



OSS-DB Exam Silver 技術解説セミナー

2012/11/10

特定非営利活動法人エルピーアイジャパン
テクノロジー・マネージャー
松田 神一



- OSS-DB技術者認定試験の概要
- PostgreSQLのインストール
- ポイント解説: 運用管理
- ポイント解説: SQL
- OSS-DB Exam Silverの例題



- Linux Professional Institute Japan (本部はカナダ)
- Linux/OSS技術者の技術力の認定制度の運用を通じて、日本のLinux/OSS技術者の育成、Linux/OSSビジネスの促進に寄与する活動を展開するNPO法人

- 2000年から、Linux技術者認定試験LPICを実施
- 2011年7月から、オープンソースデータベース技術者認定試験OSS-DBを実施



■ 松田 神一(まつだ しんいち)

LPI-JAPAN テクノロジー・マネージャー

■ NEC、オラクル、トレンドマイクロなどで約20年間、ソフトウェア開発に従事（専門はアプリケーション開発）

うち10年間はデータベース、およびデータベースアプリケーションの開発
(Oracle、C言語、SQL言語)

■ 2010年7月から現職



今日のゴール

OSS-DB

■ OSS-DB (PostgreSQL) の特徴の理解

- ・主な機能
- ・他のRDBMSとの違い

■ OSS-DB技術者認定試験についてのポイントの理解

- ・PostgreSQLの設定、運用管理
- ・SQLによるデータ操作

■ 受験準備のために何をすべきかの理解

- ・実機で試せる環境の準備
- ・出題範囲、試験の目的、合格基準



OSS-DB技術者 認定試験の概要



■ 使う前に設定が必要（インストールしただけでは利用できない）

- ・ユーザ
- ・アクセス権
- ・テーブルの作成
- ・プログラムの開発

■ 重要な用途

- ・基幹業務での利用
- ・バックアップ
- ・セキュリティ

■ 複雑な用途

- ・分散DB
- ・パフォーマンスチューニング
- ・トラブルシューティング

■ 製品による違い

- ・一般論だけ学んでも、現場で活躍できない



■認定の種類

- Silver (ベーシックレベル)
 - OSS-DB Exam Silverに合格すれば認定される
- Gold (アドバンストレベル)
 - OSS-DB Silverの認定を取得し、OSS-DB Exam Goldに合格すれば認定される

■Silver認定の基準

- データベースの導入、DBアプリケーションの開発、DBの運用管理ができること
- OSS-DBの各種機能やコマンドの目的、使い方を正しく理解していること

■Gold認定の基準

- トラブルシューティング、パフォーマンスチューニングなどOSS-DBに関する高度な技術を有すること
- コマンドの出力結果などから、必要な情報を読み取る知識やスキルがあること



■一般知識 (20%)

- OSS-DBの一般的特徴
- ライセンス
- コミュニティと情報収集
- RDBMSに関する一般的知識

■運用管理 (50%)

- インストール方法
- 標準付属ツールの使い方
- 設定ファイル
- バックアップ方法
- 基本的な運用管理作業

■開発 / SQL (30%)

- SQLコマンド
- 組み込み関数
- トランザクションの概念



■最新の出題範囲は

<http://www.oss-db.jp/outline/examarea.shtml>
で確認できる

■前提とするRDBMSはPostgreSQL 9.0

■SilverではOSに依存する問題は出題しないが、記号や用語がOS によって異なるものについては、Linuxのものを採用している

- OSのコマンドプロンプトには \$ を使う
- 「フォルダ」ではなく「ディレクトリ」と呼ぶ
- ディレクトリの区切り文字には / を使う

■出題範囲に関するFAQ

<http://www.oss-db.jp/faq/#n02>



■ Silverの合格基準は、各機能やコマンドについて

- ・ その目的を正しく理解していること
 - XXXコマンドを使うと何が起きるか
 - YYYをするためにはどのコマンドを使えば良いか
- ・ 利用法を正しく理解していること
 - コマンドのオプションやパラメータ
 - 設定ファイルの記述方法

■ 基本的な出題形式は

- ・ 最も適切なものを1つ(2つ)選びなさい
- ・ 誤っているものを1つ(2つ)選びなさい

■ 出題範囲にあるすべての項目について、試験問題が用意されている

■ 出題範囲詳細に載っている項目すべてについて、マニュアルなどで調査した上で、実際に試して理解する

- ・ 実機で試すことは極めて重要



OSS-DBと 商用DBの違い



■ 主な商用RDBMS: Oracle, DB2 (IBM), SQL Server (MS)

■ 主なOSS RDBMS: PostgreSQL, MySQL, Firebird

■ 共通点

- RDBMSとしての各種機能

- データ管理／入出力
- ユーザ管理、アクセス権限管理、セキュリティ
- バックアップ、リカバリ
- レプリケーション

- SQL言語 (ANSI/ISOで標準化)

■ 違い

- 各種機能の使い方

- コマンドとオプション
- 設定ファイルとパラメータ

- SQLの方言

- 独自拡張機能



■ OSS-DB vs 商用DB

- ・製品化はOracleが1977年、DB2が1982年、PostgreSQL/MySQLはいずれも1995年
- ・OSS-DBの継続的機能強化により、機能・性能とも、商用DBに比べて遜色ないレベル

■ PostgreSQL vs MySQL

- ・機能的な差は小さい
- ・PostgreSQLはレプリケーションのサポートが遅かった（2010年、MySQLは2003年、Oracleは1992年）
- ・MySQLはストアドプロシージャのサポートが遅かった（2005年、PostgreSQLは1998年、Oracleは1988年）
- ・PostgreSQLはBSDに基づくライセンス、MySQLはGPLと商用のデュアルライセンス



■ 商用DBの方が機能的に上？

- OSS-DBの継続的機能強化により、RDBMSとして必要な機能は揃っている
- 商用DBにしかない機能もあるが、逆にOSS-DBにしかない機能もある
- 個別機能の有無 (○×比較) ではなく、必要な機能の有無を検討する
- 同じ機能がなくても、代替策で同等の機能を実現できることが多い

■ 商用DBの方が性能的に上？

- マルチコアCPUの対応、性能向上のための機能強化も進んでおり、性能的にも商用DBと比べても遜色ない
 - PostgreSQL 9.2は64コアまでスケールする
- ライセンス料金が安い分、ハードウェアに投資すれば、性能的には商用DBを凌駕する
 - SSDやioDriveなどの高速ストレージの利用

■ OSS-DBを使うには特別なスキルが必要？

- 製品の利用にあたって学習が必要なのは商用でもOSSでも同じ
- PostgreSQLはOracleと類似点が多いので、Oracle技術者がPostgreSQLを習得するのは比較的容易



■ OSS-DBはサポートがない？

- ・ 商用DBと同じようなサポートサービスがある
- ・ 24x365の商用サポートもある
- ・ コミュニティのサポートが終了しても、商用サポート（パッチ提供を含む）を継続するサービスがある

■ OSS-DBは品質的に不安がある？

- ・ OSSか商用かということと品質は無関係
 - OSSだからこそ品質が高くなる（Linux、PostgreSQLなど）
 - ユーザ数が多く、開発が活発であれば、品質が高くなる

■ OSS-DBではパッチが提供されない？

- ・ 商用DBこそパッチが提供されない（アップグレードを強要される）
- ・ 商用DBでは、高額のサポート料金を払わなければセキュリティパッチさえ提供されない

■ 開発やサポートが継続されない（開発コミュニティの解散）リスクがある？

- ・ 開発元の倒産や買収で製品がなくなるリスクは商用も同じ
- ・ 開発およびユーザコミュニティがしっかりしている製品を選ぶことが肝要



- どの製品にも共通の機能もあれば、同じ機能でも製品によって実行方法の異なるもの、特定の製品にしかない機能もある
- まずはDBの種類による差分はあまり気にせずに、特定のDBについて学習し、マスターする
- 次のステップは…
- 横展開
他のDBについて、最初に学習したDBとの差分に注意しながら学習する
- 深掘り
その製品のエキスパートとなるべく、更に深く学ぶ



PostgreSQLの インストール



■ インストールに必要な環境

- ・インターネットにつながっているマシン (Windows/Mac/Linux)
- ・インストーラの入ったメディアがあれば、オフラインのPCでもインストール可能

■ おススメの環境

- ・ある程度、Linuxの知識がある方にはLinuxを使うことを勧める
- ・VirtualBox あるいは VMware Player (いずれも無料) を使えば、Windows PC上に仮想Linux環境を構築し、そこにPostgreSQLをインストールして学習することができる
- ・仮想環境の良い点は、それを破壊しても、簡単に最初からやり直せるところ
- ・もちろん、WindowsやMacの環境に直接、PostgreSQLをインストールするのでもOK

■ 参考書などを読むだけでは、十分な学習をすることはできません。
自分専用の環境を作り、そこでいろいろ試すことで学習してください。



■ インストール方法

- ・ソースコードから自分でビルドしてインストール
- ・ビルド済みのパッケージをインストール（様々なビルド済みパッケージがある）

■ ダウンロードサイト（ソースコードや各種パッケージへのリンクがある）

- ・<http://www.postgresql.org/download/>

■ インストール後の初期設定

- ・データベースのスーパーユーザ（postgresユーザ）の作成
- ・環境変数（PATH, PGDATAなど）の設定
- ・データベースの初期化（データベースクラスタの作成）
- ・データベース（サーバープロセス）の起動
- ・データベース（サーバープロセス）起動の自動化

■ インストール方法によっては、初期設定の一部が自動的に実行される

■ インストール方法によって、プログラムがインストールされる場所、データベースファイルが作られる場所が大きく異なるので注意



■ Windows/Mac/Linuxいずれでも利用可能

- EnterpriseDB社のサイトから、ビルド済みのパッケージをダウンロードしてインストールする
<http://www.enterprisedb.com/products-services-training/pgdownload>
- GUIの管理ツール (pgAdmin III) も同時にインストールされる
- ApacheやPHPなど、PostgreSQLと一緒に使われるソフトウェアも、同時にインストール可能
- Windowsではワンクリックインストールの利用を推奨

■ インストールガイド(英語)は

<http://www.enterprisedb.com/resources-community/pginst-guide>

■ 多くの項目はデフォルト値のままで良い

- スーパーユーザ (postgres) のパスワードの設定を求められるので、適切に設定し、それを忘れないようにすること
- ロケール (Locale) の設定を求められるが、"Default locale"となっているのを"C"に変更することを推奨する
- インストール終了時にスタックビルダ (Stack Builder) を起動するかどうか尋ねられるが、ここはチェックボックスを外して終了してよい。必要なら後でスタックビルダを起動することができる



- **postgres ユーザは自動的に作成される**
- **データベースの初期化、起動はインストール時に実行されるので、インストール後、すぐにデータベースに接続できる**
- **データベースの自動起動の設定ががあるので、マシンを再起動したときもデータベースが自動的に起動する**
- **Windowsでは C:\Program Files\PostgreSQL\9.0 の下にインストールされる。**
データベースは C:\Program Files\PostgreSQL\9.0\data の下に作られる。環境変数 PATH に C:\Program Files\PostgreSQL\9.0\bin を追加するか、あるいは C:\Program Files\PostgreSQL\9.0 の下の pg_env.bat を実行する
- **Linuxでは /opt/PostgreSQL/9.0 の下にインストールされる。データベースは /opt/PostgreSQL/9.0/data の下に作られる。環境変数 PATH に /opt/PostgreSQL/9.0/bin を追加するか、あるいは /opt/PostgreSQL/9.0 の下の pg_env.sh を読み込む ("." pg_env.sh" を実行する)**



- CentOSなどでは、`yum` コマンドでインストールするのが基本だが、CentOS 6.xで
`# yum install postgresql-server`
とすると、PostgreSQL 8.4がインストールされるので注意
- PostgreSQL 9.0 (あるいは他のバージョン) を `yum` コマンドでインストールする場合について
<http://www.postgresql.org/download/linux/redhat/>
に手順の説明 (英語) がある
- リポジトリを `rpm` でインストール、パッケージを `yum` でインストール、という手順でインストールする
- 上記ページの“repository RPM listing”のリンクをクリック
<http://yum.postgresql.org/repopackages.php>
に表示されているリストから、インストールするPostgreSQLのバージョン、Linuxディストリビューションのバージョンに合ったリンクをクリック。
PostgreSQL 9.0をCentOS 5.x (32bit版) にインストールする場合は
<http://yum.postgresql.org/9.0/redhat/rhel-5-i386/pgdg-centos90-9.0-5.noarch.rpm> をダウソロード
`# rpm -ivh pgdg-centos-9.0-5.noarch.rpm`
としてリポジトリをインストールする



■リポジトリのインストールが終わったら、

```
# yum install postgresql90-server  
とすればパッケージがインストールされる
```

■ディストリビューションの種類 (RedHat/CentOS/Fedora/SL) とバージョン (5.x/6.x)、マシンアーキテクチャ (32bit/64bit)、PostgreSQLのバージョン (9.0/9.1/9.2) によって、ダウンロードするrpmファイルが異なるが手順やインストール後の環境は基本的に同じ。

■yum コマンドを使わず、パッケージだけダウンロードして、rpm コマンドでインストールしても良い。

必要なパッケージは、postgresql90 (クライアント)、postgresql90-libs (ライブラリ)、postgresql90-server (サーバ) の3つ。

ライブラリ、クライアント、サーバの順で、rpmコマンドでインストールする。

パッケージは次のサイトからダウンロードできる

<http://yum.postgresql.org/packages.php>



- **postgres ユーザは自動的に作成される**
- **プログラムは /usr/pgsql-9.0 の下にインストールされる。データベースは /var/lib/pgsql/9.0/data の下に作成される**
- **主なコマンドは /usr/bin の下にシンボリックリンクが作られるが、pg_ctl や initdb など一部のコマンドについてはリンクが作成されないので、PATH を設定するか、絶対パスで起動する必要がある。**
- **インストールしただけでは、データベースの初期化、起動、自動起動の設定などはされない。rootユーザで以下を実行する**
 - # service postgresql-9.0 initdb (データベース初期化)
 - # service postgresql-9.0 start (データベース起動)
 - # chkconfig postgresql-9.0 on (データベース自動起動の設定)
- **参考:RPMで複数バージョンのPostgreSQLをインストール**
 - http://lets.postgresql.jp/documents/tutorial/new_rpm



■ Fedoraへのインストール方法

方法自体はCentOSと同じだが、
yum install postgresql-server
とすると、PostgreSQL 9.1がインストールされる

■ postgres ユーザは自動的に作成される

■ データベースは /var/lib/pgsql/data の下に作成される

■ インストールしただけでは、データベースの初期化、起動、自動起動の設定などはされない。rootユーザで（あるいはsudoコマンドを使って）以下を実行する

- # postgresql-setup initdb (データベース初期化)
- # systemctl start postgresql.service (データベース起動)
- # systemctl enable postgresql.service (データベース自動起動の設定)



- Ubuntuでは標準的なapt-getで最新版(バージョン9.1)がインストールされる

```
$ sudo apt-get install postgresql
```

- プログラムは /usr/lib/postgresql/9.1 の下にインストールされる

- 設定ファイルは /etc/postgresql/9.1/main の下、データベースは /var/lib/postgresql/9.1/main の下に作成される

- postgres ユーザは自動的に作成され、データベースの作成、起動、自動起動の設定も自動的に行われるので、すぐに利用可能

- 主なコマンドは /usr/bin の下にシンボリックリンクが作られるので環境変数の設定は不要。ただし、pg_ctl や initdb など一部のコマンドについてはリンクが作成されない

- 環境がやや特殊。pg_ctl コマンドなど一部の機能の学習には不適

- (参考)

http://www.oss-db.jp/measures/dojo_info_01.shtml



- Linuxでは、コンパイラなどの開発環境が標準で用意されており（インストールされていなくても簡単にセットアップ可能）、ソースコードから自分でビルドしてインストールするのも難しくない
- ソースコードはPostgreSQLの公式サイトからダウンロード
<http://www.postgresql.org/ftp/source/>
- ビルド、およびインストールの手順は、オンラインマニュアル
<http://www.postgresql.jp/document/9.0/html/>
の15章(Linux)、16章(Windows)に解説されている
- 基本的には、

```
$ ./configure  
$ make          (あるいは $ make world)  
# make install  (あるいは # make install-world)
```

を実行するだけ
- 多くの環境では `configure` の実行でいくつかエラーが出るが、これを自力で解決できる人には、ソースからのインストールを勧める
- 市販書籍では、ソースからビルドを前提に解説された記述が多い



- `make install` は、プログラムを `/usr/local/pgsql` の下にコピーするだけなので、その後の初期設定をすべて実行する必要がある
- 初期設定の手順はオンラインマニュアルの17章に解説がある

■ `postgres` ユーザの作成

```
# useradd postgres
```

■ 環境変数の設定 (`~postgres/.bash_profile`、およびPostgreSQLを利用するユーザの `~/.bash_profile` に追記)

```
export PATH=$PATH:/usr/local/pgsql/bin  
export PGDATA=/usr/local/pgsql/data  
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/pgsql/lib  
export MANPATH=$MANPATH:/usr/local/pgsql/share/man
```

■ データベース用ディレクトリの作成 (データベース初期化の準備)

```
# mkdir /usr/local/pgsql/data  
# chown postgres /usr/local/pgsql/data  
# chmod 700 /usr/local/pgsql/data
```



■ データベースの初期化と起動 (postgres ユーザで実行)

```
$ initdb -E UTF8 --no-locale  
$ pg_ctl start
```

■ 自動起動の設定 (RedHat系)

```
contrib/start-scripts/linux を  
/etc/rc.d/init.d/postgresql-9.0 にコピー  
# chmod +x /etc/rc.d/init.d/postgresql-9.0  
# chkconfig --add postgresql-9.0  
# chkconfig postgresql-9.0 on
```

■ 自動起動の設定 (Debian系)

```
contrib/start-scripts/linux を  
/etc/init.d/postgresql-9.0 にコピー  
$ sudo chmod +x /etc/init.d/postgresql-9.0  
$ sudo update-rc.d postgresql-9.0 defaults 98 02
```



インストールに関する注意事項

OSS-DB

- インストール方法によっては、`initdb`, `pg_ctl` など（試験範囲に含まれる）一部のコマンドへの PATH が通っていないので、PATH 変数を変更する、あるいは `/usr/local/bin` にリンクを張る、などの必要がある
 - ・ 実運用の環境では回避策がある（これらのコマンドを使わなくても良い）が、試験対策としてはこれらのコマンドの使用法を理解する必要がある
- PostgreSQL の実行ファイル、ライブラリなどが置かれる場所、データベースファイルが作成される場所がどこか、インストール後に確認しておくこと
 - ・ インストール方法によって大きく異なるので注意
- `yum`, `rpm`, `apt-get`, `dpkg` 等、OS やパッケージに依存したインストールコマンドや手順は出題しない
- ネットワーク経由で PostgreSQL を使うとき、PostgreSQL 本体の設定だけでなく、OS のファイアウォールなどの設定も変更が必要なことが多いことに注意。
 - ・ 例えば CentOS 6.x では、PostgreSQL が使うポート 5432 はファイアウォールでブロックされ、また SELinux が Enforcing になっている



ポイント解説：運用管理



■ 必要な人に、適切なDBサービスを提供すること（セキュリティ管理）

- ・ 必要ない人にはサービスを提供しない
- ・ 不正なアクセスを拒絶する
- ・ 設定と監視

■ サービスレベルの維持

- ・ 定められた水準のサービスを提供し続けること
 - サービスを提供する時間
 - パフォーマンスの維持

■ トラブルシューティング（予防と対処）

- ・ DBに接続できない
- ・ DBが遅い
- ・ DBが起動しない
- ・ ディスク、ファイル、データの破損
- ・ バックアップ、リストア、リカバリ



- 運用管理に必要とされる機能、実現されている機能はほぼ同じだが、使用するコマンド、パラメータ、設定ファイルなどは全く異なる
- それぞれのRDBMSについて基本からマスターする
- データベース構造の違いに注意する
- 同じ用語を使っていても、その意味がRDBMSの種類によって異なることや、同じ機能をRDBMSの種類によって別の名称で呼んでいることもあるので注意が必要



■ データベースインスタンス

- ・データベースを構成するプロセス、共有メモリ、ファイルを合わせたものをインスタンスと呼ぶ

■ プロセスの構成

- ・PostgreSQLのサーバプロセスはマルチプロセス構成で、データアクセス、ログ出力などのために、それぞれ別のプロセスが起動している

■ ファイルの構成

- ・データベースファイルについては、その置き場所となるディレクトリを指定すると、PostgreSQLサーバがその下にファイルを作成する



■ PostgreSQLのサーバは

- ・マルチプロセス構成
- ・全体を管理するpostmasterプロセス①
- ・目的別に複数のpostgresプロセス②
- ・クライアント1つに対して1つのpostgresプロセス③

```
[matsuda@fedora ~]$ ps aux | grep postgres
postgres 2039 0.0 0.1 150676 1700 ? S 15:36 0:00
/usr/local/pgsql/bin/postmaster -D /usr/local/pgsql/data
postgres 2071 0.0 0.0 150676 948 ? Ss 15:37 0:00 postgres: writer process
postgres 2072 0.0 0.0 150676 672 ? Ss 15:37 0:00 postgres: wal writer
process
postgres 2073 0.0 0.1 151512 1400 ? Ss 15:37 0:00 postgres: autovacuum
launcher process
postgres 2074 0.0 0.0 119160 804 ? Ss 15:37 0:00 postgres: stats collector
process
postgres 12086 0.0 0.2 151792 2772 ? Ss 17:04 0:00 postgres: matsuda matsuda
[local] idle
postgres 12216 0.0 0.2 151792 2748 ? Ss 17:11 0:00 postgres: matsuda matsuda
[local] idle
```



■ PostgreSQLのデータベースファイルは

- ・データベースクラスタの `base` ディレクトリ配下に格納
- ・オブジェクト(テーブルなど)1つにつき、1個のファイル
- ・ファイル名とオブジェクト名の対応は `oid2name` コマンドで確認できる

```
[postgres@fedora data]$ ls -F
PG_VERSION pg_hba.conf pg_serial/ pg_twophase/ postmaster.pid
base/      pg_ident.conf pg_stat_tmp/ pg_xlog/    serverlog
global/     pg_multixact/ pg_subtrans/ postgresql.conf
pg_clog/    pg_notify/   pg_tblspc/  postmaster.opts
[postgres@fedora data]$ ls -F base
1/ 12794/ 12802/ 16385/
[postgres@fedora data]$ oid2name
All databases:
  Oid Database Name Tablespace
-----  
16385  matsuda pg_default
  12802  postgres pg_default
  12794  template0 pg_default
    1    template1 pg_default
[postgres@fedora data]$ ls -F base/16385
12539  12586  12623  12665_fsm  12738  12784_vm
12539_fsm 12587  12624  12665_vm   12740  12786
...
[postgres@fedora ~]$ oid2name -d matsuda -f 12539
From database "matsuda":
  Filenode Table Name
-----  
12539  pg_statistic
```

各ディレクトリが1つのデータベースに対応

各ファイルが1つのテーブルに対応



■ データベースクラスタ

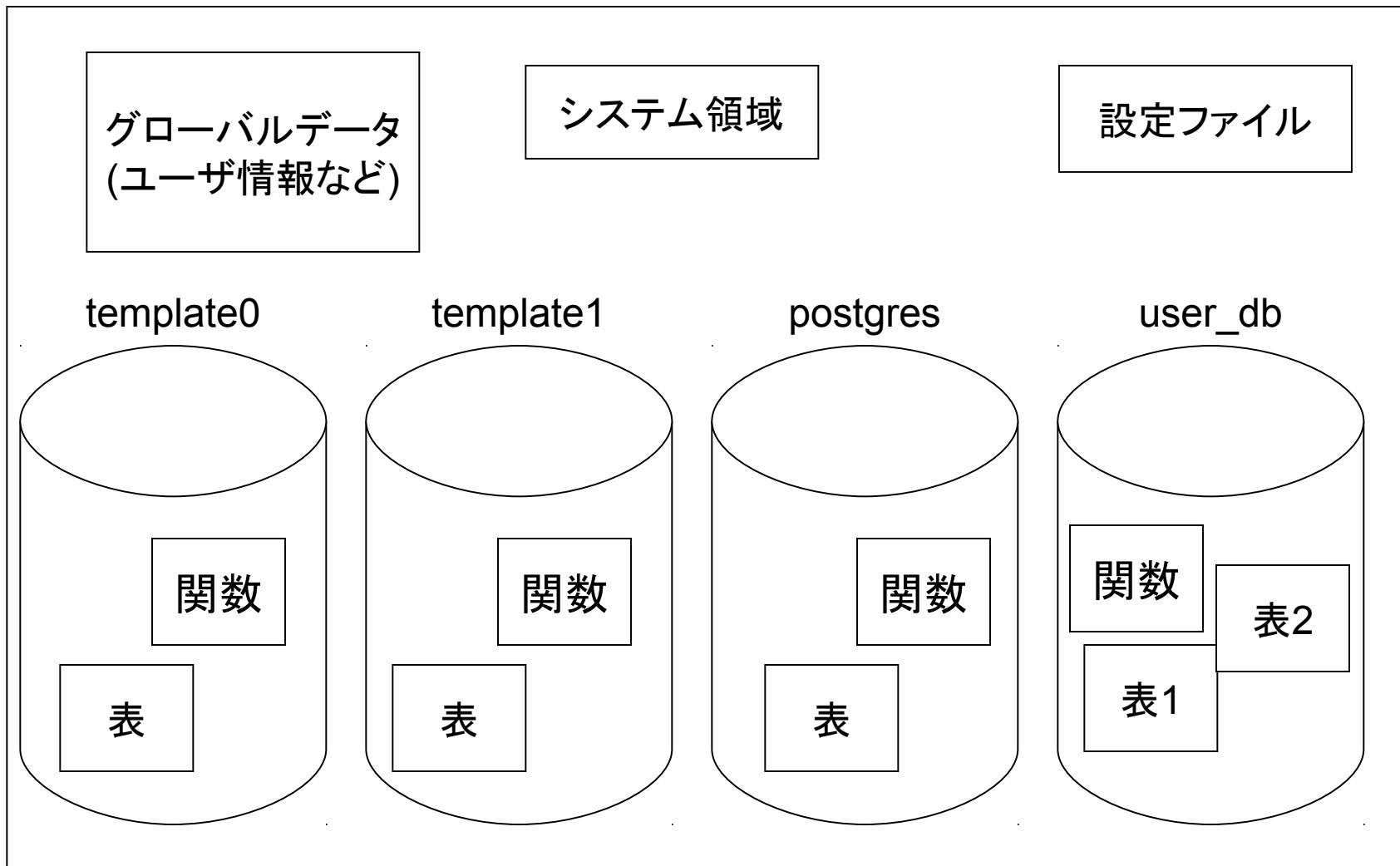
- ・初期化された直後のPostgreSQLのインスタンスには、`template0`, `template1` という2つのテンプレートデータベースと、`postgres` というユーザ用のデータベースが含まれる
- ・これ以外にも用途に応じてデータベースを追加できる
- ・これら複数のデータベースの集合体をデータベースクラスタと呼んでいる（PostgreSQL独自の用語）
- ・個々のデータベースは、表や関数などのオブジェクトを独立して管理する
- ・ユーザ情報はグローバルデータなので、全データベースで共有される
- ・PostgreSQLのサーバプロセスは、1つのデータベースクラスタを管理できる、つまりクラスタ内の複数のデータベースを管理できる



データベースクラスタのイメージ

OSS-DB

データベースクラスタ





データベースの初期化、起動と終了

OSS-DB

■ いざれもデータベース管理者（通常はpostgres）ユーザで実行すること

■ データベースクラスタの新規作成

- `initdb` コマンド

- 主なオプション

- `--D` : データベースクラスタを作成するディレクトリ
- `--E` : デフォルトのエンコーディング(UTF8など)
- `--locale` : ロケール(ja_JPなど)
- `--no-locale` : ロケールを使用しない(cにする)

■ データベースの起動

- `pg_ctl start`

- 主なオプション

- `--D` : データベースクラスタのあるディレクトリ

■ データベースの終了

- `pg_ctl stop`

- 主なオプション

- `--D` : データベースクラスタのあるディレクトリ
- `-m` : 停止モード(smart/fast/immediate)

■ `--D` オプションを省略すると、環境変数 PGDATA が使われる



■ DBサーバーのリソースなど、各種パラメータの設定をするファイル

- ・データベースクラスタのある（環境変数 PGDATA で指定される）ディレクトリにある
- ・'#'で始まる行はコメント
- ・"パラメータ名 = 値" という形式でパラメータを設定
- ・主なパラメータと設定の例
 - `listen_address = '*' (TCP接続を許可する)`
 - `shared_buffers = 256MB (共有バッファのサイズを256MBにする)`
 - `log_line_prefix = '%t %p' (ログ出力時に、時刻とプロセスIDを付加)`
- ・この他、パフォーマンスチューニングなどのための多数のパラメータが設定できるが、OSS-DB Silverの試験で問われるのは、以下の4つ（数字はバージョン9.0のマニュアルの節番号）
 - 記述方法 (18.1)
 - 接続と認証 (18.3)
 - クライアント接続デフォルト (18.10)
 - エラー報告とログ取得 (18.7)



■マニュアルの18.7節（エラー報告とログ取得）を参照

- ・ログ機能自体は充実しているが、デフォルトの設定では必要な情報が出力されない

■log_destination

- ・ログの出力先
- ・stderr（デフォルト）、csvlog、syslog、eventlog（Windowsのみ）
から、カンマ区切りで複数指定可能

■logging_collector

- ・on に設定すると stderr/csvlog で出力されたログをリダイレクトできる

■log_filename

- ・logging_collectorにより出力されるファイル名を指定
- ・デフォルトは postgresql-%Y-%m-%d_%H%M%S.log で、csvlog の場合は拡張子
が .csv になる

■log_connections

- ・on に設定すると、クライアントの接続認証をログに出力する

■log_line_prefix

- ・各ログ行の先頭に出力する文字列を printf 形式で指定
- ・デフォルトは空文字列
- ・リダイレクトを使う場合、%t（時刻）、%p（プロセスID）などを入れることは必須



■ 共有メモリなどのデフォルト設定が小さいので、ハードウェア資源を有効に活用して最高のパフォーマンスを出すためには、設定を変更する必要がある

■ `shared_buffers`

- ・ 共有メモリバッファのサイズ、デフォルトは32MB
- ・ RAMが1GB以上あるシステムでの推奨サイズはシステムメモリの25%

■ `checkpoint_segments`

- ・ このパラメータで指定した個数のWALファイル (トランザクションログ、16MB) が書き出されると、自動的にチェックポイントが発生する
- ・ デフォルトは3
- ・ 10以上が推奨、更新が多いシステムでは大きめ (32以上) にする。

■ `wal_buffers`

- ・ WAL出力に使われるバッファのメモリサイズ
- ・ デフォルトは64kB (PostgreSQL 9.0まで)
- ・ PostgreSQL 9.1ではデフォルトが変更、`shared_buffers`の1/32とWALファイルのサイズ (16MB) の小さい方



wal_level

- WALに書き出す情報の種類を指定
- 値は、minimal (default), archive, hot_standby
- ログアーカイブ (PITR) を使うには archive または hot_standby に設定

archive_mode

- ログアーカイブを使うには on に設定

archive_command

- WALファイルの退避に使うシェルコマンド
- 例:archive_command = 'cp %p /mnt/pg-arch/%f'

archive_timeout

- WALファイルが一杯にならなくても (16MBに達しなくても) 強制的にアーカイブさせる (次のWALファイルに切り替える)までの時間を秒数で指定
- デフォルトは0 (強制切り替えしない)
- 数分程度 (例えば300) に設定するのが合理的
 - 強制アーカイブした場合でもファイルサイズは16MB
 - 5分だと、1日あたり、16MB*12*24~5GB のアーカイブが作成されることにも注意
 - レプリケーションの運用も検討する



設定ファイル (pg_hba.conf)

OSS-DB

■ HBA=Host Based Authentication

■ DBへの接続を許可(あるいは拒否)する接続元、データベース、ユーザの組み合わせを設定

- ・先頭行から順に調べて、マッチする組み合わせが見つかったところで終了
- ・マッチする組み合わせが見つからなければ、接続拒否

■ 主な記述形式

- ・ local database名 ユーザ名 認証方法
- ・ host database名 ユーザ名 接続元IPアドレス 認証方法

■ 主な認証方法

- ・ md5 (パスワード認証)
- ・ ident (OSと同じユーザ名の時のみ接続可(パスワード不要))
- ・ trust (常に接続可(パスワード認証などを実行せずに接続を許可する))
- ・ reject (常に接続不可(パスワード認証などを実行せずに接続を拒絶する))

■ 記述例

- ・ local all postgres md5 (postgresユーザでの接続はパスワードを要求)
- ・ local all all ident (OSのユーザ名とDBのユーザ名が一致すれば接続可)
- ・ host all all 127.0.0.1/32 trust (ローカルホストからは常に接続可)
- ・ host db1 all 192.168.0.0/24 reject (192.168.0.1-255からdb1には接続不可)

■ デフォルト設定はパッケージによって異なるが、多くの場合、localhostからは、trustかidentで接続できるようになっている

- ・ Linuxでは、su - postgres とすれば、パスワードなしで管理者権限でDBに接続できる、ということ



■ データベースに接続してSQLを実行するには psql コマンドを使う

```
psql [option...] [dbname [username]]
```

■ 主なオプション

- -d, --dbname : 接続先データベース名
- -U, --username : 接続時のユーザ名
- -h, --host : 接続先サーバのホスト名
- -p, --port : 接続先ホストのポート番号
- -f, --file : 使用するファイル名 (psql では入力スクリプト)
 - 以上は他のツール (pg_dumpなど) でも共通に使われるオプション
- -l, --list : 利用可能なデータベースの一覧表示して終了

■ '\' (環境によっては'¥') で始まるのは psql の独自コマンド (メタコマンド)。
改行によって終了し、psql ツールによって処理される。

■ それ以外のものはSQL文と判断され、データベースのサーバープロセスに送信される。SQL文は";" (セミコロン) で終了する。改行では終了せず、次行以降に継続される (改行はスペースと同じ)。



■ 主な psql のメタコマンド ('=>' は psql のプロンプト)

- => \d (テーブル一覧の表示)
- => \d 表名 (指定した表の列名、データ型の表示)
- => \du (ユーザー一覧の表示)
- => \set (内部変数の表示・設定)
- => \c *db_name* (他のデータベースに接続)
- => \? (psql で使える各種コマンドに関するヘルプの表示)
- => \h (SQL に関するヘルプの表示)
 - => \h SELECT (SELECT の使い方に関するヘルプの表示)
- => \! *command* (OSコマンドの実行)
 - => \! ls (カレントディレクトリのファイル一覧の表示)
- => \q (終了)



■ 実行時パラメータの設定値は、データベースに接続して SHOW コマンドを実行することで確認できる

- => SHOW log_destination;
- => SHOW ALL;

■ 実行時パラメータの多くは、データベースに接続して SET コマンドを実行することで変更できる。ただし、その変更は現行セッション（あるいはトランザクション）内でのみ有効。

- => SET client_encoding TO 'UTF8';
- (注意) psql の \set メタコマンド（内部変数の表示と変更）と混同しないこと

■ postgresql.conf や pg_hba.conf の設定変更は、ファイルを変更しただけでは有効にならない。多くのパラメータは postgres ユーザで

```
$ pg_ctl reload
```

を実行することで反映される。一部のパラメータはデータベースの再起動

```
$ pg_ctl restart
```

をしないと変更が反映されない。

■ Linuxの場合、pg_ctl を使う代わりに、root ユーザで

```
# service postgresql-9.0 reload あるいは
```

```
# /etc/rc.d/init.d/postgresql-9.0 reload
```

としても良い（試験対策としては pg_ctl を覚えること）



■一般ユーザと管理者ユーザ（スーパーユーザ）

- OSに一般ユーザと管理者ユーザがあるのと同じように、データベースにも一般ユーザと管理者ユーザがある。
- 一般ユーザには限られた権限しかないが、管理者ユーザにはすべての権限がある。
- OSの管理者ユーザと、データベースの管理者ユーザは異なる。
例えば、`root` で `pg_ctl` コマンドを実行することはできない。

■権限とは？

- 多くの種類の権限があるが、例えば
 - 新規にテーブルを作成する権限、あるいは削除する権限
 - テーブルからデータを検索 (SELECT) する権限
 - テーブルのデータを更新 (UPDATE) する権限
- デフォルトでは、テーブルの所有者 (作成者) だけが、そのテーブルに対する SELECT/UPDATE などの権限を持つ (管理者ユーザは別)。
つまり、権限を与えられなければ、他人のDBやテーブルを参照 / 更新できない。



■ ユーザ作成

- `postgres` ユーザで `createuser` コマンドを使う。
 - `$ createuser [option] [username]`
- オプションで指定しなかった場合、以下を対話的に入力する。
 - 新規ユーザ名
 - 新規ユーザを管理者ユーザとするかどうか
 - 新規ユーザにデータベース作成の権限を与えるかどうか
 - 新規ユーザにユーザ作成の権限を与えるかどうか
 - (注意) PostgreSQL 9.2では仕様が変更になり、`--interactive` オプションを指定しなければ、対話的入力を行わない
- あるいは、`CREATEROLE` 権限のあるユーザで `psql` を使って接続し、`CREATE USER` 文を使う。
 - `=# CREATE USER name [option];`
 - 対話的な入力による権限設定はできない。

■ ユーザ削除

- `dropuser` コマンド、または `DROP USER` 文を使う
- 当該ユーザがテーブルなど何らかのオブジェクトを所有している場合、それらすべて削除しなければユーザを削除することはできない



■データベースクラスタに関する権限

- CREATEDB, CREATEROLE などデータベースクラスタに関する権限は、ユーザ作成時に付与するか、あるいは ALTER USER 文で付与・剥奪する
 - =# ALTER USER *username* CREATEDB NOCREATEROLE;

■データベース内のオブジェクトに関する権限

- テーブルなどのオブジェクトに対する権限の付与・剥奪には、GRANT 文とREVOKE 文を使う。
- 個々のユーザに対して、GRANT/REVOKEすることもできるが、ユーザ名として public を指定すれば、全ユーザに対するGRANT/REVOKEも可能。
 - => GRANT SELECT ON *table1* TO public;
 - => GRANT SELECT, UPDATE ON *table2* TO *user3*;
 - => REVOKE DELETE ON *table4* FROM public;
- GRANT/REVOKEの対象となるオブジェクトはテーブルだけではない
 - =# GRANT CREATE ON DATABASE *db5* TO *user6*;
(データベース db5 上にスキーマを作成する権限を user6 に付与)
 - =# GRANT CREATE ON SCHEMA *sc7* TO *user8*;
(スキーマ sc7 上にオブジェクト(テーブルなど)を作成する権限を user8 に付与)



■ デフォルトのアクセス権限

- オブジェクトの所有者 (=作成者) は、そのオブジェクトに対するすべての権限を有するが、他の一般ユーザは権限を与えられなければ、そのオブジェクトには一切、アクセスできない
- GRANT/REVOKE は、作成済みのオブジェクトに対するアクセス権限を変更する
- ALTER DEFAULT PRIVILEGES により、将来作成されるオブジェクトのデフォルトアクセス権限を設定できる
 - ALTER DEFAULT PRIVILEGES FOR USER *user1* GRANT SELECT ON TABLES TO *user2*;
(*user1* が将来作成するテーブルについて、*user2* は SELECT 権限を持つ)
 - テーブル、ビュー、シーケンス、関数についてのデフォルト権限を設定できる
 - 既存のオブジェクトには影響しない

■ アクセス権限の確認

- 既存のテーブルのアクセス権限は、psql の \dp メタコマンドで確認できる
- 他のオブジェクトについても、\d で始まるメタコマンドがある
- デフォルトのアクセス権限は \ddp メタコマンドで確認できる



■ PostgreSQLにおいて、ユーザとロールはまったく同じもの

- USER はデフォルトで LOGIN、ROLE はデフォルトで NOLOGIN
- CREATE USER は内部的に CREATE ROLE を呼び出す
- ALTER USER も内部的に ALTER ROLE を呼び出す
- ユーザ作成の権限は CREATEROLE
- CREATE USER と CREATE ROLE は、LOGIN 属性のデフォルト値以外はまったく同じ処理を行う

■ ロールの権限の管理

- ユーザと同じように、ALTER USER、ALTER ROLE、GRANT／REVOKEにより権限の付与、剥奪を行う
- ユーザに対してロール自身をGRANTすることができる。ロールに付与されたすべての権限が、一括してユーザに付与される。

```
-=# CREATE ROLE role1;
-=# GRANT SELECT ON table2 TO role1;
-=# GRANT role1 TO user3;
```



■ データベースクラスタ内に新規にデータベースを作成するには、`createdb` コマンドを使う、あるいはデータベースに接続して、`CREATE DATABASE` 文を使う

- `$ createdb [option...] dbname [comment]`
- `=> CREATE DATABASE dbname [option];`
- いずれの場合も `CREATEDB` 権限が必要

■ 新規に作成されるデータベースは、(オプションで指定しなければ) テンプレートデータベース `template1` のコピーとなる

- すべてのデータベースで共通に利用したいオブジェクトや関数定義などは、事前に `template1` に作成しておく
- 文字セットが異なる場合はコピーできない、例えば `template1` が `UTF8` のとき、`EUC` のデータベースを `template1` のコピーとして作成することはできないので、`template0` のコピーとして作成する

```
- $ createdb -E EUC_JP -T template0 dbname  
- => CREATE DATABASE dbname TEMPLATE template0 'EUC_JP';
```

■ データベースを削除するには、`dropdb` コマンド、または `DROP DATABASE` 文を使う

- 元に戻せないので要注意
- データベースの所有者、または管理者ユーザだけが実行できる



- データベースでは重要なデータを管理している。ディスクの故障などによるデータの損失に備え、バックアップを取得することが重要
- データベースではメモリ上のデータ（キャッシュ）が最新。キャッシュとディスク上のデータファイルの内容が一致するとは限らない、つまり、OSコマンドを使ってファイルをコピーしてもバックアップにはならない
 - ・ データベースのバックアップには特殊な方法（専用のコマンド）が必要
- データベースがクラッシュしたとき、一週間前のバックアップからデータベースが復元（リストア）できても、ありがたくないかもしれない
 - ・ クラッシュ直前の状態にデータを復旧（リカバリ）するためのバックアップ手段がある
- バックアップの方法とリストア・リカバリの方法をセットで覚えること
 - ・ バックアップを作っても、いざというときに使えなければ役に立たない
- (参考) http://www.oss-db.jp/measures/dojo_04.shtml



■ pg_dump コマンド

- ・データベース単位でバックアップを作成
- ・psql または pg_restore コマンドを使ってリストア

■ pg_dumpall コマンド

- ・データベースクラスタ全体のバックアップを作成
- ・psql コマンドを使ってリストア

■ コールドバックアップ (ディレクトリコピー)

- ・OS付属のコピー、アーカイブ用コマンドを使ってバックアップを作成
- ・簡単で確実な方法だが、データベースを停止する必要がある

■ ポイント・イン・タイム・リカバリ (PITR)

- ・使い方がやや複雑
- ・WAL (Write Ahead Logging) 機能と組み合わせて、任意の時点にリカバリ可能

■ COPY 文、\copy メタコマンド

- ・テーブル単位でCSV形式ファイルの入出力



■ データベースを停止せずに、データベース単位のバックアップを取得

- \$ pg_dump [options] -f *dumpfilename* *dbname* あるいは
- \$ pg_dump [options] *dbname* > *dumpfilename*
- -F オプションで、出力形式を指定できる。p (plain) はテキスト形式（デフォルト）、c (custom) はカスタム（バイナリ）形式、t (tar) はTAR形式
 - (参考) PostgreSQL 9.1 で新しいオプション d (directory) が新設された
- データベースクラスタ内のすべてのデータベースのバックアップを取得するには、`pg_dumpall` コマンドを使う。（出力形式はテキストのみ）

■ テキスト形式 (p) のバックアップは `psql` コマンドで、バイナリ形式 (c/t/d) のバックアップは `pg_restore` コマンドでリストアする。

- \$ psql -f *dumpfilename* *dbname* あるいは
- \$ psql *dbname* < *dumpfilename*
- \$ pg_restore -d *dbname* *dumpfilename*

■ `pg_dump` が作成するテキスト形式のバックアップはSQLのスクリプト (CREATE TABLE, COPY など) となっており、エディタで修正可能



■ データベースを停止せずに、データベースクラスタ全体のバックアップを取得

- \$ pg_dumpall [options] -f *dumpfilename* あるいは
- \$ pg_dumpall [options] > *dumpfilename*

■ ユーザ情報などのグローバルオブジェクトもバックアップ可能 (pg_dump では取得できない)

- -g オプションを指定すると、グローバルオブジェクトのみバックアップする

■ 出力フォーマットはテキスト形式のみなので psql コマンドでリストアする。データベース名は任意。空のクラスタにロードするときは postgres を指定すればよい

- \$ psql -f *dumpfilename* postgres あるいは
- \$ psql postgres < *dumpfilename*



■ pg_dump の -F オプションで出力ファイルのフォーマットを指定

- p (plain、デフォルト) はテキスト形式
 - CREATE TABLE、COPY などの SQL スクリプトが出力される
 - --inserts オプションを指定すると、COPY 文の代わりに INSERT 文を使うので、他のデータベースへのデータインポートにも利用可能
- c (custom) は圧縮されたアーカイブ形式 (バイナリ)
 - リストア時の取り扱いが最も柔軟
 - マルチプロセスによる高速リストアが可能
- t (tar) はLinuxなどのTARによるアーカイブ形式 (バイナリ、非圧縮形式)
 - リストア用の SQL スクリプトと、各テーブルごとのデータファイルがTAR形式で1つのファイルにアーカイブされている
- d (directory) は、指定のディレクトリの下に、リストア用の SQL スクリプト (バイナリ形式) と各オブジェクトのデータファイル (圧縮形式) を作成
 - TARで1ファイルにまとめる代わりに、圧縮された多数のファイルを作成
 - ダンプ、リストアとも、-f オプションでディレクトリ名を指定する
 - PostgreSQL 9.1で新設されたオプション

■ pg_dumpall はテキスト形式のみ

- -g オプションを指定すると、グローバルデータのみのバックアップ

■ テキスト形式は psql、バイナリ形式は pg_restore でリストア



■ディレクトリコピーによるバックアップ

- ・データベースを停止すれば、物理的なデータファイルをディレクトリごとコピーすることでバックアップを作成できる。(コールドバックアップ)
- ・コピーの方法は自由に選んで良い。(cp, tar, cpio, zip…)
 - \$ cp -r data backupdir
 - \$ tar czf backup.tgz data
- ・簡単で確実な方法だが、頻繁には実行できない

■バックアップを、同じ構成の別のマシンにコピーして動かすこともできる

- ・バックアップ作成と逆のことをすればリストアできる
 - \$ cp -r backupdir data
 - \$ tar xzf backup.tgz
- ・コピー元とコピー先で、PostgreSQLのメジャーバージョンが一致していること

■参考:コールドバックアップに対し、データベースの稼働中に取得するバックアップをホットバックアップと呼ぶ



■ PITR (Point In Time Recovery)

- ・障害の直前の状態までデータを復旧 (リカバリ) できる。
- ・間違ってデータを削除した場合でも、任意の時点まで戻すことができる。

■ PITRの仕組み

- ・WAL (Write Ahead Logging) により、データファイルへの書き込み前に、変更操作についてログ出力される。 (トランザクションログ)
- ・WALファイルをアーカイブして保存しておく
- ・最後のバックアップ (ベースバックアップ) に対して、障害発生直前までのWALを適用することで、データを復旧できる。

■ PITRによるベースバックアップの取得手順

- ・スーパーユーザで接続し、バックアップ開始をサーバに通知
 - =# SELECT pg_start_backup('label');
- ・tar, cpio などのOSコマンドでバックアップを取得 (サーバーは止めない)
- ・再度、スーパーユーザで接続し、バックアップ終了をサーバに通知
 - =# SELECT pg_stop_backup();
- ・(参考) PostgreSQL 9.1では pg_basebackup コマンドにより、上記の手順をまとめて実行できる
- ・(参考) レプリケーションはPITRと同じ原理で動作している。同じ手順でベースバックアップを取得し、WALデータを転送して適用することでデータベースを複製している



■必要な設定 (postgresql.conf)

- wal_level を archive または hot_standby にする
- archive_mode を on にする
- archive_command を適切に設定し、WAL ファイルが安全な場所にコピーされるようにする

■リカバリの方法

- ベースバックアップからリストア
- pg_xlog ディレクトリ内の古いファイルはすべて削除
- アーカイブされていない新しいWALファイルがあれば、pg_xlog ディレクトリにコピー
- recovery.conf ファイルを作成し、restore_command を適切に設定
- サーバを起動すれば、自動的にリカバリされる
- recovery.conf ファイルの名前を変更する (または移動する)

■より安全な運用のために

- pg_xlog ディレクトリは、データベースクラスタと物理的に異なるディスクにする
- archive_command によるコピー先も、物理的に異なるディスクにする
- archive_timeout を適切な値にする (パフォーマンス上、問題がない範囲で短く)
- 定期的にベースバックアップを取得する (リカバリに要する時間を短くするため、また保存すべきアーカイブログの量を削減するため)
- レプリケーションなど他の手段も組み合わせて運用する
- pg_xlog ディレクトリが失われると未アーカイブのトランザクションはリカバリできない (不完全リカバリとなる) ことに注意



CSVファイルの入出力

OSS-DB

■ psql の \copy メタコマンド、あるいは SQL の COPY 文を使うと、データベースのテーブルと、OSファイルシステム上のファイル(CSVなど)の間で入出力ができる。

■ \copy メタコマンドの基本的な使い方

- => \copy *table_name* to *file_name* [*options*]
- => \copy *table_name* from *file_name* [*options*]
- デフォルトではタブ区切りのテキストファイルを入出力、
オプションに"csv"と指定すれば、カンマ区切りのCSVファイルになる。

■ SQLの COPY 文はPostgreSQLの独自拡張機能。使い方の違いに注意。

- =# COPY *table_name* TO '*file_name*' [*options*];
- =# COPY *table_name* FROM '*file_name*' [*options*];
- \copy メタコマンドは psql によって処理されるのでクライアント上のファイルの入出力、COPY 文は SQL として実行されるのでサーバ上のファイルの入出力。
- SQL文として扱われる所以、ファイル名(文字列)は引用符で括る必要がある。
- COPY 文によるファイル入出力は、サーバー上のファイルを読み書きすることになるため、データベース管理者ユーザでしか実行できない、という制限がある。
- COPY 文でファイル名を STDOUT あるいは STDIN (引用符なし) とすると、標準入出力とのデータのやり取りになる。この場合は一般ユーザでも実行できる。

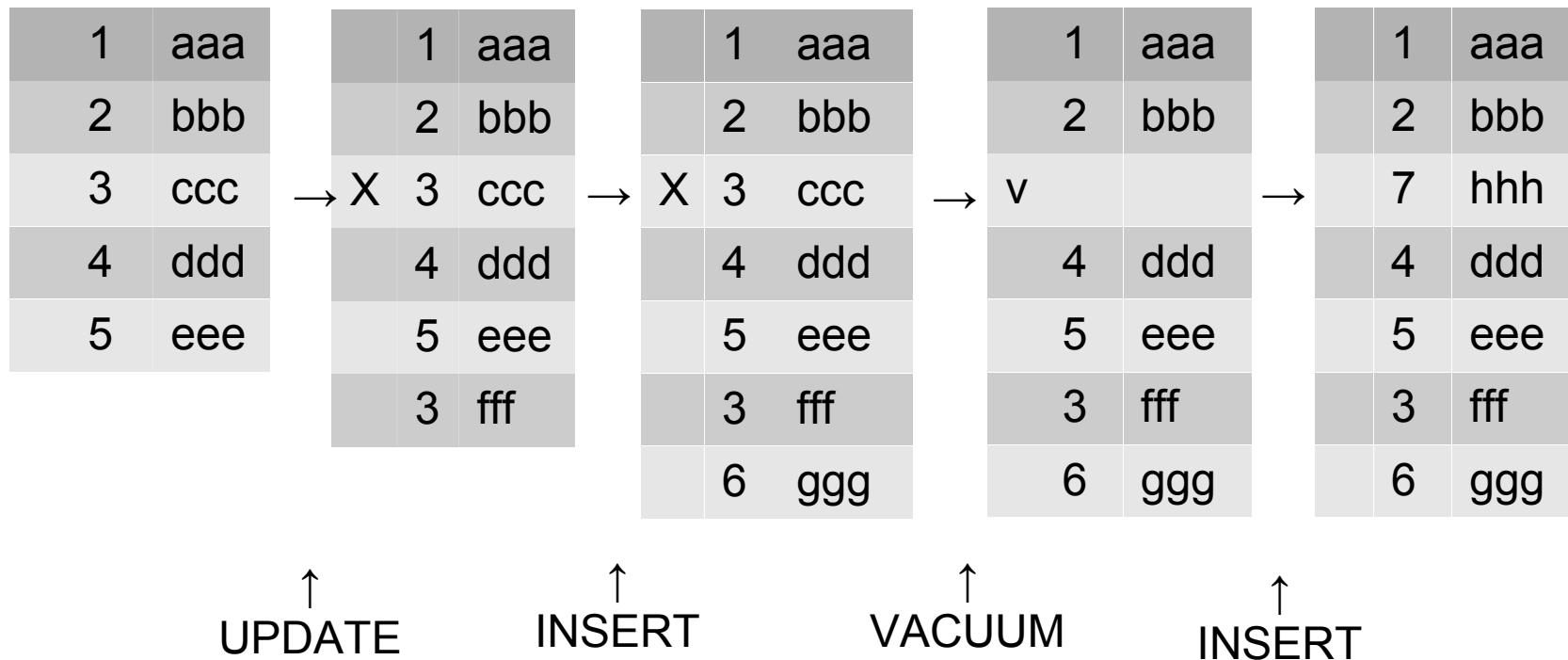
■ (参考) http://www.oss-db.jp/measures/dojo_07.shtml



- PostgreSQLのデータファイルは追記型の構造。データが更新されると、旧データには削除マークが付けられ、新データはファイルの末尾に追加される。削除マークの付いた領域は、そのまま残る(再利用されない)
 - ・これにより、読み取り一貫性の機能が実現されている
 - ・UPDATE=DELETE+INSERT だが、DELETEされた領域がそのまま残るので、更新のたびにデータファイルが大きくなる
- データの更新が繰り返されると、ファイルサイズが増大し、ディスク容量不足やパフォーマンス問題を引き起こす
- VACUUM は削除マークがついたデータ領域を回収し、再利用可能にする
 - ・VACUUMを実行した後のINSERTは、回収された領域を使うので、データファイルが大きくならない

■ PostgreSQLでは

- UPDATEはDELETE+INSERTとして処理される
- DELETEされた行の領域はそのままでは再利用されない
- 削除された領域を再利用可能にするための仕組みがVACUUM





■ コマンドラインから `vacuumdb` コマンド、あるいはデータベースに接続して `VACUUM` 文を実行する。

■ `VACUUM`, `vacuumdb` の主なオプション

- `ANALYZE`, `-z`, `--analyze` : 統計情報の取得も同時に実施
- `FULL`, `-f`, `--full` : データを移動し、ファイルサイズを小さくする
 - 時間がかかる上、テーブルロックが発生するので注意。原則として利用しない
- `VERBOSE`, `-v`, `--verbose` : 処理内容の詳細を画面に出力する
- `-a`, `--all` : クラスタ内の全データベースに対して `VACUUM` を実施

■ (参考) `VACUUM`と`VACUUM FULL`

- http://wiki.postgresql.org/wiki/VACUUM_FULL/ja



- VACUUM を自動的に実行する機能
- デフォルトの設定では、自動的に実行されるようになっており、これが推奨の設定でもある
- VACUUM と ANALYZE が自動的に実行される
- テーブル毎に、データの変更量が設定値を超えると実行される

- PostgreSQLの古いバージョンでは、手動で、あるいは cron で定期的に VACUUM を実行する必要があった
- autovacuum により、管理者が VACUUM を意識する必要性が低くなっているが、機能については理解しておくこと
 - PostgreSQL 7.4 で contrib として提供
 - PostgreSQL 8.1 で本体機能に組み込み
 - PostgreSQL 8.3 からデフォルトで ON



ポイント解説：SQL



■ SQLとは

- Structured Query Language
- RDBMSにアクセス（データの検索と更新）するときに使われる言語

■ RDBMSで重要な概念

- 表 (table)
- 列 (column, field)
- 行 (row, record)

■ SQLの区分

- DDL (Data Definition Language)、DML (Data Manipulation Language)、DCL (Data Control Language) に大別される
- DDL (CREATE TABLE, ALTER TABLE) で表と列を定義し、DML (SELECT, INSERT, UPDATE, DELETE) でデータの検索と更新を行う

■ 言語としての特徴

- ANSI/ISOで標準化されている（どのRDBMSでも利用できる）
- 大文字/小文字を区別しない（文字列を除く）
- IF/THEN/ELSEやGOTOなど、あるいは変数や配列を使った、いわゆるプログラミングはSQLだけではできない（他の言語のプログラム中にSQLを埋め込むことで実現する）



- SQLはANSIで標準化されており、RDBMSの種類による違いは小さい
- SQL文 (DML/DDL/DCL) については差分が小さいが、データ型 (種類と実装)、関数 (特に文字列関数や時間関数) はRDBMSの種類による違いが大きい
- 標準準拠の程度はRDBMSの種類によるが、PostgreSQLは準拠度が比較的高い
- PostgreSQLのマニュアルでは、各所にその機能がANSI標準なのか、PostgreSQLの独自拡張なのかの別が記述されている
- OracleなどANSI標準の策定前から存在していたRDBMSには、標準にならない仕様が数多く残っているが、現在のバージョンでは標準の仕様の多くが取り入れられている



■ オープンソースデータベース標準教科書

- <http://www.oss-db.jp/ossdbtext/text.shtml>
- SQLについて何も知らない人を対象に基礎から解説
- PDF版とEPUB版(スマートフォンなどで利用可能)を無料でダウンロード可能





■ 数値型

- SMALLINT (2バイト)、INTEGER (4バイト)、BIGINT (8バイト)
- NUMERIC (最大1000桁)、DECIMAL (NUMERICと同じ)
- REAL (4バイト)、DOUBLE PRECISION (8バイト)
- SERIAL (自動増分4バイト)、BIGSERIAL (自動増分8バイト)

■ 文字列型

- CHARACTER VARYING (可変長、最大4096文字)、VARCHAR (CHARACTER VARYINGと同じ)
- CHARACTER (固定長)、CHAR (CHARACTERと同じ)
- TEXT (可変長、無制限)

■ 日付型

- DATE (日付のみ)
- TIME (時刻のみ)
- TIMESTAMP (日付+時刻)

■ 論理値型

- BOOLEAN (TRUE/FALSE)



■ 共通のものが多いが、微妙に仕様が異なることがある

- **INTEGER 型**: PostgreSQLでは4バイトの整数、Oracleでは38桁の10進数
- **VARCHAR 型**: PostgreSQLでは文字数を指定、最大4096文字、Oracleではバイト数を指定、最大4000バイト
- **DATE 型**: PostgreSQLでは日付のみ、Oracleでは日付 + 時刻

■ 多くのRDBMSでほぼ同じように使えるもの

- INTEGER, NUMERIC
- CHAR, VARCHAR
- TIMESTAMP

■ PostgreSQL独自のデータ型

- **SERIAL/BIGSERIAL** : 自動的にシーケンスが作成され、列値を連番にできる
- **TEXT** : 可変長文字列だが、最大長を指定しなくて良いので便利
- **BOOLEAN** : 論理値型
 - TRUE/'t'/'true'/'y'/'yes'/'on'/'1'
 - FALSE/'f'/'false'/'n'/'no'/'off'/'0'
 - 大文字・小文字は区別しない、TRUE/FALSE はキーワード、他は文字列

■ (参考) http://www.oss-db.jp/measures/dojo_05.shtml



表(テーブル)の作成 - 基礎編

OSS-DB

- 表は CREATE TABLE 文で作成する。

表名 → CANDIDATE(受験者表)

```
CREATE TABLE table_name (
    column_name1 data_type1,
    column_name2 data_type2...
);
```

列名 →

CID(受験者番号)	NAME(氏名)
1	小沢次郎
2	石原伸子
3	戌井玄太郎
4	山本花子

行 →

↑
列

■ 例:

- ```
CREATE TABLE candidate(
 cid INTEGER,
 name VARCHAR (20)
);
```

- CREATE TABLE 文はデータの入れ物を作るだけなので、実行した直後はデータは入っていない

- SQLでは(文字列を除き)大文字と小文字は区別されない。コマンドだけでなく、表名や列名でも大文字と小文字は区別されない。本資料内では予約語を大文字、他を小文字で記述しているが、すべて小文字(あるいは大文字)で書いて構わない

- 表や列の名前に日本語(漢字)を使用しても問題なく動作することが多いが、一般的には望ましくないので、表名、列名には英数字のみを使うことを推奨する



# 表(テーブル)の作成 - 応用編

OSS-DB

■ 表の列に、一意、**非NULL**、外部キーなどの制約をつけたり、デフォルト値を設定したりできる。制約は、CREATE TABLE による作成時に指定することも、作成後に ALTER TABLE 文で追加することもできる

## ■ 主な制約

- NOT NULL : 値が NULL でない
- UNIQUE : 値が一意 (列値が同じである行が他に存在しない)
- PRIMARY KEY : 主キー (UNIQUE かつ NOT NULL)
- FOREIGN KEY (REFERENCES) : 外部キー (別テーブルに列値が同じ行が存在する)
- CHECK : 列の有効値を数式などで定義

## ■ 例:

- candidate表に主キー制約を追加

```
ALTER TABLE candidate ADD CONSTRAINT cid_p PRIMARY KEY (cid);
```

- exam表の作成時に各種制約を指定

```
CREATE TABLE exam (
 eid INTEGER PRIMARY KEY,
 cid INTEGER REFERENCES candidate(cid),
 exam_name VARCHAR(10) NOT NULL,
 exam_date DATE,
 score INTEGER DEFAULT 0,
 grade VARCHAR(10));
```

## ■ 制約に違反するデータ挿入、データ更新はエラーになる



■ CREATE TABLE については、制約の付与を含め、差異はほとんどない

- データ型の差異があるので、同じ DDL がそのまま使えるとは限らない

■ ALTER TABLE については、文法上の微妙な差異が多い。PostgreSQL では、ALTER TABLE *table\_name* に続いて、

- 列の追加は ADD [COLUMN] *column\_definition*
- 列の削除は DROP [COLUMN] *column\_name*
- 列属性の変更は ALTER [COLUMN] *column\_name* *new\_def*
- 列名の変更は RENAME *column\_name* TO *new\_column\_name*

■ RDBMSの種類による主な違い(参考)

- ADD/DROP/ALTER/RENAME の後に COLUMN と書くかどうか
- ADD などで指定する列定義を括弧で囲うかどうか
- 列属性の変更は ALTER か MODIFY か



■ データを検索 (問い合わせ) して表示するには SELECT 文を使う

■ SELECT 文には3つの基本機能がある

- 選択 (selection)
  - 行の抽出
  - 条件を指定し、それに合った行だけを表示する
- 射影 (projection)
  - 列の抽出
  - 指定した列だけを取り出して表示する
- 結合 (join)
  - 複数の表を結合して1つの表として扱う

■ 複雑な問い合わせも、これらの基本機能の組み合わせ



# SELECT 文 - 選択

OSS-DB

## ■ 指定した条件に合致した行を表示する

| EID | CID | EX_NAME | EX_DATE   | SCORE | GRADE |
|-----|-----|---------|-----------|-------|-------|
| 1   | 1   | Silver  | 2011/7/1  | 80    | Pass  |
| 2   | 2   | Silver  | 2011/7/1  | 75    | Pass  |
| 3   | 3   | Silver  | 2011/7/2  | 50    | Fail  |
| 4   | 1   | Gold    | 2011/7/4  | 40    | Fail  |
| 5   | 2   | Gold    | 2011/7/12 | 85    | Pass  |
| 6   | 1   | Gold    | 2011/7/14 | 70    | Pass  |



| EID | CID | EX_NAME | EX_DATE   | SCORE | GRADE |
|-----|-----|---------|-----------|-------|-------|
| 1   | 1   | Silver  | 2011/7/1  | 80    | Pass  |
| 2   | 2   | Silver  | 2011/7/1  | 75    | Pass  |
| 5   | 2   | Gold    | 2011/7/12 | 85    | Pass  |
| 6   | 1   | Gold    | 2011/7/14 | 70    | Pass  |

## ■ 基本的構文は

```
SELECT * FROM table_name WHERE condition;
```

## ■ WHERE 以下は省略できて、省略するとすべての行を表示する

## ■ 例えば、GRADE が Pass のデータだけを表示するには

```
SELECT * FROM exam WHERE grade = 'Pass';
```



# SELECT 文 - 射影

OSS-DB

## ■ 指定した列だけを表示する

| CID | NAME  |
|-----|-------|
| 1   | 小沢次郎  |
| 2   | 石原伸子  |
| 3   | 戌井玄太郎 |
| 4   | 山本花子  |



| NAME  |
|-------|
| 小沢次郎  |
| 石原伸子  |
| 戌井玄太郎 |
| 山本花子  |

## ■ 基本的な構文は

```
SELECT column_list FROM table_name;
```

## ■ 表示する列をカンマで区切って1つ以上指定する

## ■ 例えば、candidate 表の name 列だけを表示するには

```
SELECT name FROM candidate;
```



# SELECT 文 - 結合

OSS-DB

## 複数の表を結合する

| CID | NAME  |
|-----|-------|
| 1   | 小沢次郎  |
| 2   | 石原伸子  |
| 3   | 戌井玄太郎 |
| 4   | 山本花子  |

| EID | CID | EX_NAME | EX_DATE   | SCORE | GRADE |
|-----|-----|---------|-----------|-------|-------|
| 1   | 1   | Silver  | 2011/7/1  | 80    | Pass  |
| 2   | 2   | Silver  | 2011/7/1  | 75    | Pass  |
| 3   | 3   | Silver  | 2011/7/2  | 50    | Fail  |
| 4   | 1   | Gold    | 2011/7/4  | 40    | Fail  |
| 5   | 2   | Gold    | 2011/7/12 | 85    | Pass  |
| 6   | 1   | Gold    | 2011/7/14 | 70    | Pass  |

| CID | NAME  | EID | CID | EX_NAME | EX_DATE   | SCORE | GRADE |
|-----|-------|-----|-----|---------|-----------|-------|-------|
| 1   | 小沢次郎  | 1   | 1   | Silver  | 2011/7/1  | 80    | Pass  |
| 2   | 石原伸子  | 2   | 2   | Silver  | 2011/7/1  | 75    | Pass  |
| 3   | 戌井玄太郎 | 3   | 3   | Silver  | 2011/7/2  | 50    | Fail  |
| 1   | 小沢次郎  | 4   | 1   | Gold    | 2011/7/4  | 40    | Fail  |
| 2   | 石原伸子  | 5   | 2   | Gold    | 2011/7/12 | 85    | Pass  |
| 1   | 小沢次郎  | 6   | 1   | Gold    | 2011/7/14 | 70    | Pass  |



## ■ 表結合の SELECT 文には2通りの記述方法

- FROM 句にカンマで区切って表を並べ、WHERE 句に結合条件を記述

```
SELECT * FROM candidate, exam
WHERE candidate.cid = exam.cid;
```

- JOIN 句に結合対象表、ON 句に結合条件を記述

```
SELECT * FROM candidate
JOIN exam ON candidate.cid = exam.cid;
```

- どの表の列であるかを明示するため、列名は table\_name.column\_name の形式で記述する（1つの表にしか列名が現れない時は表名を省略可）

## ■ 結合に使う列の名前が同じなら、USING 句で簡潔に表記しても良い

- SELECT \* FROM candidate JOIN exam USING (cid);

## ■ 表に別名をつけることができる（列名の記述が簡潔になる）

- SELECT \* FROM candidate [AS] c  
JOIN exam [AS] e ON c.cid = e.cid;

- 表別名を明示する AS は省略可能（あってもなくても良い）だが、Oracleでは指定してはいけない



## ■ 選択、射影、結合を組み合わせて利用できる

```
■ SELECT column_list FROM table_name1
JOIN table_name2 ON join_condition
WHERE select_condition;
```

- 表示したい列をカンマで区切って複数並べる
- すべての列を表示するには *column\_list* を \* とする
- WHERE 句を省略すると、すべての行が表示される
- WHERE 句の条件に合致した行がないときは、1行も表示されないが、これ自体はエラーとは扱われない
- 列や条件には複雑な式や関数を利用しても良い



## ■ 単なる計算や関数の実行にも SELECT 文を使うことができる。 このとき、FROM 句は指定しない。

- ・ 単なる計算: 1日は何秒?

```
SELECT 60 * 60 * 24;
```

- ・ 複数の計算をまとめて実行: 1週間は何時間? 1年は何時間?

```
SELECT 24 * 7, 24 * 365;
```

- ・ 関数の呼び出し: 文字列の長さ?

```
SELECT length('How long is this?');
```

## ■ 指定する列や条件は、必ずしも表のデータと関係しなくてもよい

- ・ 実験: 単なる計算の FROM に通常のテーブルを指定すると何が起きるか?

```
SELECT 60 * 60 * 24 FROM candidate;
```

- ・ 実験: 表と関係のない条件を WHERE 句に書くと何が起きるか?

```
SELECT * FROM candidate WHERE 1 = 2;
```

```
SELECT * FROM candidate WHERE 1 = 1;
```

## ■ (参考) OracleやDB2では FROM 句が必須なので、ダミー表から SELECT する

- ・ SELECT 60 \* 60 \* 24 FROM dual; (Oracle)

## ■ (参考) [http://www.oss-db.jp/measures/dojo\\_09.shtml](http://www.oss-db.jp/measures/dojo_09.shtml)



# SELECT 文の応用 - 行の並べ替えと行数制限

OSS-DB

■ ORDER BY 句をすることで、表示順をソートできる。

降順にソートする場合は DESC と追記する。

デフォルトは昇順だが、明示的に ASC と追記しても良い。

- cidについて昇順、cidが同じときはexam\_dateについて降順にソート

```
SELECT * FROM exam ORDER BY cid, exam_date DESC;
```

■ 表示する行数を制限するには、LIMIT 句を使う (PostgreSQL, MySQLなど、一部の RDBMSでのみ利用可能)、OFFSET 句を組み合わせて、表示しない行数を指定できる

- exam\_dateでソートし、先頭の3行だけ表示

```
SELECT * FROM exam ORDER BY exam_date LIMIT 3;
```

- cidでソートし、3行をスキップして次の2行、つまり4行目と5行目を表示

```
SELECT * FROM exam ORDER BY cid LIMIT 2 OFFSET 3;
```

■ ORDER BY 句がないときの SELECT 文の出力順はまったく保証されないことに注意

■ (参考) Oracleでは ROWNUM という擬似列を使うことで表示する行数を制限できるが、ORDER BY 句と組み合わせて使うことができない (ROWNUM の値がソートの前に付与されるため)



# SELECT 文の応用 - データの集約

OSS-DB

- SELECT 文で、データを集約(合計、平均、最大、最小などを計算)できる
- GROUP BY 句を指定すると、特定の列の値が同じグループ同士でデータを集約できる

- 最高得点、最低得点、平均点の計算

```
SELECT max(score), min(score), avg(score) FROM exam;
```

- cidごとにグループ分けしてデータ数と平均点を表示、つまり受験者ごとの受験回数と平均点

```
SELECT cid, count(*), avg(score) FROM exam GROUP BY cid;
```

## GROUP BY, WHERE, HAVING の関係(処理順)に注意

- WHERE の条件に合致した行をすべて抽出 → GROUP BY の条件に従ってグループ分けして集約 → HAVING の条件に合致した集約行を抽出
- WHERE には集約前に判定できる条件をすべて、HAVING には集約後にしか判定できない条件を記述する
- エラーとなる SELECT の例: WHERE/HAVING に不適切な条件

- ```
SELECT cid, count(*), avg(score) FROM exam WHERE avg(score) > 75  
GROUP BY cid;
```

- ```
SELECT cid, count(*), avg(score) FROM exam GROUP BY cid
HAVING grade = 'Pass';
```

- 動作するが、適切でない SELECT の例: HAVING でなく WHERE に記述するべき

- ```
SELECT cid, count(*), avg(score) FROM exam GROUP BY cid HAVING cid < 3;
```

- 正しい SELECT の例: gradeがPassの結果についての平均点が75を超えてる受験者のデータ

- ```
SELECT cid, count(*), avg(score) FROM exam WHERE grade = 'Pass'
GROUP BY cid HAVING avg(score) > 75;
```



## ■通常の表結合（内部結合）

- ```
SELECT * FROM candidate c
JOIN exam e ON c.cid = e.cid;
```
- candidate表にデータがあっても、対応するデータがexam表になければ、データが表示されない

■外部結合を使うと、結合対象の行にデータがなくても、結合元のデータが表示される

- ```
SELECT * FROM candidate c
LEFT JOIN exam e ON c.cid = e.cid;
```
- この他に、RIGHT JOIN, FULL JOIN, CROSS JOINがある
  - (参考) いくつかのRDBMSでは、FULL JOIN や CROSS JOIN がない
- JOIN は INNER JOIN, LEFT JOIN は LEFT OUTER JOIN と書いても同じ意味になる

## ■実験：簡単なテーブルを作り、各種 JOIN の動作の違いを調べ、理解する

```
• CREATE TABLE t1 (id INTEGER, val VARCHAR(5));
CREATE TABLE t2 (id INTEGER, val VARCHAR(5));
INSERT INTO t1 VALUES (1, 'a'), (2, 'b');
INSERT INTO t2 VALUES (1, 'x'), (3, 'z');
SELECT * FROM t1 [LEFT/RIGHT/FULL] JOIN t2 USING (id);
```



- SELECT 文の中に、別の SELECT 文を書くことができる。これを副問い合わせ (subquery、サブクエリー) と呼ぶ
- 副問い合わせは、SELECT 句、FROM 句、WHERE 句のいずれにでも使うことができる
  - UPDATE 文や DELETE 文など、SELECT 文以外でも使うことができる
  - 構文によっては、副問い合わせが複数行を返すとエラーになる
- WHERE 句の副問い合わせが最もよく使われる
  - GRADE が Pass である試験結果のある受験者の一覧の出力:
    - `SELECT * FROM candidate WHERE cid IN  
(SELECT cid FROM exam WHERE grade = 'Pass');`
    - `SELECT * FROM candidate c WHERE EXISTS  
(SELECT * FROM exam e  
WHERE e.cid = c.cid AND grade = 'Pass');`
  - IN、EXISTS の他に、NOT IN、NOT EXISTS、ALL、ANY、SOME なども副問い合わせと組み合わせて利用される (ANY と SOME は同じ意味)



■複数の SELECT 文の結果に対して、和 (UNION)、差 (EXCEPT)、積 (INTERSECT) の集合演算を行うことができる

■SELECT ... UNION SELECT ... ;

複数の SELECT 文の結果の和集合を返す

- SELECT する列の数と型が一致している必要がある
- まったく同じ結果が複数あった場合、1行だけが返される
- UNION の代わりに UNION ALL とすると、重複行も含めてすべての行が返る

■SELECT ... EXCEPT SELECT ... ;

最初の問い合わせの結果のうち、EXCEPT 以降の問い合わせの結果に含まれるもののが除外される

- 一部のRDBMSではサポートされない
- Oracleでは EXCEPT の代わりに MINUS を使う

■SELECT ... INTERSECT SELECT ... ;

両方の問い合わせの結果に含まれるものだけが返される

- 一部のRDBMSではサポートされない



# SELECT 文の応用 - ビュー (VIEW)

OSS-DB

## ■ SELECT 文をビューとして定義することで、SELECT 文の結果をテーブルであるかのごとく扱うことができる

- 表の結合をビューで表現:

```
CREATE VIEW exam_view AS
SELECT e.eid, c.cid, c.name, e.exam_name, e.exam_date,
e.score, e.grade
FROM exam e JOIN candidate c ON e.cid = c.cid;
```

- データの集約をビューで表現:

```
CREATE VIEW exam_summary AS
SELECT cid, count(*), avg(score), max(exam_date)
FROM exam GROUP BY cid;
```

## ■ ビューからの SELECT はテーブルからと同じように実行できる

- SELECT \* FROM exam\_summary;
- SELECT name, exam\_name, exam\_date FROM exam\_view  
WHERE cid = 1;

## ■ ビューは更新 (INSERT/UPDATE/DELETE) できない

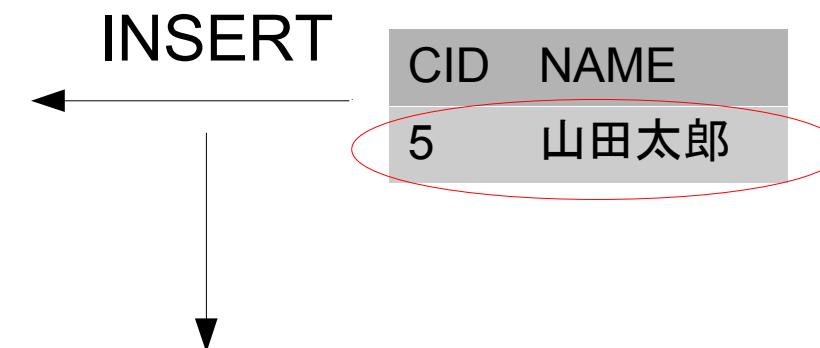
- ルール (RULE) を定義すれば更新可能
- 他のRDBMSでは、ビューが更新可能なものもある (ただし、更新可能かどうかはビューの定義にも依存する)



## ■ 表にデータを追加(挿入)するには INSERT 文を使う

- RDBMSの表はデータの「集合」であって、データ間に順序はない
- INSERTは「挿入」という意味だが、実態としてはデータの「追加」

| CID | NAME  |
|-----|-------|
| 1   | 小沢次郎  |
| 2   | 石原伸子  |
| 3   | 戌井玄太郎 |
| 4   | 山本花子  |



| CID | NAME  |
|-----|-------|
| 1   | 小沢次郎  |
| 2   | 石原伸子  |
| 3   | 戌井玄太郎 |
| 4   | 山本花子  |
| 5   | 山田太郎  |



# INSERT 文 - 構文と実施例

OSS-DB

## ■ INSERT INTO *table\_name* (*column\_list*) VALUES (*value\_list*);

- *column\_list* に指定しなかった列には、列のデフォルト値（設定がなければ NULL）が入る
- 全列にデータを入れるときは *column\_list* を省略しても良い（プログラムとして書く場合は、習慣として必ず列リストを指定すること）
- PostgreSQLでは、*column\_list*を省略した上で、一部の列だけ値を指定することも可能（OracleやMySQLではエラーになる、プログラムとして書く時は必ず列リストを指定すること）
- PostgreSQL, MySQLなど一部のRDBMSでは、（*value\_list*）をカンマで区切り複数行を1回の INSERT で追加できる（Oracleなどでは不可）

## ■ 例：candidate表に行を追加

- 対象列を（表定義とは異なる順で）指定して1行追加

```
INSERT INTO candidate(name, cid) VALUES ('山田太郎', 5);
```

- 対象列を省略して2行追加（RDBMSの種類によってはエラーになる）

```
INSERT INTO candidate VALUES (6, '鈴木イチロー'), (7, '松尾秀樹');
```

- 一部の列だけを指定して1行追加

```
INSERT INTO candidate (cid) VALUES (8);
```

- (参考) PostgreSQLでは上のINSERTを以下のように記述しても同じ（RDBMSの種類によってはエラーになる）

```
INSERT INTO candidate VALUES (8);
```



# INSERT 文 - 応用編

OSS-DB

## 他のテーブルのデータを参照した INSERT を実行できる

| new_exam表 |     |      |         |
|-----------|-----|------|---------|
| EID       | CID | NAME | EX_DATE |
|           |     |      |         |

SELECT+INSERT

| candidate表 |       |
|------------|-------|
| CID        | NAME  |
| 1          | 小沢次郎  |
| 2          | 石原伸子  |
| 3          | 戌井玄太郎 |
| 4          | 山本花子  |

exam表

| EID | CID | EX_NAME | EX_DATE   | SCORE | GRADE |
|-----|-----|---------|-----------|-------|-------|
| 1   | 1   | Silver  | 2011/7/1  | 80    | Pass  |
| 2   | 2   | Silver  | 2011/7/1  | 75    | Pass  |
| 3   | 3   | Silver  | 2011/7/2  | 50    | Fail  |
| 4   | 1   | Gold    | 2011/7/4  | 40    | Fail  |
| 5   | 2   | Gold    | 2011/7/12 | 85    | Pass  |
| 6   | 1   | Gold    | 2011/7/14 | 70    | Pass  |

| EID | CID | NAME  | EX_DATE   |
|-----|-----|-------|-----------|
| 1   | 1   | 小沢次郎  | 2011/7/1  |
| 2   | 2   | 石原伸子  | 2011/7/1  |
| 3   | 3   | 戌井玄太郎 | 2011/7/2  |
| 4   | 1   | 小沢次郎  | 2011/7/4  |
| 5   | 2   | 石原伸子  | 2011/7/12 |
| 6   | 1   | 小沢次郎  | 2011/7/14 |



## ■ VALUES 句の代わりに SELECT 文を書くこともできる

- (準備) 新しいテーブルを作成:

```
CREATE TABLE new_exam (eid INTEGER, cid INTEGER,
name VARCHAR(20), exam_date DATE, score INTEGER, grade
VARCHAR(10));
```

- INSERT ~ SELECT によるデータの追加:

```
INSERT INTO new_exam (eid, cid, name, exam_date)
SELECT e.eid, c.cid, c.name, e.exam_date FROM exam e
JOIN candidate c ON e.cid = c.cid;
```

## ■ 参考: CREATE TABLE AS あるいは SELECT INTO を使うと、新規テーブルを作成すると同時に SELECT の結果をテーブルに入れることができる。ただし、いずれも一部のRDBMSでしか利用できない

- CREATE TABLE new\_exam1 AS  
SELECT e.eid, c.cid, c.name, e.exam\_date FROM exam e  
JOIN candidate c ON e.cid = c.cid;
- SELECT e.eid, c.cid, c.name, e.exam\_date INTO new\_exam2  
FROM exam e JOIN candidate c ON e.cid = c.cid;



## ■ 表のデータを変更するには UPDATE 文を使う

| EID | CID | EX_NAME | EX_DATE   | SCORE | GRADE |
|-----|-----|---------|-----------|-------|-------|
| 1   | 1   | Silver  | 2011/7/1  | 80    | Pass  |
| 2   | 2   | Silver  | 2011/7/1  | 75    | Pass  |
| 3   | 3   | Silver  | 2011/7/2  | 50    | Fail  |
| 4   | 1   | Gold    | 2011/7/4  | 40    | Fail  |
| 5   | 2   | Gold    | 2011/7/12 | 85    | Pass  |
| 6   | 1   | Gold    | 2011/7/14 | 70    | Pass  |

UPDATE

| EID | SCORE | GRADE |
|-----|-------|-------|
| 5   | 65    | Fail  |



| EID | CID | EX_NAME | EX_DATE   | SCORE | GRADE |
|-----|-----|---------|-----------|-------|-------|
| 1   | 1   | Silver  | 2011/7/1  | 80    | Pass  |
| 2   | 2   | Silver  | 2011/7/1  | 75    | Pass  |
| 3   | 3   | Silver  | 2011/7/2  | 50    | Fail  |
| 4   | 1   | Gold    | 2011/7/4  | 40    | Fail  |
| 5   | 2   | Gold    | 2011/7/12 | 65    | Fail  |
| 6   | 1   | Gold    | 2011/7/14 | 70    | Pass  |



■ UPDATE *table\_name* SET *col\_name* = *new\_val*  
WHERE *condition*;

- “*col\_name=new\_val*” の部分をカンマで区切って複数並べれば、複数の列の値を同時に更新できる
- WHERE 句を省略すると、すべての行が更新される（要注意）
- WHERE 句の条件に合致したデータがなければ1行も更新されないが、これ自体はエラーとはならない

■ 例：exam表で、eidが5の行について、scoreとgradeの値を変更

- UPDATE exam SET score = 65, grade = 'Fail'  
WHERE eid = 5;
- (参考) 同じ更新を、リスト形式を使って  
UPDATE exam SET (score, grade) = (65, 'Fail')  
WHERE cid = 5;  
と書くこともできるが、RDBMSの種類によってはエラーになる

■ トランザクションの機能を使っていなければ、データは即座に更新され、取り消し不可以（OracleやDB2に慣れた人は要注意）



# UPDATE 文 - 応用編

OSS-DB

## ■他のテーブルを参照したデータ更新(副問い合わせの利用)

new\_exam表

| EID | CID | NAME  | EX_DATE   | SCORE | GRADE |
|-----|-----|-------|-----------|-------|-------|
| 1   | 1   | 小沢次郎  | 2011/7/1  |       |       |
| 2   | 2   | 石原伸子  | 2011/7/1  |       |       |
| 3   | 3   | 戌井玄太郎 | 2011/7/2  |       |       |
| 4   | 1   | 小沢次郎  | 2011/7/4  |       |       |
| 5   | 2   | 石原伸子  | 2011/7/12 |       |       |
| 6   | 1   | 小沢次郎  | 2011/7/14 |       |       |

UPDATE

exam表

| EID | SCORE | GRADE |
|-----|-------|-------|
| 1   | 80    | Pass  |
| 2   | 75    | Pass  |
| 3   | 50    | Fail  |
| 4   | 40    | Fail  |
| 5   | 85    | Pass  |
| 6   | 70    | Pass  |

| EID | CID | NAME  | EX_DATE   | SCORE | GRADE |
|-----|-----|-------|-----------|-------|-------|
| 1   | 1   | 小沢次郎  | 2011/7/1  | 80    | Pass  |
| 2   | 2   | 石原伸子  | 2011/7/1  | 75    | Pass  |
| 3   | 3   | 戌井玄太郎 | 2011/7/2  | 50    | Fail  |
| 4   | 1   | 小沢次郎  | 2011/7/4  | 40    | Fail  |
| 5   | 2   | 石原伸子  | 2011/7/12 | 85    | Pass  |
| 6   | 1   | 小沢次郎  | 2011/7/14 | 70    | Pass  |



## ■ UPDATE 文の SET 句に副問い合わせを書くことができる

- 例: new\_exam 表の score 列に、exam 表から該当するデータをコピーする

```
- UPDATE new_exam n
SET score = (SELECT score FROM exam e WHERE n.eid = e.eid);
```

## ■ 注意事項

- SET 句に記述した SELECT 文が複数の行を返した場合は、UPDATE 文自体がエラーとなり、データは更新されない (RDBMSの種類によっては、一部の行が更新される)

```
- UPDATE new_exam n
SET score = (SELECT score FROM exam e WHERE n.cid = e.cid);
→ 副問い合わせが複数行を返すのでエラーになる
```

- SET 句に記述した SELECT 文が行を返さなかった場合、列の値は NULL に更新される。更新されたくない場合は、WHERE 句に適切な条件を記述する必要がある

```
- UPDATE new_exam n
SET score =
(SELECT score FROM exam e WHERE e.eid = n.eid)
WHERE EXISTS
(SELECT * FROM exam e WHERE e.eid = n.eid);
```



## ■副問い合わせを利用した UPDATE 文で、複数列を更新したい

- UPDATE new\_exam  
SET score = (SELECT score...), grade = (SELECT grade...)  
WHERE ... ;  
とすれば、どの RDBMS でも動作するが、ちょっと冗長

## ■RDBMS依存だが、それに簡潔な記述法がある

- PostgreSQLの場合 ~ FROM 句を使って表を結合できる

```
UPDATE new_exam n
SET (score, grade) = (e.score, e.grade)
FROM exam e WHERE n.eid = e.eid;
```

- Oracleの場合 ~ SET 句で SELECT リストを指定可能

```
UPDATE new_exam n SET (score, grade) =
(SELECT score, grade FROM exam e WHERE e.eid = n.eid);
```

- MySQLの場合 ~ 更新対象表を複数指定することで、表を結合できる

```
UPDATE new_exam n, exam e
SET n.score = e.score, n.grade = e.grade
WHERE n.eid = e.eid;
```



## ■ 表のデータを削除するには DELETE 文を使う

■ `DELETE FROM table_name WHERE condition;`

- WHERE 句を省略すると、すべての行が削除される（要注意）
- WHERE 句の条件に合致した行がなければ1行も削除されないが、これ自体はエラーとはならない

■ トランザクションの機能を使っていなければ、データは即座に削除され、取り消しできない（OracleやDB2に慣れた人は要注意）

■ 例：candidate表から行を削除

- cidの値が7の行を削除

```
DELETE FROM candidate WHERE cid = 7;
```

- nameの値がNULLである行をすべて削除

```
DELETE FROM candidate WHERE name IS NULL;
```

■ (参考) すべての行を削除するには、DELETE よりも TRUNCATE が高速

- `TRUNCATE [TABLE] table_name;`



## ■他のテーブルを参照 (副問い合わせを利用) した DELETE の例

- 試験データのない受験者を削除

- ```
DELETE FROM candidate c WHERE NOT EXISTS
(SELECT * FROM exam e WHERE e.cid = c.cid);
```
- ```
DELETE FROM candidate
WHERE cid NOT IN (SELECT cid FROM exam);
```

- new\_exam 表にコピー済みのデータを exam 表から削除

- ```
DELETE FROM exam e WHERE EXISTS
(SELECT * FROM new_exam n WHERE n.eid = e.eid);
```
- ```
DELETE FROM exam
WHERE eid IN (SELECT eid FROM new_exam);
```

- DELETE FROM で表別名が使えないRDBMSもあるので注意

## ■(参考) PostgreSQL では USING 句を使ってテーブル結合できる (独自拡張) ので、コピー済みのデータの削除は以下でも実行できる

- ```
DELETE FROM exam e
USING new_exam n
WHERE n.eid = e.eid;
```



■ RDBMSにおけるトランザクションとは？

- ・複数の更新を1つの処理としてまとめたもの、例えば…
- ・銀行口座間の資金の移動で、口座Aの残高を減らす UPDATE 文と、口座Bの残高を増やす UPDATE 文
- ・売上伝票の入力で、伝票ヘッダを入力する INSERT 文と、明細行を入力する INSERT 文

■トランザクション内の複数の更新は、そのすべてがデータベースに反映されているか、あるいはまったく反映されていないか、のどちらか。
一部だけが反映されている状態には（一時的であっても）ならない。

■トランザクション内で1つ以上の更新SQLを発行した後、COMMIT を実行すると、すべての更新がまとめて反映される。COMMIT の代わりに ROLLBACK を実行すると、トランザクション内のすべての更新が破棄される

■トランザクション内で実行され、まだ COMMIT されていない更新 SQL の結果について、トランザクション内では参照できるが、他のクライアントからは見ることができない（更新前のデータが見える）

■（参考） http://www.oss-db.jp/measures/dojo_01.shtml



■ PostgreSQLでは、BEGIN または START TRANSACTION 文でトランザクションが開始され、COMMIT または ROLLBACK 文で終了する

- (デフォルトでは) BEGIN を実行しないとトランザクションは開始しない
- COMMIT/ROLLBACK の後は、再度 BEGIN を実行する必要がある

■ SAVEPOINT, ROLLBACK TO *savepoint* などの基本を理解する

- COMMIT: すべての更新をデータベースに反映させる
- SAVEPOINT *sp_name*: トランザクションの一時保存
- ROLLBACK TO *sp_name*: 更新を一部キャンセル、一時保存した状態まで戻る
- ROLLBACK: 一時保存を含め、すべての更新をキャンセル

■ トランザクションの外部で実行されるSQL文(INSERT/UPDATE/DELETE)は自動的に COMMIT される (OracleやDB2に慣れた人は要注意)

■ (参考) PostgreSQLでは CREATE TABLE, DROP TABLE などのDDLもトランザクションの一部になるので、DDLによる自動 COMMIT は発生せず、ROLLBACK すれば DROP TABLE されたテーブルも元に戻る

- Oracleなどでは、DDLを実行すると、トランザクションが自動的に COMMIT される



■ PostgreSQLでは、トランザクションの途中でエラーが発生すると、以後のSQLはすべてエラーとなり、ROLLBACKするしかなくなるので注意が必要

- SQLの文法エラー、DBの制約違反(一意性、外部参照など)によるエラー、いずれの場合も ROLLBACKが必要
- この状態で COMMIT を発行すると、ROLLBACKが実行される
- 自動コミットを offにして操作すると、この問題に陥ることが多くなる
- 回避策は、エラーになる可能性のあるSQLを実行する前に SAVEPOINTを実行し、エラーが発生したらその SAVEPOINTまで ROLLBACKすること
- Oracleなどでは、エラーが発生しても、処理の継続が可能

■例

```
• CREATE TABLE tableau (id INTEGER UNIQUE, val VARCHAR(10));
BEGIN;
INSERT INTO tableau VALUES (1, 'aaa'), (2, 'bbb');
SAVEPOINT sp1;
INSERT INTO tableau VALUES (2, 'ccc'); ←エラー!! (UNIQUE 制約に違反)
SELECT * FROM tableau; ←すべてのSQLがエラーになってしまう
ROLLBACK TO sp1; ←これがないと、次の COMMITで ROLLBACKされる!
COMMIT;
```



■ CREATE SEQUENCE 文で明示的に作成することができる他、SERIAL 型(4バイト)または BIGSERIAL 型(8バイト)の列を作ることで自動的に作成される

- CREATE SEQUENCE *seq_name* [*options*] ;
- デフォルトでは8バイト

■ シーケンス名と同じ名前の特別なテーブルが自動的に作成される

- SELECT * FROM *seq_name*;

■ シーケンスの現在値は `currval()`、次の値は `nextval()` 関数で取得。現在値の変更には `setval()` 関数を使う

- SELECT `currval('seq_name')` ;
- SELECT `nextval('seq_name')` ;
- SELECT `setval('seq_name', 100)` ;
- シーケンスを利用するための関数名は他の RDBMS と同じだが、呼び出し方が違うので注意

■ SERIAL/BIGSERIAL 型の列については、INSERT 時に列を指定しない、あるいは列の値として DEFAULT を指定すると、シーケンスの次の値が使われる

- CREATE TABLE *seq_test* (*id* BIGSERIAL, *val* VARCHAR(50)) ;
INSERT INTO *seq_test* (*val*) VALUES ('abc') ;
INSERT INTO *seq_test* (*id*, *val*) VALUES (DEFAULT, 'xyz') ;

■ (参考) http://www.oss-db.jp/measures/dojo_06.shtml



■ 集約関数

- `count`, `sum`, `avg`, `max`, `min`
- **NULL 値の扱いに注意**
 - `count(*)` はすべての列が `NULL` であっても1件のデータとしてカウントする
 - `count(col)` は、`col` の値が `NULL` のものを除いたデータ数を返す
 - `avg(col)` は `NULL` を除いたデータの平均値、つまり `sum(col) / count(col)` を返す
- (参考) http://www.oss-db.jp/measures/dojo_08.shtml

■ 算術演算子、算術関数

- +、-、*、/ の算術演算子は標準通り
- 剰余計算に MOD 関数の他、% 演算子が使える (Oracle, DB2などは MOD のみ)
- べき乗計算に POWER 関数の他、^ 演算子が使える (他の主なRDBMSは POWER のみ)
- 乱数発生に RANDOM 関数が用意されており、0と1の間の小数值を返す (PostgreSQL独自)



■ 文字列リテラル（文字列定数）

- SQLの文字列リテラルはシングルクオートで囲まれ、大文字と小文字は区別される
 - ダブルクオートで囲った文字列をリテラルとして使えるRDBMSもあるが、一般には列別名などシングルクオートとは異なる特定の用途でしか使えない
 - `SELECT col1 "col #1" FROM table1 WHERE...;`
- 空文字列('')とNULLは別のもの('' IS NULLはFALSE、'' = ''はTRUE)
 - `SELECT 1 WHERE '' = '';`
`SELECT 1 WHERE '' IS NULL;`
を試してみると良い
 - (参考) Oracleでは''とNULLは同じものとして扱われるので、'' IS NULLはTRUE、'' = ''はFALSEとなる(上のSELECT文の結果が逆になる)
- 文字列中にシングルクオートを入れるにはシングルクオートを2つ並べる
 - 'I can't do it.'
 - '''' とあつたら、これは「'」という文字列を表す
- (参考) \$tag\$で文字列リテラルを記述することも可能(PostgreSQL独自)
 - \$xyz\$I can't do it.\$xyz\$: 'I can't do it.'と同じ
 - tagはなくても良く、\$\$I can't do it.\$\$という記述でもOK
 - Oracleでは、Q'XstringX' (Xは任意の文字、Qは小文字でも可)という記述がある
例えば、q'xI can't do it.x'
- (参考) http://www.oss-db.jp/measures/dojo_02.shtml



■ 文字列結合

- 文字列の結合には ANSI 標準の `||` 演算子を使う
 - `||` が利用できないRDBMS、代わりに `+` を使うRDBMS、`concat` 関数を使うRDBMSもある
- `'abc' || NULL` は `NULL` になる
 - Oracleでは `'abc'` になる
 - (参考) http://www.oss-db.jp/measures/dojo_08.shtml

■ 文字列比較

- `LIKE` で、ワイルドカードの `_%` を使ったマッチングは非常に重要
 - `SELECT * FROM table1 WHERE col1 LIKE 'a_c%';`
- `=` を使った比較では、`_%` がワイルドカードにならないことにも注意
- 大文字と小文字は区別されるが、MySQLのように(デフォルトでは)区別しないRDBMSもある

■ 正規表現

- `~` 演算子で、指定の正規表現を含む文字列とマッチさせられる
 - `SELECT * FROM table1 WHERE col1 ~ '^ [a-c] ';`
- `SIMILAR TO` は `LIKE` とほぼ同じ使い方だが、正規表現の一部をサポートする
 - `SELECT * FROM table1 WHERE col1 SIMILAR TO '[a-c] %';`
 - 上記の例はいずれも、`col1`の先頭文字がa, b, cのいずれかである行の検索
- 正規表現は多くのRDBMSが何らかの方法でサポートしているが、実装方法はRDBMSの種類によって大きく異なる



■ 文字列関数

- RDBMSの種類によって実装されている関数に違いがある
- 以下のものはANSI標準で定義されており、大多数のRDBMSで利用可能
- 文字列の変換:UPPER, LOWER
- 文字列の置換:REPLACE, TRANSLATE
- 文字の削除:TRIM, RTRIM, LTRIM
- 文字列の長さ:LENGTH, CHARACTER_LENGTH, OCTET_LENGTH
- 部分文字列:SUBSTRING, POSITION
- ASCII変換:ASCII, CHR

■ 現在では、どのRDBMSでもマルチバイト文字は当然のようにサポートされており、(CHARACTER_) LENGTH 関数はバイト数ではなく文字数を返す。バイト数を調べたいときは OCTET_LENGTH 関数を使う(Oracleでは LENGTHB 関数)。

■ (参考) Oracleには SUBSTRING 関数ではなく、代わりに SUBSTR 関数を使う



■ 時間関数

- RDBMSの種類によって実装されている関数に大きな違いがある
- 現在日時の取得: `CURRENT_DATE`, `CURRENT_TIME`,
`CURRENT_TIMESTAMP`
 - これらは関数名の後に括弧を付けずに使うことに注意
- 日時から要素の取得: `EXTRACT`, `TO_CHAR`

■ 期間リテラル

- 記述方法はRDBMSの種類によって大きく異なる
 - `INTERVAL '10' YEAR` (Oracle)
 - `10 YEARS` (DB2)
 - `INTERVAL '10 YEAR'` (PostgreSQL)
 - `INTERVAL 10 YEAR` (MySQL)
- 例えば、1ヶ月後の日付をPostgreSQLで表示するには
 - `SELECT current_date + INTERVAL '1 MONTH';` あるいは
 - `SELECT current_date + '1 MONTH'::INTERVAL;`



■ 変換関数

- TO_CHAR, TO_NUMBER, TO_DATE などは、OracleでもPostgreSQLでも使えるが、他のRDBMSには使えないものが多い
- 型変換: CAST (ANSI標準)
 - TO_xxx の代わりに CAST 関数を使う
 - `SELECT cast('2011-10-01' AS DATE) + 10;`
- PostgreSQL独自の型変換方式として :: 演算子を使う方法がある
 - `SELECT '2011-10-01'::DATE + 10;`
- NULL の代替値: COALESCE
 - `SELECT coalesce(val1, val2...);`
val1, val2...のうち、最初の NULL でないものが返る
 - Oracleの NVL の拡張
- 場合分け: CASE/WHEN/THEN/ELSE/END
 - `SELECT CASE col1 WHEN val1 THEN 'xxx' WHEN val2 THEN 'yyy' ELSE 'zzz' END FROM table1;`
 - Oracleの DECODE と同じ機能 (PostgreSQL／MySQLでは DECODE は復号化)



■スキーマとは

- データベース内の名前空間
 - データベース内に1つ以上のスキーマが存在する
 - どのテーブルもいずれかのスキーマに属する
- 階層化できないディレクトリ、と考えると良い
 - スキーマの下にさらにスキーマを作ることはできない
 - 異なるスキーマの下に同じ名前のテーブルが存在してもよく、これらはまったく独立した別のテーブルである
- SQLでテーブル名を指定するときは、`schema_name.table_name` の形式で指定する。ただし、スキーマ名は省略可能で、テーブル名だけ指定しても良い
- さらにデータベース名をつけて、`db_name.schema_name.table_name` の形式で指定しても良いが、(現行バージョンの) PostgreSQLでは、接続中のDB以外にはアクセスないので、`db_name` は接続中のDB名と同じでなければならない

■PostgreSQLにおけるスキーマの利用

- CREATE SCHEMA 文で作成
- DROP SCHEMA 文で削除
- ALTER SCHEMA 文で名前や所有者を変更できる
- `public` というスキーマがデフォルトで存在する



■スキーマ検索パス

- => SHOW search_path; で確認できる
- デフォルトでは、"\$user", public となっている
- この設定を変えると、スキーマ名省略時の動作が変わる。例えば、自分のユーザ名と異なるスキーマのテーブルに対して、スキーマ名を明示しなくてもアクセスできる。

■スキーマ名を省略した時の動作 (デフォルト設定の場合)

- ユーザ foo が SELECT * FROM table1; を実行
 - テーブル foo.table1 があれば、そこから SELECT する
 - なければ、public.table1 から SELECT する
 - いずれのテーブルもなければエラーとなる
- ユーザ foo が CREATE TABLE table2 ...; を実行
 - スキーマ foo があれば foo.table2 を作成する
 - なければ public.table2 を作成する
- デフォルトの状態では public スキーマのみが存在しているので、新規テーブルはすべて public スキーマに作成され、テーブルの検索や更新もすべて public スキーマのテーブルに対して実行される
- public スキーマを削除 (DROP SCHEMA) してもよい



■ PostgreSQL独自の概念

- public スキーマ
- スキーマ検索パス

■ Oracleの場合

- ユーザ名とスキーマ名はほぼ同義で、ユーザが作られると、同じ名前のスキーマが自動的に作成される
- CREATE/DROP/ALTER SCHEMA はない
- シノニム (SYNONYM) を使えば、他人のスキーマ内のオブジェクトに、スキーマ名を指定せずにアクセスできる

■ MySQLの場合

- スキーマという概念が存在しない
- 同じインスタンス内の他のデータベース内のテーブルにアクセスできるので、これを利用すると、構文的にはスキーマがあるのと同じになる(これをスキーマと呼ぶ人も多い)
 - `SELECT * FROM db_name.table_name;`
 - `SELECT * FROM information_schema.tables;`



■ PL/pgSQLという、OracleのPL/SQLに似た言語でストアドプログラムを作成できる

- ・マニュアルに、Oracle PL/SQLからの移植についての節(39.12)もある

■ 事前に、`createlang plpgsql` を実行して、手続き言語の使用についてDBに登録しておく必要があるが、PostgreSQL 9.0ではデフォルトで登録済み(`createlang -l` で確認できる)

- ・(参考) PostgreSQL 9.1 のマニュアルによると、`createlang` コマンドは将来のバージョンで廃止される可能性があり、代わりに、データベースに接続して `CREATE EXTENSION` 文を使うべき、とのこと

■ PL/pgSQLでなく、SQLで関数定義をすることもできる。この他、標準で、PL/Tcl, PL/Perl, PL/Pythonを提供

- ・標準以外でも、PL/Java, PL/PHP, PL/Rubyなど多数の言語が利用可能



■ PL/pgSQLによる関数の例

- ```
CREATE FUNCTION test2(INTEGER) RETURNS INTEGER AS $$
DECLARE
 di ALIAS FOR $1;
 d INTEGER;
BEGIN
 d := di * 2;
 RETURN d;
END;
$$ LANGUAGE plpgsql;
```
- ```
SELECT test2(3);
```

■ SQLによる関数の例

- ```
CREATE FUNCTION cname(INTEGER) RETURNS TEXT AS $$
SELECT name FROM candidate WHERE cid = $1;
$$ LANGUAGE SQL;
```
- ```
SELECT cname(2);
```



■ トリガーとは？

- ・データベースが更新されるたびに呼び出される手続き
- ・テーブルの更新 (INSERT/UPDATE/DELETE) が実行される直前、あるいは直後に呼び出すことができる
- ・1つのSQL文の実行について1度だけ実行することも、更新される各行について別々に呼び出すこともできる
- ・データベースの整合性を保持する、変更履歴を記録する、などの目的で利用できる
 - 不正な更新をエラーにする、省略された値を設定する、更新日時フィールドに値を設定する、といった処理が可能

■ トリガーの作成方法

- ・PL/pgSQLなどによるFUNCTIONを作成する
 - 引数を取らない
 - 戻り値は trigger 型として宣言
- ・作成済みの関数をCREATE TRIGGER文により割り当てる



■PL/pgSQLによる関数の作成

- ```
CREATE FUNCTION exam_grade() RETURNS TRIGGER AS $$
BEGIN
IF NEW.grade IS NULL THEN
 IF NEW.score >= 70 THEN
 NEW.grade := 'Pass';
 ELSE
 NEW.grade := 'Fail';
 END IF;
END IF;
RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

## ■CREATE TRIGGERによる割り当て

- ```
CREATE TRIGGER exam_grade  
BEFORE INSERT ON exam FOR EACH ROW  
EXECUTE PROCEDURE exam_grade();
```

■実行例

- ```
INSERT INTO exam (eid, cid, exam_name, exam_date, score)
VALUES (10, 4, 'Gold', current_date, 60);
```
- ```
SELECT * FROM exam WHERE eid = 10;
```



■ ビュー (VIEW) の更新

- ・ ビューの更新 (INSERT/UPDATE/DELETE) の可否はRDBMSに依存する
 - PostgreSQLではビューを更新できない
- ・ ビューの更新の可否はビューの定義にも依存する
 - ビューの更新をサポートするRDBMSでも、ビューの定義によってはビューを更新できない
 - 複数のテーブルを結合しているとき、どのテーブルを更新する？ →わからない
 - 集約関数を使っているとき、どの行を更新する？ →そもそも更新対象となるデータがない

■ ルール (RULE) とは？

- ・ ビューの更新を実現するためのPostgreSQL独自の方式
- ・ ビューに対して更新のSQLが実行された時、代わりにルールとして定義された手続きを呼び出す
 - (やろうと思えば) どんなビューでも更新できる、直感と異なる更新もできる
- ・ テーブルに対するルールを作ることもでき、トリガーの代わりに使うことも可能



■ルールの作成

- `CREATE [OR REPLACE] RULE rule_name AS
ON event TO view_name [WHERE condition]
DO INSTEAD command;`

■例:ビューエクスアーヴィーへのINSERTを表エクスアーマーへのINSERTとする

- `CREATE RULE exam_view_insert AS
ON INSERT TO exam_view
DO INSTEAD
INSERT INTO exam VALUES(NEW.eid, NEW.cid,
NEW.exam_name, NEW.exam_date, NEW.score, NEW.grade);`

■実行例

- `INSERT INTO exam_view (eid, cid, name, exam_name,
exam_date, score)
VALUES (11, 1, 'Foo Bar', 'Gold', current_date, 90);`
- `SELECT * FROM exam_view WHERE eid = 11;`



例題解説



■一般知識 - ライセンス

PostgreSQLの利用条件、ライセンスについて、正しいものを2つ選びなさい。

- A. PostgreSQLのソースコードは公開されており、誰でもダウンロードできる
- B. PostgreSQLのソースコードを改変して再配布するとき、ソースコードを公開する義務がある
- C. PostgreSQLのソースコードを改変して再配布するとき、著作権表示をする必要はない
- D. PostgreSQLの開発者は、致命的なバグについて修正の義務を負っている
- E. PostgreSQLはユーザ登録することなく、無料で利用できる



■運用管理 - 標準付属ツールの使い方

以下の記述から、誤っているものを2つ選びなさい。

- A. `createdb` コマンドでデータベースを作成するには CREATEDB 権限が必要である
- B. `dropdb` コマンドでデータベースを削除するには CREATEDB 権限が必要である
- C. `dropdb` コマンドでデータベースを削除する前に、そのデータベース内のテーブルなどすべてのオブジェクトを削除しておく必要がある
- D. `dropuser` コマンドでユーザを削除するには、CREATEROLE 権限が必要である
- E. `dropuser` コマンドでユーザを削除する前に、そのユーザが所有するすべてのテーブルを削除しておく必要がある



■運用管理 - バックアップ方法

PostgreSQLのバックアップに関する以下の記述から、誤っているものを1つ選びなさい。

- A. `pg_dump` コマンドを使ってバックアップを作成し、
`psql` コマンドを使ってそれをリストアした
- B. `pg_dumpall` コマンドを使ってバックアップ作成し、
`pg_restore` コマンドを使ってそれをリストアした
- C. ハードディスクが破損してデータベースが起動しなくなってしまったが、
ポイントインタイムリカバリ (PITR) 機能を使っていたので、
クラッシュ直前の状態にまで復旧させることができた
- D. テーブルを CSV 形式でバックアップするために、
`psql` でデータベースに接続し、`COPY` 文を実行した
- E. CSV 形式のファイルをデータベースにアップロードするために、
`psql` でデータベースに接続し、`\copy` メタコマンドを実行した



■運用管理 - 基本的な運用管理作業

バキューム (VACUUM) 機能に関する説明について、正しいものを2つ選んでください。

- A. バキュームを実行するには、コマンドラインから vacuumdb コマンドを実行する
- B. 自動バキュームを実行するには、コマンドラインから autovacuumdb コマンドを実行する
- C. バキュームにより削除領域が回収されると、通常はデータベースファイルのサイズが小さくなる
- D. psql でデータベースに接続して VACUUM 文を実行する際、テーブル単位あるいはデータベース単位でバキュームを実行することができる
- E. 自動バキュームでは、指定した時間おきに、自動的にバキュームが実行される



■SQL - 集約関数

以下のSQL文を順次実行した。最後の SELECT 文が返す値の組み合わせとして適切なものはどれか。

```
CREATE TABLE test1 (id INTEGER, val INTEGER);
INSERT INTO test1 VALUES (1, 10), (2, 20);
INSERT INTO test1 VALUES (3, NULL), (4, 30);
INSERT INTO test1 VALUES (NULL, NULL);
SELECT count(*), count(val), avg(val) FROM test1;
```

- A. 5, 5, 12
- B. 5, 5, 20
- C. 5, 3, 20
- D. 4, 4, 15
- E. 4, 3, 20



■SQL - トランザクション

以下のSQL文を順次実行した。実行後のテーブル t1 の行数は何行か。

```
CREATE TABLE t1 (id INTEGER, val VARCHAR(10));
BEGIN;
INSERT INTO t1 VALUES (1, 'aaa'), (2, 'bbb');
SAVEPOINT sp1;
DELETE FROM t1 WHERE id = 1;
SAVEPOINT sp2;
INSERT INTO t1 VALUES (3, 'ccc');
ROLLBACK to sp1;
INSERT INTO t1 VALUES (4, 'ddd'), (5, 'eee');
COMMIT;
```



参考資料

OSS-DB

■ OSS教科書OSS-DB Silver

- ・認定教材

■ オープンソースデータベース 標準教科書

- ・初心者向けにSQLの初步から
Webアプリケーション開発まで

■ PostgreSQL徹底入門

- ・PostgreSQL 9.0対応
- ・9.0.1のインストーラ、ソースコード

■ PostgreSQL全機能バイブル

- ・PostgreSQL 9.2対応
- ・11/16発売予定

■ PostgreSQL – Up and Running

- ・PostgreSQL 9.1/9.2対応
- ・現在は英語のみ

■ 日本PostgreSQLユーザ会

<http://www.postgresql.jp/>

■ Let's Postgres

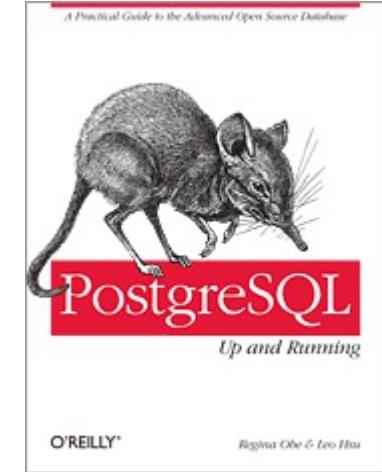
<http://lets.postgresql.jp/>

■ オンラインマニュアル

<http://www.postgresql.jp/document/9.0/html/>



オープンソースデータベース標準教科書
— PostgreSQL —
(Ver.1.0.0)





ご清聴ありがとうございました。

■お問い合わせ■

LPI-Japan
テクノロジー・マネージャー
松田 神一
matsuda@lpi.or.jp



■ 使用するサンプルデータの作成

- CREATE TABLE candidate (cid INTEGER PRIMARY KEY, name VARCHAR (20));
- CREATE TABLE exam (eid INTEGER PRIMARY KEY, cid INTEGER REFERENCES candidate (cid) , exam_name VARCHAR (10) , exam_date DATE, score INTEGER, grade VARCHAR (10));
- INSERT INTO candidate (cid, name) VALUES (1, '小沢次郎'), (2, '石原伸子'), (3, '戌井玄太郎'), (4, '山本花子');
- INSERT INTO exam (eid, cid, exam_name, exam_date, score, grade) VALUES (1, 1, 'Silver', '2011-07-01', 80, 'Pass'), (2, 2, 'Silver', '2011-07-01', 75, 'Pass'), (3, 3, 'Silver', '2011-07-02', 50, 'Fail'), (4, 1, 'Gold', '2011-07-04', 40, 'Fail'), (5, 2, 'Gold', '2011-07-12', 85, 'Pass'), (6, 1, 'Gold', '2011-07-14', 70, 'Pass');



参考資料：SQLの説明で使用しているテーブル

OSS-DB

CANDIDATE(受験者表)

CID(受験者番号)	NAME(氏名)
1	小沢次郎
2	石原伸子
3	戌井玄太郎
4	山本花子

EXAM(試験結果表)

EID (試験ID)	CID (受験者ID)	EXAM_NAME (試験名)	EXAM_DATE (試験日)	SCORE (得点)	GRADE (合否)
1	1	Silver	2011/7/1	80	Pass
2	2	Silver	2011/7/1	75	Pass
3	3	Silver	2011/7/2	50	Fail
4	1	Gold	2011/7/4	40	Fail
5	2	Gold	2011/7/12	85	Pass
6	1	Gold	2011/7/14	70	Pass