オープンソースデータベース標準教科書

― PostgreSQL ―

（Ver.2.0.1）

LPI-JAPAN

目次

まえがき

このたび、特定非営利活動法人エルピーアイジャパンは、オープンソースデータベース技術者教育に利用していただくことを目的とした教材、「オープンソースデータベース標準教科書」を開発し、インターネット上にて公開し、提供することとなりました。

この「オープンソースデータベース標準教科書」は、データベース技術習得のニーズの高まりに応えるべく、まったく初めてデータベースについて学習する人のために開発されました。既にリリースされ好評を得ている「Linux 標準教科書」「Linux サーバー構築標準教科書」の姉妹版となります。

公開にあたっては、「オープンソースデータベース標準教科書」に添付されたライセンス（クリエイティブ・コモンズ・ライセンス表示 - 非営利 - 改変禁止）の下に公開されています。

本教材は、最新の技術動向に対応するため、随時アップデートを行っていきます。また、テキスト作成やアップデートについては、意見交換のメーリングリストで、誰でも、オープンに参加できます。詳しくはオープンソースデータベース標準教科書のホームページをご覧ください。

オープンソースデータベース標準教科書 https://oss-db.jp/ossdbtext

執筆者・制作担当者紹介

松田神一（第 1 版企画・進行担当）
オープンソースソフトウェアの進歩・普及と PC の低価格化・高性能化により、誰でも簡単に Web アプリケーションを構築できるようになりました。ほとんどの Web アプリケーションでは、そのバックエンドにデータベースを利用しており、データベースの利用・運用管理技術は IT 技術者にとって必須のものとなっています。

しかし、データベースの利用に使われる SQL 言語は、他のプログラム言語とは特性が大きく異なるため、習得が簡単ではありません。データベースの初心者が、SQL 言語とデータベース運用管理技術の基礎を学ぶことのできる、教科書のようなものが必要と考えて、本書を企画しました。

宮原徹（第 1 版企画・執筆担当、第 2 版監修／株式会社びぎねっと）
本教科書は、データベースを初めて触る方でも迷わないためのガイドとなるよう、できるだけ簡潔に分かりやすく、実際に動かしてみて理解できることを目標に執筆しました。一方で、データベースは OS と同じく奥が深いソフトウェアのため、本教科書で解説できたのはほんのさわりに過ぎません。特に運用管理やパフォーマンス、データベース設計については、より詳細な解説書にあたってみてください。この教科書を手に取った方のデータベーススキル修得の一助になれば幸いです。

喜田 紘介（第 2 版執筆担当／株式会社アシスト／ NPO 法人日本 PostgreSQL ユーザ会理事）
本教科書の第 1 版が登場した当時、ちょうど自社で PostgreSQL 担当に任命されたのを良く覚えています。データベースや Linux については、仕事で必要になる度に手を止めて調べている程度にまだまだ初級者でした。ふと思い立って参加した LPI-Japan 様の Linux セミナー（講師は本書の第 1 版を執筆された宮原さんでした。）をきっかけに多くのボランティアの方で成り立つ技術コミュニティや勉強会に触れ、本教科書シリーズも当然のように全部ダウンロードしていました。本書のレベル感は、当時の私は本業でデータベースをやっていたこともあり、1 周はサラッと手を動かしてみて、短時間で入門レベルを総ざらいできたな、と感じていました。そのまま実行できるようコマンドが紹介されていますので初心者でも十分読み進めることができますし、私の場合は各実行例ごとに、もう少し違うコマンドでも状態を確認してみるなど、本書をきっかけに幅を広げようと思って 2 周目、3 周目に取り組みました。それ以降は PostgreSQL のマニュアルを隅々まで読み込むような、詳細を知らなければならない仕事をしてきたわけですが、自身がセミナー講師を担当する側になって、たまに本書を読み返しています。これから学習される方向けの外せない部分や難しいポイントを確認しては「なるほど、この説明か。」と、その度になにか発見があり、より良い仕事の助けとさせていただきました。

さて、改訂版のお話をいただいたのは、2017 年夏、PostgreSQL の大規模メジャーバージョンアップであるバージョン 10 のリリースが 2017 年秋に決定した頃でした。第 1 版当時の PostgreSQL 9.0 からは 7 つのバージョンを経ており、PostgreSQL の機能・安定性・性能いずれも格段に向上しています。データベース製品の進化だけでなく、企業での OSS 利用が当たり前に検討されるようになったことや、クラウド化をはじめとした IT インフラ環境の移り変わりなど、IT を取り巻く世の中の変化も

著しいもので、それに伴い有用な OSS 製品への興味は増していくばかりです。そんな中で、今から学習される方向けに本教科書をアップデートし、より多くの方にお役立ていただけることは、このような世界で活動することのきっかけをくれた皆様に対する一番の恩返しであると思い、ありがたくお話をお請けすることになりました。

改訂にあたって注力したことは、そのまま試せる豊富な実行例が本書の良い点ですので、2018 年時点で主流の OS、PostgreSQL 環境をお使いいただき、初めての方でも迷わず手を動かせるよう、コマンドの修正や補足説明、デフォルト値の変更などを行いました。少し応用的と思うような説明も追加しています。「そのまま試して結果を確認できる」という点では初心者の方にも変わらずご理解いただけるようにしていますし、とはいえ少し高度な内容まで扱う事で一歩ステップアップしたい方にもより満足いただけるようにできたと思います。

高橋征義（第 1 版 PDF 版・EPUB 版制作担当／株式会社達人出版会）
本教科書の PDF・EPUB 作成のお手伝いをいたしました。技術の習得において、紙の書籍で学ぶことの優位性はまだまだ大きいものがありますが、特に進化の激しい業界においては、素早い制作と頻繁な改訂を得意とする電子書籍のメリットも小さくありません。さらに、今回採用した EPUB は、PC・タブレットからスマートフォンまで、一つのファイルで対応できるのが特徴です。このような便利なテキストを提供することで、エンジニアの方々の技術力向上に貢献できれば幸いです。

中谷 徹（第 2 版 PDF 版・EPUB 版・Kindle 版制作担当／ NPO 法人 LPI-Japan）
本教科書第 2 版の PDF・EPUB の作成で協力させていただきました。第 2 版は第 1 版とは原稿のファイル形式も生成工程も異なるため、第 1 版とルック&フィールが異なってますが、ご利用いただく上で最も良いワークアラウンドを選択したつもりですのでご容赦ください。皆様の学習のお役に立てていただければ幸いです。

木村 真之介 (意見交換用 Wiki サイト作成)
皆様の意見交換用の Wiki サイト (http://oss-db.jp/oss-db-wiki/) を開設しました。本教科書に関するご意見・ご質問・誤植等の報告はぜひ Wiki サイトにお寄せください。本教科書が皆様の学習のお役に立てれば幸いです。

教科書開発に協力いただいた皆さん (第 1 版)
本教科書は、オープンソースソフトウェア開発の手法を取り入れ、何回かのフェーストゥフェースのミーティングと、メーリングリストを使ったコミュニケーションで構成の企画および原稿のレビューなどを行いました。

- 案浦浩二
- 石井達夫（SRA OSS, Inc. 日本支社）
- 伊津野匡
- 上田和章（ネットプラン松山）
- 岡田賢治（株式会社ネットマイスター）
- 加藤剛（株式会社アークシステム）
- 綱川貴之（富士通株式会社）
- 遠山洋平（株式会社びぎねっと）
- 永安悟史（アップタイム・テクノロジーズ合同会社）
- 濱田大助（日本文理大学）
- 早坂一王（株式会社クレスコ）
- 古橋勇作（日本 HP）
- 本間裕一（株式会社エヌサイト）
- 三谷篤（SRA 西日本）
- 吉田貴紀（SRA OSS, Inc. 日本支社）
- 吉田敏和（NTT コムウェア株式会社）

著作権

使用に関する権利

本教材の使用に関するお問合せ先
特定非営利活動法人エルピーアイジャパン（LPI-Japan）事務局

〒100-0011 東京都千代田区内幸町 2-1-1　飯野ビルディング 9 階

TEL：03-6205-7025　　E-Mail：info@lpi.or.jp

この教科書の目的

本書の目的は、データベースの経験の無い技術者を対象に、基本的なデータベースの操作方法について実習を通して学習することにあります。SQL 文を使ってデータベースを操作したり、データベースの作成や管理についての基礎を学習します。OSS-DB エンジニアのスキルを認定する OSS-DB 技術者認定試験のための教育および学習にも役立てていただけます。

想定している実習環境

本書での実習環境として、以下の環境を構築しています。

データベース

本書では、PostgreSQL バージョン 10.1 を利用します。バージョンに依存する内容はほとんど無いため、その他のバージョンでも学習は可能ですが、表示など一部異なる場合があります。インストールは RPM パッケージから行っていますが、独自にソースコードからインストールしてもかまいません。

OS

本書では、CentOS バージョン 7 を利用します。PostgreSQL のバージョン 10.1 が動作すれば、その他のディストリビューションでもかまいません。

マシンの構成と HDD

マシンの構成は、市販されている一般的な構成の PC を想定しています。その PC に Linux と PostgreSQL をインストールします。よって、HDD の内容は完全にクリアされます。そのため HDD の中を消してよい PC を用意するか、HDD の中身をあらかじめバックアップしておく必要があります。

ネットワーク

利用する PC は、ネットワークで接続されており、インターネットにも接続できることを前提としています。接続されていない場合には、別の方法で PostgreSQL のパッケージをダウンロードおよびコピーしてインストールを行ってください。

仮想環境

専用の PC 環境が用意できない場合は仮想環境を使う方法もあります。仮想環境とは Windows や Mac OS X 上で PC をソフトウェアで実現し、稼働している OS 上にあたかも別のマシンが動作しているかの様に振る舞うもので、たとえば VMware 社の VMware Workstation(Windows) や VMware Fusion(Mac OS X)、Parallels 社の Parallels Desktop(Mac OS X) や VirtualBox、Linux KVM 等があげられます。

クラウド環境

クラウド上に用意した環境を使う方法もあります。CentOS、またはその他の Linux が動作する環境に PostgreSQL がインストールしてあれば、本書の内容を学習できます。

その他の情報源

- オープンソースデータベース技術者認定試験
  - https://oss-db.jp/
- 日本 PostgreSQL ユーザ会（JPUG）
  - https://www.postgresql.jp/
- PostgreSQL のマニュアル (JPUG のサイトから「日本語ドキュメント」)
  - https://www.postgresql.jp/document/
- メーリングリスト（pgsql-jp）
  - https://www.postgresql.jp/npo/mailinglist
- Let's PostgreSQL

1　SQL によるデータベースの操作基礎編

データベースの操作には SQL を使用します。この章では SQL を利用したデータベースの操作の基礎を学びます。すでに作成されているデータベースに対して SQL を使ってデータの検索や更新などを行ってみましょう。

1.1　データベース利用の基本パターン

データベースの役目は、利用者からの要求に応じて様々なデータを管理することです。それらを分類すると、以下のようなパターンに分けることができます。

- **表を作成する（CREATE TABLE）**
  データベースにデータを保管するには、「表（TABLE）」を作成する必要があります。表の項目名や、データの種類（文字や数値など）を指定する必要があります。
- **データを挿入する（INSERT）**
  データベースにデータを保管することを「挿入 (INSERT)」と呼びます。データの種類に合わせたデータを挿入する必要があります。
- **データを検索する（SELECT）**
  データベースからデータを取り出すことを「検索 (SELECT)」と呼びます。条件を指定して一部を取り出したり、複数の表から組み合わせでデータを取り出すなど、様々な検索が行えるのがデータベースのメリットです。
- **データを更新する（UPDATE）**
  データベースのデータを修正することを「更新 (UPDATE)」と呼びます。すべてのデータを一括で修正したり、条件を指定して一部のデータだけを修正したりできます。
- **データを削除する（DELETE）**
  データベースからデータを取り除くことを「削除 (DELETE)」と呼びます。条件を指定して、取り除きたいデータを絞り込みます。

1.2　psql ツールの利用

PostgreSQL に対して SQL を実行してデータベースを操作するには、psql ツール（以下 psql）を利用します。

注意：ここからの作業は第 11 章で説明しているセットアップが行われていることを前提としています。うまく作業が行えない場合には、セットアップが正しく行われていることを確認してください。

1.2.1　psql でデータベースに接続する

psql は Linux 上で実行できるコマンドとして提供されているので、利用するには Linux にログインする必要があります。

■1.2.1.1　ユーザー postgres で作業可能にする

psql を実行するには、シェルで作業しているユーザーをユーザー postgres に切り替えます。

ユーザー postgres は、Linux に PostgreSQL をパッケージでインストールした時に自動的に作成されているユーザーです。管理者ユーザー root でログインした後、su コマンドでユーザー postgres に切り替えます。

su コマンドでユーザーを切り替えるには、以下のように su コマンドに-（ハイフン）をつけて実行するようにしてください。

```
[root@localhost ~]# su - postgres
[postgres@localhost ~]$
```

環境によっては、root ユーザーでログインすることができず、代わりに sudo 権限を有する管理者ユーザーが提供されている場合があります。

以下の例では sudo 権限を持つユーザー centos でログインし、postgres ユーザーに変更する例です。

```
[centos@localhost ~]$ sudo su - postgres
[postgres@localhost ~]$
```

本書では OS の管理者ユーザー root で操作可能なものとして例示しますが、自身の環境に応じて適宜読み替えるようにしてください。

■1.2.1.2　データベース一覧を表示する

psql に-l オプションをつけて実行します。psql は PostgreSQL に接続し、現在作成されているデータベースの一覧を表示します。

```
[postgres@localhost ~]$ psql -l
                              List of databases
   Name    |  Owner   | Encoding | Collate | Ctype |   Access privileges
-----------+----------+----------+---------+-------+-----------------------
 ossdb     | postgres | UTF8     | C       | C     |
 postgres  | postgres | UTF8     | C       | C     |
 template0 | postgres | UTF8     | C       | C     | =c/postgres          +
           |          |          |         |       | postgres=CTc/postgres
 template1 | postgres | UTF8     | C       | C     | =c/postgres          +
           |          |          |         |       | postgres=CTc/postgres
(4 rows)
```

■1.2.1.3　データベースに接続する

psql を利用してデータベースに接続します。PostgreSQL は同時に複数のデータベースを管理できますが、psql ではそのうちの 1 つを選んで接続して操作します。

以下のように、psql の引数として接続したいデータベースの名前を指定して実行します。

```
[postgres@localhost ~]$ psql ossdb
psql (10.1)
Type "help" for help.

ossdb=#
```

接続に成功すると、プロンプトが表示されて SQL 文などの実行命令を受け付ける状態になります。接続できない場合には、PostgreSQL が正しく実行されているか、ユーザー postgres で psql を実行しているかを確認してください。

1.2.2　psql のヘルプを表示する

psql のヘルプを表示します。help と入力します。

```
ossdb=# help
You are using psql, the command-line interface to PostgreSQL.
Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help with psql commands
       \g or terminate with semicolon to execute query
       \q to quit
```

1.2.3　メタコマンド

psql は SQL を受け付けて、PostgreSQL に対して SQL を実行する他に、\（バックスラッシュ）で始まるメタコマンドを受け付けます。メタコマンドは、ヘルプを表示したり、データベースに対する操作を行ったり、様々な種類のコマンドが用意されています。

■1.2.3.1　psql メタコマンド\h

利用できる SQL のヘルプが確認できます。

```
ossdb=# \h
Available help:
  ABORT
  ALTER AGGREGATE
  ALTER COLLATION
  ALTER CONVERSION
  ALTER DATABASE
(以下略)
```

■1.2.3.2　psql メタコマンド\h SQL コマンド

psql メタコマンド\h の引数に SQL コマンドを指定すると、その SQL コマンドのヘルプが確認できます。SQL コマンドの指定は大文字でも小文字でも構いません。

```
ossdb=# \h DELETE
Command:     DELETE
Description: delete rows of a table
Syntax:
[ WITH [ RECURSIVE ] with_query [, ...] ]
DELETE FROM [ ONLY ] table_name [ * ] [ [ AS ] alias ]
    [ USING using_list ]
    [ WHERE condition | WHERE CURRENT OF cursor_name ]
    [ RETURNING * | output_expression [ [ AS ] output_name ] [, ...] ]
```

■1.2.3.3　psql メタコマンド\?

psql メタコマンド\?で、利用できる psql メタコマンドのヘルプが確認できます。

```
ossdb=# \?
General
  \copyright             show PostgreSQL usage and distribution terms
  \crosstabview [COLUMNS] execute query and display results in crosstab
  \errverbose            show most recent error message at maximum verbosity
  \g [FILE] or ;         execute query (and send results to file or |pipe)
(以下略)
```

1.2.4　psql を終了する

psql を終了するには、psql メタコマンドの\q を入力します。

```
ossdb=# \q
[postgres@localhost ~]$
```

1.2.5　表の確認

作成されている表を確認するには psql メタコマンド\d を利用します。

```
ossdb=# \d
          List of relations
 Schema |   Name   | Type  |  Owner
--------+----------+-------+----------
 public | customer | table | postgres
 public | orders   | table | postgres
 public | prod     | table | postgres
(3 rows)
```

1.2.6　表定義の確認

表がどのような項目を持っているのかを確認するには psql メタコマンド\d に確認したい表名を付けて実行します。

```
ossdb=# \d customer
                Table "public.customer"
    Column     |  Type   | Collation | Nullable | Default
---------------+---------+-----------+----------+---------
 customer_id   | integer |           |          |
 customer_name | text    |           |          |

ossdb=# \d orders
                          Table "public.orders"
   Column    |            Type             | Collation | Nullable | Default
-------------+-----------------------------+-----------+----------+---------
 order_id    | integer                     |           |          |
 order_date  | timestamp without time zone |           |          |
 customer_id | integer                     |           |          |
 prod_id     | integer                     |           |          |
 qty         | integer                     |           |          |

ossdb=# \d prod
                Table "public.prod"
  Column   |  Type   | Collation | Nullable | Default
-----------+---------+-----------+----------+---------
 prod_id   | integer |           |          |
 prod_name | text    |           |          |
 price     | numeric |           |          |
```

1.2.7　表とテーブル、リレーション

本書ではデータの格納先を「表」と記述していますが、psql の実行結果には「リレーション」や「テーブル」と表記されています。基本的に表とテーブルは同じものと考えて構いません。リレーションは、本来的な意味ではデータの集合を指していますが、リレーショナルデータベースではリレーションは表形式で表されますので、リレーションと表もほぼ同義と考えて良いでしょう。同様に、「行」と「レコード」や「タプル」、「列」と「カラム」も同義語になります。

1.3　pgAdmin4（GUI ツール）

psql に並ぶ標準ツールとして pgAdmin4 も用意されています。pgAdmin4 は GUI 管理ツールで、Windows などの GUI 環境では psql よりも簡単に使うことができます。（Linux 版の pgAdmin4 も提供されています。Linux の構築時点で GUI 利用を前提に作成している場合は有用でしょう。）

- pgAdmin プロジェクト
  - https://www.pgadmin.org/

Windows 版の PostgreSQL をインストールすると、一緒に pgAdmin4 もインストールされますし、上記 URL から pgAdmin4 を個別にインストールして外部のデータベースサーバーに接続して使用することもできます。

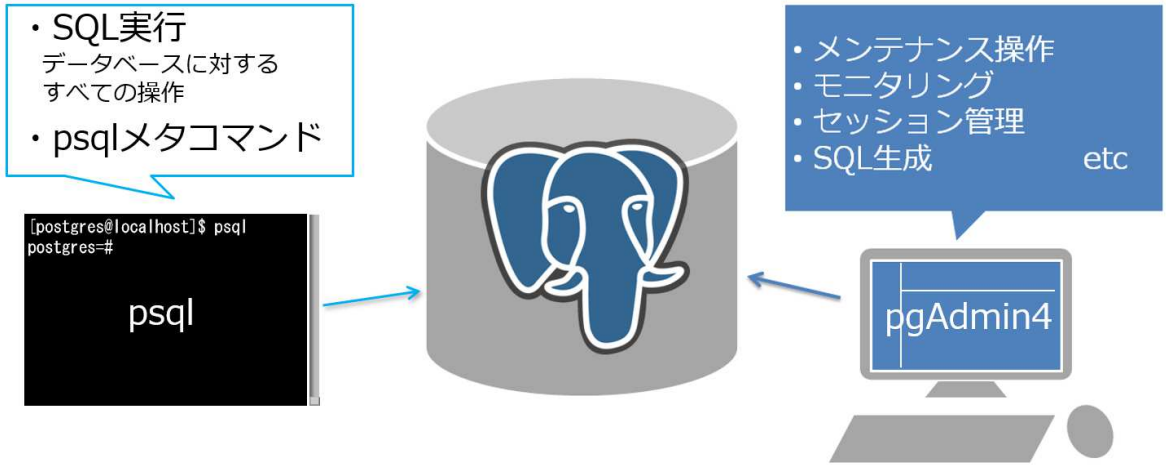


図 1　psql と pgAdmin4

複数の管理対象データベースを登録しておいて、操作対象を切り替えながらデータベースの稼動状態、表や索引の一覧やそれぞれの定義情報、状態を確認できます。

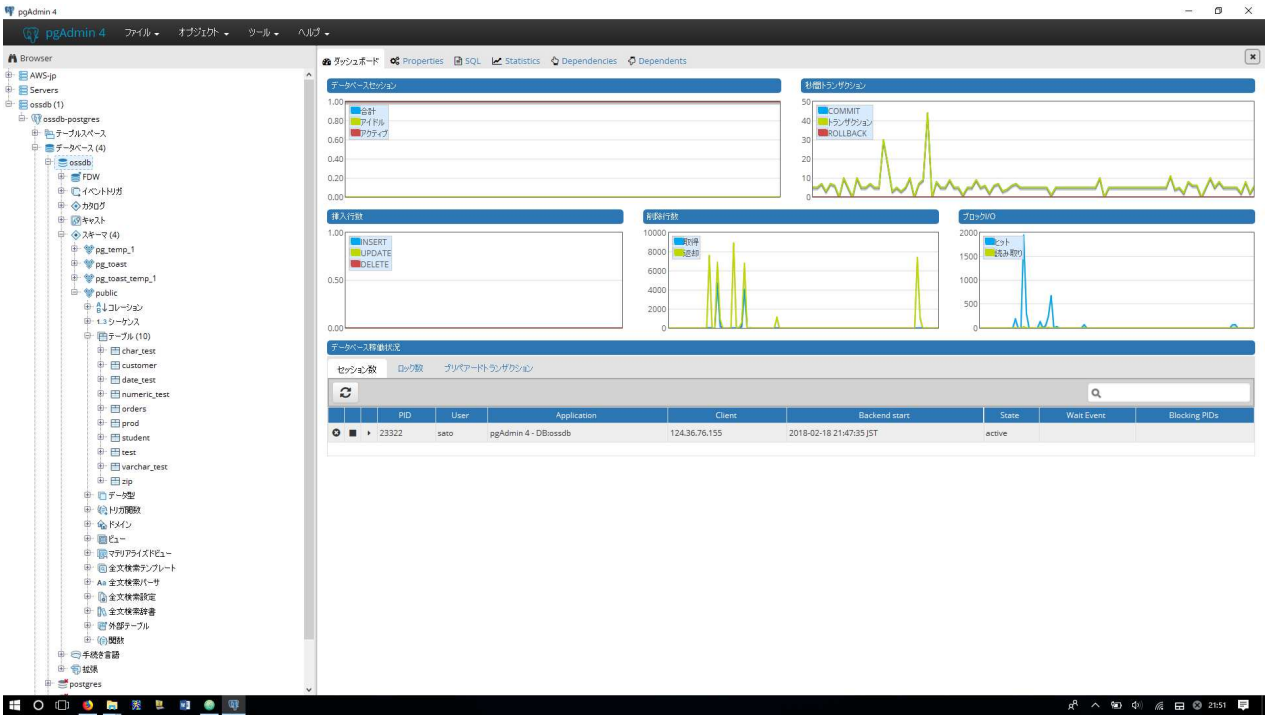


図 2　pgAdmin4 を用いた状態確認

代表的な機能としては、データベースを選択した状態で上部メニューより「ツール」→「クエリツール」を選択すると SQL を実行したり、よく使う SQL を保存しておくことができます。

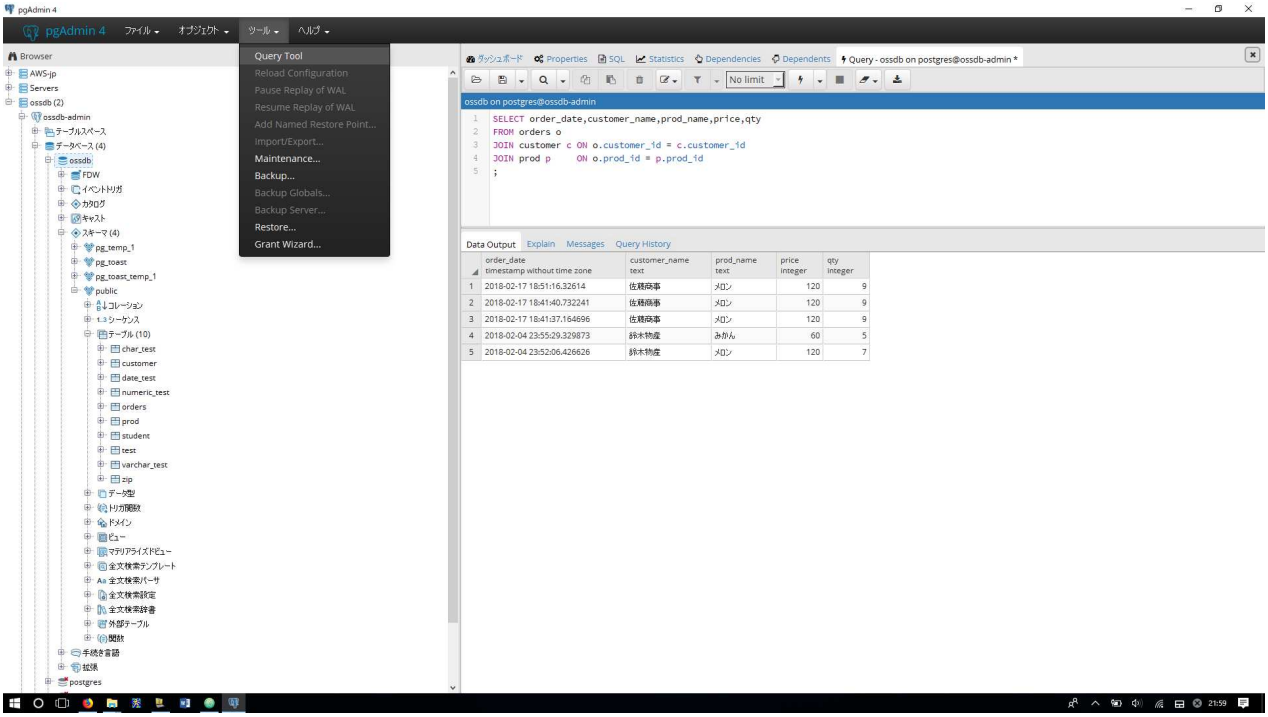


図 3　pgAdmin4 から SQL 実行

データベース名を右クリックして表示されるメニューから「Backup」したり、バックアップを使った「Restore」で簡単にデータをファイルに出力したり、戻すこともできます。テーブル名を右クリックすると「先頭 n 件のデータを取得」したり、テーブルのメンテナンスが行えるといった形で、GUI ならではの直感的な操作が可能です。ぜひいろいろ試してみてください。

1.4　SQL の実行方法

psql は、psql メタコマンドと SQL コマンドの 2 つを受け付けて実行します。psql メタコマンドは必ず\から始まるので区別されます。psql メタコマンド以外は、PostgreSQL データベースに対する SQL コマンドとして実行されます。psql では、改行やスペースは単に SQL コマンドの整形のために用いるもので、SQL 構文としての意味を持ちません。SQL コマンドの実行をサーバーに指示する;（セミコロン）または psql メタコマンド\g を文の末尾に記述することで、文全体をひとつの命令としてサーバーに送信し、処理が行われます。SQL コマンドの実行は、行末に;（セミコロン）をつけて入力を行うか、psql メタコマンド\g を実行します。改行しても複数行に渡って入力を受け付けるので、SQL 文の入力が終わったら忘れずに;（セミコロン）か、psql メタコマンド\g を実行してください。

1.4.1　複数行入力のプロンプト表示

psql で複数行入力をした時に表示されるプロンプトは、通常時と複数行入力中の 2 行目以降で異なります。

■1.4.1.1　psql のプロンプト

表記	動作モード
データベース名=#	通常のプロンプト
データベース名-#	2 行目以降のプロンプト

プロンプトに何を表示するかは、psql の内部変数 PROMPT1、PROMPT2 で定義されています。たとえば 2 行目以降のプロンプトに何も表示したくない場合、psql メタコマンドの`\unset` で PROMPT2 の変数値を解除します。

```
ossdb=# 1st line
ossdb-# 2nd line
ossdb-# 3rd line;
ERROR:  syntax error at or near "1"
行 1: 1st line
      ^
ossdb=# \unset PROMPT2
ossdb=# 1st line
2nd line
3rd line;
ERROR:  syntax error at or near "1"
行 1: 1st line
      ^
```

本書では、複数行に渡る SQL を電子版からのコピー＆ペーストで簡単に実行できるように実行例を掲載しています。実習時には 2 行目以降のプロンプトが出ているのが正しい表示です。

1.4.2　テキストファイルからの読み込み実行

psql はテキストファイルから読み込んだ内容を実行する機能があります。同じ処理を何度も繰り返し実行したい場合には、あらかじめメタコマンドや SQL コマンドをファイルに記述して、psql に読み込ませて実行できます。

■1.4.2.1　ファイルを読み込ませる場合の psql の構文

```
$ psql -f ファイル名 [データベース名] [ユーザー名]
```

以下の例では、メタコマンド\d を記述したファイル test.sql を psql に読み込ませて実行しています。

```
[postgres@localhost ~]$ cat test.sql
\d
[postgres@localhost ~]$ psql -f test.sql ossdb
          List of relations
 Schema |   Name   | Type  |  Owner
--------+----------+-------+----------
 public | customer | table | postgres
 public | orders   | table | postgres
 public | prod     | table | postgres
(3 rows)
```

1.5　データの検索（SELECT）

データの検索はデータベース利用の一番の基本です。データベースは様々な種類のデータを表形式で保管しているので、そのデータを必要な形で取り出すのがデータの検索です。データの検索には SQL の SELECT 文を使用します。

■1.5.0.1　SELECT 文の基本構文

```
SELECT [DISTINCT] * | SELECT 項目リスト
    FROM 表名 [,…]
    [WHERE 検索条件式]
    [GROUP BY グループ化式]
    [HAVING 検索条件式]
    [ORDER BY 並べ替え式]
```

1.5.1　全件全項目検索

データベースの表データを全件、全項目で取り出すのが全件全項目検索です。*(アスタリスク) を指定することで、対象となる検索表の全項目を検索します。

■1.5.1.1　全件全項目検索の SELECT 文

```
SELECT * FROM 表
```

以下の例では、customer 表、prod 表、orders 表の全件全項目検索を行っています。

```
ossdb=# SELECT * FROM customer;
 customer_id | customer_name
-------------+---------------
           1 | 佐藤商事
           2 | 鈴木物産
           3 | 高橋商店
(3 rows)

ossdb=# SELECT * FROM prod;
 prod_id | prod_name | price
---------+-----------+-------
       1 | みかん    |    50
       2 | りんご    |    70
       3 | メロン    |   100
(3 rows)

ossdb=# SELECT * FROM orders;
 order_id |         order_date         | customer_id | prod_id | qty
----------+----------------------------+-------------+---------+-----
        1 | 2018-01-22 12:34:51.510398 |           1 |       1 |  10
        2 | 2018-01-22 12:34:58.875188 |           2 |       2 |   5
        3 | 2018-01-22 12:35:07.440391 |           3 |       3 |   8
        4 | 2018-01-22 12:35:16.847541 |           2 |       1 |   3
        5 | 2018-01-22 12:35:26.164922 |           3 |       2 |   4
(5 rows)
```

1.5.2　SELECT 項目リスト

SELECT 文で検索したい列名をカンマ区切りで並べて指定します。

■1.5.2.1　SELECT 項目リストを使った SELECT 文

```
SELECT 列名 [, 列名...] FROM 表
```

以下の例では、prod 表から prod_name 列、price 列のみ検索しています。

```
ossdb=# SELECT prod_name,price FROM prod;
 prod_name | price
-----------+-------
 みかん    |    50
 りんご    |    70
```

```
 メロン     |   100
(3 rows)
```

1.5.3　WHERE 句による絞り込み検索

検索で取り出すデータ行を条件で絞り込むには、WHERE 句を使用します。

■1.5.3.1　WHERE 句の構文

```
WHERE 列名 条件式 条件値
```

■1.5.3.2　主な条件式

条件式	意味
=	等しい
<>	等しくない
>	よりも大きい
<	よりも小さい（未満）
>=	以上
<=	以下
BETWEEN	範囲指定
LIKE	部分一致

それぞれの条件式を使ってどのような結果が得られるか見てみましょう。

■1.5.3.3　等号、不等号

- **等しい (=)、等しくない (<>)**
  ある列の値が指定した条件値と等しい、あるいは等しくないデータを取り出します。条件値を文字列として指定する場合には'(シングルクォート) で括ります。以下の例では、customer 表から customer_id 列の値が 2 の行データを検索します。

```
ossdb=# SELECT * FROM customer WHERE customer_id = 2;
 customer_id | customer_name
-------------+---------------
           2 | 鈴木物産
(1 row)
```

以下の例では、customer 表から customer_id 列の値が 2 以外の行データを検索します。

```
ossdb=# SELECT * FROM customer WHERE customer_id <> 2;
 customer_id | customer_name
-------------+---------------
           1 | 佐藤商事
           3 | 高橋商店
(2 rows)
```

以下の例では、customer 表から customer_name 列の値が「佐藤商事」の行データを検索します。文字列データを指定する場合、'（シングルクォート）で前後を括ります。

```
ossdb=# SELECT * FROM customer WHERE customer_name = '佐藤商事';
 customer_id | customer_name
-------------+---------------
           1 | 佐藤商事
(1 row)
```

- **よりも大きい (>)、よりも小さい（未満）(<)**
  ある列の値が指定した条件値よりも大きい、あるいは小さい（未満）データを取り出します。以下の例では、prod 表から price 列の値が 70 よりも大きい行データを検索します。price 列の値が 70 のりんごは含まれません。

```
ossdb=# SELECT * FROM prod WHERE price > 70;
 prod_id | prod_name | price
---------+-----------+-------
       3 | メロン    |   100
(1 row)
```

- **以上 (>=)、以下 (<=)**
  ある列の値が指定した条件値以上、あるいは以下のデータを取り出します。以下の例では、prod 表から price 列の値が 70 以上の行データを検索します。price 列の値が 70 のりんごも含まれます。

```
ossdb=# SELECT * FROM prod WHERE price >= 70;
 prod_id | prod_name | price
---------+-----------+-------
       2 | りんご    |    70
       3 | メロン    |   100
(2 rows)
```

■1.5.3.4　範囲検索

- **A 以上 B 以下**
  条件式に 2 つの値を指定し、その間の範囲に該当するデータを取り出します。以下の例では、prod 表から price 列の値が 10 以上 80 以下の行データを検索します。price 列が 50 のみかん、70 のりんごが含まれ、範囲外であるメロンは含まれません。

```
ossdb=# SELECT * FROM prod WHERE price BETWEEN 10 AND 80;
 prod_id | prod_name | price
---------+-----------+-------
       1 | みかん    |    50
       2 | りんご    |    70
(2 rows)
```

この結果は、不等号を使った「以上、以下」を組み合わせた場合と同じです。

```
ossdb=# SELECT * FROM prod WHERE price >= 10 AND price <= 80;
 prod_id | prod_name | price
---------+-----------+-------
       1 | みかん    |    50
       2 | りんご    |    70
(2 rows)
```

■1.5.3.5　部分一致検索

部分一致検索には LIKE 演算子を用います。文字列の一部に指定した文字列を含むデータを取り出します。

- **指定した値がデータのどこかに一致（中間一致検索）**
  文字列中に「ん」を含むものを検索します。検索したい文字を % で囲みました。文字列なので’（**シングルクォート**）で囲む点はこれまでの検索と同じです。以下の例では、文字列中に「ん」を含む「みかん」「りんご」が該当します。（「メロン」は「ン」を含みますが、コンピューターは「ん」と「ン」を通常は区別します。）

```
ossdb=# SELECT * FROM prod WHERE prod_name LIKE '%ん%';
 prod_id | prod_name | price
---------+-----------+-------
       1 | みかん    |    50
       2 | りんご    |    70
(2 rows)
```

- **前方一致検索、後方一致検索**
  文字列の最後に「ん」を含むものを検索します。先ほど同様 % と組み合わせますが、文字列の最後であることを表すために、「ん」の後の % は指定しません。このような検索を後方一致検索と呼びます。以下の例では、文字列の最後に「ん」を含む「みかん」が該当します。

```
ossdb=# SELECT * FROM prod WHERE prod_name LIKE '%ん';
 prod_id | prod_name | price
---------+-----------+-------
       1 | みかん    |    50
(1 row)
```

前方一致検索の場合は、検索したい文字列を先頭にします。（例：’り %’ とすると「りんご」が該当）

- **該当する文字位置を指定する場合**
  指定した値が文字列中の何文字目にあるか指定する検索です。先ほどの % は複数文字を表しましたが、その代わりに 1 文字につきひとつの_（アンダースコア）を使います。以下の例では、3 文字目が「ん」であるものを検索するためにアンダースコアを 2 つ並べて「__ん」を条件にしており、結果として「みかん」が検索されています。アンダースコア 1 文字分では結果は得られません。

```
ossdb=# SELECT * FROM prod WHERE prod_name LIKE '__ん';
 prod_id | prod_name | price
---------+-----------+-------
       1 | みかん    |    50
(1 row)
ossdb=# SELECT * FROM prod WHERE prod_name LIKE '_ん';
 prod_id | prod_name | price
---------+-----------+-------
(0 rows)
```

- **指定した文字列を「含まない」検索**
  一致=に対して、不一致を表す<>があったように、LIKE に対してその文字列を含まないことを表す NOT LIKE が用意されています。以下の例では、NOT LIKE を使って「ん」を含まないものを検索します。「メロン」のみ該当します。

```
ossdb=# SELECT * FROM prod WHERE prod_name NOT LIKE '%ん%';
 prod_id | prod_name | price
---------+-----------+-------
       3 | メロン    |   100
(1 row)
```

同様に「前方（後方）不一致検索」も可能です。

```
ossdb=# SELECT * FROM prod WHERE prod_name NOT LIKE '%ん';
 prod_id | prod_name | price
---------+-----------+-------
       2 | りんご    |    70
```

```
      3 | メロン    |   100
(2 rows)
```

1.6　ORDER BY 句による並べ替え

検索結果を指定した順番に並べ替えるには、ORDER BY 句を使用します。リレーショナルデータベースではデータを集合として扱うため、並び順を指定しない限り結果の表示順を保証しません。ORDER BY 句で並べ替えを指定することで、期待した順番で行データを取り出すことができます。DESC を指定すると、降順で並べ替えが行われます。

■1.6.0.1　ORDER BY 句の構文

```
ORDER BY 列名 [DESC]
```

以下の例では、prod 表から price 列の値で昇順、降順で並べ替えをしています。

```
ossdb=# SELECT * FROM prod ORDER BY price;
 prod_id | prod_name | price
---------+-----------+-------
       1 | みかん    |    50
       2 | りんご    |    70
       3 | メロン    |   100
(3 rows)

ossdb=# SELECT * FROM prod ORDER BY price DESC;
 prod_id | prod_name | price
---------+-----------+-------
       3 | メロン    |   100
       2 | りんご    |    70
       1 | みかん    |    50
(3 rows)
```

1.7　表の結合

リレーショナルデータベースでは、表と表を結びつけてデータを取り出すことができます。表を結びつけることを「結合」と呼びます。

1.7.1　JOIN 句による結合

結合を行うには JOIN 句で結合したい列を指定します。指定された列の値を比較し、同じ値のデータを結合します。

■1.7.1.1　JOIN 〜 ON 構文

```
FROM 表 1
    JOIN 表 2 ON 表 1. 列 = 表 2. 列
```

単一の表に対する SELECT では、対象の表を FROM 句で指定してきましたが、結合では二つ目の表を指定するために JOIN 句のあとに表名を続けます。このとき表の指定とセットで使う ON 句も重要です。ON 句の後には結合条件を記述します。二つの表同士で関連する列を結合条件に指定し、この値が一致するデータが結果として取得されます。

結合の使い方を順を追ってみていきましょう。

- **単一の表を検索**
  以下の例では、結合を行う元となる表である orders 表の order_id 列、customer_id 列、prod_id 列、qty 列を SELECT

項目リストに指定した SELECT 文です。

```
ossdb=# SELECT order_id,customer_id,prod_id,qty FROM orders;
 order_id | customer_id | prod_id | qty
----------+-------------+---------+-----
        1 |           1 |       1 |  10
        2 |           2 |       2 |   5
        3 |           3 |       3 |   8
        4 |           2 |       1 |   3
        5 |           3 |       2 |   4
(5 rows)
--                    ↑         ↑
--「customer 表」「prod 表」から合致する値を取得する
```

この検索結果のうち、customer_id 列と prod_id 列はそれぞれ customer 表、prod 表から別の値を持ってきて置き換えることにします。

- **orders 表と customer 表を結合**
  FROM 句または JOIN 句で複数の表を指定した場合には、SELECT 項目リストや結合条件では「表名. 列名」と指定します。まずは orders 表と customer 表の 2 つの表を JOIN 句で結合します。ここでは先の結果と比較するため、order_id 列、customer_id 列と並べて、新たに customer 表から得られた customer_name 列の値を表示します。

```
ossdb=# SELECT orders.order_id,orders.customer_id,customer.customer_name
FROM orders
JOIN customer ON orders.customer_id = customer.customer_id;
 order_id | customer_id | customer_name
----------+-------------+---------------
        1 |           1 | 佐藤商事
        2 |           2 | 鈴木物産
        3 |           3 | 高橋商店
        4 |           2 | 鈴木物産
        5 |           3 | 高橋商店
(5 rows)
```

customer 表には以下のデータが格納されていましたので、customer_id が「1」のときは「佐藤商事」、「2」のときは「鈴木物産」というように、行ごとに関連する値が customer 表から検索されていることがわかります。

customer_id	customer_name
1	佐藤商事
2	鈴木物産
3	高橋商店

- **orders 表と prod 表を結合**
  同様に、orders 表と prod 表を JOIN 句で結合します。

```
ossdb=# SELECT orders.order_id,prod.prod_name
FROM orders
JOIN prod ON orders.prod_id = prod.prod_id;
 order_id | prod_name
----------+-----------
        1 | みかん
        2 | りんご
        3 | メロン
        4 | みかん
        5 | りんご
```

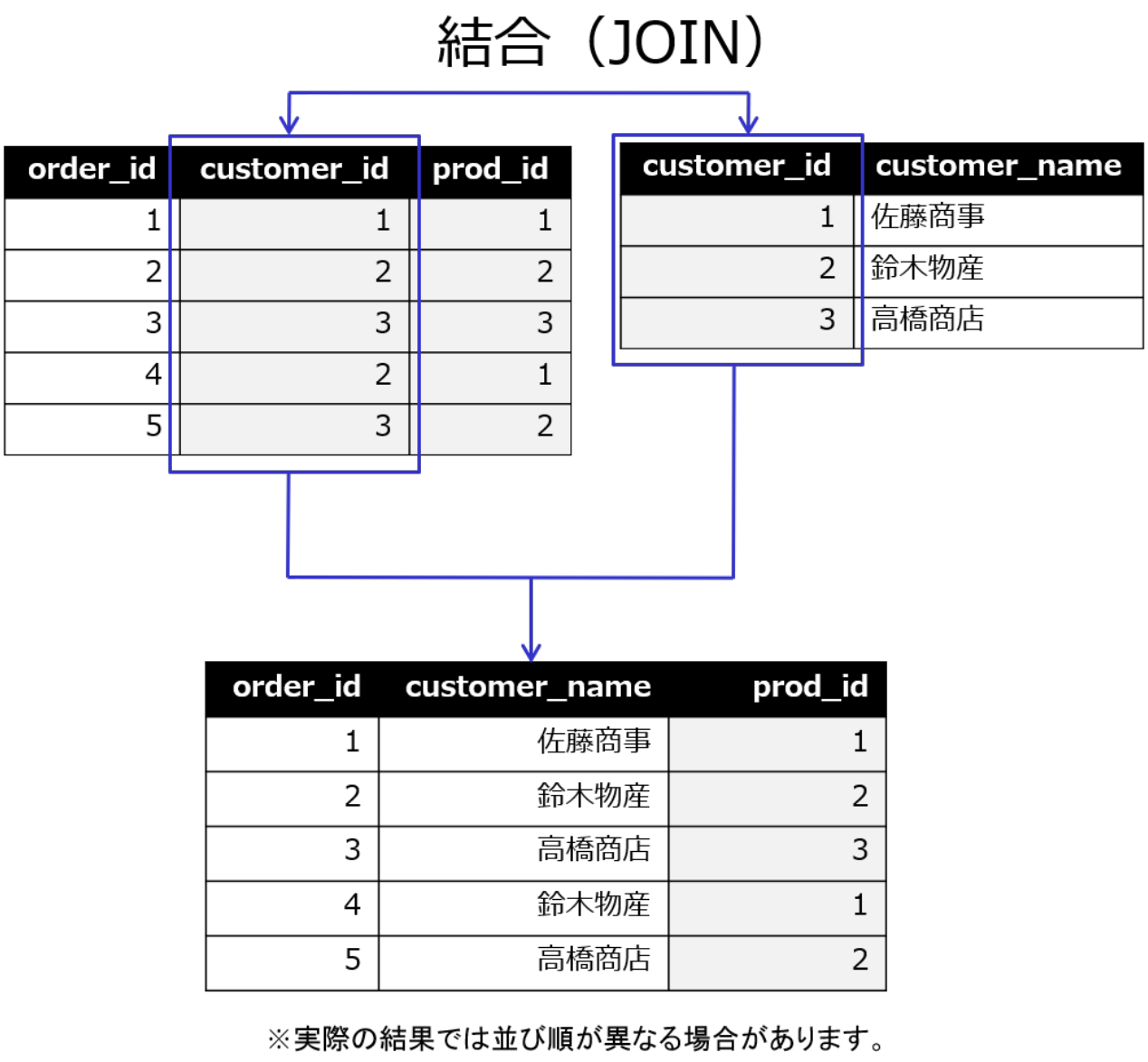


図 4　orders 表と customer 表の JOIN

```
(5 rows)
```

- **3 つの表を結合**
  この 2 つの JOIN 句を 1 つの SELECT 文にまとめます。最初に実行した SELECT 文は以下の通りでした。

```
SELECT order_id,customer_id,prod_id,qty
FROM orders
```

最初の SELECT 文と比較して、SELECT 項目リストの置き換わりと、JOIN 句が 2 つ並べて追加されている点に注目してください。

```
ossdb=# SELECT orders.order_id,customer.customer_name,prod.prod_name,orders.qty
FROM orders
JOIN customer ON orders.customer_id = customer.customer_id
JOIN prod     ON orders.prod_id = prod.prod_id;
 order_id | customer_name | prod_name | qty
----------+---------------+-----------+-----
        1 | 佐藤商事      | みかん    |  10
        2 | 鈴木物産      | りんご    |   5
```

```
        3 | 高橋商店       | メロン     |   8
        4 | 鈴木物産       | みかん     |   3
        5 | 高橋商店       | りんご     |   4
(5 rows)
```

customer_id を customer 表から取得した店名、prod_id を prod 表から取得した商品名に置き換えることができました。

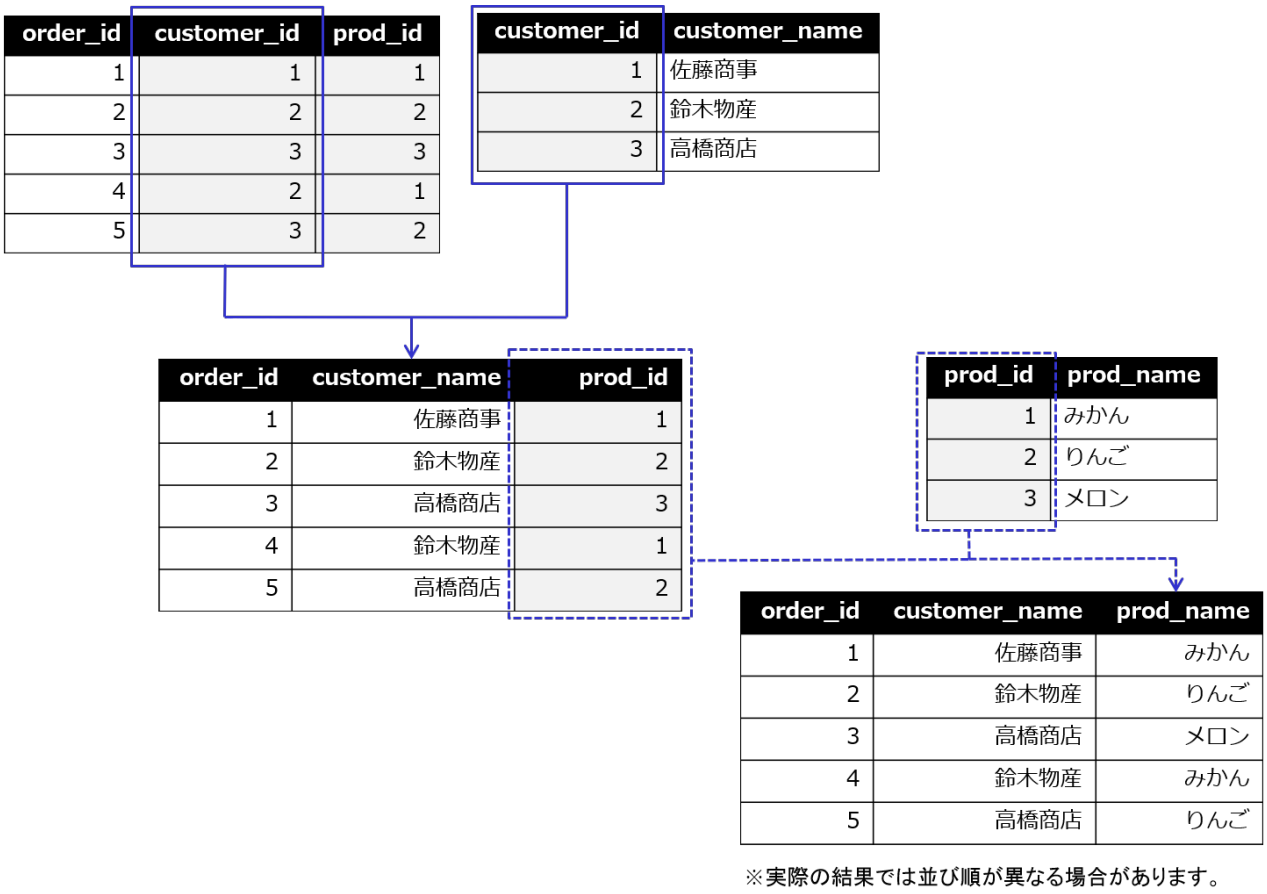


図 5　3 つの表の JOIN

1.7.2　表別名の利用

SQL 文の中で何度も使用する表名は、FROM 句、または JOIN 句で別名を指定できます。短い別名を指定すれば、SQL 文を短くして見やすくすることができます。

■1.7.2.1　表別名の指定方法

```
FROM 表名 別名
JOIN 表名 別名
```

前の JOIN 句による結合を行う SQL 文に表別名を適用すると、以下のようになります。

```
ossdb=# SELECT o.order_id,c.customer_name,p.prod_name,o.qty
FROM orders o
JOIN customer c ON o.customer_id = c.customer_id
JOIN prod p     ON o.prod_id = p.prod_id;
 order_id | customer_name | prod_name | qty
----------+---------------+-----------+-----
        1 | 佐藤商事       | みかん     |  10
```

```
       2 | 鈴木物産       | りんご     |   5
       3 | 高橋商店       | メロン     |   8
       4 | 鈴木物産       | みかん     |   3
       5 | 高橋商店       | りんご     |   4
(5 rows)
```

1.8　行データの入力（INSERT）

表に行データを入力するには、INSERT 文を使用します。

■1.8.0.1　INSERT 文の構文

```
INSERT INTO 表名（列名 [,…])
    VALUES（値 [,...])
```

値に文字列データを指定する場合には、'（シングルクォート）で括る必要があります。

```
ossdb=# INSERT INTO prod(prod_id,prod_name,price) VALUES (4,'バナナ',30);
INSERT 0 1
ossdb=# SELECT * FROM prod;
 prod_id | prod_name | price
---------+-----------+-------
       1 | みかん     |    50
       2 | りんご     |    70
       3 | メロン     |   100
       4 | バナナ     |    30
(4 rows)
```

1.9　データの更新（UPDATE）

行データを更新するには、UPDATE 文を使用します。

■1.9.0.1　UPDATE 文の構文

```
UPDATE 表名
    SET 列名 = 値
    WHERE 条件式
```

以下の例では、prod 表の prod_id 列の値が 4 の行データを、price 列の値を 40 に更新しています。

```
ossdb=# UPDATE prod SET price = 40 WHERE prod_id = 4;
UPDATE 1
ossdb=# SELECT * FROM prod;
 prod_id | prod_name | price
---------+-----------+-------
       1 | みかん     |    50
       2 | りんご     |    70
       3 | メロン     |   100
       4 | バナナ     |    40
(4 rows)
```

更新値は、既存の行データの値に対する演算で設定することもできます。以下の例では、prod 表の price 列の値を一律 10 増加させています（その後、元に戻しています）。

```
ossdb=# UPDATE prod SET price = price + 10;
UPDATE 4
ossdb=# SELECT * FROM prod;
 prod_id | prod_name | price
---------+-----------+-------
       1 | みかん    |    60
       2 | りんご    |    80
       3 | メロン    |   110
       4 | バナナ    |    50
(4 rows)

ossdb=# UPDATE prod SET price = price - 10;
UPDATE 4
```

1.10　行データの削除（DELETE）

行データを削除するには、DELETE 文を使用します。DELETE 文は WHERE 句で指定した条件式に適合した行データをすべて削除します。WHERE 句を指定しなかった場合には、指定された表の行データはすべて削除されます。

■1.10.0.1　DELETE 文の構文

```
DELETE FROM 表名
    WHERE 条件式
```

以下の例では、prod 表から prod_id 列の値が 4 の行データを削除しています。

```
ossdb=# DELETE FROM prod WHERE prod_id = 4;
DELETE 1
ossdb=# SELECT * FROM prod;
 prod_id | prod_name | price
---------+-----------+-------
       1 | みかん    |    50
       2 | りんご    |    70
       3 | メロン    |   100
(3 rows)
```

2　データ型

PostgreSQL では、データをいくつかのデータ型に分類して保管しています。これまで利用してきたサンプルデータベースでは、以下の 3 つのデータ型を使用しています。

- 数値データ型
- 文字列データ型
- 日付データ型

2.1　数値データ型

数値データ型は、いわゆる「数字」を保管するためのデータ型です。データベースでは四則演算をはじめ、様々な数値演算のための関数などが用意されているので、簡単に数値データを加工して取り扱うことができます。

2.1.1　integer 型

整数のデータ型です。-2147483648 から +2147483647 までの整数値を格納することができます。整数値のため、小数点以下の値は格納されません。小数点以下の値は四捨五入されます。

以下の例では、値を 30.4 で INSERT すると小数点以下は四捨五入されて 30 で格納されています。値を 30.5 に UPDATE すると四捨五入されて 31 で格納されています。

```
ossdb=# INSERT INTO prod(prod_id,prod_name,price) VALUES (4,'バナナ',30.4);
INSERT 0 1
ossdb=# SELECT * FROM prod WHERE prod_id = 4;
 prod_id | prod_name | price
---------+-----------+-------
       4 | バナナ    |    30
(1 row)

ossdb=# UPDATE prod SET price = 30.5 WHERE prod_id = 4;
UPDATE 1
ossdb=# SELECT * FROM prod WHERE prod_id = 4;
 prod_id | prod_name | price
---------+-----------+-------
       4 | バナナ    |    31
(1 row)
```

2.1.2　numeric 型

任意の精度の数値データ型です。小数点以下の値を含む数値を格納することができます。小数点より上は 131072 桁まで、小数点より下は 16383 桁までの桁数を指定できます。桁数を指定する場合、整数と小数点以下を合わせた桁数を「精度」、小数点以下の桁数を「位取り」と呼び、以下のように指定します。

```
numeric(精度, 位取り)
```

たとえば「numeric(6,2)」と指定すると、全体の桁数は 6 桁、小数点以下は 2 桁、整数部は 6-2 で 4 桁なので最大 9999.99 までの値を格納することができます。

以下の例では、numeric 型の id 列を持った numeric_test 表を作成しています。整数部は 4 桁なので、5 桁の値である 19999 を指定した INSERT 文はエラーとなっています。

```
ossdb=# CREATE TABLE numeric_test(
id numeric(6,2));
CREATE TABLE
```

```
ossdb=# \d numeric_test
              Table "public.numeric_test"
 Column |     Type     | Collation | Nullable | Default
--------+--------------+-----------+----------+---------
 id     | numeric(6,2) |           |          |

ossdb=# INSERT INTO numeric_test VALUES (9999.99);
INSERT 0 1
ossdb=# INSERT INTO numeric_test VALUES (19999.99);
ERROR:  numeric field overflow
DETAIL:  A field with precision 6, scale 2 must round to an absolute value less than 10^4.
```

2.1.3 その他の数値型

PostgreSQL は integer 型、numeric 型以外の数値型も備えています。これらはデータの性質に合わせたデータベースへの保管や、互換性の維持などの目的のために用意されています。

■2.1.3.1 数値型一覧表

データ型	サイズ	範囲
smallint	2 バイト	-32768 から +32767
integer	4 バイト	-2147483648 から +2147483647
bigint	8 バイト	-9223372036854775808 から +9223372036854775807
decimal	可変長	小数点より上は 131072 桁まで、小数点より下は 16383 桁まで
numeric	可変長	小数点より上は 131072 桁まで、小数点より下は 16383 桁まで
real	4 バイト	6 桁精度
double precision	8 バイト	15 桁精度
serial	4 バイト	1 から 2147483647
bigserial	8 バイト	1 から 9223372036854775807

2.2 文字列データ型

文字列データ型は、文字のデータを保管することができます。数値型のような演算に比べて大小比較や演算などを行うことはできませんが、LIKE 演算子のように部分一致検索などを行うことができます。

2.2.1 character varying 型 (varchar 型)

文字数に上限のある可変長の文字列型です。可変長ですから、文字数制限内であれば何文字の文字データでも構いません。上限を超える文字列を格納しようとするとエラーになります。

以下の例では、長さ 3 の varchar 型を持った varchar_test 表を作成しています。文字数の長さは半角、全角にかかわらず長さ 3（3 文字）までを許容します。

```
ossdb=# CREATE TABLE varchar_test(
varstring varchar(3));
CREATE TABLE
ossdb=# INSERT INTO varchar_test VALUES ('ABC');
INSERT 0 1
ossdb=# INSERT INTO varchar_test VALUES ('あいうえお');
ERROR:  value too long for type character varying(3)
ossdb=# INSERT INTO varchar_test VALUES ('あいう');
```

```
INSERT 0 1
ossdb=# INSERT INTO varchar_test VALUES ('AIUEO');
ERROR:  value too long for type character varying(3)
ossdb=# SELECT * FROM varchar_test;
 varstring
-----------
 ABC
 あいう
(2 rows)
```

2.2.2　character 型 (char 型)

文字数に上限のある固定長の文字列型です。固定長のため、足りない分の文字数は空白で埋められます。

以下の例では、長さ 3 の char 型を持った char_test 表を作成しています。一文字を格納すると末尾まで空白で埋められており、後方一致検索を使って空白を検索することで確認できます。

```
ossdb=# CREATE TABLE char_test(
string char(3));
CREATE TABLE
ossdb=# INSERT INTO char_test VALUES ('あ');
INSERT 0 1
ossdb=# SELECT * FROM char_test WHERE string LIKE 'あ ';
 string
--------
 あ
(1 row)
```

上限である 3 文字を超える文字列を格納しようとするとエラーになります。文字列数の長さは半角、全角にかかわらず長さ 3（3 文字）までです。

```
ossdb=# INSERT INTO char_test VALUES ('ABC');
INSERT 0 1
ossdb=# INSERT INTO char_test VALUES ('あいうえお');
ERROR:  value too long for type character(3)
ossdb=# INSERT INTO char_test VALUES ('あいう');
INSERT 0 1
ossdb=# INSERT INTO char_test VALUES ('AIUEO');
ERROR:  value too long for type character(3)
ossdb=# SELECT * FROM char_test;
 string
--------
 あ
 ABC
 あいう
(3 rows)
```

2.2.3　text 型

文字数に上限のない可変長の文字列型です。文字列長の指定が必要ないため便利ですが、ANSI SQL 標準には定義されていないデータ型です。

■2.2.3.1　文字列型一覧表

データ型	説明
character varying(n), varchar(n)	上限付き可変長
character(n), char(n)	空白で埋められた固定長
text	制限なし可変長

2.3　日付・時刻データ型

日付・時刻データ型は、日付だけを格納するデータ型、時刻だけを格納するデータ型、両方を同時に格納するデータ型の 3 つが使用できます。目的に応じて使い分けるとよいでしょう。なお、本項では日付・時刻データ型の紹介とし、日付・時刻データを検索する際の特有の記述については後述します。

■2.3.0.1　日付・時刻データ型一覧表

データ型	説明
date	日付のみ格納（時刻データは切り捨てられ 00:00:00 となる）
time	時刻のみ格納（日付データを持たず日付型への変換不可）
timestamp	日付と時刻を格納

以下の例では、date 型、time 型、timestamp 型の 3 つの列を持った date_test 表を作成しています。それぞれに現在時刻を挿入すると、各データ型で定められた「日付」「時刻」「日付と時刻」が格納されていることを確認します。（現在時刻を取得する now() 関数については後述します。）

```
ossdb=# CREATE TABLE date_test ( d_test  date,
                                 t_test  time,
                                 ts_test timestamp);
CREATE TABLE
ossdb=# \d date_test
                      Table "public.date_test"
 Column  |            Type             | Collation | Nullable | Default
---------+-----------------------------+-----------+----------+---------
 d_test  | date                        |           |          |
 t_test  | time without time zone      |           |          |
 ts_test | timestamp without time zone |           |          |

ossdb=# INSERT INTO date_test VALUES (now(),now(),now());
INSERT 0 1
ossdb=# SELECT * FROM date_test;
   d_test   |     t_test      |          ts_test
------------+-----------------+----------------------------
 2018-01-23 | 12:34:56.526066 | 2018-01-23 12:34:56.526066
(1 row)
```

3 表

3.1 表の作成（CREATE TABLE）

前の章では例文中で簡単な表を作成し、その表に対して SQL 文を実行していました。実際のデータベースでは利用者が目的に応じて表を作成します。表の作成方法を見ていましょう。

3.1.1 表を作成する

表の作成は、CREATE TABLE 文を使用します。

■3.1.1.1 CREATE TABLE 文の構文

```
CREATE TABLE 表名
    (列名 データ型 [NULL|NOT NULL]
        | [UNIQUE]
        | [PRIMARY KEY (列名 [,...])]
        | [REFERNCES 外部参照表名 (参照列名)]
    [,...])
```

表を作成するには、列の名前と、その列にどのようなデータが入るのかを決めなければいけません。ここでは、以下のような 3 つの列を持つテーブルを作成します。

■3.1.1.2 staff 表の定義

	列名	データ型
社員番号	id	integer 型
氏名	name	text 型
誕生日	birthday	date 型

これを SQL 文にすると、以下のようになります。作成ができたら、表の一覧に入っていることと、表の定義を確認しておきましょう。

```
ossdb=# CREATE TABLE staff
                    (id       integer,
                     name     text,
                     birthday date);
CREATE TABLE

ossdb=# \d
            List of relations
 Schema |     Name     | Type  |  Owner
--------+--------------+-------+----------
 public | char_test    | table | postgres
 public | customer     | table | postgres
 public | date_test    | table | postgres
 public | numeric_test | table | postgres
 public | orders       | table | postgres
 public | prod         | table | postgres
 public | staff        | table | postgres
```

```
 public | varchar_test | table | postgres
(8 rows)

ossdb=# \d staff
               Table "public.staff"
  Column  |  Type   | Collation | Nullable | Default
----------+---------+-----------+----------+---------
 id       | integer |           |          |
 name     | text    |           |          |
 birthday | date    |           |          |
```

3.1.2 staff 表にデータを格納する

INSERT 文を使って staff 表にデータを格納してみましょう。

```
ossdb=# INSERT INTO staff (id,name,birthday) VALUES (1,'宮原徹','1972-01-09');
INSERT 0 1
ossdb=# SELECT * FROM staff;
 id |  name  |  birthday
----+--------+------------
  1 | 宮原徹 | 1972-01-09
(1 row)
```

3.2 表定義の修正（ALTER TABLE）

表を作成した後で表定義を修正するには ALTER TABLE 文を使用します。

3.2.1 ALTER TABLE による変更

ALTER TABLE 文では表そのものの定義や動作の修正に加え、表内の列定義の修正も可能で、その分指定の方法が複雑になっています。

メタコマンド\h で構文のヘルプを確認してみましょう。ここでは詳細を覚える必要はありませんが、変更対象の表を指定し、さらに「新しい列を追加する（=ADD COLUMN）」のような action を指定することをヘルプから読み取ってみてください。

```
ossdb=# \h ALTER TABLE
Command:     ALTER TABLE
Description: change the definition of a table
Syntax:
ALTER TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
    action [, ... ]

(略)

where action is one of:

    ADD [ COLUMN ] [ IF NOT EXISTS ] column_name data_type [ COLLATE collation ] [ column_constraint [ ... ] ]
    DROP [ COLUMN ] [ IF EXISTS ] column_name [ RESTRICT | CASCADE ]
    ALTER [ COLUMN ] column_name [ SET DATA ] TYPE data_type [ COLLATE collation ] [ USING expression ]

(略)
```

ここでは新たに列を追加する ALTER TABLE 文を試してみましょう。staff 表に所属部署コードを表す dept_cd 列を追加し、その列に UPDATE 文で値を格納しています。

```
ossdb=# ALTER TABLE staff ADD COLUMN dept_cd integer;
ALTER TABLE
ossdb=# \d staff
                Table "public.staff"
  Column   |  Type   | Collation | Nullable | Default
-----------+---------+-----------+----------+---------
 id        | integer |           |          |
 name      | text    |           |          |
 birthday  | date    |           |          |
 dept_cd   | integer |           |          |

 ossdb=# UPDATE staff SET dept_cd = 1 WHERE id = 1;
 UPDATE 1
 ossdb=# SELECT * FROM staff;
  id |  name   |  birthday  | dept_cd
 ----+---------+------------+---------
   1 | 宮原徹 | 1972-01-09 |       1
 (1 row)
```

3.2.2 表定義の修正は原則として行わない

ALTER TABLE 文で修正できる内容は様々ですが、すでにデータが格納されている状態で安易に列の定義を修正することは望ましくありません。少なくとも以下のような設計の面や、作業による影響を考慮して実施方法を検討しましょう。

- **正しい設計になっているか**
  例えば、上記では staff 表に dept_cd 列を追加しましたが、所属部門を staff 表で管理することは理想的な対応だったでしょうか。`SELECT * FROM staff;` とすれば所属情報込みでスタッフを一覧表示できることは確かに便利かもしれません。しかし、1 名のスタッフが複数部門に所属するケースは起こりえないでしょうか。
  このような場合は、新たに所属部門表を作成し「01 番のスタッフは A という組織に所属する」という事実を表すようにします。「01 番のスタッフは同時に C という組織に所属する」場合はもう一行追加すればよいのです。

■3.2.2.1 所属部門表の例

スタッフ ID	部門コード	意味
01	A	スタッフ 01 は A という組織に所属
02	B	スタッフ 02 は B という組織に所属
03	C	スタッフ 03 は C という組織に所属
01	C	スタッフ 01 は C という組織に所属（同時に A と C に所属していることがわかる）

- **表定義の変更作業そのものがどんな影響を及ぼすか**
  仮に、スタッフ一覧に部門コードを持つことが適切であったとして、変更作業そのものがシステム全体に悪影響を及ぼす可能性があります。
  小規模なスタッフ表であれば問題は少ないかもしれませんが、数百万人のデータを収めたウェブサービスの会員表である場合はどうでしょうか。利用者がログインするたびに参照されているような場合、表の変更作業中は誰もログインできなくなり社会的影響にまで及ぶことを考慮します。（これは極端な例で、実際のデータベース製品の実装では変更中の参照は許されていることが多いです。しかし、データの更新などはブロックされてしまいますので無視できるものではありません。）
  この場合もやはり上記のような所属部門表を新規に作成することで、利用者の操作を妨げることなく必要なデータは保持することができます。

- **短期の対策と中長期の対策**

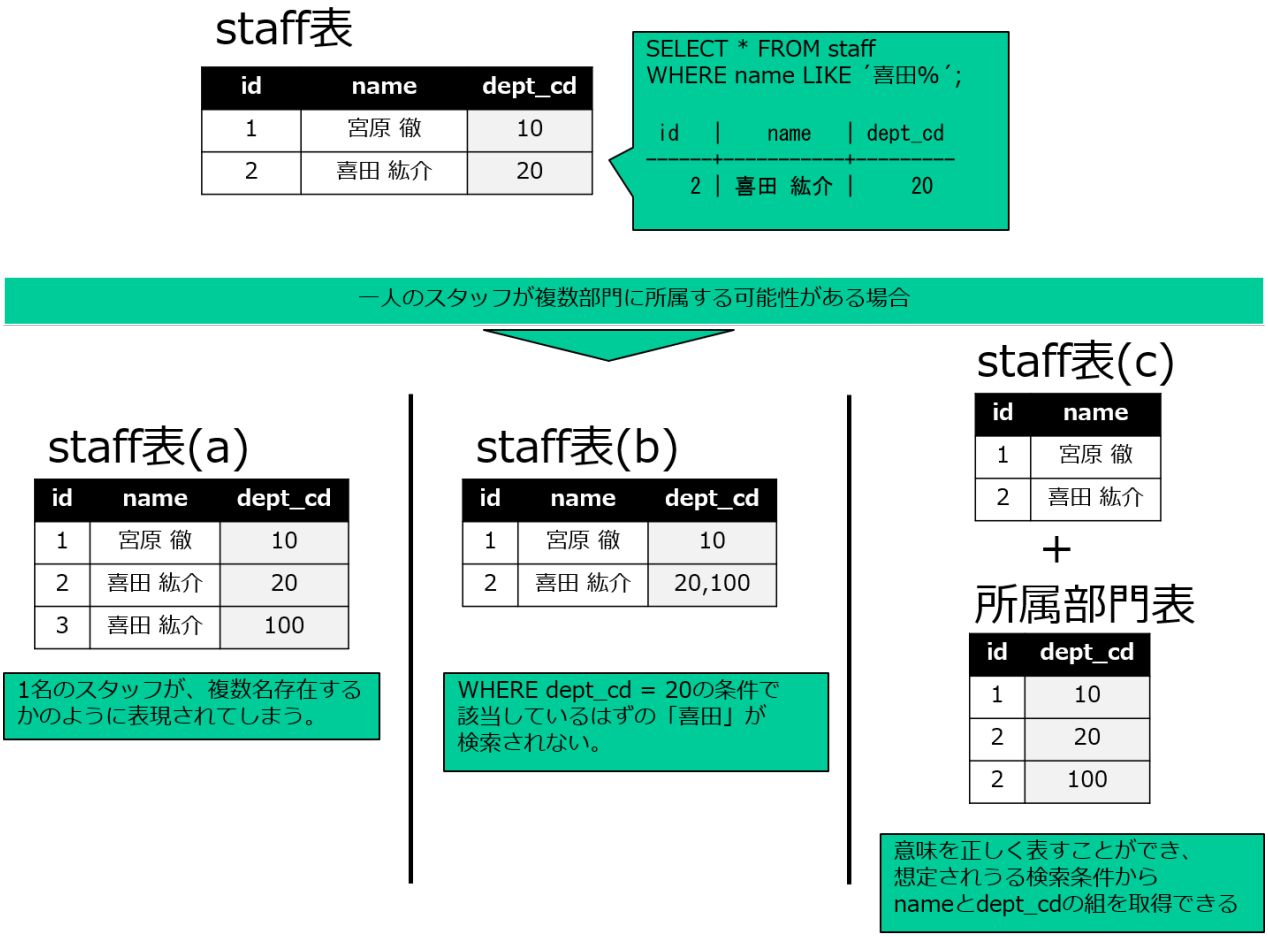


図 6　ALTER TABLE は行うべきか

さしあたって必要なデータを格納するために、新規に所属部門表を作成したとします。しかし、やはり設計上はスタッフ一覧に部門コードを持つことが最適という場合もあるでしょう。
大規模メンテナンスのためのサービス休止期間を利用者にアナウンスし、理想的な表に変更することは長期的な対策としては有用です。

ある操作が他の SQL やプログラムにどのような影響を与えるかは、いかなる操作であっても考慮が必要です。その中でも特に表のメンテナンスは影響範囲が大きくなりやすいですので十分に注意しましょう。

3.3　表の削除

3.3.1　DROP TABLE 文による表の削除

表を削除するには、DROP TABLE 文を使用します。表を削除すると、表に格納されているデータも一緒に削除されて元に戻すことができません。

```
ossdb=# \d
            List of relations
 Schema |     Name      | Type  |  Owner
--------+---------------+-------+----------
 public | char_test     | table | postgres
 public | customer      | table | postgres
 public | date_test     | table | postgres
 public | numeric_test  | table | postgres
 public | orders        | table | postgres
 public | prod          | table | postgres
 public | staff         | table | postgres
```

```
 public | varchar_test | table | postgres
(8 rows)

ossdb=# DROP TABLE staff;
DROP TABLE
ossdb=# \d
            List of relations
 Schema |     Name     | Type  |  Owner
--------+--------------+-------+----------
 public | char_test    | table | postgres
 public | customer     | table | postgres
 public | date_test    | table | postgres
 public | numeric_test | table | postgres
 public | orders       | table | postgres
 public | prod         | table | postgres
 public | varchar_test | table | postgres
(7 rows)
```

3.3.2 DROP TABLE、DELETE、TRUNCATE の利用

データを削除する操作には、表を定義ごと削除する DROP TABLE の他にも、WHERE 条件に該当した行だけを削除する DELETE、表のデータのみ全件削除する TRUNCATE があります。条件に該当する特定の行のデータだけを削除したい場合には DELETE 文を使用しますが、DELETE 文でのデータ削除は対象となるデータの件数が多いと時間がかかることがあります。例えば時系列に沿って蓄積されるデータのうち、保管期限を過ぎたものを一括削除するようなケースでは、表を月別などに分割しておき、月単位で TRAUNCATE することも考えます。

DROP TABLE 文は表と表データだけでなく、関連する索引やビューなど、その他のものも併せて削除してしまうため、再作成する場合はこれらも再定義する必要に時間がかかるなどの問題が発生することがあります。行データだけを一括で削除する場合には TRUNCATE 文を使用します。

■3.3.2.1 TRUNCATE 文の構文

```
TRAUNCATE 表名
```

以下の例では、char_test 表のすべての行データをすべて TRUNCATE 文で削除しています。

```
ossdb=# SELECT * FROM char_test;
 string
--------
 あ
 ABC
 あいう
(3 rows)

ossdb=# TRUNCATE char_test;
TRUNCATE TABLE
ossdb=# SELECT * FROM char_test;
 string
--------
(0 rows)
```

3.4 行データのセーブ・ロード

COPY 文を使用すると、行データをファイルにセーブしたり、ファイルからロードすることができます。FORMAT 句で csv を指定することで、CSV 形式のファイルをセーブ、ロードができます。

注意：COPY は DB サーバー上のファイルに直接データを書き出す操作で、PostgreSQL のスーパーユーザーでのみ実行することができます。似た操作ではクライアント端末側にデータを出力する psql メタコマンド\copy や、バックアップの章で解説する pg_dump などが利用できますので、目的に応じて使い分けるようにしてください。

3.4.1 行データのセーブ

COPY TO 文で行データをファイルにセーブできます。

■3.4.1.1 COPY 文によるデータのセーブ

```
COPY 表名 TO ファイル（FORMAT 形式）
```

以下の例は、customer 表のデータを CSV 形式でファイルにセーブしています。\!メタコマンドは Linux のシェルコマンドを実行しています。

```
ossdb=# COPY customer TO '/home/postgres/customer.csv' (FORMAT csv);
COPY 3
ossdb=# \! ls /home/postgres
customer.csv
ossdb=# \! cat /home/postgres/customer.csv
1,佐藤商事
2,鈴木物産
3,高橋商店
```

3.4.2 CSV ファイルのロード

COPY FROM 文でファイルから行データをロードできます。

■3.4.2.1 COPY 文によるデータのロード

```
COPY 表名 FROM ファイル（FORMAT 形式）
```

以下の例は、customer 表のデータを CSV 形式のファイルからロードしています。

```
ossdb=# DELETE FROM customer;
DELETE 3
ossdb=# SELECT * FROM customer;
 customer_id | customer_name
-------------+---------------
(0 rows)

ossdb=# COPY customer FROM '/home/postgres/customer.csv' (FORMAT csv);
COPY 3
ossdb=# SELECT * FROM customer;
 customer_id | customer_name
-------------+---------------
           1 | 佐藤商事
           2 | 鈴木物産
```

```
          3 | 高橋商店
(3 rows)
```

3.4.3 \copy メタコマンド

psql は COPY 文と同様の動作をする\copy メタコマンドが使用できます。異なるのは以下の点です。

- COPY 文は SQL であり、DB サーバーが実行する。出力ファイルは DB サーバー内に作成される。
- psql メタコマンド\copy はクライアントからのデータ取得操作であり、出力ファイルはクライアント端末内に作成される。
- ファイル指定が絶対指定のほか、psql 実行時のクライアント端末のカレントディレクトリからの相対指定でも可能
- 形式の指定が with csv

具体的な使用例はこの後の演習で解説します。

4　基礎編 演習

これまでの章で、SQL を使ったデータベースの基礎について学びました。2 つの演習問題で、学習した内容を確認してみましょう。

4.1　演習 1：データ操作

4.1.1　データ操作演習

1. すべての商品の価格を 10% アップします
2. 価格が 100 以上の商品の価格を元に戻します
3. prod 表のデータをファイルにコピーします
4. prod 表を削除します
5. prod 表を再度作成します
6. データをファイルからコピーします

表の定義はあらかじめ確認しておきましょう。また、巻末の付録にも表を作成するための CREATE TABLE 文の例がありますので、参考にしてください。

4.1.2　解答例

- **演習 1-1**：すべての商品の価格を 10% アップします

- 商品表の価格列を指定して UPDATE

```
ossdb=# \d prod
                Table "public.prod"
  Column   |  Type   | Collation | Nullable | Default
-----------+---------+-----------+----------+---------
 prod_id   | integer |           |          |
 prod_name | text    |           |          |
 price     | integer |           |          |

ossdb=# SELECT * FROM prod;
 prod_id | prod_name | price
---------+-----------+-------
       1 | みかん    |    50
       2 | りんご    |    70
       3 | メロン    |   100
       4 | バナナ    |    31
(4 rows)
ossdb=# UPDATE prod SET price = price * 1.1;
UPDATE 4
ossdb=# SELECT * FROM prod;
 prod_id | prod_name | price
---------+-----------+-------
       1 | みかん    |    55
       2 | りんご    |    77
       3 | メロン    |   110
       4 | バナナ    |    34
(4 rows)
```

- **演習 1-2**：価格が 100 以上の商品の価格を元に戻します

- 価格が 100 以上の商品を指定して UPDATE

```
ossdb=# UPDATE prod SET price = price/1.1 WHERE price >= 100;
UPDATE 1
ossdb=# SELECT * FROM prod;
 prod_id | prod_name | price
---------+-----------+-------
       1 | みかん    |    55
       2 | りんご    |    77
       4 | バナナ    |    34
       3 | メロン    |   100
(4 rows)
```

- **演習 1-3**：prod 表のデータをファイルにコピーします
- COPY コマンドでデータをファイルにセーブ

```
ossdb=# COPY prod TO '/home/postgres/prod.csv' (FORMAT csv);
COPY 4
ossdb=# \! ls -l prod.csv
-rw-r--r--. 1 postgres postgres 61  1月 29 02:01 prod.csv
```

- **演習 1-4**：prod 表を削除します
- 表の削除には DROP TABLE を使用

```
ossdb=# DROP TABLE prod;
DROP TABLE
```

- **演習 1-5**：表を再度作成します
- 表の作成時は列ごとに格納するデータに合わせた型を指定する
- ID のような整数には integer 型、文字には text 型、計算に用いる数値は numeric 型

```
ossdb=# CREATE TABLE prod ( prod_id     integer,
                            prod_name   text,
                            price       numeric  );
CREATE TABLE
```

- **演習 1-6**：データをファイルからコピーします
- セーブ時と同様、COPY コマンドを使用

```
ossdb=# COPY prod FROM '/home/postgres/prod.csv' (FORMAT csv);
COPY 4
ossdb=# SELECT * FROM prod;
 prod_id | prod_name | price
---------+-----------+-------
       1 | みかん    |    55
       2 | りんご    |    77
```

```
      4 | バナナ     |    34
      3 | メロン     |   100
(4 rows)
```

4.2 演習 2：郵便番号データベース

郵便番号データベースを設計してみましょう。郵便番号のデータは CSV 形式で公開されています。この郵便番号データを格納するデータベースを設計し、実際にデータを格納してみましょう。

4.2.1 郵便番号データのダウンロード

郵便番号データは以下の Web ページからダウンロードできます。

- **郵便番号データダウンロード Web ページの利用**
  データは ZIP 形式で配布されています。「住所の郵便番号（CSV 形式）」-「読み仮名データの促音・拗音を小書きで表記するもの」をクリックして ZIP 形式でダウンロードしてください。ここでは「全国一括」のデータを用います。
- http://www.post.japanpost.jp/zipcode/download.html
- **郵便番号データのダウンロード**
  以下の例は、wget コマンドを使って郵便番号 CSV データをダウンロードして、unzip コマンドで解凍しています。

```
[postgres@localhost ~]$ wget http://www.post.japanpost.jp/zipcode/dl/kogaki/zip/ken_all.zip
--2018-01-29 01:14:02--  http://www.post.japanpost.jp/zipcode/dl/kogaki/zip/ken_all.zip
www.post.japanpost.jp (www.post.japanpost.jp) を DNS に問いあわせています... 43.253.37.203
www.post.japanpost.jp (www.post.japanpost.jp)|43.253.37.203|:80 に接続しています... 接続しました。
HTTP による接続要求を送信しました、応答を待っています... 200 OK
長さ: 1686409 (1.6M) [application/zip]
`ken_all.zip' に保存中

100%[=======================================================================================>] 1,686,409   10.4MB/s 時間

2018-01-29 01:14:02 (10.4 MB/s) - `ken_all.zip' へ保存完了 [1686409/1686409]

[postgres@localhost ~]$ ls
customer.csv  ken_all.zip  test.sql
[postgres@localhost ~]$ unzip ken_all.zip
Archive:  ken_all.zip
  inflating: KEN_ALL.CSV
[postgres@localhost ~]$ ls -l KEN_ALL.CSV
-rw-rw-r--. 1 postgres postgres 12288638 12 月 22 14:22 KEN_ALL.CSV
```

4.2.2 郵便番号データベース表の作成

郵便番号データを格納するための表をデータベースに作成します。

- **郵便番号データ項目**
  データの項目は以下の通りです。

内容	備考
全国地方公共団体コード (JIS X0401、X0402)	半角数字
(旧) 郵便番号 (5 桁)	半角数字
郵便番号 (7 桁)	半角数字
都道府県名	半角カタカナ (コード順に掲載)
市区町村名	半角カタカナ (コード順に掲載)

内容	備考
町域名	半角カタカナ (五十音順に掲載)
都道府県名	漢字 (コード順に掲載)
市区町村名	漢字 (コード順に掲載)
町域名	漢字 (五十音順に掲載)
一町域が二以上の郵便番号で表される場合の表示	
小字毎に番地が起番されている町域の表示	
丁目を有する町域の場合の表示	
一つの郵便番号で二以上の町域を表す場合の表示	
更新の表示	
変更理由	

- **郵便番号データ用の表定義**
  以下の例は、文字列データを char 型および text 型で定義した表作成のための CREATE TABLE 文です。

```
ossdb=# CREATE TABLE zip (
                        lgcode    char(5),
                        oldzip    char(5),
                        newzip    char(7),
                        prefkana  text,
                        citykana  text,
                        areakana  text,
                        pref      text,
                        city      text,
                        area      text,
                        largearea integer,
                        koaza     integer,
                        choume    integer,
                        smallarea integer,
                        change    integer,
                        reason    integer
                        );
CREATE TABLE
```

- **郵便番号データサンプル**
  データは以下のような内容です。

```
[postgres@localhost ~]$ head -5 KEN_ALL_UTF8.CSV
01101,"060  ","0600000","ホツカイドウ","サツポロシチユウオウク","イカニケイサイガナイバアイ","北海道","札幌市中央区","以下
01101,"064  ","0640941","ホツカイドウ","サツポロシチユウオウク","アサヒガオカ","北海道","札幌市中央区","旭ケ丘",0,0,1,0,
01101,"060  ","0600041","ホツカイドウ","サツポロシチユウオウク","オオドオリヒガシ","北海道","札幌市中央区","大通東",0,0,1
01101,"060  ","0600042","ホツカイドウ","サツポロシチユウオウク","オオドオリニシ（1-19 チヨウメ）","北海道","札幌市中央区","
01101,"064  ","0640820","ホツカイドウ","サツポロシチユウオウク","オオドオリニシ（20-28 チヨウメ）","北海道","札幌市中央区",
```

4.2.3　データのロードと文字コードについて

ダウンロードできる CSV データは、日本語部分がシフト JIS で作成されています。一方、現在使用しているデータベースは日本語を UTF-8 で格納するようにしているため、文字コードを UTF-8 に揃える必要があります。

シフト JIS のデータを UTF-8 に変換するには、psql で\encoding メタコマンドを使用する方法と、Linux のコマンドで文字コード変換をする方法があります。

- **psql メタコマンドを使用する方法**
  \encoding メタコマンドを使用すると、psql が扱うデータの文字コードを変更できます。
  以下の例は、\encoding メタコマンドで psql が扱うデータの文字コードをシフト JIS に変更しています。データベー

スは UTF-8 で格納するので、シフト JIS から UTF-8 への文字コード変換が行われます。

```
ossdb=# \encoding SJIS
ossdb=# \copy zip from KEN_ALL.CSV with csv
COPY 124165
ossdb=# \encoding UTF-8
```

- **Linux のコマンドを使用する方法**
  Linux であれば nkf コマンドで文字コード変換が行えます。他に iconv コマンドも使用できますが、改行コードを別途 dos2unix コマンドで変換する必要があります。Linux 環境によっては nkf コマンドがインストールされていない場合がありますので、その場合には yum コマンドでインストールしてください。
- nkf コマンドで郵便番号データを UTF-8 に変換

```
[postgres@localhost ~]$ nkf -w KEN_ALL.CSV > KEN_ALL_UTF8.CSV
```

- iconv コマンドと dos2unix コマンドで郵便番号データを UTF-8 に変換

```
[postgres@localhost ~]$ iconv -f SHIFT-JIS -t UTF-8 KEN_ALL.CSV | dos2unix > KEN_ALL_UTF8.CSV
```

- 変換後、psql から\copy メタコマンドでロードします。

```
ossdb=# \copy zip from KEN_ALL_UTF8.CSV with csv
```

4.2.4 郵便番号データの確認

ロードされた郵便番号データを確認します。

以下の例では、現在使用されている郵便番号のデータが格納されている newzip 列で絞り込み検索を行っています。

```
ossdb=# SELECT * FROM zip WHERE newzip = '1500002';
 lgcode | oldzip | newzip  | prefkana | citykana | areakana |  pref  |  city  | area | largearea | koaza | choum
--------+--------+---------+----------+----------+----------+--------+--------+------+-----------+-------+------
 13113  | 150    | 1500002 | ﾄｳｷｮｳﾄ   | ｼﾌﾞﾔｸ    | ｼﾌﾞﾔ     | 東京都 | 渋谷区 | 渋谷 |         0 |     0 |      1 |
(1 row)
```

5　SQL によるデータベースの操作応用編

データベースの操作に使用する SQL 文には、他にも様々な文法が存在しています。この章では、特に多用する SQL 文の文法について解説します。

- 以降の実行例は、prod 表が以下の状態になっている前提で記載します。

```
ossdb=# DROP TABLE prod;
DROP TABLE
ossdb=# CREATE TABLE prod ( prod_id     integer,
                            prod_name   text,
                            price       numeric  );
CREATE TABLE
ossdb=# INSERT INTO prod(prod_id,prod_name,price) VALUES
 (1,'みかん',50),
 (2,'りんご',70),
 (3,'メロン',100),
 (4,'バナナ',30);
INSERT 0 4
ossdb=# SELECT * FROM prod;
 prod_id | prod_name | price
---------+-----------+-------
       1 | みかん    |    50
       2 | りんご    |    70
       3 | メロン    |   100
       4 | バナナ    |    30
(4 rows)
```

5.1　演算子と関数

5.1.1　演算子

- **AND/OR 演算子**
  SELECT 文などに指定する WHERE 句の条件で、複数の条件を設定したい場合には AND 演算子、OR 演算子が使用できます。
  AND 演算子は指定した条件が両方満たされる場合、OR 演算子は指定した条件のいずれかが満たされる場合に SQL 文の結果が表示されます。
  以下の例は、prod 表の price 列の値が 50 よりも大きく、100 よりも小さい行データのみを検索しています。

```
ossdb=# SELECT * FROM prod WHERE price > 50 AND price < 100;
 prod_id | prod_name | price
---------+-----------+-------
       2 | りんご    |    70
(1 row)
```

以下の例は、customer 表の customer_id 列が 1 または 2 の行データを検索しています。

```
ossdb=# SELECT * FROM customer WHERE customer_id = 1 OR customer_id = 2;
 customer_id | customer_name
-------------+---------------
           1 | 佐藤商事
           2 | 鈴木物産
(2 rows)
```

- **LIKE 演算子**

ある列の値が指定した条件に部分的に一致する行データを取り出します。
以下の例では、customer 表の customer_name 列が「鈴木」で始まる行データを検索しています。

```
ossdb=# SELECT * FROM customer WHERE customer_name LIKE '鈴木%';
 customer_id | customer_name
-------------+---------------
           2 | 鈴木物産
(1 row)
```

以下の例では、customer 表の customer_name 列に「商」が含まれる行データを検索しています。前後に % がついているので、値のどこに「商」があっても検索条件に一致します。

```
ossdb=# SELECT * FROM customer WHERE customer_name LIKE '%商%';
 customer_id | customer_name
-------------+---------------
           1 | 佐藤商事
           3 | 高橋商店
(2 rows)
```

なお LIKE 演算子は便利な反面、性能上検索速度が遅くなる場合があるので、注意して使う必要があるでしょう。

■5.1.1.1　LIKE 演算子で使用できるワイルドカード

ワイルドカード	内容
_	1 文字
%	0 文字以上の文字列

- **BETWEEN 演算子**
  ある列の値が指定した 2 つの条件値の範囲内にあるデータを取り出します。2 つの条件値は AND で指定します。条件値そのものも含まれるので「○以上、○以下」という条件であると考えればよいでしょう。
  以下の例では、prod 表の price 列が 50 から 70 の間の行データを検索しています。

```
ossdb=# SELECT * FROM prod WHERE price BETWEEN 50 AND 70;
 prod_id | prod_name | price
---------+-----------+-------
       1 | みかん    |    50
       2 | りんご    |    70
(2 rows)
```

5.1.2　関数

- **集約関数**
  集約関数を使用すると、データを SQL 文で集計することができ、データを一括で処理して 1 つの結果を返します。

■5.1.2.1　主な集約関数

関数	説明
count 関数	対象データの件数を返す
sum 関数	対象データ（数値）の合計値を返す
avg 関数	対象データ（数値）の平均値を返す
max 関数	対象データ（数値または文字）の最大値を返す 文字の場合、コード順で大小を評価

関数	説明
min 関数	対象データ（数値または文字）の最小値を返す 文字の場合、コード順で大小を評価

- **count 関数**
  count 関数はデータの行数を数える関数です。

```
ossdb=# SELECT count(order_id) FROM orders;
 count
 ------
     5
(1 row)
```

- **sum 関数**
  sum 関数は指定された列の合計を計算する関数です。

```
ossdb=# SELECT sum(qty) FROM orders;
 sum
 ----
  30
(1 row)
```

- **avg 関数**
  avg 関数は指定された列の平均を計算する関数です。

```
ossdb=# SELECT avg(qty) FROM orders;
        avg
 --------------------
 6.0000000000000000
(1 row)
```

- **max 関数**
  max 関数は指定された列の最大値を計算する関数です。

```
ossdb=# SELECT max(qty) FROM orders;
 max
 -----
  10
(1 row)
```

- **min 関数**
  min 関数は指定された列の最小値を計算する関数です。

```
ossdb=# SELECT min(qty) FROM orders;
 min
 -----
   3
(1 row)
```

参考：文字データの最大/最小
文字データの大小関係は、文字コードの並び順により評価されます。
以下の例では、「あ」～「お」を表す UTF-8 の文字コードを表示し、その順序通りに並べ替えが行われていることを確認しています。max 関数や min 関数の結果も並び順に従っていることがわかります。

```
ossdb=# TRUNCATE char_test;
TRUNCATE TABLE
ossdb=# INSERT INTO char_test VALUES ('あ'),('い'),('う'),('え'),('お');
INSERT 0 5
```

```
ossdb=# SELECT string,convert_to(string,'utf8') FROM char_test
ORDER BY string DESC;
 string | convert_to
--------+------------
 お     | \xe3818a
 え     | \xe38188
 う     | \xe38186
 い     | \xe38184
 あ     | \xe38182
(5 rows)
ossdb=# SELECT max(string),min(string) FROM char_test ;
 max  | min
------+------
 お   | あ
(1 row)
```

5.1.3　GROUP BY 句と集約関数の組み合わせ

GROUP BY 句を使うと、指定された列で行をグループ化し、それぞれのグループ毎に集約関数の計算を行うことができます。

以下の例は、orders 表の行データを prod_id 列の値毎にグループ化し、各グループ毎に集約関数で計算を行っています。

```
ossdb=# SELECT prod_id,count(qty),sum(qty),avg(qty),min(qty),max(qty)
FROM orders
GROUP BY prod_id;
 prod_id | count | sum |        avg         | min | max
---------+-------+-----+--------------------+-----+-----
       3 |     1 |   8 | 8.0000000000000000 |   8 |   8
       2 |     2 |   9 | 4.5000000000000000 |   4 |   5
       1 |     2 |  13 | 6.5000000000000000 |   3 |  10
(3 rows)
```

prod 表では、prod_id の 1 番は「みかん」でした。みかんが 2 回販売され、そのときに販売された数量 qty を集計しています。合計数量は 13 個、もっとも多く売れたときは一度に 10 個売れた、というようにみかんについての情報が得られます。同様に「りんご」は 9 個、「メロン」は 8 個というように、GROUP BY で指定した列の各データに対して集計が行われます。

5.1.4　HAVING 句

HAVING 句を使うと、グループ化した後のグループに対して条件による絞り込みを行うことができます。HAVING 句はグループに対しての絞り込みを行うため、比較対象は集約関数である必要があります。

以下の例では、orders 表の行データを prod_id 列の値毎にグループ化し、qty 列の合計値が 10 未満の結果のみ取得しています。

```
ossdb=# SELECT prod_id,sum(qty) FROM orders
GROUP BY prod_id;
 prod_id | sum
---------+-----
       3 |   8
       2 |   9
       1 |  13
(3 rows)

ossdb=# SELECT prod_id,sum(qty) FROM orders
GROUP BY prod_id
HAVING sum(qty) < 10;
```

```
 prod_id | sum
---------+-----
       3 |   8
       2 |   9
(2 rows)
```

5.1.5　WHERE 句、GROUP BY 句、HAVING 句の適用順序

集約関数を使った検索では、WHERE 句、GROUP BY 句、HAVING 句が以下の順序で適用されます。

1. WHERE 句による行に対する絞り込み
2. GROUP BY 句によるグループ化
3. HAVING 句によるグループに対する絞り込み

まず WHERE 句で検索対象になる行全体に対して絞り込みが行われます。この時点で除外された行は集約関数の対象にはなりません。次に GROUP BY 句によるグループ化が行われます。HAVING 句はこのグループに対して集約関数で計算を行い、指定された演算子に基づいて絞り込みを行います。

例えば、ある学習塾の “優秀な” 高校生向けプロモーションのために、現在の生徒情報から①「年齢別」「地域別」に生徒数を集計し生徒数の傾向を調べることを考えます。②今回は「高校生向け」という条件があるので、年齢を絞ることで集計対象になる件数集計対象の年齢を大きく減らし絞ります。③優秀な学生というプロモーションの趣旨をどう検索に落とし込むかは担当者の腕の見せ所かと思いますが、ここでは「進学校がある」のような地域性を考え「地域別にみたときのテストの平均点が 80 点を超える」とします。

SQL を考えると以下のようになるでしょう。

```
SELECT 年齢, 地域,count(*),avg(点数)
FROM   生徒一覧
WHERE  年齢 BETWEEN 16 AND 18 --------- ② 年齢の条件で分類に関係ないデータを除外
GROUP BY 年齢, 地域---------------------- ① 年齢、地域別に分類
HAVING avg(点数) >= 80;----------------- ③ 平均点が 80 点を超えるグループを抽出
```

ここで注目して欲しいのは②の年齢の条件と、③の平均点の条件です。実は、両条件を HAVING 句にまとめて書いても良いのです。得られる結果は全く同じです。

```
SELECT 年齢, 地域,count(*),avg(点数)
FROM   生徒一覧
GROUP BY 年齢, 地域
HAVING avg(点数) >= 80
AND    年齢  BETWEEN 16 AND 18;------- 年齢の条件を HAVING 句に記述
```

2 つの SQL の違いは、その検索パフォーマンスに現れます。ある「年齢・地域」グループの平均点が 80 点以上であることは、集計された結果をみて初めて適用できる条件ですので、HAVING 句に書くほかありません。前者は、集計対象とする 16 歳以上 18 歳以下という条件を WHERE 句で記述し、絞り込んだ行のみを集計しています。後者は年齢による絞り込みをせずに全件（おそらく小学生、中学生が名簿には多数含まれるでしょう）を対象に集計し、最後に HAVING 句で年齢を絞り込んでいます。無駄な集計作業に時間をかけているのです。

5.2　副問い合わせ

副問い合わせは、SELECT 文の中でさらに SELECT 文を実行する SQL の記述です。副問い合わせで検索された結果に基づいて主問い合わせを実行することができるので、動的な条件での検索が可能になります。副問い合わせは EXISTS 演算子、IN 演算子と組み合わせて実行できます。

5.2.1　EXISTS 演算子

EXISTS 演算子は、副問い合わせの結果が行を 1 行以上返した場合、主問い合わせが結果を返します。

EXISTS 演算子では、まず主問い合わせが実行され、返された行データの値が 1 行ずつ副問い合わせに渡されて実行されます。副問い合わせが行を 1 行以上返すと、主問い合わせが返した行データが最終的に結果として返されます。

以下の例では、主問い合わせで prod 表から 1 行ずつ行データを取り出し、副問い合わせで orders 表に対して prod_id 列に同じ値が存在するかを確認します。みかん、りんご、メロンは orders 表に行データが存在していますが、prod_id 列の値が 4 のバナナは orders 表に行データが存在していないため、主問い合わせの結果として返されません。

```
ossdb=# SELECT * FROM prod;
 prod_id | prod_name | price
---------+-----------+-------
       1 | みかん    |    50
       2 | りんご    |    70
       3 | メロン    |   100
       4 | バナナ    |    30
(4 rows)

ossdb=# SELECT * FROM orders;
 order_id |         order_date         | customer_id | prod_id | qty
----------+----------------------------+-------------+---------+-----
        1 | 2018-01-22 12:34:51.510398 |           1 |       1 |  10
        2 | 2018-01-22 12:34:58.875188 |           2 |       2 |   5
        3 | 2018-01-22 12:35:07.440391 |           3 |       3 |   8
        4 | 2018-01-22 12:35:16.847541 |           2 |       1 |   3
        5 | 2018-01-22 12:35:26.164922 |           3 |       2 |   4
(5 rows)

ossdb=# SELECT prod_id,prod_name FROM prod
WHERE EXISTS (SELECT * FROM orders WHERE orders.prod_id = prod.prod_id);
 prod_id | prod_name
---------+-----------
       1 | みかん
       2 | りんご
       3 | メロン
(3 rows)
```

5.2.2　IN 演算子

IN 演算子は、副問い合わせの結果を主問い合わせの WHERE 句の条件に対する値として実行できます。

以下の例では、orders 表の qty 列の値が 5 よりも大きい行データの prod_id 列の値から、prod 表の prod_id 列と prod_name 列の値を取得しています。

```
ossdb=# SELECT prod_id FROM orders WHERE qty > 5;
 prod_id
---------
       1
       3
(2 rows)

ossdb=# SELECT prod_id,prod_name FROM prod
WHERE prod_id IN (SELECT prod_id FROM orders WHERE qty > 5);
 prod_id | prod_name
---------+-----------
       1 | みかん
       3 | メロン
```

```
(2 rows)
```

5.3　日付・時刻型データの取り扱い

日付・時刻型データは数値型や文字列型と異なり、特別な取り扱い方が用意されています。

5.3.1　日付形式を確認・設定する

日付の形式は国や環境によって異なります。PostgreSQL がどのような日付形式に設定されているかを確認するには、SHOW DATESTYLE を実行します。

以下の例では、ISO 書式で MDY、つまり月日年のスタイルであることが分かります。西暦を 2 桁で表すとき「01-12-05」は何年の何月何日を表しますか？ 本例では、それを月-日-年で解釈するよう設定されていましたので「1 月―12 日―2005 年」と扱われています。

```
ossdb=# SHOW DATESTYLE;
 DateStyle
-----------
 ISO, MDY
(1 row)

ossdb=# select '01-12-05'::date;
    date
------------
 2005-01-12
(1 row)
```

なお、DATESTYLE を変更することもできます。日本人にはこちらのほうが馴染みがあるであろう、年―月―日の解釈に変更します。同じ「01-12-05」という文字列は、今度は「2001 年―12 月―5 日」と解釈されました。

```
ossdb=# set DATESTYLE to 'ISO, YMD';
SET
ossdb=# SHOW DATESTYLE;
 DateStyle
-----------
 ISO, YMD
(1 row)

ossdb=# select '01-12-05'::date;
    date
------------
 2001-12-05
(1 row)
```

5.3.2　現在時刻を取得する

- **now() 関数**
  now() 関数は、現在の日付と時刻を取得する関数です。
  以下の例では、SELECT 文で使用していますが、INSERT 文や UPDATE 文でも使用できます。

```
ossdb=# SELECT now();
              now
 -------------------------------
  2018-01-23 12:34:56.297675+09
 (1 row)
```

- **CURRENT_DATE/CURRENT_TIME/CURRENT_TIMESTAMP 関数**
  CURRENT_xxx 関数は、それぞれ現在の日付、時刻、日付と時刻を取得する関数です。

```
ossdb=# SELECT CURRENT_DATE;
 current_date
 --------------
 2018-01-23
 (1 row)

ossdb=# SELECT CURRENT_TIME;
    current_time
 --------------------
 12:34:56.496858+09
 (1 row)

ossdb=# SELECT CURRENT_TIMESTAMP;
       current_timestamp
 -------------------------------
 2018-01-23 12:34:56.677673+09
 (1 row)
```

5.3.3　文字列の入力値を日付型の列に格納する

上記は日付・時刻データを返す関数の結果として日付・時刻を取得しています。この結果をそのまま日付・時刻型の列に挿入することができます。

```
ossdb=# CREATE TABLE date_sample (ts_test timestamp);
CREATE TABLE
ossdb=# INSERT INTO date_sample VALUES (now());
INSERT 0 1
ossdb=# SELECT * FROM date_sample;
          ts_test
----------------------------
 2018-01-23 12:34:56.907503
(1 row)
```

次は「現在時刻」ではなく、「2018 年 1 月 23 日」のように文字列として指定された数の並びを日付型の列に格納する方法を紹介します。to_timestamp 関数を用います。

■5.3.3.1　to_timestamp 関数の使い方

```
to_timestamp('文字列','フォーマット')
```

ここで文字列は日付・時刻として解釈可能でなければならず（2018 年 1 月 23 日、23-JAN-2018 など）、その書式をフォーマット部分で指定します。フォーマットとは、文字列中の数字部分が日付・時刻に変換するときに、月・日・時・分・秒のどれを表すか指定するものです。

■5.3.3.2　フォーマットの指定

パターン	説明
YYYY	西暦 4 桁
YY	西暦 2 桁

パターン	説明
MM	数値で月を表す
MON	月名の省略形（JAN、FEB・・・など）
MONTH	月名（JANUARY、FEBRUARY・・・など）
DD	日付
HH	12 時間表記の時（AM/PM と一緒に指定）
HH24	24 時間表記の時
MI	分
SS	秒
MS	ミリ秒
US	マイクロ病

使い方の例を見てみましょう。日本語の「2018 年 1 月 23 日」はまさに文字列ですが、その文字列中には日付を表す「2018」「1」「23」が含まれています。この数値が日付のどの桁を表すかフォーマット指定します。

```
ossdb=# INSERT INTO date_sample VALUES (to_timestamp('2018 年 1 月 23 日','YYYY 年 MM 月 DD 日'));
INSERT 0 1
ossdb=# SELECT * FROM date_sample;
          ts_test
----------------------------
 2018-01-23 12:34:56.907503
 2018-01-23 00:00:00
(2 rows)
```

次の例では、さらに時・分・秒まで格納しています。「12 時」だけでは AM/PM が明確ではありませんが、フォーマットでは「HH24」とすることで 24 時間表記の 12 時（正午をまわったところ）であることを明示できます。

```
ossdb=# INSERT INTO date_sample
VALUES (to_timestamp('23-JAN-2018 12:34:56','DD-MON-YYYY HH24:MI:SS'));
INSERT 0 1
ossdb=# SELECT * FROM date_sample;
          ts_test
----------------------------
 2018-02-19 02:55:32.907503
 2018-01-23 00:00:00
 2018-01-23 12:34:56
(3 rows)
```

さらに 2 行のデータを追加します。今度は AM/PM を明記して 12 時間表記でフォーマットを指定しています。

```
ossdb=# INSERT INTO date_sample
 VALUES (to_timestamp('23-JAN-2018 AM12:34:00','DD-MON-YYYY AMHH:MI:SS')),
        (to_timestamp('23-JAN-2018 PM12:34:00','DD-MON-YYYY AMHH:MI:SS'));
INSERT 0 2
ossdb=# SELECT * FROM date_sample;
          ts_test
----------------------------
 2018-02-19 02:55:32.907503
 2018-01-23 00:00:00
 2018-01-23 12:34:56
 2018-01-23 00:34:00
 2018-01-23 12:34:00
(5 rows)
```

5.4　複雑な結合

JOIN 句による通常の結合の他にも、外部結合や自己結合といった結合が存在します。

5.4.1　外部結合

JOIN 句による通常の結合（等価結合）では、結合する表の両方に、結合条件に合うような行データが存在しないと検索結果には含まれません。
以下の例では、customer 表に新たな行データを追加し、通常の結合でどの店舗でどの商品がいくつ売れたかを取得しています。しかし、cusotmer 表に追加したばかりの藤原流通は、orders 表を見ても販売実績がありませんのでこの結果には藤原流通という行がまったく現れません。

```
ossdb=# INSERT INTO customer(customer_id,customer_name) VALUES (4,'藤原流通');
INSERT 0 1
ossdb=# SELECT * FROM customer;
 customer_id | customer_name
-------------+---------------
           1 | 佐藤商事
           2 | 鈴木物産
           3 | 高橋商店
           4 | 藤原流通
(4 rows)
ossdb=# SELECT c.customer_name,o.prod_id,o.qty
FROM customer c
JOIN orders   o ON c.customer_id = o.customer_id;
 customer_name | prod_id | qty
---------------+---------+-----
 佐藤商事      |       1 |  10
 鈴木物産      |       2 |   5
 高橋商店      |       3 |   8
 鈴木物産      |       1 |   3
 高橋商店      |       2 |   4
(5 rows)
```

外部結合は、片方の表にしか存在しないため結合で消えてしまった行データも検索結果に含むことができる結合方式です。LEFT OUTER JOIN 句を使うと、結合の左側に来た表の行データがすべて検索結果に含まれるようになります。
customer 表を左側にした orders 表との左外部結合を行っています。上の例と異なるのは、「JOIN」を「LEFT OUTER JOIN」に変更しただけですが、こうすることで orders 表に該当する行が存在しない場合も、customer 表に含むデータはもれなく結果に含むようになります。藤原流通という店舗があって検索結果には載っているものの、販売実績が無いことがわかるのです。

```
ossdb=# SELECT c.customer_name,o.prod_id,o.qty
FROM customer c
LEFT OUTER JOIN orders   o ON c.customer_id = o.customer_id;
 customer_name | prod_id | qty
---------------+---------+-----
 佐藤商事      |       1 |  10
 鈴木物産      |       2 |   5
 高橋商店      |       3 |   8
 鈴木物産      |       1 |   3
 高橋商店      |       2 |   4
 藤原流通      |         |
(6 rows)
```

複数表の外部結合も可能です。以下の例では、さらに別の prod 表を LEFT OUTER JOIN 句で結合しています。prod_id を

商品名に置き換えています。

```
ossdb=# SELECT c.customer_name,p.prod_name,o.qty
FROM customer c
LEFT OUTER JOIN orders o ON c.customer_id = o.customer_id
LEFT OUTER JOIN prod   p ON o.prod_id = p.prod_id;
 customer_name | prod_name | qty
---------------+-----------+-----
 鈴木物産      | みかん    |   3
 佐藤商事      | みかん    |  10
 高橋商店      | りんご    |   4
 鈴木物産      | りんご    |   5
 高橋商店      | メロン    |   8
 藤原流通      |           |
(6 rows)
```

状況に応じて、FULL OUTER JOIN や RIGHT OUTGER JOIN も同じような考え方で使うことができます。

5.4.2　クロス結合

全ての店舗と商品の組み合わせを取得するような問い合わせでは、結合条件を指定しないクロス結合を使用します。以下の例では、curotmer 表（4 行）と prod 表（4 行）から取得されうる全組み合わせのパターン（4×4=16 行）を取得しています。先の外部結合の結果から、「藤原流通で販売されたバナナ」という組み合わせは実際のデータには存在しないことがわかっていますが、クロス結合の場合はすべての可能性のある組み合わせを取得しているのです。

```
ossdb=# SELECT customer_name,prod_name
FROM customer c
CROSS JOIN prod p;
 customer_name | prod_name
---------------+-----------
 佐藤商事      | みかん
 鈴木物産      | みかん
 高橋商店      | みかん
 藤原流通      | みかん
 佐藤商事      | りんご
 鈴木物産      | りんご
 高橋商店      | りんご
 藤原流通      | りんご
 佐藤商事      | メロン
 鈴木物産      | メロン
 高橋商店      | メロン
 藤原流通      | メロン
 佐藤商事      | バナナ
 鈴木物産      | バナナ
 高橋商店      | バナナ
 藤原流通      | バナナ
(16 rows)
```

以下のように、結合に JOIN 句を用いず、FROM 句の後にカンマ区切りで複数の表を並べることでも同じ結果が得られます。（この場合、結合条件が必要な場合は ON 句のかわりに WHERE 句を用います。）

```
ossdb=# SELECT customer_name,prod_name
FROM customer c, prod p;
 customer_name | prod_name
---------------+-----------
 佐藤商事      | みかん
```

```
鈴木物産       | みかん
高橋商店       | みかん
  :
```

5.4.3　自己結合

自己結合は 1 つの表を 2 つの表に見立てて結合する結合方式です。このとき、2 つの表として区別するため、表に別名を使う必要があります。

以下の例では、すべての商品の組み合わせのうち、価格の合計が 100 未満となる組み合わせのみを検索しています。prod 表は 1 つしかありませんが、prod 表を別名で p1 表と p2 表の 2 つの表に見立てて、p1 と p2 を単純結合（すべての組み合わせを取り出す結合）し、価格の合計に対して WHERE 句の条件で絞り込みを行っています。

```
ossdb=# SELECT p1.prod_name,p2.prod_name,p1.price + p2.price AS pricesum
FROM prod p1,prod p2
WHERE p1.price + p2.price < 100;
 prod_name | prod_name | pricesum
-----------+-----------+----------
 みかん    | バナナ    |       80
 バナナ    | みかん    |       80
 バナナ    | バナナ    |       60
(3 rows)
```

これだけでは少しわかりにくいかもしれませんが、途中経過を実行してみるとよくわかります。prod 表には以下のデータが入っています。

```
ossdb=# SELECT prod_name,price FROM prod;
 prod_name | price
-----------+-------
 みかん    |    50
 りんご    |    70
 メロン    |   100
 バナナ    |    30
(4 rows)
```

これを自己結合することで、2 つの商品を選ぶ場合の組み合わせを生成します。組み合わせにはクロス結合を使います。

```
ossdb=# SELECT p1.prod_name,p1.price,
              p2.prod_name,p2.price
FROM prod p1,prod p2;
 prod_name | price | prod_name | price
-----------+-------+-----------+-------
 みかん    |    50 | みかん    |    50
 みかん    |    50 | りんご    |    70
 みかん    |    50 | メロン    |   100
 みかん    |    50 | バナナ    |    30
 りんご    |    70 | みかん    |    50
   :
```

さらにこの結果の右に、行ごとの合計金額を表示してみましょう。

```
ossdb=# SELECT p1.prod_name,p1.price "価格 1",
              p2.prod_name,p2.price "価格 2",
              p1.price + p2.price   "合計"
FROM prod p1,prod p2;
 prod_name | 価格 1 | prod_name | 価格 2 | 合計
```

```
-----------+-------+-----------+-------+------
 みかん    |    50 | みかん    |    50 |  100
 みかん    |    50 | りんご    |    70 |  120
 みかん    |    50 | メロン    |   100 |  150
 みかん    |    50 | バナナ    |    30 |   80
 りんご    |    70 | みかん    |    50 |  120
   :
```

この結果に対して、「合計金額が 100 円未満」という WHERE 条件を追加します。

```
ossdb=# SELECT p1.prod_name,p1.price "価格 1",
               p2.prod_name,p2.price "価格 2",
               p1.price + p2.price   "合計"
FROM prod p1,prod p2
WHERE p1.price + p2.price < 100;
 prod_name | 価格 1 | prod_name | 価格 2 | 合計
-----------+-------+-----------+-------+------
 みかん    |    50 | バナナ    |    30 |   80
 バナナ    |    30 | みかん    |    50 |   80
 バナナ    |    30 | バナナ    |    30 |   60
(3 rows)
```

あとは SELECT リストに指定するものを必要なものに絞れば元の結果が得られます。

5.5　LIMIT 句による検索行数制限

LIMIT 句を使うと、検索で取り出す行データの数を制限することができます。通常の SQL 文の検索では、条件に当てはまる行データはすべて表示されてしまいますが、LIMIT 句を指定すると必要とする行数だけを取り出せます。

5.5.1　LIMIT と並び順の指定

問い合わせの結果は順番が保証されていませんから、確実に指定した行を取り出すには、ORDER BY 句を使って行データの並びを指定する必要があります。

以下の例では、orders 表から 3 行だけ行データを取り出しています。順序を確定させるために、order_id 列で並べ替えを行っています。

```
ossdb=# SELECT * FROM orders ORDER BY order_id;
 order_id |         order_date         | customer_id | prod_id | qty
----------+----------------------------+-------------+---------+-----
        1 | 2018-01-22 12:34:51.510398 |           1 |       1 |  10
        2 | 2018-01-22 12:34:58.875188 |           2 |       2 |   5
        3 | 2018-01-22 12:35:07.440391 |           3 |       3 |   8
        4 | 2018-01-22 12:35:16.847541 |           2 |       1 |   3
        5 | 2018-01-22 12:35:26.164922 |           3 |       2 |   4
(5 rows)

ossdb=# SELECT * FROM orders ORDER BY order_id LIMIT 3;
 order_id |         order_date         | customer_id | prod_id | qty
----------+----------------------------+-------------+---------+-----
        1 | 2018-01-22 12:34:51.510398 |           1 |       1 |  10
        2 | 2018-01-22 12:34:58.875188 |           2 |       2 |   5
        3 | 2018-01-22 12:35:07.440391 |           3 |       3 |   8
(3 rows)
```

5.5.2　OFFSET 句

OFFSET 句を組み合わせることで、先頭から不要な行数を飛ばしてから行データを取り出すことができます。

以下の例では、OFFSET 句の値として 1 を与えているので、1 行飛ばして 2 行目から 3 行の行データを取り出しています。

```
ossdb=# SELECT * FROM orders ORDER BY order_id LIMIT 3 OFFSET 1;
 order_id |         order_date         | customer_id | prod_id | qty
----------+----------------------------+-------------+---------+-----
        2 | 2018-01-22 12:34:58.875188 |           2 |       2 |   5
        3 | 2018-01-22 12:35:07.440391 |           3 |       3 |   8
        4 | 2018-01-22 12:35:16.847541 |           2 |       1 |   3
(3 rows)
```

6　データベース定義の応用

データベースのデータの整合性を高め、より効率的に使うためにはデータベースの定義をしっかりと行う必要があります。この章では主キー、外部キーなどのデータベースの定義に必要となる基礎知識や、シーケンスなどの便利な機能について解説します。

6.1　主キー

主キーは、表のデータを一意に識別できる 1 つ以上の列のことです。「一意」(UNIQUE) とは、列の値が重複していないことです。一意である列のことを「一意キー」とも呼びます。主キーは一意キーである他に、必ず値が入っている必要があります。必ず値が入っていることを「NULL ではない」(NOT NULL) と呼びます。NULL についての詳細は後述します。

以下の例では、orders 表の order_id 列を条件にすれば特定の一行だけを取り出すことができますが、customer_id 列を条件にすると複数の行データが取り出されてしまいます。この場合、order_id 列は一意の値を持つので主キーとして使用できますが、customer_id 列は一意の値を持つことはできないと考えられますので、主キーにはなりません。

```
ossdb=# SELECT * FROM orders WHERE order_id = 1;
 order_id |         order_date         | customer_id | prod_id | qty
----------+----------------------------+-------------+---------+-----
        1 | 2018-01-22 12:34:51.510398 |           1 |       1 |  10
(1 row)

ossdb=# SELECT * FROM orders WHERE customer_id = 2;
 order_id |         order_date         | customer_id | prod_id | qty
----------+----------------------------+-------------+---------+-----
        2 | 2018-01-22 12:34:58.875188 |           2 |       2 |   5
        4 | 2018-01-22 12:35:16.847541 |           2 |       1 |   3
(2 rows)
```

6.1.1　主キーを指定する

主キーを指定するには、CREATE TABLE 文で表の作成時に指定するか、すでに作成されている表に対して ALTER TABLE 文で指定します。

■6.1.1.1　主キーを指定する ALTER TABLE 文の構文

```
ALTER TABLE 表名 ADD PRIMARY KEY  (列名 [,...])
```

以下の例では、prod 表の prod_id 列を主キーに指定しています。主キーを指定すると、検索を高速化するためのインデックスが自動的に（暗黙的に）作成されます。主キーに指定された列には、NOT NULL 制約が設定され、「列名_pkey」という名前の一意インデックスが作成されます。

```
ossdb=# ALTER TABLE prod ADD PRIMARY KEY(prod_id);
ALTER TABLE
ossdb=# \d prod
                Table "public.prod"
  Column   |  Type   | Collation | Nullable | Default
-----------+---------+-----------+----------+---------
 prod_id   | integer |           | not null |
 prod_name | text    |           |          |
 price     | numeric |           |          |
Indexes:
```

```
    "prod_pkey" PRIMARY KEY, btree (prod_id)
```

orders 表の order_id 列、customer 表の customer_id 列も主キーとして指定しておきます。

```
ossdb=# ALTER TABLE orders ADD PRIMARY KEY(order_id);
ALTER TABLE
ossdb=# ALTER TABLE customer ADD PRIMARY KEY(customer_id);
ALTER TABLE
```

参考：主キーを指定する際、以前のバージョンではインデックスを暗黙的に作成する旨のメッセージが記録されていましたが、PostgreSQL 10 ではメッセージ出力はありません。管理者にとって重要なメッセージに注力されるよう、バージョンによってメッセージの影響度が見直されています。

```
ossdb=# SHOW client_min_messages ;
 client_min_messages
---------------------
 notice
(1 row)

ossdb=# SET client_min_messages TO 'DEBUG1';
SET
ossdb=# ALTER TABLE customer ADD PRIMARY KEY(customer_id);
DEBUG:  ALTER TABLE / ADD PRIMARY KEY will create implicit index "customer_pkey" for table "customer"
DEBUG:  building index "customer_pkey" on table "customer"
ALTER TABLE
```

6.1.2　主キーの動作を確認する

主キーを指定すると、「一意」(UNIQUE) で「NULL ではない」(NOT NULL) という制約が設定されます。つまり、そのような制約に違反する行データの入力などができなくなります。

以下の例では、prod 表の主キーである prod_id の値を指定しない（NULL になる）INSERT 文や、既に存在している値を指定している INSERT 文がエラーになっています。

```
ossdb=# INSERT INTO prod (prod_name,price) VALUES ('すいか',60);
ERROR:  null value in column "prod_id" violates not-null constraint
DETAIL:  Failing row contains (null, すいか, 60).
ossdb=# INSERT INTO prod (prod_id,prod_name,price) VALUES (4,'すいか',60);
ERROR:  duplicate key value violates unique constraint "prod_pkey"
DETAIL:  Key (prod_id)=(4) already exists.
ossdb=# INSERT INTO prod (prod_id,prod_name,price) VALUES (5,'すいか',60);
INSERT 0 1
ossdb=# SELECT * FROM prod;
 prod_id | prod_name | price
---------+-----------+-------
       1 | みかん    |    50
       2 | りんご    |    70
       3 | メロン    |   100
       4 | バナナ    |    30
       5 | すいか    |    60
(5 rows)
```

6.1.3　複数列からなる主キー

「主キーは一意に行データを識別する 1 つ以上の列」と説明した通り、複数列からなる主キーを設定することも可能です。これを複合主キー、または複合キーと呼びます。

たとえば「1 年 2 組出席番号 3 番」のように、複数の要素から成り立つような場合が複合主キーの良い例です。

```
ossdb=# CREATE TABLE student (class TEXT,no INTEGER,name TEXT);
CREATE TABLE
ossdb=# ALTER TABLE student ADD PRIMARY KEY (class,no);
ALTER TABLE
ossdb=# \d student
              Table "public.student"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 class  | text    |           | not null |
 no     | integer |           | not null |
 name   | text    |           |          |
Indexes:
    "student_pkey" PRIMARY KEY, btree (class, no)
```

6.2　外部キー

外部キーは、その列の値が他の表の主キー（または一意キー。以後省略）に存在している列のことです。外部キーが他の表の主キーの値を参照することを「外部キー参照」、参照している先の主キーを「参照キー」と呼びます。

6.2.1　参照整合性制約

外部キー参照を行い、必ず参照キーに値があることを保証することを「外部キー制約」、あるいは「参照整合性制約」と呼びます。外部キー制約が設定されると、参照キーに存在しない値を外部キーに挿入したり、参照キーに存在しない値に外部キーの値を更新しようとするとエラーになるので、外部キーの値が間違えた値になってしまうのを防ぐことができます。また、外部キーから参照されている値を参照キーから削除することもできなくなるので、他の表の外部キーで使用されている値が参照できなくなることもありません。

6.2.2　外部キーを指定する

外部キーを指定するには、CREATE TABLE 文で表の作成時に指定するか、すでに作成されている表に対して ALTER TABLE 文で指定します。

■6.2.2.1　外部キーを指定する ALTER TABLE 文の構文

```
ALTER TABLE 表名 ADD FOREIGN KEY（列名）REFERENCES 参照表（参照キー名）
```

以下の例では、orders 表の customer_id 列と prod_id 列に外部キーを設定しています。

```
ossdb=# ALTER TABLE orders ADD FOREIGN KEY (customer_id) REFERENCES customer(customer_id);
ALTER TABLE
ossdb=# ALTER TABLE orders ADD FOREIGN KEY (prod_id) REFERENCES prod(prod_id);
ALTER TABLE
ossdb=# \d orders
                          Table "public.orders"
   Column    |            Type             | Collation | Nullable | Default
-------------+-----------------------------+-----------+----------+---------
 order_id    | integer                     |           | not null |
 order_date  | timestamp without time zone |           |          |
 customer_id | integer                     |           |          |
 prod_id     | integer                     |           |          |
 qty         | integer                     |           |          |
Indexes:
```

```
    "orders_pkey" PRIMARY KEY, btree (order_id)
Foreign-key constraints:
    "orders_customer_id_fkey" FOREIGN KEY (customer_id) REFERENCES customer(customer_id)
    "orders_prod_id_fkey" FOREIGN KEY (prod_id) REFERENCES prod(prod_id)

ossdb=# \d customer
                Table "public.customer"
    Column     |  Type   | Collation | Nullable | Default
---------------+---------+-----------+----------+---------
 customer_id   | integer |           | not null |
 customer_name | text    |           |          |
Indexes:
    "customer_pkey" PRIMARY KEY, btree (customer_id)
Referenced by:
    TABLE "orders" CONSTRAINT "orders_customer_id_fkey" FOREIGN KEY (customer_id) REFERENCES customer(customer_i
```

以下の例では、まず customer 表に customer_id 列の値が 4 の行データがないため、外部キー制約に違反してエラーになっています。次に、prod 表に prod_id 列の値が 6 の行データがないため、外部キー制約に違反してエラーになっています。

```
ossdb=# INSERT INTO orders(order_id,order_date,customer_id,prod_id,qty)
VALUES (6,now(),4,6,6);
ERROR:  insert or update on table "orders" violates foreign key constraint "orders_customer_id_fkey"
DETAIL:  Key (customer_id)=(4) is not present in table "customer".
ossdb=# INSERT INTO orders(order_id,order_date,customer_id,prod_id,qty)
VALUES (6,now(),3,6,6);
ERROR:  insert or update on table "orders" violates foreign key constraint "orders_prod_id_fkey"
DETAIL:  Key (prod_id)=(6) is not present in table "prod".
```

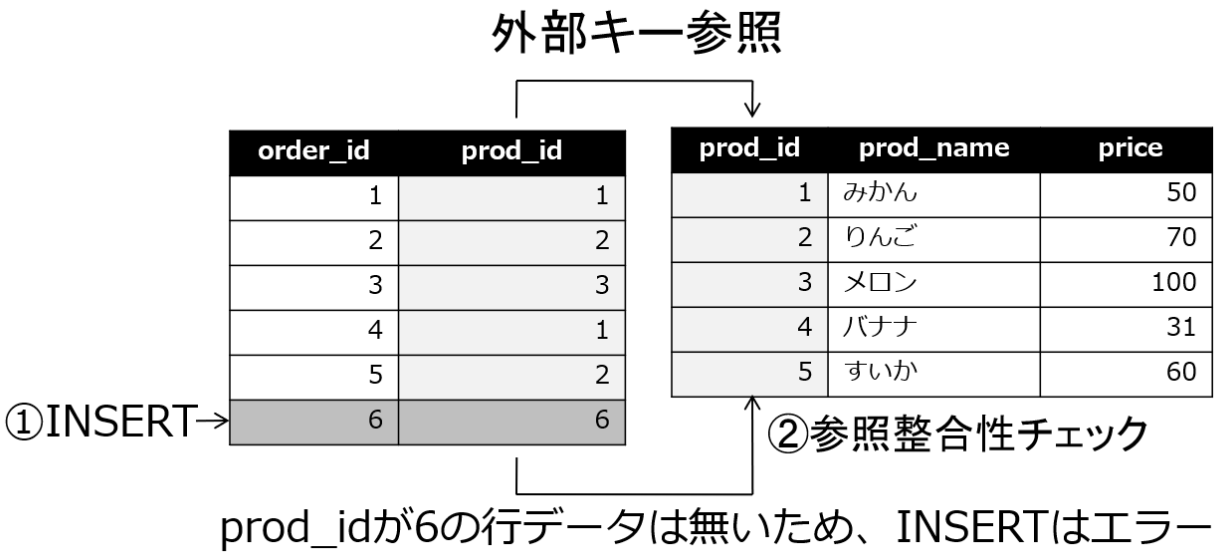


図 7　外部キー制約違反

次の INSERT では、先ほど失敗した prod_id = 6 をやめ、prod_id = 5 を挿入します。今度は制約違反は発生せず、正常に INSERT が完了しました。

```
ossdb=# INSERT INTO orders(order_id,order_date,customer_id,prod_id,qty)
VALUES (6,now(),3,5,6);
INSERT 0 1
ossdb=# SELECT * FROM orders WHERE order_id = 6;
```

```
 order_id |         order_date         | customer_id | prod_id | qty
----------+----------------------------+-------------+---------+-----
        6 | 2018-01-23 12:34:56.149141 |           3 |       5 |   6
(1 row)

ossdb=# SELECT o.order_id,c.customer_name,p.prod_name,o.qty
FROM orders o
JOIN customer c ON o.customer_id = c.customer_id
JOIN prod p ON o.prod_id = p.prod_id
WHERE order_id = 6;
 order_id | customer_name | prod_name | qty
----------+---------------+-----------+-----
        6 | 高橋商店      | すいか    |   6
(1 row)
```

6.2.3　CREATE TABLE 文で主キー、外部キーを設定する

主キー、外部キーは CREATE TABLE 文でも設定することができます。

以下の例は、prod 表、customer 表、orders 表に主キー、外部キーを設定する CREATE TABLE 文です。外部キーを設定するには参照キーとなる他の表の主キーが必要となるため、先に参照先の表を作成します。

```
ossdb=# DROP TABLE orders;
ossdb=# DROP TABLE prod;
ossdb=# DROP TABLE customer;

ossdb=# CREATE TABLE prod
        (prod_id   INTEGER PRIMARY KEY,
         prod_name TEXT,
         price     INTEGER);
CREATE TABLE
ossdb=# CREATE TABLE customer
        (customer_id   INTEGER PRIMARY KEY,
         customer_name TEXT);
CREATE TABLE
ossdb=# CREATE TABLE orders
        (order_id    INTEGER   PRIMARY KEY,
         order_date  TIMESTAMP,
         customer_id INTEGER   REFERENCES customer (customer_id),
         prod_id     INTEGER   REFERENCES prod (prod_id),
         qty         INTEGER);
CREATE TABLE
```

6.2.4　主キー、外部キーは必要か？

主キーや外部キーを設定することで、表に誤って重複した値を入れたり、間違えて行データを削除してしまったりすることを防ぐことができます。反面、一時的であってもに整合性が取れていない状態を許容しなくなってしまうので、データをメンテナンスする際などに不便です。不便さを嫌って、データの整合性チェックをデータベースの制約で行うのではなく、アプリケーション側で行うようにすることもあります。

どちらが良いかは一概には言えませんが、基本的にデータベース側で主キーや外部キーの設定を行い、システム運用上問題がある場合には制約を取り去ると考えておけばよいでしょう。少なくとも設計上は、主キー、外部キーの考え方は重要です。また、データベース設計の際に、後述する正規化をきちんと行う必要があります。

6.3 正規化

正規化とは、リレーショナルデータベースにどのようにデータを格納するかを決めるデータベース設計の手法です。リレーショナルデータベースは表の形式で行データを格納するので、実際の行データを表形式で格納しやすいように複数の表に分解していく作業だと考えれば良いでしょう。

正規化には第 1 正規形（1NF）や第 2 正規形（2NF）、第 3 正規形（3NF）などの形があります。簡単に言えば、正規化が進む度に表は分割されて増えていきます。分割することでそれぞれの表はシンプルな構造になっていくので、行データに対する修正や削除、データの追加などを行った時に問題が発生する可能性が低くなっていきます。

本書では正規化について詳細には解説しませんが、データベースの設計を行うためには知っておかなければならない考え方ですので、専門書などで学習してみてください。

6.4 NULL について

NULL（ヌル）は「未知」または「未定」と定義されるもので、「ゼロ」や「空白」「空文字」とは区別されます。数値のゼロや文字の空白、空文字はそれぞれデータが「有る」状態ですが、NULL はデータが「無い」という状態を表しています。

6.4.1 NOT NULL 制約

列に NULL を許さない場合、表定義で NOT NULL 制約を設定します。NOT NULL 制約が設定された列には必ず値が必要となります。主キーは必ず値を必要とするので、主キーを定義すると自動的に NOT NULL 制約が設定されます。

以下の例では、customer 表の customer_id 列が主キーとして設定されているため、NOT NULL 制約が設定されています。

```
ossdb=# \d customer
                Table "public.customer"
    Column     |  Type   | Collation | Nullable | Default
---------------+---------+-----------+----------+---------
 customer_id   | integer |           | not null |
 customer_name | text    |           |          |
Indexes:
    "customer_pkey" PRIMARY KEY, btree (customer_id)
Referenced by:
    TABLE "orders" CONSTRAINT "orders_customer_id_fkey" FOREIGN KEY (customer_id) REFERENCES customer(customer_i
```

6.4.2 NULL の判定

NULL は値を持たないため、通常の演算子を使った条件式では検索することができません。列の値が NULL かどうかを条件判定するには、IS NULL 演算子、IS NOT NULL 演算子を使用します。以下の例では、price 列が NULL でない行として「みかん」を、price 列が NULL である行「ぶどう」を INSERT し、それぞれ検索しています。（prod 表にデータがない状態から INSERT しています。）

```
ossdb=# INSERT INTO prod VALUES (1,'みかん',50);
ossdb=# INSERT INTO prod VALUES (6,'ぶどう',NULL);
ossdb=# SELECT * FROM prod;
 prod_id | prod_name | price
---------+-----------+-------
       1 | みかん    |    50
       6 | ぶどう    |
(2 rows)

ossdb=# SELECT * FROM prod WHERE price IS NULL;
 prod_id | prod_name | price
---------+-----------+-------
```

```
       6 | ぶどう     |
(1 row)

ossdb=# SELECT * FROM prod WHERE price IS NOT NULL;
 prod_id | prod_name | price
---------+-----------+-------
       1 | みかん     |    50
(1 row)
```

6.4.3　NULL の集約関数での取り扱い

各種集約関数では NULL は無視されることがあります。以下の例では、count(*) で表そのものの行数を数える場合と、count(price) で price 列の値の数を数える場合を比べています。

```
ossdb=# SELECT count(*) FROM prod;
 count
-------
     2
(1 row)

ossdb=# SELECT count(price) FROM prod;
 count
-------
     1
(1 row)
```

price 列の合計 sum(price) や最大、最小 max(price) などの集約関数で計算結果に影響しないことは明白ですが、平均 avg(price) ではどうでしょうか。

（みかんの金額「50 円」＋ぶどうの金額「不明」）÷ 2 行 = 25 円

と思いませんか？

```
ossdb=# SELECT sum(price),count(price),avg(price) FROM prod;
 sum | count |         avg
-----+-------+---------------------
  50 |     1 | 50.0000000000000000
(1 row)
```

実際には、先ほどの count(price) の結果からもわかる通り、NULL は平均値を算出するための行数には含められず、

50 円 ÷ 1 行 = 50 円

という結果になりました。

6.4.4　空文字

NULL と似たものとして「空文字」があります。空文字は、INSERT 文で文字列型の列データに「''」（シングルクォートを 2 つ連続）で指定できます。空文字は NULL ではないので、NOT NULL 制約に違反しません。

以下の例では、prod 表の prod_name 列に空文字の行データを入力しています。price 列は NULL としました。prod_name 列を IS NULL 条件で検索しましたが、空文字は「空の値がある」状態ですので、IS NULL 演算子では検索されません。一方、NULL を挿入した price 列は IS NULL で検索されます。

```
ossdb=# INSERT INTO prod (prod_id,prod_name) VALUES (7,'');
INSERT 0 1
ossdb=# SELECT * FROM prod;
 prod_id | prod_name | price
---------+-----------+-------
       1 | みかん     |    50
```

```
       6 | ぶどう     |
       7 |            |
(3 rows)

ossdb=# SELECT * FROM prod WHERE prod_name IS NULL;
 prod_id | prod_name | price
---------+-----------+-------
(0 rows)

ossdb=# SELECT * FROM prod WHERE price IS NULL;
 prod_id | prod_name | price
---------+-----------+-------
       6 | ぶどう     |
       7 |            |
(2 rows)
```

6.5　シーケンス

シーケンス（順序）は、連番を生成する機能です。たとえばシーケンスを INSERT 文の中で使用すると、自動的に連番が値として挿入されるので、ID 番号など重複なく一意にしたい列の値を取るのに適しています。

6.5.1　シーケンスの作成

シーケンスを作成するには、CREATE SEQUENCE 文を使用します。

■6.5.1.1　シーケンスの作成構文

```
CREATE SEQUENCE シーケンス名;
```

■6.5.1.2　シーケンスのデフォルト値

項目	値
開始値	1
増加量	1
最大値	2 の 63 乗-1(9,223,372,036,854,775,807)

6.5.2　シーケンスの操作

シーケンス操作用の関数を使うことで値を取り出したり、値を設定したりすることができます。

- **現在の値を返す**
  シーケンスの現在の値を返します。シーケンスの値は更新されません。セッション内で一度もシーケンスの値を取り出していない場合はエラーになります。

```
currval('シーケンス名')
```

- **次の値を返し、値を更新する**
  現在値の次の値を返して、シーケンスの値を次の値に更新します。デフォルトではシーケンスの値は 1 ずつ増えていくので、現在値が 1 だった時には次の値は 2 となります。最大値に達すると、デフォルトでは nextval の呼び出しはエラーになります。

```
nextval('シーケンス名')
```

- **シーケンスの値を設定する**
  シーケンスの値を指定した値に設定します。

```
setval('シーケンス名', 値)
```

以下の例では、order_id_seq シーケンスから値を取り出しています。

```
ossdb=# CREATE SEQUENCE order_id_seq;
CREATE SEQUENCE
ossdb=# SELECT currval('order_id_seq');
ERROR:  currval of sequence "order_id_seq" is not yet defined in this session
ossdb=# SELECT nextval('order_id_seq');
 nextval
---------
       1
(1 row)

ossdb=# SELECT currval('order_id_seq');
 currval
---------
       1
(1 row)

ossdb=# SELECT nextval('order_id_seq');
 nextval
---------
       2
(1 row)
```

setval() 関数を使うと、シーケンスの値を再設定できます。設定可能な値は 1 からの値です。

```
ossdb=# SELECT setval('order_id_seq',0);
ERROR:  setval: value 0 is out of bounds for sequence "order_id_seq" (1..9223372036854775807)
ossdb=# SELECT setval('order_id_seq',100);
 setval
--------
    100
(1 row)

ossdb=# SELECT currval('order_id_seq');
 currval
---------
     100
(1 row)

ossdb=# SELECT nextval('order_id_seq');
 nextval
---------
     101
(1 row)
```

6.5.3 シーケンスを SQL 文で使用する

シーケンスは、たとえば INSERT 文に組み込んで使用します。

以下の例では、orders 表へ行データを入力する INSERT 文で、order_id 列の値を order_id_seq シーケンスから取得しています。

```
ossdb=# TRUNCATE orders;
TRUNCATE TABLE
ossdb=# SELECT setval('order_id_seq',100);
 setval
--------
    100
(1 row)

ossdb=# INSERT INTO orders(order_id,order_date,customer_id,prod_id,qty)
VALUES (nextval('order_id_seq'),now(),2,3,7);
INSERT 0 1
ossdb=# SELECT * FROM orders;
 order_id |         order_date         | customer_id | prod_id | qty
----------+----------------------------+-------------+---------+-----
      101 | 2018-01-23 12:34:56.426626 |           2 |       3 |   7
(1 row)
```

6.5.4 シーケンスと飛び番

シーケンスを使って簡単に連番を作ることができますが、実行した SQL 文が失敗した場合でもシーケンスの値は進んでしまいます。この時、連番になっていない、いわゆる「飛び番」が発生します。シーケンスは完全な連番を保証するものではなく、また完全な連番を作るのは困難です。たとえば途中の行が削除されてしまえば、完全な連番ではなくなってしまいます。シーケンスはあくまでも行データを区別するための重複していない値と割り切る方がシステム設計上は楽になります。

以下の例では、INSERT 文がエラーで失敗しても、シーケンスの値が進んでしまって次の INSERT 文で飛び番が発生しています。

```
ossdb=# SELECT currval('order_id_seq');
 currval
---------
     101
(1 row)

ossdb=# INSERT INTO orders(order_id,order_date,customer_id,prod_id,qty)
VALUES (nextval('order_id_seq'),now(),10,4,7);
ERROR:  insert or update on table "orders" violates foreign key constraint "orders_customer_id_fkey"
DETAIL:  Key (customer_id)=(10) is not present in table "customer".
ossdb=# SELECT currval('order_id_seq');
 currval
---------
     102
(1 row)

ossdb=# INSERT INTO orders(order_id,order_date,customer_id,prod_id,qty)
VALUES (nextval('order_id_seq'),now(),1,2,5);
INSERT 0 1
ossdb=# SELECT * FROM orders;
 order_id |         order_date         | customer_id | prod_id | qty
```

```
----------+----------------------------+-------------+---------+-----
       101 | 2018-01-23 12:34:56.426626 |           2 |       3 |   7
       103 | 2018-01-23 12:35:59.329873 |           1 |       2 |   5
(2 rows)
```

7　マルチユーザーでの利用

PostgreSQL はマルチユーザーのデータベースです。複数のユーザーを使い分けることで、あるユーザーは表のデータを更新でき、あるユーザーは検索のみ行えるようにするなど、ユーザー毎に行えるデータベースの操作を変えたりすることができます。

7.1　ユーザーの作成

PostgreSQL のユーザーを作成するには、CREATE USER 文を使用します。また、Linux のコマンドラインから createuser コマンドを使用してもユーザーを作成できます。

ユーザーを確認するには\du メタコマンドを実行します。

以下の例では、ユーザー sato を作成しています。このユーザーでログインするときに必要なパスワードも指定しています。

```
[postgres@localhost ~]$ psql ossdb
ossdb=# CREATE USER sato PASSWORD 'sato';
CREATE ROLE
ossdb=# \du
                                   List of roles
 Role name |                         Attributes                         | Member of
-----------+------------------------------------------------------------+-----------
 postgres  | Superuser, Create role, Create DB, Replication, Bypass RLS | {}
 sato      |                                                            | {}
```

以下の例では、Linux のコマンドラインから、createuser コマンドを使用してユーザー suzuki を作成しています。-P オプションをつけると、このユーザーでログインするときに必要なパスワードを対話式で指定することができます。

```
[postgres@localhost ~]$ createuser -P suzuki
新しいロールのためのパスワード:
もう一度入力してください：
[postgres@localhost ~]$ psql ossdb
ossdb=# \du
                                   List of roles
 Role name |                         Attributes                         | Member of
-----------+------------------------------------------------------------+-----------
 postgres  | Superuser, Create role, Create DB, Replication, Bypass RLS | {}
 sato      |                                                            | {}
 suzuki    |                                                            | {}
```

7.1.1　ユーザーとロール

CREATE USER 文や createuser コマンドの結果表示にはユーザーではなくロール (ROLE) と表示されています。PostgreSQL ではログイン属性を持つロールをユーザーと呼びます。理想的なアクセス制御は、権限の集合としてのロール（グループロール）と、ログイン可能な属性を持つロール（ユーザーロール）を適切に使い分けることで実現します。例えば、あるサービスを運用するために必要な複数の表に対するアクセス権限（A 表に対する更新可能、B 表に対しては参照のみ可能のような細やかな設定）をまとめたグループロールを作成し、そのグループロールを特定の管理ユーザーに付与するようにします。ユーザーの棚卸やサービスの変更（テーブル構成の見直し）などがあった場合に柔軟に対処するためです。本書では、単にデータベースに接続し SQL の基礎を学習するという目的に沿って、ログイン時に使用するユーザーという概念を重視し、ユーザーの作成として解説しています。

7.1.2　スーパーユーザー

データベースには一番最初に初期化した際に、データベースに対するすべての権限を持ったユーザーが作成されます。これをスーパーユーザーと呼びます。Linux における root ユーザーや、Windows における Administrator ユーザーのようなものと

考えればよいでしょう。PostgreSQL では、Linux 上でデータベースの初期化（initdb コマンドの実行）を行った OS ユーザーの名前でスーパーユーザーが作成されます。このユーザーのユーザ名は慣習的に postgres となっています。

7.2　接続と認証

マルチユーザーで利用する際には、外部からネットワーク経由での PostgreSQL への接続や、接続時の認証が必要となります。これらはデフォルトでは設定されていないので、設定方法について解説します。

7.2.1　接続認証の設定を確認

PostgreSQL での接続認証の設定は、設定ファイルの一つである pg_hba.conf に記述します。デフォルトでは、以下のように設定が記述されています。

```
[postgres@localhost ~]$ cat $PGDATA/pg_hba.conf
------
(略)
# TYPE  DATABASE        USER            ADDRESS                 METHOD

# "local" is for Unix domain socket connections only
local   all             all                                     trust
# IPv4 local connections:
host    all             all             127.0.0.1/32            trust
# IPv6 local connections:
host    all             all             ::1/128                 trust
(略)
------
```

各項目の設定は、左から順に以下のようになっています。

- 接続方法（TYPE）
  クライアントがどのように PostgreSQL に接続するかを指定します。

■7.2.1.1　接続方法の設定

接続タイプ	説明
local	PostgreSQL が実行されているホストと同じホストからの接続
host	外部からの TCP/IP を使った接続
hostssl	外部からの SSL を使った接続

- データベース（DATABASE）
  接続認証の対象となるデータベースを指定します。all と記述するとすべてのデータベースが対象となります。
- ユーザー（USER）
  接続認証の対象となるユーザーを指定します。all と記述するとすべてのユーザーが対象となります。
- 認証方法（METHOD）
  認証方式を指定します。

■7.2.1.2　認証方法の設定

認証メソッド	説明
trust	認証なしに接続
reject	接続拒否
md5	MD5 パスワード認証
password	平文パスワード認証
scram-sha-256	SCRAM 認証
gss	GSSAPI 認証
sspi	SSPI
ident	IDENT 認証
peer	peer 認証
pam	PAM 認証
ldap	LDAP 認証
radius	RADIUS 認証
cert	SSL クライアント証明書認証

RPM からインストールした場合のデフォルトの設定では、ローカルホストでの接続で、すべてのデータベース、ユーザーに対して trust 認証を行うことが設定されています。

7.2.2　接続ユーザーの指定

psql でデータベースに接続する際、本来はどのユーザーで接続するか指定する必要がありますが、明示的に指定されなかった場合には psql を実行した Linux のユーザー名が暗黙の内に指定されます。接続ユーザーは`\set` メタコマンドを実行して変数 USER の値で確認できます。

以下の例では、`id` コマンドの結果で Linux のユーザー名が postgres であること、接続のユーザーも postgres になっていることを確認しています。psql にはデータベース名として ossdb のみ指定しているので、接続ユーザーは暗黙の内に postgres が指定されているのが分かります。

```
[postgres@localhost ~]$ id
uid=1001(postgres) gid=1001(postgres) groups=1001(postgres) context=unconfined_u:unconfined_r:unconfined_t:s0-s0
[postgres@localhost ~]$ psql ossdb
psql (10.1)
Type "help" for help.

ossdb=# \set
AUTOCOMMIT = 'on'
COMP_KEYWORD_CASE = 'preserve-upper'
DBNAME = 'ossdb'
ECHO = 'none'
ECHO_HIDDEN = 'off'
ENCODING = 'UTF8'
FETCH_COUNT = '0'
HISTCONTROL = 'none'
HISTSIZE = '500'
HOST = '/var/run/postgresql'
IGNOREEOF = '0'
ON_ERROR_ROLLBACK = 'off'
ON_ERROR_STOP = 'off'
PORT = '5432'
PROMPT1 = '%/%R%# '
PROMPT2 = '%/%R%# '
PROMPT3 = '>> '
QUIET = 'off'
```

```
SERVER_VERSION_NAME = '10.1'
SERVER_VERSION_NUM = '100001'
SHOW_CONTEXT = 'errors'
SINGLELINE = 'off'
SINGLESTEP = 'off'
USER = 'postgres'
VERBOSITY = 'default'
VERSION = 'PostgreSQL 10.1 on x86_64-pc-linux-gnu, compiled by gcc (GCC) 4.8.5 20150623 (Red Hat 4.8.5-16), 64-b
VERSION_NAME = '10.1'
VERSION_NUM = '100001'
```

次に、ユーザー sato で接続した場合の例です。ログイン時にユーザー名を指定して、その通りになっていることがわかります。また、psql のプロンプトが「=#」ではなく「=>」になっており、これはユーザー sato が、スーパーユーザーでなく一般ユーザーであることを表します。

```
[postgres@localhost ~]$ psql ossdb sato
psql (10.1)
Type "help" for help.

ossdb=> \set
(該当箇所のみ抜粋)
DBNAME = 'ossdb'
USER = 'sato'
```

7.2.3 パスワード認証の設定

データベースへの接続時にパスワード認証を行うように設定してみます。パスワード認証を設定するには pg_hba.conf に以下のような設定を行います。設定の変更は OS ユーザーを postgres にして行います。

```
vi $PGDATA/pg_hba.conf
-----
# TYPE  DATABASE        USER            ADDRESS                 METHOD

# "local" is for Unix domain socket connections only
local   all             all                                     md5
# IPv4 local connections:
host    all             all             127.0.0.1/32            md5
# IPv6 local connections:
host    all             all             ::1/128                 md5
```

設定の変更は PostgreSQL を再起動（または設定の再読み込み）をするまでは有効になりません。ユーザーにパスワードの設定を行ってから `pg_ctl reload` で設定を再読み込みします。パスワードの設定を行わないまま本設定を読み込むと、TRUST 認証が無効になってしまうためデータベースに接続できなくなります。

7.2.4 パスワードの設定、変更

パスワード認証が有効になると、パスワードが設定されていないユーザーはデータベースに接続できなくなるので、パスワードを設定しておきます。

- 既存ユーザーにパスワード設定
  既存のユーザーに対して ALTER USER 文でパスワードを設定します。初期ユーザーである postgres ユーザーにはパスワードが設定されていませんので、インストール直後に必ず本操作を実施しましょう。#### ALTER USER 文でパスワードを設定する構文

```
ALTER USER ユーザー名 PASSWORD 'パスワード'
```

以下の例では、ユーザー postgres に対してパスワードを postgres に設定しています。

```
[postgres@localhost ~]$ psql ossdb
ossdb=# ALTER USER postgres PASSWORD 'postgres';
ALTER ROLE
```

- 強度の高いパスワード格納方式
  pg_hba.conf で指定する認証方式には、password、md5、scram-sha-256 という 3 つのパスワード格納方式が指定できます。このうち、デフォルトの格納方式は md5 ですが、近年では暗号化強度が弱いとされ、PostgreSQL 10 からは scram-sha-256 が利用可能になっています。

```
[postgres@localhost ~]$ psql ossdb
ossdb=# SET password_encryption = 'scram-sha-256';
SET
ossdb=# ALTER USER postgres PASSWORD 'postgres';
ALTER ROLE
```

この場合、pg_hba.conf の記述も上記の md5 の代わりに scram-sha-256 とします。

```
vi $PGDATA/pg_hba.conf
# TYPE  DATABASE        USER            ADDRESS                 METHOD
# "local" is for Unix domain socket connections only
local   all             all                                     scram-sha-256
# IPv4 local connections:
host    all             all             127.0.0.1/32            scram-sha-256
# IPv6 local connections:
host    all             all             ::1/128                 scram-sha-256
```

7.2.5　設定値の再読み込み

PostgreSQL の各設定には、その設定が反映されるタイミングが定められています。ほぼすべてのパラメーターが起動時に読み込まれるほか、pg_hba.conf の設定やその他の一部のパラメーターは設定の再読み込みで反映されるようになっています。

以下の例は、ユーザー postgres で pg_ctl コマンドで PostgreSQL の設定を再読み込みしています。

```
[postgres@localhost ~]$ pg_ctl reload
サーバにシグナルを送信しました
```

PostgreSQL の停止または再起動は、他のユーザーがデータベースを使用している場合、デフォルトでは実行中の処理を中断して停止処理が優先される動作になっています。重要な処理の実行に影響しないよう、メンテナンス時間帯を設けて作業を行うことが大切ですが、このように他の処理への影響を軽微にできる方法が用意されている場合もあります。

7.2.6　パスワード認証による接続

パスワード認証が有効になったかどうかを確認します。

以下の例では、ユーザー postgres、ユーザー sato での接続を確認しています。

```
[postgres@localhost ~]$ psql ossdb
Password:
psql (10.1)
Type "help" for help.

ossdb=# \q
[postgres@localhost ~]$ psql ossdb sato
Password for user sato:
psql (10.1)
Type "help" for help.
```

```
ossdb=> \q
```

7.3　ネットワーク経由接続

PostgreSQL は、TCP/IP を利用したネットワーク経由での接続も受け付けることができます。

7.3.1　ネットワーク経由接続の設定

ネットワーク経由接続を受け付けるには、postgresql.conf に listen_addresses の設定を行って、PostgreSQL を再起動します。デフォルトでは、`listen_addresses = 'localhost'` が設定されていて、PostgreSQL が実行されているホストでのローカルループバック接続のみが有効になっています。値を**`*`（アスタリスク）**に設定することで、ホストが用意しているすべてのインターフェースからの接続を受け付けるようになります。もし特定のインターフェースからのみ受け付けたい場合には、そのインターフェースに設定された IP アドレスを記述します。接続受付のポート番号はデフォルトで 5432 に設定されています。ポート番号を変更したい場合には、port の設定値を変更します。

以下の例では、すべてのインターフェースからの接続を受け付けるように設定しています。

```
[postgres@localhost ~]$ vi $PGDATA/postgresql.conf
-----
#listen_addresses = 'localhost'         # what IP address(es) to listen on;
listen_addresses = '*'                  # what IP address(es) to listen on;
                                        # comma-separated list of addresses;
                                        # defaults to 'localhost', '*' for all
                                        # (change requires restart)
#port = 5432                            # (change requires restart)
```

あわせて、接続認証の設定も行います。pg_hba.conf の host アクセス制御を設定します。

以下の例では、ネットワーク経由接続で ossdb データベースに sato ユーザーがアクセスする際にパスワード認証するように設定しています。pg_hba.conf の設定反映は再読み込みで良いですが、今回は listen_addresses も変更していますので、PostgreSQL の再起動で両方の設定を反映させます。

```
[postgres@localhost ~]$ vi $PGDATA/pg_hba.conf
# TYPE  DATABASE        USER            ADDRESS                 METHOD
# "local" is for Unix domain socket connections only
local   all             all                                     scram-sha-256
# IPv4 local connections:
host    ossdb           sato            0.0.0.0/0               scram-sha-256
host    all             all             127.0.0.1/32            scram-sha-256
# IPv6 local connections:
host    all             all             ::1/128                 scram-sha-256
```

7.3.2　PostgreSQL の再起動

設定変更後に設定を反映させるため PostgreSQL を再起動します。ユーザー root で `systemctl restart postgresql-10.service` を実行するか、ユーザー postgres で `pg_ctl restart` コマンドを実行します。

以下の例は、ユーザー root で PostgreSQL を再起動しています。

```
[root@localhost ~]# systemctl restart postgresql-10.service
```

以下の例は、ユーザー postgres で pg_ctl コマンドで PostgreSQL を再起動しています。

```
[postgres@localhost ~]$ pg_ctl restart
サーバ停止処理の完了を待っています.... 完了
サーバは停止しました
サーバの起動完了を待っています....
```

7.3.3　psql を使ったネットワーク経由接続

psql を使って PostgreSQL にネットワーク経由接続するにはオプション-h でホスト名、-p でポート番号を指定します。ユーザー、データベース、クライアント端末の組み合わせが pg_hba.conf で許可されている必要があります。

■7.3.3.1　psql でネットワーク経由接続

```
psql -h ホスト名 -p ポート番号 -U ユーザー名 データベース名
```

以下の例では、サーバーの IP アドレスに対してネットワーク経由接続を行っています。

```
[postgres@localhost ~]$ psql -h 192.168.101.10 -p 5432 -U sato ossdb
Password for user sato:
psql (10.1)
Type "help" for help.

ossdb=>
```

pg_hba.conf に定義した組み合わせ（sato ユーザー、ossdb データベース）以外では、認証エラーが出ています。

```
[postgres@localhost ~]$ psql -h 192.168.101.10 -p 5432 -U sato postgres
psql: FATAL:  no pg_hba.conf entry for host "192.168.101.10", user "sato", database "postgres", SSL off
```

7.4　アクセス権限

1 つのデータベースに複数のユーザーが接続できる場合、アクセス権限を設定することで表などに対する操作を制御することができます。

7.4.1　アクセス権限の付与

アクセス権限を付与するには、GRANT 文を使用します。

■7.4.1.1　GRANT 文の構文

```
GRANT {ALL | SELECT | INSERT | DELETE | UPDATE}
    ON object TO {user | PUBLIC}
```

以下の例は、ユーザー sato に対して prod 表に対するすべての権限を付与しています。prod 表の所有者（owner）である postgres ユーザーで操作してユーザー sato に対して権限を与えます。

```
[postgres@localhost ~]$ psql -U postgres ossdb
Password for user postgres:
psql (10.1)
Type "help" for help.

ossdb=# \dt prod
        List of relations
 Schema | Name | Type  |  Owner
--------+------+-------+----------
 public | prod | table | postgres
(1 row)
ossdb=# GRANT all ON prod TO sato;
GRANT
```

7.4.2　アクセス権限の確認

アクセス権限を確認するには、\dp メタコマンドを使用します。

```
ossdb=# \dp prod
                              Access privileges
 Schema | Name | Type  |     Access privileges     | Column privileges | Policies
--------+------+-------+---------------------------+-------------------+----------
 public | prod | table | postgres=arwdDxt/postgres+|                   |
        |      |       | sato=arwdDxt/postgres     |                   |
(1 row)
```

■7.4.2.1　アクセス権の表記

権限を付与されたユーザー=権限種別/権限を付与したユーザー

権限を付与されたユーザーが空白のときは public（全ユーザー）に対して許可されていることを表します。アクセス権限の後ろにある「/posgres」は、この権限を与えたユーザーを表しています。表のオーナーや、相当する操作が許可されたユーザーが当てはまります。

権限種別	説明
a	INSERT(Append の意)
r	SELECT(Read の意)
w	UPDATE(Write の意)
d	DELETE
D	TRUNCATE
x	REFERENCES
t	TRIGGER

REFERENCES 権限は、外部キー制約を作成する両方の表に権限を持っている必要があります。TRIGGER 権限は、表に対する操作をトリガー（引き金）にして別の処理を行う「トリガー機能」を表に対して作成できる権限です。

7.4.3　アクセス権限の取り消し

与えたアクセス権限を取り消すには、REVOKE 文を使用します。

■7.4.3.1　REVOKE 文の構文

```
REVOKE {ALL | SELECT | INSERT | DELETE | UPDATE}
    ON object FROM {user|PUBLIC}
```

以下の例では、ユーザー sato から prod 表に対するすべての権限を取り消しています。

```
[postgres@localhost ~]$ psql -U postgres ossdb
Password for user postgres:
psql (10.1)
Type "help" for help.

ossdb=# REVOKE all ON prod FROM sato;
REVOKE
ossdb=# \dp prod
```

```
                              Access privileges
 Schema | Name | Type  |     Access privileges     | Column privileges | Policies
--------+------+-------+---------------------------+-------------------+----------
 public | prod | table | postgres=arwdDxt/postgres |                   |
(1 row)
```

7.5　トランザクション

トランザクションとは、データベースに対する 1 つ以上の処理のまとまりのことです。データベースに対する処理が開始されると同時にトランザクションが開始 (BEGIN) され、まとまった処理の結果を確定するコミット (COMMIT)、あるいは処理を破棄するロールバック (ROLLBACK) が行われるまでが 1 つのトランザクションです。

これまで PostgreSQL の操作に使用してきた SQL 文は、自動コミット（AUTOCOMMIT）がデフォルトで有効になっていたため、すべての操作が成功すると自動的に COMMIT が発行されていました。自動コミットを無効にするには、psql メタコマンドの`\set AUTOCOMMIT=off` を実行するか、BEGIN を実行して明示的にトランザクションを開始する必要があります。

以下の例では、customer 表に 1 行 INSERT しますが、それを `BEGIN` で開始することでトランザクションとして実行しています。最後に `ROLLBACK` して INSERT が取り消されている例と、最後に `COMMIT` して挿入した結果が確定されている例です。

```
ossdb=# BEGIN;
BEGIN
ossdb=# INSERT INTO customer VALUES (5,'田中産業');
INSERT 0 1
ossdb=# SELECT * FROM customer WHERE customer_id = 5;
 customer_id | customer_name
-------------+---------------
           5 | 田中産業
(1 row)

ossdb=# ROLLBACK;
ROLLBACK

ossdb=# SELECT * FROM customer WHERE customer_id = 5;
 customer_id | customer_name
-------------+---------------
(0 rows)

ossdb=# BEGIN;
BEGIN
ossdb=# INSERT INTO customer VALUES (5,'田中産業');
INSERT 0 1

ossdb=# COMMIT;
COMMIT

ossdb=# SELECT * FROM customer WHERE customer_id = 5;
 customer_id | customer_name
-------------+---------------
           5 | 田中産業
(1 row)
```

7.5.1　読み取り一貫性

読み取り一貫性とは、あるトランザクション内で行われているデータ更新はコミットして確定されない限り、他の検索トランザクションに対して影響を及ぼさないことです。

以下の例では、まず 1 つのデータベース接続でトランザクションを開始します。メロンの価格を 120 円にアップデートしました。まだ COMMIT も ROLLBACK もせず、トランザクションの途中で他のセッションから prod 表を検索すると元の 100 円が得られます。更新トランザクションが確定されたあとは他のセッションでも更新後の 120 円を得ることができます。

```
ossdb=# SELECT * FROM prod;
 prod_id | prod_name | price
---------+-----------+-------
       1 | みかん    |    50
       2 | りんご    |    70
       3 | メロン    |   100
(3 rows)
ossdb=# BEGIN;
BEGIN
ossdb=# UPDATE prod SET price = 120 WHERE prod_id = 3;
UPDATE 1
ossdb=# SELECT * FROM prod;
 prod_id | prod_name | price
---------+-----------+-------
       1 | みかん    |    50
       2 | りんご    |    70
       3 | メロン    |   120
(3 rows)

/* 別のターミナルで検索した後にトランザクションを確定 */

ossdb=# COMMIT;
COMMIT
```

別のターミナルを立ち上げて、別の psql を実行します。

```
[root@localhost ~]# su - postgres
[postgres@localhost ~]$ psql ossdb
Password:
psql (10.1)
Type "help" for help.

/* 更新後、COMMIT は実行していない状態 */

ossdb=# SELECT * FROM prod;
 prod_id | prod_name | price
---------+-----------+-------
       1 | みかん    |    50
       2 | りんご    |    70
       3 | メロン    |   100
(3 rows)

/* 更新トランザクションを COMMIT した後 */

ossdb=# SELECT * FROM prod;
```

```
 prod_id | prod_name | price
---------+-----------+-------
       1 | みかん    |    50
       2 | りんご    |    70
       3 | メロン    |   120
(3 rows)
```

図 8　読み取り一貫性

このように、最初のトランザクションが行データを追加していても、そのトランザクションがコミットされる前では、別のトランザクションからはその行データ追加は見えません。もし見えてしまうと、その後ロールバックされて追加が取り消された時、結局存在しないことになった行データを読んでしまうことになるからです。このようにコミットされる前の行データを読んでしまうことを「ダーティーリード」と呼びます。PostgreSQL はダーティーリードが発生しないよう、読み取り一貫性を維持しています。なお、本項では詳しく扱いませんが、図中にあるように、PostgreSQL では更新が確定された後、古い行を読み飛ばす制御も行っています。

7.5.2　ロック機構と更新の競合

データベースは、先に行われたトランザクションの対象をロックし、他のトランザクションが書き換えないように保護します。ロックされているデータ行を他のトランザクションが更新しようとすると更新の競合が発生し、コミットまたはロールバックでロックが解除されるまで処理が待たされます。

以下の例では、まず一つのトランザクションが prod 表の行データを更新します。

```
ossdb=# BEGIN;
BEGIN
```

```
ossdb=# UPDATE prod SET price = price * 1.1 WHERE prod_id = 1;
UPDATE 1
ossdb=# SELECT * FROM prod WHERE prod_id =1;
 prod_id | prod_name | price
---------+-----------+-------
       1 | みかん    |    55
(1 row)
```

別のターミナルを立ち上げて、別の psql を実行します。前のトランザクションが更新している prod 表の同じ行データを更新します。

```
[postgres@localhost ~]$ psql ossdb
Password:
psql (10.1)
Type "help" for help.

ossdb=# BEGIN;
BEGIN
ossdb=# UPDATE prod SET price = price * 1.2 WHERE prod_id = 1;
```

図 9　更新の競合

2 つめの UPDATE 文が実行されたところで更新の競合が発生し、前のトランザクションが完了するまで待たされます。

以下のように、前のトランザクションをコミットかロールバックするとロックが解除され、後から実行された UPDATE 文の更新が完了します。

```
ossdb=# BEGIN;
BEGIN
ossdb=# UPDATE prod SET price = price * 1.1 WHERE prod_id = 1;
UPDATE 1
ossdb=# SELECT * FROM prod WHERE prod_id =1;
 prod_id | prod_name | price
---------+-----------+-------
       1 | みかん    |    55
(1 row)

/* この時点で競合が発生 */
```

```
ossdb=# ROLLBACK;
ROLLBACK
```

前のトランザクションがロールバックされると、後から実行された UPDATE 文の更新が完了することを確認します。

```
ossdb=# BEGIN;
BEGIN
ossdb=# UPDATE prod SET price = price * 1.2 WHERE prod_id = 1;

/* この時点で更新の競合が発生 */
/* 前のトランザクションがロールバックされた直後に UPDATE 文が実行される */

UPDATE 1
ossdb=# SELECT * FROM prod WHERE prod_id = 1;
 prod_id | prod_name | price
---------+-----------+-------
       1 | みかん    |    60
(1 row)
ossdb=# COMMIT;
COMMIT
```

後から実行された UPDATE 文の実行時の price 列の値は 50 だったため、1.2 倍の 60 に更新されています。

7.5.3　デッドロック

もし、2 つのトランザクションがお互いに相手のロックしているデータ行を更新しようとして、更新の競合が同時に発生したらどうなるでしょうか。トランザクションが互いに相手のロック解除待ちで止まってしまう状態になることを「デッドロック」と呼びます。

PostgreSQL ではデッドロックが発生すると、どちらかのトランザクション全体を失敗させて強制的にロールバックさせます。片方のトランザクションが行っていたロックは解除されるので、もう一方のトランザクションは正常に終了することができます。

まず、一方のトランザクションが price 表の prod_id 列の値が 1 の行データを更新します。

```
ossdb=# BEGIN;
BEGIN
ossdb=# UPDATE prod SET price = price * 1.1 WHERE prod_id = 1;
UPDATE 1
```

次に、もう一方のトランザクションが price 表の prod_id 列の値が 2 の行データを更新します。

```
ossdb=# BEGIN;
BEGIN
ossdb=# UPDATE prod SET price = price * 1.1 WHERE prod_id = 2;
UPDATE 1
```

さらに、前のトランザクションが price 表の prod_id 列の値が 2 の行データを更新します。この更新はもう一方のトランザクションが更新してロックしている行データなので、更新の競合が発生します。

```
ossdb=# UPDATE prod SET price = price * 1.2 WHERE prod_id = 2;
/* 更新の競合が発生 */
```

もう一方のトランザクションが、前のトランザクションが更新している price 表の prod_id 列の値が 1 の行データを更新しようとすると、デッドロックが発生し、トランザクションはロールバックされます。この結果、前のトランザクションのロック待ちが解除され、更新が実行されます。

```
ossdb=# UPDATE prod SET price = price * 1.2 WHERE prod_id = 1;
ERROR:  deadlock detected
```

```
DETAIL:  Process 3027 waits for ShareLock on transaction 728; blocked by process 2775.
Process 2775 waits for ShareLock on transaction 729; blocked by process 3027.
HINT:  See server log for query details.
CONTEXT:  while updating tuple (0,6) in relation "prod"
ossdb=#
```

図 10　デッドロック

デッドロックが発生しないようにする、または影響を軽微に抑えるいくつかの方法が論じられますが、完全に防ぐことは難しいとも言われる話題です。ひとつの有力な案としては、複数個所の更新が必要な場合に更新の順序を一定にすることです。例のように更新対象行を交差させてしまうとデッドロックが発生しやすくなります。上記では「prod_id が小さいほうから更新する」ようなルールを定めておけば、後からのトランザクションはひとつ目の更新を待機しますのでデッドロックは発生しません。そしてトランザクションがデータ行をロックしている時間を短くするために、できるだけ早くトランザクションを終了することです。トランザクションの時間を短くすればするほど、デッドロックが発生しにくくなります。ただし、アプリケーションによっては「prod_id が小さいほう」を決めることが容易でない場合や、大規模な開発では規約が守られないケースもあるでしょう。上記は同一テーブル内の複数行ですが、複数テーブル間で発生するデッドロックや、索引や制約に起因するユーザーが意図せず発生してしまうデッドロックもあります。

参考：デッドロックの検知方法はデータベース製品によって異なり、それによる影響も異なります。PostgreSQL では、パラメーター `deadlock_timeout` で定められた間隔（デフォルトでは 1 秒間隔）でその瞬間に実行されているトランザクションのロックの状態（獲得済みまたは獲得待ち）を確認することで検知します。そのため、先に検知されたどちらか一方のトランザクションがキャンセルされるのです。他のデータベース製品では、後から処理を実行したトランザクションがキャンセルされるなど一定のルールがあります。

データベースの最も重要な機能は、トランザクションを正しく実行し、深刻なデータ破壊を防ぐことです。データを守るために、誤った更新はデッドロックのような形をとり実行させないようにしているのです。データベースの利用者はこのような問題が発生しうることを知り、利用しているデータベース製品の機能や、アプリケーション側で考えられる代表的な対策を知っておくことで、いざ必要になったときにその場に最も適した対策を考えられるようにしておきましょう。

8　パフォーマンスチューニング

データベースの性能を高めるためには、様々な性能向上のための仕組みやパフォーマンスチューニングの方法について理解しておく必要があります。

8.1　インデックス（索引）

インデックスは、検索の際に目的となる行データを素早く見つけるための仕組みです。その名の通り、本の索引のようにデータがどこにあるかを直接指し示してくれます。インデックスが無いと、検索する度に表全体を検索する必要があります。行データが多いと検索対象のデータが増えるため、性能が大幅に劣化して遅くなります。このように表全体を検索することを「シーケンシャルスキャン」や「フルスキャン」、インデックスから検索することを「インデックススキャン」と呼びます。

8.1.1　主キーのインデックス

インデックススキャンが意図したとおりに行なわれるよう設計し、インデックスを作成しなければなりません。正しく正規化された表でインデックスが最も有効に働くのは、主キーに対して作成したインデックスです。主キーは検索の条件検索や表の結合などで検索されることが多いので、インデックスを作成しておくとインデックススキャンで高速に必要な行データを見つけ出すことができます。

以下の例のように、主キーを定義すると自動的にインデックスが作成されます。

```
ossdb=# \d prod
                Table "public.prod"
  Column   |  Type   | Collation | Nullable | Default
-----------+---------+-----------+----------+---------
 prod_id   | integer |           | not null |
 prod_name | text    |           |          |
 price     | integer |           |          |
Indexes:
    "prod_pkey" PRIMARY KEY, btree (prod_id)
Referenced by:
    TABLE "orders" CONSTRAINT "orders_prod_id_fkey" FOREIGN KEY (prod_id) REFERENCES prod(prod_id)
```

8.1.2　インデックスの作成

インデックスを作成するには、CREATE INDEX 文を使用します。インデックスを作成したい表の列を指定します。列は 1 列でも良いですし、複数列を指定することもできます。複数列を指定したインデックスを「複合インデックス」と呼びます。複合インデックスは指定された列の一部だけを検索条件にすると有効にならない場合もあるので、検索時の SQL 文がどのような検索条件となるかを考慮して作成する必要があります。

以下の例では、orders 表の customer_id 列にインデックスを作成しています。

```
ossdb=# CREATE INDEX orders_customer_id_idx
ON orders(customer_id);
CREATE INDEX
ossdb=# \d orders
                         Table "public.orders"
   Column    |            Type             | Collation | Nullable | Default
-------------+-----------------------------+-----------+----------+---------
 order_id    | integer                     |           | not null |
 order_date  | timestamp without time zone |           |          |
 customer_id | integer                     |           |          |
 prod_id     | integer                     |           |          |
 qty         | integer                     |           |          |
```

```
Indexes:
    "orders_pkey" PRIMARY KEY, btree (order_id)
    "orders_customer_id_idx" btree (customer_id)
Foreign-key constraints:
    "orders_customer_id_fkey" FOREIGN KEY (customer_id) REFERENCES customer(customer_id)
    "orders_prod_id_fkey" FOREIGN KEY (prod_id) REFERENCES prod(prod_id)
```

8.1.3　インデックスを削除する

インデックスを削除するには DROP INDEX 文を使用します。

以下の例では、orders_customer_id_idx インデックスを削除しています。

```
ossdb=# DROP INDEX orders_customer_id_idx;
DROP INDEX
ossdb=# \d orders
                          Table "public.orders"
   Column    |            Type             | Collation | Nullable | Default
-------------+-----------------------------+-----------+----------+---------
 order_id    | integer                     |           | not null |
 order_date  | timestamp without time zone |           |          |
 customer_id | integer                     |           |          |
 prod_id     | integer                     |           |          |
 qty         | integer                     |           |          |
Indexes:
    "orders_pkey" PRIMARY KEY, btree (order_id)
Foreign-key constraints:
    "orders_customer_id_fkey" FOREIGN KEY (customer_id) REFERENCES customer(customer_id)
    "orders_prod_id_fkey" FOREIGN KEY (prod_id) REFERENCES prod(prod_id)
```

8.1.4　インデックスは万能ではない

インデックスは検索を高速化する手段ですが、万能ではありません。まず、インデックスの対象となる列の値が頻繁に書き換わる場合、インデックスも頻繁に更新する必要があります。行データが増えればそれだけインデックス更新の負荷も高くなるため、あまり書き換わることのない列の方がインデックスに向いているということになります。また、列の値が適度にばらついていないと、インデックススキャンよりもシーケンシャルスキャンの方が効率が良い場合があります。インデックスが有効に働いているかどうかは、次に説明する分析などを行って確認する必要があるでしょう。

8.2　SQL 実行プランの分析

SQL が実際にどのようにデータベース内部で実行されているかを確認するには、SQL 実行プランを分析します。分析を行うには、分析したい SQL 文の前に EXPLAIN 文をつけて実行します。

8.2.1　インデックスが存在しない場合の SQL 実行プラン

インデックスが存在しない表に対する検索は、フルスキャンになることが分かります。

以下の例では、郵便番号データベースの検索を行う SELECT 文を分析しています。インデックスは存在しないのでシーケンシャルスキャン（Seq Scan）になっています。

```
ossdb=# EXPLAIN SELECT * FROM zip WHERE newzip = '1500002';
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on zip  (cost=0.00..4297.06 rows=1 width=145)
   Filter: (newzip = '1500002'::bpchar)
```

```
(2 rows)
```

8.2.2　インデックスが存在する場合の SQL 実行プラン

インデックスが存在していて、インデックスを利用した方が良いと判断される場合には、インデックススキャンが行われることが分かります。

```
ossdb=# CREATE INDEX zip_newzip_idx ON zip(newzip);
CREATE INDEX
ossdb=# EXPLAIN SELECT * FROM zip WHERE newzip = '1500002';
                                QUERY PLAN
---------------------------------------------------------------------------
 Index Scan using zip_newzip_idx on zip  (cost=0.42..8.44 rows=1 width=145)
   Index Cond: (newzip = '1500002'::bpchar)
(2 rows)
```

8.2.3　インデックスが存在しても必ず使われるわけではない

インデックスが存在していても、検索条件によってはインデックスを使う必要は無いと判断されます。

以下の例では、zip 表の largearea 列にインデックスを作成していますが、largearea 列は 0 か 1 といずれかの値しか持たない列のため、必ずインデックスが使われるとは限らなくなっています。最後に 0、1 それぞれ何件格納されているかを count 関数を使って調べています。

```
ossdb=# CREATE INDEX zip_largearea ON zip(largearea);
CREATE INDEX
ossdb=# EXPLAIN SELECT * FROM zip WHERE largearea = 0;
                         QUERY PLAN
------------------------------------------------------------
 Seq Scan on zip  (cost=0.00..4297.06 rows=121342 width=145)
   Filter: (largearea = 0)
(2 rows)

ossdb=# EXPLAIN SELECT * FROM zip WHERE largearea = 1;
                                 QUERY PLAN
-----------------------------------------------------------------------------
 Index Scan using zip_largearea on zip  (cost=0.42..750.22 rows=2823 width=145)
   Index Cond: (largearea = 1)
(2 rows)

ossdb=# SELECT largearea,count(*) FROM zip GROUP BY largearea;
 largearea | count
-----------+--------
         0 | 121390
         1 |   2775
(2 rows)
```

largearea 列の値はほとんどが 0 のため、インデックスを利用しても性能が出ないと判断してフルスキャンを選択しています。また、largearea 列の値が 1 の行データは比較的少ないため、インデックススキャンが選択されています。

この例も書籍の索引と同じで、PostgreSQL に関する技術書で「PostgreSQL」という単語が書かれたページを巻末の索引で探すことはしないでしょう。ほとんどのページが該当してしまうことが予想できる場合は、そのような探し方はせずに先頭からページをめくって調べるほうがはるかに効率的だからです。

8.3 バキューム処理

PostgreSQL を継続して利用していくには、バキューム処理が必要になってきます。バキューム処理は、必要の無くなった行データが格納されている領域を回収して、再度利用可能な状態にする処理です。バキューム処理は、PostgreSQL のデータ管理方式に密接に関わっています。

8.3.1 PostgreSQL のデータ管理方式

PostgreSQL では、行データが更新されたり削除されたりした時に、実際の行データは消しません。それまでの行データに削除されて使用されなくなった印をつけて検索の対象から外すようにします。バキューム処理は、これらの不要な行データを回収する処理を行います。この方式の利点は、更新や削除の段階で物理的に行データを削除せず印を付けておくだけなので、性能面では有利となります。反面、更新処理が多くなると、不要になった行データの量が増えすぎて物理的なデータ量が大きくなってしまうので、ディスク容量を圧迫したり、シーケンシャルスキャンの性能が劣化してしまいます。

8.3.2 VACUUM と VACUUM FULL

バキューム処理を行うには、VACUUM 文あるいは VACUUM FULL 文を使用します。VACUUM 文は、不要な行データを回収して再利用可能な状態にします。データファイルのサイズは変わりません。VACUUM FULL 文はさらに行データの物理的な配置を移動させてデータファイルのサイズを縮小することができます。VACUUM FULL 文は行データの移動を伴うため、実行すると表全体に強いロックが取得されて、他のユーザーが処理を行えなくなります。VACUUM FULL 文は副作用が大きいので、大量にデータを削除した後、データファイルのサイズを縮小してディスクの空き容量を増やしたい場合などに使用すると良いでしょう。

8.3.3 VACUUM ANALYZE

VACUUM 文は、データの分布を調査して SQL 実行プランの決定に役立てる統計情報を再作成する ANALYZE 文と同時実行できます。

以下の例は、データベースに対して VACUUM ANALYZE を実行しています。

```
ossdb=# VACUUM ANALYZE;
VACUUM
```

8.3.4 自動バキュームデーモン

自動バキュームデーモンは、VACUUM と ANALYZE を自動的に実行してくれる仕組みです。自動バキュームデーモンはデフォルトで動作しており、バキューム処理、統計情報再作成処理が必要になるタイミングを監視しています。

自動バキュームデーモンが動作しているかどうかを確認してみましょう。Linux 上での動作している autovacuum という名前のプロセスが自動バキュームデーモンです。

```
[postgres@localhost ~]$ ps ax | grep autovacuum
11425 ?        Ss     0:56 postgres: autovacuum launcher process
 3925 pts/2    R+     0:00 grep --color=auto autovacuum
```

8.4 クラスタ

クラスタ化は、物理的なデータの配置をインデックスに従って再配置します。再配置により、インデックスを使って検索される行データが物理ディスク上でまとめられるため、ディスクアクセスが減り性能が向上することが期待できます。

クラスタ化を行うには CLUSTER 文を使用します。初めてクラスタ化を行う場合にはインデックスを USING 句で明示的に指定する必要がありますが、2 回目以降はクラスタ化するために使用したインデックスが記録されているのでインデックスを指定しないで CLUSTER 文を実行しても大丈夫です。

以下の例では、orders 表の orders_pkey インデックスを使用してクラスタ化を行っています。クラスタ化に使用したインデックスは、情報の後ろに CLUSTER と表示されます。

```
ossdb=# CLUSTER orders;
ERROR:  there is no previously clustered index for table "orders"
ossdb=# CLUSTER orders USING orders_pkey;
CLUSTER
ossdb=# \d orders
                          Table "public.orders"
   Column    |            Type             | Collation | Nullable | Default
-------------+-----------------------------+-----------+----------+---------
 order_id    | integer                     |           | not null |
 order_date  | timestamp without time zone |           |          |
 customer_id | integer                     |           |          |
 prod_id     | integer                     |           |          |
 qty         | integer                     |           |          |
Indexes:
    "orders_pkey" PRIMARY KEY, btree (order_id) CLUSTER
Foreign-key constraints:
    "orders_customer_id_fkey" FOREIGN KEY (customer_id) REFERENCES customer(customer_id)
    "orders_prod_id_fkey" FOREIGN KEY (prod_id) REFERENCES prod(prod_id)

ossdb=# CLUSTER orders;
CLUSTER
```

9　バックアップとリストア

データベースのバックアップとリストアは、重要なデータを扱うデータベースにとって重要な作業です。ハードディスクの障害などでデータが失われた時、あらかじめ取得しておいたバックアップから確実にリストアが行えるようにしておく必要があります。

9.1　バックアップ手法の整理

データベースのデータは、多数のユーザーにより同時に更新される可能性があり、テキスト文書や表計算ソフトのファイルをUSB メモリにコピーするような単純な手法ではバックアップになりません。

以下のようなバックアップ手法を知り、利用目的に応じて使い分ける必要があります。

9.1.1　主なバックアップ手法一覧

手法	復旧ポイント	説明
ファイルコピー	バックアップ取得時点	データベースを停止して、OS コマンドでファイルをコピーする方法。アプリケーションを停止できる場合は最も簡単。
SELECT 文	バックアップ取得時点	データベースを停止せずに、SELECT した結果をファイルに書き出す。COPY 文や\o メタコマンドを使用。
pg_dump	バックアップ取得時点	データベースを停止せずに専用のコマンドでデータをファイルに書き出す。内部的には COPY 文相当の処理が行われる。バックアップ対象や出力形式を柔軟に選択でき、復旧ポイント次第では有力な選択肢である。
pg_basebackup	障害発生直前	バックアップ取得以降に発生した障害に対して有効な選択肢である。バックアップファイルに変更履歴を順に適用することで障害発生直前の状態まで復旧することができるほか、時刻やトランザクション位置を指定して、変更履歴に含まれる任意の時点に復旧することもできる。
レプリケーション	リアルタイム～指定間隔	レプリケーション機能でデータベースまたはテーブル単位の複製を作成する。標準機能では**ストリーミングレプリケーション**や**ロジカルレプリケーション**、他にもさまざまなレプリケーションツールが公開されており、リアルタイム性の高いものやクラウドで保管するなどツールにより特徴がある。

本書では、バックアップ手法の一例として、環境準備や事前設定が不要で簡単に試すことができる「ファイルコピー」と「pg_dump」を紹介します。

9.2　ファイルのコピー

最も確実で簡単なバックアップは、ファイルレベルでのバックアップです。必要となるファイルをすべてコピーすることでバックアップができます。ただし、ファイルのコピーを行うには PostgreSQL を完全に停止する必要があります。

以下の例では、tar コマンドを使用して PostgreSQL の関連ファイルが格納されているデータディレクトリ$PGDATA 以下をアーカイブしています。

```
[postgres@localhost ~]$ pg_ctl stop -m fast
サーバ停止処理の完了を待っています.... 完了
サーバは停止しました
[postgres@localhost ~]$ tar cvf backup.tar $PGDATA
tar: メンバ名から先頭の `/' を取り除きます
/var/lib/pgsql/10/data/
/var/lib/pgsql/10/data/pg_wal/
/var/lib/pgsql/10/data/pg_wal/archive_status/
```

```
/var/lib/pgsql/10/data/pg_wal/000000010000000000000003
/var/lib/pgsql/10/data/pg_wal/000000010000000000000004
    :
[postgres@localhost ~]$ ls -l backup.tar
-rw-rw-r--. 1 postgres postgres 95887360  1月  23 12:34 backup.tar
[postgres@localhost ~]$ pg_ctl start
サーバの起動完了を待っています....
2018-02-05 02:09:45.616 JST [4866] LOG:  redirecting log output to logging collector process
2018-02-05 02:09:45.616 JST [4866] HINT:  Future log output will appear in directory "log".
完了
サーバ起動完了
```

注意：この方法はデータベースを構成する **すべてのファイル**を利用者自身が指定する必要があります。テーブルスペース機能を用いて複数ディレクトリに分かれて保持されているデータがある場合、もれなく取得しなければなりません。

9.3 pg_dump コマンドによるバックアップ

pg_dump コマンドは、データベースを SQL 文としてバックアップするコマンドです。ファイルのコピーと違い、データベースを停止することなくバックアップすることができます。pg_dump コマンドを実行すると、データベースまたは表単位で、指定した形式でファイルに出力することができます。

pg_dump コマンドは指定したデータベースや表だけをバックアップするコマンドですが、pg_dumpall コマンドを使うと、すべてのデータベース、作成したユーザーなどの情報をまとめてバックアップすることができます。

以下の例では、pg_dump コマンドでデータベース ossdb をバックアップしています。実行時オプションとして、-F で出力形式「ディレクトリ」、-f で「ディレクトリ名」、最後の引数は対象データベース名を指定しています。ディレクトリ形式とは、指定したディレクトリ配下にテーブル毎にダンプファイルを作成するものです。リストア時に対象テーブルがわかっていれば必要なファイルのみを渡せばよい点や、-j オプションと組み合わせることで複数コアを使った並列ダンプに対応しています。

```
[postgres@localhost ~]$ mkdir backup.d
[postgres@localhost ~]$ ls -l | grep backup.d
drwxrwxr-x. 2 postgres postgres     4096  1月 23 12:34 backup.d

[postgres@localhost ~]$ pg_dump -Fd -f backup.d ossdb
パスワード:<postgres ユーザーのパスワード>

[postgres@localhost ~]$ ls backup.d
3711.dat.gz  3713.dat.gz  3715.dat.gz  3717.dat.gz  3719.dat.gz  toc.dat
3712.dat.gz  3714.dat.gz  3716.dat.gz  3718.dat.gz  3720.dat.gz
```

9.4 pg_restore によるリストア

pg_dump コマンドを使ってバックアップしたファイルは、専用の pg_restore コマンドでデータベースに読み込ませることができます。pg_restore の実行時オプションでは、ダンプファイルに含まれる「データベース」または「表」を指定して対象のみをリストアできます。ダンプファイルに含まれるデータベース定義を元に CREATE DATABSE から実施する方法や、データベースに対して CREATE TABLE したり、既存のテーブルにデータのみ INSERT する方法などが柔軟に指定可能です。

以下の例では、障害に見立てて ossdb データベースを削除した後、先ほど取得したバックアップ「backup.d」を指定してデータベース全体をリストアしています。pg_restore コマンド実行時のポイントとしては、いまは ossdb データベースが存在しないため、リストア操作のために一時的に postgres データベースに接続し、-C オプションで新規に ossdb データベースを作成していることです。

```
/* 障害に見立てて ossdb データベースを削除 */

[postgres@localhost ~]$ dropdb ossdb
```

```
パスワード:
[postgres@localhost ~]$ psql ossdb
Password:
psql: FATAL:  database "ossdb" does not exist

/* ossdb データベースを復旧 */

[postgres@localhost ~]$ pg_restore -d postgres -c -C  backup.d
パスワード:
pg_restore: [アーカイバ (db)] TOC 処理中にエラーがありました:
pg_restore: [アーカイバ (db)] TOC エントリ 3726; 1262 16384 DATABASE ossdb postgres のエラーです
pg_restore: [アーカイバ (db)] could not execute query: ERROR:  database "ossdb" does not exist
    コマンド: DROP DATABASE ossdb;

警告: リストアにてエラーを無視しました: 1

/* ここではデータベース ossdb が存在しない旨のエラーが表示されますが、問題ありません。
 * pg_restore ではリストア対象（今回は ossdb データベース）と重複するデータを最初に削除する処理（今回は dropdb ossdb）を実行し
 * 直前の dropdb コマンドで削除対象の ossdb データベースがすでに存在しないため、内部で実行しようとした dropdb 処理のみが失敗して
 */

[postgres@localhost ~]$ psql ossdb
Password:
psql (10.1)
Type "help" for help.

ossdb=# \d
              List of relations
 Schema |     Name     |   Type   |  Owner
--------+--------------+----------+----------
 public | char_test    | table    | postgres
 public | customer     | table    | postgres
 public | date_test    | table    | postgres
 public | numeric_test | table    | postgres
 public | order_id_seq | sequence | postgres
 public | orders       | table    | postgres
 public | prod         | table    | postgres
 public | student      | table    | postgres
 public | varchar_test | table    | postgres
 public | zip          | table    | postgres
(10 rows)

ossdb=# SELECT count(*) FROM zip;
 count
--------
 124165
(1 row)
```

10　Web アプリケーションとの連携

データベースの一般的な利用方法として、Web アプリケーションと連携してデータベースでデータの管理を行う形があります。このように Web アプリケーションの背後で動作するデータベースを「バックエンドデータベース」と呼びます。この章では、Web アプリケーション開発でよく利用されている言語「PHP」との組み合わせでのバックエンドデータベースとしての利用方法を学びます。

10.1　PHP とは？

PHP とは、Web サーバー上で動作する Web アプリケーション開発でよく利用される言語です。HTML ファイルにアプリケーションのコードを埋め込むことができるので、Web ページの一部分にアプリケーションの動作を組み込んだりすることができます。

10.2　PHP の動作イメージ

PHP は Web サーバーの「Apache HTTP サーバー（以下、Apache）」に組み込んで利用するモジュールが配布されており、これを利用すると簡単に Web アプリケーションの開発を始められます。Web ブラウザから拡張子が.php になっているファイルを要求されると、そのファイル内に含まれている PHP のコードが実行され、最終的な結果は HTML として Web ブラウザに渡されます。

10.3　Apache と PHP 環境の設定

Apache と PHP を利用できるようにインストールと設定を行います。インストールするのは PostgreSQL が動作しているマシン上でも、別々のマシンでも構いません。別々のマシンにインストールした場合、PHP と PostgreSQL の間はネットワーク経由で接続することになります。ここでは PostgreSQL が動作している同じマシンに Apache と PHP をインストールします。

10.3.1　Apache と PHP をパッケージでインストール

CentOS では Apache と PHP が RPM パッケージで提供されているので、yum コマンドを使ってインストールします。yum コマンドでインストールが行えない場合には、インストールメディアから rpm コマンドを使って必要なパッケージをインストールします。インストールするパッケージは php と php-pgsql です。

```
[root@localhost ~]# yum install php php-pgsql
読み込んだプラグイン:fastestmirror, langpacks
base                                                     | 3.6 kB  00:00:00
extras                                                   | 3.4 kB  00:00:00
pgdg10                                                   | 4.1 kB  00:00:00
updates                                                  | 3.4 kB  00:00:00
Loading mirror speeds from cached hostfile
 * base: ftp.iij.ad.jp
 * extras: ftp.iij.ad.jp
 * updates: ftp.iij.ad.jp
依存性の解決をしています
--> トランザクションの確認を実行しています。
---> パッケージ php.x86_64 0:5.4.16-43.el7_4 を インストール
--> 依存性の処理をしています: php-common(x86-64) = 5.4.16-43.el7_4 のパッケージ: php-5.4.16-43.el7_4.x86_64
--> 依存性の処理をしています: php-cli(x86-64) = 5.4.16-43.el7_4 のパッケージ: php-5.4.16-43.el7_4.x86_64
--> 依存性の処理をしています: httpd-mmn = 20120211x8664 のパッケージ: php-5.4.16-43.el7_4.x86_64
--> 依存性の処理をしています: httpd のパッケージ: php-5.4.16-43.el7_4.x86_64
---> パッケージ php-pgsql.x86_64 0:5.4.16-43.el7_4 を インストール
--> 依存性の処理をしています: php-pdo(x86-64) = 5.4.16-43.el7_4 のパッケージ: php-pgsql-5.4.16-43.el7_4.x86_64
--> トランザクションの確認を実行しています。
```

```
---> パッケージ httpd.x86_64 0:2.4.6-67.el7.centos.6 を インストール
--> 依存性の処理をしています: httpd-tools = 2.4.6-67.el7.centos.6 のパッケージ: httpd-2.4.6-67.el7.centos.6.x86_64
--> 依存性の処理をしています: /etc/mime.types のパッケージ: httpd-2.4.6-67.el7.centos.6.x86_64
---> パッケージ php-cli.x86_64 0:5.4.16-43.el7_4 を インストール
---> パッケージ php-common.x86_64 0:5.4.16-43.el7_4 を インストール
--> 依存性の処理をしています: libzip.so.2()(64bit) のパッケージ: php-common-5.4.16-43.el7_4.x86_64
---> パッケージ php-pdo.x86_64 0:5.4.16-43.el7_4 を インストール
--> トランザクションの確認を実行しています。
---> パッケージ httpd-tools.x86_64 0:2.4.6-67.el7.centos.6 を インストール
---> パッケージ libzip.x86_64 0:0.10.1-8.el7 を インストール
---> パッケージ mailcap.noarch 0:2.1.41-2.el7 を インストール
--> 依存性解決を終了しました。

依存性を解決しました

================================================================================
 Package                アーキテクチャー      バージョン                  リポジトリー     容量
================================================================================
インストール中:
 php                    x86_64              5.4.16-43.el7_4             updates        1.4 M
 php-pgsql              x86_64              5.4.16-43.el7_4             updates         86 k
依存性関連でのインストールをします:
 httpd                  x86_64              2.4.6-67.el7.centos.6       updates        2.7 M
 httpd-tools            x86_64              2.4.6-67.el7.centos.6       updates         88 k
 libzip                 x86_64              0.10.1-8.el7                base            48 k
 mailcap                noarch              2.1.41-2.el7                base            31 k
 php-cli                x86_64              5.4.16-43.el7_4             updates        2.7 M
 php-common             x86_64              5.4.16-43.el7_4             updates        565 k
 php-pdo                x86_64              5.4.16-43.el7_4             updates         98 k

トランザクションの要約
================================================================================
インストール  2 パッケージ（+7 個の依存関係のパッケージ）

総ダウンロード容量: 7.7 M
インストール容量: 27 M
Is this ok [y/d/N]: y
Downloading packages:
(1/9): libzip-0.10.1-8.el7.x86_64.rpm                       |  48 kB  00:00:01
(2/9): mailcap-2.1.41-2.el7.noarch.rpm                      |  31 kB  00:00:01
(3/9): httpd-tools-2.4.6-67.el7.centos.6.x86_64.rpm         |  88 kB  00:00:01
(4/9): php-5.4.16-43.el7_4.x86_64.rpm                       | 1.4 MB  00:00:01
(5/9): httpd-2.4.6-67.el7.centos.6.x86_64.rpm               | 2.7 MB  00:00:02
(6/9): php-pdo-5.4.16-43.el7_4.x86_64.rpm                   |  98 kB  00:00:00
(7/9): php-common-5.4.16-43.el7_4.x86_64.rpm                | 565 kB  00:00:00
(8/9): php-pgsql-5.4.16-43.el7_4.x86_64.rpm                 |  86 kB  00:00:00
(9/9): php-cli-5.4.16-43.el7_4.x86_64.rpm                   | 2.7 MB  00:00:00
--------------------------------------------------------------------------------
合計                                                3.1 MB/s | 7.7 MB  00:00:02
Running transaction check
Running transaction test
Transaction test succeeded
```

```
Running transaction
  インストール中          : libzip-0.10.1-8.el7.x86_64                                  1/9
  インストール中          : php-common-5.4.16-43.el7_4.x86_64                           2/9
  インストール中          : php-cli-5.4.16-43.el7_4.x86_64                              3/9
  インストール中          : php-pdo-5.4.16-43.el7_4.x86_64                              4/9
  インストール中          : httpd-tools-2.4.6-67.el7.centos.6.x86_64                    5/9
  インストール中          : mailcap-2.1.41-2.el7.noarch                                 6/9
  インストール中          : httpd-2.4.6-67.el7.centos.6.x86_64                          7/9
  インストール中          : php-5.4.16-43.el7_4.x86_64                                  8/9
  インストール中          : php-pgsql-5.4.16-43.el7_4.x86_64                            9/9
  検証中                  : php-cli-5.4.16-43.el7_4.x86_64                              1/9
  検証中                  : php-common-5.4.16-43.el7_4.x86_64                           2/9
  検証中                  : php-5.4.16-43.el7_4.x86_64                                  3/9
  検証中                  : php-pgsql-5.4.16-43.el7_4.x86_64                            4/9
  検証中                  : mailcap-2.1.41-2.el7.noarch                                 5/9
  検証中                  : httpd-2.4.6-67.el7.centos.6.x86_64                          6/9
  検証中                  : httpd-tools-2.4.6-67.el7.centos.6.x86_64                    7/9
  検証中                  : libzip-0.10.1-8.el7.x86_64                                  8/9
  検証中                  : php-pdo-5.4.16-43.el7_4.x86_64                              9/9

インストール:
  php.x86_64 0:5.4.16-43.el7_4          php-pgsql.x86_64 0:5.4.16-43.el7_4

依存性関連をインストールしました:
  httpd.x86_64 0:2.4.6-67.el7.centos.6  httpd-tools.x86_64 0:2.4.6-67.el7.centos.6
  libzip.x86_64 0:0.10.1-8.el7          mailcap.noarch 0:2.1.41-2.el7
  php-cli.x86_64 0:5.4.16-43.el7_4      php-common.x86_64 0:5.4.16-43.el7_4
  php-pdo.x86_64 0:5.4.16-43.el7_4

完了しました!
```

10.3.2　Apache+PHP の設定とテスト

インストールが終わったら、動作テストを行います。

テスト用の PHP を埋め込んだファイル index.php を、Apache が HTML ファイルを参照する/var/www/html ディレクトリに作成します。

```
[root@localhost ~]# echo "<?php phpinfo(); ?>" > /var/www/html/index.php
[root@localhost ~]# cat /var/www/html/index.php
<?php phpinfo(); ?>
```

Web サーバーを起動します。

以下の例では、systemctl のサブコマンド list-unit-files で httpd の起動に用いるサービス名を確認し、start でサービスを起動、status や ps コマンドで正常に起動したことを確認しています。

```
[root@localhost ~]# systemctl list-unit-files | grep httpd
httpd.service                                 disabled
[root@localhost ~]# systemctl start httpd.service
[root@localhost ~]# systemctl status httpd.service
● httpd.service - The Apache HTTP Server
   Loaded: loaded (/usr/lib/systemd/system/httpd.service; disabled; vendor preset: disabled)
   Active: active (running) since 月 2018-02-05 04:31:17 JST; 7s ago
     Docs: man:httpd(8)
           man:apachectl(8)
```

```
 Main PID: 6807 (httpd)
   Status: "Processing requests..."
   CGroup: /system.slice/httpd.service
           tq6807 /usr/sbin/httpd -DFOREGROUND
           tq6809 /usr/sbin/httpd -DFOREGROUND
           tq6810 /usr/sbin/httpd -DFOREGROUND
           tq6811 /usr/sbin/httpd -DFOREGROUND
           tq6812 /usr/sbin/httpd -DFOREGROUND
           mq6814 /usr/sbin/httpd -DFOREGROUND

 2月 05 04:31:16 localhost systemd[1]: Starting The Apache HTTP Server...
 2月 05 04:31:17 localhost httpd[6807]: AH00558: httpd: Could not reliably determine the server's fully qualifie
 2月 05 04:31:17 localhost systemd[1]: Started The Apache HTTP Server.
Hint: Some lines were ellipsized, use -l to show in full.
[root@localhost ~]# ps ax | grep httpd
 6807 ?        Ss     0:00 /usr/sbin/httpd -DFOREGROUND
 6809 ?        S      0:00 /usr/sbin/httpd -DFOREGROUND
 6810 ?        S      0:00 /usr/sbin/httpd -DFOREGROUND
 6811 ?        S      0:00 /usr/sbin/httpd -DFOREGROUND
 6812 ?        S      0:00 /usr/sbin/httpd -DFOREGROUND
 6814 ?        S      0:00 /usr/sbin/httpd -DFOREGROUND
 6825 pts/0    S+     0:00 grep --color=auto httpd
```

ブラウザで Web サーバーに接続します。Apache や PHP、PostgreSQL が動作しているマシン上で Firefox などの Web ブラウザを起動し、以下のアドレスを入力してアクセスします。

```
http://localhost/
```

同じマシンで Web ブラウザを実行できない場合には、別のマシンから Web サーバーに接続してください。

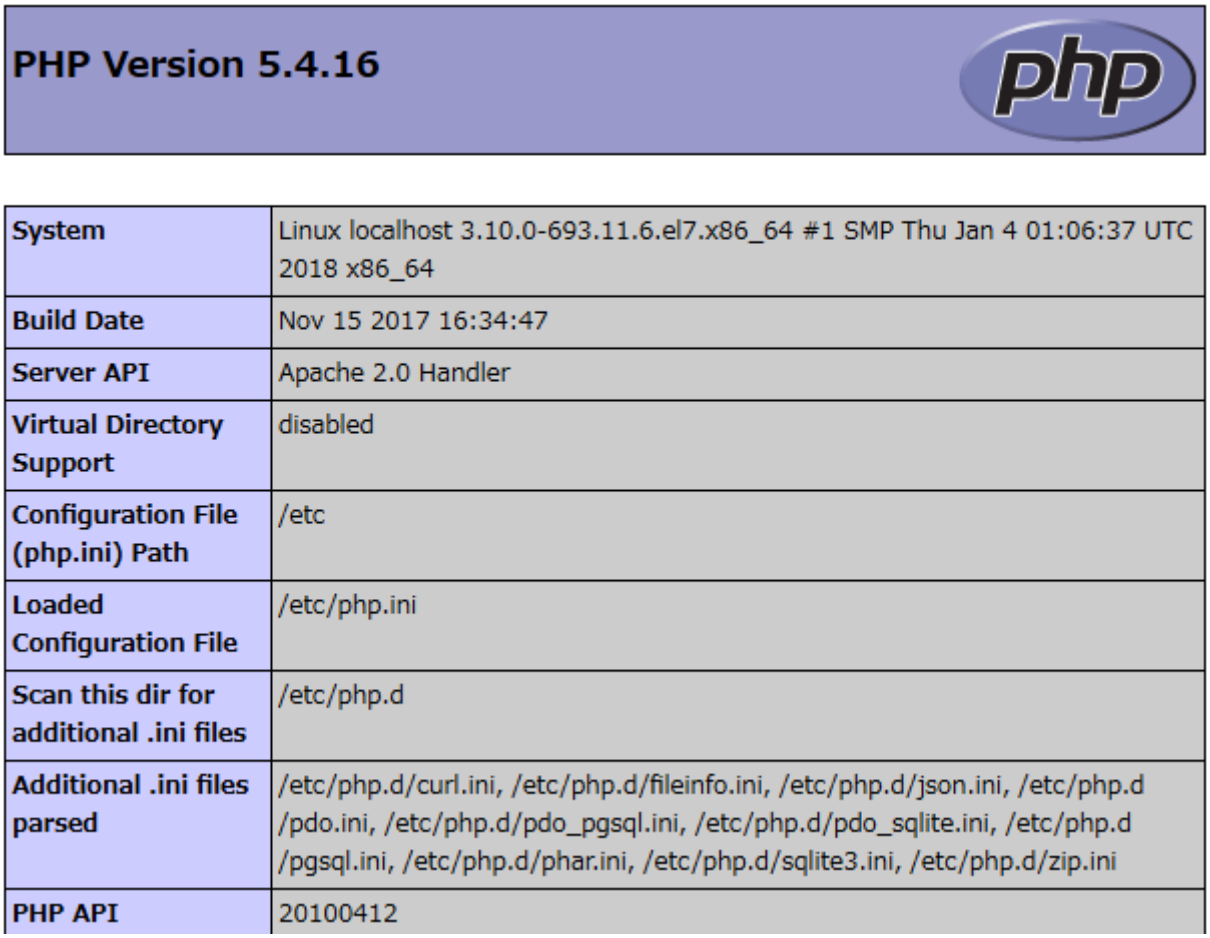
上記 URL にアクセスすると、`index.php` に書いておいた `phpinfo()` 関数が実行され、インストールされている PHP の詳細な情報が Web ページで表示されます。

10.4　PHP と PostgreSQL の連携

PHP から PostgreSQL に接続し、データを取得して HTML ファイルに組み込む簡単なアプリケーションを作成します。サンプルアプリケーションを作成して、実行してみましょう。サンプルで使用している PHP で用意された PostgreSQL 用の関数は以下の通りです。

■10.4.0.1　サンプルで使用している PostgreSQL 用関数

関数名	説明
pg_connect()	PostgreSQL に接続します。
pg_query()	SQL を実行します。
pg_last_error()	エラーを取得します。
pg_num_rows()	行数を取得します。
pg_num_fields()	列数を取得します。
pg_field_name()	列名を取得します。
pg_fetch_object()	行データを取得します。
pg_close()	PostgreSQL との接続を終了します。



PHP Version 5.4.16

System	Linux localhost 3.10.0-693.11.6.el7.x86_64 #1 SMP Thu Jan 4 01:06:37 UTC 2018 x86_64
Build Date	Nov 15 2017 16:34:47
Server API	Apache 2.0 Handler
Virtual Directory Support	disabled
Configuration File (php.ini) Path	/etc
Loaded Configuration File	/etc/php.ini
Scan this dir for additional .ini files	/etc/php.d
Additional .ini files parsed	/etc/php.d/curl.ini, /etc/php.d/fileinfo.ini, /etc/php.d/json.ini, /etc/php.d/pdo.ini, /etc/php.d/pdo_pgsql.ini, /etc/php.d/pdo_sqlite.ini, /etc/php.d/pgsql.ini, /etc/php.d/phar.ini, /etc/php.d/sqlite3.ini, /etc/php.d/zip.ini
PHP API	20100412

図 11　phpinfo

また、使用している他の（PostgreSQL 操作以外の）関数は以下の通りです。

関数名	説明
htmlspecialchars()	引数の文字列の中から HTML で特別な意味を持つ文字を正しく表示可能な形式に変換します。変換されるのは&や<>といった文字です。
strtoupper()	引数の文字列をすべて大文字にします。

10.4.1　データ検索ページの作成

サンプルアプリケーションを作成します。ossdb データベースに PostgreSQL ユーザー postgres で接続し、SQL 文「SELECT * FROM prod」を実行します。実行した結果は HTML のテーブルとして加工し、Web ブラウザで表示します。

以下のサンプルを適当なファイル名（例 selectProd.php）でディレクトリ/var/www/html に保存します。作業はユーザー root で行うか、ディレクトリ/var/www/html のパーミッションを変更して適当なユーザーで行ってください。データベースへの接続情報を記述している pg_connect 関数内のパスワードは、各自で設定済みの postgres ユーザーのパスワードに変更してください。

selectProd.php

```
<html>
<body>
<?php
$dbcon = pg_connect("dbname=ossdb user=postgres password=password");
if (!$dbcon) {
        die("<hr>pg_connect 失敗<hr>");
}
$sql =  "SELECT * FROM prod";
echo  "SQL=$sql<brx>\n";
$result = pg_query ($dbcon, $sql);
if (!$result) {
        pg_last_error($dbcon);
        die( "pg_exec 失敗<hr>");
}
$numrows = pg_num_rows($result);
$fnum = pg_num_fields($result);
echo "<table border>";
echo "<tr>";
for ($x = 0; $x < $fnum; $x++) {
        echo "<td><b>";
        echo htmlspecialchars(strtoupper(pg_field_name($result, $x)));
        echo "</b></td>";
}
echo "</tr>";
for ($i = 0; $i < $numrows; $i++) {
        $row = pg_fetch_object($result, $i);
        echo "<tr align='center'>";
        for ($x = 0; $x < $fnum; $x++) {
                $fieldname = pg_field_name($result, $x);
                echo "<td>";
                echo htmlspecialchars($row->$fieldname);
                echo "</td>";
        }
        echo"</tr>";
}
echo "</table>";
pg_close($dbcon);
?>
</body>
</html>
```

ファイルを作成したら、Web ブラウザからアクセスして結果を確認します。

```
http://localhost/selectProd.php
```

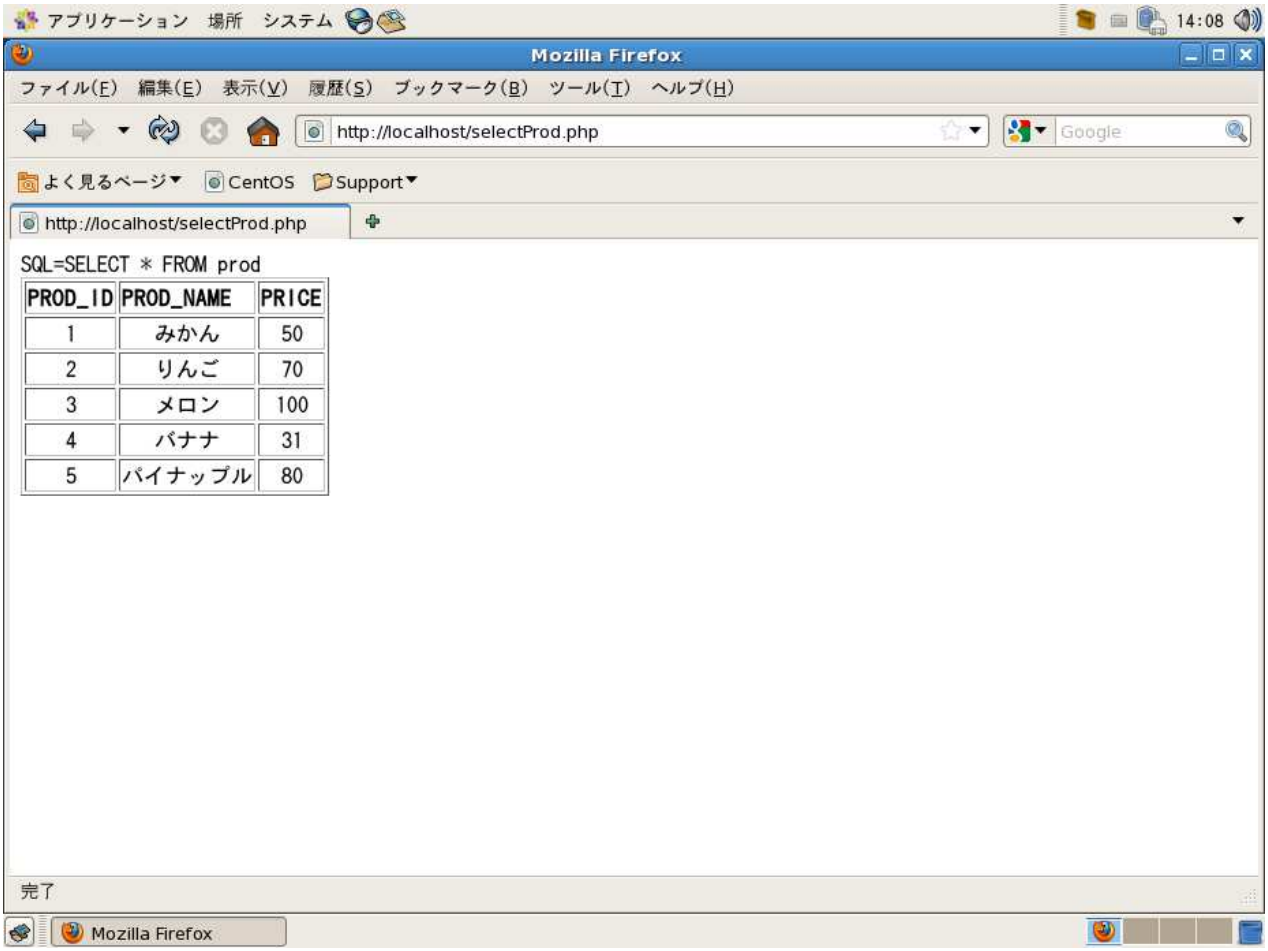


図 12　selectProd

10.4.2　フォームからのデータの取得方法

HTML フォームに入力されたデータを PHP で取得するには、$_POST 変数を使用します。HTML フォームから ACTION として呼び出された PHP は POST メソッドで送信された値を連想配列として保持しています。$_POST 変数はこの連想配列にアクセスするための変数です。動作を確認するために、HTML フォームの test.html と PHP プログラム test.php を/var/www/html ディレクトリに作成します。

test.html

```
<html>
<body>
<form action="test.php" method="post">
<input type=text name="foo">
<input type=submit>
</form>
</body>
</html>
```

test.php

```
<?php
echo htmlspecialchars($_POST['foo']);
?>
```

Web ブラウザから test.html を呼び出し、フォームに値を入力して実行ボタンをクリックします。PHP の echo は変数の値を

表示する命令ですので、フォームからの値を Web ブラウザに表示します。

```
http://localhost/test.html
```

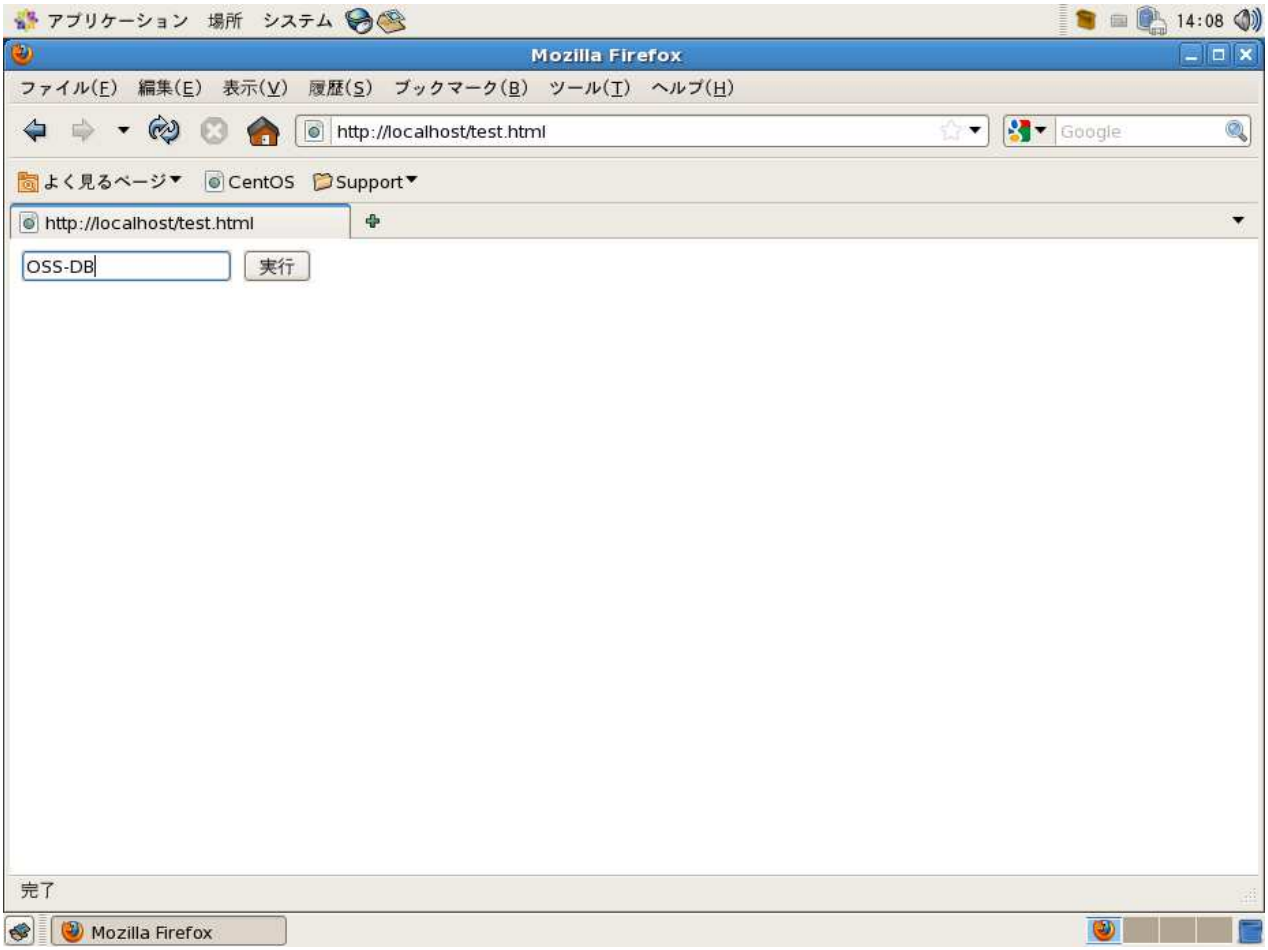


図 13　testhtml

10.4.3　フォームからの入力を SQL 文に組み込む

PHP で SQL 文を実行するには、一度 SQL 文を文字列変数に代入し、その変数を pg_query 関数に渡します。文字列変数に代入する際にフォームからの入力を SQL 文に組み込めば、フォーム入力に応じて動作を変更することができます。selectForm.php ではデータベースにアクセスしますので、pg_connect 関数内のパスワードは各自で設定したパスワードに変更します。

selectForm.html

```
<html>
<body>
検索したいテーブル名を入力して下さい。
<form action="selectForm.php" method="post">
<input type=text name="table">
<input type=submit>
</form>
</body>
</html>
```

selectForm.php

```
<html>
<body>
<?php
```

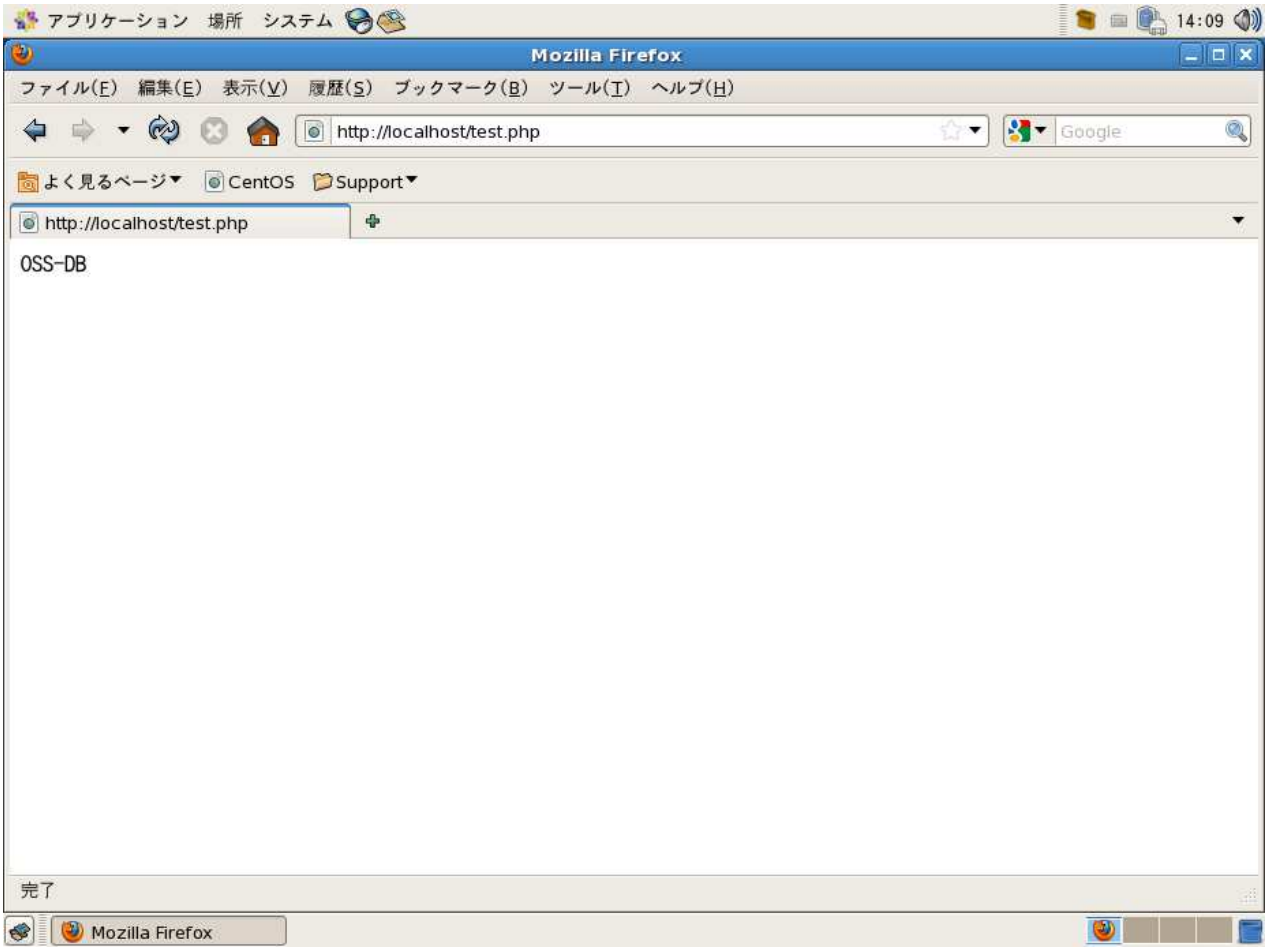


図 14　testphp

```
$dbcon = pg_connect("dbname=ossdb user=postgres password=password");
if (!$dbcon) {
        die("<hr>pg_connect 失敗<hr>");
}
$table = $_POST['table'];
$sql =  "SELECT * FROM $table";
echo  "SQL=$sql<brx>\n";
$result = pg_query ($dbcon, $sql);
if (!$result) {
        pg_last_error($dbcon);
        die( "pg_exec 失敗<hr>");
}
$numrows = pg_num_rows($result);
$fnum = pg_num_fields($result);
echo "<table border>";
echo "<tr>";
for ($x = 0; $x < $fnum; $x++) {
        echo "<td><b>";
        echo htmlspecialchars(strtoupper(pg_field_name($result, $x)));
        echo "</b></td>";
}
echo "</tr>";
for ($i = 0; $i < $numrows; $i++) {
```

```
        $row = pg_fetch_object($result, $i);
        echo "<tr align='center'>";
        for ($x = 0; $x < $fnum; $x++) {
                $fieldname = pg_field_name($result, $x);
                echo "<td>";
                echo htmlspecialchars($row->$fieldname);
                echo "</td>";
        }
        echo"</tr>";
}
echo "</table>";
pg_close($dbcon);
?>
</body>
</html>
```

作成したフォームにアクセスし、検索したい表の名前を入力します。

```
http://localhost/selectForm.html
```

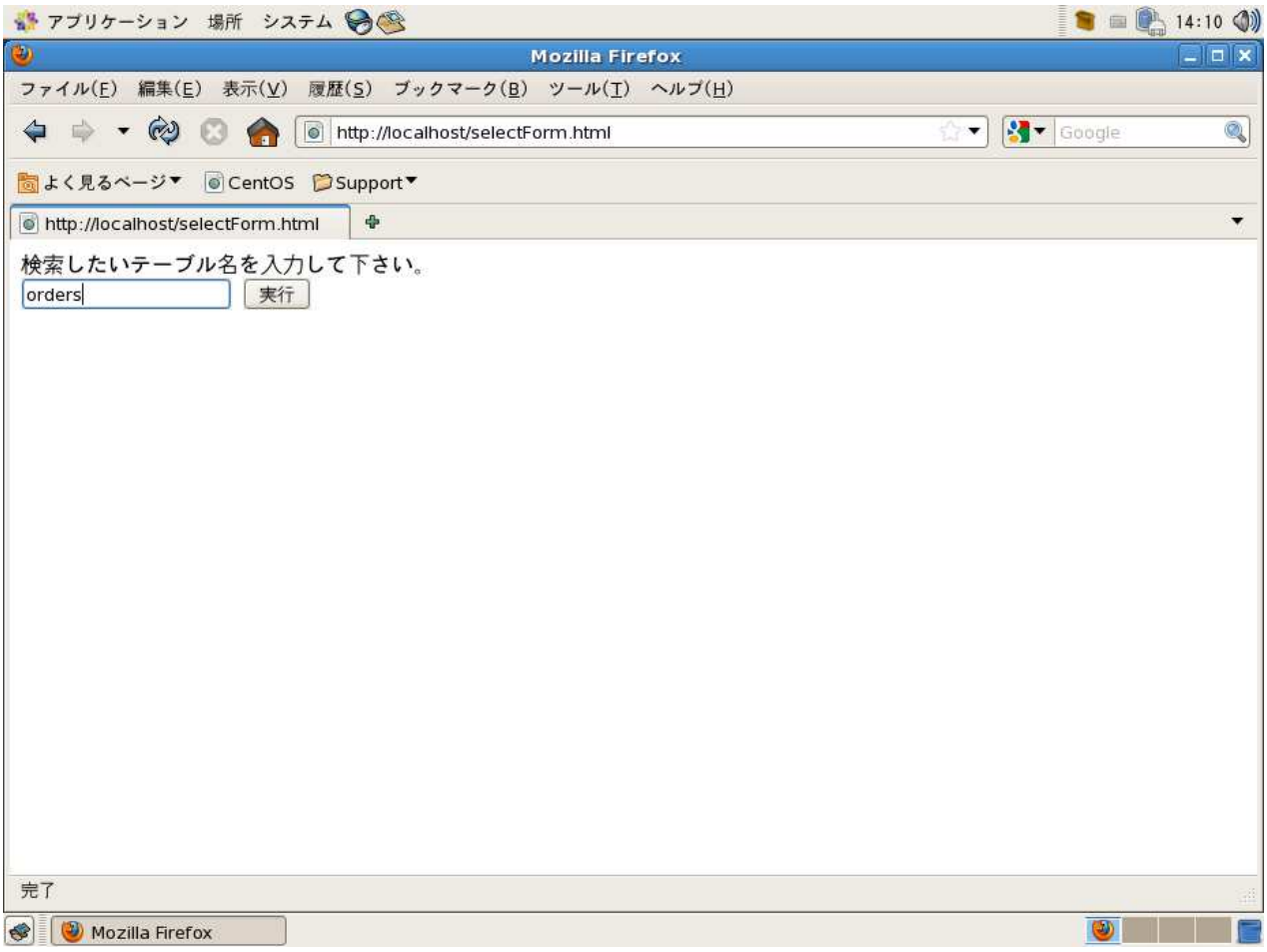


図 15　selectFormhtml

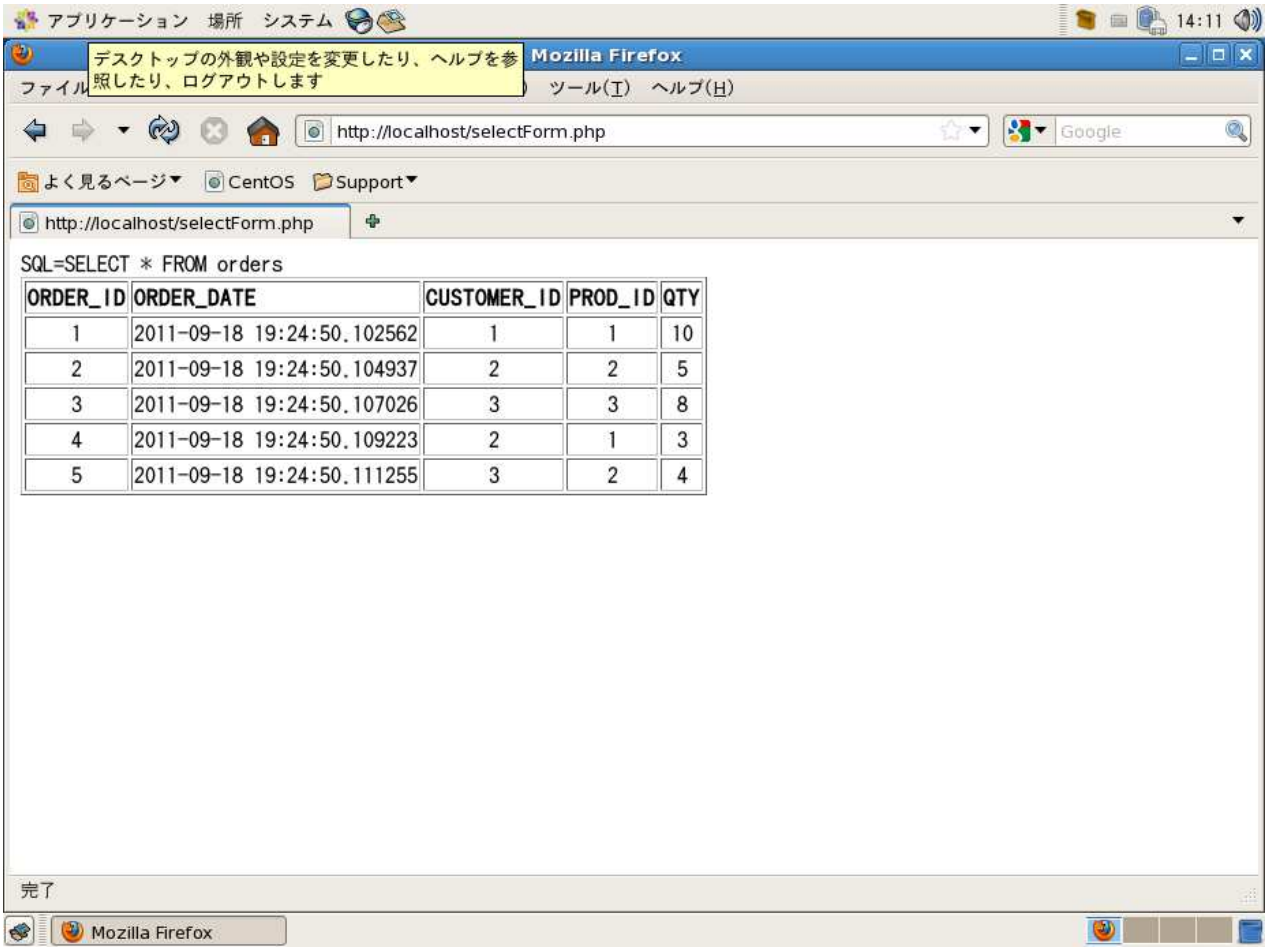


図 16　selectFormphp.png

11　実習環境の構築方法

11.1　OS のインストール

本書では Linux ディストリビューションは CentOS 7.4 64 ビット版を使用しています。インストール手順には特別なところはありませんので、インストーラーのデフォルト設定でインストールします。

11.1.1　OS ユーザーの作成

yum や RPM で PostgreSQL をインストールすると、OS ユーザー postgres が内部的に作成され各プログラムの実行権限が付与されます。この postgres ユーザーをあらかじめ作成しておくことで、OS ユーザーとしての設定（ホームディレクトリや環境変数など）の管理がしやすくなりますので本書ではそのように進めます。

以下では useradd コマンドで OS ユーザー postgres を作成、passwd コマンドでユーザーのパスワードを設定しています。その後、postgres ユーザーにログインしてプロンプトの表示やホームディレクトリ位置を確認し、ログアウトしておきます。

```
[root@localhost ~]# useradd postgres
[root@localhost ~]# passwd postgres
ユーザー postgres のパスワードを変更。
新しいパスワード:
新しいパスワードを再入力してください:
passwd: すべての認証トークンが正しく更新できました。
[root@localhost ~]# su - postgres
[postgres@localhost ~]$ pwd
```

```
/home/postgres
[postgres@localhost ~]$ exit
[root@localhost ~]#
```

11.1.2　セキュリティの設定

セキュリティの設定では、外部からの攻撃などを受けない環境であることを確認した上で、ファイアーウォールや SELinux は無効にしていることを想定しています。

```
[root@localhost ~]# setenforce 0
[root@localhost ~]# systemctl stop firewalld.service
```

11.2　PostgreSQL のインストール

CentOS では PostgreSQL のバージョン 10 が標準で提供されていないので、yum の外部リポジトリを利用して、PostgreSQL 10 をインストールします。本書では 2018 年 1 月現在の最新版、PostgreSQL 10.1 をインストールします。

11.2.1　手順 1　yum リポジトリの設定

PostgreSQL の最新版は PostgreSQL 開発コミュニティ内の PostgreSQL RPM Building Project により配布されています。web ブラウザで以下の URL にアクセスすると現在サポートされているバージョンの yum リポジトリ設定用の RPM パッケージが配布されています。

PostgreSQL RPM Building Project * https://yum.postgresql.org/repopackages.php

注意：開発中の最新版（α版、β版や RC 版）が同ページで配布されていることがありますが、不具合修正の後、数か月後に正式リリースされるものですので、新機能検証などの目的以外では利用しないことを推奨します。

利用する OS ディストリビューションに対応したリンクを右クリックし、リンクの URL をコピーしておきます。例えば、CentOS 7 系、PostgreSQL 10 では、以下のような RPM パッケージが直接リンクされています。https://download.postgresql.org/pub/repos/yum/10/redhat/rhel-7-x86_64/pgdg-centos10-10-2.noarch.rpm

これを Linux サーバー上の `yum install` コマンドで指定します。

```
[root@localhost ~]# yum install https://download.postgresql.org/pub/repos/yum/10/redhat/rhel-7-x86_64/pgdg-cento
読み込んだプラグイン:fastestmirror, langpacks
pgdg-centos10-10-2.noarch.rpm                                                    | 4.6 kB  00:0

(中略)

Is this ok [y/d/N]: y
Downloading packages:
Running transaction check
Running transaction test
Transaction test succeeded
Running transaction
  インストール中          : pgdg-centos10-10-2.noarch
  検証中                  : pgdg-centos10-10-2.noarch

インストール:
  pgdg-centos10.noarch 0:10-2

完了しました!
```

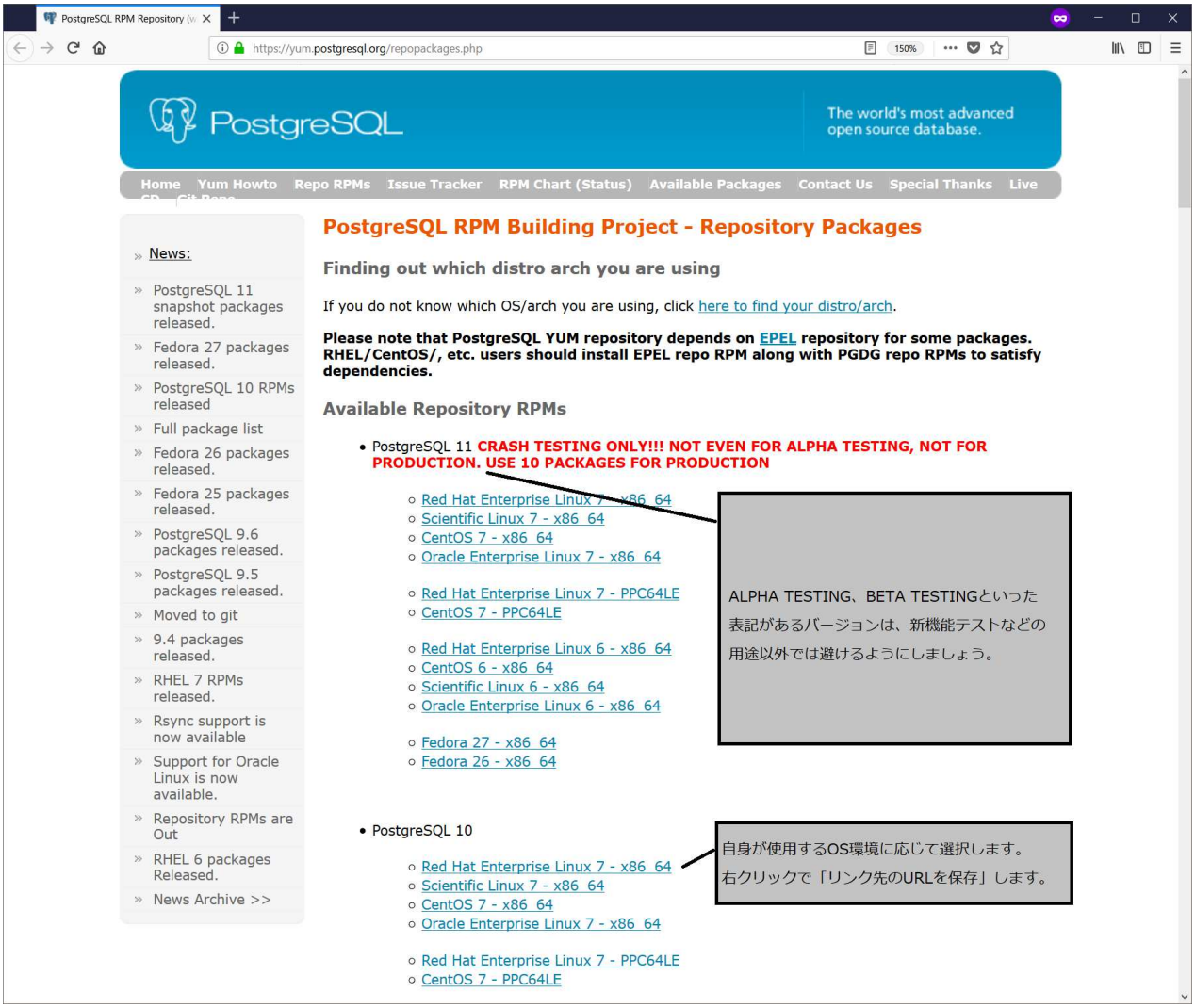


図 17　PostgreSQL RPM Building Project

11.2.2　手順 2　PostgreSQL のインストール

手順 1 で PostgreSQL 10 の yum リポジトリが利用できるようになりましたので、必要なパッケージを指定してインストールします。

```
[root@localhost ~]# yum install postgresql10 postgresql10-server postgresql10-contrib postgresql10-devel
読み込んだプラグイン:fastestmirror, langpacks
pgdg10                                                   | 4.1 kB  00:00:00
(1/2): pgdg10/7/x86_64/group_gz                          |  245 B  00:00:01
(2/2): pgdg10/7/x86_64/primary_db                        | 145 kB  00:00:01
Loading mirror speeds from cached hostfile
 * base: ftp.iij.ad.jp
 * extras: ftp.iij.ad.jp
 * updates: ftp.iij.ad.jp
依存性の解決をしています
--> トランザクションの確認を実行しています。
---> パッケージ postgresql10.x86_64 0:10.1-1PGDG.rhel7 を インストール
--> 依存性の処理をしています: postgresql10-libs(x86-64) = 10.1-1PGDG.rhel7 のパッケージ: postgresql10-10.1-1PGDG.rhel
--> 依存性の処理をしています: libpq.so.5()(64bit) のパッケージ: postgresql10-10.1-1PGDG.rhel7.x86_64
---> パッケージ postgresql10-contrib.x86_64 0:10.1-1PGDG.rhel7 を インストール
```

```
---> パッケージ postgresql10-devel.x86_64 0:10.1-1PGDG.rhel7 を インストール
--> 依存性の処理をしています: libicu-devel のパッケージ: postgresql10-devel-10.1-1PGDG.rhel7.x86_64
---> パッケージ postgresql10-server.x86_64 0:10.1-1PGDG.rhel7 を インストール
--> トランザクションの確認を実行しています。
---> パッケージ libicu-devel.x86_64 0:50.1.2-15.el7 を インストール
---> パッケージ postgresql10-libs.x86_64 0:10.1-1PGDG.rhel7 を インストール
--> 依存性解決を終了しました。

依存性を解決しました

================================================================================================
 Package                    アーキテクチャー     バージョン          リポジトリー     容量
================================================================================================
インストール中:
 postgresql10               x86_64               10.1-1PGDG.rhel7    pgdg10          1.5 M
 postgresql10-contrib       x86_64               10.1-1PGDG.rhel7    pgdg10          587 k
 postgresql10-devel         x86_64               10.1-1PGDG.rhel7    pgdg10          2.0 M
 postgresql10-server        x86_64               10.1-1PGDG.rhel7    pgdg10          4.3 M
依存性関連でのインストールをします:
 libicu-devel               x86_64               50.1.2-15.el7       base            702 k
 postgresql10-libs          x86_64               10.1-1PGDG.rhel7    pgdg10          347 k

トランザクションの要約
================================================================================================
インストール  4 パッケージ（+2 個の依存関係のパッケージ）

総ダウンロード容量: 9.3 M
インストール容量: 40 M
Is this ok [y/d/N]: y
Downloading packages:
警告: /var/cache/yum/x86_64/7/base/packages/libicu-devel-50.1.2-15.el7.x86_64.rpm: ヘッダー V3 RSA/SHA256 Signatu
libicu-devel-50.1.2-15.el7.x86_64.rpm の公開鍵がインストールされていません
(1/6): libicu-devel-50.1.2-15.el7.x86_64.rpm                   | 702 kB  00:00:00
(2/6): postgresql10-contrib-10.1-1PGDG.rhel7.x86_64.rpm        | 587 kB  00:00:02
(3/6): postgresql10-10.1-1PGDG.rhel7.x86_64.rpm                | 1.5 MB  00:00:03
(4/6): postgresql10-libs-10.1-1PGDG.rhel7.x86_64.rpm           | 347 kB  00:00:00
(5/6): postgresql10-devel-10.1-1PGDG.rhel7.x86_64.rpm          | 2.0 MB  00:00:01
(6/6): postgresql10-server-10.1-1PGDG.rhel7.x86_64.rpm         | 4.3 MB  00:00:03
------------------------------------------------------------------------------------------------
合計                                                1.4 MB/s | 9.3 MB  00:00:06
file:///etc/pki/rpm-gpg/RPM-GPG-KEY-CentOS-7 から鍵を取得中です。
Importing GPG key 0xF4A80EB5:
 Userid     : "CentOS-7 Key (CentOS 7 Official Signing Key) <security@centos.org>"
 Fingerprint: 6341 ab27 53d7 8a78 a7c2 7bb1 24c6 a8a7 f4a8 0eb5
 Package    : centos-release-7-4.1708.el7.centos.x86_64 (@anaconda)
 From       : /etc/pki/rpm-gpg/RPM-GPG-KEY-CentOS-7
上記の処理を行います。よろしいでしょうか？ [y/N]y
Running transaction check
Running transaction test
Transaction test succeeded
Running transaction
  インストール中          : postgresql10-libs-10.1-1PGDG.rhel7.x86_64                    1/6
```

```
  インストール中          : postgresql10-10.1-1PGDG.rhel7.x86_64                    2/6
  インストール中          : libicu-devel-50.1.2-15.el7.x86_64                       3/6
  インストール中          : postgresql10-devel-10.1-1PGDG.rhel7.x86_64              4/6
  インストール中          : postgresql10-server-10.1-1PGDG.rhel7.x86_64             5/6
  インストール中          : postgresql10-contrib-10.1-1PGDG.rhel7.x86_64            6/6
  検証中                  : postgresql10-server-10.1-1PGDG.rhel7.x86_64             1/6
  検証中                  : postgresql10-libs-10.1-1PGDG.rhel7.x86_64               2/6
  検証中                  : postgresql10-devel-10.1-1PGDG.rhel7.x86_64              3/6
  検証中                  : postgresql10-contrib-10.1-1PGDG.rhel7.x86_64            4/6
  検証中                  : libicu-devel-50.1.2-15.el7.x86_64                       5/6
  検証中                  : postgresql10-10.1-1PGDG.rhel7.x86_64                    6/6

インストール:
  postgresql10.x86_64 0:10.1-1PGDG.rhel7          postgresql10-contrib.x86_64 0:10.1-1PGDG.rhel7
  postgresql10-devel.x86_64 0:10.1-1PGDG.rhel7    postgresql10-server.x86_64 0:10.1-1PGDG.rhel7

依存性関連をインストールしました:
  libicu-devel.x86_64 0:50.1.2-15.el7             postgresql10-libs.x86_64 0:10.1-1PGDG.rhel7

完了しました!
```

インストールが完了すると、以下の通りディレクトリとバイナリが配置されています。

```
[postgres@localhost ~]$ ls /usr/pgsql-10/
bin  doc  include  lib  share
[postgres@localhost ~]$ ls /usr/pgsql-10/bin
clusterdb   ecpg               pg_config       pg_isready      pg_rewind       pg_waldump                postm
createdb    initdb             pg_controldata  pg_receivewal   pg_standby      pgbench                   psql
createuser  oid2name           pg_ctl          pg_recvlogical  pg_test_fsync   postgres                  reind
dropdb      pg_archivecleanup  pg_dump         pg_resetwal     pg_test_timing  postgresql-10-check-db-dir  vacuu
dropuser    pg_basebackup      pg_dumpall      pg_restore      pg_upgrade      postgresql-10-setup       vacuu
```

11.2.3　手順 3　PostgreSQL 利用環境の初期設定

インストール直後はデータベースが作成されておらず、次のステップ以降で利用者が作成します。インストールした PostgreSQL は、OS ユーザー postgres が初期化ユーザーとして管理権限を持っているので、su コマンドでユーザー postgres に変更して操作を行います。PostgreSQL に対する各種操作がしやすいように OS 側の設定を行います。なお、データディレクトリの位置はデータベース作成時に指定できますが、その際に本項の設定（環境変数と起動スクリプト）が正しく設定されている必要があります。本書ではデフォルトの位置（/var/lib/pgsql/10/data）に作成することとします。

■11.2.3.1　環境変数の設定

環境変数	説明
PGDATA	データディレクトリ位置を指定します。
PGHOME	PostgreSQL のインストールディレクトリを指定します。
PATH	PostgreSQL インストールディレクトリ配下の bin を指定します。

以下では、postgresql ユーザーの環境変数設定ファイル .bash_profile を編集し、PGDATA、PGHOME、PATH 環境変数を追加しています。

```
[root@localhost ~]# su - postgres
[postgres@localhost ~]$ vi .bash_profile
---------
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
        . ~/.bashrc
fi

# User specific environment and startup programs

PATH=$PATH:$HOME/.local/bin:$HOME/bin

export PATH

### edit for PostgreSQL 10
export PGDATA=/var/lib/pgsql/10/data
export PGHOME=/usr/pgsql-10
export PATH=$PGHOME/bin:.:$PATH
-------
[postgres@localhost ~]$ source .bash_profile
```

■11.2.3.2　起動スクリプトの確認

yum や RPM で PostgreSQL をインストールすると、起動停止スクリプトが自動作成されます。先の手順で設定した環境変数 PGDATA が Location of database directory と一致していることを確認します。本書の範囲では変更する必要はありません。

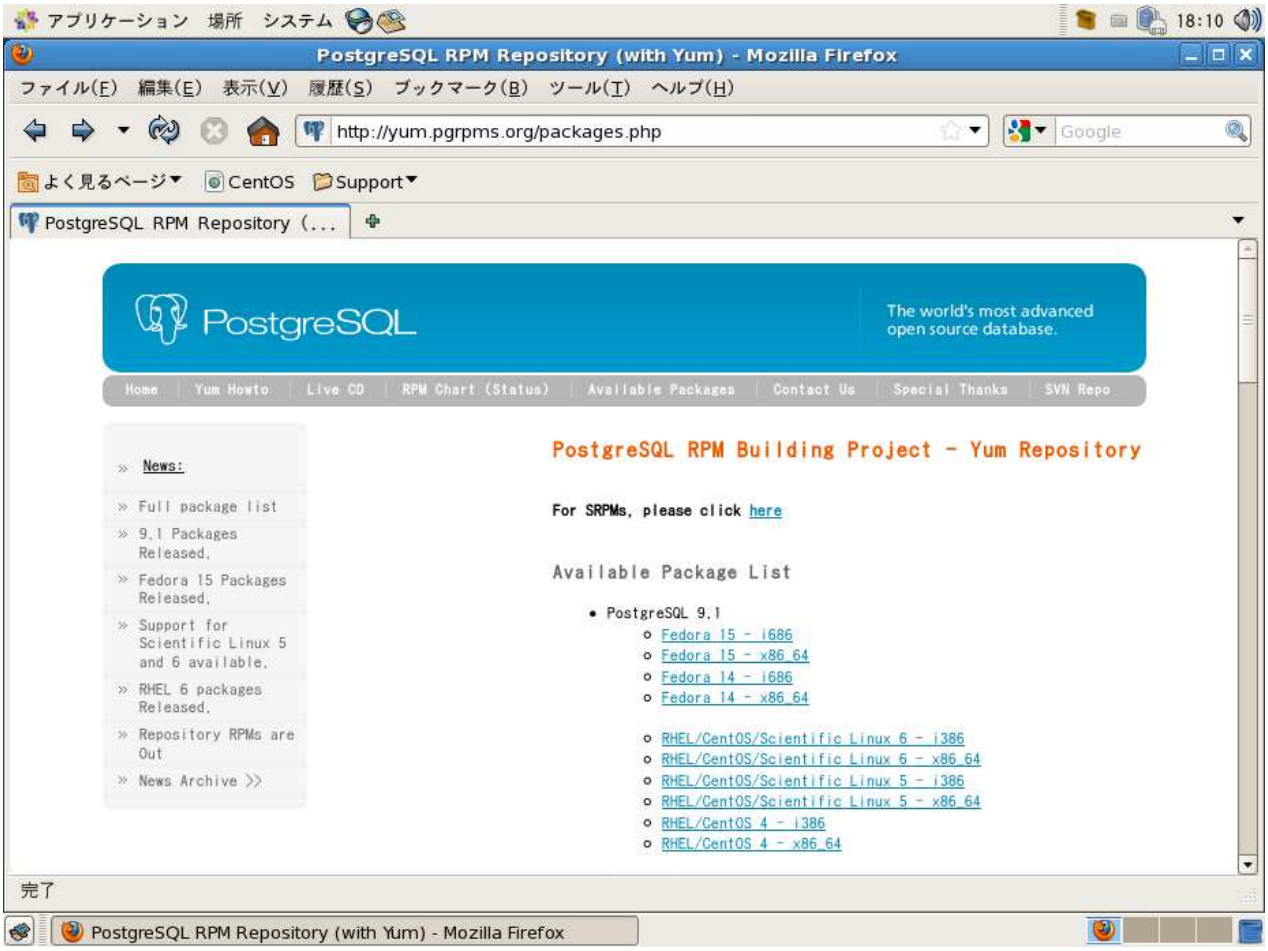
```
[root@localhost ~]# vi /usr/lib/systemd/system/postgresql-10.service
------
# Location of database directory
Environment=PGDATA=/var/lib/pgsql/10/data/
------
```

11.2.4　参考　yum を使わないインストール

インターネットへの接続が行えないなどの制限がある場合には、Web サイトから以下のパッケージをダウンロードしてサーバーに配置し、RPM コマンドでインストールしてください。

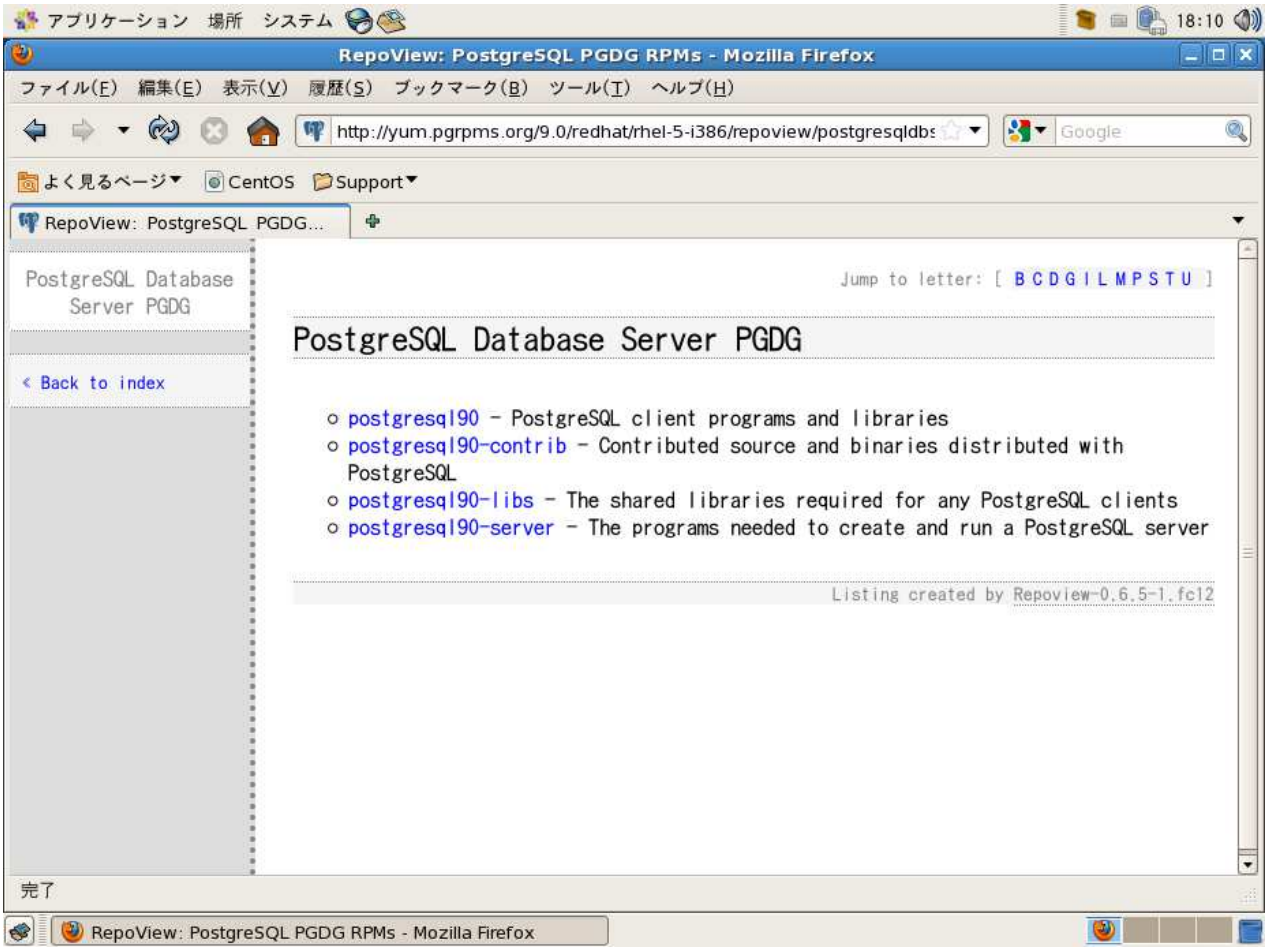
11.2.4.0.1　ダウンロード Web ページ

http://yum.pgrpms.org/packages.php



11.2.4.0.2　CentOS 7 64 ビット版用ダウンロード Web ページ

https://yum.postgresql.org/10/redhat/rhel-7-x86_64/repoview/postgresqldbserver10.group.html



以下のパッケージをダウンロードしサーバーに配置します。

パッケージ名	説明
postgresql10	PostgreSQL を利用する上で必須のクライアントプログラムやライブラリ
postgresql10-libs	PostgreSQL を利用する上で必須の共有ライブラリ
postgresql10-server	サーバープログラムの本体
postgresql10-contrib	拡張機能（本書の範囲では必須ではありません）

各パッケージのリンク先にはさらに複数マイナーバージョンが配布されている場合があります。本書ではいずれも 10.1 を利用しています。

■11.2.4.1　RPM コマンドでインストール

実際にダウンロードするファイルは `postgresql10-10.1-1PGDG.rhel7.x86_64.rpm` のような RPM 形式です。RPM コマンドでインストールします。postgresql10-libs、postgresql10-server、postgresql10-contrib も同様にインストールします。

```
[root@localhost ~]# cd <ファイル配置先ディレクトリ>
[root@localhost ~]# rpm -ivh postgresql10-10.1-1PGDG.rhel7.x86_64.rpm
```

11.3　データベースの初期化

データベースクラスタを作成します。この作業をデータベースの初期化と呼び、インストール後に 1 回だけ行います。

11.3.1　データベースクラスタと initdb コマンド

PostgreSQL が管理するデータベースそのもの（実体は OS 上のファイル）や各種設定ファイル、変更履歴ファイル、ログファイルなどをひとまとめにしたものをデータベースクラスタと呼びます。初期化とはデータベースクラスタを構成するすべてのファイルやディレクトリを新規作成することを指します。データベースの初期化は `initdb` コマンドを使用し、日本語環境で利用するうえで推奨されている`-E utf8` および`--no-locale` オプションを指定してデータベースを初期化します。

11.3.2　データディレクトリ

データベースクラスタを構成するすべてのファイルやディレクトリは 1 つのディレクトリ配下にまとめて配置されます。このディレクトリをデータディレクトリと呼び、環境変数 PGDATA で指定されます。initdb 時、環境変数 PGDATA が参照され、ここで指定した位置にデータベースクラスタが作成されます。

11.3.3　initdb コマンドの実行

以下の例では、前述の手順の従ってデータディレクトリ位置が環境変数 PGDATA に設定された状態で initdb コマンドを実行しています。initdb 完了後、cd コマンドでデータディレクトリに移動し、作成されたファイルを確認しています。

```
[postgres@localhost ~]$ env | grep PGDATA
PGDATA=/var/lib/pgsql/10/data
[postgres@localhost ~]$ initdb -E utf8 --no-locale
データベースシステム内のファイルの所有者は"postgres"ユーザでした。
このユーザがサーバプロセスを所有しなければなりません。

データベースクラスタはロケール"C"で初期化されます。
デフォルトのテキスト検索設定は english に設定されました。

データベージのチェックサムは無効です。

ディレクトリ/var/lib/pgsql/10/data の権限を設定しています ... ok
サブディレクトリを作成しています ... ok
デフォルトの max_connections を選択しています ... 100
デフォルトの shared_buffers を選択しています ... 128MB
selecting dynamic shared memory implementation ... posix
設定ファイルを作成しています ... ok
running bootstrap script ... ok
performing post-bootstrap initialization ... ok
データをディスクに同期しています...ok

警告: ローカル接続向けに"trust"認証が有効です。
pg_hba.conf を編集する、もしくは、次回 initdb を実行する時に-A オプショ
ン、または、--auth-local および--auth-host を使用することで変更するこ
とができます。

Success. You can now start the database server using:

    pg_ctl -D /var/lib/pgsql/10/data -l logfile start

[postgres@localhost ~]$ cd $PGDATA
[postgres@localhost data]$ ls
PG_VERSION  pg_commit_ts  pg_ident.conf  pg_notify    pg_snapshots  pg_subtrans  pg_wal                postgresq
base        pg_dynshmem   pg_logical     pg_replslot  pg_stat       pg_tblspc    pg_xact
global      pg_hba.conf   pg_multixact   pg_serial    pg_stat_tmp   pg_twophase  postgresql.auto.conf
```

11.4　データベースを起動

PostgreSQL の起動・停止には systemctl コマンドを使用します。

```
[root@localhost ~]# systemctl start postgresql-10.service
```

PostgreSQL が正しく起動されている場合、ステータスは以下のようになります。

```
[root@localhost ~]# systemctl status postgresql-10.service
● postgresql-10.service - PostgreSQL 10 database server
   Loaded: loaded (/usr/lib/systemd/system/postgresql-10.service; disabled; vendor preset: disabled)
   Active: active (running) since 月 2018-01-22 01:59:14 JST; 6s ago
     Docs: https://www.postgresql.org/docs/10/static/
  Process: 22602 ExecStartPre=/usr/pgsql-10/bin/postgresql-10-check-db-dir ${PGDATA} (code=exited, status=0/SUCC
 Main PID: 22611 (postmaster)
   CGroup: /system.slice/postgresql-10.service
           tq22611 /usr/pgsql-10/bin/postmaster -D /var/lib/pgsql/10/data/
           tq22614 postgres: logger process
           tq22616 postgres: checkpointer process
           tq22617 postgres: writer process
           tq22618 postgres: wal writer process
           tq22619 postgres: autovacuum launcher process
           tq22620 postgres: stats collector process
           mq22621 postgres: bgworker: logical replication launcher
(以下略)
```

デフォルトでは手動起動になっているので、システムの起動毎に自動的に起動したい場合には systemctl で enable サブコマンドを指定します。自動起動を無効にする場合は disable を指定します。

```
[root@localhost ~]# systemctl enable postgresql-10.service
Created symlink from /etc/systemd/system/multi-user.target.wants/postgresql-10.service to /usr/lib/systemd/syste
[root@localhost ~]# systemctl list-unit-files | grep postgres
postgresql-10.service                         enabled
```

11.5　動作の確認

データベースの動作確認を行います。PostgreSQL サーバーに対するすべての操作は postgres ユーザーで実施します。

```
[root@localhost ~]# su - postgres
```

psql に-l オプションを付けて実行し、作成されているデータベースを確認します。

```
[postgres@localhost ~]$ psql -l
                                  List of databases
   Name    |  Owner   | Encoding | Collate | Ctype |   Access privileges
-----------+----------+----------+---------+-------+-----------------------
 postgres  | postgres | UTF8     | C       | C     |
 template0 | postgres | UTF8     | C       | C     | =c/postgres          +
           |          |          |         |       | postgres=CTc/postgres
 template1 | postgres | UTF8     | C       | C     | =c/postgres          +
           |          |          |         |       | postgres=CTc/postgres
(3 rows)
```

12　付録　実習の準備方法

実習で使用するデータベースを作成し、データベースに接続します。そして、表を作成し、初期データを入力します。

12.1　データベースの作成

実習用のデータベース ossdb を作成します。データベースの作成は OS ユーザー postgres で行います。作成後、接続できることを確認しておきます。

```
[root@localhost ~]# su - postgres
[postgres@localhost ~]$ createdb ossdb
[postgres@localhost ~]$ psql ossdb
psql (10.1)
Type "help" for help.

ossdb=#
```

12.2　表の作成

表を作成します。prod 表、customer 表、orders 表の 3 つを作成します。

以下の SQL 文を psql を実行している端末にコピー＆ペーストすれば、必要な表が作成されます。

```
CREATE TABLE prod
(prod_id   integer,
 prod_name text,
 price     integer);

CREATE TABLE customer
 (customer_id   integer,
  customer_name text);

CREATE TABLE orders
 (order_id    integer,
  order_date  timestamp,
  customer_id integer,
  prod_id     integer,
  qty         integer);
```

以下は実行例です。

```
ossdb=# CREATE TABLE prod
 (prod_id   integer,
  prod_name text,
  price     integer);
CREATE TABLE
(略)
```

12.3　データの入力

作成した表に初期データを入力します。以下の SQL 文を psql を実行している端末にコピー＆ペーストすれば、初期データがそれぞれの表に入力されます。

```
-- 複数行を同時に INSERT
INSERT INTO customer(customer_id,customer_name) VALUES
 (1,'佐藤商事'),
 (2,'鈴木物産'),
 (3,'高橋商店');

INSERT INTO prod(prod_id,prod_name,price) VALUES
 (1,'みかん',50),
 (2,'りんご',70),
 (3,'メロン',100);

-- 一行ずつ個別に INSERT し、now() 関数で取得される時刻に差をつける
INSERT INTO orders(order_id,order_date,customer_id,prod_id,qty) VALUES (1,now(),1,1,10);
INSERT INTO orders(order_id,order_date,customer_id,prod_id,qty) VALUES (2,now(),2,2,5);
INSERT INTO orders(order_id,order_date,customer_id,prod_id,qty) VALUES (3,now(),3,3,8);
INSERT INTO orders(order_id,order_date,customer_id,prod_id,qty) VALUES (4,now(),2,1,3);
INSERT INTO orders(order_id,order_date,customer_id,prod_id,qty) VALUES (5,now(),3,2,4);
```

以下は実行例です。

```
ossdb=# -- 複数行を同時に INSERT
ossdb=# INSERT INTO customer(customer_id,customer_name) VALUES
 (1,'佐藤商事'),
 (2,'鈴木物産'),
 (3,'高橋商店');
INSERT 0 3
(略)
```

オープンソースデータベース標準教科書 - PostgreSQL -　(Ver. 2.0.1)

2022 年　8 月　9 日 v2.0.1 版発行
2019 年　3 月　1 日 v2.0.0 版発行
2011 年 10 月 14 日 v1.0.1 版発行
2011 年 10 月　5 日 v1.0.0 版発行

発行所 LPI-Japan