PostgreSQLデータベース

データベースとは

Databaseという単語自体は造語であり、Dataは情報。Baseは基地という意味になります。
第二次大戦後の米軍が、膨大な量の資料をひとつの基地に集約して効率化を図ったことから、
「データベース」という言葉が生まれました。

データベースは、ただのデータの集まりではなく、それを利用するユーザー又はプログラムが、
必要とするデータを効率的に取得することができるように整頓されたデータの集まりなのです。

そのデータベースと利用者との間に介在し、利用者からの指示によって、データベースに対する操作
（テーブル定義、データの追加・修正・削除・検索など）を行ったり、複数ユーザーからの同時制御を
管理したりするシステムを**「データベース管理システム（DBMS）」**と言います。

今回みなさんが使用するデータベースは、**PostgreSQL**というデータベースです。

データベースの特徴

データベースの特徴としては下記のようなものが挙げられます。

・データの非重複性
データベースとして情報を一元管理するためデータに重複がない。

・同時処理の実現
同時の書き込みや削除などは、DBMS が制御するため、データの矛盾が発生することはない。

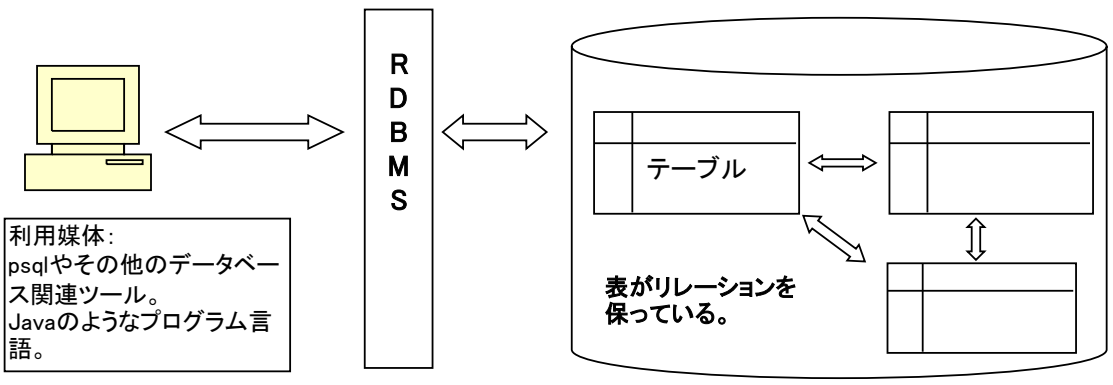
・データの機密性
データベースへのユーザのアクセスをDBMS によって制御できる。（アクセス制御）

・データの障害回復機能
データベースに何らかの障害が発生した場合でも、DBMSはこれを回復するための手段を用意
している。

リレーショナル型データベース（RDBMS）

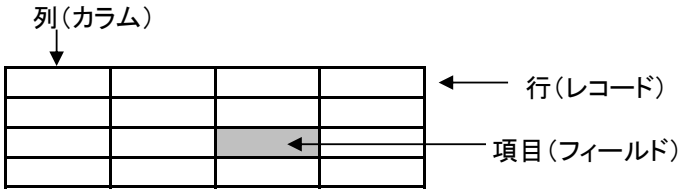
DBMSのデータ管理の概念には、階層型、ネットワーク型、リレーショナル型、オブジェクト指向型など、
歴史的に様々なタイプのモデルが存在しています。
しかし現在は、リレーショナル型が圧倒的なシェアを持っており、現在利用されているほとんどのDBMS
がこの種類になります。
リレーショナル型のデータベース管理システムのことを、特にリレーショナルデータベース管理システム
（Relational Database Management System: **RDBMS**）と言います。

リレーショナル型データベースの特徴は下記のようなものです。
・データを2次元の表で管理する。
・複数の表を一元管理することで、巨大なデータベースを構築できる。
・リレーショナル型データベースに対応するデータベース言語として**SQL**が使用されている。



テーブル

データベースは、データを2次元の表で管理しています。
その表のことを、一般的に**「テーブル」**と呼びます。
表の横のまとまりが**「行」**で、縦のまとまりが**「列」**になります。



SQL

リレーショナル型データベースは、**問い合わせ言語（クエリー言語）**を用いることで、効率的に目的のデータを取得することができます。
この問い合わせ言語は、現在は、**「SQL（Structured Query Language）」**という言語が標準となっているので、ほとんどのデータベース製品で共通に使うことができます。

> SQLは、1974年に考案されたSEQUEL（シークエル）言語を前身とし、ANSI（米国規格協会）/ISO（国際標準化機構）により標準化が進められ、
> 1992年に「SQL-92」という規格が定められた。
> その後、1999年には、オブジェクト指向の考え方を取り入れた規格として、
> 「SQL-99」が標準化された。

SQL はデータの取り出し（検索）だけではなく、テーブルの作成、レコードの追加、削除、更新なども行うことができます。

SQLの命令文をカテゴリ分けすると以下のようなものがあります。

・データ操作言語（DML：Data Manipulation Language）
データの検索・追加・更新・削除などを行う
SELECT、INSERT、UPDATE、DELETE

・データ定義言語（DDL：Data Definition Language）
テーブルの作成・変更・削除などを行う
CREATE、ALTER、DROP

・データ制御言語（DCL：Data Control Language）
権限の付与や剥奪、トランザクションの制御などを行う
（トランザクションとはデータの論理的な整合性を確保する一連の処理単位のこと）
GRANT、REVOKE、COMMIT、ROLLBACK

ユーザーの概念

データベースには、Linuxと同じようなユーザーの概念があります。
どのユーザーはどのテーブルを操作できるかなど、権限が分けられています。
postgreSQLでは**postgres**ユーザーが最高権限のユーザーになります。
このpostgresユーザーだけは、Linux上にも同じ名称のユーザーを用意しておく必要があります。

データベース製品について

データベース市場は、シェアで見るとOracle(日本オラクル)、DB2（IBM）、SQL Server（Microsoft）の3つが安定した地位を確立していますが、近年のLinux市場の台頭により、オープンソースのデータベース製品である、PostgreSQLやMySQLを採用する企業も増えています。

・フリー　・・・　PostgreSQL、MySQL
・商用　　・・・　Oracle、DB2、SQL Server、Sybaseなど

PostgreSQLの起動と停止

では、早速PostgreSQLを使いながら、SQL操作に慣れていきたいと思います。
PostgreSQLの起動は、postgresユーザーで行います。

起動

①postgresユーザーになる。「su － postgres」
パスワード(himitu)を入力します。
②以下のコマンドで起動します。
「pg_ctl start」
→これにより、postmasterが起動する。
postmasterは、クライアントからの接続要求を待ち受け、postgresとのやり取りを仲介します。

停止・再起動

停止は、**「pg_ctl stop」**
再起動は、**「pg_ctl restart」**と入力します。

まずは、起動したままの状態にしておいてください。

psqlの使用

PosrgreSQLのデータベースに対して接続し、SQL文を発行するための標準的なクライアントとして、**psql**というものが付属しています。

まず、以下のコマンドでデータベースにアクセスします。

```
psql 〈データベース名〉
psql 〈データベース名〉 〈ユーザー名〉
psql -U 〈ユーザー名〉 〈データベース〉
```

これでPostgreSQLへの接続が完了し、以下のプロンプトが表示されます。
jmdtbs=>

では、psqlの基本的なコマンドとして下記のコマンドを打ってみましょう。
jmdtbs=>**¥l （¥マークはLinuxではバックスラッシュ）**
～データベースの一覧が表示される～
jmdtbs=>**¥d**
～現在アクセスしているデータベースのテーブルの一覧が表示される～
jmdtbs=>**¥d test**
～testデータベースのテーブル詳細情報が表示される～

確認できましたか？

SQL参考資料

SQL文

SQLでは、文の終わりをセミコロン「;」で表します。
途中で改行が入っていても、セミコロンまでが1つの文として認識されます。
大文字小文字は区別しません。

SELECT文

データベース内のデータ（行）を取り出します。

```
SELECT 列名 , 列名 , .... FROM  テーブル名;
```

テーブルから全ての列を取得する場合は、以下のように「＊」指定できます。

```
SELECT ＊ FROM  テーブル名;
```

WHERE 句を用いることで条件を付けてデータを参照することができます。

例：顧客コード='0003'の情報を取り出したい時

```
SELECT ＊ FROM テーブル名 WHERE 顧客コード = '0003';
```

INSERT文

テーブルへデータを入力します。

```
INSERT INTO テーブル名 (列名 , 列名 ,.....)
      VALUES(データ , データ ,.....);
```

VALUES の後ろのカッコには、実際に入力するデータを記述します。
列名を指定しない場合は、テーブルの全列が対象となる。

UPDATE文

データの変更（更新）をします。

```
UPDATE テーブル名 SET 列名 = データ, 列名 = データ ,.... ;
```

DELETE文

データの削除を行います。

```
DELETE FROM テーブル名 ;
```

ここまでの、UPDATE文やDELETE文においても、WHERE句を使って、条件に合致するレコードだけを更新したり、削除したりすることができます。

CREATE TABLE文

テーブルを作成します。

```
CREATE TABLE テーブル名 (列名 データ型 , .....);
```

CREATE TABLE の後にテーブル名を書き、その後ろにカッコで囲んでフィールド名称（列名）とデータ型を設定します。

テーブルの列（フィールド）には、型という概念があります。
この型の種類は、データベース製品によって用意されているものが異なります。
PostgreSQLでよく使われる型を挙げておきます。

```
text ・・ 長さ無制限の可変長文字列
integer ・・ 4バイトの符号付整数
DATE ・・ 日付
TIMESTAMP ・・ 日付と時刻
```

PostgreSQLのデータ型

数値型

SMALLINT（INT2）	2バイト符号付整数
INTEGER（INT4）	4バイト符号付整数
BIGINT	8バイト符号付整数

浮動小数点型

FLOAT4（REAL）	4バイト浮動小数点
FLOAT8 （DOUBLE PRECISION）	8バイト浮動小数点

日付型

DATA	日付
TIME	時刻
TIME WITH TIME ZONE	タイムゾーン付き時刻
TIMESTAMP	日付と時刻
INTERVAL	期間

文字列型

CHARACTER（CHAR）	文字列。
CHARACTER(n) （CHAR(n)）	固定長文字列。
CHARACTER VARYING(n) （VARCHAR(n)）	可変長文字列。
TEXT	長さ無制限の可変長文字列。

日付関数

CURRENT_DATE ・・ 現在の日付を得る
CURRENT_TIME ・・ 現在の時刻を得る
CURRENT_TIMESTAMP ・・ 現在の日時を得る

例： select current_date

変換関数

文字列変換関数：
TO_CHAR (文字列に変換したい式 , 表記フォーマット)

日付型変換関数：
TO_DATE (日付に変換したい式 , 表記フォーマット)

日付フォーマットについて
'Y' ・・ 年を表す（桁数分'Y'を並べる）
'MM' ・・ 月（1～12）
'DD' ・・ 日(1～31)
'DY' ・・ 曜日
'HH' ・・ 時
'HH24' ・・ 時（24時間制）
'HH12' ・・ 時（12時間制）
'MI' ・・ 分（0～59）

デフォルト、シーケンス等

デフォルト値

列には、属性の1つとして、デフォルト値を設定することができる。
デフォルト値は、「DEFAULT」キーワードを使って設定する。
INSERT時に、その列に対する値が設定されなかった場合、デフォルト値が挿入される。

```
create  table  foo (
   a  integer  default  0  not null,
   b  varchar(20)  null
)
```

オラクルの場合、NOT NULL制約より前にDEFAULTを指定する必要があるが、
他の製品では、順番はどちらでもよい。

シーケンス

順番を管理するオブジェクト。PostgreSQL、オラクル、DB2に存在する。
シーケンスを使うには、まずcreate sequence文で、シーケンスを作成しておく必要がある。
構文は、各製品により多少違いがある。
PostgreSQLの場合

```
create sequence シーケンス名
   [ increment 増分値 ]
   [ minvalue 最小値 ]
   [ maxvalue 最大値 ]
   [ start 初期値 ]
   [ cycle ]
```

increment以下の指定は、オプションであり、何も指定しなければ、
最小値1から最大値2,147,483,647まで、1ずつ増加するシーケンスが作成される。
初期値は、startで指定されなければ、最小値になります。(
cycleは、値をサイクリックに使うかどうかの指定。何も指定しなければ、NOCYCLEという状態であり、最大値か最小値に達した後エラーになる。

PostgreSQLでは、シーケンスを使うために、以下の三つの関数が用意されています。
nextval('シーケンス名')
　次のシーケンス値を取り出します。前回この関数を呼んだ時に返った値より、
　増分値だけ大きい値が返却されます。
currval('シーケンス名')
　現在のシーケンス値を取り出します。ただし、この関数を呼ぶ前に同じセッションで
　同じシーケンスに対してnextval('シーケンス名')を呼び出していないと、エラーになります。
setval('シーケンス名',値)
　シーケンスの値を指定した値でリセットします。

自動インクリメント列

PostgreSQLでは、**SERIAL型**により、自動インクリメント列を定義できる。
内部的にシーケンスを作ってくれるが、初期値や増分値などの細かい設定ができない。

CREATE TABLE AS

Oracle、PostgreSQL、MySQLでは、「CREATE TABLE AS」命令によって、
SELECTした結果をそのまま新しいテーブルに保存することができる。
構文:

```
CREATE  TABLE  テーブル名  AS  SELECT文;
```

テーブルの変更

ALTER TABLE文

ALTER TABLE文は、既存のテーブルの構造を変更するSQL文。
以下のようなことができる。

・列の追加、削除
・テーブル制約の追加、削除
・列へのデフォルト値を設定、削除

列の追加と削除

・列の追加
ALTER TABLE テーブル名 ADD [COLUMN] 追加する列の定義

・列の削除
ALTER TABLE テーブル名 DROP [COLUMN] 削除する列名

制約の追加と削除

制約の追加
制約を追加するには、テーブル制約の構文が使用される。

ALTER TABLE テーブル名 ADD [CONSTRAINT 制約名] テーブル制約;
例:
ALTER TABLE テーブル名 ADD CHECK (name <> '');
ALTER TABLE テーブル名 ADD CONSTRAINT 制約名 UNIQUE (列名リスト);
ALTER TABLE テーブル名 ADD FOREIGN KEY (列名リスト) REFERENCES 参照先テーブル名(列名リスト);

制約の削除
ALTER TABLE テーブル名 DROP CONSTRAINT 制約名;

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

NOT-NULL制約の追加
ALTER TABLE テーブル名 ALTER COLUMN 列名 SET NOT NULL;

NOT-NULL制約の削除

デフォルト値の設定と削除

デフォルト値の追加
ALTER TABLE テーブル名 ALTER COLUMN 列名 SET DEFAULT デフォルト値;
デフォルト値の削除
ALTER TABLE テーブル名 ALTER COLUMN 列名 DROP DEFAULT;

データ型と演算子

PostgreSQL の主要なデータ型

データ型	説明
int2	符合付正数(smallint)
int4	符合付正数(integer)
float8	倍精度浮動少数(real)
char(n)	固定長文字列
varchar(n)	可変長文字列
date	日付(年月日)
time	時刻(時分秒)
datetime	日付と時刻

← データ型の後ろに数値がついているものは、それぞれ確保されるバイト数を表している。

char と varchar について
char と varchar は引数として桁数を指定する。
char の場合、固定長の文字列となる。
たとえば、char(10) として10文字の文字を格納できるカラムを定義した場合、7文字しか格納しないと残りの部分にはスペースが挿入される。
それに対し、varchar は文字が少なくても後ろに何も付けられない。
もっとも、PostgreSQL では、文字列には、最大文字数を指定する必要がない「text」を使った方が効率が良いとされている。

日付データ型

PostgreSQLは、SQL標準に準拠した日付データ型を実装している。
DATE(年月日)、TIME(時分秒)、TIMESTAMP(年月日時分秒)、INTERVAL(時間差)。

DATESTYLE

DATESTYLEというパラメータにより、日付の出力形式と年月日の順番を指定できる。
デフォルトでは、**「ISO , MDY」**となっている。
表示する際は、**「show datestyle」**
変更する際は、**「SET DATESTYLE TO ISO , YMD;」**などとする。
→基本的に、年を4桁で表記しておけば、自動的にそれが年と判断してくれる。

タイムゾーン

タイムゾーンは、日本語では「時間帯」の意味だが、協定世界時からの差を表したもの。
日本のタイムゾーンの略語は「JST」であり、大文字で指定する。
日本は、世界協定時から9時間進んでいる。

演算子

算子	説明
+ − * /	四則演算子
%	剰余演算
^	ベキ乗を求める(2^5 = 32)
#	排他的論理和
::	キャスト(型変換)
||	文字列を連結します
~	否定を表す(NOT)
<<	左算術シフト
>>	右算術シフト

正規表現

パターンマッチング

ある条件に一致する文字列を表現する

```
=> select * from table where name like 'a%'
```

name カラムの値が a で始まるものが選択される。
パーセント「%」がワイルドカード（0文字以上の任意の文字）。
１文字だけに限定したい場合は、アンダーバー「_」を指定。

正規表現

PostgreSQL では、正規表現検索が行うことができる。
正規表現を使う直前部分に、チルダ「~」を使う。

```
=> select * from table where name  ~  '^a'
```

^a.*f$
^は、何々で始まる文字
ドット「.」が任意の一文字。
＊が直前文字の繰り返し。

制約とは

制約とは無効なデータがデータベースに入力されてしまうことを防ぐためのルール設定。
制約には「列制約」と「テーブル制約」という 2つの基本制約があります。両者の違いは、
列制約が列のみに適用されるのに対し、テーブル制約が列のグループに適用される。
CREATE TABLE 文では、列定義のデータ型の後ろに列制約を追加し、最後にカンマを付けます。
テーブル制約は、テーブル定義の最後の列定義の後ろに配置し、最後に閉じカッコをつけます。

```
CREATE TABLE テーブル名
  (列名　データ型　列制約....,
   ....................,
   ....................,
   テーブル制約) ;
```

PRIMARY KEY(主キー)

主キーとは、ベーステーブルの各行を一意に識別するための 1つ以上の列のグループのこと。
主キーは、NULL を持たず、一意(UNIQUE)である。

```
/* 列制約としてのPRIMARY KEYの指定 */
 CREATE TABLE 受注表
 (
   受注番号     INTEGER PRIMARY KEY ,
   得意先コード CHAR(5) ,
   商品コード   CHAR(4) ,
   受注個数     INTEGER
```

```
/* テーブル制約でのPRIMARY KEYの指定 */
 CREATE TABLE 価格表
 (
   販売店コード CHAR(4) ,
   商品コード   CHAR(4) ,
   価格         INTEGER ,
   PRIMARY KEY(販売店コード , 商品コード)
```

UNIQUE

すでに他の行の同じ列に存在する値の設定を拒否することができる。
UNIQUE 制約を持つ列には NULL が含まれる可能性がある。

NOT NULL

列に NULL が許可されるのを防ぐことができます。この制約は列に対してのみ使用できます。

CHECK

テーブルに入力するデータを必要に応じてある条件に従ったデータしか入力できないようにする。

```
/* CHECK制約による入力値の制限 */
 CREATE TABLE 受注表
 (
   受注番号     INTEGER PRIMARY KEY ,
   得意先コード CHAR(5) ,
   商品コード   CHAR(4) ,
   受注個数     INTEGER CHECK(受注個数 >=
10)
 ) ;
```

```
/* CHECK制約による入力値の制限 */
 CREATE TABLE 受注表
 (
   受注番号     INTEGER PRIMARY KEY ,
   得意先コード CHAR(5) ,
   商品コード   CHAR(4) ,
   受注個数     INTEGER ,
   CHECK(受注個数 >= 10 OR 商品コード = 0003)
 ) ;
```

制約の名前付けと削除

「 CONSTRAINT 制約名」という書式で制約に名前をつけることができる。
「ALTER TABLE 受注表 DROP CONSTRAINT制約名 ;」で削除する。

外部キー制約（FOREIGN KEY / REFERENCES）

外部キー制約は他のテーブルのプライマリキーによる制約です。
たとえば部門テーブルが既に存在し、社員登録で所属部門コードを入力したら部門テーブルを
チェックし該当がなければエラーを表示して欲しいような時に、
社員テーブルの所属部門コードに外部キーを設定します。

外部キー制約の設定は、列制約とテーブル制約の2パターンで設定可能。

構文：
fooテーブルのb列が、barテーブルのc列を参照する場合。

```
列制約：

CREATE TABLE foo (
    a INTEGER NOT NULL PRIMARY KEY,
    b VARCHAR(20) NOT NULL REFERENCES bar(c)
)

テーブル制約

CREATE TABLE foo (
    a INTEGER NOT NULL PRIMARY KEY,
    b VARCHAR(20) NOT NULL,
    FOREIGN KEY(b) REFERENCES bar(c)
```

外部キーが設定されると逆に設定された方（この例では部門テーブル）のプライマリキーはUPDATEや
DELETEを禁止されます。
しかし外部キーにオプションを設定することによって、プライマリキーの更新を外部キーの更新に
連動させることができます。

```
解説）オプションには下記4つがある。
    NO ACTION     何もしない。デフォルト。
    CASCADE       プライマリキーの値を該当箇所全てにセットする。
    SET NULL      NULLをセットする。
    SET DEFAULT   DEFAULT句で指定してある値に戻す。
```

```
CREATE TABLE bumon_tbl (
  bu_cd   INTEGER PRIMARY KEY,      ←この部門コードに制約される。
  bu_name VERCHAR(20)
);

CREATE TABLE syain_tbl (
  sya_cd   INTEGER PRIMARY KEY,
  sya_name VERCHAR(20) ,
  sya_bucd INTEGER REFERENCES bumon_tbl(bu_cd)   ←外部キー設定
                ON UPDATE CASCADE       ←オプション（更新連動）
                ON DELETE SET NULL      ←オプション（削除はNULLセット）
```

ビュー

ビューとは、仮想的なテーブルのことです。
ビューを作成するには **CREATE VIEW 文**を用います。

構文:

```
CREATE VIEW ビュー名 [(列名, 列名, .....)] AS SELECT文;
```

→ビューの列名を明示的に指定することもできる。

重複行の扱い

SELECT文で取得した選択列の重複する行を省きたい場合、**「DISTINCT」**キーワードを使用します。

```
SELECT DISTINCT 列名 FROM テーブル名;
```

出力の並べ替え

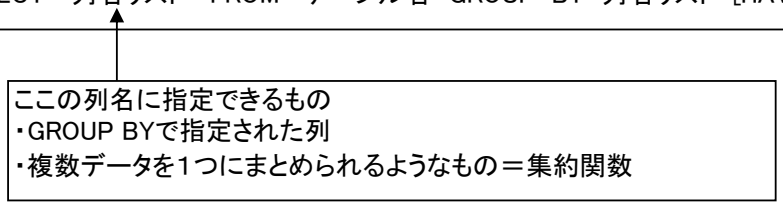
SQL では ORDER BY 句を使って出力に順番を適用できるようにしています。
列ごとに昇順 (ASC) 降順 (DESC) を指定することができます。
各列ごとに、列名の後ろに指定する形。デフォルトでは昇順になっているので、普通ASCはいらない。

```
SELECT * FROM テーブル名 ORDER BY 列名 DESC ;
```

グループ化

同じ列の値の中で、同じ値を持つデータごとにグループとしてまとめることをグループ化と言います。
グループ化には **GROUP BY 句**を用います。
グループ化された情報に対して条件を設定し、その条件に合致するものだけを抽出する場合は、
HAVING 句を続けて抽出条件を記述します。

```
SELECT 列名リスト FROM テーブル名 GROUP BY 列名リスト [HAVING 条件句];
```



集約関数とは、1つの列グループに対して施すことのできる演算機能で1つの結果を返します。
集約関数は以下の5つ

```
SUM( ) ・・ 指定条件によって得られた列の値の合計を求める関数
AVG( ) ・・ 指定条件によって得られた列の値の平均値を求める関数
MAX( ) ・・ 指定条件によって得られた列の値の中で最大値を返す関数
MIN( ) ・・ 指定条件によって得られた列の値の中で最小値を返す関数
COUNT( ) ・・ 指定条件によって得られた表の基数、すなわち行数を求める関数
```

副問い合わせ

あるSELECT文での問い合わせの結果を基に、さらにその外側のSQL文が本来の条件処理を行う場合には、入れ子状にしたSQLを構築することができます。
これをサブクエリー（副問い合わせ）と言います。

```
例：
SELECT　列名　FROM　テーブル名
　WHERE　列名 =
　（SELECT　列名　FROM　テーブル名　WHERE　列名　=　値 ）;
```

副問い合わせの結果が、1行1列の場合、1行複数列の場合、複数行複数列の3パターンがあるので、その際にWHERE句の条件部分に使用できる演算子も、それぞれ異なる。

結果が1行1列の場合は、等号や不等号といった関係演算子を使い、
1行複数列の場合は、行値コンストラクタが使える。
さらに、複数行複数列の場合は、IN演算子やEXISTS演算子, ANY演算子, ALL演算子などが使える。

IN演算子

IN演算子を使うと、カッコ内のいずれかの値と、INの前の値が一致する場合に、真を返す。
カッコの中には、列を並べることもできるし、サブクエリーを記述することもできる。

```
式　IN（式 [,式...]）
式　IN（サブクエリー）
```

EXISTS演算子

カッコ内に指定されたサブクエリーの中で、1つ以上の行がある場合に真を返す。

```
EXIST（サブクエリー）
```

「NOT EXIST」とすると、サブクエリーの中に行が全くない場合が真となる。

ANY演算子

ANY演算子は、比較演算子とサブクエリーから成り立つ。
サブクエリーで返された結果の中のいずれかの値と比較したい場合に使用する。

```
式　比較演算子　ANY（サブクエリー）
```

ALL演算子

ALL演算子は、比較演算子とサブクエリーから成り立つ。
サブクエリーで返された結果の中の全ての値と比較したい場合に使用する。

```
式　比較演算子　ALL（サブクエリー）
```

CASE演算子

式を評価して、任意の値に変換する。
CASE文の書式は2種類ある。
たとえば、性別という列の値が'男'だったら'male'、'女'だったら'female'としたい場合

```
--単純CASE式
CASE 性別
WHEN '男' THEN 'male'
WHEN '女' THEN 'female'
ELSE  'その他'
END

--検索CASE式
CASE WHEN 性別 = '男' THEN 'male'
       WHEN 性別 = '女' THEN 'female'
ELSE  'その他'
```

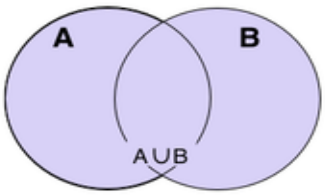
クエリの結合

クエリの結合

クエリの結合とは、2つのクエリの出力結果を和、差、積、の 3 種類の方法で結合することです。
和結合は、「UNION 演算子」、積結合は「INTERSECT 演算子」、差結合は「EXCEPT 演算子」を用います。

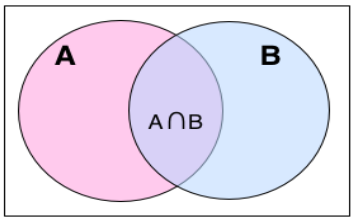
UNION 演算子

UNION 演算子は、2つのクエリの出力を和結合します。
すなわち、2つのクエリの結果を結合し、そこから重複する行を削除します。



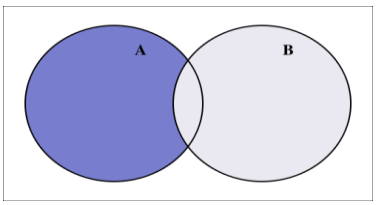
INTERSECT 演算子

INTERSECT 演算子は、2つのクエリの出力を積結合します。
すなわち、2つのクエリの結果から同じものだけを出力するものです。



EXCEPT 演算子

EXCEPT 演算子は、2つのクエリの出力の差結合します。



クエリの結合

JOIN

SELECT　表名1.列名 ,　表名2.列名
FROM　表名1

という指定の後ろに以下のJOIN句をつけた場合の意味

CROSS JOIN 表名2　・・・2つの表から直積演算を戻します

NATURAL JOIN 表名2　・・・2つの表の中の同じ列名に基づいて結合します

JOIN　表名2　USING (列名)　・・・列名に基づいて等価結合をします

JOIN　表名2　ON (表名1.列名 = 表名2.列名)
　・・・ON句の条件に基づいて結合します

LEFT | RIGHT | FULL OUTER JOIN 表名2 ON (表名1.列名 = 表名2.列名)
　・・・外部結合をします

外部結合

SQL:1999では「内部結合」とは一致する行のみを戻す2つの表の結合のことを言います。
「外部結合」とは内部結合の結果とともに一致しない行も戻す結合のことを言います。

LEFT OUTER JOIN ――― 左側外部結合
RIGHT OUTER JOIN ――― 右側外部結合
FULL OUTER JOIN ――― 完全外部結合

ユーザー定義関数

ストアドプロシージャとは

ストアドプロシージャは、データベースに対する一連の処理を一つのプログラムにまとめ（PROCEDURE）、データベース管理システムに保存（STORE）したものです。
複雑な SQL 文の呼び出しを、論理的に一つの処理単位にまとめて、
簡単にその名前で呼び出せるようになっています。
一つのプロシージャには、複数の SQL 文が含まれていたり、繰り返しや条件分岐などの制御構造をもつこともあります。
引数をとって処理をしたり、処理結果を返すこともできます。

ストアドプロシージャを利用することにより、次のようなメリットがある

・RDBMS に一つずつ SQL 文を発行する必要がなくなる
・ネットワーク上のトラフィックを削減できる
・あらかじめ処理内容が RDBMS に登録され、構文解析や機械語への変換が済んでいるため、処理時間が軽減される

ORACLE では PL/SQL という拡張 SQL を用いてストアドプロシージャを記述します。
一方、PostgreSQL ではユーザ定義関数を利用することでストアドプロシージャに近い機能を実現。

SQL関数

SQL 関数は、PostgreSQL で実装されている SQL のみで表現される関数のこと。
SQLだけで実現できる簡単な処理をまとめたい場合に使える。

```
CREATE [OR REPLACE] FUNCTION 関数名 ( [ 引数リスト] )
RETURNS [ SETOF ] 戻り値のデータ型
AS
'
～実際の具体的な処理～
'
LANGUAGE sql ;
```

PL/pgSQL関数

PL/pgSQL 言語で記述される関数で、SQL 関数よりもより複雑な動作を定義することができる。
繰り返しや条件分岐などの制御構造を定義できる。

```
CREATE [OR REPLACE] FUNCTION 関数名 ( [ 引数リスト] )
RETURNS [ SETOF ] 戻り値のデータ型
AS
'
[ラベル]
[ DECLARE 変数名 データ型 … ]
BEGIN
    処理内容
END ;
'
```

変数の宣言や実際のプログラム部分にあたる処理を書く。

関数の実行

SELECT 関数名([引数,...]) ; で実行する。

関数の削除

DROP FUNCTION 関数名([引数のデータ型]) ;

PL/pgSQL関数

・関数が複数行を返す場合は、SETOFキーワードを付ける。
さらに、返す複数行をためこむために、RETURN NEXTを使い、最後にRETURNで返す。

・変数の宣言は、DECLAREブロック内に記述する。
宣言の仕方は、「変数名　型　:=　値」。

IF文

構文:

```
IF  条件式1  THEN
  ～条件式1がtrueだった場合の処理～
ELSIF  条件式2  THEN
  ～条件式2がtrueだった場合の処理～
ELSE
  ～どの条件にもヒットしなかった場合の処理～
END IF
```

LOOP文

構文:

```
LOOP

  ～具体的な処理～
  EXIT  WHEN   条件式;

END LOOP;
```

EXIT WHENを使って、このループを抜けるための条件を指定しておかないと、
ループが無限ループとなってしまう点に注意！

WHILE文

構文:

```
WHILE  条件式  LOOP
  ～条件式がtrueの間だけ行う具体的な処理～
END  LOOP;
```

FOR文

構文:

```
FOR  カウンタ変数  in  初期値. . 終了値  LOOP
  ～具体的な処理～
END  LOOP;
```

カウンタ変数は、ループが1回まわるたびに1カウントアップされる。
また、この変数だけは特別にDECLAREでの宣言の必要がない。

CREATE TYPE文

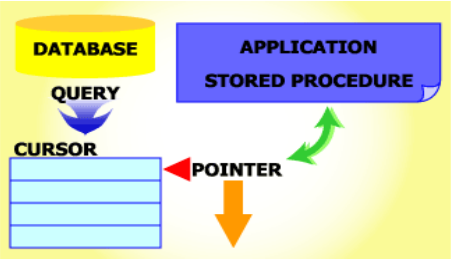
現在のデータベースで使用できる新しいデータ型を登録する。
概念的な複数列の集まりなので、テーブルのようなものと思ったほうがよい。

構文:

```
CREATE  TYPE  型の名前
  AS  （ 属性名   データ型 [, ... ] );
```

カーソル

SQL は、データベースから複数行のデータを取り出し、一括して処理することに向いています。
他のプログラムや SQL 非標準のストアドプロシージャにおけるループでは、
一行ごとにデータを処理したい場合があります。
カーソルとは、クエリの結果集合を一時的に蓄えておくための仮想的な作業領域のことです。
この中の現在のレコードの位置を示すポインタと呼ばれるものがあり、
ループ処理が実行されるたびに一行ずつ進んでいき、次の処理対象を行います。



カーソルを使用するときの手順

①カーソルの宣言
②カーソルのオープン
③FETCH処理（一行ごとにデータを取り出す）
④カーソルを閉じる

構文： 以下は束縛カーソルの場合

①カーソルの宣言

```
DECLARE  カーソル名  CURSOR   FOR  SELECT文;
```

②カーソルを開く

```
OPEN  カーソル名;
```

③FETCH処理
FETCH 文が実行されると、行の各データを変数リストへ格納します。

```
FETCH  カーソル名  INTO  変数リスト ;
```

④カーソルを閉じる

```
CLOSE カーソル名 ;
```

トリガー

トリガーは、表に対して何らかの変更処理が加えられたときに、その変更処理をきっかけとして
自動的に実行される特殊なストアドプロシージャのこと。
トリガーを定義するときには、その対象となるテーブル、トリガーが起動するきっかけとなる
表に対する変更処理、トリガーの処理内容、トリガーの起動するタイミングなどを指定します。

トリガーは指定したテーブルを監視し、指定した変更処理がテーブルに対して行われると、
指定したタイミングで指定した処理を実行します。

構文:

```
<<PostgreSQL>>

CREATE   TRIGGER  トリガー名
      { BEFORE | AFTER }
      { INSERT | UPDATE [OF 列名,...] | DELETE }
        [ OR { INSERT | UPDATE [OF 列名,...] | DELETE}]
        [ ... ]
      ON テーブル名
      FOR EACH { ROW | STATEMENT }
      [ WHEN 条件式 ]
```

PostgreSQL では BEFORE と AFTER のみサポートしています。
PostgreSQL では、複数の行に対するデータ操作文が発行されるときに行ごとに起動するのか、
一度だけ起動するのかを指定するために、どちらの場合も明示的に指定します。
一行ごとに起動するときは FOR EACH ROW、一度だけしか起動しない場合は
FOR EACH STATEMENTとします。

トリガーの処理内容

PostgreSQL の場合は予め定義しておいた関数名を EXECUTE PROCEDURE に続けて記述します。
引数が必要な場合は引数も記述します。このとき利用できる関数は、PL/pgSQL 関数か
C 関数でなければなりません。