# 主要RDB比較

|  |  |  |
| --- | --- | --- |
|  | SQL Server | PostgreSQL |
| タイプ | RDBMS | ORDBMS |
| サポート | あり | ほぼなし |
| プロセス管理 | マルチスレッド | マルチプロセス |
| ライセンス | Microsoftの製品 | BSD |
| 開発言語 | poswer shell, F#なども対応（MS系寄り） |  |
| 特徴 | Windows製品と親和  アドオンサービス豊富 | 幅広い用途 |
| メモリ管理 | ローカルメモリのみ | プロセスとSQL実行 |
| 注意点 | プライマリーキー列は後から変更不可。 | 「バキューム処理」を行わないとパフォーマンス低下 |
| 備考 | 原則Windows Serverでのみ動作。 | Oracleに匹敵する機能を持つと言われている |

|  |  |  |
| --- | --- | --- |
|  | MariaDB(MySQL) | sqlite |
| タイプ | RDBMS | 軽量RDBMS |
| サポート | MySQLの有料版なら | なし |
| プロセス管理 | マルチスレッド | 単一トランザクションでファイルをロック |
| ライセンス | GPLv2(MariaDB)  商用/GPLv2(MySQL) | パブリックドメイン |
| 開発言語 | .NET非対応？ |  |
| 特徴 | シンプルかつ軽量  対アプリ向き | ソフトウェアに内蔵。外から接続不可。  データベース毎にテキストファイル１つ  \*.sqlite3 |
| メモリ管理 | 細分化 |  |
| 注意点 |  |  |
| 備考 | 一番人気  堅牢性やや低い？ | 軽量だが高速  プロシージャ、Checkが使えない |

カーソル、ファンクション、マテリアライズドビュー、ラージオブジェクト、パーティション

・インデックス、トリガー、ビュー　はどのデータベースにも存在。

|  |  |  |  |
| --- | --- | --- | --- |
|  | ストアド  プロシージャ |  | 備考 |
| Postgresql | 〇 |  |  |
| SQLServer |  |  | パーティション作成、難。 |
| MySQL |  |  |  |
| sqlite | × |  |  |

# 共通項目

## 基本的な概念

## データモデル

### モデル概要

概念データモデル

現実世界を概念的に表現したモデル。

論理データモデル

データ的に表現されたデータ。

論理設計の最終目的はER図を作成する事。

（右図参照）

階層型データモデル

ツリー構造で表現され、ある１つのEntityが他の複数のEntityと親子関係を持つ。

リレーショナルモデル

分離性が高く，データ構造に修正が入ってもプログラムへの影響が最小限に抑えられる

ネットワークデータモデル

網の目状の構造で表現され，1つの子Entityに対して，複数の親Entityを関連づけることができる

各データが多対多に結びつくことでデータ構造を表現する

データベース設計

論理データモデルに変換する作業のこと

カーディナリティ(cardinality)

1つの列に格納されているデータの種類がどのくらいあるのかを示す値

例えば性別なら=2になり，idの場合は=レコード数になる。

### ER図

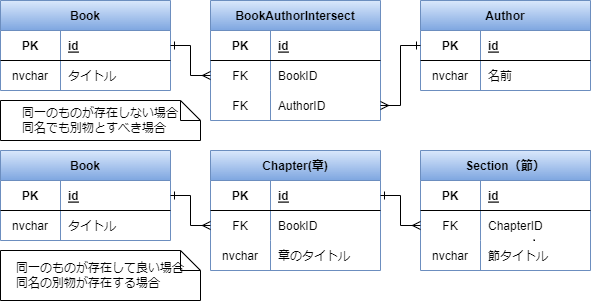
Entity Relationship

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 記号 | 概要 | 備考 | 記号 | 概要 | 備考 |
|  | one（１） |  |  | 0または1 |  |
|  | Many（多） |  |  | 1以上 |  |
|  | 1to1（1対1） |  |  | 0以上 |  |

補足：多の方がFOREIGNKEY制約を設け、参照される側になる

比較：Data Flow Diagram - 機能と入出力データとの関係

### 多対多と１対多の多層の違い



ポイントは末端のエンティティが**複数存在して問題ないか**どうか。

唯一無二の場合 ⇒多対多

重複が認められる ⇒ １対多の多層

## スキーマ

データの構造、性質、データベースを操作するときのルールや表現法などを定義したもの。

データベースの設計図のようなもの。

通常、RDBのように多機能なDBが持つ機能で、NoSQLのようにシンプルなDBはスキーマが無い事が多い。

3層スキーマアーキテクチャ

クライアントサーバシステムを以下の３階層に分割して構築するシステム形態。

・外部スキーマ （SQLの）viewなど

・概念スキーマ テーブル定義など

・内部スキーマ データの物理（ハード）的な格納定義 データ層

用途：

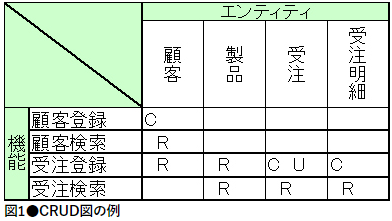
・アクセス件をスキーマごとに管理する

・１つのデータベースを多数のユーザーが干渉無しに使用できるようにする

逆にスキーマレスの特徴

・レコードごとにデータ構造が違うデータを持つことができる（特にドキュメント型NoSQL）

## CRUDマトリックス



どの機能でどのデータをどのように（CRUD）編集できるかを表にしたもの。

補足：CRUDはDB関連でよく使われる言葉で、Create、Read、Update、Deleteの意。

## charとvarchar

対応表

|  |  |  |
| --- | --- | --- |
|  | char [n] | varchar [n] |
| 名称 | 空白埋め固定長 | 上限付き可変長 |
| 対応文字数 | 0~255文字 | 0~65535 |
| 指定文字数未満 | 空白文字で埋められる | byte数を変えてストレージに保存 |
| 指定文字数超 | 切り捨てられる | エラーが発生 |
| 取出し時の末尾空白 | 取り除かれる | 取り除かれる |
| 注意点 | ・指定文字数未満の場合．ストレージを僅かに圧迫する | ・格納時にサイズ上限確認が行われるため，極僅かな負荷になる。 |
| 用途 | 郵便番号のように桁数が変わらないID。  ⇒ 変わった事がある事実をどう捉えるか。 | 名前、表題など文字数が定まらない場合。 |
| 備考 | ・原始的な型であり，現在においてはほとんど利点がない。 | ・textという型とあまり違いが無い  ・迷ったらvarchar（文字数指定無）で |

補足：[n]は「文字数」を示す場合（PostgreSQL，MySQL）もあれば，バイト数を表す場合（Oracle）もあり注意が必要。

## text と varchar

・varchar では文字数を制限することができるが、文字数制限くらいでは壊れたデータの投入を抑止するのは不十分。

## varcharとnvarchar

一番の違いは文字数の扱われ方で：

varchar 全角は1文字で2文字として扱う？DBによって違うらしい。全角非推奨。

nvarchar Unicode（全角）文字で使われる。文字数に関しては全角半角で違いなし。

nchar nvarcharと同様だが文字数固定。あまり使われない？

## ACID特性

下のそれぞれの頭文字をとってACID特性と呼ぶ。

原子性（atomicity）

複数の処理を１つにまとめて，それらの処理がすべて実行されたか，またはまったく実行されていないかのどちらかの結果になる事。

一貫性（consistency）

トランザクションの開始および終了時点で，業務として規定された整合性を満たす事。

独立性（isolation）

作業中のトランザクションによる更新は，確定するまで他のトランザクションから不可視。

永続性（durability）

確定したトランザクションの結果はデータベースに永続的に保存される。

## CAP定理

CAP定理とは?

CAP定理\*とは、 分散システムにおいてCAPのうち2つしか提供できないことを示した定理。

|  |  |
| --- | --- |
|  | 概要 |
| Consistency  一貫性 | 別々のノード上に接続したどのクライアントにおいても データの一貫性がある性質。 |
| Availability  可用性 | 1つ以上のノードがダウンしたとしても システムとして正常に動作し続ける性質。 |
| Partition Tolerance  分断耐性 | ノード間の通信障害 (接続が切れたり、 遅延したりした状態)が 起きたとしても、システムとして正常に動作し続ける性質。 |

\* Eric A. Brewer教授によって提唱されたため、 ブリュワーの定理 (Brewer's Theorem) とも言う。

## Raftアルゴリズム

安全な State Machine Replication (SMR) を実装するための分散合意アルゴリズム。

分散システムにおいて一貫性 (分散トランザクション) と可用性 (障害耐性) を実装するための基本的な部品として使用することができる。

一貫性 (Consistency) に振るのであればトランザクション (強い一貫性) を導入し、分断耐性 (Partition Tolerance) に振るのであれば結果整合性を導入

参考：<https://qiita.com/torao@github/items/5e2c0b7b0ea59b475cce>

## 別DBの結合

### 概要

### 同一エンジン内の場合

普通にできる。

補足：同一サーバー内の同一エンジンの別DBへのアクセス

SELECT \*

FROM db1.dbo.table1

JOIN db2.dbo.table1

ON db1.dbo.table1.col1 = db2.dbo.table1.col1

### 別サーバーの場合

#### SQL Serverの場合

リンクサーバーを設定する

リンクサーバー名称とサーバーのIPアドレスを設定

（右図はイメージ）

[ SQL（例 ）]

SELECT \*

FROM db1.dbo.table1

JOIN [linked\_server\_name].db2.dbo.table1

ON db1.dbo.table1.col1 = db2.dbo.table1.col1

注意点

通信レイテンシなどのオーバーヘッドが発生する

#### 一般的なRDB製品の場合

**基本的には無理**と考える。

（以下、編集中）

但し、ODBCを使用すると一応できる（らしい）

例としてMySQLとPostgreSQLのテーブルをJOINする場合

MySQLとPostgreSQLそれぞれのODBCドライバをインストール。

ODBCデータソースを設定する際にそれぞれにデータソース名を付ける

例：mysql、postgresql

[ SQL（例 ）]

SELECT \*

FROM mysql.dbo.table1 mydb

JOIN postgresql.db2.dbo.table1 psql

ON mydb.table1.col1 = psql.table1.col1

## データベース管理システム（DBMS）

ユーザーとデータベースの仲介をするシステムの事

DBMSの主な機能

|  |  |
| --- | --- |
| 項目 | 詳細 |
| データ管理 | データベースで扱うデータ形式を管理し，データの追加・削除・更新・検索などの**操作**を行う |
| 整合性の維持 | 不正なデータや重複データをチェックし，**データの整合性**を保つ  値の入力範囲など。 |
| トランザクション管理 | トランザクション実行中の一連の処理を途中で停止しないよう，データベースの**操作の一貫性を保証**する。 |
| 同時実行制御 | 複数のユーザーが同時に実行しても矛盾が生じないようにする  データベース上の，不整合の発生を防ぐ |
| セキュリティ機能 | データのアクセス権限を管理し，不正アクセスから保護する  暗号化する事により，高い**安全性**を実現する |
| 障害回復機能 | 障害発生時に復旧を行い障害発生前の状態に回復する  バックアップやリカバリ機能によりデータベースを復元する。  トランザクション障害，システム障害，ハードウェアなどの記憶媒体の障害から復旧する |

## レプリケーション

|  |  |  |
| --- | --- | --- |
|  |  |  |
| MySQL | Master-Slaveレプリケーション |  |
| PostgreSQL | Streamingレプリケーション |  |
| SQLServer |  |  |

## 共通用語

ストアドプロシージャ

プログラム的な処理記述

リードレプリカ

マスターから複製された読み込み専用のレプリカ。主に負荷分散の為に用いられる。

RDBでは一般的だが、NoSQL等では使われる事は少ないらしい。

# SQL

## 標準SQL

### 概要

SQL 2011 標準に準拠

ISO/IEC 9075:2016

### 主な機能

・DOMAIN（定義域）[SQL92]

・UNION（論理和集合）[SQL99]

・TRIGGER[SQL99]

・ストアドファンクション[SQL99]

・ラージオブジェクト [SQL99]

・CURSOR（SELECT結果の特定行の取り出し）

・PREPARE（コンパイル化＋関数化したVIEW）

・WITH 句と SQL 関数を使用したコンポーザビリティ[SQL2011]

・SELECT リストと WHERE 句内のサブクエリ[SQL2011]

・相関サブクエリ[SQL2011]

・ARRAY データ型と STRUCT データ型[SQL2011]

### 命名規則

|  |  |  |
| --- | --- | --- |
| 対象 | 内容 | 例 |
| テーブル名 | 小文字アルファベットに統一  基本は複数形の英単語を使う | table  categories |
| 多対多のテーブル名 | 複数形 ＋ \_ ＋ 複数形 | users\_categories |
| 複数単語 | スネークケースを利用 | table\_name |
| ユニーク制約 | uq\_<テーブル名>\_NN | uq\_users\_01 |

その他

ハイフンは原則禁止。アンダーバー(\_)を用いる。

名前の最初に数字を使わない。 例 2019\_salary×

data・info など汎用過ぎる単語は使わない

性別に関して。ISO 5218という規格がある。<https://ja.wikipedia.org/wiki/ISO_5218>

UNIQUE\_\_TOGETHER？？

参考：<https://qiita.com/genzouw/items/35022fa96c120e67c637>

### SQL書き順

SELECT

FROM

JOIN

ON

WHERE

GROUPBY

HAVING

ORDERBY

LIMIT

### 具体例

備忘録として

JOINの例

SELECT \* FROM gyoumu\_dailyreport

INNER JOIN gyoumu\_project ON gyoumu\_dailyreport.project\_number = gyoumu\_project.project\_number

WHERE gyoumu\_project.asset\_number IN ('EZA0007', 'EZJ4820080')

countとbetween

SELECT COUNT(\*)

FROM [DATA\_COLLECTION].[dbo].[PRODUCTION\_LOG]

WHERE ライン名='CHボデー' and 日時 BETWEEN '2022/07/29 09:00:00' AND '2022/07/29 10:00:00';

distinctのcount

select count(distinct 品名) FROM [DATA\_COLLECTION].[dbo].[PRODUCTION\_LOG]

where ライン名='CHボデー' and CONVERT(date,日時) = '2022-08-01'

全レコード削除

DELETE FROM table\_name;

日付比較

SELECT TOP \*

FROM [gyoumu\_mng].[dbo].[gyoumu\_project]

where start\_date > '2022-01-01'

LIKE

SELECT \*

FROM [DATA\_COLLECTION].[dbo].[PRODUCTION\_LOG]

Where PRODUCTION\_LOG.品名 like 'CH%'

INSERT

INSERT INTO gyoumu\_mng.dbo.gyoumu\_asset (asset\_number, asset\_name) VALUES ('G229999','TEST')

OUTER JOIN

SELECT \* FROM gyoumu\_purchaseitem

LEFT OUTER JOIN "gyoumu\_purchaseinfo" ON ("gyoumu\_purchaseitem"."purchase\_info\_id" = "gyoumu\_purchaseinfo"."id")

WHERE project\_number='10J0039';

GroupBy

SELECT 品名, MIN(日時), MAX(日時)

FROM [DATA\_COLLECTION].[dbo].[PRODUCTION\_LOG]

WHERE ライン名='CHボデー' and CAST(日時 AS DATE) = CAST(GETDATE() AS DATE)

GROUP BY 品名 ORDER BY MIN(日時)

## 主キーについて

### 概要

|  |  |  |
| --- | --- | --- |
|  |  |  |
| 主キー | Primary Key |  |
| 疑似キー | Pseudo Key | テーブル中に主キーとなり得る列が存在しない時に、便宜的に行を一意に特定するために付与するキー。自然キーが存在しない場合の苦肉の策。  代理キー（Surrogate Key）とも呼ばれる。 |
| 自然キー |  | テーブル中の主キーとなる得る列。 |

### 外部キー

作成・更新・削除するときに一緒に親子テーブル同時に行うものを「整合性が強い」と呼ぶ。

外部キーは整合性が強い時にのみ使用するべき、という意見がある。

また、「弱い整合性」を「結果整合性」と呼ぶ。

２つ以上外部キーを使う際は、それぞれの制約が保証されない？？

### 複合主キー制約

「主キーに対するあらゆる操作」が複雑になる。

複合ユニーク＋疑似キーの方が管理しやすい場合が多い（らしい）。

## GROUP BYとDISTINCT

DISTINCTは実行した結果のテーブルから、重複している行を削除した結果を出す。

GROUP BY は実行した結果をグループ化して更に集計する際に用いる。

## SQLアンチパターンより

### まとめ

Jaywalk（信号無視）

本来１つずつ列に格納すべき複数のデータを、配列的に１つの列に入れない。

多対多（たたいた）のリレーションシップの時にやってしまいがちだが、多対多の時はintersection tableを利用する。

補足：例えば(x,y,z)の３次元座標などの場合は問題ない。

Naïve Tree（隣接リスト）

ツリー構造のデータを扱う方法。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 設計 | テーブル数 | 子へのクエリ実行 | ツリーへのクエリ実行 | 挿入 | 削除 | 参照整合性維持 |
| 隣接リスト | 1 | 簡単 | 難しい | 簡単 | 簡単 | 可能 |
| 再帰クエリ | 1 | 簡単 | 簡単 | 簡単 | 簡単 | 可能 |
| 経路列挙 | 1 | 簡単 | 簡単 | 簡単 | 簡単 | 不可 |
| 入れ子集合 | 1 | 難しい | 難しい | 難しい | 難しい | 不可 |
| 閉包テーブル | 2 | 簡単 | 簡単 | 簡単 | 簡単 | 可能 |

入れ子集合（改）とも言える「入れ子区間モデル」というのも存在するらしい

ID required

主キー名を「とりあえずid」するのは止める。「id」は疑似キーを連想させる。

主キー 社員番号、免許番号など、レコードを確実に特定できる値（Natural Key）の事。

疑似キー 主キーとなり得る列が存在しない時に、便宜的に付与するキー。

Keyless Entry

外部キーを使ったリレーションの場合、子の扱いがめんどくさい事がある。

それでも、外部キーを使用した時の利点の方が勝る。外部キーの扱い方を覚える。

Entity Attribute Value

動的に列の数が変化するようなテーブル設計が必要だからと言って、列で持つべき情報を行に展開しない。

継承的発想の方が無難。

Single Table 継承 限定的に使用される列を用意しておく。使わない場合はNull。

具象テーブル継承 似たような列構造を持つテーブルを２つ以上作成。

Class Table継承 外部テーブルを用いた、オブジェクト指向の継承的発想。

半構造化データ

Polymorphic Associations

―対多 ⇔ ２つ以上のFKを持つテーブル ⇔ 多対―　という構成で、真ん中のテーブルで集計する際に片方は必ずnullになるような設計。

・左右両方のテーブルの間にIntersection Tableを挟む。Jaywalkと同じ種類の問題。

・共通の親テーブルを用い、そのテーブルで集計を行う。

[https://qiita.com/Natsumag/items/c9c91f6fa7a3af6dd1d1#](https://qiita.com/Natsumag/items/c9c91f6fa7a3af6dd1d1#第6章-polymorphic-associationspolymorphic-associationsポリモーフィック関連)第6章-polymorphic-associationspolymorphic-associationsポリモーフィック関連

Multi Column Attribute（複数列属性）

Metadata Trimble（メタデータ大増殖）

大体のDBはレコード数が増えるとパフォーマンスは落ちる。

だからといって、例えば年別にテーブルを作成したりしない事。

水平パーティション 　データベースの標準機能の「パーティション」

垂直パーティション　　外部テーブルを利用した継承的発想。

ラージオブジェクトやTEXT型で対応（非推奨）

Rounding Error（丸め誤差）

floatの精度は荒い。精度が必要なときはnumericかdecimalを使う。

31 flavors

CHECK、ENUMのメンテナンス性の悪さ。外部テーブルを使った方が良い。

phantom files（幻のファイル）

バイナリデータ対応で、VARCHARで内部ストレージへのパス文字を格納する事は止める。

ラージオブジェクトを使う。

Index shotgun

インデックスは必ずしもパフォーマンス向上とは成らない。目的を持って行う。

MENTORの原則。

<https://qiita.com/mizunokura/items/fbe8dbdca278764985df>

The Fear of Unknown

データベースに於けるNullの扱いを知る。

COALESCE、IS DISTINCT FROM

Ambiguous Group

単一値の原則(Single-Value Rule)を守る

Ramdom Selection

poor man's search engine

spaghetti query

implicit column

Readable Password

SQL Injection

Sedue Key Neat Freak（疑似キー潔癖症）

See No Evil

Diplomatic Immunity

Magic Beans

Sandcastle

### 連番型

どのDBシステムにおいても、トランザクションと無関係のところでカウントアップが行われ、ロールバックしてもその値が元に戻ることはない。その結果、複数のクライアントが同時に新番号を取得しても重複することが無い。（反対に言うと、SQL内で関数などを駆使して新番号を取得するべきでは無い）

|  |  |  |
| --- | --- | --- |
| DB | 機能の名前 | 備考 |
| MySQL | LAST\_INSERT\_ID |  |
| SQL Server | SCOPE\_IDENTITY |  |
| Oracle | .CURRVAL |  |
| PostgreSQL | serial |  |

### 再帰クエリ

Tree構造を扱うためのクエリ

WITHで仮想テーブルを作成した後に本命のクエリを発行する、という流れ。

例）

WITH recursive AS (

SELECT \* FROM tree WHERE id = 1

UNION ALL

SELECT tree.\* FROM tree, r WHERE tree.parent\_id = r.id

)

SELECT \* FROM recursive

ツリー構造の設計

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 設計 | テーブル数 | 子へのクエリ実行 | ツリーへのクエリ実行 | 挿入 | 削除 | 参照整合性維持 |
| 隣接リスト | 1 | 簡単 | 難しい | 簡単 | 簡単 | 可能 |
| 再帰クエリ | 1 | 簡単 | 簡単 | 簡単 | 簡単 | 可能 |
| 経路列挙 | 1 | 簡単 | 簡単 | 簡単 | 簡単 | 不可 |
| 入れ子集合 | 1 | 難しい | 難しい | 難しい | 難しい | 不可 |
| 閉包テーブル | 2 | 簡単 | 簡単 | 簡単 | 簡単 | 可能 |

入れ子集合（改）とも言える「入れ子区間モデル」というのも存在するらしい

### 拡張性の高いテーブル設計

シングルテーブル継承

１つのテーブルで、場合によって使う列、使わない列が混在しているテーブル。

issue\_type VARCHAR(20);

severity VARCHAR(20); -- issue\_typeがBugの時だけ使う

sponsor VARCHAR(20); -- issue\_typeがFeatureRequestの時だけ使う

クラステーブル継承

「基底クラス」にあたる列だけ定義しておき、「継承クラス」にあたる部分は1対1の外部キーを利用する方法。

# 比較的新しいものなど

## NewSQL

### 概要

リレーショナルデータベース（RDB）の整合性を保ちながら、NoSQLのスケーラビリティや柔軟性を持つデータベース。

従来のRDBは一貫性と安定性が強みだが、大量のデータを高速に処理する能力には限界がある。

NoSQLはスケーラビリティと柔軟性が強みですが、トランザクション整合性の維持が難しい。

その両方を達成を達成すべく、2020年頃から話題を集めつつある。

NewSQLデータベースの一覧

Amazon Aurora

Apache Trafodion（英語版）

Clustrix（英語版）

CockroachDB（英語版）

Couchbase Server

Google Spanner

NuoDB（英語版）

Pivotal Software（英語版） XD

SingleStore（英語版） 以前はMemSQLとして知られていた

TIBCO Software（英語版） Spaces

TiDB（英語版）

TokuDB（英語版）

TransLattice（英語版）Elastic Database

VoltDB

YugabyteDB（英語版）

## 時系列データベース

### 概要

時刻情報を主キーとしたデータベースで、時系列データ（Time Series Data）を扱う。

例えばIoTにおけるセンサーからデータは、ミリ秒単位で発生する高頻度のデータとなり得る。

そのため蓄積されると大容量のデータとなり、取り扱いが難しい。そういったデータを扱うのが得意なデータベース。

また、BIツールなどでの分析専用データとして使われる。

（分類としてはDWH；Data Warehouse になる？）

InfluxDB、Prometheus、Graphite、OpenTSDBなどがある

### 用語

#### メジャーメント

リレーショナルDBのテーブルに相当するもの。タグ、フィールド、タイムスタンプをカラムに持つ二次元の表をイメージするとよい。メジャーメントにはユーザが文字列の名前をつけることができる

### 資料

#### InfluxDBの概要がまとまった記事

<https://qiita.com/dedepi92/items/eaf9f012cc79f784f2d5>

#### 時系列DBの人気情報

<https://db-engines.com/en/ranking/time+series+dbms>

### InfluxDB

#### 概要

時系列データに特化したデータベース

Go言語で書かれている（ここが少し厄介？）

「時系列データのための高機能データストア」という位置づけ。

IoT センサー・データおよびリアルタイム解析を含む大きな量のタイムスタンプされたデータに関する利用シーンに適す。

小規模データや列が多い複雑な構造を扱うのは苦手。

なお、InfluxDBだけでデータの可視化（Chronograf）までできるが、InfluxDBはGrafana（BIツール）と共に使われる事が多い。

ライセンス

シングル構成 MIT ライセンス

クラスタ構成 製品版

詳細

InfluxQL という SQL ライクなクエリ言語を提供

動作OSはLinuxのみなので注意

#### チュートリアル

インストール

wget -qO- https://repos.influxdata.com/influxdb.key | sudo apt-key add -

source /etc/os-release

echo "deb https://repos.influxdata.com/debian $(lsb\_release -cs) stable" | sudo tee /etc/apt/sources.list.d/influxdb.list

sudo apt install –y influxdb

### Prometheus

プロメテウス

## NoSQL

### 概要

NoSQL = Not only SQL

構造がシンプルである故に高速処理が可能で、**ビッグデータなどの分野で利用**されることが多い。

結合が必要なデータ、トランザクションによる一貫性確保などは苦手。

・書き込み負荷の分散が可能

具体的製品

Amazon DynamoDB

OSSの製品

Redis、MongoDB、Cassandra、HBase、Neo4J、InfluxDB

4種類のデータモデル

|  |  |  |  |
| --- | --- | --- | --- |
| 種類 | 概要 | 備考 | 製品 |
| キーバリュー型 | 文字通り、KeyとValueの対 | シンプルな構造  アプリのキャッシュとして使う事も。 | Redis、DynamoDB |
| 列指向型  （ワイドカラムストア） | 通常のRDBが「行指向」に行う処理を「列方向」に行う。 | 列を限定した検索に向く  ログデータの分析など | Cassandra、HBase、BigTable、Redshift |
| ドキュメント指向型 | JSONやXML形式で保存 | 別々のデータ構造や入れ子式のデータに対応できる | MongoDB |
| グラフ型 |  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| id | 1 | 2 | 3 | 4 |
| 商品名 | ノート | 定規 | ペン | 消しゴム |
| 価格 | 100 | 250 | 300 | 50 |
| サイズ | 40 | 32 | 21 | 18 |

列指向に関して：

これだけ理解が少し難しいので補足。

RDB的に考えると、全レコードの特定列の値のみを全部取得 > そのデータをzipさせて処理　といったイメージ。

右の図で行くと 商品名、価格のデータを全て抜き出し、商品名⇒index⇒価格 の順で素早く抜き取るイメージ。

メモ：

・実は世間で思われているような高速用途でのメリットはほとんど無いらしい

・それよりもSQLにないデータ構造の扱いやすさで使用を考えるべき。

### Redis

DBというよりはpythonのpickleのような感じで配列やマップをサーバー上でアトミックに安全に読み書きできる共有メモリ構造

### Elasticsearch

概要

ドキュメント指向型の NoSQL

RESTfulに対応している（らしい）

Apache Lucene基盤の分散処理マルチテナント対応検索エンジン

github：https://github.com/elastic/elasticsearch

### MongoDB

#### 概要

ドキュメント指向型の NoSQL

ライセンス

AGPL v3.0

製品ライセンス

適している用途

見出しがあって明細があってメタデータがあってタグの配列があって、といったような複雑なデータの格納

シンプルな構造のBig Data の格納

コンテンツの管理、配信

#### MongoDB Compass

MongoDBの管理ツール

### 資料

#### NOSQLの関する面白い考察

<https://jp.quora.com/MongoDB-no-sama-na-NoSQL-ni-ikioi-ga-aru-no-ha-naze-desu-ka-SQL-to-kurabe-te-donna-riten-ya-ketten-ga-arima-suka>

# 応用的な項目

## 別用途でDBを使う際のベストプラクティス

### 概要

システム運用で使用しているRDBサーバーがあり、例えばBIでの分析用途などの目的でも同一のデータを使用したい場合のメモ

第一に：

システム運用で使用しているDBはそれ以外の目的で使うべきではない

理由

・パフォーマンス低下 分析クエリがリソースを消費する為

・セキュリティの観点 分析アプリを使用するユーザーに本番データにアクセスしてほしくない

・障害

### 複製の方法

|  |  |
| --- | --- |
| 方法 | 概要 |
| レプリケーション | 単純にマスターサーバーのコピーを作成する。  当然別々のデータソースを結合する事ができない。 |
| ETL；  Extract-Transform-Load | 必要なデータを抽出して、別のフォーマットに変換、別DBに転送するソフト。特にスクラッチ開発するものを差す場合が多い。 |
| データパイプライン | 現在では特に、半自動型でETLと同様のデータ転送を行うソフトウェアを差す。 |

### ETL

大きく分けて以下3ステップに分類されます。

データ抽出処理：API連携やDB接続などを用いて、ETL処理サーバにデータを持ってくる

データ変換処理：持ってきたデータを、処理可能な形式に変換し、変換・加工などを行う

データ出力処理：変換処理したデータを、目的の形式に変換後、目的地に転送する

### データパイプライン

CSV→JSONやセンサー情報→DBなどを行う？

具体的製品

Kafka(Apache)

NiFi（Apache）

Kinesis（AWS）

Pub/Sub（GCP）

Event Hubs（Azure）

Apache NiFi

Apache PLC4X

データフロー

概要

製品

### インメモリデータベース

ストレージでは無く、メモリ上にデータをほじさせるデータベース。

処理の高速化を目的としている。

具体的製品

TimesTen(Oracle)

HANA(SAP)

SQL Server In-Memory OLTP(MS)

Ignite(Apache)

Redis(OSS)

Memcached(OSS)

### 用語

ストリーム処理

バッチ処理せず、リアルタイムのデータをそのまま渡す処理

## DB更新時にメソッドを実行

PostgreSQLの場合はPL/Pythonを用いる事ができる

# データベース種類

## リレーショナルデータベース（RDB）

・表構造で表現するため，シンプルで分かり易い

・操作対象のデータとデータの操作を分けているため，データへの依存がない

・分離性が高く，データ構造に修正が入ってもプログラムへの影響が最小限に抑えられる

・データベース言語であるSQLを用いてデータを操作することができる

・独立した複数のテーブルを関連付けることで，柔軟に操作を行うことができる

・表と表を関連付ける共通の情報を持つ列を外部キーという

・個々のレコードを1件分のデータとし，データを構成する個々の項目を列として扱う

リレーショナルモデル

ものの集まりやその要素を，集合とよばれる数学的対象として扱う数学理論，集合論をベースにしている。

リレーションとは各テーブルの関連付けの事。

・各リレーションの関連付けによってデータ構造を表現する

・1970年，エドガー.F.コッドによって提案され，数学の集合論をベースにしている

## データレイク

## データウェアハウス（DWH）

Data Ware House

データ分析に特化したデータベース。

・1つのクエリ処理を同時並行で行う

・サブジェクト指向

・列ベースでデータを管理しているため、検索処理が速い。

製品

Amazon Redshift、Azure、Google BigQuery、Snowflakeなど

HUE、

## データマート

１つの目的（例えばレポート）向きに抜粋されたデータ群。例えば、企業内データというビッグデータから、営業課の情報だけ抜き出しまとめたもの、など。

BI内のツールとして実装される場合も多い。

## 時系列データベース

製品：InfluxDB、GridDB、Prometheus

## サーバーレスDB

Amazon

Aurora Serverless

Google

Cloud Datastore

Cloud SQL(容量上限が存在するので、定義に反するが容量が大きいので記載)

Microsoft

Azure Cosmos DB

## 資料

OSSDBについて、非常によくまとまっている

<https://qiita.com/kanegoon/items/5ade64336d389feea4bc>

SurrealDB

Rust製データベース

<https://qiita.com/silane1001/items/795c3539675e588c2c4d>

## その他

# postgresql

## 概要

オブジェクト・リレーショナル・データベース

オープンソフトウェア

BSDライセンスをベースにしたPostgreSQLライセンス。

使用，変更，複製，修正，商用が可能。サブライセンスが認められる。

著作権とライセンス条文，無保証である事を記載すれば再配布も可能。

GPLのようなソース公開義務が無い。

（比較）MITはトレードマークの使用を禁止。GPLはソースコードの公開が義務。

クライアント／サーバー型

複数台のコンピューターをサーバーとクライアントに役割を分けて運用する仕組み

（対）ピアツーピア

追記型

フィールド（列）変更が大変なので最初によく考える

そのほか

・TCP/IPまたはUNIXドメインによる通信が可能。

・PostgreSQL自体をライブラリとして利用することはできない。

・クライアントからの接続ごとにプロセスが生成される。

・サーバーでデータを管理し，クライアントから処理を要求する仕組み。

・標準SQLの大部分をサポート

・Linux，UNIX系OS，Windows，様々なOSで動作

・日本語を含む，様々な言語のエンコーディング(マルチバイト文字)に対応している

・ユーザー定義により，機能の拡張ができる

・スクリプトの記述を簡便化したり、様々なタスクを自動化することが可能？？？

使い方

psql -h ホスト名 -p ポート番号 -U ユーザー名 -d データベース名 で対話式端末プログラムを起動。

例

psql # 引数無しで起動する場合

psql -U *user\_name* -d *db\_name* # ログインユーザーやデータベースを指定する場合

・ロール名を省略すると，現在のOS上のログイン名が使用される。

オプション

-f fileName SQLコマンドが記載されたテキストファイルを実行する --file

注意点

デフォルトがpeer認証（※）になっている為，以下のどちらかが必要：

方法１：suコマンドなどでpostgresqlのユーザー名と同一のものに切り替える。

方法２：ログイン方法をpeer認証からパスワード認証に切り替える。

※（参考）[用語(postgre sql)](#_用語(postgre_sql))

postgresql.serviceを停止してからコンピューターの電源を落とさないと，次回起動時にpsqlでログインできない。systemctl restartすると正常に戻る。

## チュートリアル

### インストール

Linux

普通にパッケージマネージャでインストール

terminal:

sudo apt install postgresql # Debian系

sudo apt install postgresql-contrib # 拡張モジュール，便利ツールのインストール。（任意）

psql --version # バージョンの確認

sudo apt install postgresql-client-12 # クライアント側

# 補足 client-commonもインストールされる。-commonだけではアプリは動かない。

補足：

ソースコードのビルドでインストールする方法もある（らしい）

PostgreSQL 11.5以降？、OSでロケール関連の環境変数を設定していないと警告文が出る

~/.bashrc などで設定するのが良い

export TZ=Asia/Tokyo

export LANG=ja\_JP.UTF-8

export LANGUAGE=ja\_JP:ja

export LC\_ALL=ja\_JP.UTF-8

情報元：<https://yamap55.hatenablog.com/entry/2021/03/13/145223>（Dockerの際の設定なので、注意）

setlocale ja\_JP.UTF-8がエラー

対策：<https://qiita.com/libra_lt/items/feb65b1742d9508740a9>

### データベースクラスタの作り直し

初心者は読み飛ばして良い。一度、このチュートリアルが終わった人が試す項目。

データベースクラスタを削除

$ sudo service postgresql stop # psqlの停止；systemctlだと停止できない事があった。

=# SHOW data\_directory; # 自動で作られたデータベータクラスタのディレクトリパスが表示される。

> /var/lib/postgresql/11/main

$ ps axf | grep -E "postmaster|postgres" # もしくはこの方法で確認できる

$ sudo rm -rf /var/lib/postgresql/9.6/main # 削除

データベースクラスタを作成

残っている /var/lib/postgresql/11/ディレクトリの所有者はpostgresである事に注意

$ export PGADTA=/var/lib/postgresql/11/main # initdbに必要な環境変数を設定

$ sudo -E -u postgres /usr/lib/postgresql/11/bin/initd --no-locale -E UTF-8 # localeなしエンコーディングUTF-8で作成

$ sudo service postgresql start # psqlの再開

補足：sudo -E 現在のユーザーの環境変数を引き継いでsudoを実行する

環境変数

以下の環境変数はほぼ必須なので設定を行う。

様々な理由から、~/.bash\_profile(一般ユーザー)や /etc/profile（root用）に記述する。（下の方に追記）

export PGDATA=/var/lib/postgresql/ver/main # データベースクラスタのディレクトリ

export PATH=$PATH:/usr/lib/postgresql/ver/bin/　　 # パスを通す

補足：以下の理由で ~/.bash\_profileと/etc/profileどちらを使うかは状況に依る

・postgresはシステムユーザーなので、~postgres/.bash\_profileは存在しないし、機能しない。

・postgresはrootに属していない為、/etc/profileは直接的には意味がない。

→ 結局rootか一般ユーザーからのsudo -u かsuで環境変数を引き継ぐしか無い。

postgresqlバイナリの場所

find /usr | grep pg\_ctl # Linuxの場合、これで探す事もできる

Win10 C:\Program Files\PostgreSQL\ver\bin

mac /usr/local/opt/postgresql@ver /bin

Linux(Debian) /usr/lib/postgresql/ver/bin/

### 最初の起動

psqlコマンドで起動する。以下の点に注意。

・postgresql用のシステムユーザー（※１）でpsqlを起動する。（アプリへのログインという感覚に近い）

・psql コマンドで -U オプションを省略すると，OSへのログインユーザー名がそのまま使用される。

※１　Linuxのユーザー区分で、特定のアプリケーションを動作させる為のユーザー。OS（Linux）へのログインはできない。

最初の状態ではシステムユーザーはpostgresと言うスーパーユーザーのみ。

以下は具体的な２つのpsql起動方法。

# 方法１（推奨）

sudo -u postgres psql # sudoの-u オプションで一時的にOSのユーザーを変更

# 方法２

sudo passwd postgres # パスワードを設定しておく。（任意）

su postgres # 完全にユーザーを切り替える。

postgre # 実行

postgres =# \q # postgresqlの終了

本ドキュメントでは以後「 =# 」という表現は，「psql上のスーパーユーザーでpsqlを起動」した事を明示する為に使用。

### ユーザーとデータベース作成

注意点

・各コマンドの最後に**；（セミコロン）**を忘れないように

・各コマンドは大文字英語で記述してあるが，**小文字でも問題無い**

・ロール（≒ユーザー）名も大文字小文字を区別しない。 ※以後「ロール」で統一

=# CREATE ROLE user1 LOGIN CREATEDB; -- ロールの作成。ログインとデータベースの管理者権限を付与。

=# CREATE DATABASE user1; -- ロール名と同じ名前のデータベースを作成しておく。

=# CREATE DATABASE user1 OWNER user1; -- 所有者を指定する場合（今回はどちらでも良い）

=# ALTER DATABASE user1 OWNER TO user1; -- あとで所有者を変更する場合

=# ALTER USER user1 WITH PASSWORD ’pass’; -- パスワードの設定

=# \q -- 一旦postgreSQLを終了

# psql -h ホスト名 -p ポート番号 -U ロール名 -d データベース名

# 補足：接続先がlocalhostの場合は-hを省略可能。ポート番号=5432番(def.) の場合-pを省略可能。

# 補足：OSのユーザー名=psqlのユーザー名の場合-U省略可能。接続先データベース名=ユーザー名の場合-dを省略可能。

psql　-h 192.168.1.30　-p 5432　-U user1　-d testdb

user1=> -- =# では無く=> になっている事に注目。これは、一般ユーザーを示す。

user1=> exit -- psqlを終了

今回はテスト用なので，作成するデータベース名はOSのユーザー名と全く同じにしておく（psqlへのログインが楽になる）。

### テーブルの作成

以下のテーブルを作成する。なお，テーブルの事をリレーションと呼ぶ事があるらしい。

departments

|  |  |  |
| --- | --- | --- |
| groupNo | groupName | groupTel |
| 1 | 営業部 | 01209234567 |
| 2 | 技術部 | 01209234569 |
| 3 | 管理部 | 01209234566 |

CREATE TABLE departments (

groupNo int UNIQUE NOT NULL, --PRIMARY KEYの方が良い？2021-08-25

groupName text,

groupTel char(12),

);

--CREATE TABLE と表示されたら成功

(

INSERT INTO departments VALUES(1,’営業部’,’01209234567’);

INSERT INTO departments VALUES(2,’技術部’,’01209234569’);

INSERT INTO departments VALUES(3,’管理部’,’01209234566’);

SELECT \* FROM departments --確認方法

UPDATE departments SET grouptel='01209234567' WHERE groupno=1; --修正方法

### トランザクション

トランザクションを用いて以下のテーブルを作成する。

（詳細）[DCL（トランザクション）](#_DCL（トランザクション）)

members

|  |  |  |  |
| --- | --- | --- | --- |
| no | name | tel | groupNo |
| 1 | 佐藤 | 01201234567 | 2 |
| 2 | 鈴木 | 01201234568 | 1 |
| 3 | 渡辺 | 01201234569 | 3 |

BEGIN; --トランザクションを開始

CREATE TABLE members (

groupNo int UNIQUE NOT NULL,

groupName text,

groupTel char(12),

)

INSERT INTO members VALUES(1, ’佐藤’, ’01209234567’, 2), (2, ’鈴木’, ’01201234568’, 1); --レコード追加：佐藤，鈴木

COMMIT; --一度コミット（確定）する

BEGIN; --再びトランザクションを開始

INSERT INTO members VALUES(3, ‘渡辺’, ’01201234569’, 3); --レコード追加：渡辺

ROLLBACK; --中断

select \* from members; --確認

--渡辺が追加されていない事が確認できる。

BEGIN; --再びトランザクションを開始

INSERT INTO members VALUES(3, ‘渡辺’, ’01201234569’, 3); --レコード追加：渡辺

SAVEPOINT s1; --セーブポイントの作成

DELETE FROM

members WHERE no = 3; --誤って消したとする

ROLLBACK TO s1; --元に戻す

COMMIT; --確定

select \* from members; --確認

--渡辺が追加されている状態である事が確認できる。

### クエリの実行

### 外部からの接続

#### サーバー側設定

接続を許可するアドレス。あくまでテストなので広い範囲にアクセス許可を持たせているが，通常はこんな設定はしないように。

[ /etc/postgresql/ver/main/postgresql.conf ] （参考）[postgresql.conf](#_postgresql.conf)

listen\_addresses ='192.168.1.1' # listenするサーバー側のアドレスを指定。

listen\_addresses ='\*' # 分からなかったらこうする。（すべての使用可能なアドレスをlisten）

port = 5432 # ポート番号を変える場合。[def.= 5432]

[ /etc/postgresql/ver/main/pg\_hba.conf ] （参考）[pg\_hba.conf](#_pg_hba.conf)

TYPE DATABASE USER ADDRESS METHOD # この行はただのヘッダ行

local all all 192.168.1.0/24 trust # 対象セグメントをすべて許可

local user1 all 192.168.1.0/24 md5 # ユーザー名指定＋パスワード認証の場合

設定の有効化

terminal：

sudo systemctl restart postgresql # 再起動する事で上記設定変更を適用する

sudo service postgresql restart # なぜか systemctlではうまく行かない時がある

psql -V # ついでに，先の処理のためにバージョンを確認　--version でも良い

#### クライアント側設定

一般ツールのインストール方法

Linux

sudo apt install postgresql-client-[ver] # postgresql-client-common というのがある。

# psql -h ホスト名 -p ポート番号 -U ロール名 -d データベース名

psql　-h 192.168.1.30　-p 5432　-U user1　-d testdb

Windows

（ダウンロード先）<https://www.postgresql.org/ftp/odbc/versions/msi/>

・管理ツール　スタート → Windows管理ツール → ODBCデータソース

ユーザーDSNタブ： → PostgreSQL UTF8

設定 （補足）サーバー名：IPアドレス等を入れても良い

接続確認 構成 →

ANSIかUTF8かはODBC ドライバを使用する Windows アプリケーションのビルドオプションと合わせる。

（通常UTF8）（winverコマンドで確認可能？）

次ページへ続く

補足

パスワード認証するpsqlユーザーのパスワード指定方法

[ psql ]

=# ALTER USER user1 WITH PASSWORD 'your\_pass'; --パスワードの設定（任意）

=# exit

[ terminal ]

sudo nano ~/.psql\_history # psqlのhistoryファイルでパスワードが丸見えなので対処する

# →該当箇所を削除

エクセルからの接続

・エクセルの32/64bitの確認。

エクセルのメニュー：ファイル → アカウント→ エクセルのバージョン情報

・Excelのbit数とpsql -VにあったpsqlODBC をダウンロード，インストール

・エクセルのファイルで：データ → 外部データの取り込み → その他のデータソース →

データ接続ウイザードから → ODBC DSN → 先ほどのデータソース名 → テーブルを選択

⇒ データの閲覧はできるが，レコードの追加などはできないかも 2021-03-10

C#

npgsql というものを使う（らしい）

https://github.com/npgsql/npgsql/releases/tag/v4.0.13

### 基本の記述法

テーブルの指定方法は “myschema”.”mytable”　と言った形で行う。スキーマを省略するとsearch\_pathに登録されているスキーマの中からmytableを探し，最初に見つけたものが対象となる。

テーブル，列とその値の指定方法

構文：table\_name(field1, field2) VALES(val1, ‘val2’)

・列名無しでVALESのみを使う事もできる。その場合は先頭の列から順番に値が格納される。

・値が文字列の場合はシングルクォーテーションで囲う。

### データベース削除

DROP DATABASE user1; --チュートリアルで使用したデータベースを削除

DROP TABLE department; --1つずつテーブルを削除したい場合

### トラブルシューティング

Is the server running on host "192.168.1.1" and accepting TCP/IP connections on port 5432?

## 詳細

### ソフトウェア情報

#### postgreSQLのバージョン

・バージョンはx.yのように2要素で表記される。(10.1以降)

※9.6以前はx.y.z 形式で，x.y がメジャーバージョンを示していた

・メジャーバージョンアップは，設定機能，大きな機能などの追加・変更を含む

・メジャーverが異なると，データベースを構成するファイルに互換性がない。異なるマイナーverではOK

・メジャーバージョンアップは，1年に1回**程度**，行われる。

・マイナーバージョンのリリースは2，3ヶ月に1回程度行われる。

・マイナーバージョンアップでは，バグの修正やセキュリティ対策などが行われる

・メジャーでもマイナーでもバージョンアップ時にはPostgreSQLを停止させる必要がある。

また，リリースノートに関しては：

・メジャー，マイナー共にバージョンアップ時に公開される

・セキュリティ機能に関する変更も記載されている

・リリースノートは英語だが，日本PostgreSQLユーザー会により，和訳されたものも公開される。

・内部的でユーザーには分からない変更点については，記載されていない。

・Postgres95から現在verまでのリリースノートが閲覧できる。

#### 開発コミュニティ

名前をPostgreSQL Global Development Group(PGDG)と言う。

Webサイトでソースコードやバイナリコードを無償で配布している。

バグ修正の義務を負う保守サポートを提供していない

最終的な意思決定は，開発コミュニティの中心を担う，ごく数名のコアメンバが行う

様々な種類のメーリングリストを運営している

カンファレンスや会合などのイベントを世界各地で開催している

メーリングリストの公用語は英語。

#### プロセス

PostgreSQLは複数のプロセスで構成され，さまざまな処理を行っている。

各プロセスの起動状況は以下のようにして確認できる

ps -ef | grep postgres

postgres: logger process

|  |  |  |
| --- | --- | --- |
| プロセス名 | ps名 | 説明 |
| マスタサーバー |  | 最初に起動される親プロセス |
| ライタ | writer process | 共有バッファの内容をデータファイルに書き出す |
| WALライタ | wal writer process | WALバッファの内容をWALファイルに書き出す |
| チェックポインタ | checkpointer process | すべてのダーティページをデータファイルに書き出す |
| 自動バキュームランチャ | autovacuum launcher process | 設定に従って自動バキュームワーカを起動する |
| 自動バキュームワーカ | autovacuum worker process | 設定に従って自動バキューム処理を行う |
| 統計情報コレクタ | stats collector process | データベースの活動状況に関する統計情報を収集する |
| バックエンド | username databasename [接続] 状態 | クライアントから接続要求に対して起動され，クエリを処理する |
| バックグラウンドワーカ | bgworker: [module\_name] | ロジカルレプリケーション用のワーカ。またユーザー定義のバックグラウンドワーカを組み込んだ場合にも表示される |
| パラレルワーカ | bgworker: parallel worker for PID <PID> | パラレルスキャン実行時に起動され，クエリを処理する |

パラレルスキャン

max\_parallel\_workers\_per\_gather

#### ロックモード一覧

|  |  |
| --- | --- |
| ロックモード | 獲得タイミング |
| ACCESS SHARE | SELECT |
| ROW SHARE　行共有ロック | SELECT FOR UPDATE, SELECT FOR SHARE, KEY UPDATE, KEY SHARE |
| ROW EXCLUSIVE　行排他ロック | INSERT, UPDATE, DELETE |
| SHARE UPDATE EXCLUSIVE  他ユーザーのテーブルに対しての処理を妨げない | VACUUM, ANALYZE, CREATE INDEX CONCURRENTLY, ALTER TABLE, SET, RESET, VALIDATE CONSTRAINT, CLUSTER ON, SET WITHOUT CLUSTER |
| SHARE　共有ロック | CREATE INDEX |
| SHARE ROW EXCLUSIVE | 同一セッション内での競合を防止する為のロックモード。  明示的にこのロックモードを獲得するSQLコマンドは無い。 |
| EXCLUSIVE　排他ロック | 明示的にこのロックモードを獲得するSQLコマンドは無い。 |
| ACCESS EXCLUSIVE  （排他ロック） | ALTER TABLE, DROP TABLE, TRUNCATE, REINDEX, CLUSTER, VACUUM FULL  （SELECTをブロックする唯一のロック） |

表内で明示されていない動作はACCESS EXCLUSIVEになる。

・ロックモードが解放されるまで，ほかのユーザーのトランザクションは待たされる。

ロックモード間の競合（〇：競合しない　×：競合する）

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 現在のロック  　　\  要求するロック | ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE | SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE |
| ACCESS SHARE | 〇 | 〇 | 〇 | 〇 | 〇 | 〇 | 〇 | × |
| ROW SHARE | 〇 | 〇 | 〇 | 〇 | 〇 |  | × | × |
| ROW EXCLUSIVE | 〇 | 〇 | 〇 | 〇 | × | × | × | × |
| SHARE UPDATE EXCLUSIVE | 〇 | 〇 | 〇 | × | × | × | × | × |
| SHARE | 〇 | 〇 | × | × | 〇 | × | × | × |
| SHARE ROW EXCLUSIVE | 〇 | 〇 | × | × | × | × | × | × |
| EXCLUSIVE | 〇 | × | × | × | × | × | × | × |
| ACCESS EXCLUSIVE | × | × | × | × | × | × | × | × |

（情報元）<https://www.postgresql.jp/document/11/html/explicit-locking.html>

（関連）[DCL（トランザクション）](#_DCL（トランザクション）)

#### 初期データベース

initdbコマンドで新しいデータベースクラスタを作った際にも，これらのデータベースは作成される。

|  |  |  |
| --- | --- | --- |
| データベース名 | 説明 | 備考 |
| template0 | テンプレート | 変更不可。  削除不能。  システム標準のオブジェクトのみが含まれる。 |
| template1 | テンプレート | 変更可。  初期状態ではtemplate0と全く同じ。 |
| postgres | 普通のデータベース |  |

補足

・初期状態ではtemplate0，template1は全く同じ内容。

・template0は削除できない

・文字エンコードやロケールを指定してのcreatedbの際，template1を対象とする事はできない。

（template1に先に変更を適用してからcreatedbを行う）

・createdbでテンプレートの指定がない場合はtemplate1が適用される。

#### 分離レベル

トランザクションが同時に実行された場合に，それぞれのトランザクションに与える影響の度合い

下に行くほど分離レベルが高い。

|  |  |  |
| --- | --- | --- |
| 分離レベル | 意味 | 備考 |
| READ UNCOMMITTED | コミットされていないデータが参照されている可能性がある | postgreSQLでは  READ COMMITTEDとして扱われる。どれも防げない |
| READ COMMITTED | 問い合わせが実行される直前までにコミットされたデータのみを参照する | **デフォルト**  ダーティを防ぐ |
| REPEATABLE READ | トランザクションが開始される前までにコミットされたデータのみを参照する。  単一トランザクション内の連続するSELECT文は，常に同じデータを参照する。 | PostgreSQL9.1からサポート  ダーティ，ファジーを防ぐ  ファントムだけ防げない |
| SERIALIZABLE | もっとも厳しいトランザクションの分離レベル。並列実行された複数のトランザクションの実行であっても，逐次的に扱われたものと同じ結果を要求される。 | ファントムリードを含め全て防ぐ |

分離レベルが不十分な場合の挙動

|  |  |  |
| --- | --- | --- |
| 事象名 | 意味 | 抑止可能な分離レベル |
| Dirty Read  ダーティーリード | 他のトランザクションのコミット前のデータを読み込んでしまう。 | READ COMMITTED以上 |
| Fazzy Read  ファジーリード  （反復不可能読み取り） | 反復読込みの際に，1回目と2回目で値が異なる。他トランザクションによる**削除や追加によって**データが変更される。UPDATEかDELETE? | REPEATABLE REA以上 |
| Phantom Read  ファントムリード | 他のトランザクションのコミットの影響で，同一トランザクション内で以前は取得されなかったデータが追加される。INSERT? | SERIALIZABLEのみ |

FAZZY READ

あるデータをトランザクション内で複数回読み込んだ場合に，対象のデータが他トランザクションで更新・削除されてコミットされることにより，取得されるデータ内容が変わってしまう現象のこと

PostgreSQLでは存在しない設定

例

BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ; -- 分離レベルの指定１：開始時

その他の分離レベル指定方法

START TRANSACTION ISOLATION LEVEL SERIALIZABLE; -- BEGIN と全く同じ（トランザクション単位）

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE; -- （トランザクション単位）

SET default\_transaction\_isolation TO ‘SERIALIZABLE’; -- （セッション単位）

#### グローバルオブジェクト

概要

全てのデータベースで共有する，ロール，テーブル空間、データベース全体に適用するアクセス制限などの属性に関する情報の事。pg\_dumpではバックアップの対象とならず，対象としたい場合はpg\_dumpallコマンドを使用する必要がある。

#### バキューム

データベース内の使われていない領域などを監視，管理する事。

・実行しないまま運用を進めるとデータファイルが肥大化する為、必須ではないが、実行は推奨。

・MVCC；Multi-Version Concurrency Control　多版型同時実行制御　をとっている。

autovacuum

autovacuumデーモン（プロセス）による自動バキューム処理の事。VACUUM ANALYZEと同等の処理を行う。

・処理は複数のプロセスによって実行される

・テーブルに対して大量のデータ更新があると実行される

・PostgreSQLのver8.3以降ではデフォルトで有効。（関連）[postgresql.conf](#_postgresql.conf)のautobacuum

・[postgresql.conf](#_postgresql.conf)のautovacuumを無効に設定した場合、自動バキュームは実行されないが、VACUUM（またはvacuumdb）コマンドの手動実行は問題なくできる。

・処理が結構重たいらしいので，バキュームが動作する時間帯などを適切に設定する。

VACUUM

手動でバキューム処理を行うデータベースコマンド。クライアントコマンドであるvacuumdbのラッパーコマンドらしい。

・バキューム対象に対しての所有者権限が必要

（関連）[VACUUM](#_VACUUM)コマンド

言葉の定義

統計情報

不要領域を回収

不要な領域を判別できるようにマークして、その領域を再利用できるようにする。

#### 関連パッケージ

libpq-dev

postgresql-contrib

補足：contribはcontributionの略らしい

python3-dev

psycopg2

psycopg2-binary

#### 環境変数

$POSTGRES\_HOME よくわからん 2021-09-21

export POSTGRES\_HOME=/usr/local/pgsql

情報元は少し古い： <http://linux.kororo.jp/cont/server/postgresql.php>

$PGDATA データベースクラスタのルートディレクトリの絶対パスを格納。

initdb などの-Dオプションの代わりになる環境変数。

export PGDATA=$POSTGRES\_HOME/data

SHOW data\_directoryの結果と同じにしておけば良い？2021-10-05

#### そのほか

psql起動後の　=>　と =# の違いは？2021-08-24

|  |  |  |
| --- | --- | --- |
|  | 文字 | 説明 |
| 1文字目 | # | スーパーユーザーでpsqlを起動している。（おそらく） |
| 1文字目 | > | 一般ユーザーでpsqlを起動している。（おそらく） |
| 2文字目 | = | 途中コマンドが無い，まっさらな状態。 |
| 2文字目 | - | ；（セミコロン）で確定されていない，途中コマンドがある。 |

#### libpq

PostgreSQLデータベースサーバにアクセスするためのC言語のライブラリです。PHP,Ruby,Perl,Python,NodeJS,etcはlibpqを利用してPostgreSQLにアクセスするAPIを提供している。

新しいバージョンで追加された機能

13 VAGUUM PARALLEL

14 COPY FREEZE

15 MERGE

### 設定ファイル

#### 概要

postgresql.conf 全体の設定

pg\_hba.conf クライアントからの接続を制御する

pg\_ident.conf ident認証およびGSSAPI認証で使用される

recovery.conf アーカイブリカバリ用の設定ファイル

pg\_service.conf libpqライブラリの接続情報をサービスとしてまとめて管理。

設定ファイルの値を変更したら，PostgreSQLサーバー（サービスだけで良い？）を再起動する必要がある。

# テーブルのように扱うことができる？ 2021-08-16

=# SELECT name, setting, unit FROM pg\_settings WHERE name LIKE ‘%wal%’;

2021-08-31 設定ファイルを見ることができるVIEWが存在する事が判明

場所

Debian：/etc/postgresql/ver/main/

#### postgresql.conf

保存場所：

Debian：/etc/postgresql/ver/main/postgresql.conf

※データベースクラスタは別ディレクトリ /var/lib/postgresql/ver/main/ に存在する。

注意事項

・メモリの単位：B(バイト), kB(キロ), MB(メガ), GB(ギガ), TB(テラ)　(乗数は1024)　kは小文字にする

時間の単位：ms(ミリ秒), s(秒), min(分), h(時間), d(日)　※month, yearは無いらしい

・単位が無くてもエラーにはならない。各設定値ごとにデフォルト単位が定められている。

・全てのパラメータ名は大文字と小文字の区別がない

・# 以降の文字列はコメントとして扱われる。

反映タイミング

以下の表では[postmaster]などと表記　情報元：pg\_settingテーブルのcontext列より

internal データベースクラスタの構築後は変更できない

postmaster PostgreSQLサーバーの起動・再起動時。post=postgreの意？？

sighup 「postgresql.conf」の再読み込み時

superuser-backend スーパーユーザーで新しいセッションを開始時

backend 一般ユーザーで新しいセッションを開始時

superuser スーパーユーザーでSETコマンド実行時

user 一般ユーザーでSETコマンド実行時。（postgresql.conf再読み込みも適用？）

ファイルの場所

|  |  |  |
| --- | --- | --- |
| 項目 | 説明 | 備考／値 |
| data\_directory | データベースクラスタを格納するディレクトリのフルパス |  |
| hba\_file |  |  |
| indent\_file |  |  |

接続と認証

|  |  |  |
| --- | --- | --- |
| 項目 | 説明 | 備考／値 |
| listen\_addresses  [postmaster ?] | 接続を受付(listen)するサーバー側のアドレス。 | ’localhost’ local機のみ’  127.0.0.1 自身のloop  空文字　local機のみ(def. コメントアウト)  ’\*’　　 全て  ’0.0.0.0’ IPv4全て ’::’　　 IPv6全て  '192.168.1.1, localhost' 複数 |
| port  [postmaster] | 接続を待ち受けるポート | def.=5432 |
| max\_connections  [postmaster] | 許可する最大接続数  超えるとエラーで接続が拒否 | def.=100 |
| ssl\_cert\_file |  |  |
| ssl\_key\_file |  |  |

次ページへ続く

～postgresql.conf　続き

リソース管理

|  |  |  |
| --- | --- | --- |
| 項目 | 説明 | 備考／値 |
| shared\_buffers |  |  |
| temp\_buffers |  |  |
| work\_mem |  |  |
| max\_stack\_depth |  |  |

クライアント接続

|  |  |  |
| --- | --- | --- |
| 項目 | 説明 | 備考／値 |
| client\_encoding  [user] | クライアント機のエンコーディング | def.=’SQL\_ASCII’  UTF8 など |
| default\_transaction\_isolation  [user] | 新しいトランザクションのデフォルトの分離レベル | （関連）[分離レベル](#_分離レベル)  read uncommitted  (def.) **read committed**  repeatable read  serializable |
| search\_pach  [user] | 予め登録しておくスキーマ | def.=”$USER”, public |

次ページへ続く

～postgresql.conf　続き

レポートとロギング

統計情報のデータは出力できない仕様？

|  |  |  |
| --- | --- | --- |
| 項目 | 説明 | 備考／値 |
| logging\_collector | **標準エラー**をファイルに書き出すかどうか | def.=off  on/off, true/false, yes,no, 1/0　(boolean) |
| log\_destination | サーバーログをどこに出力するのかを設定 | 'syslog'　Linuxのsyslog（※１）  'stderr'　標準エラー(def.)  csvlog　CSV形式（どこ？）  注意：off という設定値は無い |
| log\_connections | 接続クライアントの接続認証成否などの情報を出力するかどうか | def.=off  on/off, true/false, yes,no, 1/0　(boolean) |
| log\_directory | ログファイルを格納するディレクトリを設定 | def.=log |
| log\_filename | ログのファイル名を設定 | def.='postgresql-%Y-%m-%d\_%H%M.log'  → 1999-12-31\_2359.log といった形 |
| log\_line\_prefix | サーバログメッセージの行頭に、ユーザー名やデータベース名、日付などの書式文字列を設定。 | 値  %t  %r  %u  %d  def.=‘%m [%p]’ |
| log\_min\_messages | サーバーログに書き込むログのレベルを設定 | 値  SQL？？  INFO  ERROR  WARNING (def.) |
| log\_statement | サーバーログに書込むSQL文の種類  ※スーパーユーザーのみ変更可 | all　実行した全てのSQL文が対象  ddl　データ定義（DDL）が対象  mod　DDL+DML  non[def.] |
| log\_truncate\_on\_rotation |  |  |

※１journald環境の場合、どこに出力される？

次ページへ続く

～postgresql.conf　続き

バックアップ

|  |  |  |
| --- | --- | --- |
| 項目 | 説明 | 備考／値 |
| wal\_level  [postmaster] | WALにどの程度の情報を書き込むか | サーバーの起動時のみ変更可能  minimal クラッシュリカバリのみ実施  archive　minimalに加えストリーミングレプリケーションを実施  hot\_standby　archiveに加えホットスタンドバイの実施  **replica**(def.)　ver9.6以降 =hotstandby  logical　hot\_standbyに加え、WALの復号化 |
| archive\_mode | WALファイルをWALアーカイブとして保存するかどうか | on  **off**(def.)  always |
| max\_wal\_size |  | def.=1GB |
| min\_wal\_size |  | def=80MB |
| archive\_command | WALアーカイブとして保存するためのシェルコマンドを指定する | def.=空文字  %pパス名　　%fファイル名 |

PITRを使用する為には

wal\_levelをreplicaまたはlogical

archive\_modeをonまたはalways

archive\_commandの設定例　‘cp %p [WALアーカイブの保存先のパス]/%f’

archive\_command = 'test ! -f /mnt/server/archivedir/%f && cp %p /mnt/server/archivedir/%f' # Unix

archive\_command = 'copy "%p" "C:\\server\\archivedir\\%f"' # Windows

バキューム

|  |  |  |
| --- | --- | --- |
| 項目 | 説明 | 参考／値 |
| autovacuum  [ sighup ] | autovacuumプロセスの実行可否 | on/off, true/false, yes,no, 1/0　(boolean) |
| track\_counts  [ superuser ] | データベースの活動についての統計情報の収集の可否 |  |
| autovacuum\_freeze\_max\_age  [ postmaster ] |  |  |
| autovacuum\_vacuum\_threshold | vacuum起動に必要な更新レコード数 | def.=50 |

資源の消費

レプリケーション

問い合わせ計画

エラー報告とログ取得

実行時統計情報

ロック管理

バージョンとプラットフォーム

エラー処理

そのほか

|  |  |  |  |
| --- | --- | --- | --- |
| 項目 | 説明 | 反映 | 備考／値 |
| work\_mem | クエリ実行時に使われる領域。 |  |  |
| maintenance\_work\_mem | データベースメンテナンスの作業で使われる領域 |  |  |

#### pg\_hba.conf

概要

権限ファイル。

レコードの上から順に行われ，先に一致した行の認証方式を採用する仕様。

その為基本的には，先に範囲を狭く許可設定し，後ろに行くほど広く厳しい認証（または拒否）を行う。

ファイルパス

Debian /etc/postgresql/ver/main/pg\_hba.conf

内容

構文：TYPE　DATABASE　USER[…]　ADDRESS　METHOD

local all postgres trust

host all postgres localhost trust

host db1,db2 admin 10.4.1.0/24 peer

host db1,db2 admin 192.168.100.10 md5

TYPE

local サーバーlocal機からの直接アクセス。

host IP接続のhost機からのアクセス。ssl有無は問わない。

hostssl SSLを用いたhost機からのアクセス。

hostnossl SSLを用いないhost機からのアクセス。

DATABASE

all

sameuser 指定したユーザーと同じ名前のデータベースへの接続に対応。

samerole データベースと同じ名前のロールのメンバに対応します。

replication レプリケーション接続に対応します。

*db\_name* 特定のデータベースを指定。**コンマ区切りで複数指定**できる。

USER

all 全ユーザーを対象。但し，個別のユーザー設定がある場合はそちらが優先される。

*user\_name* 特定のユーザー名を指定

\* これは使えない。エラーになる？　postgresql.confのlistenアドレスと混同しないように。

次ページへ続く

pg\_hba.conf　続き

ADDRESS

IPv4単一 172.20.143.89/32

IPv4複数（小規模） 172.20.143.0/24

IPv4複数（大規模） 172.20.0.0/16

IPv6単一 ::1/128

IPv6複数（小規模） fe80::7a31:c1ff:0000:0000/96

IPv6すべて ::0/0

\* これは使えない。エラーになる？　postgresql.confのlistenアドレスと混同しないように。

METHOD

認証方式

|  |  |
| --- | --- |
| 設定値 | 説明 |
| trust | 無条件で許可。認証なし。 |
| reject | 無条件で拒絶。 |
| md5 | 暗号化パスワード。（psqlで普通にパスワード設定した時はこの設定を用いる） |
| password | 平文パスワード。（めったに使わない？） |
| scram-sha-256 | PostgreSQL10以降 |
| gss | GSSAPI（Linux環境からTCP/IP接続でのみ使用可能） |
| sspi | Windows環境のみ |
| ident | Identサーバーからクライアント情報を入手して認証 |
| peer | peer認証。（OSのユーザー名とpsqlのユーザー名の一致） |
| ldap | LDAPサーバーを使ったパスワード認証 |
| radius | RADIUSサーバーを使ったパスワード認証 |
| cert | SSLクライアント証明書を使った認証 |
| pam | Pluggable Authentication Module　パスワード認証 |
| bsd | OSによって提供されたBSD認証サービス |

～設定ファイル　つづき

#### pg\_ident.conf

#### postgresql.auto.conf

ALTER SYSTEM を行う事で自動的に作成されるファイル。ver9.4で追加。

データベースクラスタ直下（ルートディレクトリ）に配置される。

設定の再ロードやサーバーの再起動でpostgresql.confに変更点の上書きを行う。

（参考）ALTER SYSTEM

#### recovery.conf

PITR（オンラインで行う物理バックアップ）時に使用する。ユーザーが手動で作成する必要がある。

|  |  |  |
| --- | --- | --- |
| 項目 | 説明 | 備考／値 |
| recovery\_target |  | 厳密には、immediateだけが有効な値 |
| recovery\_target\_time | リカバリを実施するタイムスタンプの上限を指定 |  |
| recovery\_target\_xid | リカバリを進行させるトランザクションIDの上限を指定 |  |

（補足）ver12ではpostgresql.confに統合された。

#### pg\_service.conf

#### rc.conf

保存場所：

Debian：/etc/rc.conf

postgresql\_enable="YES"

postgresql\_data="/home/pgdata"

postgresql\_user="hogeuser"

#### ALTER SYSTEM

postgresql.confの内容を編集する

構文：ALTER SYSTEM SET パラメータ名 TO {設定値 | DEFAULT};

構文：ALTER SYSTEM SET パラメータ名 = {設定値 | DEFAULT};

=# ALTER SYSTEM SET shared\_buffers = ‘256MB’

=# ALTER SYSTEM SET shared\_buffers TO ‘256MB’

ALTER SYSTEM RESET??

・コマンドを実行すると，設定値は一時ファイルpostgresql.auto.confに書込まれる。

・コマンドで設定された値の反映タイミングは、各パラメータによって異なる。

・設定の再ロードやサーバーの再起動でpostgresql.confに変更点の上書きを行う。

・実行にはスーパーユーザー権限が必要

・ver9.4で追加された。

（比較）SET：postgresql.confに直接書き込む。

#### SET

実行時パラメータを変更し、postgresql.conf に直接書き込む。

構文：SET [ 有効範囲 ] parameter\_name { TO | = } { value | 'value' | DEFAULT }

構文：SET [ 有効範囲 ] TIME ZONE { timezone | LOCAL | DEFAULT }

有効範囲

SESSION 有効範囲を現在のセッションのみにする(def.)

※セッション　PostgreSQLサーバーへの接続

LOCAL 有効範囲を現在のトランザクションのみにする

・実行時パラメータの中には，スーパーユーザーでないと設定できない項目もある。

・トランザクションがロールバックされるとパラメータ値は**トランザクション開始時の値**に戻る。

・トランザクションがコミットされるとそのパラメータ値は同一セッション内で保存される。

（比較）ALTER SYSTEM：postgresql.auto.confを経由してpostgresql.confに書き込む。

注意

メタコマンドの\set は環境変数の設定

#### SHOW

実行時パラメータ（≒環境変数）の値を表示

SHOW ALL --すべてのパラメータを表示

SHOW search\_path -- 実効時パラメータ（この場合sarch\_path）を表示

#### search\_path

SQLでスキーマを指定しない場合に候補となるスキーマ名を格納する。

SHOW search\_path; --

> ”$user", public -- 結果 "$user" は現在のロール名を返す。

SET search\_path TO "$user", groupA, public; -- 設定の仕方

注意点

・左側に記載されているスキーマから優先的にアクセスされる。後ろのスキーマに同名のテーブルが有っても無視される。

・スーパーユーザーでなくともSETの実行は可能。

未確認情報

ALTER USER [ユーザー名] SET search\_path TO [スキーマ名];

特定のロールにのみ追加？

#### pg\_settings

設定値を扱いやすくしたVIEW。

・UPDATEにより更新が可能。WHERE name = *pram* で対象を指定，settingで値を指定。

UPDATE pg\_settings SET setting = 8192 WHERE name = ‘work\_mem’

・更新や参照に特別な権限は必要無い。

・行の挿入，削除は出来ない。

context列 更新タイミング

internal データベースクラスタの構築後は変更できない

postmaster PostgreSQLサーバーの起動・再起動時

sighup 「postgresql.conf」の再読み込み時

superuser-backend スーパーユーザーで新しいセッションを開始時

backend 一般ユーザーで新しいセッションを開始時

superuser スーパーユーザーでSETコマンド実行時

user 一般ユーザーでSETコマンド実行時

name列 対象のパラメータ値が表示

setting列 現在のパラメータ値が表示

select name, context, setting from pg\_settings;

#### その他

postgresql.auto.conf

ALTER SYSTEMコマンドによって変更されたパラメータが保存される一時ファイル。

同パラメータがpostgresql.confにもある場合，postgresql.auto.confのものが優先される。

・データベースクラスタのディレクトリ直下に配置される。

・ver9.4で追加

### ロギング

#### 概要

ログに関するパラメータはpostgresql.confで設定するものがほとんど。

一部SETコマンドの実行で変更できるものもある。

/var/log/postgresql/postgresql-11-main.log がデフォルト？

#### ログレベル

　INFO ユーザーから出力を要求された情報

　NOTICE ユーザーにとって役立つ情報

　LOG データベース管理者にとって役立つ、パフォーマンスや内部の処理に関する情報

　WARNING 不適切なコマンド使用等に対するユーザーへの警告。（デフォルト）

　ERROR **特定のコマンド**を中断させたエラー

　FATAL **特定のセッション**を中断させたエラー

　PANIC **全てのセッション**を中断させた致命的なエラー

### システムカタログ

#### 概要

プログラムの設定ファイル的に使われるビュー。**PostgreSQL固有**の機能。（比較）情報スキーマ

データベースクラスタに関する，**詳細なメタ情報の確認**に使用する。

・ビューとして提供され，そのままの形でユーザーが値を読み取る事ができる。

・情報スキーマより詳細な情報が確認できる

・システムカタログのスキーマは pg\_catalog

（参考）https://www.postgresql.jp/document/11/html/catalogs.html

比較

|  |  |  |
| --- | --- | --- |
|  | システムカタログ | 情報スキーマ |
| 規格 | postgre独自 | SQL標準 |
| 対象 | データベースクラスタの管理 | データベースオブジェクトに関する定義情報 |
| ビュー | pg\_tablesなど | public.talbesなど |
| 権限 |  | 参照に特別な設定や権限は不要  スーパーユーザーが所有？ |
| その他 | スキーマの名称はpg\_catalog | より詳細な情報 |

次ページへ続く

～システムカタログ　続き

#### ビュー

pg\_authid ロール名・ロールの権限・暗号化されたパスワードなど、データベースクラスタ全体のロールに関する情報を格納

pg\_tables データベースクラスタ全体のテーブル情報を格納。スキーマ、所有者など。

SELECT \* FROM pg\_catalog.pg\_tables; -- スキーマ名を含んだ，正式名称？

SELECT \* FROM pg\_tables; -- 同名のテーブルは無いはずなのでスキーマ名は省略できる

pg\_database データベースクラスタ全体のデータベース情報を格納。名前、所有者など。

データベースクラスタにつき１つ存在する。

pg\_indexes インデックスを有するスキーマ名，テーブル名など。

pg\_user データベースクラスタ全体のユーザー情報を格納。ユーザーの名前・ID・権限など。

pg\_roles ？

rolname, rolsuper ユーザー情報

rolcreaterole, rolcreatedb, rolcanlogin 権限情報

SELECT rolname, rolcreatedb FROM pg\_roles; --ロール（≒ユーザー）の確認

pg\_views

pg\_timezone\_names タイムゾーンの一覧

### 情報スキーマ

information\_schema

データベースクラスタに関するメタ情報を，**標準SQL規格に準拠する形で取得**する。

→ スキーが標準　と覚える 2021-10-11

・スーパーユーザーに所有される、ビューとテーブルのグループ。

・SQLの標準規格に当てはまる。

・データベースオブジェクトに関する定義情報が格納されている。

・システムカタログとは異なり、参照に特別な設定や権限は不要。

・情報スキーマよりシステムカタログの方が詳しい情報が取得できる。

SELECT \* FROM information\_schema.tables; -- 情報スキーマのテーブル情報の一覧を表示

pg\_settings

pending\_restart 再起動による適用待ち状態。

### データ型

#### 数値系

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 別名 | byte数 | 説明 |
| smallint | int2 | 2 | 符号付き整数 2byte(16bit) 　C言語のshortに相当 |
| integer | int, int4 | 4 | 符号付き整数 4byte(32bit) 　C言語のintに相当 |
| bigint | int8 | 8 | 符号付き整数 8byte(64bit) |
| real | float4 | 4 | 単精度浮動小数点（4バイト）　C言語のfloatに相当  最低6桁の精度 |
| double precision | float8 | 8 | 倍精度浮動小数点（8バイト）　C言語のdoubleに相当  最低15桁の精度 |
| numeric [ (p, s) ] | decimal [ (p, s) ] | 可変 | 精度の選択が可能な高精度数値。最大1000桁。  int8では足りない場合やfloatなどの誤差を許さない場合に使う。  範囲外の数値を格納しようとするとエラーとなる  p数字全体の有効桁数　　　s 小数部分の桁数 |
| smallserial | serial2 | 2 | 連番型と呼ばれる。自動採番　整数のみ  連番抜けの可能性あり。シーケンスも同時に作成される。  NULLを格納しようとするとエラー。負の値は格納可能。 |
| serial | serial4 | 4 |
| bigserial |  | 8 |

UNSIGNEDオプションで符号なしにできる？

#### 基本の型

|  |  |  |
| --- | --- | --- |
| 名称 | 別名 | 説明 |
| boolean | bool | 論理値（真/偽）　‘on’/’off’, ‘true’/’false’, ‘yes’,’no’, ‘1’/’0’, の文字列，またはTRUE/FALSE　NULLを格納できる。  int型の値は格納できない。 |
| bit [ (n) ] |  | 固定長ビット列 |
| bit varying [ (n) ] | 可変長ビット列[ (n) ] |  |
| uuid |  | 汎用一意識別子 |
| bytea |  | バイナリデータ（「バイトの配列（byte array）」）  最大1GBまで。ただし推奨は100kBまでで，それ以上はラージオブジェクトを推奨。ラージオブジェクトは4TBまで。 |

#### 文字列系

‘値’　と言った形でシングルクオーテーションで囲う

|  |  |  |
| --- | --- | --- |
| 名称 | 別名 | 説明 |
| text |  | 制限なし可変長　ヘッダ4byte。text[] で配列？  1GBまで格納可能  =NEW どうなる？？2021-09-09 |
| character [ (n) ] | char [ (n) ]  DBeaverではbpcharだった | 空白埋め固定長　ヘッダ1byte。0~255文字まで  埋められた空白文字はLIKEなどでの比較の際には無視される。  指定文字数を超えると，超えた分は切り捨て。  (n) を省略すると(1) になる。n文字**以内**の長さが入力可能  **あくまで文字数**なので「田中」は4byteだが2文字となる |
| character varying [ (n) ] | varchar [ (n) ] | ・上限付き可変長　ヘッダ1byte。0~65535文字まで。  ・格納時にサイズ上限確認が行われ，極僅かな負荷になる。  ・指定長さに満たない部分は…　どうなる？2021-09-07  ・(n) 省略で1GBまで格納可。  ・(n) 指定でn文字**以内**の長さまで格納可。 |

補足

・char, varchar共に上限を超えるとエラーになるが，超えた文字列が空白文字の場合はエラーにならない。

・システムカタログ内のみで使用される“char”型とname型が存在する。

・varcharとtextはあまり違いが無いと言える。

#### 日時系

|  |  |  |
| --- | --- | --- |
| 名称 | 別名 | 説明 |
| date |  | 暦の日付（年月日）　4byte |
| time [ (p) ] [ without time zone ] |  | 時刻（時間帯なし）　8byte  μ秒を含む。 |
| time [ (p) ] with time zone | timetz | 時間帯付き時刻　12byte  UTC（世界協定時）で格納。μ秒を含む。 |
| timestamp [ (p) ] [ without time zone ] |  | 日付と時刻（時間帯なし）　8byte  日付がある為に夏時間などへの対応も可能。μ秒を含む。  2000-01-01 00:00:00からの経過μ秒　らしい |
| timestamp [ (p) ] with time zone | timestamptz | 時間帯付き日付と時刻　8byte  UTC（世界協定時）で格納。μ秒を含む。 |
| interval [ fields ] [ (p) ] |  | 時間間隔　16byte |
|  |  |  |

=# select '1999-10-20 01:30:00 JST'::timestamptz; -- ::型　で文字列を型変換

timestamptz

------------------------

1999-10-20 01:30:00+09

(1 行)

現在日時の3か月前の日時を取得

SELECT now() - '3 month'::interval;

SELECT now() - interval '3 month';

#### ネットワーク系

|  |  |  |
| --- | --- | --- |
| 名称 | 別名 | 説明 |
| cidr |  | IPv4もしくはIPv6ネットワークアドレス |
| inet |  | IPv4もしくはIPv6ホストアドレス |
| macaddr |  | MAC（Media Access Control）アドレス |
| macaddr8 |  | MAC (Media Access Control) アドレス (EUI-64 形式) |
| tsquery |  | テキスト検索問い合わせ |
| tsvector |  | テキスト検索文書 |
| xml |  | XMLデータ |
| json |  | テキストのJSONデータ |
| jsonb |  | バイナリ JSON データ, 展開型  高速に処理する |

#### 図形系

|  |  |  |
| --- | --- | --- |
| 名称 | 別名 | 説明 |
| box |  | 平面上の矩形 |
| circle |  | 平面上の円 |
| line |  | 平面上の無限直線 |
| lseg |  | 平面上の線分 |
| path |  | 平面上の幾何学的経路 |
| point |  | 平面上の幾何学的点 |
| polygon |  | 平面上の閉じた幾何学的経路 |

#### 配列型

・全てのデータ型に使用することができる

・配列型「name」の1番目の要素を指定する場合は、name[1]と指定する

⇒なぜ，一般的なプログラムの方式と違うことをやるのか理解に苦しむ。（通常，1番目の要素はname[0]）

・配列型の要素数に制限は無い。

CREATE TABLE sal\_emp (

name text,

pay\_by\_quarter integer[3], -- 一次配列を作成。要素数も指定。

schedule text[2][4] -- 二次配列を作成。

);

INSERT INTO sal\_emp VALUES ('Bill',

'{10000, 10000, 10000, 10000}',

'{

{"meeting", "lunch"},

{"training", "presentation"}

}');

column1 integer[], -- 要素数が指定されない場合は配列無しとみなされる？？

#### その他

|  |  |  |
| --- | --- | --- |
| 名称 | 別名 | 説明 |
| money |  | 貨幣金額 |
| pg\_lsn |  | PostgreSQLログシーケンス番号 |
| txid\_snapshot |  | ユーザレベルのトランザクションIDスナップショット |
|  |  |  |

（情報元）<https://www.postgresql.jp/document/11/html/datatype.html>

NULLについて

NULLはIS NULL/IS NOT NULL で判定する。（通常は=や<>を用いる）

補足：NULL値の判定に=や<>を用いると，その結果がNULLとして返る。

### 各言語からの接続

Java PostgreSQL JDBCドライバ

python

libpg

C言語で作られたPostgreSQL用のインターフェイス。大体必要になるので入れておく

sudo apt install libpq-dev

•psycopg 一番人気

•pg8000

•py-postgresql

### 任意のディレクトリでinitdb

情報元：<https://www.servernote.net/article.cgi?id=postgresql-install-and-initdb-to-customdir>

### 一般情報

#### PGDG ；PostgreSQL Global Development Group

PostgreSQLの開発コミュニティの事。

University of California, Berkeley

後のPostgreSQLとなる，postgresを開発していた。PostgreSQLの開発コミュニティとは異なる。

#### SQL

リレーショナルデータベースを操作・定義するための，データベース言語(問い合わせ言語)。

1986年，ANSIによって標準規格が制定された後，1987年にはISOにおいて国際標準化。

数年ごとに改定されており，SQL：2008，SQL：2003というように制定された年号で規格を区別する。

PostgreSQLは標準SQLの大部分をサポート。

#### 正規化

元のレコードがこうだったとする。（この状態を「繰り返し」が存在すると言う）

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 注文ID | ユーザーID | ユーザー名 | 商品ID | 商品名 | 単価 | 個数 | 商品ID | 商品名 | 単価 | 個数 |
| T001 | U001 | 鈴木 | S001 | りんご | 100 | 1 | S002 | バナナ | 150 | 2 |
| T002 | U002 | 佐藤 | S003 | ブドウ | 200 | 3 |  |  |  |  |
| T003 | U001 | 鈴木 | S004 | みかん | 80 | 4 | S005 | もも | 300 | 2 |

注文ID，商品IDが主キーとなっている

第１正規形

繰り返し部分（同じ列名の重複など）を除く。上のテーブルは商品ID～個数が２つずつあるのでこれを解消。

・主キーを設定する。言い換えると，レコードを一意に特定できるようにする。

・導出項目(他の属性から算出できる項目)を削除する

※属性≒データ型，値

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 注文ID | ユーザーID | ユーザー名 | 商品ID | 商品名 | 単価 | 個数 |
| T001 | U001 | 鈴木 | S001 | りんご | 100 | 1 |
| T001 | U001 | 鈴木 | S002 | バナナ | 150 | 2 |
| T002 | U002 | 佐藤 | S003 | ブドウ | 200 | 3 |
| T003 | U001 | 鈴木 | S004 | みかん | 80 | 4 |
| T003 | U001 | 鈴木 | S005 | もも | 300 | 2 |

第２正規形

複合主キーを１つのテーブルで持たない状態にする。主キーを１つにするとも言える。（外部キーは2つ以上あっても良い）

主キーに対してすべての非キー属性（＝外部キーに関連する項目以外）が完全関数従属したもの。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 注文ID | ユーザーID | ユーザー名 | 商品ID | 個数 |
| T001 | U001 | 鈴木 | S001 | 1 |
| T001 | U001 | 鈴木 | S002 | 2 |
| T002 | U002 | 佐藤 | S003 | 3 |
| T003 | U001 | 鈴木 | S004 | 4 |
| T003 | U001 | 鈴木 | S005 | 2 |

|  |  |  |
| --- | --- | --- |
| 商品ID | 商品名 | 単価 |
| S001 | りんご | 100 |
| S002 | バナナ | 150 |
| S003 | ブドウ | 200 |
| S004 | みかん | 80 |
| S005 | もも | 300 |

第３正規形

主キー以外の列が依存関係を持たない状態。

推移関数従属する属性を取り除いたもの。どの候補キーに対しても依存関係になっていない。

例となっている右テーブルでは主キーではないが1つの値が決まればもう一方が決まる（関数従属）状態なので，それも別テーブルとする。

|  |  |  |  |
| --- | --- | --- | --- |
| 注文ID | ユーザーID | 商品ID | 個数 |
| T001 | U001 | S001 | 1 |
| T001 | U001 | S002 | 2 |
| T002 | U002 | S003 | 3 |
| T003 | U001 | S004 | 4 |
| T003 | U001 | S005 | 2 |

|  |  |
| --- | --- |
| ユーザーID | ユーザー名 |
| U001 | 鈴木 |
| U002 | 佐藤 |

次ページに続く

～正規化　続き

補足：

・第４正規形，第５正規形も存在するが，実行されるのは非常に稀。

・正規化されすぎる事でパフォーマンスが下がる場合もある。

Xを決定項または候補キー，Yを被決定項または非キー項目と呼ぶ。

関数従属(functional dependency)　ある値が決まると別の値が決まる関係にある状態。

XがYに関数従属する場合 Y→Xと表す。　例：社員ID → 氏名　 ※X, Yの順番に注意！

部分関数従属性

XがABからなる場合は {A,B} →YかつA→Y（またはB→Y）と表す。

{A,B} →YかつA→Y の時，YはAに部分関数従属する　と表現する。

完全関数従属性

候補キーが決まると，すべての被決定項が決定する

XがABからなる場合は { A, B }→Yと表す。YはA,Bに完全関数従属する　と表現する。

推移関数従属性

X→Y かつY→ZでY→XやX→Zは不成立。社員ID→住所→郵便番号　のような流れ。

ZはXに推移関数従属する　と表現する。

#### サポート

PostgreSQL Global Development Group(PGDG)」という開発コミュニティは無償のサポートを提供しており，サポートの期限は「最初のメジャーバージョンのリリースから5年間」となっている。

例

2012年9月：PostgreSQL 9.2

2013年9月：PostgreSQL 9.3

2017年8月：PostgreSQL 9.2.23

→ 上記全て，2017年の9月にサポートが終了する。

### 特有の機能

#### PL/Python

ストアドプロシージャあるいはストアドファンクションをPython言語で記述するための手続き言語

## データベースクラスタ

PostgreSQLの処理で使用する一連のディレクトリとファイル群。1つのデータベースのルートディレクトリという捉え方もできる。

場所

/var/lib/postgresql/ver/main など

=# SHOW data\_directory; -- 格納ディレクトリが表示される。（スーパーユーザー権限が必要）

特徴

・1つのデータベースクラスタには複数のデータベースを作成できる。

・PostgreSQLサーバー（プロセス群やサービス）1つに対しデータベース1つが対応する。

但しPostgreSQLサーバーを1台のマシンに複数インストールする事は可能。

主要ディレクトリ

base

データベースごとにOID名でディレクトリが作成される。

インデックスファイル，TOASTファイル，Free Space Mapファイル，Visibility Mapファイルなどが作成される。

global

データベースクラスタ内で共有するテーブルを保持する。

（ここに格納されるオブジェクトをグローバルオブジェクトと呼ぶ？2021-11-16）

pg\_wal

WALファイルを格納する。

pg\_xact

トランザクションのコミット状態を管理するファイルが格納。

pg\_tblspc

テーブル空間（デフォルトではbaseディレクトリ？）のファイルへのシンボリックリンクを保有するサブディレクトリ

### テーブルスペース

別名：テーブル空間

#### 概要

データベースオブジェクトの実ファイルを格納するファイルシステム上の場所の事。

データベースオブジェクトの容量管理や，データベースオブジェクトの使用頻度などに応じてストレージを割り当てられる。

複数台のストレージを指定すれば，並列的にデータベースサーバとしての性能を上げる事もできる。

テーブルスペースの確認方法：

=# select oid, \* from pg\_tablespace; -- まずはoidを確認（なぜか明示的に列名の指定が必要）

=# select pg\_tablespace\_location(1663); -- これで具体的な場所を確認できる

pg\_default template0, template1用

pg\_global 共有システムカタログ用

注意点

・デフォルトでは $PGDATA/base。CREATE TABLESPACEコマンドで作成する。

・作成する場合は，ディレクトリパスの指定が必要

・「場所」で指定するディレクトリは事前に作成する必要がある

・

#### 関連コマンド

構文：CREATE TABLESPACE name [ OWNER {所有者名 | CURRENT\_USER | SESSION\_USER} ]

LOCATION 'テーブルスペースのパス';

OWNER

CURRENT\_USER：コマンドを実行しているユーザー名

SESSION\_USER：セッションを所有しているユーザー名

※セッション　PostgreSQLサーバーへの接続

LOCATION

**必須オプション**。対象パスのディレクトリは事前に作成する必要がある。

例

=# CREATE TABLESPACE tbspc\_name LOCATION ‘/tmp/name\_tbspc’

CREATE TABLESPACE tbspc\_name -- 作成

ALTER TABLESPACE tbspc\_name -- 修正

DROP TABLESPACE tbspc\_name -- 削除

ALTER

ALTER TABLESPACE tbspc\_name RENAME\_TO new\_name

ALTER TABLESPACE tbspc\_name OWNER TO { new\_owner | CURRENT\_USER | SESSION\_USER }

ALTER TABLESPACE tbspc\_name SET tablespace\_option = value…

ALTER TABLESPACE tbspc\_name RESET tablespace\_option…

TABLEへの指定方法

CREATE TABLE table1 TABLESPACE tbspc\_name LOCATION ‘/data/dbs’; --

ALTAR TABLE

CREATE DATABASE WITH TABLESPACE tbspc\_name; -- データベースにデフォルトのテーブルスペースを指定

### ストレージ

#### 関連コマンド

SET STORAGE

列の保管モードを設定。インラインで保持するかTOASTなどで保持するかを指定する。

PLAIN integerのような固定長の値に対して使用します。インラインで保持され，圧縮されない。

MAIN インラインで保持されていて，圧縮可能なデータに使用します。

EXTERNAL 圧縮されていない外部データに使用します。

EXTENDED 圧縮された外部データに使用します。 EXTERNALより処理速度は落ちる。

PLAIN以外の保管をサポートするほとんどのデータ型におけるデフォルト。

※テーブルの現状はメタコマンド \d+ tableName で確認できる。

### インデックスファイル

CREATE INDEXで作られたインデックスの実体。

## バックアップ

### 概要

なお，データベースクラスタのバックアップ方法では，ディレクトリを直でコピーする方法もある。

tarやrsyncを使用

・PostgreSQLの停止中のみバックアップが可能

・論理バックアップに比べてバックアップデータのサイズが大きい

・「postgresql.conf」等の設定ファイルも一括でバックアップされる

・PostgreSQLのメジャーバージョンが異なる場合はリストアできない

|  |  |  |  |
| --- | --- | --- | --- |
| バックアップ | | | 復旧方法 |
| 種類 | 方式 | 方法 |
| オフライン  (PostgreSQLを停止) | 物理 | tar,rsyncなどによるディレクトリコピー | コピーしたディレクトリの配置 |
| オンライン  (PostgreSQLを稼働させたまま) | ・各種パラメータの設定  ・pg\_start\_backup(), pg\_stop\_backup()  ・又はpg\_basebackup（ベースバックアップ） | PITR  WALのコピー  recovery.confファイルの設定 |
| 論理 | pg\_dump　pg\_dumpall | psql  またはpg\_restoreコマンド |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| バックアップ手法 | サイズ | 時間 | リカバリ時間 | コスト | 業務影響 |
| 論理 | 小 | 小 | 大 | 小 | 小 |
| 物理（オフライン） | 大 | 中 | 小 | 小 | 大 |
| 物理（オンライン） | 大+α | 中～大 | 中 | 中～大 | 小 |

情報元： https://qiita.com/bwtakacy/items/65260e29a25b5fbde835#バックアップ手法の比較

手法

フルバックアップのみ

システムを停止する必要がある。データ量が多く，バックアップに時間がかかる。

週一，月一など規模に合わせ，比較的低めの頻度で行う。

フルバックアップ + 差分バックアップ

復元時の負荷が少なめだが，バックアップ時の負荷はやや大きい。

フルバックアップ + 増分バックアップ

バックアップ時の負荷は少ないが，復元時の負荷が大きい。

### 用語（バックアップ）

#### WAL

Write Ahead Logging

実際のデータベース更新処理(INSERT，UPDATE，DELETE)の前にログを記述するシステム。

更新処理の度にLSNというカウンタ値がカウントアップされ，WALレコードが挿入される。

データ更新とWALへの書き込みで負荷が倍になる感じがするが、実際はこの方式の方が性能は良い。

pg\_wal/archive\_statusディレクトリに.readyファイルが作成される。

補足

ハードディスクにはシーケンシャルアクセスによる書込みの方が高速である為，WALの書込みを同期で行い，実際のデータ領域へのランダムアクセスは非同期で行う。

#### LSN

Log Sequence Number

具体的には 000000010000000000000001，000000010000000000000002 といった数値になる（らしい）

#### その他

ホットスタンドバイ

サーバーがアーカイブリカバリを実行している最中にサーバーに接続し読み取り専用の問い合わせを実行することができる機能

オンラインバックアップ

PostgreSQLを稼働させたままバックアップを取る事

ストリーミングレプリケーション；streaming replication

データベースクラスタ全体を対象にデータを複製する機能

デフォルト状態から数ステップで使用可能。冗長性の確保にも使えるらしい。

（参考）<https://changineer.info/server/postgresql/postgresql_streaming_replication01.html>

ベースバックアップ

データベースクラスタ全体のバックアップを取得する。

WALも自動でアーカイブされる。その為，実質的にはバックアップ取得中の更新もバックアップ対象といえる。

論理バックアップ

データベースクラスタをスクリプトファイルへ抽出する。

その際，設定ファイルなどは含まれない。

物理バックアップ

データベースクラスタのデータを直接バックアップする。

### 具体的な方法

特に理解しにくいもののみを記述

#### コールドバックアップ

オフライン（サーバー機を直接操作）での物理バックアップの事。

psql -U postgres -h 192.168.1.1 -c ‘SHOW data\_directory;’ # データベースクラスタのパスを確認

# 注意。スーパーユーザーでしか実行できない。もちろんpsqlの対話コマンドで実行しても良い。

> /var/lib/postgresql/11/main # 結果が右のようだったとする。

pg\_ctl stop # postgresqlを停止

pg\_ctl #

tar cvJf pg\_full.tar.xz /var/lib/postgresql/11/main # 丸ごとtarでアーカイブ作成

pg\_ctl start # postgresqlの動作を開始

テーブル空間も忘れずにバックアップを取得するようにする。

データベースクラスタ直下のpg\_tblspcに入っているのはシンボリックリンクなので，実体データのバックアップをとるようにする。

補足

・近年のDebian系ではsystemctl start/stop postgresql を使った方が楽かもしれない。

#### PITR

オンライン物理バックアップの事。実際これが一番理解しにくい。

Point In Time Recovery　指定した時間まで遡ってリカバリする，の意

注意点

・パフォーマンス，リカバリ時間，ディスク容量を考慮してバランスの取れた設定にする必要がある

・WAL領域のデータも確実にバックアップをとるようにする。

・定期的に実行することで、WALアーカイブの容量を抑え、復旧処理の時間を短縮できる。

・ベースバックアップを取得した以降のポイントなら，障害発生の直前に近い任意のポイントに復旧できる。

・古いWALファイルが削除されないようにアーカイブとして保存する必要がある

事前準備（設定）

ログファイルをアーカイブして保存する設定を行っておく。

[ postgresql.comf ]

wal\_level = archive

archive\_mode = on

archive\_command = '/bin/cp %p /home/kazu/local/pg952/arch/%f'

・wal\_levelをreplicaまたはlogical，archive\_modeをonまたはalwaysにする。

・archive\_commandは例えば ‘cp %p [WALアーカイブの保存先のパス]/%f’

次ページに続く

～PITR　つづき

ベースバックアップ

1) pg\_start\_backup()関数を呼び出す (例) SELECT pg\_start\_backup('ラベル名');

なお，pg\_start\_backup()では以下の処理が行われる。

・共有メモリ上のステータスを「バックアップ中」にする

・WALをスイッチする　？？

・チェックポイントを発行し，そのLSN；Log Sequence Numberを保持。

・上のLSNを元にWALファイルを特定し，そのLSN番号の文字列はbackup\_labelというテキストファイル内に書き出される。また，backup\_labelというファイルはベースバックアップに含まれる。

・LSNを返却

2) データベースクラスタ全体の物理バックアップ（ディレクトリコピー）を取得

これを具体的にどうやってやるのかの情報がない 2021-10-09

terminal:

$ cp –r ${PGDATA} ./backup/ # どうやらこうゆう感じらしい

3) pg\_stop\_backup()関数を呼び出す (例) SELECT pg\_stop\_backup();

なお，pg\_stop\_backupでは以下の処理が行われる。

・共有メモリ上のステータスを元に戻す。

・backup\_labelを読み込み，開始位置（LSN）を取得する。

・上のLSNを含むWALレコードを書き出す

・WALをスイッチする

・バックアップ履歴ファイルを書き出す

・上２つのWALファイルがアーカイブされるのを待つ

・WALレコードの書き出しで取得したLSNを返却する。

補足

・pg\_basebackupでは上記の**一連のベースバックアップ処理を自動的に行う**。

※手順2) でpg\_basebackupを実行する訳では無いので勘違いしないように

詳細：[pg\_basebackup](#_pg_basebackup)

・あくまで物理バックアップなので，異なるバージョンのリストア／バックアップには非対応

リカバリ

手動でrecovery.confファイル（必須）を作成して配置。（どこに？）

リストアポイントは、「recovery.conf」のrecovery\_target\_timeパラメータやrecovery\_target\_xidパラメータで指定。

流れ（本より）

・pg\_controlファイルを読み込む

・recovery.confを読み込む

・backup\_labelファイルを読み込む。

・pg\_controlファイルを更新し，backup\_labelを削除する。

・必要なWALを繰り返し適用する。

### recovery.conf

PITRによるリストア処理で用いる設定ファイル。

項目

recovery\_target\_time リストアポイントを指定する。

recovery\_target\_xid

### 関連コマンド

#### pg\_restore

論理バックアップのファイルからリストアを行う。

対象はテーブルデータ，ラージオブジェクト，シーケンス値

ロール情報は対象外。

構文：pg\_restore [接続オプション] [オプション] [ファイル名]

オプション

-d database\_name リストアを行うデータベースを指定。 --dbname=database\_name

省略すると標準出力にテキスト形式のSQL文が出力

-c リストア前に既存のデータベースオブジェクトを削除 --clean

-j ジョブ数 リストア処理を同時に実行するジョブ数を指定 --jobs=ジョブ数

$ pg\_restore -j 2 -d db001 db001.bak

#バックアップファイル「db001.bak」をデータベースdb001へ、同時に2つのジョブを実行してリストア

-1 リストア処理を1つのトランザクションとして実行。--single-transaction

例

$ pg\_restore -U postgres -d database1 file.dump # database1に対してfile.dumpからリストア

・**tarとバイナリ形式**（カスタム形式とも呼ばれる？）のバックアップファイルをリストアする。テキストは非対応。（テキスト形式の時はpsqlコマンド）

・ラージオブジェクトやシーケンス値もリストア対象となる。

・psqlと異なり，-fオプションは無いらしい。

#### pg\_dump

データベース単位の論理バックアップ（スクリプトファイルへ抽出）。

構文：pg\_dump [接続オプション] [オプション] [データベース名]

オプション

-f file\_name 出力先のファイル名を指定。省略すると標準出力に出力。≒ pgdump > file\_name

-d db\_pattern 対象のデータベース名を指定。（-dオプション仕様） --dbname=dbname

$ pg\_dump -d database1

-F 出力フォーマットを指定。省略するとテキスト(plain)になる。

p **テキスト形式**(def.) plain

c バイナリ custom

t tar形式 tar

-a データのみを取得テーブルデータ、ラージオブジェクト、シーケンス値が対象。--data-only

（関連）--section

-t table\_pattern（pattern文字列）バックアップ対象のテーブルを指定する。--table = table\_pattern

-T table\_pattern（pattern文字列）**対象外**のテーブルを指定する。 --exclude-table = table\_pattern

-n schema\_pattern（pattern文字列）バックアップ対象のスキーマを指定する。--table = schema\_pattern

-N schema\_pattern（pattern文字列）**対象外**のスキーマを指定する。 --exclude-table = schema\_pattern

・ラージオブジェクトをバックアップすることもできる。（むしろデフォルト）

・.shファイルのような，コマンドの羅列ファイルが作成される

・データベースクラスタを対象としているが，バックアップ対象はデータベース単位

・テーブル定義のみのバックアップも可能。

・実行には各テーブルの読み取り権限さえあれば良いが、現実的にはスーパーユーザーが適切。

・ロールやテーブルスペース定義は取得できない（pg\_dumpallでは取得する）

（参考）https://www.postgresql.jp/document/11/html/backup-dump.html

# テキスト形式でdatabase1のバックアップファイルを作成

$ pg\_dump -U user1 -h 192.168.1.30 -Fp -f file.dump database1

復元方法

$ pg\_restore -U postgres -d database1 file.dump # バイナリかtarの場合はpg\_restoreを用いる

$ psql -f dumped\_file database1 # テキスト形式の場合はpsql -f を用いる

#### pg\_dumpall

全てのデータベースの論理バックアップを作成する。

構文：pg\_dumpall [接続オプション] [オプション]

・PostgreSQLサーバーを**稼働させたままで行わなければならない**。

・対象はテーブルデータはもちろん，ロールやテーブルスペース定義，グローバルオブジェクト（ロールとテーブル空間）も含む。（pg\_dumpでは取得できない）。

postgresql.conf等の設定ファイルはバックアップ対象外。

・バックアップ時のPostgreSQLと比べてリストア先のバージョンが新しい場合もリストアできる。

（ディレクトリコピーによる物理バックアップではバージョンが異なるとリストアできない）

・バックアップの形式は**テキスト形式のみ**対応。

・取得されるロックはaccess share。他ユーザーのデータベースアクセスを妨げることはない。

オプション

-g グローバルオブジェクトのみをバックアップ対象に

-f file\_name 出力先のファイル名を指定。省略すると標準出力に出力。

$ pg\_dumpall -f db\_all.bak # pg\_dumpall > db\_all.bak と同等

テキスト形式のダンプはpsql -f でリストアできる（pg\_restoreでは.tarかバイナリのみ）

$ psql -f dumped\_file database1 #

#### pg\_start\_backup

PITR手法によるバックアップの開始を通知する

pg\_stop\_backupと併せて通知関数と呼ぶ（らしい）

※注意：pg\_dumpと混同しないように！pg\_dumpは論理バックアップ。

構文：

第二引数 true：積極的なチェックポイント処理を行う(def.) false：リソース的に控えめな処理

第三引数 true：並行したバックアップを許可しない(def.) false：許可する

・pg\_start\_backupはチェックポイントの実行も行う。 チェックポイント=WALにおけるリカバリポイント

・PostgreSQLを稼働したまま行う。

・バックアップ取得中に作成されたWALもアーカイブ対象となる。

#### pg\_basebackup

PITR手法によりベースバックアップ（データベース全てのバックアップ）を行う。

オンラインで行う物理バックアップ。

pg\_start\_backup～pg\_stop\_backupまでの一連の処理を実行する。

・PITR(Point In Time Recovery)に使用

・デフォルトではWALファイルを含めたバックアップが取得される。

補足：postgresql.confで設定する。

・スーパーユーザー権限またはREPLICATION権限が必要。

・リモートによるバックアップの取得も可能。

・ストリーミングレプリケーション(データベースクラスタ全体を対象にデータを複製する機能)において、プライマリのベースバックアップを取得する際に使用される（2021-09-03意味不明）

・**ver9.1**で追加された機能

構文：pg\_basebackup [オプション]

オプション

-D バックアップ保存先を指定。対象ディレクトリが空でない場合はエラー。

$ pg\_basebackup -D /backup

-X method WALバックアップの方法を指定 (also) --wal-method= method

method

STREAM 並行してWALファイルをアーカイブ。(also) s (def.)

NONE WALファイルをバックアップ対象から外す。(also) n

FETCH 最後にWALバックアップを行う。(also) f

-P 進捗を表示 ふつう-vでは？？

#### pg\_rman

#### pg\_current\_wal\_insert\_lsn

WALレコードの次の書き出し位置（LSNの文字列）を確認する。

#### pg\_walfile\_name

具体的なWALファイル名を確認する。

=# SELECT pg\_walfile\_name(pg\_current\_wal\_lsn());

> 000000010000000000000003

#### pg\_waldump

WALを可読性が高い表現で表示する

（公式情報）<https://pgsql-jp.github.io/current/html/pgwaldump.html>

#### pg\_archivecleanup

ver9.5から

## クライアントコマンド

≒bashやcommand promptのコマンドと考えて良い。

### 接続オプション

ほぼすべてのクライアントコマンドで利用される，postgreSQLへの接続オプション

|  |  |  |  |
| --- | --- | --- | --- |
|  | 説明 | 詳細 | 環境変数 |
| -U | ユーザー名を指定 | sudo の -u オプションと混同しないように | $PGUSER |
| -h | ホスト名を指定 | def.=UNIXドメインでの接続 | $PGHOST |
| -d | データベース名を指定 | def.=データベースのユーザー名 | $PGDATABASE |
| -p | ポート番号を指定 | def.=5432 | $PGPORT |

補足

-d オプションの仕様

-d db\_name で指定された場合は、そのデータベースが対象になる。

-d が省略された場合は$PGDATABASEで指定してあるデータベースが対象になる。

$PGDATABASE が空白の時はpsqlのユーザー名と同名のデータベースが対象になる。

※これを本ドキュメントでは「-dオプション仕様」と呼ぶことにする。

ついでに接続オプションではないが，共通オプション。

-v 詳細表示

### psql

psqlはPostgreSQLのターミナル型のフロントエンド。

psql -U sakura -h localhost sakuradb

構文：psql [接続オプション] [オプション] [データベース名 [ユーザー名] ]

オプション

-c command 接続と同時にデータベースコマンドを実行して終了　　--command=command

$ psql -c ‘\l’ # 現在のデータベースのリストを表示

-l 全てのデータベースのリストを表示した後，psqlを終了する。 --list

-f fileName 指定したファイル実行し，結果を表示し，psqlを終了　 --file=fileName

psql -f 'sample.sql'

psql database1 < 'sample.sql' # リダイレクト(<) を使用しても同じ結果になる。

-s 各コマンドで実行するかキャンセルするかの確認をする　--single-step

psql --single-step -f 'sample.sql' -U postgres # sample.sqlの実行

-S シングル行モード。（非推奨） --single-line

-V バージョンの表示 --version

-1 複数のコマンドを1つのトランザクションで実行する。　--single-transaction

psql --single-transaction -f 'test.sql' # test.sqlのSQL文が1つのトランザクションとして実行される

処理中にエラーが発生した場合は全ての処理が取り消される。

¥! [command] サブシェルでシェルコマンドを実行する。commandを省略した場合は処理をサブシェルに渡す。終了するとpsqlが再開する。

リダイレクトの使用？2021-08-30

### initdb

**データベースクラスタ**を作成する。

コマンドプロンプト，terminalからも実行できる？？

（関連）creatdb

構文：initdb [オプション] *directory\_name*

オプション

-D

-D *directory\_name* データベースクラスタを作成するディレクトリを指定　--pgdata=*directory\_name*

指定しない場合は，環境変数$PGDATAの値が適用される。

-E *encording* データベースのエンコーディングを指定する。 --encording=*encording*

省略した場合はOSのロケールから自動的に設定される。

initdb --encoding=EUC\_JP

日本語対応：EUC\_JP，EUC\_JIS\_2004，SJIS，SHIFT\_JIS\_2004，UTF-8

--locale=*locale* ロケールを指定する。--locale=C はロケールを無効にする。

省略した場合はOSのロケールが使われる。

**SJISは設定できない。**

注意：ロケールと互換性がないデータベースエンコーディングを設定しているとエラーを起こすことがある

--no-locale ロケールを無効に。--locale=Cと同等。

-U *user\_name* 作成するデータベースの管理ユーザーを指定。--username=*user\_name*

省略するとinitdbコマンドを実行したOSユーザが指定される

-k データベースのチェックサムを有効にする --data-checksums

-X *directory\_name* WALを格納するディレクトリを変更する。 --waldir=*directory\_name*

補足

・ローカルホスト機のみで有効で，ネットワーク経由で他のホストから実行する事はできない。

・既にデータベースクラスタが作成されているディレクトリを指定した場合initdbは実行されない。

・pg\_ctl initdb でも同様。

### createdb

データベースを作成する

SQLコマンドCREATE DATABASEのラッパ。全く違いなし。

構文：createdb [connection-option...] [option...] [*db\_name* [description]]

接続オプション

-h ホスト名またはIPアドレスを指定

オプション

db\_name 省略すると現在のOSのログインユーザー名のデータベースを作成する。

-E encoding エンコードを指定。 --encoding=encoding

-O userName OWNERオプションに同等。データベースの所有者を指定。 --owner=userName

スーパーユーザー権限が無ければ，自分以外を所有者に指定する事はできない。

-l localName ロケールを指定する --locale=localName

-T template0でtemplate0を指定する必要がある。

-T tempName テンプレートを指定する。--template=tempName

※-tオプションが存在しない癖に，なぜか大文字Tとなっている

-E，-lオプションはテンプレートがtemplate0の時のみ有効

（参考）[CREATE DATABASE](#_CREATE_DATABASE)

### createuser

※データベースコマンドのCREATE ROLEに合わせてユーザーでなくロールと呼ぶ。

構文：createuser [接続オプション] [オプション] [ユーザー名]

オプション

--interactive 対話形式でユーザーを作成。ver9.1までデフォルトだった。

注意：iは--inherit，Iは--no-inherit。--interactiveに1文字オプションは無い。

-P 対話形式でパスワード設定を行う。パスワード認証を予定する場合のみ。--pwprompt

-l 新しいロールに対してログイン権限を与える。（def.）　--login

※CREATE ROLEではNOLOGINが規定値な点に注意

-L ログインを禁止する。　--no-login

-d CREATEDB権限を与える --createdb

-D 新しいロールに対してデータベースの作成を禁止する（デフォルト） --no-createdb

-s スーパーユーザーとして作成。 --superuser

-S スーパーユーザーにしない（デフォルト） --no-superuser

-r CREATEROLE権限を付与 --createrole

-R 新しいロール（ユーザー）の作成を禁止する（デフォルト） --no-createrole

-U name createuserコマンドの実行ロール（＝psql接続ユーザー名）を指定する。

（参考）[CREATE ROLE](#_CREATE_ROLE_1)

補足

・クライアントコマンドでは修正（alter）ができないかも？ALTER ROLEが必要2021-08-31

・スーパーユーザー権限を持ったロールを作る際には，実行ロールにもスーパーユーザー権限が必要。

・実行時のOSユーザーの権限などは全く影響を及ぼさない。あくまで，オプションなどで指定する。

### dropuser

PostgreSQLのユーザアカウントを削除する。

構文：dropuser [接続オプション] [オプション] [ユーザー名]

オプション

-i 削除前に確認メッセージを表示させる --interactive

dropuser -U postgres -i user

補足

・PostgreSQLのユーザーを削除できるのはスーパーユーザーとCREATEROLE権限があるユーザーのみ

・対象ユーザーがスーパーユーザーな場合は，dropuserの実行ユーザーもスーパーユーザーである必要がある。

（参考）[DROP](#_DROP)

### dropdb

PostgreSQLのユーザアカウントを削除する。

DROP DATABASEのラッパー。

構文：dropdb [接続オプション] [オプション] [データベース名]

例

sudo -u postgres dropdb plcpi # 一時的にpostgresユーザーになってデータベースを削除

オプション

-i 実行前に確認用の対話メッセージを表示　--interactive

・データベースが削除できるのはスーパーユーザーとデータベースの所有者のみ

（参考）[DROP](#_DROP)

### pg\_ctl

サーバーの起動・シャットダウン(停止)や，データベースクラスタの初期化などを行う。

また，起動中の状態を表示するための管理ツールとして使用。

・postgreSQLが起動するローカル機からしか実行できない。

・データベースを作成したユーザーで行う必要がある。（rootでもダメ）

注意点

・pg\_ctlの実行ファイルが格納されているディレクトリへパスを通す必要がある

・毎回 -Dオプションを使用するか，環境変数 $PGDATA を設定する必要がある。

/usr/local/pgsql/bin/pg\_ctl stop # フルパスで指定しても良い

/usr/lib/postgresql/ver/bin/pg\_ctl stop # ラズパイ、Ubuntuではここだった 2021-10-05

・postgreSqlのスーパーユーザーでないと実行できない？

su - postgres -c "pg\_ctl stop”

共通オプション

-D name データベースクラスタを指定する --pgdata=data\_directory

補足

debian系ではsu postgresにしないとpg\_ctlを実行できなかった。2021-10-05

postgresql.serviceの起動／停止はsystemctlでやった方がはるかに簡単。

インストール時にpg\_ctlcluster 11 main startというコマンドを実行しろと言っているが？2022-05-21

次ページへ続く

pg\_ctlつづき

サブコマンド

initdb 新規データベースクラスタを作成。≒initdbコマンド

start バックグラウンドでPostgreSQLサーバーを起動する

stop サーバーをシャットダウン。

構文：pg\_ctl stop [オプション]

オプション

-m モード シャットダウンの方式を指定する(s,f,i)

-W シャットダウンの完了を待たずにコマンド発行元に制御を戻す

-t 最大待ち時間秒 シャットダウンの完了を待つ時間。指定が無い場合は60秒

指定時間内に停止が完了しない場合は失敗として処理されるが，停止処理は行われる。

・コマンド実行直後から、新しいクライアントからの接続は禁止

$ pg\_ctl stop -m smart # kill TERM と同等

restart 再起動する。stop+start

オプション　stopサブコマンドのオプションと同じ。

reload 設定ファイルpostgresql.confとpg\_hba.confを再読み込みする

-D *name* データベースクラスタを指定する --pgdata

指定しない場合は環境変数｢$PGDATA｣が選択される

変更が反映されないパラメータもある。

status PostgreSQLサーバーが稼働しているか確かめる

kill プロセスにシグナルを送信する

pg\_ctl kill TERM 1234

|  |  |  |  |
| --- | --- | --- | --- |
| シグナル | pg\_ctl stop | -m | 説明 |
| TERM | smart | s | 実行中の処理を終了し，全ての接続が切断されてからシャットダウン。  （クライアント接続が残っている限りシャットダウンされない） |
| INT | fast（高速） | f | 高速シャットダウン。すべての処理がロールバック。(def,) |
| QUIT | immediate | i | 一切の終了処理無し。クラッシュと同等で次回起動時に**回復処理が必要**。 |
| HUP | reload | × | 設定ファイルを再読み込み。 |

補足：INTインターラプト（割込み），HUPハングアップ（応答なし）

※systemctlに似ているように感じた 2021-08-30

### pg\_controldata

initdbで初期化されるデータベースクラスタ全体の**制御情報を取得**するコマンド

構文：pg\_controldata [オプション]　[ [-D] [ディレクトリ名] ]

ディレクトリ名はデータベータクラスタの保存ディレクトリ。

ディレクトリ名を省略すると，環境変数「$PGDATA」に設定されているディレクトリの制御情報を取得。

注意点

・initdbを実行したデータベースクラスタの管理ユーザーで行う必要がある。

・データベース単位の情報の取得はできない。あくまでデータベースクラスタ単位。

### vacuumdb

データベースのバキューム（不要領域の回収）を行う

構文：vacuumdb [接続オプション] [オプション]　[ -t テーブル名, … [ (column1[, …] ) ] ]　[データベース名]

オプション

-a **全てのデータベース**を対象とする　--all

-f 回収した**不要領域を削除**する。コマンド実行中はテーブルに排他ロックをかける　--full

-z 不要領域の回収を行い，**統計情報の収集・更新**を行う　--analy**z**e

-Z 不要領域の回収はせずに，**統計情報の収集・更新のみ**を行う --analyze-only ≒ANALYZE

補足

・VACUUMへのラッパコマンドらしい。VACCUM = vaccumdb -a

・テーブル指定を省略すると，現在のユーザーが権限を持つ現在のデータベース内が対象。

### pg\_rman

## データベースコマンド

### 概要

DDL(Data Definition Language)

テーブルなどのオブジェクトを作成するためのコマンド群

例：CREATE, DROP, ALTER

DL=Disney Landで覚えようかな。たくさん「オブジェクト（ミッキーとか）」を作る

DML(Data Manipulation Language)

データの挿入／更新／削除などを行うためのコマンド群。

例：SELECT，INSERT，UPDATE，DELETE

DCL(Data Control Language)

トランザクション関連のコマンド

例：COMMIT，ROLLBACK，GRANT，REVOKE，BEGIN

### 一般コマンド

#### メタコマンド

これらコマンドを実行するときは最後に；（セミコロン）は必要ない。

（大文字小文字を区別する）

\l データベース覧表示。

\d リレーション（テーブル，ビュー，シーケンス）の一覧。

\dp リレーションのアクセス権の一覧を表示。 \zでもよい

\d [*tb\_pattern* ] リレーションの列名などの情報が確認できる。列名、型、照合順序、Null許容、def.値

\d+ [*tb\_pattern*] \dで見られる情報に加え、ストレージ、統計の対象、説明が確認できる。

\dt テーブルの一覧

\du ユーザーの一覧

\dn スキーマの一覧

pattern文字列 （patternパラメータ）

各種 \d コマンドでは，「pattern文字列」を渡す事で，関連するオブジェクトを複数対象とできる。

\* 0文字を含む任意の１文字

? 任意の一文字

\q postgresqlの終了

\? メタコマンドの一覧と簡単な説明を表示

\h [SQLコマンド] SQLコマンドのヘルプ一覧。「SQLコマンド」は必須オプションに近い。

\! [SQLコマンド] OSコマンドの実行結果

postgres =# \! ls -- 例として ls コマンドの実行

\c *db\_name* データベースの切り換え

\password 現在ログイン中のユーザーのパスワードを変更する。

\set 環境変数を設定する。

#### \copy

クライアント側のファイルとサーバー側テーブルでデータのコピーを行うメタコマンド。

構文 \copy テーブル名 TO {ファイル名 | stdout} [with] [オプション]

構文 \copy テーブル名 FROM {ファイル名 | stdout} [with] [オプション]

オプション（withは有っても無くても良いらしい）

補足：基本的にオプションはデータベースコマンドであるCOPYと同じ

csv ファイルをcsv形式で扱う

delimiter [as] ‘char’ 区切り文字の指定

・ファイル名に ’’や ””は不要らしい（使用できない？） 一方COPYでは’ を使用する。

・特別な権限を必要としない。一方COPYコマンドではスーパーユーザー権限が必要（標準出力の場合は不要）

・デフォルトではタブ区切り（？）

\copy colors FROM colors.txt CSV DELEMITER E’\t’ HEADER -- タブ区切りテキストを読み込み

（関連）[COPY](#_COPY)

（公式）<https://www.postgresql.jp/document/11/html/app-psql.html#APP-PSQL-META-COMMANDS-COPY>

#### 特別な文字列

|| 2つの文字列を結合する

=# SELECT c || v FROM table1; -- cの値vの値　といった結果が出力される

=# WHERE name = 'SATO' || ' ' || 'HITOMI'; -- 「SATO HITOMI」が対象になる

:: データ型を指定（キャスト）する

=# SELECT myfield::integer FROM mytable

myfieldをintegerとして表示（SELECT）する。

-- コメント

IS NULL値の判定に使用する。IS NULL/IS NOT NULL

補足：NULL値の判定に=や<>を用いると，その結果がNULLとして返る。

#### 正規表現

POSIX正規表現

構文：~ '条件'

|  |  |  |
| --- | --- | --- |
| 演算子 | 説明 | 例 |
| ~ | 正規表現にマッチ，大文字小文字の区別あり | 'thomas' ~ '.\*thomas.\*’　？？？ |
| ~\* | 正規表現にマッチ，大文字小文字の区別なし | 'thomas' ~\* '.\*Thomas.\*'　？？？ |
| !~ | 正規表現にマッチしない，大文字小文字の区別あり | 'thomas' !~ '.\*Thomas.\*'　？？？ |
| !~\* | 正規表現にマッチしない，大文字小文字の区別なし | 'thomas' !~\* '.\*vadim.\*'　？？？ |

例

SELECT count(\*) FROM city WHERE eng !~ 'RI'; --cityテーブルのeng列にRIを含まないレコード数を返す

※つまり eng=’RI’ の「=（イコール）」の代わりに「~」を使う

### SELECT

SELECTは使い方が多彩なので，独立した項目を作る。

なお，SELECTはData Manipulation Language（データ操作言語）

#### 基本

構文：SELECT {列名 [AS 別名][, 列名 [AS 別名]...] | \*} FROM テーブル名 [WHERE 条件];

select \* from table1; 基本中の基本。table1のすべての列のすべてのレコードを表示

SELECTの対象となるテーブルを指定する。

通常はWHEREを介して抽出条件を指定する。

=# SELECT \*

FROM fruits f CROSS JOIN kansou k WHERE f.name = 'banana';

JOINなどを用いて，複数のテーブルを結合することもある。　（参考）[架空テーブルの作成](#_架空テーブルの作成)

クエリ内変数

変数の宣言より先に，変数を使うことができる。

SELECT s1.name, tel from *table1* t1, *table2* t2 WHERE t1.no = t2.no; --変数を使った例

SELECT table1.name, tel from *table1*, *table2* WHERE table1.no = table2.no; --変数を使わない例

#### 条件

WHERE 最も一般的で簡単な条件の指定方法

WHERE id < 50 AND groupName = '総務部'; -- 複数の条件の例

WHERE pending\_restart'; -- boolをwhereにする場合

WHERE NOTpending\_restart'; -- boolをwhereにする場合（NOT）

CASE

CASE WHEN condition1 THEN result

[WHEN condition2...] --第二条件。C言語のelse ifみたいな。

[ELSE result]

END

例

SELECT id FROM sample WHERE CASE WHEN age < 30 THEN FALSE ELSE TRUE END;

UPDATE sample SET boss = (CASE WHEN groupNo = 2 THEN '佐藤' ELSE '鈴木' END);

--boss列はgroupNoの値によって変化。　⇒ groupsテーブルみたいなのを作って参照させるべきでは？

次ページに続く

～SELECT　条件　続き

LIKE 前方一致や後方一致。

構文：LIKE ‘パターン’ [ESCAPE escape-character]

構文：NOT LIKE ‘パターン’ [ESCAPE escape-character]

・大文字と小文字は**区別されない**。

・WHERE id ~ 'A001'　と同等なのは　LIKE　‘%A001%’ （~は正規表現）

・postgreSQLの拡張で，ILIKEというものがあり，こちらは大文字と小文字の区別をする。

|  |  |
| --- | --- |
| 文字 | 意味 |
| % | 0文字以上の文字  ※Linuxでは「\*」，正規表現では「.\*」に近い。**0文字も含む**点に注意。 |
| \_（アンダーバー） | 任意の一文字 |
| ~~ | LIKEと同等（PostgreSQL独自の表現） |
| ~~\* | ILIKEと同等（PostgreSQL独自の表現） |

SIMILAR TO

標準SQL（SQL:1999）の正規表現定義に沿ったもの。

≒LIKE　大文字と小文字は**区別されない**。（LIKE + 正規表現？）

|  |  |
| --- | --- |
| 文字 | 意味 |
| % | 任意の文字列　という解説(from ping-t) があったが、正規表現で%は使わないはず |
| | | 二者択一（2つの選択肢のうちいずれか）を意味します。 |
| \* | 直前の項目の0回以上の繰り返しを意味します。 |
| + | 直前の項目の1回以上の繰り返しを意味します。 |
| ? | 直前の項目の0回もしくは1回の繰り返しを意味します。 |
| {m} | 直前の項目の正確なm回の繰り返しを意味します。 |
| {m,} | 直前の項目のm回以上の繰り返しを意味します。 |
| {m,n} | 直前の項目のm回以上かつn回以下の繰り返しを意味します。 |
| 丸括弧() | 項目を1つの論理項目にグループ化することができます。 |
| 大括弧式[...] | は、POSIX正規表現と同様に文字クラスを指定します。 |

公式情報：https://www.postgresql.jp/document/11/html/functions-matching.html

#### DISTINCT

対象列の値が重複されるレコードを排除して返す。

⇒ いっそSELECT DISTINCT と覚えた方が良い。

構文：SELECT DISTINCT [ ON (対象列名,...) ] SELECTの対象列,...　FROM table；

・DISTINCTはSELECTの直後に記述する。

・ONを省略した場合にはSELECTの対象列の重複が除去される。

SELECT DISTINCT ON (club) club from sample; --重複が削除されたclub列の値が表示される

SELECT DISTINCT ON (groupNo) groupNo FROM sample WHERE sex = '男';

SELECT DISTINCT groupNo FROM sample WHERE sex = '男'; -- 非重複対象とSELECT列が同じ場合

#### LIMIT

指定された行数より多くの行を返すことがなくなる。（少ないことはありうる）

構文：LIMIT { number | ALL }

既定値はLIMIT ALL で制限なし

注意点：

・データが件数分存在しない場合は、存在する件数分のみ取得される。（エラーにはならない）

#### OFFSET

返す行の開始位置を移動させる。

構文：OFFSET number

#### ORDER BY

構文：ORDER BY 対象列名 [**ASC** | DESC][, 対象名 [ASC | DESC]...]

ASC昇順（小→大）[def.] DESC降順（大→小）

SELECT \* FROM table1 ORDER BY id, age DESC; --片方が降順，もう片方が昇順

LIMIT 取得するレコード件数に上限を設ける

構文：LIMIT 上限件数 [OFFSET 結果の取得時にスキップする件数]

SELECT name FROM member ORDER BY id LIMIT 3 OFFSET 1;

-- idを降順で3件取得。但し，取得結果を1つオフセットする。⇒ 取得件数は依然として3件である点に注意。

例

SELECT no FROM table1 ORDER BY groupNo,no; --ソート対象を2つ指定する場合

#### SHOW

configファイルの設定値を確認する

postgres=# SHOW shared\_buffers;

postgres=# SHOW ALL; # 全ての設定項目の表示

※スーパーユーザー権限が必要な場合

$ sudo -u postgres psql -d dbname1 # これが一番確実に思える

memo

よく使いそうなものをピックアップ

config\_file，ssl\_cert\_file,

hba\_file,ident\_file, log\_filename

#### EXISTS

１件でも条件に一致するレコードが存在するかどうか。true/falseを返す，と考える。

副問い合わせの結果が存在するかどうか。任意のSELECT文。副問い合わせ。

副問い合わせの対象列において，**テーブル名が指定されている**場合。（結果的にINに近い）

サブクエリ（）内で1度でもテーブル名が出てきたら，この条件は適用になる？

EXISTS ： 対象テーブルから条件に一致するデータのみを取得

NOT EXISTS ：対象テーブルから条件に一致しないデータのみを取得

SELECT name FROM animal WHERE EXISTS (SELECT zooA.name FROM zooA WHERE num = 2);

--対象テーブル名がある：

副問い合わせの対象列において，**テーブル名が指定されていない**場合はTRUE/FALSE

EXISTS ： サブクエリの結果が1件でもあればtrueとなり，主クエリは全て対象となる。

NOT EXISTS ： サブクエリの結果が1件も無ければtrueとなり，主クエリは全て対象となる。

SELECT name FROM animal WHERE EXISTS (SELECT name FROM zooA WHERE num = 2);

--対象テーブル名が無い→num=2のレコードが1件でもあればanimalテーブルの全レコードが対象：

EXIST SELECT 1 WHERE　最大1つの出力行を生成する（true/falseなので結果は変わらない）

これが対象テーブルの条件が指定されていない場合の効果を端的に表すコマンド。

例：

SEL ECT name FROM animal WHERE NOT EXISTS (SELECT name FROM zooA WHERE zooA.name = animal.name);

-- かなり複雑な例。サブクエリはzooAにもanimalにも存在するnameを返す。animalテーブルからそのサブクエリの結果を含まないレコードをSELECTする。⇒ zooAには存在しないanimalテーブルのnameを返す。

#### IN

副問い合わせの結果に一致するものを対象とする

SELECT \* FROM animal WHERE name IN (SELECT name FROM zooB);

--zooBテーブルのname列 = animalテーブルのname列　であるレコードが対象になる

・INでもNOT INでも副問い合わせの結果**NULL値になるレコードは無視**される。

#### FOR UPDATE

ROW SHAREのロックモードを取得する。＝レコード（行）に対する処理のブロック。

COMMITを実行するまでロックは有効となり，他ユーザーは処理待ち状態となる。

※SELECTでしか使わないらしいのでSELECT FOR UPDATE　と覚えてもよいかも

SELECT \* FROM item FOR UPDATE;

SELECT FOR SHAREとの違い？？ 2021-08-27

・どちらもロックレベル「ROW SHARE」を取得する

FOR UPDATE 自分が更新する目的で行をロックしておく。

FOR SHARE 他からの更新がかからないように行のロックを行う。

#### そのほか

複数のテーブルで列名が重複する場合にはテーブル名を明示しないといけない。

テーブル名：test1 列：no, name, tel

テーブル名：test2 列：no, name, fax

SELECT no, tel from sample1 s1, sample2 s2 WHERE s1.no = s2.no;

⇒ noはテーブルtest1にもtest2にも存在する為，テーブル名を明示しないとエラーになる

逆にこの場合はエラーにならない

SELECT fax from *test1* s1, *test2* s2 WHERE s1.no = s2.no;

条件に当てはまるデータが存在しない場合には値が返されない。

（？　プログラム的にそんな事ありえるのか？2021-09-13）

SELECT id FROM (SELECT id FROM sample1) s1; -- 無駄に変数定義

### CREATE TABLE

SELECTに同じく，情報が多岐にわたるので，独立した項目を作る。

#### 基本

構文：CREATE TABLE [ IF NOT EXISTS ] table\_name (column\_name data\_type [, ... ] )

table\_name 日本語などの2byte文字にも対応している

“”（ダブルクォーテーション）で囲むと大文字・小文字を区別する事ができる。

列数はデータ型にもよるが250～1,600

特定のスキーマに作る場合：CREATE TABLE *myschema*.*mytable* ()

スキーマを指定しない場合は実行時パラメータsearch\_pathに記述があるスキーマの内，最初に指定テーブル名と一致したスキーマが選択される。（参考）[実行時パラメータ](#_実行時パラメータ)

CREATE TABLE *teble1* (

id PRIMARY KEY,

name varchar, -- こんな形でドメインを指定する

)

#### 列設定

DEFAULT 既定値を設定

price INT DEFAULT 2000

次ページへ続く

CREATE TABLE　続き

#### 制約

CONSTRAINT 名前付き制約の指定 （参考）[CONSTRAINT](#_CONSTRAINT)

CHECK 検査制約。値に制約を課す

price numeric CHECK (price > 0),

agecity CHECK(age >= 18 OR city\_code = '0003'),

gender CHECK(gender IN('M', 'F')))

price numeric CHECK (price > 0), --CHECKの複合化の例

discounted\_price numeric CHECK (discounted\_price > 0),

CHECK (price > discounted\_price)

UNIQUE 一意制約。重複する値の禁止

・NULL値も認められる（PRIMARY KEYではNULL値は認められない）

・対象列にインデックスが自動で作成される

column1 name VARCHAR(10) UNIQUE, -- 新規作成時1

(no INTEGER, name TEXT, UNIQUE(no, name)) -- 新規作成時2

ALTER TABLE table1 ADD UNIQUE(column1) -- 修正時

ALTER TABLE table1 ADD UNIQUE(no, name ) -- 2つ以上の列の組み合わせが一意になる

PRIMARY KEY 主キー制約。指定列を主キーにする。

・主キー制約はテーブル毎に１つだけ設定可能。

・１つの主キー制約で複数列を指定する事が可能。（ややこしい；後述）

・値にNULLは認められない。

・対象列にインデックスが自動で作成される。（指定しない限り連番型には成らない点に注意）

複数列の組み合わせを主キーに指定することもできる。

product\_no integer,

order\_id integer,

PRIMARY KEY (product\_no, order\_id)

CREATE TABLE sample (no INTEGER PRIMARY KEY, name TEXT); -- 列ごとに定義するやり方

CREATE TABLE sample (no INTEGER, name TEXT, PRIMARY KEY (no)); -- 後で定義するやり方

NOT NULL NULL値をとる事を禁止する

ALTER TABLE table1 ALTER COLUMN column1 DROP NOT NULL; --NOT NULL 設定の解除

・IS NULL / IS NOT NULLで判定

・NULLを格納した列の値は，データ型にかかわらずNULLとなる。(int型で0になったりはしない)

・NULLを含む列であってもソート対象に指定しても問題ない。

注意点

・１つのテーブルに指定できる名前付き制約の数に制限は無い

次ページへ続く

CREATE TABLE　制約　続き

外部キー

値の入力制限として外部テーブルを参照先とする。（被参照列と呼ばれる）

構文：column\_name column\_type REFERENCES ref\_table1 (ref\_column1));

構文：FOREIGN KEY (self\_column1) REFERENCES ref\_table1 (ref\_column1));

product\_no integer REFERENCES products (product\_no),

product\_no integer REFERENCES products, -- 列名が等しい場合はこのように省略できる

product\_no integer, FOREIGN KEY (no) REFERENCES products(no)); -- FOREIGN KEY を使う場合

仕様

・参照先の列の値には，被参照テーブルに存在する値or NULLしか指定できない。

・１つでも他テーブルで参照されている被参照レコードの変更（ALTER）や削除（DELETE）は不可。

・外部キーの被参照列は被参照テーブル側でPRIMARYかUNIQUE制約が設定されている必要がある。

・外部キーの被参照列のデータ型に制限は無く，ドメインも使用できる。

・REFERENCESコマンドだけを使用してもFOREIGN KEYが作成される（らしい 2021-09-01）

・tableName\_columnName\_fkey という制約が作成される。

CREATE TABLE tableName -- 新規作成時

group\_id integer,

FOREIGN KEY (group\_id) REFERENCES group\_table(id);

ALTER TABLE tableName ADD FOREIGN KEY (group\_id) REFERENCES group\_table(id); --修正時

違い

主キー ：最小限の列セットにより自身のテーブル内のレコードを一意に指定する。

外部キー ：自身のテーブルの列（の集まり）が，他テーブルのレコードを一意に指定する。

補足

・参照整合性制約とも呼ばれる。

次ページへ続く

CREATE TABLE　続き

#### ON UPDATE, ON DELETEにつく制約

被参照側のレコードが削除される時の参照側（制約を付ける側）の挙動を指定する。

NO ACTION 既定の動作。制約違反としてエラーになる

RESTRICT NO ACTIONと同じだが，検査の遅延ができない

CASCADE 参照しているフィールドもあわせて更新（削除）する。

SET NULL 参照列をNULLにする。

SET DEFAULT 参照列を既定値にする。

以下の例ではordersのレコードが削除されるとorder\_itemsテーブルの関連レコードも削除される。

productsのレコードが削除されるとorder\_itemsテーブルの１レコードでも参照削除される。

CREATE TABLE products (

product\_no integer PRIMARY KEY,

name text,

price numeric );

CREATE TABLE orders (

order\_id integer PRIMARY KEY,

shipping\_address text,　　...　);

CREATE TABLE order\_items (

product\_no integer REFERENCES products ON DELETE RESTRICT,

order\_id integer REFERENCES orders ON DELETE CASCADE,

quantity integer,

PRIMARY KEY (product\_no, order\_id)　);

外部キー制約の検査を遅らせる

外部キーが張られているとテーブルにレコードを加えていくときに，どうしても一時的に制約違反の状態になってしまい，それを回避するのが難しいケースがある。

PostgreSQLでは，「SET CONSTRAINTS ALL DEFERRED」というコマンドを使うことで，外部キー制約に限ってトランザクションの終わりまで制約の検査を遅延させることができる。

次ページへ続く

CREATE TABLE　続き

DEFAULT 規定値を指定する。 ※ALTERの場合はSET DEFAULT

name varchar(10) DEFAULT '営業部',

※既定値を指定せず，INSERT時にも指定されなかったデータはNULLになる。

・既定値にNULLを指定する事ができる。

テーブルに対する設定項目

DEFAULT CHARSET=utf8;

ENGINE=InnoDB

その他

TABLESPACE tablespace

テーブル空間名を指定する。省略するとdefault\_tablespaceになる。

USING INDEX TABLESPACE tablespace

UNIQUE、PRIMARY KEY、またはEXCLUDE制約に関連したインデックスを作成するテーブル空間を指定する。

#### 列設定の参考例

id

interger NOT NULL

idとしてはあまり意味がない

interger UNIQUE [NOT NULL] 単に重複してほしくない。

NULL値は許容される。

UNIQUEをつけるだけでCONSTRAINT, btreeが付与される。

インデックスも付与されるが tableName\_id\_key となり，pkeyでは無い。

補足：OracleではNULLの重複が許可される。

interger PRIMARY KEY

インデックス作成＋NOT NULL

serial PRIMARY KEY

自動連番＋インデックス作成＋NOT NULL

便利そうに見えて逆に不便かも。管理者しか触らないテーブルならSEQUENCEを避けた方が無難。

文字列

varchar

text

### 仮想テーブルの作成

#### 概要

SQLクエリの実行結果により1つ以上のテーブルをあたかも1つのテーブルとして扱うことができる。

そのような処理に関連したコマンドをまとめる。

注意点：

複数テーブルを結合したクエリの中で使われる列名には注意が必要。

片方のみでしか使われていない列名は列名のみで良いが，同名の列名が存在する場合はテーブル名も必要

|  |  |  |  |
| --- | --- | --- | --- |
| memberテーブル | groupテーブル | クエリ中の列名 | 備考 |
| 列名no | 列名no | member.no　group.no | 同名の列名があるのでテーブル名も必要 |
| 列名groupNo | 列名no | groupNo　no | 列名が独立しているので列名のみで良い |

#### 交差結合

別名：単純結合

例えば以下のテーブルに以下のレコードが存在するとする。

shopA\_shoesテーブル (id int, size float) VALES (001, 25.0), (003, 23.5)

shopB\_shoesテーブル (id int, size float) VALES (001, 25.0), (002, 22.5), (003, 23.5)

（厳密にいうと，このようなテーブルの分け方をするべきでない。あくまで，これは例）

ここで以下のSQLを実行

SELECT count(\*) FROM shopA\_shoes, shopB\_shoes; -- 2つのテーブルの掛け合わせとなる為，6が出力される。

備考

・CROSS JOIN（後述）と全く同じ結果になる。

（比）UNION ALL JOIN：タプルの結合 UNION：サブクエリの結果の結合

・同名の列が存在する保証がないので，SELECT側でできることが限られる。あまり使えないかも。

架空テーブルの作成　つづき

#### 副問い合わせ

サブクエリ

SELECT, FROM, WHERE句などで条件や範囲を指定するためのクエリ文。()で括る。

・SELECT句で使用する副問い合わせの結果が**複数列または2件以上の場合はエラー**となる。

・FROM句で使用する副問い合わせでは**別名を指定しないとエラー**となる。また，複数の副問い合わせで同じ列名を取得する際に，SELECT句の列名に対象テーブルを明記しない場合もエラーとなる。

SELECT \* FROM (SELECT name FROM zooA); -- エラーになる

・WHERE句で使用する副問い合わせに対し，件数結果に対して適切な演算子を使用しない場合はエラー。

【副問い合わせの結果が1件以下】

=，<，>，<=，>=，<>，!=

SELECT \* FROM table1 WHERE id = (SELECT id FROM table1 WHERE name = '田中');　-- 田中が2件あったらエラー

補足：2件以上あったらどれと等号比較すれば良いのか分からないから　と考えると自然と合点が行く。

【副問い合わせの結果が複数件を含む】

IN，NOT IN，EXISTS，NOT EXISTS，

ANY 副問い合わせの結果のいずれかと一致するかどうか（≒IN）

=ANY／=SOME 副問い合わせの結果のいずれかと一致するかどうか

<>ANY／<>SOME 副問い合わせの結果のいずれかと一致しないかどうか（≒NOT IN）

=ALL 副問い合わせの結果全てと一致するかどうか

<>ALL 副問い合わせの結果全てと一致しないかどうか

【そのほか】

（）　何の意味があるのか，以下のような事ができる。

SELECT (SELECT id FROM sample1 WHERE name = '田中');

⇒ サブクエリの結果をそのまま返す為，普通にSELECTしたのと変わらない。

#### GROUP BY

SELECT groupName FROM table1 GROUP BY groupName HAVING groupName != '人事部';

HAVING

条件を指定する。≒whereと考えれば良い。

「GROUP BYより先に働く条件」といった認識で考えると分かりやすい。

SELECT groupName FROM table1 GROUP BY groupName HAVING max(age) > 30;

--この例ではageが30以上のレコードをgroupNameでグループ化。

#### JOIN

2つのFROM項目を結び付ける。2つ以上のタプルの結合。日本語では「結合」。

入れ子の場合，（）を付けた方が良い。ない場合は左から右へ入れ子。

SELECT column\_name FROM table1 INNER JOIN table2 ON hoge=fuga

ONには条件が記述される。（WHEREに近い？）

以下のいずれか

• [ INNER ] JOIN 内部結合。双方のテーブルの共通の列を結合キーにする。

SELECT \* FROM members INNER JOIN departments ON members.groupNo=departments.no;

SELECT \* FROM members INNER JOIN departments ON true; --CROSS JOINと同じ結果になる

左側（FROMで指定）テーブルの列、右側（JOINで指定）テーブルの列の順で表示される。

結合時に列名の指定が必要。ONによる条件の付与かNATURAL INNER JOINの形で使用。

• LEFT [ OUTER ] JOIN 左テーブル（fromで指定）のレコードは無条件で対象。

• RIGHT [ OUTER ] JOIN 右テーブル版。OUTER JOINはどちらか一方が条件に一致すれば対象。

SELECT \* FROM members OUTER INNER JOIN departments;

-- この場合membersが左テーブル、departmentsが右テーブル

• FULL [ OUTER ] JOIN 内部結合＋結合条件不一致のレコードの値をNULL（空白）で結合

SELECT \* FROM t1 FULL JOIN t2 ON t1.num = t2.num;

• CROSS JOIN 掛け算的に総当りで結合。あまり使わない。

SELECT \* FROM members CROSS JOIN departments;

SELECT \* FROM member, department; -- このようにしても同じ結果が得られる

・NATURAL 名前が同じ列全てを結合キーに指定

usingのように，キーとなった列が最初（左端）の列に表示されて結合される。

共通名の列を左側のテーブルの列順を基準にして，**出力データの列が入れ替わる**のが特徴。

NATURAL INNER JOIN / NATURAL OUTTER JOIN

JOIN～USING 値が等しいことを条件に結合。結合キーは出力の一番左にまとめられる。

まったく同じ名前の列がある時に便利。

SELECT \* FROM member INNER JOIN department USING (groupNo); --INNER JOINの場合

SELECT \* FROM member FULL OUTER JOIN department USING (groupNo); --OUTER JOINの場合

Join参考資料

<https://www.codeproject.com/KB/database/Visual_SQL_Joins/Visual_SQL_JOINS_orig.jpg>

#### UNION

テーブルとテーブルの論理和集合（合算）を作り出す。

構文：query1 UNION [ALL] query2

query2の結果をquery1に付加するが，行の順序がquery1,2となる保証はない。

例

SELECT \* FROM teama UNION \* FROM teamb --teama，teambの和集合。（重複は省かれる）

（反）EXCEPT （比）CROSS JOIN

#### ALL

UNION，INTERSECTと共に使い，重複レコードも対象に含める

SELECT \* FROM teama UNION \* FROM teamb --teama，teambの和集合，かつ重複も表示される

具体例

テーブルprojectAにA社社員，B社社員，C社社員，C社社員 のレコードがあり，

テーブルprojectBにC社社員，D社社員　のレコードがある場合（INTERSECTの項と同じ条件）

SELECT \* FROM projectA INTERSECT ALL SELECT \* FROM projectB;

を実行すると ＞C社社員，C社社員　と出力される。

#### ANY

別名：SOME

INとほぼ同じで，真のときは副問い合わせの結果をそのまま返し，**偽の時はNULL**を返す。

SELECT \* FROM sample1 WHERE id =ANY (SELECT id FROM sample1 WHERE name = '佐藤');

#### EXCEPT

テーブルとテーブルの論理差集合（除外），かつ重複レコードも削除。

構文：query1 EXCEPT [ALL] query2

SELECT \* FROM teama EXCEPT \* FROM teamb --teama，teambの和集合。（重複は省かれる）

具体例

テーブルprojectAにA社社員，B社社員，C社社員 のレコードがあり，

テーブルprojectBにC社社員，D社社員　のレコードがある場合に

SELECT \* FROM projectA EXCEPT SELECT \* FROM projectB;

を実行すると ＞A社社員，B社社員　と出力される。

（反）UNION

（比較）NOT EXISTS() ()内で示される条件に対しtrue/false を返す

#### INTERSECT

テーブルとテーブルの論理積集合（一致）を作り出す。

テーブルprojectAにA社社員，B社社員，C社社員，C社社員 のレコードがあり，

テーブルprojectBにC社社員，D社社員　のレコードがある場合に

SELECT \* FROM projectA INTERSECT SELECT \* FROM projectB;

を実行すると　C社社員　と出力される。（projectAには2レコードあるが，重複レコードは1件のみ）

### ロール

#### 概要

ロールはデータベータクラスタごとに作成され，同クラスタ内のデータベースで共有できる。

#### ROLL とUSERの違い

基本的にはroll≒user。どちらかと言うとROLEが正当。

近年のバージョンではユーザーとして使う場合もグループとして使う場合もROLEを用いる。（USERはPostgreSQL 8.0.x以前に「USER」と「GROUP」に分けられていた事に起因する。）

またCREATE USERというコマンドは例えばver11.5でも使用する事が可能で，CREATE USER ではデフォルトでLOGIN権限が付く（CREATE ROLEでは付かない）など多少の違いもある。

#### CREATE ROLE

構文：CREATE ROLE user\_name [ [ WITH ] オプション [ ... ] ]

概要

オプション（太文字がデフォルト）

SUPERUSER | **NOSUPERUSER** スーパーユーザーかどうかを決める

CREATEDB | **NOCREATEDB** 新しいデータベースを作成する権限

CREATEROLE | **NOCREATEROLE** 対象ロールが別の新しいロールを作成できるかどうか

CREATE ROLE user1 WITH CREATEROLE; --ロール作成権限付きでuser1を作成

**INHERIT** | NOINHERIT 他の（複数の）ロールの権限を引き継ぐ

？？具体例？？ 2021-09-09

LOGIN | **NOLOGIN** ユーザー(LOGIN)かグループ(NOLOGIN)かを分ける，と考える。

※createuserでは-l；--loginがデフォルトな点に注意

REPLICATION | **NOREPLICATION** レプリケーションロールかどうか。（pg\_basebackupなど）

[ENCRYPTED] PASSWORD 'password'　パスワードの指定。

VALID UNTIL 'timestamp' ロールのパスワードが無効になる日時を設定。def.=永遠に有効

IN ROLE 'group\_role\_name' 今回追加するロールを，他のロールの新規メンバとする。

CREATE ROLE user1 IN ROLE group1;

ROLE 'member\_role\_name' 他のロールをメンバとして，今回追加するロールに含める。

CONNECTION LIMIT ロールの最大同時接続数を指定。def.=**-1**(無制限)

PASSWORD ‘password’ PostgreSQLへのログインパスワードを設定

IN ROLE group\_role1 メンバとして追加する既存のロール名を設定

補足

・CREATE USER というコマンドがあり，効果はほとんど同じ。（旧バージョンとの互換の為？）

⇒ CREATE USER ではデフォルトでLOGIN権限が付く

・スーパーユーザー権限を持ったロールを作る際には，実行ロールにもスーパーユーザー権限が必要。

（参考）[createuser](#_createuser)

（参考）<https://www.postgresql.jp/document/11/html/sql-createrole.html>

（補足）オプションは小文字でも良いらしい。

#### ALTER ROLE

=# ALTER ROLE role1 PASSWORD 'your\_password' -- パスワードの設定を行う

# 特にCREATE ROLEと書式が異なるもの

ALTER ROLE user1 RENAME TO new\_name1; --データ型の変更

・CREATE ROLEと同様に，ALTER USERでも良いらしい

#### INHERITS

継承。同テーブルの中で別グループを作るようなときに役に立つ。

基本

table1の列がno INTEGER, name VARCHAR(11) の時下記のクエリを実行すると

CREATE TABLE table2 (tel CHAR(11)) INHERITS (table1);

列 no INTEGER, name VARCHAR(11), tel CHAR(11) を持ったtable2が作成される。

継承と言うより，実レコードの記録先メモリアドレスを共有し（C言語のポインタに近い），かつtable1と2では列の表示範囲の定義が異なる，と捉えた方が良さそう。

具体例

cities（市）テーブルを作り，その中で県庁所在地の市があり，通常県庁所在地の市のレコードにのみアクセスしたい場合。以下のように継承テーブルを作成する。

CREATE TABLE cities (name text, population int, altitude int);

CREATE TABLE capitals ( prefecture char(4) ) INHERITS (cities);

### DDL（オブジェクト）

Data Definition Language データ定義言語 [ デカい object と覚えようかな ]

（補足）大文字でも小文字でも良いが，web上の情報では大文字が好まれている様子。

CREATE ROLE ロールを追加（後述）

ALTER ROLE ロールに付属するオプションを変更する。

構文：ALTER ROLE *username* LOGIN;

DROP ROLE *username* 　ロールを削除

#### CREATE DATABASE

構文：CREATE DATABASE *db\_name* [ [ WITH ] [ OWNER [=] *user\_name* ] ]

備考

データベースの作成するユーザーにはsuperuserかcreatedbオプションが必要。

データベース名は先頭がアルファベット文字から始まる，63バイトまでの長さ。

createdbというラッパコマンドが存在する。

作成元となるテンプレートを指定する必要がある。

オプション

TEMPLATE [=] template 作成元となるテンプレートの指定

ENCODING [=] encoding 使用される文字セット。-E

WITH ENCODING 'UTF8'

（使用できる文字セット）<https://www.postgresql.jp/document/9.4/html/multibyte.html>

LC\_COLLATE [=] lc\_collate 照合順。ORDER BYなどで使用。

LC\_CTYPE [=] lc\_ctype 使用する文字のクラス。小文字，大文字，数字など。

TABLESPACE [=] tablespace\_name デフォルトのテーブル空間名。

ALLOW\_CONNECTIONS [=] true/false 　接続許可。def.=true

CONNECTION LIMIT [=] *number* 同時接続数。def.=-1 無制限

IS\_TEMPLATE [=] ture/false 複製可能かどうか。falseでもスーパーユーザーは複製可能。

OWNER [=] *user\_name* 所有者を指定する。

-E，-lオプションはテンプレートがtemplate0の時のみ有効

補足

データベースを2つ以上にするメリットは特に無いらしい。

特に小規模の使用では1つで十分だし，予期せぬテーブル同士の連携を想定するとむしろ1つの方が良い。

関連

DROP DATABASE

（参考）<https://www.postgresql.jp/document/11/html/sql-createdatabase.html>

（参考）[createdb](#_createdb)

#### CREATE TABLE

独立した項目を作成[CREATE TABLE](#_CREATE_TABLE_3)

#### ALTER 共通

ALTER TABLE table1 RENAME TO newTable1; --テーブル名の変更

ALTER FUNCTION func1(integer) OWNER TO user1; --関数の所有者を変更

（比較）UPDATE：レコードの変更

#### ALTER TABLE

列設定の変更の場合はALTER TABLE table\_name ALTER COLUMN column\_nameとなる

ALTER TABLE table1 RENAME TO table1\_re; -- テーブル名の変更

ALTER TABLE table1 ADD COLUMN column1 text; -- 新規列の追加

ALTER TABLE table1 DROP COLUMN column1; -- 列の削除　行名がなければエラー

ALTER TABLE table1 ALTER COLUMN column1 TYPE [新しいデータ型]; -- データ型の変更

ALTER TABLE table1 ADD PRIMARY KEY(id) ; -- 主キー制約

ALTER TABLE table1 ADD UNIQUE(id) ; -- 一意制約

# 特にCREATE TABLEと書式が異なるもの

ALTER TABLE table1 ALTER COLUMN column1 SET DEFAULT '営業部'; -- 既定値の変更

ALTER TABLE table1 ALTER COLUMN column1 SET DATA TYPE char(50); -- データ型の変更 ver9.1で追加

ALTER TABLE table1 ALTER COLUMN “column1” TYPE text [USING “name”::text;]

スーパーユーザー権限が無ければ，自分以外を所有者に指定する事はできない。

ALTER TABLE table1 ALTER COLUMN column1 TYPE text;

A

#### DROP

DROP DATABASE データベースの削除

構文：DROP DATABASE [IF EXISTS] *db\_name*

・データベースが削除できるのはスーパーユーザーとデータベースの所有者のみ

DROP ROLE

DROP ROLE user1; -- ロール user1の削除

DROP USER user1; -- CREATE ROLEと同様に，DROP USERでも良いらしい

・コマンドの実行にはCREATEROLE権限が必要

・スーパーユーザーを削除する場合は，実行ユーザーもスーパーユーザーである必要がある。

・データベースやテーブルなどの所有者は削除できない

#### TRUNCATE

テーブルからすべてのレコードを削除する。

構文：TRUNCATE table1

DELETEの連続，と考えてしまうとDMLに分類したくなってしまう。テーブルの初期化，みたいに考える。

（類似？）intersect　積集合を作成する

### DML（データ）

Data Manipulation Language データ操作言語Mani query

#### SELECT

別項目を作成：[SELECT](#_SELECT)

#### INSERT

レコードを追加

構文：INSERT INTO tablename VALUES(val1, val2, val3...);

※INSETではINTOが必ず付く。

INSERT INTO schema1.table1 VALUES (1, 'hoge'); -- 基本。3つ目以降の列の値はNULLになる。

INSERT INTO table1(id, name) VALUES (1, 'hoge'); -- 列名を指定する場合

INSERT INTO table1 VALUES(6, 'バレー部'), (7, '卓球部'); -- レコードを2つ追加する事もできる

<https://www.postgresql.jp/document/11/html/sql-insert.html>

INSERT…SELECT クエリ結果から別テーブルのレコードを追加する

INSERT INTO table1 (id)

SELECT table2.fld\_order\_id

FROM table2 WHERE table2.fld\_order\_id > 100;

サブコマンドにSELECTを使用する事もできる。

INSERT INTO member SELECT 'Y0099', '2017-03-17', 0, 'f'; -- あまり意味が無いが，こんな事もできる。

VALUESの注意点

先に（）で列名の指定がある場合は，VALUES内の要素の数も合わせなければエラーになる。

INSERT INTO table1 VALUES (1); -- これはOK。2つ目以降の列の値はNULLになる。

INSERT INTO table1 (id, name) VALUES (1); -- これはエラー。

#### UPDATE

レコードなど，特定のデータを更新。

※UPDATEはテーブルにしか使わない。その為か，UPDATE TABLEでは無いので注意。

構文：UPDATE tablename SET field = value WHERE id = id\_val;

例

UPDATE members SET group\_id = 5 WHERE id = 2; -- 通常の値変更

UPDATE members SET (id, name) = (3, ‘fuga’) WHERE id = 2; -- 複数の値変更1

UPDATE members SET id = 3, name ‘fuga’ WHERE id = 2; -- 複数の値変更2

UPDATE members SET score = score + 2000 WHERE login; -- 条件によって値を変更する

（参考）<https://www.postgresql.jp/document/11/html/sql-update.html>

（比較）ALTER：テーブル，ロールなどの変更

#### DELETE FROM

レコードなど，特定のデータを削除。

※DELETEは必ずFROMが付くので，「DELETE FROMというコマンド」という感じで覚える。

DELETE FROM products WHERE product\_no=101;

DELETE FROM products WHERE no = 1 OR no = 2; --現時点でnoが1か2のレコードを削除

DELETE FROM products WHERE name ~ ‘sa’; -- POSIX正規表現で削除するレコードを選択

DELETE FROM products; -- products テーブルの**全レコード**を削除　≒DROP TABLE products

### DCL（トランザクション）

Data Control Language データ制御言語 [ conTranxactionとか覚えようかな ]

#### 概要

・トランザクション内でエラーが発生するとROLLBACKが実行されるまで，すべてのSQLがエラーになる。

（関連）[ロックモード一覧](#_ロックモード一覧)

#### 自動COMMIT

明示的にトランザクションを開始しない限り、SQLが処理された時点でその内容はコミットされる。

#### デッドロック

例：

user1=>：BEGIN;

user2=>：BEGIN;

user1=>：SELECT \* FROM item WHERE itemId = '0011' FOR UPDATE;

user2=>：SELECT \* FROM sales WHERE itemId = '0011' FOR UPDATE;

user1=>：SELECT \* FROM sales WHERE itemId = '0011' FOR UPDATE;

user2=>：SELECT \* FROM item WHERE itemId = '0011' FOR UPDATE;

この場合，どちらかがCOMMITをするまでお互いにロックの解放待ちとなりデッドロックとなる。

なお，PostgreSQLにはデッドロックを回復させる機能があり，この場合どちらかのトランザクションがROLLBACKされる。（どちらかは分からない）

#### BEGIN

トランザクションの開始。

BEGINではロックを得られない点に注意。

（関連）[ロックモード一覧](#_ロックモード一覧)

詳細

・または START TRANSACTIONでも良い。

BEGIN ISOLATION LEVEL serializable; -- トランザクションの分離レベルを明示的に指定する。

（関連）[分離レベル](#_分離レベル)

補足

START TRANSACTIONという別名コマンドが有り，BEGINと全く同じ？

#### COMMIT

トランザクションの確定

詳細

・または ENDでも良い。

#### ROLLBACK

トランザクションの取り消し

BEGIN; --トランザクションを開始

INSERT INTO members VALUES(3, ‘渡辺’, ’01201234569’, 3); --レコード追加：渡辺

ROLLBACK; --取り消し（INSERTが無効化）

詳細

・またはABORTでも良い。

#### SET TRANSACTION ISOLATION LEVEL

構文：SET TRANSACTION ISOLATION LEVEL分離レベル;

分離レベル

READ COMMITTED／REPEATABLE READ／SERIALIZABLE

スナップショットの例

BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;

SET TRANSACTION SNAPSHOT '00000003-0000001B-1';

#### SAVEPOINT

構文SAVEPOINT セーブポイント名;

BEGIN; -- 再びトランザクションを開始

INSERT INTO members VALUES(1, ‘渡辺’); -- レコード追加：渡辺

SAVEPOINT s1; -- セーブポイントの作成

INSERT INTO members VALUES(2, ‘伊藤’); -- レコード追加：渡辺

SAVEPOINT s1; -- 同名のセーブポイント。s1が上書きされる。上のs1は無かった事に。

DELETE FROM members WHERE no = 2; -- 誤ってレコードを消したとする

ROLLBACK TO s1; -- 元に戻す

COMMIT; -- コミット

select \* from members; -- 確認。「伊藤」が存在している事が確認できる。

RELEASE SAVEPOINT s1; -- 補足：直前のセーブポイント（s1）を削除。

#### GRANT

アクセス権限を定義する。（特権を与える？）

※意外にもDCLに分類される

構文：GRANT 権限 ON [TABLE] *table\_name*  TO { *role* | public }; など

GRANT all ON table1 to user1; # 一般ユーザーに権限付与）

GRANT INSERT ON table1 TO PUBLIC;

権限

SELECT SELECTとCOPY TO（テーブルからファイルへ保存）

INSERT INSERTとCOPY FROM（テーブルへの行の挿入）

UPDATE UPDATE　但し複雑なUPDATEコマンドではSELECT権限が必要な場合が多い。

DELETE DELETE　但し複雑なDELETEコマンドではSELECT権限が必要な場合が多い。

※テーブルの削除はDROPで行う上，CREATE権限が影響する点に注意

TRUNCATE TRUNCATEテーブルの全データを高速で削除

TRIGGER

CREATE スキーマに対するオブジェクトの作成。スキーマ，テーブル，インデックス…。

CREATE ROLEは含まない。CREATE ROLE権限はALTER ROLEなどで変更する。

CONNECT 指定されたデータベースに接続する事ができる。

TEMPORARY 一時テーブルの作成を許可する　also：TEMP

EXECUTE FUNCTIONとPROCEDUREの使用を許可する

USAGE 手続き言語の関数を作成することを許可　2021-09-09 意味不明

REFERENCES 外部キー作成

ALL [PRIVILEGES] 全ての権限。PRIVILEGESはPostgreSQLでは省略可能。厳密なSQLでは必要。

管理できない権限

CREATEROLE, CREATEDB. LOGIN, SUPERUSER → CREATE ROLEで付与

ALTER → テーブルの所有者であるかどうか、などが実行権限に関係する。（で合ってる？2021-12-22）

#### REVOKE

アクセス権限を取り消す

構文：REVOKE 権限 ON [TABLE] *table\_name* FROM {ユーザー名 | PUBLIC};

REVOKE DELETE ON *table1* FROM user1;

PUBLIC

暗黙的に定義された全ロールからなるグループ。pg\_rolesにも存在しない。

（情報元）<https://www.postgresql.jp/document/11/html/sql-revoke.html>

・ユーザー名には ””があっても無くても良い。

### 管理区分

#### DOMAIN

ドメイン

列に代入する値に課す事ができる制限。C言語のtypedefのように型を定義する感覚に近い？

CREATE DOMAIN test\_domain AS integer DEFAULT 3 CONSTRAINT original\_type\_check CHECK (VALUE < 20);

CREATE TABLE *teble1* (

id serial NOT NULL,

value test\_domain, -- こんな形でドメインを指定する

PRIMARY KEY (ID)

)

構文：CREATE DOMAIN name [ AS ] data\_type

[COLLATE 照合順]　[DEFAULT デフォルト値]　[ [CONSTRAINT 制約名] 制約 ]... ;

data\_type CREATE TABLEの時のように，使用するデータ型を指定する。

照合順 照合順序の指定。？？？

その他はCREATE TABLEの列定義部分と同様。

CREATE DOMAIN domain\_name;

ALTER DOMAIN domain\_name SET DEFAULT デフォルト値;

ALTER DOMAIN domain\_name DROP CONSTRAINT 制約;

ALTER DOMAIN domain\_name RENAME TO domain\_new; --ドメインの名称を変更する

DROP DOMAIN domain\_new; --ドメインを削除

補足

CONSTRAIN制約とドメインは同時に使うことができる。

⇒ 条件が矛盾する時はどちらが優先される？2021-09-22

#### SCHEMA

テーブル，データ型，関数などのオブジェクトをグループ化して管理する１区画。

本質的には名前空間と言える。既定値は”$user”, public ⇒２つのスキーマが用意されている。

３層スキーマという方式が一般的らしく，外部ユーザー用，開発者向き概念設計用，postgreSQLの内部用でスキーマを分ける，といった使い方をする（らしい）

⇒ 例えば外部ユーザーにはレプリケーションだけ見せるとか　そういった使い方？2021-09-07

ひとつのスキーマに変更があったとしても他の2つのスキーマは影響しないようにするのが主たる目的。

例えばpostgresでログイン中にshow search\_pathで現在のスキーマ検索パスが設定されていた場合

SHOW search\_path; -- "$user" は現在のロール名と同じ名前のスキーマ

"$user", schemaA, public -- 結果

ここでSELECT \* from table1;　というクエリを実行すると，まず現在のユーザー名のスキーマpostgresにtable1が存在するかを確認，見つかればpostgres.table1に対してデータを取得。次にschemaA.table1と見つかるまで，または記述されている分だけ実行される。

例

CREATE SCHEMA *schema1* AUTHORIZATION *user1*; -- user1に所有権があるschema1を作成

CREATE TABLE *schema1*.*table1* (id INTEGER, name VARCHAR);

公式によるスキーマの使い方例：

・1つのデータベースを多数のユーザーが互いに干渉することなく使用できるようにするため。

• 管理しやすくなるよう，データベースオブジェクトを論理グループに編成するため。

•サードパーティのアプリケーションを別々のスキーマに入れることにより，他のオブジェクトの名前と競合しないようにするため。

次ページに続く

～SCHEMA　続き

CREATE SCHEMA

構文：CREATE SCHEMA *schema1* [ AUTHORIZATION *所有ロール名* ] [ schema\_element [ ... ] ]

AUTHORIZATION

指定しなかった場合、本コマンドの実行ロールが所有する。

ALTER SCHEMA **スキーマ名**の変更

ALTER SCHEMA *schema1* RENAME TO *new\_schema1*;

ALTER SCHEMA *new\_schema1* OWNER TO postgres; --スキーマの所有者をpostgresに変更

ALTERの場合はOWNER TO *user*となる点に注意

DROP SCHEMA

DROP SCHEMA *schema1* -- 対象スキーマにオブジェクトがある場合はエラーになる

DROP SCHEMA *schema1* CASCADE; -- 関連オブジェクトを含めて削除する。（CASCADEは省略可）

その他

pg\_ から始まるスキーマは特殊なスキーマ（システムカタログ）とみなされる為、作成できない。

#### VIEW

複雑なSELECT文に名前をつけ，仮想テーブルとして定義したもの。VIEWは実テーブルが使用できるほとんどの場合で使え，≒テーブルと捉えても良い。他のVIEWに対するVIEWの作成も珍しくない。

VIEWはテーブル構造の詳細をカプセル化するので，アプリケーションが発展するに従いテーブル構造が変わったとしても，一貫したインターフェイスを保てる。

詳細

・作成したビューの一覧はシステムカタログ「pg\_views」から確認できる

・CREATE VIEWでOR REPLACEオプションを指定すると，既存のビューを置き換えられる

・参照時にはVIEWに対するSELECT権限が必要。

・作成には元テーブルに対して**SELECT権限**が必要。（CREATE権限では無い）

・修正（ALTAR VIEW）は対象VIEWの所有者でのみ実行可。（スーパーユーザーでも無理？）

以前のverは検索処理のみの実行だったが，ver9.3以降は以下の条件を満たせば更新処理(INSERT，UPDATE，DELETE)が可能

・複数テーブルの結合を行っていない

・集約関数を使用していない。（参考）[集約関数](#_集約関数)

CREATE VIEW view1 AS SELECT city, temp, prcp, date FROM weather, cities WHERE city = name; -- 定義

SELECT \* FROM view1; -- 利用

注意点

・ALTER VIEWでの変更の対象はあくまでVIEWであって，参考元のテーブルに変更はない（？）

ALTER VIEW view1 ALTER COLUMN groupNo SET DEFAULT 9; -- 参照元テーブルの既定値は変更されない

ALTER VIEW view1 ALTER groupNo SET DEFAULT 9; -- COLUMNキーワードは省略できる

・VIEWの所有者でないとALTER VIEWの実行はできない。

・同スキーマ内の同名オブジェクトは認められない。例：animalテーブルがある時，animalビューは作れない

・VIEWはプログラム処理による仮想テーブルなので，利用するたびに参照元のテーブルへのアクセスが発生する。その為検索の効率が下がる場合がある。

比較

|  |  |  |
| --- | --- | --- |
|  | コマンド | 詳細 |
| VIEW | CREATE/DROP VIEW | 複雑なクエリに名前をつけて保存。仮想テーブルとしても扱われる。 |
| MATERIALIZED  VIEW | CREATE/DROP MATERIALIZED VIEW | キャッシュデータのあるVIEW |
| PREPARE | PREPARE/  DEALLOCATE [PREPARE] | VIEWの関数化。高速かつ変数の使えるVIEW  EXECUTE prepared1();　で実行。 |
| CURSOR | DECLARE/ | ※FUNCTIONの処理内容定義に使うのもDECLAREなので注意 |

#### MATERIALIZED VIEW

複雑なSQL文のSELECT結果を頻繁に取得する場合に使用する機能

CREATE MATERIALIZED VIEW -- 作成

REFRESH MATERIALIZED VIEW -- データ（キャッシュ）の更新

ALTER MATERIALIZED VIEW -- 修正

DROP MATERIALIZED VIEW -- 削除

・マテリアライズドビューでは対象のデータをキャッシュし，実体として保持する

・元のテーブルが変更されてもデータの実体を保持し続ける。

・マテリアライズドビューの内容を更新したい場合にはREFRESHが必要

・ビューを定義するSQL文にORDER BY句を使用できる

・インデックスを作成する事ができる。

共通

IF NOT EXISTS 同名のマテリアライズドビューの存在有無がエラーに影響しない（REFRESH以外）

WITH DATA 作成時や更新後にデータを投入する

WITH NO DATA 作成時や更新後にデータを投入しない（REFRESH MATERIALIZED VIEWが必要）

CREATE

WITH DATA マテリアライズドビューの作成時にデータを投入する

WITH NO DATA マテリアライズドビューの作成時にデータを投入しない

投入しない場合，スキャン不可という印が付き、REFRESH MATERIALIZED VIEWが使用されるまで使用できない。

DROP

CASECADE 対象のマテリアライズドビューに依存するオブジェクトも削除する

RESTRICT 対象のマテリアライズドビューに依存するオブジェクトがある場合はビューを削除しない

REFRESH

CONCURRENTLY マテリアライズドビューの更新中もSELECT処理をブロックしない

#### PREPARE

プリペアド文

https://www.postgresql.jp/document/11/html/sql-prepare.html

コンパイルており高速で，かつ変数の使えるVIEWといった感じ。

通常のSQL文の実行はSQLの書き換え→SQLの構造の解析→実行計画の作成　となる。

プリペアド文は上記の作業を一度に行い（コンパイルみたいな感じ？）性能の向上が図れる

PREPARE prepare1 (varchar) AS SELECT member\_info.name, address\_info.address

FROM member\_info INNER JOIN address\_info ON member\_info.addressId = address?info.addressId

WHERE address\_info.address = $1; --作成

EXECUTE prepare1(); --実行

DEALLOCATE prepare1; --削除

・SELECT，UPDATE，INSERTなどが指定できる

#### FUNCTION

標準SQLではストアドファンクション（ stored function）と呼ぶらしい。

構文：

CREATE [OR REPLACE] FUNCTION 関数名([引数の型]) RETURNS [SETOF] 戻り値の型 AS

$$処理内容 $$　LANGUAGE 使用言語　[引数NULLの挙動]　[実行権限]

SETOF 型　を返すよう宣言されている場合、関数の最後の問い合わせは最後まで実行される

ないと１行？

使用言語

SQL

PL/pgSQL

引数NULLの挙動

CALLED ON NULL INPUT ？？？

実行権限

SECURITY DEFINER 関数を作成したユーザーの権限に基づいて実行される

・RETURNS void にすると戻り値の無い関数が作成できる

・RETURNS [SETOF] TABLE [(columns)] ？？でテーブル

・RETURNS [SETOF] record 公式資料に，一切の説明がなくこんな書き方がされている2021-09-01

基本

=#　CREATE [OR REPLACE] FUNCTION funcName( argtType1 ) RETURNS SETOF retType AS $$

$#　SELECT item\_name FROM sample WHERE price > $1;

$#　$$ LANGUAGE SQL;

引数int，戻り値varcharの関数を作成

$1 １つ目の変数

具体例

=# CREATE [OR REPLACE] FUNCTION func1(ret\_typer) RETURNS SETOF varchar(20) AS $$

$# SELECT item\_name FROM sample WHERE price > $1;

$# $$ LANGUAGE SQL;

ALTER FUNCTION func1 (argtType1) OWNER TO user1; -- 引数型まで必要。オーバーライドがあるって事？

DROP FUNCTION [IF EXISTS] funcName( argtType1 );

（参考）<https://qiita.com/SRsawaguchi/items/411801e254ee66f511f1>

次ページに続く

～FUNCTION　続き

PL/pgSQL言語

構文

DECLARE

　　変数宣言

BEGIN

　　関数の処理

END;

例

CREATE FUNCTION sample\_func(TEXT) RETURNS BOOLEAN AS $$

DECLARE

　　x ALIAS FOR $1;

　　result BOOLEAN := FALSE;

BEGIN

　　IF x IS NOT NULL THEN

　　　　result = TRUE;

　　END IF;

RETURN result;

END

$$ LANGUAGE plpgsql CALLED ON NULL INPUT;

比較

|  |  |  |  |
| --- | --- | --- | --- |
|  | 実行方法 | コマンド | 詳細 |
| PREPARE | EXECUTE | PREPARE/ DEALLOCATE | VIEWの関数化  高速かつ変数の使えるVIEW |
| FUNCTION | クエリ内 | CREATE/DROP FUNCTION |  |
| PROCEDURE | CALL |  | ver11以降で使用可能 |

注意点

同じ関数名でも引数の型が違う場合は別の関数として扱われる。

#### PROCEDURE

英語ではStored Procedureと呼ぶらしい

ALTER PROCEDURE proc1(integer, integer) RENAME TO proc2; --変更

DROP PROCEDURE --削除

CALL func1(‘hoge’) --実行

PostgreSQLのバージョン11から対応。

\dのtypeは「proc0」となる。

CREATE FUNCTIONを用いても同じ関数が作成できる。

※TRIGGERではEXECUTEが使われる？

CREATE TRIGGER sample\_trg AFTER INSERT ON sample FOR ROW EXECUTE PROCEDURE sample\_func();

#### CURSOR

問い合わせをカプセル化し，問い合わせの結果を一度に数レコードずつ読み取る。

膨大な結果がある場合のメモリ節約が目的

構文：DECLARE *cursor\_name* CURSOR [ ( arguments ) ] FOR query;

DECLARE (cursor) FOR と覚える

例

DECLARE liahona SCROLL CURSOR FOR SELECT \* FROM films;

FETCH FORWARD 5 FROM liahona;

DECLARE カーソルを定義。

INSENSITIVE カーソルから取得したデータが取得元テーブルの更新に影響を受けない。[def.]

BINARY テキスト形式ではなくバイナリ形式でデータを取得

WITH HOLD トランザクションが終わってもカーソルは有効。（対）WITHOUT HOLD[def.]

SCROLL データのスクロールを許可する。（順逆共に可能）

NO SCROL データのスクロールを順方向のみ許可する。

OPEN

OSSDB Silverの範囲外

MOVE

データの取得なしにカーソルを進める

構文：MOVE [ direction [ FROM | IN ] ] cursor\_name

MOVE FORWARD 4 *cursor\_name* ; --カーソルを順方向に4つ移動

FETCH

構文：FETCH [ direction [ FROM | IN ] ] *cursor\_name*

FETCH BACKWARD IN *cursor\_name* ; --逆方向にデータを取得

注意点

・作りたてのCURSORは現在位置が0。

・FETCH (FORWARD) は**現在位置のレコードから**取得を開始する

・FETCH BACKWARDは現在位置のレコードを**含まず**，後方のレコードから取得を開始する

・カーソルが作成されていない場合はエラーとなる

direction

FORWARD [count] 現在行を含めcount行分を取り出す。count=0で現在行，count省略でNEXTと同等。

BACKWARD [count] 現在位置を含まず，前のcount行分。count=0で現在行，count省略でPRIORと同等。

NEXT 次の行を取り出す。　(opp) PRIOR　[def.]

PRIOR １つ前の行を取り出す。　(opp) NEXT

FIRST 先頭行を取り出す。ABSOLUE 1と同じ。

LAST 最終行を取り出す。ABSOLUE -1と同じ。

ABSOLUTE count 問い合わせのcount行を取り出す。

RELATIVE count カーソルの現在位置からcount番目の行を取り出す。

count 単体だと ＝FORWAR count

ALL 単体だと ＝FORWAR ALL　カーソルの現在値以降の全データ。

#### SEQUENCE

C言語のforループみたいに値を自動で加算させる

・初期値は0

・負の値を取ることができる

関数

currval() 対象のシーケンスの現在値。

シーケンス作成直後，curvalの実行前にnextvalが必要。さもなくばエラー。

nextval() 次のシーケンスに進み，次に採番された値を返す。

ROLLBACKで値は元に戻らない。setvalで再代入が必要。

SELECT nextval('seq1');

setval() 次に採番される値を指定。

SELECT setval('seq1', 100); -- seq1の次の値を100に。（なんか直感的じゃない）

関連：serial型

PostgreSQL独自の型で，serialというものが存在する

CREATE TABLE table1 ( id serial ); -- serial型を使ったテーブルの作成

これは以下のようにする事と同等

CREATE SEQUENCE table1\_id\_seq;

CREATE TABLE table1 (

id integer DEFAULT nextval(' table1\_id\_seq ') NOT NULL

);

次ページに続く

〜SEQUENCE　つづき

CREATE SEQUENCE

構文：CREATE [ TEMPORARY | TEMP ] SEQUENCE name [ INCREMENT [ BY ] num ]　[オプション]

INCREMENT [BY] 増分値の指定

TEMPORARY セッションが終了すると自動的に削除される

オプション

MAXVALUE val シーケンスが生成する最大値を決定

(opp) NO MAXVALUE (see also) MINVALUE

START val シーケンスの開始値を変更

RESTART val シーケンスの現在値を変更

CACHE num 番号の割り当てを高速化するためにキャッシュを使用　--def.=1（キャッシュなし）

CYCLE 限界値に達したときに周回させる (opp) NO CYCLE[def.]

（補足）NO CYCLE時に上限まで達した後にnextvalを行うとエラーになる。

（関連）MAXVALUE，MINVALUE

基本

CREATE SEQUENCE seq;

CREATE TABLE mytable (id int DEFAULT nextval('seq'), name varchar(256));

INSERT INTO mytable(name) VALUES('name1');

CREATE SEQUENCE seq1 INCREACEMENT BY 2; --増加量を2にする

CREATE SEQUENCE seq1 CACHE 5; --番号の割り当てを高速化するためにキャッシュを使用

そのほか

ALTER SEQUENCE seq1 INCREMENT BY -5 MINVALUE 100 START 1000;

ALTER SEQUENCE seq1 OWNER TO user1;

ALTER SEQUENCE seq1 RENAME TO seq1\_new;

DROP SEQUENCE -- 削除

補足

・シーケンスはPostgeSQLの独自拡張なので，MySQLなどへ移行では問題が起きる可能性が高い

#### INDEX

構文

CREATE INDEX idx\_name ON table1 [ USING method ] (column1…)

・インデックスは未使用であっても整合性が取れるように管理される為，更新処理が遅くなる可能性がある。

・インデックスはデータとの整合性が取れるように自動で管理される。

・UNIQUEとPRIMARY KEYでは自動的にインデックスが作成される。

指定しない限り連番型には成らない点に注意

外部キーでは自動的にインデックスは作成されない。

CREATE TABLE department(id CHAR(4) UNIQUE, name VARCHAR);

CREATE TABLE customer(id CHAR(4) PRIMARY KEY, name VARCHAR);

INSERT INTO department (name) VALUES('管理部'); -- idの指定なし → NULLになる

INSERT INTO customer(name) VALUES('田中一郎'); -- idの指定なし → PRIMARY KEYはNULL禁止なのでエラー

・WHEREなどで対象になる列に対してインデックスを作ると最も効果的（当たり前ではあるが）。

CREATE INDEX table1\_idx ON table1(id); -- 作成

CREATE INDEX table1\_idx ON table1(id, groupName); -- 複数列に対するインデックス

REINDEX -- インデックスの修正

ALTER INDEX -- 名前のみ変更が可能

DROP INDEX member\_idx; -- tableName\_idx という形。ON tableNameは不要

関数インデックス

列値を用いた演算結果やSQL関数の結果をインデックスとして利用する事。

XML型の列を評価する場合なども使われる。

インデックスの種類(method)

|  |  |
| --- | --- |
| 種類 | 説明 |
| B-tree | もっとも一般的なインデックス。ある順番でソート可能なデータに対する等価性や範囲を問い合わせる場合に用いる  CREATE INDEX で作成されるデフォルトのインデックス |
| Hash | 単純な等価性比較で問い合わせる場合に用いる。ver9.6以前ではHashインデックスに対する操作はWALに記録されない為，クラッシュリカバリ後に別途REINDEXが必要となる。  また、ストリーミングレプリケーションに対応できていないため使用は推奨されていない。Ver10からはWALに記録されるようになり、これらの制約はなくなった。 |
| GiST | Generalized Search Tree（汎用検索ツリー）B+treeの一般化。  汎用的なインデックス実装の基盤となるインデックス種別 |
| SP-GiST | Generalized Inverted Index（汎用転置インデックス）  GiSTと同様に汎用的なインデックス実装の基盤となるインデックス種別。  SPとはSpace Partition を示し、主に分割管理されるデータ構造をインデックスファイルとして格納する。ver9.2以降からサポート |
| GIN | 汎用転置インデックス種別。インデックスの対象となる項目が複数存在するデータに用いる。  例えば全文検索で文書に含まれる単語をGINインデックスとして構築するような用途で使われる。 |
| BRIN | ブロックレンジインデックス。論理的な値の並びと物理的な並びに強い相関があり、かつ大規模なテーブルに対する範囲検索で有効になる。 |
| bloom | contribモジュールとして提供されるブルームフィルターと利用したインデックス。任意の列の組み合わせに対する等価性比較を行う場合に有効になる。 |

#### CONSTRAINT

名前付き制約

例えば以下のようにすると，名前付き制約positive\_price を持った列が作成できる。

… price numeric CONSTRAINT positive\_price CHECK (price > 0) --データ型のあとに付ける

ALTER TABLE table1 DROP CONSTRAINT constraintName; --制約の削除

列にPRIMARY KEYを付けると，自動的にtableName\_pkey という制約が作られる

\d tableName --これで確認できる

ALTER TABLE table1 DROP CONSTRAINT tableName\_pkey --PRIMARY KEYの削除 → NOT NULLだけが残る

ALTER TABLE table1 ADD CONSTRAINT ???? --再度PRIMARY KEYを付けたい場合

遅延制約チェック

その他

ALTER TABLE table1 ADD UNIQUE(no);

ALTER TABLE table1 ADD CONSTRAINT out\_key FOREIGN KEY (no) REFERENCES table2 (no);

ALTER TABLE table1 ADD CONSTRAINT check\_rule CHECK (no <= 50);

違いまとめ（編集中）

|  |  |  |  |
| --- | --- | --- | --- |
|  | 対象 | コマンド | 意味 |
| CHECK | テーブル |  | 特定の列 |
| CONSTRAINT | テーブル | ALTER TABLE table1 ADD/DROP CONSTRAINT | CHECKに名前をつけて管理しやすく |
| RULE | テーブル，ビュー | CREATE/DROP RULE | イベント発生時のテーブル更新に関する制約  指定できるeventは1つ。 |
| TRIGGER | テーブル，ビュー | CREATE/DROP TRIGGER | イベント発生時に関数を呼び出す。  eventは複数指定可能。 |

（関連）[CREATE TABLE](#_CREATE_TABLE_3)

#### RULE

テーブルまたはビューの更新(INSERT，UPDATE，DELETE)に関する制約を定める

通常，ある条件を満たさないビューに対して更新処理を行うことはできないが，ルールを用いる事によって可能となる。

構文

CREATE RULE ルール名 AS ON イベント TO テーブル又はビュー名 [ WHERE 条件 ]

DO [ALSO | INSTEAD] { NOTHING | SQLコマンド [; SQLコマンド…] }

イベント 更新処理(INSERT，UPDATE，DELETE)のうち**いずれか１つ**。

INSTEAD ここで定義するコマンドのみが実行される。（元のコマンドの処理は実行されない）

ALSO 元のコマンドの処理に加えて，ここで定義するコマンドも実行される。

例

CREATE RULE *rule1* AS ON INSERT TO *table1* DO INSTEAD NOTHING;

DROP RULE *rule1* ON *table1*;

・規定されていない更新SQLを実行するとエラーになる。例えばINSERTのみ指定されている状態でUPDATEを実行するとエラーになる。（SELECTは問題なく実行できる）

・権限必要？

・INSERT時のレコードを指定するキーワードとして、「NEW」を使用できる。

CREATE RULE *rule1* AS ON INSERT TO *table1* DO ALSO INSERT TO *table2* values(new.c1)

CREATE [ OR REPLACE ]

ALTERでは無く REPLACE RULEでルールの更新（変更）が可能。

ALTER RILE

ルールの名前だけが変更可能。

DROP RULE

DROP RULE [ IF EXISTS ] rule\_name ON target\_table [ CASCADE | RESTRICT ]

CASCADE ルールに依存するオブジェクトを自動的に削除，さらにそれらのオブジェクトに依存するすべてのオブジェクトも削除。

RESTRICT 依存するオブジェクトがある場合，ルールの削除を拒否。[def.]

例：

CREATE RULE rule1 AS ON INSERT TO view1 DO INSTEAD(

　INSERT INTO log\_memory VALUES('view1', 'INSERT', NEW, now());

);

#### TRIGGER

特定のテーブルを対象にした更新イベント発生時に指定した関数(function) を呼び出す。

CREATE TRIGGER

構文：

CREATE TRIGGER トリガー名 タイミング イベント ON テーブル名

[影響範囲]

EXECUTE PROCEDURE トリガー関数名([引数]);

タイミング

BEFORE，AFTER

タイミングがAFTERの場合はトリガーがNULLを返してもトリガーは実行される。

⇒ トリガーがNULLを返す　とは？ 2021-12-05

イベント（「タイミング」と同時に使う）以下の内いずれか複数。

INSERT OR UPDATE OR DELETE OR TRUNCATE

影響範囲

FOR [EACH] ROW 行に対するイベント毎に１回起動。

FOR [EACH] STATEMENT SQL文ごとに１回起動(def.)

トリガー関数名

・引数をとらずTRIGGER型を返す，特別な関数を事前に宣言する必要がある。

・この関数はTriggerData構造体で入力を受け取る。

・定義されていない関数を指定してトリガ作成された場合は作成エラーとなる。

CREATE FUNCTION div\_proc() RETURNS trigger AS $div\_proc$ --引数なし、戻り値trigger型の関数として宣言

CREATE FUNCTION trigger\_func1() RETURNS trigger AS $$

BEGIN

INSERT INTO answer VALUES (NEW.id, div(NEW.operand1, NEW.operand2), mod(NEW.operand1, NEW.operand2));

RETURN NEW; --trigger型変数のNEWをRETURNする。

END;

$$ LANGUAGE plpgsql;

参考：<https://www.postgresql.jp/document/11/html/plpgsql-trigger.html>

トリガ関数はNULLまたは、トリガの発行元になったテーブルの構造を正確に持ったレコードを返す必要がある。TG\_ARGV[]で無効なインデックスやDELETEを行った場合などはNULLが返る。

次ページに続く

～TRIGGER　つづき

DROP TRIGGER

・影響範囲をオプションで指定することができる。

・定義する呼び出し対象の関数が未定義の場合，トリガ作成時にエラーが発生。

ALTER TRIGGER

変更できるのはトリガの名称のみ。

ALTER TRIGGER trgger1 ON sample RENAME TO new\_trg;

例

CREATE TRIGGER trigger1 BEFORE INSERT OR UPDATE OR DELETE ON table1 EXECUTE PROCEDURE func1();

DROP TRIGGER trigger1 ON table1;

CREATE TRIGGER trigger1 BEFORE INSERT OR DELETE ON table1 FOR EACH ROW EXECUTE PROCEDURE log\_write();

補足

・使用する関数はTRIGGER型を返すように定義する必要がある。

・変更の前の値と後の値を「OLD」「NEW」キーワードに格納することができる

#### ラージオブジェクト

Large Object；LOB

巨大なバイナリ・テキストデータを格納する機能。PostgreSQL 9.0以降。

ラージオブジェクトを格納すると，そのOIDが返される。

pg\_largeobject\_metadeta 所有者，アクセス権限などのメタデータを管理。

列：oid，lomowner, lomacl

pg\_largeobject データの実態を管理。列：loid，pageno, Data

関連コマンド

\lo\_import ファイルをラージオブジェクトへ格納する。

Oid lo\_import(PGconn \*conn, const char \*filename);

\lo\_export ラージオブジェクトから，オペレーティングシステム上のファイルに出力

int lo\_export(PGconn \*conn, Oid lobjId, const char \*filename);

lo\_open 読み書きの為に既存のラージオブジェクトを開く

int lo\_open(PGconn \*conn, Oid lobjId, int mode);

注意点

・データサイズ上限はver9.3以降 であれば4TB。それ以前は2GB。

・システムカタログのテーブルに格納される。テーブルの列に格納されるスタイルとは異なる。

#### PARTITION

パーティション。

テーブルなどのデータを複数のまとまりに分割する機能。テーブルの分割保存。

論理的には１つのテーブルとして扱える。

詳細

・ver10から実装された。

・DELETE FROMではなく，DROP TABLEで特定レコード（パーティション）を削除できる為低負荷。

partition key

リスト・パーティショニング

値によって分割する方法

LIST(key\_column)

CREATE TABLE club (id int, club varchar)

CREATE TABLE baseball PARTITION OF club FOR VALUES IN ('野球部');

レンジ・パーティショニング

値の範囲によって分割する方法

CREATE TABLE log(user varchar, date date) PARTITION BY RANGE(date);

CREATE TABLE log\_2017 PARTITION OF log FOR AVLUES FROM(‘2017-01-01’) TO (‘2017-12-31’);

ハッシュ・パーティショニング

ハッシュ値を使用して、ほぼ均等にデータを分割する方法

CREATE TABLE items(id int, name varchar) PARTITION BY HASH(id);

CREATE TABLE PARTITION item0 OF items FOR AVLUES WITH(MODULES 3, REMAINDER 0);

コマンド

ALTER TABLE {ATTACH | DETACH} PARTITION

ALTER TABLE parent\_table ATTACH PARTITION child\_table FOR VALUES IN ('バレー部');

DETACH 親テーブルから対象パーティションを切り離し，独立した別のテーブルを作成する。

ATTACH 別のテーブルをパーティションとして親テーブルに組み込む。

### 関数

#### 概要

以下のような形で使用する（らしい）

SELECT concat(c,v) FROM db\_name;

大きく分けて以下のような関数の種類がある

|  |  |  |
| --- | --- | --- |
| 名称 | 説明 | 公式資料 |
| 比較関数 |  | <https://www.postgresql.jp/document/11/html/functions-comparison.html> |
| 算術関数 |  | <https://www.postgresql.jp/document/11/html/functions-math.html> |
| 文字関数 |  | <https://www.postgresql.jp/document/11/html/functions-string.html> |
| 配列関数 |  | <https://www.postgresql.jp/document/11/html/functions-array.html> |
| 範囲関数 |  | <https://www.postgresql.jp/document/11/html/functions-range.html> |
| ウィンドウ関数 |  | <https://www.postgresql.jp/document/11/html/functions-window.html> |
| 日付/時刻関数 |  | <https://www.postgresql.jp/document/11/html/functions-datetime.html> |
| 集約関数 |  | <https://www.postgresql.jp/document/11/html/functions-aggregate.html> |

#### カテゴリ不明

collate

concat

文字列の結合？（比較）||

current\_schema()

現在のスキーマが取得できる

SELECT current\_schema();

#### 算術関数

abs 絶対値を返す。

SELECT avg(age) FROM members;

div() 商を返す。div(arg1, arg2)でarg1÷arg2

mod() 余を返す。mod(arg1, arg2)でarg1％arg2

floor() 切り捨て整数化。（1.34→1）

ceil() 切り上げ整数化。（1.34→2）

round() 四捨五入 つまりround upということ。

SELECT round(12.545); -- 結果：13

SELECT round(12.545, 2); -- 結果：12.55

trunc() 数値の小数部分を指定した位で切り捨てる

SELECT trunc(12.545); -- 結果：12

SELECT trunc(12.545, 2); -- 結果：12.54

random 0以上1**未満**の乱数

#### 文字列関数

lower 小文字にして返す。

upper 大文字にして返す。

char\_length 文字数を返す。

SELECT char\_length(‘アイウエオ’) # ⇒ 5

octet\_length バイト数を返す。

SELECT octet\_length(‘アイウエオ’) # ⇒ 15

trim 指定文字列を取り除く。( [arg, 除去する文字列] )

SELECT trim(‘ABCde’, ‘ABC’) # ⇒ de

lpad 先頭に文字列を追加。( arg, 文字数, [追加する文字列] )

SELECT lpad(‘test’, 5, ‘z’); # ⇒ ztest

rpad 末尾に文字列を追加。(arg, 文字数, [追加する文字列] )

SELECT rpad(‘test’, 5, ‘z’); # ⇒ testz

substring 文字列を部分的に取得する。( arg, 開始位置, [文字数] )

他の言語などだとsubstringの開始位置は0から始まるが，postgreSQLでは**1から始まる**点に注意。

replace 文字列を置き換える。( arg, 対象パターン, 置き換え文字列 )

#### 日付/時間関数

<https://www.postgresql.jp/document/11/html/functions-datetime.html>

extract

インターバル値から指定したフィールドの値のみを取得

構文：extract(フィールド FROM source)

SELECT extract(day FROM timestamp '2000-12-16 12:21:13'); --タイムスタンプから取得

SELECT extract(minute FROM now()); --現在時刻から取得

フィールド

century世紀，month，day日, decade10年，dow　曜日（day of the week），hour, second, microseconds

doy 年内での通算日数（1〜365/366），week 何週目かを示す。ISO 8601週番号

isodow 日曜日が7になる, isoyear

epoch　 to\_timestamp

date\_part

extractと同じ結果を返す関数。

構文：date\_part('フィールド', interval 'インターバル値')

但し第一引数が **’’**（シングルクォーテーション）付きな点に注意。

SELECT date\_part('day', timestamp '2000-12-16 12:21:13'); --タイムスタンプから取得

SELECT date\_part('minute', current\_timestamp); --現在時刻から取得

SELECT now() – interval ‘month’; -- 補足：interval はこんな感じで使用

次ページへ続く

日付/時間関数　つづき

**関数を呼び出した時の時刻**を返す。（非標準SQL）

SELECT clock\_timestamp() --タイムスタンプ型を返す。

**トランザクションを開始した時**を基準とする。

SELECT now(); -- タイムスタンプ型を返す。

SELECT current\_timestamp; -- かっこ（）が無い。now() と全く同じでタイムスタンプ型を返す。

SELECT current\_date -- DATE型を返す。(currentとあるのに現在時間ではない)

SELECT current\_time -- TIME型を返す。(currentとあるのに現在時間ではない)

そのほか

statement\_timestamp() タイムスタンプ型で**現在の文の開始時間**を返す

age タイムスタンプ間の差分を取得する

構文：age(タイムスタンプ値, タイムスタンプ値)

SELECT age(timestamp '20170707', timestamp '20150327'); --2015年3月27日から2017年7月7日の差分

#### 集約関数

注意点

・countを除き，1行も選択されなかった場合NULL値を返す。

count

NULL以外のデータ件数を返す。bigint(int8)型の値を返す。

構文：count(列名)

SELECT groupName FROM sample GROUP BY groupName HAVING count(age) >= 2;

--ageがNULL以外のレコ―ドが2件以上あるレコードを，groupNameでグループ化する

SELECT count(group\_id) FROM manage; -- 単にmanageテーブルのgroup\_id IS NOT NULL のレコード数を表示

count(\*) 入力行数の数。

NULL値でも対象？ ⇒ 恐らく， count(列名)だと値がNULLの行は対象とならない事を強調しているのだと思われる。

cout(\*) は純粋なレコード数のみ見ている。仮に全ての列の値がNULLでもレコード数としてカウントされる。

sum()

指定した列の合計値を返す。数値型の列のみが対象。NULLは無視される。

SELECT sum(amount) FROM soldItems;

max() 最大値を返す。数値，日付，時刻，文字列が対象。

min() 最小値を返す。数値，日付，時刻，文字列が対象。

文字列型で使った時は文字コード値で比較される。あくまで文字列として比較される。

CASE WHENと併用すると

SELECT SUM(CASE WHEN flag = 1 THEN 1 ELSE 0 END) FROM my\_table

比較：Count

#### システム情報関数

version() postgreSQLのバージョンを返す

select version(); -- このように使用。（ほかの方法でも確かめられるので，あまり使うことはない気がする）

current\_database() 現在接続しているデータベース情報を取得

current\_user コマンドを実行したユーザー情報を表示

SELECT current\_user; -- 関数なのに()が必要ないので注意 個人的にはめちゃくちゃ違和感ある

current\_timestamp 現在の日時を取得

current\_date 現在の日付を取得

currentはカッコなし。（但しcurrent\_database以外）「car rentは**カッコ**よく**ない**」で覚える。

#### データ型書式設定関数

to\_char() int型などの値を文字列型に変換する。

SELECT to\_char(now(), 'YYYY-MM-DD HH24:MI:SS');

to\_date() 文字列型の値を日付型に変換する。

SELECT to\_date('20170707', 'YYYYMMDD');

to\_timestamp() 文字列型の値をtimestamp型に変換する。

to\_number() 文字列型の値を数値型に変換する。

to\_number('20170707', '99999999');

#### その他

### その他コマンド

#### REFRESH

#### VACCUM

手動でバキューム処理を行う。バキューム対象に対しての所有者権限が必要。

（関連）バキューム

構文：VACUUM [ (オプション…) ] [テーブル名 [(column\_name…]] ];

テーブル名

省略すると，現在のユーザーが権限を持つ現在のデータベース内の全てのテーブルとマテリアライズドビューを処理。

オプション（同時に２つ以上使える）

FULL 広い範囲の回収を行う。コマンドの実行後、**テーブルサイズは縮小**する

テーブルに対し排他ロック（ACCESS EXCLUSIVE）を取得する。

FREEZE 積極的なタプルの「凍結」

VERBOSE バキューム処理の詳細な報告を出力

ANALYZE プランナが使用する**統計情報を更新**

SHARE UPDATE EXCLUSIVEを取得する。（テーブルの作成などが実行可能）

DISABLE\_PAGE\_SKIPPING

例

postgres=# VACUUM table1;

（関連）[バキューム](#_バキューム)

#### ANALYZE

効率的にSQLを実行するために、統計情報を収集して更新する。

実行を怠ると，効率の良いSQLの実行計画が作成できなくなる。

構文：ANALYZE [テーブル名];

・テーブル名を指定しない場合は現在のデータベースの中のすべての通常のテーブルが解析される。

・実行するとテーブルに対して，ロック；SHARE UPDATE EXCLUSIVEが取得される。

#### CLUSTER

インデックスを使用して、テーブルのデータを物理的に並び替える

#### REINDEX

インデックスを作り直し、インデックス内に生じてしまった空領域を解消する

構文：REINDEX [ ( VERBOSE ) ] { INDEX | TABLE | SCHEMA | DATABASE | SYSTEM } name

#### COPY

**サーバー側のファイル**とテーブル間のデータをコピーする

構文：COPY テーブル名 FROM {'**絶対パス**のファイル名' | STDIN} [WITH] [オプション [, ...] ];

構文：COPY テーブル名 TO {'**絶対パス**のファイル名' | STDOUT} [WITH] [オプション [, ...] ];

・**標準出力の以外**の場合（ファイル読書き）は**スーパーユーザー権限**が必要。ない場合エラー。

・読み書きの対象はpostgreSQLサーバー機。

・デフォルトのファイル形式はタブ区切り。

オプション

WITHの使い方例

COPY table1 FROM '/Users/test.csv' WITH (FORMAT csv, HEADER); -- WITH後に「()」（カッコ）があっても無くても良い

DELIMITER ‘char’ 区切り文字

COPY table1 TO STDOUT (DELIMITER '|');

\copy table1 to sample.csv csv -- \copy ではdelimiter [as] は省略できるらしい

デフォルトの区切り文字 テキスト形式：タブ CSV形式：コンマ

FORMAT format テキストの形式を指定。 text (def.)，csv，binary

COPY table1 FROM '/Users/local/sample.csv' WITH (FORMAT csv);

HEADER テキストの1行目を列ヘッダとして無視する

COPY table1 FROM '/Users/local/test.txt' WITH HEADER;

\copy table1 from test.txt header -- クライアントコマンド\copy の場合と比較

bashでやる場合

$ psql -c ‘COPY table1 TO STDOUT;’

例文

COPY table1 TO STDOUT; -- 標準出力

COPY table1 TO '/Users/local/sample.csv' WITH (FORMAT csv); -- table1をCSV形式でサーバー側に保存

（参考）[\copy](#_copy) クライアント側のファイルとテーブル間コピー

#### EXPLAIN

実行計画

EXPLAIN SELECT \* FROM table1 WHERE value =120;

⇒ costとかwidthとか表示されるが意味不明 2021-09-03

#### LOCK TABLE

明示的にテーブル全体のロックを取得する。

任意のロックモードが選択できるのが特徴。

構文：LOCK [ TABLE ] [ ONLY ] name [\*] [ IN *lockmode* MODE ] [ NOWAIT ]

例

LOCK TABLE *items* IN ACCESS EXCLUSIVE MODE;

（関連）[ロックモード一覧](#_ロックモード一覧)

## 行先不明

UNLOGGED TABLE 信頼性が落ちる反面，高速化

テーブル

スキーマの中に作成される。

複数バージョンのPostgreSQLをインストールする方法

## その他情報

### クライアントGUI

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Win | Linux | Mac | 説明 | 感想／Web |
| DBeaver | 〇 | 〇 | 〇 | 様々なDBに対応し，人気。  ポータブル版有り。 | ・画もきれいだし使いやすい。  <https://www.heidisql.com/> |
| HeidiSQL | 〇 | ― | ？ | オープンソースなのでライセンス無料。  ポータブル版あり。 | ・SQLコマンドが見えるのは面白い。  ・データ型がカテゴリ別に分かれており，分かりやすい  <https://www.heidisql.com/> |
| pgAdmin | 〇 | 〇 | ？ | サーバー側にPostgreSQLをインストール時に付属するアプリ。 | <https://www.pgadmin.org/> |
| A5:SQL Mk-2 | 〇 | ― | ― | 有志によるフリーソフト。 | <https://a5m2.mmatsubara.com/> |

その他

phpPgAdmin Webベースのツール。遅延がすごいらしい

Postico

PG Commander

Navicat Linux対応？

MySQL Workbench

TeamSQL 人気アプリだったが2019年2月末にサービス終了。

### pythonから使う

import psycopg2

conn　= psycopg2.connect("host=192.168.1.1 port=5432 dbname=*sampledb* user=*user1* password=pssword")

cur = conn.cursor() # カーソルの作成

cur.execute("SELECT \* FROM test;") # SQLの実行

one\_row = cur.fetchone() # 1レコード取り出す

rows = cur.fetchall() # まとめて取得

for row in cur: # ひとつずつ取得

print(row)

colnames = [col.name for col in cur.description] # coloum 名はdescriptionに含まれている

# 引数の渡し方に癖がある

cur.execute("INSERT INTO foo VALUES (%s)", "bar") # だめ

cur.execute("INSERT INTO foo VALUES (%s)", ("bar")) # だめ

cur.execute("INSERT INTO foo VALUES (%s)", ("bar",)) # OK

cur.execute("INSERT INTO foo VALUES (%s)", ["bar"]) # OK

cur.close() # 解放

conn.close() # ※最初から with で実行する手もある。

fetchone, fetchall

### html（Webブラウザ）で使う

フレームワークを使う

Django

html上でデータベースの管理ができたりする。

中～大規模向き

Flask

とにかくシンプル。1ファイルでもWebアプリケーションを実装できる。

必要なライブラリを自分で選択できる。（逆に言うと選択しなければならない）

Rails

Rubyを使うフレームワーク。

PHPを使う

サーバーサイドで動くのでクライアント機が何であれ使いやすい。

時代遅れ間は否めない。

### 他の種類のデータベース

XMLデータベース

XMLを扱うための機能をもっており，拡張性が非常に高い

階層型データベース

古いデータベースモデルで，データを木構造のように構築する

データを上から下へと見ていくために親データと子データという関係で高速に検索できる

Entityを階層型に管理する

ネットワーク型データベース

古いデータベースモデルで，各レコード間の関係付けが網目状になっている

カード型データベース

レコードを，必要なだけ単純に累積する構造を持つデータベースです。

## 用語(postgre sql)

### その他用語

btree 計算機科学におけるデータ構造，特に木構造の一つ。MySQLなどでも採用されている

indexによく使われている。

DBMS database management system

GSSAPI RFC 2743で定義されている安全な認証のための業界標準のプロトコル

hot\_standby スレーブDBに対して参照のみ可能で更新ができない状態。

index データに目次（index）をつける事で検索を高速化する。

peer認証 PostgreSQLのユーザー名とOSのユーザー名が一致するかで認証する方法。

peer to peer サーバー，クライアントの区別がなく，接続機が対等な関係なネットワーク形態の事。

postgres 最初に唯一設定されているスーパーユーザー。最初はこのユーザーを使う。

replication 可用性を高めるための複製の事。一般的にデータベースでは使われる。

shmmax 共有メモリのパラメータ。１プロセスごとの共有メモリの数の上限を設定する。

TOAST

過大属性格納技法：The Oversized-Attribute Storage Technique

データベースで大きなデータを扱うための方法。

TOASTで使うにはデータ型は可変長（varlena）表現をつける必要がある。

テーブル空間

テーブル定義 列名，主キー制約などのテーブルの定義情報の事。

リレーション テーブルの事。数学用語らしい。

チェックポイント WALの情報から得られるデータをディスクに反映。＝リカバリポイントと考える。

セッション PostgreSQLサーバーへの接続1回の1単位

# mysql

## 概要

注意点

ラズパイではmysql-server が使えない。代わりにmariadb-serverを使うように言われる。(2022-01-11)

補足：mariadbについて

オラクル社がmysqlを所有していたサン・マイクロシステムズを買収する時、共同設立者のミカエル・ウィデニウスがmysqlプロジェクトをフォークして作ったソフト。

補足

3.6.19以前は外部キーをサポートしない。

## 詳細

### パーティション

CREATE TABLE `logs` (

`logged\_at` datetime NOT NULL,

.,

)

PARTITION BY RANGE (year(`logged\_at`)) (

PARTITION p2019 VALUES LESS THAN (2020),

PARTITION p2020 VALUES LESS THAN (2021),

PARTITION p2021 VALUES LESS THAN (2022),

PARTITION pmax VALUES LESS THAN MAXVALUE

);

<https://www.ritolab.com/entry/236>

クライアントからのコマンド

@IP の形式となる（毎回？？ 2022-12-20）

@’%’ % は0文字以上にマッチ \_は1文字以上にマッチ

CREATE USER [HOGE@192.168.1.1](mailto:HOGE@192.168.1.1) INDENTIFIED BY ‘abcdefg’;

メモ

最初からいる管理者の名前はroot

auth\_socket

UNIXソケットを利用し、UNIX OSのユーザー名と同じ名前のMySQLアカウントにパスワードなしでログインできる

インストール

INSTALL PLUGIN auth\_socket SONAME ‘auth\_socket.so’;

ログイン

CREATE USER hoge@hocalhost IDENTIFIED WITH auth\_socktet;

### システム用スキーマ

VARIABLES

# SQLite

## 概要

・読み込みは早いが、書き込みはレコード数が多くなるともたつく。

・最大レコード数は約270,000行

## メモ

db.sqlite3-journal

# SQLServer

## 概要

## チュートリアル

### SQLServer Mangement Studio

GUIかつ、さして難しくなかったので、詳細資料を残すのは後回しとする。2022-02-04

#### ローカルのSQL Serverに接続

ConnectObjectExplorer→Server typeにDatabase Engineを選択

→ Server nameのコンボボックスの矢印をクリック→<Browse for more>

→ Local Server タブを選択 → TreeViewのDatabase Engineを開く→ PC名¥SQLEXPRESSを選択

### リモート機から（GUI）

### エクセルから表示

#### 概要

SQLServer、エクセルともにMicrosoft製品だけあって、接続ドライバの追加インストールなしで接続できる。

#### 操作

データ → 外部データの取り込み → その他のデータソース → SQLServer

サーバー名：**URLを入力**する

※「サーバー名」というラベル文字列の箇所にURL入力は初見殺し過ぎる。

ログイン時の照明

Windows認証 localhostのデータベースサーバーの場合

パスワード 外部データベースの場合

#### 接続ファイル

C:\Users\[user\_name]\Documents\My Data Sources

## 詳細

### 接続ドライバ

#### JDBCドライバ

#### pyodbc

pythonで使用するドライバ。

サンプル

python

#!/usr/bin/env python

import pyodbc

# Some other example server values are

server = 'localhost\SQLEXPRESS' # for a named instance

# server = 'myserver,port' # to specify an alternate port

# server = 'tcp:myserver.database.windows.net'

database = 'seigi\_app'

username = 'user\_nama'

password = 'password'

cnxn = pyodbc.connect('DRIVER={ODBC Driver 17 for SQL Server};SERVER='+server+';DATABASE='+database+';UID='+username+';PWD='+ password)

cursor = cnxn.cursor()

公式：<https://docs.microsoft.com/ja-jp/sql/connect/python/pyodbc/step-3-proof-of-concept-connecting-to-sql-using-pyodbc?view=sql-server-ver15#connect>

### 列のプロパティ

|  |  |  |
| --- | --- | --- |
| プロパティ名 | 日本語 | 概要 |
| Identity Seed |  | 初期値 |
| Identity Increment | IDの増分 | 増分 |

### パーティション

SQL ServerMnagerを使用

ALTER DATABASE [my\_table]

ADD FILEGROUP [my\_table\_group1]

2022-04-14に調べた感じ、学習に最低でも半日以上かかりそう

（PostgreSQLもMySQLもパーティション作成は簡単なのに…）

補足情報

以下のデータ型はインデックス作成ができないので、パーティションでも使用不可

timestamp、ntext、text、image、xml、varchar(max)、nvarchar(max)、varbinary(max)

CLRユーザ定義データ型、別名データ（エイリアス データ）型

ファイルグループの作成

ALTER DATABASE [mydb]

ADD FILEGROUP [FG2019] # FG2018, FG2019, FG2020を作成

データベースを右クリック→プロパティ→ファイルグループ　で確認可能

パーティション関数を作成

CREATE PARTITION FUNCTION funcPartitionDate(datetime) AS RANGE RIGHT FOR VALUES (

,'2019/01/01' ,'2020/01/01');

データベースを右クリック→ストレージ→パーティション関数　で確認可能

RIGHTとLEFTの違い

|  |  |  |  |
| --- | --- | --- | --- |
| pt | RIGHT | LEFT | 備考 |
| 1 | column < 2018/01/01 | column <= 2018/01/01 | 11 |
| 2 | column >= 2018/01/01 AND column < 2019/01/01 | column > 2018/01/01 AND column <= 2019/01/01 | 1 |
| 3 | column >= 2019/01/01 | column > 2019/01/01 | 1 |

補足：datetime2を使え　という情報？

CREATE PARTITION FUNCTION myRangePF1 (datetime2(0))

AS RANGE RIGHT FOR VALUES ('2022-04-01', '2022-05-01', '2022-06-01') ;

（参考）<https://qiita.com/fuk101/items/603ddb445070aebbfcfa>

（参考）<https://techinfoofmicrosofttech.osscons.jp/index.php?SQL%20Server%20パーティション分割>

（参考）<https://blog.withonoware.co.jp/tech/2021/03/08/sql-server-のデータパーティションについて/>

### データ型

#### 数値系

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 別名 | byte数 | 説明 |
| tinyint |  | 1 | 0~255　C言語のbyteに相当 |
| smallint |  | 2 | 符号付き整数　-32,768～32,767　C言語のshortに相当 |
| int |  | 4 | 符号付き整数　-2bills～2bills　C言語のintに相当 |
| bigint |  | 8 | 符号付き整数　±9,223,372,036,854,775,808 |

（情報元）<https://docs.microsoft.com/ja-jp/sql/t-sql/data-types/int-bigint-smallint-and-tinyint-transact-sql?view=sql-server-ver15>

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 別名 | byte数 | 説明 |
| datetime2 |  |  | datetimeの秒の小数部の有効桁数を0～7桁の間で変更可能。  参考<https://docs.microsoft.com/ja-jp/sql/t-sql/data-types/datetime2-transact-sql?view=sql-server-ver15> |

### 制約（Constraint）

UQ\_\_

NULLを許容するUnique constraintはSQLServerでは対応できないらしい。

（参考までに、ほかの主要RDBSでは対応している）

### テーブル／行にコメント

#### テーブル

EXEC sys.sp\_addextendedproperty @name=N'MS\_Description' -- 固定値

, @value=N'<テーブル論理名>'

, @level0type=N'SCHEMA'

, @level0name=N'<スキーマ名>'

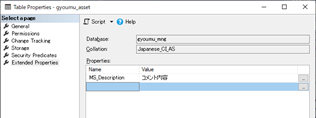
, @level1type=N'TABLE'

, @level1name=N'<テーブル名>'

具体例

EXEC sys.sp\_addextendedproperty @name=N'MS\_Description'

,@value=N'コメント内容'

,@level0type=N'SCHEMA'

,@level0name=N'dbo'

,@level1type=N'TABLE'

,@level1name=N'my\_table'

結果は右図参照

#### 行

EXEC sys.sp\_addextendedproperty @name=N'MS\_Description'

,@value=N'番号'

,@level0type=N'SCHEMA'

,@level0name=N'dbo'

,@level1type=N'TABLE'

,@level1name=N'test\_table'

,@level2type=N'COLUMN'

,@level2name=N'no'

SQLで使う場合

SELECT TOP (1000) no, name,,ep.value As comment,

FROM my\_table, sys.extended\_properties AS ep

補足

列プロパティにDescription と言うプロパティもある

### 注意点

文字列はシングルクォーテーションじゃないといけない？

### 関数

CONVERT(date,アクセス日時)

SELECT \* FROM production\_log

WHERE CONVERT(date, create\_at)='2022-07-28'

## 資料

### 問題点

・正規表現による検索がサポートされていない。

・GroupByをともなうサブクエリの発行ができない

・パーティションの作成が困難

・Djangoへの対応（Django 2.1にしか対応できない事）

- OneToOneFieldに非対応（SQLSever側のバグが原因らしい）

- bulk\_updateに非対応

・ALTAR TABLEでprimary keyを変更できない

## SQL Server Manegement Studio

### 管理

#### バックアップ

データベースを選択 → Task → backup

#### エキスポート（dump）

データベースを選択 → Task → Export

（テーブル単位のdumpファイルは作れない？）

CSV：Flat File Source

Excel：

一度やってみたが、めんどくさくてやってられない。

ExcelからアクセスしてCSVに保存した方が早い。

### ER図の表示

データベースダイアグラム

Database Diagrams→New Database Diagram

### DB初期化

### Auto Increment

IDフィールドであれば以下の方法で設定の変更／確認ができる。

・テーブルを選択して右クリック→Design→任意の列を選択→Column Propertiesの中のIdentity Specificationを選択→

・IDフィールドであれば(Is Identity) がYesになっているはず。（なっていなければ→Yesに）

・Identity Incrementが増分

### 列の追加

・テーブルを選択して右クリック→Design

・一番下の行を編集し、新規列を追加。

補足：「Nullを許容（Allow Nulls）」を強く推奨！！

・メニュー：File→Save my\_table\_nameで保存

・Ctrl+Shift+Rでキャッシュを更新（任意）

### メモ

#### IDをもとに戻す

→以下のクエリを入力 → 

SQL

DBCC CHECKIDENT ('tablename', RESEED, 0);

## その他

# データベース管理ツール

## データベース管理ツール

### DBeaver

Windows、Mac、Linuxに対応。

Enterprise Editionあり。

レコードの追加

　新しい行の追加

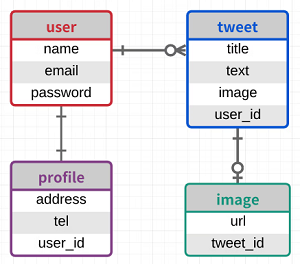
をクリックすると，今実行しようとしているトランザクションのSQLコードがみられる。

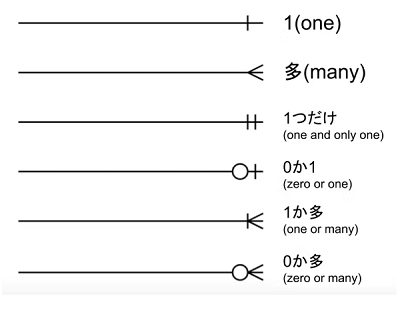
### HeidiSQL

Windows専用

## ER図

### 概要





### 使い方

#### 行の追加

### diagrams.net（旧：Draw.io）

<https://app.diagrams.net/?src=about>

#### VSCode

プラグイン：Draw.io Integration

.drawioまたは.dioという拡張子でファイルを作成する。

（補足）.drawio.pngとかdrawio.svgでファイルを生成すると、importやexportの必要なくそのまま画像として利用できたり、編集する

でERと入力しエンター。

例

# 用語

シャーディング

sharding　テーブルの水平分割。（Horizontal Partitioning）

特定の列の値によって、保存先のストレージを選択する方式のパーティショニング。

レコードIDを素数で割った余りで分散させる方法がよくとられる。例 ID÷3の余り。

Assertion

表明とも呼ばれる。スキーマ内の複数のテーブルに対して制約を適用する。

CREATE ASSERTION句で指定。

JDBC

JavaScript用のODBCの事

射影

Projection。

テーブルから指定された列を抽出する事。

（関連）選択；selection

商

表Aのうち表Bのすべての項目を含む行の集合と返す。（INの事？）

スケールアウト サーバ台数を増やして、複数台のサーバで処理を分散化・並列化すること

保全機能

排他制御や参照制約、表制約などによってデータの完全性を保つ機能。

フルマネージド

、組織のITスタッフではなく、サードパーティ・プロバイダによって管理および維持される、ストレージ、データおよびコンピュート・サービスを含むデータベース

2相コミットメント

複数のデータベースノードに跨って作用するトランザクション

# 資料

<https://scrapbox.io/gosyujin/SQLアンチパターンで言及されている名称>