/tsearch2を変更しました。
- 間違った第2パラメータからrecord_outを保護しました。
- ある目的（例えばSELECT FOR UPDATE）用にコミットされるように見えるトランザクションが直後に別の目的用にコミットされるという従来から存在した競合状況を修正しました。

一貫性が明らかにないデータが短期間アプリケーションから参照できてしまうため、これはきわめて重大な不具合です。

- リレーションの拡張とVACUUMとの間の競合状態を修正しました。

理論上、これにより新たに挿入されたページの価値が失われます。しかしこうした状況が発生する頻度はきわめてまれで、アサートエラー以外に発生した事例はありません。

- TIME WITH TIME ZONE型の値の比較を修正しました。

--enable-integer-datetimes設定スイッチが使用されている場合、この比較コードに間違いがありました。注意: この修正により列値のソート順が正しくなるように変更されましたので、TIME WITH TIME ZONE型の列に対するインデックスを持つ場合、この更新をインストールした後にREINDEXを行う必要があります。

- TIME WITH TIME ZONE型の値に対するEXTRACT(EPOCH)を修正しました。
- INTERVAL型の値の秒の小数部分において、負の値が間違って表示される点を修正しました。

このエラーは、--enable-integer-datetimes設定スイッチが使用された場合にのみ発生します。

- %を含む名前のトリガを正しくダンプできるようにpg_dumpを修正しました。
- contrib/intaggに対する更なる64ビット向けの修正を行いました。

RECORD型を返す関数の最適化の間違いを修正しました。

- COALESCE(NULL, NULL) でのクラッシュを防止しました。
- libpq用のBorland向けmakefileを修正しました。
- timetz型に対するcontrib/btree_gistを修正しました。(Teodor)
- プロセスがまだ稼働しているかどうかを確認するために、pg_ctlでpostmaster.pid内のPIDがあるかどうかを検査するようにしました。
- ダンプのタイムスタンプ追加により起こるpg_dump/pg_restoreの問題を修正しました。
- 保持可能なカーソルの実現とトランザクションコミット時の遅延されたトリガの発行との間の相互作用を修正しました。
- 参照渡しの変数型を返すSQL関数におけるメモリリークを修正しました。

[前のページ](#)

リリース 8.0.4

[ホーム](#)
[上に戻る](#)

[次のページ](#)

リリース8.0.2

E.144. リリース8.0.2

リリース日: 2005-04-07

このリリースは、8.0.1の各種不具合を修正したものです。 8.0メジャーリリースにおける新機能については[項E.146](#)を参照してください。

E.144.1. バージョン8.0.2への移行

8.0.*からの移行ではダンプ/リストアは不要です。 このリリースではPostgreSQLライブラリの主バージョン番号が更新されました。 従って、ユーザアプリケーションが適切な番号の共有ライブラリを検出できなかった場合、再リンクしなければならないかもしれません。

E.144.2. 変更点

- すべてのインタフェースライブラリの主バージョン番号を増加させました(Bruce)

これは8.0.0の時に行われるはずでした。 psqlなどの7.4.x版のPostgreSQLクライアントアプリケーションを8.0.xアプリケーションと同じマシンでできるようにするために要求されました。 こうしたライブラリを使用するユーザアプリケーションの再リンクが必要になるかもしれません。

- Windows専用のwal_sync_methodの設定fsync_writethroughを追加しました(Magnus, Bruce)

この設定により、PostgreSQLはWALへの書き出し時にディスクドライブのライトキャッシュを経由して書き出すようになります。 この振舞いは以前はfsyncと呼ばれていましたが、他のプラットフォームにおけるfsyncとかなり異なる動作であるため別の名前にしました。

- Windows上でwal_sync_methodの設定open_datasyncを有効にし、このプラットフォームにおけるデフォルトにしました(Magnus, Bruce)

デフォルトがfsync_writethroughではありませんので、ディスクドライブのライトキャッシュが有効な場合、電源障害時にデータ損失の可能性があります。 Windows上でライトキャッシュを無効にするためには、デバイスマネージャからドライブのプロパティを選択し、そしてポリシーを選択してください。

- 新しいキャッシュ管理アルゴリズム2QでARCを置き換えました(Tom)

これはARCに関する審査待ちの米国特許を避けるために実施しました。 2Qコードは、ある種の作業状態ではARCより数パーセント速度が落ちる可能性があります。 より優れたキャッシュアルゴリズムが8.1で導入される予定です。

- 新規に作成されたテーブルに対する動作を改良するためのプランナの調整 (Tom)

- plpgsqlにて初期値がNULLの配列要素に代入できるようにしました(Tom)

これまではこうした配列はNULLのままでしたが、現在では単一要素の配列になりました。 8.0にてNULL配列値のUPDATEの主SQLエンジンにおける扱いがこの方法に変更されましたが、plpgsqlにおける同じ状況が見落とされていました。

- plpythonの関数本体において¥r¥nと¥rを¥nに変換しました(Michael Fuhr)

これにより、plpythonのコードがWindowsクライアントやMacクライアントで作成された場合の構文エラーを防止できます。

- SPIカーソルでEXPLAINなどの行を返すユーティリティコマンドを扱えるようにしました (Tom)

- ALTER TABLE SET WITHOUT OIDS後のCLUSTERの障害を修正しました (Tom)

- ALTER TABLE ADD COLUMNのメモリ消費量を減少させました(Neil)

- ALTER LANGUAGE RENAMEを修正しました (Tom)

- pg_ctlのWindows独自のregisterオプションとunregisterオプションを文書化しました(Magnus)

- バックエンドシャットダウン中に行われた操作が統計情報コレクタで確実に集計されるようにしました。

これは、システムカタログのバキューム処理が十分な間隔で行われていないというpg_autovacuumの報告を解消するためのものです。 バックエンド終了時の一時テーブル削除が原因のカタログの削除に関するものではありません。

- `log_destination` 設定パラメータのWindowsのデフォルトをeventlogに変更しました (Magnus)

デフォルトでは、Windowsで稼動しているサーバはログ出力を標準エラーではなく、Windowsイベントロガーに送信します。

- Kerberos認証をWindowsで動作できるようにしました(Magnus)
- `ALTER DATABASE RENAME`を、CREATEDB権限を持たないスーパーユーザで実行できるようにしました(Tom)
- `CREATE`および`DROP DATABASE`のWALログ項目を絶対パスを指定しないように変更しました(Tom)

これにより、ポイントインタイムリカバリをデータベースの格納場所が異なる可能性がある別のマシンで実行できるようになりました。 こうした状況はまだ`CREATE TABLESPACE`では危険であることに注意してください。

- テーブルを作成し、そのテーブルへのカーソルを開いているトランザクションが実行中に、そのバックエンドが終了した場合のクラッシュを修正しました。(Tom)
- `array_map()`を修正し、PL関数からそれを呼び出すことができるようにしました(Tom)
- `contrib/tsearch2`および`contrib/btree_gist`に関する複数の修正 (Teodor)
- あるプラットフォームにおける`contrib/pgcrypto`関数のクラッシュを修正しました (Marko Kreen)
- 64ビットプラットフォーム用に`contrib/intagg`を修正しました (Tom)

- `CREATE`文の解析におけるECPGの不具合を修正しました (Michael)
- ECPGの問題の原因となる、powerpcおよびamd64におけるgccの不具合を回避しました (Christof Petig)
- ロケールがCの場合に、ロケールを考慮した`upper()`、`lower()`、`initcap()`を使用しないようにしました (Bruce)

これにより、ロケールがCの場合に非7ビットデータに対してエラーを生成するプラットフォームで、こうした関数が動作するようになります。

- `quote_ident()`がキーワードに一致する名前を引用符で括るように修正しました (Tom)
- `to_date()`がCCフィールドとYYフィールドの両方が使用された場合に合理的に振舞うように修正しました (Karel)
- `to_char(interval)`が月の間隔として0が指定された場合に失敗することを防止しました (Tom)
- `date_trunc('week')`が間違った週を返さないように修正しました (Bruce)

`date_trunc('week')`は、ある年の1月の初めの数日に対して間違った年を返しました。

- `INET`データ型において、クラスDアドレスのデフォルトのマスク長を正しく使用するようにしました (Tom)

E.145. リリース8.0.1

リリース日: 2005-01-31

このリリースは、複数のセキュリティ関連の問題を含む、8.0.0の各種不具合を修正したものです。8.0メジャーリリースにおける新機能については[項E.146](#)を参照してください。

E.145.1. バージョン8.0.1への移行

8.0.0からの移行ではダンプ/リストアは不要です。

E.145.2. 変更点

- スーパーユーザ以外のLOADを不許可としました

共有ライブラリの初期化関数を自動的に行うプラットフォーム(少なくともWindowsやELFベースのUnixを含みます)では、LOADを使用して任意のコードを実行させることができました。この問題を報告したNGSソフトウェアに感謝します。

- 集約関数の作成者が指定した遷移関数を実行する権利を持つかどうかを検査します。

この見落としのため、関数のEXECUTE権限を回避することができてしまいました。

- contrib/intaggにおけるセキュリティと64ビット問題を修正しました。
- 一部のcontrib関数で必要であったSTRICT印を追加しました。(Kris Jurka)
- plpgsqlのカーソル定義の括弧が多すぎた場合のバッファオーバーランを防止しました。(Neil)
- すべての場合においてALTER TABLE ADD COLUMNがドメイン制約を強制するようにしました。
- FULLおよびRIGHT外部結合用の計画作成エラーを修正しました。

結合結果が間違っ、左側入力と同じようにソートされたものとみなされていました。これはユーザに間違っソート結果を導くものではありませんが、入れ子状マージ結合の場合は全く間違っ回答を与えるかもしれませんでした。

- グループ化された集約関数の計画作成の改良
- ROLLBACK TO savepointで、セーブポイント以降に作成されたカーソルをクローズした。
- Windowsにおいて不適切なバックエンドスタックサイズを修正しました。
- WindowsにおいてSHGetSpecialFolderPath()を取りやめました(Magnus)
- Windowsサービスとして実行する場合のpg_autovacuumの問題を修正しました(Dave Page)
- pg_dump/pg_restoreにおける複数の小さな不具合を修正しました。
- typedefsで名前付き構造体を使用した場合のecpgのsegfaultを修正しました(Michael)

E.146. リリース8.0

リリース日: 2005-01-19

E.146.1. 概要

以下に本リリースにおける大きな変更点を示します。

Microsoft Windowsネイティブサーバ

このリリースは、Microsoft Windows®上でサーバとしてネイティブに稼動する最初のPostgreSQLリリースです。これはWindowsサービスとしても稼動可能です。このリリースでは、Windows 2000 SP4、Windows XP、および、Windows 2003などのNTベースのWindowsをサポートします。Windows 95、Windows 98、および、Windows MEなどの古めのリリースはサポートされていません。これらのオペレーティングシステムではPostgreSQLをサポートするための基盤が存在しないためです。別に、Windows上でのインストールを簡単に行うために、インストーラプロジェクトが設立されました。 <http://www.postgresql.org/ftp/win32/> を参照してください。

リリース工程を通して試験を行いました。Windowsへの移植においてはPostgreSQLがUnixプラットフォームで培ってきた長年の製作環境の経験という利点がありません。そのため、新しい製品を使用する時と同等の注意をもってこれを扱ってください。

以前のリリースでは、WindowsオペレーティングシステムでサーバをサポートするためにはCygwin Unixエミュレーションツールキットが必要でした。PostgreSQLは数年に渡ってWindows上のネイティブなクライアントをサポートしていました。

セーブポイント

セーブポイントにより、トランザクションの特定の一部を他の部分に影響することなく、アボートさせることができます。以前のリリースではこうした機能はありませんでした。トランザクション全体をアボートさせる以外に、トランザクション内のある文の失敗から復旧させる方法はありませんでした。この機能は、複雑なトランザクション内でエラーからの復旧が必要なアプリケーション作成者にとって有意です。

ポイントインタイムリカバリ

これまでのリリースでは、以前のバックアップからリストアする、もしくは、スタンバイ・リプリケーションサーバを使用する以外に、ディスク装置の障害から復旧させる方法はありませんでした。ポイントインタイムリカバリにより、サーバの連続的なバックアップが可能です。障害発生時点まで復旧させることも過去のあるトランザクションまで復旧させることも可能です。

テーブル空間

テーブル空間を使用して、管理者は個々のテーブル、インデックス、および、データベース全体を格納するために異なるファイルシステムを選ぶことができます。これは、性能を改善し、また、ディスク領域全体の使用状況を制御できます。これまでのリリースでは、initlocationを使用し、こうした処理のためには手作業によってシンボリックリンクを管理していました。

バッファ管理、CHECKPOINT、VACUUMの改良

このリリースでは、より高度なバッファ交換戦略を使用します。これにより、利用可能な共有バッファの使用が改善され、性能が向上します。バキューム処理やチェックポイント処理が性能に与える影響も軽減されます。

列データ型の変更

`ALTER TABLE` で列のデータ型が変更できるようになりました。

新しいPerlサーバサイド言語

新しいバージョンのplperlサーバサイド言語では、永続的共有格納領域、トリガ、レコードを返すこと、レコードの配列を返すこと、データベースにアクセスするSPI呼び出しをサポートするようになりました。

COPYにおけるカンマ区切り値書式(CSV)サポート

`COPY` で、カンマ区切り値書式ファイルの読み書きが可能になりました。これはまた、非標準的な引用符や文字の区切りを柔軟に解釈できます。

E.146.2. バージョン8.0への移行

全ての以前のリリースからデータの移行を行うためには、pg_dumpを使用したダンプ、リストアが必要です。

以下の非互換性に注意してください。

- `READ COMMITTED` シリアライゼーションモードにおいて、揮発性関数が、その関数の呼び出し元となったコマンドが始まるまでではなく、その関数内の各文が始まるまでに同時実行中のトランザクションによってコミットされた結果を参照できるようにになりました。
- `STABLE` もしくは `IMMUTABLE` と宣言された関数は常に呼び出し中の問い合わせのスナップショットを使用します。そのため、呼び出し中の問い合わせが始まってから行われた活動の影響を、その活動が自身のトランザクションであっても他のトランザクションであっても、参照しません。こうした関数は読み取りのみでなければなりません。つまり、`SELECT` 以外のSQLコマンドを全く使用することができません。
- 非遅延の `AFTER` トリガが、現在の対話的コマンドの完了後ではなく、トリガの発行元となる問い合わせの完了後、即座に発行されるようになりました。これはトリガの発行元となる問い合わせが関数内にある時に差異が現れます。この場合トリガは、関数が次の操作を進める前に呼び出されます。
- サーバ設定パラメータ `virtual_host` と `tcpip_socket` が、より一般的なパラメータ `listen_addresses` に置き換えられました。また、サーバはデフォルトで `localhost` に対する監視を行うようになりました。これにより、多くの状況において `-i` `postmaster` スイッチの必要性がなくなりました。
- サーバ設定パラメータ `SortMem` および `VacuumMem` の名前が、より使用状況に合った `work_mem` および `maintenance_work_mem` という名前に変更されました。元の名前は `SET` や `SHOW` でまだサポートされています。
- サーバ設定パラメータ `log_pid`、`log_timestamp`、`log_source_port` が、より柔軟な `log_line_prefix` に置き換えられました。
- サーバ設定パラメータ `syslog` がより論理的な `log_destination` 変数に置き換えられ、この変数によってログの出力先を制御するようになりました。
- データ変更のみやデータ定義文のみを選択できるように、サーバ設定パラメータ `log_statement` が変更されました。サーバ設定パラメータ `log_duration` は、`log_statement` が問い合わせを出力する場合にのみ出力するようになりました。
- サーバ設定パラメータ `max_expr_depth` が `max_stack_depth` に置き換えられました。これは、式の入れ子レベルではなく物理的なスタックサイズを単位とします。再帰的な関数によって引き起こされるスタックオーバーフローが原因で、セッションが終了してしまうことを防止するために、これは役に立ちます。
- `length()` が、`CHAR(n)` の値の最後に含まれる空白を数えないようになりました。
- 整数から `BIT(N)` へのキャストにおいて、整数の右から `N` ビットが選択されるようになりました。以前は左から `N` ビットを選択していました。
- `NULL` 配列値の一要素もしくは一部を更新することで、非 `NULL` の配列、すなわち、代入された位置だけからなる配列が生成されるようになりました。
- 配列への入力値の構文チェックがかなり厳しくなりました。これまではおかしい場所に入力でき、おかしい結果を返していたごみが、エラーになるようになりました。空文字列の要素値は、何も入力しないのではなく、`""` と記述しなければなりません。また、配列要素の前後の空白に関する動作も変更されました。後にある空白が、前にある空白と同義（これはこれまででも無視されていました）となるように無視されるようになりました。
- 整数演算操作のオーバーフローが検出されるようになりました。エラーとして報告されます。
- 単一バイトの `char` データ型に関連する算術演算子が削除されました。
- `extract()` 関数 (`date_part` と呼ばれます) が紀元前の時間に対して正確な年を返すようになりました。以前は正しい値よりも1少ない値が返されていました。また、この関数が千年紀や世紀についても正しい値を返すようになりました。
- `CIDR` 値のマスクされないビットがゼロでなければならなくなりました。例えば、`204.248.199.1/31` は `CIDR` の値としてもはや許されません。こうした値はこれまでもPostgreSQLで受け付けられないはずでしたが、これが拒絶されるようになりました。
- `EXECUTE` が、実行した文に合った完了タグを返すようになりました。
- `psql` の `%copy` コマンドが、`psql` の `stdin/stdout` ではなく問い合わせの `stdin/stdout` を読み書きするようになりました。新しい `pstdin/pstdout` パラメータ経由でアクセスすることで以前の動作を行うことができます。

- JDBCクライアントインタフェースが標準配布から削除され、<http://jdbc.postgresql.org>から提供されるようになりました。
- TCLクライアントインタフェースも削除されました。<http://gborg.postgresql.org>で、複数のTCLインタフェースが提供されています。
- サーバが、オペレーティングシステムで提供されるものではなく、独自の時間帯データベースを使用するようになりました。これにより全てのプラットフォームにおいて一貫性のある振舞いが提供されます。ほとんどの場合、時間帯の動作における注意すべき差異はほぼありませんが、`SET/SHOW TimeZone`で使用される時間帯名称はプラットフォームが提供する名称と異なる可能性があることに注意してください。
- `configure`のスレッド用オプションのために、ユーザによる試験やファイルの編集が不要になりました。スレッド用オプションが自動的に検出されるようになりました。
- テーブル空間が実装されました。そのため`initlocation`が削除されました。
- ユーザ定義のGiSTインデックス用のAPIが変更されました。UnionおよびPickSplitメソッドで、`bytea`ではなく、特別な`GistEntryVector`構造体へのポインタが渡されるようになりました。

E.146.3. 廃止予定の機能

PostgreSQLの動作の一部は準最適となることが決定されています。後方互換性のために、これらは8.0では削除されませんが、非推奨であり、次のメジャーリリースで削除される予定です。

- リリース8.1では`interval`に対する`to_char()`関数は削除される予定です。
- `oid/float4/float8`データ型に空文字列を代入した時に、サーバが警告を発するようになりました。これまで同様に、0として解釈して処理を継続します。次のメジャーリリースでは、空文字列はこうしたデータ型では無効とみなす予定です。
- デフォルトでは、PostgreSQL 8.0以前のテーブルは`OID`付きで作成されます。次のリリースでは、これは事実でなくなる予定です。`OID`付きでテーブルを作成するためには、`WITH OIDS`句を指定する、あるいは、`default_with_oids`設定パラメータを設定する必要があります。将来のリリースのPostgreSQLとの互換性のために、`OID`が必要なテーブルを使用しているユーザは明示的に`WITH OIDS`を指定することを勧めます。

E.146.4. 変更点

以下に、リリース8.0とこれまでのメジャーリリースとの間での変更点の詳細な内容を示します。

E.146.4.1. 性能向上

- データ型を跨るインデックスの使用をサポート(Tom)

この変更を行うまで、データ型が完全に一致しない場合、多くの問い合わせがインデックスを使用しませんでした。この改良によりインデックスの使用がより直感的かつ一貫的になります。

- 新しいバッファ交換戦略によるキャッシュ処理の改良(Jan)

これまでのリリースでは最長未使用時間 (LRU) キャッシュを使用して、最近に参照されたページをメモリ内に保持していました。このLRUアルゴリズムでは、特定のキャッシュ項目が何回アクセスされたかを考慮しませんでした。そのため、大規模なテーブルスキャンにより、よく使用されるキャッシュページがメモリ外に追いやられてしまいます。新しいキャッシュアルゴリズムは、4つの独立したリストを使用して、最も最近に使用されたキャッシュページ、最もよく使用されたキャッシュページを管理し、作業負荷に基づいてその交換を動的に最適化します。これにより共有バッファキャッシュが非常に効率的に使用されるようになるはずです。これまで共有バッファのサイズを試行錯誤して決定していた管理者は、この新しいキャッシュ交換ポリシーで再度決定してください。

- 定期的にダーティバッファを書き出すサブプロセスの追加によりチェックポイント時の書き出しを低減(Jan)

これまでのリリースでは、数分程度の周期で実行されるチェックポイントプロセスが全てのダーティバッファをオペレーティングシステムのキャッシュに書き込み、オペレーティングシステムのダーティバッファをディスクに書き出していました。これによりディスクの使用に関して周期的なピークが現れるようになり、性能を悪化させることがよく起こりました。新しいコードでは、バックグラウンドライタを使用して、一定のペースで小刻みに書き出すようにしたので、チェックポイントの際にダーティページの書き出しがかなり少なくなりました。更に新しいコードでは、大域的な`sync()`を呼び出さずに、直前のチェックポイント以降に書き込まれたファイルのみに対する`fsync()`を呼び出します。これにより性能は向上し、チェックポイント時の性能劣化も最小限になります。

- バキュームを延長する機能の追加により性能への影響を軽減(Jan)

負荷が高いシステムにおいてVACUUM処理は、多くのI/O要求を行いますので、他のユーザにとっての性能を悪化させてしまいます。このリリースでは、VACUUM速度を低めることで、他のユーザへの影響を低減させることができます。しかし、これはVACUUMの処理時間を多くしてしまいます。

- 重複キーに対するB-treeインデックス性能の向上(Dmitry Tkach, Tom)

インデックス内に重複する値が多い場合のインデックスのスキャン方法が改良されました。

- 計画作成時、動的に生成されるテーブルサイズ推定値を使用する(Tom)

これまでプランナは、最後に実行したVACUUMまたはANALYZEによって判明した値を使用して、テーブルの物理サイズ(ページ数)や行数などのテーブルサイズを推定していました。現在のテーブルの物理サイズをカーネルから取得し、このテーブルサイズに最後に実行したVACUUMまたはANALYZEによって判明した行密度(1ページあたりの行数)を乗じて行数を推定するようになりました。これにより、最後に保守用のコマンドを実行してからテーブルサイズに大きな変更があった場合でも、より信頼できる推定ができるようになるはずです。

- OR句のある場合のインデックス使用の改良(Tom)

これにより、オプティマイザは、インデックス付けされていない、多くのOR句を持つ文に対してインデックスを使用できるようになりました。また、複数列に対するインデックスが、その最初の列が指定され、2番目の列がOR句の一部である場合でも使用できるようになりました。

- 部分インデックス句の一致処理の改良(Tom)

サーバが、複雑なWHERE句を持つ問い合わせにおける部分インデックスの使用に関してより賢明になりました。

- GEQOオプティマイザの性能向上(Tom)

多く(デフォルトでは12以上)のテーブルを持つ問い合わせの計画を作成する時、GEQOオプティマイザが使用されます。このリリースでは、最適化に費やされる時間を減らすために、問い合わせの解析方法を高速化しました。

- 雑多なオプティマイザの改良

実施された小規模な改良を全てここで示すことはできませんが、特定状況における多くの改善がこれまでのリリースよりなされています。

- C関数の検索速度の改良(Tom)

このリリースでは、動的にロードされたC関数の情報を検索する時にハッシュテーブルを使用します。これにより、検索がサーバの実行形式に組み込まれた関数とほぼ同等の速度で実行されますので、速度が向上します。

- 型固有のANALYZE統計処理機能の追加(Mark Cave-Ayland)

この機能により、より柔軟な非標準のデータ型用の統計情報の生成が可能です。

- ANALYZEが式インデックスに対する統計情報を収集するようになりました(Tom)

式インデックス(関数インデックスとも呼ばれます)により、ユーザは単なる列だけではなく式や関数呼び出しの結果に対してインデックスを付けることができます。このリリースでは、オプティマイザは式インデックスの内容に関する統計情報を収集、使用することができます。これにより式インデックスが最適な問い合わせに対して作成される計画の質がかなり向上します。

- ANALYZE用の新しい2段階サンプリング手法(Manfred Koizar)

これにより、テーブルの領域によって有効な行の密度が非常に異なる場合により優れた統計情報が得られます。

- TRUNCATEの速度向上(Tom)

7.4で見受けられたTRUNCATEの性能劣化を、トランザクションセーフのまま取り戻しました。

E.146.4.2. サーバにおける変更

- WALファイルアーカイブとポイントインタイムリカバリの追加(Simon Riggs)
- ディスクレイアウトの制御ができるテーブル空間の追加(Gavin)
- 組み込みログ回転プログラムの追加(Andreas Pflug)

syslogや他のログ回転プログラムに依存することなく簡単にサーバメッセージのログができるようになりました。

- 以下のサーバコンパイル時の設定を表示するための、読み取りのみのサーバ設定パラメータの追加
block_size、integer_datetimes、max_function_args、max_identifier_length、max_index_keys (Joe)
- sameuser、samegroup、allという引用符付けによるpg_hba.confでの特殊な意味付けを取消し(Andrew)
- デフォルトのpg_hba.conf内にてより明瞭なlocalhost用のIPv6名::1/128を使用(Andrew)
- pg_hba.confの例においてCIDR書式を使用(Andrew)
- サーバ設定パラメータSortMemとVacuumMemの名前をwork_memとmaintenance_work_memに変更(古い名前もまだサポートします)(Tom)

この変更は、インデックスや外部キーの作成などのバルク操作にはmaintenance_work_mem、問い合わせの実行中に使用される作業空間用にはwork_mem、と明確に分けるために行われました。

- log_disconnectionsサーバ設定を使用したセッション切断のログを可能にした (Andrew)
- 各ログ行に発行される情報を制御できるようにlog_line_prefixサーバ設定パラメータを新規追加(Andrew)
ユーザ名、データベース名、リモートのIPアドレス、セッションの開始時刻を含めることができます。
- log_pid、log_timestamp、log_source_portサーバ設定パラメータの削除。これらの機能はlog_line_prefixで置き換えられました。(Andrew)
- virtual_hostとtcpip_socketパラメータを統合しlisten_addressesパラメータに置き換えた(Andrew, Tom)

virtual_hostでは単一の接続監視用IPアドレスしか指定できませんでした。listen_addressesでは複数のアドレスを指定することができます。

- デフォルトでローカルホストの接続監視を行い、多くの状況で-i postmasterスイッチの必要性を省いた(Andrew)
ローカルホスト(127.0.0.1)の接続監視によってセキュリティ問題が発生することはありません。更に、ローカルソケットをサポートしないWindowsやJDBCなどの設定において特別な調整を行うことなく動作させることができます。
- syslogサーバ設定パラメータを削除。ログ出力先を制御する、より論理的なlog_destination変数を追加。
- どの問い合わせをログに記録するかを選択するためにlog_statementサーバ設定パラメータがall、mod、ddl、noneを取るように変更(Bruce)

これにより、管理者はデータ定義の変更のみやデータ変更文のみのログを取得することができます。

- 一部のログ取得関連の設定パラメータは以前から普通のユーザによって調整可能でしたが、"より冗長"方向でのみにしか調整できませんでした。これらがより厳密に扱われるようになり、スーパーユーザのみがこれらを設定することができます。しかし、スーパーユーザはALTER USERを使用して、ユーザ単位で非スーパーユーザ用の設定を提供することができます。また、スーパーユーザはPGOPTIONSを介してスーパーユーザのみの設定パラメータの値を設定することもできます。
- 設定ファイルをデータディレクトリ外に格納できるようにした(mlw)

デフォルトでは、設定ファイルはクラスタの最上位ディレクトリにあります。この追加によって、設定ファイルをデータディレクトリ以外に格納することができるようになり、管理しやすくなりました。

- 最初に実行される時のみに準備済み問い合わせの計画を作成。これで定数を統計処理に使用することが可能。(Oliver Jowett)

準備済み文はその計画を一度作成し、何回も実行します。準備済み文では、その使用時に再計画分のオーバーヘッドを防ぐことができますが、計画の質は、問い合わせで使用される正確なパラメータが不明である分劣化します。このリリースでは、無名の準備済み分の計画作成は最初の実行まで遅延され、そしてその実行時に使用された実際のパラメータ値を最適化のヒントとして使用します。これにより、行外のパラメータ渡しを性能を落とすことなく使用できるようになりました。

- パラメータを取るDECLARE CURSORが可能(Oliver Jowett)

パラメータ付きParseメッセージにてDECLARE CURSORを発行する際に有用です。Bind時点で送信されるパラメータ値が、カーソルの問い合わせの実行時に置き換えられます。

- inetとcidrデータ型に対するハッシュ結合と集約の修正(Tom)

リリース7.4における、`inet`と`cidr`値が混在した場合のハッシュ処理に不具合がありました。（これらの型のハッシュを作成することはありませんので、この不具合はそれまでのリリースでは存在しません。）

- `log_statement`が問い合わせを出力する場合のみに `log_duration`を出力するようにしました。(Ed L.)

E.146.4.3. 問い合わせにおける変更

- セーブポイント(入れ子のトランザクション)機能の追加(Alvaro)
- 未サポートの隔離レベルを受け付け、サポートされたレベルの中で最も近いレベルに移るようになりました。(Peter)

SQL仕様では、データベースが指定した隔離レベルをサポートしない場合はその次に厳しいレベルを使用すべきであると規定しています。 この変更はこの推奨に従ったものです。

- `START TRANSACTION`同様`BEGIN WORK`でトランザクション隔離レベルを指定可能です。
- 受け付けた元の問い合わせと異なる問い合わせ型を生成するルールについてテーブル権限の検査を修正(Tom)
- 単一引用符の使用を簡単にするドル引用符の実装(Andrew, Tom, David Fetter)

これまでのリリースでは、関数本体に引用符を付けるために単一引用符が使用されていたので、関数テキストの内部の単一引用符を、二重の単一引用符、もしくはエラーが起こりやすい記述とする必要がありました。 このリリースでは、テキストブロックの引用符付けに「ドル引用符付け」を使用できるように機能を追加しました。 この機能により、異なる入れ子レベルで異なる引用符区切り文字を使用できますので、正確な引用符付けという処理がかなり簡単になりました。

- `CASE val WHEN compval1 THEN ...`における`val`の評価を1回だけにしました(Tom)

`CASE`が試験済みの式を複数回評価しないようになりました。 式が複雑な場合や揮発性がある場合に利点があります。

- 集約問い合わせの目的リストを計算する前の`HAVING`検査(Tom)

`SELECT SUM(win)/SUM(lose) ... GROUP BY ... HAVING SUM(lose) > 0`などの場合における不適切な失敗を改修しました。 これは正常動作すべきでしたが、これまではゼロ割で失敗することがありました。

- `max_expr_depth`パラメータを、スタックサイズをキロバイトで指定した`max_stack_depth`パラメータで置き換えました(Tom)

再帰的な関数の暴走によるクラッシュに対する防弾処理として提供される機能です。 式の入れ子の深さを単位にするのではなく、直接実行スタックのサイズを単位にするようになりました。

- 任意の行式の許可(Tom)

このリリースでは、SQL式が任意の複合型、つまり行値を持つことができます。 これにより、関数がより簡単に行を引数として取ることも行値を返すことができます。

- 行内の演算子や副問い合わせの比較として`LIKE/ILIKE`が使用できます(Fabien Coelho)
- 識別子やキーワード内の基本ASCII文字についてロケール固有の大文字小文字の変換を行わない(Tom)

`İ`と`ı`を含む単語が誤動作する"トルコ語問題"を解消します。 7ビットASCIIセット以外の文字の組み合わせは依然ロケールを無視します。

- 構文エラー報告の改良(Fabien, Tom)

構文エラーの報告が以前より使いやすくなりました。

- 実行した文に対応する完全なタグを返すよう`EXECUTE`を変更(Kris Jurka)

これまでのリリースでは、全ての`EXECUTE`呼び出しは`EXECUTE`タグを返していました。 このリリースでは、実行されたコマンドを反映したタグを返します。

- ルールの展開において`NATURAL CROSS JOIN`の発行を行わない(Tom)

こうした句は論理的な意味がありません。 しかし、ルールでコンパイラがこの構文を生成することがありました。

E.146.4.4. オブジェクト操作における変更

- キャスト、変換、言語、演算子クラス、ラージオブジェクトに対する`COMMENT ON`の追加(Christopher)

- デフォルトでOID付きのテーブルを作成するかどうかを制御する `default_with_oids` サーバ設定パラメータの新規追加(Neil)

これにより管理者は `CREATE TABLE` をデフォルトでOID列を持つテーブルを作成するか持たないテーブルを作成するかを制御することができます。(現在の出荷時でデフォルトでは、`default_with_oids` は `TRUE` ですが、将来のリリースではデフォルトで `FALSE` となります。)

- `CREATE TABLE AS` に `WITH / WITHOUT OIDS` 句を追加(Neil)
- `ALTER TABLE DROP COLUMN` でOID列を削除することが可能(`ALTER TABLE SET WITHOUT OIDS` もまだ動作します) (Tom)
- テーブル列として複合型が可能(Tom)

- SQL仕様通りの、`ALTER ... ADD COLUMN` でのデフォルトと `NOT NULL` 制約が可能(Rod)
NULLではなく指定されたデフォルト値を初期値として列を生成する `ADD COLUMN` が可能になりました。

- 列の型を変更する `ALTER COLUMN TYPE` が可能(Rod)
列を削除し、再度追加することなく、列のデータ型が変更できるようになりました。

- 複数の `ALTER` 操作を単一の `ALTER TABLE` コマンドにまとめることが可能(Rod)
これは特に、テーブルの書き換えを行う `ALTER` コマンド(`ALTER COLUMN TYPE` とデフォルト付きの `ADD COLUMN` があります)で便利です。 `ALTER` コマンドをまとめることにより、必要なテーブルの書き換えは1回だけになります。

- `SERIAL` 列を追加する `ALTER TABLE` が可能(Tom)
これは、新規列にデフォルトを付与する新機能の結果によるものです。
- 集約、変換、データベース、関数、演算子、演算子クラス、スキーマ、型、テーブル空間の所有者の変更が可能(Christopher, Euler Taveira de Oliveira)
これまでは、直接システムテーブルを変更する必要がありました。

- 一時オブジェクトの作成を `SECURITY DEFINER` 関数に制限することを許可(Sean Chittenden)
- `ALTER TABLE ... SET WITHOUT CLUSTER` を追加 (Christopher)
これまでのリリースでは、システムテーブルを変更する他、自動クラスタ指定を削除する方法がありませんでした。
- スキーマ内での一意性を保証するために制約/インデックス/`SERIAL` 名が番号付きの `table_column_type` になりました(Tom)

SQL仕様では、こうした名前はスキーマ内で一意であるべきと規定してます。

- `SERIAL` 列のシーケンス名を返す `pg_get_serial_sequence()` の追加(Christopher)
これにより自動スクリプトが確実に `SERIAL` シーケンス名を見つけることができます。
- プライマリ/外部キーのデータ型の不一致が高価な検索になる時の警告
- テーブル空間間でインデックスを移動することができる新しい `ALTER INDEX` コマンドの追加(Gavin)
- `ALTER TABLE OWNER` により依存するシーケンスの所有者も変更するようにした(Alvaro)

E.146.4.5. ユーティリティコマンドにおける変更

- `CREATE SCHEMA` によりトリガ、インデックス、シーケンスの作成が可能(Neil)
- `CREATE RULE` に `ALSO` キーワードを追加(Fabien Coelho)
`ALSO` がルール生成に追加できるようになりました。これは `INSTEAD` ルールと反対です。
- `LOCK` に `NOWAIT` オプションを追加(Tatsuo)
`LOCK` が、要求したロックを待機する必要がある場合に失敗することができるようになりました。
- `COPY` により、カンマ区切り値(CSV)ファイルの読み書きが可能(Andrew, Bruce)

- COPYの区切りとNULL文字列が競合する場合にエラーを生成(Bruce)
- GRANT/REVOKEの動作をよりSQL仕様に従うように変更
- CREATE INDEXとCHECKPOINT間のロック競合を防止(Tom)

7.3と7.4では、長時間実行されるB-treeインデックスの構築により同時実行されるCHECKPOINTの完了がブロックされることがありました。これにより、WALログの再利用を行うことができないためWALが膨れ上がることがありました。

- データベース全体のANALYZEはテーブルを跨ってロックを保持しない(Tom)

これは、テーブルの排他ロックを必要とする他のバックエンドがデッドロックする可能性を低減します。この変更をうまく使用するために、データベース全体に対するANALYZEをトランザクションブロック(BEGINブロック)内で実行しないでください。これは各テーブルに対してコミットされ新しいトランザクションが生成できなければなりません。

- REINDEXはインデックスの親テーブルの排他的ロックを全く取得しません。

インデックス自体はまだ排他的にロックされます。しかし構築中のインデックスを使用していない場合はテーブルの読み込みを続けることができます。

- ユーザ名を変更したときにMD5ユーザパスワードを消去(Bruce)

PostgreSQLは、MD5を使用してパスワードを暗号化する時にユーザ名をソルトとして使用します。ユーザ名が変更された場合、そのソルトは保存されたMD5パスワードと一致しませんので、過去のパスワードを使用することができなくなります。このリリースでは、注意が表示され、パスワードが削除されます。その後、そのユーザでパスワード認証を行う場合には、新しいパスワードを割り当てなければなりません。

- Windows用の新しいpg_ctl kill オプション(Andrew)

Windowsはバックエンドにシグナルを送信するためのkillコマンドがありません。そこでpg_ctlにこの機能が追加されました。

- 情報スキーマの改良
- GUIツールで初期パスワードを設定できるようにinitdbに--pwfileオプションを追加(Magnus)
- initdbにおけるロケール/符号化方式の不一致の検出 (Peter)
- Windowsオペレーティングシステムサービスに登録するためのregisterコマンドをpg_ctlに追加(Dave Page)

E.146.4.6. データ型および関数における変更

- より完全な複合型(行型)のサポート(Tom)

これまでスカラ型の値のみが動作していた多くの場所で複合型の値が使用できるようになりました。

- 非長方配列値をエラーとして拒絶(Joe)

これまではarray_inは何もメッセージを出さずにおかしな結果を構築しました。

- 整数算術計算におけるオーバーフローの検出(Tom)
- 単一バイトの"char"データ型に関する算術演算子の削除

これまでは、パーサは、`null * null`など"unable to select an operator"エラーとなることがより適切である多くの場合で、こうした演算子を選択していました。実際に"char"列に対する算術演算を行いたいのであれば、明示的に整数型にキャストしてください。

- 配列への入力における構文検査をできるだけ厳密化(Joe)

これまでおかしな場所に許され、おかしな結果を生成したごみがERRORになるようになりました。例えば、右閉じ括弧の後の空白以外の文字。

- 空文字列配列要素値は何も記述しないのではなく""と記述しなければならない(Joe)

これまでは、どちらの方法でも空文字列要素値を記述できましたが、引用符付きの文字列のみが有効になりました。何も指定しないところは、おそらく将来のリリースではNULL要素値とみなされる予定です。

- 配列要素の終わりの空白を無視(Joe)

これまでは要素値と区切り文字もしくは右括弧の間で、前方の空白は無視され、後方の空白は意味がありました。現在は、後方の空白も無視されます。

- 下限が1でない場合、明示的な配列境界付きで配列値を発行 (Joe)
- 日付文字列としてYYYY-monthname-DDを受け付ける (Tom)
- netmaskとhostmask関数が最大のマスク長を返す (Tom)
- 階乗関数がnumericを返すように変更 (Gavin)

numericを返すことにより、階乗関数はより広範な入力値で動作するようになりました。

- to_char/to_date()による日付変換の改良 (Kurt Roeckx, Fabien Coelho)
- CHAR(n)においてlength()が最後の空白を考慮しないようにした (Gavin)

この変更は整合性を高めるために行われました。CHAR(n)データにおける最後に続く空白は重要な意味を持ちません。ですので、これらをlength()で数えるべきではありません。

- OID/float4/float8データ型に空文字列を渡した場合の警告 (Neil)

8.1ではエラーになる予定です。

- int2/int4/int8/float4/float8入力関数で値の前後の空白文字を許可 (Neil)
- float4, float8におけるIEEE InfinityとNaNのサポートの改良 (Neil)

IEEE準拠の浮動小数点演算をサポートする全てのプラットフォームで動作するようになりました。

- date_trunc()にweekオプションの追加 (Robert Creager)
- 1 BCに対するto_charの修正 (以前は1 ADを返しました) (Bruce)
- BC日付用のdate_part(year)を修正 (以前は正しい年から1引いた値を返していました) (Bruce)
- date_part()が適切な千年紀と世紀を返すように修正 (Fabien Coelho)

これまでのバージョンでは、世紀と千年紀は、標準的な計算方法と比べて、間違った値を返し、また、間違った年から数えていました。

- 標準に従うように、ceil()の別名ceiling()、pow()の別名power()を追加 (Neil)
- 特定のエラー条件でSQL:2003で指定された正しいSQLSTATEエラーコードを発行するようにln()、log()、power()、sqrt()を変更 (Neil)
- SQL:2003の定義通りのwidth_bucket()関数の追加 (Neil)
- 数値集合への操作を簡略化するgenerate_series()関数の追加 (Joe)

- マルチバイト符号化方式を扱うupper/lower/initcap()関数の修正 (Tom)
- booleanおよびビット単位の整数に対するAND/OR集約の追加 (Fabien Coelho)
- クライアント、サーバのネットワークアドレスを返すセッション情報関数の追加 (Sean Chittenden)
- 閉経路の領域を求める関数の追加 (Sean Chittenden)
- 他のバックエンドにキャンセル要求を送信する関数の追加 (Magnus)
- intervalにdatetimeを加算する演算子 (Tom)

順番が反対のdatetimeにintervalを加算するほうは既にサポートされていましたが、標準SQLでは両方とも必要です。

- 整数からBIT(N)へのキャストにて、整数の右側からNビットを選択 (Tom)

これまでのリリースでは、左から数えてNビットが選択されていましたが、いうまでもありませんが、BITからintへのキャスト時に整合性がないため、役に立つものとは思われません。

- CIDR値の非マスクビットは0でなければなりません (Kevin Brintnall)

E.146.4.7. サーバサイド言語における変更

- `READ COMMITTED` シリアライゼーションモードにて、`volatile`関数は、その関数の呼び出しに対応するコマンドの始まるまでではなく、その関数内の各行が始まるまでに同時実行トランザクションがコミットした結果を参照するようになりました。
- `STABLE`または`IMMUTABLE`と宣言された関数は常に問い合わせ呼び出し時のスナップショットを使用します。従って、問い合わせ呼び出し開始後に行われた操作は、それが自身のトランザクション内であろうと他のトランザクション内であろうと、その影響を受けません。関数は読み取りのみでなければなりません。つまり、`SELECT`以外のSQLコマンドを使用することはできません。関数を`VOLATILE`で宣言するのではなく`STABLE`または`IMMUTABLE`で宣言することで、かなりの性能向上があります。
- 非遅延の`AFTER`トリガは、対応するコマンドの完了時ではなく、トリガを発行した問い合わせの完了直後に発行されるようになりました。関数内にある問い合わせがトリガを発行する時に違いが現れます。トリガは関数が次の操作に進む前に実行されます。例えば、関数がテーブルに新規行を挿入する場合、関数の処理が進む前に非遅延の外部キー検査が行われます。
- 名前付きで宣言した関数パラメータの許可(Dennis Björklund)
関数の文書化を改善することができます。この名前を実際にどう使用できるかは使用する関数言語に依存します。
- PL/pgSQLパラメータ名による関数内での参照を許可(Dennis Björklund)
これは基本的に、名前付きパラメータに対する別名を自動的に作成します。
- PL/pgSQL生成時の最低限の構文検査の実施(Tom)
これにより、より早く構文エラーを検出することができます。
- PL/pgSQLにおける複合型(行変数とレコード変数)のサポートの強化
例えば、関数に単一の値として行型の値を渡すことができるようになりました。
- 前もって宣言された変数を参照するPL/pgSQL変数のデフォルト値の許可
- PL/pgSQL `FOR`ループの解析の改良(Tom)
`FOR`変数のデータ型ではなく`".."`の存在によって解析が開始されるようになりました。これは、正常な関数では違いはありませんが、結果として失敗があった時のエラーメッセージがより判りやすくなります。
- PL/Perlサーバサイド言語の大規模な改良(Command Prompt, Andrew Dunstan)
- PL/Tclにおいて、SPIコマンドが副トランザクション内で実行されるようになりました。エラーが発生した場合、副トランザクションは消え、通常のTclエラーと同様にエラーが報告されます。このエラーは`catch`で捕捉可能です。これまでは、こうしたエラーを捕捉できませんでした。
- PL/pgSQLで`ELSEIF`を受け付けるようになりました。(Neil)
これまではPL/pgSQLでは`ELSIF`しか解釈できませんでした。しかし、多くのユーザはこの`ELSEIF`キーワードという綴りに慣れています。

E.146.4.8. psqlにおける変更

- データベースオブジェクトに関するpsqlの情報表示の改良(Christopher)
- psqlの`%du`と`%dg`におけるグループメンバシップの表示を許可(Markus Bertheau)
- psqlの`%dn`による一時スキーマを表示を防止(Bruce)
- psqlのファイル名の扱いにてチルダユーザ展開を許可(Zach Irmen)
- readline経由、配色を含む、psqlの装飾付きプロンプトの表示を許可(Reece Hart, Chet Ramey)
- psqlの`%copy`を`COPY`コマンドの構文に完全に一致するように変更(Tom)
- 構文エラーの場所を表示(Fabien Coelho, Tom)
- psqlの`%d`表示に`CLUSTER`情報を追加 (Bruce)

- `psql`の`%copy stdin/stdout`をコマンドの入出力を使用するように変更(Bruce)
- `psql`の`stdin/stdout`から読み取る`pstdin/pstdout`を追加(Mark Feit)
- グローバルな`psql`設定ファイル`psqlrc.sample`の追加 (Bruce)
これにより、全体的な`psql`起動コマンドを格納できる中核的なファイルを持つことができます。
- テーブルが`OID`列を持つ場合の`psql %d+`に指示子を持たせる(Neil)
- Windowsでは、Control-Zがend-of-fileと取られないように`psql`内でバイナリモードを使用する
- `%dn+`でスキーマの権限と説明を表示させる(Dennis Björklund)
- タブ補完サポートの改良(Stefan Kaltenbrunn, Greg Sabino Mullane)
- 論理値を大文字でも小文字でも設定可能にした(Michael Paesold)

E.146.4.9. `pg_dump`における変更

- `pg_dump`の信頼性を向上させるために依存性情報を使用(Tom)
これにより、関連するオブジェクトが稀に間違った順序でダンプされるという長い間存在した問題が解決したはずです。
- 可能な限りアルファベット順で`pg_dump`にオブジェクトの出力を行う(Tom)
これによりダンプファイル間の変更を簡単に区別できるようになるはずです。
- `pg_restore`におけるいくつかのSQLエラーを無視(Fabien Coelho)
これにより、`pg_restore`の動作は、`pg_dump`の出力スクリプトを`psql`に渡した時の動作と同様になりました。ほとんどの場合、エラーを無視して先に進めることは使いやすくなるでしょう。また、エラー時に終了する旧来の動作を指定するために`pg_restore`にオプションが追加されました。
- `pg_restore -l`で、含有するオブジェクトのスキーマ名を表示
- `pg_dump`のテキスト出力にbegin/end印を追加(Bruce)
- `pg_dump/pg_dumpall`の冗長モードで開始時刻、終了時刻を追加(Bruce)
- `pg_dumpall`で`pg_dump`のほとんどのオプションを許可(Christopher)
- `pg_dump`はデフォルトで`SET SESSION AUTHORIZATION`ではなく`ALTER OWNER`を使用する(Christopher)

E.146.4.10. `libpq`における変更

- `libpq`の`SIGPIPE`例外処理をスレッドセーフに変更(Bruce)
- 表示用の文字長を返す`PQmbdsplen()`の追加(Tatsuo)
- SSLとKerberos接続をロックするスレッドを追加(Manfred Spraul)
- `PQoidValue()`、`PQcmdTuples()`、および`PQoidStatus()`が`EXECUTE`コマンドで動作可能(Neil)
- サーバのバージョン番号に、より簡単なアクセスを提供する`PQserverVersion()`の追加
- パラメータのデータ型を指定する必要がある文の準備をサポートする`PQprepare/PQsendPrepared()`の追加(Abhijit Menon-Sen)
- `SET DESCRIPTOR`など、多くのECPGの改良(Michael)

E.146.4.11. ソースコードにおける変更

- データベースサーバがWindows上でネイティブに動作可能(Claudio, Magnus, Andrew)
- WindowsをサポートするためにシェルスクリプトコマンドをC言語版に変換(Andrew)
- 拡張メークファイル基盤の作成(Fabien Coelho, Peter)

元のソースツリーの外部で拡張を構築する作業を簡単にします。

- 再割当て可能なインストレーションのサポート(Bruce)

再設定や再構築を行わずにインストレーションのツリーを別の場所に移動できるように、インストールされるファイルのディレクトリパス(/shareディレクトリなど)が実行ファイルの実際のパスから相対的に計算されるようになりました。

- 文書のインストール先を指定するための`--with-docdir`の使用。`--infodir`も可能(Peter)
- 文書のインストールを防止する`--without-docdir`の追加(Peter)
- DocBook V4.2 SGMLへの更新 (Peter)
- 新しいPostgreSQL CVSタグ(Marc)

PostgreSQLのCVSレポジトリの独自のコピーを管理する組織でより簡単に行うことができるようにするために変更されました。マスタリポジトリのファイルバージョンスタンプが、コピーされたレポジトリへのチェックイン、チェックアウトによって破壊されないようになります。

- ロック処理の明確化(Manfred Koizar)
- バッファ管理の整理(Neil)
- プラットフォーム試験からCPUスピンロックコードを分離(Bruce, Tom)
- PA-RISCのgcc用のインラインのtest-and-setコードを追加(ViSolve, Tom)
- i386スピンロックコードの改良(Manfred Spraul)
- 最近のgccリリースでの警告を防ぐためのスピンロックアセンブルコードの整理(Tom)
- ソースツリーからJDBCを削除。別のプロジェクトになりました。
- ソースツリーからlibpgtclを削除。別のプロジェクトになりました。
- メモリとファイル記述子の使用状況の推定を高精度化(Tom)
- Mac OS X用の起動スクリプトの改良(Ray A.)
- 新しい`fsync()` 試験プログラム(Bruce)
- 文書の大規模な改良(Neil, Peter)
- `pg_encoding`の削除。もはや不要
- `pg_id`の削除。もはや不要
- `initlocation`の削除。もはや不要
- スレッドフラグの自動検出(手作業での試験は不要) (Bruce)
- Olsonのパブリックドメインのtimezoneライブラリの使用(Magnus)
- スレッド処理が有効であれば、Unixwareではバックエンド実行形式同様にスレッドフラグを使用(Bruce)

Unixwareでは、スレッド化されたオブジェクトとスレッド化されないオブジェクトを混在させることができません。そのため全てスレッド化されるものとしてコンパイルされなければなりません。

- コマンド文字列の解析にpsqlはflexが生成する字句解析を使用
- バックエンド全体で使用されるリンクリストデータ構造を再実装(Neil)

リストへの追加ができることにより性能が向上します。また、length操作がより効率的になります。

- 動的にロードされるモジュールが独自のサーバ設定パラメータを生成可能(Thomas Hallgren)
- ブラジル語によるFAQの追加(Euler Taveira de Oliveira)
- フランス語によるFAQの追加(Guillaume Lelarge)
- Windowsのログ処理用のpgeventを追加

- OS Xにおいて、libpqとECPGが適切な共有ライブラリとして構築可能(Tom)

E.146.4.12. Contribにおける変更

- contrib/dblinkの修正(Joe)
- contrib/dbmirrorの改良(Steven Singer)
- contrib/xml2の追加(John Gray, Torchbox)
- contrib/mysqlの更新
- 新バージョンのcontrib/btree_gist (Teodor)
- PostgreSQL用trigramマッチング、contrib/trgmの追加。(Teodor)
- contrib/tsearch2の多くの改良(Teodor)
- contrib/fuzzystrmatchに二重metaphoneを追加(Andrew)
- contrib/pg_autovacuumがWindowsサービスとして動作可能(Dave Page)
- contrib/dbsizeに機能追加(Andreas Pflug)
- contrib/pg_loggerの削除。 ログ取得副プロセスへの統合によって廃止されました。
- contrib/rservの削除。 各種の別プロジェクトにより廃棄されました。

[前のページ](#)
リリース8.0.1

[ホーム](#)
[上に戻る](#)

[次のページ](#)
リリース7.4.30

E.147. リリース7.4.30

リリース日: 2010-10-04

このリリースは7.4.29に対し、各種の不具合を修正したものです。7.4メジャーリリースにおける新機能については[項E.177](#)を参照してください。

これはPostgreSQL 7.4.X系列の最終リリースとなるはずですが。ユーザにはより新しい系列のリリースへの更新を強く勧めます。

E.147.1. バージョン7.4.30への移行

7.4.Xからの移行ではダンプ/リストアは不要です。しかし7.4.26以前のバージョンからアップデートする場合は7.4.26のリリースノートを参照してください。

E.147.2. 変更点

- PL/PerlおよびPL/Tclにおいて、呼び出し元のSQLユーザIDごとに別のインタプリタを使用します。(Tom Lane)

この変更は、同じセッション内でその後に異なるSQLユーザIDにより実行されるPerlまたはTclコード（例えばSECURITY DEFINERの内部）を破壊することで発生する可能性があるというセキュリティ問題を防止するものです。ほとんどのスクリプト用言語は、対象とする関数で呼び出される標準関数や演算子の再定義など、これを行うことができる多くの方法を提供します。この変更がないと、PerlまたはTcl言語の使用権限を持つ任意のSQLユーザは基本的に、対象の関数の所有者が持つSQL権限でできることを何でもできるようになります。

この変更の欠点はPerlおよびTcl関数間の内部通信がより難しくなるという点です。逃げ道を提供するために、PL/PerlUおよびPL/TclU関数ではまだセッション当たり1つのインタプリタのみを使用します。これらのすべての関数は従来よりデータベーススーパーユーザの信頼レベルで実行されますので、これはセキュリティ問題とはみなされません。

信頼された実行を提供することを主張するサードパーティ製の手続き言語は似たようなセキュリティ問題を持つ可能性が大きくあります。セキュリティが非常に重要な目的のために依存している手続き言語があれば、その作者に連絡を取ることを勧めます。

この問題を指摘いただいたTim Bunceに感謝します(CVE-2010-3433)。

- pg_get_expr() の用途として意図されたシステムカタログ列の一つではない引数で呼び出されることを拒絶することでpg_get_expr() 内のクラッシュする可能性を防止します。(Heikki Linnakangas, Tom Lane)
- "cannot handle unplanned sub-select"エラーを修正しました。(Tom Lane)

これは、副選択が他の副選択を含む式に展開される結合の別名参照を含む場合に発生しました。

- ロックファイル (postmaster.pidおよびソケット用のロックファイル) を書き出す時に、注意してその内容のfsyncします。(Tom Lane)

これがないと、postmasterの起動後しばらくしてマシンがクラッシュした場合に、ロックファイルの内容が破損することがあり得ました。これが起きると、このロックファイルを手作業で削除するまで、その後のpostmasterの起動試行が成功できなくなります。

- 削除された列を含むテーブルに対するcontrib/dblinkの処理を改良しました。(Tom Lane)
- contrib/dblinkにおいて"duplicate connection name"エラー後の接続リークを修正しました。(Itagaki Takahiro)
- 構築手順や文書を更新して、ソースコードリポジトリがCVSからGitに変更されたことを反映しました。(Magnus Hagander他)

E.148. リリース7.4.29

リリース日: 2010-05-17

このリリースは7.4.28に対し、各種の不具合を修正したものです。 7.4メジャーリリースにおける新機能については[項E.177](#)を参照してください。

PostgreSQLコミュニティは2010年7月に7.4.X系のリリースの更新版の公開を停止します。 ユーザには新しいリリース版まで更新することを奨励します。

E.148.1. バージョン7.4.29への移行

7.4.Xからの移行ではダンプ/リストアは不要です。 しかし7.4.26以前のバージョンからアップデートする場合は7.4.26のリリースノートを参照してください。

E.148.2. 変更点

- Safe.pmを使用するのではなく、インタプリタ全体に適用されるopmaskを使用してplperlを強制的に制限します。(Tim Bunce, Andrew Dunstan)

最近の開発によりplperlを信頼できるとする根拠とするにはSafe.pmが非常に安全ではないことを知らされました。 この変更により、Safe.pmの使用がなくなり、常に適用されるopcodeマスクを持つ別のインタプリタを使用するようになります。 この変更の喜ばしい副作用として、plperl内で普通の方法でPerlのstrict プラグマが使用できるようになりました。 また、Perlの\$aと\$bがソート関数で想定される動作を行うようになり、関数のコンパイルが非常に高速になりました。(CVE-2010-1169)

- PL/Tclにおいてpltcl_modules経由で信頼できないコードが実行されることを防止しました。(Tom)

PL/TclのデータベーステーブルからTclコードを自動読み込みする機能はトロイの木馬攻撃によりつけこまれる可能性があります。 そのテーブルの作成や挿入のできるユーザに対する制限がなかったためです。 この変更により、pltcl_modulesがスーパーユーザにより所有されていない限り、この機能は無効になります。(しかし、このテーブルに対する権限は検査されませんので、安全性が低いモジュールを本当に必要とするインストレーションでは、信頼された非スーパーユーザに適切な権限を付与することができます。) また、 pltclu関数を本当に実行しようとしていない限り、制限がない"通常"のTclにコードが読み込まれることを防止しました。(CVE-2010-1170)

- 非特権ユーザがスーパーユーザのみのパラメータ設定を再設定することを許しません。(Alvaro)

これまで、非特権ユーザが自身に対するALTER USER ... RESET ALLまたは自身が所有するデータベースに対するALTER DATABASE ... RESET ALLを行った場合、そのユーザまたはデータベースに対する特別なパラメータ設定は、たとえスーパーユーザのみにより変更可能なものとして提供されるものであっても、すべて削除されました。 ALTERはそのユーザが変更権限を持つパラメータのみを削除するようになりました。

- CONTEXT追加がログ項目になされる時にシャットダウンが発生した場合に、バックエンドのシャットダウン時にクラッシュする可能性を防ぎました。(Tom)

現在のトランザクションがログメッセージを出力する時点に達した時にすでにロールバックされているために文脈出力関数が失敗することがありました。

- pl/perlのppport.hを最近のPerlバージョンに更新しました。(Andrew)
- pl/python内の雑多なメモリリークを修正しました。(Andreas Freund, Tom)
- contrib/pgstattuple関数がキャンセル割り込みに即座に応答することを確実にしました。(Tatsuhito Kasahara)
- サーバ起動時に、既存の共有メモリセグメントに対してshmget() がEINVALを返す状況を適切に扱うようにしました。(Tom)

この振舞いはOS Xを含むBSD派生のカーネルにおいて観測されました。 共有メモリ要求サイズが大きすぎるという警告の、完全に間違った起動失敗という結果になりました。

E.149. リリース7.4.28

リリース日: 2010-03-15

このリリースは7.4.27に対し、各種の不具合を修正したものです。 7.4メジャーリリースにおける新機能については[項E.177](#)を参照してください。

PostgreSQLコミュニティは2010年7月に7.4.X系のリリースの更新版の公開を停止します。 ユーザには新しいリリース版まで更新することを奨励します。

E.149.1. バージョン7.4.28への移行

7.4.Xからの移行ではダンプ/リストアは不要です。 しかし7.4.26以前のバージョンからアップデートする場合は7.4.26のリリースノートを参照してください。

E.149.2. 変更点

- SSL接続においてセッションキーの再調停をどの程度の頻度で行うかを制御する`ssl_renegotiation_limit`設定パラメータを新しく追加しました。(Magnus)

これをゼロに設定することで、再調停を完全に無効にすることができます。 これは壊れたSSLライブラリを使用している場合に必要になる可能性があります。 具体的には一部のベンダは再調停の試行を失敗させるCVE-2009-3555に対する一時しのぎのパッチを提供しています。

- `bit`型の`substring()`では、いかなる負の長さも"文字列の残りすべて"を意味するようにしました。(Tom)

これまでのコードでは-1のみを上のように扱い、他の負の値についてはクラッシュする可能性がある無効な結果値を生成しました。(CVE-2010-0442)

- 正規表現マッチが異常なまでに低速になる一部の状況を修正しました。(Tom)

- `pg_hba.conf`とその関連ファイルを読み取る時、`@`が引用符記号の内部にある場合に`@something`をファイル包含要求と扱わないようにしました。 また、`@`自身をファイル包含要求として扱いません。(Tom)

ロールまたはデータベース名が`@`から始まる場合の一定しない動作を防止します。 空白文字を含むパス名のファイルを包含する必要がある場合、まだ実現させることはできませんが、式全体を引用符でくくるのではなく`@"/path to/file"`と記述しなければなりません。

- ディレクトリが`pg_hba.conf`とその関連ファイル内で含有対象と指定されていた場合の一部のプラットフォームにおける無限ループを防止します。(Tom)

- PL/Tclは確実にTclインタプリタを完全に初期化します。(Tom)

このミスの既知の兆候は、Tcl 8.5以降を使用している場合のTclの`clock`コマンドの誤動作だけです。

- `dblink_build_sql_*`関数であまりに多くのキー列が指定された場合の`contrib/dblink`のクラッシュを防止します。(Rushabh Lathia, Joe Conway)

E.150. リリース7.4.27

リリース日: 2009-12-14

このリリースは7.4.26に対し、各種の不具合を修正したものです。 7.4メジャーリリースにおける新機能については[項E.177](#)を参照してください。

E.150.1. バージョン7.4.27への移行

7.4.Xからの移行ではダンプ/リストアは不要です。 しかし7.4.26以前のバージョンからアップデートする場合は7.4.26のリリースノートを参照してください。

E.150.2. 変更点

- セッション固有の状態を変更するインデックス関数により引き起こされる間接的なセキュリティ脅威から保護します。(Gurjeet Singh, Tom)

この変更により不変インデックス関数がスーパーユーザのセッションを破壊する可能性(CVE-2009-4136)を防ぎます。

- ヌルバイトが埋め込まれたコモンネーム (CN) フィールドをもつSSL証明書を拒絶します。(Magnus)

これは、SSL検証においてサーバまたはクライアント名に対する証明書が不正に一致しないようにします。(CVE-2009-4034)

- バックエンド起動時のキャッシュ初期化の間でクラッシュする可能性を修正しました。(Tom)

- 安全ではない時点でのVACUUMを中断するシグナルを防止します。(Alvaro)

この修正により、VACUUM FULLがすでにタプルの移動をコミットした後にキャンセルされた場合のパニックや、通常のVACUUMがテーブルを切り詰めた後に割り込みを受けた場合の一時的なエラーを防止します。

- ハッシュテーブルサイズの計算における整数オーバーフローによるクラッシュの可能性を修正しました。(Tom)

これは、ハッシュ結合の結果のプランナの推定サイズが極端に大きい場合に発生することがありました。

- inet/cidrの比較における非常にまれなクラッシュを修正しました。(Chris Mikkelsen)

- PAMパスワード処理をより堅牢に修正しました。(Tom)

以前のコードでは、Linuxのpam_krb5 PAMモジュールとドメインコントローラとしてのMicrosoft Active Directoryを組み合わせると失敗することが知られていました。 PAMスタックに渡される引数に関する正当ではない仮定を作成することであるため、どこかで問題になるかもしれません。

- 今後のバージョンのlibpqとの互換性を高めるために、postmasterが接続要求パケットにおけるapplication_nameパラメータをすべて無視するようにしました。(Tom)

E.152. リリース7.4.25

リリース日: 2009-03-16

このリリースは7.4.24に対し、各種の不具合を修正したものです。 7.4メジャーリリースにおける新機能については[項E.177](#)を参照してください。

E.152.1. バージョン7.4.25への移行

7.4.Xからの移行ではダンプ/リストアは不要です。 しかし、7.4.11より前のバージョンからアップグレードする場合は、7.4.11のリリースノートを参照してください。

E.152.2. 変更点

- 符号化方式の変換に失敗した時、再帰的なエラーによるクラッシュを防止しました。(Tom)

この変更は、最近の2つのマイナーリリースにて関連する失敗の状況に対してなされた改修を拡張したものです。 前回の修正は元の問題報告に特化したものでしたが、符号化方式変換関数で発生するすべてのエラーがそのエラーを報告しようとして、無限に再帰される可能性があることが分かりました。 したがって、再帰的なエラー報告を行う状況になったことがわかった時の解決策は、変換と符号化方式を無効にし、通常のASCII形式のエラーメッセージで報告することです。(CVE-2009-0922)

- 特定の変換関数に対する、間違った符号化方式を用いたCREATE CONVERSIONを許可しません。(Heikki)

これにより、符号化方式に関する失敗における、あり得る状況を防止します。 前回の変更は、同じ問題における別の種類の失敗に対する防止策でした。

- `to_char()` に引数のデータに対して不適切な整形用のコードが渡された時のコアダンプを修正しました。(Tom)
- デフォルトの既知の時間帯省略形のリストにMUST (モーリシャス島夏時間) を追加しました。(Xavier Bugaud)

E.153. リリース7.4.24

リリース日: 2009-02-02

このリリースは7.4.23に対し、各種の不具合を修正したものです。 7.4メジャーリリースにおける新機能については[項E.177](#)を参照してください。

E.153.1. バージョン7.4.24への移行

7.4.Xからの移行ではダンプ/リストアは不要です。 しかし、7.4.11より前のバージョンからアップグレードする場合は、7.4.11のリリースノートを参照してください。

E.153.2. 変更点

- `headline()` 関数におけるURLの扱いを改良しました。(Teodor)
- `headline()` 関数における延長見出しの扱いを改良しました。(Teodor)
- 符号化方式の変換が符号化方式の組み合わせの指定を間違えた変換関数で作成された場合にあり得る、アサートの失敗や誤変換を防止します。(Tom, Heikki)
- `VACUUM`において小規模なテーブルに対する不要なロックを防止しました。(Heikki)
- `contrib/tsearch2`の`get_covers()` 関数における未初期化変数を修正しました。(Teodor)
- `to_char()` の`TH`書式を扱うコードの不具合を修正しました。(Andreas Scherbaum)
- すべての文書において、現在活動していない`pgsql-ports`と`pgsql-patches`メーリングリストの代わりに、`pgsql-bugs`または`pgsql-hackers`を参照するようにしました。

E.154. リリース7.4.23

リリース日: 2008-11-03

このリリースは7.4.22に対し、各種の不具合を修正したものです。 7.4メジャーリリースにおける新機能については[項E.177](#)を参照してください。

E.154.1. バージョン7.4.23への移行

7.4.Xからの移行ではダンプ/リストアは不要です。 しかし、7.4.11より前のバージョンからアップグレードする場合は、7.4.11のリリースノートを参照してください。

E.154.2. 変更点

- クライアントの符号化方式が翻訳されたエラーメッセージを表すことができない時のバックエンドのクラッシュを修正しました。(Tom)

以前にも同様の問題に対応しましたが、"等価な文字を持たない"メッセージ自体が変換されない場合にまだ失敗しました。 今回の修正では、こうした状況を検知した時、翻訳を無効にし、通常のASCIIエラーメッセージを送信します。

- 1つの問い合わせ項目がテキストの先頭の単語に一致した場合、テキスト検索の間違った見出し生成を修正しました。(Sushant Sinha)
- `--enable-integer-datetimes` で構築された場合、ISO以外の日付書式を使用した時の時間間隔値における不適切な秒端数表示を修正しました。(Ron Mayer)
- 渡されたタプルとタプル記述子が異なる列数を持っていた場合に `SPI_getvalue` および `SPI_getbinval` が適切に動作することを確実にしました。(Tom)

テーブルに列が追加または削除された場合、こうした状況は正常です。 しかしこの2つの関数は適切に扱っていませんでした。 起こり得る唯一の結果は間違ったエラーが示されることです。

- `ecpg` の `CREATE USER` 解析を修正しました。(Michael)

E.155. リリース7.4.22

リリース日: 2008-09-22

このリリースは7.4.21に対し、各種の不具合を修正したものです。 7.4メジャーリリースにおける新機能については[項E.177](#)を参照してください。

E.155.1. バージョン7.4.22への移行

7.4.Xからの移行ではダンプ/リストアは不要です。 しかし、7.4.11より前のバージョンからアップグレードする場合は、7.4.11のリリースノートを参照してください。

E.155.2. 変更点

- 64ビットプラットフォームで稼働している場合、日付時刻入力関数が正しく整数オーバーフローを検知できるよう修正しました。(Tom)
- 非常に長いログメッセージをsyslogに書き出す際の性能を向上しました。(Tom)
- `SELECT DISTINCT ON`問い合わせに対するカーソルの後方スキャンにおける不具合を修正しました。(Tom)
- プランナが論理型の結果を生成する`GROUP BY`式を、式の内容に関係なく、常に2つのグループになると推定することを修正しました。(Tom)
これにより、`col IS NULL`などの特定の論理試験に対して、通常の`GROUP BY`推定より非常により良く精度が上がります。
- `pg_dump`および`pg_restore`がSQLコマンドの送信に失敗した後に報告するエラーを改良しました。(Tom)

E.156. リリース7.4.21

リリース日: 2008-06-12

このリリースは7.4.20に対し、重大な1つの不具合と軽微な1つの不具合を修正したものです。7.4メジャーリリースにおける新機能については[項E.177](#)を参照してください。

E.156.1. バージョン7.4.21への移行

7.4.Xからの移行ではダンプ/リストアは不要です。しかし、7.4.11より前のバージョンからアップグレードする場合は、7.4.11のリリースノートを参照してください。

E.156.2. 変更点

- `pg_get_ruledef()` で負の定数を括弧でくるようにしました。(Tom)

この修正前では、ビューまたはルールにおける負の定数がそのまま、例えば`-42::integer`のようにダンプされる可能性があります。これは若干正しくありません。演算子の前置規則に従うと、`(-42)::integer`となるべきです。通常は大した違いが起こることはありませんが、最近の別のパッチと相互作用して、PostgreSQLが有効な`SELECT DISTINCT`ビューの問い合わせを拒絶することになりました。pg_dumpの出力のリロード失敗という結果になることもあり得たため、これは高優先度の修正として扱われました。実際にダンプ出力が正しくなかったバージョンは8.3.1と8.2.7だけです。

E.157. リリース7.4.20

リリース日: リリースされませんでした

このリリースには7.4.19からの各種修正が含まれています。 7.4メジャーリリースにおける新機能については[項E.177](#)を参照してください。

E.157.1. バージョン7.4.20への移行

7.4.Xからの移行ではダンプ/リストアは不要です。 しかし、7.4.11より前のバージョンからアップグレードする場合は、7.4.11のリリースノートを参照してください。

E.157.2. 変更点

- キリル語の"Yo"文字 (2つのドットを持つёとЕ) を扱うISO-8859-5と他の符号化方式との間の変換を修正しました。(Sergey Burladyan)
- 複数のデータ型入力関数、特に、結果に初期化されず予測できない値を含む未使用のバイトを許す`array_in()`を修正しました。(Tom)

これは、2つの明らかに同じリテラル値が等しいとみなされず、結果パーサは一致しない`ORDER BY`および`DISTINCT`式に関して不平をいうといったエラーをもたらします。

- 正規表現副文字列一致 (`substring(string from pattern)`) における境界状態を修正しました。(Tom)

この問題は、パターン全体は一致するが、ユーザが括弧付きの副式を指定し、その副式が一致しなかった時に発生します。例えば、`substring('foo' from 'foo(bar)?')`です。(bar) は一致しませんので、これはNULLを返すべきです。しかし、間違ってパターン全体が一致 (つまりfoo) という結果を返していました。

- `ecpg`の`PGTYPEStimestamp_sub()`関数からの不正確な結果を修正しました。(Michael)
- `DatumGetBool`マクロがgcc 4.3でエラーにならないよう修正しました。(Tom)

この問題は、論理値を返す"古い形式" (V0)のC関数に影響します。これは8.3ではすでに修正されていましたが、当時なかったバックパッチが必要になりました。

- 長い間存在した`LISTEN/NOTIFY`の競合条件を修正しました。(Tom)

まれな状況において、`NOTIFY`を実行する同時実行のトランザクションが後にコミットされたことが観測されたために想定されたとしても、`LISTEN`だけを実行するセッションが注意を受け取らない可能性があります。

この修正の副作用として、`LISTEN`コマンドをまだコミットしていないトランザクションでは`LISTEN`用の`pg_listener`内の行を参照しなくなります。これは検索するために選択すべきもので、これまではできていました。この動作はできるともできないとも文書化されていませんが、一部のアプリケーションで古い動作に依存している可能性があります。

- `ORDER BY`および`GROUP BY`内の定数式の表示を修正しました。(Tom)

明示的にキャストされた定数が不正確に表示されました。これは例えば、ダンプ/リストア時にビュー定義を壊す可能性があります。

- `COPY OUT`時に`NOTICE`メッセージを正しく扱うよう`libpq`を修正しました。(Tom)

このエラーは、ユーザ定義のデータ型出力ルーチンが`NOTICE`を発行する時に起こることのみが観測されていました。しかし、他が原因で発生していないという保証はありません。

E.158. リリース7.4.19

リリース日: 2008-01-07

このリリースは、重大なセキュリティ問題の修正を含む、7.4.18の各種不具合を修正したものです。7.4メジャーリリースにおける新機能については[項E.177](#)を参照してください。

E.158.1. バージョン7.4.19への移行

7.4.Xからの移行ではダンプ/リストアは不要です。しかし、7.4.11より前のバージョンからアップグレードする場合は、7.4.11のリリースノートを参照してください。

E.158.2. 変更点

- インデックス内の関数がVACUUM、ANALYZEなどを実行するユーザの権限で実行されることを防止します。(Tom)

インデックス式内で使用される関数および部分インデックスの述語は、新しいテーブル項目が作成されると評価されます。これが、もし信頼できないユーザが所有するテーブルを変更したとすると、トロイの木馬となるコードの実行という危険性を引き起こすことは以前から解っていました。(トリガ、デフォルト、検査制約なども同種の危険性を引き起こします。)しかしインデックス内の関数は、VACUUM FULLなどの日常の保守作業で実行される可能性があるため、さらに危険です。こうした保守作業は通常自動的にスーパーユーザ権限で行われます。例えば、極悪なユーザがトロイの木馬となるインデックス定義を設定し、次の日常のバキューム処理を待つことで、スーパーユーザ権限でコードを実行することが可能です。この修正は、SECURITY DEFINER関数で使用されていた権限変更機構を使用して、標準的な保守作業(VACUUM、ANALYZE、REINDEX、CLUSTERを含む)が呼び出したユーザではなくテーブル所有者として実行するように調整しました。こうしたセキュリティ制限の迂回を防ぐために、SET SESSION AUTHORIZATIONおよびSET ROLEがSECURITY DEFINERコンテキストで禁止されるようになりました。(CVE-2007-6600)

- 正規表現パッケージ内のさまざまな不具合を修正しました。(Tom, Will Drewry)

より巧妙な正規表現パターンにより、信頼できないソースから正規表現検索パターンを受け付けるアプリケーションでは、クラッシュ、無限または無限に近いループ、大量のメモリの浪費、DoS攻撃を引き起こすすべてが発生する可能性があります。(CVE-2007-4769、CVE-2007-4772、CVE-2007-6067)

- /contrib/dblinkを使用して、パスワード認証のみをセキュリティ制限として使用するには非スーパーユーザでなければなりません。

7.4.18で行った修正は不十分で、一部のdblink関数に対してのみ問題を解消していました。(CVE-2007-6601、CVE-2007-3278)

- 一部のWHERE false AND var IN (SELECT ...)におけるプランナの失敗を修正しました。(Tom)
- マルチバイトデータベース符号化方式を使用している時に潜在する、translate()のクラッシュを修正しました。
- 長い例外メッセージでクラッシュしないようPL/Pythonを修正しました。(Alvaro)
- ecpgパーサを修正しました。(Michael)
- contrib/tablefuncのcrosstab()が、クラッシュするのではなく、独自の意味でカテゴリとしてNULL行識別子を扱うようにしました。(Joe)
- tsvectorおよびtsquery出力関数が正しくバックスラッシュをエスケープするように修正しました。(Teodor, Bruce)
- 入力文字列が大きな場合にto_tsvector()がクラッシュすることを修正しました。(Teodor)
- configureスクリプトを再生成する場合、特定のバージョンのAutoconfを使用することが必要になりました。(Peter)

これは開発者とパッケージ作成者のみに影響します。この変更は、AutoconfとPostgreSQLの未確認のバージョンの組み合わせを使用する事故を防ぐためになされました。その結果がうまくいくかどうかについて責任を負う限り、異なるバージョンのAutoconfを本当に使用したい場合は、このバージョン検査を取り除くことができます。

E.159. リリース7.4.18

リリース日: 2007-09-17

このリリースは7.4.17の不具合を修正したものです。 7.4メジャーリリースにおける新機能については[項E.177](#)を参照してください。

E.159.1. バージョン7.4.18への移行

7.4.Xからの移行ではダンプ/リストアは不要です。 しかし、7.4.11より前のバージョンからアップグレードする場合は、7.4.11のリリースノートを参照してください。

E.159.2. 変更点

- トランザクションが、行を挿入した後に、同一テーブルに対する同時実行中のVACUUMの終了時刻とほぼ同じ時点でアポートした場合のインデックス破損を防止します。(Tom)
- CREATE DOMAIN ... DEFAULT NULLが適切に動作するようにしました。(Tom)
- 過度なSSLエラーメッセージのログ処理を修正しました。(Tom)
- log_min_error_statementログ処理がメモリ不足になった時のクラッシュを修正しました。(Tom)
- 他のセッションによる一時テーブル処理を防ぐために起こるCLUSTERの失敗を防ぎます。(Alvaro)
- セキュリティ強化のため、パスワード認証のみを使用する/contrib/dblinkを使用するユーザが非特権ユーザであることを要求します。(Joe)

E.160. リリース7.4.17

リリース日: 2007-04-23

このリリースは7.4.16の不具合を修正したもので、セキュリティ問題の修正を1つ含みます。 7.4メジャーリリースにおける新機能については[項E.177](#)を参照してください。

E.160.1. バージョン7.4.17への移行

7.4.Xからの移行ではダンプ/リストアは不要です。しかし、7.4.11より前のバージョンからアップグレードする場合は、7.4.11のリリースノートを参照してください。

E.160.2. 変更点

- `search_path`における一時テーブルスキーマの明示的な指定をサポートしました。また、関数や演算子ではその検索を無効にしました(Tom)。

これは、`SECURITY DEFINER`関数が`search_path`の値を本当に安全に設定するために必要です。こうしないと、一時オブジェクトを使用して権限のないSQLユーザが`SECURITY DEFINER`関数の権限でコードを実行することができてしまうためです (CVE-2007-2138)。詳細は`CREATE FUNCTION`を参照してください。

- `/contrib/tsearch2`のクラッシュを修正しました(Teodor)。
- `VACUUM FULL`の`UPDATE`連鎖方法において潜在したデータ破損不具合を修正しました(Tom, Pavan Deolasee)。
- ハッシュインデックスの拡張時のパニックを修正しました(7.4.15で入ってしまった不具合です)(Tom)。

E.161. リリース 7.4.16

リリース日: 2007-02-05

このリリースは7.4.15の各種不具合を修正したもので、セキュリティ問題の修正も含みます。 7.4メジャーリリースにおける新機能については[項E.177](#)を参照してください。

E.161.1. バージョン7.4.16への移行

7.4.Xからの移行ではダンプ/リストアは不要です。 しかし、7.4.11より前のバージョンからアップグレードする場合は、7.4.11のリリースノートを参照してください。

E.161.2. 変更点

- 接続ユーザがバックエンドのメモリを読み取ることができるセキュリティ脆弱性を取り除きました。(Tom)

この脆弱性には、SQL関数が宣言されたデータ型を返すかどうか、そのテーブル列のデータ型が変更されたかどうかに関する通常の検査の抑制も含まれています (CVE-2007-0555、CVE-2007-0556)。 これらのエラーを悪用して簡単にバックエンドをクラッシュさせることができます。 また原理的には、アクセスを許していないはずのユーザがデータベースの内容を読み取ることができてしまいます。

- 実行不可能な分割点の選択によりB-Treeインデックスページの分割が失敗する可能性がある、稀に起こる不具合を修正しました。(Heikki Linnakangas)
- UNION をきっかけとした、稀に発生するAssert()クラッシュを修正しました。(Tom)
- 3バイト長を越えるUTF8シーケンスに関する、複数バイト文字処理のセキュリティを強化しました。(Tom)

E.162. リリース 7.4.15

リリース日: 2007-01-08

このリリースは7.4.14の各種不具合を修正したものです。 7.4メジャーリリースにおける新機能については[項E.177](#)を参照してください。

E.162.1. バージョン7.4.15への移行

7.4.Xからの移行ではダンプ/リストアは不要です。 しかし、7.4.11より前のバージョンからアップグレードする場合は、7.4.11のリリースノートを参照してください。

E.162.2. 変更点

- AIXにおける`getaddrinfo()`の扱いを改良しました。(Tom)
これは、特に統計情報コレクタの起動に関する問題を修正するものです。
- `VACUUM`における"親キーの再検索に失敗する"エラーを修正しました。(Tom)
- 数ギガバイトのハッシュインデックスに影響する不具合を修正しました。(Tom)
- 複数の空要素からなる`ARRAY[]`生成時の不具合を修正しました。(Tom)
- 新しく`initdb`を行ったインストレーションでは、`to_number()`と`to_char(numeric)`が`IMMUTABLE`ではなく`STABLE`になりました。(Tom)

これは、`lc_numeric`がこれらの関数の出力を変更する可能性があるためです。

- 括弧を使用した正規表現を使用するインデックスを改良しました。(Tom)

この改良により、`psql` `¥d`の性能も向上しました。

E.163. リリース7.4.14

リリース日: 2006-10-16

このリリースは7.4.13の各種不具合を修正したものです。 7.4メジャーリリースにおける新機能については[項E.177](#)を参照してください。

E.163.1. バージョン7.4.14への移行

7.4.Xからの移行ではダンプ/リストアは不要です。 しかし、7.4.11より前のバージョンからアップグレードする場合は、7.4.11のリリースノートを参照してください。

E.163.2. 変更点

- 型指定がないリテラルがANYARRAYと解釈される場合のコアダンプを修正しました。
- 区切り文字と重なって一致する文字を扱うよう `string_to_array()` を修正しました。

例えば `string_to_array('123xx456xxx789', 'xx')` です。

- `psql` の `\d` コマンドに対するパターンマッチにおけるちょっとした部分を修正しました。
- `/contrib/ltree` におけるインデックス破損不具合を修正しました。(Teodor)
- `/contrib/dbmirror` におけるバックスラッシュエスケープ処理を修正しました。
- リグレッション試験をUS DST法の最近の変更に合わせました。

E.164. リリース7.4.13

リリース日: 2006-05-23

このリリースは7.4.12の各種不具合を修正したもので、非常に深刻なセキュリティ問題を解消するパッチを含みます。7.4メジャーリリースにおける新機能については[項E.177](#)を参照してください。

E.164.1. バージョン7.4.13への移行

7.4.Xからの移行ではダンプ/リストアは不要です。しかし、7.4.11より前のバージョンからアップグレードする場合は、7.4.11のリリースノートを参照してください。

CVE-2006-2313およびCVE-2006-2314に示されたSQLインジェクション攻撃を完全に防ぐためには、アプリケーション側のコードの変更が必要となる場合があります。SQLコマンド内に信頼できない文字列を埋め込むアプリケーションでは、できる限り早く、その文字列が推奨するエスケープ技法を使用していることを確実に検証しなければなりません。ほとんどの場合、アプリケーションは文字列のエスケープ処理に、その場しのぎのコードではなく、ライブラリやドライバが提供する (libpqのPQescapeStringConn()) のような関数を使用しなければなりません。

E.164.2. 変更点

- すべての場合において、符号化が無効なマルチバイト文字を拒絶するようにサーバを変更しました。(Tatsuo, Tom)

PostgreSQLは少し前からこのような方向に移行していましたが、この検査がすべての符号化方式とすべてのテキスト入力に対して統一的に適用されるようになりました。さらに、単なる警告ではなく常にエラーとなるようになりました。この変更はCVE-2006-2313で示されるような種類のSQLインジェクション攻撃から保護します。

- 文字列リテラル内の`'\'`の安全ではない使用を拒絶します。

CVE-2006-2314で示されるような種類のSQLインジェクション攻撃からのサーバ側の保護として、サーバは、SQL文字列リテラル内のASCII単一引用符の表現として、`''`のみを受け付け、`'\'`を受け付けなくなりました。デフォルトでは、SQLインジェクションが可能となる状況である、`client_encoding`がクライアント側のみの符号化方式 (SJIS、BIG5、GBK、GB18030、UHC) に設定された場合にのみ`'\'`は拒絶されます。新しい設定パラメータ`backslash_quote`により、必要な場合にこの動作を調整できます。CVE-2006-2314に対して完全に保護するには、クライアント側を変更する必要があるかもしれないことに注意してください。安全ではないクライアントを安全ではないものとして明らかにすることが、`backslash_quote`の目的の一つです。

- libpqの文字列エスケープルーチンを、符号化方式と`standard_conforming_strings`を考慮するように変更しました。

これは、CVE-2006-2313およびCVE-2006-2314で示されるセキュリティ問題に対し、libpqを使用したアプリケーションを修正します。また、将来予定されるSQL標準文字列リテラル構文への移行に対しても保護しています。同時に複数のPostgreSQL接続を使用するアプリケーションは、各データベース接続において使用される設定に合わせてエスケープ処理が正しく行われるように、`PQescapeStringConn()`と`PQescapeByteaConn()`に移行しなければなりません。"独自に"文字列エスケープ処理を行うアプリケーションはライブラリルーチンを使用するように変更しなければなりません。

- 一部の不正な符号化方式変換関数を修正しました。

`win1251_to_iso`、`win866_to_iso`、`euc_tw_to_big5`、`euc_tw_to_mic`、`mic_to_euc_tw`はすべて可変拡張に関して正しくありませんでした。

- 文字列における偶然残る`'\'`の使用を整理しました。(Bruce, Jan)
- インデックススキャンの論理和が本来返すべき行を失う場合がある不具合を修正しました。
- Btreeインデックスが消去された場合のWAL再生を修正しました。
- `|`を含むパターンに対する`SIMILAR TO`を修正しました。(Tom)
- 独自のDH SSLパラメータを正しく使用するようにサーバを修正しました。(Michael Fuhr)
- Intel MacでのBonjourを修正しました。(Ashley Clark)
- さまざまな小規模のメモリリークを修正しました。

E.165. リリース7.4.12

リリース日: 2006-02-14

このリリースは7.4.11の各種不具合を修正したものです。7.4メジャーリリースにおける新機能については[項E.177](#)を参照してください。

E.165.1. バージョン7.4.12への移行

7.4.Xからの移行ではダンプ/リストアは不要です。しかし、7.4.11より前のバージョンからアップグレードする場合は、7.4.11のリリースノートを参照してください。

E.165.2. 変更点

- `SET SESSION AUTHORIZATION`によるクラッシュの可能性を修正しました。(CVE-2006-0553)

アサートを有効にしてコンパイルした場合（これはデフォルトではありません）、非特権ユーザがサーバをクラッシュでき、その結果、他のユーザへのサービスが一時的に不能になります。この問題を報告してくれたAkio Ishidaに感謝します。

- 自身が挿入した行に関する行の可視性ロジックについての不具合を修正しました。(Tom)

非常に稀な状況で、現在のコマンドが挿入した行が、本来無効であるべき時に既に有効になっているものとして扱われることがありました。リリース7.4.9、7.3.11で組み込まれた不具合を修正しました。

- `pg_clog`および`pg_subtrans`ファイル生成時に"ファイルが既に存在する"というエラーを引き起こす競合条件を修正しました。(Tom)
- 準備済み文内の`UNKNOWN`パラメータ用の`DOMAIN`制約を適切に検査します。(Neil)
- 独自の演算子や演算子クラスを参照するクロススキーマを持つダンプをリストアできるように修正しました。(Tom)
- `configure`時の`finite`および`isinf`の存在検査に関する移植性を修正しました。(Tom)

E.166. リリース7.4.11

リリース日: 2006-01-09

このリリースは7.4.10の各種不具合を修正したものです。 7.4メジャーリリースにおける新機能については[項E.177](#)を参照してください。

E.166.1. バージョン7.4.11への移行

7.4.Xからの移行ではダンプ/リストアは不要です。 しかし、7.4.8より以前のバージョンからアップグレードする場合は、7.4.8のリリースノートを参照してください。 また、後述のロケールやplperlの問題の影響がある場合、移行後にテキストの列に対するインデックスをREINDEXする必要があるかもしれません。

E.166.2. 変更点

- トランザクション外または失敗したトランザクション内から発行される、プロトコルレベルのDescribeメッセージを修正しました。(Tom)
- 韓国語など、異なる文字の組み合わせを同一のものとみなすロケールにおける文字列比較を修正しました。(Tom)
既存のテキスト列に対するインデックスを修正するにはREINDEXが必要になります。
- plperlが後でロケールを変更できないように、postmasterの起動時にロケール用の環境変数を設定します。
これは、initdb実行時に通知したロケールと異なるロケールを表す環境変数の下でpostmasterが起動された場合に発生する問題を修正します。 このような条件では、plperlの使用によりインデックスが破損される可能性があります。 こうした問題が発生していた場合、既存のテキスト列に対するインデックスのREINDEXが必要になります。
- 長期間存在した、ごく一部の特定のアジア向けマルチバイト文字セットを取り扱う場合のstrpos()と正規表現の不具合を修正しました。(Tatsuo)
- MD5およびXDESアルゴリズム用のソルト空間をすべて利用できないという、/contrib/pgcryptoのgen_saltの不具合を修正しました。(Marko Kreen, Solar Designer)
Blowfishおよび標準DES用のソルトには影響ありません。
- 指定した列数が実際に問い合わせで返される列数と異なる場合にクラッシュせずにエラーを返すように/contrib/dblinkを修正しました。(Joe)

E.167. リリース7.4.10

リリース日: 2005-12-12

このリリースは7.4.9の各種不具合を修正したものです。 7.4メジャーリリースにおける新機能については[項E.177](#)を参照してください。

E.167.1. バージョン7.4.10への移行

7.4.Xからの移行ではダンプ/リストアは不要です。 しかし、7.4.8より以前のバージョンからアップグレードする場合は、7.4.8のリリースノートを参照してください。

E.167.2. 変更点

- トランザクションログ管理における競合条件を修正しました。

I/O操作により間違ったページが初期化されてしまうわずかな状態がありました。 これは、アサート失敗またはデータ破壊をもたらします。

- 現在のトランザクションがすでにアボートしている時にクライアントがBindプロトコルメッセージを送信した場合の失敗を防ぎます。
- `/contrib/ltree`を修正しました。(Teodor)
- AIX、HPUXにおけるコンパイルを修正しました。(Tom)
- 長期間残っていた、外部結合の実行計画作成エラーを修正しました。

この不具合により、"マージ結合可能な結合条件のみをサポートするRIGHT JOIN"という不正なエラーが発生することがありました。

- テーブルが削除された場合のpg_autovacuumのコアダンプを防止します。

E.168. リリース 7.4.9

リリース日: 2005-10-04

このリリースは7.4.8の各種不具合を修正したものです。 7.4メジャーリリースにおける新機能については[項E.177](#)を参照してください。

E.168.1. バージョン7.4.9への移行

7.4.Xからの移行ではダンプ/リストアは不要です。 しかし、7.4.8より以前のバージョンからアップグレードする場合は、7.4.8のリリースノートを参照してください。

E.168.2. 変更点

- `VACUUM`によって`ctid`チェーンがすぐに削除されてしまうエラーを修正し、`ctid`リンクを追跡するコードにおいてより多くの検査を行うようにしました。

これにより、長い間存在した、非常に稀な環境におけるクラッシュという問題が修正されました。

- マルチバイト文字セットを使用する場合に、`CHAR()`の空白文字埋め込みが適切に指定した長さになるように修正しました。(Yoshiyuki Asaba)

以前のリリースでは、`CHAR()`の空白文字埋め込みが正しくありませんでした。 格納する文字数を考慮せずに、単に指定されたバイト数で空白文字を埋め込んでいただけでした。

- `COPY`における読みとりのみのトランザクションに関する試験の意味を修正しました。

このコードで`COPY TO`を禁止していましたが、本来ならば`COPY FROM`を禁止しなければなりません。

- リレーションの内部のみを参照する、外部結合のON句に関する計画作成の問題を修正しました。

- `x FULL JOIN y ON true`の境界に関して更に改修しました。

- `array_in`と`array_recv`のOIDパラメータの検証をより偏執的に行うようにしました。

- `a`列にGiSTインデックスを持つ場合の`UPDATE a=... WHERE a...`といった問い合わせにおいて行が見つからない点を修正しました。

- 日付時刻の解析の堅牢性を向上しました。

- 部分的に書き込まれたWALページに対する検査を改良しました。

- SSL有効時のシグナルの扱いに対する堅牢性を向上しました。

- `postmaster`の起動時に、`max_files_per_process`以上のファイルを開かないようにしました。

- さまざまなメモリリークを修正しました。

- 移植性の向上をいろいろと行いました。

- PL/pgSQLにおいて、変数が参照渡し型の場合の`var := var`の扱いを修正しました。

- 正確なSnowballコードを使用するように`contrib/tsearch2`を更新しました。

E.169. リリース7.4.8

リリース日: 2005-05-09

このリリースは、複数のセキュリティ関連の問題を含む、7.4.7の各種不具合を修正したものです。7.4メジャーリリースにおける新機能については[項E.177](#)を参照してください。

E.169.1. バージョン7.4.8への移行

7.4.Xからの移行ではダンプ/リストアは不要です。しかし、7.4.Xのシステムカタログで見つかった2つの重大なセキュリティ問題を突かれる可能性があります。ダンプ、7.4.8のinitdbを使用したinitdb、リロードを行うことで、自動的にこれらの問題を修正します。

より重大なセキュリティ問題は、組み込みの文字セット符号化変換関数により、権限を持たないユーザがSQLコマンドを呼び出すことができるという点です。このような用途のためにこれらの関数を設計していませんでしたが、悪意のある引数の設定に対する安全性がありませんでした。この修正により、これらの関数の宣言されたパラメータリストがSQLコマンドから呼び出されないように変更されました。（通常の符号化変換機構の使用には影響はありません。）

もうひとつの問題は、contrib/tsearch2モジュールが、internal型の引数を取らない場合に間違ってinternalを返すものと宣言された関数を複数作成することです。これによりinternal型の引数を使用するすべての関数について、型の安全性が損なわれます。

initdb、もしくは、後述の手作業による修正手順に従って、すべてのインストールにおいてこれらのエラーを修正することを強く勧めます。これらのエラーにより、少なくとも、権限を持たないデータベースユーザがサーバプロセスをクラッシュさせることができます。また、権限を持たないユーザがデータベーススーパーユーザ権限を手に入れることができる可能性もあります。

initdbを行いたくないのであれば、スーパーユーザ権限で以下の手続きを行ってください。

```
BEGIN;
UPDATE pg_proc SET proargtypes[3] = 'internal'::regtype
WHERE pronamespace = 11 AND pronargs = 5
    AND proargtypes[2] = 'cstring'::regtype;
-- このコマンドは90行を更新したと報告するはずです。
-- 異なる場合は、コミットせずにロールバックして調査を行ってください。
COMMIT;
```

contrib/tsearch2をインストールしている場合、次に以下を実行してください。

```
BEGIN;
UPDATE pg_proc SET proargtypes[0] = 'internal'::regtype
WHERE oid IN (
    'dex_init(text)'::regprocedure,
    'snb_en_init(text)'::regprocedure,
    'snb_ru_init(text)'::regprocedure,
    'spell_init(text)'::regprocedure,
    'syn_init(text)'::regprocedure
);
-- このコマンドは5行を更新したと報告するはずです。
-- 異なる場合は、コミットせずにロールバックして調査を行ってください。
COMMIT;
```

このコマンドが"`dex_init(text)`関数は存在しません"といったメッセージで失敗したら、データベースにtsearch2がインストールされていないか、すでに更新済みであるかのどちらかです。

上の手続きを、template1を含むインストール内のすべてのデータベースで行わなければなりません。理想を言えば、template0に対しても実施してください。テンプレートデータベースで修正を行わなかった場合、この後に作成されるデータベースにこのエラーが含まれてしまいます。template1は他のデータベースと同じ方法で修正できますが、template0では更に行わなければならないことがあります。まず、任意のデータベースから以下を実行してください。

```
UPDATE pg_database SET dataallowconn = true WHERE datname = 'template0';
```

次に、template0に接続し、上記修正手続きを実施してください。最後に以下を実行してください。

```
-- 再度template0を凍結させます
VACUUM FREEZE;
-- そして、今後の変更に対し保護します。
UPDATE pg_database SET dataallowconn = false WHERE datname = 'template0';
```

E.169.2. 変更点

- 誤使用を防ぐために符号化関数のシグネチャを変更しました。
- `INTERNAL`結果の関数の危険な使用を防止するように、`contrib/tsearch2`を変更しました。
- ある目的（例えばSELECT FOR UPDATE）用にコミットされるように見えるトランザクションが直後に別の目的用にコミットされるという従来から存在した競合状況を修正しました。

一貫性が明らかにないデータが短期間アプリケーションから参照できてしまうため、これはきわめて重大な不具合です。

- リレーションの拡張とVACUUMとの間の競合状態を修正しました。

理論上、これにより新たに挿入されたページの価値が失われます。しかしこうした状況が発生する頻度はきわめてまれで、アサートエラー以外に発生した事例はありません。

- `TIME WITH TIME ZONE`型の値の比較を修正しました。

`--enable-integer-datetimes`設定スイッチが使用されている場合、この比較コードに間違いがありました。注意: この修正により列値のソート順が正しくなるように変更されましたので、`TIME WITH TIME ZONE`型の列に対するインデックスを持つ場合、この更新をインストールした後に`REINDEX`を行う必要があります。

- `TIME WITH TIME ZONE`型の値に対する`EXTRACT(EPOCH)`を修正しました。
- `INTERVAL`型の値の秒の小数部分において、負の値が間違って表示される点を修正しました。

このエラーは、`--enable-integer-datetimes`設定スイッチが使用された場合にのみ発生します。

- バックエンドのシャットダウン中に行われた操作が統計情報コレクタで確実に集計されるようにしました。

これは、システムカタログのバキューム処理が十分な間隔で行われていないという`pg_autovacuum`の報告を解消するためのものです。バックエンド終了時の一時テーブル削除が原因のカタログの削除に関するものではありません。

- `plpgsql`において更にバッファオーバーランの検査を追加しました。(Neil)
- `%`を含む名前のトリガを正しくダンプできるように`pg_dump`を修正しました。
- より最近のOpenSSLによる構築のために`contrib/pgcrypto`を修正しました。(Marko Kreen)
- `contrib/intagg`に対する更なる64ビット向けの修正を行いました。
- `RECORD`型を返す関数の最適化の間違いを修正しました。
- 月に関連した書式のために`to_char(interval)`がコアダンプすることを防止しました。
- `COALESCE(NULL, NULL)`でのクラッシュを防止しました。
- PL関数を正しく呼び出すように`array_map`を修正しました。
- `ALTER DATABASE RENAME`における権限検査を修正しました。
- `ALTER LANGUAGE RENAME`を修正しました。
- `RemoveFromWaitQueue`がその終了時に整理するようにしました。

これは、トランザクションがロック待ち状態（問い合わせのキャンセルによりよく発生します）に陥り、ロック獲得側が非常に短期間でロックを解放する場合にのみ発生する、ロック管理エラーを修正します。

- `INSERT ... SELECT`内の型付けされていないパラメータがある場合の問題を修正しました。
- `ALTER TABLE SET WITHOUT OIDS`後の`CLUSTER`の失敗を修正しました。

E.170. リリース7.4.7

リリース日: 2005-01-31

このリリースは複数のセキュリティ問題を含む、7.4.6の各種不具合を修正したものです。7.4メジャーリリースにおける新機能については[項E.177](#)を参照してください。

E.170.1. バージョン7.4.7への移行

7.4.Xからの移行の場合は ダンプ/リストアは必要ありません。

E.170.2. 変更点

- Disallow `LOAD` to non-superusers

On platforms that will automatically execute initialization functions of a shared library (this includes at least Windows and ELF-based Unixen), `LOAD` can be used to make the server execute arbitrary code. Thanks to NGS Software for reporting this.

- Check that creator of an aggregate function has the right to execute the specified transition functions

This oversight made it possible to bypass denial of `EXECUTE` permission on a function.

- Fix security and 64-bit issues in contrib/intagg
- Add needed `STRICT` marking to some contrib functions (Kris Jurka)
- Avoid buffer overrun when plpgsql cursor declaration has too many parameters (Neil)
- Fix planning error for `FULL` and `RIGHT` outer joins

The result of the join was mistakenly supposed to be sorted the same as the left input. This could not only deliver mis-sorted output to the user, but in case of nested merge joins could give outright wrong answers.

- Fix plperl for quote marks in tuple fields
- Fix display of negative intervals in SQL and GERMAN datestyles
- Make `age(timestamptz)` do calculation in local timezone not GMT

E.171. リリース7.4.6

リリース日: 2004-10-22

このリリースは7.4.5の各種不具合を修正したものです。 7.4メジャーリリースにおける新機能については[項E.177](#)を参照してください。

E.171.1. バージョン7.4.6への移行

7.4.Xからの移行の場合は ダンプ/リストアは必要ありません。

E.171.2. 変更点

- ディスク上のヒントビットの更新に失敗する可能性を修正

稀な環境下で、この誤りにより"could not access transaction status"という失敗になる可能性があります。 これはデータ損失の可能性がある不具合です。

- ハッシュ外部結合がタプルを喪失しないことを確実にしました。

ハッシュ結合計画を使用する非常に大規模な左結合が、右側のデータ分布のみが与えられた一致しない左側の行を出力する可能性があります。

- rootによるpg_ctlの実行を禁止

潜在するセキュリティ問題から保護します。

- make_oidjoins_checkにおける/tmp内の中間ファイルの使用を避けました。

これはセキュリティ問題として報告されたものです。 しかし、開発者以外がこのスクリプトを使用する理由がありませんので、ほとんど気にする必要はありません。

- Prevent forced backend shutdown from re-emitting prior command result

稀な場合ですが、実際には強制的なデータベースの停止によってアボートされた最後のコマンドをクライアントが成功したものとなす可能性があります。

- pg_stat_get_backend_idsetの不具合を修正

システムの統計情報ビューの一部で誤動作する可能性があります。

- Fix small memory leak in postmaster

- Fix "expected both swapped tables to have TOAST tables" bug

ALTER TABLE DROP COLUMN後のCLUSTERなどの場合に発生する可能性があります。

- Prevent pg_ctl restart from adding -D multiple times

- Fix problem with NULL values in GiST indexes

- :: is no longer interpreted as a variable in an ECPG prepare statement

E.172. リリース 7.4.5

リリース日: 2004-08-18

このリリースは7.4.4の深刻な不具合を修正したものです。 7.4メジャーリリースにおける新機能については[項E.177](#)を参照してください。

E.172.1. 7.4.4バージョンへの移行

7.4.Xからの移行の場合は ダンプ/リストアは必要ありません。

E.172.2. 変更点

- B-treeインデックスの同時挿入時にクラッシュする可能性を修正

このパッチは、B-treeインデックスへの同時挿入時にサーバパニックを引き起こす稀な問題を修正します。 永続的な損傷にはなりませんが、リリースし直すべきであると判断しました。 7.4より前のリリースには、この不具合はありません。

E.173. リリース 7.4.4

リリース日: 2004-08-16

このリリースは7.4.3の様々な不具合を修正したものです。 7.4メジャーリリースにおける新機能については[項E.177](#)を参照してください。

E.173.1. 7.4.4バージョンへの移行

7.4.Xからの移行の場合は ダンプ/リストアは必要ありません。

E.173.2. 変更点

- クラッシュ時のコミット済みのトランザクションの欠落の可能性を修正

トランザクションのコミットとチェックポイント処理との間の内部ロック処理が不十分であったため、最近のチェックポイント直前にコミットされたトランザクションの全部あるいは一部がデータベースのクラッシュの後の再起動時に欠落する可能性があります。 この深刻な不具合はPostgreSQL 7.1以降から存在していました。

- Check HAVING restriction before evaluating result list of an aggregate plan
- Avoid crash when session's current user ID is deleted
- Fix hashed crosstab for zero-rows case (Joe)
- Force cache update after renaming a column in a foreign key
- Pretty-print UNION queries correctly
- Make psql handle `¥r¥n` newlines properly in COPY IN
- pg_dump handled ACLs with grant options incorrectly
- Fix thread support for OS X and Solaris
- Updated JDBC driver (build 215) with various fixes
- ECPG fixes
- Translation updates (various contributors)

E.174. リリース7.4.3

リリース日: 2004-06-14

このリリースは7.4.2の様々な不具合を修正したものです。 7.4メジャーリリースにおける新機能については[項E.177](#)を参照してください。

E.174.1. 7.4.3バージョンへの移行

7.4.Xからの移行の場合は ダンプ/リストアは必要ありません。

E.174.2. 変更点

- Fix temporary memory leak when using non-hashed aggregates (Tom)
- ECPG fixes, including some for Informix compatibility (Michael)
- Fixes for compiling with thread-safety, particularly Solaris (Bruce)
- Fix error in COPY IN termination when using the old network protocol (ljb)
- Several important fixes in pg_autovacuum, including fixes for large tables, unsigned oids, stability, temp tables, and debug mode (Matthew T. O'Connor)
- Fix problem with reading tar-format dumps on NetBSD and BSD/OS (Bruce)
- Several JDBC fixes
- Fix ALTER SEQUENCE RESTART where last_value equals the restart value (Tom)
- Repair failure to recalculate nested sub-selects (Tom)
- Fix problems with non-constant expressions in LIMIT/OFFSET
- Support FULL JOIN with no join clause, such as X FULL JOIN Y ON TRUE (Tom)
- Fix another zero-column table bug (Tom)
- Improve handling of non-qualified identifiers in GROUP BY clauses in sub-selects (Tom)
副selectにおける選択リストの別名が外部の問い合わせレベルの名前より優先されるようになりました。
- Do not generate "NATURAL CROSS JOIN" when decompiling rules (Tom)
- Add checks for invalid field length in binary COPY (Tom)
この修正はセキュリティホールの発生を抑えるものです。
- Avoid locking conflict between `ANALYZE` and `LISTEN/NOTIFY`
- Numerous translation updates (various contributors)

E.175. リリース7.4.2

リリース日: 2004-03-08

このリリースは7.4.1の様々な不具合を修正したものです。 7.4メジャーリリースにおける新機能については[項E.177](#)を参照してください。

E.175.1. 7.4.2バージョンへの移行

7.4.Xからの移行の場合は ダンプ/リストアは必要ありません。 しかし、これは初期の7.4.Xシステムカタログの内容に存在した2つのエラーの修正を組み込む最も簡単な方法としてお勧めできます。 7.4.2のinitdbを使用したダンプ、initdb、再ロードという流れにより、自動的にこの問題が修正されます。

この2つのエラーのより深刻な点は、`anyarray`データ型が間違った整列ラベルを持っていたことです。 `pg_statistic`システムカタログが`anyarray`型の列を使用しているため、これは問題です。 この間違ったラベル付けにより、`double`で整列された列(`float8`や`timestamp`など)を`WHERE`句に含む問い合わせの計画を作成中、プランナは間違った推定を行ない、最悪はクラッシュしてしまいます。 `initdb`や後述の手作業による修復手順に従って、全てのインストレーションでこのエラーを修復することを強く推奨します。

あまり深刻ではないエラーは、`SET`の代わりに`UPDATE pg_settings`を使用できるように`pg_settings`システムビューの出力が`public`に対して更新アクセス可能すべきでした。 これは、`initdb`や手作業で修正することができますが、`UPDATE pg_settings`を使用したいのでない限り修正する必要はありません。

`initdb`を実行したくなくれば、以下の手順で`pg_statistic`を修正することができます。 データベーススーパーユーザで実行してください。

```
-- clear out old data in pg_statistic:
DELETE FROM pg_statistic;
VACUUM pg_statistic;
-- this should update 1 row:
UPDATE pg_type SET typalign = 'd' WHERE oid = 2277;
-- this should update 6 rows:
UPDATE pg_attribute SET attalign = 'd' WHERE atttypid = 2277;
--
-- At this point you MUST start a fresh backend to avoid a crash!
--
-- repopulate pg_statistic:
ANALYZE;
```

これは稼働中のデータベースに対して行なうことができますが、変更したデータベースで実行中の全てのバックエンドは、`pg_statistic`を安全に更新させるために、再起動されなければなりません。

`pg_settings`エラーを修復するには単に以下を実行してください。

```
GRANT SELECT, UPDATE ON pg_settings TO PUBLIC;
```

上述の手順は、`template1`を含むインストレーションの全てのデータベースに対して行なわなければなりません。理想をいえば`template0`にも行なってください。 テンプレートデータベースに対して修復を行なわないと、将来作成されるデータベースに同じエラーが含まれることになります。 `template1`は他のデータベースと同様の方法で修正できますが、`template0`の修正には更に手順が必要です。 まず、任意のデータベースから以下を発行してください。

```
UPDATE pg_database SET datallowconn = true WHERE datname = 'template0';
```

次に`template0`に接続し、上の手順を実行してください。 最後に以下を実行してください。

```
-- re-freeze template0:
VACUUM FREEZE;
-- and protect it against future alterations:
UPDATE pg_database SET datallowconn = false WHERE datname = 'template0';
```

E.175.2. 変更点

リリース7.4.2には、リリース7.3.6で行なわれた修正に加え、以下の修正が組み込まれています。

- Fix `pg_statistics` alignment bug that could crash optimizer

See above for details about this problem.

- Allow non-super users to update `pg_settings`
- Fix several optimizer bugs, most of which led to "variable not found in subplan target lists" errors
- Avoid out-of-memory failure during startup of large multiple index scan
- Fix multibyte problem that could lead to "out of memory" error during `COPY IN`
- Fix problems with `SELECT INTO / CREATE TABLE AS` from tables without OIDs
- Fix problems with `alter_table` regression test during parallel testing
- Fix problems with hitting open file limit, especially on OS X (Tom)
- Partial fix for Turkish-locale issues

`initdb` will succeed now in Turkish locale, but there are still some inconveniences associated with the `i/I` problem.

- Make `pg_dump` set client encoding on restore
- Other minor `pg_dump` fixes
- Allow `ecpg` to again use C keywords as column names (Michael)
- Added `ecpg` `WHENEVER NOT_FOUND` to `SELECT/INSERT/UPDATE/DELETE` (Michael)
- Fix `ecpg` crash for queries calling set-returning functions (Michael)
- Various other `ecpg` fixes (Michael)
- Fixes for Borland compiler
- Thread build improvements (Bruce)
- Various other build fixes
- Various JDBC fixes

E.176. リリース7.4.1

リリース日: 2003-12-22

このリリースは7.4の様々な不具合を修正したものです。 7.4メジャーリリースにおける新機能については[項E.177](#)を参照してください。

E.176.1. 7.4.1バージョンへの移行

7.4 からの移行の場合は ダンプ/リストアは必要ありません。

情報スキーマ内の修正をインストールする場合は、データベース内にリロードする必要があります。 これは、`initdb`を実行して新しいクラスタを初期化すること、もしくは、新しいリリースをインストールした後に、スーパーユーザとして`psql`を使用して各データベースで以下の一連のSQLコマンドを実行することで行なうことができます。

```
DROP SCHEMA information_schema CASCADE;  
¥i /usr/local/pgsql/share/information_schema.sql
```

2つ目のコマンド内のパスはインストレーションに合わせてください。

E.176.2. 変更点

- Fixed bug in `CREATE SCHEMA` parsing in ECPG (Michael)
- Fix compile error when `--enable-thread-safety` and `--with-perl` are used together (Peter)
- Fix for subqueries that used hash joins (Tom)

不適切な構造体の共有のため、特定のハッシュ結合を使用する副問い合わせクラッシュしました。

- Fix free space map compaction bug (Tom)
- Fix for Borland compiler build of `libpq` (Bruce)
- Fix `netmask()` and `hostmask()` to return the maximum-length masklen (Tom)

7.4より前のリリースと一貫性を持つ値を返すように関数を修正しました。

- Several `contrib/pg_autovacuum` fixes

この修正には、不適切な変数の初期化、`TRUNCATE`後のvacuumの抜け、vacuum期間の計算のオーバーフローを含んでいます。

- Allow compile of `contrib/cube` under Cygwin (Jason Tishler)
- Fix Solaris use of password file when no passwords are defined (Tom)

Solarisにおいてパスワードが定義されていない場合に何らかのパスワード認証を行なうとクラッシュする点を修正しました。

- JDBC fix for thread problems, other fixes
- Fix for `bytea` index lookups (Joe)
- Fix information schema for bit data types (Peter)
- Force `zero_damaged_pages` to be on during recovery from WAL
- Prevent some obscure cases of "variable not in subplan target lists"
- Make `PQescapeBytea` and `byteaout` consistent with each other (Joe)
- Escape `bytea` output for bytes > 0x7e (Joe)

異なるクライアント符号化方式がbyteaの入出力に使用された場合、符号化方式の違いによりbytea値が破壊される可能性があります。この修正では、影響を受ける可能性があるバイトを全てエスケープしました。

- Added missing `SPI_finish()` calls to dblink's `get_tuple_of_interest()` (Joe)
- New Czech FAQ
- Fix information schema view `constraint_column_usage` for foreign keys (Peter)
- ECPG fixes (Michael)
- Fix bug with multiple `IN` subqueries and joins in the subqueries (Tom)
- Allow `COUNT('x')` to work (Tom)
- Install ECPG include files for Informix compatibility into separate directory (Peter)

Informix互換用のECPGヘッダファイルの名前の一部がオペレーティングシステムのヘッダファイルと競合しました。独自のディレクトリにインストールすることで、名前の競合が防止されました。

- Fix SSL memory leak (Neil)

このリリースは、7.4における、SSLが割り当てたメモリを全て解放しない不具合を修正しました。

- Prevent `pg_service.conf` from using service name as default dbname (Bruce)
- Fix local ident authentication on FreeBSD (Tom)

[前のページ](#)

リリース7.4.2

[ホーム](#)
[上に戻る](#)

[次のページ](#)

リリース 7.4

E.177. リリース 7.4

リリース日: 2003-11-17

E.177.1. 概要

このリリースの大きな変更点を以下に示します。

`IN/NOT IN` 副問い合わせが非常に効率的になりました。

以前のリリースでは、`IN/NOT IN` 副問い合わせは、シーケンシャルスキャンによって副問い合わせと一致する上位の問い合わせと結合されていました。7.4のコードでは、通常の結合で使用する洗練された技術と同じものを使用し、そのため、非常に高速になりました。これで `IN` は通常、等価な `EXISTS` と同程度の速度、場合によってはより高速になりました。これにより、これまでのリリースにおいて行なっていた慣習的常識はくつがえりました。

ハッシュバケットを使用した `GROUP BY` 処理の改良

以前のリリースでは、グループ化される行が最初にソートされました。7.4のコードでは、グループ当たり1項目で、ハッシュテーブルに結果を積むことによって、ソートを行わずに `GROUP BY` ができるようになりました。しかし、ハッシュテーブルが `sort_mem` に収まるには大き過ぎると推定した場合、まだソート技術を使用します。

複数キーによるハッシュ結合能力(新規)

以前のリリースでは、ハッシュ結合は1つのキー上でのみ存在できました。このリリースでは、複数列に対するハッシュ結合が可能です。

明示的な `JOIN` 構文を使用した問い合わせはより良く最適化されるようになりました。

以前のリリースでは、明示的な `JOIN` 構文を使用する問い合わせを、単に、構文によって示される順番で評価していました。7.4ではこれらの問い合わせを完全に最適化できるようになりました。つまり、オブティマイザは全ての結合順序を考慮し、最も効率的なものを選択します。しかし、外部結合はまだ宣言順に従わなければなりません。

より強力、より高速な正規表現コード。

正規表現モジュール全体が、もともとはTclで作成されたHenry Spencer氏による新しいバージョンに置き換えられました。このコードではかなり性能が向上し、様々な種類の正規表現をサポートしています。

簡単なSQL関数のインライン関数

簡単なSQL関数を元の問い合わせ内のSQLに含めることによりインライン化できるようになりました。呼び出しに伴うオーバーヘッドを抑制することにより、性能が向上します。つまり、簡単なSQL関数がマクロのように動作するようになりました。

完全なIPv6接続とIPv6アドレスデータ型のサポート

以前のリリースでは、IPv4接続のみが可能、そしてIPデータ型としてはIPv4アドレスのみをサポートしていました。このリリースでは両者に関して完全なIPv6をサポートしています。

SSLの性能と信頼性の大きな改良

SSL APIに非常に詳しい方々により、SSLコードは書き直され、SSLキー調停やエラー復旧が改良されました。

空き領域マップを効率的に空インデックスページを再利用させました。また、他の空き領域の管理を改良しました。

以前のリリースでは、削除行によって空のまま残されたB-treeインデックスページは、元々そのページにインデックス付けされた行に近いインデックス値を持つ行によってのみ再利用されました。7.4では、`VACUUM` は空のインデックスページを記録し、それらを将来のインデックス行用に再利用することができます。

標準SQLの情報スキーマ。

情報スキーマは、データベース内で定義されたスキーマオブジェクトについての情報にアクセスするための、標準化され、安定した方法を提供します。

カーソルはより標準SQLに近く従うようになりました。

`FETCH`と`MOVE`コマンドは、より標準SQLに近く従うよう改良されました。

カーソルがトランザクションの外側でも存在できるようになりました。

これらのカーソルは保持可能カーソルとも呼ばれます。

新しいクライアントからサーバへのプロトコル。

新しいプロトコルでは、エラーコード、より多くの状況情報、より高速な起動、より優れたバイナリデータの通信、SQLコマンドとパラメータ値の分離、プロトコルレベルで利用できる準備された文が付与されました。また、`COPY`失敗時の復旧がより明確になりました。昔のプロトコルもまだ、サーバ、クライアント共サポートされています。

`libpq` と `ECPG` アプリケーションが完全にスレッドセーフになりました。

以前の`libpq` のリリースでも既にスレッドをサポートしていましたが、このリリースでは、データベース接続開始期間に使用されていたスレッドセーフでないいくつかのコードを修正し、スレッドセーフに改良されました。この機能を有効にするには、`configure`の`--enable-thread-safety`を使用しなければなりません。

新しいバージョンのフルテキストインデックス

`contrib/tsearch2`にある新しいフルテキストインデックス手法が利用可能です。

新しい自動vacuumツール。

`contrib/autovacuum`にある新しい自動vacuumツールは、`INSERT/UPDATE/DELETE`の活動について、統計情報テーブルを監視し、必要に応じて自動的にvacuumを行ないます。

配列の扱いは、改良され、サーバの中核部分に移りました。

多くの配列の制限が取り除かれました。これにより、配列は、ほぼ完全にデータ型をサポートできるように動作します。

E.177.2. バージョン7.4への移行

以前のリリースからデータを移行する場合、`pg_dump` によるダンプ/リストアが必要です。

以下の非互換部分を確認して下さい。

- サーバ側での自動コミット設定は削除され、クライアントアプリケーションと言語内で再実装されました。サーバ側の自動コミットは、自動コミットの振舞いを自身で制御したい言語やアプリケーションで、多くの問題の原因となっていました。そのため、サーバから自動コミットは取り除かれ、個々のクライアントAPIに適切に追加されました。
- エラーメッセージの言葉使いがこのリリースで大きく変わりました。これらのメッセージをより正確に、よりユーザ向けにするために非常に多くの労力を費やしました。アプリケーションがエラーメッセージを解析してエラー条件の違いを検知するようであれば、その代わりに、新しいエラーコード機能を使用するように強く勧めます。
- 明示的な`JOIN`を使用する内部結合構文の振舞いが変わる可能性があります。これらはより良く最適化されるようになったためです。
- 多くのサーバ設定パラメータの名前が、明確になるように変更されました。主にログに関連したパラメータです。
- `FETCH 0` や `MOVE 0` は何も行なわないようになりました。以前のリリースでは、`FETCH 0`は残存する全ての行を取り出し、`MOVE 0` はカーソルの最後まで移動していました。
- `FETCH` と `MOVE` は、実際に取り出し/移動した行の数を返すようになりました。なお、カーソルがその先頭/最後にあった場合は0を返します。以前のリリースは、そのコマンドに渡した行数を返し、実際に取り出した、あるいは、移動した行数を返しませんでした。
- `COPY` は、行末を表すシーケンスとして、復帰のみ、あるいは、復帰/改行を使用したファイルも処理できるようになりました。復帰と改行そのものは、データ値としては受け付けられません。その代わりに`¥r` と`¥n`を使用して下さい。
- `char(n)` から `varchar(n)` もしくは `text` への変換時、最後の空白は除去されるようになりました。これは、ほとんどのユーザが常にそうなってほしいと期待している動作です。
- `float(p)` データ型は、10進数ではなく、2進数での p 桁を表すようになりました。新しいこの動作は、標準SQLに従います。
- 曖昧な日付値は、`datestyle`設定で指定した順序に一致しなければならなくなりました。以前のリリースでは、10/20/03というデータ指定は、`datestyle`で日が最初であると指定していたとしても、10月と解釈されていました。

7.4では、日付の指定がその時点の`datestyle`設定では無効である場合、エラーが発生します。

- `oidrand`、`oidsrand`、`userfntest` 関数が削除されました。これらの関数は、もう使用されることはないとは決定されました。
- `'now'` or `'today'` 時間が変動する日付/時刻値を指定する文字列リテラルは、列のデフォルト式で想定したとおりに働かなくなりました。これらは、挿入時の時刻ではなく、テーブルを生成した時刻をそのデフォルトに設定するようになりました。その代わりに、`now()`、`current_timestamp`、`current_date`といった関数を使用して下さい。

以前のリリースでは、`'now'` といった文字列がテーブル生成時刻ではなく、`INSERT`時刻と解釈されるような特殊なコードがありました。しかし、この回避方法は全ての場合に適用されませんでした。リリース7.4では、デフォルト式は`now()` や `current_timestamp` といった関数を適切に使用して定義することが必須となりました。これらは全ての状況で動作します。

- ドル印(`$`)は演算子名として使用できなくなりました。その代わりに、識別子の文字(先頭文字は除く)として使用できるようになりました。これは、他のデータベースシステムとの互換性を向上させるため、および、パラメータのプレースホルダ(`$n`)が演算子に近接していた場合の構文問題を防止するために行なわれました。

E.177.3. 変更点

以下に、リリース7.4と以前のメジャーリリースとの間になされた変更点について、詳細な説明を示します。

E.177.3.1. サーバ操作についての変更

- Allow IPv6 server connections (Nigel Kukard, Johan Jordaan, Bruce, Tom, Kurt Roeckx, Andrew Dunstan)
- Fix SSL to handle errors cleanly (Nathan Mueller)

In prior releases, certain SSL API error reports were not handled correctly. This release fixes those problems.

- SSL protocol security and performance improvements (Sean Chittenden)

SSLの性能の低さのために、SSLキーの再調停があまりにも頻繁に発生していました。また、初期のキーの取扱いが改良されました。

- Print lock information when a deadlock is detected (Tom)

これによりデッドロック状況のデバッグが容易になります。

- Update `/tmp` socket modification times regularly to avoid their removal (Tom)

これは、`/tmp` ディレクトリを整理する管理スクリプトによってサーバソケットファイルが削除されてしまうことを防ぐ助けになるでしょう。

- Enable PAM for Mac OS X (Aaron Hillegass)

- Make B-tree indexes fully WAL-safe (Tom)

以前のリリースでは、ある特定かつ稀な場合、サーバクラッシュによりB-treeインデックスが破損してしまいました。このリリースでは、これらのもっとも稀な場合について削除しました。

- Allow B-tree index compaction and empty page reuse (Tom)

- Fix inconsistent index lookups during split of first root page (Tom)

以前のリリースでは、単一ページのインデックスを2ページに分割する際に、他のデータベースセッションがインデックス項目を探すことができない、短い時間がありました。このリリースはこの稀に起こる失敗を修正しました。

- Improve free space map allocation logic (Tom)

- 空領域マップの割り当てロジックの改良 (Tom)

- Preserve free space information between server restarts (Tom)

以前のリリースでは、`postmaster`が停止した時に空領域は保存されませんでした。そのため、新しく稼働したサーバは、空領域情報を持ちませんでした。このリリースでは、空領域マップを保存し、サーバの再起動時にそれを再読み込みします。

- Add start time to `pg_stat_activity` (Neil)
- New code to detect corrupt disk pages; erase with `zero_damaged_pages` (Tom)
- New client/server protocol: faster, no username length limit, allow clean exit from `COPY` (Tom)
- Add transaction status, table ID, column ID to client/server protocol (Tom)
- Add binary I/O to client/server protocol (Tom)
- Remove autocommit server setting; move to client applications (Tom)
- New error message wording, error codes, and three levels of error detail (Tom, Joe, Peter)

E.177.3.2. 性能の向上

- Add hashing for `GROUP BY` aggregates (Tom)
- Make nested-loop joins be smarter about multicolumn indexes (Tom)
- Allow multikey hash joins (Tom)
- Improve constant folding (Tom)
- Add ability to inline simple SQL functions (Tom)
- Reduce memory usage for queries using complex functions (Tom)

以前のリリースでは、割り当てられたメモリを返す関数は、問い合わせが完了するまで、それを解放しませんでした。このリリースでは、関数が割り当てたメモリを関数呼び出しの終了時に解放することができ、関数で使用する総メモリ量が減少します。

- Improve GEQO optimizer performance (Tom)

このリリースでは、GEQOオプティマイザの潜在的な問い合わせ経路の管理方法において複数の非効率な点を修正しました。

- Allow `IN/NOT IN` to be handled via hash tables (Tom)
- Improve `NOT IN (subquery)` performance (Tom)
- Allow most `IN` subqueries to be processed as joins (Tom)
- Pattern matching operations can use indexes regardless of locale (Peter)

非ASCIIのロケールでは、`LIKE`比較用の標準的なインデックスを使用する方法がありません。このリリースでは`LIKE`用に特別なインデックスを作成する方法を追加しました。

- Allow the postmaster to preload libraries using `preload_libraries` (Joe)

長時間読み込まれる共有ライブラリでは、このオプションを利用でき、そのライブラリはpostmaster内であらかじめ読み込まれ、全てのデータベースセッションで継承されます。

- Improve optimizer cost computations, particularly for subqueries (Tom)
- Avoid sort when subquery `ORDER BY` matches upper query (Tom)
- Deduce that `WHERE a.x = b.y AND b.y = 42` also means `a.x = 42` (Tom)
- Allow hash/merge joins on complex joins (Tom)
- Allow hash joins for more data types (Tom)
- Allow join optimization of explicit inner joins, disable with `join_collapse_limit` (Tom)
- Add parameter `from_collapse_limit` to control conversion of subqueries to joins (Tom)
- Use faster and more powerful regular expression code from Tcl (Henry Spencer, Tom)

Use bit-mapped relation sets in the optimizer (Tom)

- Improve connection startup time (Tom)

新しいクライアント/サーバプロトコルでは、データベースセッション開始のために必要なネットワークパケットがより少なくなりました。

- Improve trigger/constraint performance (Stephan)
- Improve speed of `col IN (const, const, const, ...)` (Tom)
- Fix hash indexes which were broken in rare cases (Tom)
- Improve hash index concurrency and speed (Tom)

以前のリリースでは、特に、同時実行数が多い状況では、ハッシュインデックスの低性能が見られました。このリリースではそれを修正し、開発グループでは、B-treeとハッシュインデックスとの性能比較についての報告をまとめました。

- Align shared buffers on 32-byte boundary for copy speed improvement (Manfred Spraul)

特定のCPUの性能は、アドレスが32バイトで整列された時のデータコピーがより高速になります。

- Data type `numeric` reimplemented for better performance (Tom)

`numeric`は100を基として保存していました。かなり良い性能のために、新しいコードでは10000を基として使用します。

E.177.3.3. サーバ設定の変更

- Rename server parameter `server_min_messages` to `log_min_messages` (Bruce)

これは`log_`から始まるサーバのログを制御するほとんどのパラメータに対して行なわれました。

- Rename `show_*_stats` to `log_*_stats` (Bruce)
- Rename `show_source_port` to `log_source_port` (Bruce)
- Rename `hostname_lookup` to `log_hostname` (Bruce)
- Add `checkpoint_warning` to warn of excessive checkpointing (Bruce)

以前のリリースでは、チェックポイントが頻繁に発生し過ぎているかどうかを認識することは困難でした。この機能により、過度なチェックポイントの発生時にサーバログに警告が追加されます。

- New read-only server parameters for localization (Tom)
- Change debug server log messages to output as `DEBUG` rather than `LOG` (Bruce)
- Prevent server log variables from being turned off by non-superusers (Bruce)

非スーパーユーザが管理者によって有効にされたログ取得を無効にすることを不可能とする、セキュリティに関するものです。

- `log_min_messages/client_min_messages` now controls `debug_*` output (Bruce)

これはクライアントデバッグ情報を集中化し、これにより、全てのデバッグ出力がクライアントログもしくはサーバログに送信できます。

- Add Mac OS X Rendezvous server support (Chris Campbell)

これにより、Mac OS Xホストが利用可能なPostgreSQLサーバにネットワークから問い合わせできるようになります。

- Add ability to print only slow statements using `log_min_duration_statement` (Christopher)

これは、よく要求のあったデバッグ機能で、これにより、管理者はサーバログ内で低速な問い合わせのみを見られるようになります。

- Allow `pg_hba.conf` to accept netmasks in CIDR format (Andrew Dunstan)

これにより、管理者は`pg_hba.conf`内でホストIPアドレスとネットマスクフィールドを単一CIDRフィールドにまとめるこ

とができます。

- New read-only parameter `is_superuser` (Tom)
- New parameter `log_error_verbosity` to control error detail (Tom)

これは、新しいエラー報告機能と一緒に動作し、ヒントやファイル名、行番号などのエラー情報を更に提供します。

- `postgres --describe-config` now dumps server config variables (Aizaz Ahmed, Peter)

このオプションは、設定変数名とその最小、最大、デフォルト、説明文を把握する必要がある管理ツールで有用です。

- Add new columns in `pg_settings`: `context`, `type`, `source`, `min_val`, `max_val` (Joe)
- Make default `shared_buffers` 1000 and `max_connections` 100, if possible (Tom)

以前のバージョンではデフォルトで64共有バッファでした。そのため、非常に古いシステムでもPostgreSQLが起動できました。このリリースでは、プラットフォームで許可された共有メモリ量を試験し、可能であれば、より理想的なデフォルト値を選択します。もちろん、ユーザはリソースの使用状況を評価し、それに合わせて`shared_buffers`の大きさを決定することもまだ可能です。

- New `pg_hba.conf` record type `hostnossl` to prevent SSL connections (Jon Jensen)

以前のリリースでは、クライアントとサーバがSSLをサポートする場合、SSL接続を防ぐ方法がありませんでした。このオプションでその機能が可能です。

- Remove parameter `geqo_random_seed` (Tom)
- Add server parameter `regex_flavor` to control regular expression processing (Tom)
- Make `pg_ctl` better handle nonstandard ports (Greg)

E.177.3.4. 問い合わせに関する変更

- New SQL-standard information schema (Peter)
- Add read-only transactions (Peter)
- Print key name and value in foreign-key violation messages (Dmitry Tkach)
- Allow users to see their own queries in `pg_stat_activity` (Kevin Brown)

以前のリリースでは、スーパーユーザのみが`pg_stat_activity`を使用して問い合わせ文字列を確認することができました。通常のユーザが自身の問い合わせ文字列を確認できるようになりました。

- Fix aggregates in subqueries to match SQL standard (Tom)

標準SQLでは、入れ子になった副問い合わせ内の集約関数は、その引数に問い合わせ外部の変数のみが含まれる場合、その外部の問い合わせに属すると規定しています。以前のPostgreSQLのリリースでは、この点を正しく扱っていませんでした。

- Add option to prevent auto-addition of tables referenced in query (Nigel J. Andrews)

デフォルトでは、問い合わせ内にあり、`FROM`句にまだ存在しないテーブルは自動的に`FROM`句に追加されました。これは、歴史的なPOSTGRESの振舞いと互換性がありますが、標準SQLとは反対です。このオプションにより、標準と互換性がある動作を選択することができます。

- Allow `UPDATE ... SET col = DEFAULT` (Rod)

これにより`UPDATE`は列を宣言されたデフォルト値に設定することができます。

- Allow expressions to be used in `LIMIT/OFFSET` (Tom)

以前のリリースでは、`LIMIT/OFFSET`には定数のみを使用でき、式は使用できませんでした。

- Implement `CREATE TABLE AS EXECUTE` (Neil, Peter)

E.177.3.5. オブジェクト操作に関する変更

- Make `CREATE SEQUENCE` grammar more conforming to SQL:2003 (Neil)

- Add statement-level triggers (Neil)

これにより文の終了時にトリガを発行することができます。しかし、そのトリガは文によって変更された全ての行にアクセスすることはできません。この能力は将来のリリースで計画しています。

- Add check constraints for domains (Rod)

ドメインに検査制約を使用できるようになり、ドメインの有用性が非常に高くなりました。

- Add `ALTER DOMAIN` (Rod)

これにより既存のドメインの操作が可能です。

- Fix several zero-column table bugs (Tom)

PostgreSQLはゼロ列のテーブルをサポートします。そうしたテーブルの使用時に発生する各種の不具合を修正しました。

- Have `ALTER TABLE ... ADD PRIMARY KEY` add not-null constraint (Rod)

以前のリリースでは、`ALTER TABLE ... ADD PRIMARY`は一意性制約を追加しますが、非NULL制約を追加しませんでした。このリリースで修正されました。

- Add `ALTER TABLE ... WITHOUT OIDS` (Rod)

これにより新規行、更新された行がOID列を持つかどうかを制御することができます。これは、ほとんどの場合、格納領域の節約に有用です。

- Add `ALTER SEQUENCE` to modify minimum, maximum, increment, cache, cycle values (Rod)

- Add `ALTER TABLE ... CLUSTER ON` (Alvaro Herrera)

このコマンドは`pg_dump`で使用され、以前にクラスタ化された各テーブルのクラスタ用列を記録します。この情報はデータベース全体のクラスタによって、以前にクラスタ化されたテーブルをクラスタ化するために使用されます。

- Improve automatic type casting for domains (Rod, Tom)

- Allow dollar signs in identifiers, except as first character (Tom)

- Disallow dollar signs in operator names, so `x=$1` works (Tom)

- Allow copying table schema using `LIKE subtable`, also SQL:2003 feature `INCLUDING DEFAULTS` (Rod)

- Add `WITH GRANT OPTION` clause to `GRANT` (Peter)

これにより、`GRANT`は他のユーザにオブジェクトの権限を付与する能力を与えることができるようになります。

E.177.3.6. ユーティリティコマンドに関する変更

- Add `ON COMMIT` clause to `CREATE TABLE` for temporary tables (Gavin)

これにより、トランザクションのコミット時にテーブルの削除、もしくは、全行の削除を行なう機能が追加されました。

- Allow cursors outside transactions using `WITH HOLD` (Neil)

以前のリリースでは、カーソルはそのカーソルを作成したトランザクションの終端で削除されました。カーソルは`WITH HOLD`オプションを付けて作成できるようになり、これにより、作成したトランザクションがコミットされた後でもそのカーソルにアクセスできるようになります。

- `FETCH 0` and `MOVE 0` now do nothing (Bruce)

以前のリリースでは、`FETCH 0`は全ての行を取り出し、`MOVE 0`はカーソルの終端まで移動しました。

- Cause `FETCH` and `MOVE` to return the number of rows fetched/moved, or zero if at the beginning/end of cursor, per SQL standard (Bruce)

以前のリリースでは、`FETCH` と `MOVE` で返される行数は、実際に処理された行数を反映したものではありませんでした。

- Properly handle `SCROLL` with cursors, or report an error (Neil)

ある種の問い合わせに(前方、および、後方にスクロールする)ランダムアクセスを許可することは、何らかの追加作業なしでは行なうことができません。カーソル生成時に`SCROLL`が付与された場合、この追加作業が行なわれます。さらに、そのカーソルが`NO SCROLL`で生成された場合は、ランダムアクセスは許可されません。

- Implement SQL-compatible options `FIRST`, `LAST`, `ABSOLUTE n`, `RELATIVE n` for `FETCH` and `MOVE` (Tom)
- Allow `EXPLAIN` on `DECLARE CURSOR` (Tom)
- Allow `CLUSTER` to use index marked as pre-clustered by default (Alvaro Herrera)
- Allow `CLUSTER` to cluster all tables (Alvaro Herrera)

これにより、過去にクラスタ化されたテーブルは全て、単一コマンドで再度クラスタ化することができます。

- Prevent `CLUSTER` on partial indexes (Tom)
- Allow DOS and Mac line-endings in `COPY` files (Bruce)
- Disallow literal carriage return as a data value, backslash-carriage-return and `¥r` are still allowed (Bruce)
- `COPY` changes (binary, `¥.`) (Tom)
- Recover from `COPY` failure cleanly (Tom)
- Prevent possible memory leaks in `COPY` (Tom)
- Make `TRUNCATE` transaction-safe (Rod)

`TRUNCATE` がトランザクション内部で使えるようになりました。トランザクションが中断した場合、`TRUNCATE` で行なわれた変更は自動的にロールバックされます。

- Allow prepare/bind of utility commands like `FETCH` and `EXPLAIN` (Tom)
- Add `EXPLAIN EXECUTE` (Neil)
- Improve `VACUUM` performance on indexes by reducing WAL traffic (Tom)
- Functional indexes have been generalized into indexes on expressions (Tom)

以前のリリースでは、関数インデックスは、1つ以上の列名に適用される単一の関数のみをサポートしていました。このリリースでは、任意の種類のスカラ式を使用できます。

- Have `SHOW TRANSACTION ISOLATION` match input to `SET TRANSACTION ISOLATION` (Tom)
- Have `COMMENT ON DATABASE` on nonlocal database generate a warning, rather than an error (Rod)

データベースのコメントは、データベースに局所的なテーブルに保存されます。そのため、データベース上のコメントは各データベース内に保存されなければなりません。

- Improve reliability of `LISTEN/NOTIFY` (Tom)
- Allow `REINDEX` to reliably reindex nonshared system catalog indexes (Tom)

これにより、以前のリリースでは必要であった、スタンドアロンセッションの必要なしにシステムテーブルのインデックスを再作成できます。現時点で、インデックスの再作成のためにスタンドアロンセッションが必要なテーブルは、`pg_database`、`pg_shadow`、`pg_group`、グローバルシステムテーブルのみです。

E.177.3.7. データ型と関数に関する変更

- New server parameter `extra_float_digits` to control precision display of floating-point numbers (Pedro Ferreira, Tom)

これにより、レグレッション試験で問題の原因となった、出力制度を制御することができます。

- Allow `+1300` as a numeric time-zone specifier, for `FJST` (Tom)
- Remove rarely used functions `oidrand`, `oidsrand`, and `userfntest` functions (Neil)

- Add `md5()` function to main server, already in `contrib/pgcrypto` (Joe)
MD5関数がしばしば要求されました。より複雑な暗号化のためには`contrib/pgcrypto`を使用して下さい。
- Increase date range of `timestamp` (John Cochran)
- Change `EXTRACT(EPOCH FROM timestamp)` so `timestamp without time zone` is assumed to be in local time, not GMT (Tom)
- Trap division by zero in case the operating system doesn't prevent it (Tom)
- Change the `numeric` data type internally to base 10000 (Tom)
- New `hostmask()` function (Greg Wickham)
- Fixes for `to_char()` and `to_timestamp()` (Karel)
- Allow functions that can take any argument data type and return any data type, using `anyelement` and `anyarray` (Joe)
これにより、任意のデータ型で動作できる関数を作成できるようになります。
- Arrays can now be specified as `ARRAY[1,2,3]`, `ARRAY[['a','b'],['c','d']]`, or `ARRAY[ARRAY[ARRAY[2]]]` (Joe)
- Allow proper comparisons for arrays, including `ORDER BY` and `DISTINCT` support (Joe)
- Allow indexes on array columns (Joe)
- Allow array concatenation with `||` (Joe)
- Allow `WHERE` qualification `expr op ANY/SOME/ALL (array_expr)` (Joe)
これにより、`SELECT * FROM tab WHERE col IN (array_val)`のような目的のために、配列が値のリストのように振舞うことができます。
- New array functions `array_append`, `array_cat`, `array_lower`, `array_prepend`, `array_to_string`, `array_upper`, `string_to_array` (Joe)
- Allow user defined aggregates to use polymorphic functions (Joe)
- Allow assignments to empty arrays (Joe)
- Allow 60 in seconds fields of `time`, `timestamp`, and `interval` input values (Tom)
閏秒には60秒という値が必要でした。
- Allow `cidr` data type to be cast to `text` (Tom)
- Disallow invalid time zone names in `SET TIMEZONE`
- Trim trailing spaces when `char` is cast to `varchar` or `text` (Tom)
- Make `float(p)` measure the precision p in binary digits, not decimal digits (Tom)
- Add IPv6 support to the `inet` and `cidr` data types (Michael Graff)
- Add `family()` function to report whether address is IPv4 or IPv6 (Michael Graff)
- Have `SHOW datestyle` generate output similar to that used by `SET datestyle` (Tom)
- Make `EXTRACT(TIMEZONE)` and `SET/SHOW TIME ZONE` follow the SQL convention for the sign of time zone offsets, i.e., positive is east from UTC (Tom)
- Fix `date_trunc('quarter', ...)` (Böjthe Zoltán)
以前のリリースのこの関数の呼び出しは不正確な値を返しました。
- Make `initcap()` more compatible with Oracle (Mike Nolan)
`initcap()` は空白文字の直後だけではなく、非英数字の直後の文字を大文字にするようになりました。

- Allow only `datestyle` field order for date values not in ISO-8601 format (Greg)
- Add new `datestyle` values `MDY`, `DMY`, and `YMD` to set input field order; honor `US` and `European` for backward compatibility (Tom)
- String literals like `'now'` or `'today'` will no longer work as a column default. Use functions such as `now()`, `current_timestamp` instead. (change required for prepared statements) (Tom)
- Treat NaN as larger than any other value in `min()`/`max()` (Tom)

NaNは既にほとんどの目的で、通常の数値の後にソートされてましたが、`min()` と `max()` ではこれを正しく取っていませんでした。

- Prevent interval from suppressing `:00` seconds display
- New functions `pg_get_triggerdef(prettyprint)` and `pg_conversion_is_visible()` (Christopher)
- Allow time to be specified as `040506` or `0405` (Tom)
- Input date order must now be `YYYY-MM-DD` (with 4-digit year) or match `datestyle`
- Make `pg_get_constraintdef` support unique, primary-key, and check constraints (Christopher)

E.177.3.8. サーバ側の言語に関する変更

- Prevent PL/pgSQL crash when `RETURN NEXT` is used on a zero-row record variable (Tom)
- Make PL/Python's `spi_execute` interface handle null values properly (Andrew Bosma)
- Allow PL/pgSQL to declare variables of composite types without `%ROWTYPE` (Tom)
- Fix PL/Python's `_quote()` function to handle big integers
- Make PL/Python an untrusted language, now called `plpythonu` (Kevin Jacobs, Tom)

Python言語は制限付き実行環境をサポートしなくなりました。そのため、信頼されたバージョンのPL/Pythonは削除されました。この状態が変われば、非スーパーユーザで使用可能なバージョンのPL/Pythonは再び追加されるでしょう。

- Allow polymorphic PL/pgSQL functions (Joe, Tom)
- Allow polymorphic SQL functions (Joe)
- Improved compiled function caching mechanism in PL/pgSQL with full support for polymorphism (Joe)
- Add new parameter `$0` in PL/pgSQL representing the function's actual return type (Joe)
- Allow PL/Tcl and PL/Python to use the same trigger on multiple tables (Tom)
- Fixed PL/Tcl's `spi_prepare` to accept fully qualified type names in the parameter type list (Jan)

E.177.3.9. psql の変更

- Add `¥pset pager` always to always use pager (Greg)

これにより行数がスクリーンの高さよりも少なかったとしても強制的にページャを使用します。これは、複数のスクリーンの行を越えた行では価値があります。

- Improve tab completion (Rod, Ross Reedstrom, Ian Barwick)
- Reorder `¥?` help into groupings (Harald Armin Massa, Bruce)
- Add backslash commands for listing schemas, casts, and conversions (Christopher)
- `¥encoding` now changes based on the server parameter `client_encoding` (Tom)

以前のリリースでは、`SET client_encoding`を使用した符号化方式の変更は`¥encoding`に影響しませんでした。

- Save editor buffer into readline history (Ross)

`\e`を使用して問い合わせを編集する場合、その結果はreadlineの履歴に保存され、上矢印キーを使用して取り出すことができます。

- Improve `\d` display (Christopher)
- Enhance HTML mode to be more standards-conforming (Greg)
- New `\set AUTOCOMMIT off` capability (Tom)

これは、削除された`autocommit`サーバパラメータの代わりになります。

- New `\set VERBOSITY` to control error detail (Tom)
これは新しいエラー報告の詳細を制御します。
- New prompt escape sequence `%x` to show transaction status (Tom)
- Long options for `psql` are now available on all platforms

E.177.3.10. `pg_dump` の変更

- Multiple `pg_dump` fixes, including tar format and large objects
- Allow `pg_dump` to dump specific schemas (Neil)
- Make `pg_dump` preserve column storage characteristics (Christopher)
これは`ALTER TABLE ... SET STORAGE`の情報を保持します。
- Make `pg_dump` preserve `CLUSTER` characteristics (Christopher)
- Have `pg_dumpall` use `GRANT/REVOKE` to dump database-level privileges (Tom)
- Allow `pg_dumpall` to support the options `-a`, `-s`, `-x` of `pg_dump` (Tom)
- Prevent `pg_dump` from lowercasing identifiers specified on the command line (Tom)
- `pg_dump` options `--use-set-session-authorization` and `--no-reconnect` now do nothing, all dumps use `SET SESSION AUTHORIZATION`

`pg_dump`はユーザを切替えるための再接続を行なわなくなりました。その代わりに常に`SET SESSION AUTHORIZATION`を使用します。これはリストア時のパスワード問い合わせを減少させます。

- Long options for `pg_dump` are now available on all platforms
PostgreSQLはその内部に長い名前のオプションを処理する関数を含むようになりました。

E.177.3.11. `libpq` の変更

- Add function `PQfreemem` for freeing memory on Windows, suggested for `NOTIFY` (Bruce)

Windowsは、あるライブラリで割り当てられたメモリを同じライブラリ内の関数で解放することを要求します。そのため、`free()`では、`libpq`で割り当てられたメモリの解放を行なうことができません。特にWindowsでは`PQfreemem`が`libpq`のメモリを解放する適切な方法です。そして、これは他のプラットフォームでも推奨されます。

- Document service capability, and add sample file (Bruce)
これにより、クライアントはクライアントマシン上で中央にまとめられたファイル内の接続情報を検索することができます。
- Make `PQsetdbLogin` have the same defaults as `PQconnectdb` (Tom)
- Allow `libpq` to cleanly fail when result sets are too large (Tom)
- Improve performance of function `PQunescapeBytea` (Ben Lamb)
- Allow thread-safe `libpq` with `configure` option `--enable-thread-safety` (Lee Kindness, Philip Yarra)
- Allow function `pqInternalNotice` to accept a format string and arguments instead of just a preformatted

message (Tom, Sean Chittenden)

- Control SSL negotiation with `sslmode` values `disable`, `allow`, `prefer`, and `require` (Jon Jensen)
- Allow new error codes and levels of text (Tom)
- Allow access to the underlying table and column of a query result (Tom)

指定した結果セットに関連する、背後にあるテーブルと列の名前を把握したい問い合わせ構築ツールで役に立ちます。

- Allow access to the current transaction status (Tom)
- Add ability to pass binary data directly to the server (Tom)
- Add function `PQexecPrepared` and `PQsendQueryPrepared` functions which perform bind/execute of previously prepared statements (Tom)

E.177.3.12. JDBC に関する変更

- Allow `setNull` on updateable result sets
- Allow `executeBatch` on a prepared statement (Barry)
- Support SSL connections (Barry)
- Handle schema names in result sets (Paul Sorenson)
- Add refcursor support (Nic Ferrier)

E.177.3.13. インタフェースに関する雑多な変更

- Prevent possible memory leak or core dump during libpqctl shutdown (Tom)
- Add Informix compatibility to ECPG (Michael)
これにより、ECPGは、Informix拡張を使用して作成された組み込みCプログラムを処理できます。
- Add type `decimal` to ECPG that is fixed length, for Informix (Michael)
- Allow thread-safe embedded SQL programs with `configure` option `--enable-thread-safety` (Lee Kindness, Bruce)
これにより、複数のスレッドが同時にデータベースにアクセスできます。
- Moved Python client PyGreSQL to <http://www.pygresql.org> (Marc)

E.177.3.14. ソースコードに関する変更

- Prevent need for separate platform geometry regression result files (Tom)
- Improved PPC locking primitive (Reinhard Max)
- New function `palloc0` to allocate and clear memory (Bruce)
- Fix locking code for s390x CPU (64-bit) (Tom)
- Allow OpenBSD to use local ident credentials (William Ahern)
- Make query plan trees read-only to executor (Tom)
- Add Darwin startup scripts (David Wheeler)
- Allow libpq to compile with Borland C++ compiler (Lester Godwin, Karl Waclawek)
- Use our own version of `getopt_long()` if needed (Peter)
- Convert administration scripts to C (Peter)
- Bison `>= 1.85` is now required to build the PostgreSQL grammar, if building from CVS

- Merge documentation into one book (Peter)
- Add Windows compatibility functions (Bruce)
- Allow client interfaces to compile under MinGW (Bruce)
- New `ereport()` function for error reporting (Tom)
- Support Intel compiler on Linux (Peter)
- Improve Linux startup scripts (Slawomir Sudnik, Darko Prenosil)
- Add support for AMD Opteron and Itanium (Jeffrey W. Baker, Bruce)
- Remove `--enable-recode` option from `configure`

`CREATE CONVERSION` がありますので、これはもはや不要です。

- Generate a compile error if spinlock code is not found (Bruce)

スピンロック用のコードがないプラットフォームは、セマフォをこっそりと使用するのではなく、コンパイルに失敗するようになります。この失敗は、新しい`configure`オプションで無効にすることができます。

E.177.3.15. 寄贈物に関する変更

- Change dbmirror license to BSD
- Improve earthdistance (Bruno Wolff III)
- Portability improvements to pgcrypto (Marko Kreen)
- Prevent crash in xml (John Gray, Michael Richards)
- Update oracle
- Update mysql
- Update cube (Bruno Wolff III)
- Update earthdistance to use cube (Bruno Wolff III)
- Update btree_gist (Oleg)
- New tsearch2 full-text search module (Oleg, Teodor)
- Add hash-based crosstab function to tablefuncs (Joe)
- Add serial column to order `connectby()` siblings in tablefuncs (Nabil Sayegh, Joe)
- Add named persistent connections to dblink (Shridhar Daithanka)
- New `pg_autovacuum` allows automatic `VACUUM` (Matthew T. O'Connor)
- Make pgbench honor environment variables `PGHOST`, `PGPORT`, `PGUSER` (Tatsuo)
- Improve intarray (Teodor Sigaev)
- Improve pgstattuple (Rod)
- Fix bug in `metaphone()` in `fuzzystrmatch`
- Improve adddepend (Rod)
- Update spi/timetravel (Böjthe Zoltán)
- Fix `dbase -s` option and improve non-ASCII handling (Thomas Behr, Márcio Smiderle)
- Remove array module because features now included by default (Joe)

E.178. リリース7.3.21

リリース日: 2008-01-07

このリリースは、重大なセキュリティ問題の修正を含む、7.3.20の各種不具合を修正したものです。

これが7.3.X系の最後のPostgreSQLリリースとなる予定です。 ユーザはより新しいリリースブランチに早めの更新を行うことが推奨されています。

E.178.1. バージョン7.3.21への移行

7.3.Xからの移行ではダンプ/リストアは不要です。 しかし、7.3.13より前のバージョンからアップグレードする場合は、7.3.13のリリースノートを参照してください。

E.178.2. 変更点

- インデックス内の関数がVACUUM、ANALYZEなどを実行するユーザの権限で実行されることを防止します。(Tom)

インデックス式内で使用される関数および部分インデックスの述語は、新しいテーブル項目が作成されると評価されます。 これが、もし信頼できないユーザが所有するテーブルを変更したとすると、トロイの木馬となるコードの実行という危険性を引き起こすことは以前から解っていました。(トリガ、デフォルト、検査制約なども同種の危険性を引き起こします。) しかしインデックス内の関数は、VACUUM FULLなどの日常の保守作業で実行される可能性があるため、さらに危険です。 こうした保守作業は通常自動的にスーパーユーザ権限で行われます。 例えば、極悪なユーザがトロイの木馬となるインデックス定義を設定し、次回の日常のバキューム処理を待つことで、スーパーユーザ権限でコードを実行することが可能です。 この修正は、SECURITY DEFINER関数で使われていた権限変更機構を使用して、標準的な保守作業(VACUUM、ANALYZE、REINDEX、CLUSTERを含む)が呼び出したユーザではなくテーブル所有者として実行するように調整しました。 こうしたセキュリティ制限の迂回を防ぐために、SET SESSION AUTHORIZATIONおよびSET ROLEがSECURITY DEFINER コンテキストで禁止されるようになりました。(CVE-2007-6600)

- /contrib/dblinkを使用して、パスワード認証のみをセキュリティ制限として使用するには非スーパーユーザでなければなりません。

7.3.20で行った修正は不十分で、一部のdblink関数に対してのみ問題を解消していました。(CVE-2007-6601,CVE-2007-3278)

- マルチバイトデータベース符号化方式を使用している時に潜在する、translate()のクラッシュを修正しました。
- contrib/tablefuncのcrosstab()が、クラッシュするのではなく、独自の意味でカテゴリとしてNULL行識別子を扱うようにしました。(Joe)
- configureスクリプトを再生成する場合、特定のバージョンのAutoconfを使用することが必要になりました。(Peter)

これは開発者とパッケージ作成者のみに影響します。 この変更は、AutoconfとPostgreSQLの未確認のバージョンの組み合わせを使用する事故を防ぐためになされました。 その結果がうまくいくかどうかについて責任を負う限り、異なるバージョンのAutoconfを本当に使用したい場合は、このバージョン検査を取り除くことができます。

E.179. リリース7.3.20

リリース日: 2007-09-17

このリリースは7.3.19の不具合を修正したものです。

E.179.1. バージョン7.3.20への移行

7.3.Xからの移行ではダンプ/リストアは不要です。しかし、7.3.13より前のバージョンからアップグレードする場合は、7.3.13のリリースノートを参照してください。

E.179.2. 変更点

- トランザクションが、行を挿入した後に、同一テーブルに対する同時実行中のVACUUMの終了時刻とほぼ同じ時点でアポートした場合のインデックス破損を防止します。(Tom)
- CREATE DOMAIN ... DEFAULT NULLが適切に動作するようにしました。(Tom)
- log_min_error_statementログ処理がメモリ不足になった時のクラッシュを修正しました。(Tom)
- セキュリティ強化のため、パスワード認証のみを使用する/contrib/dblinkを使用するユーザが非特権ユーザであることを要求します。(Joe)

E.180. リリース7.3.19

リリース日: 2007-04-23

このリリースは7.3.18の不具合を修正したもので、セキュリティ問題の修正を1つ含みます。

E.180.1. バージョン7.3.19への移行

7.3.Xからの移行ではダンプ/リストアは不要です。しかし、7.3.13より前のバージョンからアップグレードする場合は、7.3.13のリリースノートを参照してください。

E.180.2. 変更点

- `search_path`における一時テーブルスキーマの明示的な指定をサポートしました。また、関数や演算子ではその検索を無効にしました(Tom)。

これは、`SECURITY DEFINER`関数が`search_path`の値を本当に安全に設定するために必要です。こうしないと、一時オブジェクトを使用して権限のないSQLユーザが`SECURITY DEFINER`関数の権限でコードを実行することができてしまうためです (CVE-2007-2138)。詳細は`CREATE FUNCTION`を参照してください。

- `VACUUM FULL`の`UPDATE`連鎖方法において潜在したデータ破損不具合を修正しました(Tom, Pavan Deolasee)。

E.181. リリース 7.3.18

リリース日: 2007-02-05

このリリースは7.3.17の各種不具合を修正したもので、セキュリティ問題の修正も含まれます。

E.181.1. バージョン7.3.18への移行

7.3.Xからの移行ではダンプ/リストアは不要です。しかし、7.3.13より前のバージョンからアップグレードする場合は、7.3.13のリリースノートを参照してください。

E.181.2. 変更点

- 接続ユーザがバックエンドのメモリを読み取ることができるセキュリティ脆弱性を取り除きました。(Tom)

この脆弱性には、SQL関数が宣言されたデータ型を返すかどうか、そのテーブル列のデータ型が変更されたかどうかに関する通常の検査の抑制も含まれています (CVE-2007-0555、CVE-2007-0556)。これらのエラーを悪用して簡単にバックエンドをクラッシュさせることができます。また原理的には、アクセスを許していないはずのユーザがデータベースの内容を読み取ることができてしまいます。

- 実行不可能な分割点の選択によりB-Treeインデックスページの分割が失敗する可能性がある、稀に起こる不具合を修正しました。(Heikki Linnakangas)
- 3バイト長を越えるUTF8シーケンスに関する、複数バイト文字処理のセキュリティを強化しました。(Tom)

E.182. リリース 7.3.17

リリース日: 2007-01-08

このリリースは7.3.16の各種不具合を修正したものです。

E.182.1. バージョン7.3.17への移行

7.3.Xからの移行ではダンプ/リストアは不要です。しかし、7.3.13より前のバージョンからアップグレードする場合は、7.3.13のリリースノートを参照してください。

E.182.2. 変更点

- 新しくinitdbを行ったインストレーションでは、`to_number()` と `to_char(numeric)` が `IMMUTABLE` ではなく `STABLE` になりました。(Tom)

これは、`lc_numeric` がこれらの関数の出力を変更する可能性があるためです。

- 括弧を使用した正規表現を使用するインデックスを改良しました。(Tom)

この改良により、`psql \d` の性能も向上しました。

E.183. リリース7.3.16

リリース日: 2006-10-16

このリリースは7.4.13の各種不具合を修正したものです。

E.183.1. バージョン7.3.16への移行

7.3.Xからの移行ではダンプ/リストアは不要です。しかし、7.3.13より前のバージョンからアップグレードする場合は、7.3.13のリリースノートを参照してください。

E.183.2. 変更点

- `psql's \d` コマンドに対するパターンマッチにおけるちょっとした部分を修正しました。
- `/contrib/ltree` におけるインデックス破損不具合を修正しました。(Teodor)
- 64bitアーキテクチャの性能とサポートをより向上させるために、7.4のスピンロックコードを移植しました。
- `libpq` におけるSSLに関するメモリリークを修正しました。
- `/contrib/dbmirror` におけるバックスラッシュエスケープ処理を修正しました。
- リグレッション試験をUS DST法の最近の変更に合わせました。

E.184. リリース7.3.15

リリース日: 2006-05-23

このリリースは7.3.14の各種不具合を修正したもので、非常に深刻なセキュリティ問題を解消するパッチを含みます。

E.184.1. バージョン7.3.15への移行

7.3.Xからの移行ではダンプ/リストアは不要です。しかし、7.3.13より前のバージョンからアップグレードする場合は、7.3.13のリリースノートを参照してください。

CVE-2006-2313およびCVE-2006-2314に示されたSQLインジェクション攻撃を完全に防ぐためには、アプリケーション側のコードの変更が必要となる場合があります。SQLコマンド内に信頼できない文字列を埋め込むアプリケーションでは、できる限り早く、その文字列が推奨するエスケープ技法を使用していることを確実に検証しなければなりません。ほとんどの場合、アプリケーションは文字列のエスケープ処理に、その場しのぎのコードではなく、ライブラリやドライバが提供する (libpqのPQescapeStringConn()) のような関数を使用しなければなりません。

E.184.2. 変更点

- すべての場合において、符号化が無効なマルチバイト文字を拒絶するようにサーバを変更しました。(Tatsuo, Tom)

PostgreSQLは少し前からこのような方向に移行していましたが、この検査がすべての符号化方式とすべてのテキスト入力に対して統一的に適用されるようになりました。さらに、単なる警告ではなく常にエラーとなるようになりました。この変更はCVE-2006-2313で示されるような種類のSQLインジェクション攻撃から保護します。

- 文字列リテラル内の`\`の安全ではない使用を拒絶します。

CVE-2006-2314で示されるような種類のSQLインジェクション攻撃からのサーバ側の保護として、サーバは、SQL文字列リテラル内のASCII単一引用符の表現として、`'`のみを受け、`\"`を受けないようになりました。デフォルトでは、SQLインジェクションが可能となる状況である、`client_encoding`がクライアント側のみの符号化方式 (SJIS、BIG5、GBK、GB18030、UHC) に設定された場合にのみ`\"`は拒絶されます。新しい設定パラメータ`backslash_quote`により、必要な場合にこの動作を調整できます。CVE-2006-2314に対して完全に保護するには、クライアント側を変更する必要があるかもしれないことに注意してください。安全ではないクライアントを安全ではないものとして明らかにすることが、`backslash_quote`の目的の一つです。

- libpqの文字列エスケープルーチンを、符号化方式と`standard_conforming_strings`を考慮するように変更しました。

これは、CVE-2006-2313およびCVE-2006-2314で示されるセキュリティ問題に対し、libpqを使用したアプリケーションを修正します。また、将来予定されるSQL標準文字列リテラル構文への移行に対しても保護しています。同時に複数のPostgreSQL接続を使用するアプリケーションは、各データベース接続において使用される設定に合わせてエスケープ処理が正しく行われるように、`PQescapeStringConn()`と`PQescapeByteaConn()`に移行しなければなりません。"独自に"文字列エスケープ処理を行うアプリケーションはライブラリルーチンを使用するように変更しなければなりません。

- 一部の不正な符号化方式変換関数を修正しました。

`win1251_to_iso`、`win866_to_iso`、`euc_tw_to_big5`、`euc_tw_to_mic`、`mic_to_euc_tw`はすべて可変拡張に関して正しくありませんでした。

- 文字列における偶然残る`\"`の使用を整理しました。(Bruce, Jan)
- 独自のDH SSLパラメータを正しく使用するようにサーバを修正しました。(Michael Fuhr)
- さまざまな小規模のメモリリークを修正しました。

E.185. リリース7.3.14

リリース日: 2006-02-14

このリリースは7.3.13の各種不具合を修正したものです。

E.185.1. バージョン7.3.14への移行

7.3.Xからの移行ではダンプ/リストアは不要です。しかし、7.3.13より前のバージョンからアップグレードする場合は、7.3.13のリリースノートを参照してください。

E.185.2. 変更点

- `SET SESSION AUTHORIZATION`によるクラッシュの可能性を修正しました。(CVE-2006-0553)

アサートを有効にしてコンパイルした場合（これはデフォルトではありません）、非特権ユーザがサーバをクラッシュでき、その結果、他のユーザへのサービスが一時的に不能になります。この問題を報告してくれたAkio Ishidaに感謝します。

- 自身が挿入した行に関する行の可視性ロジックについての不具合を修正しました。(Tom)

非常に稀な状況で、現在のコマンドが挿入した行が、本来無効であるべき時に既に有効になっているものとして扱われることがありました。リリース7.3.11で組み込まれた不具合を修正しました。

- `pg_clog`および`pg_subtrans`ファイル生成時に"ファイルが既に存在する"というエラーを引き起こす競合条件を修正しました。(Tom)
- 独自の演算子や演算子クラスを参照するクロススキーマを持つダンプをリストアできるように修正しました。(Tom)
- `configure`時の`finite`および`isinf`の存在検査に関する移植性を修正しました。(Tom)

E.186. リリース7.3.13

リリース日: 2006-01-09

このリリースは7.3.12の各種不具合を修正したものです。

E.186.1. バージョン7.3.13への移行

7.3.Xからの移行ではダンプ/リストアは不要です。しかし、7.3.10より以前のバージョンからアップグレードする場合は、7.3.10のリリースノートを参照してください。また、後述のロケールやplperlの問題の影響がある場合、移行後にテキストの列に対するインデックスをREINDEXする必要があるかもしれません。

E.186.2. 変更点

- 韓国語など、異なる文字の組み合わせを同一のものとみなすロケールにおける文字列比較を修正しました。(Tom)

既存のテキスト列に対するインデックスを修正するにはREINDEXが必要になります。

- plperlが後でロケールを変更できないように、postmasterの起動時にロケール用の環境変数を設定します。

これは、initdb実行時に通知したロケールと異なるロケールを表す環境変数の下でpostmasterが起動された場合に発生する問題を修正します。このような条件では、plperlの使用によりインデックスが破損される可能性があります。こうした問題が発生していた場合、既存のテキスト列に対するインデックスのREINDEXが必要になります。

- 長期間存在した、ごく一部の特定のアジア向けマルチバイト文字セットを取り扱う場合のstrpos()と正規表現の不具合を修正しました。(Tatsuo)
- MD5およびXDESアルゴリズム用のソルト空間をすべて利用できないという、/contrib/pgcryptoのgen_saltの不具合を修正しました。(Marko Kreen, Solar Designer)

Blowfishおよび標準DES用のソルトには影響ありません。

- 指定した列数が実際に問い合わせで返される列数と異なる場合にクラッシュせずにエラーを返すように/contrib/dblinkを修正しました。(Joe)

E.187. リリース7.3.12

リリース日: 2005-12-12

このリリースは7.3.11の各種不具合を修正したものです。

E.187.1. バージョン7.3.12への移行

7.3.Xからの移行ではダンプ/リストアは不要です。しかし、7.3.10より以前のバージョンからアップグレードする場合は、7.3.10のリリースノートを参照してください。

E.187.2. 変更点

- トランザクションログ管理における競合条件を修正しました。

I/O操作により間違ったページが初期化されてしまうわずかな状態がありました。これは、アサート失敗またはデータ破壊をもたらします。

- `/contrib/ltree`を修正しました。(Teodor)
- 長期間残っていた、外部結合の実行計画作成エラーを修正しました。

この不具合により、"マージ結合可能な結合条件のみをサポートするRIGHT JOIN"という不正なエラーが発生することがありました。

- テーブルが削除された場合のpg_autovacuumのコアダンプを防止します。

E.188. リリース 7.3.11

リリース日: 2005-10-04

このリリースは7.3.10の各種不具合を修正したものです。

E.188.1. バージョン7.3.11への移行

7.3.Xからの移行ではダンプ/リストアは不要です。しかし、7.3.10より以前のバージョンからアップグレードする場合は、7.3.10のリリースノートを参照してください。

E.188.2. 変更点

- `VACUUM`によって`ctid`チェーンがすぐに削除されてしまうエラーを修正し、`ctid`リンクを追跡するコードにおいてより多くの検査を行うようにしました。

これにより、長い間存在した、非常に稀な環境におけるクラッシュという問題が修正されました。

- マルチバイト文字セットを使用する場合に、`CHAR()`の空白文字埋め込みが適切に指定した長さになるように修正しました。(Yoshiyuki Asaba)

以前のリリースでは、`CHAR()`の空白文字埋め込みが正しくありませんでした。格納する文字数を考慮せずに、単に指定されたバイト数で空白文字を埋め込んでいただけでした。

- `a`列にGiSTインデックスを持つ場合の`UPDATE a=... WHERE a...`といった問い合わせにおいて行が見つからない点を修正しました。
- 部分的に書き込まれたWALページに対する検査を改良しました。
- SSL有効時のシグナルの扱いに対する堅牢性を向上しました。
- さまざまなメモリリークを修正しました。
- 移植性の向上をいろいろと行いました。
- PL/pgsqlにおいて、変数が参照渡し型の場合の`var := var`の扱いを修正しました。

E.189. リリース7.3.10

リリース日: 2005-05-09

このリリースは、複数のセキュリティ関連の問題を含む、7.3.9の各種不具合を修正したものです。

E.189.1. バージョン7.3.10への移行

7.3.Xからの移行ではダンプ/リストアは不要です。しかし、7.3.Xのシステムカタログで見つかった重大なセキュリティ問題を突かれる可能性があります。ダンプ、7.3.10のinitdbを使用したinitdb、リロードを行うことで、自動的にこれらの問題を修正します。

セキュリティ問題は、組み込みの文字セット符号化変換関数により、権限を持たないユーザがSQLコマンドを呼び出すことができるという点です。このような用途のためにこれらの関数を設計していませんでしたが、悪意のある引数の設定に対する安全性がありませんでした。この修正により、これらの関数の宣言されたパラメータリストがSQLコマンドから呼び出されないように変更されました。（通常の符号化変換機構の使用には影響はありません。）initdb、もしくは、後述の手作業による修正手順に従って、すべてのインストレーションにおいてこれらのエラーを修正することを強く勧めます。これらのエラーにより、少なくとも、権限を持たないデータベースユーザがサーバプロセスをクラッシュさせることができます。また、権限を持たないユーザがデータベーススーパーユーザ権限を手に入れることができる可能性もあります。

initdbを行いたくないのであれば、スーパーユーザ権限で以下の手続きを行ってください。

```
BEGIN;
UPDATE pg_proc SET proargtypes[3] = 'internal'::regtype
WHERE pronamespace = 11 AND pronargs = 5
    AND proargtypes[2] = 'cstring'::regtype;
-- このコマンドは90行を更新したと報告するはずです。
-- 異なる場合は、コミットせずにロールバックして調査を行ってください。
COMMIT;
```

上の手続きを、`template1`を含むインストレーション内のすべてのデータベースで行わなければなりません。理想を言えば、`template0`に対しても実施してください。テンプレートデータベースで修正を行わなかった場合、この後に作成されるデータベースにこのエラーが含まれてしまいます。`template1`は他のデータベースと同じ方法で修正できますが、`template0`では更に行わなければならないことがあります。まず、任意のデータベースから以下を実行してください。

```
UPDATE pg_database SET dataallowconn = true WHERE datname = 'template0';
```

次に、`template0`に接続し、上記修正手続きを実施してください。最後に以下を実行してください。

```
-- 再度template0を凍結させます
VACUUM FREEZE;
-- そして、今後の変更に対し保護します。
UPDATE pg_database SET dataallowconn = false WHERE datname = 'template0';
```

E.189.2. 変更点

- 誤使用を防ぐために符号化関数のシグネチャを変更しました。
- ある目的（例えばSELECT FOR UPDATE）用にコミットされるように見えるトランザクションが直後に別の目的用にコミットされるという従来から存在した競合状況を修正しました。

一貫性が明らかにないデータが短期間アプリケーションから参照できてしまうため、これはきわめて重大な不具合です。

- リレーションの拡張とVACUUMとの間の競合状態を修正しました。

理論上、これにより新たに挿入されたページの価値が失われます。しかしこうした状況が発生する頻度はきわめてまれで、アサートエラー以外に発生した事例はありません。

- TIME WITH TIME ZONE型の値の比較を修正しました。

--enable-integer-datetimes設定スイッチが使用されている場合、この比較コードに間違いがありました。注意: この修正により列値のソート順が正しくなるように変更されましたので、TIME WITH TIME ZONE型の列に対するインデックスを持つ場合、この更新をインストールした後¥にREINDEXを行う必要があります。

- TIME WITH TIME ZONE型の値に対するEXTRACT(EPOCH)を修正しました。

- `INTERVAL`型の値の秒の小数部分において、負の値が間違っ表示される点を修正しました。
このエラーは、`--enable-integer-datetimes`設定スイッチが使用された場合にのみ発生します。
- `plpgsql`において更にバッファオーバーランの検査を追加しました。(Neil)
- `%`を名前含むトリガを正しくダンプできるように`pg_dump`を修正しました。
- 月に関連した書式のために`to_char(interval)`がコアダンプすることを防止しました。
- より最近のOpenSSLによる構築のために`contrib/pgcrypto`を修正しました。(Marko Kreen)
- `contrib/intagg`に対する更なる64ビット向けの修正を行いました。
- `RECORD`型を返す関数の最適化の間違いを修正しました。

[前のページ](#)
リリース 7.3.11

[ホーム](#)
[上に戻る](#)

[次のページ](#)
リリース7.3.9

E.190. リリース7.3.9

リリース日: 2005-01-31

このリリースは複数のセキュリティ問題を含む、7.3.8の各種不具合を修正したものです。

E.190.1. バージョン7.3.9への移行

7.3.Xからの移行の場合は ダンプ/リストアは必要ありません。

E.190.2. 変更点

- Disallow `LOAD` to non-superusers

On platforms that will automatically execute initialization functions of a shared library (this includes at least Windows and ELF-based Unixen), `LOAD` can be used to make the server execute arbitrary code. Thanks to NGS Software for reporting this.

- Check that creator of an aggregate function has the right to execute the specified transition functions

This oversight made it possible to bypass denial of `EXECUTE` permission on a function.

- Fix security and 64-bit issues in contrib/intagg
- Add needed `STRICT` marking to some contrib functions (Kris Jurka)
- Avoid buffer overrun when plpgsql cursor declaration has too many parameters (Neil)
- Fix planning error for `FULL` and `RIGHT` outer joins

The result of the join was mistakenly supposed to be sorted the same as the left input. This could not only deliver mis-sorted output to the user, but in case of nested merge joins could give outright wrong answers.

- Fix plperl for quote marks in tuple fields
- Fix display of negative intervals in SQL and GERMAN datestyles

E.191. リリース7.3.8

リリース日: 2004-10-22

このリリースは7.3.7の各種不具合を修正したものです。

E.191.1. バージョン7.3.8への移行

7.3.Xからの移行の場合は ダンプ/リストアは必要ありません。

E.191.2. 変更点

- ディスク上のヒントビットの更新に失敗する可能性を修正

稀な環境下で、この誤りにより"could not access transaction status"という失敗になる可能性があります。 これはデータ損失の可能性がある不具合です。

- ハッシュ外部結合がタプルを喪失しないことを確実にしました。

ハッシュ結合計画を使用する非常に大規模な左結合が、右側のデータ分布のみが与えられた一致しない左側の行を出力する可能性があります。

- rootによるpg_ctlの実行を禁止

潜在するセキュリティ問題から保護します。

- make_oidjoins_checkにおける/tmp内の中間ファイルの使用を避けました。

これはセキュリティ問題として報告されたものです。 しかし、開発者以外がこのスクリプトを使用する理由がありませんので、ほとんど気にする必要はありません。

E.192. リリース7.3.7

リリース日: 2004-08-16

このリリースは7.3.6の深刻な不具合を改修したものです。その他若干の修正があります。

E.192.1. バージョン7.3.7への移行

7.3.X からの移行の場合は ダンプ/リストアは必要ありません。

E.192.2. 変更点

- クラッシュ時のコミット済みのトランザクションの欠落の可能性を修正

トランザクションのコミットとチェックポイント処理との間の内部ロック処理が不十分であったため、最近のチェックポイント直前にコミットされたトランザクションの全部あるいは一部がデータベースのクラッシュの後の再起動時に欠落する可能性があります。この深刻な不具合はPostgreSQL 7.1以降から存在していました。

- Remove asymmetrical word processing in tsearch (Teodor)
- Properly schema-qualify function names when pg_dump'ing a CAST

E.193. リリース7.3.6

リリース日: 2004-03-02

7.3.5の様々な不具合を修正しました。

E.193.1. バージョン7.3.6への移行

7.3.* からの移行の場合は ダンプ/リストアは必要ありません。

E.193.2. 変更点

- Revert erroneous changes in rule permissions checking

A patch applied in 7.3.3 to fix a corner case in rule permissions checks turns out to have disabled rule-related permissions checks in many not-so-corner cases. This would for example allow users to insert into views they weren't supposed to have permission to insert into. We have therefore reverted the 7.3.3 patch. The original bug will be fixed in 8.0.

- Repair incorrect order of operations in GetNewTransactionId()

This bug could result in failure under out-of-disk-space conditions, including inability to restart even after disk space is freed.

- Ensure configure selects -fno-strict-aliasing even when an external value for CFLAGS is supplied

On some platforms, building with -fstrict-aliasing causes bugs.

- Make pg_restore handle 64-bit off_t correctly

This bug prevented proper restoration from archive files exceeding 4 GB.

- Make contrib/dblink not assume that local and remote type OIDs match (Joe)

- Quote connectby()'s start_with argument properly (Joe)

- Don't crash when a rowtype argument to a plpgsql function is NULL

- Avoid generating invalid character encoding sequences in corner cases when planning LIKE operations

- Ensure text_position() cannot scan past end of source string in multibyte cases (Korea PostgreSQL Users' Group)

- Fix index optimization and selectivity estimates for LIKE operations on bytea columns (Joe)

E.194. リリース 7.3.5

リリース日: 2003-12-03

7.3.4の様々な不具合を修正しました。

E.194.1. バージョン7.3.5への移行

7.3.X からの移行の場合は ダンプ/リストアは必要ありません。

E.194.2. 変更点

- Force zero_damaged_pages to be on during recovery from WAL
- Prevent some obscure cases of "variable not in subplan target lists"
- Force stats processes to detach from shared memory, ensuring cleaner shutdown
- Make PQescapeBytea and byteaout consistent with each other (Joe)
- Added missing SPI_finish() calls to dblink's get_tuple_of_interest() (Joe)
- Fix for possible foreign key violation when rule rewrites INSERT (Jan)
- Support qualified type names in PL/Tcl's spi_prepare command (Jan)
- Make pg_dump handle a procedural language handler located in pg_catalog
- Make pg_dump handle cases where a custom opclass is in another schema
- Make pg_dump dump binary-compatible casts correctly (Jan)
- Fix insertion of expressions containing subqueries into rule bodies
- Fix incorrect argument processing in clusterdb script (Anand Ranganathan)
- Fix problems with dropped columns in plpython triggers
- Repair problems with to_char() reading past end of its input string (Karel)
- Fix GB18030 mapping errors (Tatsuo)
- Fix several problems with SSL error handling and asynchronous SSL I/O
- Remove ability to bind a list of values to a single parameter in JDBC (prevents possible SQL-injection attacks)
- Fix some errors in HAVE_INT64_TIMESTAMP code paths
- Fix corner case for btree search in parallel with first root page split

E.195. リリース 7.3.4

リリース日: 2003-07-24

7.3.3の様々な不具合を修正しました。

E.195.1. バージョン7.3.4への移行方法

7.3.X からの移行の場合は ダンプ/リストアは必要ありません。

E.195.2. 変更点

- Repair breakage in timestamp-to-date conversion for dates before 2000
- Prevent rare possibility of server startup failure (Tom)
- Fix bugs in interval-to-time conversion (Tom)
- Add constraint names in a few places in pg_dump (Rod)
- Improve performance of functions with many parameters (Tom)
- Fix to_ascii() buffer overruns (Tom)
- Prevent restore of database comments from throwing an error (Tom)
- Work around buggy strxfrm() present in some Solaris releases (Tom)
- Properly escape jdbc setObject() strings to improve security (Barry)

E.196. リリース 7.3.3

リリース日: 2003-05-22

バージョン7.3.2の様々な不具合を修正しました。

E.196.1. バージョン7.3.3への移行方法

7.3.X からの移行の場合は ダンプ/リストアは必要ありません。

E.196.2. 変更点

- Repair sometimes-incorrect computation of StartUpID after a crash
- Avoid slowness with lots of deferred triggers in one transaction (Stephan)
- Don't lock referenced row when `UPDATE` doesn't change foreign key's value (Jan)
- Use `-fPIC` not `-fpic` on Sparc (Tom Callaway)
- Repair lack of schema-awareness in contrib/reindexdb
- Fix contrib/intarray error for zero-element result array (Teodor)
- Ensure createuser script will exit on control-C (Oliver)
- Fix errors when the type of a dropped column has itself been dropped
- `CHECKPOINT` does not cause database panic on failure in noncritical steps
- Accept 60 in seconds fields of timestamp, time, interval input values
- Issue notice, not error, if `TIMESTAMP`, `TIME`, or `INTERVAL` precision too large
- Fix `abstime-to-time` cast function (fix is not applied unless you initdb)
- Fix `pg_proc` entry for `timestamp_tz_izone` (fix is not applied unless you initdb)
- Make `EXTRACT(EPOCH FROM timestamp without time zone)` treat input as local time
- `'now'::timestamp_tz` gave wrong answer if timezone changed earlier in transaction
- `HAVE_INT64_TIMESTAMP` code for time with timezone overwrote its input
- Accept `GLOBAL TEMP/TEMPORARY` as a synonym for `TEMPORARY`
- Avoid improper schema-privilege-check failure in foreign-key triggers
- Fix bugs in foreign-key triggers for `SET DEFAULT` action
- Fix incorrect time-qual check in row fetch for `UPDATE` and `DELETE` triggers
- Foreign-key clauses were parsed but ignored in `ALTER TABLE ADD COLUMN`
- Fix createlang script breakage for case where handler function already exists
- Fix misbehavior on zero-column tables in `pg_dump`, `COPY`, `ANALYZE`, other places
- Fix misbehavior of `func_error()` on type names containing `'%'`
- Fix misbehavior of `replace()` on strings containing `'%'`
- Regular-expression patterns containing certain multibyte characters failed
- Account correctly for `NULLs` in more cases in join size estimation

- Avoid conflict with system definition of `isblank()` function or macro
- Fix failure to convert large code point values in EUC_TW conversions (Tatsuo)
- Fix error recovery for `SSL_read/SSL_write` calls
- Don't do early constant-folding of type coercion expressions
- Validate page header fields immediately after reading in any page
- Repair incorrect check for ungrouped variables in unnamed joins
- Fix buffer overrun in `to_ascii` (Guido Notari)
- contrib/ltree fixes (Teodor)
- Fix core dump in deadlock detection on machines where char is unsigned
- Avoid running out of buffers in many-way indexscan (bug introduced in 7.3)
- Fix planner's selectivity estimation functions to handle domains properly
- Fix dbmirror memory-allocation bug (Steven Singer)
- Prevent infinite loop in `ln(numeric)` due to roundoff error
- `GROUP BY` got confused if there were multiple equal `GROUP BY` items
- Fix bad plan when inherited `UPDATE/DELETE` references another inherited table
- Prevent clustering on incomplete (partial or non-NULL-storing) indexes
- Service shutdown request at proper time if it arrives while still starting up
- Fix left-links in temporary indexes (could make backwards scans miss entries)
- Fix incorrect handling of `client_encoding` setting in `postgresql.conf` (Tatsuo)
- Fix failure to respond to `pg_ctl stop -m fast` after `Async_NotifyHandler` runs
- Fix SPI for case where rule contains multiple statements of the same type
- Fix problem with checking for wrong type of access privilege in rule query
- Fix problem with `EXCEPT` in `CREATE RULE`
- Prevent problem with dropping temp tables having serial columns
- Fix `replace_vars_with_subplan_refs` failure in complex views
- Fix regexp slowness in single-byte encodings (Tatsuo)
- Allow qualified type names in `CREATE CAST` and `DROP CAST`
- Accept `SETOF type[]`, which formerly had to be written `SETOF _type`
- Fix `pg_dump` core dump in some cases with procedural languages
- Force ISO datestyle in `pg_dump` output, for portability (Oliver)
- `pg_dump` failed to handle error return from `lo_read` (Oleg Drokin)
- `pg_dumpall` failed with groups having no members (Nick Eskelinen)
- `pg_dumpall` failed to recognize `--globals-only` switch
- `pg_restore` failed to restore blobs if `-X disable-triggers` is specified
- Repair intrafunction memory leak in `plpgsql`

pltcl's `core` command dumped core if given wrong parameters (Ian Harding)

- ppython used wrong value of `atttypmod` (Brad McLean)
- Fix improper quoting of boolean values in Python interface (D'Arcy)
- Added `addDataType()` method to `PGConnection` interface for JDBC
- Fixed various problems with updateable ResultSets for JDBC (Shawn Green)
- Fixed various problems with DatabaseMetaData for JDBC (Kris Jurka, Peter Royal)
- Fixed problem with parsing table ACLs in JDBC
- Better error message for character set conversion problems in JDBC

[前のページ](#)

リリース 7.3.4

[ホーム](#)
[上に戻る](#)

[次のページ](#)

リリース 7.3.2

E.197. リリース 7.3.2

リリース日: 2003-02-04

バージョン7.3.1の様々な不具合を修正しました。

E.197.1. バージョン7.3.2への移行方法

7.3.X からの移行の場合は ダンプ/リストアは必要ありません。

E.197.2. 変更点

- Restore creation of OID column in CREATE TABLE AS / SELECT INTO
- Fix pg_dump core dump when dumping views having comments
- Dump DEFERRABLE/INITIALLY DEFERRED constraints properly
- Fix UPDATE when child table's column numbering differs from parent
- Increase default value of max_fsm_relations
- Fix problem when fetching backwards in a cursor for a single-row query
- Make backward fetch work properly with cursor on SELECT DISTINCT query
- Fix problems with loading pg_dump files containing contrib/lo usage
- Fix problem with all-numeric user names
- Fix possible memory leak and core dump during disconnect in libpqtdl
- Make plpython's spi_execute command handle nulls properly (Andrew Bosma)
- Adjust plpython error reporting so that its regression test passes again
- Work with bison 1.875
- Handle mixed-case names properly in plpgsql's %type (Neil)
- Fix core dump in pltcl when executing a query rewritten by a rule
- Repair array subscript overruns (per report from Yichen Xie)
- Reduce MAX_TIME_PRECISION from 13 to 10 in floating-point case
- Correctly case-fold variable names in per-database and per-user settings
- Fix coredump in plpgsql's RETURN NEXT when SELECT into record returns no rows
- Fix outdated use of pg_type.typprtlen in python client interface
- Correctly handle fractional seconds in timestamps in JDBC driver
- Improve performance of getImportedKeys() in JDBC
- Make shared-library symlinks work standardly on HP/UX (Giles)
- Repair inconsistent rounding behavior for timestamp, time, interval
- SSL negotiation fixes (Nathan Mueller)
- Make libpq's ~/.pgpass feature work when connecting with PQconnectDB
- Update my2pg, ora2pg

- Translation updates
- Add casts between types lo and oid in contrib/lo
- fastpath code now checks for privilege to call function

[前のページ](#)
リリース 7.3.3

[ホーム](#)
[上に戻る](#)

[次のページ](#)
リリース 7.3.1

E.198. リリース 7.3.1

リリース日: 2002-12-18

バージョン7.3の様々な不具合を修正しました。

E.198.1. バージョン7.3.1への移行方法

7.3 からの移行の場合は ダンプ/リストアは必要ありません。しかし、PostgreSQLの主なインタフェースライブラリであるのメジャーバージョン番号がこのリリースから変更されましたので、libpqを使用したクライアントコードを再リンクする必要があります。

E.198.2. 変更点

- Fix a core dump of COPY TO when client/server encodings don't match (Tom)
- Allow pg_dump to work with pre-7.2 servers (Philip)
- contrib/adddepend fixes (Tom)
- Fix problem with deletion of per-user/per-database config settings (Tom)
- contrib/vacuumlo fix (Tom)
- Allow 'password' encryption even when pg_shadow contains MD5 passwords (Bruce)
- contrib/dbmirror fix (Steven Singer)
- Optimizer fixes (Tom)
- contrib/tsearch fixes (Teodor Sigaev, Magnus)
- Allow locale names to be mixed case (Nicolai Tufar)
- Increment libpq library's major version number (Bruce)
- pg_hba.conf error reporting fixes (Bruce, Neil)
- Add SCO Openserver 5.0.4 as a supported platform (Bruce)
- Prevent EXPLAIN from crashing server (Tom)
- SSL fixes (Nathan Mueller)
- Prevent composite column creation via ALTER TABLE (Tom)

E.199. リリース 7.3

リリース日: 2002-11-27

E.199.1. 概要

このリリースにおける大きな変更は以下の通りです。

スキーマ

スキーマを使用すると、ユーザはオブジェクトを独立した名前空間で作成することができます。これにより、2 人の人や、2 つのアプリケーションが同じ名前のテーブルを作成できます。共有テーブル用のパブリックスキーマもあります。テーブル / インデックスの作成は、パブリックスキーマへのアクセス許可を禁止することで制御できます。

列の削除

PostgreSQLは `ALTER TABLE ... DROP COLUMN` 機能をサポートするようになりました。

テーブル関数

複数の行および / または複数の列を返す関数が、以前よりもずっと使いやすくなりました。そのような "テーブル関数" を、`SELECT FROM` 句で呼び出し、その出力をテーブルのように扱うことができます。また、PL/pgSQL の関数はセットを返すことができるようになりました。

ブリペアドクエリ

パフォーマンス向上のため、PostgreSQLはブリペアドクエリをサポートするようになりました。

依存関係追跡

PostgreSQLはオブジェクトの依存関係を記録するようになりました。これにより様々な点が向上されます。`DROP` 文は、`CASCADE` または `RESTRICT` を使用して、依存関係のあるオブジェクトも削除されるかどうかを制御するようになりました。

権限

関数や手続き言語は権限をサポートするようになり、関数は作成者の権限で実行されるように定義可能になりました。

国際化

マルチバイトとロケールのサポートは、どちらも常に有効となりました。

ロギング

様々なロギングオプションが強化されました。

インタフェース

多数のインタフェースが <http://gborg.postgresql.org> に移動され、それぞれ独立して開発およびリリースされることが可能になりました。

関数 / 識別子

デフォルトで、関数は 32 個までのパラメータを取ることができ、識別子は 63 バイトの長さにできるようになりました。また、`OPAQUE` の使用は非難されるようになったため、関数の引数および結果の型において、以前 `OPAQUE` がそれぞれ意味していたことを表すために、特定の "擬似データ型" がサポートされるようになりました。

E.199.2. バージョン 7.3 への移行

以前のリリースからデータを移行する場合、`pg_dump` によるダンプ/リストアが必要です。アプリケーションがシステムカタログを参照する場合、7.3 におけるスキーマの導入のために、追加的な変更が必要になります。詳しくは、http://developer.postgresql.org/~momjian/upgrade_tips_7.3 を参照してください。

以下の非互換性について検討して下さい。 >

- 6.3 よりも前のバージョンのクライアントはサポートされません。
- `pg_hba.conf` には、ユーザ名のための列および新たな機能が追加されました。既存のファイルは調整する必要があります。
- `postgresql.conf` のいくつかのロギングパラメータの名前が変更されました。
- `LIMIT #, #` は無効になりましたので `LIMIT # OFFSET #` を使用してください。
- 列リストを使用する `INSERT` 文では、それぞれの列に値を指定する必要があります。例えば、`INSERT INTO tab (col1, col2) VALUES ('val1')` は無効となります。 `INSERT` が列リストを持たない場合に、期待されるよりも少ない列を指定することは依然として許可されます。
- `serial` 列は、自動的に `UNIQUE` とはならなくなりました。したがって、インデックスは自動的に作成されません。
- 中断されたトランザクション内の `SET` コマンドはロールバックされるようになりました。
- `COPY` は欠落した後続の列をヌルと見なさなくなりました。全ての列を指定する必要があります。(ただし、`COPY` コマンドに列リストを指定することにより、同様の結果を実現することができます。)
- データ型 `timestamp` は、タイムゾーン付きタイムスタンプではなく、タイムゾーンなしタイムスタンプと同等になりました。
- PostgreSQL 7.3 にロードされた、7.3 よりも前のデータベースは、`serial` 列、一意的制約、および外部キーに対する新しいオブジェクト依存関係を持ちません。詳細な説明と、そのような依存関係を追加するスクリプトについては、`contrib/adddepend/` ディレクトリを参照して下さい。
- 空の文字列 ('') は、整数フィールドに対する入力として許可されなくなりました。以前は、これは暗黙的に 0 と解釈されていました。

E.199.3. 変更点

E.199.3.1. サーバの操作

- Add `pg_locks` view to show locks (Neil)
- Security fixes for password negotiation memory allocation (Neil)
- Remove support for version 0 FE/BE protocol (PostgreSQL 6.2 and earlier) (Tom)
- Reserve the last few backend slots for superusers, add parameter `superuser_reserved_connections` to control this (Nigel J. Andrews)

E.199.3.2. パフォーマンス

- Improve startup by calling `localtime()` only once (Tom)
- Cache system catalog information in flat files for faster startup (Tom)
- Improve caching of index information (Tom)
- Optimizer improvements (Tom, Fernando Nasser)
- Catalog caches now store failed lookups (Tom)
- Hash function improvements (Neil)
- Improve performance of query tokenization and network handling (Peter)
- Speed improvement for large object restore (Mario Weilguni)
- Mark expired index entries on first lookup, saving later heap fetches (Tom)
- Avoid excessive NULL bitmap padding (Manfred Koizar)
- Add BSD-licensed `qsort()` for Solaris, for performance (Bruce)
- Reduce per-row overhead by four bytes (Manfred Koizar)

- Fix GEQO optimizer bug (Neil Conway)
- Make WITHOUT OID actually save four bytes per row (Manfred Koizar)
- Add default_statistics_target variable to specify ANALYZE buckets (Neil)
- Use local buffer cache for temporary tables so no WAL overhead (Tom)
- Improve free space map performance on large tables (Stephen Marshall, Tom)
- Improved WAL write concurrency (Tom)

E.199.3.3. 権限

- Add privileges on functions and procedural languages (Peter)
- Add OWNER to CREATE DATABASE so superusers can create databases on behalf of unprivileged users (Gavin Sherry, Tom)
- Add new object privilege bits EXECUTE and USAGE (Tom)
- Add SET SESSION AUTHORIZATION DEFAULT and RESET SESSION AUTHORIZATION (Tom)
- Allow functions to be executed with the privilege of the function owner (Peter)

E.199.3.4. サーバの設定

- Server log messages now tagged with LOG, not DEBUG (Bruce)
- Add user column to pg_hba.conf (Bruce)
- Have log_connections output two lines in log file (Tom)
- Remove debug_level from postgresql.conf, now server_min_messages (Bruce)
- New ALTER DATABASE/USER ... SET command for per-user/database initialization (Peter)
- New parameters server_min_messages and client_min_messages to control which messages are sent to the server logs or client applications (Bruce)
- Allow pg_hba.conf to specify lists of users/databases separated by commas, group names prepended with +, and file names prepended with @ (Bruce)
- Remove secondary password file capability and pg_password utility (Bruce)
- Add variable db_user_namespace for database-local user names (Bruce)
- SSL improvements (Bear Giles)
- Make encryption of stored passwords the default (Bruce)
- Allow pg_statistics to be reset by calling pg_stat_reset() (Christopher)
- Add log_duration parameter (Bruce)
- Rename debug_print_query to log_statement (Bruce)
- Rename show_query_stats to show_statement_stats (Bruce)
- Add param log_min_error_statement to print commands to logs on error (Gavin)

E.199.3.5. 問い合わせ

- Make cursors insensitive, meaning their contents do not change (Tom)
- Disable LIMIT #,# syntax; now only LIMIT # OFFSET # supported (Bruce)
- Increase identifier length to 63 (Neil, Bruce)

- UNION fixes for merging ≥ 3 columns of different lengths (Tom)
- Add DEFAULT key word to INSERT, e.g., INSERT ... (... , DEFAULT, ...) (Rod)
- Allow views to have default values using ALTER COLUMN ... SET DEFAULT (Neil)
- Fail on INSERTs with column lists that don't supply all column values, e.g., INSERT INTO tab (col1, col2) VALUES ('val1'); (Rod)
- Fix for join aliases (Tom)
- Fix for FULL OUTER JOINS (Tom)
- Improve reporting of invalid identifier and location (Tom, Gavin)
- Fix OPEN cursor(args) (Tom)
- Allow 'ctid' to be used in a view and currtid(viewname) (Hiroshi)
- Fix for CREATE TABLE AS with UNION (Tom)
- SQL99 syntax improvements (Thomas)
- Add statement_timeout variable to cancel queries (Bruce)
- Allow prepared queries with PREPARE/EXECUTE (Neil)
- Allow FOR UPDATE to appear after LIMIT/OFFSET (Bruce)
- Add variable autocommit (Tom, David Van Wie)

E.199.3.6. オブジェクトの操作

- Make equals signs optional in CREATE DATABASE (Gavin Sherry)
- Make ALTER TABLE OWNER change index ownership too (Neil)
- New ALTER TABLE tablename ALTER COLUMN colname SET STORAGE controls TOAST storage, compression (John Gray)
- Add schema support, CREATE/DROP SCHEMA (Tom)
- Create schema for temporary tables (Tom)
- Add variable search_path for schema search (Tom)
- Add ALTER TABLE SET/DROP NOT NULL (Christopher)
- New CREATE FUNCTION volatility levels (Tom)
- Make rule names unique only per table (Tom)
- Add 'ON tablename' clause to DROP RULE and COMMENT ON RULE (Tom)
- Add ALTER TRIGGER RENAME (Joe)
- New current_schema() and current_schemas() inquiry functions (Tom)
- Allow functions to return multiple rows (table functions) (Joe)
- Make WITH optional in CREATE DATABASE, for consistency (Bruce)
- Add object dependency tracking (Rod, Tom)
- Add RESTRICT/CASCADE to DROP commands (Rod)
- Add ALTER TABLE DROP for non-CHECK CONSTRAINT (Rod)
- Autodestroy sequence on DROP of table with SERIAL (Rod)

Prevent column dropping if column is used by foreign key (Rod)

- Automatically drop constraints/functions when object is dropped (Rod)
- Add CREATE/DROP OPERATOR CLASS (Bill Studenmund, Tom)
- Add ALTER TABLE DROP COLUMN (Christopher, Tom, Hiroshi)
- Prevent inherited columns from being removed or renamed (Alvaro Herrera)
- Fix foreign key constraints to not error on intermediate database states (Stephan)
- Propagate column or table renaming to foreign key constraints
- Add CREATE OR REPLACE VIEW (Gavin, Neil, Tom)
- Add CREATE OR REPLACE RULE (Gavin, Neil, Tom)
- Have rules execute alphabetically, returning more predictable values (Tom)
- Triggers are now fired in alphabetical order (Tom)
- Add /contrib/adddepend to handle pre-7.3 object dependencies (Rod)
- Allow better casting when inserting/updating values (Tom)

E.199.3.7. ユーティリティコマンド

- Have COPY TO output embedded carriage returns and newlines as ¥r and ¥n (Tom)
- Allow DELIMITER in COPY FROM to be 8-bit clean (Tatsuo)
- Make pg_dump use ALTER TABLE ADD PRIMARY KEY, for performance (Neil)
- Disable brackets in multistatement rules (Bruce)
- Disable VACUUM from being called inside a function (Bruce)
- Allow dropdb and other scripts to use identifiers with spaces (Bruce)
- Restrict database comment changes to the current database
- Allow comments on operators, independent of the underlying function (Rod)
- Rollback SET commands in aborted transactions (Tom)
- EXPLAIN now outputs as a query (Tom)
- Display condition expressions and sort keys in EXPLAIN (Tom)
- Add 'SET LOCAL var = value' to set configuration variables for a single transaction (Tom)
- Allow ANALYZE to run in a transaction (Bruce)
- Improve COPY syntax using new WITH clauses, keep backward compatibility (Bruce)
- Fix pg_dump to consistently output tags in non-ASCII dumps (Bruce)
- Make foreign key constraints clearer in dump file (Rod)
- Add COMMENT ON CONSTRAINT (Rod)
- Allow COPY TO/FROM to specify column names (Brent Verner)
- Dump UNIQUE and PRIMARY KEY constraints as ALTER TABLE (Rod)
- Have SHOW output a query result (Joe)
- Generate failure on short COPY lines rather than pad NULLs (Neil)

Fix CLUSTER to preserve all table attributes (Alvaro Herrera)

- New pg_settings table to view/modify GUC settings (Joe)
- Add smart quoting, portability improvements to pg_dump output (Peter)
- Dump serial columns out as SERIAL (Tom)
- Enable large file support, >2G for pg_dump (Peter, Philip Warner, Bruce)
- Disallow TRUNCATE on tables that are involved in referential constraints (Rod)
- Have TRUNCATE also auto-truncate the toast table of the relation (Tom)
- Add clusterdb utility that will auto-cluster an entire database based on previous CLUSTER operations (Alvaro Herrera)
- Overhaul pg_dumpall (Peter)
- Allow REINDEX of TOAST tables (Tom)
- Implemented START TRANSACTION, per SQL99 (Neil)
- Fix rare index corruption when a page split affects bulk delete (Tom)
- Fix ALTER TABLE ... ADD COLUMN for inheritance (Alvaro Herrera)

E.199.3.8. データ型と関数

- Fix factorial(0) to return 1 (Bruce)
- Date/time/timezone improvements (Thomas)
- Fix for array slice extraction (Tom)
- Fix extract/date_part to report proper microseconds for timestamp (Tatsuo)
- Allow text_substr() and bytea_substr() to read TOAST values more efficiently (John Gray)
- Add domain support (Rod)
- Make WITHOUT TIME ZONE the default for TIMESTAMP and TIME data types (Thomas)
- Allow alternate storage scheme of 64-bit integers for date/time types using --enable-integer-datetimes in configure (Thomas)
- Make timezone(timestamptztz) return timestamp rather than a string (Thomas)
- Allow fractional seconds in date/time types for dates prior to 1BC (Thomas)
- Limit timestamp data types to 6 decimal places of precision (Thomas)
- Change timezone conversion functions from timetz() to timezone() (Thomas)
- Add configuration variables datestyle and timezone (Tom)
- Add OVERLAY(), which allows substitution of a substring in a string (Thomas)
- Add SIMILAR TO (Thomas, Tom)
- Add regular expression SUBSTRING(string FROM pat FOR escape) (Thomas)
- Add LOCALTIME and LOCALTIMESTAMP functions (Thomas)
- Add named composite types using CREATE TYPE typename AS (column) (Joe)
- Allow composite type definition in the table alias clause (Joe)
- Add new API to simplify creation of C language table functions (Joe)

Remove ODBC-compatible empty parentheses from calls to SQL99 functions for which these parentheses do not match the standard (Thomas)

- Allow macaddr data type to accept 12 hex digits with no separators (Mike Wyer)
- Add CREATE/DROP CAST (Peter)
- Add IS DISTINCT FROM operator (Thomas)
- Add SQL99 TREAT() function, synonym for CAST() (Thomas)
- Add pg_backend_pid() to output backend pid (Bruce)
- Add IS OF / IS NOT OF type predicate (Thomas)
- Allow bit string constants without fully-specified length (Thomas)
- Allow conversion between 8-byte integers and bit strings (Thomas)
- Implement hex literal conversion to bit string literal (Thomas)
- Allow table functions to appear in the FROM clause (Joe)
- Increase maximum number of function parameters to 32 (Bruce)
- No longer automatically create index for SERIAL column (Tom)
- Add current_database() (Rod)
- Fix cash_words() to not overflow buffer (Tom)
- Add functions replace(), split_part(), to_hex() (Joe)
- Fix LIKE for bytea as a right-hand argument (Joe)
- Prevent crashes caused by SELECT cash_out(2) (Tom)
- Fix to_char(1,'FM999.99') to return a period (Karel)
- Fix trigger/type/language functions returning OPAQUE to return proper type (Tom)

E.199.3.9. 国際化

- Add additional encodings: Korean (JOHAB), Thai (WIN874), Vietnamese (TCVN), Arabic (WIN1256), Simplified Chinese (GBK), Korean (UHC) (Eiji Tokuya)
- Enable locale support by default (Peter)
- Add locale variables (Peter)
- Escape bytes $\geq 0x7f$ for multibyte in PQescapeBytea/PQunescapeBytea (Tatsuo)
- Add locale awareness to regular expression character classes
- Enable multibyte support by default (Tatsuo)
- Add GB18030 multibyte support (Bill Huang)
- Add CREATE/DROP CONVERSION, allowing loadable encodings (Tatsuo, Kaori)
- Add pg_conversion table (Tatsuo)
- Add SQL99 CONVERT() function (Tatsuo)
- pg_dumpall, pg_controldata, and pg_resetxlog now national-language aware (Peter)
- New and updated translations

E.199.3.10. サーバサイド言語

- Allow recursive SQL function (Peter)
- Change PL/Tcl build to use configured compiler and Makefile.shlib (Peter)
- Overhaul the PL/pgSQL FOUND variable to be more Oracle-compatible (Neil, Tom)
- Allow PL/pgSQL to handle quoted identifiers (Tom)
- Allow set-returning PL/pgSQL functions (Neil)
- Make PL/pgSQL schema-aware (Joe)
- Remove some memory leaks (Nigel J. Andrews, Tom)

E.199.3.11. psql

- Don't lowercase psql %connect database name for 7.2.0 compatibility (Tom)
- Add psql %timing to time user queries (Greg Sabino Mullane)
- Have psql %d show index information (Greg Sabino Mullane)
- New psql %dD shows domains (Jonathan Eisler)
- Allow psql to show rules on views (Paul ?)
- Fix for psql variable substitution (Tom)
- Allow psql %d to show temporary table structure (Tom)
- Allow psql %d to show foreign keys (Rod)
- Fix %? to honor %pset pager (Bruce)
- Have psql reports its version number on startup (Tom)
- Allow %copy to specify column names (Tom)

E.199.3.12. libpq

- Add ~/.pgpass to store host/user password combinations (Alvaro Herrera)
- Add PQunescapeBytea() function to libpq (Patrick Welche)
- Fix for sending large queries over non-blocking connections (Bernhard Herzog)
- Fix for libpq using timers on Win9X (David Ford)
- Allow libpq notify to handle servers with different-length identifiers (Tom)
- Add libpq PQescapeString() and PQescapeBytea() to Windows (Bruce)
- Fix for SSL with non-blocking connections (Jack Bates)
- Add libpq connection timeout parameter (Denis A Ustimenko)

E.199.3.13. JDBC

- Allow JDBC to compile with JDK 1.4 (Dave)
- Add JDBC 3 support (Barry)
- Allows JDBC to set loglevel by adding ?loglevel=X to the connection URL (Barry)
- Add Driver.info() message that prints out the version number (Barry)
- Add updateable result sets (Raghu Nidagal, Dave)
- Add support for callable statements (Paul Bethe)

- Add query cancel capability
- Add refresh row (Dave)
- Fix MD5 encryption handling for multibyte servers (Jun Kawai)
- Add support for prepared statements (Barry)

E.199.3.14. 種々のインターフェイス

- Fixed ECPG bug concerning octal numbers in single quotes (Michael)
- Move src/interfaces/libpq to <http://gborg.postgresql.org> (Marc, Bruce)
- Improve Python interface (Elliot Lee, Andrew Johnson, Greg Copeland)
- Add libpqctl connection close event (Gerhard Hintermayer)
- Move src/interfaces/libpq++ to <http://gborg.postgresql.org> (Marc, Bruce)
- Move src/interfaces/odbc to <http://gborg.postgresql.org> (Marc)
- Move src/interfaces/libpqe to <http://gborg.postgresql.org> (Marc, Bruce)
- Move src/interfaces/perl5 to <http://gborg.postgresql.org> (Marc, Bruce)
- Remove src/bin/pgaccess from main tree, now at <http://www.pgaccess.org> (Bruce)
- Add pg_on_connection_loss command to libpqctl (Gerhard Hintermayer, Tom)

E.199.3.15. ソースコード

- Fix for parallel make (Peter)
- AIX fixes for linking Tcl (Andreas Zeugswetter)
- Allow PL/Perl to build under Cygwin (Jason Tishler)
- Improve MIPS compiles (Peter, Oliver Elphick)
- Require Autoconf version 2.53 (Peter)
- Require readline and zlib by default in configure (Peter)
- Allow Solaris to use Intimate Shared Memory (ISM), for performance (Scott Brunza, P.J. Josh Rovero)
- Always enable syslog in compile, remove --enable-syslog option (Tatsuo)
- Always enable multibyte in compile, remove --enable-multibyte option (Tatsuo)
- Always enable locale in compile, remove --enable-locale option (Peter)
- Fix for Win9x DLL creation (Magnus Naeslund)
- Fix for link() usage by WAL code on Windows, BeOS (Jason Tishler)
- Add sys/types.h to c.h, remove from main files (Peter, Bruce)
- Fix AIX hang on SMP machines (Tomoyuki Nijima)
- AIX SMP hang fix (Tomoyuki Nijima)
- Fix pre-1970 date handling on newer glibc libraries (Tom)
- Fix PowerPC SMP locking (Tom)
- Prevent gcc -ffast-math from being used (Peter, Tom)
- Bison >= 1.50 now required for developer builds

- Kerberos 5 support now builds with Heimdal (Peter)
- Add appendix in the User's Guide which lists SQL features (Thomas)
- Improve loadable module linking to use RTLD_NOW (Tom)
- New error levels WARNING, INFO, LOG, DEBUG[1-5] (Bruce)
- New src/port directory holds replaced libc functions (Peter, Bruce)
- New pg_namespace system catalog for schemas (Tom)
- Add pg_class.relnamespace for schemas (Tom)
- Add pg_type.typnamespace for schemas (Tom)
- Add pg_proc.pronamespace for schemas (Tom)
- Restructure aggregates to have pg_proc entries (Tom)
- System relations now have their own namespace, pg_* test not required (Fernando Nasser)
- Rename TOAST index names to be *_index rather than *_idx (Neil)
- Add namespaces for operators, opclasses (Tom)
- Add additional checks to server control file (Thomas)
- New Polish FAQ (Marcin Mazurek)
- Add Posix semaphore support (Tom)
- Document need for reindex (Bruce)
- Rename some internal identifiers to simplify Windows compile (Jan, Katherine Ward)
- Add documentation on computing disk space (Bruce)
- Remove KSQO from GUC (Bruce)
- Fix memory leak in rtree (Kenneth Been)
- Modify a few error messages for consistency (Bruce)
- Remove unused system table columns (Peter)
- Make system columns NOT NULL where appropriate (Tom)
- Clean up use of sprintf in favor of snprintf() (Neil, Jukka Holappa)
- Remove OPAQUE and create specific subtypes (Tom)
- Cleanups in array internal handling (Joe, Tom)
- Disallow pg_atoi("") (Bruce)
- Remove parameter wal_files because WAL files are now recycled (Bruce)
- Add version numbers to heap pages (Tom)

E.199.3.16. 寄贈

- Allow inet arrays in /contrib/array (Neil)
- GiST fixes (Teodor Sigaev, Neil)
- Upgrade /contrib/mysql
- Add /contrib/dbsize which shows table sizes without vacuum (Peter)

Add /contrib/intagg, integer aggregator routines (mlw)

- Improve /contrib/oid2name (Neil, Bruce)
- Improve /contrib/tsearch (Oleg, Teodor Sigaev)
- Cleanups of /contrib/rserver (Alexey V. Borzov)
- Update /contrib/oracle conversion utility (Gilles Darold)
- Update /contrib/dblink (Joe)
- Improve options supported by /contrib/vacuumlo (Mario Weilguni)
- Improvements to /contrib/intarray (Oleg, Teodor Sigaev, Andrey Oktyabrski)
- Add /contrib/reindexdb utility (Shaun Thomas)
- Add indexing to /contrib/isbn_issn (Dan Weston)
- Add /contrib/dbmirror (Steven Singer)
- Improve /contrib/pgbench (Neil)
- Add /contrib/tablefunc table function examples (Joe)
- Add /contrib/ltree data type for tree structures (Teodor Sigaev, Oleg Bartunov)
- Move /contrib/pg_controldata, pg_resetxlog into main tree (Bruce)
- Fixes to /contrib/cube (Bruno Wolff)
- Improve /contrib/fulltextindex (Christopher)

[前のページ](#)

リリース 7.3.1

[ホーム](#)
[上に戻る](#)

[次のページ](#)

リリース7.2.8

E.200. リリース7.2.8

リリース日: 2005-05-05

このリリースは、セキュリティ関連の問題を含む、7.2.7の各種不具合を修正したものです。

E.200.1. バージョン7.2.8への移行

7.2.Xからの移行ではダンプ/リストアは不要です。

E.200.2. 変更点

- `TIME WITH TIME ZONE` 型の値に対する `EXTRACT(EPOCH)` を修正しました。
- `plpgsql` において更にバッファオーバーランの検査を追加しました。(Neil)
- `%` を名前含むトリガを正しくダンプできるように `pg_dump` を修正しました。
- 月に関連した書式のために `to_char(interval)` がコアダンプすることを防止しました。
- より最近の OpenSSL による構築のために `contrib/pgcrypto` を修正しました。(Marko Kreen)

E.201. リリース7.2.7

リリース日: 2005-01-31

このリリースは複数のセキュリティ問題を含む、7.2.6の各種不具合を修正したものです。

E.201.1. バージョン7.2.7への移行

7.2.Xからの移行の場合は ダンプ/リストアは必要ありません。

E.201.2. 変更点

- Disallow `LOAD` to non-superusers

On platforms that will automatically execute initialization functions of a shared library (this includes at least Windows and ELF-based Unixen), `LOAD` can be used to make the server execute arbitrary code. Thanks to NGS Software for reporting this.

- Add needed STRICT marking to some contrib functions (Kris Jurka)
- Avoid buffer overrun when plpgsql cursor declaration has too many parameters (Neil)
- Fix planning error for FULL and RIGHT outer joins

The result of the join was mistakenly supposed to be sorted the same as the left input. This could not only deliver mis-sorted output to the user, but in case of nested merge joins could give outright wrong answers.

- Fix display of negative intervals in SQL and GERMAN datestyles

E.202. リリース7.2.6

リリース日: 2004-10-22

このリリースは7.2.5の各種不具合を修正したものです。

E.202.1. バージョン7.2.6への移行

7.2.Xからの移行の場合は ダンプ/リストアは必要ありません。

E.202.2. 変更点

- ディスク上のヒントビットの更新に失敗する可能性を修正

稀な環境下で、この誤りにより"could not access transaction status"という失敗になる可能性があります。 これはデータ損失の可能性がある不具合です。

- ハッシュ外部結合がタプルを喪失しないことを確実にしました。

ハッシュ結合計画を使用する非常に大規模な左結合が、右側のデータ分布のみが与えられた一致しない左側の行を出力する可能性があります。

- rootによるpg_ctlの実行を禁止

潜在するセキュリティ問題から保護します。

- make_oidjoins_checkにおける/tmp内の中間ファイルの使用を避けました。

これはセキュリティ問題として報告されたものです。 しかし、開発者以外がこのスクリプトを使用する理由がありませんので、ほとんど気にする必要はありません。

- より新しいバージョンのBisonに変更

E.203. リリース7.2.5

リリース日: 2004-08-16

このリリースは7.2.4のさまざまな不具合を改修したものです。

E.203.1. バージョン7.2.5への移行

7.2.Xからの移行の場合、ダンプ/リストアは必要ありません。

E.203.2. 変更点

- クラッシュ時のコミット済みのトランザクションの欠落の可能性を修正

トランザクションのコミットとチェックポイント処理との間の内部ロック処理が不十分であったため、最近のチェックポイント直前にコミットされたトランザクションの全部あるいは一部がデータベースのクラッシュの後の再起動時に欠落する可能性があります。この深刻な不具合はPostgreSQL 7.1以降から存在していました。

- Fix corner case for btree search in parallel with first root page split
- Fix buffer overrun in `to_ascii` (Guido Notari)
- Fix core dump in deadlock detection on machines where char is unsigned
- Fix failure to respond to `pg_ctl stop -m fast` after `Async_NotifyHandler` runs
- Repair memory leaks in `pg_dump`
- Avoid conflict with system definition of `isblank()` function or macro

E.204. リリース7.2.4

リリース日: 2003-01-30

このリリースはバージョン 7.2.3 の不具合を改修したものです。 データ損失の可能性を防止するための改修も含まれています。

E.204.1. バージョン 7.2.4 への移行方法

7.2.X からの移行の場合は ダンプ/リストアは必要ありません。

E.204.2. 変更点

- Fix some additional cases of VACUUM "No one parent tuple was found" error
- Prevent VACUUM from being called inside a function (Bruce)
- Ensure pg_clog updates are sync'd to disk before marking checkpoint complete
- Avoid integer overflow during large hash joins
- Make GROUP commands work when pg_group.grolist is large enough to be toasted
- Fix errors in datetime tables; some timezone names weren't being recognized
- Fix integer overflows in circle_poly(), path_encode(), path_add() (Neil)
- Repair long-standing logic errors in lseg_eq(), lseg_ne(), lseg_center()

E.205. リリース 7.2.3

リリース日: 2002-10-01

このリリースはバージョン 7.2.2 の不具合を改修したものです。 データ損失の可能性を防止するための改修も含まれています。

E.205.1. バージョン 7.2.3 への移行方法

7.2.X からの移行の場合は ダンプ/リストアは必要ありません。

E.205.2. 変更点

- Prevent possible compressed transaction log loss (Tom)
- Prevent non-superuser from increasing most recent vacuum info (Tom)
- Handle pre-1970 date values in newer versions of glibc (Tom)
- Fix possible hang during server shutdown
- Prevent spinlock hangs on SMP PPC machines (Tomoyuki Nijima)
- Fix pg_dump to properly dump FULL JOIN USING (Tom)

E.206. リリース 7.2.2

リリース日: 2002-08-23

このリリースはバージョン 7.2.1 の不具合を改修したものです。

E.206.1. バージョン 7.2.2 への移行方法

7.2.X からの移行の場合は ダンプ/リストアは必要ありません。

E.206.2. 変更点

- Allow EXECUTE of "CREATE TABLE AS ... SELECT" in PL/pgSQL (Tom)
- Fix for compressed transaction log id wraparound (Tom)
- Fix PQescapeBytea/PQunescapeBytea so that they handle bytes > 0x7f (Tatsuo)
- Fix for psql and pg_dump crashing when invoked with non-existent long options (Tatsuo)
- Fix crash when invoking geometric operators (Tom)
- Allow OPEN cursor(args) (Tom)
- Fix for rtree_gist index build (Teodor)
- Fix for dumping user-defined aggregates (Tom)
- contrib/intarray fixes (Oleg)
- Fix for complex UNION/EXCEPT/INTERSECT queries using parens (Tom)
- Fix to pg_convert (Tatsuo)
- Fix for crash with long DATA strings (Thomas, Neil)
- Fix for repeat(), lpad(), rpad() and long strings (Neil)

E.207. リリース 7.2.1

リリース日: 2002-03-21

このリリースはバージョン 7.2 の不具合を改修したものです。

E.207.1. バージョン 7.2.1 への移行方法

7.2 からの移行の場合は ダンプ/リストアは必要ありません。

E.207.2. 変更点

- Ensure that sequence counters do not go backwards after a crash (Tom)
- Fix pgaccess kanji-conversion key binding (Tatsuo)
- Optimizer improvements (Tom)
- Cash I/O improvements (Tom)
- New Russian FAQ
- Compile fix for missing AuthBlockSig (Heiko)
- Additional time zones and time zone fixes (Thomas)
- Allow psql ¥connect to handle mixed case database and user names (Tom)
- Return proper OID on command completion even with ON INSERT rules (Tom)
- Allow COPY FROM to use 8-bit DELIMITERS (Tatsuo)
- Fix bug in extract/date_part for milliseconds/microseconds (Tatsuo)
- Improve handling of multiple UNIONs with different lengths (Tom)
- contrib/btree_gist improvements (Teodor Sigaev)
- contrib/tsearch dictionary improvements, see README.tsearch for an additional installation step (Thomas T. Thai, Teodor Sigaev)
- Fix for array subscripts handling (Tom)
- Allow EXECUTE of "CREATE TABLE AS ... SELECT" in PL/pgSQL (Tom)

E.208. リリース 7.2

リリース日: 2002-02-04

E.208.1. 概要

このリリースは、大容量アプリケーションでの使用のための PostgreSQL の改良です。

このリリースにおける大きな変更は以下の通りです。

VACUUM

`vacuum` がテーブルをロックしなくなりました。これにより、通常のユーザは `vacuum` 処理の間にアクセスすることができます。`VACUUM FULL` という新しいコマンドは、テーブルをロックする古い `vacuum` 処理を行い、ディスク上のテーブルのコピーを縮小させます。

トランザクション

トランザクションが 40 億を越えたインストレーションの問題はなくなりました。

OIDs

OID は省略できるようになりました。ユーザは OID の使用が多過ぎる場合に OID 無しのテーブルを作成することができます。

オプティマイザ

システムは、`ANALYZE` 時に列統計情報の度数分布を計算するようになりました。オプティマイザはより良い選択ができるようになります。

セキュリティ

新しい MD5 暗号化オプションによって、より安全な格納やパスワード通信ができるようになりました。新しい Unix ドメインソケットの認証オプションが Linux や BSD システムで使用できます。

統計情報

管理者は、新しいテーブルアクセス統計モジュールを使用して、テーブルやインデックスの使用に関する、粒度の良い情報を入手することができます。

国際化

プログラムとライブラリのメッセージが、複数の言語で表示されるようになりました。

E.208.2. バージョン 7.2 への移行

以前のリリースからデータを移行する場合、`pg_dump` によるダンプ/リストアが必要です。

以下の非互換性について検討して下さい。 >

- `VACUUM` コマンドのセマンティックスがこのリリースで変更されました。これに伴い、保守手順を更新する方が良いでしょう。
- このリリースでは、`= NULL` による比較は常に偽(より正確にいうと `NULL`)を返します。以前のリリースでは、自動的にこの構文を `IS NULL` に変換していました。`postgresql.conf` のパラメータを使用して、この以前の動作を有効にすることができます。
- `pg_hba.conf` と `pg_ident.conf` 設定ファイルは接続時に毎回ではなく、`SIGHUP` シグナルを受け取った時にのみ再読み込みされるようになりました。
- `octet_length()` 関数は圧縮しない状態のデータ長を返すようになりました。
- 'current' という日付/時刻の値は使用できなくなりました。アプリケーションを書き換える必要があります。

- `timestamp()`、`time()`、および `interval()` 関数が使用できなくなりました。`timestamp()` の代わりに `timestamp 'string'` を使用するか、`CAST` を使用してください。

`SELECT ... LIMIT #, #` 構文は次のリリースから削除されます。問い合わせを例えば `LIMIT 10 OFFSET 20` のように `LIMIT` と `OFFSET` 句を使用するように問い合わせを変更して下さい。

E.208.3. 変更点

E.208.3.1. サーバの操作

- Create temporary files in a separate directory (Bruce)
- Delete orphaned temporary files on postmaster startup (Bruce)
- Added unique indexes to some system tables (Tom)
- System table operator reorganization (Oleg Bartunov, Teodor Sigaev, Tom)
- Renamed `pg_log` to `pg_clog` (Tom)
- Enable `SIGTERM`, `SIGQUIT` to kill backends (Jan)
- Removed compile-time limit on number of backends (Tom)
- Better cleanup for semaphore resource failure (Tatsuo, Tom)
- Allow safe transaction ID wraparound (Tom)
- Removed OIDs from some system tables (Tom)
- Removed "triggered data change violation" error check (Tom)
- SPI portal creation of prepared/saved plans (Jan)
- Allow SPI column functions to work for system columns (Tom)
- Long value compression improvement (Tom)
- Statistics collector for table, index access (Jan)
- Truncate extra-long sequence names to a reasonable value (Tom)
- Measure transaction times in milliseconds (Thomas)
- Fix TID sequential scans (Hiroshi)
- Superuser ID now fixed at 1 (Peter E)
- New `pg_ctl "reload"` option (Tom)

E.208.3.2. パフォーマンス

- Optimizer improvements (Tom)
- New histogram column statistics for optimizer (Tom)
- Reuse write-ahead log files rather than discarding them (Tom)
- Cache improvements (Tom)
- IS NULL, IS NOT NULL optimizer improvement (Tom)
- Improve lock manager to reduce lock contention (Tom)
- Keep `relcache` entries for index access support functions (Tom)
- Allow better selectivity with NaN and infinities in NUMERIC (Tom)
- R-tree performance improvements (Kenneth Been)

- B-tree splits more efficient (Tom)

E.208.3.3. 権限

- Change UPDATE, DELETE privileges to be distinct (Peter E)
- New REFERENCES, TRIGGER privileges (Peter E)
- Allow GRANT/REVOKE to/from more than one user at a time (Peter E)
- New has_table_privilege() function (Joe Conway)
- Allow non-superuser to vacuum database (Tom)
- New SET SESSION AUTHORIZATION command (Peter E)
- Fix bug in privilege modifications on newly created tables (Tom)
- Disallow access to pg_statistic for non-superuser, add user-accessible views (Tom)

E.208.3.4. クライアント認証

- Fork postmaster before doing authentication to prevent hangs (Peter E)
- Add ident authentication over Unix domain sockets on Linux, *BSD (Helge Bahmann, Oliver Elphick, Teodor Sigaev, Bruce)
- Add a password authentication method that uses MD5 encryption (Bruce)
- Allow encryption of stored passwords using MD5 (Bruce)
- PAM authentication (Dominic J. Eidson)
- Load pg_hba.conf and pg_ident.conf only on startup and SIGHUP (Bruce)

E.208.3.5. サーバの設定

- Interpretation of some time zone abbreviations as Australian rather than North American now settable at run time (Bruce)
- New parameter to set default transaction isolation level (Peter E)
- New parameter to enable conversion of "expr = NULL" into "expr IS NULL", off by default (Peter E)
- New parameter to control memory usage by VACUUM (Tom)
- New parameter to set client authentication timeout (Tom)
- New parameter to set maximum number of open files (Tom)

E.208.3.6. 問い合わせ

- Statements added by INSERT rules now execute after the INSERT (Jan)
- Prevent unadorned relation names in target list (Bruce)
- NULLs now sort after all normal values in ORDER BY (Tom)
- New IS UNKNOWN, IS NOT UNKNOWN Boolean tests (Tom)
- New SHARE UPDATE EXCLUSIVE lock mode (Tom)
- New EXPLAIN ANALYZE command that shows run times and row counts (Martijn van Oosterhout)
- Fix problem with LIMIT and subqueries (Tom)
- Fix for LIMIT, DISTINCT ON pushed into subqueries (Tom)

- Fix nested EXCEPT/INTERSECT (Tom)

E.208.3.7. スキーマ操作

- Fix SERIAL in temporary tables (Bruce)
- Allow temporary sequences (Bruce)
- Sequences now use int8 internally (Tom)
- New SERIAL8 creates int8 columns with sequences, default still SERIAL4 (Tom)
- Make OIDs optional using WITHOUT OIDS (Tom)
- Add %TYPE syntax to CREATE TYPE (Ian Lance Taylor)
- Add ALTER TABLE / DROP CONSTRAINT for CHECK constraints (Christopher Kings-Lynne)
- New CREATE OR REPLACE FUNCTION to alter existing function (preserving the function OID) (Gavin Sherry)
- Add ALTER TABLE / ADD [UNIQUE | PRIMARY] (Christopher Kings-Lynne)
- Allow column renaming in views
- Make ALTER TABLE / RENAME COLUMN update column names of indexes (Brent Verner)
- Fix for ALTER TABLE / ADD CONSTRAINT ... CHECK with inherited tables (Stephan Szabo)
- ALTER TABLE RENAME update foreign-key trigger arguments correctly (Brent Verner)
- DROP AGGREGATE and COMMENT ON AGGREGATE now accept an aggtype (Tom)
- Add automatic return type data casting for SQL functions (Tom)
- Allow GiST indexes to handle NULLs and multikey indexes (Oleg Bartunov, Teodor Sigaev, Tom)
- Enable partial indexes (Martijn van Oosterhout)

E.208.3.8. ユーティリティコマンド

- Add RESET ALL, SHOW ALL (Marko Kreen)
- CREATE/ALTER USER/GROUP now allow options in any order (Vince)
- Add LOCK A, B, C functionality (Neil Padgett)
- New ENCRYPTED/UNENCRYPTED option to CREATE/ALTER USER (Bruce)
- New light-weight VACUUM does not lock table; old semantics are available as VACUUM FULL (Tom)
- Disable COPY TO/FROM on views (Bruce)
- COPY DELIMITERS string must be exactly one character (Tom)
- VACUUM warning about index tuples fewer than heap now only appears when appropriate (Martijn van Oosterhout)
- Fix privilege checks for CREATE INDEX (Tom)
- Disallow inappropriate use of CREATE/DROP INDEX/TRIGGER/VIEW (Tom)

E.208.3.9. データ型と関数

- SUM(), AVG(), COUNT() now uses int8 internally for speed (Tom)
- Add convert(), convert2() (Tatsuo)
- New function bit_length() (Peter E)

- Make the "n" in CHAR(n)/VARCHAR(n) represents letters, not bytes (Tatsuo)
- CHAR(), VARCHAR() now reject strings that are too long (Peter E)
- BIT VARYING now rejects bit strings that are too long (Peter E)
- BIT now rejects bit strings that do not match declared size (Peter E)
- INET, CIDR text conversion functions (Alex Pilosov)
- INET, CIDR operators << and <=< indexable (Alex Pilosov)
- Bytea ¥### now requires valid three digit octal number
- Bytea comparison improvements, now supports =, <>, >, >=, <, and <=
- Bytea now supports B-tree indexes
- Bytea now supports LIKE, LIKE...ESCAPE, NOT LIKE, NOT LIKE...ESCAPE
- Bytea now supports concatenation
- New bytea functions: position, substring, trim, btrim, and length
- New encode() function mode, "escaped", converts minimally escaped bytea to/from text
- Add pg_database_encoding_max_length() (Tatsuo)
- Add pg_client_encoding() function (Tatsuo)
- now() returns time with millisecond precision (Thomas)
- New TIMESTAMP WITHOUT TIMEZONE data type (Thomas)
- Add ISO date/time specification with "T", yyyy-mm-ddThh:mm:ss (Thomas)
- New xid/int comparison functions (Hiroshi)
- Add precision to TIME, TIMESTAMP, and INTERVAL data types (Thomas)
- Modify type coercion logic to attempt binary-compatible functions first (Tom)
- New encode() function installed by default (Marko Kreen)
- Improved to_*() conversion functions (Karel Zak)
- Optimize LIKE/ILIKE when using single-byte encodings (Tatsuo)
- New functions in contrib/pgcrypto: crypt(), hmac(), encrypt(), gen_salt() (Marko Kreen)
- Correct description of translate() function (Bruce)
- Add INTERVAL argument for SET TIME ZONE (Thomas)
- Add INTERVAL YEAR TO MONTH (etc.) syntax (Thomas)
- Optimize length functions when using single-byte encodings (Tatsuo)
- Fix path_inter, path_distance, path_length, dist_ppath to handle closed paths (Curtis Barrett, Tom)
- octet_length(text) now returns non-compressed length (Tatsuo, Bruce)
- Handle "July" full name in date/time literals (Greg Sabino Mullane)
- Some datatype() function calls now evaluated differently
- Add support for Julian and ISO time specifications (Thomas)

E.208.3.10. 国際化

- National language support in `psql`, `pg_dump`, `libpq`, and `server` (Peter E)
- Message translations in Chinese (simplified, traditional), Czech, French, German, Hungarian, Russian, Swedish (Peter E, Serguei A. Mokhov, Karel Zak, Weiping He, Zhenbang Wei, Kovacs Zoltan)
- Make `trim`, `ltrim`, `rtrim`, `btrim`, `lpad`, `rpadd`, `translate` multibyte aware (Tatsuo)
- Add LATIN5,6,7,8,9,10 support (Tatsuo)
- Add ISO 8859-5,6,7,8 support (Tatsuo)
- Correct LATIN5 to mean ISO-8859-9, not ISO-8859-5 (Tatsuo)
- Make `mic2ascii()` non-ASCII aware (Tatsuo)
- Reject invalid multibyte character sequences (Tatsuo)

E.208.3.11. PL/pgSQL

- Now uses portals for `SELECT` loops, allowing huge result sets (Jan)
- `CURSOR` and `REFCURSOR` support (Jan)
- Can now return open cursors (Jan)
- Add `ELSEIF` (Klaus Reger)
- Improve PL/pgSQL error reporting, including location of error (Tom)
- Allow `IS` or `FOR` key words in cursor declaration, for compatibility (Bruce)
- Fix for `SELECT ... FOR UPDATE` (Tom)
- Fix for `PERFORM` returning multiple rows (Tom)
- Make PL/pgSQL use the server's type coercion code (Tom)
- Memory leak fix (Jan, Tom)
- Make trailing semicolon optional (Tom)

E.208.3.12. PL/Perl

- New untrusted PL/Perl (Alex Piliusov)
- PL/Perl is now built on some platforms even if `libperl` is not shared (Peter E)

E.208.3.13. PL/Tcl

- Now reports `errorInfo` (Vsevolod Lobko)
- Add `spi_lastoid` function (bob@redivi.com)

E.208.3.14. PL/Python

- ...is new (Andrew Bosma)

E.208.3.15. psql

- `%d` displays indexes in unique, primary groupings (Christopher Kings-Lynne)
- Allow trailing semicolons in backslash commands (Greg Sabino Mullane)
- Read password from `/dev/tty` if possible
- Force new password prompt when changing user and database (Tatsuo, Tom)
- Format the correct number of columns for Unicode (Patrice)

E.208.3.16. libpq

- New function PQescapeString() to escape quotes in command strings (Florian Weimer)
- New function PQescapeBytea() escapes binary strings for use as SQL string literals

E.208.3.17. JDBC

- Return OID of INSERT (Ken K)
- Handle more data types (Ken K)
- Handle single quotes and newlines in strings (Ken K)
- Handle NULL variables (Ken K)
- Fix for time zone handling (Barry Lind)
- Improved Druid support
- Allow eight-bit characters with non-multibyte server (Barry Lind)
- Support BIT, BINARY types (Ned Wolpert)
- Reduce memory usage (Michael Stephens, Dave Cramer)
- Update DatabaseMetaData (Peter E)
- Add DatabaseMetaData.getCatalogs() (Peter E)
- Encoding fixes (Anders Bengtsson)
- Get/setCatalog methods (Jason Davies)
- DatabaseMetaData.getColumns() now returns column defaults (Jason Davies)
- DatabaseMetaData.getColumns() performance improvement (Jeroen van Vianen)
- Some JDBC1 and JDBC2 merging (Anders Bengtsson)
- Transaction performance improvements (Barry Lind)
- Array fixes (Greg Zoller)
- Serialize addition
- Fix batch processing (Rene Pijlman)
- ExecSQL method reorganization (Anders Bengtsson)
- GetColumn() fixes (Jeroen van Vianen)
- Fix isWriteable() function (Rene Pijlman)
- Improved passage of JDBC2 conformance tests (Rene Pijlman)
- Add bytea type capability (Barry Lind)
- Add isNullable() (Rene Pijlman)
- JDBC date/time test suite fixes (Liam Stewart)
- Fix for SELECT 'id' AS xxx FROM table (Dave Cramer)
- Fix DatabaseMetaData to show precision properly (Mark Lillywhite)
- New getImported/getExported keys (Jason Davies)
- MD5 password encryption support (Jeremy Wohl)

- Fix to actually use type cache (Ned Wolpert)

E.208.3.18. ODBC

- Remove query size limit (Hiroshi)
- Remove text field size limit (Hiroshi)
- Fix for SQLPrimaryKeys in multibyte mode (Hiroshi)
- Allow ODBC procedure calls (Hiroshi)
- Improve boolean handing (Aidan Mountford)
- Most configuration options now settable via DSN (Hiroshi)
- Multibyte, performance fixes (Hiroshi)
- Allow driver to be used with iODBC or unixODBC (Peter E)
- MD5 password encryption support (Bruce)
- Add more compatibility functions to odbc.sql (Peter E)

E.208.3.19. ECPG

- EXECUTE ... INTO implemented (Christof Petig)
- Multiple row descriptor support (e.g. CARDINALITY) (Christof Petig)
- Fix for GRANT parameters (Lee Kindness)
- Fix INITIALLY DEFERRED bug
- Various bug fixes (Michael, Christof Petig)
- Auto allocation for indicator variable arrays (int *ind_p=NULL)
- Auto allocation for string arrays (char **foo_pp=NULL)
- ECPGfree_auto_mem fixed
- All function names with external linkage are now prefixed by ECPG
- Fixes for arrays of structures (Michael)

E.208.3.20. その他のインタフェース

- Python fix fetchone() (Gerhard Haring)
- Use UTF, Unicode in Tcl where appropriate (Vsevolod Lobko, Reinhard Max)
- Add Tcl COPY TO/FROM (ljb)
- Prevent output of default index op class in pg_dump (Tom)
- Fix libpgeasy memory leak (Bruce)

E.208.3.21. 構築作業とインストール

- Configure, dynamic loader, and shared library fixes (Peter E)
- Fixes in QNX 4 port (Bernd Tegge)
- Fixes in Cygwin and Windows ports (Jason Tishler, Gerhard Haring, Dmitry Yurtaev, Darko Prenosil, Mikhail Terekhov)
- Fix for Windows socket communication failures (Magnus, Mikhail Terekhov)

- Hurd compile fix (Oliver Elphick)
- BeOS fixes (Cyril Velter)
- Remove configure --enable-unicode-conversion, now enabled by multibyte (Tatsuo)
- AIX fixes (Tatsuo, Andreas)
- Fix parallel make (Peter E)
- Install SQL language manual pages into OS-specific directories (Peter E)
- Rename config.h to pg_config.h (Peter E)
- Reorganize installation layout of header files (Peter E)

E.208.3.22. ソースコード

- Remove SEP_CHAR (Bruce)
- New GUC hooks (Tom)
- Merge GUC and command line handling (Marko Kreen)
- Remove EXTEND INDEX (Martijn van Oosterhout, Tom)
- New pgjindent utility to indent java code (Bruce)
- Remove define of true/false when compiling under C++ (Leandro Fanzone, Tom)
- pgindent fixes (Bruce, Tom)
- Replace strcasecmp() with strcmp() where appropriate (Peter E)
- Dynahash portability improvements (Tom)
- Add 'volatile' usage in spinlock structures
- Improve signal handling logic (Tom)

E.208.3.23. 寄贈

- New contrib/rtree_gist (Oleg Bartunov, Teodor Sigaev)
- New contrib/tsearch full-text indexing (Oleg, Teodor Sigaev)
- Add contrib/dblink for remote database access (Joe Conway)
- contrib/ora2pg Oracle conversion utility (Gilles Darold)
- contrib/xml XML conversion utility (John Gray)
- contrib/fulltextindex fixes (Christopher Kings-Lynne)
- New contrib/fuzzystrmatch with levenshtein and metaphone, soundex merged (Joe Conway)
- Add contrib/intarray boolean queries, binary search, fixes (Oleg Bartunov)
- New pg_upgrade utility (Bruce)
- Add new pg_resetxlog options (Bruce, Tom)

E.209. リリース 7.1.3

リリース日: 2001-08-15

E.209.1. 7.1.3 への移行方法

7.1.X からの移行の場合は ダンプ/リストアは必要ありません。

E.209.2. 変更点

Remove unused WAL segments of large transactions (Tom)
Multiaction rule fix (Tom)
PL/pgSQL memory allocation fix (Jan)
VACUUM buffer fix (Tom)
Regression test fixes (Tom)
pg_dump fixes for GRANT/REVOKE/comments on views, user-defined types (Tom)
Fix subselects with DISTINCT ON or LIMIT (Tom)
BeOS fix
Disable COPY TO/FROM a view (Tom)
Cygwin build (Jason Tishler)

E.210. リリース 7.1.2

リリース日: 2001-05-11

このリリースは 7.1.1 の不具合を改修したものです。

E.210.1. バージョン 7.1.2 への移行方法

7.1.X からの移行の場合は ダンプ/リストアは必要ありません。

E.210.2. 変更点

```
Fix PL/pgSQL SELECTs when returning no rows
Fix for psql backslash core dump
Referential integrity privilege fix
Optimizer fixes
pg_dump cleanups
```


E.211. リリース 7.1.2

リリース日: 2001-05-05

このリリースは 7.1 の不具合を改修したものです。

E.211.1. バージョン 7.1.1 への移行方法

7.1.X からの移行の場合は ダンプ/リストアは必要ありません。

E.211.2. 変更点

Fix for numeric MODULO operator (Tom)
pg_dump fixes (Philip)
pg_dump can dump 7.0 databases (Philip)
readline 4.2 fixes (Peter E)
JOIN fixes (Tom)
AIX, MSWIN, VAX, N32K fixes (Tom)
Multibytes fixes (Tom)
Unicode fixes (Tatsuo)
Optimizer improvements (Tom)
Fix for whole rows in functions (Tom)
Fix for pg_ctl and option strings with spaces (Peter E)
ODBC fixes (Hiroshi)
EXTRACT can now take string argument (Thomas)
Python fixes (Darcy)

E.212. リリース 7.1

リリース日: 2001-04-13

このリリースは、過去長い間PostgreSQLのコードにあった、制限を取り去ることに焦点をあてたものです。

このリリースにおける大きな変更は以下の通りです。

ログ先行書き込み(WAL)

オペレーティングシステムのクラッシュ時のデータベースの一貫性を保つために、以前の PostgreSQLのリリースでは、各トランザクションのコミットの前に変更されたデータを全て強制的にディスクに書き込んでいました。WAL によって、ログファイルのみがディスクに吐き出されることになり、性能が大幅に改善されました。以前のリリースで -F を使用して、ディスクの吐き出しを止めていた場合、それを取り止めることを検討する方が良いでしょう。

TOAST

TOAST。これまでのリリースでは行の長さ制限はコンパイル時に決定されました。通常は 8Kから 32 k です。この制限は、長めのテキストフィールドの格納が困難でした。TOAST によって、任意長の大きな行を優れた性能で格納できます。

外部結合

外部結合をサポートするようになりました。外部結合の代替方法としてのUNION/NOT IN は不要となりました。SQL92の外部結合の構文を使用しています。

関数マネージャ

これまでの C 関数マネージャは NULL 値を適切に扱いませんでした。また、64ビット CPU (Alpha) をサポートしませんでした。新しい関数マネージャではこれらを行います。古い独自の関数を使用し続けることは可能ですが、新しい関数マネージャ呼び出しインタフェースを使用するように書き換えることを勧めます。

複雑な問い合わせ

これまでサポートされなかった多数の複雑な問い合わせが動作するようになりました。ビュー、集約、UNION、LIMIT、カーソル、副問い合わせ、継承されたテーブルを多く組み合わせたものが適切に動作するようになりました。継承されたテーブルはデフォルトでアクセスされるようになりました。FROM での副問い合わせもサポートされるようになりました。

E.212.1. バージョン 7.1 への移行

以前のリリースからデータを移行する場合、pg_dump によるダンプ/リストアが必要です。

E.212.2. 変更点

```
Bug Fixes
-----
Many multibyte/Unicode/locale fixes (Tatsuo and others)
More reliable ALTER TABLE RENAME (Tom)
Kerberos V fixes (David Wragg)
Fix for INSERT INTO...SELECT where targetlist has subqueries (Tom)
Prompt username/password on standard error (Bruce)
Large objects inv_read/inv_write fixes (Tom)
Fixes for to_char(), to_date(), to_ascii(), and to_timestamp() (Karel,
Daniel Baldoni)
Prevent query expressions from leaking memory (Tom)
Allow UPDATE of arrays elements (Tom)
Wake up lock waiters during cancel (Hiroshi)
Fix rare cursor crash when using hash join (Tom)
Fix for DROP TABLE/INDEX in rolled-back transaction (Hiroshi)
Fix psql crash from ¥1+ if MULTIBYTE enabled (Peter E)
Fix truncation of rule names during CREATE VIEW (Ross Reedstrom)
Fix PL/perl (Alex Kapranoff)
Disallow LOCK on views (Mark Hollomon)
Disallow INSERT/UPDATE/DELETE on views (Mark Hollomon)
Disallow DROP RULE, CREATE INDEX, TRUNCATE on views (Mark Hollomon)
Allow PL/pgsql accept non-ASCII identifiers (Tatsuo)
Allow views to properly handle GROUP BY, aggregates, DISTINCT (Tom)
```

Fix rare failure with TRUNCATE command (Tom)
 Allow UNION/INTERSECT/EXCEPT to be used with ALL, subqueries, views, DISTINCT, ORDER BY, SELECT...INTO (Tom)
 Fix parser failures during aborted transactions (Tom)
 Allow temporary relations to properly clean up indexes (Bruce)
 Fix VACUUM problem with moving rows in same page (Tom)
 Modify pg_dump to better handle user-defined items in template1 (Philip)
 Allow LIMIT in VIEW (Tom)
 Require cursor FETCH to honor LIMIT (Tom)
 Allow PRIMARY/FOREIGN Key definitions on inherited columns (Stephan)
 Allow ORDER BY, LIMIT in subqueries (Tom)
 Allow UNION in CREATE RULE (Tom)
 Make ALTER/DROP TABLE rollback-able (Vadim, Tom)
 Store initdb collation in pg_control so collation cannot be changed (Tom)
 Fix INSERT...SELECT with rules (Tom)
 Fix FOR UPDATE inside views and subselects (Tom)
 Fix OVERLAPS operators conform to SQL92 spec regarding NULLs (Tom)
 Fix lpad() and rpad() to handle length less than input string (Tom)
 Fix use of NOTIFY in some rules (Tom)
 Overhaul btree code (Tom)
 Fix NOT NULL use in Pl/pgSQL variables (Tom)
 Overhaul GIST code (Oleg)
 Fix CLUSTER to preserve constraints and column default (Tom)
 Improved deadlock detection handling (Tom)
 Allow multiple SERIAL columns in a table (Tom)
 Prevent occasional index corruption (Vadim)

Enhancements

Add OUTER JOINS (Tom)
 Function manager overhaul (Tom)
 Allow ALTER TABLE RENAME on indexes (Tom)
 Improve CLUSTER (Tom)
 Improve ps status display for more platforms (Peter E, Marc)
 Improve CREATE FUNCTION failure message (Ross)
 JDBC improvements (Peter, Travis Bauer, Christopher Cain, William Webber, Gunnar)
 Grand Unified Configuration scheme/GUC. Many options can now be set in data/postgresql.conf, postmaster/postgres flags, or SET commands (Peter E)
 Improved handling of file descriptor cache (Tom)
 New warning code about auto-created table alias entries (Bruce)
 Overhaul initdb process (Tom, Peter E)
 Overhaul of inherited tables; inherited tables now accessed by default; new ONLY key word prevents it (Chris Bitmead, Tom)
 ODBC cleanups/improvements (Nick Gorham, Stephan Szabo, Zoltan Kovacs, Michael Fork)
 Allow renaming of temp tables (Tom)
 Overhaul memory manager contexts (Tom)
 pg_dumpall uses CREATE USER or CREATE GROUP rather using COPY (Peter E)
 Overhaul pg_dump (Philip Warner)
 Allow pg_hba.conf secondary password file to specify only username (Peter E)
 Allow TEMPORARY or TEMP key word when creating temporary tables (Bruce)
 New memory leak checker (Karel)
 New SET SESSION CHARACTERISTICS (Thomas)
 Allow nested block comments (Thomas)
 Add WITHOUT TIME ZONE type qualifier (Thomas)
 New ALTER TABLE ADD CONSTRAINT (Stephan)
 Use NUMERIC accumulators for INTEGER aggregates (Tom)
 Overhaul aggregate code (Tom)
 New VARIANCE and STDDEV() aggregates
 Improve dependency ordering of pg_dump (Philip)
 New pg_restore command (Philip)
 New pg_dump tar output option (Philip)
 New pg_dump of large objects (Philip)
 New ESCAPE option to LIKE (Thomas)
 New case-insensitive LIKE - ILIKE (Thomas)
 Allow functional indexes to use binary-compatible type (Tom)
 Allow SQL functions to be used in more contexts (Tom)
 New pg_config utility (Peter E)
 New PL/pgSQL EXECUTE command which allows dynamic SQL and utility statements (Jan)
 New PL/pgSQL GET DIAGNOSTICS statement for SPI value access (Jan)
 New quote_identifiers() and quote_literal() functions (Jan)
 New ALTER TABLE table OWNER TO user command (Mark Hollomon)
 Allow subselects in FROM, i.e. FROM (SELECT ...) [AS] alias (Tom)
 Update PyGreSQL to version 3.1 (D'Arcy)
 Store tables as files named by OID (Vadim)
 New SQL function setval(seq,val,bool) for use in pg_dump (Philip)
 Require DROP VIEW to remove views, no DROP TABLE (Mark)
 Allow DROP VIEW view1, view2 (Mark)
 Allow multiple objects in DROP INDEX, DROP RULE, and DROP TYPE (Tom)
 Allow automatic conversion to/from Unicode (Tatsuo, Eiji)
 New /contrib/pgcrypto hashing functions (Marko Kreen)
 New pg_dumpall --globals-only option (Peter E)
 New CHECKPOINT command for WAL which creates new WAL log file (Vadim)
 New AT TIME ZONE syntax (Thomas)

Allow location of Unix domain socket to be configurable (David J. MacKenzie)
Allow postmaster to listen on a specific IP address (David J. MacKenzie)
Allow socket path name to be specified in hostname by using leading slash
(David J. MacKenzie)
Allow CREATE DATABASE to specify template database (Tom)
New utility to convert MySQL schema dumps to SQL92 and PostgreSQL (Thomas)
New /contrib/rserv replication toolkit (Vadim)
New file format for COPY BINARY (Tom)
New /contrib/oid2name to map numeric files to table names (B Palmer)
New "idle in transaction" ps status message (Marc)
Update to pgaccess 0.98.7 (Constantin Teodorescu)
pg_ctl now defaults to -w (wait) on shutdown, new -l (log) option
Add rudimentary dependency checking to pg_dump (Philip)

Types

Fix INET/CIDR type ordering and add new functions (Tom)
Make OID behave as an unsigned type (Tom)
Allow BIGINT as synonym for INT8 (Peter E)
New int2 and int8 comparison operators (Tom)
New BIT and BIT VARYING types (Adriaan Joubert, Tom, Peter E)
CHAR() no longer faster than VARCHAR() because of TOAST (Tom)
New GIST seg/cube examples (Gene Selkov)
Improved round(numeric) handling (Tom)
Fix CIDR output formatting (Tom)
New CIDR abbrev() function (Tom)

Performance

Write-Ahead Log (WAL) to provide crash recovery with less performance
overhead (Vadim)
ANALYZE stage of VACUUM no longer exclusively locks table (Bruce)
Reduced file seeks (Denis Perchine)
Improve BTREE code for duplicate keys (Tom)
Store all large objects in a single table (Denis Perchine, Tom)
Improve memory allocation performance (Karel, Tom)

Source Code

New function manager call conventions (Tom)
SGI portability fixes (David Kaelbling)
New configure --enable-syslog option (Peter E)
New BSDI README (Bruce)
configure script moved to top level, not /src (Peter E)
Makefile/configuration/compilation overhaul (Peter E)
New configure --with-python option (Peter E)
Solaris cleanups (Peter E)
Overhaul /contrib Makefiles (Karel)
New OpenSSL configuration option (Magnus, Peter E)
AIX fixes (Andreas)
QNX fixes (Maurizio)
New heap_open(), heap_openr() API (Tom)
Remove colon and semi-colon operators (Thomas)
New pg_class.relkind value for views (Mark Hollomon)
Rename ichar() to chr() (Karel)
New documentation for btrim(), ascii(), chr(), repeat() (Karel)
Fixes for NT/Cygwin (Pete Forman)
AIX port fixes (Andreas)
New BeOS port (David Reid, Cyril Velter)
Add proofreader's changes to docs (Addison-Wesley, Bruce)
New Alpha spinlock code (Adriaan Joubert, Compaq)
UnixWare port overhaul (Peter E)
New Darwin/Mac OS X port (Peter Bierman, Bruce Hartzler)
New FreeBSD Alpha port (Alfred)
Overhaul shared memory segments (Tom)
Add IBM S/390 support (Neale Ferguson)
Moved macmanuf to /contrib (Larry Rosenman)
Syslog improvements (Larry Rosenman)
New template0 database that contains no user additions (Tom)
New /contrib/cube and /contrib/seg GIST sample code (Gene Selkov)
Allow NetBSD's libedit instead of readline (Peter)
Improved assembly language source code format (Bruce)
New contrib/pg_logger
New --template option to createdb
New contrib/pg_control utility (Oliver)
New FreeBSD tools ipc_check, start-scripts/freebsd

E.213. リリース 7.0.3

リリース日: 2000-11-11

このリリースは 7.0.2 の不具合を改修したものです。

E.213.1. バージョン 7.0.3 への移行方法

7.0.* からの移行の場合は ダンプ/リストアは必要ありません。

E.213.2. 変更点

```
Jdbc fixes (Peter)
Large object fix (Tom)
Fix lean in COPY WITH OIDS leak (Tom)
Fix backwards-index-scan (Tom)
Fix SELECT ... FOR UPDATE so it checks for duplicate keys (Hiroshi)
Add --enable-syslog to configure (Marc)
Fix abort transaction at backend exit in rare cases (Tom)
Fix for psql ¥l+ when multibyte enabled (Tatsuo)
Allow PL/pgSQL to accept non ascii identifiers (Tatsuo)
Make vacuum always flush buffers (Tom)
Fix to allow cancel while waiting for a lock (Hiroshi)
Fix for memory allocation problem in user authentication code (Tom)
Remove bogus use of int4out() (Tom)
Fixes for multiple subqueries in COALESCE or BETWEEN (Tom)
Fix for failure of triggers on heap open in certain cases (Jeroen van
Vianen)
Fix for erroneous selectivity of not-equals (Tom)
Fix for erroneous use of strcmp() (Tom)
Fix for bug where storage manager accesses items beyond end of file
(Tom)
Fix to include kernel errno message in all smgr elog messages (Tom)
Fix for '.' not in PATH at build time (SL Baur)
Fix for out-of-file-descriptors error (Tom)
Fix to make pg_dump dump 'iscachable' flag for functions (Tom)
Fix for subselect in targetlist of Append node (Tom)
Fix for mergejoin plans (Tom)
Fix TRUNCATE failure on relations with indexes (Tom)
Avoid database-wide restart on write error (Hiroshi)
Fix nodeMaterial to honor chgParam by recomputing its output (Tom)
Fix VACUUM problem with moving chain of update row versions when source
and destination of a row version lie on the same page (Tom)
Fix user.c CommandCounterIncrement (Tom)
Fix for AM/PM boundary problem in to_char() (Karel Zak)
Fix TIME aggregate handling (Tom)
Fix to_char() to avoid coredump on NULL input (Tom)
Buffer_fix (Tom)
Fix for inserting/copying longer multibyte strings into char() data
types (Tatsuo)
Fix for crash of backend, on abort (Tom)
```

E.214. リリース 7.0.2

リリース日: 2000-06-05

このリリースは 7.0.1 のパッケージに文書を追加したものです。

E.214.1. バージョン 7.0.2 への移行方法

7.* からの移行の場合は ダンプ/リストアは必要ありません。

E.214.2. 変更点

tarball に文書を追加した。

E.215. リリース 7.0.1

リリース日: 2000-06-01

このリリースは 7.0 を整理したものです。

E.215.1. バージョン 7.0.1 への移行方法

7.0 からの移行の場合は ダンプ/リストアは必要ありません。

E.215.2. 変更点

Fix many CLUSTER failures (Tom)
Allow ALTER TABLE RENAME works on indexes (Tom)
Fix plpgsql to handle datetime->timestamp and timespan->interval (Bruce)
New configure --with-setproctitle switch to use setproctitle() (Marc, Bruce)
Fix the off by one errors in ResultSet from 6.5.3, and more.
jdbc ResultSet fixes (Joseph Shraibman)
optimizer tunings (Tom)
Fix create user for pgaccess
Fix for UNLISTEN failure
IRIX fixes (David Kaelbling)
QNX fixes (Andreas Kardos)
Reduce COPY IN lock level (Tom)
Change libpqeasy to use PQconnectdb() style parameters (Bruce)
Fix pg_dump to handle OID indexes (Tom)
Fix small memory leak (Tom)
Solaris fix for createdb/dropdb (Tatsuo)
Fix for non-blocking connections (Alfred Perlstein)
Fix improper recovery after RENAME TABLE failures (Tom)
Copy pg_ident.conf.sample into /lib directory in install (Bruce)
Add SJIS UDC (NEC selection IBM kanji) support (Eiji Tokuya)
Fix too long syslog message (Tatsuo)
Fix problem with quoted indexes that are too long (Tom)
JDBC ResultSet.getTimestamp() fix (Gregory Krasnow & Floyd Marinescu)
ecpg changes (Michael)

E.216. リリース 7.0

リリース日: 2000-05-08

このリリースには広範囲に渡る改良が行われており、PostgreSQLの成長が続いていることを示しています。7.0 で行われた改良や改修はこれまでのリリースよりも多くなっています。開発者はこれが最善のリリースであるという確信を持っています。 確かなリリースのみを発表することに全力を注いでいます。 このリリースも例外ではありません。

このリリースにおける大きな変更は以下の通りです。

外部キー

PARTIAL MATCH 外部キー以外の外部キーが実装されました。多くのユーザから求められていた機能ですので、提供することができて嬉しく思っています。

オプティマイザの見直し

1年ほど作業を行い、オプティマイザの改良を行いました。 より少ないメモリ、とより高速な性能でより良い問い合わせ計画の選択を行うことができます。

psql の更新

対話式端末モニタ psql は、多くの新しい機能を持つように更新されました。詳細は psql のマニュアルを参照して下さい。

結合の構文

このリリースでは INNER JOIN のみですが、SQL92 の結合構文がサポートされました。JOIN、NATURAL JOIN、JOIN/USING、JOIN/ON が、列の相関名として使用することができます。

E.216.1. バージョン 7.0 への移行

以前のリリースからデータを移行する場合、pg_dump によるダンプ/リストアが必要です。6.5.* からのアップグレードの場合、pg_upgrade を使用してこのリリースにアップグレードすることもできます。しかし、完全なインストールのダンプをリストアする方法が、常に堅牢なアップグレードの方法です。

新しいリリースにて検討すべきインタフェースと互換性に関する問題として、以下の項目があります。

- `datetime` と `timespan` 日付/時刻型はSQL92 で定義された `timestamp` と `interval` 型に変更されました。移行を簡単にするために PostgreSQL が 古い名前の型を認識して、新しい型に変換するといった対応を行っていますが、この機構は、既存のアプリケーションでは完全に透過的ではないかもしれません。
- オプティマイザは、問い合わせコスト概算の部分で更に改良されました。これにより、準備された計画をより良く選択した時間問い合わせ時間は減少する場合があります。しかし、あまりありませんが、病的にデータが分散したような場合、問い合わせ時間は増加する可能性があります。大量のデータを扱う場合、問い合わせを試験して性能を確認することを勧めます。
- JDBC および ODBC インタフェースがアップグレードされ、また、拡張されました。
- `CHAR_LENGTH` 文字列関数が固有の関数となりました。以前のバージョンでは、これは `LENGTH` 関数呼び出しに変換されており、幾何データ型などの他のデータ型における `LENGTH` の実装と多義になっていました。

E.216.2. 変更点

Bug Fixes

```

Prevent function calls exceeding maximum number of arguments (Tom)
Improve CASE construct (Tom)
Fix SELECT coalesce(f1,0) FROM int4_tbl GROUP BY f1 (Tom)
Fix SELECT sentence.words[0] FROM sentence GROUP BY sentence.words[0] (Tom)
Fix GROUP BY scan bug (Tom)
Improvements in SQL grammar processing (Tom)
Fix for views involved in INSERT ... SELECT ... (Tom)
Fix for SELECT a/2, a/2 FROM test_missing_target GROUP BY a/2 (Tom)
Fix for subselects in INSERT ... SELECT (Tom)
Prevent INSERT ... SELECT ... ORDER BY (Tom)
Fixes for relations greater than 2GB, including vacuum
Improve propagating system table changes to other backends (Tom)

```

Improve propagating user table changes to other backends (Tom)
 Fix handling of temp tables in complex situations (Bruce, Tom)
 Allow table locking at table open, improving concurrent reliability (Tom)
 Properly quote sequence names in pg_dump (Ross J. Reedstrom)
 Prevent DROP DATABASE while others accessing
 Prevent any rows from being returned by GROUP BY if no rows processed (Tom)
 Fix SELECT COUNT(1) FROM table WHERE ... if no rows matching WHERE (Tom)
 Fix pg_upgrade so it works for MVCC (Tom)
 Fix for SELECT ... WHERE x IN (SELECT ... HAVING SUM(x) > 1) (Tom)
 Fix for "f1 datetime DEFAULT 'now'" (Tom)
 Fix problems with CURRENT_DATE used in DEFAULT (Tom)
 Allow comment-only lines, and ;;; lines too. (Tom)
 Improve recovery after failed disk writes, disk full (Hiroshi)
 Fix cases where table is mentioned in FROM but not joined (Tom)
 Allow HAVING clause without aggregate functions (Tom)
 Fix for "---" comment and no trailing newline, as seen in perl interface
 Improve pg_dump failure error reports (Bruce)
 Allow sorts and hashes to exceed 2GB file sizes (Tom)
 Fix for pg_dump dumping of inherited rules (Tom)
 Fix for NULL handling comparisons (Tom)
 Fix inconsistent state caused by failed CREATE/DROP commands (Hiroshi)
 Fix for dbname with dash
 Prevent DROP INDEX from interfering with other backends (Tom)
 Fix file descriptor leak in verify_password()
 Fix for "Unable to identify an operator =" problem
 Fix ODBC so no segfault if CommLog and Debug enabled (Dirk Niggemann)
 Fix for recursive exit call (Massimo)
 Fix for extra-long timezones (Jeroen van Vianen)
 Make pg_dump preserve primary key information (Peter E)
 Prevent databases with single quotes (Peter E)
 Prevent DROP DATABASE inside transaction (Peter E)
 ecpg memory leak fixes (Stephen Birch)
 Fix for SELECT null::text, SELECT int4fac(null) and SELECT 2 + (null) (Tom)
 Y2K timestamp fix (Massimo)
 Fix for VACUUM 'HEAP MOVED IN was not expected' errors (Tom)
 Fix for views with tables/columns containing spaces (Tom)
 Prevent privileges on indexes (Peter E)
 Fix for spinlock stuck problem when error is generated (Hiroshi)
 Fix ipcclean on Linux
 Fix handling of NULL constraint conditions (Tom)
 Fix memory leak in odbc driver (Nick Gorham)
 Fix for privilege check on UNION tables (Tom)
 Fix to allow SELECT 'a' LIKE 'a' (Tom)
 Fix for SELECT 1 + NULL (Tom)
 Fixes to CHAR
 Fix log() on numeric type (Tom)
 Deprecate ':' and ';' operators
 Allow vacuum of temporary tables
 Disallow inherited columns with the same name as new columns
 Recover or force failure when disk space is exhausted (Hiroshi)
 Fix INSERT INTO ... SELECT with AS columns matching result columns
 Fix INSERT ... SELECT ... GROUP BY groups by target columns not source columns (Tom)
 Fix CREATE TABLE test (a char(5) DEFAULT text '', b int4) with INSERT (Tom)
 Fix UNION with LIMIT
 Fix CREATE TABLE x AS SELECT 1 UNION SELECT 2
 Fix CREATE TABLE test(col char(2) DEFAULT user)
 Fix mismatched types in CREATE TABLE ... DEFAULT
 Fix SELECT * FROM pg_class where oid in (0,-1)
 Fix SELECT COUNT('asdf') FROM pg_class WHERE oid=12
 Prevent user who can create databases can modifying pg_database table (Peter E)
 Fix btree to give a useful elog when key > 1/2 (page - overhead) (Tom)
 Fix INSERT of 0.0 into DECIMAL(4,4) field (Tom)

Enhancements

 New CLI interface include file sqlcli.h, based on SQL3/SQL98
 Remove all limits on query length, row length limit still exists (Tom)
 Update jdbc protocol to 2.0 (Jens Glaser <jens@jens.de>)
 Add TRUNCATE command to quickly truncate relation (Mike Mascari)
 Fix to give super user and createdb user proper update catalog rights (Peter E)
 Allow ecpg bool variables to have NULL values (Christof)
 Issue ecpg error if NULL value for variable with no NULL indicator (Christof)
 Allow ^C to cancel COPY command (Massimo)
 Add SET FSYNC and SHOW PG_OPTIONS commands (Massimo)
 Function name overloading for dynamically-loaded C functions (Frankpitt)
 Add CmdTuples() to libpq++ (Vince)
 New CREATE CONSTRAINT TRIGGER and SET CONSTRAINTS commands (Jan)
 Allow CREATE FUNCTION/WITH clause to be used for all language types
 configure --enable-debug adds -g (Peter E)
 configure --disable-debug removes -g (Peter E)
 Allow more complex default expressions (Tom)
 First real FOREIGN KEY constraint trigger functionality (Jan)
 Add FOREIGN KEY ... MATCH FULL ... ON DELETE CASCADE (Jan)
 Add FOREIGN KEY ... MATCH <unspecified> referential actions (Don Baccus)
 Allow WHERE restriction on ctid (physical heap location) (Hiroshi)
 Move pginterface from contrib to interface directory, rename to pgeasy (Bruce)
 Change pgeasy connectdb() parameter ordering (Bruce)
 Require SELECT DISTINCT target list to have all ORDER BY columns (Tom)
 Add Oracle's COMMENT ON command (Mike Mascari <mascari@yahoo.com>)
 libpq's PQsetNoticeProcessor function now returns previous hook (Peter E)
 Prevent PQsetNoticeProcessor from being set to NULL (Peter E)
 Make USING in COPY optional (Bruce)
 Allow subselects in the target list (Tom)
 Allow subselects on the left side of comparison operators (Tom)
 New parallel regression test (Jan)
 Change backend-side COPY to write files with permissions 644 not 666 (Tom)
 Force permissions on PGDATA directory to be secure, even if it exists (Tom)
 Added psql LASTOID variable to return last inserted oid (Peter E)
 Allow concurrent vacuum and remove pg_vlock vacuum lock file (Tom)
 Add privilege check for vacuum (Peter E)
 New libpq functions to allow asynchronous connections: PQconnectStart(),
 PQconnectPoll(), PQresetStart(), PQresetPoll(), PQsetenvStart(),
 PQsetenvPoll(), PQsetenvAbort (Ewan Mellor)

New libpq PQsetenv() function (Ewan Mellor)
 create/alter user extension (Peter E)
 New postmaster.pid and postmaster.opts under \$PGDATA (Tatsuo)
 New scripts for create/drop user/db (Peter E)
 Major psql overhaul (Peter E)
 Add const to libpq interface (Peter E)
 New libpq function PQoidValue (Peter E)
 Show specific non-aggregate causing problem with GROUP BY (Tom)
 Make changes to pg_shadow recreate pg_pwd file (Peter E)
 Add aggregate(DISTINCT ...) (Tom)
 Allow flag to control COPY input/output of NULLs (Peter E)
 Make postgres user have a password by default (Peter E)
 Add CREATE/ALTER/DROP GROUP (Peter E)
 All administration scripts now support --long options (Peter E, Karel)
 Vacuumdb script now supports --all option (Peter E)
 ecpg new portable FETCH syntax
 Add ecpg EXEC SQL IFDEF, EXEC SQL IFNDEF, EXEC SQL ELSE, EXEC SQL ELIF
 and EXEC SQL ENDIF directives
 Add pg_ctl script to control backend start-up (Tatsuo)
 Add postmaster.opts.default file to store start-up flags (Tatsuo)
 Allow --with-mb=SQL_ASCII
 Increase maximum number of index keys to 16 (Bruce)
 Increase maximum number of function arguments to 16 (Bruce)
 Allow configuration of maximum number of index keys and arguments (Bruce)
 Allow unprivileged users to change their passwords (Peter E)
 Password authentication enabled; required for new users (Peter E)
 Disallow dropping a user who owns a database (Peter E)
 Change initdb option --with-mb to --enable-multibyte
 Add option for initdb to prompts for superuser password (Peter E)
 Allow complex type casts like col::numeric(9,2) and col::int2::float8 (Tom)
 Updated user interfaces on initdb, initlocation, pg_dump, ipcclean (Peter E)
 New pg_char to encoding() and pg_encoding_to_char() functions (Tatsuo)
 libpq non-blocking mode (Alfred Perlstein)
 Improve conversion of types in casts that don't specify a length
 New plperl internal programming language (Mark Hollomon)
 Allow COPY IN to read file that do not end with a newline (Tom)
 Indicate when long identifiers are truncated (Tom)
 Allow aggregates to use type equivalency (Peter E)
 Add Oracle's to_char(), to_date(), to_datetime(), to_timestamp(), to_number()
 conversion functions (Karel Zak <zakkr@zf.jcu.cz>)
 Add SELECT DISTINCT ON (expr [, expr ...]) targetlist ... (Tom)
 Check to be sure ORDER BY is compatible with the DISTINCT operation (Tom)
 Add NUMERIC and int8 types to ODBC
 Improve EXPLAIN results for Append, Group, Agg, Unique (Tom)
 Add ALTER TABLE ... ADD FOREIGN KEY (Stephan Szabo)
 Allow SELECT .. FOR UPDATE in PL/pgSQL (Hiroshi)
 Enable backward sequential scan even after reaching EOF (Hiroshi)
 Add btree indexing of boolean values, >= and <= (Don Baccus)
 Print current line number when COPY FROM fails (Massimo)
 Recognize POSIX time zone e.g. "PST+8" and "GMT-8" (Thomas)
 Add DEC as synonym for DECIMAL (Thomas)
 Add SESSION USER as SQL92 key word, same as CURRENT USER (Thomas)
 Implement SQL92 column aliases (aka correlation names) (Thomas)
 Implement SQL92 join syntax (Thomas)
 Make INTERVAL reserved word allowed as a column identifier (Thomas)
 Implement REINDEX command (Hiroshi)
 Accept ALL in aggregate function SUM(ALL col) (Tom)
 Prevent GROUP BY from using column aliases (Tom)
 New psql %encoding option (Tatsuo)
 Allow PQrequestCancel() to terminate when in waiting-for-lock state (Hiroshi)
 Allow negation of a negative number in all cases
 Add ecpg descriptors (Christof, Michael)
 Allow CREATE VIEW v AS SELECT fl::char(8) FROM tbl
 Allow casts with length, like foo::char(8)
 New libpq functions PQsetClientEncoding(), PQclientEncoding() (Tatsuo)
 Add support for SJIS user defined characters (Tatsuo)
 Larger views/rules supported
 Make libpq's PQconnndefaults() thread-safe (Tom)
 Disable // as comment to be ANSI conforming, should use -- (Tom)
 Allow column aliases on views CREATE VIEW name (collist)
 Fixes for views with subqueries (Tom)
 Allow UPDATE table SET fld = (SELECT ...) (Tom)
 SET command options no longer require quotes
 Update pgaccess to 0.98.6
 New SET SEED command
 New pg options.sample file
 New SET FSYNC command (Massimo)
 Allow pg descriptions when creating tables
 Allow pg descriptions when creating types, columns, and functions
 Allow psql %copy to allow delimiters (Peter E)
 Allow psql to print nulls as distinct from "" [null] (Peter E)

Types

 Many array fixes (Tom)
 Allow bare column names to be subscripted as arrays (Tom)
 Improve type casting of int and float constants (Tom)
 Cleanups for int8 inputs, range checking, and type conversion (Tom)
 Fix for SELECT timespan('21:11:26'::time) (Tom)
 netmask('x.x.x.x/0') is 255.255.255.255 instead of 0.0.0.0 (Oleg Sharoyko)
 Add btree index on NUMERIC (Jan)
 Perl fix for large objects containing NUL characters (Douglas Thomson)
 ODBC fix for large objects (free)
 Fix indexing of cidr data type
 Fix for Ethernet MAC addresses (macaddr type) comparisons
 Fix for date/time types when overflows happened in computations (Tom)
 Allow array on int8 (Peter E)
 Fix for rounding/overflow of NUMERIC type, like NUMERIC(4,4) (Tom)
 Allow NUMERIC arrays
 Fix bugs in NUMERIC ceil() and floor() functions (Tom)
 Make char length()/octet_length including trailing blanks (Tom)
 Made abstime/reftime use int4 instead of time_t (Peter E)
 New lztext data type for compressed text fields

Revise code to handle coercion of int and float constants (Tom)
Start at new code to implement a BIT and BIT VARYING type (Adriaan Joubert)
NUMERIC now accepts scientific notation (Tom)
NUMERIC to int4 rounds (Tom)
Convert float4/8 to NUMERIC properly (Tom)
Allow type conversion with NUMERIC (Thomas)
Make ISO date style (2000-02-16 09:33) the default (Thomas)
Add NATIONAL CHAR [VARYING] (Thomas)
Allow NUMERIC round and trunc to accept negative scales (Tom)
New TIME WITH TIME ZONE type (Thomas)
Add MAX()/MIN() on time type (Thomas)
Add abs(), mod(), fac() for int8 (Thomas)
Rename functions to round(), sqrt(), cbrt(), pow() for float8 (Thomas)
Add transcendental math functions (e.g. sin(), acos()) for float8 (Thomas)
Add exp() and ln() for NUMERIC type
Rename NUMERIC power() to pow() (Thomas)
Improved TRANSLATE() function (Edwin Ramirez, Tom)
Allow X=-Y operators (Tom)
Allow SELECT float8(COUNT(*))/(SELECT COUNT(*) FROM t) FROM t GROUP BY f1; (Tom)
Allow LOCALE to use indexes in regular expression searches (Tom)
Allow creation of functional indexes to use default types

Performance

Prevent exponential space consumption with many AND's and OR's (Tom)
Collect attribute selectivity values for system columns (Tom)
Reduce memory usage of aggregates (Tom)
Fix for LIKE optimization to use indexes with multibyte encodings (Tom)
Fix r-tree index optimizer selectivity (Thomas)
Improve optimizer selectivity computations and functions (Tom)
Optimize btree searching for cases where many equal keys exist (Tom)
Enable fast LIKE index processing only if index present (Tom)
Re-use free space on index pages with duplicates (Tom)
Improve hash join processing (Tom)
Prevent descending sort if result is already sorted (Hiroshi)
Allow commuting of index scan query qualifications (Tom)
Prefer index scans in cases where ORDER BY/GROUP BY is required (Tom)
Allocate large memory requests in fix-sized chunks for performance (Tom)
Fix vacuum's performance by reducing memory allocation requests (Tom)
Implement constant-expression simplification (Bernard Frankpitt, Tom)
Use secondary columns to be used to determine start of index scan (Hiroshi)
Prevent quadruple use of disk space when doing internal sorting (Tom)
Faster sorting by calling fewer functions (Tom)
Create system indexes to match all system caches (Bruce, Hiroshi)
Make system caches use system indexes (Bruce)
Make all system indexes unique (Bruce)
Improve pg statistics management for VACUUM speed improvement (Tom)
Flush backend cache less frequently (Tom, Hiroshi)
COPY now reuses previous memory allocation, improving performance (Tom)
Improve optimization cost estimation (Tom)
Improve optimizer estimate of range queries $x > \text{lowbound}$ AND $x < \text{highbound}$ (Tom)
Use DNF instead of CNF where appropriate (Tom, Taral)
Further cleanup for OR-of-AND WHERE-clauses (Tom)
Make use of index in OR clauses ($x = 1$ AND $y = 2$) OR ($x = 2$ AND $y = 4$) (Tom)
Smarter optimizer computations for random index page access (Tom)
New SET variable to control optimizer costs (Tom)
Optimizer queries based on LIMIT, OFFSET, and EXISTS qualifications (Tom)
Reduce optimizer internal housekeeping of join paths for speedup (Tom)
Major subquery speedup (Tom)
Fewer fsync writes when fsync is not disabled (Tom)
Improved LIKE optimizer estimates (Tom)
Prevent fsync in SELECT-only queries (Vadim)
Make index creation use psort code, because it is now faster (Tom)
Allow creation of sort temp tables > 1 Gig

Source Tree Changes

Fix for linux PPC compile
New generic expression-tree-walker subroutine (Tom)
Change form() to varargform() to prevent portability problems
Improved range checking for large integers on Alphas
Clean up #include in /include directory (Bruce)
Add scripts for checking includes (Bruce)
Remove un-needed #include's from *.c files (Bruce)
Change #include's to use <> and "" as appropriate (Bruce)
Enable Windows compilation of libpq
Alpha spinlock fix from Uncle George <gatgul@voicenet.com>
Overhaul of optimizer data structures (Tom)
Fix to cygipc library (Yutaka Tanida)
Allow pgsqll to work on newer Cygwin snapshots (Dan)
New catalog version number (Tom)
Add Linux ARM
Rename heap replace to heap update
Update for QNX (Dr. Andreas Kardos)
New platform-specific regression handling (Tom)
Rename oid8 -> oidvector and int28 -> int2vector (Bruce)
Included all yacc and lex files into the distribution (Peter E.)
Remove lextest, no longer needed (Peter E.)
Fix for libpq and psql on Windows (Magnus)
Internally change datetime and timespan into timestamp and interval (Thomas)
Fix for plpgsql on BSD/OS
Add SQL ASCII test case to the regression test (Tatsuo)
configure --with-mb now deprecated (Tatsuo)
NT fixes
NetBSD fixes (Johnny C. Lam <lamj@stat.cmu.edu>)
Fixes for Alpha compiles
New multibyte encodings

E.217. リリース 6.5.3

リリース日: 1999-10-13

このリリースは基本的には 6.5.2 を整理したものです。 6.5.2 でなくなったPgAccess を新しく追加し、また、NT に特化した改修を組み込みました。

E.217.1. バージョン 6.5.3 への移行方法

6.5.* からの移行の場合は ダンプ/リストアは必要ありません。

E.217.2. 変更点

```
Updated version of pgaccess 0.98
NT-specific patch
Fix dumping rules on inherited tables
```

E.218. リリース 6.5.2

リリース日: 1999-09-15

このリリースは基本的には 6.5.1 を整理したものです。 6.5.1 のユーザから報告があった不具合を改修しています。

E.218.1. バージョン 6.5.2 への移行方法

6.5.* からの移行の場合は ダンプ/リストアは必要ありません。

E.218.2. 変更点

```
subselect+CASE fixes(Tom)
Add SHLIB_LINK setting for solaris_i386 and solaris_sparc ports(Daren Sefcik)
Fixes for CASE in WHERE join clauses(Tom)
Fix BTScan abort(Tom)
Repair the check for redundant UNIQUE and PRIMARY KEY indexes(Thomas)
Improve it so that it checks for multicolumn constraints(Thomas)
Fix for Windows making problem with MB enabled(Hiroki Kataoka)
Allow BSD yacc and bison to compile pl code(Bruce)
Fix SET NAMES working
int8 fixes(Thomas)
Fix vacuum's memory consumption(Hiroshi,Tatsuo)
Reduce the total memory consumption of vacuum(Tom)
Fix for timestamp(datetime)
Rule deparsing bugfixes(Tom)
Fix quoting problems in mkMakefile.tcldefs.sh.in and mkMakefile.tkdefs.sh.in(Tom)
This is to re-use space on index pages freed by vacuum(Vadim)
document -x for pg_dump(Bruce)
Fix for unary operators in rule deparser(Tom)
Comment out FileUnlink of excess segments during mdtruncate() (Tom)
IRIX linking fix from Yu Cao >yucaofalcon.kla-tencor.com<
Repair logic error in LIKE: should not return LIKE_ABORT
    when reach end of pattern before end of text(Tom)
Repair incorrect cleanup of heap memory allocation during transaction abort(Tom)
Updated version of pgaccess 0.98
```

E.219. リリース 6.5.1

リリース日: 1999-07-15

このリリースは基本的には 6.5 を整理したものです。6.5 のユーザから報告があった不具合を改修しています。

E.219.1. バージョン 6.5.1 への移行方法

6.5 からの移行の場合は ダンプ/リストアは必要ありません。

E.219.2. 変更点

```
Add NT README file
Portability fixes for linux_ppc, IRIX, linux_alpha, OpenBSD, alpha
Remove QUERY_LIMIT, use SELECT...LIMIT
Fix for EXPLAIN on inheritance(Tom)
Patch to allow vacuum on multisegment tables(Hiroshi)
R-Tree optimizer selectivity fix(Tom)
ACL file descriptor leak fix(Atsushi Ogawa)
New expression subtree code(Tom)
Avoid disk writes for read-only transactions(Vadim)
Fix for removal of temp tables if last transaction was aborted(Bruce)
Fix to prevent too large row from being created(Bruce)
plpgsql fixes
Allow port numbers 32k - 64k(Bruce)
Add ^ precedence(Bruce)
Rename sort files called pg_temp to pg_sorttemp(Bruce)
Fix for microseconds in time values(Tom)
Tutorial source cleanup
New linux_m68k port
Fix for sorting of NULL's in some cases(Tom)
Shared library dependencies fixed (Tom)
Fixed glitches affecting GROUP BY in subselects(Tom)
Fix some compiler warnings (Tomoaki Nishiyama)
Add Win1250 (Czech) support (Pavel Behal)
```


E.220. リリース 6.5

リリース日: 1999-06-09

このリリースは、開発チームが、バークレイから継承したソースコードに熟達した段階であることを示します。大きな機能を簡単に追加できるようになったことが判るでしょう。全世界にわたる開発チームの規模と経験が増えたおかげです。

以下に、より注目すべき変更点を簡単に説明します。

多版型同時実行制御(MVCC)

これは、古いテーブルレベルロックを取り去り、ほとんどの商用データベースシステムよりも優れたロックシステムで置き換えます。伝統的なシステムでは、変更された各行はコミットされるまでは、他のユーザから読み込まれないように、ロックされます。MVCCはPostgreSQL固有の多版性を使用して、書き込み活動中のデータに対する読み込み側からの一貫性のある読み込みを可能にします。書き込み側は、小さな `pg_log` トランザクションシステムを継続して使用することができます。これによって、伝統的なデータベースシステムのような、各行に対するロックの割り当てを行うことなく、全てを行うことができます。ですから、基本的には、単純なテーブルレベルのロックによって制限することはなくなります。行レベルのロックよりも多少優れたものを持つことになりました。

pg_dump によるホットバックアップ

pg_dump は新しい MVCC 機能を使用して、問い合わせを実行できるオンライン状態のデータベースに対する、完全なデータベースのダンプ/バックアップができるようになりました。

数値データ型

ユーザ指定の精度を持つ、本当の数値データ型を持つようになりました。

一時テーブル

一時テーブルはデータベースセッションにおいて一意な名前を持つこと、および、セッション終了時に削除されることが保証されます。

新しい SQL 機能

CASE、INTERSECT、EXCEPT 文をサポートするようになりました。新しく LIMIT/OFFSET、SET TRANSACTION ISOLATION LEVEL、SELECT ... FOR UPDATE を用意しました。また、LOCK TABLE コマンドを改良しました。

高速化

チーム内の才能による、PostgreSQLの高速化を継続して行いました。メモリ割り当て、最適化、テーブルの結合、行転送ルーチンを高速化しました。

移植

移植リストの拡張を継続して行いました。今回は Windows NT/ix86 と NetBSD/arm32 を含めました。

インタフェース

ほとんどのインタフェースは新しいバージョンになり、既存の機能は改良されました。

文書

文書全体に対して、新規のもの、更新されたものがあります。SGI と AIX プラットフォーム用のFAQ が新しく配布されました。Tutorial は Stefan Simkovics による SQL の入門情報です。ユーザガイド には、postmaster からより多くのユーティリティプログラムまでを範囲とするリファレンスページがあります。またその付録には日付/時刻の動作の詳細があります。管理者用ガイド には、Tom Lane によるトラブル対応に関する新しい章が追加されました。プログラマガイド には、これもまた Stefan による、問い合わせ処理の説明があります。また、匿名 CVS と CVSup 経由の PostgreSQL ソースツリーの入手についても説明があります。

E.220.1. バージョン 6.5 への移行

以前のリリースのPostgreSQL からデータを移行する場合、pg_dump によるダンプ/リストアが必要です。pg_upgrade を使用してこのリリースにアップグレードすることは できません。

テーブルのディスク上の構造がこれまでのリリースから変更されたからです。多版型同時実行制御(MVCC)機能は、マルチユーザ環境下においてどこか異なる動作をもたらします。以下の節を熟読して、既存のアプリケーションが必要な動作を行うことを確認して下さい。

E.220.1.1. 多版型同時実行制御

6.5 における読み込み操作はデータをロックしませんので、トランザクション隔離レベルに関係なく、あるトランザクションで読み込まれたデータは、他の操作によって変更されることがあります。言い替えると、`SELECT` で読み込まれた行は、返された(つまり、文またはトランザクションの始まりから少し後の)時点に存在した行を意味し、現在のトランザクションがコミット、またはロールバックするまでの間、同時に実行されるトランザクションによる更新や削除から保護される行であることは意味しません。

行が実際に存在することを確実にする、または、同時に行われる更新から保護するためには、`SELECT FOR UPDATE` または、適切な `LOCK TABLE` 文を使用しなければなりません。これは、PostgreSQL の以前のリリース、もしくは、他の環境からアプリケーションを移植する場合には考慮しなければなりません。

`contrib/refint.*` トリガを参照整合性のために使用している場合は、上のことに注意して下さい。今のところ更に技術を追加しなければなりません。1 つの方法は、トランザクションがプライマリキーを更新、削除する時に `LOCK parent_table IN SHARE ROW EXCLUSIVE MODE` コマンドを使用し、トランザクションが外部キーを更新、挿入する時に、`LOCK parent_table IN SHARE MODE` コマンドを使用することです。

注意: `SERIALIZABLE` モードでトランザクションを実行している場合、上の `LOCK` コマンドを、そのトランザクションで何らかの DML 文(`SELECT/INSERT/DELETE/UPDATE/FETCH/COPY_TO`)を実行する前に実行しなければなりません。

これらの不便な点は、ダーティー(コミットされていない)データを(隔離レベルに関係なく)読み込む機能と本当の参照整合性が実装された時になくなります。

E.220.2. 変更点

```
Bug Fixes
-----
Fix text<->float8 and text<->float4 conversion functions(Thomas)
Fix for creating tables with mixed-case constraints(Billy)
Change exp()/pow() behavior to generate error on underflow/overflow(Jan)
Fix bug in pg_dump -z
Memory overrun cleanups(Tatsuo)
Fix for lo_import crash(Tatsuo)
Adjust handling of data type names to suppress double quotes(Thomas)
Use type coercion for matching columns and DEFAULT(Thomas)
Fix deadlock so it only checks once after one second of sleep(Bruce)
Fixes for aggregates and PL/pgsql(Hiroshi)
Fix for subquery crash(Vadim)
Fix for libpq function PQfnnumber and case-insensitive names(Bahman Rafatjoo)
Fix for large object write-in-middle, no extra block, memory consumption(Tatsuo)
Fix for pg_dump -d or -D and quote special characters in INSERT
Repair serious problems with dynahash(Tom)
Fix INET/CIDR portability problems
Fix problem with selectivity error in ALTER TABLE ADD COLUMN(Bruce)
Fix executor so mergejoin of different column types works(Tom)
Fix for Alpha OR selectivity bug
Fix OR index selectivity problem(Bruce)
Fix so %d shows proper length for char()/varchar() (Ryan)
Fix tutorial code(Clark)
Improve destroyuser checking(Oliver)
Fix for Kerberos(Rodney McDuff)
Fix for dropping database while dirty buffers(Bruce)
Fix so sequence nextval() can be case-sensitive(Bruce)
Fix != operator
Drop buffers before destroying database files(Bruce)
Fix case where executor evaluates functions twice(Tatsuo)
Allow sequence nextval actions to be case-sensitive(Bruce)
Fix optimizer indexing not working for negative numbers(Bruce)
Fix for memory leak in executor with fjIsNull
Fix for aggregate memory leaks(Erik Riedel)
Allow user name containing a dash to grant privileges
Cleanup of NULL in inet types
Clean up system table bugs(Tom)
Fix problems of PAGER and %? command(Masaaki Sakaida)
Reduce default multisegment file size limit to 1GB(Peter)
Fix for dumping of CREATE OPERATOR(Tom)
Fix for backward scanning of cursors(Hiroshi Inoue)
Fix for COPY FROM STDIN when using %i(Tom)
Fix for subselect is compared inside an expression(Jan)
Fix handling of error reporting while returning rows(Tom)
Fix problems with reference to array types(Tom, Jan)
```

Prevent UPDATE SET oid(Jan)
 Fix pg_dump so -t option can handle case-sensitive tablenames
 Fixes for GROUP BY in special cases(Tom, Jan)
 Fix for memory leak in failed queries(Tom)
 DEFAULT now supports mixed-case identifiers(Tom)
 Fix for multisegment uses of DROP/RENAME table, indexes(Ole Gjerde)
 Disable use of pg_dump with both -o and -d options(Bruce)
 Allow pg_dump to properly dump group privileges(Bruce)
 Fix GROUP BY in INSERT INTO table SELECT * FROM table2(Jan)
 Fix for computations in views(Jan)
 Fix for aggregates on array indexes(Tom)
 Fix for DEFAULT handles single quotes in value requiring too many quotes
 Fix security problem with non-super users importing/exporting large objects(Tom)
 Rollback of transaction that creates table cleaned up properly(Tom)
 Fix to allow long table and column names to generate proper serial names(Tom)

Enhancements

 Add "vacuumdb" utility
 Speed up libpq by allocating memory better(Tom)
 EXPLAIN all indexes used(Tom)
 Implement CASE, COALESCE, NULLIF expression(Thomas)
 New pg_dump table output format(Constantin)
 Add string min()/max() functions(Thomas)
 Extend new type coercion techniques to aggregates(Thomas)
 New moddatetime contrib(Terry)
 Update to pgaccess 0.96(Constantin)
 Add routines for single-byte "char" type(Thomas)
 Improved substr() function(Thomas)
 Improved multibyte handling(Tatsuo)
 Multiversion concurrency control/MVCC(Vadim)
 New Serialized mode(Vadim)
 Fix for tables over 2gigs(Peter)
 New SET TRANSACTION ISOLATION LEVEL(Vadim)
 New LOCK TABLE IN ... MODE(Vadim)
 Update ODBC driver(Byron)
 New NUMERIC data type(Jan)
 New SELECT FOR UPDATE(Vadim)
 Handle "NaN" and "Infinity" for input values(Jan)
 Improved date/year handling(Thomas)
 Improved handling of backend connections(Magnus)
 New options ELOG_TIMESTAMPS and USE_SYSLOG options for log files(Massimo)
 New TCL ARRAYS option(Massimo)
 New INTERSECT and EXCEPT(Stefan)
 New pg_index.indisprimary for primary key tracking(D'Arcy)
 New pg_dump option to allow dropping of tables before creation(Brook)
 Speedup of row output routines(Tom)
 New READ COMMITTED isolation level(Vadim)
 New TEMP tables/indexes(Bruce)
 Prevent sorting if result is already sorted(Jan)
 New memory allocation optimization(Jan)
 Allow psql to do %p%g(Bruce)
 Allow multiple rule actions(Jan)
 Added LIMIT/OFFSET functionality(Jan)
 Improve optimizer when joining a large number of tables(Bruce)
 New intro to SQL from S. Simkovics' Master's Thesis (Stefan, Thomas)
 New intro to backend processing from S. Simkovics' Master's Thesis (Stefan)
 Improved int8 support(Ryan Bradetich, Thomas, Tom)
 New routines to convert between int8 and text/varchar types(Thomas)
 New bushy plans, where meta-tables are joined(Bruce)
 Enable right-hand queries by default(Bruce)
 Allow reliable maximum number of backends to be set at configure time
 (--with-maxbackends and postmaster switch (-N backends))(Tom)
 GEQO default now 10 tables because of optimizer speedups(Tom)
 Allow NULL=Var for MS-SQL portability(Michael, Bruce)
 Modify contrib check_primary_key() so either "automatic" or "dependent"(Anand)
 Allow psql %d on a view show query(Ryan)
 Speedup for LIKE(Bruce)
 Ecpq fixes/features, see src/interfaces/ecpq/ChangeLog file(Michael)
 JDBC fixes/features, see src/interfaces/jdbc/CHANGELOG(Peter)
 Make % operator have precedence like /(Bruce)
 Add new postgres -O option to allow system table structure changes(Bruce)
 Update contrib/pginterface/findoidjoins script(Tom)
 Major speedup in vacuum of deleted rows with indexes(Vadim)
 Allow non-SQL functions to run different versions based on arguments(Tom)
 Add -E option that shows actual queries sent by %dt and friends(Masaaki Sakaida)
 Add version number in start-up banners for psql(Masaaki Sakaida)
 New contrib/vacuumlo removes large objects not referenced(Peter)
 New initialization for table sizes so non-vacuumed tables perform better(Tom)
 Improve error messages when a connection is rejected(Tom)
 Support for arrays of char() and varchar() fields(Massimo)
 Overhaul of hash code to increase reliability and performance(Tom)
 Update to PyGreSQL 2.4(D'Arcy)
 Changed debug options so -d4 and -d5 produce different node displays(Jan)
 New pg_options: pretty_plan, pretty_parse, pretty_rewritten(Jan)
 Better optimization statistics for system table access(Tom)

Better handling of non-default block sizes(Massimo)
Improve GEQO optimizer memory consumption(Tom)
UNION now supports ORDER BY of columns not in target list(Jan)
Major libpq++ improvements(Vince Vielhaber)
pg_dump now uses -z(ACL's) as default(Bruce)
backend cache, memory speedups(Tom)
have pg_dump do everything in one snapshot transaction(Vadim)
fix for large object memory leakage, fix for pg_dumping(Tom)
INET type now respects netmask for comparisons
Make VACUUM ANALYZE only use a readlock(Vadim)
Allow VIEWS on UNIONS(Jan)
pg_dump now can generate consistent snapshots on active databases(Vadim)

Source Tree Changes

Improve port matching(Tom)
Portability fixes for SunOS
Add Windows NT backend port and enable dynamic loading(Magnus and Daniel Horak)
New port to Cobalt Qube(Mips) running Linux(Tatsuo)
Port to NetBSD/m68k(Mr. Mutsuki Nakajima)
Port to NetBSD/sun3(Mr. Mutsuki Nakajima)
Port to NetBSD/macppc(Toshimi Aoki)
Fix for tcl/tk configuration(Vince)
Removed CURRENT key word for rule queries(Jan)
NT dynamic loading now works(Daniel Horak)
Add ARM32 support(Andrew McMurry)
Better support for HP-UX 11 and UnixWare
Improve file handling to be more uniform, prevent file descriptor leak(Tom)
New install commands for plpgsql(Jan)

[前のページ](#)

リリース 6.5.1

[ホーム](#)
[上に戻る](#)

[次のページ](#)

リリース 6.4.2

E.221. リリース 6.4.2

リリース日: 1998-12-20

リリース 6.4.1 は正しくパッケージ化されていませんでした。また、不具合を1つ改修しています。

E.221.1. バージョン 6.4.2 への移行方法

6.4.* からの移行の場合は ダンプ/リストアは必要ありません。

E.221.2. 変更点

Fix for datetime constant problem on some platforms (Thomas)

E.222. リリース 6.4.1

リリース日: 1998-12-18

このリリースは基本的には 6.4 を整理したものです。6.4 のユーザから報告があった不具合を改修しています。

E.222.1. バージョン 6.4.1 への移行方法

6.4 からの移行の場合は ダンプ/リストアは必要ありません。

E.222.2. 変更点

```
Add pg_dump -N flag to force double quotes around identifiers. This is
the default(Thomas)
Fix for NOT in where clause causing crash(Bruce)
EXPLAIN VERBOSE coredump fix(Vadim)
Fix shared-library problems on Linux
Fix test for table existence to allow mixed-case and whitespace in
the table name(Thomas)
Fix a couple of pg_dump bugs
Configure matches template/.similar entries better(Tom)
Change builtin function names from SPI_* to spi_*
OR WHERE clause fix(Vadim)
Fixes for mixed-case table names(Billy)
contrib/linux/postgres.init.csh/sh fix(Thomas)
libpq memory overrun fix
SunOS fixes(Tom)
Change exp() behavior to generate error on underflow(Thomas)
pg_dump fixes for memory leak, inheritance constraints, layout change
update pgaccess to 0.93
Fix prototype for 64-bit platforms
Multibyte fixes(Tatsuo)
New ecpg man page
Fix memory overruns(Tatsuo)
Fix for lo_import() crash(Bruce)
Better search for install program(Tom)
Timezone fixes(Tom)
HP-UX fixes(Tom)
Use implicit type coercion for matching DEFAULT values(Thomas)
Add routines to help with single-byte (internal) character type(Thomas)
Compilation of libpq for Windows fixes(Magnus)
Upgrade to PyGreSQL 2.2(D'Arcy)
```

E.223. リリース 6.4

リリース日: 1998-10-30

このリリースでは、多くの新機能や改良があります。前回のリリース以来、システムのほとんど全てに対して要求がありました。以下に、簡単に概要を示します。

- Jan Wieck による、書換えルールシステムの高度な新しいコードのおかげで、ビューとルールが実用的なものになりました。また、彼は、この機能に関する章をプログラマガイドに書きました。
- Jan はまた、前回のリリースで彼により寄贈された、元々の PL/pgTCL と調和する、2 番目の手続き言語、PL/pgSQL を寄贈しました。
- 石井達夫による、省略可能なマルチバイト文字セットをサポートするようになりました。これは既存のロケールサポートを補足するものです。
- Tom Lane による、非同期メッセージとその割り込みサポートの改良のおかげで、クライアント/サーバ間の通信が整理されました。
- パーサが、利用できる演算子と関数に引数を一致させる、または、対象とする列に列や式を一致させる、自動型強制を行うようになりました。これは、PostgreSQL の型拡張機能をサポートする、汎用機構を使用しています。この機能を説明する新しい章が ユーザガイド にあります。
- 新しく 3 つの型が追加されました。inet と cidr という 2 つの型は、IP ネットワーク、サブネット、マシンアドレスの各種形式をサポートします。いくつかのプラットフォームで利用できる、8 バイト整数型があります。詳しくは、ユーザガイド のデータ型に関する章を参照して下さい。4 番目の型、serial が、int4 型、シーケンス、一意性インデックスの混合物として、パーサによってサポートされるようになりました。
- INSERT DEFAULT VALUES など、より多くの SQL92 互換の構文機能も追加されました。
- 自動設定・インストールシステムにいくつかの変更要求がありました。従来よりもより多くのプラットフォームでより堅牢になったはずです。

E.223.1. バージョン 6.4 への移行

以前のリリースの PostgreSQL からデータを移行する場合、pg_dump もしくは pg_dumpall によるダンプ/リストアが必要です。

E.223.2. 変更点

Bug Fixes

```
-----
Fix for a tiny memory leak in PQsetdb/PQfinish(Bryan)
Remove char2-16 data types, use char/varchar(Darren)
Pqfn not handles a NOTICE message(Anders)
Reduced busywaiting overhead for spinlocks with many backends (dg)
Stuck spinlock detection (dg)
Fix up "ISO-style" timespan decoding and encoding(Thomas)
Fix problem with table drop after rollback of transaction(Vadim)
Change error message and remove non-functional update message(Vadim)
Fix for COPY array checking
Fix for SELECT 1 UNION SELECT NULL
Fix for buffer leaks in large object calls(Pascal)
Change owner from oid to int4 type(Bruce)
Fix a bug in the oracle compatibility functions btrim() ltrim() and rtrim()
Fix for shared invalidation cache overflow(Massimo)
Prevent file descriptor leaks in failed COPY's(Bruce)
Fix memory leak in libpgtcl's pg_select(Constantin)
Fix problems with username/passwords over 8 characters(Tom)
Fix problems with handling of asynchronous NOTIFY in backend(Tom)
Fix of many bad system table entries(Tom)
```

Enhancements

```
-----
Upgrade ecpg and ecpglib, see src/interfaces/ecpg/ChangeLog(Michael)
Show the index used in an EXPLAIN(Zeugswetter)
EXPLAIN invokes rule system and shows plan(s) for rewritten queries(Jan)
Multibyte awareness of many data types and functions, via configure(Tatsuo)
New configure --with-mb option(Tatsuo)
New initdb --pgencoding option(Tatsuo)
New createdb -E multibyte option(Tatsuo)
Select version(); now returns PostgreSQL version(Jeroen)
libpq now allows asynchronous clients(Tom)
Allow cancel from client of backend query(Tom)
```


psql now cancels query with Control-C(Tom)
libpq users need not issue dummy queries to get NOTIFY messages(Tom)
NOTIFY now sends sender's PID, so you can tell whether it was your own(Tom)
PGresult struct now includes associated error message, if any(Tom)
Define "tz hour" and "tz minute" arguments to date_part()(Thomas)
Add routines to convert between varchar and bpchar(Thomas)
Add routines to allow sizing of varchar and bpchar into target columns(Thomas)
Add bit flags to support timezonehour and minute in data retrieval(Thomas)
Allow more variations on valid floating point numbers (e.g. ".1", "1e6")(Thomas)
Fixes for unary minus parsing with leading spaces(Thomas)
Implement TIMEZONE HOUR, TIMEZONE MINUTE per SQL92 specs(Thomas)
Check for and properly ignore FOREIGN KEY column constraints(Thomas)
Define USER as synonym for CURRENT USER per SQL92 specs(Thomas)
Enable HAVING clause but no fixes elsewhere yet.
Make "char" type a synonym for "char(1)" (actually implemented as bpchar)(Thomas)
Save string type if specified for DEFAULT clause handling(Thomas)
Coerce operations involving different data types(Thomas)
Allow some index use for columns of different types(Thomas)
Add capabilities for automatic type conversion(Thomas)
Cleanups for large objects, so file is truncated on open(Peter)
Readline cleanups(Tom)
Allow psql %f % to make spaces as delimiter(Bruce)
Pass pg attribute.atttypmod to the frontend for column field lengths(Tom,Bruce)
Mysql compatibility library in /contrib(Aldrin)
Remove the requirement that ORDER/GROUP BY clause identifiers be included in the target list(David)
Convert columns to match columns in UNION clauses(Thomas)
Remove fork()/exec() and only do fork()(Bruce)
Jdbc cleanups(Peter)
Show backend status on ps command line(only works on some platforms)(Bruce)
Pg hba.conf now has a sameuser option in the database field
Make lo unlink take oid param, not int4
New DISABLE_COMPLEX_MACRO for compilers that cannot handle our macros(Bruce)
libpqtc1 now handles NOTIFY as a Tcl event, need not send dummy queries(Tom)
libpqtc1 cleanups(Tom)
Add -error option to libpqtc1's pg_result command(Tom)
New locale patch, see docs/README/locale(Oleg)
Fix for pg_dump so CONSTRAINT and CHECK syntax is correct(ccb)
New contrib/lo code for large object orphan removal(Peter)
New psql command "SET CLIENT_ENCODING TO 'encoding'" for multibytes feature, see /doc/README.mb(Tatsuo)
contrib/noupdate code to revoke update permission on a column
libpq can now be compiled on Windows(Magnus)
Add PQsetdbLogin() in libpq
New 8-byte integer type, checked by configure for OS support(Thomas)
Better support for quoted table/column names(Thomas)
Surround table and column names with double-quotes in pg_dump(Thomas)
PQreset() now works with passwords(Tom)
Handle case of GROUP BY target list column number out of range(David)
Allow UNION in subselects
Add auto-size to screen to %d? commands(Bruce)
Use UNION to show all %d? results in one query(Bruce)
Add %d? field search feature(Bruce)
Pg_dump issues fewer %connect requests(Tom)
Make pg_dump -z flag work better, document it in manual page(Tom)
Add HAVING clause with full support for subselects and unions(Stephan)
Full text indexing routines in contrib/fulltextindex(Maarten)
Transaction ids now stored in shared memory(Vadim)
New PGCLIENTENCODING when issuing COPY command(Tatsuo)
Support for SQL92 syntax "SET NAMES"(Tatsuo)
Support for LATIN2-5(Tatsuo)
Add UNICODE regression test case(Tatsuo)
Lock manager cleanup, new locking modes for LLL(Vadim)
Allow index use with OR clauses(Bruce)
Allows "SELECT NULL ORDER BY 1;"
Explain VERBOSE prints the plan, and now pretty-prints the plan to the postmaster log file(Bruce)
Add indexes display to %d command(Bruce)
Allow GROUP BY on functions(David)
New pg_class.relkind for large objects(Bruce)
New way to send libpq NOTICE messages to a different location(Tom)
New %w write command to psql(Bruce)
New /contrib/findoidjoins scans oid columns to find join relationships(Bruce)
Allow binary-compatible indexes to be considered when checking for valid
Indexes for restriction clauses containing a constant(Thomas)
New ISBN/ISSN code in /contrib/isbn issn
Allow NOT LIKE, IN, NOT IN, BETWEEN, and NOT BETWEEN constraint(Thomas)
New rewrite system fixes many problems with rules and views(Jan)
* Rules on relations work
* Event qualifications on insert/update/delete work
* New OLD variable to reference CURRENT, CURRENT will be remove in future
* Update rules can reference NEW and OLD in rule qualifications/actions
* Insert/update/delete rules on views work
* Multiple rule actions are now supported, surrounded by parentheses
* Regular users can create views/rules on tables they have RULE permits
* Rules and views inherit the privileges of the creator
* No rules at the column level
* No UPDATE NEW/OLD rules
* New pg_tables, pg_indexes, pg_rules and pg_views system views
* Only a single action on SELECT rules
* Total rewrite overhaul, perhaps for 6.5
* handle subselects
* handle aggregates on views
* handle insert into select from view works
System indexes are now multikey(Bruce)
Oidint2, oidint4, and oidname types are removed(Bruce)
Use system cache for more system table lookups(Bruce)
New backend programming language PL/pgSQL in backend/pl(Jan)
New SERIAL data type, auto-creates sequence/index(Thomas)
Enable assert checking without a recompile(Massimo)
User lock enhancements(Massimo)
New setval() command to set sequence value(Massimo)
Auto-remove unix socket file on start-up if no postmaster running(Massimo)

Conditional trace package(Massimo)
New UNLISTEN command(Massimo)
psql and libpq now compile under Windows using win32.mak(Magnus)
Lo read no longer stores trailing NULL(Bruce)
Identifiers are now truncated to 31 characters internally(Bruce)
Createuser options now available on the command line
Code for 64-bit integer supported added, configure tested, int8 type(Thomas)
Prevent file descriptor leak from failed COPY(Bruce)
New pg_upgrade command(Bruce)
Updated /contrib directories(Massimo)
New CREATE TABLE DEFAULT VALUES statement available(Thomas)
New INSERT INTO TABLE DEFAULT VALUES statement available(Thomas)
New DECLARE and FETCH feature(Thomas)
libpq's internal structures now not exported(Tom)
Allow up to 8 key indexes(Bruce)
Remove ARCHIVE key word, that is no longer used(Thomas)
pg_dump -n flag to suppress quotes around identifiers
disable system columns for views(Jan)
new INET and CIDR types for network addresses(TomH, Paul)
no more double quotes in psql output
pg_dump now dumps views(Terry)
new SET QUERY_LIMIT(Tatsuo,Jan)

Source Tree Changes

/contrib cleanup(Jun)
Inline some small functions called for every row(Bruce)
Alpha/linux fixes
HP-UX cleanups(Tom)
Multibyte regression tests(Soonmyung.)
Remove --disabled options from configure
Define PGDOC to use POSTGRES DIR by default
Make regression optional
Remove extra braces code to pgindent(Bruce)
Add bsd shared library support(Bruce)
New --without-CXX support configure option(Brook)
New FAQ CVS
Update Backend flowchart in tools/backend(Bruce)
Change atttypmod from int16 to int32(Bruce, Tom)
Getusage() fix for platforms that do not have it(Tom)
Add PQconnectdb, PGUSER, PGPASSWORD to libpq man page
NS32K platform fixes(Phil Nelson, John Buller)
SCO 7/UnixWare 2.x fixes(Billy,others)
Sparc/Solaris 2.5 fixes(Ryan)
Pgbuiltin.3 is obsolete, move to doc files(Thomas)
Even more documentation(Thomas)
Nextstep support(Jacek)
Aix support(David)
pginterface manual page(Bruce)
shared libraries all have version numbers
merged all OS-specific shared library defines into one file
smarter TCL/TK configuration checking(Billy)
smarter perl configuration(Brook)
configure uses supplied install-sh if no install script found(Tom)
new Makefile.shlib for shared library configuration(Tom)

[前のページ](#)

リリース 6.4.1

[ホーム](#)
[上に戻る](#)

[次のページ](#)

リリース 6.3.2

E.224. リリース 6.3.2

リリース日: 1998-04-07

このリリースは 6.3.x の不具合を改修したものです。新機能の完全な要約については バージョン 6.3 のリリースノートを参照して下さい。

要約:

- Linux を含むいくつかのプラットフォームの自動設定サポートにおいて、バージョン 6.3.1 で間違って導入された障害を修復しました。
- BETWEEN と LIKE 句の左側での関数呼び出しの扱いを修正しました。

6.3 や 6.3.1 からはダンプ/リストアは必要ありません。 `make distclean`、`make`、`make install` だけが必要です。この最後の段階は、`postmaster` が稼働していない時に行わなければなりません。PostgreSQL ライブラリを使用して、独自のアプリケーションを再リンクしなければなりません。

6.3 以前のインストレーションからのアップグレードの場合は、バージョン 6.3 用のインストール・移行手順を参照して下さい。

E.224.1. 変更点

Configure detection improvements for tcl/tk(Brook Milligan, Alvin)
Manual page improvements(Bruce)
BETWEEN and LIKE fix(Thomas)
fix for psql %connect used by pg_dump(Oliver Elphick)
New odbc driver
pgaccess, version 0.86
qsort removed, now uses libc version, cleanups(Jeroen)
fix for buffer over-runs detected(Maurice Gittens)
fix for buffer overrun in libpgtcl(Randy Kunkee)
fix for UNION with DISTINCT or ORDER BY(Bruce)
gettimeofday configure check(Doug Winterburn)
Fix "indexes not used" bug(Vadim)
docs additions(Thomas)
Fix for backend memory leak(Bruce)
libreadline cleanup(Erwan MAS)
Remove DISTDIR(Bruce)
Makefile dependency cleanup(Jeroen van Vianen)
ASSERT fixes(Bruce)

E.225. リリース 6.3.1

リリース日: 1998-03-23

要約:

- マルチバイト文字セットのサポートの追加。
- クライアント、サーバ間でエンディアンが異なる場合のバイトオーダーの修正。
- 使用できる SQL 構文の若干の更新。
- インストール時の設定自動検出の改良。

6.3 からはダンプ/リストアは必要ありません。make distclean、make、make install だけです。この最後の段階は、postmaster が稼働していない時に行わなければなりません。PostgreSQL ライブラリを使用して、独自のアプリケーションを再リンクしなければなりません。

6.3 以前のインストレーションからのアップグレードの場合は、バージョン 6.3 用のインストール・移行手順を参照して下さい。

E.225.1. 変更点

```
ecpg cleanup/fixes, now version 1.1(Michael Meskes)
pg_user cleanup(Bruce)
large object fix for pg_dump and tclsh (alvin)
LIKE fix for multiple adjacent underscores
fix for redefining builtin functions(Thomas)
ultrix4 cleanup
upgrade to pg_access 0.83
updated CLUSTER manual page
multibyte character set support, see doc/README.mb(Tatsuo)
configure --with-pgport fix
pg_ident fix
big-endian fix for backend communications(Kataoka)
SUBSTR() and substring() fix(Jan)
several jdbc fixes(Peter)
libpgtcl improvements, see libpgtcl/README(Randy Kunkee)
Fix for "Datasize = 0" error(Vadim)
Prevent %do from wrapping(Bruce)
Remove duplicate Russian character set entries
Sunos4 cleanup
Allow optional TABLE key word in LOCK and SELECT INTO(Thomas)
CREATE SEQUENCE options to allow a negative integer(Thomas)
Add "PASSWORD" as an allowed column identifier(Thomas)
Add checks for UNION target fields(Bruce)
Fix Alpha port(Dwayne Bailey)
Fix for text arrays containing quotes(Doug Gibson)
Solaris compile fix(Albert Chin-A-Young)
Better identify tcl and tk libs and includes(Bruce)
```

E.226. リリース 6.3

リリース日: 1998-03-01

このリリースでは、多くの 新機能や改良があります。以下に、簡単に概要を示します。

- 完全な SQL92 の副セレクト機能を含む、多くの新しい SQL 機能 (ターゲットリストの副セレクトを除く全て)。
- タイムゾーン、日付書式を指定する、クライアント側の環境変数のサポート。
- クライアント/サーバ接続用のソケットインタフェース。現在、これがデフォルトとなりましたので、postmaster 起動時に `-i` フラグが必要になる可能性があります。
- より優れたパスワード認証機構。デフォルトのテーブル権限が変更されました。
- *time travel* という古い形式が削除されました。性能が向上しました。

注意: Bruce Momjian による、このリリースを紹介する以下のノートがあります。

いくつか言及すべき、6.3 の一般的な問題点があります。一行では書き表せない程度の項目です。詳細な変更点リストの再検査がまだ必要です。

まず、副セレクトが可能になりました。これが可能になった今、副セレクトがない SQL は非常に限定された言語であるということが出来ます。副セレクトは大きな機能であり、副セレクトによってより良い問い合わせが行うことができるかどうか、コードを見直すべきです。想像するよりも多くのことが副セレクトで行うことができることが判るでしょう。Vadim は副セレクトを使用した大きな SQL マップを残し、また、完全な副セレクト関数も残しました。副セレクトを使うことができないものは、目的リストだけです。

次に、6.3 はデフォルトで、TCP/IPではなく Unix ドメインソケットを使用するようになりました。他のマシンからの接続を可能にするには、postmaster の新規の `-i` オプションを使用し、もちろん、`pg_hba.conf` ファイルを編集しなければなりません。また、このため、`pg_hba.conf` の書式が変更されました。

3番目に、`char()` フィールドは、`varchar()` や `text` よりも高速にアクセスできるようになりました。特に、`text` や `varchar()` には、この型の最初の列の後にある列へのアクセスにペナルティがあります。`char()` にもこのアクセスペナルティがありましたが、今回なくなりました。特に、短い文字用の列に `varchar()` や `text` を使用しているような場合、テーブルの再設計を行うことを勧めます。この点や他の変更点により、6.3 は以前のリリースよりも高速になっています。

Unix ファイルに依存しないパスワード定義が可能になりました。新しい SQL コマンドがあります。より詳細は 管理者用ガイドを参照して下さい。ユーザの情報やそのパスワードを格納する、`pg_shadow` という新しいテーブルがあります。また、デフォルトではこのテーブルは `postgres` スーパーユーザからのみ SELECT することができます。`pg_user` は `pg_shadow` のビューとなり、`PUBLIC` より SELECT 可能です。アプリケーションを変更することなく、`pg_user` を使用し続けることができます。

ユーザが作成したテーブルは、デフォルトで、`PUBLIC` から SELECT できなくなりました。これは ANSI 標準の要求に従うために変更されました。もちろんテーブルを作成した後に必要な権限を GRANT することができます。システムテーブルは現在でも `PUBLIC` から SELECT 可能です。

本物のデッドロック検出コードが追加されました。60秒のタイムアウトはもうありません。この新しいロックコードは よりよい FIFO を実装していますので、高負荷時のリソースの消費量が少なくなるはずです。

以前のリリースでは、不適当な文書に関する批判が多くありました。Thomas は頑張ってこのリリースで多くの新しいマニュアルを作成しました。doc/ディレクトリに注目して下さい。

性能上の理由により、*time travel* はなくなりました。トリガを使用して実装することが出来ます(`pgsql/contrib/spi/README`を参照して下さい)。型、演算子などの `%d` コマンドをチェックして下さい。また、ビューは元となったテーブルとは別に、独自の権限を持つようになりました。ですから、これらの権限は別々に設定しなければなりません。いくつかの新しい PostgreSQL との通信方法については `/pgsql/interfaces` をチェックして下さい。

これは既存のユーザへの説明が必要となる初めてのリリースです。多くの点において、この新しいリリースでは多くの制限が削除され、そのために行っていた回避策が不要になったため、この説明が必要でした。

E.226.1. バージョン 6.3 への移行

以前のリリースの PostgreSQL からデータを移行する場合、pg_dump もしくは pg_dumpall によるダンプ/リストアが必要です。

E.226.2. 変更点

Bug Fixes

Fix binary cursors broken by MOVE implementation(Vadim)
Fix for tcl library crash(Jan)
Fix for array handling, from Gerhard Hintermayer
Fix acl error, and remove duplicate pgtrace(Bruce)
Fix psql ¥e for empty file(Bruce)
Fix for textcat on varchar() fields(Bruce)
Fix for DBT Sendproc (Zeugswetter Andres)
Fix vacuum analyze syntax problem(Bruce)
Fix for international identifiers(Tatsuo)
Fix aggregates on inherited tables(Bruce)
Fix substr() for out-of-bounds data
Fix for select 1=1 or 2=2, select 1=1 and 2=2, and select sum(2+2)(Bruce)
Fix notty output to show status result. -q option still turns it off(Bruce)
Fix for count(*), aggs with views and multiple tables and sum(3)(Bruce)
Fix cluster(Bruce)
Fix for PQtrace start/stop several times(Bruce)
Fix a variety of locking problems like newer lock waiters getting
 lock before older waiters, and having readlock people not share
 locks if a writer is waiting for a lock, and waiting writers not
 getting priority over waiting readers(Bruce)
Fix crashes in psql when executing queries from external files(James)
Fix problem with multiple order by columns, with the first one having
 NULL values(Jeroen)
Use correct hash table support functions for float8 and int4(Thomas)
Re-enable JOIN= option in CREATE OPERATOR statement (Thomas)
Change precedence for boolean operators to match expected behavior(Thomas)
Generate elog(ERROR) on over-large integer(Bruce)
Allow multiple-argument functions in constraint clauses(Thomas)
Check boolean input literals for 'true','false','yes','no','1','0'
 and throw elog(ERROR) if unrecognized(Thomas)
Major large objects fix
Fix for GROUP BY showing duplicates(Vadim)
Fix for index scans in MergeJoin(Vadim)

Enhancements

Subselects with EXISTS, IN, ALL, ANY key words (Vadim, Bruce, Thomas)
New User Manual(Thomas, others)
Speedup by inlining some frequently-called functions
Real deadlock detection, no more timeouts(Bruce)
Add SQL92 "constants" CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP,
 CURRENT_USER(Thomas)
Modify constraint syntax to be SQL92-compliant(Thomas)
Implement SQL92 PRIMARY KEY and UNIQUE clauses using indexes(Thomas)
Recognize SQL92 syntax for FOREIGN KEY. Throw elog notice(Thomas)
Allow NOT NULL UNIQUE constraint clause (each allowed separately before)(Thomas)
Allow PostgreSQL-style casting ("::") of non-constants(Thomas)
Add support for SQL3 TRUE and FALSE boolean constants(Thomas)
Support SQL92 syntax for IS TRUE/IS FALSE/IS NOT TRUE/IS NOT FALSE(Thomas)
Allow shorter strings for boolean literals (e.g. "t", "tr", "tru")(Thomas)
Allow SQL92 delimited identifiers(Thomas)
Implement SQL92 binary and hexadecimal string decoding (b'10' and x'1F')(Thomas)
Support SQL92 syntax for type coercion of literal strings
 (e.g. "DATETIME 'now'")(Thomas)
Add conversions for int2, int4, and OID types to and from text(Thomas)
Use shared lock when building indexes(Vadim)
Free memory allocated for an user query inside transaction block after
 this query is done, was turned off in <= 6.2.1(Vadim)
New SQL statement CREATE PROCEDURAL LANGUAGE(Jan)
New PostgreSQL Procedural Language (PL) backend interface(Jan)
Rename pg_dump -H option to -h(Bruce)
Add Java support for passwords, European dates(Peter)
Use indexes for LIKE and ~, !~ operations(Bruce)
Add hash functions for datetime and timespan(Thomas)
Time Travel removed(Vadim, Bruce)
Add paging for ¥d and ¥z, and fix ¥i(Bruce)
Add Unix domain socket support to backend and to frontend library(Goran)
Implement CREATE DATABASE/WITH LOCATION and initlocation utility(Thomas)
Allow more SQL92 and/or PostgreSQL reserved words as column identifiers(Thomas)
Augment support for SQL92 SET TIME ZONE...(Thomas)
SET/SHOW/RESET TIME ZONE uses TZ backend environment variable(Thomas)
Implement SET keyword = DEFAULT and SET TIME ZONE DEFAULT(Thomas)
Enable SET TIME ZONE using TZ environment variable(Thomas)
Add PGDATESTYLE environment variable to frontend and backend initialization(Thomas)
Add PGTZ, PGCOSTHEAP, PGCOSTINDEX, PGRPLANS, PGGEQO
 frontend library initialization environment variables(Thomas)
Regression tests time zone automatically set with "setenv PGTZ PST8PDT"(Thomas)

Add pg_description table for info on tables, columns, operators, types, and aggregates(Bruce)
 Increase 16 char limit on system table/index names to 32 characters(Bruce)
 Rename system indexes(Bruce)
 Add 'GERMAN' option to SET DATESTYLE(Thomas)
 Define an "ISO-style" timespan output format with "hh:mm:ss" fields(Thomas)
 Allow fractional values for delta times (e.g. '2.5 days')(Thomas)
 Validate numeric input more carefully for delta times(Thomas)
 Implement day of year as possible input to date_part()(Thomas)
 Define timespan_finite() and text_timespan() functions(Thomas)
 Remove archive stuff(Bruce)
 Allow for a pg_password authentication database that is separate from the system password file(Todd)
 Dump ACLs, GRANT, REVOKE privileges(Matt)
 Define text, varchar, and bpchar string length functions(Thomas)
 Fix Query handling for inheritance, and cost computations(Bruce)
 Implement CREATE TABLE/AS SELECT (alternative to SELECT/INTO)(Thomas)
 Allow NOT, IS NULL, IS NOT NULL in constraints(Thomas)
 Implement UNIONs for SELECT(Bruce)
 Add UNION, GROUP, DISTINCT to INSERT(Bruce)
 varchar() stores only necessary bytes on disk(Bruce)
 Fix for BLOBs(Peter)
 Mega-Patch for JDBC...see README_6.3 for list of changes(Peter)
 Remove unused "option" from PQconnectdb()
 New LOCK command and lock manual page describing deadlocks(Bruce)
 Add new psql %da, %dd, %df, %do, %dS, and %dT commands(Bruce)
 Enhance psql %z to show sequences(Bruce)
 Show NOT NULL and DEFAULT in psql %d table(Bruce)
 New psql .psqlrc file start-up(Andrew)
 Modify sample start-up script in contrib/linux to show syslog(Thomas)
 New types for IP and MAC addresses in contrib/ip_and_mac(TomH)
 Unix system time conversions with date/time types in contrib/unixdate(Thomas)
 Update of contrib stuff(Massimo)
 Add Unix socket support to DBD::Pg(Goran)
 New python interface (PyGreSQL 2.0)(D'Arcy)
 New frontend/backend protocol has a version number, network byte order(Phil)
 Security features in pg_hba.conf enhanced and documented, many cleanups(Phil)
 CHAR() now faster access than VARCHAR() or TEXT
 ecpg embedded SQL preprocessor
 Reduce system column overhead(Vadmin)
 Remove pg_time table(Vadim)
 Add pg_type attribute to identify types that need length (bpchar, varchar)
 Add report of offending line when COPY command fails
 Allow VIEW privileges to be set separately from the underlying tables.
 For security, use GRANT/REVOKE on views as appropriate(Jan)
 Tables now have no default GRANT SELECT TO PUBLIC. You must explicitly grant such privileges.
 Clean up tutorial examples(Darren)

Source Tree Changes

 Add new html development tools, and flow chart in /tools/backend
 Fix for SCO compiles
 Stratus computer port Robert Gillies
 Added support for shlib for BSD44 derived & i386_solaris
 Make configure more automated(Brook)
 Add script to check regression test results
 Break parser functions into smaller files, group together(Bruce)
 Rename heap_create to heap_create_and_catalog, rename heap_creatr to heap_create()(Bruce)
 Sparc/Linux patch for locking(TomS)
 Remove PORTNAME and reorganize port-specific stuff(Marc)
 Add optimizer README file(Bruce)
 Remove some recursion in optimizer and clean up some code there(Bruce)
 Fix for NetBSD locking(Henry)
 Fix for libptcl make(Tatsuo)
 AIX patch(Darren)
 Change IS TRUE, IS FALSE, ... to expressions using "=" rather than function calls to isttrue() or isfalse() to allow optimization(Thomas)
 Various fixes NetBSD/Sparc related(TomH)
 Alpha linux locking(Travis,Ryan)
 Change elog(WARN) to elog(ERROR)(Bruce)
 FAQ for FreeBSD(Marc)
 Bring in the PostODBC source tree as part of our standard distribution(Marc)
 A minor patch for HP/UX 10 vs 9(Stan)
 New pg_attribute.atttypmod for type-specific info like varchar length(Bruce)
 UnixWare patches(Billy)
 New i386 'lock' for spinlock asm(Billy)
 Support for multiplexed backends is removed
 Start an OpenBSD port
 Start an AUX port
 Start a Cygnus port
 Add string functions to regression suite(Thomas)
 Expand a few function names formerly truncated to 16 characters(Thomas)
 Remove un-needed malloc() calls and replace with palloc()(Bruce)

E.227. リリース 6.2.1

リリース日: 1997-10-17

6.2.1 はリリース 6.2 の不具合の改修と利便性を上げたものです。

要約:

- 1つの SQL92 を複数行に跨った文字列とすることができます。
- テーブル更新時にユーザ名を挿入する、トリガ関数の例を含めました。

これは 6.2 の小さな不具合を改修したものです。6.2 よりも前のシステムからアップグレードを行うためには、完全なダンプ/リロードが必要です。手順については 6.2 のリリースノートを参照して下さい。

E.227.1. バージョン 6.2 からバージョン 6.2.1 への移行

これは小さな不具合を改修したリリースです。バージョン 6.2 からの場合は、ダンプ/リロードは必要ありませんが、それ以前のリリースでは必要です。

バージョン 6.2 からのアップグレードで、ダンプ/リロードを行った場合、avg(money) の計算が正確になったことが判るでしょう。この他全ての不具合の改修は、実行形式ファイルを更新した時点で効果が得られます。

ダンプ/リロードを行わない他の方法として、以下の SQL コマンドを `psql` から実行して、既存のシステムテーブルを更新する方法があります。

```
update pg_aggregate set aggfinalfn = 'cash div flt8'
where aggname = 'avg' and aggbasetype = 790;
```

これは、template1 を含め、全ての既存データベースに対して行う必要があります。

E.227.2. 変更点

```
Allow TIME and TYPE column names(Thomas)
Allow larger range of true/false as boolean values(Thomas)
Support output of "now" and "current"(Thomas)
Handle DEFAULT with INSERT of NULL properly(Vadim)
Fix for relation reference counts problem in buffer manager(Vadim)
Allow strings to span lines, like ANSI(Thomas)
Fix for backward cursor with ORDER BY(Vadim)
Fix avg(cash) computation(Thomas)
Fix for specifying a column twice in ORDER/GROUP BY(Vadim)
Documented new libpq function to return affected rows, PQcmdTuples(Bruce)
Trigger function for inserting user names for INSERT/UPDATE(Brook Milligan)
```

E.228. リリース 6.2

リリース日: 1997-10-02

以前のリリースの PostgreSQL からデータを移行する場合はダンプ/リストアが必要です。

E.228.1. バージョン 6.1 からバージョン 6.2 への移行

この移行の場合、6.1 データベースの完全なダンプと、6.2 のデータベースへのリストアが必要です。

6.1 のデータベースをダンプするには、6.2 の `pg_dump` や `pg_dumpall` ユーティリティを使用しなければならないことに注意して下さい。

E.228.2. バージョン 1.x からバージョン 6.2 への移行

1.* の初期リリースからの移行の場合は、まず、1.09 へアップグレードしなければなりません。COPY の出力書式がリリース 1.02 から改良されたからです。

E.228.3. 変更点

Bug Fixes

```

-----
Fix problems with pg_dump for inheritance, sequences, archive tables(Bruce)
Fix compile errors on overflow due to shifts, unsigned, and bad prototypes
    from Solaris(Diab Jerius)
Fix bugs in geometric line arithmetic (bad intersection calculations)(Thomas)
Check for geometric intersections at endpoints to avoid rounding ugliness(Thomas)
Catch non-functional delete attempts(Vadim)
Change time function names to be more consistent(Michael Reifenberg)
Check for zero divides(Michael Reifenberg)
Fix very old bug which made rows changed/inserted by a command
    visible to the command itself (so we had multiple update of
    updated rows, etc.)(Vadim)
Fix for SELECT null, 'fail' FROM pg_am (Patrick)
SELECT NULL as EMPTY FIELD now allowed(Patrick)
Remove un-needed signal stuff from contrib/pginterface
Fix OR (where x != 1 or x isnull didn't return rows with x NULL) (Vadim)
Fix time cmp function (Vadim)
Fix handling of functions with non-attribute first argument in
    WHERE clauses (Vadim)
Fix GROUP BY when order of entries is different from order
    in target list (Vadim)
Fix pg_dump for aggregates without sfunc1 (Vadim)

```

Enhancements

```

-----
Default genetic optimizer GEQO parameter is now 8(Bruce)
Allow use parameters in target list having aggregates in functions(Vadim)
Added JDBC driver as an interface(Adrian & Peter)
pg_password utility
Return number of rows inserted/affected by INSERT/UPDATE/DELETE etc.(Vadim)
Triggers implemented with CREATE TRIGGER (SQL3)(Vadim)
SPI (Server Programming Interface) allows execution of queries inside
    C-functions (Vadim)
NOT NULL implemented (SQL92)(Robson Paniago de Miranda)
Include reserved words for string handling, outer joins, and unions(Thomas)
Implement extended comments ("/* ... */") using exclusive states(Thomas)
Add "/*" single-line comments(Bruce)
Remove some restrictions on characters in operator names(Thomas)
DEFAULT and CONSTRAINT for tables implemented (SQL92)(Vadim & Thomas)
Add text concatenation operator and function (SQL92)(Thomas)
Support WITH TIME ZONE syntax (SQL92)(Thomas)
Support INTERVAL unit TO unit syntax (SQL92)(Thomas)
Define types DOUBLE PRECISION, INTERVAL, CHARACTER,
    and CHARACTER VARYING (SQL92)(Thomas)
Define type FLOAT(p) and rudimentary DECIMAL(p,s), NUMERIC(p,s) (SQL92)(Thomas)
Define EXTRACT(), POSITION(), SUBSTRING(), and TRIM() (SQL92)(Thomas)
Define CURRENT DATE, CURRENT TIME, CURRENT_TIMESTAMP (SQL92)(Thomas)
Add syntax and warnings for UNION, HAVING, INNER and OUTER JOIN (SQL92)(Thomas)
Add more reserved words, mostly for SQL92 compliance(Thomas)
Allow hh:mm:ss time entry for timespan/reftime types(Thomas)
Add center() routines for lseg, path, polygon(Thomas)
Add distance() routines for circle-polygon, polygon-polygon(Thomas)
Check explicitly for points and polygons contained within polygons
    using an axis-crossing algorithm(Thomas)
Add routine to convert circle-box(Thomas)
Merge conflicting operators for different geometric data types(Thomas)
Replace distance operator "<==>" with "<->"(Thomas)
Replace "above" operator "!^" with ">^" and "below" operator "!" with "<^"(Thomas)
Add routines for text trimming on both ends, substring, and string position(Thomas)
Added conversion routines circle(box) and poly(circle)(Thomas)
Allow internal sorts to be stored in memory rather than in files(Bruce & Vadim)
Allow functions and operators on internally-identical types to succeed(Bruce)

```

Speed up backend start-up after profiling analysis(Bruce)
Inline frequently called functions for performance(Bruce)
Reduce open() calls(Bruce)
psql: Add PAGER for Yh and Y?,YC fix
Fix for psql pager when no tty(Bruce)
New entab utility(Bruce)
General trigger functions for referential integrity (Vadim)
General trigger functions for time travel (Vadim)
General trigger functions for AUTOINCREMENT/IDENTITY feature (Vadim)
MOVE implementation (Vadim)

Source Tree Changes

HP-UX 10 patches (Vladimir Turin)
Added SCO support, (Daniel Harris)
MkLinux patches (Tatsuo Ishii)
Change geometric box terminology from "length" to "width"(Thomas)
Deprecate temporary unstored slope fields in geometric code(Thomas)
Remove restart instructions from INSTALL(Bruce)
Look in /usr/ucb first for install(Bruce)
Fix c++ copy example code(Thomas)
Add -o to psql manual page(Bruce)
Prevent relname unallocated string length from being copied into database(Bruce)
Cleanup for NAMEDATALEN use(Bruce)
Fix pg_proc names over 15 chars in output(Bruce)
Add strNcpy() function(Bruce)
remove some (void) casts that are unnecessary(Bruce)
new interfaces directory(Marc)
Replace fopen() calls with calls to fd.c functions(Bruce)
Make functions static where possible(Bruce)
enclose unused functions in #ifdef NOT_USED(Bruce)
Remove call to difftime() in timestamp support to fix SunOS(Bruce & Thomas)
Changes for Digital Unix
Portability fix for pg_dumpall(Bruce)
Rename pg attribute.attnvals to attdispersion(Bruce)
"intro/unix" manual page now "pgintro"(Bruce)
"built-in" manual page now "pgbuiltin"(Bruce)
"drop" manual page now "drop table"(Bruce)
Add "create trigger", "drop trigger" manual pages(Thomas)
Add constraints regression test(Vadim & Thomas)
Add comments syntax regression test(Thomas)
Add PGINDENT and support program(Bruce)
Massive commit to run PGINDENT on all *.c and *.h files(Bruce)
Files moved to /src/tools directory(Bruce)
SPI and Trigger programming guides (Vadim & D'Arcy)

[前のページ](#)

リリース 6.2.1

[ホーム](#)

[上に戻る](#)

[次のページ](#)

リリース 6.1.1

E.229. リリース 6.1.1

リリース日: 1997-07-22

E.229.1. バージョン 6.1 からバージョン 6.1.1 への移行

これは小さな不具合を改修したリリースです。バージョン 6.1 からの移行の場合はダンプ/リロードは不要ですが、それ以前のリリースからの場合は全てダンプ/リロードは必要です。より詳細は 6.1 のリリースノートを参照して下さい。

E.229.2. 変更点

```
fix for SET with options (Thomas)
allow pg_dump/pg_dumpall to preserve ownership of all tables/objects(Bruce)
new psql ¥connect option allows changing usernames without changing databases
fix for initdb --debug option(Yoshihiko Ichikawa)
lextest cleanup(Bruce)
hash fixes(Vadim)
fix date/time month boundary arithmetic(Thomas)
fix timezone daylight handling for some ports(Thomas, Bruce, Tatsuo)
timestamp overhauled to use standard functions(Thomas)
other code cleanup in date/time routines(Thomas)
psql's ¥d now case-insensitive(Bruce)
psql's backslash commands can now have trailing semicolon(Bruce)
fix memory leak in psql when using ¥g(Bruce)
major fix for endian handling of communication to server(Thomas, Tatsuo)
Fix for Solaris assembler and include files(Yoshihiko Ichikawa)
allow underscores in usernames(Bruce)
pg_dumpall now returns proper status, portability fix(Bruce)
```

E.230. リリース 6.1

リリース日: 1997-06-08

PostgreSQL 6.1 リリースでは、リグレッションテストが追加され、また、多くの変更がなされました。

新しい 3 個のデータ型(`datetime`、`timespan`、`circle`) が PostgreSQL 固有の型セットとして追加されました。points、boxes、paths、polygons の出力書式において、これら型の間で一貫性を持たせました。misc.out における polygon 出力は元のリグレッション出力からの相対値の正確さをチェックする部分的なものになりました。

PostgreSQL 6.1 では、遺伝的アルゴリズムを使用した、新しい、もう一つのオプティマイザを導入しました。問い合わせに複数の条件や複数のテーブルが含まれる(オプティマイザが評価の順番を選択することになる)場合、このアルゴリズムは問い合わせ結果の順番において不規則な動作をもたらします。多くのリグレッションテストがその結果を明示的に順序付けするように変更されたので、オプティマイザの選択の影響はありません。ごく一部の (例えば、point や time interval などの) 生来順番を持たないデータ型用のリグレッションテストやこれらの型を使用するテスト、明示的に `set geqo to 'off'` と `reset geqo` の間で行われます。

配列指示子(大括弧に囲まれた原子値)の解釈が、元となるリグレッションテストが生成された後に数回変更されたようです。現在の `./expected/*.out` はこの新しい解釈を反映していますが、間違っているかもしれません。

float8 リグレッションテストは少なくともいくつかのプラットフォームで失敗します。これは、`pow()` と `exp()` の実装やオーバーフロー、アンダーフロー検出に使用される通知機構の違いが原因です。

random テストの結果は"不規則"なものです。また、リグレッションテストの評価は単なる `diff` コマンドを使用しますので、"random" テストは"失敗"します。しかし、私の試験用マシン(Linux/gcc/i686)では "random" は不規則な結果を生成しないようです。

E.230.1. バージョン 6.1 への移行

この移行の場合、6.0 データベースの完全なダンプと、6.1 のデータベースへのリストアが必要です。

1.* の初期リリースからの移行の場合は、まず、1.09 へアップグレードしなければなりません。COPY の出力書式がリリース 1.02 から改良されたからです。

E.230.2. 変更点

```

Bug Fixes
-----
packet length checking in library routines
lock manager priority patch
check for under/over flow of float8(Bruce)
multitable join fix(Vadim)
SIGPIPE crash fix(Darren)
large object fixes(Sven)
allow btree indexes to handle NULLs(Vadim)
timezone fixes(D'Arcy)
select SUM(x) can return NULL on no rows(Thomas)
internal optimizer, executor bug fixes(Vadim)
fix problem where inner loop in < or <= has no rows(Vadim)
prevent re-commuting join index clauses(Vadim)
fix join clauses for multiple tables(Vadim)
fix hash, hashjoin for arrays(Vadim)
fix btree for abstime type(Vadim)
large object fixes(Raymond)
fix buffer leak in hash indexes (Vadim)
fix rtree for use in inner scan (Vadim)
fix gist for use in inner scan, cleanups (Vadim, Andrea)
avoid unnecessary local buffers allocation (Vadim, Massimo)
fix local buffers leak in transaction aborts (Vadim)
fix file manager memory leaks, cleanups (Vadim, Massimo)
fix storage manager memory leaks (Vadim)
fix btree duplicates handling (Vadim)
fix deleted rows reincarnation caused by vacuum (Vadim)
fix SELECT varchar()/char() INTO TABLE made zero-length fields(Bruce)
many psql, pg_dump, and libpq memory leaks fixed using Purify (Igor)

Enhancements
-----
attribute optimization statistics(Bruce)

```

much faster new btree bulk load code(Paul)
BTREE UNIQUE added to bulk load code(Vadim)
new lock debug code(Massimo)
massive changes to libpg++(Leo)
new GEQO optimizer speeds table multitable optimization(Martin)
new WARN message for non-unique insert into unique key(Marc)
update x=-3, no spaces, now valid(Bruce)
remove case-sensitive identifier handling(Bruce,Thomas,Dan)
debug backend now pretty-prints tree(Darren)
new Oracle character functions(Edmund)
new plaintext password functions(Dan)
no such class or insufficient privilege changed to distinct messages(Dan)
new ANSI timestamp function(Dan)
new ANSI Time and Date types (Thomas)
move large chunks of data in backend(Martin)
multicolumn btree indexes(Vadim)
new SET var TO value command(Martin)
update transaction status on reads(Dan)
new locale settings for character types(Oleg)
new SEQUENCE serial number generator(Vadim)
GROUP BY function now possible(Vadim)
re-organize regression test(Thomas,Marc)
new optimizer operation weights(Vadim)
new psql %z grant/permit option(Marc)
new MONEY data type(D'Arcy,Thomas)
tcp socket communication speed improved(Vadim)
new VACUUM option for attribute statistics, and for certain columns (Vadim)
many geometric type improvements(Thomas,Keith)
additional regression tests(Thomas)
new datestyle variable(Thomas,Vadim,Martin)
more comparison operators for sorting types(Thomas)
new conversion functions(Thomas)
new more compact btree format(Vadim)
allow pg_dumpall to preserve database ownership(Bruce)
new SET GEQO=# and R_PLANS variable(Vadim)
old (!GEQO) optimizer can use right-sided plans (Vadim)
typechecking improvement in SQL parser(Bruce)
new SET, SHOW, RESET commands(Thomas,Vadim)
new %connect database USER option
new destroydb -i option (Igor)
new %dt and %di psql commands (Darren)
SELECT "%n" now escapes newline (A. Duursma)
new geometry conversion functions from old format (Thomas)

Source tree changes

new configuration script(Marc)
readline configuration option added(Marc)
OS-specific configuration options removed(Marc)
new OS-specific template files(Marc)
no more need to edit Makefile.global(Marc)
re-arrange include files(Marc)
nextstep patches (Gregor Hoffleit)
removed Windows-specific code(Bruce)
removed postmaster -e option, now only postgres -e option (Bruce)
merge duplicate library code in front/backends(Martin)
now works with eBones, international Kerberos(Jun)
more shared library support
c++ include file cleanup(Bruce)
warn about buggy flex(Bruce)
DG/UX, Ultrix, IRIX, AIX portability fixes

[前のページ](#)
リリース 6.1.1

[ホーム](#)
[上に戻る](#)

[次のページ](#)
リリース 6.0

E.231. リリース 6.0

リリース日: 1997-01-29

以前のリリースの PostgreSQL からデータを移行する場合はダンプ/リストアが必要です。

E.231.1. バージョン 1.09 からバージョン 6.0 への移行

この移行の場合、1.09 データベースの完全なダンプと、6.0 のデータベースへのリストアが必要です。

E.231.2. 1.09より前のバージョンからバージョン 6.0 への移行

1.* の初期リリースからの移行の場合は、まず、1.09 へアップグレードしなければなりません。COPY の出力書式がリリース 1.02 から改良されたからです。

E.231.3. 変更点

Bug Fixes

```

-----
ALTER TABLE bug - running postgres process needs to re-read table definition
Allow vacuum to be run on one table or entire database(Bruce)
Array fixes
Fix array over-runs of memory writes(Kurt)
Fix elusive btree range/non-range bug(Dan)
Fix for hash indexes on some types like time and date
Fix for pg_log size explosion
Fix permissions on lo_export() (Bruce)
Fix uninitialized reads of memory(Kurt)
Fixed ALTER TABLE ... char(3) bug(Bruce)
Fixed a few small memory leaks
Fixed EXPLAIN handling of options and changed full_path option name
Fixed output of group acl privileges
Memory leaks (hunt and destroy with tools like Purify(Kurt)
Minor improvements to rules system
NOTIFY fixes
New asserts for run-checking
Overhauled parser/analyze code to properly report errors and increase speed
Pg_dump -d now handles NULL's properly(Bruce)
Prevent SELECT NULL from crashing server (Bruce)
Properly report errors when INSERT ... SELECT columns did not match
Properly report errors when insert column names were not correct
psql %g filename now works(Bruce)
psql fixed problem with multiple statements on one line with multiple outputs
Removed duplicate system OIDs
SELECT * INTO TABLE . GROUP/ORDER BY gives unlink error if table exists(Bruce)
Several fixes for queries that crashed the backend
Starting quote in insert string errors(Bruce)
Submitting an empty query now returns empty status, not just " " query(Bruce)

```

Enhancements

```

-----
Add EXPLAIN manual page(Bruce)
Add UNIQUE index capability(Dan)
Add hostname/user level access control rather than just hostname and user
Add synonym of != for <>(Bruce)
Allow "select oid,* from table"
Allow BY,ORDER BY to specify columns by number, or by non-alias table.column(Bruce)
Allow COPY from the frontend(Bryan)
Allow GROUP BY to use alias column name(Bruce)
Allow actual compression, not just reuse on the same page(Vadim)
Allow installation-configuration option to auto-add all local users(Bryan)
Allow libpq to distinguish between text value '' and null(Bruce)
Allow non-postgres users with createdb privs to destroydb's
Allow restriction on who can create C functions(Bryan)
Allow restriction on who can do backend COPY(Bryan)
Can shrink tables, pg_time and pg_log(Vadim & Erich)
Change debug level 2 to print queries only, changed debug heading layout(Bruce)
Change default decimal constant representation from float4 to float8(Bruce)
European date format now set when postmaster is started
Execute lowercase function names if not found with exact case
Fixes for aggregate/GROUP processing, allow 'select sum(func(x),sum(x+y) from z'
Gist now included in the distribution(Marc)
Ident authentication of local users(Bryan)
Implement BETWEEN qualifier(Bruce)
Implement IN qualifier(Bruce)
libpq has PQgetisnull() (Bruce)
libpq++ improvements
New options to initdb(Bryan)
Pg_dump allow dump of OIDs(Bruce)
Pg_dump create indexes after tables are loaded for speed(Bruce)
Pg_dumpall dumps all databases, and the user table
Pginterface additions for NULL values(Bruce)
Prevent postmaster from being run as root
psql %h and %? is now readable(Bruce)
psql allow backslashed, semicolons anywhere on the line(Bruce)

```

psql changed command prompt for lines in query or in quotes(Bruce)
psql char(3) now displays as (bp)char in %d output(Bruce)
psql return code now more accurate(Bryan?)
psql updated help syntax(Bruce)
Re-visit and fix vacuum(Vadim)
Reduce size of regression diffs, remove timezone name difference(Bruce)
Remove compile-time parameters to enable binary distributions(Bryan)
Reverse meaning of HBA masks(Bryan)
Secure Authentication of local users(Bryan)
Speed up vacuum(Vadim)
Vacuum now had VERBOSE option(Bruce)

Source tree changes

All functions now have prototypes that are compared against the calls
Allow asserts to be disabled easily from Makefile.global(Bruce)
Change oid constants used in code to #define names
Decoupled sparc and solaris defines(Kurt)
Gcc -Wall compiles cleanly with warnings only from unfixable constructs
Major include file reorganization/reduction(Marc)
Make now stops on compile failure(Bryan)
Makefile restructuring(Bryan, Marc)
Merge bsdi 2.1 to bsdi(Bruce)
Monitor program removed
Name change from Postgres95 to PostgreSQL
New config.h file(Marc, Bryan)
PG VERSION now set to 6.0 and used by postmaster
Portability additions, including Ultrix, DG/UX, AIX, and Solaris
Reduced the number of #define's, centralized #define's
Remove duplicate OIDS in system tables(Dan)
Remove duplicate system catalog info or report mismatches(Dan)
Removed many os-specific #define's
Restructured object file generation/location(Bryan, Marc)
Restructured port-specific file locations(Bryan, Marc)
Unused/uninitialized variables corrected

[前のページ](#)
リリース 6.1

[ホーム](#)
[上に戻る](#)

[次のページ](#)
リリース 1.09

E.232. リリース 1.09

リリース日: 1996-11-04

すいません。 1.02 から 1.09 での変更履歴は残っていません。6.0 にて示した変更点の中には、実際には 1.02.1 から 1.09 リリースでの変更も含まれています。

[前のページ](#)

リリース 6.0

[ホーム](#)[上に戻る](#)[次のページ](#)

リリース 1.02

E.233. リリース 1.02

リリース日: 1996-08-01

E.233.1. バージョン 1.02 からバージョン 1.02.1 への移行

1.02.1 用の新しい移行ファイルがあります。このファイルには 'copy' の変更や古い ASCII ファイルの変換スクリプトがあります。

注意: 以下の注意は、Postgres95 1.01 および 1.02 から Postgres95 1.02.1 への移行を行うユーザ向けのものです。

新規に Postgres95 1.02.1 を使用し始める場合は古いデータベースを移行する必要はありませんので、これ以上読み続ける必要はありません。

古い Postgres95 バージョン 1.01 または 1.02 データベースからバージョン 1.02.1 へアップグレードするためには、以下の手順が必要です。

1 11 新しい 1.02.1 の postmaster を起動します。

1 11 1.02.1 の新しい組み込み関数と演算子を 1.01 または 1.02 データベースに追加します。これは所有する 1.01 または 1.02 データベースに対して、新しい 1.02.1 サーバを実行させ、このファイルの最後にある問い合わせを実行することで行われます。psql を使用して簡単に実施できます。1.01 または 1.02 データベースの名前が testdb であり、また、このファイルの最後の部分を addfunc.sql に保存したとすると、以下のようになります。

```
% psql testdb -f addfunc.sql
```

1.02 では既に存在するため、1.02 データベースのアップグレードでは、ファイルの最後の2文の実行の際に警告が表示されます。気にする必要はありません。

E.233.2. ダンプ/リロード手順

以前のバージョンで生成した、pg_dump や テキストモードの copy tablename to stdout をリロードする場合、データベースにロードする前に、その ASCII ファイルに対して以下の sed を実行する必要があります。古い書式では '.' をデータ終端として使用していますが、'\.' がデータ終端を示すようになりました。また、空文字列は NULL ではなく '' としてロードされるようになりました。完全な詳細については copy マニュアルページを参照して下さい。

```
sed 's/^\.\$/\.\./g' <in_file >out_file
```

古いバイナリ copy や標準出力以外の copy からロードする場合は、データ終端文字はありませんので、変換する必要はありません。

```
-- following lines added by agc to reflect the case-insensitive
-- regexp searching for varchar (in 1.02), and bpchar (in 1.02.1)
create operator ~* (leftarg = bpchar, rightarg = text, procedure = texticregexeq);
create operator !~* (leftarg = bpchar, rightarg = text, procedure = texticregexne);
create operator ~* (leftarg = varchar, rightarg = text, procedure = texticregexeq);
create operator !~* (leftarg = varchar, rightarg = text, procedure = texticregexne);
```

E.233.3. 変更点

Source code maintenance and development
* worldwide team of volunteers
* the source tree now in CVS at ftp.ki.net

Enhancements
* psql (and underlying libpq library) now has many more options for formatting output, including HTML
* pg_dump now output the schema and/or the data, with many fixes to enhance completeness.
* psql used in place of monitor in administration shell scripts. monitor to be deprecated in next release.
* date/time functions enhanced
* NULL insert/update/comparison fixed/enhanced
* TCL/TK lib and shell fixed to work with both tcl7.4/tk4.0 and tcl7.5/tk4.1

Bug Fixes (almost too numerous to mention)

- * indexes
- * storage management
- * check for NULL pointer before dereferencing
- * Makefile fixes

New Ports

- * added SolarisX86 port
- * added BSD/OS 2.1 port
- * added DG/UX port

[前のページ](#)

リリース 1.09

[ホーム](#)
[上に戻る](#)

[次のページ](#)

リリース 1.01

E.234. リリース 1.01

リリース日: 1996-02-23

E.234.1. バージョン 1.0 からバージョン 1.01 への移行

以下の注意は、Postgres95 1.0 から Postgres95 1.01 への移行を行うユーザ向けのものです。

新規に Postgres95 1.02.1 を使用し始める場合は古いデータベースを移行する必要はありませんので、これ以上読み続ける必要はありません。

Postgres95 バージョン 1.0 で作成したデータベースをPostgres95 バージョン 1.01 で使用するためには、以下の手順が必要です。

1 11 src/Makefile.global 内の NAMEDATALEN の定義を 16 に、OIDNAMELEN の定義を 20 に設定します。

1 11 ホストベース認証方式を使用するかどうか決定します。

1 11 使用する場合、最上位データディレクトリ(通常は \$PGDATA の値)に pg_hba という名前のファイルを作成しなければなりません。src/libpq/pg_hba に構文の例があります。

1 11 ホストベース認証方式を使用しない場合、src/Makefile.global 内の以下の行をコメントアウトすることができます。

```
HBA = 1
```

```
in src/Makefile.global
```

ホストベース認証方式はデフォルトで有効であり、上の A または B の手順を行わなかった場合、1.01 ボックス以外から 1.0 データベースへ接続することはできなくなることにご注意下さい。

1 11 1.01 をコンパイルし、インストールします。しかし、initdb を実行してはいけません。

1 11 これ以上作業する前に、1.0 のpostmaster を終了させ、既存の \$PGDATA ディレクトリをバックアップします。

1 11 PGDATA 環境変数を 1.0 データベースに設定し、パスを 1.01のバイナリが使用されるように設定します。

1 11 \$PGDATA/PG_VERSION ファイルを 5.0から5.1へ変更します。

1 11 新しい 1.01 postmaster を起動します。

1 11 1.01 の新しい組み込み関数と演算子を 1.0 データベースに追加します。これは所有する 1.0 データベースに対して、新しい 1.01 サーバを実行させ、添付の問い合わせを 1.0_to_1.01.sql に保存して実行して下さい。psql を使用して簡単に実施できます。1.0 データベースの名前が testdb の場合は以下のようになります。

```
% psql testdb -f 1.0_to_1.01.sql
```

(以下を切り出して使用して下さい。)

```
-- add builtin functions that are new to 1.01

create function int4eqoid (int4, oid) returns bool as 'foo'
language 'internal';
create function oideqint4 (oid, int4) returns bool as 'foo'
language 'internal';
create function char2icregexeq (char2, text) returns bool as 'foo'
language 'internal';
create function char2icregexne (char2, text) returns bool as 'foo'
language 'internal';
create function char4icregexeq (char4, text) returns bool as 'foo'
language 'internal';
create function char4icregexne (char4, text) returns bool as 'foo'
language 'internal';
create function char8icregexeq (char8, text) returns bool as 'foo'
language 'internal';
create function char8icregexne (char8, text) returns bool as 'foo'
language 'internal';
create function char16icregexeq (char16, text) returns bool as 'foo'
language 'internal';
create function char16icregexne (char16, text) returns bool as 'foo'
```

```

language 'internal';
create function texticregexeq (text, text) returns bool as 'foo'
language 'internal';
create function texticregexne (text, text) returns bool as 'foo'
language 'internal';

-- add builtin functions that are new to 1.01

create operator = (leftarg = int4, rightarg = oid, procedure = int4eqoid);
create operator = (leftarg = oid, rightarg = int4, procedure = oideqint4);
create operator ~* (leftarg = char2, rightarg = text, procedure = char2icregexeq);
create operator !~* (leftarg = char2, rightarg = text, procedure = char2icregexne);
create operator ~* (leftarg = char4, rightarg = text, procedure = char4icregexeq);
create operator !~* (leftarg = char4, rightarg = text, procedure = char4icregexne);
create operator ~* (leftarg = char8, rightarg = text, procedure = char8icregexeq);
create operator !~* (leftarg = char8, rightarg = text, procedure = char8icregexne);
create operator ~* (leftarg = char16, rightarg = text, procedure = char16icregexeq);
create operator !~* (leftarg = char16, rightarg = text, procedure = char16icregexne);
create operator ~* (leftarg = text, rightarg = text, procedure = texticregexeq);
create operator !~* (leftarg = text, rightarg = text, procedure = texticregexne);

```

E.234.2. 変更点

Incompatibilities:

- * 1.01 is backwards compatible with 1.0 database provided the user follow the steps outlined in the `MIGRATION_from_1.0_to_1.01` file. If those steps are not taken, 1.01 is not compatible with 1.0 database.

Enhancements:

- * added `PQdisplayTuples()` to `libpq` and changed `monitor` and `psql` to use it
- * added `NeXT` port (requires `SysVIPC` implementation)
- * added `CAST .. AS ...` syntax
- * added `ASC` and `DESC` key words
- * added `'internal'` as a possible language for `CREATE FUNCTION` internal functions are C functions which have been statically linked into the postgres backend.
- * a new type `"name"` has been added for system identifiers (table names, attribute names, etc.) This replaces the old `char16` type. The of name is set by the `NAMEDATALEN` #define in `src/Makefile.global`
- * a readable reference manual that describes the query language.
- * added host-based access control. A configuration file (`$PGDATA/pg_hba`) is used to hold the configuration data. If host-based access control is not desired, comment out `HBA=1` in `src/Makefile.global`.
- * changed regex handling to be uniform use of Henry Spencer's regex code regardless of platform. The regex code is included in the distribution
- * added functions and operators for case-insensitive regular expressions. The operators are `~*` and `!~*`.
- * `pg_dump` uses `COPY` instead of `SELECT` loop for better performance

Bug fixes:

- * fixed an optimizer bug that was causing core dumps when functions calls were used in comparisons in the `WHERE` clause
- * changed all uses of `getuid` to `geteuid` so that effective uids are used
- * `psql` now returns non-zero status on errors when using `-c`
- * applied public patches 1-14

E.235. リリース 1.0

リリース日: 1995-09-05

E.235.1. 変更点

Copyright change:

- * The copyright of Postgres 1.0 has been loosened to be freely modifiable and modifiable for any purpose. Please read the COPYRIGHT file.
- Thanks to Professor Michael Stonebraker for making this possible.

Incompatibilities:

- * date formats have to be MM-DD-YYYY (or DD-MM-YYYY if you're using EUROPEAN STYLE). This follows SQL-92 specs.
- * "delimiters" is now a key word

Enhancements:

- * sql LIKE syntax has been added
- * copy command now takes an optional USING DELIMITER specification. delimiters can be any single-character string.
- * IRIX 5.3 port has been added.
- Thanks to Paul Walmsley and others.
- * updated pg_dump to work with new libpq
- * %d has been added psql
- Thanks to Keith Parks
- * regexp performance for architectures that use POSIX regex has been improved due to caching of precompiled patterns.
- Thanks to Alistair Crooks
- * a new version of libpq++
- Thanks to William Wanders

Bug fixes:

- * arbitrary userids can be specified in the createuser script
- * %c to connect to other databases in psql now works.
- * bad pg_proc entry for float4inc() is fixed
- * users with usecreatedb field set can now create databases without having to be usesuper
- * remove access control entries when the entry no longer has any privileges
- * fixed non-portable datetimes implementation
- * added kerberos flags to the src/backend/Makefile
- * libpq now works with kerberos
- * typographic errors in the user manual have been corrected.
- * btrees with multiple index never worked, now we tell you they don't work when you try to use them

E.236. Postgres95 リリース 0.03

リリース日: 1995-07-21

E.236.1. 変更点

Incompatible changes:

- * BETA-0.3 IS INCOMPATIBLE WITH DATABASES CREATED WITH PREVIOUS VERSIONS (due to system catalog changes and indexing structure changes).
- * double-quote (") is deprecated as a quoting character for string literals; you need to convert them to single quotes (').
- * name of aggregates (eg. int4sum) are renamed in accordance with the SQL standard (eg. sum).
- * CHANGE ACL syntax is replaced by GRANT/REVOKE syntax.
- * float literals (eg. 3.14) are now of type float4 (instead of float8 in previous releases); you might have to do typecasting if you depend on it being of type float8. If you neglect to do the typecasting and you assign a float literal to a field of type float8, you might get incorrect values stored!
- * LIBPQ has been totally revamped so that frontend applications can connect to multiple backends
- * the usesysid field in pg user has been changed from int2 to int4 to allow wider range of Unix user ids.
- * the netbsd/freebsd/bsd o/s ports have been consolidated into a single BSD44_derived port. (thanks to Alistair Crooks)

SQL standard-compliance (the following details changes that makes postgres95 more compliant to the SQL-92 standard):

- * the following SQL types are now built-in: smallint, int(eger), float, real, char(N), varchar(N), date and time.

The following are aliases to existing postgres types:

```
smallint -> int2
integer, int -> int4
float, real -> float4
```

- char(N) and varchar(N) are implemented as truncated text types. In addition, char(N) does blank-padding.
- * single-quote (') is used for quoting string literals; ' (in addition to ¥') is supported as means of inserting a single quote in a string
- * SQL standard aggregate names (MAX, MIN, AVG, SUM, COUNT) are used (Also, aggregates can now be overloaded, i.e. you can define your own MAX aggregate to take in a user-defined type.)
- * CHANGE ACL removed. GRANT/REVOKE syntax added.
 - Privileges can be given to a group using the "GROUP" key word.

For example:

```
GRANT SELECT ON foobar TO GROUP my_group;
```

The key word 'PUBLIC' is also supported to mean all users.

Privileges can only be granted or revoked to one user or group at a time.

"WITH GRANT OPTION" is not supported. Only class owners can change access control
 - The default access control is to grant users readonly access. You must explicitly grant insert/update access to users. To change this, modify the line in


```
src/backend/utils/acl.h
```

that defines ACL_WORLD_DEFAULT

Bug fixes:

- * the bug where aggregates of empty tables were not run has been fixed. Now, aggregates run on empty tables will return the initial conditions of the aggregates. Thus, COUNT of an empty table will now properly return 0. MAX/MIN of an empty table will return a row of value NULL.
- * allow the use of ¥; inside the monitor
- * the LISTEN/NOTIFY asynchronous notification mechanism now work
- * NOTIFY in rule action bodies now work
- * hash indexes work, and access methods in general should perform better. creation of large btree indexes should be much faster. (thanks to Paul Aoki)

Other changes and enhancements:

- * addition of an EXPLAIN statement used for explaining the query execution plan (eg. "EXPLAIN SELECT * FROM EMP" prints out the execution plan for the query).
- * WARN and NOTICE messages no longer have timestamps on them. To turn on timestamps of error messages, uncomment the line in


```
src/backend/utils/elog.h:
```

```
/* define ELOG_TIMESTAMPS */
```
- * On an access control violation, the message


```
"Either no such class or insufficient privilege"
```

 will be given. This is the same message that is returned when a class is not found. This dissuades non-privileged users from guessing the existence of privileged classes.
- * some additional system catalog changes have been made that are not visible to the user.

libpgtcl changes:

- * The -oid option has been added to the "pg_result" tcl command. pg_result -oid returns oid of the last row inserted. If the last command was not an INSERT, then pg_result -oid returns "".
- * the large object interface is available as pg_lo* tcl commands: pg_lo_open, pg_lo_close, pg_lo_creat, etc.

Portability enhancements and New Ports:

- * flex/lex problems have been cleared up. Now, you should be able to use flex instead of lex on any platforms. We no longer make assumptions of what lexer you use based on the platform you use.
- * The Linux-ELF port is now supported. Various configuration have been tested: The following configuration is known to work:
kernel 1.2.10, gcc 2.6.3, libc 4.7.2, flex 2.5.2, bison 1.24
with everything in ELF format,

New utilities:

- * ipcclean added to the distribution
ipcclean usually does not need to be run, but if your backend crashes and leaves shared memory segments hanging around, ipcclean will clean them up for you.

New documentation:

- * the user manual has been revised and libpq documentation added.

[前のページ](#)

リリース 1.0

[ホーム](#)
[上に戻る](#)

[次のページ](#)

Postgres95 リリース 0.02

E.237. Postgres95 リリース 0.02

リリース日: 1995-05-25

E.237.1. 変更点

Incompatible changes:

- * The SQL statement for creating a database is 'CREATE DATABASE' instead of 'CREATEDB'. Similarly, dropping a database is 'DROP DATABASE' instead of 'DESTROYDB'. However, the names of the executables 'createdb' and 'destroydb' remain the same.

New tools:

- * pgperl - a Perl (4.036) interface to Postgres95
- * pg_dump - a utility for dumping out a postgres database into a script file containing query commands. The script files are in a ASCII format and can be used to reconstruct the database, even on other machines and other architectures. (Also good for converting a Postgres 4.2 database to Postgres95 database.)

The following ports have been incorporated into postgres95-beta-0.02:

- * the NetBSD port by Alistair Crooks
- * the AIX port by Mike Tung
- * the Windows NT port by Jon Forrest (more stuff but not done yet)
- * the Linux ELF port by Brian Gallew

The following bugs have been fixed in postgres95-beta-0.02:

- * new lines not escaped in COPY OUT and problem with COPY OUT when first attribute is a '.'
- * cannot type return to use the default user id in createuser
- * SELECT DISTINCT on big tables crashes
- * Linux installation problems
- * monitor doesn't allow use of 'localhost' as PGHOST
- * psql core dumps when doing ¥c or ¥l
- * the "pgtclsh" target missing from src/bin/pgtclsh/Makefile
- * libpgtcl has a hard-wired default port number
- * SELECT DISTINCT INTO TABLE hangs
- * CREATE TYPE doesn't accept 'variable' as the internallength
- * wrong result using more than 1 aggregate in a SELECT

E.238. Postgres95 リリース 0.01

リリース日: 1995-05-01

最初のリリースです。

付録 F. 追加で提供されるモジュール

目次

- F.1. [adminpack](#)
- F.2. [auth_delay](#)
- F.3. [auto_explain](#)
- F.4. [btree_gin](#)
- F.5. [btree_gist](#)
- F.6. [chpasswd](#)
- F.7. [citext](#)
- F.8. [cube](#)
- F.9. [dblink](#)
- F.10. [dict_int](#)
- F.11. [dict_xsyn](#)
- F.12. [dummy_seclabel](#)
- F.13. [earthdistance](#)
- F.14. [file_fdw](#)
- F.15. [fuzzystrmatch](#)
- F.16. [hstore](#)
- F.17. [intagg](#)
- F.18. [intarray](#)
- F.19. [isn](#)
- F.20. [lo](#)
- F.21. [ltree](#)
- F.22. [pageinspect](#)
- F.23. [パスワード検査](#)
- F.24. [pg_buffercache](#)
- F.25. [pgcrypto](#)
- F.26. [pg_freespacemap](#)
- F.27. [pgrowlocks](#)
- F.28. [pg_stat_statements](#)
- F.29. [pgstattuple](#)
- F.30. [pg_trgm](#)
- F.31. [seq](#)
- F.32. [sepgsql](#)
- F.33. [spi](#)
- F.34. [sslinfo](#)
- F.35. [tablefunc](#)
- F.36. [tcn](#)
- F.37. [test_parser](#)
- F.38. [tsearch2](#)
- F.39. [unaccent](#)
- F.40. [uuid-ossdp](#)
- F.41. [xml2](#)

この付録と次の付録にはPostgreSQL配布物のcontribディレクトリにあるモジュールに関する情報があります。 ここには、移植用のツール、解析ユーティリティ、限定した利用者を対象にしていること、または、主ソースツリーに含めるには実験的すぎることが主な理由でPostgreSQLのコアシステムにはないプラグイン機能が含まれます。 これはその有用性を妨げるものではありません。

この付録では、contribにあるエクステンションやその他のサーバプラグインモジュールを説明します。

ソース配布から構築する場合、"world"を対象に構築しない限り、これらのモジュールは自動的に構築されません([ステップ2](#)参照)。 以下をconfigureを実行した後のソースツリー内のcontribディレクトリで実行することで、これらすべてを構築しインストールすることができます。

```
gmake
gmake install
```

選択した1つのモジュールのみを構築しインストールするには、そのモジュールのディレクトリで同じコマンドを行ってください。 多くのモジュールにはリグレーション試験があり、インストール前であれば以下のコマンドで、

```
gmake check
```

PostgreSQLサーバが動いている状態であれば以下のコマンドで実行できます。

```
gmake installcheck
```

PostgreSQLのパッケージ化されたバージョンを使用している場合は通常、例えば`postgresql-contrib`のような別途副パッケージとしてこれらのモジュールが利用可能です。

多くのモジュールは新しいユーザ定義関数、演算子、型を提供します。こうしたモジュールの1つを使用できるようにするためには、コードをインストールした後に、新しいオブジェクトをデータベースサーバに登録する必要があります。PostgreSQL 9.1以降では、これは[CREATE EXTENSION](#) コマンドを実行することで行われます。新しいデータベースでは、以下のように簡単に行うことができます。以下に例を示します。

```
CREATE EXTENSION module_name;
```

このコマンドはデータベーススーパーユーザにより実行されなければなりません。これは現在のデータベースの中にのみ新しいオブジェクトを登録します。このため、そのモジュールの機能を利用可能にさせたいデータベース毎にこのコマンドを実行しなければなりません。その拡張が今後作成されるデータベースにデフォルトでコピーされるように`template1`データベースに対して実行する方法もあります。

多くのモジュールはユーザが選択したスキーマ内にそのオブジェクトをインストールすることができます。これを行うためには`CREATE EXTENSION` コマンドに`SCHEMA schema_name`を追加してください。デフォルトでは、現在の作成対象スキーマ、通常は`public`内に格納されます。

9.1より前のバージョンのPostgreSQLからダンプしてリストアすることでデータベースができあがった場合、もし9.1より前のバージョンのモジュールを使用していたのであれば、代わりに以下を行わなければなりません。

```
CREATE EXTENSION module_name FROM unpackaged;
```

これは9.1より前のモジュール内のオブジェクトを適切な拡張オブジェクトに更新します。今後のモジュールの更新は[ALTER EXTENSION](#)により管理されます。拡張の更新に関する詳細については[項35.15](#)を参照してください。

しかしながら、いくつかのモジュールはこの意味での"エクステンション"ではなく、例えば[shared_preload_libraries](#)といった他の方法でサーバにロードされることに注意してください。各モジュールの詳細はドキュメントを参照してください。

[前のページ](#)

Postgres95 リリース 0.01

[ホーム](#)
[上に戻る](#)

[次のページ](#)
adminpack

F.1. adminpack

adminpackは、pgAdminやその他の管理・運用ツールがサーバログファイルの遠隔管理を行うなどの、追加的な機能を提供できるようにするための数多くのサポート機能を提供します。

F.1.1. 実装された機能

adminpackにより実装された機能はスーパーユーザのみが実行可能です。 以下がそれらの機能一覧です。

```
int8 pg_catalog.pg_file_write(fname text, data text, append bool)
bool pg_catalog.pg_file_rename(oldname text, newname text, archivename text)
bool pg_catalog.pg_file_rename(oldname text, newname text)
bool pg_catalog.pg_file_unlink(fname text)
setof record pg_catalog.pg_logdir_ls()

/* pgAdmin互換性用の現在残されているバックエンド機能 */
int8 pg_catalog.pg_file_read(fname text, data text, append bool)
bigint pg_catalog.pg_file_length(text)
int4 pg_catalog.pg_logfile_rotate()
```

F.2. auth_delay

`auth_delay`はパスワードの総当たり攻撃をより難しくするために認証エラーの報告を行う前にわずかにサーバを停止させます。これはDOS攻撃を防ぐためのものではありません。認証エラーを待たせ、コネクションスロットを消費させるため、DOS攻撃の影響を増長させるかもしれません。

この機能を有効にするためには`postgresql.conf`の [shared_preload_libraries](#)よりモジュールをロードする必要があります。

F.2.1. 設定パラメータ

`auth_delay.milliseconds (int)`

指定されたミリ秒数認証エラーを返す前に待機します。デフォルトは0です。

これらのパラメータを`postgresql.conf`ファイルに設定する必要があります。典型的な使用方法是以下ようになります。

```
# postgresql.conf
shared_preload_libraries = 'auth_delay'

auth_delay.milliseconds = '500'
```

F.2.2. 著作者

KaiGai Kohei <kaigai@ak.jp.nec.com>

F.3. auto_explain

auto_explainモジュールは、手動でEXPLAINの実行を必要とせず、自動的に遅い文の実行計画をログ記録する手段を提供します。大きなアプリケーションにおける最適化されていない問い合わせを追跡するのに特に有用です。

このモジュールはSQLでアクセスできる関数を提供しません。使用するには、サーバに単に読み込ませます。ある個別のセッションに読み込ませることができます。

```
LOAD 'auto_explain';
```

(実行するためにはスーパーユーザでなければなりません。) より一般的な使用方法は、postgresql.confの[shared_preload_libraries](#)にauto_explainを含めて、すべてのセッションで事前にロードしておくことです。すると、想定外に低速な問い合わせを発生時に何も行うことなく追跡することができます。当然ながらこのためのオーバーヘッドという代償があります。

F.3.1. 設定パラメータ

auto_explainの動作を制御するいくつかの構成パラメータが存在します。デフォルトの動作は何もしないことなので、なんらかの結果を望むのであれば少なくともauto_explain.log_min_durationを設定しなければならないことに注意してください。

auto_explain.log_min_duration (integer)

auto_explain.log_min_durationは、文の実行計画がログに記録されるようになる、ミリ秒単位の最小の文実行時間です。これをゼロにすれば、すべての計画が記録されます。マイナス1 (デフォルト) は計画の記録を無効にします。例えば、250msに設定すると、250ms以上実行する文すべてが記録されます。スーパーユーザのみがこの設定を変更することができます。

auto_explain.log_analyze (boolean)

auto_explain.log_analyzeは、実行計画のログが取得されたときに出力されるものとして、単にEXPLAIN出力ではなく、EXPLAIN ANALYZE出力を行います。このパラメータはデフォルトで無効です。スーパーユーザのみ、この設定を変更できます。

注意: このパラメータが有効の場合、計画ノードごとの時間的調整は事実上ログされるまで如何に時間が掛かろうと、全ての実行文に対して引き起こります。極端に性能上のマイナスの影響が起こり得ます。

auto_explain.log_verbose (boolean)

auto_explain.log_verboseは、実行計画のログが取得されたときに出力されるものとして、単にEXPLAIN出力ではなく、EXPLAIN VERBOSE出力を行います。このパラメータはデフォルトで無効です。スーパーユーザのみ、この設定を変更できます。

auto_explain.log_buffers (boolean)

auto_explain.log_buffersにより、実行計画のログを記録するときに、単なるEXPLAIN出力よりもEXPLAIN (ANALYZE, BUFFERS) 出力で記録ようになります。このパラメータはデフォルトで無効です。スーパーユーザのみがこの設定を変更することができます。auto_explain.log_analyzeパラメータが設定されていないければ、このパラメータは効果がありません。

auto_explain.log_format (enum)

auto_explain.log_formatは使用するEXPLAIN出力書式を選びます。許容される値はtext、xml、json、yamlです。デフォルトはtextです。スーパーユーザのみがこの設定を変更することができます。

auto_explain.log_nested_statements (boolean)

auto_explain.log_nested_statementsにより、入れ子状の文 (関数内から実行される文) を考慮して記録するようになります。無効ならば、最上位の問い合わせ計画のみが記録されます。このパラメータはデフォルトで無効です。スーパーユーザのみがこの設定を変更することができます。

これらパラメータをpostgresql.confに設定しなければなりません。典型的な使用法は以下のようになります。

```
# postgresql.conf
shared_preload_libraries = 'auto_explain'
```

```
auto_explain.log_min_duration = '3s'
```

F.3.2. 例

```
postgres=# LOAD 'auto_explain';
postgres=# SET auto_explain.log_min_duration = 0;
postgres=# SELECT count(*)
           FROM pg_class, pg_index
           WHERE oid = indrelid AND indisunique;
```

これにより、以下のようなログ出力が作成されます。

```
LOG:  duration: 3.651 ms  plan:
      Query Text: SELECT count(*)
                  FROM pg_class, pg_index
                  WHERE oid = indrelid AND indisunique;
Aggregate  (cost=16.79..16.80 rows=1 width=0) (actual time=3.626..3.627 rows=1 loops=1)
  -> Hash Join  (cost=4.17..16.55 rows=92 width=0) (actual time=3.349..3.594 rows=92 loops=1)
        Hash Cond: (pg_class.oid = pg_index.indrelid)
        -> Seq Scan on pg_class  (cost=0.00..9.55 rows=255 width=4) (actual time=0.016..0.140 rows=255
loops=1)
        -> Hash  (cost=3.02..3.02 rows=92 width=4) (actual time=3.238..3.238 rows=92 loops=1)
              Buckets: 1024  Batches: 1  Memory Usage: 4kB
              -> Seq Scan on pg_index  (cost=0.00..3.02 rows=92 width=4) (actual time=0.008..3.187
rows=92 loops=1)
              Filter: indisunique
```

F.3.3. 作者

板垣 貴裕 <itagaki.takahiro@oss.ntt.co.jp>

[前のページ](#)
auth_delay

[ホーム](#)
[上に戻る](#)

[次のページ](#)
btree_gin

F.4. btree_gin

btree_ginは、次に列挙するデータ型に対しB-treeと同等な動作を実装するGIN演算子クラスを提供します。データ型は、int2、int4、int8、float4、float8、timestamp with time zone、timestamp without time zone、time with time zone、time without time zone、date、interval、oid、money、"char"、varchar、text、bytea、bit、varbit、macaddr、inet、およびcidrです。

一般的に、これらの演算子クラスは同等な標準B-treeインデックス方式を性能的に凌駕する物ではなく、標準B-treeコードの1つの重要機能である一意性強要の能力を欠いています。しかしながら、GINの試験、およびその他のGIN演算子クラスの開発の基礎として便利です。同時に、GINインデックス化可能列およびB-treeインデックス化可能列双方を試験する問い合わせに対し、ビットマップを介してANDを取り一体化されるべき2つの別々のインデックスを作成するよりも、これらの演算子クラスの1つを使用する複数列GINインデックスを作成するほうがより効率的です。

F.4.1. 使用例

```
CREATE TABLE test (a int4);
-- create index
CREATE INDEX testidx ON test USING gin (a);
-- query
SELECT * FROM test WHERE a < 10;
```

F.4.2. 著作者

Teodor Sigaev (<teodor@stack.net>) および Oleg Bartunov (<oleg@sai.msu.su>)。追加情報は<http://www.sai.msu.su/~megera/oddmuse/index.cgi/Gin>を参照ください。

F.5. btree_gist

btree_gistは、次に列挙するデータ型に対しB-treeと同等な動作を実装するGiSTインデックス演算子クラスを提供します。

データ型は、int2、int4、int8、float4、float8、numeric、timestamp with time zone、timestamp without time zone、time with time zone、time without time zone、date、interval、oid、money、char、varchar、text、bytea、bit、varbit、macaddr、inet、およびcidrです。

一般的に、これらの演算子クラスは同等な標準B-treeインデックス方式を性能的に凌駕する物ではなく、標準B-treeコードの1つの重要機能である一意性強要の能力を欠いています。しかしながら、以下で述べるようにB-treeインデックスにはない特徴をいくつか備えています。また、これらの演算子クラスは、GiSTでのみインデックス可能なデータ型の列もあれば、単純なデータ型の列もあるような複数列のGiSTインデックスが必要な場合に便利です。最後に、GiSTの試験、およびその他のGiST演算子クラスの開発の基礎として便利です。

典型的なB-tree検索演算子に加えて、btree_gistは<> ("等しくない") に対してもインデックスのサポートを提供します。これは、後で述べるような[排他制約](#)と組み合わせると便利です。

また、自然な距離のあるデータ型には、btree_gistは距離演算子<->を定義し、この演算子を使った最近接検索へのGiSTインデックスのサポートを提供します。距離演算子はint2、int4、int8、float4、float8、timestamp with time zone、timestamp without time zone、time without time zone、date、interval、oid、moneyに提供されます。

F.5.1. 使用例

btreeの代わりにbtree_gistを使った簡単な例

```
CREATE TABLE test (a int4);
-- インデックスの作成
CREATE INDEX testidx ON test USING gist (a);
-- 問い合わせ
SELECT * FROM test WHERE a < 10;
-- 最近接検索: "42"に一番近い10個のエントリを見つける
SELECT *, a <-> 42 AS dist FROM test ORDER BY a <-> 42 LIMIT 10;
```

動物園の一つの檻に1種類の動物しかいないというルールを強制するために[排他制約](#)を使います。

```
=> CREATE TABLE zoo (
    cage    INTEGER,
    animal  TEXT,
    EXCLUDE USING gist (cage WITH =, animal WITH <>)
);

=> INSERT INTO zoo VALUES(123, 'zebra');
INSERT 0 1
=> INSERT INTO zoo VALUES(123, 'zebra');
INSERT 0 1
=> INSERT INTO zoo VALUES(123, 'lion');
ERROR:  conflicting key value violates exclusion constraint "zoo_cage_animal_excl"
DETAIL:  Key (cage, animal)=(123, lion) conflicts with existing key (cage, animal)=(123, zebra).
=> INSERT INTO zoo VALUES(124, 'lion');
INSERT 0 1
```

F.5.2. 著作者

Teodor Sigaev (<teodor@stack.net>)、Oleg Bartunov (<oleg@sai.msu.su>)、およびJanko Richter (<jankorichter@yahoo.de>)。追加情報は<http://www.sai.msu.su/~megera/postgres/gist/>を参照ください。

F.6. chkpass

本モジュールは暗号化したパスワードを格納するために設計された`chkpass`データ型を実装します。各パスワードは入力時に自動的に暗号化形式に変換され、常に暗号化して格納されます。比較する際は単に平文のパスワードと比較してください。そうすると、比較関数が比較する前に平文を暗号化してくれます。

パスワードが簡単にクラックできると判断される場合にエラーを報告する対策がコードに含まれています。しかし、現在は何も行わない単なるスタブです。

コロン付きの入力文字列を付けると、それはすでに暗号化済みのパスワードとみなされ、それ以上暗号化することなく格納されます。これにより事前に暗号化されたパスワードを入力することができます。

出力時はコロンが先頭に付けられます。これにより、暗号化を再度行うことなくパスワードのダンプと再ロードを行うことができます。コロンがない暗号化パスワードが必要ならば、`raw()` 関数を使用してください。これにより、Apacheの`Auth_PostgreSQL`モジュールのような作業にこの型を使用することができます。

暗号化は標準Unix関数`crypt()`を使用します。このため、その関数全ての制限の影響を受けます。とりわけ、パスワードの先頭8文字のみが考慮されます。

`chkpass`データ型のインデックス付けができないことに注意してください。

使用例を以下に示します。

```
test=# create table test (p chkpass);
CREATE TABLE
test=# insert into test values ('hello');
INSERT 0 1
test=# select * from test;
      p
-----
:dVGkpXdOrE3ko
(1 row)

test=# select raw(p) from test;
      raw
-----
dVGkpXdOrE3ko
(1 row)

test=# select p = 'hello' from test;
?column?
-----
t
(1 row)

test=# select p = 'goodbye' from test;
?column?
-----
f
(1 row)
```

F.6.1. 作者

D'Arcy J.M. Cain (<darcy@druid.net>)

F.7. citext

`citext` モジュールは、大文字小文字の区別がない文字列型を提供します。これは値の比較の際、基本的に内部的で `lower` を呼び出します。この他はほぼ `text` と同様に動作します。

F.7.1. 論拠

PostgreSQLにおいて大文字小文字の区別のない比較を行う標準的な手法は、値を比べる際に以下のように `lower` 関数を使用することでした。例です。

```
SELECT * FROM tab WHERE lower(col) = LOWER(?);
```

これはまあまあ動作しますが、数多くの欠点があります。

- 作成するSQL文を冗長にします。また常に列と問い合わせの値両方に `lower` を使用することを忘れないようにしなければなりません。
- `lower` を使用して関数インデックスを作成していない限り、インデックスを使用しません。
- `UNIQUE` または `PRIMARY KEY` として列を宣言するのであれば、暗黙的に生成されるインデックスは大文字小文字を区別します。このため、大文字小文字を区別しない検索では使えず、また、大文字小文字を区別しない一意性を強制させられません。

`citext` データ型によりSQL問い合わせ内の `lower` 呼び出しを省くことができます。さらに、大文字小文字の区別がないプライマリーキーを実現できます。`citext` は `text` と同様にロケールも考慮します。つまり大文字と小文字のマッチングは、`LC_CTYPE` データベース設定の規則に依存します。繰り返しますが、この動作は `lower` を使用した問い合わせと同一です。しかしこのデータ型により、ロケールの考慮は透過的に行われますので、問い合わせで特殊なことを行うことを覚えておく必要はありません。

F.7.2. 使用法

簡単な例を示します。

```
CREATE TABLE users (
    nick CITEXT PRIMARY KEY,
    pass TEXT NOT NULL
);

INSERT INTO users VALUES ( 'larry', md5(random()::text) );
INSERT INTO users VALUES ( 'Tom', md5(random()::text) );
INSERT INTO users VALUES ( 'Damian', md5(random()::text) );
INSERT INTO users VALUES ( 'NEAL', md5(random()::text) );
INSERT INTO users VALUES ( 'Bjørn', md5(random()::text) );

SELECT * FROM users WHERE nick = 'Larry';
```

`SELECT` 文は、`nick` 列が `larry` に設定され、問い合わせが `Larry` に対してであっても、1つのタプルを返します。

F.7.3. 文字列比較の動作

`citext` はそれぞれの文字列を（`lower` が呼ばれますが）小文字に変換して結果を普通に比較します。よって、例えば `lower` で小文字にした場合に同じ結果となるような2つの文字列が等しいとみなされます。

大文字小文字の区別のない照合をできる限り正確にエミュレートするために、数多くの `citext` 独自版の各種文字列処理演算子と関数があります。したがって、例えば正規表現演算子 `~` および `~*` は、`citext` に適用する時に同じ動作を提供します。これら両方は大文字小文字を区別することなくマッチさせます。`!~` や `!~*` だけではなく `LIKE` 演算子、`~~`、`~~*`、`!~~`、`!~~*` でも同じことが言えます。もし大文字小文字を区別して比較したい場合は、演算子の引数を `text` にキャストすることができます。

引数が `citext` であれば、同様にして以下の関数は大文字小文字を区別しない一致を実行します。

- `regexp_replace()`
- `regexp_split_to_array()`
- `regexp_split_to_table()`

- `replace()`
- `split_part()`
- `strpos()`
- `translate()`

正規表現関数（RegExp関数）では、大文字小文字を区別して一致させたい場合"i"フラグを付けて、強制的に大文字小文字を区別して一致させることができます。 そうしないと、大文字小文字を区別させたい場合にはこれらの関数のいずれかを使用する前段階で`text`にキャストしなければなりません。

F.7.4. 制限

- `citext`の大文字小文字を区別しない動作は使用するデータベースの`LC_CTYPE`に依存します。 どのように値を比較するかは、データベースが作成されたときに決定されます。 Unicode標準の定義という観点では、真に大文字小文字の区別がないわけではありません。 実質的に何を意味しているかという、使用している照合が十分なものであれば、`citext`による比較も十分なものになるはずです。 しかしデータベースに様々な言語でデータを格納している場合は、ある言語のユーザは照合が他の言語用のものであった場合想定外の問い合わせ結果を得るかもしれません。
- PostgreSQL 9.1では、`COLLATE`指定を`citext`列もしくはデータ値に付け加えることができます。 現状では、`citext`演算子は大文字小文字を含んだ文字列を比較する際に、デフォルトではない`COLLATE`指定を重んじます。 しかし、最初の小文字変換はデータベースの`LC_CTYPE`設定にしたがって、常に実行されます（つまり、`COLLATE "default"`が指定されたようになります） これは、両方のステップが入力された`COLLATE`指定に従うように、将来のリリースにおいて変更されるでしょう。
- 演算子関数およびB-Tree比較関数でデータの複製を作成しそれを比較のために小文字に変換しなければならないので、`citext`は`text`ほど効率的ではありません。 しかし大文字小文字の区別をしない一致をさせるために`lower`を使用する場合よりかなり効率的です。
- `citext`は、ある文脈では大文字小文字の区別を行い、またある文脈では大文字小文字の区別を行わない比較をする必要がある場合、あまり役に立ちません。 標準的な解法は`text`型を使用し、大文字小文字を区別する比較が必要であれば手作業で`lower`関数を使用することです。 これは大文字小文字を区別しない比較の必要性がまれであれば、問題なく動作します。 大文字小文字を区別しない比較がほとんどで、大文字小文字を区別する比較の必要性がまれである場合は、データを`citext`として格納し、大文字小文字を区別する比較の際にその列を明示的に`text`にキャストすることを検討してください。 どちらの場合でも、2種類の検索の両方を高速にするために2つのインデックスを作成しなければならないでしょう。
- `citext`演算子を含んだスキーマは、現在の`search_path`(典型的には`public`)に 存在しなければいけません。 もし無い場合は通常の大文字小文字が区別される`text`比較が代わりに呼び出されます。

F.7.5. 作者

David E. Wheeler <david@kineticcode.com>

Donald Fraserによる`citext`モジュール原本からのヒント

[前のページ](#)
chpasswd

[ホーム](#)
[上に戻る](#)

[次のページ](#)
cube

F.8. cube

本モジュールは、多次元立体を表すためのcubeデータ型を実装します。

F.8.1. 構文

[表F-1](#)はcube型で有効な外部表現を示します。 x 、 y などは浮動小数点数を意味します。

表 F-1. Cubeの外部表現

x	1次元の点。（または幅ゼロの一次元間隔）
(x)	同上
$x_1, x_2, \dots, x_n$	n 次元空間の点。 内部的には体積0の立方体として表されます。
$(x_1, x_2, \dots, x_n)$	同上
$(x), (y)$	x から y まで（またはその逆）の1次元の間隔。 順序は関係ありません。
$[(x), (y)]$	同上
$(x_1, \dots, x_n), (y_1, \dots, y_n)$	対角の組み合わせで表される n 次元の立体。
$[(x_1, \dots, x_n), (y_1, \dots, y_n)]$	同上

立体の対角の入力順序は関係ありません。 統一的な"左下 — 右上"という内部表現を作成するために必要ならば、cube関数は自動的に値を交換します。

空白文字は無視されます。 このため、 $[(x), (y)]$ は $[(\ x), (\ y)]$ と同じです。

F.8.2. 精度

値は内部的に64ビットの浮動小数点数値として格納されます。 つまり、有効桁が16より大きい数値は切り詰められることを意味します。

F.8.3. 使用方法

cubeモジュールには、cube型の値に対するGiSTインデックス演算子クラスが含まれます。 GiST演算子クラスでサポートされる演算子は[表F-2](#)に書かれている通りです。

表 F-2. cubeのGiST演算子

演算子	説明
$a = b$	立体aと立体bは同じです。
$a \ \&\& \ b$	立体aと立体bには重なりがあります。
$a \ @> \ b$	立体aは立体bを含みます。
$a \ <@ \ b$	立体aは立体bに含まれます。

（PostgreSQL 8.2以前では、含有演算子 $@>$ と $<@$ はそれぞれ $@$ と $\sim$ と呼ばれていました。 これらの名前はまだ利用できますが、廃止予定であり、最終的にはなくなります。 古い名前がコアの幾何データ型が従う規約と反対であることに注意してください。）

標準的なB-tree演算子も提供されます。 以下に例を示します。

演算子	説明
$[a, b] < [c, d]$	より小さい
$[a, b] > [c, d]$	より大きい

これらの演算子はソート以外の実用的な目的に対して大きな意味を持ちません。 これらの演算子はまず(a)と(c)を比べ、等しけ

れば(b)と(d)を比べます。 論理的にはほとんどの場合優れたソート処理とれます。 こうした型をORDER BYで使いたい場合に有用です。

表F-3は有効な関数を示します。

表 F-3. Cubeの関数

<code>cube(float8) returns cube</code>	同じ軸をもつ、1次元の立体を作成します。 <code>cube(1) == '(1)'</code>
<code>cube(float8, float8) returns cube</code>	1次元の立体を作成します。 <code>cube(1,2) == '(1),(2)'</code>
<code>cube(float8[]) returns cube</code>	配列で定義される座標を使用した体積0の立体を作成します。 <code>cube(ARRAY[1,2]) == '(1,2)'</code>
<code>cube(float8[], float8[]) returns cube</code>	2つの配列で定義される右上および左下座標を持つ立体を作成します。 配列長は同じでなければなりません。 <code>cube('{1,2'}::float[], '{3,4'}::float[]) == '(1,2),(3,4)'</code>
<code>cube(cube, float8) returns cube</code>	新しい座標の両部分に同一の値をを持たせて、既存の立体に次元を加えた立体を新しく作成します。 これは計算した値で部品を追加しながら立体を構築する場合に有用です。 <code>cube('(1)',2) == '(1,2),(1,2)'</code>
<code>cube(cube, float8, float8) returns cube</code>	既存の立体に次元を加えた立体を新しく作成します。 これは計算した値で部品を追加しながら立体を構築する場合に有用です。 <code>cube('(1,2)',3,4) == '(1,3),(2,4)'</code>
<code>cube_dim(cube) returns int</code>	立体の次元数を返します。
<code>cube_ll_coord(cube, int) returns double</code>	立体の左下隅のn次座標の値を返します。
<code>cube_ur_coord(cube, int) returns double</code>	立体の右上隅のn次座標の値を返します。
<code>cube_is_point(cube) returns bool</code>	立体が点、つまり2つの定義された隅が同一の場合真を返します。
<code>cube_distance(cube, cube) returns double</code>	2つの立体間の距離を返します。 両方の立体が点の場合、これは通常の距離測定関数です。
<code>cube_subset(cube, int[]) returns cube</code>	配列内の次元インデックスの一覧を使用して、既存の立体から新しい立体を作成します。 例えば、 <code>cube_subset(cube('(1,3,5),(6,7,8)'), ARRAY[2]) = '(3),(7)'</code> のように1次元のLLおよびUR座標を見つけるために使用することができます。 また、例えば、 <code>cube_subset(cube('(1,3,5),(6,7,8)'), ARRAY[3,2,1,1]) = '(5, 3, 1, 1),(8, 7, 6, 6)'</code> のように次元を除去することや希望通りの順序に並び替えることもできます。
<code>cube_union(cube, cube) returns cube</code>	2つの立体の和集合を作成します。
<code>cube_inter(cube, cube) returns cube</code>	2つの立体の共通部分を作成します。
<code>cube_enlarge(cube c, double r, int n) returns cube</code>	最小でn次元において指定した径で立体のサイズを増加させます。 径が負の場合、立体は縮小されます。 これは、近傍点を見つけるための点を囲む境界矩形を作成する際に有用です。 定義済のすべての次元は径rにより変わります。 LL座標をrにより減少し、UR座標をrにより増加します。 LL座標を対応するUR座標より大きくするために増加する場合（これはr < 0の場合にのみ発生します）、両方の座標はその平均値に設定されます。 nが定義済の次元より多く、かつ、立体が増加される（r >= 0）場合、余計な座標の基として0が使われます。

F.8.4. デフォルト

```
select cube_union('(0,5,2),(2,3,1)', '0');
cube_union
-----
(0, 0, 0),(2, 5, 2)
(1 row)
```

この和集合および以下の共通集合は一般常識と矛盾しないと思います。

```
select cube_inter('(0,-1),(1,1)', '(-2),(2)');
cube_inter
-----
(0, 0),(1, 0)
(1 row)
```

次元が異なる立体のバイナリ操作すべてにおいて、より低い次元の方がデカルト投影、つまり、文字列表現で省略された座標に0を持つものになると仮定します。 上の例は以下と同じです。

```
cube_union('(0,5,2),(2,3,1)', '(0,0,0),(0,0,0)');
cube_inter('(0,-1),(1,1)', '(-2,0),(2,0)');
```

以下の含有の述部は点構文を使用しますが、実際内部的には第2引数は矩形として表されます。この構文により、別の点用の型や(box,point)という述部用の関数を定義することが不要になります。

```
select cube_contains(' (0,0), (1,1)', '0.5,0.5');
cube_contains
-----
t
(1 row)
```

F.8.5. 注釈

使用例については、`sql/cube.sql` リグレーション試験を参照してください。

破壊防止のために立体の次元数に100までという制限を行いました。これは`cubedata.h`で設定されており、必要に応じて多少大きくすることができます。

F.8.6. クレジット

原作者: Gene Selkov, Jr. <selkovjr@mcs.anl.gov>, Mathematics and Computer Science Division, Argonne National Laboratory.

GiST (<http://gist.cs.berkeley.edu/>)の要点を説明していただいたJoe Hellerstein博士 (<http://db.cs.berkeley.edu/jmh/>)に感謝します。また、Illustra用の例 <http://best.berkeley.edu/~adong/rtree/index.html>の作者である彼の以前の生徒Andy Dong (<http://best.me.berkeley.edu/~adong/>)に感謝します。また、自分の世界を作成できるようにし、静かに生活できるようにしてもらった、過去から現在までのすべてのpostgres開発者に感謝します。データベース研究に長年誠実にサポートしてくれたArgonne LabとU.S. Department of Energyにも感謝します。

2002年8月/9月にBruno Wolff III <bruno@wolff.to>による小規模な改修がこのパッケージになされました。この改修には、単精度から倍精度への精度の変更といくつかの関数の追加が含まれます。

2006年7月にJoshua Reich <josh@root.net>による改修がさらになされました。この改修には`cube(float8[], float8[])`が含まれ、また、古いV0プロトコルからV1呼び出しプロトコルを使用するようコードが整理されました。

[前のページ](#)

citext

[ホーム](#)
[上に戻る](#)

[次のページ](#)

dblink

F.9. dblink

目次

[dblink_connect](#) -- リモートデータベースへの永続的な接続を開きます。
[dblink_connect_u](#) -- リモートデータベースへの永続的な危険な接続を開きます。
[dblink_disconnect](#) -- リモートデータベースへの永続的な接続を閉じます。
[dblink](#) -- リモートデータベースで問い合わせを実行します
[dblink_exec](#) -- リモートデータベースでコマンドを実行します
[dblink_open](#) -- リモートデータベースでカーソルを開きます
[dblink_fetch](#) -- リモートデータベースで開いているカーソルから行を取り出します
[dblink_close](#) -- リモートデータベースでカーソルを閉じます
[dblink_get_connections](#) -- 接続中の名前付きdblink接続すべての名前を返します
[dblink_error_message](#) -- 名前付き接続上の最後のエラーメッセージを入手します
[dblink_send_query](#) -- リモートデータベースに非同同期間問い合わせを送信します
[dblink_is_busy](#) -- 接続において非同同期間問い合わせが実行中か検査します
[dblink_get_notify](#) -- 接続上の非同同期間通知を取り出す
[dblink_get_result](#) -- 非同同期間問い合わせの結果を入手します
[dblink_cancel_query](#) -- 名前付き接続上の実行中の問い合わせをすべて取り消します。
[dblink_get_pkey](#) -- リレーションのプライマリキーフィールドの位置とフィールド名を返します
[dblink_build_sql_insert](#) -- ローカル側のタプルを使用し、プライマリキーフィールドの値を別の提供される値に置き換えてINSERT文を構築します。
[dblink_build_sql_delete](#) -- プライマリキーフィールドの値として提供された値を使用したDELETE文を構築します。
[dblink_build_sql_update](#) -- プライマリキーフィールドの値として提供された値を使用したUPDATE文を構築します。

dblinkは、データベースセッション内から他のPostgreSQLデータベースへの接続をサポートするモジュールです。

dblink_connect

名前

dblink_connect -- リモートデータベースへの永続的な接続を開きます。

概要

```
dblink_connect(text connstr) returns text
dblink_connect(text connname, text connstr) returns text
```

説明

`dblink_connect()` はリモートのPostgreSQLデータベースへの接続を確立します。 接続先のサーバとデータベースは標準のlibpq接続文字列を通して識別されます。 省略可能ですが、名前を接続に割り当てることも可能です。 複数の名前付きの接続を一度に開くことができますが、無名の接続は同時に1つしか許されません。 接続は、閉ざされるまで、または、データベースセッションが終わるまで永続します。

接続文字列は同時に既存の外部サーバ名であっても構いません。 対応する外部データラッパーを定義する場合、`postgresql_fdw_validator`を使用することを推奨します。 後述の例と以下の一覧を参照してください。 [CREATE FOREIGN DATA WRAPPER](#), [CREATE SERVER](#), [CREATE USER MAPPING](#)

引数

connname

接続に使用する名前です。 省略した場合、既存の無名の接続を閉ざし、無名の接続を開きます。

connstr

例えば`hostaddr=127.0.0.1 port=5432 dbname=mydb user=postgres password=mypasswd`といったlibpq形式の接続情報文字列です。 詳細は[項31.1](#)の `PQconnectdb`を参照してください。

戻り値

状態を返します。 これは常にOKです（何らかのエラーが起きるとこの関数は戻らずエラーとなるためです）。

注釈

スーパーユーザのみがパスワード認証がない接続を作成するために`dblink_connect`を使用することができます。 スーパーユーザ以外でこの機能が必要ならば`dblink_connect_u`を代わりに使用してください。

他の`dblink`関数内で接続情報文字列が混乱する危険が発生しますので、等号記号を含む接続名を選択することは勧めません。

例

```
SELECT dblink_connect('dbname=postgres');
 dblink_connect
-----
 OK
(1 row)

SELECT dblink_connect('myconn', 'dbname=postgres');
 dblink_connect
-----
 OK
(1 row)
```



```

-- 外部データラッパー (FOREIGN DATA WRAPPER) の機能
-- 注意:これが正常に機能するにはローカル接続にパスワード認証が必須です。
--      さもないと、dblink_connect():から以下のエラーを受け取ります。
--
-----
--      ERROR:  password is required
--      DETAIL:  Non-superuser cannot connect if the server does not request a password.
--      HINT:   Target server's authentication method must be changed.
CREATE USER dblink_regression_test WITH PASSWORD 'secret';
CREATE FOREIGN DATA WRAPPER postgresql VALIDATOR postgresql fdw_validator;
CREATE SERVER fdtest FOREIGN DATA WRAPPER postgresql OPTIONS (hostaddr '127.0.0.1', dbname
'contrib_regression');

CREATE USER MAPPING FOR dblink_regression_test SERVER fdtest OPTIONS (user 'dblink_regression_test', password
'secret');
GRANT USAGE ON FOREIGN SERVER fdtest TO dblink_regression_test;
GRANT SELECT ON TABLE foo TO dblink_regression_test;

¥set ORIGINAL_USER :USER
¥c - dblink_regression_test
SELECT dblink_connect('myconn', 'fdtest');
    dblink_connect
-----
    OK
(1 row)

SELECT * FROM dblink('myconn','SELECT * FROM foo') AS t(a int, b text, c text[]);
 a | b | c
-----+-----+-----
 0 | a | {a0,b0,c0}
 1 | b | {a1,b1,c1}
 2 | c | {a2,b2,c2}
 3 | d | {a3,b3,c3}
 4 | e | {a4,b4,c4}
 5 | f | {a5,b5,c5}
 6 | g | {a6,b6,c6}
 7 | h | {a7,b7,c7}
 8 | i | {a8,b8,c8}
 9 | j | {a9,b9,c9}
10 | k | {a10,b10,c10}
(11 rows)

¥c - :ORIGINAL_USER
REVOKE USAGE ON FOREIGN SERVER fdtest FROM dblink_regression_test;
REVOKE SELECT ON TABLE foo FROM dblink_regression_test;
DROP USER MAPPING FOR dblink_regression_test SERVER fdtest;
DROP USER dblink_regression_test;
DROP SERVER fdtest;
DROP FOREIGN DATA WRAPPER postgresql;

```

dblink_connect_u

名前

dblink_connect_u -- リモートデータベースへの永続的な危険な接続を開きます。

概要

```
dblink_connect_u(text connstr) returns text
dblink_connect_u(text connname, text connstr) returns text
```

説明

dblink_connect_u() は、非スーパーユーザが任意の認証方式を使用して接続することができる点を除き、dblink_connect() と同じです。

リモートサーバがパスワードを含まない認証方式を選択していた場合、セッションがローカルなPostgreSQLサーバを実行するユーザから構成されたものとなりますので、なりすましやその後の権限昇格が起こり得ます。また、リモートサーバがパスワードを要求したとしても、サーバ側のユーザに属する~/.pgpassファイルなどサーバの環境から提供されるパスワードになる可能性があります。これはなりすましの危険性だけでなく、信頼できないリモートサーバへのパスワードが漏れる可能性を引き起こします。このためdblink_connect_u() はまず、スーパーユーザ以外から呼び出すことができないように、PUBLICからすべての権限を取り除いた形でインストールされます。状況によっては、信頼できるとみなされた特定のユーザにdblink_connect_u() のEXECUTE権限を与えることが適切となる場合があります。しかしこれは注意して行わなければなりません。また、サーバのユーザに属する~/.pgpassファイルにはすべて、ホスト名としてワイルドカードを指定した項目をまったく含めないことを推奨します。

その他の情報はdblink_connect() を参照してください。

dblink_disconnect

名前

dblink_disconnect -- リモートデータベースへの永続的な接続を閉ざします。

概要

```
dblink_disconnect() returns text
dblink_disconnect(text connname) returns text
```

説明

dblink_disconnect() は dblink_connect() で開かれた既存の接続を閉ざします。 引数がない構文では無名の接続を閉ざします。

引数

connname

閉ざす名前付き接続の名前です。

戻り値

状態を返します。 これは常にOKです（何らかのエラーが起きるとこの関数は戻らずエラーとなるためです）。

例

```
SELECT dblink_disconnect();
 dblink_disconnect
-----
OK
(1 row)

SELECT dblink_disconnect('myconn');
 dblink_disconnect
-----
OK
(1 row)
```

dblink

名前

dblink -- リモートデータベースで問い合わせを実行します

概要

```
dblink(text connname, text sql [, bool fail_on_error]) returns setof record
dblink(text connstr, text sql [, bool fail_on_error]) returns setof record
dblink(text sql [, bool fail_on_error]) returns setof record
```

説明

`dblink`はリモートデータベースで問い合わせ（通常は`SELECT`ですが行を返す任意のSQLコマンドを行うことができます）を実行します。

2つの`text`型の引数が与えられた場合、一番目の引数はまず永続接続の名前を検索するために使われます。もし見つければ、コマンドがその接続上で実行されます。見つからなければ、一番目の引数は`dblink_connect`用の接続情報文字列として扱われ、このコマンド実行時と同様に指定された接続が開きます。

引数

`connname`

使用する接続の名前です。無名の接続を使用する場合はこのパラメータを省略します。

`connstr`

上で`dblink_connect`で説明した接続情報文字列です。

`sql`

例えば`select * from foo`といった、リモートデータベースで実行させるSQL問い合わせです。

`fail_on_error`

真（省略時のデフォルト）の場合、接続のリモート側で発生したエラーによりローカル側でもエラーが発生します。偽の場合リモート側のエラーはローカル側にはNOTICEとして報告され、この関数は行を返しません。

戻り値

この関数は問い合わせにより生成された行を返します。`dblink`は任意の問い合わせで使うことができますので、これは特定の列集合を指定するのではなく、`record`型を返すものと宣言されています。これは呼び出し元の問い合わせで想定列集合を指定しなければならないことを意味します。さもないとPostgreSQLは何が想定されているかわかりません。以下に例を示します。

```
SELECT *
FROM dblink('dbname=mydb', 'select pronomame, prosrc from pg_proc')
AS t1(pronomame name, prosrc text)
WHERE pronomame LIKE 'bytea%';
```

`FROM`句の"別名"部分は関数が返す列名とその型を指定しなければなりません。（別名内の列名の指定はSQL標準の構文ですが、列の型の指定はPostgreSQLの拡張です。）これによりシステムは、関数を実行する前に、*がどのように展開されるか、`WHERE`句内の`pronomame`が何を参照するかを理解します。実行時、リモートデータベースから返る実際の問い合わせの結果が`FROM`句で示された列数と異なる場合エラーが発生します。しかし、列名は一致する必要はありません。また、`dblink`は正確な型一致も強制しません。返されるデータ文字列が`FROM`句で宣言された列型の有効な入力である限り成功します。

注釈

前もって判明している問い合わせをdblinkで使用する簡便な方法はビューを作成することです。これにより問い合わせの度に列型の情報を記載することなく、ビュー内に隠すことができます。以下に例を示します。

```
CREATE VIEW myremote_pg_proc AS
SELECT *
FROM dblink('dbname=postgres', 'select proname, prosrc from pg_proc')
AS t1(proname name, prosrc text);

SELECT * FROM myremote_pg_proc WHERE proname LIKE 'bytea%';
```

例

```
SELECT * FROM dblink('dbname=postgres', 'select proname, prosrc from pg_proc')
AS t1(proname name, prosrc text) WHERE proname LIKE 'bytea%';
proname | prosrc
-----+-----
byteacat | byteacat
byteaeq | byteaeq
bytealt | bytealt
byteale | byteale
byteagt | byteagt
byteage | byteage
byteane | byteane
byteacmp | byteacmp
bytealike | bytealike
byteanlike | byteanlike
byteain | byteain
byteaout | byteaout
(12 rows)
```

```
SELECT dblink_connect('dbname=postgres');
dblink_connect
-----
OK
(1 row)
```

```
SELECT * FROM dblink('select proname, prosrc from pg_proc')
AS t1(proname name, prosrc text) WHERE proname LIKE 'bytea%';
proname | prosrc
-----+-----
byteacat | byteacat
byteaeq | byteaeq
bytealt | bytealt
byteale | byteale
byteagt | byteagt
byteage | byteage
byteane | byteane
byteacmp | byteacmp
bytealike | bytealike
byteanlike | byteanlike
byteain | byteain
byteaout | byteaout
(12 rows)
```

```
SELECT dblink_connect('myconn', 'dbname=regression');
dblink_connect
-----
OK
(1 row)
```

```
SELECT * FROM dblink('myconn', 'select proname, prosrc from pg_proc')
AS t1(proname name, prosrc text) WHERE proname LIKE 'bytea%';
proname | prosrc
-----+-----
bytearecv | bytearecv
byteasend | byteasend
byteale | byteale
byteagt | byteagt
byteage | byteage
byteane | byteane
byteacmp | byteacmp
bytealike | bytealike
byteanlike | byteanlike
byteacat | byteacat
byteaeq | byteaeq
bytealt | bytealt
byteain | byteain
byteaout | byteaout
(14 rows)
```

dblink_exec

名前

dblink_exec -- リモートデータベースでコマンドを実行します

概要

```
dblink_exec(text connname, text sql [, bool fail_on_error]) returns text
dblink_exec(text connstr, text sql [, bool fail_on_error]) returns text
dblink_exec(text sql [, bool fail_on_error]) returns text
```

説明

dblink_exec はリモートデータベースでコマンド（つまり行を返さない任意のSQL文）を実行します。

2つのtext型の引数が与えられた場合、一番目の引数はまず永続接続の名前を検索するために使われます。もし見つければ、コマンドがその接続上で実行されます。見つからなければ、一番目の引数はdblink_connect用の接続情報文字列として扱われ、このコマンド実行時と同様に指定された接続が開きます。

引数

connname

使用する接続の名前です。無名の接続を使用する場合はこのパラメータを省略します。

connstr

上でdblink_connectで説明した接続情報文字列です。

sql

例えばinsert into foo values(0,'a',{'a0","b0","c0"})といった、リモートデータベースで実行させるSQL問い合わせです。

fail_on_error

真（省略時のデフォルト）の場合、接続のリモート側で発生したエラーによりローカル側でもエラーが発生します。偽の場合リモート側のエラーはローカル側にはNOTICEとして報告され、この関数の戻り値はERRORになります。

戻り値

状態、つまりコマンドの状態またはERRORを返します。

例

```
SELECT dblink_connect('dbname=dblink_test_standby');
dblink_connect
-----
OK
(1 row)

SELECT dblink_exec('insert into foo values(21, 'z', '{a0","b0","c0"}');');
dblink_exec
-----
INSERT 943366 1
(1 row)

SELECT dblink_connect('myconn', 'dbname=regression');
dblink_connect
-----
OK
```

```
(1 row)

SELECT dblink_exec('myconn', 'insert into foo values(21,'z','{"a0","b0","c0"}');');
      dblink_exec
-----
INSERT 6432584 1
(1 row)

SELECT dblink_exec('myconn', 'insert into pg_class values ('foo'),false);
NOTICE:  sql error
DETAIL:  ERROR:  null value in column "relnamespace" violates not-null constraint

      dblink_exec
-----
ERROR
(1 row)
```

[前のページ](#)
dblink

[ホーム](#)
[上に戻る](#)

[次のページ](#)
dblink_open

dblink_open

名前

dblink_open -- リモートデータベースでカーソルを開きます

概要

```
dblink_open(text cursorname, text sql [, bool fail_on_error]) returns text
dblink_open(text connname, text cursorname, text sql [, bool fail_on_error]) returns text
```

説明

dblink_open() はリモートデータベースでカーソルを開きます。その後カーソルをdblink_fetch() と dblink_close() で操作することができます。

引数

connname

使用する接続の名前です。無名の接続を使用する場合はこのパラメータを省略します。

cursorname

このカーソルに割り当てる名前です。

sql

例えばselect * from pg_classといった、リモートデータベースで実行させたいSELECT文です。

fail_on_error

真（省略時のデフォルト）の場合、接続のリモート側で発生したエラーによりローカル側でもエラーが発生します。偽の場合リモート側のエラーはローカル側にはNOTICEとして報告され、この関数の戻り値はERRORになります。

戻り値

状態、つまりOKまたはERRORを返します。

注釈

カーソルはトランザクション内でのみ持続することができますので、リモート側がまだトランザクションの内部でない場合、dblink_openはリモート側で明示的なトランザクションブロックを開始（BEGIN）します。このトランザクションは対応するdblink_closeが実行された時に同様に閉ざされます。dblink_openとdblink_closeの間にdblink_execを使用してデータを変更した場合、エラーが発生することに注意してください。また、dblink_closeの前にdblink_disconnectを使用すると、トランザクションがアボートしますので変更が失われることに注意してください。

例

```
SELECT dblink_connect('dbname=postgres');
 dblink_connect
-----
 OK
(1 row)

SELECT dblink_open('foo', 'select proname, prosrc from pg_proc');
 dblink_open
-----
```


dblink_fetch

名前

dblink_fetch -- リモートデータベースで開いているカーソルから行を取り出します

概要

```
dblink_fetch(text cursorname, int howmany [, bool fail_on_error]) returns setof record
dblink_fetch(text connname, text cursorname, int howmany [, bool fail_on_error]) returns setof record
```

説明

dblink_fetchはdblink_openによりあらかじめ確立したカーソルから行を取り出します。

引数

connname

使用する接続の名前です。 無名の接続を使用する場合はこのパラメータを省略します。

cursorname

行を取り出すカーソルの名前です。

howmany

受け取る行の最大数です。 カーソルの現在位置から次のhowmany行を取り出し、カーソルの位置を前方に移動します。 カーソルが終端まで達すると、これ以上の行は生成されません。

fail_on_error

真（省略時のデフォルト）の場合、接続のリモート側で発生したエラーによりローカル側でもエラーが発生します。 偽の場合リモート側のエラーはローカル側にはNOTICEとして報告され、この関数は行を返しません。

戻り値

この関数はカーソルから取り出された行を返します。 この関数を使用するためには、dblinkで説明したように、想定する列集合を指定する必要があります。

注釈

リモートカーソルから返る実際の列数とFROM句で指定された列数と異なる場合エラーが発生します。 この場合リモート側のカーソルは、エラーが発生しなかった場合と同じ行数分位置が変わります。 リモート側のFETCHが完了した後にローカル側でこの他のエラーが発生した場合も同じです。

例

```
SELECT dblink_connect('dbname=postgres');
dblink_connect
-----
OK
(1 row)

SELECT dblink_open('foo', 'select proname, prosrc from pg_proc where proname like ''bytea%'');
dblink_open
-----
OK
(1 row)
```

```

SELECT * FROM dblink_fetch('foo', 5) AS (funcname name, source text);
  funcname | source
-----+-----
 byteacat  | byteacat
 byteacmp  | byteacmp
 byteaeq   | byteaeq
 byteage   | byteage
 byteagt   | byteagt
(5 rows)

```

```

SELECT * FROM dblink_fetch('foo', 5) AS (funcname name, source text);
  funcname | source
-----+-----
 byteain   | byteain
 byteale   | byteale
 bytealike | bytealike
 bytealt   | bytealt
 byteane   | byteane
(5 rows)

```

```

SELECT * FROM dblink_fetch('foo', 5) AS (funcname name, source text);
  funcname | source
-----+-----
 byteanlike | byteanlike
 byteaout   | byteaout
(2 rows)

```

```

SELECT * FROM dblink_fetch('foo', 5) AS (funcname name, source text);
  funcname | source
-----+-----
(0 rows)

```

[前のページ](#)
dblink_open

[ホーム](#)
[上に戻る](#)

[次のページ](#)
dblink_close

dblink_close

名前

dblink_close -- リモートデータベースでカーソルを閉ざします

概要

```
dblink_close(text cursorname [, bool fail_on_error]) returns text
dblink_close(text connname, text cursorname [, bool fail_on_error]) returns text
```

説明

dblink_closeは前もってdblink_openで開かれたカーソルを閉ざします。

引数

connname

使用する接続の名前です。 無名の接続を使用する場合はこのパラメータを省略します。

cursorname

閉ざすカーソルの名前です。

fail_on_error

真（省略時のデフォルト）の場合、接続のリモート側で発生したエラーによりローカル側でもエラーが発生します。 偽の場合リモート側のエラーはローカル側にはNOTICEとして報告され、この関数の戻り値はERRORになります。

戻り値

状態、つまりOKまたはERRORを返します。

注釈

dblink_openが明示的なトランザクションブロックを開始し、これが接続上で最後まで開き続けているカーソルであった場合、dblink_closeは対応するCOMMITを発行します。

例

```
SELECT dblink_connect('dbname=postgres');
 dblink_connect
-----
OK
(1 row)

SELECT dblink_open('foo', 'select proname, prosrc from pg_proc');
 dblink_open
-----
OK
(1 row)

SELECT dblink_close('foo');
 dblink_close
-----
OK
(1 row)
```


dblink_get_connections

名前

dblink_get_connections -- 接続中の名前付きdblink接続すべての名前を返します

概要

```
dblink_get_connections() returns text[]
```

説明

dblink_get_connections は、接続中の名前付きのdblink接続すべての名前を配列として返します。

戻り値

接続名のテキスト型配列を返します。なければNULLです。

例

```
SELECT dblink_get_connections();
```


dblink_error_message

名前

dblink_error_message -- 名前付き接続上の最後のエラーメッセージを入手します

概要

```
dblink_error_message(text connname) returns text
```

説明

dblink_error_message は指定された接続における、最後のリモートエラーメッセージを取り出します。

引数

connname

使用する接続名です。

戻り値

最後のエラーメッセージを返します。 接続においてエラーが存在しなかった場合は空文字列を返します。

例

```
SELECT dblink_error_message('dtest1');
```

dblink_send_query

名前

`dblink_send_query` -- リモートデータベースに非同同期間い合わせを送信します

概要

```
dblink_send_query(text connname, text sql) returns int
```

説明

`dblink_send_query` は非同同期に、つまり、結果をすぐに待機することなく実行する問い合わせを送信します。 接続上で進行中の非同同期間い合わせが存在してはなりません。

非同同期間い合わせの登録が成功した後、`dblink_is_busy` を使用して完了状況を検査することができます。 そして最後に、`dblink_get_result` を使用して結果を収集します。 また、`dblink_cancel_query` を使用して実行中の非同同期間い合わせを取り消すことができます。

引数

`connname`

使用する接続名です。

`sql`

例えば `select * from pg_class` といった、リモートデータベースで実行させたいSQL文です。

戻り値

問い合わせの登録に成功した場合1を返します。 失敗した場合は0を返します。

例

```
SELECT dblink_send_query('dtest1', 'SELECT * FROM foo WHERE f1 < 3');
```

dblink_is_busy

名前

dblink_is_busy -- 接続において非同同期間い合わせが実行中か検査します

概要

```
dblink_is_busy(text connname) returns int
```

説明

dblink_is_busy は非同同期間い合わせが進行中かどうか試験します。

引数

connname

検査対象の接続名です。

戻り値

接続において進行中だった場合は1、さもなければ0を返します。この関数が0を返した場合、dblink_get_result がブロックされないことが保証されます。

例

```
SELECT dblink_is_busy('dtest1');
```

dblink_get_notify

名前

dblink_get_notify -- 接続上の非同期通知を取り出す

概要

```
dblink_get_notify() returns setof (notify_name text, be_pid int, extra text)
dblink_get_notify(text connname) returns setof (notify_name text, be_pid int, extra text)
```

説明

dblink_get_notifyは名前の付いていない接続、または、もし指定されて名前が付いている接続いずれからも通知を取り出します。 dblink経由で通知を受け取るには、 dblink_execを使用してLISTENを最初に発行しなければなりません。 詳細は[LISTEN](#)と[NOTIFY](#)を参照ください。

引数

connname

通知を受け取る名前付きの接続の名前

戻り値

setof (notify_name text, be_pid int, extra text) または存在しない場合は空集合を返します。

例

```
SELECT dblink_exec('LISTEN virtual');
 dblink_exec
-----
LISTEN
(1 row)

SELECT * FROM dblink_get_notify();
 notify_name | be_pid | extra
-----+-----+-----
(0 rows)

NOTIFY virtual;
NOTIFY

SELECT * FROM dblink_get_notify();
 notify_name | be_pid | extra
-----+-----+-----
virtual      |    1229 |
(1 row)
```

dblink_get_result

名前

dblink_get_result -- 非同同期間い合わせの結果を入手します

概要

```
dblink_get_result(text connname [, bool fail_on_error]) returns setof record
```

説明

dblink_get_resultは、事前にdblink_send_queryで送信された非同同期間い合わせの結果を収集します。 問い合わせがまだ完了していなかった場合、dblink_get_resultは終わるまで待機します。

引数

connname

使用する接続名です。

fail_on_error

真（省略時のデフォルト）の場合、接続のリモート側で発生したエラーによりローカル側でもエラーが発生します。 偽の場合リモート側のエラーはローカル側にはNOTICEとして報告され、この関数は行を返しません。

戻り値

非同同期間い合わせ（行を返すSQL文の場合）について、この関数は問い合わせで生成された行を返します。 この関数を使用するためには、上のdblinkで説明したように想定する列集合を指定する必要があります。

非同同期コマンド（行を返さないSQL文の場合）について、この関数はコマンド状態文字列からなるテキスト列を1つ持つ1行を返します。 この場合も呼び出し元のFROM句で結果が単一のテキスト列を持つことを指定する必要があります。

注釈

dblink_send_queryが1を返した場合にこの関数を呼び出さなければなりません。 接続を再度利用できるようになる前に、送信した問い合わせに対し一度呼び出されなければなりません。 もう一度実行すると空の結果集合を得ることになります。

dblink_send_queryとdblink_get_resultを使う場合には、dblinkはリモート側の問い合わせ結果をローカルの問い合わせ処理に渡す前にすべて取り込みます。 問い合わせが大量の行を返す場合、ローカルセッションで一時的なメモリ膨張が起こるかも知れません。 そのような問い合わせはdblink_openでカーソルとして開き、それから一度に管理可能な行数を取り出す方が良いでしょう。 あるいは、普通のdblink()を使って下さい。 大きな結果集合をディスクにスプールすることでメモリ膨張を回避します。

例

```
contrib_regression=# SELECT dblink_connect('dtest1', 'dbname=contrib_regression');
dblink_connect
-----
OK
(1 row)

contrib_regression=# SELECT * FROM
contrib_regression-# dblink_send_query('dtest1', 'select * from foo where f1 < 3') AS t1;
t1
----
```

```

1
(1 row)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2 text, f3 text[]);
 f1 | f2 |      f3
-----+-----
  0 | a  | {a0,b0,c0}
  1 | b  | {a1,b1,c1}
  2 | c  | {a2,b2,c2}
(3 rows)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2 text, f3 text[]);
 f1 | f2 | f3
-----+-----
(0 rows)

contrib_regression=# SELECT * FROM
contrib_regression-# dblink_send_query('dtest1', 'select * from foo where f1 < 3; select * from foo where f1
> 6') AS t1;
 t1
----
  1
(1 row)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2 text, f3 text[]);
 f1 | f2 |      f3
-----+-----
  0 | a  | {a0,b0,c0}
  1 | b  | {a1,b1,c1}
  2 | c  | {a2,b2,c2}
(3 rows)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2 text, f3 text[]);
 f1 | f2 |      f3
-----+-----
  7 | h  | {a7,b7,c7}
  8 | i  | {a8,b8,c8}
  9 | j  | {a9,b9,c9}
 10 | k  | {a10,b10,c10}
(4 rows)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2 text, f3 text[]);
 f1 | f2 | f3
-----+-----
(0 rows)

```

[前のページ](#)

dblink_get_notify

[ホーム](#)
[上に戻る](#)

[次のページ](#)

dblink_cancel_query

dblink_cancel_query

名前

dblink_cancel_query -- 名前付き接続上の実行中の問い合わせをすべて取り消します。

概要

```
dblink_cancel_query(text connname) returns text
```

説明

`dblink_cancel_query` は名前付き接続上で進行中の問い合わせをすべて取り消そうとします。これは成功するとは限らないことに注意してください（例えばリモート問い合わせがすでに終わっているかもしれないからです）。取り消し要求は単に問い合わせがすぐに失敗する可能性を大きくするだけです。例えば、`dblink_get_result` を呼び出すなど、通常の問い合わせ手順を完了させる必要があります。

引数

connname

使用する接続名です。

戻り値

取り消し要求が送信された場合 `OK` を、さもなければ失敗についてのエラーメッセージテキストを返します。

例

```
SELECT dblink_cancel_query('dtest1');
```


dblink_get_pkey

名前

dblink_get_pkey -- リレーシヨンのプライマリキーフィールドの位置とフィールド名を返します

概要

```
dblink_get_pkey(text relname) returns setof dblink_pkey_results
```

説明

dblink_get_pkeyは、ローカルデータベース内のリレーシヨンのプライマリキーに関する情報を提供します。これはリモートデータベースに送信する問い合わせを生成する際に役に立つことがあります。

引数

relname

例えばfooやmyschema.mytabといった、ローカル側のリレーシヨンの名前です。例えば"FooBar"のように名前に大文字小文字が混在する場合や特殊文字が含まれる場合は二重引用符で括ってください。引用符がないと文字列は小文字に変換されます。

戻り値

プライマリキー毎に1行を返します。リレーシヨンがプライマリキーを持たない場合は行は返されません。結果の行型は以下のよう定義されます。

```
CREATE TYPE dblink_pkey_results AS (position int, colname text);
```

position列は単に1からNを返します。それは、主キー内にあるフィールドの数で、テーブルの列内にある数ではありません。

例

```
CREATE TABLE foobar (  
    f1 int,  
    f2 int,  
    f3 int,  
    PRIMARY KEY (f1, f2, f3)  
);  
CREATE TABLE  
  
SELECT * FROM dblink_get_pkey('foobar');  
 position | colname  
-----+-----  
         1 | f1  
         2 | f2  
         3 | f3  
(3 rows)
```

dblink_build_sql_insert

名前

`dblink_build_sql_insert` -- ローカル側のタプルを使用し、プライマリキーフィールドの値を別の提供される値に置き換えてINSERT文を構築します。

概要

```
dblink_build_sql_insert(text relname,
                        int2vector primary_key_attnums,
                        integer num_primary_key_atts,
                        text[] src_pk_att_vals_array,
                        text[] tgt_pk_att_vals_array) returns text
```

説明

`dblink_build_sql_insert`はローカル側のテーブルの一部を選択した複製をリモートデータベースに行う場合に有用になる可能性があります。これはプライマリキーによりローカルテーブルから行を選択し、そのプライマリキー値を最後の引数で与えた値に置き換えて、行を複製するINSERT SQLコマンドを構築します。（行をそのまま複製する場合は、単に最後の2つの引数に同じ値を指定してください。）

引数

`relname`

例えば`foo`や`myschema.mytab`といったローカル側のリレーションの名前です。例えば`"FooBar"`のように名前に大文字小文字が混在する場合や特殊文字が含まれる場合は二重引用符で括ってください。引用符がないと文字列は小文字に変換されます。

`primary_key_attnums`

例えば`1 2`といった、プライマリキーフィールドの属性番号（1始まり）です。

`num_primary_key_atts`

プライマリキーフィールドの個数です。

`src_pk_att_vals_array`

ローカルタプルを検索するために使用されるプライマリキーフィールドの値です。各フィールドはテキスト形式で表されます。これらのプライマリキーの値を持つ行がローカル側に存在しない場合はエラーが発生します。

`tgt_pk_att_vals_array`

最終的なINSERTコマンドにおいて置き換えられるプライマリキーフィールドの値です。各フィールドはテキスト形式で表されます。

戻り値

要求したSQL文をテキストとして返します。

注釈

PostgreSQL 9.0の段階で、`primary_key_attnums`の中の属性数は、`SELECT * FROM relname`内の列の位置に対応する、論理的列数として翻訳されます。以前のバージョンは物理的な列の位置として数を翻訳しました。テーブルの存続期間中に、表示された列の左側のどんな列でも削除されると差異が生じます。

例

```
SELECT dblink_build_sql_insert('foo', '1 2', 2, '{"1", "a"}', '{"1", "b''a"}');
      dblink_build_sql_insert
-----
INSERT INTO foo(f1,f2,f3) VALUES('1','b''a','1')
(1 row)
```

[前のページ](#)

dblink_get_pkey

[ホーム](#)

[上に戻る](#)

[次のページ](#)

dblink_build_sql_delete

dblink_build_sql_delete

名前

dblink_build_sql_delete -- プライマリキーフィールドの値として提供された値を使用したDELETE文を構築します。

概要

```
dblink_build_sql_delete(text relname,
                        int2vector primary_key_attnums,
                        integer num_primary_key_atts,
                        text[] tgt_pk_att_vals_array) returns text
```

説明

dblink_build_sql_deleteはローカル側のテーブルの一部を選択した複製をリモートデータベースに行う場合に有用になる可能性があります。これは指定したプライマリキーの値を持つ行を削除するDELETE SQLコマンドを構築します。

引数

relname

例えばfooやmyschema.mytabといったローカル側のリレーションの名前です。例えば"FooBar"のように名前に大文字小文字が混在する場合や特殊文字が含まれる場合は二重引用符で括ってください。引用符がないと文字列は小文字に変換されます。

primary_key_attnums

例えば1 2といった、プライマリキーフィールドの属性番号（1始まり）です。

num_primary_key_atts

プライマリキーフィールドの個数です。

tgt_pk_att_vals_array

最終的なDELETEコマンドにおいて使用されるプライマリキーフィールドの値です。各フィールドはテキスト形式で表されます。

戻り値

要求したSQL文をテキストとして返します。

注釈

PostgreSQL 9.0の段階で、primary_key_attnumsの中の属性数は、SELECT * FROM relname内の列の位置に対応する、論理的列数として翻訳されます。以前のバージョンは物理的な列の位置として数を翻訳しました。テーブルの存続期間中に、表示された列の左側のどんな列でも削除されると差異が生じます。

例

```
SELECT dblink_build_sql_delete('"MyFoo"', '1 2', 2, '{"1", "b"}');
        dblink_build_sql_delete
-----
DELETE FROM "MyFoo" WHERE f1='1' AND f2='b'
(1 row)
```


dblink_build_sql_update

名前

dblink_build_sql_update -- プライマリキーフィールドの値として提供された値を使用したUPDATE文を構築します。

概要

```
dblink_build_sql_update(text relname,
                        int2vector primary_key_attnums,
                        integer num_primary_key_atts,
                        text[] src_pk_att_vals_array,
                        text[] tgt_pk_att_vals_array) returns text
```

説明

dblink_build_sql_updateはローカル側のテーブルの一部を選択した複製をリモートデータベースに行う場合に有用になる可能性があります。これはプライマリキーによりローカルテーブルから行を選択し、そのプライマリキー値を最後の引数で与えた値に置き換えて、行を複製するUPDATE SQLコマンドを構築します。（行をそのまま複製する場合は、単に最後の2つの引数に同じ値を指定してください。）このUPDATE コマンドは常に行のすべてのフィールドを代入します。この関数とdblink_build_sql_insertの主な違いは、対象の行がリモート側のテーブルにすでに存在すると仮定している点です。

引数

relname

例えばfooやmyschema.mytabといったローカル側のリレーションの名前です。例えば"FooBar"のように名前に大文字小文字が混在する場合や特殊文字が含まれる場合は二重引用符で括ってください。引用符がないと文字列は小文字に変換されます。

primary_key_attnums

例えば1 2といった、プライマリキーフィールドの属性番号（1始まり）です。

num_primary_key_atts

プライマリキーフィールドの個数です。

src_pk_att_vals_array

ローカルタプルを検索するために使用されるプライマリキーフィールドの値です。各フィールドはテキスト形式で表されます。これらのプライマリキーの値を持つ行がローカル側に存在しない場合はエラーが発生します。

tgt_pk_att_vals_array

最終的なUPDATE コマンドにおいて置き換えられるプライマリキーフィールドの値です。各フィールドはテキスト形式で表されます。

戻り値

要求したSQL文をテキストとして返します。

注釈

PostgreSQL 9.0の段階で、primary_key_attnumsの中の属性数は、SELECT * FROM relname内の列の位置に対応する、論理的列数として翻訳されます。以前のバージョンは物理的な列の位置として数を翻訳しました。テーブルの存続期間中に、表示された列の左側のどんな列でも削除されると差異が生じます。

例

```
SELECT dblink_build_sql_update('foo', '1 2', 2, '{"1", "a"}', '{"1", "b"}');
      dblink_build_sql_update
-----
UPDATE foo SET f1='1',f2='b',f3='1' WHERE f1='1' AND f2='b'
(1 row)
```

[前のページ](#)

dblink_build_sql_delete

[ホーム](#)
[上に戻る](#)

[次のページ](#)

dict_int

F.10. dict_int

`dict_int` は、全文検索用の辞書テンプレートの追加例です。この辞書例の目的は、検索性能に大きく影響する一意な単語数の急激な増大を防ぎながら、こうした数のインデックス付けを行うことができるように、整数（符号付きおよび符号無）のインデックス付けを制御することです。

F.10.1. 設定

この辞書は2つのオプションを受け付けます。

- `maxlen` パラメータは整数型の単語で許される最大桁数を指定します。デフォルト値は6です。
- `rejectlong` パラメータは、桁数を超える整数を切り詰めるか無視するかを指定します。`rejectlong` が `false`（デフォルト）ならば、辞書は整数の先頭の `maxlen` 桁を返します。`rejectlong` が `true` ならば、辞書は桁数を超えた整数をストップワードとして扱います。このためインデックス付けされません。これはまた、こうした整数を検索することができないことを意味します。

F.10.2. 使用方法

`dict_int` 拡張機能をインストールすると、`intdict_template` テキスト検索テンプレートとこれに基づき、そのデフォルト値で `intdict` 辞書が作成されます。以下のようにパラメータを変更することができます。

```
mydb# ALTER TEXT SEARCH DICTIONARY intdict (MAXLEN = 4, REJECTLONG = true);
ALTER TEXT SEARCH DICTIONARY
```

または、このテンプレートを基に新しい辞書を作成してください。

辞書を試験するためには以下を試してください。

```
mydb# select ts_lexize('intdict', '12345678');
 ts_lexize
-----
 {123456}
```

しかし、現実世界で使用する場合は、[第12章](#)で説明されるテキスト検索設定内にこれを含むようになるでしょう。以下のようになります。

```
ALTER TEXT SEARCH CONFIGURATION english
ALTER MAPPING FOR int, uint WITH intdict;
```

F.11. dict_xsyn

`dict_xsyn` (拡張類義語辞書) は全文検索用の辞書テンプレートの追加例です。この種類の辞書は、単語を類義語の集まりに置き換え、その類義語のいずれかを使用して単語を検索できるようにします。

F.11.1. 設定

`dict_xsyn` 辞書は以下のオプションを受け付けます。

- `matchorig` は辞書で元の単語が受け付けられるか否かを制御します。デフォルトは `true` です。
- `matchsynonyms` は類義語が辞書で受け付けられるか否かを制御します。デフォルトは `false` です。
- `keeporig` は元の単語を辞書出力に含めるか否かを制御します。デフォルトは `true` です。
- `keepsynonyms` は類義語が辞書出力に含められるか否かを制御します。デフォルトは `true` です。
- `rules` は、類義語リストを含むファイルのベース名です。このファイルは `$SHAREDIR/tsearch_data/` (`$SHAREDIR` は PostgreSQL インストールの共有データ用ディレクトリを示します) に格納しなければなりません。この名前は `.rules` で終わらなければなりません (これは `rules` パラメータには含まれません)。

`rules` ファイルは以下の書式です。

- 各行は、行の先頭で与えられる1つの単語に対する類義語の集まりを表します。類義語は以下のように空白文字で区切られます。

```
word syn1 syn2 syn3
```

- シャープ記号 (#) はコメント区切り記号です。行の任意の位置に記載することができます。行の残りの部分は飛ばされます。

例として `$SHAREDIR/tsearch_data/` にインストールされる `xsyn_sample.rules` を参照してください。

F.11.2. 使用方法

`dict_xsyn` 拡張機能をインストールすると、`xsyn_template` テキスト検索テンプレートが作成され、それに基づき、デフォルトのパラメータを持った `xsyn` 辞書が作成されます。例えば以下のように、パラメータを変更することができます。

```
mydb# ALTER TEXT SEARCH DICTIONARY xsyn (RULES='my_rules', KEEPORIG=false);
ALTER TEXT SEARCH DICTIONARY
```

またこのテンプレートに基づいた新しい辞書を作成することもできます。

辞書を試験するためには以下を試してください。

```
mydb=# SELECT ts_lexize('xsyn', 'word');
         ts_lexize
-----
{syn1,syn2,syn3}

mydb# ALTER TEXT SEARCH DICTIONARY xsyn (RULES='my_rules', KEEPORIG=true);
ALTER TEXT SEARCH DICTIONARY

mydb=# SELECT ts_lexize('xsyn', 'word');
         ts_lexize
-----
{word,syn1,syn2,syn3}

mydb# ALTER TEXT SEARCH DICTIONARY xsyn (RULES='my_rules', KEEPORIG=false, MATCHSYNONYMS=true);
ALTER TEXT SEARCH DICTIONARY

mydb=# SELECT ts_lexize('xsyn', 'syn1');
         ts_lexize
-----
{syn1,syn2,syn3}

mydb# ALTER TEXT SEARCH DICTIONARY xsyn (RULES='my_rules', KEEPORIG=true, MATCHORIG=false,
KEEPSYNONYMS=false);
```

```
ALTER TEXT SEARCH DICTIONARY

mydb=# SELECT ts_lexize('xsyn', 'syn1');
       ts_lexize
-----
{word}
```

現実世界で使用する場合は、[第12章](#)で説明されるテキスト検索設定内にこれを含むようになるでしょう。 以下のようになります。

```
ALTER TEXT SEARCH CONFIGURATION english
ALTER MAPPING FOR word, asciiword WITH xsyn, english_stem;
```

[前のページ](#)
dict_int

[ホーム](#)
[上に戻る](#)

[次のページ](#)
dummy_seclabel

F.12. dummy_seclabel

dummy_seclabel モジュールは SECURITY LABEL 文のリグレーション試験をサポートするためだけに存在します。実運用においての使用を意図したものではありません。

F.12.1. 原理的な説明

SECURITY LABEL 文によりユーザはデータベースオブジェクトにセキュリティラベルを割り当てることができます。しかしセキュリティラベルはロード可能なモジュールにより特別に許された時のみ割り当てることができます。このためこのモジュールは適切にリグレーション試験ができるように提供されています。

実運用で使用されることを意図したセキュリティラベルプロバイダは通常、SE-Linuxなどのプラットフォームに特化した機能に依存します。このモジュールはプラットフォームに依存しませんので、リグレーション試験により適切なものです。

F.12.2. 使用法

ここで使用方法の簡単な例を示します。

```
# postgresql.conf
shared_preload_libraries = 'dummy_seclabel'

postgres=# CREATE TABLE t (a int, b text);
CREATE TABLE
postgres=# SECURITY LABEL ON TABLE t IS 'classified';
SECURITY LABEL
```

dummy_seclabel モジュールは、unclassified、classified、secret、top secret という 4 つの固定のラベルのみを提供します。この他の文字列はすべてセキュリティラベルとして許されません。

これらのラベルはアクセス制御を強制するために使用されません。これらは SECURITY LABEL 文が想定通りに動作するかどうかを検査するためだけに使用されます。

F.12.3. 作者

KaiGai Kohei <kaigai@ak.jp.nec.com>

F.13. earthdistance

`earthdistance`は地球上の大圏距離を計算する、2つの異なる方式を提供します。最初に説明する方式は`cube`モジュールに依存します。（これは`earthdistance`をインストールする前にインストールしなければなりません。）2番目の方式は、座標系として緯度経度を使用した、組み込みの`point`データ型を基にしたものです。

このモジュールでは地球は完全な球体であると仮定します。（この精度が不十分な場合は、[PostGIS](#)プロジェクトを参照することを勧めます。）

F.13.1. Cubeを基にした地表距離

地球中心からのx、y、z距離をあらわす3次元を使用した点（両隅が同じ）である`cube`として、データは格納されます。`cube`上に`earth`ドメインが提供されます。これには、値がこれら制限を満たすか、また値が理論的に実際の地球表面に近いかどうかの整合性検査を含みます。

地球の半径は`earth()`関数から入手されます。この単位はメートルです。しかしこの1つの関数を変更することで、何らかの他の単位を使用するようにしたり、より適切と考える別の半径を使用したりするようにこのモジュールを変更することができます。

このパッケージは天文学データベースへの応用もあります。天文学者はおそらく`earth()`が度単位の距離になるように`180/pi()`を返すものと変更したいでしょう。

緯度経度（度単位）の入力をサポート、緯度経度の出力をサポート、2点間の大圏距離を計算、インデックス検索に使用可能な簡単に境界矩形を指定するための関数が提供されます。

提供されている関数は[表F-4](#)に示されています。

表 F-4. Cubeを基にしたEarthdistanceの関数

関数	戻り値	説明
<code>earth()</code>	<code>float8</code>	地球の想定半径を返します。
<code>sec_to_gc(float8)</code>	<code>float8</code>	地表の2点間の通常の直線（割線）距離を大圏距離に変換します。
<code>gc_to_sec(float8)</code>	<code>float8</code>	地表の2点間の大圏距離を通常の直線（割線）距離に変換します。
<code>ll_to_earth(float8, float8)</code>	<code>earth</code>	度単位で指定された緯度（第1引数）と経度（第2引数）に対する地表位置を返します。
<code>latitude(earth)</code>	<code>float8</code>	地表上の点の緯度を度単位で返します。
<code>longitude(earth)</code>	<code>float8</code>	地表上の点の経度を度単位で返します。
<code>earth_distance(earth, earth)</code>	<code>float8</code>	地表上の2点間の大圏距離を返します。
<code>earth_box(earth, float8)</code>	<code>cube</code>	位置から指定した大圏距離内の点に対する <code>cube</code> の <code>@></code> 演算子を使用するインデックス検索に適した矩形を返します。矩形内の点の一部は指定した大圏距離の外部にあります。このため、 <code>earth_distance</code> を使用した第2の検査を問い合わせに含めなければなりません。

F.13.2. Pointを基にした地表距離

このモジュールの第2部分は`point`型の値として地球上の位置を表現することに依存します。ここで第1要素は経度を度単位で、第2要素は緯度を度単位で表現されていると見なします。直感的に経度はX軸、緯度はY軸という考えがより合うため、点は(経度, 緯度)として見なされますが、逆には見なされません。

[表F-5](#)に示されている1つの演算子が提供されます。

表 F-5. Pointを基にしたearthdistanceの演算子

演算子	戻り値	説明
<code>point <@> point</code>	<code>float8</code>	法定マイル単位の地表の2点間の距離を返します。

このモジュールの`cube`を基にした場合と異なり、ここでの単位はコード内に固定で記載されることに注意してください。

earth() 関数を変更しても、この演算子の結果には影響しません。

緯度経度表現の1つの欠点は、極近辺と経度±180度近辺の限界条件に注意する必要があります。 cube を基にした表現ではこうした不連続性を防止できます。

[前のページ](#)
dummy_seclabel

[ホーム](#)
[上に戻る](#)

[次のページ](#)
file_fdw

F.14. file_fdw

file_fdwモジュールは、サーバのファイルシステムにあるデータファイルにアクセスするのに使用できる外部データラッパfile_fdwを提供します。データファイルはCOPY FROMで読むことのできるフォーマットでなければなりません。詳細は [COPY](#) を参照してください。

このラッパで作成された外部テーブルには以下のオプションを設定することができます。

filename

読み込むファイルを指定します。必須です。絶対パスでなければなりません。

format

ファイルフォーマットを指定するもので、COPYのFORMATオプションと同じです。

header

ファイルがヘッダ行を持つか指定するもので、COPYのHEADERオプションと同じです。

delimiter

ファイルのデリミタ文字を指定するもので、COPYのDELIMITERオプションと同じです。

quote

ファイルの引用符文字を指定するもので、COPYのQUOTEオプションと同じです。

escape

ファイルのエスケープ文字を指定するもので、COPYのESCAPEオプションと同じです。

null

ファイルのnull文字列を指定するもので、COPYのNULLオプションと同じです。

encoding

ファイルのエンコーディングを指定するもので、COPYのENCODINGオプションと同じです。

このラッパを使って作られた外部テーブルのカラムは、以下のオプションを持つことができます。

force_not_null

これはbooleanオプションです。真の場合は、このカラムの値はヌル値文字列(これはファイルレベルのnullオプションです)と比較されません。これは、COPYのFORCE_NOT_NULLオプションに列名を指定するのと同じ効果があります。

COPYのOIDS、FORCE_QUOTEの各オプションはfile_fdwでは現在サポートされていません。

これらのオプションは外部テーブルまたはその列にのみ指定可能で、file_fdw外部データラッパやそれを使用するサーバ、ユーザマッピングのオプションには指定できません。

どのファイルから読み込むかを決定できるのはスーパーユーザーのみでなければならない、というセキュリティ上の理由から、テーブルレベルのオプションを変更するにはスーパーユーザ権限が必要です。原則としては非スーパーユーザはその他のオプションを変更することを許されてもよいのですが、現時点ではサポートされていません。

file_fdwを使用する外部テーブルでは、EXPLAINは読み込むファイルの名前を表示します。COSTS OFFが指定されない場合は(バイト単位の)ファイルサイズも表示されます。

例 F-1. PostgreSQL CSV ログ用の外部テーブル作成

file_fdwの明確な用途の一つはPostgreSQLの活動ログをテーブルとして検索できるようにすることです。これを実現するには、ここではpglog.csvと呼ぶCSVファイルにログを記録する必要があります。まず、file_fdwをエクステンションとしてインストールします。

```
CREATE EXTENSION file_fdw;
```

続いて外部サーバを作成します。

```
CREATE SERVER pglog FOREIGN DATA WRAPPER file_fdw;  
2323/2500
```


これで外部テーブルを作成する準備ができました。CREATE FOREIGN TABLE コマンドを使って、テーブルのカラム、CSVファイル名とそのフォーマットを定義する必要がありますでしょう。

```
CREATE FOREIGN TABLE pglog (  
  log_time timestamp(3) with time zone,  
  user_name text,  
  database_name text,  
  process_id integer,  
  connection_from text,  
  session_id text,  
  session_line_num bigint,  
  command_tag text,  
  session_start_time timestamp with time zone,  
  virtual_transaction_id text,  
  transaction_id bigint,  
  error_severity text,  
  sql_state_code text,  
  message text,  
  detail text,  
  hint text,  
  internal_query text,  
  internal_query_pos integer,  
  context text,  
  query text,  
  query_pos integer,  
  location text,  
  application_name text  
) SERVER pglog  
OPTIONS ( filename '/home/josh/9.1/data/pg_log/pglog.csv', format 'csv' );
```

これで全てですーもうあなたはログに直接検索を実行することができます。実運用においては、もちろん何らかの方法でログを切り替える必要があるでしょう。

F.15. fuzzystmatch

fuzzystmatchモジュールは、文字列間の類似度や相違度を決める複数の関数を提供します。

注意

現時点で、soundex、metaphone、dmetaphone、dmetaphone_altは（UTF-8のような）マルチバイト符号化方式では十分に動作しません。

F.15.1. Soundex

Soundexシステムは、同一コードに変換することで似ているように見える名称を一致させる手法です。これは、1880年、1900年、1910年の米国国政調査で初めて使用されました。Soundexは非英語圏の名称では特に有用なものではないことに注意してください。

fuzzystmatchはSoundexコードを使用して動作する2つの関数を提供します。

```
soundex(text) returns text
difference(text, text) returns int
```

soundex関数は文字列をSoundexコードに変換します。difference関数は2つの文字列をそのSoundexコードに変換し、コード位置が一致する個数を報告します。Soundexコードは4文字からなりますので、結果は0から4までの範囲になります。0はまったく一致しないことを、4は完全に一致することを示します。（したがってこの関数の名前は間違っています。similarityの方がより優れた名前だったかもしれません。）

以下に使用例をいくつか示します。

```
SELECT soundex('hello world!');

SELECT soundex('Anne'), soundex('Ann'), difference('Anne', 'Ann');
SELECT soundex('Anne'), soundex('Andrew'), difference('Anne', 'Andrew');
SELECT soundex('Anne'), soundex('Margaret'), difference('Anne', 'Margaret');

CREATE TABLE s (nm text);

INSERT INTO s VALUES ('john');
INSERT INTO s VALUES ('joan');
INSERT INTO s VALUES ('wobbly');
INSERT INTO s VALUES ('jack');

SELECT * FROM s WHERE soundex(nm) = soundex('john');

SELECT * FROM s WHERE difference(s.nm, 'john') > 2;
```

F.15.2. レーベンシュタイン

この関数は2つの文字列間のレーベンシュタイン距離を計算します。

```
levenshtein(text source, text target, int ins_cost, int del_cost, int sub_cost) returns int
levenshtein(text source, text target) returns int
levenshtein_less_equal(text source, text target, int ins_cost, int del_cost, int sub_cost, int max_d) returns int
levenshtein_less_equal(text source, text target, int max_d) returns int
```

sourceおよびtargetは255バイトまでの任意の非NULL文字列を取ることができます。コストパラメータはそれぞれ、文字の挿入、削除、置換に負わせる文字数を指定します。この関数の2番目のバージョンのようにコストパラメータを省略することができます。この場合デフォルトですべて1になります。levenshtein_less_equalは距離の小さい値についてのlevenshtein関数の高速化版です。実際の距離がmax_dより小さいまたは等しい場合、levenshtein_less_equalはその正確な値を返します。さもなければ、この関数はmax_dより大きな値を返します。

以下に例を示します。

```
test=# SELECT levenshtein('GUMBO', 'GAMBOL');
       levenshtein
-----
                2
(1 row)
```

```

test=# SELECT levenshtein('GUMBO', 'GAMBOL', 2,1,1);
 levenshtein
-----
          3
(1 row)

test=# SELECT levenshtein_less_equal('extensive', 'exhaustive',2);
 levenshtein_less_equal
-----
          3
(1 row)

test=# SELECT levenshtein_less_equal('extensive', 'exhaustive',4);
 levenshtein_less_equal
-----
          4
(1 row)

```

F.15.3. Metaphone

Metaphoneは、Soundex同様、入力文字に対する対応するコードを構築するという考えに基づいたものです。2つの文字列が同一コードを持つ場合、類似とみなされます。

以下の関数は入力文字列に対するmetaphoneコードを計算します。

```
metaphone(text source, int max_output_length) returns text
```

sourceは255文字までの非NULL文字列を取ることができます。max_output_lengthは出力metaphoneコードの最大長を設定します。出力は長すぎるとこの長さに切り詰められます。

以下に例を示します。

```

test=# SELECT metaphone('GUMBO', 4);
 metaphone
-----
      KM
(1 row)

```

F.15.4. Double Metaphone

Double Metaphoneシステムは与えられた入力文字列に対する、"primary"と"alternate"という2つの"似たように見える"文字列を計算します。ほとんどの場合、これらは同じですが、英語以外の名称では特に発音に応じて多少異なる場合があります。以下の関数はprimaryコードとalternateコードを計算します。

```

dmetaphone(text source) returns text
dmetaphone_alt(text source) returns text

```

入力文字列長に関する制限はありません。

以下に例を示します。

```

test=# select dmetaphone('gumbo');
 dmetaphone
-----
      KMP
(1 row)

```

F.16. hstore

本モジュールはキー、値の組み合わせの集合を単一のPostgreSQL値に格納するためのhstoreデータ型を実装します。あまり厳密に検査されない属性を多く持つ行や半構造化データなど、多くの状況で有用になる可能性があります。キーと値は単純なテキスト文字列です。

F.16.1. hstoreの外部表現

入力および出力で使用されるhstore値のテキスト表現はカンマで区切られた、ゼロ以上の`key => value`という組み合わせを含みます。以下に例を示します。

```
k => v
foo => bar, baz => whatever
"l-a" => "anything at all"
```

組み合わせの順序は重要ではありません（出力時に再現されないこともあります）。組み合わせ間や=>記号の前後の空白文字は無視されます。キーや値が空白文字、カンマ、=、>を含む場合は二重引用符でくくります。キーや値に二重引用符やバックスラッシュを含めるには、バックスラッシュでエスケープしてください。

hstore内の各キーは一意です。重複するキーを持つhstoreを宣言すると、hstoreには1つしか保存されません。またどちらが残るかは保証されません。

```
SELECT 'a=>1,a=>2'::hstore;
 hstore
-----
"a"=>"1"
```

値はSQLのNULLを取ることができます（キーは不可）。以下に例を示します。

```
key => NULL
```

NULLキーワードは大文字小文字の区別をしません。nullを普通の文字列"NULL"として扱うためには二重引用符でくくってください。

注意: 入力として使用される場合hstoreテキスト書式は、前もって必要な引用符付けやエスケープ処理を適用することに注意してください。パラメータとしてhstoreリテラルを渡す場合、追加処理は必要ありません。しかし、引用符付けしたリテラル定数として渡す場合には、単一引用符および(`standard_conforming_strings`設定パラメータに依存しますが)バックスラッシュ文字をすべて正しくエスケープしなければなりません。文字列定数の取り扱いについては[項4.1.2.1](#)を参照してください。

出力の場合、厳密に必要な場合であっても、常にキーと値は二重引用符でくくられます。

F.16.2. hstoreの演算子と関数

hstoreモジュールで提供される演算子を[表F-6](#)に、関数を[表F-7](#)に示します。

表 F-6. hstore演算子

演算子	説明	例	結果
<code>hstore -> text</code>	キーの値を取り出します（存在しなければNULL）	<code>'a=>x, b=>y'::hstore -> 'a'</code>	x
<code>hstore -> text[]</code>	キーの値を取り出します（存在しなければNULL）	<code>'a=>x, b=>y, c=>z'::hstore -> ARRAY['c','a']</code>	{ "z", "x" }
<code>hstore hstore</code>	hstoreを連結します。	<code>'a=>b, c=>d'::hstore 'c=>x, d=>q'::hstore</code>	"a"=>"b", "c"=>"x", "d"=>"q"
<code>hstore ? text</code>	hstoreがキーを含むかどうか。	<code>'a=>1'::hstore ? 'a'</code>	t
<code>hstore ?& text[]</code>	hstoreが指定したキーをすべて含むかどうか。	<code>'a=>1,b=>2'::hstore ?& ARRAY['a','b']</code>	t
<code>hstore ? text[]</code>	hstoreが指定したキーのいずれかを含むかどうか。	<code>'a=>1,b=>2'::hstore ? ARRAY['b','c']</code>	t

hstore @> hstore	左辺は右辺を含むかどうか。	'a=>b, b=>1, c=>NULL'::hstore @> 'b=>1'	t
hstore <@ hstore	左辺は右辺に含まれるかどうか。	'a=>c'::hstore <@ 'a=>b, b=>1, c=>NULL'	f
hstore - text	左辺からキーを削除します。	'a=>1, b=>2, c=>3'::hstore - 'b'::text	"a"=>"1", "c"=>"3"
hstore - text[]	左辺からキー(複数)を削除します。	'a=>1, b=>2, c=>3'::hstore - ARRAY['a','b']	"c"=>"3"
hstore - hstore	左辺から一致する組み合わせを削除します。	'a=>1, b=>2, c=>3'::hstore - 'a=>4, b=>2'::hstore	"a"=>"1", "c"=>"3"
record #= hstore	record内のフィールドをhstoreの対応する値で置換します。	後述の例を参照	
%% hstore	キーと値を変更してhstoreを配列に変換します。	%% 'a=>foo, b=>bar'::hstore	{a,foo,b,bar}
%# hstore	hstoreをキーと値の2次元配列に変換します。	%# 'a=>foo, b=>bar'::hstore	{{a,foo},{b,bar}}

注意: PostgreSQL 8.2より前では、含有演算子@>と<@はそれぞれ@と~と呼ばれていました。これらの名前はまだ利用できますが、廃止予定であり、最終的にはなくなります。古い名前がコアの幾何データ型が従う規約と反対であることに注意してください。

表 F-7. hstoreの関数

関数	戻り値の型	説明	例	結果
hstore(record)	hstore	レコードまたは行からhstoreを生成します。	hstore(ROW(1,2))	f1=>1, f2=>2
hstore(text[])	hstore	配列からhstoreを生成します。配列はキー、値の配列でも2次元の配列でも構いません	hstore(ARRAY['a','1','b','2']) hstore(ARRAY[['c','3'],['d','4']])	a=>1, b=>2, c=>3, d=>4
hstore(text[], text[])	hstore	キー、値で分けた配列からhstoreを作成します。	hstore(ARRAY['a','b'], ARRAY['1','2'])	"a"=>"1", "b"=>"2"
hstore(text, text)	hstore	hstore型の単一項目を作成します。	hstore('a', 'b')	"a"=>"b"
akeys(hstore)	text[]	hstoreのキーを配列として入手します。	akeys('a=>1,b=>2')	{a,b}
skeys(hstore)	setof text	hstoreのキーを集合として入手します。	skeys('a=>1,b=>2')	a b
avals(hstore)	text[]	hstoreの値を配列として入手します。	avals('a=>1,b=>2')	{1,2}
svals(hstore)	setof text	hstoreの値を集合として入手します。	svals('a=>1,b=>2')	1 2
hstore_to_array(hstore)	text[]	hstoreのキーと値を、キーと値を交互に	hstore_to_array('a=>1,b=>2')	{a,1,b,2}

		表した配列として入手します。								
hstore_to_matrix(hstore)	text[]	hstoreのキーと値を2次元配列として入手します。	hstore_to_matrix('a=>1,b=>2')	{{a,1},{b,2}}						
slice(hstore, text[])	hstore	hstoreの部分集合を取り出します。	slice('a=>1,b=>2,c=>3'::hstore, ARRAY['b','c','x'])	"b"=>"2", "c"=>"3"						
each(hstore)	setof(key text, value text)	hstoreのキーと値を集合として入手します。	select * from each('a=>1,b=>2')	<table><tr><th>key</th><th>value</th></tr><tr><td>a</td><td>1</td></tr><tr><td>b</td><td>2</td></tr></table>	key	value	a	1	b	2
key	value									
a	1									
b	2									
exist(hstore,text)	boolean	hstoreがキーを含むかどうか。	exist('a=>1','a')	t						
defined(hstore,text)	boolean	hstoreがキーに対して非NULLの値を持つかどうか	defined('a=>NULL','a')	f						
delete(hstore,text)	hstore	キーに一致する組み合わせを削除します。	delete('a=>1,b=>2','b')	"a"=>"1"						
delete(hstore,text[])	hstore	キー(複数)に対応する組み合わせを削除します。	delete('a=>1,b=>2,c=>3',ARRAY['a','b'])	"c"=>"3"						
delete(hstore,hstore)	hstore	第二引数内の組み合わせと一致する組み合わせを削除します。	delete('a=>1,b=>2','a=>4,b=>2'::hstore)	"a"=>"1"						
populate_record(record,hstore)	record	record内のフィールドをhstoreの対応する値で置換します。	後述の例を参照							

注意: populate_record関数の最初の引数は実際にはrecordではなく anyelementと宣言されています。しかし、実行時にはレコード型以外は拒絶されます。

F.16.3. インデックス

hstoreは@>, ?, ?&および?| 演算子向けのGiSTおよびGINインデックスをサポートします。 以下に例を示します。

```
CREATE INDEX hidx ON testhstore USING GIST (h);

CREATE INDEX hidx ON testhstore USING GIN (h);
```

hstoreはまた、=演算子向けにbtreeまたはhashインデックスをサポートします。 これによりhstoreの列をUNIQUEと宣言すること、また、GROUP BY、ORDER BY、DISTINCTの式で使うことができます。 hstore値のソート順序付けはあまり有用ではありません。 しかしこれらのインデックスは等価値の検索の際に有用になるかもしれません。 =比較用のインデックスを以下のように作成します。

```
CREATE INDEX hidx ON testhstore USING BTREE (h);

CREATE INDEX hidx ON testhstore USING HASH (h);
```

F.16.4. 例

キーを追加、または、既存のキーを新しい値で更新します。

```
UPDATE tab SET h = h || hstore('c', '3');
```

キーを削除します。

```
UPDATE tab SET h = delete(h, 'k1');
```

recordをhstoreに変換します。

```
CREATE TABLE test (col1 integer, col2 text, col3 text);
INSERT INTO test VALUES (123, 'foo', 'bar');

SELECT hstore(t) FROM test AS t;
      hstore
-----
"col1"=>"123", "col2"=>"foo", "col3"=>"bar"
(1 row)
```

hstoreを事前に定義されたrecord型に変換します。

```
CREATE TABLE test (col1 integer, col2 text, col3 text);

SELECT * FROM populate_record(null::test,
                                '"col1"=>"456", "col2"=>"zzz"');
  col1 | col2 | col3
-----+-----+-----
  456 | zzz  |
(1 row)
```

hstoreの値を使用して既存のレコードを変更します。

```
CREATE TABLE test (col1 integer, col2 text, col3 text);
INSERT INTO test VALUES (123, 'foo', 'bar');

SELECT (r).* FROM (SELECT t #= '"col3"=>"baz"' AS r FROM test t) s;
  col1 | col2 | col3
-----+-----+-----
   123 | foo  | baz
(1 row)
```

F.16.5. 統計情報

内在する自由度のため、hstore型は異なるキーを多く含むことができます。有効なキーを検査することはアプリケーション側の作業です。以下の例では、キー検査および統計情報の入手に関する複数の技法を示します。

簡単な例を示します。

```
SELECT * FROM each('aaa=>bq, b=>NULL, ""=>1');
```

テーブルを使用する例です。

```
SELECT (each(h)).key, (each(h)).value INTO stat FROM testhstore;
```

オンライン統計値です。

```
SELECT key, count(*) FROM
  (SELECT (each(h)).key FROM testhstore) AS stat
GROUP BY key
ORDER BY count DESC, key;
  key      | count
-----+-----
line       |    883
query      |    207
pos        |    203
node       |    202
space      |    197
status     |    195
public     |    194
title      |    190
org        |    189
.....
```

F.16.6. 互換性

PostgreSQL 9.0からhstoreの内部表現はこれまでから変更されました。(ダンプ内で使用される)テキスト表現には変更がありませんので、ダンプ/リストアによる更新の妨げにはなりません。

バイナリによる更新の際、新しいコードで古い書式のデータを認識させることにより、上位互換が保持されます。これには、新しいコードによりまだ変更されていないデータを処理する際に、性能の劣化を多少伴います。以下のようにUPDATE文を実行することによりテーブル列内のすべての値を強制的に更新することができます。

```
UPDATE tablename SET hstorecol = hstorecol || '';
```

上を行う他の方法を以下に示します。

```
ALTER TABLE tablename ALTER hstorecol TYPE hstore USING hstorecol || '';
```

ALTER TABLEによる方法はテーブルに対して排他ロックを必要とします。しかし、古いバージョンの行でテーブルが膨張することはありません。

F.16.7. 作者

Oleg Bartunov <oleg@sai.msu.su>, Moscow, Moscow University, Russia

Teodor Sigaev <teodor@sigaev.ru>, Moscow, Delta-Soft Ltd., Russia

追加の改良はAndrew Gierth <andrew@tao11.riddles.org.uk>,United Kingdomによりなされました。

[前のページ](#)
fuzzystmatch

[ホーム](#)
[上に戻る](#)

[次のページ](#)
intagg

F.17. intagg

intaggモジュールは整数型の集約子と列挙子を提供します。 その能力の上位集合を提供する組み込み関数が存在しますので、intaggは現在使われません。 しかし、このモジュールは組み込み関数の互換ラップとして今でもまだ提供されています。

F.17.1. 関数

集約子は、正確に提供する整数のみを含む整数型配列を生成する`int_array_aggregate(integer)` 集約関数です。 これは任意の配列型で同じことを行う`array_agg`のラップです。

列挙子は、`setof integer`を返す`int_array_enum(integer[])` 関数です。 これは基本的に上記集約子の反対の操作を行います。 指定された整数型配列を行集合に拡張します。 これは任意の配列型で同じことを行う`unnest`のラップです。

F.17.2. 使用例

多くのデータベースシステムは1対多のテーブルを持ちます。 こうしたテーブルは通常、以下のように2つのインデックス用のテーブルの間に存在します。

```
CREATE TABLE left (id INT PRIMARY KEY, ...);
CREATE TABLE right (id INT PRIMARY KEY, ...);
CREATE TABLE one_to_many(left INT REFERENCES left, right INT REFERENCES right);
```

通常以下のように使用されます。

```
SELECT right.* from right JOIN one_to_many ON (right.id = one_to_many.right)
WHERE one_to_many.left = item;
```

これは、左辺のテーブル内にある項目に対応した、右辺のテーブル内のすべての項目を返します。 これはSQLで非常によく使用される式です。

さて、この方法論は`one_to_many`テーブル内に非常に多数の項目がある場合に扱いにくくなることがあり得ます。 しばしばこうした結合は、インデックススキャンと特定された左辺の項目に対応した右辺のテーブル内の項目をそれぞれ取り出すこととなります。 非常に動的なシステムでは、できることは多くありません。 しかし、ほぼ静的なデータが一部にある場合、集約子を使用して要約テーブルを作成することができます。

```
CREATE TABLE summary AS
SELECT left, int_array_aggregate(right) AS right
FROM one_to_many
GROUP BY left;
```

これは左辺項目毎に1行を持ち、右辺の項目の配列をもつテーブルを作成します。 さて、これは配列を使用する何らかの方法がないとかなり使い勝手が悪くなります。 これが配列列挙子が存在する理由です。 以下を行うことができます。

```
SELECT left, int_array_enum(right) FROM summary WHERE left = item;
```

上の`int_array_enum`を使用した問い合わせは、以下と同じ結果を生成します。

```
SELECT left, right FROM one_to_many WHERE left = item;
```

違いは、要約テーブルに対する問い合わせはテーブルから1行だけを取り出す必要があるのに対し、直接`one_to_many`に問い合わせる場合はインデックススキャンと各項目に対し行を取り出さなければならないという点です。

あるシステムでは`EXPLAIN`を行うと8488というコストを持つ問い合わせが329というコストまで減少しました。 元の問い合わせは`one_to_many`テーブルを含む結合でしたが、以下のように置き換えられました。

```
SELECT right, count(right) FROM
( SELECT left, int_array_enum(right) AS right
  FROM summary JOIN (SELECT left FROM left_table WHERE left = item) AS lefts
    ON (summary.left = lefts.left)
) AS list
GROUP BY right
ORDER BY count DESC;
```

F.18. intarray

`intarray`モジュールはNULLのない整数の配列の操作に便利な関数と演算子を多く提供します。 また、一部の演算子を使用したインデックス検索をサポートします。

配列にNULL要素が一つでも含まれていれば、これらの操作はすべてエラーを発生します。

これらの操作の多くは一次元配列に対してのみ適当なものです。 高次元の入力配列を受け付けますが、データは格納された順の一次元の配列であるかのように扱われます。

F.18.1. intarrayの関数および演算子

`intarray`モジュールで提供される関数を[表F-8](#)に、演算子を[表F-9](#)に示します。

表 F-8. intarray関数

関数	戻り値 の型	説明	例	結果
<code>icount(int[])</code>	<code>int</code>	配列内の要素数	<code>icount('{1,2,3}'::int[])</code>	3
<code>sort(int[], text dir)</code>	<code>int[]</code>	配列のソート。 <code>dir</code> は <code>asc</code> または <code>desc</code> のいずれかでなければなりません。	<code>sort('{1,2,3}'::int[], 'desc')</code>	{3,2,1}
<code>sort(int[])</code>	<code>int[]</code>	昇順ソート	<code>sort(array[11,77,44])</code>	{11,44,77}
<code>sort_asc(int[])</code>	<code>int[]</code>	昇順ソート		
<code>sort_desc(int[])</code>	<code>int[]</code>	降順ソート		
<code>uniq(int[])</code>	<code>int[]</code>	隣接する重複を削除	<code>uniq(sort('{1,2,3,2,1}'::int[]))</code>	{1,2,3}
<code>idx(int[], int item)</code>	<code>int</code>	<code>item</code> に一致する要素番号（存在しなければ0）	<code>idx(array[11,22,33,22,11], 22)</code>	2
<code>subarray(int[], int start, int len)</code>	<code>int[]</code>	<code>start</code> から始まり <code>len</code> 個の要素の部分配列	<code>subarray('{1,2,3,2,1}'::int[], 2, 3)</code>	{2,3,2}
<code>subarray(int[], int start)</code>	<code>int[]</code>	<code>start</code> から始まる部分配列	<code>subarray('{1,2,3,2,1}'::int[], 2)</code>	{2,3,2,1}
<code>intset(int)</code>	<code>int[]</code>	単一要素の配列を作成	<code>intset(42)</code>	{42}

表 F-9. intarray演算子

演算子	戻り値	説明
<code>int[] && int[]</code>	<code>boolean</code>	重なり。配列間で少なくとも1つの共通要素がある場合 <code>true</code> を返します。
<code>int[] @> int[]</code>	<code>boolean</code>	含有。左辺の配列が右辺の配列を含む場合 <code>true</code> を返します。
<code>int[] <@ int[]</code>	<code>boolean</code>	含有される。左辺の配列が右辺の配列に含まれる場合 <code>true</code> を返します。
<code># int[]</code>	<code>int</code>	配列内の要素数
<code>int[] # int</code>	<code>int</code>	インデックス（ <code>idx</code> 関数と同じです）
<code>int[] + int</code>	<code>int[]</code>	配列に要素をプッシュ（配列の末尾に追加）します
<code>int[] + int[]</code>	<code>int[]</code>	配列を連結（右辺配列を左辺配列の末尾に追加）します
<code>int[] - int</code>	<code>int[]</code>	配列から右辺の引数に一致する項目を削除します
<code>int[] - int[]</code>	<code>int[]</code>	左辺の配列から右辺の配列要素を削除します
<code>int[] int</code>	<code>int[]</code>	引数をまとめます
<code>int[] int[]</code>	<code>int[]</code>	配列をまとめます
<code>int[] & int[]</code>	<code>int[]</code>	配列の共通部分
<code>int[] @@ query_int</code>	<code>boolean</code>	配列が問い合わせを満たす場合 <code>true</code> （後述）
<code>query_int ~~ int[]</code>	<code>boolean</code>	配列が問い合わせを満たす場合 <code>true</code> （ <code>@@</code> の交代演算子）

（PostgreSQL 8.2以前では、含有演算子`@>`と`<@`はそれぞれ`@`と`~`と呼ばれていました。これらの名前はまだ利用できますが、廃

止予定であり、最終的にはなくなります。古い名前はコアの幾何データ型が以前従っていた規約とは反対であることに注意してください。)

演算子`&&`、`@>`と`<@`は、これらはNULLを含まない整数配列のみで動作し、組み込み演算子はどの配列型に対しても動作する点を除き、同じ名前のPostgreSQLの組み込み演算子とそれぞれほぼ等価です。この制限により、多くの場合、組み込み演算子より高速です。

`@@`および`~~`演算子は、配列が特化したデータ型`query_int`で表現される問い合わせを満たすかどうかを試験します。問い合わせは、おそらく`&`(論理積)、`|`(論理和)、`and` `!`(否定)演算子を組み合わせて使用した、配列要素に対して検査される整数値からなります。例えば`1&(2|3)`という問い合わせは1および、2か3のいずれかを含む配列に一致します。

F.18.2. インデックスサポート

`intarray`は`&&`、`@>`、`<@`、`@@`演算子に関して通常の配列等価性と同様にインデックスサポートを提供します。

2つのGiSTインデックス演算子クラスが提供されます。`gist_int_ops`(デフォルトで使用されます)は小中規模要素数のデータセットに適します。一方、`gist_intbig_ops`はより大きな署名を使用しますので、大規模データセット(つまり、異なった配列値を多数持つ列)のインデックスにより適しています。実装は組込みの非可逆圧縮を持ったRD Treeデータ構造を使用します。

また、同じ演算子をサポートするデフォルトではないGIN演算子クラス`gin_int_ops`も存在します。

GiSTおよびGINインデックスのどちらを選択するかは、別途説明されるGiSTとGINの相対的な性能特徴に依存します。まとめると、GINインデックスはGiSTインデックスより高速に検索することができますが、構築または更新は低速です。このため、GINは静的データに適し、GiSTは頻繁に更新されるデータに適します。

F.18.3. 例

```
-- a message can be in one or more "sections"
CREATE TABLE message (mid INT PRIMARY KEY, sections INT[], ...);

-- create specialized index
CREATE INDEX message_rdtree_idx ON message USING GIST (sections gist__int_ops);

-- select messages in section 1 OR 2 - OVERLAP operator
SELECT message.mid FROM message WHERE message.sections && '{1,2}';

-- select messages in sections 1 AND 2 - CONTAINS operator
SELECT message.mid FROM message WHERE message.sections @> '{1,2}';

-- the same, using QUERY operator
SELECT message.mid FROM message WHERE message.sections @@ '1&2'::query_int;
```

F.18.4. ベンチマーク

ソースディレクトリ以下の`contrib/intarray/bench`にはベンチマーク試験一式があります。以下のように実行します。

```
cd ../bench
createdb TEST
psql TEST < ../_int.sql
./create_test.pl | psql TEST
./bench.pl
```

`bench.pl`スクリプトには多くのオプションがあります。これらは引数を付けずに実行すると表示されます。

F.18.5. 作者

Teodor Sigaev (teodor@sigae.ru)とOleg Bartunov (oleg@sai.msu.su)によりすべての作業がなされました。さらなる情報については<http://www.sai.msu.su/~megeera/postgres/gist/>を参照してください。Andrey Oktyabrskiは新しい関数、演算子の追加において素晴らしい作業を行いました。

[前のページ](#)

intagg

[ホーム](#)

[上に戻る](#)

[次のページ](#)

isn

F.19. isbn

isbnモジュールは、EAN13、UPC、ISBN（書籍）、ISMN（音楽）、ISSN（連番）という国際的な標準製品番号に従うデータ型を提供します。番号は入力時にハードコードされた接頭辞の一覧に基づいて検証されます。この接頭辞の一覧は出力時に数字にハイフンを付けるのにも使われます。新しい接頭辞が時々追加されますので、接頭辞の一覧は古くなっているかもしれません。このモジュールの将来のバージョンでは、必要なときにユーザが簡単に更新できる一つもしくは複数のテーブルから接頭辞の一覧を取得することが望まれます。しかし、現時点では、一覧はソースコードを修正し再コンパイルすることでしか更新できません。あるいは、接頭辞の検証とハイフン付けのサポートはこのモジュールの将来のバージョンからは外されるかもしれません。

F.19.1. データ型

表F-10にisbnモジュールで提供されるデータ型を示します。

表 F-10. isbnデータ型

データ型	説明
EAN13	ヨーロッパ統一商品コード。常にEAN13用表示形式で表示されます。
ISBN13	国際標準図書番号。新しいEAN13用表示形式で表示されます。
ISMN13	国際標準楽譜番号。新しいEAN13用表示形式で表示されます。
ISSN13	国際標準逐次刊行物番号。新しいEAN13用表示形式で表示されます。
ISBN	国際標準図書番号。旧式の簡略表示形式で表示されます。
ISMN	国際標準楽譜番号。旧式の簡略表示形式で表示されます。
ISSN	国際標準逐次刊行物番号。旧式の簡略表示形式で表示されます。
UPC	統一商品コード。

数点注意事項があります。

- 1 11 ISBN13、ISMN13、ISSN13番号はすべてEAN13数値です。
- 1 11 EAN13番号は常にISBN13、ISMN13、ISSN13のいずれかではありません（一部はそうです）。
- 1 11 一部のISBN13番号はISBNとして表示可能です。
- 1 11 一部のISMN13番号はISMNとして表示可能です。
- 1 11 一部のISSN13番号はISSNとして表示可能です。
- 1 11 UPC番号はEAN13番号の部分集合です（基本的にはEAN13から先頭の0桁を取り除いたものです）。
- 1 11 すべてのUPC、ISBN、ISMN、ISSNはEAN13番号として表現可能です。

すべての型は内部的には同一表現（64ビット整数）を使用し、すべて相互交換が可能です。複数の型は、表示書式を制御する、および、番号のある特定の型を表さなければならない入力に対する有効性検査をより強く行えるようにするために提供されています。

ISBN、ISMN、ISSN型では、可能ならば番号の簡略表示形式（ISxN 10）で表示されます。簡略形式に合わない番号ではISxN 13書式で表示されます。EAN13、ISBN13、ISMN13、ISSN13型では常にISxNの長めの形式（EAN13）で表示されます。

F.19.2. キャスト

isbnモジュールは以下の型の組合せに関するキャストを提供します。

- ISBN13 <=> EAN13
- ISMN13 <=> EAN13
- ISSN13 <=> EAN13

- ISBN <=> EAN13
- ISMN <=> EAN13
- ISSN <=> EAN13
- UPC <=> EAN13
- ISBN <=> ISBN13
- ISMN <=> ISMN13
- ISSN <=> ISSN13

EAN13から他の型へキャストする時、その値が他の型のドメイン内であるかどうか実行時に検査が行われます。ドメイン内になればエラーが発生します。他のキャストでは単にラベル付けを再実行するだけですので、常に成功します。

F.19.3. 関数と演算子

isnモジュールは標準的な比較演算子とこれらデータ型すべてに対するB-treeおよびハッシュインデックスサポートを提供します。さらに、表F-11で示される複数の特化した関数も存在します。以下の表ではisnはこのモジュールのデータ型のいずれか1つを意味します。

表 F-11. isn関数

関数	戻り値	説明
isn_weak(boolean)	boolean	弱めの入力モードを設定します。（新しい設定を返します）
isn_weak()	boolean	強弱モードの現在の状態を返します。
make_valid(isn)	isn	無効な番号を有効にします（無効フラグを消去します）。
is_valid(isn)	boolean	無効フラグの有無を検査します。

強弱モードは無効なデータをテーブルに挿入できるようにするために使用されます。無効とは間違った検査桁を意味するものであり、番号自体は存在します。

この弱めのモードを使いたいと考えるのは何故でしょうか。大規模なISBN番号群があり、その内の多くが何らかの理由で間違った検査桁を持つことはあり得ます。（印刷された一覧をスキャンしてOCRした結果番号を間違えた場合、手作業で番号を取り出した場合などがあり得ます。）とにかく、こうした混乱は整理したいことですが、データベース内に番号をすべて取り込んで、より簡単に情報を検査し有効にすることができるように、外部ツールを使用してデータベース内の無効な番号の位置を特定したいと思うかも知れません。例えば、テーブル内の無効な番号をすべて選択したいと思うかも知れません。

弱めのモードを使用して無効な番号をテーブルに挿入する時、番号は修正された検査桁付きで挿入されますが、最後にイクスクラメーション印 (!) 付きで、例えば0-11-000322-5!と表示されます。この無効印はis_valid関数を使って検査することができます、また、make_valid関数で消去することができます。

また、番号の最後に!文字を付与することで、弱めのモードでなくとも無効な番号を強制的に挿入することもできます。

この他の入力における特殊な機能として、検査桁として?を書くことができます。これにより正確な検査桁が自動的に挿入されます。

F.19.4. 例

```

--Using the types directly:
SELECT isbn('978-0-393-04002-9');
SELECT isbn13('0901690546');
SELECT issn('1436-4522');

--Casting types:
-- note that you can only cast from ean13 to another type when the
-- number would be valid in the realm of the target type;
-- thus, the following will NOT work: select isbn(ean13('0220356483481'));
-- but these will:
SELECT upc(ean13('0220356483481'));
SELECT ean13(upc('220356483481'));

--Create a table with a single column to hold ISBN numbers:
CREATE TABLE test (id isbn);
INSERT INTO test VALUES('9780393040029');
```

```
--Automatically calculate check digits (observe the '?'):
INSERT INTO test VALUES('220500896?');
INSERT INTO test VALUES('978055215372?');

SELECT issn('3251231?');
SELECT ismn('979047213542?');

--Using the weak mode:
SELECT isn_weak(true);
INSERT INTO test VALUES('978-0-11-000533-4');
INSERT INTO test VALUES('9780141219307');
INSERT INTO test VALUES('2-205-00876-X');
SELECT isn_weak(false);

SELECT id FROM test WHERE NOT is_valid(id);
UPDATE test SET id = make_valid(id) WHERE id = '2-205-00876-X!';

SELECT * FROM test;

SELECT isbn13(id) FROM test;
```

F.19.5. 参考文献

本モジュールを実装するための情報は以下を含むいくつかのサイトを通して集められました。

- <http://www.isbn-international.org/>
- <http://www.issn.org/>
- <http://www.ismn-international.org/>
- <http://www.wikipedia.org/>

ハイフン付けに使用した接頭辞も以下から集められました。

- http://www.gs1.org/productssolutions/idkeys/support/prefix_list.html
- <http://www.isbn-international.org/en/identifiers.html>
- <http://www.ismn-international.org/ranges.html>

アルゴリズムの作成には注意を払い、公式ISBN、ISMN、ISSNユーザマニュアルで提示されたアルゴリズムに対して念入り過ぎるほど検証されました。

F.19.6. 作者

Germán Méndez Bravo (Kronuz), 2004 - 2006

本モジュールはGarrett A. Wollmanのisbn_issnコードに触発されたものです。

[前のページ](#)
intarray

[ホーム](#)
[上に戻る](#)

[次のページ](#)
lo

F.20. lo

loモジュールはラージオブジェクト（LOやBLOBとも呼ばれます）保守作業のサポートを提供します。 loデータ型とlo_manageトリガが含まれます。

F.20.1. 原理

JDBCドライバにおける問題の1つ（ODBCドライバでもこれは影響します）は、規定ではBLOB（バイナリラージオブジェクト）への参照はテーブル内に格納され、その項目が変更されると、関連するBLOBがデータベースから削除されると想定している点です。

PostgreSQLの立場では、これは起こりません。 ラージオブジェクトは独自の権限をもったオブジェクトとして扱われます。 テーブル項目はOIDによりラージオブジェクトを参照することはできますが、同じラージオブジェクトOIDを参照するテーブル項目を複数持つことも可能です。 このため、システムは、こうした項目の変更または削除されたという理由だけでは、ラージオブジェクトを削除しません。

さて、これはPostgreSQL固有のアプリケーションでは問題ありませんが、JDBCやODBCを使用する標準的なコードでは、オブジェクトが削除されず、孤児、つまりどこからも参照されずディスクを消費するだけのオブジェクトになります。

loモジュールによりLO参照列を持つテーブルにトリガを付与して、これを解消することができます。 このトリガは基本的には、ラージオブジェクトを参照する値を削除または変更した時常にlo_unlinkを単に行います。 このトリガを使用する時は、単一のデータベースのみがトリガの対象列で参照されるラージオブジェクトを参照することを前提とします。

また、本モジュールは、単にoid型のドメインに過ぎないloデータ型を提供します。 ラージオブジェクトへの参照を持つデータベース列とこの他のOIDを持つデータベース列との間に違いを持たせるために有用です。 実際このトリガを使用するためにlo型を使用する必要はありません。 しかし、データベース内のどの列がトリガで管理されているラージオブジェクトを示しているかを保持するために、これを使用することは簡便かもしれません。 また、BLOB列でloを使用しない場合、ODBCドライバが混乱してしまうと取りざたされています。

F.20.2. 使用方法

簡単な使用例を示します。

```
CREATE TABLE image (title TEXT, raster lo);

CREATE TRIGGER t_raster BEFORE UPDATE OR DELETE ON image
FOR EACH ROW EXECUTE PROCEDURE lo_manage(raster);
```

一意なラージオブジェクト参照を含む列それぞれに対し、BEFORE UPDATE OR DELETEトリガを作成してください。 そして、単一のトリガ引数として列名を指定してください。 同一テーブル上に複数のlo型の列を持たせる必要がある場合、それぞれに対して別のトリガを作成してください。 同一テーブル上の各トリガに別の名前を与えることは忘れないでください。

F.20.3. 制限

- トリガが実行されませんので、テーブル削除により含まれるオブジェクトは孤児化します。 DROP TABLEの前にDELETE FROM tableを行うことで防止することができます。

TRUNCATEも同様の危険があります。

ラージオブジェクトを孤児化させた、または孤児化させた疑いがある場合は、消去するための手助けとなるvacuumloモジュールを参照してください。 lo_manageトリガのバックネットとしてvacuumloを時々実行することを勧めます。

- フロントエンドの中には独自のテーブルを作成するものがあり、その場合、関連するトリガは作成されません。 また、ユーザはトリガを作成することを忘れる（または知らない）かもしれません。

F.20.4. 作者

Peter Mount <peter@retap.org.uk>

F.21. ltree

本モジュールは階層ツリーを模擬した構造に格納されたデータのラベルを表現する `ltree` データ型を実装します。ラベルツリー全体を検索する高度な機能を提供します。

F.21.1. 定義

ラベルは、アルファベット文字とアンダースコア（例えばCロケールではA-Za-z0-9_文字が許されます。）の並びです。ラベルの長さは256バイト未満でなければなりません。

例えば42、Personal_Servicesです。

ラベル経路は、例えばL1.L2.L3のようなドットで区切られた0個以上のラベルの並びであり、階層ツリーのルートから特定のノードまでの経路を表します。ラベル経路の長さは65キロバイトまでに制限されていますが、2キロバイト以下のサイズがよく使われます。実際のところこれは主要な制限ではありません。例えばDMOZカタログ (<http://www.dmoz.org>) における最大ラベル経路はおよそ240バイトです。

例: 'Top.Countries.Europe.Russia'

`ltree` モジュールは以下の複数のデータ型を提供します。

- `ltree` はラベル経路を格納します。
- `lquery` は、`ltree` 値に一致する正規表現のようなパターンを表現します。単一の単語は経路内のラベルに一致します。スター記号 (*) は0個以上のラベルに一致します。以下に例を示します。

<code>foo</code>	正確に <code>foo</code> というラベル経路に一致します。
<code>*.foo.*</code>	<code>foo</code> というラベルを含むラベル経路すべてに一致します。
<code>*.foo</code>	<code>foo</code> というラベルで終わるラベル経路すべてに一致します。

スター印は一致可能なラベル数を制限するために量指定を行うことができます。

<code>*{n}</code>	正確に n 個のラベルに一致します。
<code>*{n,}</code>	少なくとも n 個のラベルに一致します。
<code>*{n,m}</code>	少なくとも n 個に一致し、多くても m 個を超えないラベルに一致します。
<code>*{,m}</code>	最大 m 個のラベルに一致します。つまり <code>*{0,m}</code> と同じです。

単なる正確な一致以上の一致を行うために、`lquery` の非スターラベルの終端に記述することができる複数の修飾子が存在します。

<code>@</code>	大文字小文字を区別しない一致。例えば <code>a@</code> は <code>A</code> に一致します。
<code>*</code>	この接頭辞を持つすべてのラベルに一致。例えば <code>foo*</code> は <code>foobar</code> に一致します。
<code>%</code>	最初のアンダースコアで区切られた単語に一致。

`%`の動作は多少複雑です。ラベル全体ではなく単語一致を試みます。例えば`foo_bar%`は`foo_bar_baz`に一致しますが`foo_barbaz`に一致しません。`*`と組み合わせる場合、接頭辞一致が各単語ごとに適用されます。例えば`foo_bar%*`は`foo1_bar2_baz`に一致しますが、`foo1_br2_baz`に一致しません。

また、ラベルのいずれかに一致させるために「`|`」（論理和）で区切って、複数のおそらく修飾子が付いたラベルを記述することもできます。さらに、先頭に「`!`」（否定）を記述して選択肢のいずれかにも一致しないすべてのラベルに一致させることもできます。

以下に注釈付きの`lquery`の例を示します。

```
Top.*{0,2}.sport*@.!football|tennis.Russ*|Spain
a.   b.       c.       d.       e.
```

この問い合わせは以下のようなラベルに一致します。

- a aa Topラベルから始まる。
- a aa 次いで0から2個のラベルを持つ。
- a aa 直後に`sport`接頭辞（大文字小文字の区別無）から始まるラベルを持つ。
- a aa そして、`football`と`tennis`に一致しないラベルを持つ。

a aa Russから始まる、または、正確にSpainに一致するラベルで終わる。

- ltxtqueryはltree値に対する全文検索のようなパターンを表します。 ltxtquery値は、おそらく最後に@、*、%修飾子を持った単語からなります。 修飾子の意味はlqueryと同じです。 単語は& (論理積)、| (論理和)、! (否定)、括弧を組み合わせることが可能です。 主なlqueryとの違いは、 ltxtqueryはラベル経路上の位置を考慮せずに単語に一致することです。

ltxtqueryの例を示します。

Europe & Russia*@ & !Transportation

これはEurope ラベルとRussia (大文字小文字の区別無) から始まるラベルを含む経路に一致します。 しかし、Transportation ラベルを含む経路は一致しません。 経路内の単語の位置は重要ではありません。 また、%が使用された場合、位置に関係なく、単語をラベル内のアンダースコアで区切られた何らかの単語に一致させることができます。

注意：ltxtqueryではシンボルの間に空白を入れることができますが、ltreeとlqueryではできません。

F.21.2. 演算子と関数

ltree型は、通常の比較演算子=、<>、<、>、<=、>=を持ちます。 比較では、ツリーの巡回順でソートされ、ノードの子要素はラベルテキストでソートされます。 さらに、表F-12に示す特殊な演算子が使用可能です。

表 F-12. ltree演算子

演算子	戻り値	説明
ltree @> ltree	boolean	左辺の引数が右辺の祖先要素 (か同じ) かどうか
ltree <@ ltree	boolean	左辺の引数が右辺の子孫要素 (か同じ) かどうか
ltree ~ lquery	boolean	ltreeがlqueryに一致するかどうか
lquery ~ ltree	boolean	ltreeがlqueryに一致するかどうか
ltree ? lquery[]	boolean	ltreeが配列内のいずれかのlqueryに一致するかどうか
lquery[] ? ltree	boolean	ltreeが配列内のいずれかのlqueryに一致するかどうか
ltree @ ltxtquery	boolean	ltreeがltxtqueryに一致するかどうか
ltxtquery @ ltree	boolean	ltreeがltxtqueryに一致するかどうか
ltree ltree	ltree	ltree経路を連結します
ltree text	ltree	テキストをltreeに変換し、連結します
text ltree	ltree	テキストをltreeに変換し、連結します
ltree[] @> ltree	boolean	配列にltreeの祖先要素が含まれるかどうか
ltree <@ ltree[]	boolean	配列にltreeの祖先要素が含まれるかどうか
ltree[] <@ ltree	boolean	配列にltreeの子孫要素が含まれるかどうか
ltree @> ltree[]	boolean	配列にltreeの子孫要素が含まれるかどうか
ltree[] ~ lquery	boolean	配列にlqueryに一致する経路が含まれるかどうか
lquery ~ ltree[]	boolean	配列にlqueryに一致する経路が含まれるかどうか
ltree[] ? lquery[]	boolean	ltree配列にいずれかのlqueryに一致する経路が含まれるかどうか
lquery[] ? ltree[]	boolean	ltree配列にいずれかのlqueryに一致する経路が含まれるかどうか
ltree[] @ ltxtquery	boolean	配列にltxtqueryに一致する経路が含まれるかどうか
ltxtquery @ ltree[]	boolean	配列にltxtqueryに一致する経路が含まれるかどうか
ltree[] ?@> ltree	ltree	ltreeの祖先要素となる配列内の最初の要素。存在しなければNULL
ltree[] ?<@ ltree	ltree	ltreeの子孫要素となる配列内の最初の要素。存在しなければNULL
ltree[] ?~ lquery	ltree	lqueryに一致する配列内の最初の要素。存在しなければNULL
ltree[] ?@ ltxtquery	ltree	ltxtqueryに一致する配列内の最初の要素。存在しなければNULL

<@、@>、@、~演算子は^<@、^@>、^@、^~と類似です。 これらはインデックスを使用しない点を除き、同一です。 これらは試験の際にだけ役に立ちます。

使用可能な関数を表F-13に示します。

表 F-13. ltree関数

関数	戻り値 型	説明	例	結果
subltree(ltree, int start, int end)	ltree	start 位置からend-1位置まで のltreeの部分経路（位置は0から始 まります）。	subltree('Top.Child1.Child2',1,2)	Child1
subpath(ltree, int offset, int len)	ltree	offset 位置からlen個のltreeの部分 経路（位置は0から始まります）。 offset が負の場合、部分経路は経路 の終端から数えた位置から始ま ります。 len が負の場合、経路の終端 から指定個のラベルを除きます。	subpath('Top.Child1.Child2',0,2)	Top.Child1
subpath(ltree, int offset)	ltree	offset 位置から経路の終端まで のltreeの部分経路（位置は0から始 まります）。 offset が負の場合、 部分経路は経路の終端から数えた位 置から始まります。	subpath('Top.Child1.Child2',1)	Child1.Child2
nlevel(ltree)	integer	経路内のラベル数	nlevel('Top.Child1.Child2')	3
index(ltree a, ltree b)	integer	a内でbが最初に出現する位置。存在 しなければ-1	index('0.1.2.3.5.4.5.6.8.5.6.8','5.6')	6
index(ltree a, ltree b, int offset)	integer	a内でoffset から検索を始めてbが最 初に出現する位置。 負のoffset は 経路終端から -offset ラベルから検 索を始めることを意味します。	index('0.1.2.3.5.4.5.6.8.5.6.8','5.6',- 4)	9
text2ltree(text)	ltree	text をltree にキャスト		
ltree2text(ltree)	text	ltree をtext にキャスト		
lca(ltree, ltree, ...)	ltree	最少共通祖先。つまり、経路で共通 する最長接頭辞。（最大8個の引数 をサポート）	lca('1.2.2.3','1.2.3.4.5.6')	1.2
lca(ltree[])	ltree	最少共通祖先。つまり、経路で共通 する最長接頭辞。	lca(array['1.2.2.3'::ltree,'1.2.3'])	1.2

F.21.3. インデックス

ltree は、以下で示された演算子を高速化できる、複数種類のインデックスをサポートします。

- ltree に対する B-tree インデックス：<、<=、=、>=、>
- ltree に対する GiST インデックス：<、<=、=、>=、>、@>、<@、@、~、？

インデックスの作成例を以下に示します。

```
CREATE INDEX path_gist_idx ON test USING GIST (path);
```

- ltree[] に対する GiST インデックス：ltree[] <@ ltree、ltree @> ltree[]、@、~、？

インデックスの作成例を以下に示します。

```
CREATE INDEX path_gist_idx ON test USING GIST (array_path);
```

注意：この種類のインデックスは非可逆です。

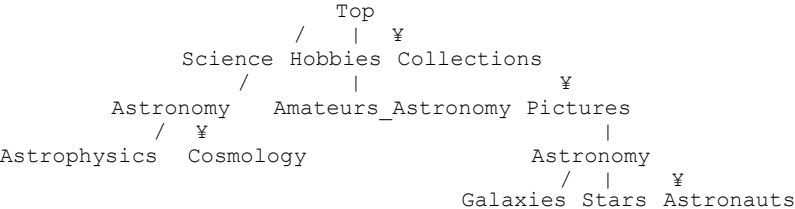
F.21.4. 例

この例は、後述のデータを使用します（ソース配布内の contrib/ltree/ltreetest.sql ファイルでも利用可能です）。

```
CREATE TABLE test (path ltree);
INSERT INTO test VALUES ('Top');
INSERT INTO test VALUES ('Top.Science');
INSERT INTO test VALUES ('Top.Science.Astronomy');
```

```
INSERT INTO test VALUES ('Top.Science.Astronomy.Astrophysics');
INSERT INTO test VALUES ('Top.Science.Astronomy.Cosmology');
INSERT INTO test VALUES ('Top.Hobbies');
INSERT INTO test VALUES ('Top.Hobbies.Amateurs_Astronomy');
INSERT INTO test VALUES ('Top.Collections');
INSERT INTO test VALUES ('Top.Collections.Pictures');
INSERT INTO test VALUES ('Top.Collections.Pictures.Astronomy');
INSERT INTO test VALUES ('Top.Collections.Pictures.Astronomy.Stars');
INSERT INTO test VALUES ('Top.Collections.Pictures.Astronomy.Galaxies');
INSERT INTO test VALUES ('Top.Collections.Pictures.Astronomy.Astronauts');
CREATE INDEX path_gist_idx ON test USING gist(path);
CREATE INDEX path_idx ON test USING btree(path);
```

これで、以下の階層を記述するデータが投入されたtestテーブルができます。



継承を行うことができます。

```
ltreetest=> SELECT path FROM test WHERE path <@ 'Top.Science';
path
-----
Top.Science
Top.Science.Astronomy
Top.Science.Astronomy.Astrophysics
Top.Science.Astronomy.Cosmology
(4 rows)
```

経路一致の例をいくつか示します。

```
ltreetest=> SELECT path FROM test WHERE path ~ '.*.Astronomy.*';
path
-----
Top.Science.Astronomy
Top.Science.Astronomy.Astrophysics
Top.Science.Astronomy.Cosmology
Top.Collections.Pictures.Astronomy
Top.Collections.Pictures.Astronomy.Stars
Top.Collections.Pictures.Astronomy.Galaxies
Top.Collections.Pictures.Astronomy.Astronauts
(7 rows)

ltreetest=> SELECT path FROM test WHERE path ~ '.*!pictures@.*.Astronomy.*';
path
-----
Top.Science.Astronomy
Top.Science.Astronomy.Astrophysics
Top.Science.Astronomy.Cosmology
(3 rows)
```

全文検索の例をいくつか示します。

```
ltreetest=> SELECT path FROM test WHERE path @ 'Astro%' & !pictures@';
path
-----
Top.Science.Astronomy
Top.Science.Astronomy.Astrophysics
Top.Science.Astronomy.Cosmology
Top.Hobbies.Amateurs_Astronomy
(4 rows)

ltreetest=> SELECT path FROM test WHERE path @ 'Astro*' & !pictures@';
path
-----
Top.Science.Astronomy
Top.Science.Astronomy.Astrophysics
Top.Science.Astronomy.Cosmology
(3 rows)
```

関数を使用した経路構築の例です。

```
ltreetest=> SELECT subpath(path,0,2)||'Space'||subpath(path,2) FROM test WHERE path <@
'Top.Science.Astronomy';
?column?
-----
Top.Science.Space.Astronomy
Top.Science.Space.Astronomy.Astrophysics
```

```
Top.Science.Space.Astronomy.Cosmology
(3 rows)
```

経路内の位置にラベルを挿入するSQL関数を作成することで、これを簡略化することができます。

```
CREATE FUNCTION ins_label(ltree, int, text) RETURNS ltree
AS 'select subpath($1,0,$2) || $3 || subpath($1,$2);'
LANGUAGE SQL IMMUTABLE;

ltreetest=> SELECT ins_label(path,2,'Space') FROM test WHERE path <@ 'Top.Science.Astronomy';
           ins_label
-----
Top.Science.Space.Astronomy
Top.Science.Space.Astronomy.Astrophysics
Top.Science.Space.Astronomy.Cosmology
(3 rows)
```

F.21.5. 作者

開発はすべてTeodor Sigaev (<teodor@stack.net>)とOleg Bartunov (<oleg@sai.msu.su>)によりなされました。さらなる情報については<http://www.sai.msu.su/~megera/postgres/gist/>を参照してください。作者は有用な議論を行ったEugeny Rodichevに感謝しています。コメントや不具合報告を歓迎します。

[前のページ](#)
lo

[ホーム](#)
[上に戻る](#)

[次のページ](#)
pageinspect

F.22. pageinspect

pageinspectモジュールは、デバッグの際に有用となる低レベルなデータベースページの内容を調べることができる関数を提供します。これらの関数はすべて、スーパーユーザのみが使用することができます。

F.22.1. 関数

get_raw_page(relname text, fork text, blkno int) returns bytea

get_raw_pageは、指定されたテーブルの指定されたブロックを読み取り、bytea値としてそのコピーを返します。これにより、単一ブロックの時間的に一貫性を持つコピーを入手することができます。forkは、主データフォークでは'main'、空き領域マップでは'fsm'、可視性マップでは'vm'としなければなりません。

get_raw_page(relname text, blkno int) returns bytea

get_raw_pageの簡略形であり、主フォークから読み取ります。get_raw_page(relname, 'main', blkno)と同じです。

page_header(page bytea) returns record

page_headerは、すべてのPostgreSQLヒープとインデックスページで共通するフィールドを表示します。

get_raw_pageで得られたページイメージを引数として渡さなければなりません。以下に例を示します。

```
test=# SELECT * FROM page_header(get_raw_page('pg_class', 0));
lsn      | tli | flags | lower | upper | special | pagesize | version | prune_xid
-----+-----+-----+-----+-----+-----+-----+-----+-----
0/24A1B50 | 1   |      1 | 232   | 368   |      8192 | 8192    | 4       | 0
```

返却される列は、PageHeaderData構造体のフィールドに対応します。詳細はsrc/include/storage/bufpage.hを参照してください。

heap_page_items(page bytea) returns setof record

heap_page_itemsはヒープページ上の行ポインタをすべて表示します。使用中の行ポインタでは、タプルヘッダも表示されます。生ページがコピーされた時点のMVCCスナップショットでタプルが可視かどうかは関係なく、すべてのタプルが表示されます。

get_raw_pageで得られたヒープページイメージを引数として渡さなければなりません。以下に例を示します。

```
test=# SELECT * FROM heap_page_items(get_raw_page('pg_class', 0));
```

返却されるフィールドの説明については、src/include/storage/itemid.hおよびsrc/include/access/htup.hを参照してください。

bt_metap(relname text) returns record

bt_metapはB-treeインデックスのメタページに関する情報を返却します。以下に例を示します。

```
test=# SELECT * FROM bt_metap('pg_cast_oid_index');
-[ RECORD 1 ]-----
magic      | 340322
version    | 2
root       | 1
level      | 0
fastroot   | 1
fastlevel  | 0
```

bt_page_stats(relname text, blkno int) returns record

bt_page_statsは、B-treeインデックスの個別のページについての要約情報を返却します。以下に例を示します。

```
test=# SELECT * FROM bt_page_stats('pg_cast_oid_index', 1);
-[ RECORD 1 ]-----
blkno      | 1
type       | 1
live_items | 256
dead_items | 0
avg_item_size | 12
page_size  | 8192
free_size  | 4056
btpo_prev  | 0
btpo_next  | 0
btpo       | 0
```


bt_page_items(relname text, blkno int) returns setof record

bt_page_itemsは、B-treeインデックスページ上の全項目についての詳細情報を返却します。 以下に例を示します。

```
test=# SELECT * FROM bt_page_items('pg_cast_oid_index', 1);
```

itemoffset	ctid	itemlen	nulls	vars	data
1	(0,1)	12	f	f	23 27 00 00
2	(0,2)	12	f	f	24 27 00 00
3	(0,3)	12	f	f	25 27 00 00
4	(0,4)	12	f	f	26 27 00 00
5	(0,5)	12	f	f	27 27 00 00
6	(0,6)	12	f	f	28 27 00 00
7	(0,7)	12	f	f	29 27 00 00
8	(0,8)	12	f	f	2a 27 00 00

fsm_page_contents(page bytea) returns text

fsm_page_contentsは、FSMページの内部ノード構造を表示します。 出力は複数行からなる文字列で、各行がページ内のバイナリツリー（二分木）の1ノードを表します。 それらのノードのうち、非ゼロのノードのみが出力されます。 そのページから返される次のスロットを指し示すための"next（次）"と呼ばれるポインタも出力されます。

FSMページの構造に関する更に詳しい情報は、src/backend/storage/freespace/READMEを参照してください。

F.23. パスワード検査

passwordcheckモジュールは[CREATE ROLE](#)または[ALTER ROLE](#)によって設定したユーザのパスワードを検査します。パスワードが弱すぎると考えられた場合、パスワードは拒絶されてコマンドはエラーで終わります。

このモジュールを有効にするには、postgresql.conf中の[shared_preload Libraries](#)に'\$libdir/passwordcheck'を追加して、サーバを再起動してください。

ソースコードの変更により、このモジュールをユーザの用途に適合できます。例えば、パスワード検査のため[CrackLib](#)を使用できます。これに必要な作業はMakefile中の2行のコメントアウトを外してモジュールを再構築することだけです。（ライセンスの理由からCrackLibをデフォルトで組み込むことができません。）CrackLibがなくても、モジュールはパスワードの強度に関する幾つかの単純な規則を強制します。ユーザはこの規則を、適切となるように修正または拡張できます。

注意

暗号化されないパスワードが、ネットワークを送信されること、サーバログに記録されることおよびデータベース管理者に盗聴されることを防ぐために、PostgreSQLはパスワードの一方向暗号化をユーザに提供できます。この機能を使用して、多くのクライアントプログラムはサーバへ送信する前にパスワードを暗号化できます。

一方向暗号化されたパスワードは復号できないので、これがpasswordcheckモジュールの有効性の限界となります。この理由により、高度のセキュリティが要求される場合、passwordcheckは推奨されません。データベース内部に保存したパスワードに依存するよりは、Kerberosのような外部の認証方法を使用した方が安全です（[第19章](#)参照）。

その他の方法として、一方向暗号化されたパスワードを拒否するためにpasswordcheckを修正できます。しかし、この方法ではパスワードが平文のテキストとして送信されるため、ユーザに多大なセキュリティリスクを負担させます。

F.24. pg_buffercache

pg_buffercacheモジュールは、共有バッファキャッシュで何が起きているかをリアルタイムに確認する方法を提供します。

このモジュールはレコード集合を返すpg_buffercache_pages C関数と、簡単に利用できるようにこの関数を隠蔽するpg_buffercacheビューを提供します。

どちらに対しても、潜在的なセキュリティ問題がありますので、デフォルトではPUBLICアクセスは取り除かれています。

F.24.1. The pg_buffercacheビュー

ビューによって公開されている列の定義を[表F-14](#)に示します。

表 F-14. pg_buffercacheの列

名前	型	参照	説明
bufferid	integer		1からshared_buffersまでの範囲で示されるID
relfilenode	oid	pg_class.relfilenode	リレーシヨンのファイルノード番号
reltablespace	oid	pg_tablespace.oid	リレーシヨンのテーブル空間OID
reldatabase	oid	pg_database.oid	リレーシヨンのデータベースOID
relblocknumber	bigint		リレーシヨン内のページ番号
relforknumber	smallint		リレーシヨン内のフォーク番号
isdirty	boolean		ダーティページかどうか
usagecount	smallint		ページのLRU数

共有キャッシュ内の各バッファに対して、1行が存在します。未使用のバッファは、bufferidを除き、すべてのフィールドがNULLになります。共有システムカタログは、OIDがゼロのデータベースに属するものとして表示されます。

キャッシュはすべてのデータベースで共有されているため、現在のデータベースに属さないリレーシヨンのページも表示されます。これは、一部の行に対して一致するpg_classの結合行が存在しない、間違った結合をしてしまう可能性すらあることを意味します。pg_classに対して結合しようとする場合、現在のデータベースのOIDまたは0と等しいreldatabaseを持つ行に限定して結合することをお勧めします。

pg_buffercacheビューにアクセスがあると、ビューが表示するすべてのバッファ状態をコピーするために十分な期間、内部バッファマネージャはロックを取得します。これにより、一貫した結果集合が生成されることが、また、必要以上に長く通常のバッファ操作がブロックされることが保証されます。とは言え、このビューが頻繁に読み取られると、データベース性能に多少影響が発生する可能性があります。

F.24.2. サンプル出力

```
regression=# SELECT c.relname, count(*) AS buffers
              FROM pg_buffercache b INNER JOIN pg_class c
              ON b.relfilenode = pg_relation_filenode(c.oid) AND
                 b.reldatabase IN (0, (SELECT oid FROM pg_database
                                     WHERE datname = current_database()))
              GROUP BY c.relname
              ORDER BY 2 DESC
              LIMIT 10;
```

relname	buffers
tenk2	345
tenk1	141
pg_proc	46
pg_class	45
pg_attribute	43
pg_class_relname_nsp_index	30
pg_proc_prname_args_nsp_index	28
pg_attribute_relid_attnam_index	26
pg_depend	22
pg_depend_reference_index	20

(10 rows)

F.24.3. 作者

Mark Kirkwood <markir@paradise.net.nz>

設計協力: Neil Conway <neilc@samurai.com>

デバッグのアドバイス: Tom Lane <tgl@sss.pgh.pa.us>

[前のページ](#)
パスワード検査

[ホーム](#)
[上に戻る](#)

[次のページ](#)
pgcrypto

F.25. pgcrypto

pgcryptoモジュールはPostgreSQL用の暗号関数を提供します。

F.25.1. 汎用ハッシュ関数

F.25.1.1. digest()

```
digest(data text, type text) returns bytea
digest(data bytea, type text) returns bytea
```

与えられたdataのバイナリハッシュを計算します。typeは使用するアルゴリズムです。標準アルゴリズムはmd5、sha1、sha224、sha256、sha384、およびsha512です。pgcryptoがOpenSSL付きで構築された場合、[表F-18](#)で詳解する、より多くのアルゴリズムを利用することができます。

ダイジェストを16進数表記の文字列としたい場合は、結果に対してencode()を使用してください。以下に例を示します。

```
CREATE OR REPLACE FUNCTION sha1(bytea) returns text AS $$
    SELECT encode(digest($1, 'sha1'), 'hex')
$$ LANGUAGE SQL STRICT IMMUTABLE;
```

F.25.1.2. hmac()

```
hmac(data text, key text, type text) returns bytea
hmac(data bytea, key text, type text) returns bytea
```

keyをキーとしたdataのハッシュ化MACを計算します。typeはdigest()の場合と同じです。

digest()と似ていますが、ハッシュはキーを知った上で再計算されるだけです。これは、誰かがデータを変更し、同時に一致するようにハッシュを変更するという状況を防ぎます。

キーがハッシュブロックサイズより大きい場合、まずハッシュ化され、その結果をキーとして使用します。

F.25.2. パスワードハッシュ化関数

crypt()およびgen_salt()関数は特にパスワードのハッシュ化のために設計されたものです。crypt()がハッシュ処理を行い、gen_salt()はハッシュ処理用のアルゴリズム上のパラメータを準備します。

crypt()アルゴリズムは、以下の点でMD5やSHA1のような通常のハッシュ処理アルゴリズムと異なります。

- 1 11 低速です。データ量が少ないためパスワード総当たり攻撃に対して頑健にする唯一の方法です。
- 1 11 結果にはソルトというランダムな値が含まれます。このため同じパスワードのユーザでも異なった暗号化パスワードを持ちます。これはアルゴリズムの逆処理に対する追加の防御です。
- 1 11 結果内にアルゴリズムの種類が含まれます。このため異なるアルゴリズムでハッシュ化したパスワードが混在可能です。
- 1 11 一部は適応型です。つまり、コンピュータが高速になったとしても、既存のパスワードとの互換性を損なうことなくアルゴリズムを低速に調整することができます。

crypt()関数がサポートするアルゴリズムを[表F-15](#)に列挙します。

表 F-15. crypt()がサポートするアルゴリズム

アルゴリズム	パスワード最大長	適応型かどうか	ソルトビット長	説明
bfb	72	はい	128	Blowfishベース、2a版
md5	無制限	いいえ	48	MD5ベースの暗号
xdes	8	はい	24	拡張DES
des	8	いいえ	12	元来のUNIX crypt

F.25.2.1. crypt()

crypt(password text, salt text) returns text

passwordのcrypt(3)形式のハッシュを計算します。新しいパスワードを保管する時には、gen_salt()を使用して新しいsaltを生成する必要があります。パスワードを検査する時、既存のハッシュ値をsaltとして渡し、結果が格納された値と一致するかどうかを確認します。

新しいパスワードの設定例を以下に示します。

```
UPDATE ... SET pswhash = crypt('new password', gen_salt('md5'));
```

認証の例です。

```
SELECT pswhash = crypt('entered password', pswhash) FROM ... ;
```

入力パスワードが正しければtrueを返します。

F.25.2.2. gen_salt()

gen_salt(type text [, iter_count integer]) returns text

crypt() で使用するランダムなソルト文字列を新規に生成します。 また、このソルト文字列はcrypt() にどのアルゴリズムを使用するかを通知します。

typeパラメータはハッシュ化アルゴリズムを指定します。 受付可能な種類は、des、xdes、md5、bf です。

繰り返し回数を持つアルゴリズムでは、ユーザはiter_countパラメータを使用して繰り返し回数を指定できます。 指定する回数を高くすれば、パスワードのハッシュ化にかかる時間が長くなり、それを破るための時間も長くなります。 しかし、あまりに多くの回数を指定すると、ハッシュ計算にかかる時間は数年に渡ってしまう可能性があります。 これは実用的ではありません。 iter_countパラメータを省略した場合、デフォルトの繰り返し回数が使用されます。 iter_countで受け付けられる値はアルゴリズムに依存し、表F-16に示す通りです。

表 F-16. crypt() 用の繰り返し回数

アルゴリズム	デフォルト	最小	最大
xdes	725	1	16777215
bf	6	4	31

xdesの場合は他にも、回数が奇数でなければならないという制限があります。

適切な繰り返し回数を選択するために、元々のDES暗号は当時のハードウェアで1秒あたり4個のハッシュを持つことができるように設計されたことを考えてください。 毎秒4ハッシュより遅いと、おそらく使い勝手が悪いでしょう。 毎秒100ハッシュより速いというのは、十中八九、あまりにも速すぎるでしょう。

ハッシュ化アルゴリズム別に相対的な速度に関する概要を表F-17にまとめました。 この表は、8文字のパスワード内のすべての文字の組合せを取るためにかかる時間を示します。 また、すべて小文字の英字のみのパスワードである場合と大文字小文字が混在した英字と数字のパスワードの場合を仮定します。 crypt-bfの項では、スラッシュの後の数値はgen_saltのiter_countです。

表 F-17. ハッシュアルゴリズムの速度

アルゴリズム	1秒当たりのハッシュ数	[a-z]の場合	[A-Za-z0-9]の場合
crypt-bf/8	28	246年	251322年
crypt-bf/7	57	121年	123457年
crypt-bf/6	112	62年	62831年
crypt-bf/5	211	33年	33351年
crypt-md5	2681	2.6年	2625年
crypt-des	362837	7日	19年
sha1	590223	4日	12年
md5	2345086	1日	3年

注意：

- Pentium 4 1.5GHzのマシンを使用しました。
- `crypt-des`および`crypt-md5`アルゴリズムの数値はJohn the Ripper v1.6.38の`-test`出力から得たものです。
- `md5`の数値はmdcrack 1.2のものです。
- `sha1`の数値はlcrack-20031130-betaのものです。
- `crypt-bf`の数は、1000個の8文字パスワードをループする単純なプログラムを使用して得たものです。こうして、異なる回数の速度を示すことができました。参考までに、`john -test`は`crypt-bf/5`で213 loops/secでした。（結果の差異が非常に小さいことは、pgcryptoにおける`crypt-bf`実装がJohn the Ripperで使用されるものと同じであるという事実と一致します。）

"すべての組み合わせを試行する"ことは現実的な行使ではありません。通常パスワード推定は、普通の単語とその変形の両方を含む辞書を使用して行われます。ですので、いささかなりとも言葉に似たパスワードは上で示した数値よりも速く推定されます。また6文字の単語に似ていないパスワードは推定を免れるかもしれませんが、免れないかもしれません

F.25.3. PGP暗号化関数

ここで示す関数はOpenPGP（RFC 4880）標準の暗号処理部分を実装します。対称キーおよび公開キー暗号化がサポートされます。

暗号化されたPGPメッセージは次の2つの部品（またはパケット）から構成されます。

- セッションキーを含むパケット。対称キーまたは公開キー暗号化の両方。
- セッションキーにより暗号化されたデータを含むパケット。

対称キー（つまりパスワード）で暗号化する場合

- 1 11 与えられたパスワードはString2Key(S2K)アルゴリズムでハッシュ化されます。これはどちらかというと`crypt()`アルゴリズムと似て、意図的に低速で、かつランダムなソルトを使用します。しかし、全長のバイナリキーを生成します。
- 1 11 分離したセッションキーが要求された場合、新しいランダムなキーが生成されます。さもなければ、S2Kキーがそのままセッションキーとして使用されます。
- 1 11 S2Kキーがそのまま使用される場合、S2K設定のみがセッションキーパケットに格納されます。さもなければ、セッションキーはS2Kキーで暗号化され、セッションキーパケットに格納されます。

公開キーで暗号化する場合

- 1 11 新しいランダムなセッションキーが生成されます。
- 1 11 これは公開キーを使用して暗号化され、セッションキーパケットに格納されます。

どちらの場合でもデータ暗号化は以下のように処理されます。

- 1 11 省略可能なデータ操作として、圧縮、UTF-8への変換、改行の変換があります。
- 1 11 データの前にはランダムなバイト数のブロックが付きます。これはrandom IVを使用する場合と同じです。
- 1 11 ランダムな前置ブロックとデータのSHA1ハッシュが後に付けられます。
- 1 11 これをすべてセッションキーで暗号化し、データパケットに格納します。

F.25.3.1. `pgp_sym_encrypt()`

```
pgp_sym_encrypt(data text, psd text [, options text ]) returns bytea
pgp_sym_encrypt_bytea(data bytea, psd text [, options text ]) returns bytea
```

対称PGPキー`psd`で`data`を暗号化します。`options`パラメータには後述のオプション設定を含めることができます。

F.25.3.2. `pgp_sym_decrypt()`

```
pgp_sym_decrypt(msg bytea, psd text [, options text ]) returns text
pgp_sym_decrypt_bytea(msg bytea, psd text [, options text ]) returns bytea
```

対称キーで暗号化されたPGPメッセージを復号化します。

`pgp_sym_decrypt`で`bytea`型のデータを復号化することはできません。これは無効な文字データの出力を防止するためです。元のテキストのデータを`pgp_sym_decrypt_bytea`で復号化することが正しい方法です。

`options`パラメータには後述のオプション設定を含めることができます。

F.25.3.3. `pgp_pub_encrypt()`

```
pgp_pub_encrypt(data text, key bytea [, options text ]) returns bytea
pgp_pub_encrypt_bytea(data bytea, key bytea [, options text ]) returns bytea
```

公開PGPキー`key`で`data`を暗号化します。この関数に秘密キーを与えるとエラーになります。

`options`パラメータには後述のオプション設定を含めることができます。

F.25.3.4. `pgp_pub_decrypt()`

```
pgp_pub_decrypt(msg bytea, key bytea [, psw text [, options text ]]) returns text
pgp_pub_decrypt_bytea(msg bytea, key bytea [, psw text [, options text ]]) returns bytea
```

公開キーで暗号化されたメッセージを復号化します。`key`は、暗号化に使用した公開キーに対応する秘密キーでなければなりません。秘密キーがパスワードで保護されている場合は、そのパスワードを`psw`で指定しなければなりません。パスワードはないが、オプションを指定したい場合は空のパスワードを指定する必要があります。

`pgp_pub_decrypt`で`bytea`型のデータを復号化することはできません。これは無効な文字データの出力を防止するためです。元のテキストのデータを`pgp_pub_decrypt_bytea`で復号化することが正しい方法です。

`options`パラメータには後述のオプション設定を含めることができます。

F.25.3.5. `pgp_key_id()`

```
pgp_key_id(bytea) returns text
```

`pgp_key_id`はPGP公開キーまたは秘密キーのキーIDを取り出します。暗号化されたメッセージが指定された場合は、データの暗号化に使用されたキーIDを与えます。

2つの特殊なキーIDを返すことがあります。

- `SYMKEY`

メッセージは対称キーで暗号化されました。

- `ANYKEY`

メッセージは公開キーで暗号化されましたが、キーIDが消去されていました。つまり、どれで復号化できるかを判定するためにはすべての秘密キーを試行しなければならないことを意味します。`pgcrypto`自身はこうしたメッセージを生成しません。

異なるキーが同一IDを持つ場合があることに注意してください。これは稀ですが、正常なイベントです。この場合クライアントアプリケーションはどちらが当てはまるかを調べるために、`ANYKEY`の場合と同様に、それぞれのキーで復号化を試行しなければなりません。

F.25.3.6. `armor()`, `dearmor()`

```
armor(data bytea) returns text
dearmor(data text) returns bytea
```

PGPのASCIIアーマー形式にデータを隠す、または、データを取り出します。ASCIIアーマーは基本的にCRC付きのBASE64という形式で、追加のフォーマットがあります。

F.25.3.7. PGP関数用のオプション

オプションはGnuPGに似せて命名しています。オプションの値は等号記号の後に指定しなければなりません。複数のオプションはカンマで区切ってください。以下に例を示します。

```
pgp_sym_encrypt(data, psw, 'compress-algo=1, cipher-algo=aes256')
```

`convert-crlf`を除くすべてのオプションは暗号化関数にのみ適用可能です。復号化関数はPGPデータからこうしたパラメータを入手します。

もっとも興味深いオプションはおそらく `compress-algo` と `unicode-mode` でしょう。残りはデフォルトで問題ないはずです。

F.25.3.7.1. cipher-algo

使用する暗号アルゴリズム。

値: `bf`, `aes128`, `aes192`, `aes256` (OpenSSL-only: `3des`, `cast5`)

デフォルト: `aes128`

適用範囲: `pgp_sym_encrypt`, `pgp_pub_encrypt`

F.25.3.7.2. compress-algo

使用する圧縮アルゴリズム。PostgreSQLがzlib付きで構築されている場合のみ利用可能です。

値:

0 - 非圧縮

1 - ZIP圧縮

2 - ZLIB圧縮 (ZIPにメタデータとブロックCRCを加えたもの)

デフォルト: 0

適用範囲: `pgp_sym_encrypt`, `pgp_pub_encrypt`

F.25.3.7.3. compress-level

どの程度圧縮するかです。レベルが大きい程小さくなりますが、低速になります。0は圧縮を無効にします。

値: 0, 1-9

デフォルト: 6

適用範囲: `pgp_sym_encrypt`, `pgp_pub_encrypt`

F.25.3.7.4. convert-crlf

暗号化の際に `\n` を `\r\n` に、復号化の際に `\r\n` を `\n` に変換するかどうか。RFC 4880では、テキストデータは改行コードとして `\r\n` を使用して格納すべきであると規定されています。完全にRFC準拠の動作を行いたければ、これを使用してください。

値: 0, 1

デフォルト: 0

適用範囲: `pgp_sym_encrypt`, `pgp_pub_encrypt`, `pgp_sym_decrypt`, `pgp_pub_decrypt`

F.25.3.7.5. disable-mdc

データをSHA-1で保護しません。このオプションを使用することが良い唯一の理由は、SHA-1で保護されたパケットがRFC 4880に追加される前の、古いPGP製品との互換性を得るためです。最近のgnupg.orgおよびpgp.comのソフトウェアではこれを正しくサポートしています。

値: 0, 1

デフォルト: 0

適用範囲: `pgp_sym_encrypt`, `pgp_pub_encrypt`

F.25.3.7.6. enable-session-key

分離したセッションキーを使用します。公開キー暗号では常に分離したセッションキーを使用します。これは対称キー暗号向けのもので、デフォルトではS2Kをそのまま使用します。

値: 0, 1

デフォルト: 0

適用範囲: `pgp_sym_encrypt`

F.25.3.7.7. s2k-mode

使用するS2Kアルゴリズム。

値:

0 - ソルト無。危険です

1 - ソルト有。固定繰り返し回数。

3 - 可変繰り返し回数

デフォルト: 3

適用範囲: `pgp_sym_encrypt`

F.25.3.7.8. s2k-digest-algo

S2K計算で使用するダイジェストアルゴリズム。

値: md5, sha1

デフォルト: sha1

適用範囲: pgp_sym_encrypt

F.25.3.7.9. s2k-cipher-algo

分離したセッションキーの暗号化に使用する暗号。

値: bf, aes, aes128, aes192, aes256

デフォルト: cipher-algoを使用

適用範囲: pgp_sym_encrypt

F.25.3.7.10. unicode-mode

テキストデータをデータベース内部符号化方式からUTF-8に変換して戻すかどうかです。 データベースがすでにUTF-8であれば、変換は起こらず、データにUTF-8としてタグが付くのみです。 このオプションがないと、何も行われません。

値: 0, 1

デフォルト: 0

適用範囲: pgp_sym_encrypt, pgp_pub_encrypt

F.25.3.8. GnuPGを使用したキーの生成

新しいキーを生成します。

```
gpg --gen-key
```

推奨するキー種類は"DSA and Elgamal"です。

RSA暗号化のためには、マスタとしてDSAまたはRSAで署名のみのキーを作成し、そして `gpg --edit-key` を使用してRSAで暗号化された副キーを追加しなければなりません

キーを列挙します。

```
gpg --list-secret-keys
```

ASCIIアーマー形式で公開キーをエクスポートします。

```
gpg -a --export KEYID > public.key
```

ASCIIアーマー形式の秘密キーをエクスポートします。

```
gpg -a --export-secret-keys KEYID > secret.key
```

PGP関数にこれらのキーを渡す前に `dearmor()` `を使用する必要があります。 バイナリデータを扱うことができる場合、`gpg` から `-a` フラグを省略することができます。

詳細は `man gpg`、<http://www.gnupg.org/gph/en/manual.html> The GNU Privacy Handbook、<http://www.gnupg.org> サイトの各種文書を参照してください。

F.25.3.9. PGPコードの制限

- 署名に関するサポートはありません。 これはまた、暗号化副キーがマスタキーに属しているかどうか検査しないことを意味します。
- マスタキーとして暗号化キーをサポートしません。 一般的にこうした状況は現実的ではありませんので、問題にならないはずです。
- 複数の副キーに関するサポートはありません。 よくありますので、これは問題になりそうに見えます。 一方、通常のGPG/PGPキーを `pgcrypto` で使用すべきではありません。 使用する状況が多少異なりますので新しく作成してください。

F.25.4. 暗号化そのものを行う関数

これらの関数はデータ全体を暗号化するためだけに実行します。 PGP暗号化の持つ先端的な機能はありません。 したがって、大きな問題がいくつか存在します。

1 11 暗号キーとしてユーザキーをそのまま使用します。

1 11 暗号化されたデータが変更されたかどうかを確認するための整合性検査をまったく提供しません。

1 11 ユーザが、IVをも含め暗号化パラメータ自体をすべて管理していることを想定しています。

1 11 テキストは扱いません。

このため、PGP暗号化の導入もあり、暗号化のみの関数はあまり使用されません。

```
encrypt(data bytea, key bytea, type text) returns bytea
decrypt(data bytea, key bytea, type text) returns bytea

encrypt_iv(data bytea, key bytea, iv bytea, type text) returns bytea
decrypt_iv(data bytea, key bytea, iv bytea, type text) returns bytea
```

`type` で指定した暗号方法を使用してデータの暗号化・復号化を行います。 `type` 文字列の構文は以下の通りです。

```
algorithm [ - mode ] [ /pad: padding ]
```

ここで`algorithm`は以下の1つです。

- `bf` — Blowfish
- `aes` — AES (Rijndael-128)

また`mode`は以下の1つです。

- `cbc` — 次のブロックは前ブロックに依存します（デフォルト）
- `ecb` — 各ブロックは独自に暗号化されます（試験用途のみ）

`padding`は以下の1つです。

- `pkcs` — データ長に制限はありません（デフォルト）
- `none` — データは暗号ブロックサイズの倍数でなければなりません

このため、例えば以下は同じです。

```
encrypt(data, 'fooz', 'bf')
encrypt(data, 'fooz', 'bf-cbc/pad:pkcs')
```

`encrypt_iv` および `decrypt_iv` では、`iv` パラメータはCBCモード用の初期値となります。 ECBでは無視されます。 正確にブロック長でない場合、切り抜かれるか、もしくはゼロで埋められます。 このパラメータがない場合、関数のデフォルト値はすべてゼロです。

F.25.5. ランダムデータ関数

```
gen_random_bytes(count integer) returns bytea
```

暗号論的に強いランダムな`count`バイトを返します。 一度に最大で1024バイトを抽出することができます。 ランダム性ジェネレータプールを空にすることを防止するためのものです。

F.25.6. 注釈

F.25.6.1. 構築

`pgcrypto`は自身で主PostgreSQLの`configure`スクリプトの検出結果に従って構築します。 構築に影響するオプションは`--with-zlib`と`--with-openssl`です。

ZLIB付きでコンパイルされた場合、PGP暗号化関数は暗号化前にデータを圧縮することができます。

OpenSSL付きでコンパイルされた場合、より多くのアルゴリズムが利用できるようになります。 また、OpenSSLがより最適化されたBIGNUM関数を持つため、公開キー暗号化関数は高速になります。

表 F-18. OpenSSLの有無による機能のまとめ

機能	組み込み	OpenSSL付き
MD5	○	○
SHA1	○	○
SHA224/256/384/512	○	○ (注1)
その他のダイジェストアルゴリズム	×	○ (注2)
Blowfish	○	○
AES	○	○ (注3)
DES/3DES/CAST5	×	○
暗号化そのもの	○	○
PGP対称暗号化	○	○
PGP公開キー暗号化	○	○

注

- 1 11 SHA2はOpenSSLバージョン0.9.8から含まれています。これより古いバージョンでは、pgcryptoは組み込みのコードを使用します。
- 1 11 OpenSSLがサポートする任意のダイジェストアルゴリズムが自動的に選択されます。これは、明示的にサポートされなければならない暗号では使用できません。
- 1 11 AESはOpenSSLバージョン0.9.7から含まれています。これより古いバージョンでは、pgcryptoは組み込みのコードを使用します。

F.25.6.2. NULLの扱い

標準SQLの通り、引数のいずれかがNULLの場合、すべての関数はNULLを返します。注意せずに使用すると、これがセキュリティ上の問題になるかもしれません。

F.25.6.3. セキュリティ上の制限

pgcryptoの関数はすべてデータベースサーバ内部で実行されます。これは、pgcryptoとクライアントアプリケーションとの間でやり取りされるデータはすべて平文であることを意味します。したがって、以下を行う必要があります。

- 1 11 ローカルまたはSSL接続で接続
- 1 11 システム管理者およびデータベース管理者を信頼

これらが不可能であれば、クライアントアプリケーション内で暗号化する方が望まれます。

F.25.6.4. 有用な文書

- <http://www.gnupg.org/gph/en/manual.html>
GNUプライバシーハンドブック
- <http://www.openwall.com/crypt/>
blowfish暗号アルゴリズムの説明
- <http://www.stack.nl/~galactus/remailers/passphrase-faq.html>
優れたパスワードの選び方
- <http://world.std.com/~reinhold/diceware.html>
パスワード決定に関する面白い考え
- <http://www.interhack.net/people/cmcurtin/snake-oil-faq.html>
良い暗号、悪い暗号に関する説明

F.25.6.5. 技術的な参考情報

- <http://www.ietf.org/rfc/rfc4880.txt>
OpenPGPメッセージフォーマット
- <http://www.ietf.org/rfc/rfc1321.txt>
MD5 メッセージダイジェストアルゴリズム
- <http://www.ietf.org/rfc/rfc2104.txt>
HMAC: Keyed-Hashing for Message Authentication.
- <http://www.usenix.org/events/usenix99/provos.html>
DES暗号、MD5暗号、bcryptアルゴリズムの比較
- <http://csrc.nist.gov/cryptval/des.htm>
DES、3DES、AES標準
- [http://en.wikipedia.org/wiki/Fortuna_\(PRNG\)](http://en.wikipedia.org/wiki/Fortuna_(PRNG))
Fortuna CSPRNGの説明
- <http://jlcooke.ca/random/>
Linux用Jean-Luc Cooke Fortunaに基づく /dev/random ドライバ
- <http://research.cyber.ee/~lipmaa/crypto/>
暗号学に関するリンク集

F.25.7. 作者

Marko Kreen <markokr@gmail.com>

pgcryptoは以下のソースを使用しています。

アルゴリズム	作者	元ソース
DES crypt	David Burren and others	FreeBSD libcrypt
MD5 crypt	Poul-Henning Kamp	FreeBSD libcrypt
Blowfish crypt	Solar Designer	www.openwall.com
Blowfish cipher	Simon Tatham	PuTTY
Rijndael cipher	Brian Gladman	OpenBSD sys/crypto
MD5 and SHA1	WIDE Project	KAME kame/sys/crypto
SHA256/384/512	Aaron D. Gifford	OpenBSD sys/crypto
BIGNUM math	Michael J. Fromberger	dartmouth.edu/~sting/sw/imath

F.26. pg_freespacemap

`pg_freespacemap` モジュールは、空き領域マップ (FSM) を検査する手法を提供します。 `pg_freespace` と呼ばれる関数、正確に言うと、二つの多重定義された関数を提供します。 これらの関数は、指定されたページ、あるいはリレーシヨンのすべてのページについての、空き領域マップに記録されている値を表示します。

デフォルトでは、潜在的なセキュリティ問題が存在するため、これらの関数の PUBLIC アクセスは無効にされています。

F.26.1. 関数

```
pg_freespace(rel regclass IN, blkno bigint IN) returns int2
```

FSMを参照して、`blkno` で指定されたリレーシヨンのページ上の空き領域のサイズを返します。

```
pg_freespace(rel regclass IN, blkno OUT bigint, avail OUT int2)
```

FSMを参照して、リレーシヨンの各ページの空き領域のサイズを表示します。 リレーシヨンの各ページに対して (`blkno` `bigint`, `avail` `int2`) が1タプルとなり、これらのタプルのセットが返却されます。

空き領域マップに格納された値は、正確ではありません。 これらの値は `BLCKSZ` (デフォルト `BLCKSZ` では32バイト) の1/256の精度で丸められ、また、タプルが挿入や更新されるのと同じように、完全に最新に保たれているわけではありません。

インデックスでは、ページ内の空き領域ではなく、完全に未使用のページが追跡されます。 したがって、その値には意味がなく、単にページが一杯か空かを表します。

注意: バージョン8.4で導入された新しいFSM実装を反映するために、同バージョンでインタフェースが変更されました。

F.26.2. サンプル出力

```
postgres=# SELECT * FROM pg_freespace('foo');
 blkno | avail 
-----+-----
      0 |      0
      1 |      0
      2 |      0
      3 |     32
      4 |    704
      5 |    704
      6 |    704
      7 |   1216
      8 |    704
      9 |    704
     10 |    704
     11 |    704
     12 |    704
     13 |    704
     14 |    704
     15 |    704
     16 |    704
     17 |    704
     18 |    704
     19 |   3648
(20 rows)

postgres=# SELECT * FROM pg_freespace('foo', 7);
 pg_freespace 
-----
           1216
(1 row)
```

F.26.3. 作者

オリジナルバージョンは Mark Kirkwood <markir@paradise.net.nz> によるものです。 バージョン8.4では、Heikki Linnakangas <heikki@enterprisedb.com> により、新しいFSM実装に合うよう書き直されました。

F.27. pgrowlocks

pgrowlocks モジュールは、指定したテーブルにおける行ロックの情報を示す関数を提供します。

F.27.1. 概要

pgrowlocks(text) returns setof record

パラメータはテーブルの名前です。結果はレコードの集合となり、各レコードはテーブル内のロックされた1行を示します。出力列は表F-19の通りです。

表 F-19. pgrowlocksの出力列

名前	型	説明
locked_row	tid	ロックされた行のタプルID (TID)
lock_type	text	共有ロックの場合はShared、排他ロックの場合はExclusive
locker	xid	ロックを獲得したトランザクションのトランザクションID、もしマルチトランザクションの場合はmultixact ID
multi	boolean	ロックをマルチトランザクションが獲得していた場合は真
xids	xid[]	ロックを獲得しているトランザクションのトランザクションID (マルチトランザクションの場合は複数)
pids	integer[]	ロックを獲得しているバックエンドのプロセスID (マルチトランザクションの場合は複数)

pgrowlocks は対象テーブルに対してAccessShareLockを獲得し、ロック情報の収集のために1行ずつ行を読み取ります。これは大規模テーブルにおいては高速とは言えません。以下に注意してください:

1 11 テーブル全体が他から排他ロックされている場合、pgrowlocks はブロックされます。

1 11 pgrowlocks では、自己矛盾のないスナップショットを生成することは保証されません。その実行中に、新しい行ロックが獲得されることも、古いロックが解放されることもあり得ます。

pgrowlocks は、ロックされた行の内容は表示しません。同時に行の内容を参照したい場合には、以下のようして実現することができます:

```
SELECT * FROM accounts AS a, pgrowlocks('accounts') AS p
WHERE p.locked_row = a.ctid;
```

しかし (PostgreSQL 8.3の時点では) こうした問い合わせが非常に非効率であることに注意してください。

F.27.2. サンプル出力

```
test=# SELECT * FROM pgrowlocks('t1');
 locked_row | lock_type | locker | multi |  xids  |      pids
-----+-----+-----+-----+-----+-----
      (0,1) | Shared   |    19 | t     | {804,805} | {29066,29068}
      (0,2) | Shared   |    19 | t     | {804,805} | {29066,29068}
      (0,3) | Exclusive |   804 | f     | {804}      | {29066}
      (0,4) | Exclusive |   804 | f     | {804}      | {29066}
(4 rows)
```

F.27.3. 作者

Tatsuo Ishii

F.28. pg_stat_statements

pg_stat_statementsモジュールは、サーバで実行されたすべてのSQL文の実行時の統計情報を記録する手段を提供します。

このモジュールは追加の共有メモリを必要とするため、`postgresql.conf`の[shared_preload_libraries](#)にpg_stat_statementsを追加してモジュールをロードしなければなりません。このことは、このモジュールを追加もしくは削除するには、サーバを再起動する必要があるということを意味しています。

F.28.1. pg_stat_statements ビュー

このモジュールによって収集された統計情報は、pg_stat_statementsというシステムビューを通して利用することができます。このビューは、1行に対して、それぞれ個々の問い合わせ、データベースID、およびユーザIDを含んでいます（モジュールが記録できるSQL文の最大数まで）。ビューの列は、[表F-20](#)に示す通りです。

表 F-20. pg_stat_statementsの列

名前	型	参照元	説明
userid	oid	pg_authid.oid	SQL文を実行したユーザのOID
dbid	oid	pg_database.oid	SQL文が実行されたデータベースのOID
query	text		代表的な文の文字列 (track_activity_query_size バイトまで)
calls	bigint		実行回数
total_time	double precision		SQL文の処理に費やした総時間（ミリ秒単位）
rows	bigint		SQL文によって取得された、あるいは影響を受けた行の総数
shared_blks_hit	bigint		SQL文によってヒットした共有ブロックキャッシュの総数
shared_blks_read	bigint		SQL文によって読み込まれた共有ブロックの総数
shared_blks_dirtied	bigint		文によりダーティ状態となった共有ブロックの総数
shared_blks_written	bigint		SQL文によって書き込まれた共有ブロックの総数
local_blks_hit	bigint		SQL文によってヒットしたローカルブロックキャッシュの総数
local_blks_read	bigint		SQL文によって読み込まれたローカルブロックの総数
local_blks_dirtied	bigint		SQL文によりダーティ状態となったローカルブロックの総数
local_blks_written	bigint		SQL文によって書き込まれたローカルブロックの総数
temp_blks_read	bigint		SQL文によって読み込まれた一時ブロックの総数
temp_blks_written	bigint		SQL文によって書き込まれた一時ブロックの総数
blk_read_time	double precision		SQL文がブロック読み取りに費やした、ミリ秒単位の総時間 (track_io_timing が有効な場合。無効であればゼロ)
blk_write_time	double precision		SQL文がブロック書き出しに費やした、ミリ秒単位の総時間 (track_io_timing が有効な場合。無効であればゼロ)

これらビューと関数pg_stat_statements_resetは、pg_stat_statements拡張機能をインストールすることで明示的にインストールされたデータベースのみで利用可能です。しかしながら、統計情報はpg_stat_statementsモジュールがサーバにロードされていれば、ビューの存在に関わらず、サーバのすべてのデータベースについて記録されます。

セキュリティ上の理由から、スーパーユーザ以外のユーザは、他のユーザによって実行された問い合わせの文字列を見ることができません。ただし、ユーザのデータベースにビューがインストールされている場合、統計情報についてはそれらユーザから見ることができます。

計画作成が可能な問い合わせ（つまりSELECT、INSERT、UPDATE、DELETE）は、内部のハッシュ計算に従った、同一の問い合わせ構造を持つ限り、1つのpg_stat_statements項目に組み合わせられます。典型的には、2つの問い合わせは、問い合わせの中に現れるリテラル定数の値以外、意味的に等価である場合、この目的では同一とみなされます。しかし、ユーティリティコマンド（つまりこの他のコマンドすべて）は問い合わせ文字列のテキストを基に厳密に比較されます。

他の問い合わせと合致させるために定数値が無視された場合、pg_stat_statementsの表示の中で定数は?に置換されます。問い合わせの残りのテキストは、pg_stat_statements項目に関連付いた特定のハッシュ値を持つ、1つ目の問い合わせのテキストで

す。

一部の状況では、見た目上異なるテキストを持つ問い合わせが1つの`pg_stat_statements`項目にまとめられることがあります。通常これは意味的に等しい問い合わせでのみ発生しますが、関連がない問い合わせが1つの項目にまとめられるハッシュ競合の可能性がわずかながら存在します。（しかしこれは別のユーザまたは別のデータベースに属する問い合わせでは発生することはありません。）

ハッシュ値は問い合わせの解析後の表現に対して計算されますので、`search_path`の設定が異なる等の要因の結果として異なる意味を持つ場合、同じテキストを持つ問い合わせが別の項目として現れるという、反対もまたあります。

F.28.2. 関数

`pg_stat_statements_reset()` returns void

`pg_stat_statements_reset`は`pg_stat_statements`によってこれまでに収集したすべての統計情報を削除します。デフォルトでは、この関数はスーパーユーザのみ実行することができます。

F.28.3. 設定パラメータ

`pg_stat_statements.max` (integer)

`pg_stat_statements.max`は、このモジュールによって記録されるSQL文の最大数(すなわち、`pg_stat_statements`ビューにおける行の最大数)です。これを超過して異なるSQL文を検出した場合は、最も実行回数の低いSQL文の情報が捨てられます。デフォルトは1000です。このパラメータはサーバの起動時にのみ指定できます。

`pg_stat_statements.track` (enum)

`pg_stat_statements.track`は、どのSQL文をモジュールによって計測するかを制御します。`top`を指定した場合は（直接クライアントによって発行された）最上層のSQL文を記録します。`all`は（関数の中から呼び出された文などの）入れ子になった文も記録します。`none`は文に関する統計情報収集を無効にします。デフォルトは`top`です。この設定はスーパーユーザだけが変更できます。

`pg_stat_statements.track_utility` (boolean)

`pg_stat_statements.track_utility`は、このモジュールがユーティリティコマンドを記録するかどうかを指定します。ユーティリティコマンドとは、`SELECT`、`INSERT`、`UPDATE`および`DELETE`以外のすべてです。デフォルトは`on`です。この設定はスーパーユーザのみが変更できます。

`pg_stat_statements.save` (boolean)

`pg_stat_statements.save`は、サーバを終了させる際に文の統計情報を保存するかどうかを指定します。`off`の場合、統計情報は終了時に保存されず、サーバ開始時に再読み込みもされません。デフォルト値は`on`です。このパラメータは`postgresql.conf`ファイル、またはサーバコマンドラインでのみ設定できます。

このモジュールは、約`pg_stat_statements.max * track\_activity\_query\_size`バイトの追加の共有メモリを必要とします。`pg_stat_statements.track`に`none`が設定されていても、モジュールがロードされている限り常にこのメモリが消費されることに注意してください。

これらのパラメータは`postgresql.conf`の中で設定しなければなりません。典型的な使用法は以下のようになります。

```
# postgresql.conf
shared_preload_libraries = 'pg_stat_statements'

pg_stat_statements.max = 10000
pg_stat_statements.track = all
```

F.28.4. サンプル出力

```
bench=# SELECT pg_stat_statements_reset();

$ pgbench -i bench
$ pgbench -c10 -t300 bench

bench=# \x
bench=# SELECT query, calls, total_time, rows, 100.0 * shared_blks_hit /
          nullif(shared_blks_hit + shared_blks_read, 0) AS hit_percent
          FROM pg_stat_statements ORDER BY total_time DESC LIMIT 5;
-[ RECORD 1 ]-----
query      | UPDATE pgbench_branches SET bbalance = bbalance + ? WHERE bid = ?;
calls      | 3000
```

```

total_time | 9609.001000000002
rows       | 2836
hit_percent | 99.9778970000200936
-[ RECORD 2 ]-----
query      | UPDATE pgbench_tellers SET tbalance = tbalance + ? WHERE tid = ?;
calls      | 3000
total_time | 8015.156
rows       | 2990
hit_percent | 99.9731126579631345
-[ RECORD 3 ]-----
query      | copy pgbench_accounts from stdin
calls      | 1
total_time | 310.624
rows       | 100000
hit_percent | 0.30395136778115501520
-[ RECORD 4 ]-----
query      | UPDATE pgbench_accounts SET abalance = abalance + ? WHERE aid = ?;
calls      | 3000
total_time | 271.7419999999997
rows       | 3000
hit_percent | 93.7968855088209426
-[ RECORD 5 ]-----
query      | alter table pgbench_accounts add primary key (aid)
calls      | 1
total_time | 81.42
rows       | 0
hit_percent | 34.4947735191637631

```

F.28.5. 作者

Takahiro Itagaki <itagaki.takahiro@oss.ntt.co.jp>。 Peter Geoghegan <peter@2ndquadrant.com>により問い合わせの正規化が追加されました。

[前のページ](#)
pgrowlocks

[ホーム](#)
[上に戻る](#)

[次のページ](#)
pgstattuple

F.29. pgstattuple

pgstattupleモジュールはタプルレベルの統計情報を入手するための各種関数を提供します。

F.29.1. 関数

pgstattuple(text) returns record

pgstattupleはリレーシヨンの物理的な長さ、"不要"なタプルの割合、およびその他の情報を返します。これはバキュームが必要かどうかユーザが判断する時に有用かもしれません。引数は対象とするリレーシヨンの名前（スキーマ修飾可）です。以下に例を示します。

```
test=> SELECT * FROM pgstattuple('pg_catalog.pg_proc');
-[ RECORD 1 ]-----+-----
table_len          | 458752
tuple_count        | 1470
tuple_len          | 438896
tuple_percent      | 95.67
dead_tuple_count   | 11
dead_tuple_len     | 3157
dead_tuple_percent | 0.69
free_space         | 8932
free_percent       | 1.95
```

出力列を[表F-21](#)で説明します。

表 F-21. pgstattupleの出力列

列	型	説明
table_len	bigint	リレーシヨンのバイト単位の物理長
tuple_count	bigint	有効なタプル数
tuple_len	bigint	有効なタプルの物理長（バイト単位）
tuple_percent	float8	有効タプルの割合
dead_tuple_count	bigint	無効なタプル数
dead_tuple_len	bigint	バイト単位の総不要タプル長
dead_tuple_percent	float8	不要タプルの割合
free_space	bigint	バイト単位の総空き領域
free_percent	float8	空き領域の割合

pgstattuple はリレーシヨン上で読み取りロックのみを獲得します。ですので、結果はこの瞬間のスナップショットを考慮しません。つまり、同時実行の更新がその結果に影響を与えます。

pgstattuple は、HeapTupleSatisfiesNowが偽を返すかどうかで、タプルが"不要"かどうか判定します。

pgstattuple(oid) returns record

OIDで対象リレーシヨンを指定する点を除き、これはpgstattuple(text)と同じです。

pgstatindex(text) returns record

pgstatindexはB-treeインデックスに関する情報を示すレコードを返します。以下は例です。

```
test=> SELECT * FROM pgstatindex('pg_cast_oid_index');
-[ RECORD 1 ]-----+-----
version          | 2
tree_level       | 0
index_size       | 8192
root_block_no    | 1
internal_pages   | 0
leaf_pages       | 1
empty_pages      | 0
deleted_pages    | 0
avg_leaf_density | 50.27
leaf_fragmentation | 0
```

出力列は以下の通りです。

列	型	説明
version	integer	B-treeバージョン番号
tree_level	integer	ルートページのツリーレベル
index_size	bigint	インデックス内の総ページ数
root_block_no	bigint	ルートブロックの場所
internal_pages	bigint	"内部"（上位レベル）ページ数
leaf_pages	bigint	リーフページ数
empty_pages	bigint	空ページ数
deleted_pages	bigint	削除ページ数
avg_leaf_density	float8	リーフページの平均密度
leaf_fragmentation	float8	リーフページの断片化

pgstattupleでは、結果はページ毎に累積されます。この瞬間のインデックス全体のスナップショットが存在すると想定してはいけません。

pg_relpages(text) returns bigint

pg_relpagesはリレーション内のページ数を返します。

F.29.2. 作者

Tatsuo Ishii and Satoshi Nagayasu

[前のページ](#)
pg_stat_statements

[ホーム](#)
[上に戻る](#)

[次のページ](#)
pg_trgm

F.30. pg_trgm

pg_trgmモジュールは、類似文字列の高速検索をサポートするインデックス演算子クラスだけではなく、トリグラム一致に基づくASCII英数字の類似度の決定に関する関数と演算子も提供します。

F.30.1. トリグラム（またはトリグラフ）の概念

トリグラムは文字列から3つの連続する文字を取り出したグループです。共有するトリグラムの個数を数えることで、2つの文字列の類似度を測定することができます。この単純な考えが、多くの自然言語における単語の類似度を測定する際に非常に効率的であることが判明しています。

注意: 文字列内に含まれるトリグラム集合を決める際、文字列の前に2つの空白、後に1つの空白が付いているものとみなされます。例えば、"cat"という文字列のトリグラム集合は、" c"、" ca"、"cat"、"at "です。

F.30.2. 関数と演算子

pg_trgmモジュールで提供される関数を表F-22に、演算子を表F-23に示します。

表 F-22. pg_trgm関数

関数	戻り値	説明
similarity(text, text)	real	2つの引数がどの程度似ているかを示す数値を返します。結果はゼロ（2つの文字列にまったく類似度がないことを示す）から1（2つの文字列が同一であることを示す）までの範囲です。
show_trgm(text)	text[]	与えられた文字列内のすべてのトリグラムからなる配列を返します。（実際これはデバッグ時を除いて役に立つことはほぼありません。）
show_limit()	real	%演算子で使用する現在の類似度閾値を返します。これは、例えば2つの単語それぞれでミススペルがあったとしても類似しているものとみなす、その2つの単語間の最低の類似度を設定します。
set_limit(real)	real	%演算子で使用する現在の類似度閾値を設定します。閾値は0から1までの値でなければなりません（デフォルトは0.3です）。渡された値と同じ値が返ります。

表 F-23. pg_trgm演算子

演算子	戻り値	説明
text % text	boolean	2つの引数がset_limitで設定された類似度閾値以上の類似度を持つ場合trueを返します。
text <-> text	real	引数間の"距離"、つまり1 - similarity()の値を返します。

F.30.3. インデックスサポート

pg_trgmモジュールは、テキスト列全体に非常に高速な類似度検索を行うためのインデックスを作成することができるように、GiSTおよびGINインデックス演算子クラスを提供します。これらのインデックス種類は上記類似度演算子をサポートし、さらにLIKEおよびILIKE問い合わせにおけるトリグラムを基にしたインデックス検索をサポートします。（これらのインデックスは等価性や単純な比較演算子をサポートしません。ですので通常のB-treeインデックスも必要になるかもしれません。）

例：

```
CREATE TABLE test_trgm (t text);
CREATE INDEX trgm_idx ON test_trgm USING gist (t gist_trgm_ops);
```

または

```
CREATE INDEX trgm_idx ON test_trgm USING gin (t gin_trgm_ops);
```

この段階で、テキスト列tに類似度検索で使用可能なインデックスがあります。典型的な問い合わせを以下に示します。

```
SELECT t, similarity(t, 'word') AS sml
FROM test_trgm
WHERE t % 'word'
ORDER BY sml DESC, t;
```


これは`word`に十分似たテキスト列の値をすべて、類似度の高い順番に返します。インデックスは非常に大規模なデータ群に対する高速操作を行うために使用されます。

以下は上の問い合わせを変形させたものです。

```
SELECT t, t <-> 'word' AS dist
FROM test_trgm
ORDER BY dist LIMIT 10;
```

これはGINインデックスではなくGiSTインデックスにより非常に効率的に実装することができます。通常、類似度が高いものの中から少ない個数のみを必要とする場合、1番目の式よりも効率的です。

PostgreSQL 9.1から、これらのインデックス種類は`LIKE`および`ILIKE`におけるインデックス検索をサポートします。以下に例を示します。

```
SELECT * FROM test_trgm WHERE t LIKE '%foo%bar';
```

インデックス検索は、検索文字列からトリグラムを抽出し、それらをインデックスから検索することによって動作します。検索文字列内のトリグラムが多ければ、よりインデックス検索が効率的になります。B-treeを基にした検索とは異なり、検索文字列の左側が固定されている必要はありません。

GiSTまたはGINインデックスの選択は、他で説明されるGiSTとGINの相対的な性能特性に依存します。まとめとしては、GINインデックスはGiSTインデックスより検索が高速ですが、構築または更新は低速です。このためGINは静的なデータに、GiSTはよく更新されるデータに適しています。

F.30.4. テキスト検索の統合

トリグラム一致は全文テキストインデックスと一緒に使用する時、非常に有用なツールです。特に、全文検索機構では直接一致しない、ミススペルがある入力単語の認識を行うために役に立ちます。

第一段階は文書内で一意な単語からなる補助テーブルを生成することです。

```
CREATE TABLE words AS SELECT word FROM
ts_stat('SELECT to_tsvector(''simple'', bodytext) FROM documents');
```

ここで`documents`は、検索対象の`bodytext`テキストフィールドを持つテーブルです。言語固有の設定を使用するのではなく、`to_tsvector`関数で`simple`設定を使用する理由は、元の（語幹抽出されていない）単語のリストが欲しいためです。

次に`word`列にトリグラムインデックスを作成します。

```
CREATE INDEX words_idx ON words USING gin(word gin_trgm_ops);
```

これで、上の例に似た`SELECT`問い合わせを使用して、ユーザの検索語内でミススペルのある単語を提示できるようになります。有用な追加された試験は、選択された単語の長さがミススペルのある単語の長さとはほぼ同じになることを要求するものです。

注意: `words` テーブルは別に生成された静的なテーブルですので、文書群の更新に合理的に追従できるよう定期的に再生成する必要があります。正確に最新状態を維持する必要性は通常ありません。

F.30.5. 参考

GiST開発サイト <http://www.sai.msu.su/~megera/postgres/gist/>

Tsearch2開発サイト <http://www.sai.msu.su/~megera/postgres/gist/tsearch/V2/>

F.30.6. 作者

Oleg Bartunov <oleg@sai.msu.su>, Moscow, Moscow University, Russia

Teodor Sigaev <teodor@sigaev.ru>, Moscow, Delta-Soft Ltd., Russia

文書作成: Christopher Kings-Lynne

本モジュールはロシアモスクワのDelta-Soft Ltd.による後援です。

F.31. seg

本モジュールは線分、浮動小数点区間を表現するsegデータ型を実装します。 segは区間の終端内の不確定性を表すことができ、特に実験計測の表現に有用です。

F.31.1. 原理

計測の幾何は数値の連続におけるものより通常より複雑です。 計測は通常、多少あいまいな制限を持つ連続の部分部分となります。 不確実性と不規則性のため、さらに、タンパク質を安定させる温度範囲など計測される値は本質的に何らかの状態を示す区間となる可能性があるため、計測は区間として表れます。

一般的な見方を使うと、こうしたデータは値の組合せではなく区間としてデータを格納する方が便利なようです。 実際、ほとんどのアプリケーションでより効率的であると判明してさえいます。

一般的な見方をさらに進めると、制限の曖昧さは、伝統的な数値データ型を使用することで情報がある程度損失してしまうことを暗示しています。 これを考えてみましょう。 計測機器で6.50と読み取り、読み取ったデータをデータベースに格納します。 それを取り出す時にどうなるでしょう。 見てみましょう。

```
test=> select 6.50 :: float8 as "pH";
      pH
-----
6.5
(1 row)
```

計測という世界では6.50は6.5と同じではありません。 時としてこれが致命的な違いになる場合があります。 実験者は信頼する桁を書き出し（公開し）ます。 6.50は実質それ以上の値を含むあいまいな区間であり、（おそらく）その分布におけるもっとも特徴的なものを中央値として持つ6.5という区間よりもあいまいです。 私達は厳密にこうした異なるデータ項目が同じものとして現れることを好みません。

まとめ？ 任意の可変精度を持つ区間の制限を記録できる特別なデータ型を持つことは素晴らしいことでしょう。 変数はある意味、各データ要素が独自の精度を持ちます。

以下を見てください。

```
test=> select '6.25 .. 6.50'::seg as "pH";
      pH
-----
6.25 .. 6.50
(1 row)
```

F.31.2. 構文

区間の外部表現は、1つまたは2つの浮動小数点値を範囲演算子（..または...）で結び付けた形になります。 他にも、中央値と正負の偏差として指定することも可能です。 省略可能な確実性指示子（<、>、~）を格納することもできます。（しかし、確実性指示子はすべての組込みの演算子で無視されます。） 表F-24に許される表現についての概要を、表F-25にいくつか例を示します。

表F-24では、x、y、deltaは浮動小数点数値を表します。 delta以外のxとyの前に確実性指示子を付与することができます。

表 F-24. seg外部表現

x	単一値（幅0の区間）
x .. y	xからyまでの区間
x (+-) delta	x - deltaからx + deltaまでの区間
x ..	下限値xを持つ閉じていない区間
.. x	上限値xを持つ閉じていない区間

表 F-25. 有効なSEG入力例

5.0	幅0のセグメントを作成します（こうすると点になります）。
~5.0	幅0のセグメントを作成し、データ内に~を記録します。 ~はseg型の演算では無視されますが、コメントとして

	保持されます。
<5.0	5.0という点を作成します。 <は無視されますが、コメントとして保持されます。
>5.0	5.0という点を作成します。 >は無視されますが、コメントとして保持されます。
5 (+-) 0.3	4.7 .. 5.3という区間を作成します。 (+-) という記述は保持されないことに注意してください。
50 ..	50以上のすべて
.. 0	0以下のすべて
1.5e-2 .. 2E-2	0.015 .. 0.02という区間を作成します
1 ... 2	1...2、1 .. 2、1..2と同じです（範囲演算子前後の空白は無視されます）。

データソースで...が広く使用されるため、..の代わりの綴りとして許可されています。 残念なことにこれにより解析上の曖昧性が生じました。 0...23の上限が23なのか0.23なのかが明確ではありません。 これは、segの入力において、少なくとも1つの桁を数値内の小数点の前に書くことを要求することで解決されます。

健全性検査としてsegは、5 .. 2のような、下限値が上限値より大きな区間を拒絶します。

F.31.3. 精度

内部的にseg値には32ビット浮動小数点数値の組合せが格納されます。 これは7桁以上の有効桁を持つ数値が切り詰められることを意味します。

有効桁が正確に7桁、または7桁未満の数値は元の精度が保たれます。 つまり、問い合わせが0.00を返す場合、最後の0が書式付けのための誤差にならないことが確実です。 これは元のデータの精度を反映します。 最初から続く0は精度の影響を受けません。 0.0067は2有効桁のみを持つものと考えられます。

F.31.4. 使用方法

segモジュールにはseg値用のGiSTインデックス演算子クラスが含まれます。 GiST演算子クラスでサポートされる演算子を[表F-26](#)に示します。

表 F-26. Seg GiST 演算子

演算子	説明
[a, b] << [c, d]	[a, b]が完全に[c, d]の左側に存在します。 つまり、b <cならば[a, b] << [c, d]は真です。 さもなくば偽です。
[a, b] >> [c, d]	[a, b]が完全に [c, d]の右側に存在します。 つまり、a > dならば[a, b] >> [c, d]は真、さもなくば偽です。
[a, b] &< [c, d]	重なるまたは左側。 これは"右側にはみださない"ととらえた方がいいかもしれません。 b <= dならば真です。
[a, b] &> [c, d]	重なるか右側。 これは"左側にはみださない"ととらえた方がいいかもしれません。 a >= cならば真です。
[a, b] = [c, d]	同一。 セグメント[a, b]と [c, d]が同一、つまり、a == bかつc == dです。
[a, b] && [c, d]	セグメント[a, b]と[c, d]に重なりがあります。
[a, b] @> [c, d]	セグメント[a, b]はセグメント[c, d]を含有します。 つまり、a <= cかつb >= dです。
[a, b] <@ [c, d]	セグメント[a, b]はセグメント[c, d]に含有されます。 つまり、a >= cかつb <= dです。

（PostgreSQL 8.2以前では、含有演算子@>および<@はそれぞれ@および~という名前でした。 以前の名前もまだ利用できますが、廃止予定であり、いずれなくなります。 以前の名前は、コアの幾何データ型で以前従っていた規約と反対であることに注意してください。）

以下の例のように、標準B-Tree演算子も提供されます。

演算子	説明
[a, b] < [c, d]	より小さい
[a, b] > [c, d]	より大きい

これらの演算子はソート以外の実用的な目的に対して大きな意味を持ちません。これらの演算子はまず(a)と(c)を比べ、等しければ(b)と(d)を比べます。論理的にはほとんどの場合優れたソート処理と思えます。こうした型をORDER BYで使いたい場合に有用です。

F.31.5. 注釈

使用方法の例はリグレーション試験のsql/seg.sqlを参照してください。

(+-)を通常範囲に変換する機構は、境界で有効な桁数を決定するという点で完全に正確ではありません。例えば以下のように、結果の区間に10の冪乗が含まれる場合、下限値に余計な桁を追加します。

```
postgres=> select '10(+-)1'::seg as seg;
      seg
-----
9.0 .. 11          -- should be: 9 .. 11
```

R-Treeインデックスの性能は入力値の初期の順序に大きく依存する可能性があります。seg列で入力テーブルをソートすることは非常に役に立つでしょう。例としてsort-segments.plスクリプトを参照してください。

F.31.6. クレジット

原作者：Gene Selkov, Jr. <selkovjr@mcs.anl.gov>, Mathematics and Computer Science Division, Argonne National Laboratory.

GiST (<http://gist.cs.berkeley.edu/>)の要旨 (gist) を説明していただいたJoe Hellerstein博士 (<http://db.cs.berkeley.edu/jmh/>) に感謝します。また、自分の世界を作成できるようにし、静かに生活できるようにしてもらった、過去から現在までのすべてのpostgres開発者に感謝します。データベース研究に長年誠実にサポートしてくれたArgonne LabとU.S. Department of Energyにも感謝します。

[前のページ](#)
pg_trgm

[ホーム](#)
[上に戻る](#)

[次のページ](#)
sepgsql

F.32. sepysql

`sepysql` は、SELinuxのセキュリティポリシーに基づいた、ラベルベースの強制アクセス制御（MAC; Mandatory Access Control）機能を提供するモジュールです。

警告

現在の実装にはいくつかの重要な制限事項があり、そのため、全ての操作に対して強制アクセス制御を適用するわけではありません。詳細は [項F.32.7](#) をご覧ください。

F.32.1. 概要

このモジュールは、PostgreSQLが標準で提供しているものに加えて、SELinuxと統合されたアクセス制御のレイヤーを追加します。SELinuxの観点からは、このモジュールがPostgreSQLをユーザ空間オブジェクトマネージャとして機能することを可能にします。すなわち、DMLクエリによる個々のテーブルや関数へのアクセスは、オペレーティングシステムのセキュリティポリシーによってチェックされます。このチェックは、PostgreSQLによる通常のSQLパーミッションチェックに対して追加的に実施されます。

SELinuxにおけるアクセス制御の意思決定は、`system_u:object_r:sepysql_table_t:s0`のような書式を持ったセキュリティラベルと呼ばれる文字列を用いて行われます。個々のアクセス制御の意思決定には、2種類のラベルが利用されます。すなわち、ある操作を行おうとする主体（サブジェクト）のラベルと、その操作の対象となるオブジェクトのラベルです。これらのラベルはあらゆる種類のオブジェクトに対して適用されるため、（このモジュールを用いる事で）データベースに格納されたオブジェクトに対するアクセス制御は、他の一般的なオブジェクト、例えばファイルに対するものと同一の基準に従って意思決定される事になります。このデザインは、情報資産を格納する方法とは独立に、一元管理されたセキュリティポリシーによって情報資産を保護することを意図しています。

[SECURITY LABEL](#)を用いてデータベースオブジェクトにセキュリティラベルを設定することができます。

F.32.2. インストール

このモジュールはSELinuxが有効なLinuxカーネル 2.6.28 以上で動作します。その他のプラットフォーム上では利用する事はできません。加えて、`libselinux` 2.0.99以上、`selinux-policy` 3.9.13以上（ディストリビューションによっては、必要なルールを古いバージョンのポリシーにバックポートしているか可能性があります）が必要で、`--with-selinux` オプションによって明示的に有効化しなければいけません。

`sestatus` コマンドを用いてSELinuxの状態を確認することができます。典型的な出力例は以下の通りです。

```
$ sestatus
SELinux status:           enabled
SELinuxfs mount:         /selinux
Current mode:             enforcing
Mode from config file:    enforcing
Policy version:           24
Policy from config file:  targeted
```

SELinuxが無効化されている、あるいはインストールされていない場合、このモジュールのインストールの前に、SELinuxのセットアップを行わねばなりません。

このモジュールをビルドするには、PostgreSQLの`configure` コマンドに`--with-selinux` オプションを追加してください。これは、ビルド時に`libselinux-devel` パッケージがインストール されている事を確認します。

このモジュールを利用するには、[shared_preload Libraries](#) パラメータに `sepysql` を追記する必要があります。これ以外の方法でロードされた場合、このモジュールは機能しません。このモジュールのロード後、各データベースに対して`sepysql.sql`を実行し、セキュリティラベル管理のための関数のインストールや、初期セキュリティ ラベルの設定を行うべきです。

以下に`sepysql` 関数およびセキュリティラベルと共にデータベース クラスタを初期化する手順を示します。インストール先に応じて、適宜パス名を読み替えるようにしてください。

```
$ export PGDATA=/path/to/data/directory
$ initdb
$ vi $PGDATA/postgresql.conf
change
    #shared_preload_libraries = ''                                # (change requires restart)
```



```
to
    shared_preload_libraries = 'sepgsql'          # (change requires restart)
$ for DBNAME in template0 template1 postgres; do
    postgres --single -F -c exit_on_error=true $DBNAME ¥
    </usr/local/pgsql/share/contrib/sepgsql.sql >/dev/null
done
```

libselinuxとselinux-policyのバージョンによっては以下のようなメッセージが出力される事があります。

```
/etc/selinux/targeted/contexts/sepgsql_contexts: line 33 has invalid object type db_blobs
/etc/selinux/targeted/contexts/sepgsql_contexts: line 36 has invalid object type db_language
/etc/selinux/targeted/contexts/sepgsql_contexts: line 37 has invalid object type db_language
/etc/selinux/targeted/contexts/sepgsql_contexts: line 38 has invalid object type db_language
/etc/selinux/targeted/contexts/sepgsql_contexts: line 39 has invalid object type db_language
/etc/selinux/targeted/contexts/sepgsql_contexts: line 40 has invalid object type db_language
```

これらのメッセージは無害ですので無視してください。

インストール手順が正常に終了したら、通常通り、サーバを起動することができます。

F.32.3. リグレッションテスト

SELinuxの性質上、sepgsqlのリグレッションテストを実行するには、いくつかの追加的な設定が必要で、そのうちのいくつかはrootで実行する必要があります。リグレッションテストは通常のmake checkやmake installcheck コマンドで実行する事はできません。必要な設定を行い、テスト用スクリプトを手動で実行する必要があります。このテストはPostgreSQLビルドツリーのcontrib/sepgsqlディレクトリで実行する必要があります。しかしビルドツリーを必要とする一方、このテストはインストールされたサーバに対して実行する必要があります。これは、make checkではなく、make installcheckによく似ています。

最初に、[項F.32.2](#)に従ってsepgsqlをデータベースにセットアップします。使用するOS上のユーザは、認証無しでデータベース特権ユーザとして接続できる 必要があることに留意してください。

次に、リグレッションテスト用のポリシーパッケージのビルドとインストールを行ってください。sepgsql-regtest.ppはリグレッションテストの実行に必要な一連のルールを含む特別な目的のポリシーパッケージです。ポリシーのソースファイルであるsepgsql-regtest.teから、SELinuxの提供するMakefileを用いてmakeコマンドでビルドする事ができます。この時、インストール先システムにおいて、適切なMakefileの位置を指定する必要があります。以下の例で示されているパスは一例です。ビルドが完了したら、semoduleを用いてこのポリシーパッケージをインストールする事ができます。このコマンドは、指定されたポリシーパッケージをリンクし、カーネル空間にロードする役割を果たします。インストールが正常終了したら、semodule -lにより 有効なパッケージの一覧としてsepgsql-regtestが表示されるはずですが。

```
$ cd ../contrib/sepgsql
$ make -f /usr/share/selinux/devel/Makefile
$ sudo semodule -u sepgsql-regtest.pp
$ sudo semodule -l | grep sepgsql
sepgsql-regtest 1.04
```

次に、sepgsql_regression_test_modeブーリアンを有効化してください。安全のため、デフォルトではsepgsql-regtest.ppに含まれる全てのルールが有効化されている訳ではありません。sepgsql_regression_test_modeブーリアンはリグレッションテストを起動するためのいくつかのルールに関連付けされており、setsebool コマンドによって有効化する事ができます。

```
$ sudo setsebool sepgsql_regression_test_mode on
$ getsebool sepgsql_regression_test_mode
sepgsql_regression_test_mode --> on
```

次に、シェルがunconfined_tドメインで動作している事を確認して下さい。

idコマンドを使って現在のドメインを確認する事ができます。以下のように、シェルプロセスがunconfined_tドメインで動作している事を確認してください。

```
$ id -Z
unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
```

利用者の動作ドメインを設定する方法について、必要であれば、詳細は[項F.32.8](#)をご覧ください。

最後に、リグレッションテストのスクリプトを実行します。

```
$ ./test_sepgsql
```

このスクリプトは全ての設定ステップが正常に行われていることを確認し、その後、sepgsqlモジュールに対するリグレッションテストを実行します。

テストの実行後はsepgsql_regression_test_modeブーリアンを 無効化する事をお勧めします。

```
$ sudo setsebool sepgsql_regression_test_mode off
```

sepgsql-regtest ポリシーをアンロードする際は、以下のコマンドを 実行してください。

```
$ sudo semodule -r sepgsql-regtest
```

F.32.4. GUCパラメータ

sepgsql.permissive (boolean)

このパラメータにより、オペレーティングシステムの設定に関わらず、 sepgsql をパーミッシブモードで動作させる事ができます。 デフォルトの設定値はoffです。 postgresql.conf 内、およびサーバ起動時のコマンドラインでのみ、 このパラメータを設定する事ができます。

このパラメータがonの場合、たとえオペレーティングシステムがエンフォーシングモードで動作していたとしても、SE-PostgreSQLはパーミッシブモードで動作します。 このパラメータは主としてテストの目的に有用です。

sepgsql.debug_audit (boolean)

このパラメータにより、セキュリティポリシーの設定とは無関係に監査ログを出力する事が可能になります。 デフォルト値はoff（セキュリティポリシーの設定に従う）です。

SELinuxのセキュリティポリシーには、特定のアクセスを監査ログに記録するか否かを制御するルールも存在します。 デフォルトでは、ポリシーに違反したアクセスを記録し、それ以外はログに記録されません。

システムのポリシーとは無関係に、このパラメータは全ての監査ログの出力を強制します。

F.32.5. 機能

F.32.5.1. 制御されるオブジェクトの種類

SELinuxのセキュリティモデルでは、全てのアクセス制御 ルールは動作主体（サブジェクト；典型的にはデータベースクライアント）と 対象オブジェクト間の関係として記述し、これらはセキュリティラベルによって 識別されます。 ラベル付けされていないオブジェクトに対するアクセスが発生した場合、 そのオブジェクトはあたかもunlabeled_tタイプが割り当てられているかのように振舞います。

現在のsepgsqlでは、スキーマ、テーブル、カラム、シーケンス、ビューおよび関数に対するラベル付けがサポートされています。 sepgsqlの利用時には、これらのデータベースオブジェクトに対して、その生成時に自動的にセキュリティラベルが割り当てられます。 このラベルはデフォルトセキュリティラベルと呼ばれ、作成者のラベルと親関係にあたるオブジェクトのラベルに基づいて、システムのセキュリティポリシーが決定します。

新しいデータベースオブジェクトのラベルは、タイプ遷移と呼ばれる異なったラベルを設定するための特別なルールがセキュリティポリシーに設定されている場合を除き、親関係にあるオブジェクトのラベルを引き継ぎます。 スキーマの親オブジェクトはデータベースであり、テーブル、シーケンス、ビュー、および関数はその属するスキーマが、カラムはその属するテーブルが親オブジェクトという事になります。

F.32.5.2. DMLパーミッション

テーブルに対しては、構文の種類に応じてdb_table:select、db_table:insert、db_table:updateあるいはdb_table:delete権限が全ての被参照テーブルに対してチェックされます。 加えて、WHERE 句やRETURNING 句で参照されるカラム、又はUPDATEの際のデータ元として利用されるカラムの属するテーブルに対してdb_table:select権限もチェックされます。

参照された全てのカラムに対して列レベルの権限チェックが行われます。 SELECT構文で読み出されるカラムに対してだけでなく、他のDML構文で参照 されているカラムに対してもdb_column:select権限がチェックされます。 また、UPDATEやINSERTによって操作の対象となっているカラム に対しても、db_column:updateやdb_column:insert権限が それぞれチェックされます。

以下の例を見てください

```
UPDATE t1 SET x = 2, y = md5sum(y) WHERE z = 100;
```

ここでは、更新されるt1.xに対してdb_column:update権限が、更新と同時に参照されるt1.yに対してはdb_column:{select update}権限が、そして参照されるだけのt1.zにはdb_column:select権限がチェックされます。 また、テーブルレベルではdb_table:{select update}権限がチェックされます。

SELECT構文を用いてシーケンスを参照する場合、db_sequence:get_valueがチェックされます。 しかし、現在のところlastval() など関連する関数の実行時にはパーミッションチェックを行わない事に留意してください。

ビューに対してはdb_view:expand権限がチェックされ、次いで、ビューから展開されたオブジェクトに対するパーミッションが

個別にチェックされます。

関数に対して`db_procedure:{execute}`権限が定義されていますが、現バージョンではチェックされません。

クライアントは全ての被参照テーブル・カラムに対して参照の権限を有している必要があります。それらがビューに由来し、展開されたものであっても同様です。これにより、テーブルの内容がどのような方法によって参照されるかに関係なく、一貫したアクセス制御ルールを適用する事ができます。

データベーススーパーユーザに対して、標準のデータベース権限システムはDMLを用いたシステムカタログの更新と、TOASTテーブルの参照および更新を許していますが、`sepgsql`が有効なとき、これらの操作は禁止されます。

F.32.5.3. DDLパーミッション

オブジェクトの作成、変更、削除やセキュリティラベルの変更など、SELinuxは各オブジェクトに共通の操作を制御するためのパーミッションを何個か定義しています。また、特定のスキーマに対する名前の追加や削除など、いくつかのオブジェクトにはそれ固有の操作を制御するための特別なパーミッションも定義されています。

`CREATE`構文の実行時、オブジェクトタイプ毎の`create`権限が作成されたオブジェクトに対してチェックされます。新しく作成されたオブジェクトにはデフォルトのセキュリティラベルが割り当てられ、`create`権限は利用者とオブジェクト自身のセキュリティコンテキストの対に対してチェックされます。`CREATE TABLE`はテーブルとそれに含まれるカラムを同時に作成するものとして扱います。したがって、利用者はテーブルとカラムの両方を作成する権限を有している必要があります。

オブジェクトタイプによってはいくつか追加的なチェックが適用される場合があります。`CREATE DATABASE`においては、新しいデータベースに対する`create`権限だけでなく、その複製元あるいはテンプレートのデータベースに対する`getattr`権限もチェックされます。特定のスキーマに属するオブジェクト（テーブル、ビュー、シーケンスや関数）の作成においては、新しいオブジェクト自身に対する`create`権限だけでなく、スキーマに対する`add_name`権限もチェックされます。

`DROP`構文の実行時、削除されるオブジェクトに対してその種類に応じた`drop`権限がチェックされます。`CASCADE`により間接的に削除されるオブジェクトに対してもチェックは行われます。特定のスキーマに含まれるオブジェクト（テーブル、ビュー、シーケンスや関数）の削除に際しては、スキーマに対する`remove_name`権限も併せてチェックされます。

`SECURITY LABEL`コマンドの実行時、ラベル付けされるオブジェクトの古いラベルに対して`setattr`権限と`relabelfrom`権限が、入力された新しいラベルに対して`relabelto`権限がチェックされます。

複数のラベルプロバイダがインストールされており、利用者がSELinuxの管理下でないラベルを設定しようとした場合、`setattr`権限だけがチェックされるべきです。しかし実装上の制約により、現在はこれをチェックしていません。

F.32.5.4. トラストッド プロシジャ

トラストッド・プロシジャは`SECURITY DEFINER`関数や`Set-UID`コマンドに似ています。通常、機密データに対する高度にコントロールされたアクセス手段を提供する目的で、SELinuxは利用者のものとは異なるセキュリティラベルで信頼済みのコードを実行するための機能を持っています。関数がトラストッド・プロシジャとして振舞うかどうかは、関数のセキュリティラベルおよびオペレーティングシステムのセキュリティポリシーに従って決まります。例えば：

```
postgres=# CREATE TABLE customer (
            cid      int primary key,
            cname     text,
            credit     text
        );
CREATE TABLE
postgres=# SECURITY LABEL ON COLUMN customer.credit
            IS 'system_u:object_r:sepgsql_secret_table_t:s0';
SECURITY LABEL
postgres=# CREATE FUNCTION show_credit(int) RETURNS text
            AS 'SELECT regexp_replace(credit, '-'[0-9]+$',' -xxxx', 'g')
            FROM customer WHERE cid = $1'
            LANGUAGE sql;
CREATE FUNCTION
postgres=# SECURITY LABEL ON FUNCTION show_credit(int)
            IS 'system_u:object_r:sepgsql_trusted_proc_exec_t:s0';
SECURITY LABEL
```

これらの操作は管理権限を持つ利用者で行ってください。

```
postgres=# SELECT * FROM customer;
ERROR:  SELinux: security policy violation
postgres=# SELECT cid, cname, show_credit(cid) FROM customer;
 cid | cname | show_credit
-----+-----+-----
  1  | taro  | 1111-2222-3333-xxxx
  2  | hanako | 5555-6666-7777-xxxx
(2 rows)
```

この場合、一般の利用者はcustomer.creditを直接参照することはできませんが、トラステッド・プロシージャであるshow_creditを用いる事で、一部の桁がマスクされた顧客のクレジットカード番号をプリントする事が可能になります。

F.32.5.5. 動的ドメイン遷移

セキュリティポリシーによって許可されている場合、SELinuxの動的ドメイン遷移機能を用いて、利用者のラベルを新しいものに切り替える事ができます。 利用者のドメインはsetcurrent権限および、古いドメインから新しいドメインに遷移するためのdyntransition権限を有している必要があります。

利用者に自身のラベル、すなわち権限を切り替える事を可能にする動的ドメイン遷移は、システムによる自動切り替え（トラステッド・プロシージャの場合）に比べて慎重に取り扱う必要があります。 dyntransition権限は元々のドメインよりも小さな権限セットを持つドメインに切り替える場合에만安全であると考えられます。例えば、

```
regression=# select sepgsql_getcon();
              sepgsql_getcon
-----
unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
(1 row)

regression=# SELECT sepgsql_setcon('unconfined_u:unconfined_r:unconfined_t:s0-s0:c1.c4');
              sepgsql_setcon
-----
t
(1 row)

regression=# SELECT sepgsql_setcon('unconfined_u:unconfined_r:unconfined_t:s0-s0:c1.c1023');
ERROR:  SELinux: security policy violation
```

上記の例では、広いMCSレンジc0.c1023から狭いMCSレンジc1.c4への遷移は許可されているものの、その逆は禁止されています。

動的ドメイン遷移とトラステッド・プロシージャの組み合わせは、典型的なコネクションプーラのライフサイクルに適合する興味深い利用法を提供します。 たとえコネクションプーリングソフトウェアが大半のSQLの実行を許可されていない場合でも、トラステッド・プロシージャの内側からsepysql_setcon()関数を用いて利用者のセキュリティラベルを切り替える事ができます。（トラステッド・プロシージャは利用者のセキュリティラベルを切り替えるための認証情報を要求すべきです） この後、現在のセッションはコネクションプーラの権限ではなく、対象となる利用者の権限で動作する事になります。 また、sepysql_setcon()にNULL引数を与えて（適切な権限チェックを行う）トラステッド・プロシージャから再び呼び出す事で、コネクションプーラは後でセキュリティラベルを元に戻す事ができます。 ここでのポイントは、トラステッド・プロシージャだけが実際に有効なセキュリティラベルを変更する権限を持っており、正しい認証情報が与えられた場合にのみそれを実行するという事です。 言うまでもなく、安全な操作のためには、権限のないアクセスから認証情報を保持するテーブルや関数定義などを保護しなければなりません。

F.32.5.6. その他

ロードされたモジュールは、セキュリティポリシーの適用を容易にバイパスできるため、LOADコマンドの実行は全面的に禁止されています。

F.32.6. sepysql関数

表F-27に利用可能な関数を示します。

表 F-27. Sepysql 関数

sepysql_getcon() returns text	利用者のドメイン、つまり、現在の利用者ラベルを返却します。
sepysql_setcon(text) returns bool	セキュリティポリシーで許可されている場合、現在のセッションの利用者ドメインを新しいドメインへと切り替えます。 NULLを引数に取る事も可能で、その場合、元々の利用者ドメインへの遷移を意味します。
sepysql_mcstrans_in(text) returns text	mcstransデーモンが動作している場合、入力されたMLS/MCSレンジを修飾フォーマットから直接フォーマットに変換します。
sepysql_mcstrans_out(text) returns text	mcstransデーモンが動作している場合、入力されたMLS/MCSレンジを直接フォーマットから修飾フォーマットに変換します。
sepysql_restorecon(text) returns bool	現在のデータベース内の全てのオブジェクトに対して初期ラベルを割り当てます。引数はNULLもしくはシステムの標準に代わる定義ファイルの名前です。

F.32.7.

制限事項

ユーザ空間アクセスベクタ・キャッシュ

まだsepgsqlはユーザ空間アクセスベクタ・キャッシュをサポートしていません。この機能はパフォーマンスを改善するでしょう。

Data Definition Language (DDL) パーミッション

実装上の制約により、いくつかのDDL操作はパーミッションをチェックしません。

Data Control Language (DCL) パーミッション

実装上の制約により、DCL操作はパーミッションをチェックしません。

Row-level access control

PostgreSQLは行レベルアクセス制御をサポートしていません。したがって、sepgsqlも同様です。

隠れチャネル

たとえ利用者が参照を許可されていないオブジェクトであっても、sepgsqlはその存在を隠すことを意図していません。例えば、我々があるオブジェクトの内容を参照する事ができなくても、主キーの競合や外部キー違反、その他の方法によって不可視なオブジェクトが存在する事を推測できます。"最高機密"テーブルの存在を隠すことは不可能であり、その内容を秘匿することだけを意図しています。

F.32.8. 外部リソース

[SE-PostgreSQL Introduction](#)

このwikiページでは、概要、セキュリティ・デザイン、アーキテクチャ、管理、および将来の機能について紹介しています。

[Fedora SELinux User Guide](#)

このドキュメントはSELinuxシステム管理に対する広範な知識を提供しています。主としてFedoraを対象としています。が、それに限ったものではありません。

[Fedora SELinux FAQ](#)

このドキュメントはSELinuxに関するよくある質問と回答(FAQ)です。主としてFedoraを対象としていますが、それに限ったものではありません。

F.32.9. 著者

KaiGai Kohei <kaigai@ak.jp.nec.com>

[前のページ](#)

seg

[ホーム](#)

[上に戻る](#)

[次のページ](#)

spi

F.33. spi

spiモジュールは、SPIおよびトリガを使用した、動作可能な例を複数提供します。これらの関数は独自の何らかの価値を持つものですが、目的に応じて変更するための例としてより有用です。関数は任意のテーブルと使用できるほど一般的なものですが、トリガを作成する場合は（後述のように）テーブル名とフィールド名を指定する必要があります。

以下で説明する関数グループのそれぞれは、別々にインストールすることができる拡張として提供されます。

F.33.1. refint — 参照整合性を実装する関数

`check_primary_key()` および `check_foreign_key()` は、外部キー制約を検査するために使用されます。（当然ながら、この機能はかなり前に組み込みの外部キー機能に取って代わりました。しかし例としてはまだ有用です。）

`check_primary_key()` は参照テーブルを検査します。使用法は、この関数を使用する `BEFORE INSERT OR UPDATE` トリガを他のテーブルを参照するテーブルに作成することです。トリガ引数は、外部キーを形成する参照テーブルの列名、被参照テーブル名、プライマリ/一意キーを形成する被参照テーブルの列名です。複数の外部キーを扱うためには、各参照に対してトリガを作成してください。

`check_foreign_key()` は被参照テーブルを検査します。使用法は、この関数を使用する `BEFORE DELETE OR UPDATE` トリガを他のテーブルで参照されるテーブルに作成することです。トリガ引数は、この関数が検査を実行しなければならない参照テーブル数、参照キーが見つかった場合の動作（`cascade` — 参照行を削除、`restrict` — 参照キーが存在する場合トランザクションをアボート、`setnull` — 参照キーフィールドをNULLに設定）、プライマリ/一意キーを形成するトリガを発行したテーブルの列名、参照テーブルの名前と列名（最初の引数で指定された数のテーブル分繰り返す）です。プライマリ/一意キー列はNOT NULLと指定されていなければならない、また、一意性インデックスを持つべきであることに注意してください。

`refint.example` に例があります。

F.33.2. timetravel — 時間旅行を実装する関数

かなり前にPostgreSQLは各タプルで挿入時間、削除時間を保持する時間旅行機能が組み込まれました。これをこれらの関数を使用して模擬することができます。これらの関数を使用するためには、`abstime` 型の、タプルの挿入日付（`start_date`）および変更/削除日付（`stop_date`）を格納するために2つの列をテーブルに追加しなければなりません。

```
CREATE TABLE mytab (
    ...
    start_date      abstime,
    stop_date       abstime
    ...
);
```

この列には好み名前を付けることができますが、以下の説明では`start_date`、`stop_date`を使用します。

新しく行が挿入される時、`start_date`は通常現在時刻に、`stop_date`は`infinity`に設定されるはずですが。挿入されるデータにおけるこれらの列がNULLの場合、トリガは自動的にこれらの値を置き換えます。一般的には、これらの列に非NULLのデータを明示的に挿入することは、ダンプデータの再ロードを行う時にしかありません。

`stop_date`が`infinity`の`stop_date`を持つタプルは"現在有効"で、変更可能です。トリガが防止するため、有限の`stop_date`を持つタプルを変更することはできません。（変更する必要がある場合は、以下のように時間旅行を無効にすることができます。）

変更可能な行では、更新時、更新されようとしているタプルの`stop_date`のみが（現在時刻に）変更され、変更されたデータを持った新しいタプルが挿入されます。この新しいタプルの`start_date`は現在時刻となり、`stop_date`は`infinity`になります。

削除では実際にタプルの削除は行われず、その`stop_date`が現在時刻になります。

"現在有効"なタプルを問い合わせるには、問い合わせのWHERE条件に`stop_date = 'infinity'`を含めてください。（これをビューに組み込もうと考えるかもしれません。）同様に、`start_date`と`stop_date`に適切な条件を付けることで任意の時点で有効だったタプルを問い合わせることもできます。

`timetravel()` は、こうした動作をサポートする、一般的なトリガ関数です。時間旅行を行うテーブル毎にこの関数を使用した `BEFORE INSERT OR UPDATE OR DELETE` トリガを作成してください。2つのトリガ引数で、`start_date`と`stop_date`列の実際の名前を指定してください。省略可能ですが、1から3つの引数を追加して指定することもできます。これらは`text`型の列を参照しなければなりません。トリガは現在のユーザ名を、INSERT時に最初の列、UPDATE時に2番目の列、DELETE時に3番目の列に格納します。

`set_timetravel()` により、テーブル単位で時間旅行を有効または無効にすることができます。 `set_timetravel('mytab', 1)` は `mytab` テーブルの時間旅行を有効にします。 `set_timetravel('mytab', 0)` は `mytab` テーブルの時間旅行を無効にします。 時間旅行が無効な時、`start_date` と `stop_date` 列を自由に変更することができます。 有効状態は現在のデータベースセッション内で局所的な状態であることに注意してください。 新規セッションでは常に、すべてのテーブルの時間旅行は有効状態で始まります。

`get_timetravel()` は、状態変更を行うことなく、時間旅行の状態を返します。

`timetravel.example` に例が存在します。

F.33.3. autoinc — フィールド自動増分用の関数

`autoinc()` は、整数型フィールドにシーケンスの次の値を格納するトリガです。 これは、組み込みの"連番列"機能と一部重複しますが、同一ではありません。 `autoinc()` は挿入時に別のフィールド値に置き換える試みを上書きし、さらに省略可能ですが、更新時にフィールドを増加させるために使用することもできます。

使用方法は、この関数を使用する `BEFORE INSERT` (または `BEFORE INSERT OR UPDATE`) トリガを作成することです。 2つのトリガ引数、変更する整数型列の名前と値を生み出すシーケンスオブジェクトの名前を指定します。 (実際、自動増分列を複数更新したい場合、これらの名前の組み合わせを任意の数指定することができます。)

`autoinc.example` に例があります。

F.33.4. insert_username — 誰がテーブルを変更したかを追跡する関数

`insert_username()` は現在のユーザ名をテキスト型のフィールドに格納するトリガです。 これはテーブル内のある行を最後に変更したユーザを追跡する際に有用です。

使用方法は、この関数を使用する `BEFORE INSERT`、`UPDATE` またはその両方のトリガを作成することです。 1つのトリガ引数、変更するテキスト型の列の名前を指定してください。

`insert_username.example` に例があります。

F.33.5. moddatetime — 最終更新時刻を追跡する関数

`moddatetime()` は現在時刻を `timestamp` 型のフィールドに格納するトリガです。 これは、テーブル内のある行の最終更新時刻を追跡する際に有用です。

使用方法は、この関数を使用する `BEFORE UPDATE` トリガを作成することです。 1つのトリガ引数、変更する列名を指定してください。 この列は `timestamp` 型または `timestamp with time zone` 型でなければなりません。

`moddatetime.example` に例があります。

[前のページ](#)
sepgsql

[ホーム](#)
[上に戻る](#)

[次のページ](#)
sslinfo

F.34. sslinfo

現在のクライアントがPostgreSQLに接続する際に提供する SSL 証明書に関する情報を、`sslinfo`モジュールは提供します。現在の接続が SSL を使用しない場合、モジュールは無用です（大部分の関数は NULL を返します）。

インストールを`--with-openssl`オプション付きで構築しない限り、この拡張は全く構築されません。

F.34.1. 提供される関数

`ssl_is_used()` returns boolean

サーバへの現在の接続において SSL を使用する場合 TRUE、使用しない場合 FALSE を返します。

`ssl_version()` returns text

SSL接続に使われているプロトコルの名前（例えば、SSLv2、SSLv3、またはTLSv1）を返します。

`ssl_cipher()` returns text

SSL接続に使われている暗号の名前（例えば、DHE-RSA-AES256-SHA）を返します。

`ssl_client_cert_present()` returns boolean

現在のクライアントがサーバに対して、有効な SSL クライアント証明書を提示した場合 TRUE、そうでない場合 FALSE を返します。（サーバがクライアントに対して、クライアント証明書を要求する方式と要求しない方式があります）。

`ssl_client_serial()` returns numeric

現在のクライアント証明書のシリアル番号を返します。証明書のシリアル番号と証明書の発行者との組み合わせにより、証明書が一意に識別されることが保証されます（しかし、証明書の所有者の保証ではありません。所有者は定期的にその鍵を変更し、発行者から新しい証明書を取得すべきだからです）。

したがって、自分で認証局を設立し、その認証局の証明書だけをサーバが受理する場合、シリアル番号は利用者を識別するのに最も信頼できる方法です（あまり記憶の助けにはなりません）。

`ssl_client_dn()` returns text

現在のクライアント証明書の所有者の内容を全て返します。文字データは現在のデータベースのエンコーディングに変換されます。なお、証明書名で非 ASCII 文字を用いる場合、データベースでもその文字を使用できると仮定します。データベースが SQL_ASCII エンコーディングを使用する場合、証明書名で用いる非 ASCII 文字は UTF-8 のユニコードとして表示されます。

その結果は `/CN=Somebody /C=Some country /O=Some organization` のようになります。

`ssl_issuer_dn()` returns text

現在のクライアント証明書の発行者名を全て返します。文字データは現在のデータベースのエンコーディングに変換されます。エンコーディングの変換法は `ssl_client_dn` と同じです。

本関数の戻り値と証明書シリアル番号の組み合わせにより、証明書を一意に識別します。

実際に本関数が有用となるのは、サーバの `root.crt` ファイルの中に信頼できる認証局の証明書を複数保有している場合、またはこの認証局が中間認証局の証明書を発行している場合だけです。

`ssl_client_dn_field(fieldname text)` returns text

この関数は証明書の所有者の指定した項目の内容を返します。指定した項目が存在しない場合は NULL を返します。項目の名前は OpenSSL オブジェクトデータベースを使用して ASN1 オブジェクト識別子に変換された文字列定数です。以下の項目が受理できます。

```
commonName (alias CN)
surname (alias SN)
name
givenName (alias GN)
countryName (alias C)
localityName (alias L)
stateOrProvinceName (alias ST)
organizationName (alias O)
organizationUnitName (alias OU)
title
```

```
description
initials
postalCode
streetAddress
generationQualifier
description
dnQualifier
x500UniqueIdentifier
pseudonym
role
emailAddress
```

`commonName` を除き、全ての項目は任意です。認証局の方針によって、どの項目を含み、どの項目を含まないかが全て決まります。しかし、X.500 および X.509 標準によって、項目の意味は厳格に定義されています。したがって、項目に任意の意味を持たせることはできません。

`ssl_issuer_field(fieldname text) returns text`

証明書の所有者に対するものではなく証明書の発行者に対するものであるという点を除き、`ssl_client_dn_field` と同様の関数です。

F.34.2. 著者

Victor Wagner <vitus@cryptocom.ru>, Cryptocom LTD

Cryptocom 社 OpenSSL 開発グループのメールアドレス <openssl@cryptocom.ru>

[前のページ](#)

[spi](#)

[ホーム](#)
[上に戻る](#)

[次のページ](#)

[tablefunc](#)

F.35. tablefunc

tablefuncモジュールにはテーブル（つまり複数行）を返す各種関数があります。これらの関数は、その独自の目的として、および、複数行を返すC関数の作成方法を示す例として、有用です。

F.35.1. 提供される関数

tablefuncモジュールにより提供される関数を表F-28に示します。

表 F-28. tablefuncの関数

関数	戻り値	説明
normal_rand(int numvals, float8 mean, float8 stddev)	setof float8	正規分布乱数値の集合を生成します。
crosstab(text sql)	setof record	行の名前と N 個の値列からなる"ピボット表"を生成します。ここで N は呼出元の問い合わせで指定される行型で決定します。
crosstab N (text sql)	setof table_crosstab_ N	行の名前と N 個の値列からなる"ピボット表"を生成します。crosstab2、crosstab3、crosstab4が定義されていますが、後述する手順で追加のcrosstab N 関数を作成することが可能です。
crosstab(text source_sql, text category_sql)	setof record	2番目の問い合わせで指定された値列を持つ"ピボット表"を生成します。
crosstab(text sql, int N)	setof record	廃止予定のcrosstab(text)です。値列の数は呼び出す問い合わせで常に決まりますので、現在引数 N は無視されます。
connectby(text relname, text keyid fld, text parent keyid fld [, text orderby fld], text start_with, int max_depth [, text branch_delim])	setof record	階層ツリー構造表現を生成します。

F.35.1.1. normal_rand

normal_rand(int numvals, float8 mean, float8 stddev) returns setof float8

normal_randは正規乱数値の集合（ガウシアン分布）を生成します。

ここでnumvalsはこの関数が返す値の数です。meanは正規分布の平均値、stddevは正規分布値の標準偏差です。

例えば、以下の呼出しは、平均5、標準偏差3で1000個の値を要求します。

```
test=# SELECT * FROM normal_rand(1000, 5, 3);
normal_rand
-----
 1.56556322244898
 9.10040991424657
 5.36957140345079
-0.369151492880995
 0.283600703686639
      .
      .
      .
 4.82992125404908
 9.71308014517282
 2.49639286969028
(1000 rows)
```

F.35.1.2. crosstab(text)

```
crosstab(text sql)
crosstab(text sql, int N)
```

crosstab関数は"ピボット"表示を生成するために使用されます。ここでは、データは下方向にではなくページ横方向に渡って列

挙げられます。例えば、以下のようなデータがあるとします。

```
row1    val11
row1    val12
row1    val13
...
row2    val21
row2    val22
row2    val23
...
```

これを次のように表示したいとします。

```
row1    val11    val12    val13    ...
row2    val21    val22    val23    ...
...
```

`crosstab`関数は、最初のような書式を持つ生データを生成するSQL問い合わせとなるテキストパラメータを取り、2番目のような書式を持つテーブルを生成します。

`sql`パラメータは元となるデータ集合を生成するSQL文です。この文は`row_name`列を1つ`category`列を1つ、`value`列を1つ返さなければなりません。`N`は廃れたパラメータであり、指定されたとしても無視されます。（これまでは、これは出力値列の数と一致する必要がありました。しかし、現在これは呼び出し元の問い合わせにより決まります。）

例：指定したSQLは以下のような集合を生成しても構いません。

```
row_name  cat    value
-----+-----+-----
row1      cat1   val1
row1      cat2   val2
row1      cat3   val3
row1      cat4   val4
row2      cat1   val5
row2      cat2   val6
row2      cat3   val7
row2      cat4   val8
```

`crosstab`関数は`setof record`を返すものとして宣言されています。このため、出力列の実際の名前と型を呼び出し元の`SELECT`文の`FROM`内で宣言しなければなりません。以下に例を示します。

```
SELECT * FROM crosstab('...') AS ct(row_name text, category_1 text, category_2 text);
```

この例は以下のような集合を生成します。

```
          <== value    columns ==>
row_name  category_1    category_2
-----+-----+-----
row1      val1          val2
row2      val5          val6
```

`FROM`句は出力を1つの`row_name`列（SQL問い合わせの最初の結果列と同一データ型）と続く`N`個の`value`列（SQL問い合わせの3番目の結果列とすべて同じデータ型）を持つものとして定義しなければなりません。必要なだけの個数の値列を出力するように設定することができます。出力列の名前は使用者に任されています。

`crosstab`関数は、同じ`row_name`値を持つ入力行の各連続的なグループに対して、1つの出力行を生成します。左から右へこれらの行の`value`フィールドで出力`value`列を埋めていきます。もしグループ内の行が存在する出力`value`列より少なければ、余った出力列は`NULL`になります。もし行が多ければ、余った入力行は無視されます。

実際のところ、入力行の順序が適切になるように、つまり、同じ`row_name`を持つ値がまとまり、行内で正しく順序付けられるように、SQL問い合わせは常に`ORDER BY 1,2`を指定しなければなりません。`crosstab`自体が問い合わせ結果の2番目の列に注意を払わないことに注意してください。これは順序付けのため、3番目の列の値がページに渡って現れる順序を制御するためだけに存在します。

以下に複雑な例を示します。

```
CREATE TABLE ct(id SERIAL, rowid TEXT, attribute TEXT, value TEXT);
INSERT INTO ct(rowid, attribute, value) VALUES('test1','att1','val1');
INSERT INTO ct(rowid, attribute, value) VALUES('test1','att2','val2');
INSERT INTO ct(rowid, attribute, value) VALUES('test1','att3','val3');
INSERT INTO ct(rowid, attribute, value) VALUES('test1','att4','val4');
INSERT INTO ct(rowid, attribute, value) VALUES('test2','att1','val5');
INSERT INTO ct(rowid, attribute, value) VALUES('test2','att2','val6');
INSERT INTO ct(rowid, attribute, value) VALUES('test2','att3','val7');
INSERT INTO ct(rowid, attribute, value) VALUES('test2','att4','val8');

SELECT *
FROM crosstab(
    'select rowid, attribute, value
```

```

from ct
where attribute = 'att2' or attribute = 'att3'
order by 1,2')
AS ct(row_name text, category_1 text, category_2 text, category_3 text);

row_name | category_1 | category_2 | category_3
-----+-----+-----+-----
test1    | val2       | val3       |
test2    | val6       | val7       |
(2 rows)

```

必要な出力行型をその定義に反映した独自のcrosstab関数を構築することで、常に出力列を定義するためのFROM句を書く必要性をなくすることができます。これは次節で説明します。他にも必要なFROM句をビュー定義に埋め込むことでも実現可能です。

F.35.1.3. crosstabN(text)

```
crosstabN(text sql)
```

crosstabN関数は、呼び出し元のSELECT問い合わせで列名と型を書き出す必要性をなくすることができるように、一般的なcrosstab関数に対する独自のラッパを構築する方法の例です。tablefuncモジュールには、次のように出力行型が定義されたcrosstab2、crosstab3、crosstab4が含まれています。

```

CREATE TYPE tablefunc_crosstab_N AS (
    row_name TEXT,
    category_1 TEXT,
    category_2 TEXT,
    .
    .
    .
    category_N TEXT
);

```

このように、入力問い合わせがtext型のrow_name列とvalue列を生成し、かつ、2、3、または4個の出力値列を持つ場合、これらの関数を直接使用することができます。この他の点はすべて、上述の一般的なcrosstab関数で説明した通りの動作をします。

例えば、上で挙げた例は下のように動作します。

```

SELECT *
FROM crosstab3(
    'select rowid, attribute, value
    from ct
    where attribute = 'att2' or attribute = 'att3'
    order by 1,2');

```

これらの関数はほぼ説明を目的として提供されたものです。背後のcrosstab()関数に基いた独自の戻り型と関数を作成することができます。独自のcrosstab関数を構築する方法は2つあります。

- contrib/tablefunc/tablefunc--1.0.sqlの例と同様にして、必要な出力列を記述する複合型を作成します。そして、text型のパラメータを1つ取り、setof your_type_nameを返す一意な名前の関数を、同じ背後のcrosstab C関数をリンクさせて定義します。例えば、元データが行名としてtext型を、値としてFLOAT8を生成し、5つの値列を希望する場合、以下ようになります。

```

CREATE TYPE my_crosstab_float8_5_cols AS (
    my_row_name text,
    my_category_1 float8,
    my_category_2 float8,
    my_category_3 float8,
    my_category_4 float8,
    my_category_5 float8
);

CREATE OR REPLACE FUNCTION crosstab_float8_5_cols(text)
RETURNS setof my_crosstab_float8_5_cols
AS '$libdir/tablefunc','crosstab' LANGUAGE C STABLE STRICT;

```

- 暗黙的に戻り値の型を定義する場合はOUTパラメータを使用してください。同じ例を以下のように書くこともできます。

```

CREATE OR REPLACE FUNCTION crosstab_float8_5_cols(
    IN text,
    OUT my_row_name text,
    OUT my_category_1 float8,
    OUT my_category_2 float8,
    OUT my_category_3 float8,
    OUT my_category_4 float8,
    OUT my_category_5 float8)
RETURNS setof record
AS '$libdir/tablefunc','crosstab' LANGUAGE C STABLE STRICT;

```

F.35.1.4. crosstab(text, text)

```
crosstab(text source_sql, text category_sql)
```

単一パラメータのcrosstab構文の大きな制限は、各値を最初の利用可能な列に挿入して、すべての値をグループのように扱う点です。 値列を特定のデータカテゴリに対応させ、グループの一部はカテゴリの一部のデータを持たない可能性がある場合は、うまく動作しません。 2パラメータを取るcrosstab構文は、出力列に対応するカテゴリのリストを明示的に提供することで、こうした状況を扱います。

source_sqlは元となるデータ集合を生成するSQL文です。 このSQL文はrow_name列を1つcategory列を1つ、value列を1つ返さなければなりません。 また1つ以上の"追加"の列を持つこともできます。 row_name列が先頭でなければなりません。 categoryとvalue列は、この順番で最後の2列でなければなりません。 row_nameとcategoryとの間の列はすべて"追加"の列とみなされます。 "追加"の列は同じrow_name値を持つ行すべてで同一であるということが前提です。

例えば、source_sqlは以下のような集合を生成しなければなりません。

```
SELECT row_name, extra_col, cat, value FROM foo ORDER BY 1;
```

row_name	extra_col	cat	value
row1	extra1	cat1	val1
row1	extra1	cat2	val2
row1	extra1	cat4	val4
row2	extra2	cat1	val5
row2	extra2	cat2	val6
row2	extra2	cat3	val7
row2	extra2	cat4	val8

category_sqlはカテゴリの集合を生成するSQL文でなければなりません。 このSQL文は1つの列のみを返さなければなりません。 また、少なくとも1つの結果行を生成しなければならず、さもないと、エラーになります。 さらに重複するカテゴリを生成してはなりません。 さもないとエラーとなります。 category_sqlは以下のようなものになります。

```
SELECT DISTINCT cat FROM foo ORDER BY 1;
```

cat
cat1
cat2
cat3
cat4

crosstab関数はsetof recordを返すものとして宣言されていますので、出力列の実際の名前と型を、以下の例のように、呼出元のSELECTのFROM句で定義しなければなりません。

```
SELECT * FROM crosstab('...', '...')
AS ct(row_name text, extra text, cat1 text, cat2 text, cat3 text, cat4 text);
```

これは以下のような集合を生成します。

row_name	extra	<== value	columns	==>	
		cat1	cat2	cat3	cat4
row1	extra1	val1	val2		val4
row2	extra2	val5	val6	val7	val8

FROM句は、出力列の適切な個数、およびその適切なデータ型を定義しなければなりません。 source_sql問い合わせ結果にN個の列がある場合、最初のN-2は最初のN-2出力列と一致しなければなりません。 残りの出力列はsource_sql問い合わせ結果の最後の列の型を持たなければならず、かつ、category_sql問い合わせ結果内の行と同じ個数でなければなりません。

crosstab関数は、同一row_name値を持つ入力行の連続したグループ毎に1つの出力行を生成します。 row_name出力列と任意の"追加"列はグループの最初の行からコピーされます。 value出力列は、category値と一致する行のvalueで埋められます。 行のcategoryがcategory_sql問い合わせの出力とまったく一致しなかった場合、そのvalueは無視されます。 グループの入力行内にまったくカテゴリに一致する出力列が存在しない場合、NULLで埋められます。

実際は、同じrow_nameを持つ値をまとめられるように、source_sql問い合わせでは常にORDER BY 1を指定すべきです。 しかし、グループ内のカテゴリの順序は重要ではありません。 また、category_sql問い合わせの出力順序が指定された出力列の順序と一致することを確実にすることが重要です。

以下に複雑な例を2つ示します。

```
create table sales(year int, month int, qty int);
insert into sales values(2007, 1, 1000);
insert into sales values(2007, 2, 1500);
insert into sales values(2007, 7, 500);
insert into sales values(2007, 11, 1500);
insert into sales values(2007, 12, 2000);
insert into sales values(2008, 1, 1000);

select * from crosstab(
```

```
'select year, month, qty from sales order by 1',
'select m from generate_series(1,12) m'
) as (
year int,
"Jan" int,
"Feb" int,
"Mar" int,
"Apr" int,
"May" int,
"Jun" int,
"Jul" int,
"Aug" int,
"Sep" int,
"Oct" int,
"Nov" int,
"Dec" int
);
year | Jan | Feb | Mar | Apr | May | Jun | Jul | Aug | Sep | Oct | Nov | Dec
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
2007 | 1000 | 1500 |      |      |      |      | 500 |      |      |      | 1500 | 2000
2008 | 1000 |      |      |      |      |      |      |      |      |      |      |
(2 rows)

CREATE TABLE cth(rowid text, rowdt timestamp, attribute text, val text);
INSERT INTO cth VALUES('test1','01 March 2003','temperature','42');
INSERT INTO cth VALUES('test1','01 March 2003','test_result','PASS');
INSERT INTO cth VALUES('test1','01 March 2003','volts','2.6987');
INSERT INTO cth VALUES('test2','02 March 2003','temperature','53');
INSERT INTO cth VALUES('test2','02 March 2003','test_result','FAIL');
INSERT INTO cth VALUES('test2','02 March 2003','test_startdate','01 March 2003');
INSERT INTO cth VALUES('test2','02 March 2003','volts','3.1234');

SELECT * FROM crosstab
(
'SELECT rowid, rowdt, attribute, val FROM cth ORDER BY 1',
'SELECT DISTINCT attribute FROM cth ORDER BY 1'
)
AS
(
rowid text,
rowdt timestamp,
temperature int4,
test_result text,
test_startdate timestamp,
volts float8
);
rowid | rowdt | temperature | test_result | test_startdate | volts
-----+-----+-----+-----+-----+-----
test1 | Sat Mar 01 00:00:00 2003 | 42 | PASS | Sat Mar 01 00:00:00 2003 | 2.6987
test2 | Sun Mar 02 00:00:00 2003 | 53 | FAIL | Sat Mar 01 00:00:00 2003 | 3.1234
(2 rows)
```

各問い合わせで結果列の名前と型を記述する必要性をなくすために、事前定義した関数を作成することができます。 前節の例を参照してください。 このcrosstab構文用の背後のC関数はcrosstab_hashという名前です。

F.35.1.5. connectby

```
connectby(text relname, text keyid_fld, text parent_keyid_fld
[, text orderby_fld ], text start_with, int max_depth
[, text branch_delim ])

connectby関数はテーブル内に格納された階層データ表示を生成します。 テーブルは行を一意に識別するキーフィールドと各行の親（もしあれば）を参照する親キーフィールドを持たなければなりません。 connectbyは任意の行から辿った部分ツリーを表示することができます。
```

表F-29ではパラメータを解説します。

表 F-29. connectbyパラメータ

パラメータ	説明
relname	元となるリレーション名
keyid_fld	キーフィールドの名前
parent_keyid_fld	親のキーフィールドの名前
orderby_fld	兄弟の順序付け用のフィールド名（省略可能）
start_with	開始行のキー値
max_depth	辿る深さに対する制限。無制限の場合はゼロ

branch_delim	キーと分岐出力で区切る文字列（省略可能）

キーおよび親キーフィールドは任意のデータ型を取ることができますが、これらは同じデータ型でなければなりません。キーフィールドのデータ型に関係なく、start_withはテキスト文字列として入力されなければならないことに注意してください。

connectby関数はsetof recordを返すものとして宣言されていますので、以下の例のように、出力列の実際の名前と型を呼出し元のSELECT文のFROM句で定義しなければなりません。

```
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'pos', 'row2', 0, '~')
AS t(keyid text, parent_keyid text, level int, branch text, pos int);
```

先頭から2つの出力列は、現在の行のキーおよび親行のキーとして使用されます。これらはテーブルのキーフィールドのデータ型と一致する必要があります。3番目の出力列はツリーの深さであり、integer型である必要があります。branch_delimパラメータが与えられた場合、次の出力列は分岐表示であり、text型である必要があります。最後に、orderby_fldパラメータが与えられた場合、最後の出力列は連番であり、integer型である必要があります。

"分岐"出力列は現在の行まで達するために取られるキーの経路を示します。キーは指定されたbranch_delim文字列で区切られます。分岐表示が不要ならば、branch_delimパラメータと出力列リスト内の分岐列を省略してください。

同じ親を持つ兄弟の順序が重要な場合、どのフィールドで兄弟の順序付けを行うかを指定するorderby_fldパラメータを含めてください。このフィールドは任意のソート可能なデータ型を取ることができます。orderby_fldが指定された場合のみ、出力列リストには、最終整数型連番列を含めなければなりません。

テーブルおよびフィールド名を表すパラメータはそのままconnectbyが内部的に生成するSQL問い合わせにコピーされます。したがって、大文字小文字が混在した名前または特殊文字を含む名前の場合は二重引用符で括ってください。またテーブル名をスキーマで修飾する必要があるかもしれません。

大規模なテーブルでは、親キーフィールド上にインデックスがないと性能が劣化します。

branch_delim文字列がキー値内にまったく出現しないことが重要です。さもないと、connectbyは無限再帰エラーを間違って報告するかもしれません。branch_delimが提供されていない場合、再帰を検知するためにデフォルト値~が使用されます。

以下に例を示します。

```
CREATE TABLE connectby_tree(keyid text, parent_keyid text, pos int);
```

```
INSERT INTO connectby_tree VALUES('row1',NULL, 0);
INSERT INTO connectby_tree VALUES('row2','row1', 0);
INSERT INTO connectby_tree VALUES('row3','row1', 0);
INSERT INTO connectby_tree VALUES('row4','row2', 1);
INSERT INTO connectby_tree VALUES('row5','row2', 0);
INSERT INTO connectby_tree VALUES('row6','row4', 0);
INSERT INTO connectby_tree VALUES('row7','row3', 0);
INSERT INTO connectby_tree VALUES('row8','row6', 0);
INSERT INTO connectby_tree VALUES('row9','row5', 0);
```

```
-- with branch, without orderby_fld (order of results is not guaranteed)
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'row2', 0, '~')
AS t(keyid text, parent_keyid text, level int, branch text);
```

keyid	parent_keyid	level	branch
row2		0	row2
row4	row2	1	row2~row4
row6	row4	2	row2~row4~row6
row8	row6	3	row2~row4~row6~row8
row5	row2	1	row2~row5
row9	row5	2	row2~row5~row9

(6 rows)

```
-- without branch, without orderby_fld (order of results is not guaranteed)
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'row2', 0)
AS t(keyid text, parent_keyid text, level int);
```

keyid	parent_keyid	level
row2		0
row4	row2	1
row6	row4	2
row8	row6	3
row5	row2	1
row9	row5	2

(6 rows)

```
-- with branch, with orderby_fld (notice that row5 comes before row4)
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'pos', 'row2', 0, '~')
AS t(keyid text, parent_keyid text, level int, branch text, pos int);
```

keyid	parent_keyid	level	branch	pos
row2		0	row2	0
row5	row2	1	row2~row5	1
row4	row2	1	row2~row4	2
row6	row4	2	row2~row4~row6	3
row8	row6	3	row2~row4~row6~row8	4
row9	row5	2	row2~row5~row9	5

```

row2 | | 0 | row2 | 1
row5 | row2 | 1 | row2~row5 | 2
row9 | row5 | 2 | row2~row5~row9 | 3
row4 | row2 | 1 | row2~row4 | 4
row6 | row4 | 2 | row2~row4~row6 | 5
row8 | row6 | 3 | row2~row4~row6~row8 | 6
(6 rows)

-- without branch, with orderby fld (notice that row5 comes before row4)
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'pos', 'row2', 0)
AS t(keyid text, parent_keyid text, level int, pos int);
keyid | parent_keyid | level | pos
-----+-----+-----+----
row2 | | 0 | 1
row5 | row2 | 1 | 2
row9 | row5 | 2 | 3
row4 | row2 | 1 | 4
row6 | row4 | 2 | 5
row8 | row6 | 3 | 6
(6 rows)

```

F.35.2. 作者

Joe Conway

F.36. tcn

tcnモジュールは 関連づけさせた任意のテーブル上の変更を監視者に通知するトリガ関数を提供します。これはFOR EACH ROWのAFTERトリガとして使用しなければなりません。

CREATE TRIGGER文の中で与えることができるパラメータは1つしかありませんが、省略することができます。与えられた場合、それは通知のチャンネル名として使用されます。省略された場合はチャンネル名としてtcnが使用されます。

通知のペイロードにはテーブル名、どのようなsy類の操作が行われたかを示す文章、プライマリキー列における列名と値の組み合わせが含まれます。部位はそれぞれカンマで区切られています。正規表現を使用して簡単に解析するために、テーブル名と列名は常に二重引用符で括られ、またデータ値は常に単一引用符で括られています。内部に含まれる引用符は二重化されます。

この閣僚を使用する簡単な例を以下に示します。

```
test=# create table tcndata
test=# (
test=#     a int not null,
test=#     b date not null,
test=#     c text,
test=#     primary key (a, b)
test=# );
NOTICE: CREATE TABLE / PRIMARY KEY will create implicit index "tcndata_pkey" for table "tcndata"
CREATE TABLE
test=# create trigger tcndata_tcn_trigger
test=#     after insert or update or delete on tcndata
test=#     for each row execute procedure triggered_change_notification();
CREATE TRIGGER
test=# listen tcn;
LISTEN
test=# insert into tcndata values (1, date '2012-12-22', 'one'),
test=#                                     (1, date '2012-12-23', 'another'),
test=#                                     (2, date '2012-12-23', 'two');
INSERT 0 3
Asynchronous notification "tcn" with payload ""tcndata",I,"a"='1',"b"='2012-12-22'" received from server
process with PID 22770.
Asynchronous notification "tcn" with payload ""tcndata",I,"a"='1',"b"='2012-12-23'" received from server
process with PID 22770.
Asynchronous notification "tcn" with payload ""tcndata",I,"a"='2',"b"='2012-12-23'" received from server
process with PID 22770.
test=# update tcndata set c = 'uno' where a = 1;
UPDATE 2
Asynchronous notification "tcn" with payload ""tcndata",U,"a"='1',"b"='2012-12-22'" received from server
process with PID 22770.
Asynchronous notification "tcn" with payload ""tcndata",U,"a"='1',"b"='2012-12-23'" received from server
process with PID 22770.
test=# delete from tcndata where a = 1 and b = date '2012-12-22';
DELETE 1
Asynchronous notification "tcn" with payload ""tcndata",D,"a"='1',"b"='2012-12-22'" received from server
process with PID 22770.
```

F.37. test_parser

test_parserは、全文検索用の独自パーサの例です。これは特に有用なことはまったく行いませんが、独自パーサ開発の開始点として提供することができます。

test_parserは空白で区切られた単語を認知し、2つのトークン種類だけを返します。

```
mydb=# SELECT * FROM ts_token_type('testparser');
 tokid | alias | description
-----+-----+-----
      3 | word  | Word
     12 | blank | Space symbols
(2 rows)
```

これらのトークン数はデフォルトパーサの数え方と互換性があるように選択されました。これにより、headline() 関数として使用することができますので、例を単純にしています。

F.37.1. 使用方法

test_parser拡張をインストールすると、testparser テキスト検索パーサが作成されます。ユーザが設定可能なパラメータはありません。

例えば、このパーサを試験することができます。

```
mydb=# SELECT * FROM ts_parse('testparser', 'That''s my first own parser');
 tokid | token
-----+-----
      3 | That's
     12 |
      3 | my
     12 |
      3 | first
     12 |
      3 | own
     12 |
      3 | parser
```

実際に使用する場合は、このパーサを使用するテキスト検索設定の構築が必要です。以下に例を示します。

```
mydb=# CREATE TEXT SEARCH CONFIGURATION testcfg ( PARSER = testparser );
CREATE TEXT SEARCH CONFIGURATION

mydb=# ALTER TEXT SEARCH CONFIGURATION testcfg
mydb-# ADD MAPPING FOR word WITH english_stem;
ALTER TEXT SEARCH CONFIGURATION

mydb=# SELECT to_tsvector('testcfg', 'That''s my first own parser');
 to_tsvector
-----
'that':1 'first':3 'parser':5
(1 row)

mydb=# SELECT ts_headline('testcfg', 'Supernovae stars are the brightest phenomena in galaxies',
mydb-# to_tsquery('testcfg', 'star'));
 ts_headline
-----
Supernovae <b>stars</b> are the brightest phenomena in galaxies
(1 row)
```

F.38. tsearch2

tsearch2モジュールは、テキスト検索がリリース8.3でコアPostgreSQLに統合される前の、tsearch2を使用したアプリケーション向けの後方互換のテキスト検索機能を提供します。

F.38.1. 移植に関する問題

組み込みのテキスト検索機能はtsearch2を基にしており、大部分は似ていますが、多くの小さな違いがあります。このため、既存のアプリケーションにおいて移植に関する問題が発生します。

- 一部の関数名が変わりました。例えばrankはts_rankになりました。置き換え版のtsearch2モジュールは古い名前を別名として提供します。
- 組み込みのテキスト検索データ型と関数はすべてpg_catalogシステムスキーマ内に存在します。tsearch2を使用したインストールでは、これらのオブジェクトは通常publicスキーマ内にありましたが、ユーザによっては独自に別のスキーマに格納することを選択していました。したがって、これらのオブジェクトへの明示的にスキーマ修飾された参照はどちらの場合も失敗します。置き換え版のtsearch2モジュールは、こうした参照が動作し続けられるように、public（必要ならば他のスキーマ）に格納される別名オブジェクトを提供します。
- 組み込みのテキスト検索機能では"現在のパーサ"または"現在の辞書"という概念はなく、現在の検索設定default_text_search_configパラメータによりせつていされます。現在のパーサや現在の辞書はデバッグ目的の関数でのみ使用されていましたが、これが移植の問題を引き起こす場合があります。置き換え版のtsearch2モジュールはこれらの追加状態変数を模擬し、その設定および抽出に関する後方互換を持つ関数を提供します。

置き換え版のtsearch2で対応されていない問題もいくつか存在します。このため、以下のいずれかの場合はアプリケーションコードの変更が必要です。

- 過去のtsearch2トリガ関数では、引数リスト内の項目をtsvector書式に変換される前にテキストデータに対して呼び出される関数名にすることができました。これはセキュリティ問題になりますので削除されました。このため、呼び出される関数が意図したものであることを保証することはできません。インデックス付けされる前にデータをいじる必要がある場合の推奨方式は、専用の作業を行う独自トリガを作成することです。
- テキスト検索設定の情報は、tsearch2で使用されたテーブルと大きく異なる中核のシステムカタログに移動されました。こうしたテーブルの検査、変更を行うアプリケーションはすべて調整する必要があります。
- アプリケーションが独自のテキスト検索設定を使用していた場合、それらを新しいテキスト検索設定SQLコマンドを使用してコアカタログ内に構築する必要があります。置き換え版のtsearch2モジュールは、古いtsearch2の設定テーブルの集合をPostgreSQL 8.3にロードできるようにすることで、多少のサポートを行います。（このモジュールがなければ、regprocedure列の値を関数に解決できませんので、設定データをロードすることは不可能です。）こうした設定テーブルは実際に何も行いませんが、少なくとも8.3で同等の独自設定を構築する際に、その内容を考慮することは可能です。
- 古いreset_tsearch() およびget_covers() はサポートされません。
- 置き換え版のtsearch2モジュールは別名演算子をまったく定義しません。完全に組み込みのものに依存しています。まったく一般的ではありませんが、アプリケーションが明示的にスキーマ修飾した演算子名を使用する場合のみ、問題が発生します。

F.38.2. 8.3より前のインストールを変換

tsearch2を使用した、8.3より前のインストールからの推奨更新方法を以下に示します。

- 1 11 通常の方法で古いインストールのダンプを作成します。ただし、pg_dumpまたはpg_dumpallの-c (--clean)オプションは使用しないでください。
- 1 11 新しいインストールで、空のデータベースを作成し、置き換え版のtsearch2をテキスト検索を使用する各データベースにインストールしてください。これをダンプデータをロードする前に行う必要があります。古いインストールがpublic以外のスキーマにtsearch2のオブジェクトを持つ場合は、置き換え版のオブジェクトが同じスキーマ内に生成されるようにCREATE EXTENSIONコマンドを確実に調整してください。
- 1 11 ダンプデータをロードしてください。実際、元のtsearch2のオブジェクトの再作成に失敗するため、いくつかエラーが

報告されます。これらのエラーは無視することができますが、単一トランザクションでダンプをリストアすることができないことを意味します。（例えば、pg_restoreの-1スイッチを使用することはできません。）

1 11 リストアしたtsearch2の設定テーブル（pg_ts_cfgなど）の内容を検査してください。そして、必要に応じて同等の組み込みテキスト検索設定を作成してください。古い設定テーブルから有用な情報をすべて取り出した後、これらを削除することができます。

1 11 アプリケーションを試験します。

後で、最終的に置き換え版のtsearch2モジュールをアンインストールできるように、アプリケーション内の別名テキスト検索オブジェクトへの参照の名前を変更する方がよいでしょう。

F.38.3. 参考資料

Tsearch2開発サイト <http://www.sai.msu.su/~megera/postgres/gist/tsearch/V2/>

[前のページ](#)
test_parser

[ホーム](#)
[上に戻る](#)

[次のページ](#)
unaccent

F.39. unaccent

`unaccent`は語彙素からアクセント(発音区分記号)を取り除く全文検索用の辞書です。これはフィルタ処理を行う辞書、つまり、標準の動作と異なり、その出力が常に次の辞書（もしあれば）に渡されるものです。これにより全文検索においてアクセントを無視した処理を行うことができます。

現在の`unaccent`の実装では`thesaurus`辞書向けの正規化用辞書として使用することはできません。

F.39.1. 設定

`unaccent`辞書は以下のオプションを受け付けます。

- このファイルは`$SHAREDIR/tsearch_data/`内に格納しなければなりません。（ここで`$SHAREDIR`はPostgreSQLインストールディレクトリの共有データディレクトリを意味します。）この名前は`.rules`で終わらなければなりません。（`.rules`は`RULES`パラメータには含まれません。）

`rules`ファイルの書式は以下の通りです。

- 各行は、アクセント付き文字とその後にアクセントを取り除いた文字から構成される組み合わせを表します。一つ目が二つ目に変換されます。以下に例を示します。

À	A
Á	A
Â	A
Ã	A
Ä	A
Å	A
Æ	A

`unaccent.rules`は、ほとんどの欧州圏の言語で直接使用できる、より複雑な例です。これは`unaccent`モジュールをインストールした時に`$SHAREDIR/tsearch_data/`にインストールされます。

F.39.2. 使用方法

`unaccent`拡張をインストールすることで、`unaccent`全文検索テンプレートとそれに基づくデフォルトのパラメータを持つ`unaccent`辞書が生成されます。`unaccent`は`RULES='unaccent'`というデフォルトパラメータ設定を持ちます。これは標準の`unaccent.rules`ファイルを即座に使用可能にします。次の例のようにパラメータを変更することができます。

```
mydb=# ALTER TEXT SEARCH DICTIONARY unaccent (RULES='my_rules');
```

また、このテンプレートに基づいた辞書を新規に作成することができます。

以下を行うことで、辞書の動作を確認することができます。

```
mydb=# select ts_lexize('unaccent','Hôtel');
      ts_lexize
-----
{Hotel}
(1 row)
```

全文検索設定に`unaccent`辞書を組み込む方法を示す例を以下に示します。

```
mydb=# CREATE TEXT SEARCH CONFIGURATION fr ( COPY = french );
mydb=# ALTER TEXT SEARCH CONFIGURATION fr
      ALTER MAPPING FOR hword, hword_part, word
      WITH unaccent, french_stem;
mydb=# select to_tsvector('fr','Hôtels de la Mer');
      to_tsvector
-----
'hotel':1 'mer':4
(1 row)

mydb=# select to_tsvector('fr','Hôtel de la Mer') @@ to_tsquery('fr','Hotels');
      ?column?
-----
t
(1 row)

mydb=# select ts_headline('fr','Hôtel de la Mer',to_tsquery('fr','Hotels'));
      ts_headline
```

```
-----
  <b>Hôtel</b> de la Mer
(1 row)
```

F.39.3. 関数

`unaccent`関数は与えられた文字列からアクセント（発音区別符号）を取り除きます。基本的にこれは`unaccent`辞書のラッパです。しかし通常の全文検索以外の文脈で使うことができます。

```
unaccent([dictionary, ] string) returns text
```

下記は使用例です。

```
SELECT unaccent('unaccent', 'Hôtel');
SELECT unaccent('Hôtel');
```

F.40. uuid-ossdp

uuid-ossdp モジュールは複数の標準的なアルゴリズムの1つを使用して汎用一意識別子（UUID）を生成する関数を提供します。また、特殊なUUID定数を生成する関数も提供します。

本モジュールは<http://www.ossdp.org/pkg/lib/uuid/>にあるOSSDP UUIDライブラリに依存しています。

F.40.1. uuid-ossdp関数

UUIDを生成するために利用できる関数を表F-30に示します。関連する標準ITU-T Rec. X.667、ISO/IEC 9834-8:2005、RFC 4122はUUIDの生成に関して、バージョン番号1、3、4、5で識別される4つのアルゴリズムを規定します。（バージョン2アルゴリズムは存在しません。）これらのアルゴリズムのそれぞれは、異なるアプリケーション群に適切でしょう。

表 F-30. UUID生成用関数

関数	説明
<code>uuid_generate_v1()</code>	この関数はバージョン1 UUIDを生成します。これはコンピュータのMACアドレスとタイムスタンプが含まれます。この種のUUIDは識別子を生成したコンピュータを識別できる情報や生成した時刻をあばくことができますので、ある種のセキュリティに注意すべきアプリケーションでは適しません。
<code>uuid_generate_v1mc()</code>	この関数は、コンピュータの実MACアドレスではなくランダムなマルチキャストMACアドレスを使用して、バージョン1 UUIDを作成します。
<code>uuid_generate_v3(namespace uuid, name text)</code>	この関数は、入力で指定されたnameを使用して、与えられた名前空間でバージョン3 UUIDを生成します。名前空間は、表F-31で示す <code>uuid_ns_*</code> () 関数で生成される特殊な定数の1つでなければなりません。（理論上これは何らかのUUIDになります。）nameは選択された名前空間内の識別子です。 例えば以下のようになります。 <pre>SELECT uuid_generate_v3(uuid_ns_url(), 'http://www.postgresql.org');</pre> nameパラメータはMD5でハッシュ化されます。このため、生成されたUUIDから平文が分かることはありません。この方法によるUUIDの生成は不規則性はなく、また、環境に依存する要素もないため、再度生成されます。
<code>uuid_generate_v4()</code>	この関数はバージョン4 UUIDを生成します。これは完全にランダムな数から生成されます。
<code>uuid_generate_v5(namespace uuid, name text)</code>	この関数はバージョン5 UUIDを生成します。バージョン3 UUIDと似ていますが、ハッシュ方式としてSHA-1を使用することが異なります。SHA-1がMD5より安全であることから、バージョン5はバージョン3に比べて好まれるはずです。

表 F-31. UUID定数を返す関数

<code>uuid_nil()</code>	"nil" UUID定数です。これは実際のUUIDになることはありません。
<code>uuid_ns_dns()</code>	DNS名前空間をUUIDに選定した定数です。
<code>uuid_ns_url()</code>	URL名前空間をUUIDに選定した定数です。

uuid_ns_oid()	ISOオブジェクト識別子（OID）をUUIDに選定した定数です。（これはASN.1のOIDに関するもので、PostgreSQLで使われるOIDとは関係ありません。）
uuid_ns_x500()	X.500区分名（DN）をUUIDに選定した定数です。

F.40.2. 作者

Peter Eisentraut <peter_e@gmx.net>

F.41. xml2

xml2 モジュールはXPath問い合わせとXSLT機能を提供します。

F.41.1. 廃止予定の可能性についてのお知らせ

PostgreSQL 8.3から、SQL/XML標準に基づくXML関連の機能はコアサーバ内に存在します。この機能は、XML構文検査、本モジュールと同等のを行うXPath問い合わせなどを範囲としますが、APIには互換性はありません。新しい標準APIのため、本モジュールは今後のバージョンのPostgreSQLで削除される予定ですので、アプリケーションの変換が推奨されています。本モジュールの機能のうち新しいAPIに適用できないことがわかった場合、その不足に取り組むことができるようにpgsql-hackers@postgresql.orgにその問題を表明してください。

F.41.2. 関数の説明

表F-32に本モジュールで提供する関数を示します。これらの関数は簡単なXML解析とXPath問い合わせを提供します。すべての引数はtext型です。簡潔にするため説明しません。

表 F-32. 関数

関数	結果	説明
xml_is_well_formed(document)	bool	これはパラメータとして与えられた文書テキストを解析し、文書が整形式のXMLであれば真を返します。（注意:PostgreSQL 8.2以前ではこの関数はxml_valid()と呼ばれていました。XMLでは整形と検証が異なる意味を持つため間違った名前でした。古い名前もまだ利用できますが、廃止予定です。）
xpath_string(document, query)	text	これらの関数は与えられた文書に対するXPath問い合わせを評価し、結果を指定した型にキャストします。
xpath_number(document, query)	float4	
xpath_bool(document, query)	bool	
xpath_nodeset(document, query, toptag, itemtag)	text	これは文書に対する問い合わせを評価し、XMLタグ内に結果を包みます。結果が複数の値であれば、出力は以下のようになります。 <pre><toptag> <itemtag>Value 1 which could be an XML fragment</itemtag> <itemtag>Value 2....</itemtag> </toptag></pre> toptagまたはitemtagが空文字だった場合、対応するタグは省略されます。
xpath_nodeset(document, query)	text	xpath_nodeset(document, query, toptag, itemtag)と同様ですが、結果は両方のタグを省きます。
xpath_nodeset(document, query, itemtag)	text	xpath_nodeset(document, query, toptag, itemtag)と同様ですが、結果はtoptagを省きます。
xpath_list(document, query, separator)	text	この関数は複数の値を指定した区切り文字で区切って返します。例えば、区切り文字が、ならばValue 1,Value 2,Value 3となります。
xpath_list(document, query)	text	これは、,を区切り文字として使用する、上の関数のラップです。

F.41.3. xpath_table

xpath_table(text key, text document, text relation, text xpaths, text criteria) returns setof record

`xpath_table`は各文書集合に対するXPath問い合わせ集合を評価し、結果をテーブルとして返すテーブル関数です。元文書テーブルのプライマリキーフィールドが結果の第一列として返されますので、結果セットを容易に結合で使用することができます。パラメータについては[表F-33](#)で説明します。

表 F-33. `xpath_table`のパラメータ

パラメータ	説明
key	"key"フィールドの名前です。これは、出力テーブルの第一列として使用される単なるフィールドです。つまり、これは各出力行の出現元を識別するレコードです。（後述の複数値に関する注記を参照してください。）
document	XML文書を含むフィールドの名前です。
relation	文書を含むテーブルまたはビューの名前です。
xpaths	で区切られた、1つ以上のXPath式です。
criteria	WHERE句の内容です。これは省略することができません。リレーション内の全行を処理したい場合は <code>true</code> または <code>1=1</code> を使用してください。

（XPath文字列を除く）これらのパラメータは普通のSQL `SELECT` 文に単純に置換されます。このため、多少の柔軟性があります。

```
SELECT <key>, <document> FROM <relation> WHERE <criteria>
```

文は上の通りですので、これらのパラメータにはそれぞれの場所で有効なものであれば何でもよいわけです。このSELECTの結果は正確に2つの列を返さなければなりません（キーまたは文書に対して複数のフィールドを列挙させようとしない限りです）。この簡略された手法では、SQLインジェクション攻撃を防ぐためにユーザから与えられた値をすべて検証しなければならないことに注意してください。

この関数は、出力列を指定するための`AS`句を付けた`FROM`式内で使用されなければなりません。以下に例を示します。

```
SELECT * FROM
xpath_table('article_id',
            'article_xml',
            'articles',
            '/article/author|/article/pages|/article/title',
            'date entered > '2003-01-01' ')
AS t(article_id integer, author text, page_count integer, title text);
```

この`AS`句は、出力テーブルの列名とその型を定義します。先頭が"key"フィールド、残りがXPath問い合わせに対応します。結果列よりより多くのXPath問い合わせが存在する場合、余った問い合わせは無視されます。XPath問い合わせより多くの結果列が存在する場合は余った列はNULLになります。

この例で`page_count`結果列が整数として定義されていることに注意してください。関数は内部的に文字列表現で扱います。このため、出力内で整数で扱いたいと言っている時、XPath結果の文字列表現を取り出し、整数（または`AS`句で要求した任意の型）に変換するためにPostgreSQLの入力関数を使用します。例えば結果が空など、変換できない場合はエラーになります。ですので、データに何らかの問題があると考えられる場合、列型として`text`に限定する方がよいかもしれません。

`SELECT`文の呼び出しでは、単なる`SELECT *`でなければならない必要性はありません。出力列を名前で参照することも他のテーブルと結合することも可能です。この関数は希望の何らかの操作（例えば集約、結合、ソートなど）を行うことができる仮想テーブルを生成します。このため以下をより複雑な例として示すことができます。

```
SELECT t.title, p.fullname, p.email
FROM xpath_table('article_id', 'article_xml', 'articles',
                '/article/title|/article/author/@id',
                'xpath_string(article_xml,''/article/@date'') > '2003-03-20' ')
AS t(article_id integer, title text, author_id integer),
tblPeopleInfo AS p
WHERE t.author_id = p.person_id;
```

当然ながら、簡便にするためにこれをすべてビューとして包み隠すことができます。

F.41.3.1. 複数値の結果

`xpath_table`関数は各XPath問い合わせの結果が複数の値を持つ可能性があることを前提としています。このため、この関数が返す行数は入力文書の数と同じにならない可能性があります。返される最初の行には各問い合わせの最初の結果が、2番目の行には各問い合わせの2番目の結果が含まれます。問い合わせの1つが他よりも少ない値を持つ場合は代わりにNULL値が返されます。

指定したXPath問い合わせが単一の結果（おそらく一意な文書識別子）のみを返すことがユーザが分かっている場合があります。もしこれを複数の結果を返すXPathと一緒に使用されると、単一値の結果は結果の最初の行にのみ現れます。この解決方法はより単純なXPath問い合わせに対する結合部分としてキーフィールドを使用することです。以下に例を示します。

```
CREATE TABLE test (
  id int PRIMARY KEY,
  xml text
);

INSERT INTO test VALUES (1, '<doc num="C1">
<line num="L1"><a>1</a><b>2</b><c>3</c></line>
<line num="L2"><a>11</a><b>22</b><c>33</c></line>
</doc>');

INSERT INTO test VALUES (2, '<doc num="C2">
<line num="L1"><a>111</a><b>222</b><c>333</c></line>
<line num="L2"><a>111</a><b>222</b><c>333</c></line>
</doc>');

SELECT * FROM
  xpath_table('id','xml','test',
    '/doc/@num|/doc/line/@num|/doc/line/a|/doc/line/b|/doc/line/c',
    'true')
  AS t(id int, doc_num varchar(10), line_num varchar(10), val1 int, val2 int, val3 int)
WHERE id = 1 ORDER BY doc_num, line_num
```

id	doc_num	line_num	val1	val2	val3
1	C1	L1	1	2	3
1		L2	11	22	33

各行に`doc_num`を付けるためには、2つの`xpath_table`を呼び出し、その結果を結合することです。

```
SELECT t.*,i.doc_num FROM
  xpath_table('id', 'xml', 'test',
    '/doc/line/@num|/doc/line/a|/doc/line/b|/doc/line/c',
    'true')
  AS t(id int, line_num varchar(10), val1 int, val2 int, val3 int),
  xpath_table('id', 'xml', 'test', '/doc/@num', 'true')
  AS i(id int, doc_num varchar(10))
WHERE i.id=t.id AND i.id=1
ORDER BY doc_num, line_num;
```

id	line_num	val1	val2	val3	doc_num
1	L1	1	2	3	C1
1	L2	11	22	33	C1

(2 rows)

F.41.4. XSLT関数

`libxslt`がインストールされている場合、以下の関数を使用することができます。

F.41.4.1. xslt_process

```
xslt_process(text document, text stylesheet, text paramlist) returns text
```

この関数はXSLスタイルシートを文書に適用し、変換した結果を返します。`paramlist`は、'a=1,b=2'という形で指定された、変換で使用するパラメータ代入式のリストです。パラメータ解析はあまり熟考されたものではないことに注意してください。パラメータ値にカンマを入れることができません。

また、変換用のパラメータを渡さない、2つのパラメータを取るバージョンの`xslt_process`も存在します。

F.41.5. 作者

John Gray <igray@azuli.co.uk>

本モジュールの開発はTorchbox Ltd. (www.torchbox.com)が後援です。PostgreSQLと同じBSDライセンスです。

付録 G. 追加で提供されるプログラム

目次

G.1. [クライアントアプリケーション](#)

G.2. [サーバアプリケーション](#)

この付録と前の付録には、PostgreSQL配布物のcontribディレクトリにあるモジュールに関する情報があります。 contribセクションの概要や、特にcontribにあるサーバエクステンションやプラグインに関する情報は、[付録F](#)を参照してください。

この付録ではcontribにあるユーティリティプログラムを説明します。 ソースからでもパッケージシステムからでも、いったんインストールされるとそれらはPostgreSQLがインストールされた場所のbinディレクトリに入り、他のプログラムと同様に使用することができます。

[前のページ](#)

xml2

[ホーム](#)
[上に戻る](#)[次のページ](#)
クライアントアプリケーション

G.1. クライアントアプリケーション

目次

[oid2name](#) -- OIDとPostgreSQLデータディレクトリ内のファイルノードを解決する

[pgbench](#) -- PostgreSQLに対してベンチマーク試験を行う

[vacuumlo](#) -- PostgreSQLデータベースから孤児となったラージオブジェクトを削除する

このセクションでは、`contrib`にあるPostgreSQLのクライアントアプリケーションを説明します。それらは、データベースサーバがどこで稼働しているかに依存せず、どこからでも実行することが出来ます。PostgreSQLのコア配布物に含まれるクライアントアプリケーションに関する情報は、[参照II, PostgreSQLクライアントアプリケーション](#)を参照してください。

oid2name

名前

oid2name -- OIDとPostgreSQLデータディレクトリ内のファイルノードを解決する

概要

oid2name [*option...*]

説明

oid2nameは、管理者がPostgreSQLで使用されるファイル構造を確認することを補助するユーティリティプログラムです。 使用できるようになるためには、[第56章](#)で説明されるデータベースファイル構造についての知識が必要です。

注意: "oid2name" という名前は歴史的なものであり、これを使用する場合のほとんどでは、本当はテーブルのファイルノード番号（これはデータベースディレクトリ内で可視なファイル名）が関係しますので、実際誤解されやすいものです。 テーブルのOIDとテーブルファイルノードの違いを確実に理解してください。

oid2nameは対象データベースに接続し、OID、ファイルノード、テーブル名情報を抽出します。 また、データベースOIDまたはテーブル空間OIDを示すようにさせることもできます。

オプション

oid2nameは以下のコマンドライン引数を受け付けます。

`-f filenode`

*filenode*というファイルノードを持つテーブルの情報を表示します。

`-i`

一覧にインデックスおよびシーケンスを含めます。

`-o oid`

*oid*というOIDを持つテーブルの情報を表示します。

`-q`

ヘッダを省略します。（スクリプト処理に適しています）

`-s`

テーブル空間OIDを表示します。

`-S`

システムオブジェクト（`information_schema`、`pg_toast`、`pg_catalog`スキーマ内に存在するもの）を含めます。

`-t tablename_pattern`

*tablename_pattern*に一致するテーブル（複数可）の情報を表示します。

`-V`

`--version`

oid2nameのバージョンを表示し、終了します。

`-x`

表示対象の各オブジェクトに関してさらに情報を表示します。テーブル空間名、スキーマ名、OID。

```
-?
--help
```

oid2nameのコマンドライン引数の説明を表示し、終了します。

またoid2nameは以下の接続用のパラメータに関するコマンドライン引数を受け付けます。

```
-d database
```

接続データベース

```
-H host
```

データベースサーバのホスト

```
-p port
```

データベースサーバのポート

```
-U username
```

接続ユーザ名

```
-P password
```

パスワード(廃止予定。コマンドラインに記述することはセキュリティ的に危険です。)

特定のテーブルを表示するために、`-o`、`-f`、`-t`を使用して表示するテーブルを選択してください。 `-o`はOIDを、`-f`はファイルノードを、`-t`はテーブル名（実際はLIKEパターンです。ですので`f%%`などが使用できます）を引数として取ります。 これらのオプションを必要なだけ使用することができます。 一覧には、オプションのいずれかで一致したオブジェクトがすべて含まれます。 しかしこれらのオプションでは、`-d`で指定したデータベース内に存在するオブジェクトしか表示しないことに注意してください。

`-o`、`-f`、`-t`のいずれも指定せずに`-d`を指定した場合、`-d`で指定したデータベース上のすべてのテーブルを列挙します。 このモードでは、`-s`および`-i`スイッチが何を列挙するかを制御します。

`-d`も指定しなかった場合、データベースOIDの一覧を示します。 他にも`-s`を指定してテーブル空間の一覧を得ることもできます。

注釈

oid2nameは破損のないシステムカタログで実行中のデータベースサーバが必要です。 したがって、破滅的にデータベースが破損したような状況からの復旧での使用には制限されています。

例

```
$ # what's in this database server, anyway?
$ oid2name
All databases:
  Oid Database Name Tablespace
-----
 17228      alvherre  pg_default
 17255      regression pg_default
 17227      template0  pg_default
    1      template1  pg_default

$ oid2name -s
All tablespaces:
  Oid Tablespace Name
-----
  1663      pg_default
  1664      pg_global
 155151      fastdisk
 155152      bigdisk

$ # OK, let's look into database alvherre
$ cd $PGDATA/base/17228

$ # get top 10 db objects in the default tablespace, ordered by size
$ ls -ls * | head -10
-rw----- 1 alvherre alvherre 136536064 sep 14 09:51 155173
-rw----- 1 alvherre alvherre 17965056 sep 14 09:51 1155291
2402/2500
```

```

-rw----- 1 alvherre alvherre 1204224 sep 14 09:51 16717
-rw----- 1 alvherre alvherre 581632 sep 6 17:51 1255
-rw----- 1 alvherre alvherre 237568 sep 14 09:50 16674
-rw----- 1 alvherre alvherre 212992 sep 14 09:51 1249
-rw----- 1 alvherre alvherre 204800 sep 14 09:51 16684
-rw----- 1 alvherre alvherre 196608 sep 14 09:50 16700
-rw----- 1 alvherre alvherre 163840 sep 14 09:50 16699
-rw----- 1 alvherre alvherre 122880 sep 6 17:51 16751

```

\$ # I wonder what file 155173 is ...

\$ oid2name -d alvherre -f 155173

From database "alvherre":

Filenode	Table Name
155173	accounts

\$ # you can ask for more than one object

\$ oid2name -d alvherre -f 155173 -f 1155291

From database "alvherre":

Filenode	Table Name
155173	accounts
1155291	accounts_pkey

\$ # you can mix the options, and get more details with -x

\$ oid2name -d alvherre -t accounts -f 1155291 -x

From database "alvherre":

Filenode	Table Name	Oid	Schema	Tablespace
155173	accounts	155173	public	pg_default
1155291	accounts_pkey	1155291	public	pg_default

\$ # show disk space for every db object

\$ du [0-9]* |

> while read SIZE FILENODE

> do

> echo "\$SIZE `oid2name -q -d alvherre -i -f \$FILENODE`"

> done

16 1155287 branches_pkey

16 1155289 tellers_pkey

17561 1155291 accounts_pkey

...

\$ # same, but sort by size

\$ du [0-9]* | sort -rn | while read SIZE FN

> do

> echo "\$SIZE `oid2name -q -d alvherre -f \$FN`"

> done

133466 155173 accounts

17561 1155291 accounts_pkey

1177 16717 pg_proc_prname_args_nsp_index

...

\$ # If you want to see what's in tablespaces, use the pg_tblspc directory

\$ cd \$PGDATA/pg_tblspc

\$ oid2name -s

All tablespaces:

Oid	Tablespace Name
1663	pg_default
1664	pg_global
155151	fastdisk
155152	bigdisk

\$ # what databases have objects in tablespace "fastdisk"?

\$ ls -d 155151/*

155151/17228/ 155151/PG_VERSION

\$ # Oh, what was database 17228 again?

\$ oid2name

All databases:

Oid	Database Name	Tablespace
17228	alvherre	pg_default
17255	regression	pg_default
17227	template0	pg_default
1	template1	pg_default

\$ # Let's see what objects does this database have in the tablespace.

\$ cd 155151/17228

\$ ls -l

total 0

-rw----- 1 postgres postgres 0 sep 13 23:20 155156

\$ # OK, this is a pretty small table ... but which one is it?

\$ oid2name -d alvherre -f 155156

```
From database "alvherre":
  Filenode  Table Name
-----
    155156         foo
```

作者

B. Palmer <bpalmer@crimelabs.net>

[前のページ](#)
クライアントアプリケーション

[ホーム](#)
[上に戻る](#)

[次のページ](#)
pgbench

pgbench

名前

pgbench -- PostgreSQLに対してベンチマーク試験を行う

概要

```
pgbench -i [option...] [dbname]

pgbench [option...] [dbname]
```

説明

pgbenchはPostgreSQL上でベンチマーク試験を行う単純なプログラムです。これは同一のSQLコマンドの並びを何度も実行します。複数の同時実行データベースセッションで実行することもできます。そして、トランザクションの速度（1秒当たりのトランザクション数）の平均を計算します。デフォルトでpgbenchは、1トランザクション当たり5つのSELECT、UPDATE、INSERT コマンドを含むおおよそTPC-Bに基いたシナリオを試験します。しかし、独自のトランザクションスクリプトファイルを作成することで他の試験ケースを簡単に実行することができます。

pgbenchの典型的な出力を以下に示します。

```
transaction type: TPC-B (sort of)
scaling factor: 10
query mode: simple
number of clients: 10
number of threads: 1
number of transactions per client: 1000
number of transactions actually processed: 10000/10000
tps = 85.184871 (including connections establishing)
tps = 85.296346 (excluding connections establishing)
```

最初の6行はいくつかの最重要パラメータの設定を表示しています。次行が完了トランザクション数と予定トランザクション数です（後者は単なるクライアント数とクライアント毎のトランザクション数の積算結果です。）実行が完了する前に失敗しない限りこれは等しくなります。（-Tモードでは、トランザクションの実際の数が表示されます）最後の2行は、データベースセッションを開始するための時間を含める場合と含めない場合の1秒当たりのトランザクション数を示します。

デフォルトのTPC-Bと似たトランザクション試験では、あらかじめ設定する特定のテーブルが必要です。これらのテーブルを作成し、データを投入するためには、-i(初期化)オプションを付けてpgbenchを呼び出さなければなりません。（独自スクリプトを試験する場合、この手順は必要ありません。しかし代わりに試験に必要な何らかの設定を行わなければならないでしょう。）初期化は以下のようになります。

```
pgbench -i [ other-options ] dbname
```

ここでdbnameは試験用に前もって作成されたデータベースの名前です。（またデータベースサーバの接続方法を指定するために、-h、-p、-Uが必要になるかもしれません。）

注意

pgbench -iは4つのテーブルpgbench_accounts、pgbench_branches、pgbench_history、pgbench_tellersを作成します。もしあればこうした名前のテーブルは破壊されます。もし同じ名前のテーブルが存在する場合にはよく注意してください。

デフォルトの"倍数"の1では、テーブルは初期状態で以下の行数を含みます。

table	# of rows
-----	-----
pgbench_branches	1
pgbench_tellers	10
pgbench_accounts	100000
pgbench_history	0

-s（倍数）オプションを使用して行数を増加させることができます（また、ほとんどの目的ではおそらく増加させるべきです）。また、-F（フィルファクタ）オプションをここで使用しても構いません。

一度この必要な設定を行った後、以下のように`-i`を持たないコマンドでベンチマークを行うことができます。

```
pgbench [ options ] dbname
```

ほとんどすべての場合、有用な試験とするためにいくつかのオプションが必要になります。 最重要オプションは`-c`（クライアント数）、`-t`（トランザクション数）、`-T`（制限時間）、`-f`（独自スクリプトファイルの指定）です。 以下の全一覧を参照してください。

オプション

以下では、データベース初期化時に使用されるオプション、ベンチマーク実行時に使用されるオプション、どちらの場合でも使われるオプションの3つに分けて説明します。

初期化用のオプション

pgbenchは以下の初期化用のコマンドライン引数を受け付けます。

`-i`

初期化モードを呼び出すために必要です。

`-F fillfactor`

指定したフィルファクタで`pgbench_accounts`、`pgbench_tellers`、`pgbench_branches`テーブルを作成します。 デフォルトは100です。

`-s scale_factor`

この倍率で生成される行数を積算します。 例えば、`-s 100`は `pgbench_accounts` テーブルに10,000,000行を生成することを意味します。 デフォルトは1です。

`--index-tablespace=index_tablespace`

デフォルトのテーブル空間ではなく、指定したテーブル空間の中にインデックスを作成します。

`--tablespace=tablespace`

デフォルトのテーブル空間ではなく、指定したテーブル空間の中にテーブルを作成します。

`--unlogged-tables`

永続テーブルではなくログを取らないテーブルとしてテーブルを作成します。

ベンチマーク用オプション

pgbenchは以下のベンチマーク用コマンドライン引数を受け付けます。

`-c clients`

模擬するクライアント数、つまり、同時に実行されるデータベースセッション数です。 デフォルトは1です。

`-C`

各クライアントセッションが一度だけ接続を確立するのではなく、各トランザクションが新しい接続を確立します。 これは接続オーバーヘッドを測定する場合に有用です。

`-d`

デバッグ用出力を表示します。

`-D varname=value`

独自スクリプト（後述）で使用される変数を定義します。 複数の`-D`オプションを使用することができます。

`-f filename`

トランザクションスクリプトを`filename`から読み取ります。 詳細は後で説明します。 `-N`、`-S`、`-f`は互いに排他的です。

-j *threads*

pgbench内のワークスレッド数です。複数のスレッドを使用することはマルチCPUマシンで有用になります。各スレッドには管理するクライアントセッションを同じ数与えられるので、クライアント数はスレッド数の倍数でなければなりません。デフォルトは1です。

-l

各トランザクションで費やした時間をログファイルに書き出します。後で詳細を説明します。

-M *querymode*

サーバへ問い合わせを送信するために使用するプロトコルです。

- `simple`: 簡易問い合わせプロトコルを使用します。
- `extended`: 拡張問い合わせプロトコルを使用します。
- プリペアドステートメントを伴う拡張問い合わせプロトコルを使用します。

デフォルトは簡易問い合わせプロトコルです。（詳しい情報は[第46章](#)を参照してください）

-n

試験を実行する前にバキュームを行いません。pgbench_accounts、pgbench_branches、pgbench_history、pgbench_tellers標準テーブルを含まない独自試験シナリオを実行する場合、このオプションは必要です。

-N

pgbench_tellersとpgbench_branchesを更新しません。これは両テーブル上の重度の更新の競合を防ぎますが、TPC-Bのような試験ケースを行わなくなります。

-r

ベンチマーク完了後の各コマンドにおけるステートメント毎の平均レイテンシ(クライアントから見た実行時間)を報告します。詳しくは下を参照してください。

-s *scale_factor*

pgbenchの出力で指定した倍率を報告します。これは組み込みの試験では必要ありません。正確な倍率がpgbench_branchesテーブルの行数を数えることで検出されます。しかし、独自ベンチマーク（-fオプション）を試験している場合、このオプションを使用しない限り、倍率は1として報告されます。

-S

TPC-Bのような試験ではなく読み取りのみのトランザクションを実行します。

-t *transactions*

各クライアントが実行するトランザクション数です。デフォルトは10です。

-T *seconds*

クライアントあたりのトランザクション数を固定で指定するよりも長くテストを実行したい場合、ここに指定した秒数でテストを実行します。-tと-Tは互いに排他的です。

-v

試験前に4つの標準テーブルすべてをバキュームを行います。-nも-vもなければ、pgbenchはpgbench_tellersとpgbench_branchesテーブルをバキュームし、pgbench_history内のデータをすべて消去します。

共通オプション

pgbenchは以下の共通コマンドライン引数を受け付けます。

-h *hostname*

データベースサーバのホスト

-p *port*

データベースサーバのポート

`-U login`

接続ユーザ名

`-V`
`--version`

pgbenchのバージョンを表示し、終了します。

`-?`
`--help`

pgbenchのコマンドライン引数の説明を表示し、終了します。

注釈

pgbenchで実際に実行される"トランザクション"は何か？

デフォルトのトランザクションスクリプトは、1トランザクションで以下の7コマンドを発行します。

```
1 11 BEGIN;

1 11 UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;

1 11 SELECT abalance FROM pgbench_accounts WHERE aid = :aid;

1 11 UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;

1 11 UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;

1 11 INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (:tid, :bid, :aid, :delta,
CURRENT_TIMESTAMP);

1 11 END;
```

`-N`を指定した場合、第4コマンドと第5コマンドはトランザクションに含まれません。`-S`を指定した場合、`SELECT`のみが発行されます。

独自スクリプト

pgbenchは、ファイルから読み込んだトランザクションスクリプト (`-f` オプション) でデフォルトのトランザクションスクリプト (上述) を置き換えて独自のベンチマークシナリオを実行する機能をサポートします。この場合、"トランザクション"はスクリプトファイルの1回の実行として数えられます。複数のスクリプトを指定する (複数の `-f` オプション) ことさえ可能です。この場合、クライアントセッションが新しいトランザクションを開始する時にランダムにスクリプトの1つが選択されます。

スクリプトファイルの書式は、1行1 SQLコマンドという形で、複数行に跨るSQLコマンドはサポートされません。空行および `--` から始まる行は無視されます。スクリプトファイルの行には、pgbench自身が解釈する"メタコマンド" (後述) も記述することができます。

スクリプトファイル向けの簡単な変数置換機能があります。上で説明したように変数を `-D` コマンドラインオプションで設定することができます。また、後で説明するようにメタコマンドで設定することもできます。`-D` コマンドラインオプションで設定された変数の他に、`scale` 変数は現在の倍率に設定されています。一度設定すると、`:variablename` と記述することで変数の値はSQLコマンドに挿入されます。1つ以上のクライアントセッションを実行する時、セッション毎に独自の変数群を持ちます。

スクリプトファイルメタコマンドはバックスラッシュ (`\`) から始まります。メタコマンドへの引数は空白文字で区切られます。以下のメタコマンドがサポートされています。

`\set varname operand1 [ operator operand2 ]`

`varname` 変数を計算された整数値に設定します。各 `operand` は整数定数か整数値を持つ変数への `:variablename` 参照のいずれかです。`operator` は `+`、`-`、`*`、`/` を取ることができます。

例

```
\set ntellers 10 * :scale
```

```
¥setrandom varname min max
```

*varname*変数を*min*から*max*までの範囲のランダムな整数値に設定します。 各制限は整数定数か整数値を持つ変数への:*variablename*参照のいずれかです。

例

```
¥setrandom aid 1 :naccounts
```

```
¥sleep number [ us | ms | s ]
```

スクリプトの実行をマイクロ秒 (*us*)、ミリ秒 (*ms*)、秒 (*s*) 単位で指定した間待機させます。 単位を省略した場合、デフォルトは秒です。 *number*は整数定数か整数値を持つ変数への:*variablename*参照のいずれかです。

例

```
¥sleep 10 ms
```

```
¥setshell varname command [ argument ... ]
```

*command*シェルコマンドの結果を*varname*変数に設定します。 このコマンドは標準出力を通して整数値を返さなければなりません。

*argument*は、テキスト定数または任意の型の変数を参照する:*variablename*とすることができます。 コロンから始まる*argument*を使用したい場合、*argument*の先頭にさらにコロンを付けなければなりません。

例:

```
¥setshell variable_to_be_assigned command literal_argument :variable ::literal_starting_with_colon
```

```
¥shell command [ argument ... ]
```

¥setshellと同じですが、結果は無視されます。

例:

```
¥shell command literal_argument :variable ::literal_starting_with_colon
```

例えば、組込みのTPC-Bのようなトランザクションの完全な定義を示します。

```
¥set nbranches :scale
¥set ntellers 10 * :scale
¥set naccounts 100000 * :scale
¥setrandom aid 1 :naccounts
¥setrandom bid 1 :nbranches
¥setrandom tid 1 :ntellers
¥setrandom delta -5000 5000
BEGIN;
UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;
UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;
INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (:tid, :bid, :aid, :delta,
CURRENT_TIMESTAMP);
END;
```

このスクリプトにより、トランザクションを繰り返す度に異なる、ランダムに選ばれた行を参照することができます。（この例はまた、各クライアントセッションがなぜ独自の変数を持つことが重要なのかも表しています。 これがないと、異なる行を独立して参照することができないのです。）

トランザクション毎のログ処理

-lオプションを使用すると、pgbenchは各トランザクションで要した時間をログファイルに書き出します。 ログファイルの名前はpgbench_log.*nnn*です。 ここで*nnn*はpgbenchプロセスのPIDです。 -jオプションが2以上の場合、それぞれ独自のログファイルを持つ複数のワークスレッドが生成されます。 最初のワークは標準的な単一ワークの場合と同じ名前を持つログファイルを使用します。 他のワーク用の追加のログファイルはpgbench_log.*nnn.mmm*のように命名されます。 ここで*mmm*は1から始まる各ワークの連番です。

ログの書式は以下の通りです。

```
client_id transaction_no time file_no time_epoch time_us
```

ここで*time*はマイクロ秒単位の総トランザクション処理時間、*file_no*は使用されたスクリプトファイルを識別するもの（-fにより複数のスクリプトが指定された場合に有用）、*time_epoch/time_us*はマイクロ秒のオフセットを持つUNIXエポック書式のタイムスタンプ（秒部分付きのISO 8601タイムスタンプの作成に適します）でトランザクションの完了時刻を示します。

以下に出力例を示します。

```
0 199 2241 0 1175850568 995598
0 200 2465 0 1175850568 998079
0 201 2513 0 1175850569 608
0 202 2038 0 1175850569 2663
```

ステートメント毎のレイテンシ

`-r` オプションを付けると、pgbenchは各クライアントにより実行されたトランザクションのステートメント毎の経過時間を収集します。ベンチマークが終了した後、各値の平均値(各ステートメントのレイテンシと呼びます)が報告されます。

標準スクリプトでは、次のような出力になります。

```
starting vacuum...end.
transaction type: TPC-B (sort of)
scaling factor: 1
query mode: simple
number of clients: 10
number of threads: 1
number of transactions per client: 1000
number of transactions actually processed: 10000/10000
tps = 618.764555 (including connections establishing)
tps = 622.977698 (excluding connections establishing)
statement latencies in milliseconds:
    0.004386      ¥set nbranches 1 * :scale
    0.001343      ¥set ntellers 10 * :scale
    0.001212      ¥set naccounts 100000 * :scale
    0.001310      ¥setrandom aid 1 :naccounts
    0.001073      ¥setrandom bid 1 :nbranches
    0.001005      ¥setrandom tid 1 :ntellers
    0.001078      ¥setrandom delta -5000 5000
    0.326152      BEGIN;
    0.603376      UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
    0.454643      SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
    5.528491      UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;
    7.335435      UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;
    0.371851      INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (:tid, :bid, :aid,
:delta, CURRENT_TIMESTAMP);
    1.212976      END;
```

複数のスクリプトファイルが定義された場合、平均値はそれぞれのスクリプトファイル毎に分けて報告されます。

ステートメント毎のレイテンシを計算するために必要となる、追加のタイミング情報を収集することは、オーバーヘッドが加わることに注意してください。これは平均実行速度を遅くし、計測TPSを小さくするでしょう。低下量はプラットフォームとハードウェアに依存して著しく変わります。レイテンシの報告を有効にする、有効にしないで平均TPS値を比較することは、タイミング・オーバーヘッドが顕著かどうかを測定するには良い方法です。

優れた実践

まったく無意味な数値を生み出すようにpgbenchを使用することは非常に簡単です。以下に有意な結果を生み出す手助けとなるガイドラインをいくつか示します。

まず第一に、数秒で終わる試験を決して信用しないでください。 `-t` または `-T` オプションを使って、雑音を取り除くために、少なくとも数分試験にかかるようにしてください。再現可能な数値を得るために数時間必要になる場合もあります。数回試験を繰り返し、数値が再現できるかどうか確認することを勧めます。

デフォルトのTPC-Bのような試験シナリオでは、初期倍率 (`-s`) を試験予定のクライアント数 (`-c`) の最大値と同程度にしなければなりません。 `pgbench_branches` テーブルには `-s` 行しかありません。また、前トランザクションはその内の1つを更新しようとし、ます。ですので、`-c` 値を `-s` より大きくすると、他のトランザクションを待機するためにブロックされるトランザクションが多くなることは間違いありません。

デフォルトの試験シナリオはまた、テーブルを初期化してからの経過時間に非常に敏感です。テーブル内の不要行や不要空間の累積により結果が変わります。結果を理解するためには、更新された行数とバキューム時期を把握する必要があります。自動バキュームが有効な場合、性能を測定する上で結果は予測できないほど変わる可能性があります。

pgbenchの制限は、多くのクライアントセッションを試験しようとする際にpgbench自身がボトルネックになる可能性があることです。これは、データベースサーバとは別のマシンでpgbenchを実行することで緩和させることが可能です。しかし、多少のネットワーク遅延が重要です。同一データベースサーバに対し複数のクライアントマシンから複数のpgbenchインスタンスを同時に実行することが有用かもしれません。

vacuumlo

名前

vacuumlo -- PostgreSQLデータベースから孤児となったラージオブジェクトを削除する

概要

```
vacuumlo [option...] dbname...
```

説明

vacuumloはPostgreSQLデータベースから"孤児になった"ラージオブジェクトをすべて削除する、単純なユーティリティです。データベース内でoidまたはloデータ型列内にまったく現れないOIDを持つすべてのラージオブジェクト（LO）を"孤児になった"ラージオブジェクトとみなします。

これを使用する場合にはまた、[lo](#)モジュール内のlo_manageトリガに興味を持つかもしれません。lo_manageは初期段階で孤児になったLOの生成を防止しようと試みます。

コマンドラインで指名された全てのデータベースに対して処理が行われます。

オプション

vacuumloは以下のコマンドライン引数を受け付けます。

```
-l limit
```

1トランザクションにつき、limit個（デフォルトは1000）より多くのラージオブジェクトを削除しません。削除されるラージオブジェクト毎に一つのロックを取得するため、多数のラージオブジェクトの削除を1トランザクションで行う場合、[max_locks_per_transaction](#)を超える恐れがあります。もし1トランザクションで全ての削除を行いたい場合は、このlimit値を0に指定してください。

```
-n
```

ラージオブジェクトの削除を行わず、単に何が行われるはずかを示します。

```
-v
```

多くの進行メッセージを出力します。

```
-V  
--version
```

vacuumloのバージョンを表示し終了します。

```
-?  
--help
```

vacuumloのコマンドライン引数に関するヘルプを表示し終了します。

vacuumloは接続パラメータとして以下のコマンドライン引数も受け付けます。

```
-h hostname
```

データベースサーバのホスト名です。

```
-p port
```

データベースサーバのポート番号です。

```
-U username
```

接続ユーザ名です。

-w
--no-password

パスワード入力のプロンプトを出さなくします。もし、サーバがパスワード認証を必要としており、.pgpass ファイルを用いる様な、プロンプト入力とは別の手段を通してパスワードを利用できない場合は、接続に失敗します。このオプションは、バッチ処理やスクリプト処理の様なパスワードを入力するユーザがないケースで有効です。

-W

vacuumloは強制的にデータベースに接続する前にパスワード入力を促します。

サーバがパスワード認証を要求する場合vacuumloは自動的にパスワード入力を促しますので、これが重要になることはありません。しかし、vacuumloは、サーバにパスワードが必要かどうかを判断するための接続試行を無駄に行います。こうした余計な接続試行を防ぐために-wの入力が有意となる場合もあります。

注意

vacuumloは下記の手法で動作します。まずvacuumloは選択されたデータベース内のラージオブジェクトのOIDをすべて含む一時テーブルを構築します。そしてデータベース内でoid型またはlo型を型として持つ全列をスキャンし、一時テーブルから一致する項目を削除します。(注意:これらの名前の型のみが対象となります。特に、これらの型を伴ったドメインなどはスキャン対象にはなりませんので注意が必要です。)一時テーブルに残った項目を孤児LOと識別します。これらが削除されます。

作者

Peter Mount <peter@retap.org.uk>

[前のページ](#)
pgbench

[ホーム](#)
[上に戻る](#)

[次のページ](#)
サーバアプリケーション

G.2. サーバアプリケーション

目次

[pg_archivecleanup](#) -- PostgreSQLWALアーカイブファイルを消去します
[pg_standby](#) -- PostgreSQLウォームスタンバイサーバの作成をサポートします。
[pg_test_fsync](#) -- PostgreSQLの最も高速なwal_sync_methodを決定する
[pg_test_timing](#) -- 時間計測のオーバーヘッドを測定する
[pg_upgrade](#) -- PostgreSQLサーバインスタンスを更新する

このセクションでは、contribにあるPostgreSQLのサーバ関連のアプリケーションを説明します。それらは、一般的にはデータベースサーバが稼働するホスト上で実行されます。PostgreSQLのコア配布物に含まれるサーバアプリケーションに関する情報は、[参照III, PostgreSQLサーバアプリケーション](#)を参照してください。

[前のページ](#)

vacuumlo

[ホーム](#)
[上に戻る](#)[次のページ](#)

pg_archivecleanup

pg_archivecleanup

名前

pg_archivecleanup -- PostgreSQL WAL アーカイブファイルを消去します

概要

pg_archivecleanup [*option...*] *archivelocation oldestkeptwalfile*

説明

pg_archivecleanupは、スタンバイサーバとして動作している際のWALファイルのアーカイブを消去するための `archive_cleanup_command` として使用されるように設計されています ([項25.2](#)参照)。同時に、pg_archivecleanupはWALファイルのアーカイブを消去するためのスタンドアローンのプログラムとしても利用することができます。

pg_archivecleanupを使うためにスタンバイサーバを設定するには、以下を `recovery.conf` 設定ファイルに記述します。

```
archive_cleanup_command = 'pg_archivecleanup archivelocation %r'
```

archivelocation は、WALセグメントファイルを削除するディレクトリです。

`archive_cleanup_command` で使用される場合、論理的に `%r` より前のすべてのWALファイルは *archivelocation* から削除されます。これによって、クラッシュリスタートの機能を維持するための、保持しなければならないファイル数を最小限にします。このパラメータの使用は、*archivelocation* がスタンバイサーバにおいて短期的な保存場所となっている場合には適切ですが、*archivelocation* を長期的なWALアーカイブ領域として使っている場合、または複数のスタンバイサーバが同一のアーカイブログの場所からリカバリしている場合には適切ではありません。

スタンドアローンプログラムとして使用される場合、*oldestkeptwalfile* より論理的に前のWALファイルは、すべて *archivelocation* から削除されます。このモードでは、`.backup` ファイルが指定されると、そのプレフィックス部だけが *oldestkeptwalfile* として使用されます。この機能は、エラーを起こすことなく特定のベースバックアップより前のすべてのWALファイルを削除することを可能にします。以下の例は、`00000001000000037000000010` より古いすべてのファイルを削除する実例です。

```
pg_archivecleanup -d archive 00000001000000037000000010.00000020.backup

pg_archivecleanup: keep WAL file "archive/00000001000000037000000010" and later
pg_archivecleanup: removing file "archive/000000010000000370000000F"
pg_archivecleanup: removing file "archive/000000010000000370000000E"
```

pg_archivecleanupは、*archivelocation* がサーバを実行しているユーザによって読み書き可能なディレクトリであるものと仮定しています。

オプション

pg_archivecleanupは、以下のコマンドライン引数を受け付けます。

```
-d
    stderrに大量のデバッグログを出力します。

-n
    削除されるファイルの名前をstdoutに出力します(実際には削除しません)。

-v
--version
    pg_archivecleanupのバージョンを表示して終了します。

-x extension
```

プログラムがスタンドアローンユーティリティとして使用される場合、削除するファイルを決定する前にファイル名から取り除かれる拡張子を指定します。保存時に圧縮され、そのため圧縮プログラムにより拡張子を付けられたアーカイブを消去するのに特に役に立ちます。例: `-x.gz`

プログラムに渡されるファイル名 `.backup` は拡張子を含んではいけないことに注意して下さい。

-?
--help

`pg_archivecleanup` のコマンドライン引数に関するヘルプを表示して終了します。

注釈

`pg_archivecleanup` は、PostgreSQL 8.0 とそれ以降において、スタンドアローンユーティリティとして動作するように設計されています。また、PostgreSQL 9.0 とそれ以降においてはアーカイブのクリーンアップコマンドとして動作するように設計されています。

`pg_standby` は C 言語で書かれており、必要に応じて修正すべき部分が明確に示されているので、修正の容易なソースコードとなっています。

例

Linux や Unix のシステムでは、以下のように使います。

```
archive_cleanup_command = 'pg_archivecleanup -d /mnt/standby/archive %r 2>>cleanup.log'
```

アーカイブディレクトリは物理的にはスタンバイサーバ上に配置されていますので、`archive_command` は NFS を経由してアーカイブディレクトリにアクセスしますが、それらのファイルはスタンバイサーバにとってはローカルファイルです。この設定では、

- デバッグ出力を `cleanup.log` に生成します。
- アーカイブディレクトリの中から、不要になったファイルを削除します。

作者

Simon Riggs <simon@2ndquadrant.com>

関連項目

[pg_standby](#)

[前のページ](#)
サーバアプリケーション

[ホーム](#)
[上に戻る](#)

[次のページ](#)
`pg_standby`

pg_standby

名前

pg_standby -- PostgreSQL ウォームスタンバイサーバの作成をサポートします。

概要

```
pg_standby [option...] archiveLocation nextWalFile xlogFilePath [restartWalFile]
```

説明

pg_standbyは"ウォームスタンバイ"データベースサーバの作成をサポートします。これは、特定の変更が必要となるカスタマイズ可能なテンプレートを持ち、実運用環境で利用可能なプログラムとして設計されています。

pg_standbyは、標準のアーカイブリカバリからウォームスタンバイに切り替えるために必要な待機コマンド`restore_command`として設計されています。他の設定も必要ですが、それらはすべてメインのサーバマニュアルで説明されています ([項25.2](#)を参照してください)。

pg_standbyを使用して待機サーバを構築するには、`recovery.conf`設定ファイルに以下を追加します。

```
restore_command = 'pg_standby archiveDir %f %p %r'
```

ここで`archiveDir`は、リストアすべきWALセグメントファイルが存在するディレクトリです。

通常、`%r`マクロを使用して`restartWalFile`が指定された場合、このファイルより論理的に前のすべてのWALファイルは`archiveLocation`から削除されます。これによってクラッシュからの再起動ができることを担保しつつ、保持する必要があるファイルの数を最小化します。`archiveLocation`が、この特定の待機サーバで一時使用の領域である場合、このパラメータの使用は適切です。しかし、`archiveLocation`が長期間のWAL保管を目的とした領域である場合には、不適切となります。

pg_standbyは、`archiveLocation`がサーバを所有するユーザから読み取り可能なディレクトリであることを前提とします。また、`restartWalFile` (または`-k`) が指定される場合、`archiveLocation`ディレクトリは書き込み可能である必要があります。

マスタサーバが失敗した時の"ウォームスタンバイ"データベースサーバへフェイルオーバーする方法には、以下の2つがあります。

スマートフェイルオーバー

スマートフェイルオーバーは、アーカイブとして利用可能なすべてのWALファイルを適用した後に、待機サーバが準備完了となります。待機サーバが遅れてもデータロスとなることは全くありませんが、適用されていないWALが大量にある場合、待機サーバが利用可能になるまでには長時間かかるかもしれません。スマートフェイルオーバーのトリガとなるためには、単語`smart`を含むトリガファイルを作成するか、単に空のファイルを作成してください。

ファストフェイルオーバー

ファストフェイルオーバーでは、待機サーバはすぐに準備完了となります。アーカイブ内の未適用のWALファイルは無視され、それらのファイルに記録されていたすべてのトランザクションは失われます。ファストフェイルオーバーのトリガとなるためには、トリガファイルを作成し、単語`fast`を書き込んでください。また、指定した時間内に新しいWALファイルが出現しない場合に、自動的にファストフェイルオーバーを実行するようにpg_standbyを設定することもできます。

オプション

pg_standbyは、以下のコマンドライン引数を受け付けます。

-c

アーカイブからWALファイルをリストアするために `cp` または `copy` コマンドを使用します。これが唯一サポートされている動作ですので、このオプションには意味はありません。

-d

`stderr`に大量のデバッグログを出力します。

-k

`archive_location`からファイルを削除することによって、現在のWALファイルよりも古いWALファイルが、ここで指定した数以上アーカイブ内に保持されないようにします。ゼロ（デフォルト）は`archive_location`からファイルをまったく削除しないことを意味します。`restartwalfile`が指定された場合、このパラメータは警告なく無視されます。アーカイブ内の正確な切り捨て点を決定する際には、そちらの指定方法の方がより正確だからです。PostgreSQL 8.3の時点では、`restartwalfile`パラメータによる指定の方が安全、かつ効率的であるため、このパラメータの使用は廃止予定です。あまりにも小さな値を設定すると、待機サーバの再起動に必要とするファイルも削除されてしまう可能性があり、一方であまりにも大きな値を設定するとアーカイブ領域を無駄に消費します。

-r *maxretries*

コピーが失敗した場合のリトライ回数の最大値を設定します（デフォルトは3です）。失敗する度に、失敗回数に比例して待ち時間が増加するように`sleeptime * num_retries`秒間待機します。そのため、デフォルトでは待機サーバに失敗を返す前に、5秒、10秒、15秒待機することになります。これはリカバリの完了、およびその結果としてスタンバイが完全に成功するであろうと解釈されます。

-s *sleeptime*

リストアすべきWALがアーカイブ内で見つかるかどうか、その確認をする間隔を秒単位で設定します（最大60秒、デフォルト5秒）。デフォルト設定は必ずしも推奨するものではありません。[項25.2](#)を参考に検討してください。

-t *triggerfile*

存在すればフェイルオーバー発生のきっかけとなるトリガファイルを指定します。同一システムに複数のサーバが存在する場合、たとえば`/tmp/pgsql.trigger.5432`などのように構造を持ったファイル名を使用して、どのサーバのトリガが混乱しないようにすることを推奨します。

-V

--version

`pg_standby`のバージョンを表示して終了します。

-w *maxwaittime*

ファストフェイルオーバー実行後に、次のWALファイルを待機する最大秒数を設定します。ゼロ（デフォルト）に設定することは永久に待機することを意味します。デフォルトの設定は必ずしも推奨されません。[項25.2](#)を参考に検討してください。

-?

--help

`pg_standby`のコマンドライン引数に関するヘルプを表示して終了します。

注釈

`pg_standby`は、PostgreSQL 8.2以降で動作するよう設計されています。

PostgreSQL 8.3は、保持しておく必要がある最後のWALファイルを`pg_standby`に渡すための`%r`マクロを提供しています。PostgreSQL 8.2では、アーカイブファイルの削除が必要な場合`-k`オプションを使用しなければなりません。このオプションは8.3でもまだ利用可能ですが、非推奨です。

PostgreSQL 8.4は、`recovery_end_command`オプションを提供しています。このオプションを指定しないと、残ったトリガファイルが問題を引き起こす可能性があります。

`pg_standby`はC言語で書かれており、必要に応じて修正すべき部分が明確に示されているので、修正の容易なソースコードとなっています。

例

LinuxまたはUnixシステムでは以下のように使用できます。

```
archive_command = 'cp %p .../archive/%f'
```

```
restore_command = 'pg_standby -d -s 2 -t /tmp/pgsql.trigger.5442 .../archive %f %p %r 2>>standby.log'
recovery_end_command = 'rm -f /tmp/pgsql.trigger.5442'
```

ここでは、アーカイブディレクトリは物理的には待機サーバ上にあります。そのため、`archive_command`はNFS経由でアーカイブディレクトリにアクセスします。しかし、このファイルは（`ln`の使用を有効にした）待機サーバではローカルです。そのため、以下ようになります。

- `standby.log`にデバッグ用の出力を書き出します。
- 次のWALファイルが利用可能になったかどうかを確認するまで2秒間待機します。
- `/tmp/pgsql.trigger.5442`というトリガファイルが出現すると待機状態を解除し、トリガファイルの内容に従ってフェイルオーバーを実行します。
- 復旧が終了した時点で、トリガファイルを削除します。
- 必要なくなったファイルをアーカイブディレクトリから削除します。

Windowsでは以下のように使用できます。

```
archive_command = 'copy %p ...¥¥archive¥¥%f'
restore_command = 'pg_standby -d -s 5 -t C:¥pgsql.trigger.5442 ...¥archive %f %p %r 2>>standby.log'
recovery_end_command = 'del C:¥pgsql.trigger.5442'
```

`archive_command`ではバックスラッシュを二重にする必要がありますが、`restore_command`や`recovery_end_command`では必要ないことに注意してください。これは以下のような内容になります。

- アーカイブからWALファイルをリストアするために`copy`コマンドを使用します。
- `standby.log`にデバッグ用の出力を書き出します。
- 次のWALファイルが利用可能になったかどうかを確認するまで5秒間待機します。
- `C:¥pgsql.trigger.5442`というトリガファイルが出現すると待機を中止し、トリガファイルの内容に従ってフェイルオーバーを実行します。
- 復旧が終了した時点で、トリガファイルを削除します。
- 必要なくなったファイルをアーカイブディレクトリから削除します。

Windowsの`copy`コマンドは、ファイルが完全にコピーされる前に、最終的なファイルサイズを設定します。これは通常`pg_standby`を誤動作させます。したがって、`pg_standby`は適切なファイルサイズを見てから、いったん`sleeptime`秒待ちます。GNUWin32の`cp`は、ファイルコピーが完了した後にだけ、ファイルサイズを設定します。

Windowsの例では両方のサーバで`copy`を使用していますので、どちらか一方、または両サーバがネットワーク経由でアーカイブディレクトリにアクセスすることになります。

作者

Simon Riggs <simon@2ndquadrant.com>

関連項目

[pg_archivecleanup](#)

[前のページ](#)

[pg_archivecleanup](#)

[ホーム](#)
[上に戻る](#)

[次のページ](#)
[pg_test_fsync](#)

pg_test_fsync

名前

pg_test_fsync -- PostgreSQLの最も高速なwal_sync_methodを決定する

概要

pg_test_fsync [*option...*]

説明

pg_test_fsyncは使用するシステムにおいて最速な[wal_sync_method](#)は何かについて、合理的な見解を提供することを意図したものです。同時に識別されたI/O問題のイベントに診断情報を提供します。しかしpg_test_fsyncで示される差異が、実際のデータベーススループットにおいて差異として現れないことがあります。特に、多くのデータベースサーバはトランザクションログにより速度制限されていないからです。

オプション

pg_test_fsyncは以下のコマンドラインオプションを受け付けます。

-f
--filename

テストデータを書き込むためのファイルの名前を指定します。このファイルはpg_xlogディレクトリがある、または格納する予定のファイルシステムと同じファイルシステムになければなりません。（pg_xlogにはWALファイルが含まれます。）デフォルトは現在のディレクトリ内のpg_test_fsync.outです。

-s
--secs-per-test

1テストあたりの秒数を指定します。テストあたりの時間を多くするほど、テストの精度が向上しますが、実行により時間がかかるようになります。デフォルトは、本プログラムがおよそ30秒で完了することができる、2秒です。

-V
--version

pg_test_fsyncのバージョンを表示し、終了します。

-?
--help

pg_test_fsyncのコマンドライン引数の説明を表示し、終了します。

作者

Bruce Momjian <bruce@momjian.us>

関連項目

[postgres](#)

[前のページ](#)
pg_standby

[ホーム](#)
[上に戻る](#)

[次のページ](#)
pg_test_timing

pg_test_timing

名前

pg_test_timing -- 時間計測のオーバーヘッドを測定する

概要

pg_test_timing [*option...*]

説明

pg_test_timingはシステム上の時間計測のオーバーヘッドを測定し、またシステム時計が逆向きに進まないことを確認するためのツールです。時間のデータの収集に時間のかかるシステムでは、EXPLAIN ANALYZEの結果が不正確になることがあります。

オプション

pg_test_timingでは、以下のコマンド行オプションを指定できます。

-d *duration*
--duration=*duration*

テストの時間を秒単位で指定します。時間を長くすると、少しだけ結果が正確になり、またシステム時計が逆方向に進む問題を検出できる可能性が高くなります。デフォルトのテスト時間は3秒です。

-v
--version

pg_test_timingのバージョンを表示して終了します。

-?
--help

pg_test_timingのコマンドライン引数についてのヘルプを表示して終了します。

使用方法

結果の解釈

良い結果では、個々の時間計測のほとんど(90%以上)が1マイクロ秒未満になります。ループあたりの平均オーバーヘッド(per loop time)の値はさらに小さく、100ナノ秒未満になります。以下の例はIntel i7-860システムでTSC時計を使ったもので、非常に良い性能を示しています。

```
Testing timing overhead for 3 seconds.
Per loop time including overhead: 35.96 nsec
Histogram of timing durations:
< usec:      count    percent
    16:         2    0.00000%
     8:        13    0.00002%
     4:       126    0.00015%
     2:    2999652    3.59518%
     1:    80435604   96.40465%
```

ループあたりの時間(per loop time)とヒストグラム(histogram)で時間の単位が異なることに注意して下さい。ループは数ナノ秒(nsec)以内の精度を持つことができますが、個々の時間計測の呼び出しは1マイクロ秒(usec)までの精度にしかできません。

エグゼキュータの時間計測オーバーヘッドの測定

問い合わせエグゼキュータがEXPLAIN ANALYZEを使って文を実行するとき、要約を表示する他に、個々のオペレーションについての時間計測もされます。次のpsqlプログラムで行を数えれば、システムのオーバーヘッドを調べることができます。

```
CREATE TABLE t AS SELECT * FROM generate_series(1,100000);
\timing
SELECT COUNT(*) FROM t;
EXPLAIN ANALYZE SELECT COUNT(*) FROM t;
```

測定に使ったi7-860システムでは、countの問い合わせを9.8ミリ秒で実行しましたが、EXPLAIN ANALYZEをつけたときは16.6ミリ秒かかりました。どちらの問い合わせもちょうど10万行を処理しています。この6.8ミリ秒の差は、行あたりの時間計測のオーバーヘッドが68ナノ秒であることを示しており、これはpg_test_timingによる推定値の約2倍です。この比較的小さな量のオーバーヘッドでも、countの文の完全な時間計測をすると70%長くかかりました。もっと本質的な問い合わせでは、時間計測のオーバーヘッドはあまり問題にならないでしょう。

時間の測定源の変更

一部の新しいLinuxシステムでは、時間データの収集に使う時計をいつでも変更することができます。2番目の例は、時間の測定源を、より遅いacpi_pmに変更することで、上の速い結果で使われたのと同じシステムでも、遅い結果が出るかもしれないことを示しています。

```
# cat /sys/devices/system/clocksource/clocksource0/available_clocksource
tsc hpet acpi_pm
# echo acpi_pm > /sys/devices/system/clocksource/clocksource0/current_clocksource
# pg_test_timing
Per loop time including overhead: 722.92 nsec
Histogram of timing durations:
  < usec:      count    percent
    16:         3    0.00007%
     8:        563    0.01357%
     4:       3241    0.07810%
     2:     2990371  72.05956%
     1:     1155682  27.84870%
```

この構成では、上のEXPLAIN ANALYZEの例の実行に115.9ミリ秒かかりました。つまり、時間計測に1061ナノ秒のオーバーヘッドとなりますが、やはりこのユーティリティによって直接測定したものに小さな数を掛けた程度の値です。この大きな時間計測のオーバーヘッドは、実際の問い合わせ自体は問い合わせに要した時間の本の一部を占めるだけで、大半はオーバーヘッドによって使われている、ということを示しています。この構成では、多くの時間計測オペレーションを含むEXPLAIN ANALYZEの合計は、時間計測のオーバーヘッドにより非常に大きな値になるでしょう。

FreeBSDでも時間の測定源を実行時に変更することができ、起動時に、選択されたタイマーに関する情報をログ出力します。

```
dmesg | grep "Timecounter"
sysctl kern.timecounter.hardware=TSC
```

他のシステムでは、起動時にのみ時間の測定源を設定することができます。古いLinuxシステムでは"clock"のカーネル設定が、この種の変更をする唯一の方法でした。比較的新しいLinuxでも、時計についての唯一のオプションが"jiffies"であるということもあります。jiffiesはLinuxの古いソフトウェア時計の実装で、十分に速い時間測定のハードウェアがあれば、以下の例のように良い精度を出すことができます。

```
$ cat /sys/devices/system/clocksource/clocksource0/available_clocksource
jiffies
$ dmesg | grep time.c
time.c: Using 3.579545 MHz WALL PM GTOD PIT/TSC timer.
time.c: Detected 2400.153 MHz processor.
$ pg_test_timing
Testing timing overhead for 3 seconds.
Per timing duration including loop overhead: 97.75 ns
Histogram of timing durations:
  < usec:      count    percent
    32:         1    0.00000%
    16:         1    0.00000%
     8:         22    0.00007%
     4:        3010    0.00981%
     2:     2993204    9.75277%
     1:     27694571  90.23734%
```

時計のハードウェアと時間測定の正確さ

コンピュータでの正確な時間情報の収集は通常はハードウェアクロックを用いて行われ、その正確さのレベルは様々です。一部のハードウェアでは、オペレーティングシステムはシステムクロックの時間をほぼ直接にプログラムに渡すことができます。システムクロックは、ある一定の時間ごとの単純な時間の割り込みを起こすチップによって提供されることもあります。いずれの場合でも、オペレーティングシステムのカーネルは、これらの詳細を隠して時間の測定源を提供します。しかし時間の測定源の正確さ、および結果を返す速さは、その基となっているハードウェアに依存して変わります。

時間を正確に保つことができないと、システムが不安定になることがあります。時間の測定源へのすべての変更について、非常に注意深く検査してください。オペレーティングシステムのデフォルトは、正確であることよりも信頼できることを重視して設定されているかも知れません。仮想マシンを使っている場合は、それと互換性があるとして推奨されている時間の測定源を調べてください。仮想マシンではタイマーをエミュレートするのに通常以上の困難があり、ベンダーによって推奨されるオペレーティングシステムごとの設定があることもあります。

タイムスタンプカウンタ(TSC)は、現行世代のCPU上で利用可能な最も正確な時間の測定源です。オペレーティングシステムがTSCクロックをサポートしていて、それが信頼出来るなら、システム時間をTSCで測ることが望ましいです。TSCが正確な時間測定源とならず、信頼できなくなる場合がいくつかあります。古いシステムでは、TSCの時計がCPUの温度に依存して変化するため、時間測定には利用できません。一部の古いマルチコアCPUでTSCを使うと、複数のコアの間で一貫しない時間が得られることがあります。この場合、時計が逆戻りすることがありますが、それがこのプログラムが検査対象としている問題の1つです。また最新のシステムでも、非常に積極的に電力節減を実施する設定の場合には、TSCの時間計測が不正確になることがあります。

新しいオペレーティングシステムには、既知のTSCの問題について検査し、問題が検出されると、遅いが、より安定した時間測定源を使うようにするものもあります。システムがTSC時計をサポートしているけれど、それがデフォルトになっていないとしたら、何らかの正しい理由によって無効にされたのかもしれませんが。一部のオペレーティングシステムでは、発生しうるすべての問題について正しく検知できないかも知れませんが、またTSCが正確でないとわかっている状況でもTSCを利用可能にするかも知れません。

高精度イベントタイマー(HPET)は、それが利用可能で、TSCが不正確なシステムにおいて、望ましいタイマーです。タイマーのチップ自体は、100ナノ秒単位の精度までプログラム可能ですが、システムクロックによっては、そこまでの正確さはないこともあります。

ACPI(Advanced Configuration and Power Interface)は電源制御(PM)タイマーを提供し、Linuxはこれをacpi_pmとして参照します。acpi_pmを利用した時計は、最高で300ナノ秒までの精度を提供します。

古いPCハードウェアで使われていたタイマーには、8254プログラマブルインターバルタイマー(PIT)、リアルタイムクロック(RTC)、APIC(Advanced Programmable Interrupt Controller)タイマー、サイクロン(Cyclone)などがあります。これらのタイマーは、ミリ秒の精度を目指したものです。

作者

Ants Aasma <ants.aasma@eesti.ee>

関連項目

[EXPLAIN](#)

[前のページ](#)
pg_test_fsync

[ホーム](#)
[上に戻る](#)

[次のページ](#)
pg_upgrade

pg_upgrade

名前

pg_upgrade -- PostgreSQL サーバインスタンスを更新する

概要

```
pg_upgrade -b oldbindir -B newbindir -d olddatadir -D newdatadir [option...]
```

説明

pg_upgrade(これまではpg_migratorと呼ばれていました)を使用して、メジャーバージョンへの更新、例えば、8.4.7から現時点のメジャーリリースのPostgreSQL、に通常必要とされたデータのダンプ/リストアを行うことなく、PostgreSQLデータファイル内に格納されたデータをより最新のPostgreSQLメジャーバージョンに移行することができます。これは、例えば9.0.1から9.0.4などマイナーバージョンへの更新では必要ありません。

PostgreSQLのメジャーリリースでは通常、システムテーブルのレイアウトをよく変更する、多くの機能が追加されます。しかし内部データの格納書式はまれにしか変更されません。pg_upgradeはこの事実を使用して、システムテーブルを新しく作成し、古いユーザデータファイルを単に再利用することで、高速なアップグレードを実施します。将来のメジャーリリースで古いデータ書式を読み取ることができなくなるようにデータ格納書式を変更した場合、pg_upgradeではこうしたアップグレードを扱うことができません。(コミュニティはこうした状況を防ごうと考えています。)

pg_upgradeは古いクラスタと新しいクラスタとの間で、例えばコンパイル時の設定に互換性があるかどうか、32ビットバイナリか64ビットバイナリかなど、バイナリ互換性があることを確実にするために最善を尽くします。任意の外部モジュールがバイナリ互換であることも重要ですが、これはpg_upgradeでは検査することができません。

pg_upgradeは8.3.X以降から現時点のPostgreSQLのメジャーリリース(スナップショット版やαリリースを含む)への更新をサポートします。

オプション

pg_upgradeは以下のコマンドライン引数を受け付けます。

```
-b old_bindir
--old-bindir=old_bindir
```

古いクラスタの実行ファイル格納ディレクトリ。PGBINOLD環境変数

```
-B new_bindir
--new-bindir=new_bindir
```

新しいクラスタの実行ファイル格納ディレクトリ。PGBINNEW環境変数

```
-c
--check
```

クラスタの検査のみを行い、データの変更を行いません。

```
-d old_datadir
--old-datadir=old_datadir
```

古いクラスタのデータディレクトリ。PGDATAOLD環境変数

```
-D new_datadir
--new-datadir=new_datadir
```

新しいクラスタのデータディレクトリ。PGDATANEW環境変数

```
-k
--link
```

新しいクラスタにファイルをコピーするのではなく、ハードリンクを使用します。

```
-o options
--old-options options
```

古いpostgresコマンドに直接渡すオプションです。

```
-O options
--new-options options
```

新しいpostgresコマンドに直接渡すオプションです。

```
-p old_port_number
--old-port=old_portnum
```

古いクラスタのポート番号。PGPORTOLD環境変数

```
-P new_port_number
--new-port=new_portnum
```

新しいクラスタのポート番号。PGPORTNEW環境変数

```
-r
--retain
```

正常に完了した場合であってもSQLファイルとログファイルを保持します。

```
-u user_name
--user=user_name
```

クラスタのスーパーユーザの名称。PGUSER環境変数

```
-v
--verbose
```

冗長な内部ログを有効にします。

```
-V
--version
```

バージョン情報を表示し、終了します。

```
-?
-h
--help
```

使用方法を表示し、終了します。

使用方法

pg_upgradeを用いたアップグレードを行う手順を示します。

1 11 古いクラスタの移動（省略可能）

バージョンに関連したインストレーションディレクトリ(例えば/opt/PostgreSQL/9.1)を使用しているのであれば、古いクラスタを移動する必要はありません。この1クリックで済むインストーラはすべて、バージョンに関連したインストレーションディレクトリを使用します。

インストレーションディレクトリがバージョンに関連していない（例えば/usr/local/pgsql）のであれば、新しいPostgreSQLのインストレーションに干渉しないように、現在のPostgreSQLインストレーションディレクトリを移動しなければなりません。一度現在のPostgreSQLサーバを停止させていれば、PostgreSQLのインストレーションの名前を変更しても安全です。古いディレクトリが/usr/local/pgsqlであれば、以下のようにディレクトリ名を変更します。

```
mv /usr/local/pgsql /usr/local/pgsql.old
```

1 11 ソースからのインストールの場合の新しいバージョンの構築

古いクラスタと互換性を持つようなconfigureオプションを付けて新しいPostgreSQLソースを構築してください。pg_upgradeは更新を始める前にすべての設定が互換性を持っていることを確認するためにpg_controldataを検査します。

1 11 新しいPostgreSQLのバイナリのインストール

新しいサーバのバイナリとサポートファイルをインストールしてください。

ソースからインストールする場合、独自の場所に新しいサーバをインストールしたければ、以下のようにprefixを使用してください。

```
gmake prefix=/usr/local/pgsql.new install
```

1 11 pg_upgradeおよびpg_upgrade_supportのインストール

新しいPostgreSQLクラスタにpg_upgradeバイナリおよびpg_upgrade_supportライブラリをインストールしてください。

1 11 新しいPostgreSQLクラスタを初期化

initdbを使用して新しいクラスタを初期化してください。繰り返しますが、古いクラスタに合うように互換性があるinitdbのオプションを使用してください。あらかじめ組み込まれたインストーラの多くは、この手順を自動的に実施します。新しいクラスタを起動する必要はありません。

1 11 独自の共有オブジェクトファイルをインストール

例えば、pgcrypto.so、contribや何らかのその他のソースからインストールしたものなど、古いクラスタで使用していた独自の共有オブジェクトファイル(またはDLL)をすべて、新しいクラスタにインストールしてください。例えばpgcrypto.sqlなどのスキーマ定義はインストールしないでください。これらは古いクラスタから更新されるためです。

1 11 認証の調整

pg_upgradeは古いサーバと新しいサーバに複数回接続します。このため、パスワードの問い合わせが繰り返されることを防ぐために、pg_hba.conf内でtrust認証に設定、あるいは、md5を使用しているのであれば~/.pgpassファイル(項31.15参照)の使用するようにした方が良いでしょう。

1 11 両サーバの停止

両サーバが停止していることを確実にしてください。例えばUnixでは以下を使用します。

```
pg_ctl -D /opt/PostgreSQL/8.4 stop
pg_ctl -D /opt/PostgreSQL/9.0 stop
```

Windowsでは以下

```
NET STOP postgresql-8.4
NET STOP postgresql-9.0
```

または以下を適切なサービス名で使用します。

```
NET STOP pgsql-8.3 (PostgreSQL 8.3以前では異なるサービス名が使用されていました。)
```

1 11 pg_upgradeの実行

古いサーバのものではなく、常に新しいサーバのpg_upgradeバイナリを実行してください。pg_upgradeは古いクラスタおよび新しいクラスタのデータと実行形式ファイルの格納ディレクトリ(bin)の指定を要求します。また、ユーザやポート番号の指定や、コピー(デフォルト)ではなくデータリンクを使用するかどうかを指定することができます。

リンクモードを使用する場合、アップグレードは非常に高速になります(ファイルのコピーがありません)が、更新後に新しいクラスタを一度でも実行してしまうと、古いクラスタにアクセスすることができなくなります。リンクモードはまた、古いクラスタと新しいクラスタのデータディレクトリが同じファイルシステムにあることが必要です。すべてのオプションリストを参照するためにはpg_upgrade --helpを使用してください。

Windowsユーザの場合、管理者アカウントでログオンしなければなりません。また、postgresユーザとしてシェルを起動し、適切なパスを設定してください。

```
RUNAS /USER:postgres "CMD.EXE"
SET PATH=%PATH%;C:\Program Files\PostgreSQL\9.0\bin;
```

そして、以下の例のように、引用符でくくったディレクトリを付けてpg_upgradeを実行してください。

```
pg_upgrade.exe
--old-datadir "C:/Program Files/PostgreSQL/8.4/data"
--new-datadir "C:/Program Files/PostgreSQL/9.0/data"
--old-bindir "C:/Program Files/PostgreSQL/8.4/bin"
--new-bindir "C:/Program Files/PostgreSQL/9.0/bin"
```

起動後、pg_upgradeは2つのクラスタに互換性があるかどうか検証し、その後、アップグレードを行います。検査のみを行うpg_upgrade --checkを使用することができます。この場合は古いサーバは稼動中であっても構いません。ま

たpg_upgrade --checkは、アップグレード後に手作業で行わなければならない調整作業があればその概要を示します。
pg_upgradeは現在のディレクトリに対する書き込み権限を必要とします。

言うまでもありませんが、アップグレード中はクラスタにアクセスしてはいけません。意図しないクライアント接続を防ぐために、デフォルトではpg_upgradeは50432ポートでサーバを稼働します。古いクラスタと新しいクラスタが同時に稼動することはありませんので、両クラスタで同じポート番号を使用することができます。しかし実行中の古いクラスタを検査する時には、新旧で異なるポート番号でなければなりません。

データベーススキーマのリストア中にエラーが発生した場合、pg_upgradeは終了しますので、後述の[ステップ14](#)で示すように古いクラスタに戻さなければなりません。再びpg_upgradeを試すためには、pg_upgradeによるスキーマのリストアが成功するように古いクラスタを変更しなければなりません。問題がcontribモジュールであれば、そのモジュールがユーザデータを格納するために使用されていないことが前提ですが、古いクラスタからそのcontribモジュールをアンインストールし、アップグレードした後に新しいクラスタにそれをインストールする必要があるかもしれません。

1 11 pg_hba.confの復旧

pg_hba.confをtrustを使用するように変更している場合は、元の認証設定に戻してください。また新しいクラスタにおけるこの他の設定ファイル、例えばpostgresql.conf、も古いクラスタに合わせるように調整する必要があります。

1 11 移行後の処理

アップグレード後の処理が必要な場合、pg_upgradeはその終了時に警告を発します。また、管理者として実行しなければならないスクリプトファイルを生成します。このスクリプトファイルは、アップグレード後の処理を必要とするデータベースそれぞれに接続します。各スクリプトは以下を使用して実行しなければなりません。

```
psql --username postgres --file script.sql postgres
```

スクリプトは任意の順番で実行することができ、また、実行が終わったら削除することができます。

注意
一般的には、再構築用のスクリプトの実行が完了するより前に、再構築用のスクリプトで参照されるテーブルにアクセスすることは安全ではありません。これを行うと間違った結果が生じたり、性能が劣化することがあり得ます。再構築用のスクリプトで参照されないテーブルにはすぐにアクセスすることができます。

1 11 統計情報

オプティマイザの統計情報はpg_upgradeにより転送されませんので、アップグレード後に統計情報を再生成するコマンドを実行するように指示されます。

1 11 古いクラスタの削除

更新処理が満足いくものであれば、pg_upgradeの終了時に示されたスクリプトを使用して、古いクラスタのデータディレクトリを削除することができます。(bin、shareなど) 古いインストールディレクトリは削除することもできます。

1 11 古いクラスタへの戻し

pg_upgradeを実行した後に古いクラスタに戻したい場合、いくつかの選択肢があります。

- --checkを付けたpg_upgradeを実行した場合、古いクラスタには変更はまったくなされていませんので、いつでも再使用することができます。
- --linkを付けたpg_upgradeを実行した場合、データファイルは古いクラスタと新しいクラスタとで共有されます。もし新しいクラスタを起動していた場合、新しいサーバはこの共有されたファイルに書き出しを行いますので、古いクラスタで使用することは危険です。
- --linkを付けずにpg_upgradeを実行した場合、あるいは、新しいサーバを起動していない場合、リンク処理が始まると、\$PGDATA/global/pg_controlに.old接尾辞が追加される点を除き、古いクラスタは変更されません。古いクラスタを再度使用するためには、\$PGDATA/global/pg_controlから.old接尾辞を取り除きます。その後、古いクラスタを再起動することができます。

pg_upgradeは次のOIDを参照するreg*システムデータ型を含むデータベースのアップグレードをサポートしません。
regproc、regprocedure、regoper、regoperator、regconfig、regdictionary。 (regtypeはアップグレード可能です。)

インストールに影響する場合、失敗、再構築、インデックス再作成はすべてpg_upgradeにより報告されます。 テーブルおよびインデックスを再構築するアップグレード後処理スクリプトは自動的に生成されます。

展開試験を行うのであれば、古いクラスタのスキーマのみをコピーしたものを作成し、ダミーデータを挿入してから、アップグレードを行ってください。

設定ファイルしか持たないディレクトリを使用するPostgreSQL 9.2よりも前のクラスタをアップグレードする場合、例えば-d /real-data-directory -o '-D /configuration-directory'のように、実際のデータディレクトリの場所をpg_upgradeに通知しなければならず、さらに設定用ディレクトリの場所をサーバに通知しなければなりません。

デフォルト以外のUnixドメインソケットディレクトリを使用する、または新しいクラスタのデフォルトとは異なるデフォルトを使用する9.1より前の古いサーバを使用している場合、古いサーバのソケット位置を指し示すようにPGHOSTを設定してください。(これはWindowsでは関係ありません。)

サーバは書き込みを許可しなければなりませんので、ログ SHIPPING スタンバイサーバ(項25.2)をアップグレードすることができません。最も簡単な方法は、プライマリをアップグレードし、スタンバイを再構築するためにrsyncを使用することです。プライマリを停止している間、またはベースバックアップ(項24.3.2)の一部としてrsyncを実行して古いスタンバイクラスタを上書きすることができます。

リンクモードを使用したいが、新しいクラスタを起動した時に古いクラスタを変更させたくない場合は、古いクラスタのコピーを取得してからリンクモードでアップグレードを行ってください。有効な古いクラスタのコピーを取得するためには、サーバ稼動中にrsyncを使用して古いクラスタの変動があるかもしれないコピーを作成し、古いサーバを停止させた後に、rsyncを再度実行して一貫性を保つために何らかの変更をコピーに反映させます。例えば項24.3.3で説明したpostmaster.pidなど、一部のファイルを除外したいと考えるかもしれません。

PostgreSQL 8.3からの移行における制限

PostgreSQL 8.3からアップグレードを行う場合には、それ以降のリリースのPostgreSQLからのアップグレードの場合にはない追加の制限があります。例えば、ユーザ列が以下のように定義されていた場合、pg_upgradeは正常に8.3から移行できません。

- tsquery データ型
- name データ型が先頭列でない場合。

こうした列はすべて削除し、手作業でそれをアップグレードしなければなりません。

ltree contrib モジュールがデータベース内にインストールされている場合、pg_upgradeは動作しません。

pg_upgradeは以下の場合、テーブルの再作成を要求します。

- ユーザ列がtsvector データ型の場合。

pg_upgradeは以下の場合インデックスの再作成を要求します。

- ハッシュまたはGINインデックス。
- bpcchar_pattern_opsを使用するインデックス。

また、PostgreSQL 8.3より後では、日付時刻データのデフォルトの格納形式は整数型に変わりました。pg_upgradeは、古いクラスタと新しいクラスタで使用される日付時刻データの格納形式が合致するかどうかを検査します。確実に--disable-integer-datetimesを付けてconfigureを実行して新しいクラスタを構築してください。

Windowsユーザの場合、1クリックで済むインストーラとMSIインストーラで使用される整数日付時刻設定が異なるため、バージョン8.3の1クリックで済む配布物からバージョン8.4以降の1クリックで済む配布物への更新のみが可能であることに注意してください。MSIインストーラから1クリックで済むインストーラへの更新を行うことはできません。

関連項目

[initdb](#), [pg_ctl](#), [pg_dump](#), [postgres](#)

付録 H. 外部プロジェクト

目次

- H.1. [クライアントインタフェース](#)
- H.2. [管理ツール](#)
- H.3. [手続き言語](#)
- H.4. [拡張](#)

PostgreSQLは複雑なソフトウェアプロジェクトであり、そのプロジェクト管理は困難です。 PostgreSQLに対する拡張をコアプロジェクトと分離して開発する方がより効率的であることがわかりました。

H.1. クライアントインタフェース

PostgreSQL基本配布物内には、以下の2つのクライアントインタフェースのみが存在します。

- [libpq](#)は基本C言語インタフェースであり、他の多くのクライアントインタフェース構築に必要なため、存在します。
- [ECPG](#)はサーバサイドのSQL文法に依存し、PostgreSQL自体の変更に敏感であるため、存在します。

その他の言語についてのインタフェースは外部プロジェクトのもので、別に提供されています。[表H-1](#)にこれらのプロジェクトの一部を示します。パッケージの中にはPostgreSQLと同じライセンスで提供されていないものがあることに注意してください。ライセンスなど各言語インタフェースの詳細についてはそのWebサイトや文書を参照してください。

表 H-1. 外部管理のクライアントインタフェース

名称	言語	コメント	Webサイト
DBD::Pg	Perl	Perl DBI ドライバ	http://search.cpan.org/dist/DBD-Pg/
JDBC	JDBC	タイプ4 JDBC ドライバ	http://jdbc.postgresql.org/
libpqxx	C++	新しい様式のC++インタフェース	http://pqxx.org/
Npgsql	.NET	.NET データプロバイダ	http://npgsql.projects.postgresql.org/
pgtclng	Tcl		http://sourceforge.net/projects/pgtclng/
psqlODBC	ODBC	ODBCドライバ	http://psqlodbc.projects.postgresql.org/
psycopg	Python	DB API 2.0互換	http://www.initd.org/

H.2. 管理ツール

PostgreSQLで使える管理ツールが複数あります。最も人気のあるのは[pgAdmin III](#)ですが、複数の商用版もあります。

H.3. 手続き言語

PostgreSQLの基本配布には複数の手続き言語が存在します。 [PL/pgSQL](#)、[PL/Tcl](#)、[PL/Perl](#)および[PL/Python](#)です。

さらに、PostgreSQLのコア配布以外で開発、保守される手続き言語も多く存在します。 [表H-2](#)に一部のパッケージを示します。 これらのプロジェクトの中には、PostgreSQLと同じライセンスで提供されないものがあることに注意してください。 ライセンス情報など各手続き言語の詳細については、そのWebサイトや文書を参照してください。

表 H-2. 外部管理の手続き言語

名称	言語	Webサイト
PL/Java	Java	http://pljava.projects.postgresql.org/
PL/PHP	PHP	http://www.commandprompt.com/community/plphp/
PL/Py	Python	http://python.projects.postgresql.org/backend/
PL/R	R	http://www.joeconway.com/plr/
PL/Ruby	Ruby	http://raa.ruby-lang.org/project/pl-ruby/
PL/Scheme	Scheme	http://plscheme.projects.postgresql.org/
PL/sh	Unixシェル	http://plsh.projects.postgresql.org/

H.4. 拡張

PostgreSQLは元々拡張可能となるように設計されています。このため、データベースに読み込まれる拡張は、データベースに組み込まれた機能と同様に働きます。ソースコードに同梱されている`contrib`ディレクトリには複数の拡張が含まれています。[付録F](#)で説明します。他にも[PostGIS](#)など独立して開発されています。PostgreSQLのレプリケーションソリューションですら外部で開発することができます。例えば、人気の高いマスタ/スタンバイレプリケーションツールである [Slony-I](#)はコアプロジェクトと独立して開発されています。

付録 I. ソースコードリポジトリ

PostgreSQLのソースコードはGitバージョン管理システムを使って保存管理されています。 マスタのリポジトリの公的なミラーが使用可能です。このミラーは、マスタのリポジトリに変更があればすぐに更新されます。

Wikihttp://wiki.postgresql.org/wiki/Working_with_GitにGitを使用する際の手順が掲載されています。

ソースのリポジトリからPostgreSQLをビルドするにはbison、flexそしてPerlの適度に新しいバージョンが必要であることに注意してください。 ビルドするために使われるファイルはtarballに含まれているため、配布されているtarballからビルドするためには、これらのツールは必要ではありません。 他のツールの必要条件は、[第15章](#)に示されているものと同様です。

I.1. Gitを使ってソースを入手する

Gitを使用すると、コードのリポジトリのコピーがローカルマシンに作成されます。よってすべての履歴とブランチにオフラインでアクセスすることができます。これは開発もしくはパッチをテストするには最速で最も柔軟性のある方法です。

Git

- 1 11 Gitのインストール版が必要となります。インストール版は<http://git-scm.com>から入手可能です。多くのシステムはデフォルトでGitの比較的新しい版がインストールされているか、パッケージ配布システムにおいて利用可能です。

- 1 11 Gitリポジトリを使用するには、公式ミラーサイトのクローンを作成してください。

```
git clone git://git.postgresql.org/git/postgresql.git
```

これは、ローカルマシンにすべてのリポジトリをコピーします。よってインターネット接続が遅い場合には特に時間がかかるでしょう。ファイルは、カレントディレクトリの`postgresql`のサブディレクトリに配置されるでしょう。

Gitミラーサイトは、例えばファイアウォールがGitプロトコルのアクセスをブロックしているような場合にもHTTPプロトコルでも到達できます。URLのプレフィックスを`http`に変更して以下のようにしてください。

```
git clone http://git.postgresql.org/git/postgresql.git
```

HTTPプロトコルはGitプロトコルと比較すると、効率的ではありません。よって使用する際は遅くなるでしょう。

- 1 11 システムの最新の更新を入手する場合は、`cd`を（ローカルマシンの）リポジトリに対して実行し、次のコマンドを実行してください。

```
git fetch
```

Gitはソースコードを取得する以外に、もっと多くのことが実行できます。より詳細な情報は、Gitのmanページを参照するか、または<http://git-scm.com>のウェブサイトを参照してください。

付録 J. ドキュメント作成

目次

- J.1. [DocBook](#)
- J.2. [ツールセット](#)
- J.3. [文書の構築](#)
- J.4. [文書の起草](#)
- J.5. [スタイルガイド](#)

PostgreSQLの文書には4つの主要なフォーマットがあります。

- 平文：インストール前の情報
- HTML：オンラインでの閲覧と参照
- PDFもしくはPostScript：印刷用
- マニュアルページ：すぐ知りたい時

さらに、PostgreSQLソースツリー全体にわたり、様々な実装に関する問題を記述した平文のREADMEファイルが数多くあります。

HTML文書とマニュアルページは標準の配布物の一部でデフォルトでインストールされます。PDFとPostscript形式の文書は別途ダウンロードすることで入手できます。

J.1. DocBook

文書のソースは、見たところHTMLに似たマークアップ言語であるDocBookで作成されています。これらいずれの言語も、元々他の言語を記述する言語である*Standard Generalized Markup Language*、SGMLのアプリケーションです。ここからは、DocBookとSGML両方の用語が使用されますが、技術的に相互の互換性はありません。

DocBookを使用して作成することで、最終的な見栄えがどうなるかに気を遣わずに技術文書の構造と内容を指定できます。あるドキュメントスタイルによりいくつか存在する最終形式の1つに内容をどのように翻訳するかが定義されます。DocBookはOASISグループにより保守されています。[公式DocBookサイト](#)では役に立つ入門用と参照用の文書、そしてO'Reilly社の本の完全版をオンラインで読むことができます。[NewbieDoc Docbook Guide](#)は初心者にとりとても役に立ちます。[FreeBSDドキュメントプロジェクト](#)でも同様にDocBookを使用していて注目すべき数多くのスタイルに関する指針を含め、役立つ情報があります。

J.2. ツールセット

文書进行处理する過程で次のようなツールが使われます。 そのうちのいくつかは付記されているように省略しても構いません。

[DocBook DTD](#)

DocBookそのものの定義です。 現在は4.2バージョンを使用しており、これより古いまたは新しいバージョンは使用できません。 DocBook DTDのSGML版が必要です。 しかし、マニュアルページの構築には同じバージョンのXML版も必要です。

[ISO 8879 character entities](#)

DocBookに必要なものですが、ISOで保守されているため別に配布されています。

[DocBook DSSSL Stylesheets](#)

DocBookのソースをHTMLのような他のフォーマットに変換する処理手順が含まれています。

[DocBook XSL Stylesheets](#)

これはDocBookから他のフォーマットに変換する別のスタイルシートです。 現在これを使用してマニュアルページと一部のHTMLHelpを作成しています。 このツールチェインを使用してHTMLやPDF出力を作成することもできます。 しかし公式のPostgreSQLリリースではこの目的にはDSSSLスタイルシートを使用します。

現在最低必要なバージョンは1.74.0です。

[OpenJade](#)

SGML进行处理する基本パッケージです。 ここにはSGMLパーサ、(DSSSLスタイルシートを使用してSGMLを他のフォーマットに変換するプログラムである) DSSSLプロセッサ、および数多くの関連ツールが含まれています。 Jadeは現在James ClarkではなくOpenJadeグループが保守しています。

[Libxslt for xsltproc](#)

これは(DSSSLスタイルシート用の処理ツールであるjadeのような) XSLTスタイルシートを取り扱う処理ツールです。

[JadeTeX](#)

お望みならばJadeTeXをインストールしてJadeをフォーマットするバックエンドとしてTeXを使うこともできます。 JadeTeXによってPostscriptまたは(ブックマーク付きの) PDFファイルが作成できます。

とは言っても、JadeTeXでの出力はRTFバックエンドで得られる結果よりも品質が劣ります。 特に問題があるのは表と水平垂直方向のスペーシングに関する多彩な細工に関わる部分です。 さらに、でき上がった結果を手作業で修正することもできません。

文書を作成するために必要な様々なツールのインストール方法についての経験をまとめました。 以下に記載します。 これらのツールは別にパッケージ化されて配布されていることも考えられます。 もしそのような配布物を見つけた場合はdocのメーリングリストに報告してください。 そのような情報をここに付け加えたいと思います。

J.2.1. Linux RPMのインストール

数多くのベンダがディストリビューションの中にDocBook処理用のRPMパッケージ一式を提供しています。 インストールに際しては"SGML"オプションまたはsgml-common、docbook、stylesheets、openjade (あるいはjade) パッケージがあるか確認してください。 sgml-toolsも必要かもしれません。 お手持ちのディストリビューションで提供されていない時は、その他の互換性のある配布元のパッケージを利用できるはずです。

J.2.2. FreeBSDでのインストール

FreeBSDのドキュメントプロジェクトそのものがDocBookのヘビーユーザですから、FreeBSD上で使用できる文書作成用ツールの"port"一式が揃っていることは驚くに当たりません。 FreeBSD上で文書を構築するには以下のportsが必要です。

- textproc/sp

```
textproc/openjade
```

- textproc/iso8879
- textproc/dsssl-docbook-modular
- textproc/docbook-420

/usr/ports/print (tex、jadetex) の中にも興味を引くものが数多くあります。

portは/usr/local/share/sgml/catalog.portsにあるメインカタログファイルを更新しない、または順序が適切でない可能性があります。このファイルの先頭に

```
CATALOG "openjade/catalog"
CATALOG "iso8879/catalog"
CATALOG "docbook/dsssl/modular/catalog"
CATALOG "docbook/4.2/catalog"
```

という行があることを確認してください。もしもファイルを編集したくない場合はSGML_CATALOG_FILES環境変数をカタログファイルにコロンで分割して（上記のように）設定することもできます。

FreeBSDでの文書作成ツールについてのさらに詳しい情報は[FreeBSD Documentation Project's instructions](https://www.freebsd.org/doc/en/books/porters-handbook/chapter-ports.html)にあります。

J.2.3. Debianパッケージ

Debian GNU/Linux用の文書作成パッケージの一式が揃っています。インストールには以下を使います。

```
apt-get install docbook docbook-dsssl docbook-xsl openjade1.3 opensp xsltproc
```

J.2.4. ソースから手作業でインストール

DocBookツールを手作業でインストールするのは何かと複雑ですから、もし既にできているパッケージがあるならばそちらを使った方が良いでしょう。ここでは、標準的なインストール手順として、一般的な機能以外を持たせない標準的な設定のみを説明します。詳しいことはそれぞれのパッケージの文書とSGMLの入門資料を読んでください。

J.2.4.1. OpenJadeのインストール

- 1 11 OpenJadeのインストールは、GNU形式の./configure; make; make installという構築手順で行います。詳しいことはOpenJadeソース配布物の中にあります。要約すると次のようになります。

```
./configure --enable-default-catalog=/usr/local/share/sgml/catalog
make
make install
```

"デフォルトカタログ"をどこに置いたか覚えておいてください。後で必要になります。そのままにしても構いませんが、後でjadeを使用する時にはSGML_CATALOG_FILES環境変数を設定してファイルの場所を指定する必要があります（既にOpenJadeがインストールされていて残りのツール一式を別の場所にインストールしたい場合にはこの方法を使用する手があります）。

注意: 何人かのユーザはPDFを作成する際にOpenjade 1.4develを使用すると次のようなメッセージを受け取り、セグメンテーションフォルトになっています。

```
openjade:./stylesheet.dsl:664:2:E: flow object not accepted by port; only display flow
objects accepted
make: *** [postgres-A4.tex-pdf] Segmentation fault
```

Openjade 1.3に下げてエラーを取り除いてください。

- 1 11 さらにdssslディレクトリからどこか、例えば/usr/local/share/sgml/dssslといったディレクトリにdsssl.dtd、fot.dtd、style-sheet.dtdおよびcatalogをインストールします。ディレクトリ全てを一括コピーすることが一番簡単です。

```
cp -R dsssl /usr/local/share/sgml
```

- 1 11 最後に、/usr/local/share/sgml/catalogファイルを作成して以下の行を付け加えます。

```
CATALOG "dsssl/catalog"
```

（これは[ステップ2](#)でインストールされたファイルへの相対的参照パスです。インストール先のレイアウトを変更している場合にはそれに合わせる必要があります。）

J.2.4.2. DocBook DTDキットのインストール

1 11 [DocBook V4.2](#)を入手します。

1 11 `/usr/local/share/sgml/docbook-4.2`ディレクトリを作成してそこに移動します。（正確にこの場所でなくても構いませんが、ここで説明して行く流れの中では筋が通っているものです。）

```
$ mkdir /usr/local/share/sgml/docbook-4.2
$ cd /usr/local/share/sgml/docbook-4.2
```

1 11 アーカイブを伸長します。

```
$ unzip -a ...../docbook-4.2.zip
```

（アーカイブは現在のディレクトリに伸長されます。）

1 11 `/usr/local/share/sgml/catalog`ファイル（もしくはインストール時にjadeに指定したファイルに）に以下の1行を付け加える編集を行います。

```
CATALOG "docbook-4.2/docbook.cat"
```

1 11 [ISO 8879 character entities archive](#)をダウンロードし、伸長してできたファイルをDocBookのファイルと同じディレクトリに置きます。

```
$ cd /usr/local/share/sgml/docbook-4.2
$ unzip ...../ISOEnts.zip
```

1 11 DocBookとISOファイルを置いたディレクトリで以下のコマンドを実行します。

```
perl -pi -e 's/iso-(.*)\.gml/ISO¥1/g' docbook.cat
```

（この操作によって、DocBookカタログファイルで使われる名称と実際のISO文字エンティティファイル間での重複が整理されます。）

J.2.4.3. DocBook DSSSLスタイルシートのインストール

スタイルシートのインストールは、配布物を伸長して、例えば`/usr/local/share/sgml`のような適当な場所に置きます。（アーカイブは自動的にサブディレクトリを作成します。）

```
$ gunzip docbook-dsssl-1.XX.tar.gz
$ tar -C /usr/local/share/sgml -xf docbook-dsssl-1.XX.tar
```

`/usr/local/share/sgml/catalog`にある通常のカatalogエントリも作成されます。

```
CATALOG "docbook-dsssl-1.XX/catalog"
```

スタイルシートは頻繁に変更されますし、他に使えるバージョンを試してみることは有益ですから、PostgreSQLはこのカタログエントリを使用しません。どのようにスタイルシートを選択するかについての情報は[項J.2.5](#)を参照してください。

J.2.4.4. JadeTeXのインストール

JadeTeXをインストールし、使用するには、toolsとgraphicsパッケージBabel、AMSフォントおよびAMS-LaTeX、PSNFSS拡張と"the 35 fonts"の付属キット、PostScriptを生成するdvipsプログラム、マクロのパッケージとしてfancyhdr、hyperref、minitoc、urlおよびot2encがサポートされているTeXとLaTeX2eが既にインストールされていなくてはなりません。これら全ては、親愛なる隣人でもある[CTAN](#)サイトにあります。TeXの基本システムのインストールはこの紹介文の範囲をはるかに超えています。どのようなシステムにせよ、稼働するTeXのバイナリパッケージがどこかにあるはずです。

JadeTeXをPostgreSQLの文書のソースに適用する前に、TeX内部データ構造の容量を増やさないといけない場合があります。この件に関する詳細はJadeTeXのインストールの説明のところに記載されています。

この操作を終えたところでJadeTeXをインストールできるようになります。

```
$ gunzip jadetex-XXX.tar.gz
$ tar xf jadetex-XXX.tar
$ cd jadetex
$ make install
$ mktexlsr
```

最後の2つの入力にはrootになって行います。

J.2.5. configureによる検出

PostgreSQL本体のプログラムを構築した時のように、文書を構築する際にconfigureスクリプトを実行する必要があります。実行が終わる近辺の次のような出力を確認してください。

```
checking for onsgmls... onsgmls
checking for openjade... openjade
checking for DocBook V4.2... yes
checking for DocBook stylesheets... /usr/share/sgml/docbook/stylesheet/dsssl/modular
checking for collateindex.pl... /usr/bin/collateindex.pl
checking for xsltproc... xsltproc
checking for osx... osx
```

もしonsgmlsあるいはnsgmlsが見つからなかった場合には以降の試験の一部は飛ばされます。nsgmlsはJadeパッケージの一部です。自動検出できなかった場合、環境変数JADE、NSGMLSを渡すことで、これらのプログラムの場所を指定することができます。"DocBook V4.2"が見つからない場合はDocBook DTDキットをjadeが見つけれない場所に置いたか、もしくはカタログファイルが正しく設定されていません。前述のインストレーションのヒントを参照してください。DocBookスタイルシートはいくつか比較的標準となっている場所から探されますので、別の場所に置いた時には、DOCBOOKSTYLE環境変数でその場所を設定してからconfigureを後で再実行します。

[前のページ](#)

DocBook

[ホーム](#)
[上に戻る](#)

[次のページ](#)

文書の構築

J.3. 文書の構築

全ての設定が終了したら、`doc/src/sgml` ディレクトリに移動して以下のコマンドの1つを実行してください（GNU makeを使うのを忘れずに）。

J.3.1. HTML

HTML形式の文書を作成するには次のようにします。

```
doc/src/sgml$ gmake html
```

これはデフォルトでの対象です。出力は`html`サブディレクトリに作成されます。

適切なインデックスを作成するために、構築時何回か同一の処理を行う可能性があります。インデックスを考えず、出力の検証を行いたいだけであれば、以下のように`draft`を使用してください。

```
doc/src/sgml$ gmake draft
```

1つのHTMLページとして文書を作成するためには以下を使用します。

```
doc/src/sgml$ gmake postgres.html
```

J.3.2. マニュアルページ

DocBook `refentry` ページをマニュアルページに対応した`*roff`形式に変換するには、DocBook XSLスタイルシートを使用します。manページもHTML形式と同じようにtarアーカイブで配布されています。マニュアルページを作成するには次のようにします。

```
cd doc/src/sgml
gmake man
```

J.3.3. JadeTeX からの印刷

JadeTeXを使って印刷可能な文書を作成したい時は、以下のコマンドの中から1つを選んでください。

- A4サイズのDVI形式を作るには次のようにします。

```
doc/src/sgml$ gmake postgres-A4.ps
```

U.S.レターサイズの場合は次のようにします。

```
doc/src/sgml$ gmake postgres-US.ps
```

- PDFを作るには

```
doc/src/sgml$ gmake postgres-A4.pdf
```

または、

```
doc/src/sgml$ gmake postgres-US.pdf
```

のようにします。（もちろんPostscriptからもPDFバージョンを作成できますが、PDFを直接生成すれば、ハイパーリンクを含めた拡張機能を持たせるようにできます。）

PostgreSQL文書の構築にJadeTeXを使用する場合、おそらくいくつかのTeXの内部用パラメータを増加させなければなりません。これらは`texmf.cnf` ファイルで設定可能です。本書執筆時点では、以下の設定で動作します。

```
hash_extra.jadetex = 200000
hash_extra.pdfjadetex = 200000
pool_size.jadetex = 2000000
pool_size.pdfjadetex = 2000000
string_vacancies.jadetex = 150000
string_vacancies.pdfjadetex = 150000
max_strings.jadetex = 300000
max_strings.pdfjadetex = 300000
save_size.jadetex = 15000
save_size.pdfjadetex = 15000
```

J.3.4. テキストのはみ出し

テキストが印刷マージンよりも広くなりすぎる場合があります。さらに極端な場合、印刷ページよりも広くなりすぎる場合があります。例えば、改行がテキストや幅広の表などです。過度に幅広のテキストは"Overfull hbox"というメッセージをTeXのログファイル(postgres-US.logまたはpostgres-A4.log)に残します。インチ単位で72ポイントありますので、72ポイントを超えた時におそらく幅が広すぎて印刷ページに合わないとかしらに報告されます(1インチのマージンを仮定)。SGMLテキストがはみ出す可能性があるかどうかを見つけるためには、上のはみ出しに関するメッセージで示されている先頭ページ番号(例えば[50 ###] (page 50))を見つけ、PDFファイル内の次のページ(例えばpage 51)以降を読み、はみ出したテキストを探します。それに合わせてSGMLを調整します。

J.3.5. RTFからの印刷

PostgreSQL文書の印刷できるバージョンをRTFに変換した後、ちょっとした修正をOffice Suiteで施すことで作成できます。特定のOffice Suiteによる機能で、文書を引き続きPDFのPostScriptに変換できます。以下にApplixwareを使ってこの手順を示します。

注意: 現バージョンのPostgreSQL文書は、OpenJadeの大きさの限界とそれを越えた部分での不具合を引き起こすようです。もしRTFバージョンが構築過程で長時間にわたってハングアップし、かつ出力ファイルのサイズが0のままであるような場合は、この問題に遭遇したことになります。(とは言っても、通常の構築には5分から10分を要しますので、あせって中断しないように注意してください。)

Applixware RTFの手入れ

OpenJadeは本文に対してデフォルトのスタイルシートの特定を省略します。過去においてはこの原因不明の問題によって、表内容を生成するまで長い過程を必要としました。しかし、Applixwareに関わっている仲間の助力でこの症状が判明し、しっくり行くようになりました。

1 11 RTFバージョンを生成するためには次のように入力します。

```
doc/src/sgml$ gmake postgres.rtf
```

1 11 全てのスタイル(特にデフォルトスタイル)を正しく指定するようにRTFファイルを修正します。文書にrefentry節がある場合、前の段落を現在の段落に連結されるフォーマット指示を、現在の段落が次の段落に連結するように置き換えなければなりません。これらの修正を行ってくれるfixrtfユーティリティはdoc/src/sgmlにあります。

```
doc/src/sgml$ ./fixrtf --refentry postgres.rtf
```

このスクリプトはその文書の中でのゼロ番目のスタイルとして{\s0 Normal;}を付け加えます。ApplixwareによるとRTF規格は明示的なゼロ番目のスタイルを追加することを禁じていますが、Microsoft Wordはたまたま扱えるようになっています。refentry節を修正するためにスクリプトが\keepnタグを\keepに置き換えます。

1 11 Applixware Wordsで新規文書を開き、RTFファイルをインポートします。

1 11 Applixwareで新規目次(ToC)を生成します。

1 11 そこにあるToCの行の1行目の始めの文字から最終行の最後の文字までを選択します。

1 11 Tools->Book Building->Create Table of Contentsで新規ToCを構築します。ToCに含めるためヘッダの上から3レベルまでを選択します。こうするとRTFからインポートされた既存の行がApplixware本来のToCと入れ替わります。

1 11 Format->Styleを使ってToCのフォーマットを調整します。3つのToCスタイルのそれぞれを選択し、FirstとLeftインデントを調整します。下記の値を使用します。

Style	Firstインデント (インチ)	Leftインデント (インチ)
TOC-Heading 1	0.4	0.4
TOC-Heading 2	0.8	0.8
TOC-Heading 3	1.2	1.2

1 11 文書を通して下記に従って作業します。

- 改ページの調整をします。
- 表の列幅の調整をします。

1 11 ToCのExamplesとFigures部分の右寄せのページ番号を正しい値に修正します。ほんの数分しかかかりません。

- 1 11 索引に何も無い場合は削除します。
- 1 11 目次の再生成と調整を行います。
 - 1 11 ToCフィールドを選択します。
 - 1 11 Tools->Book Building->Create Table of Contentsを選択します。
 - 1 11 Tools->Field Editing->Unprotectを選択してToCを解放します。
 - 1 11 ToCそのもののエントリであるToCの第1行目を削除します。
- 1 11 後での最終編集を容易にするためApplixware Words固有のフォーマットで文書を保存します。
- 1 11 Postscriptフォーマットで文書をファイルに"出力"します。

J.3.6. 平文ファイル

インストールする過程の参照用としていくつかの平文によるファイルが配布されています。INSTALLファイルは[第15章](#)と関係していて、異なった内容を説明するためにちょっとした変更が加えられています。このファイルを生成するには、doc/src/sgml ディレクトリに移動し `gmake INSTALL` と入力します。この操作でNetscape Navigator用の平文INSTALL.htmlが作成され、既存ファイルと同じ場所に置かれます。Netscapeは(lynxやw3mと比べて)HTMLを平文に変換するのに最も品質の良い結果をもたらすようです。

HISTORYファイルも同じようにして `gmake HISTORY` コマンドで作成できます。src/test/regress/READMEファイルの場合は `gmake regress_README` です。

J.3.7. 構文検証

文書の構築にはとても時間がかかります。でも文書ファイルの正しい構文だけを検証する方法があります。以下のように入力します。ほんの数秒しかかかりません。

```
doc/src/sgml$ gmake check
```

[前のページ](#)
ツールセット

[ホーム](#)
[上に戻る](#)

[次のページ](#)
文書の起草

J.4. 文書の起草

SGMLとDocBookを起草する際に、オープンソースとなっている数ある文章編集ツールの中からどれを選んだらよいか悩むことはありません。最も一般的なツールは相応の編集モードを備えたEmacs/XEmacsエディタです。いくつかのシステムにおいては、典型的なフルインストレーションを行った場合、使えるようになっています。

J.4.1. Emacs/PSGML

PSGMLはSGML文書を編集する上で最も一般的で最も強力なモードです。正確に設定されていればEmacsでタグを挿入したりマークアップの整合性をチェックすることができます。HTMLに対しても同じように使えます。ダウンロード、インストール手順、そして詳しい文書については[PSGML Webサイト](#)を参照してください。

PSGMLについて1つ重要なことがあります。ここで本書では、使用する環境としてSGML DTDのメインディレクトリは/usr/local/lib/sgmlであると想定しています。本章の例題のように/usr/local/share/sgmlとなっている場合には、SGML_CATALOG_FILES環境変数で設定するかもしれない（マニュアルでどのようにするかは説明されています）PSGMLのインストールをカスタマイズしなくてはなりません。

（パス名をシステムに合わせ）以下を~/.emacsファイルに書いてください。

```
; ***** for SGML mode (psgml)

(setq sgml-omittag t)
(setq sgml-shorttag t)
(setq sgml-minimize-attributes nil)
(setq sgml-always-quote-attributes t)
(setq sgml-indent-step 1)
(setq sgml-indent-data t)
(setq sgml-parent-document nil)
(setq sgml-exposed-tags nil)
(setq sgml-catalog-files '("/usr/local/share/sgml/catalog"))

(autoload 'sgml-mode "psgml" "Major mode to edit SGML files." t )
```

そして同じファイルに（既に）あるauto-mode-alistにSGMLの項目を加えます。

```
(setq
  auto-mode-alist
  '(("¥¥.sgml$" . sgml-mode)
    ))
```

PSGMLを使う場合、分冊にまたがる独立したファイルを気軽に操作するには、ファイルを編集している時に適切なDOCTYPE宣言を挿入してください。例えば、このソースを操作しているのであれば、付録の章ですので最初の行を以下のようにして、文書をDocBook文書の"appendix"インスタンスとしてください。

```
<!DOCTYPE appendix PUBLIC "-//OASIS//DTD DocBook V4.2//EN">
```

これが意味するところは、SGMLを読む全てのまたどんなものにでも正しく処理ができるようにすることで、nsgmls -s docguide.sgml コマンドで文書を検証できます（とは言っても、全ての文書一式を作成する時にはこの行を削除しなければなりません）。

J.4.2. その他のEmacsモード

GNU Emacsには異なるSGMLモードが含まれています。これはPSGMLほど強力ではありませんが、よりわかりやすく、かつ、軽快に動きます。同時に構文を強調表示（フォントロック）させることができるのでとても便利です。src/tools/editors/emacs.samplesは、このモードに対するサンプル設定を含んでいます。

Norm Walshがフォントロックと入力作業を軽減させるいくつかの機能を付けたDocBook専用の[メジャーモード](#)を提供しています。

J.5. スタイルガイド

J.5.1. リファレンスページ

リファレンスページは標準のレイアウトに従う必要があります。このことにより、ユーザが必要な情報を素早く見つけられるようになり、同時にあるコマンドに関連する全ての特徴を文書化する書き手を励ましてもいます。一貫性は、PostgreSQLの各リファレンスページ間だけでなく、オペレーティングシステムや他のパッケージソフトのリファレンスページとの関係でも求められるものです。そのために、以下のガイドラインが作成されました。このガイドラインのほとんどの部分は、様々なオペレーティングシステムで定められている同様のガイドラインと一貫性を持つものです。

実行可能なコマンドを説明するリファレンスページには、以下の節がこの順序で含まれる必要があります。該当しない節は除外しても構いません。これらと同レベルの節は特殊な状況でのみ追加すべきです。そのような情報は多くの場合、"使用法"節に入れることができます。

名前

この節は自動的に生成されます。ここには、コマンドの名前および機能の簡単な概要が入ります。

概要

この節にはコマンドの構文図が入ります。この概要には、通常、各コマンドラインオプションを記載しません（それらは、以下の節に記載されます）。その代わり、入出力ファイルの宛先などのコマンドラインの主要なコンポーネントを記載します。

説明

コマンドによって何が行われるかを説明する文章です。

オプション

各コマンドラインオプションについて説明するリストです。オプションが数多くある場合、副節を使用することができます。

終了ステータス

成功の場合はゼロを使用し、失敗の場合には非ゼロを使用するプログラムでは、この節を記載する必要はありません。複数の非ゼロの終了コードに異なる意味があれば、ここに記載します。

使用法

プログラムの副言語またはランタイムインタフェースを全て記載します。プログラムが対話型でない場合、通常はこの節を除外されます。それ以外の場合、この節は実行時の特徴のすべてを記載するために使用されます。必要に応じて副節を作成してください。

環境

プログラムが使用する可能性のある全ての環境変数を列挙します。なるべく完全なリストを作成してください。SHELLのような些細に見える変数でも、ユーザに必要なことがあります。

ファイル

プログラムが暗黙的にアクセスする可能性のある全てのファイルを列挙します。つまり、コマンドラインで指定された入出力ファイルではなく、設定ファイルなどを列挙するということです。

診断

プログラムが作成する可能性のある全ての異常な出力についての説明を記載します。全てのエラーメッセージを列挙することは避けてください。作業が大変な割に、実際にはほとんど役に立たないからです。ただし、エラーメッセージにユーザが解析できる標準のフォーマットがあれば、ここに記載してください。

備考

特定の不具合、実装の問題点、セキュリティの考慮事項、互換性の問題など、他の節に該当しない全てのものを記載します。

例

例を記載します。

更新履歴

プログラムの更新履歴に主要な変更点があった場合、ここに列挙します。通常この節は省くことができます。

関連項目

次の順序で列挙されるクロスリファレンスです。他のPostgreSQLコマンドのリファレンスページ、PostgreSQLのSQLコマンドのリファレンスページ、PostgreSQLマニュアルの引用、他のリファレンスページ（オペレーティングシステム、他のパッケージソフトなど）、その他の文書。同一グループに属するものは、アルファベット順に列挙します。

SQLコマンドを説明するリファレンスページには、次の節を含める必要があります。名前、概要、説明、パラメータ、使用法、診断、備考、例、互換性、更新履歴、関連項目。パラメータ節はオプション節と似ていますが、リストに加えることのできるコマンドの句についてより自由度が高いものです。互換性節では、このコマンドがどの程度まで標準SQLに準拠しているか、または、他のどのデータベースシステムに対して互換性があるかを説明します。SQLコマンドの関連項目節では、プログラムへのクロスリファレンスよりも前にSQLコマンドを記載する必要があります。

[前のページ](#)
文書の起草

[ホーム](#)
[上に戻る](#)

[次のページ](#)
頭字語

付録 K. 頭字語

以下はPostgreSQLドキュメントと、PostgreSQLの論議で一般に使用される頭字語の一覧です。

ANSI

[American National Standards Institute \(米国規格協会\)](#)

API

[Application Programming Interface \(エイ・ピー・アイ\)](#)

ASCII

[American Standard Code for Information Interchange \(情報交換用米国標準コード\)](#)

BKI

[Backend Interface \(バックエンドインターフェイス\)](#)

CA

[Certificate Authority \(認証局\)](#)

CIDR

[Classless Inter-Domain Routing \(クラスレスドメイン間ルーティング\)](#)

CPAN

[Comprehensive Perl Archive Network \(シーパン\)](#)

CRL

[Certificate Revocation List \(証明書失効リスト\)](#)

CSV

[Comma Separated Values \(コンマ区切り\)](#)

CTE

[Common Table Expression \(共通テーブル式\)](#)

CVE

[Common Vulnerabilities and Exposures \(共通脆弱性識別子\)](#)

DBA

[Database Administrator \(データベース管理者\)](#)

DBI

[Database Interface \(Perl\) \(\[Perlの\]データベースインターフェイス\)](#)

DBMS

[Database Management System \(データベース管理システム\)](#)

DDL

[Data Definition Language \(データ定義言語\)](#)、CREATE TABLE、ALTER USERなどのSQLコマンド

DML

[Data Manipulation Language \(データ操作言語\)](#)、INSERT、UPDATE、DELETEなどのSQLコマンド

DST

[Daylight Saving Time \(夏時間\)](#)

ECPG

[Embedded C for PostgreSQL \(PostgreSQL用の組み込みC\)](#)

ESQL

[Embedded SQL \(組み込みSQL\)](#)

FAQ

[Frequently Asked Questions \(よくある質問\)](#)

FSM

[Free Space Map \(空き領域マップ\)](#)

GEQO

[Genetic Query Optimizer \(遺伝的問い合わせ最適マイザ\)](#)

GIN

[Generalized Inverted Index \(汎用転置インデックス\)](#)

GiST

[Generalized Search Tree \(汎用検索ツリー\)](#)

Git

[Git](#)

GMT

[Greenwich Mean Time \(グリニッジ標準時\)](#)

GSSAPI

[Generic Security Services Application Programming Interface](#)

GUC

[Grand Unified Configuration](#)、サーバ構成処理を行うPostgreSQLのサブシステム

HBA

[Host-Based Authentication \(ホストベース認証\)](#)

HOT

[Heap-Only Tuples \(ヒープ専用タプル\)](#)

IEC

[International Electrotechnical Commission \(国際電気標準会議\)](#)

IEEE

[Institute of Electrical and Electronics Engineers \(米国電気電子学会\)](#)

IPC

[Inter-Process Communication \(プロセス間通信\)](#)

ISO

[International Organization for Standardization \(国際標準化機構\)](#)

ISSN

[International Standard Serial Number \(国際標準逐次刊行物番号\)](#)

JDBC

[Java Database Connectivity](#)

LDAP

[Lightweight Directory Access Protocol](#)

MSVC

[Microsoft Visual C](#)

MVCC

[Multi-Version Concurrency Control \(多版型同時実行制御\)](#)

NLS

[National Language Support](#)

ODBC

[Open Database Connectivity](#)

OID

[Object Identifier \(オブジェクト識別子\)](#)

OLAP

[Online Analytical Processing](#)

OLTP

[Online Transaction Processing \(オンライントランザクション処理\)](#)

ORDBMS

[Object-Relational Database Management System \(オブジェクトリレーショナルデータベース管理システム\)](#)

PAM

[Pluggable Authentication Modules \(着脱可能認証モジュール\)](#)

PGSQL

[PostgreSQL](#)

PGXS

[PostgreSQL Extension System \(PostgreSQL拡張システム\)](#)

PID

[Process Identifier \(プロセス識別子\)](#)

PITR

[Point-In-Time Recovery \(ポイントインタイムリカバリ\)](#) (Continuous Archiving - 継続的アーカイブ)

PL

[Procedural Languages \(手続き言語\)](#) (サーバ側)

POSIX

[Portable Operating System Interface](#)

RDBMS

[Relational Database Management System \(リレーショナルデータベース管理システム\)](#)

RFC

[Request For Comments](#)

SGML

[Standard Generalized Markup Language \(標準汎用マークアップ言語\)](#)

SPI

[Server Programming Interface](#)

SP-GiST

[Space-Partitioned Generalized Search Tree \(空間分割汎用検索ツリー\)](#)

SQL

[Structured Query Language \(構造化問い合わせ言語\)](#)

SRF

[Set-Returning Function](#)

SSH

[Secure Shell](#)

SSL

[Secure Sockets Layer](#)

SSPI

[Security Support Provider Interface](#)

SYSV

[Unix System V](#)

TCP/IP

[Transmission Control Protocol \(TCP\) / Internet Protocol \(IP\)](#)

TID

[Tuple Identifier \(タプル識別子\)](#)

TOAST

[The Oversized-Attribute Storage Technique \(過大属性格納技法\)](#)

TPC

[Transaction Processing Performance Council \(トランザクション処理性能評議会\)](#)

URL

[Uniform Resource Locator \(統一資源位置指定子\)](#)

UTC

[Coordinated Universal Time \(協定世界時\)](#)

UTF

[Unicode Transformation Format \(ユニコード変換書式\)](#)

UTF8

[Eight-Bit Unicode Transformation Format \(8ビットユニコード変換書式\)](#)

UUID

[Universally Unique Identifier \(汎用一意識別子\)](#)

WAL

[Write-Ahead Log \(ログ先行書き込み\)](#)

XID

[Transaction Identifier \(トランザクション識別子\)](#)

XML

[Extensible Markup Language \(拡張可能マークアップ言語\)](#)

[前のページ](#)
スタイルガイド

[ホーム](#)
[上に戻る](#)

[次のページ](#)
参考文献

参考文献

厳選されたSQLとPostgreSQLに関する参考文献と図書です。

元となったPOSTGRES開発チームの白書と技術レポートが、カリフォルニア大学バークレイ校コンピュータサイエンス学部の[Webサイト](#)にあります。

SQL参考図書

Judith Bowman, Sandra Emerson, 、 Marcy Darnovsky, *The Practical SQL Handbook: Using SQL Variants*, Fourth Edition, Addison-Wesley Professional, ISBN 0-201-70309-2, 2001.

C. J. Date 、 Hugh Darwen, *A Guide to the SQL Standard: A user's guide to the standard database language SQL*, Fourth Edition, Addison-Wesley, ISBN 0-201-96426-0, 1997.

C. J. Date, *An Introduction to Database Systems*, Eighth Edition, Addison-Wesley, ISBN 0-321-19784-4, 2003.

Ramez Elmasri 、 Shamkant Navathe, *Fundamentals of Database Systems*, Fourth Edition, Addison-Wesley, ISBN 0-321-12226-7, 2003.

Jim Melton 、 Alan R. Simon, *Understanding the New SQL: A complete guide*, Morgan Kaufmann, ISBN 1-55860-245-3, 1993.

Jeffrey D. Ullman, *Principles of Database and Knowledge: Base Systems*, Volume 1, Computer Science Press, 1988.

PostgreSQLに特化した文書

Stefan Simkovics, *Enhancement of the ANSI SQL Implementation of PostgreSQL*, Department of Information Systems, Vienna University of Technology, November 29, 1998.

SQLの歴史と構文法論文およびPostgreSQLへのINTERSECTとEXCEPTの追加とその説明。ウィーン工科大学O. Univ. Prof. Dr. Georg GottlobおよびUniv. Ass. Mag. Katrin Seyrの援助の下、修士論文として準備されたもの。

A. Yu 、 J. Chen, The POSTGRES Group, *The Postgres95 User Manual*, University of California, Sept. 5, 1995.

Zelaine Fong, [The design and implementation of the POSTGRES query optimizer](#), University of California, Berkeley, Computer Science Department.

会報ならびに記事

Nels Olson, *Partial indexing in POSTGRES: research project*, University of California, UCB Engin T7.49.1993 O676, 1993.

L. Ong 、 J. Goh, "A Unified Framework for Version Modeling Using Production Rules in a Database System", *ERL Technical Memorandum M90/33*, University of California, April, 1990.

L. Rowe 、 M. Stonebraker, "[The POSTGRES data model](#)", Proc. VLDB Conference, Sept. 1987.

P. Seshadri 、 A. Swami, "Generalized Partial Indexes ([キャッシュされたバージョン](#))", Proc. Eleventh International Conference on Data Engineering, 6-10 March 1995, IEEE Computer Society Press, Cat.

No.95CH35724, 1995, 420-7.

M. Stonebraker 、 L. Rowe, "[The design of POSTGRES](#) ", Proc. ACM-SIGMOD Conference on Management of Data, May 1986.

M. Stonebraker, E. Hanson, 、 C. H. Hong, "The design of the POSTGRES rules system", Proc. IEEE Conference on Data Engineering, Feb. 1987.

M. Stonebraker, "[The design of the POSTGRES storage system](#) ", Proc. VLDB Conference, Sept. 1987.

M. Stonebraker, M. Hearst, 、 S. Potamianos, "[A commentary on the POSTGRES rules system](#) ", *SIGMOD Record* 18(3), Sept. 1989.

M. Stonebraker, "[The case for partial indexes](#) ", *SIGMOD Record* 18(4), Dec. 1989, 4-11.

M. Stonebraker, L. A. Rowe, 、 M. Hirohama, "[The implementation of POSTGRES](#) ", *Transactions on Knowledge and Data Engineering* 2(1), IEEE, March 1990.

M. Stonebraker, A. Jhingran, J. Goh, 、 S. Potamianos, "[On Rules, Procedures, Caching and Views in Database Systems](#) ", Proc. ACM-SIGMOD Conference on Management of Data, June 1990.

[前のページ](#)
頭字語

[ホーム](#)

[次のページ](#)
目次

目次

[Symbols](#) | [A](#) | [B](#) | [C](#) | [D](#) | [E](#) | [F](#) | [G](#) | [H](#) | [I](#) | [J](#) | [K](#) | [L](#) | [M](#) | [N](#) | [O](#) | [P](#) | [Q](#) | [R](#) | [S](#) | [T](#) | [U](#) | [V](#) | [W](#) | [X](#) | [Y](#) | [Z](#)

Symbols

\$, [位置パラメータ](#)
\$libdir, [動的ロード](#)
\$libdir/plugins, [その他のデフォルト](#), [説明](#)
*, [選択リスト項目](#)
.pgpass, [パスワードファイル](#)
.pg_service.conf, [接続サービスファイル](#)
10進数

参照 [numeric](#)

::, [型キャスト](#)
_PG_fini, [動的ロード](#)
_PG_init, [動的ロード](#)
アップグレード処理, [PostgreSQLクラスタのアップグレード処理](#)
インストール, [ソースコードからインストール](#)

Windowsにおける, [Windowsにおけるソースコードからのインストール](#)

インタフェース

外部管理の, [クライアントインタフェース](#)

インデックス, [インデックス](#)

B-tree, [インデックスの種類](#)

式に対する, [式に対するインデックス](#)

GIN, [インデックスの種類](#), [GINインデックス](#)

GiST, [インデックスの種類](#), [GiSTインデックス](#)

SP-GiST, [インデックスの種類](#), [SP-GiSTインデックス](#)

とORDER BY, [インデックスとORDER BY](#)

の使用状況の検証, [インデックス使用状況の検証](#)

ハッシュ, [インデックスの種類](#)

ユーザ定義データ型用の, [インデックス拡張機能へのインタフェース](#)

ロック, [ロックとインデックス](#)

一意, [一意インデックス](#)

同時作成, [インデックスの同時作成](#)

部分, [部分インデックス](#)

複数のインデックスの組み合わせ, [複数のインデックスの組み合わせ](#)

複数列, [複数列インデックス](#)

インデックス再作成, [定常的なインデックスの再作成](#)
インデックス走査, [プランナメソッド設定](#)
ウィンドウ関数, [ウィンドウ関数](#)

起動, [ウィンドウ関数呼び出し](#)

実行順, [ウィンドウ関数処理](#)

組み込み, [ウィンドウ関数](#)

ウォームスタンバイ, [高可用性、負荷分散およびレプリケーション](#)
エスケープ文字列構文, [C形式エスケープでの文字列定数](#)
エラーコード

libpq, [主要な関数](#)

の一覧, [PostgreSQLエラーコード](#)

エラーメッセージ, [接続状態関数](#)
エラー報告

PL/pgSQLにおける, [エラーとメッセージ](#)

オーバーロード

演算子, [ユーザ定義の演算子](#)

関数, [関数のオーバーロード](#)

オブジェクト指向データベース, [概念](#)
オブジェクト識別子

データ型, [オブジェクト識別子データ型](#)

カーソル

PL/pgSQLにおける, [カーソル](#)

の問い合わせ計画の表示, [EXPLAIN](#)

カスケードレプリケーション, [高可用性、負荷分散およびレプリケーション](#)
キーワード

の一覧, [SQLキーワード](#)

の構文, [識別子とキーワード](#)

キャスト

入出力変換, [CREATE CAST](#)

キャンセル

SQLコマンドの, [処理中の問い合わせのキャンセル](#)

クライアント認証, [クライアント認証](#)
クラスタ

データベースの

参照 [データベースクラスタ](#)

クロスコンパイル, [インストール手順](#)
クロス結合, [結合テーブル](#)
グループ化, [GROUP BYとHAVING句](#)
グレゴリオ暦, [単位の歴史](#)
グローバルデータ

PL/Pythonにおける, [データの共有](#)

PL/Tclにおける, [PL/Tclにおけるグローバルデータ](#)

コメント

データベースオブジェクトについて, [システム情報関数](#)

SQL内の、[コメント](#)

コンパイル

libpq アプリケーション、[libpqプログラムの構築](#)

サーバのなりすまし、[サーバのなりすまし防止](#)

サーバログ、[エラー報告とログ取得](#)

ログファイルの保守、[ログファイルの保守](#)

シーケンシャル走査、[プランナメソッド設定](#)

シーケンス

連番型、[連番型](#)

シグナル

バックエンドプロセス、[サーバシグナル送信関数](#)

システムカタログ

スキーマ、[システムカタログスキーマ](#)

シャットダウン、[サーバのシャットダウン](#)

シリアライザブル、[シリアライザブル分離レベル](#)

シリアライザブルスナップショット分離、[序文](#)

スーパーユーザ、[データベースへのアクセス](#)、[ロールの属性](#)

スカラ

参照 式

スキーマ、[スキーマ](#)、[概要](#)

public、[publicスキーマ](#)

の作成、[スキーマの作成](#)

の削除、[スキーマの作成](#)

現在の、[スキーマ検索パス](#)、[システム情報関数](#)

スタンバイサーバ、[高可用性、負荷分散およびレプリケーション](#)

ストリーミングレプリケーション、[高可用性、負荷分散およびレプリケーション](#)

スライスパン

参照 TOAST

スレッド

libpqにおける、[スレッド化プログラムの振舞い](#)

セーブポイント

の解放、[RELEASE SAVEPOINT](#)

の定義、[SAVEPOINT](#)

ロールバック、[ROLLBACK TO SAVEPOINT](#)

セマフォ、[共有メモリとセマフォ](#)

タイムアウト

デッドロック、[ロック管理](#)

タイムライン、[バックアップとリストア](#)

ダーティリード、[トランザクションの分離](#)

チェックポイント、[WALの設定](#)

テーブル、[概念](#)、[テーブルの基本](#)

の作成, [テーブルの基本](#)

の削除, [テーブルの基本](#)

の変更, [テーブルの変更](#)

の名称変更, [テーブル名の変更](#)

パーティショニング, [パーティショニング](#)

継承, [継承](#)

テーブル関数, [テーブル関数](#)

テーブル空間, [テーブル空間](#)

デフォルト, [文の動作](#)

一時的, [文の動作](#)

テーブル式, [テーブル式](#)

テキスト検索

データ型, [テキスト検索に関する型](#)

関数と演算子, [テキスト検索に関する型](#)

テスト, [リグレッションテスト](#)

データベース, [データベース管理](#)

を作成する権限, [ロールの属性](#)

作成, [データベースの作成](#)

データベースクラスタ, [概念](#), [データベースクラスタの作成](#)

データベース活動状況

の監視, [データベース活動状況の監視](#)

データ型, [データ型](#)

カテゴリ, [概要](#)

ユーザ定義の, [ユーザ定義の型](#)

基本, [PostgreSQLの型システム](#)

型キャスト, [型キャスト](#)

数値, [数値データ型](#)

定数, [他の型の定数](#)

内部構成, [C言語関数における基本型](#)

複合, [PostgreSQLの型システム](#)

変換, [型変換](#)

列挙 (enum) , [列挙型](#)

データ領域

参照 データベースクラスタ

ディスクドライブ, [WALの内部](#)

ディスク使用量, [ディスク使用量の決定](#)

ディスク容量, [ディスク容量の復旧](#)

デッドロック, [デッドロック](#)

間のタイムアウト, [ロック管理](#)

デフォルト値, [デフォルト値](#)

の変更, [列のデフォルト値の変更](#)

トークン, [語彙の構成](#)

トランザクション, [トランザクション](#)

トランザクションID

周回, [トランザクションIDの周回エラーの防止](#)

トランザクションの分離, [トランザクションの分離](#)

トランザクションログ

参照 WAL

トランザクション隔離レベル

の設定, [SET TRANSACTION](#)

デフォルト設定, [文の動作](#)

トランザクション分離レベル, [トランザクションの分離](#)

シリアライザブル, [シリアライザブル分離レベル](#)

リードコミティド, [リードコミティド分離レベル](#)

リピータブルリード, [リピータブルリード分離レベル](#)

トリガ, [トリガ](#)

Cによる, [Cによるトリガ関数の作成](#)

PL/pgSQLにおける, [トリガプロシージャ](#)

PL/Pythonにおける, [トリガ関数](#)

PL/Tclにおける, [PL/Tclのトリガプロシージャ](#)

とルールとの比較, [ルール対トリガ](#)

tsvectorから派生した列を更新する, [自動更新のためのトリガ](#)

トリガ関数の引数, [トリガ動作の概要](#)

ドル引用符付け, [ドル記号で引用符付けされた文字列定数](#)

ネットワーク

データ型, [ネットワークアドレス型](#)

ネットワークアタッチトストレージ (NAS)

参照 ネットワークファイルシステム

ネットワークファイルシステム, [ネットワークファイルシステム](#)

ハッシュ

参照 インデックス

バージョン

互換性, [PostgreSQLクラスタのアップグレード処理](#)

バイナリデータ, [バイナリ列データ型](#), [バイナリ文字列関数と演算子](#)

バイナリ文字列

結合, [バイナリ文字列関数と演算子](#)

長さ, [バイナリ文字列関数と演算子](#)

バキューム, [定常的なバキューム作業](#)

バックアップ, [バックアップ制御関数](#), [バックアップとリストア](#)
バックスラッシュエスケープ, [C形式エスケープでの文字列定数](#)
パーティショニング, [パーティショニング](#)
パス

スキーマへの, [文の動作](#)

パスワード, [ロールの属性](#)

スーパーユーザの, [データベースクラスタの作成](#)

認証, [パスワード認証](#)

パスワードファイル, [パスワードファイル](#)
パターン

psqlおよびpg_dumpにおける, [パターン](#)

パターンマッチ, [パターンマッチ](#)
パラメータ

の構文, [位置パラメータ](#)

ビットマップスキャン, [複数のインデックスの組み合わせ](#)
ビットマップ走査, [プランナメソッド設定](#)
ビット文字列

定数, [ビット文字列定数](#)

ビット列

データ型, [ビット列データ型](#)

ビュー, [ビュー](#)

ルールを使用した実装, [ビューとルールシステム](#)

更新, [ビューとの協調](#)

ファントムリード, [トランザクションの分離](#)
フィールド

計算処理された, [複合型を使用するSQL関数](#)

フィールド選択, [フィールド選択](#)
プライマリキー, [プライマリキー](#)
プリペアド文

の作成, [PREPARE](#)

プロトコル

フロントエンド/バックエンド, [フロントエンド/バックエンドプロトコル](#)

ホスト名, [パラメータキーワード](#)
ホットスタンバイ, [高可用性、負荷分散およびレプリケーション](#)
ポイントインタイムリカバリ, [バックアップとリストア](#)
マシンの起動

時のサーバ起動, [データベースサーバの起動](#)

メモリコンテキスト

SPI内部の, [メモリ管理](#)

ユーザ, [データベースロール](#)

現在の, [システム情報関数](#)

ユーザマッピング, [外部データ](#)

ユーザ名マップ, [ユーザ名マップ](#)
ユリウス日, [単位の歴史](#)
ラージオブジェクト, [ラージオブジェクト](#)
ライブラリ最終処理関数, [動的ロード](#)
ライブラリ初期化処理関数, [動的ロード](#)
ラベル

参照 別名

リードコミットド, [リードコミットド分離レベル](#)
リグレーションテスト, [インストール手順](#), [リグレーションテスト](#)
リピータブルリード, [リピータブルリード分離レベル](#)
リレーショナルデータベース, [概念](#)
リレーション, [概念](#)
ループ

PL/pgSQLにおける, [単純なループ](#)

ルール, [ルールシステム](#)

DELETE用, [INSERT、UPDATE、DELETEについてのルール](#)

INSERT用, [INSERT、UPDATE、DELETEについてのルール](#)

SELECT用, [SELECTルールの動き](#)

とトリガとの比較, [ルール対トリガ](#)

UPDATE用, [INSERT、UPDATE、DELETEについてのルール](#)

とビュー, [ビューとルールシステム](#)

ロール, [データベースロール](#)

を作成する権限, [ロールの属性](#)

レプリケーションの新規接続を行う権限, [ロールの属性](#)

適用可能, [applicable_roles](#)

内のメンバ資格, [ロールのメンバ資格](#)

ログの出力先, [ログの出力先](#)
ログイン権限, [ロールの属性](#)
ログ SHIPPING, [高可用性、負荷分散およびレプリケーション](#)
ロケール, [データベースクラスタの作成](#), [ロケールのサポート](#)
ロック, [明示的ロック](#)

勧告的, [勧告的ロック](#)

監視, [ロックの表示](#)

暗号化, [暗号化オプション](#)

特定の列の, [pgcrypto](#)

遺伝的問い合わせ最適化, [遺伝的問い合わせオブティマイザ](#)
一意性制約, [一意性制約](#)
引用符

および識別子, [識別子とキーワード](#)

エスケープ, [文字列定数](#)

右結合, [結合テーブル](#)
演算子, [関数と演算子](#)

の呼び出しにおける型の解決, [演算子](#)

ユーザ定義, [ユーザ定義の演算子](#)

呼び出し, [演算子の呼び出し](#)

構文, [演算子](#)

優先順位, [演算子の優先順位](#)

論理, [論理演算子](#)

演算子クラス, [演算子クラスと演算子族](#), [インデックスメソッドと演算子クラス](#)

演算子族, [演算子クラスと演算子族](#), [演算子クラスと演算子族](#)

可視性マップ, [可視性マップ](#)

回帰勾配, [集約関数](#)

回帰切片, [集約関数](#)

階層型データベース, [概念](#)

外部キー, [外部キー](#), [外部キー](#)

外部テーブル, [外部データ](#)

外部データ, [外部データ](#)

外部データラッパ

のハンドラ, [外部データラッパの作成](#)

外部結合, [結合テーブル](#)

拡張, [関連するオブジェクトを拡張としてパッケージ化](#)

外部で保守される, [拡張](#)

格納パラメータ, [格納パラメータ](#)

括弧で囲まれた, [評価式](#)

勧告的ロック, [勧告的ロック](#)

環境変数, [環境変数](#)

監視

データベース活動情報の, [データベース活動状況の監視](#)

管理ツール

外部で保守される, [管理ツール](#)

関数, [関数と演算子](#)

FROM句内の, [テーブル関数](#)

RETURNS TABLE, [TABLEを返すSQL関数](#)

の呼び出しにおける型解決, [関数](#)

ユーザ定義, [ユーザ定義関数](#)

Cで作成された, [C言語関数](#)

SQLで作成した, [問い合わせ言語 \(SQL\) 関数](#)

位置表記, [位置表記の使用](#)

呼び出し, [関数呼び出し](#)

混在表記, [混在表記の利用](#)

出力パラメータ, [出力パラメータを持つSQL関数](#)

多様, [多様型](#)

内部, [内部関数](#)

名前付き引数, [SQL関数用の引数](#)

名前付け表記, [名前付け表記の使用](#)

関数依存, [GROUP BYとHAVING句](#)

基本型, [PostgreSQLの型システム](#)

偽, [論理値データ型](#)

休止, [遅延実行](#)

共通テーブル式

参照 [WITH](#)

共分散

標本, [集約関数](#)

母集団, [集約関数](#)

共有メモリ, [共有メモリとセマフォ](#)

共有ライブラリ, [共有ライブラリ](#), [動的にロードされる関数のコンパイルとリンク](#)

近道, [近道インタフェース](#)

矩形, [矩形](#)

空き領域マップ, [空き領域マップ](#)

繰り延べトランザクション

デフォルト設定, [文の動作](#)

型

参照 [データ型](#)

多様, [多様型](#)

型キャスト, [数値定数](#), [型キャスト](#)

継承, [継承](#), [以前のPostgreSQLバージョン](#)

継続的アーカイブ, [バックアップとリストア](#)

計算処理されたフィールド, [複合型を使用するSQL関数](#)

警告処理

libpqでの, [警告処理](#)

結合, [テーブル間を結合](#), [結合テーブル](#)

natural, [結合テーブル](#)

クロス, [結合テーブル](#)

右, [結合テーブル](#)

外部, [テーブル間を結合](#), [結合テーブル](#)

左, [結合テーブル](#)

自己, [テーブル間を結合](#)

順番を制御する, [明示的なJOIN句でプランナを制御する](#)

検査制約, [検査制約](#)

検索パス, [スキーマ検索パス](#)

現在の, [システム情報関数](#)

権限, [権限](#)

ルールでの, [ルールと権限](#)

スキーマ用の, [スキーマおよび権限](#)

ビューでの, [ルールと権限](#)

問い合わせ, [システム情報関数](#)

権利

参照 [権限](#)

更新, [データの更新](#)
構文

SQL, [SQLの構文](#)

行, [概念](#), [テーブルの基本](#)
行に関する比較, [行と配列の比較](#)
行型, [複合型](#)

のコンストラクタ, [行コンストラクタ](#)

行推定

プランナ, [行推定の例](#)

左結合, [結合テーブル](#)
削除, [データの削除](#)
参照整合性, [外部キー](#), [外部キー](#)
時間間隔, [日付/時刻データ型](#), [時間間隔入力](#)

出力書式, [時間間隔出力](#)

参照 書式設定

時間帯, [時間帯](#), [ロケールと書式設定](#)

入力簡略形, [日付/時刻設定ファイル](#)

変換, [AT TIME ZONE](#)

時間帯データ, [インストール手順](#)
時間帯名, [ロケールと書式設定](#)
自動コミット

大量のデータロード, [自動コミットをオフにする](#)

自動増分

参照 serial

式

の構文, [評価式](#)

の評価順, [式の評価規則](#)

識別子

の構文, [識別子とキーワード](#)

長さ, [識別子とキーワード](#)

手続き言語, [手続き言語](#)

の作成, [手続き言語ハンドラの作成](#)

外部管理の, [手続き言語](#)

修飾名, [スキーマの作成](#)
集合を返す関数

関数, [集合を返す関数](#)

集合差, [問い合わせの結合](#)
集合積, [問い合わせの結合](#)
集合操作, [問い合わせの結合](#)
集合和, [問い合わせの結合](#)
集約関数, [集約関数](#)

built-in, [集約関数](#)

ユーザ定義, [ユーザ定義の集約](#)

呼び出し, [集約式](#)

重複, [テーブルへの問い合わせ](#), [DISTINCT](#)

出力関数, [ユーザ定義の型](#)

述部ロック, [シリアライズabl分離レベル](#)

準備された文

の削除, [DEALLOCATE](#)

の実行, [EXECUTE](#)

の問い合わせ計画の表示, [EXPLAIN](#)

順序付け演算子, [システムの演算子クラスに対する依存性](#)

初期化フォーク, [初期化フォーク](#)

所有者, [権限](#)

書式設定, [データ型書式設定関数](#)

照合順序

SQL関数における, [照合順序を持つSQL関数](#)

証明サーバ, [高可用性、負荷分散およびレプリケーション](#)

情報スキーマ, [情報スキーマ](#)

条件式, [条件式](#)

信頼

PL/Perl, [信頼されたPL/Perlおよび信頼されないPL/Perl](#)

真, [論理値データ型](#)

数値

定数, [数値定数](#)

制御ファイル, [拡張のファイル](#)

制約, [制約](#)

の削除, [制約の削除](#)

の追加, [制約の追加](#)

の名前, [検査制約](#)

プライマリキー, [プライマリキー](#)

一意性, [一意性制約](#)

外部キー, [外部キー](#)

検査, [検査制約](#)

排他, [排他制約](#)

非NULL, [非NULL制約](#)

制約による除外, [パーティショニングと制約による除外](#)

制約除外, [その他のプランナオプション](#)

性能, [性能に関するヒント](#)

正規表現, [POSIX正規表現](#)

参照 パターンマッチ

とロケール, [動作](#)

接続サービスファイル, [接続サービスファイル](#)

設定

サーバの

関数, [構成設定関数](#)

サーバの, [サーバの設定](#)

リカバリの

スタンバイサーバの, [リカバリの設定](#)

線形リグレーション, [集約関数](#)

線分, [線分](#)

全文テキスト検索

関数と演算子, [テキスト検索に関する型](#)

全文検索

データ型, [テキスト検索に関する型](#)

挿入, [データの挿入](#)

相関, [集約関数](#)

多版型同時実行制御, [序文](#)

多様関数, [多様型](#)

多様型, [多様型](#)

大文字小文字の区別

SQLコマンドの, [識別子とキーワード](#)

遅延, [遅延実行](#)

遅延トランザクション

の設定, [SET TRANSACTION](#)

長さ

バイナリ文字列の

参照 バイナリ文字列、長さ

文字列の

参照 文字列、長さ

直列化異常, [シリアライザブル分離レベル](#)

通知プロセッサ, [警告処理](#)

通知レシーバ, [警告処理](#)

定常的な保守, [定常的なデータベース保守作業](#)

定数, [定数](#)

適用可能なロール, [applicable_roles](#)

添字, [添字](#)

統計処理, [集約関数](#)

統計情報, [統計情報コレクタ](#)

プランナの, [プランナで使用する統計情報](#), [プランナ用の統計情報の更新](#)

動的ロード, [その他のデフォルト](#), [動的ロード](#)

同期コミット, [非同期コミット](#)

同期レプリケーション, [高可用性、負荷分散およびレプリケーション](#)

同時実行制御, [同時実行制御](#)

読み取りのみトランザクション

の設定, [SET TRANSACTION](#)

読み取り専用トランザクション

デフォルト設定, [文の動作](#)

日付

現在, [Current Date/Time](#)

入力関数, [ユーザ定義の型](#)
任意の精度数, [任意の精度を持つ数](#)
排他制約, [排他制約](#)
配列, [配列](#)

I/O, [配列の入出力構文](#)

アクセス, [配列へのアクセス](#)

コンストラクタ, [配列コンストラクタ](#)

ユーザ定義型の, [ユーザ定義の型](#)

検索, [配列内の検索](#)

宣言, [配列型の宣言](#)

定数, [配列の値の入力](#)

変更, [配列の変更](#)

反復不能読み取り, [トランザクションの分離](#)
範囲テーブル, [問い合わせツリーとは](#)
範囲型, [範囲型](#)

のインデックス, [インデックス](#)

排他, [範囲の制約](#)

否定, [論理演算子](#)
比較

演算子, [比較演算子](#)

行に関する, [行と配列の比較](#)

副問い合わせ結果行, [副問い合わせ式](#)

非NULL制約, [非NULL制約](#)
非ブロック接続, [非同期コマンドの処理](#)
非ブロック接続, [データベース接続制御関数](#)
非永続性, [永続性がない設定](#)
非修飾名, [スキーマ検索パス](#)
非数

数値（データ型）, [任意の精度を持つ数](#)

倍精度, [浮動小数点データ型](#)

非同期コミット, [非同期コミット](#)
標準偏差

標本, [集約関数](#)

母集団, [集約関数](#)

表記

関数, [関数呼び出し](#)

評価式, [評価式](#)
浮動小数点, [浮動小数点データ型](#)

表示, [ロケールと書式設定](#)

副問い合わせ, [集約関数](#), [スカラ副問い合わせ](#), [副問い合わせ](#)
複合型, [複合型](#), [PostgreSQLの型システム](#)

のコンストラクタ, [行コンストラクタ](#)

の定数, [複合型の値の入力](#)

分散, [集約関数](#)

標本, [集約関数](#)

母集団, [集約関数](#)

文字の並び

データ型, [文字型](#)

文字セット, [ロケールと書式設定](#), [設定済みのオプション](#), [文字セットサポート](#)
文字列

参照 [文字の並び](#)
エスケープに関する警告, [以前のPostgreSQLバージョン](#)
バックスラッシュによる引用, [以前のPostgreSQLバージョン](#)
結合, [文字列関数と演算子](#)
長さ, [文字列関数と演算子](#)
定数, [文字列定数](#)
標準に従う, [以前のPostgreSQLバージョン](#)

文字列のエスケープ

libpqにおける, [SQLコマンドに含めるための文字列のエスケープ処理](#)

並び替え, [行の並び替え](#)
別名

FROM句内の, [テーブルと列の別名](#)

選択リスト内の, [列ラベル](#)

問い合わせ内のテーブル名用の, [テーブル間を結合](#)

変動性

関数, [関数の変動性分類](#)

保守, [定常的なデータベース保守作業](#)
魔法のブロック, [動的ロード](#)
埋め込みSQL

C言語による, [ECPG - C言語による埋め込みSQL](#)

無名コードブロック, [DO](#)
名前

の構文, [識別子とキーワード](#)

修飾された, [スキーマの作成](#)

非修飾の, [スキーマ検索パス](#)

目的リスト, [問い合わせツリーとは](#)
問い合わせ, [テーブルへの問い合わせ](#), [問い合わせ](#)
問い合わせの準備

PL/pgSQL における, [計画のキャッシュ](#)

PL/Pythonにおける, [データベースアクセス関数](#)

PL/Tclにおける, [PL/Tclからのデータベースアクセス](#)

問い合わせツリー, [問い合わせツリーとは](#)
問い合わせ計画, [EXPLAINの利用](#)
有効数字, [ロケールと書式設定](#)
歴史

PostgreSQLの, [PostgreSQL小史](#)

列, [概念](#), [テーブルの基本](#)

の削除, [列の削除](#)

の追加, [列の追加](#)

の名称変更, [列名の変更](#)

システム列, [システム列](#)

列のデータ型

の変更, [列のデータ型の変更](#)

列の参照, [列の参照](#)

列挙型, [列挙型](#)

論理積, [論理演算子](#)

論理値

演算子

参照 [演算子](#)、[論理](#)

論理和, [論理演算子](#)

A

abbrev, [ネットワークアドレス関数と演算子](#)

ABORT, [ABORT](#)

abs, [算術関数と演算子](#)

acos, [算術関数と演算子](#)

adminpack, [adminpack](#)

age, [日付/時刻関数と演算子](#)

AIX

IPCの設定, [共有メモリとセマフォ](#)

上へのインストール, [AIX](#)

ALL, [副問い合わせ式](#), [行と配列の比較](#)

allow_system_table_mods設定パラメータ, [開発者向けのオプション](#)

ALTER AGGREGATE, [ALTER AGGREGATE](#)

ALTER COLLATION, [ALTER COLLATION](#)

ALTER CONVERSION, [ALTER CONVERSION](#)

ALTER DATABASE, [ALTER DATABASE](#)

ALTER DEFAULT PRIVILEGES, [ALTER DEFAULT PRIVILEGES](#)

ALTER DOMAIN, [ALTER DOMAIN](#)

ALTER EXTENSION, [ALTER EXTENSION](#)

ALTER FOREIGN DATA WRAPPER, [ALTER FOREIGN DATA WRAPPER](#)

ALTER FOREIGN TABLE, [ALTER FOREIGN TABLE](#)

ALTER FUNCTION, [ALTER FUNCTION](#)

ALTER GROUP, [ALTER GROUP](#)

ALTER INDEX, [ALTER INDEX](#)

ALTER LANGUAGE, [ALTER LANGUAGE](#)

ALTER LARGE OBJECT, [ALTER LARGE OBJECT](#)

ALTER OPERATOR, [ALTER OPERATOR](#)

ALTER OPERATOR CLASS, [ALTER OPERATOR CLASS](#)

ALTER OPERATOR FAMILY, [ALTER OPERATOR FAMILY](#)

ALTER ROLE, [ロールの属性](#), [ALTER ROLE](#)

ALTER SCHEMA, [ALTER SCHEMA](#)

ALTER SEQUENCE, [ALTER SEQUENCE](#)

ALTER SERVER, [ALTER SERVER](#)

ALTER TABLE, [ALTER TABLE](#)

ALTER TABLESPACE, [ALTER TABLESPACE](#)

ALTER TEXT SEARCH CONFIGURATION, [ALTER TEXT SEARCH CONFIGURATION](#)

ALTER TEXT SEARCH DICTIONARY, [ALTER TEXT SEARCH DICTIONARY](#)
ALTER TEXT SEARCH PARSER, [ALTER TEXT SEARCH PARSER](#)
ALTER TEXT SEARCH TEMPLATE, [ALTER TEXT SEARCH TEMPLATE](#)
ALTER TRIGGER, [ALTER TRIGGER](#)
ALTER TYPE, [ALTER TYPE](#)
ALTER USER, [ALTER USER](#)
ALTER USER MAPPING, [ALTER USER MAPPING](#)
ALTER VIEW, [ALTER VIEW](#)
ANALYZE, [プランナ用の統計情報の更新](#), [ANALYZE](#)
AND (演算子), [論理演算子](#)
any, [疑似データ型](#), [集約関数](#), [副問い合わせ式](#), [行と配列の比較](#)
anyarray, [疑似データ型](#)
anyelement, [疑似データ型](#)
anyenum, [疑似データ型](#)
anynonarray, [疑似データ型](#)
anyrange, [疑似データ型](#)
application_name設定パラメータ, [何をログに](#)
archive_cleanup_commandリカバリ用パラメータ, [アーカイブリカバリの設定](#)
archive_command設定パラメータ, [アーカイビング](#)
archive_mode設定パラメータ, [アーカイビング](#)
archive_timeout設定パラメータ, [アーカイビング](#)
area, [幾何関数と演算子](#)
ARRAY, [配列コンストラクタ](#)

結果型の決定, [UNION、CASEおよび関連する構文](#)

array_agg, [集約関数](#)
array_append, [配列関数と演算子](#)
array_cat, [配列関数と演算子](#)
array_dims, [配列関数と演算子](#)
array_fill, [配列関数と演算子](#)
array_length, [配列関数と演算子](#)
array_lower, [配列関数と演算子](#)
array_ndims, [配列関数と演算子](#)
array_nulls設定パラメータ, [以前のPostgreSQLバージョン](#)
array_prepend, [配列関数と演算子](#)
array_to_json, [JSON関数](#)
array_to_string, [配列関数と演算子](#)
array_upper, [配列関数と演算子](#)
ascii, [文字列関数と演算子](#)
asin, [算術関数と演算子](#)
AT TIME ZONE, [AT TIME ZONE](#)
atan, [算術関数と演算子](#)
atan2, [算術関数と演算子](#)
authentication_timeout設定パラメータ, [セキュリティと認証](#)
auth_delay, [auth_delay](#)
auth_delay.milliseconds設定パラメータ, [設定パラメータ](#)
autocommit

psql, [変数](#)

autovacuum

一般情報, [自動バキュームデーモン](#)

設定パラメータ, [自動Vacuum作業](#)

autovacuum_analyze_scale_factor設定パラメータ, [自動Vacuum作業](#)
autovacuum_analyze_threshold設定パラメータ, [自動Vacuum作業](#)
autovacuum_freeze_max_age設定パラメータ, [自動Vacuum作業](#)
autovacuum_max_workers設定パラメータ, [自動Vacuum作業](#)
autovacuum_naptime設定パラメータ, [自動Vacuum作業](#)
autovacuum_vacuum_cost_delay設定パラメータ, [自動Vacuum作業](#)
autovacuum_vacuum_cost_limit設定パラメータ, [自動Vacuum作業](#)
autovacuum_vacuum_scale_factor設定パラメータ, [自動Vacuum作業](#)
autovacuum_vacuum_threshold設定パラメータ, [自動Vacuum作業](#)

autovacuum設定パラメータ, [自動Vacuum作業](#)
auto_explain, [auto_explain](#)
auto_explain.log_analyze設定パラメータ, [設定パラメータ](#)
auto_explain.log_buffers設定パラメータ, [設定パラメータ](#)
auto_explain.log_format設定パラメータ, [設定パラメータ](#)
auto_explain.log_min_duration構成パラメータ, [設定パラメータ](#)
auto_explain.log_nested_statements設定パラメータ, [設定パラメータ](#)
auto_explain.log_verbose設定パラメータ, [設定パラメータ](#)
average, [集約関数](#)
avg, [集約関数](#)

B

B-tree

参照 [インデックス](#)

backslash_quote設定パラメータ, [以前のPostgreSQLバージョン](#)
BEGIN, [BEGIN](#)
BETWEEN, [比較演算子](#)
BETWEEN SYMMETRIC, [比較演算子](#)
bgwriter_delay設定パラメータ, [バックグラウンドライター](#)
bgwriter_lru_maxpages設定パラメータ, [バックグラウンドライター](#)
bgwriter_lru_multiplier設定パラメータ, [バックグラウンドライター](#)
bigint, [数値定数](#), [整数データ型](#)
bigserial, [連番型](#)
bison, [必要条件](#)
bit strings

functions, [ビット文字列関数と演算子](#)

bit_and, [集約関数](#)
bit_length, [文字列関数と演算子](#)
bit_or, [集約関数](#)
BLOB

参照 [ラージオブジェクト](#)

block_size設定パラメータ, [設定済みのオプション](#)
bonjour_name設定パラメータ, [接続設定](#)
bonjour設定パラメータ, [接続設定](#)
Boolean

データ型, [論理値データ型](#)

bool_and, [集約関数](#)
bool_or, [集約関数](#)
box, [幾何関数と演算子](#)
box (データ型), [矩形](#)
broadcast, [ネットワークアドレス関数と演算子](#)
btree_gin, [btree_gin](#)
btree_gist, [btree_gist](#)
btrim, [文字列関数と演算子](#), [バイナリ文字列関数と演算子](#)
bytea, [バイナリ列データ型](#)
bytea_output設定パラメータ, [文の動作](#)

C

C, [libpq - C ライブラリ](#), [ECPG - C言語による埋め込みSQL](#)
C++, [拡張へのC++の利用](#)
CASCADE

DROPの, [依存関係の追跡](#)

外部キー動作, [外部キー](#)

CASE, [条件式](#)

結果型の決定, [UNION、CASEおよび関連する構文](#)

cbrt, [算術関数と演算子](#)

ceil, [算術関数と演算子](#)

ceiling, [算術関数と演算子](#)

center, [幾何関数と演算子](#)

Certificate, [証明書認証](#)

char, [文字型](#)

character, [文字型](#)

character varying, [文字型](#)

char_length, [文字列関数と演算子](#)

CHECKPOINT, [CHECKPOINT](#)

checkpoint_completion_target設定パラメータ, [チェックポイント](#)

checkpoint_segments設定パラメータ, [チェックポイント](#)

checkpoint_timeout設定パラメータ, [チェックポイント](#)

checkpoint_warning設定パラメータ, [チェックポイント](#)

check_function_bodies設定パラメータ, [文の動作](#)

chkpass, [chkpass](#)

chr, [文字列関数と演算子](#)

cid, [オブジェクト識別子データ型](#)

cidr, [cidr](#)

circle, [円](#), [幾何関数と演算子](#)

citext, [citext](#)

client authentication

タイムアウト期間, [セキュリティと認証](#)

client_encoding設定パラメータ, [ロケールと書式設定](#)

client_min_messages設定パラメータ, [いつログを取得するか](#)

clock_timestamp, [日付/時刻関数と演算子](#)

CLOSE, [CLOSE](#)

CLUSTER, [CLUSTER](#)

clusterdb, [clusterdb](#)

clustering, [高可用性、負荷分散およびレプリケーション](#)

cmax, [システム列](#)

cmin, [システム列](#)

COALESCE, [COALESCE](#)

COLLATE, [照合順序式](#)

collation, [照合サポート](#)

PL/pgSQLにおける, [PL/pgSQL変数の照合](#)

collation for, [システム情報関数](#)

col_description, [システム情報関数](#)

COMMENT, [COMMENT](#)

COMMIT, [COMMIT](#)

COMMIT PREPARED, [COMMIT PREPARED](#)

commit_delay設定パラメータ, [諸設定](#)

commit_siblings設定パラメータ, [諸設定](#)

concat, [文字列関数と演算子](#)

concat_ws, [文字列関数と演算子](#)

configure, [インストール手順](#)

config_file設定パラメータ, [ファイルの場所](#)

conninfo, [接続文字列](#)

constraint_exclusion設定パラメータ, [その他のプランナオプション](#)

CONTINUE

in PL/pgSQL, [CONTINUE](#)

convert, [文字列関数と演算子](#)

convert_from, [文字列関数と演算子](#)

convert_to, [文字列関数と演算子](#)

COPY, [テーブルに行を挿入](#), [COPY](#)

libpqを使用した, [COPYコマンド関連関数](#)

corr, [集約関数](#)
cos, [算術関数と演算子](#)
cot, [算術関数と演算子](#)
count, [集約関数](#)
covar_pop, [集約関数](#)
covar_samp, [集約関数](#)
cpu_index_tuple_cost設定パラメータ, [プランナコスト定数](#)
cpu_operator_cost設定パラメータ, [プランナコスト定数](#)
cpu_tuple_cost設定パラメータ, [プランナコスト定数](#)
CREATE AGGREGATE, [CREATE AGGREGATE](#)
CREATE CAST, [CREATE CAST](#)
CREATE COLLATION, [CREATE COLLATION](#)
CREATE CONVERSION, [CREATE CONVERSION](#)
CREATE DATABASE, [データベースの作成](#), [CREATE DATABASE](#)
CREATE DOMAIN, [CREATE DOMAIN](#)
CREATE EXTENSION, [CREATE EXTENSION](#)
CREATE FOREIGN DATA WRAPPER, [CREATE FOREIGN DATA WRAPPER](#)
CREATE FOREIGN TABLE, [CREATE FOREIGN TABLE](#)
CREATE FUNCTION, [CREATE FUNCTION](#)
CREATE GROUP, [CREATE GROUP](#)
CREATE INDEX, [CREATE INDEX](#)
CREATE LANGUAGE, [CREATE LANGUAGE](#)
CREATE OPERATOR, [CREATE OPERATOR](#)
CREATE OPERATOR CLASS, [CREATE OPERATOR CLASS](#)
CREATE OPERATOR FAMILY, [CREATE OPERATOR FAMILY](#)
CREATE ROLE, [データベースロール](#), [CREATE ROLE](#)
CREATE RULE, [CREATE RULE](#)
CREATE SCHEMA, [CREATE SCHEMA](#)
CREATE SEQUENCE, [CREATE SEQUENCE](#)
CREATE SERVER, [CREATE SERVER](#)
CREATE TABLE, [新しいテーブルの作成](#), [CREATE TABLE](#)
CREATE TABLE AS, [CREATE TABLE AS](#)
CREATE TABLESPACE, [テーブル空間](#), [CREATE TABLESPACE](#)
CREATE TEXT SEARCH CONFIGURATION, [CREATE TEXT SEARCH CONFIGURATION](#)
CREATE TEXT SEARCH DICTIONARY, [CREATE TEXT SEARCH DICTIONARY](#)
CREATE TEXT SEARCH PARSER, [CREATE TEXT SEARCH PARSER](#)
CREATE TEXT SEARCH TEMPLATE, [CREATE TEXT SEARCH TEMPLATE](#)
CREATE TRIGGER, [CREATE TRIGGER](#)
CREATE TYPE, [CREATE TYPE](#)
CREATE USER, [CREATE USER](#)
CREATE USER MAPPING, [CREATE USER MAPPING](#)
CREATE VIEW, [CREATE VIEW](#)
createdb, [データベースの作成](#), [データベースの作成](#), [createdb](#)
createlang, [createlang](#)
createuser, [データベースロール](#), [createuser](#)
cstring, [疑似データ型](#)
ctid, [システム列](#), [非SELECT文のビュールール](#)
cube, [cube](#)
cume_dist, [ウィンドウ関数](#)
current_catalog, [システム情報関数](#)
current_database, [システム情報関数](#)
current_date, [日付/時刻関数と演算子](#)
current_query, [システム情報関数](#)
current_schema, [システム情報関数](#)
current_schemas, [システム情報関数](#)
current_setting, [構成設定関数](#)
current_time, [日付/時刻関数と演算子](#)
current_timestamp, [日付/時刻関数と演算子](#)
current_user, [システム情報関数](#)
currval, [シーケンス操作関数](#)
cursor

CLOSE, [CLOSE](#)

DECLARE, [DECLARE](#)

FETCH, [FETCH](#)

MOVE, [MOVE](#)

cursor_tuple_fraction設定パラメータ, [その他のプランナオプション](#)
Cygwin

上へのインストール, [Cygwin](#)

D

data partitioning, [高可用性、負荷分散およびレプリケーション](#)

data_directory設定パラメータ, [ファイルの場所](#)

date, [日付/時刻データ型](#), [日付](#)

出力書式, [日付/時刻出力](#)

参照 書式設定

定数, [特殊な値](#)

DateStyle設定パラメータ, [ロケールと書式設定](#)

date_part, [日付/時刻関数と演算子](#), [EXTRACT](#), [date_part](#)

date_trunc, [日付/時刻関数と演算子](#), [date_trunc](#)

dblink, [dblink](#)

db_user_namespace設定パラメータ, [セキュリティと認証](#)

deadlock_timeout設定パラメータ, [ロック管理](#)

DEALLOCATE, [DEALLOCATE](#)

debug_assertions設定パラメータ, [開発者向けのオプション](#)

debug_deadlocks設定パラメータ, [開発者向けのオプション](#)

debug_pretty_print設定パラメータ, [何をログに](#)

debug_print_parse設定パラメータ, [何をログに](#)

debug_print_plan設定パラメータ, [何をログに](#)

debug_print_rewritten設定パラメータ, [何をログに](#)

DECLARE, [DECLARE](#)

decode, [文字列関数と演算子](#), [バイナリ文字列関数と演算子](#)

decode_bytea

PL/Perlにおける, [PL/Perlのユーティリティ関数](#)

default_statistics_target設定パラメータ, [その他のプランナオプション](#)

default_tablespace設定パラメータ, [文の動作](#)

default_text_search_config設定パラメータ, [ロケールと書式設定](#)

default_transaction_deferrable 設定パラメータ, [文の動作](#)

default_transaction_isolation設定パラメータ, [文の動作](#)

default_transaction_read_only設定パラメータ, [文の動作](#)

default_with_oids設定パラメータ, [以前のPostgreSQLバージョン](#)

degrees, [算術関数と演算子](#)

DELETE, [削除](#), [データの削除](#), [DELETE](#)

dense_rank, [ウィンドウ関数](#)

diameter, [幾何関数と演算子](#)

dict_int, [dict_int](#)

dict_xsyn, [dict_xsyn](#)

Digital UNIX

参照 Tru64 UNIX

DISCARD, [DISCARD](#)

DISTINCT, [テーブルへの問い合わせ](#), [DISTINCT](#)

div, [算術関数と演算子](#)

DO, [DO](#)

document

text search, [文書とは何か?](#)

double precision, [浮動小数点データ型](#)
DROP AGGREGATE, [DROP AGGREGATE](#)
DROP CAST, [DROP CAST](#)
DROP COLLATION, [DROP COLLATION](#)
DROP CONVERSION, [DROP CONVERSION](#)
DROP DATABASE, [データベースの削除](#), [DROP DATABASE](#)
DROP DOMAIN, [DROP DOMAIN](#)
DROP EXTENSION, [DROP EXTENSION](#)
DROP FOREIGN DATA WRAPPER, [DROP FOREIGN DATA WRAPPER](#)
DROP FOREIGN TABLE, [DROP FOREIGN TABLE](#)
DROP FUNCTION, [DROP FUNCTION](#)
DROP GROUP, [DROP GROUP](#)
DROP INDEX, [DROP INDEX](#)
DROP LANGUAGE, [DROP LANGUAGE](#)
DROP OPERATOR, [DROP OPERATOR](#)
DROP OPERATOR CLASS, [DROP OPERATOR CLASS](#)
DROP OPERATOR FAMILY, [DROP OPERATOR FAMILY](#)
DROP OWNED, [DROP OWNED](#)
DROP ROLE, [データベースロール](#), [DROP ROLE](#)
DROP RULE, [DROP RULE](#)
DROP SCHEMA, [DROP SCHEMA](#)
DROP SEQUENCE, [DROP SEQUENCE](#)
DROP SERVER, [DROP SERVER](#)
DROP TABLE, [新しいテーブルの作成](#), [DROP TABLE](#)
DROP TABLESPACE, [DROP TABLESPACE](#)
DROP TEXT SEARCH CONFIGURATION, [DROP TEXT SEARCH CONFIGURATION](#)
DROP TEXT SEARCH DICTIONARY, [DROP TEXT SEARCH DICTIONARY](#)
DROP TEXT SEARCH PARSER, [DROP TEXT SEARCH PARSER](#)
DROP TEXT SEARCH TEMPLATE, [DROP TEXT SEARCH TEMPLATE](#)
DROP TRIGGER, [DROP TRIGGER](#)
DROP TYPE, [DROP TYPE](#)
DROP USER, [DROP USER](#)
DROP USER MAPPING, [DROP USER MAPPING](#)
DROP VIEW, [DROP VIEW](#)
dropdb, [データベースの削除](#), [dropdb](#)
droplang, [droplang](#)
dropuser, [データベースロール](#), [dropuser](#)
DTrace, [インストール手順](#), [動的追跡](#)
dummy_seclabel, [dummy_seclabel](#)
dynamic_library_path, [動的ロード](#)
dynamic_library_path設定パラメータ, [その他のデフォルト](#)

E

earthdistance, [earthdistance](#)
ECPG, [ECPG - C言語による埋め込みSQL](#), [ecpg](#)
effective_cache_size設定パラメータ, [プランナコスト定数](#)
effective_io_concurrency設定パラメータ, [非同期動作](#)
elog, [サーバ内部からのエラーの報告](#)

PL/Perlにおける, [PL/Perlのユーティリティ関数](#)

PL/Pythonにおける, [ユーティリティ関数](#)

PL/Tclにおける, [PL/Tclからのデータベースアクセス](#)

enabled role, [enabled_roles](#)
enable_bitmapscan設定パラメータ, [プランナメソッド設定](#)
enable_hashagg設定パラメータ, [プランナメソッド設定](#)
enable_hashjoin設定パラメータ, [プランナメソッド設定](#)
enable_indexonlyscan 設定パラメータ, [プランナメソッド設定](#)
enable_indexscan設定パラメータ, [プランナメソッド設定](#)
enable_material設定パラメータ, [プランナメソッド設定](#)
enable_mergejoin設定パラメータ, [プランナメソッド設定](#)

enable_nestloop 設定パラメータ, [プランナメソッド設定](#)
enable_seqscan 設定パラメータ, [プランナメソッド設定](#)
enable_sort 設定パラメータ, [プランナメソッド設定](#)
enable_tidscan 設定パラメータ, [プランナメソッド設定](#)
encode, [文字列関数と演算子](#), [バイナリ文字列関数と演算子](#)
encode_array_constructor

PL/Perlにおける, [PL/Perlのユーティリティ関数](#)

encode_array_literal

PL/Perlにおける, [PL/Perlのユーティリティ関数](#)

encode_bytea

PL/Perlにおける, [PL/Perlのユーティリティ関数](#)

encode_typed_literal

PL/Perlにおける, [PL/Perlのユーティリティ関数](#)

END, [END](#)

enum_first, [列挙型サポート関数](#)

enum_last, [列挙型サポート関数](#)

enum_range, [列挙型サポート関数](#)

ereport, [サーバ内部からのエラーの報告](#)

escape_string_warning 設定パラメータ, [以前のPostgreSQLバージョン](#)

event log

event log, [WindowsにおけるEvent Logの登録](#)

every, [集約関数](#)

EXCEPT, [問い合わせの結合](#)

exceptions

PL/pgSQLにおける, [エラーの捕捉](#)

EXECUTE, [EXECUTE](#)

EXISTS, [副問い合わせ式](#)

EXIT

in PL/pgSQL, [EXIT](#)

exit_on_error 設定パラメータ, [エラー処理](#)

exp, [算術関数と演算子](#)

EXPLAIN, [EXPLAINの利用](#), [EXPLAIN](#)

external_pid_file 設定パラメータ, [ファイルの場所](#)

extract, [日付/時刻関数と演算子](#), [EXTRACT](#), [date_part](#)

extra_float_digits 設定パラメータ, [ロケールと書式設定](#)

F

failover, [高可用性、負荷分散およびレプリケーション](#)

family, [ネットワークアドレス関数と演算子](#)

fdw_handler, [疑似データ型](#)

FETCH, [FETCH](#)

file_fdw, [file_fdw](#)

first_value, [ウィンドウ関数](#)

flex, [必要条件](#)

float4

参照 [real](#)

float8

参照 [double precision](#)

floor, [算術関数と演算子](#)
format, [文字列関数と演算子](#)

PL/pgSQL での使用, [動的コマンドの実行](#)

format_type, [システム情報関数](#)
FreeBSD

IPC設定, [共有メモリとセマフォ](#)

の起動スクリプト, [データベースサーバの起動](#)

共有ライブラリ, [動的にロードされる関数のコンパイルとリンク](#)

from_collapse_limit設定パラメータ, [その他のプランナオプション](#)
FSM

参照 空き領域マップ

fsync設定パラメータ, [諸設定](#)
full text search, [全文検索](#)
full_page_writes設定パラメータ, [諸設定](#)
function

SETOF 付き, [集合を返すSQL関数](#)

variadic, [可変長引数を取るSQL関数](#)

引数のデフォルト値, [引数にデフォルト値を持つSQL関数](#)

fuzzystrmatch, [fuzzystrmatch](#)

G

generate_series, [集合を返す関数](#)
generate_subscripts, [集合を返す関数](#)
GEQO

参照 遺伝的問い合わせ最適化

geqo_effort設定パラメータ, [遺伝的問い合わせ最適化](#)
geqo_generations設定パラメータ, [遺伝的問い合わせ最適化](#)
geqo_pool_size設定パラメータ, [遺伝的問い合わせ最適化](#)
geqo_seed設定パラメータ, [遺伝的問い合わせ最適化](#)
geqo_selection_bias設定パラメータ, [遺伝的問い合わせ最適化](#)
geqo_threshold設定パラメータ, [遺伝的問い合わせ最適化](#)
geqo設定パラメータ, [遺伝的問い合わせ最適化](#)
get_bit, [バイナリ文字列関数と演算子](#)
get_byte, [バイナリ文字列関数と演算子](#)
get_current_ts_config, [テキスト検索関数と演算子](#)
GIN

参照 インデックス

gin_fuzzy_search_limit設定パラメータ, [その他のデフォルト](#)
GiST

参照 インデックス

GRANT, [権限](#), [GRANT](#)
GREATEST, [GREATESTおよびLEAST](#)

結果型の決定, [UNION、CASEおよび関連する構文](#)

GROUP BY, [集約関数](#), [GROUP BYとHAVING句](#)
GSSAPI, [GSSAPI認証](#)
GUID, [UUID型](#)

H

has_any_column_privilege, [システム情報関数](#)
has_column_privilege, [システム情報関数](#)
has_database_privilege, [システム情報関数](#)
has_foreign_data_wrapper_privilege, [システム情報関数](#)
has_function_privilege, [システム情報関数](#)
has_language_privilege, [システム情報関数](#)
has_schema_privilege, [システム情報関数](#)
has_sequence_privilege, [システム情報関数](#)
has_server_privilege, [システム情報関数](#)
has_tablespace_privilege, [システム情報関数](#)
has_table_privilege, [システム情報関数](#)
HAVING, [集約関数](#), [GROUP BYとHAVING句](#)
hba_file設定パラメータ, [ファイルの場所](#)
height, [幾何関数と演算子](#)
high availability, [高可用性、負荷分散およびレプリケーション](#)
host, [ネットワークアドレス関数と演算子](#)
hostmask, [ネットワークアドレス関数と演算子](#)
hot_standby_feedback 設定パラメータ, [スタンバイサーバ](#)
hot_standby設定パラメータ, [スタンバイサーバ](#)
HP-UX

IPC 設定, [共有メモリとセマフォ](#)

共有ライブラリ, [動的にロードされる関数のコンパイルとリンク](#)

上へのインストール, [HP-UX](#)

hstore, [hstore](#)

I

ident, [Ident認証](#)
ident_file設定パラメータ, [ファイルの場所](#)
IFNULL, [COALESCE](#)
ignore_system_indexes設定パラメータ, [開発者向けのオプション](#)
IMMUTABLE, [関数の変動性分類](#)
IN, [副問い合わせ式](#), [行と配列の比較](#)
include

設定ファイル内の, [設定ファイルによるパラメータの設定](#)

include_if_exists

設定ファイル内の, [設定ファイルによるパラメータの設定](#)

index

GIN

text search, [GiSTおよびGINインデックス種類](#)

GiST

text search, [GiSTおよびGINインデックス種類](#)

index-only scan, [プランナメソッド設定](#)

inet_client_addr, [システム情報関数](#)

inet_client_port, [システム情報関数](#)

inet_server_addr, [システム情報関数](#)

inet_server_port, [システム情報関数](#)

inet (データ型), [inet](#)

initcap, [文字列関数と演算子](#)

initdb, [データベースクラスタの作成](#), [initdb](#), [使用法](#)

INSERT, [テーブルに行を挿入](#), [データの挿入](#), [INSERT](#)

instr, [移植例](#)

int2

参照 [smallint](#)

int4

参照 [integer](#)

int8

参照 [bigint](#)

intagg, [intagg](#)

intarray, [intarray](#)

integer, [数値定数](#), [整数データ型](#)

integer_datetimes [設定パラメータ](#), [設定済みのオプション](#)

internal, [疑似データ型](#)

INTERSECT, [問い合わせの結合](#)

interval, [日付/時刻データ型](#)

IntervalStyle [設定パラメータ](#), [ロケールと書式設定](#)

IRIX

共有ライブラリ, [動的にロードされる関数のコンパイルとリンク](#)

上へのインストール, [IRIX](#)

IS DISTINCT FROM, [比較演算子](#), [行と配列の比較](#)

IS DOCUMENT, [IS DOCUMENT](#)

IS FALSE, [比較演算子](#)

IS NOT DISTINCT FROM, [比較演算子](#), [行と配列の比較](#)

IS NOT FALSE, [比較演算子](#)

IS NOT NULL, [比較演算子](#)

IS NOT TRUE, [比較演算子](#)

IS NOT UNKNOWN, [比較演算子](#)

IS NULL, [比較演算子](#), [プラットフォームとクライアント互換性](#)

IS TRUE, [比較演算子](#)

IS UNKNOWN, [比較演算子](#)

isclosed, [幾何関数と演算子](#)

isempty, [範囲関数と演算子](#)

isfinite, [日付/時刻関数と演算子](#)

isn, [isn](#)

ISNULL, [比較演算子](#)

isopen, [幾何関数と演算子](#)

is_array_ref

PL/Perlにおける, [PL/Perlのユーティリティ関数](#)

J

join_collapse_limit [設定パラメータ](#), [その他のプランナオプション](#)

JSON, [JSONデータ型](#)

関数と演算子, [JSON関数](#)

justify_days, [日付/時刻関数と演算子](#)

justify_hours, [日付/時刻関数と演算子](#)

justify_interval, [日付/時刻関数と演算子](#)

K

Kerberos, [Kerberos認証](#)

krb_caseins_users [設定パラメータ](#), [セキュリティと認証](#)

krb_server_keyfile [設定パラメータ](#), [セキュリティと認証](#)

krb_srvname [設定パラメータ](#), [セキュリティと認証](#)

L

lag, [ウィンドウ関数](#)
language_handler, [疑似データ型](#)
lastval, [シーケンス操作関数](#)
last_value, [ウィンドウ関数](#)
lc_collate設定パラメータ, [設定済みのオプション](#)
lc_ctype設定パラメータ, [設定済みのオプション](#)
lc_messages設定パラメータ, [ロケールと書式設定](#)
lc_monetary設定パラメータ, [ロケールと書式設定](#)
lc_numeric設定パラメータ, [ロケールと書式設定](#)
lc_time設定パラメータ, [ロケールと書式設定](#)
LDAP, [インストール手順](#), [LDAP認証](#)
LDAPによる接続パラメータ検索, [接続パラメータのLDAP検索](#)
ldconfig, [共有ライブラリ](#)
lead, [ウィンドウ関数](#)
LEAST, [GREATESTおよびLEAST](#)

結果型の決定, [UNION、CASEおよび関連する構文](#)

left, [文字列関数と演算子](#)
length, [文字列関数と演算子](#), [バイナリ文字列関数と演算子](#), [幾何関数と演算子](#), [テキスト検索関数と演算子](#)
length(tsvector), [文書の操作](#)
lex, [必要条件](#)
libedit, [必要条件](#)
libperl, [必要条件](#)
libpq, [libpq - C ライブラリ](#)

単一行モード, [1行1行問い合わせ結果を受け取る](#)

libpq-fe.h, [libpq - C ライブラリ](#), [接続状態関数](#)
libpq-int.h, [接続状態関数](#)
LIKE, [LIKE](#)

とロケール, [動作](#)

LIMIT, [LIMITとOFFSET](#)
Linux

IPC設定, [共有メモリとセマフォ](#)

の起動スクリプト, [データベースサーバの起動](#)

共有ライブラリ, [動的にロードされる関数のコンパイルとリンク](#)

LISTEN, [LISTEN](#)
listen_addresses設定パラメータ, [接続設定](#)
ln, [算術関数と演算子](#)
lo, [lo](#)
lo-compat-privileges設定パラメータ, [以前のPostgreSQLバージョン](#)
LOAD, [LOAD](#)
load balancing, [高可用性、負荷分散およびレプリケーション](#)
localtime, [日付/時刻関数と演算子](#)
localtimestamp, [日付/時刻関数と演算子](#)
local_preload_libraries設定パラメータ, [その他のデフォルト](#)
LOCK, [テーブルレベルロック](#), [LOCK](#)
log, [算術関数と演算子](#)
logging_collector設定パラメータ, [ログの出力先](#)
log_autovacuum_min_duration設定パラメータ, [自動Vacuum作業](#)
log_btree_build_stats設定パラメータ, [開発者向けのオプション](#)
log_checkpoints設定パラメータ, [何をログに](#)
log_connections設定パラメータ, [何をログに](#)
log_destination設定パラメータ, [ログの出力先](#)
log_directory設定パラメータ, [ログの出力先](#)
log_disconnections設定パラメータ, [何をログに](#)
log_duration設定パラメータ, [何をログに](#)

log_error_verbosity設定パラメータ, [何をログに](#)
log_executor_stats設定パラメータ, [統計情報の監視](#)
log_filename設定パラメータ, [ログの出力先](#)
log_file_mode 設定パラメータ, [ログの出力先](#)
log_hostname設定パラメータ, [何をログに](#)
log_line_prefix設定パラメータ, [何をログに](#)
log_lock_waits設定パラメータ, [何をログに](#)
log_min_duration_statement設定パラメータ, [いつログを取得するか](#)
log_min_error_statement設定パラメータ, [いつログを取得するか](#)
log_min_messages設定パラメータ, [いつログを取得するか](#)
log_parser_stats設定パラメータ, [統計情報の監視](#)
log_planner_stats設定パラメータ, [統計情報の監視](#)
log_rotation_age設定パラメータ, [ログの出力先](#)
log_rotation_size設定パラメータ, [ログの出力先](#)
log_statement_stats設定パラメータ, [統計情報の監視](#)
log_statement設定パラメータ, [何をログに](#)
log_temp_files設定パラメータ, [何をログに](#)
log_timezone設定パラメータ, [何をログに](#)
log_truncate_on_rotation設定パラメータ, [ログの出力先](#)
looks_like_number

PL/Perlにおける, [PL/Perlのユーティリティ関数](#)

lower, [文字列関数と演算子](#), [範囲関数と演算子](#)

とロケール, [動作](#)

lower_inc, [範囲関数と演算子](#)
lower_inf, [範囲関数と演算子](#)
lo_close, [ラジオブジェクト記述子を閉じる](#)
lo_creat, [ラジオブジェクトの作成, サーバ側の関数](#)
lo_create, [ラジオブジェクトの作成, サーバ側の関数](#)
lo_export, [ラジオブジェクトのエクスポート, サーバ側の関数](#)
lo_import, [ラジオブジェクトのインポート, サーバ側の関数](#)
lo_import_with_oid, [ラジオブジェクトのインポート](#)
lo_lseek, [ラジオブジェクトのシーク](#)
lo_open, [既存のラジオブジェクトのオープン](#)
lo_read, [ラジオブジェクトからのデータの読み込み](#)
lo_tell, [ラジオブジェクトのシーク位置の入手](#)
lo_truncate, [ラジオブジェクトを切り詰める](#)
lo_unlink, [ラジオブジェクトの削除, サーバ側の関数](#)
lo_write, [ラジオブジェクトへのデータの書き込み](#)
lpad, [文字列関数と演算子](#)
lseg, [線分, 幾何関数と演算子](#)
ltree, [ltree](#)
ltrim, [文字列関数と演算子](#)

M

Mac OS X

IPC設定, [共有メモリとセマフォ](#)

共有ライブラリ, [動的にロードされる関数のコンパイルとリンク](#)

macaddr (データ型) , [macaddr](#)
MACアドレス

参照 [macaddr](#)

maintenance_work_mem設定パラメータ, [メモリ](#)
make, [必要条件](#)
MANPATH, [環境変数](#)
masklen, [ネットワークアドレス関数と演算子](#)
max, [集約関数](#)
max_connections設定パラメータ, [接続設定](#)

max_files_per_process設定パラメータ, [カーネル資源使用](#)
max_function_args設定パラメータ, [設定済みのオプション](#)
max_identifier_length設定パラメータ, [設定済みのオプション](#)
max_index_keys設定パラメータ, [設定済みのオプション](#)
max_locks_per_transaction設定パラメータ, [ロック管理](#)
max_pred_locks_per_transaction 設定パラメータ, [ロック管理](#)
max_prepared_transactions設定パラメータ, [メモリ](#)
max_stack_depth設定パラメータ, [メモリ](#)
max_standby_archive_delay設定パラメータ, [スタンバイサーバ](#)
max_standby_streaming_delay 設定パラメータ, [スタンバイサーバ](#)
max_wal_senders設定パラメータ, [送出サーバ群](#)
md5, [文字列関数と演算子](#), [バイナリ文字列関数と演算子](#), [パスワード認証](#)
min, [集約関数](#)
MinGW

上へのインストール, [MinGW/ネイティブWindows](#)

mod, [算術関数と演算子](#)
MOVE, [MOVE](#)
MVCC, [序文](#)

N

NaN

参照 [非数](#)

natural結合, [結合テーブル](#)
NetBSD

IPC設定, [共有メモリとセマフォ](#)

の起動スクリプト, [データベースサーバの起動](#)

共有ライブラリ, [動的にロードされる関数のコンパイルとリンク](#)

netmask, [ネットワークアドレス関数と演算子](#)
network, [ネットワークアドレス関数と演算子](#)
nextval, [シーケンス操作関数](#)
NFS

参照 [ネットワークファイルシステム](#)

NOT IN, [副問い合わせ式](#), [行と配列の比較](#)
NOTIFY, [NOTIFY](#)

libpqにおける, [非同期通知](#)

NOTNULL, [比較演算子](#)
NOT (演算子), [論理演算子](#)
now, [日付/時刻関数と演算子](#)
npoints, [幾何関数と演算子](#)
nth_value, [ウィンドウ関数](#)
ntile, [ウィンドウ関数](#)
NULLIF, [NULLIF](#)
NULL値

検査制約, [検査制約](#)

DISTINCT内の, [DISTINCT](#)

libpqにおける, [問い合わせ結果の情報の取り出し](#)

PL/Perlにおける, [PL/Perl関数と引数](#)

PL/Pythonにおける, [NullとNone](#)

一意性制約, [一意性制約](#)

デフォルト値, [デフォルト値](#)

比較, [比較演算子](#)

numeric, [数値定数](#)

numeric (データ型), [任意の精度を持つ数](#)

numnode, [テキスト検索関数と演算子](#), [問い合わせを操作する](#)

NVL, [COALESCE](#)

O

obj_description, [システム情報関数](#)

octet_length, [文字列関数と演算子](#), [バイナリ文字列関数と演算子](#)

OFFSET, [LIMITとOFFSET](#)

oid, [オブジェクト識別子データ型](#)

列, [システム列](#)

oid2name, [oid2name](#)

ONLY, [FROM句](#)

opaque, [疑似データ型](#)

OpenBSD

IPC configuration, [共有メモリとセマフォ](#)

IPC設定, [共有メモリとセマフォ](#)

の起動スクリプト, [データベースサーバの起動](#)

共有ライブラリ, [動的にロードされる関数のコンパイルとリンク](#)

OpenSSL, [インストール手順](#)

参照 SSL

Oracle

PL/SQLからPL/pgSQLへの移植, [Oracle PL/SQLからの移植](#)

ORDER BY, [テーブルへの問い合わせ](#), [行の並び替え](#)

とロケール, [動作](#)

OR (演算子), [論理演算子](#)

overlay, [文字列関数と演算子](#), [バイナリ文字列関数と演算子](#)

OVER句, [ウィンドウ関数呼び出し](#)

P

pageinspect, [pageinspect](#)

palloc, [コードの作成](#)

PAM, [インストール手順](#), [PAM認証](#)

passwordcheck, [パスワード検査](#)

password_encryption設定パラメータ, [セキュリティと認証](#)

path, [幾何関数と演算子](#), [環境変数](#)

path (データ型), [経路](#)

pause_at_recovery_target リカバリ用パラメータ, [リカバリ対象の設定](#)

pclose, [幾何関数と演算子](#)

peer, [Peer認証](#)

percent_rank, [ウィンドウ関数](#)

perl, [必要条件](#), [PL/Perl - Perl手続き言語](#)

pfree, [コードの作成](#)

PGAPPNAME, [環境変数](#)

pgbench, [pgbench](#)

PGcancel, [処理中の問い合わせのキャンセル](#)
PGCLIENTENCODING, [環境変数](#)
PGconn, [データベース接続制御関数](#)
PGCONNECT_TIMEOUT, [環境変数](#)
pgcrypto, [pgcrypto](#)
PGDATA, [データベースクラスタの作成](#)
PGDATABASE, [環境変数](#)
PGDATESTYLE, [環境変数](#)
PGEvtProc, [イベントコールバックプロシージャ](#)
PGGEQO, [環境変数](#)
PGGSSLIB, [環境変数](#)
PGHOST, [環境変数](#)
PGHOSTADDR, [環境変数](#)
PGKRBSRVNAME, [環境変数](#)
PGLOCALEDIR, [環境変数](#)
PGOPTIONS, [環境変数](#)
PGPASSFILE, [環境変数](#)
PGPASSWORD, [環境変数](#)
PGPORT, [環境変数](#)
PGREALM, [環境変数](#)
PGREQUIREPEER, [環境変数](#)
PGREQUIRESSL, [環境変数](#)
PGresult, [主要な関数](#)
pgrowlocks, [pgrowlocks](#)
PGSERVICE, [環境変数](#)
PGSERVICEFILE, [環境変数](#)
PGSSLCERT, [環境変数](#)
PGSSLCOMPRESSION, [環境変数](#)
PGSSLCRL, [環境変数](#)
PGSSLKEY, [環境変数](#)
PGSSLMODE, [環境変数](#)
PGSSLROOTCERT, [環境変数](#)
pgstattuple, [pgstattuple](#)
PGSYSCONFDIR, [環境変数](#)
PGTZ, [環境変数](#)
PGUSER, [環境変数](#)
pgxs, [拡張構築基盤](#)
pg_advisory_lock, [勧告的ロック用関数](#)
pg_advisory_lock_shared, [勧告的ロック用関数](#)
pg_advisory_unlock, [勧告的ロック用関数](#)
pg_advisory_unlock_all, [勧告的ロック用関数](#)
pg_advisory_unlock_shared, [勧告的ロック用関数](#)
pg_advisory_xact_lock, [勧告的ロック用関数](#)
pg_advisory_xact_lock_shared, [勧告的ロック用関数](#)
pg_aggregate, [pg_aggregate](#)
pg_am, [pg_am](#)
pg_amop, [pg_amop](#)
pg_amproc, [pg_amproc](#)
pg_archivecleanup, [pg_archivecleanup](#)
pg_attrdef, [pg_attrdef](#)
pg_attribute, [pg_attribute](#)
pg_authid, [pg_authid](#)
pg_auth_members, [pg_auth_members](#)
pg_available_extensions, [pg_available_extensions](#)
pg_available_extension_versions, [pg_available_extension_versions](#)
pg_backend_pid, [システム情報関数](#)
pg_basebackup, [pg_basebackup](#)
pg_buffercache, [pg_buffercache](#)
pg_cancel_backend, [サーバシグナル送信関数](#)
pg_cast, [pg_cast](#)
pg_class, [pg_class](#)
pg_client_encoding, [文字列関数と演算子](#)
pg_collation, [pg_collation](#)
pg_collation_is_visible, [システム情報関数](#)
pg_column_size, [データベースオブジェクト管理関数](#)

pg_config, [pg_config](#)

ユーザ定義C関数, [コードの作成](#)

pg_conf_load_time, [システム情報関数](#)

pg_constraint, [pg_constraint](#)

pg_controldata, [pg_controldata](#)

pg_conversion, [pg_conversion](#)

pg_conversion_is_visible, [システム情報関数](#)

pg_create_restore_point, [バックアップ制御関数](#)

pg_ctl, [データベースクラスタの作成](#), [データベースサーバの起動](#), [pg_ctl](#)

pg_current_xlog_insert_location, [バックアップ制御関数](#)

pg_current_xlog_location, [バックアップ制御関数](#)

pg_cursors, [pg_cursors](#)

pg_database, [テンプレートデータベース](#), [pg_database](#)

pg_database_size, [データベースオブジェクト管理関数](#)

pg_db_role_setting, [pg_db_role_setting](#)

pg_default_acl, [pg_default_acl](#)

pg_depend, [pg_depend](#)

pg_describe_object, [システム情報関数](#)

pg_description, [pg_description](#)

pg_dump, [pg_dump](#)

pg_dumpall, [pg_dumpall](#)

アップグレード中の使用, [pg_dumpを介したデータのアップグレード](#)

pg_enum, [pg_enum](#)

pg_export_snapshot, [スナップショット同期関数](#)

pg_extension, [pg_extension](#)

pg_extension_config_dump, [拡張設定テーブル](#)

pg_foreign_data_wrapper, [pg_foreign_data_wrapper](#)

pg_foreign_server, [pg_foreign_server](#)

pg_foreign_table, [pg_foreign_table](#)

pg_freespacemap, [pg_freespacemap](#)

pg_function_is_visible, [システム情報関数](#)

pg_get_constraintdef, [システム情報関数](#)

pg_get_expr, [システム情報関数](#)

pg_get_functiondef, [システム情報関数](#)

pg_get_function_arguments, [システム情報関数](#)

pg_get_function_identity_arguments, [システム情報関数](#)

pg_get_function_result, [システム情報関数](#)

pg_get_indexdef, [システム情報関数](#)

pg_get_keywords, [システム情報関数](#)

pg_get_ruledef, [システム情報関数](#)

pg_get_serial_sequence, [システム情報関数](#)

pg_get_triggerdef, [システム情報関数](#)

pg_get_userbyid, [システム情報関数](#)

pg_get_viewdef, [システム情報関数](#)

pg_group, [pg_group](#)

pg_has_role, [システム情報関数](#)

pg_hba.conf, [pg_hba.confファイル](#)

pg_ident.conf, [ユーザ名マップ](#)

pg_index, [pg_index](#)

pg_indexes, [pg_indexes](#)

pg_indexes_size, [データベースオブジェクト管理関数](#)

pg_inherits, [pg_inherits](#)

pg_is_in_recovery, [リカバリ制御関数](#)

pg_is_other_temp_schema, [システム情報関数](#)

pg_is_xlog_replay_paused, [リカバリ制御関数](#)

pg_language, [pg_language](#)

pg_largeobject, [pg_largeobject](#)

pg_largeobject_metadata, [pg_largeobject_metadata](#)

pg_last_xact_replay_timestamp, [リカバリ制御関数](#)

pg_last_xlog_receive_location, [リカバリ制御関数](#)

pg_last_xlog_replay_location, [リカバリ制御関数](#)

[pg_listening_channels](#), [システム情報関数](#)
[pg_locks](#), [pg_locks](#)
[pg_ls_dir](#), [汎用ファイルアクセス関数](#)
[pg_my_temp_schema](#), [システム情報関数](#)
[pg_namespace](#), [pg_namespace](#)
[pg_notify](#), [pg_notify](#)
[pg_opclass](#), [pg_opclass](#)
[pg_opclass_is_visible](#), [システム情報関数](#)
[pg_operator](#), [pg_operator](#)
[pg_operator_is_visible](#), [システム情報関数](#)
[pg_opfamily](#), [pg_opfamily](#)
[pg_opfamily_is_visible](#), [システム情報関数](#)
[pg_options_to_table](#), [システム情報関数](#)
[pg_pltemplate](#), [pg_pltemplate](#)
[pg_postmaster_start_time](#), [システム情報関数](#)
[pg_prepared_statements](#), [pg_prepared_statements](#)
[pg_prepared_xacts](#), [pg_prepared_xacts](#)
[pg_proc](#), [pg_proc](#)
[pg_range](#), [pg_range](#)
[pg_read_binary_file](#), [汎用ファイルアクセス関数](#)
[pg_read_file](#), [汎用ファイルアクセス関数](#)
[pg_receivexlog](#), [pg_receivexlog](#)
[pg_relation_filename](#), [データベースオブジェクト管理関数](#)
[pg_relation_filepath](#), [データベースオブジェクト管理関数](#)
[pg_relation_size](#), [データベースオブジェクト管理関数](#)
[pg_reload_conf](#), [サーバシグナル送信関数](#)
[pg_resetxlog](#), [pg_resetxlog](#)
[pg_restore](#), [pg_restore](#)
[pg_rewrite](#), [pg_rewrite](#)
[pg_roles](#), [pg_roles](#)
[pg_rotate_logfile](#), [サーバシグナル送信関数](#)
[pg_rules](#), [pg_rules](#)
[pg_seclabel](#), [pg_seclabel](#)
[pg_seclabels](#), [pg_seclabels](#)
[pg_service.conf](#), [接続サービスファイル](#)
[pg_settings](#), [pg_settings](#)
[pg_shadow](#), [pg_shadow](#)
[pg_shdepend](#), [pg_shdepend](#)
[pg_shdescription](#), [pg_shdescription](#)
[pg_shseclabel](#), [pg_shseclabel](#)
[pg_size_pretty](#), [データベースオブジェクト管理関数](#)
[pg_sleep](#), [遅延実行](#)
[pg_standby](#), [pg_standby](#)
[pg_start_backup](#), [バックアップ制御関数](#)
[pg_statio_all_indexes](#), [収集した統計情報の表示](#)
[pg_statio_all_sequences](#), [収集した統計情報の表示](#)
[pg_statio_all_tables](#), [収集した統計情報の表示](#)
[pg_statio_sys_indexes](#), [収集した統計情報の表示](#)
[pg_statio_sys_sequences](#), [収集した統計情報の表示](#)
[pg_statio_sys_tables](#), [収集した統計情報の表示](#)
[pg_statio_user_indexes](#), [収集した統計情報の表示](#)
[pg_statio_user_sequences](#), [収集した統計情報の表示](#)
[pg_statio_user_tables](#), [収集した統計情報の表示](#)
[pg_statistic](#), [プランナで使用する統計情報](#), [pg_statistic](#)
[pg_stats](#), [プランナで使用する統計情報](#), [pg_stats](#)
[pg_stat_activity](#), [収集した統計情報の表示](#)
[pg_stat_all_indexes](#), [収集した統計情報の表示](#)
[pg_stat_all_tables](#), [収集した統計情報の表示](#)
[pg_stat_bgwriter](#), [収集した統計情報の表示](#)
[pg_stat_database](#), [収集した統計情報の表示](#)
[pg_stat_database_conflicts](#), [収集した統計情報の表示](#)
[pg_stat_file](#), [汎用ファイルアクセス関数](#)
[pg_stat_replication](#), [収集した統計情報の表示](#)
[pg_stat_statements](#), [pg_stat_statements](#)
[pg_stat_sys_indexes](#), [収集した統計情報の表示](#)

[pg_stat_sys_tables](#), [収集した統計情報の表示](#)
[pg_stat_user_functions](#), [収集した統計情報の表示](#)
[pg_stat_user_indexes](#), [収集した統計情報の表示](#)
[pg_stat_user_tables](#), [収集した統計情報の表示](#)
[pg_stat_xact_all_tables](#), [収集した統計情報の表示](#)
[pg_stat_xact_sys_tables](#), [収集した統計情報の表示](#)
[pg_stat_xact_user_functions](#), [収集した統計情報の表示](#)
[pg_stat_xact_user_tables](#), [収集した統計情報の表示](#)
[pg_stop_backup](#), [バックアップ制御関数](#)
[pg_switch_xlog](#), [バックアップ制御関数](#)
[pg_tables](#), [pg_tables](#)
[pg_tablespace](#), [pg_tablespace](#)
[pg_tablespace_databases](#), [システム情報関数](#)
[pg_tablespace_location](#), [システム情報関数](#)
[pg_tablespace_size](#), [データベースオブジェクト管理関数](#)
[pg_table_is_visible](#), [システム情報関数](#)
[pg_table_size](#), [データベースオブジェクト管理関数](#)
[pg_terminate_backend](#), [サーバシグナル送信関数](#)
[pg_test_fsync](#), [pg_test_fsync](#)
[pg_test_timing](#), [pg_test_timing](#)
[pg_timezone_abbrevs](#), [pg_timezone_abbrevs](#)
[pg_timezone_names](#), [pg_timezone_names](#)
[pg_total_relation_size](#), [データベースオブジェクト管理関数](#)
[pg_trgm](#), [pg_trgm](#)
[pg_trigger](#), [pg_trigger](#)
[pg_try_advisory_lock](#), [勧告的ロック用関数](#)
[pg_try_advisory_lock_shared](#), [勧告的ロック用関数](#)
[pg_try_advisory_xact_lock](#), [勧告的ロック用関数](#)
[pg_try_advisory_xact_lock_shared](#), [勧告的ロック用関数](#)
[pg_ts_config](#), [pg_ts_config](#)
[pg_ts_config_is_visible](#), [システム情報関数](#)
[pg_ts_config_map](#), [pg_ts_config_map](#)
[pg_ts_dict](#), [pg_ts_dict](#)
[pg_ts_dict_is_visible](#), [システム情報関数](#)
[pg_ts_parser](#), [pg_ts_parser](#)
[pg_ts_parser_is_visible](#), [システム情報関数](#)
[pg_ts_template](#), [pg_ts_template](#)
[pg_ts_template_is_visible](#), [システム情報関数](#)
[pg_type](#), [pg_type](#)
[pg_typeof](#), [システム情報関数](#)
[pg_type_is_visible](#), [システム情報関数](#)
[pg_upgrade](#), [pg_upgrade](#)
[pg_user](#), [pg_user](#)
[pg_user_mapping](#), [pg_user_mapping](#)
[pg_user_mappings](#), [pg_user_mappings](#)
[pg_views](#), [pg_views](#)
[pg_xlogfile_name](#), [バックアップ制御関数](#)
[pg_xlogfile_name_offset](#), [バックアップ制御関数](#)
[pg_xlog_location_diff](#), [バックアップ制御関数](#)
[pg_xlog_replay_pause](#), [リカバリ制御関数](#)
[pg_xlog_replay_resume](#), [リカバリ制御関数](#)
[pi](#), [算術関数と演算子](#)
[PIC](#), [動的にロードされる関数のコンパイルとリンク](#)
[PID](#)

サーバプロセスのPIDの決定

libpqにおける, [接続状態関数](#)

[PITR](#), [バックアップとリストア](#)
[PITRスタンバイ](#), [高可用性、負荷分散およびレプリケーション](#)
[PL/Perl](#), [PL/Perl - Perl手続き言語](#)
[PL/PerlU](#), [信頼されたPL/Perlおよび信頼されないPL/Perl](#)
[PL/pgSQL](#), [PL/pgSQL - SQL手続き言語](#)
[PL/Python](#), [PL/Python - Python手続き言語](#)

PL/SQL (Oracle)

PL/pgSQLへの移植, [Oracle PL/SQLからの移植](#)

PL/Tcl, [PL/Tcl - Tcl手続き言語](#)

plainto_tsquery, [テキスト検索関数と演算子](#), [問合わせのパス](#)

plperl.on_init設定パラメータ, [設定](#)

plperl.on_plperlu_init設定パラメータ, [設定](#)

plperl.on_plperl_init設定パラメータ, [設定](#)

plperl.use_strict設定パラメータ, [設定](#)

plpgsql.variable_conflict設定パラメータ, [変数置換](#)

point, [座標点](#), [幾何関数と演算子](#)

polygon, [多角形](#), [幾何関数と演算子](#)

popen, [幾何関数と演算子](#)

port, [パラメータキーワード](#)

port設定パラメータ, [接続設定](#)

position, [文字列関数と演算子](#), [バイナリ文字列関数と演算子](#)

POSTGRES, [バークレイ校POSTGRESプロジェクト](#), [構造的な基本事項](#), [データベースサーバの起動](#), [データベースの作成](#), [postgres](#)

Postgres95, [Postgres95](#)

postgres.conf, [設定ファイルによるパラメータの設定](#)

postgresユーザ, [PostgreSQLユーザアカウント](#)

postmaster, [postmaster](#)

post_auth_delay設定パラメータ, [開発者向けのオプション](#)

power, [算術関数と演算子](#)

PQbackendPID, [接続状態関数](#)

PQbinaryTuples, [問い合わせ結果の情報の取り出し](#)

with COPY, [COPYコマンド関連関数](#)

PQcancel, [処理中の問い合わせのキャンセル](#)

PQclear, [主要な関数](#)

PQclientEncoding, [制御関数](#)

PQcmdStatus, [他の結果情報の取り出し](#)

PQcmdTuples, [他の結果情報の取り出し](#)

PQconnndefaults, [データベース接続制御関数](#)

PQconnectdb, [データベース接続制御関数](#)

PQconnectdbParams, [データベース接続制御関数](#)

PQconnectionNeedsPassword, [接続状態関数](#)

PQconnectionUsedPassword, [接続状態関数](#)

PQconnectPoll, [データベース接続制御関数](#)

PQconnectStart, [データベース接続制御関数](#)

PQconnectStartParams, [データベース接続制御関数](#)

PQconninfoFree, [雑多な関数](#)

PQconninfoParse, [データベース接続制御関数](#)

PQconsumeInput, [非同期コマンドの処理](#)

PQcopyResult, [雑多な関数](#)

PQdb, [接続状態関数](#)

PQdescribePortal, [主要な関数](#)

PQdescribePrepared, [主要な関数](#)

PQencryptPassword, [雑多な関数](#)

PQendcopy, [廃れたCOPY用関数](#)

PQerrorMessage, [接続状態関数](#)

PQescapeBytea, [SQLコマンドに含めるための文字列のエスケープ処理](#)

PQescapeByteaConn, [SQLコマンドに含めるための文字列のエスケープ処理](#)

PQescapeIdentifier, [SQLコマンドに含めるための文字列のエスケープ処理](#)

PQescapeLiteral, [SQLコマンドに含めるための文字列のエスケープ処理](#)

PQescapeString, [SQLコマンドに含めるための文字列のエスケープ処理](#)

PQescapeStringConn, [SQLコマンドに含めるための文字列のエスケープ処理](#)

PQexec, [主要な関数](#)

PQexecParams, [主要な関数](#)

PQexecPrepared, [主要な関数](#)

PQfformat, [問い合わせ結果の情報の取り出し](#)

with COPY, [COPYコマンド関連関数](#)

PQfinish, [データベース接続制御関数](#)
PQfireResultCreateEvents, [雑多な関数](#)
PQflush, [非同期コマンドの処理](#)
PQfmod, [問い合わせ結果の情報の取り出し](#)
PQfname, [問い合わせ結果の情報の取り出し](#)
PQfnumber, [問い合わせ結果の情報の取り出し](#)
PQfreeCancel, [処理中の問い合わせのキャンセル](#)
PQfreemem, [雑多な関数](#)
PQfsize, [問い合わせ結果の情報の取り出し](#)
PQftable, [問い合わせ結果の情報の取り出し](#)
PQftablecol, [問い合わせ結果の情報の取り出し](#)
PQftype, [問い合わせ結果の情報の取り出し](#)
PQgetCancel, [処理中の問い合わせのキャンセル](#)
PQgetCopyData, [COPYデータ受信用関数](#)
PQgetisnull, [問い合わせ結果の情報の取り出し](#)
PQgetlength, [問い合わせ結果の情報の取り出し](#)
PQgetline, [廃れたCOPY用関数](#)
PQgetlineAsync, [廃れたCOPY用関数](#)
PQgetResult, [非同期コマンドの処理](#)
PQgetssl, [接続状態関数](#)
PQgetvalue, [問い合わせ結果の情報の取り出し](#)
PQhost, [接続状態関数](#)
PQinitOpenSSL, [SSLライブラリの初期化](#)
PQinitSSL, [SSLライブラリの初期化](#)
PQinstanceData, [イベントサポート関数](#)
PQisBusy, [非同期コマンドの処理](#)
PQisnonblocking, [非同期コマンドの処理](#)
PQisthreadsafe, [スレッド化プログラムの振舞い](#)
PQlibVersion, [雑多な関数](#)

参照 PQserverVersion

PQmakeEmptyPGresult, [雑多な関数](#)
PQnfields, [問い合わせ結果の情報の取り出し](#)

with COPY, [COPYコマンド関連関数](#)

PQnotifies, [非同期通知](#)
PQnparams, [問い合わせ結果の情報の取り出し](#)
PQntuples, [問い合わせ結果の情報の取り出し](#)
PQoidStatus, [他の結果情報の取り出し](#)
PQoidValue, [他の結果情報の取り出し](#)
PQoptions, [接続状態関数](#)
PQparameterStatus, [接続状態関数](#)
PQparamtype, [問い合わせ結果の情報の取り出し](#)
PQpass, [接続状態関数](#)
PQping, [データベース接続制御関数](#)
PQpingParams, [データベース接続制御関数](#)
PQport, [接続状態関数](#)
PQprepare, [主要な関数](#)
PQprint, [問い合わせ結果の情報の取り出し](#)
PQprotocolVersion, [接続状態関数](#)
PQputCopyData, [COPYデータ送信用関数](#)
PQputCopyEnd, [COPYデータ送信用関数](#)
PQputline, [廃れたCOPY用関数](#)
PQputnbytes, [廃れたCOPY用関数](#)
PQregisterEventProc, [イベントサポート関数](#)
PQrequestCancel, [処理中の問い合わせのキャンセル](#)
PQreset, [データベース接続制御関数](#)
PQresetPoll, [データベース接続制御関数](#)
PQresetStart, [データベース接続制御関数](#)
PQresStatus, [主要な関数](#)
PQresultAlloc, [雑多な関数](#)
PQresultErrorField, [主要な関数](#)
PQresultErrorMessage, [主要な関数](#)

PQresultInstanceData, [イベントサポート関数](#)
PQresultSetInstanceData, [イベントサポート関数](#)
PQresultStatus, [主要な関数](#)
PQsendDescribePortal, [非同期コマンドの処理](#)
PQsendDescribePrepared, [非同期コマンドの処理](#)
PQsendPrepare, [非同期コマンドの処理](#)
PQsendQuery, [非同期コマンドの処理](#)
PQsendQueryParams, [非同期コマンドの処理](#)
PQsendQueryPrepared, [非同期コマンドの処理](#)
PQserverVersion, [接続状態関数](#)
PQsetClientEncoding, [制御関数](#)
PQsetdb, [データベース接続制御関数](#)
PQsetdbLogin, [データベース接続制御関数](#)
PQsetErrorVerbosity, [制御関数](#)
PQsetInstanceData, [イベントサポート関数](#)
PQsetnonblocking, [非同期コマンドの処理](#)
PQsetNoticeProcessor, [警告処理](#)
PQsetNoticeReceiver, [警告処理](#)
PQsetResultAttrs, [雑多な関数](#)
PQsetSingleRowMode, [1行1行間い合わせ結果を受け取る](#)
PQsetvalue, [雑多な関数](#)
PQsocket, [接続状態関数](#)
PQstatus, [接続状態関数](#)
PQtrace, [制御関数](#)
PQtransactionStatus, [接続状態関数](#)
PQtty, [接続状態関数](#)
PQunescapeBytea, [SQLコマンドに含めるための文字列のエスケープ処理](#)
PQuntrace, [制御関数](#)
PQuser, [接続状態関数](#)
PREPARE, [PREPARE](#)
PREPARE TRANSACTION, [PREPARE TRANSACTION](#)
pre_auth_delay設定パラメータ, [開発者向けのオプション](#)
primary_conninfoリカバリ用パラメータ, [スタンバイサーバの設定](#)
ps

活動状況監視用の, [標準的なUnixツール](#)

psql, [データベースへのアクセス](#), [psql](#)
Python, [PL/Python - Python手続き言語](#)

Q

querytree, [テキスト検索関数と演算子](#), [間合わせを操作する](#)
quote_all_identifiers 設定パラメータ, [以前のPostgreSQLバージョン](#)
quote_ident, [文字列関数と演算子](#)

PL/Perlにおける, [PL/Perlのユーティリティ関数](#)

PL/pgSQL での使用, [動的コマンドの実行](#)

quote_literal, [文字列関数と演算子](#)

PL/Perlにおける, [PL/Perlのユーティリティ関数](#)

PL/pgSQL での使用, [動的コマンドの実行](#)

quote_nullable, [文字列関数と演算子](#)

PL/Perlにおける, [PL/Perlのユーティリティ関数](#)

PL/pgSQL での使用, [動的コマンドの実行](#)

R

radians, [算術関数と演算子](#)

radius, [幾何関数と演算子](#), [RADIUS Authentication](#)
RAISE, [エラーとメッセージ](#)
random, [算術関数と演算子](#)
random_page_cost 設定パラメータ, [プランナコスト定数](#)
rank, [ウィンドウ関数](#)
readline, [必要条件](#)
real, [浮動小数点データ型](#)
REASSIGN OWNED, [REASSIGN OWNED](#)
record, [疑似データ型](#)
recovery.conf, [リカバリの設定](#)
recovery_end_command リカバリ用パラメータ, [アーカイブリカバリの設定](#)
recovery_target_inclusive リカバリ用パラメータ, [リカバリ対象の設定](#)
recovery_target_name リカバリ用パラメータ, [リカバリ対象の設定](#)
recovery_target_timeline リカバリ用パラメータ, [リカバリ対象の設定](#)
recovery_target_time リカバリ用パラメータ, [リカバリ対象の設定](#)
recovery_target_xid リカバリ用パラメータ, [リカバリ対象の設定](#)
regclass, [オブジェクト識別子データ型](#)
regconfig, [オブジェクト識別子データ型](#)
regdictionary, [オブジェクト識別子データ型](#)
regexp_matches, [文字列関数と演算子](#), [POSIX正規表現](#)
regexp_replace, [文字列関数と演算子](#), [POSIX正規表現](#)
regexp_split_to_array, [文字列関数と演算子](#), [POSIX正規表現](#)
regexp_split_to_table, [文字列関数と演算子](#), [POSIX正規表現](#)
regoper, [オブジェクト識別子データ型](#)
regoperator, [オブジェクト識別子データ型](#)
regproc, [オブジェクト識別子データ型](#)
regprocedure, [オブジェクト識別子データ型](#)
regr_avgx, [集約関数](#)
regr_avgy, [集約関数](#)
regr_count, [集約関数](#)
regr_intercept, [集約関数](#)
regr_r2, [集約関数](#)
regr_slope, [集約関数](#)
regr_sxx, [集約関数](#)
regr_sxy, [集約関数](#)
regr_syy, [集約関数](#)
regtype, [オブジェクト識別子データ型](#)
regular expression, [SIMILAR TO正規表現](#)
REINDEX, [REINDEX](#)
reindexdb, [reindexdb](#)
RELEASE SAVEPOINT, [RELEASE SAVEPOINT](#)
repeat, [文字列関数と演算子](#)
replace, [文字列関数と演算子](#)
replication, [高可用性、負荷分散およびレプリケーション](#)
replication_timeout 設定パラメータ, [送出サーバ群](#)
RESET, [RESET](#)
restartpoint, [WALの設定](#)
restart_after_crash 設定パラメータ, [エラー処理](#)
restore_command リカバリ用パラメータ, [アーカイブリカバリの設定](#)
RESTRICT

DROPの, [依存関係の追跡](#)

外部キー動作, [外部キー](#)

RETURN NEXT

PL/pgSQLにおける, [RETURN NEXTおよびRETURN QUERY](#)

RETURN QUERY

PL/pgSQLにおける, [RETURN NEXTおよびRETURN QUERY](#)

RETURNING INTO

PL/pgSQLにおける, [1行の結果を返す問い合わせの実行](#)

reverse, [文字列関数と演算子](#)
REVOKE, [権限](#), [REVOKE](#)
right, [文字列関数と演算子](#)
role

enabled, [enabled_roles](#)

ROLLBACK, [ROLLBACK](#)

psql, [変数](#)

ROLLBACK PREPARED, [ROLLBACK PREPARED](#)
ROLLBACK TO SAVEPOINT, [ROLLBACK TO SAVEPOINT](#)
round, [算術関数と演算子](#)
ROW, [行コンストラクタ](#)
row_number, [ウィンドウ関数](#)
row_to_json, [JSON関数](#)
rpad, [文字列関数と演算子](#)
rtrim, [文字列関数と演算子](#)

S

SAVEPOINT, [SAVEPOINT](#)
SCO

上へのインストール, [SCO OpenServer と SCO UnixWare](#)

SCO OpenServer

IPC設定, [共有メモリとセマフォ](#)

search_path, [スキーマ検索パス](#)
search_path設定パラメータ, [文の動作](#)
SECURITY LABEL, [SECURITY LABEL](#)
seg, [seg](#)
segment_size設定パラメータ, [設定済みのオプション](#)
SELECT, [テーブルへの問い合わせ](#), [問い合わせ](#), [SELECT](#)

選択リスト, [選択リスト](#)

SELECT INTO, [SELECT INTO](#)

PL/pgSQLにおける, [1行の結果を返す問い合わせの実行](#)

sepgsql, [sepgsql](#)
sepgsql.debug_audit configuration parameter, [GUCパラメータ](#)
sepgsql.permissive configuration parameter, [GUCパラメータ](#)
sequence, [シーケンス操作関数](#)
seq_page_cost設定パラメータ, [プランナコスト定数](#)
serial, [連番型](#)
serial2, [連番型](#)
serial4, [連番型](#)
serial8, [連番型](#)
server_encoding設定パラメータ, [設定済みのオプション](#)
server_version_num設定パラメータ, [設定済みのオプション](#)
server_version設定パラメータ, [設定済みのオプション](#)
session_replication_role設定パラメータ, [文の動作](#)
session_user, [システム情報関数](#)
SET, [構成設定関数](#), [SET](#)
SET CONSTRAINTS, [SET CONSTRAINTS](#)
SET ROLE, [SET ROLE](#)
SET SESSION AUTHORIZATION, [SET SESSION AUTHORIZATION](#)
SET TRANSACTION, [SET TRANSACTION](#)
SET XML OPTION, [文の動作](#)
setseed, [算術関数と演算子](#)
setval, [シーケンス操作関数](#)

setweight, [テキスト検索関数と演算子](#), [文書の操作](#)
set_bit, [バイナリ文字列関数と演算子](#)
set_byte, [バイナリ文字列関数と演算子](#)
set_config, [構成設定関数](#)
set_masklen, [ネットワークアドレス関数と演算子](#)
shared_buffers設定パラメータ, [メモリ](#)
shared_preload_libraries, [共有メモリとLWLocks](#)
shared_preload_libraries設定パラメータ, [カーネル資源使用](#)
SHMMAX, [共有メモリとセマフォ](#)
shobj_description, [システム情報関数](#)
SHOW, [構成設定関数](#), [SHOW](#)
SIGHUP, [設定ファイルによるパラメータの設定](#), [pg_hba.confファイル](#), [ユーザ名マップ](#)
SIGINT, [サーバのシャットダウン](#)
sign, [算術関数と演算子](#)
SIGQUIT, [サーバのシャットダウン](#)
SIGTERM, [サーバのシャットダウン](#)
silent_mode設定パラメータ, [ログの出力先](#)
SIMILAR TO, [SIMILAR TO正規表現](#)
sin, [算術関数と演算子](#)
smallint, [整数データ型](#)
smallserial, [連番型](#)
Solaris

IPC設定, [共有メモリとセマフォ](#)

の起動スクリプト, [データベースサーバの起動](#)

共有ライブラリ, [動的にロードされる関数のコンパイルとリンク](#)

上へのインストール, [Solaris](#)

SOME, [集約関数](#), [副問い合わせ式](#), [行と配列の比較](#)
SP-GiST

参照 [インデックス](#)

SPI, [サーバプログラミングインタフェース](#)

examples, [spi](#)

SPI_connect, [SPI_connect](#)
SPI_copytuple, [SPI_copytuple](#)
SPI_cursor_close, [SPI_cursor_close](#)

PL/Perlにおける, [PL/Perlからのデータベースアクセス](#)

SPI_cursor_fetch, [SPI_cursor_fetch](#)
SPI_cursor_find, [SPI_cursor_find](#)
SPI_cursor_move, [SPI_cursor_move](#)
SPI_cursor_open, [SPI_cursor_open](#)
SPI_cursor_open_with_args, [SPI_cursor_open_with_args](#)
SPI_cursor_open_with_paramlist, [SPI_cursor_open_with_paramlist](#)
SPI_exec, [SPI_exec](#)
SPI_execcp, [SPI_execcp](#)
SPI_execute, [SPI_execute](#)
SPI_execute_plan, [SPI_execute_plan](#)
SPI_execute_plan_with_paramlist, [SPI_execute_plan_with_paramlist](#)
SPI_execute_with_args, [SPI_execute_with_args](#)
spi_exec_prepared

PL/Perlにおける, [PL/Perlからのデータベースアクセス](#)

spi_exec_query

PL/Perlにおける, [PL/Perlからのデータベースアクセス](#)

spi_fetchrow

PL/Perlにおける、[PL/Perlからのデータベースアクセス](#)

SPI_finish, [SPI_finish](#)
SPI_fname, [SPI_fname](#)
SPI_fnumber, [SPI_fnumber](#)
SPI_freeplan, [SPI_freeplan](#)

PL/Perlにおける、[PL/Perlからのデータベースアクセス](#)

SPI_freetuple, [SPI_freetuple](#)
SPI_freetuptable, [SPI_freetuptable](#)
SPI_getargcount, [SPI_getargcount](#)
SPI_getargtypeid, [SPI_getargtypeid](#)
SPI_getbinval, [SPI_getbinval](#)
SPI_getnspname, [SPI_getnspname](#)
SPI_getrelname, [SPI_getrelname](#)
SPI_gettype, [SPI_gettype](#)
SPI_gettypeid, [SPI_gettypeid](#)
SPI_getvalue, [SPI_getvalue](#)
SPI_is_cursor_plan, [SPI_is_cursor_plan](#)
SPI_keepplan, [SPI_keepplan](#)
spi_lastoid, [PL/Tclからのデータベースアクセス](#)
SPI_modifytuple, [SPI_modifytuple](#)
SPI_palloc, [SPI_palloc](#)
SPI_pfree, [SPI_pfree](#)
SPI_pop, [SPI_pop](#)
SPI_prepare, [SPI_prepare](#)

PL/Perlにおける、[PL/Perlからのデータベースアクセス](#)

SPI_prepare_cursor, [SPI_prepare_cursor](#)
SPI_prepare_params, [SPI_prepare_params](#)
SPI_push, [SPI_push](#)
spi_query

PL/Perlにおける、[PL/Perlからのデータベースアクセス](#)

spi_query_prepared

PL/Perlにおける、[PL/Perlからのデータベースアクセス](#)

SPI_realloc, [SPI_realloc](#)
SPI_returntuple, [SPI_returntuple](#)
SPI_saveplan, [SPI_saveplan](#)
SPI_scroll_cursor_fetch, [SPI_scroll_cursor_fetch](#)
SPI_scroll_cursor_move, [SPI_scroll_cursor_move](#)
split_part, [文字列関数と演算子](#)
SQL/CLI, [SQLへの準拠](#)
SQL/Foundation, [SQLへの準拠](#)
SQL/Framework, [SQLへの準拠](#)
SQL/JRT, [SQLへの準拠](#)
SQL/MED, [SQLへの準拠](#)
SQL/OLB, [SQLへの準拠](#)
SQL/PSM, [SQLへの準拠](#)
SQL/Schemata, [SQLへの準拠](#)
SQL/XML, [SQLへの準拠](#)
sql_inheritance設定パラメータ, [以前のPostgreSQLバージョン](#)
SQLの拡張, [SQLの拡張](#)
sqrt, [算術関数と演算子](#)
ssh, [SSHトンネルを使った安全なTCP/IP接続](#)
SSI, [序文](#)
SSL, [SSLによる安全なTCP/IP接続](#), [SSLサポート](#)

libpqでの、[パラメータキーワード](#), [接続状態関数](#)

sslinfo, [sslinfo](#)
ssl_ca_file 設定パラメータ, [セキュリティと認証](#)

ssl_cert_file 設定パラメータ, [セキュリティと認証](#)
ssl_ciphers 設定パラメータ, [セキュリティと認証](#)
ssl_crl_file 設定パラメータ, [セキュリティと認証](#)
ssl_key_file 設定パラメータ, [セキュリティと認証](#)
ssl_renegotiation_limit 設定パラメータ, [セキュリティと認証](#)
ssl 設定パラメータ, [セキュリティと認証](#)
SSPI, [SSPI 認証](#)
STABLE, [関数の変動性分類](#)
standard deviation, [集約関数](#)
standard_conforming_strings 設定パラメータ, [以前のPostgreSQLバージョン](#)
standby_mode リカバリ用パラメータ, [スタンバイサーバの設定](#)
START TRANSACTION, [START TRANSACTION](#)
statement_timeout 設定パラメータ, [文の動作](#)
statement_timestamp, [日付/時刻関数と演算子](#)
stats_temp_directory 設定パラメータ, [問い合わせおよびインデックスに関する統計情報コレクタ](#)
stddev, [集約関数](#)
stddev_pop, [集約関数](#)
stddev_samp, [集約関数](#)
STONITH, [高可用性、負荷分散およびレプリケーション](#)
string_agg, [集約関数](#)
string_to_array, [配列関数と演算子](#)
strip, [テキスト検索関数と演算子](#), [文書の操作](#)
strpos, [文字列関数と演算子](#)
subquery, [副問い合わせ式](#)
substr, [文字列関数と演算子](#)
substring, [文字列関数と演算子](#), [バイナリ文字列関数と演算子](#), [SIMILAR TO 正規表現](#), [POSIX 正規表現](#)
sum, [集約関数](#)
superuser_reserved_connections 設定パラメータ, [接続設定](#)
suppress_redundant_updates_trigger, [トリガ関数](#)
synchronize_seqscans 設定パラメータ, [以前のPostgreSQLバージョン](#)
synchronous_commit 設定パラメータ, [諸設定](#)
synchronous_standby_names 設定パラメータ, [マスターサーバ](#)
syslog_facility 設定パラメータ, [ログの出力先](#)
syslog_identity 設定パラメータ, [ログの出力先](#)

T

tablefunc, [tablefunc](#)
tableoid, [システム列](#)
TABLE コマンド, [SELECT](#)
tan, [算術関数と演算子](#)
Tcl, [PL/Tcl - Tcl 手続き言語](#)
tcn, [tcn](#)
tcp_keepalives_count 設定パラメータ, [接続設定](#)
tcp_keepalives_idle 設定パラメータ, [接続設定](#)
tcp_keepalives_interval 設定パラメータ, [接続設定](#)
template0, [テンプレートデータベース](#)
template1, [データベースの作成](#), [テンプレートデータベース](#)
temp_buffers 設定パラメータ, [メモリ](#)
temp_file_limit 設定パラメータ, [ディスク](#)
temp_tablespace 設定パラメータ, [文の動作](#)
test_parser, [test_parser](#)
text, [文字型](#), [ネットワークアドレス関数と演算子](#)
text search, [全文検索](#)

indexes, [GiST および GIN インデックス種類](#)

tid, [オブジェクト識別子データ型](#)
time, [日付/時刻データ型](#), [時刻](#)

出力書式, [日付/時刻出力](#)

参照 書式設定

定数, [特殊な値](#)

time with time zone, [日付/時刻データ型](#), [時刻](#)
time without time zone, [日付/時刻データ型](#), [時刻](#)
timeofday, [日付/時刻関数と演算子](#)
timeout

クライアント認証, [セキュリティと認証](#)

timestamp, [日付/時刻データ型](#), [タイムスタンプ](#)
timestamp with time zone, [日付/時刻データ型](#), [タイムスタンプ](#)
timestamp without time zone, [日付/時刻データ型](#), [タイムスタンプ](#)
timestamptz, [日付/時刻データ型](#)
timezone_abbreviations設定パラメータ, [ロケールと書式設定](#)
TimeZone設定パラメータ, [ロケールと書式設定](#)
TOAST, [TOAST](#)

とユーザ定義型, [ユーザ定義の型](#)

対ラージオブジェクト, [はじめに](#)

列ごとの保管設定, [説明](#)

to_ascii, [文字列関数と演算子](#)
to_char, [データ型書式設定関数](#)

とロケール, [動作](#)

to_date, [データ型書式設定関数](#)
to_hex, [文字列関数と演算子](#)
to_number, [データ型書式設定関数](#)
to_timestamp, [データ型書式設定関数](#)
to_tsquery, [テキスト検索関数と演算子](#), [問合わせのパス](#)
to_tsvector, [テキスト検索関数と演算子](#), [文書のパス](#)
trace_locks設定パラメータ, [開発者向けのオプション](#)
trace_lock_oidmin設定パラメータ, [開発者向けのオプション](#)
trace_lock_table設定パラメータ, [開発者向けのオプション](#)
trace_lwlocks設定パラメータ, [開発者向けのオプション](#)
trace_notify設定パラメータ, [開発者向けのオプション](#)
trace_recovery_messages設定パラメータ, [開発者向けのオプション](#)
trace_sort設定パラメータ, [開発者向けのオプション](#)
trace_userlocks設定パラメータ, [開発者向けのオプション](#)
track_activities設定パラメータ, [問い合わせおよびインデックスに関する統計情報コレクタ](#)
track_activity_query_size設定パラメータ, [問い合わせおよびインデックスに関する統計情報コレクタ](#)
track_counts設定パラメータ, [問い合わせおよびインデックスに関する統計情報コレクタ](#)
track_functions設定パラメータ, [問い合わせおよびインデックスに関する統計情報コレクタ](#)
track_io_timing 設定パラメータ, [問い合わせおよびインデックスに関する統計情報コレクタ](#)
transaction_timestamp, [日付/時刻関数と演算子](#)
transform_null_equals設定パラメータ, [プラットフォームとクライアント互換性](#)
translate, [文字列関数と演算子](#)
trigger, [疑似データ型](#)
triggered_change_notification, [tcn](#)
trigger_fileリカバリ用パラメータ, [スタンバイサーバの設定](#)
trim, [文字列関数と演算子](#), [バイナリ文字列関数と演算子](#)
Tru64 UNIX

共有ライブラリ, [動的にロードされる関数のコンパイルとリンク](#)

trunc, [算術関数と演算子](#), [ネットワークアドレス関数と演算子](#)
TRUNCATE, [TRUNCATE](#)
tsearch2, [tsearch2](#)
tsquery (データ型), [tsquery](#)
tsvector concatenation, [文書の操作](#)
tsvector_update_trigger, [テキスト検索関数と演算子](#)
tsvector_update_trigger_column, [テキスト検索関数と演算子](#)
tsvector (データ型), [tsvector](#)
ts_debug, [テキスト検索関数と演算子](#), [設定のテスト](#)
ts_headline, [テキスト検索関数と演算子](#), [結果の強調](#)

ts_lexize, [テキスト検索関数と演算子](#), [辞書のテスト](#)
ts_parse, [テキスト検索関数と演算子](#), [パーサのテスト](#)
ts_rank, [テキスト検索関数と演算子](#), [検索結果のランキング](#)
ts_rank_cd, [テキスト検索関数と演算子](#), [検索結果のランキング](#)
ts_rewrite, [テキスト検索関数と演算子](#), [問い合わせの書き換え](#)
ts_stat, [テキスト検索関数と演算子](#), [文書の統計情報の収集](#)
ts_token_type, [テキスト検索関数と演算子](#), [パーサのテスト](#)
txid_current, [システム情報関数](#)
txid_current_snapshot, [システム情報関数](#)
txid_snapshot_xip, [システム情報関数](#)
txid_snapshot_xmax, [システム情報関数](#)
txid_snapshot_xmin, [システム情報関数](#)
txid_visible_in_snapshot, [システム情報関数](#)

U

UESCAPE, [識別子とキーワード](#), [Unicodeエスケープがある文字列定数](#)
unaccent, [unaccent](#), [関数](#)
Unicodeエスケープ

識別子中, [識別子とキーワード](#)

文字列定数中, [Unicodeエスケープがある文字列定数](#)

UNION, [問い合わせの結合](#)

結果型の決定, [UNION、CASEおよび関連する構文](#)

UnixWare

IPC設定, [共有メモリとセマフォ](#)

共有ライブラリ, [動的にロードされる関数のコンパイルとリンク](#)

上へのインストール, [SCO OpenServer と SCO UnixWare](#)

unix_socket_directory設定パラメータ, [接続設定](#)
unix_socket_group設定パラメータ, [接続設定](#)
unix_socket_permissions設定パラメータ, [接続設定](#)
Unixドメインソケット, [パラメータキーワード](#)
UNLISTEN, [UNLISTEN](#)
unnest, [配列関数と演算子](#)
UPDATE, [更新](#), [データの更新](#), [UPDATE](#)
update_process_title設定パラメータ, [問い合わせおよびインデックスに関する統計情報コレクタ](#)
upper, [文字列関数と演算子](#), [範囲関数と演算子](#)

とロケール, [動作](#)

upper_inc, [範囲関数と演算子](#)
upper_inf, [範囲関数と演算子](#)
URI, [接続文字列](#)
user, [システム情報関数](#)
UUID, [UUID型](#), [インストール手順](#)
uuid-oss, [uuid-oss](#)

V

VACUUM, [VACUUM](#)
vacuumdb, [vacuumdb](#)
vacuumlo, [vacuumlo](#)
vacuum_cost_delay設定パラメータ, [コストに基づくVacuum遅延](#)
vacuum_cost_limit設定パラメータ, [コストに基づくVacuum遅延](#)
vacuum_cost_page_dirty設定パラメータ, [コストに基づくVacuum遅延](#)
vacuum_cost_page_hit設定パラメータ, [コストに基づくVacuum遅延](#)
vacuum_cost_page_miss設定パラメータ, [コストに基づくVacuum遅延](#)

vacuum_defer_cleanup_age 設定パラメータ, [マスターサーバ](#)
vacuum_freeze_min_age 設定パラメータ, [文の動作](#)
vacuum_freeze_table_age 設定パラメータ, [文の動作](#)
VALUES, [VALUESリスト](#), [VALUES](#)

結果型の決定, [UNION、CASEおよび関連する構文](#)

varchar, [文字型](#)
variadic関数, [可変長引数を取るSQL関数](#)
var_pop, [集約関数](#)
var_samp, [集約関数](#)
version, [データベースへのアクセス](#), [システム情報関数](#)
VM

参照 可視性マップ

void, [疑似データ型](#)
VOLATILE, [関数の変動性分類](#)

W

WAL, [信頼性とログ先行書き込み](#)
wal_block_size 設定パラメータ, [設定済みのオプション](#)
wal_buffers 設定パラメータ, [諸設定](#)
wal_debug 設定パラメータ, [開発者向けのオプション](#)
wal_keep_segments 設定パラメータ, [送出サーバ群](#)
wal_level 設定パラメータ, [諸設定](#)
wal_receiver_status_interval 設定パラメータ, [スタンバイサーバ](#)
wal_segment_size 設定パラメータ, [設定済みのオプション](#)
wal_sync_method 設定パラメータ, [諸設定](#)
wal_writer_delay 設定パラメータ, [諸設定](#)
WHERE, [WHERE句](#)
WHILE

におけるPL/pgSQL, [WHILE](#)

width, [幾何関数と演算子](#)
width_bucket, [算術関数と演算子](#)
WITH

in SELECT, [WITH問い合わせ（共通テーブル式）](#)

SELECTにおける, [SELECT](#)

work_mem 設定パラメータ, [メモリ](#)

X

xid, [オブジェクト識別子データ型](#)
xmax, [システム列](#)
xmin, [システム列](#)
XML, [XML型](#)
XML option, [XML値の作成](#)
xml2, [xml2](#)
xmlagg, [xmlagg](#), [集約関数](#)
xmlbinary 設定パラメータ, [文の動作](#)
xmlcomment, [xmlcomment](#)
xmlconcat, [xmlconcat](#)
xmlelement, [xmlelement](#)
XMLEXISTS, [XMLEXISTS](#)
xmlforest, [xmlforest](#)
xmloption 設定パラメータ, [文の動作](#)
xmlparse, [XML値の作成](#)
xmlpi, [xmlpi](#)
xmlroot, [xmlroot](#)

xmlserialize, [XML値の作成](#)
xml_is_well_formed, [xml_is_well_formed](#)
xml_is_well_formed_content, [xml_is_well_formed](#)
xml_is_well_formed_document, [xml_is_well_formed](#)
XMLエクスポート, [XMLにテーブルをマップ](#)
XMLオプション, [文の動作](#)
XPath, [XMLの処理](#)
xpath_exists, [XMLの処理](#)

Y

yacc, [必要条件](#)

Z

zero_damaged_pages設定パラメータ, [開発者向けのオプション](#)
zlib, [必要条件](#), [インストール手順](#)

[前のページ](#)
参考文献

[ホーム](#)