データベースの操作、作成、管理の基礎を学べる

オープンソース データベース標準教科書

– PostgreSQL –

Ver. 3.0.1

Silver/Gold

OSS-DB技術者認定

https://oss-db.jp

open your NEXT future

LPI-JAPAN

Linux Professional Certification

https://lpi.or.jp

オープンソースデータベース標準教科書

2025年5月1日　Ver.3.0.1
LPI-Japan　発行

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」（以下、本教科書）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。

本教科書は、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」「Linux システム管理標準教科書」「高信頼システム構築標準教科書」「Linux セキュリティ標準教科書」といった標準教科書シリーズの 1 つとなります。

公開にあたっては、本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。

本教科書の最新情報は以下の Web ページをご参照ください。

オープンソースデータベース標準教科書

`https://oss-db.jp/ossdbtext`

本教科書の目的

本教科書の目的は、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 文を使ったデータベースの操作や、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。

PostgreSQL を扱うデータベースエンジニアとしてのスキルを証明する認定「OSS-DB Silver」の取得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

本教科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

本教科書では、PostgreSQL バージョン 13 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。

「Linux サーバー構築標準教科書」と共通の環境

本教科書は、姉妹書である「Linux サーバー構築標準教科書」で構築した仮想マシンおよびインストールした OS を使って実習を進めることを想定しています。具体的な環境の構築方法については「Linux サーバー構築標準教科書」の 1 章から 3 章までを参照してください。

Linux サーバー構築標準教科書

`https://linuc.org/textbooks/server/`

図 1: https://linuc.org/textbooks/server/

仮想マシンを利用

仮想マシンを利用して学習環境を構築します。仮想マシンを利用すると、Windows や Linux、macOS 上の仮想マシンに Linux をインストールし、動作させることができます。

「Linux サーバー構築標準教科書」では、VirtualBox を Windows 上で実行しています。本教科書では、どの仮想マシンソフトウェアを実行しても実習を行うことができます。

OS

本教科書では、Linux ディストリビューションとして AlmaLinux のバージョン 9.3 を利用します。

実習例では Intel/AMD x86_64 アーキテクチャに対応したバージョンを利用していますが、ARM 版などその他のアーキテクチャに対応したバージョンでも実習を行うことができます。

PostgreSQL のバージョン 13 が動作すれば、その他のディストリビューションや OS でもかまいません。

Linux 以外の OS で動作する PostgreSQL のインストーラーをダウンロードするには、以下の Web ページをご参照ください。

```
https://www.postgresql.jp/download
```

ネットワーク

実習を行うネットワークはインターネットに接続できることを前提としています。

インターネットに接続できない場合には、OS のインストール時に実習で必要となるソフトウェアをあらかじめインストールすることで実習を進めることができます。OS インストール手順解説の指示に従って事前にソフトウェアをインストールしてください。既に OS をインストールしている場合には、OS インストールに使用した ISO イメージをマウントし、必要となるパッケージからインストールが行えます。

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。AlmaLinux、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。

全体の流れ

本教科書では、以下の通りに実習を進めます。

1 章実習環境の構築方法

PostgreSQL のインストールや、実習で使用するデータベースの作成を行います。

2 章 SQL によるデータベースの操作基礎編

データベースの基本的な操作を実習を通して学びます。

3 章 データ型

データベースで取り扱う様々なデータ型について学びます。

4 章 表

データベースでデータを格納する表について学びます。

5 章 基礎編演習

前半で学んだ内容について演習形式で復習します。

6 章 SQL によるデータベースの操作応用編

より詳細なデータベースの操作について学びます。

7 章データベース定義の応用

表の作成などに付随する詳細な機能について学びます。

8 章マルチユーザーでの利用

複数のユーザーでデータベースを扱う際に必要となるユーザーの作成や認証について学びます。

9 章パフォーマンスチューニング

データベースの性能について学びます。

10 章バックアップとリストア

データのバックアップとリストアの方法について学びます。

執筆者・制作担当者紹介

本教科書は、オープンなプロジェクト形式で開発を行っています。企画段階から意見交換を行い、事前の技術的な調査、執筆、レビューなどをプロジェクトのメンバーで分担して行っています。

宮原徹（バージョン 3 執筆担当／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。

2011 年に第 1 版を公開してから 10 年以上経ちましたが、標準的な SQL を解説することを心がけて執筆したため、現在でも内容はほとんどそのままで第 3 版として改訂を行いました。現在では仮想マシンやクラウドを使って自学自習する方が増えたため、実習で使うデータベースの作成を最初に行うなど、現在の状況に合わせた実習の流れにしてあります。

SQL 文などはコピー＆ペーストでそのまま使えるようにしてありますので、是非 PDF 版を使ってまずは動作することを確認してみる。一回通しでやってみた後は、再度 SQL 文をあえて手入力する「写経」スタイルで復習してみてください。

教科書開発に協力いただいた皆さん (第 3 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、Slack 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。

- 板橋 章夫
- 井上 潮（東京電機大学）
- 鯨井 貴博 (株式会社ゼウス・エンタープライズ)
- 竹本 季史（インターノウス株式会社）
- 中 翔（フリーランス）
- 福永 昭臣（株式会社ボールド）

また、これまでバージョン 1 からバージョン 2 まで、多くの執筆者、レビュアー、そして利用者の皆様からフィードバックをいただきました。厚く御礼申し上げます。

著作権

使用に関する権利

図 2: CC BY-NC-ND 4.0

表示

本教科書は、特定非営利活動法人エルピーアイジャパンに著作権が帰属するものであることを表示してください。

非営利

本教科書は、非営利目的で教材として自由に利用することができます。

商業上の利得や金銭的報酬を主な目的とした営利目的での利用は、特定非営利活動法人エルピーアイジャパンによる許諾が必要です。ただし、本教科書を利用した教育において、本教科書自体の対価を請求しない場合は、営利目的の教育であっても基本的に使用できます。
その場合も含め、LPI-Japan 事務局までお気軽にお問い合わせください。

＊営利目的の利用とは以下のとおり規定しております。営利企業または非営利団体において、商業上の利得や金銭的報酬を目的に本教科書の印刷実費以上の対価を受講生に請求して本教科書の複製を用いた研修や講義を行うこと。

改変禁止

本教科書は、改変せず使用してください。本教科書に対する改変は、特定非営利活動法人エルピーアイジャパンまたは特定非営利活動法人エルピーアイジャパンが認める団体により行われています。

フィードバック

フィードバックは誰でも参加できる Slack で受け付けていますので、積極的にご参加ください。Slack 参加の詳細は以下の本教科書の Web ページを参照してください。

`https://oss-db.jp/ossdbtext`

図 3: https://oss-db.jp/ossdbtext

本教科書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
お問い合わせ：https://lpij.tayori.com/f/textbookinfo/

図 4: https://lpij.tayori.com/f/textbookinfo/

OSS-DB 技術者認定のご紹介

OSS-DB 技術者認定とは、オープンソースデータベース（OSS-DB）に関する技術力と知識を、公正かつ厳正に中立的な立場で認定する IT 技術者認定です。この認定では OSS-DB の中でも特に商用データベースとの連携に優れ、エンタープライズシステムでも多く活用されている「PostgreSQL」を基準の RDBMS として採用しています。

OSS-DB 技術者認定は、２つのレベルから構成されています。

OSS-DB Silver

データベースシステムの設計・開発・導入・運用ができる技術者であることを証明します。特に、以下の項目についてのスキルを備えていることの証明となります。

- OSS-DB に関する基礎的な知識を有する。
- オープンソースを利用して小規模なデータベースの開発と運用管理ができる
- PostgreSQL などの OSS-DB を使ったデータベースシステムの運用管理ができる
- PostgreSQL などの OSS-DB を利用した開発でデータベース部分を担当することができる。

OSS-DB Gold

大規模データベースシステムの改善・運用管理・コンサルティングができる技術者であることを証明します。特に、以下の項目についてのスキルを備えていることの証明となります。

- OSS-DB に関する深い知識を有する。
- オープンソースを利用して大規模なデータベースの開発と運用管理ができる。
- PostgreSQL などの OSS-DB の内部構造を熟知している。
- PostgreSQL などの OSS-DB の利用方法やデータベースの状態を検証してパフォーマンスチューニングをすることができる。
- PostgreSQL などの OSS-DB の利用方法やデータベースの状態を検証してトラブルシューティングをすることができる。

OSS-DB 技術者認定の詳細については、以下の Web サイトを参照してください。

図 5: https://oss-db.jp/outline

その他の情報源

- 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 https://lets.postgresql.jp/

その他、書籍なども多数出版されているので、それらも参考にしてみてください。

目次

1 実習環境の構築方法

本書では、データベースが動作する実習環境を構築して、実際に SQL 文を実行して実習を進めます。この章では実習環境を構築するために、PostgreSQL のインストールと、実習で使用するデータベースを作成します。

1.1 OS ユーザーの作成

dnf コマンドなどを使って RPM パッケージで PostgreSQL をインストールすると、OS で postgres ユーザーが作成され、関連するディレクトリの所有権やアクセス権が設定されます。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。

以下の例では、admin ユーザーでログイン後、useradd コマンドで postgres ユーザーを作成し、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、su コマンドで postgres ユーザーに切り替え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、admin ユーザーに戻しています。

```
[admin@host1 ~]$ sudo useradd postgres
[sudo] admin のパスワード: ※adminユーザーのパスワードを入力
[admin@host1 ~]$ sudo passwd postgres
ユーザー postgres のパスワードを変更。
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。
[admin@host1 ~]$ su - postgres
パスワード: ※設定したpostgresユーザーのパスワードを入力
最終ログイン: 2024/04/06 (土) 13:49:14 JST 日時 pts/1
[postgres@host1 ~]$ pwd
/home/postgres
[postgres@host1 ~]$ exit
[admin@host1 ~]$
```

1.2 PostgreSQL のインストール

AlmaLinux 9.3 では、ディストリビューションの標準パッケージとして PostgreSQL 13 が提供されています。このパッケージを dnf コマンドを使ってインストールします。

dnf コマンドの引数に必要なパッケージとして postgresql-server を指定してインストールします。依存関係が解消されて、postgresql パッケージと postgresql-private-libs パッケージも一緒にインストールされます。

パッケージ名	説明
postgresql	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
postgresql-private-libs	PostgreSQL を利用する上で必須の共有ライブラリ
postgresql-server	サーバープログラムの本体
postgresql-contrib	拡張機能（インストールは必須ではありません）

```
[admin@host1 ~]$ sudo dnf install postgresql-server
メタデータの期限切れの最終確認: 3:20:56 前の 2024年04月06日 11時06分42秒
    に実施しました。
依存関係が解決しました。
================================================================================
 パッケージ                  Arch        バージョン          リポジトリー   サイズ
================================================================================
インストール:
 postgresql-server           aarch64     13.14-1.el9_3       appstream     5.6 M
依存関係のインストール:
 postgresql                  aarch64     13.14-1.el9_3       appstream     1.5 M
 postgresql-private-libs     aarch64     13.14-1.el9_3       appstream     130 k
```

```
トランザクションの概要
================================================================================
インストール  3 パッケージ

ダウンロードサイズの合計: 7.2 M
インストール後のサイズ: 30 M
これでよろしいですか? [y/N]: y
パッケージのダウンロード:
(1/3): postgresql-private-libs-13.14-1.el9_3.aa 251 kB/s | 130 kB     00:00
(2/3): postgresql-13.14-1.el9_3.aarch64.rpm     664 kB/s | 1.5 MB     00:02
(3/3): postgresql-server-13.14-1.el9_3.aarch64. 2.1 MB/s | 5.6 MB     00:02
--------------------------------------------------------------------------------
合計                                            1.9 MB/s | 7.2 MB     00:03
トランザクションの確認を実行中
トランザクションの確認に成功しました。
トランザクションのテストを実行中
トランザクションのテストに成功しました。
トランザクションを実行中
  準備            :                                                        1/1
  インストール中  : postgresql-private-libs-13.14-1.el9_3.aarch64          1/3
  インストール中  : postgresql-13.14-1.el9_3.aarch64                       2/3
  scriptletの実行中: postgresql-server-13.14-1.el9_3.aarch64               3/3
  インストール中  : postgresql-server-13.14-1.el9_3.aarch64                3/3
  scriptletの実行中: postgresql-server-13.14-1.el9_3.aarch64               3/3
  検証            : postgresql-13.14-1.el9_3.aarch64                       1/3
  検証            : postgresql-private-libs-13.14-1.el9_3.aarch64          2/3
  検証            : postgresql-server-13.14-1.el9_3.aarch64                3/3

インストール済み:
  postgresql-13.14-1.el9_3.aarch64
  postgresql-private-libs-13.14-1.el9_3.aarch64
  postgresql-server-13.14-1.el9_3.aarch64

完了しました!
```

1.3 データベースクラスターの初期化

インストールが終わったら、データベースクラスターの初期化を行います。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。

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ます。このユーザーを PostgreSQL のスーパーユーザーと呼びます。su コマンドでユーザー postgres に変更して操作を行います。

以下の例では、データベースクラスターの初期化は postgresql-setup スクリプトに- initdb オプションを付けて実行します。データベースクラスター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。/var/lib/pgsql/data ディレクトリにデータベースクラスターが作成されているのが分かります。

```
[admin@host1 ~]$ su - postgres
[postgres@host1 ~]$ postgresql-setup --initdb
 * Initializing database in '/var/lib/pgsql/data'
 * Initialized, logs are in /var/lib/pgsql/initdb_postgresql.log
[postgres@host1 ~]$ ls /var/lib/pgsql/data/
PG_VERSION        pg_hba.conf    pg_snapshots  pg_xact
base              pg_ident.conf  pg_stat       postgresql.auto.conf
current_logfiles  pg_logical     pg_stat_tmp   postgresql.conf
global            pg_multixact   pg_subtrans   postmaster.opts
log               pg_notify      pg_tblspc     postmaster.pid
```

```
pg_commit_ts      pg_replslot    pg_twophase
pg_dynshmem       pg_serial      pg_wal
```

1.4 サービスの起動

データベースクラスターを作成したら、PostgreSQL のサービスを起動します。

以下の例では、systemctl コマンドで postgresql サービスを起動しています。

```
[postgres@host1 ~]$ systemctl start postgresql
==== AUTHENTICATING FOR org.freedesktop.systemd1.manage-units ====
'postgresql.service'を開始するには認証が必要です。
Authenticating as: ADMIN (admin)
Password: ※adminユーザーのパスワードを入力
==== AUTHENTICATION COMPLETE ====
[postgres@host1 ~]$ systemctl status postgresql
● postgresql.service - PostgreSQL database server
     Loaded: loaded (/usr/lib/systemd/system/postgresql.service; disabled; pres>
     Active: active (running) since Sat 2024-04-06 14:43:06 JST; 9s ago
    Process: 5145 ExecStartPre=/usr/libexec/postgresql-check-db-dir postgresql >
   Main PID: 5147 (postmaster)
      Tasks: 8 (limit: 10552)
     Memory: 16.7M
        CPU: 41ms
     CGroup: /system.slice/postgresql.service
             ├─5147 /usr/bin/postmaster -D /var/lib/pgsql/data
             ├─5148 "postgres: logger "
             ├─5150 "postgres: checkpointer "
             ├─5151 "postgres: background writer "
             ├─5152 "postgres: walwriter "
             ├─5153 "postgres: autovacuum launcher "
             ├─5154 "postgres: stats collector "
             └─5155 "postgres: logical replication launcher "
lines 1-17/17 (END)
```

1.5 PostgreSQL サービスの自動起動の設定

PostgreSQL サービスは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。

```
[postgres@host1 ~]$ systemctl enable postgresql
==== AUTHENTICATING FOR org.freedesktop.systemd1.manage-unit-files ====
Authentication is required to manage system service or unit files.
Authenticating as: ADMIN (admin)
Password: ※adminユーザーのパスワードを入力
==== AUTHENTICATION COMPLETE ====
==== AUTHENTICATING FOR org.freedesktop.systemd1.reload-daemon ====
Authentication is required to reload the systemd state.
Authenticating as: ADMIN (admin)
Password: ※adminユーザーのパスワードを入力
==== AUTHENTICATION COMPLETE ====
```

1.6 動作の確認

データベースの動作確認を行います。

以下の例では、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@host1 ~]$ psql -l
                                        データベース一覧
   名前     |  所有者  | エンコーディング |  照合順序   | Ctype(変換演算子) |
         アクセス権限
-----------+----------+------------------+-------------+-------------------+-----
-----------------
 postgres  | postgres | UTF8             | ja_JP.UTF-8 | ja_JP.UTF-8       |
 template0 | postgres | UTF8             | ja_JP.UTF-8 | ja_JP.UTF-8       |
     =c/postgres          +
           |          |                  |             |                   |
               postgres=CTc/postgres
 template1 | postgres | UTF8             | ja_JP.UTF-8 | ja_JP.UTF-8       |
     =c/postgres          +
           |          |                  |             |                   |
               postgres=CTc/postgres
(3 行)
```

1.7 実習用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。作成後、psql コマンドで接続できることを確認しておきます。

```
[postgres@host1 ~]$ createdb ossdb
[postgres@host1 ~]$ psql ossdb
psql (13.14)
"help"でヘルプを表示します。

ossdb=#
```

1.8 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)
```

1.9 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。

```
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);

INSERT INTO orders(order_id,order_date,customer_id,prod_id,qty) VALUES
    (1,CURRENT_TIMESTAMP,1,1,10);
INSERT INTO orders(order_id,order_date,customer_id,prod_id,qty) VALUES
    (2,CURRENT_TIMESTAMP,2,2,5);
INSERT INTO orders(order_id,order_date,customer_id,prod_id,qty) VALUES
    (3,CURRENT_TIMESTAMP,3,3,8);
INSERT INTO orders(order_id,order_date,customer_id,prod_id,qty) VALUES
    (4,CURRENT_TIMESTAMP,2,1,3);
INSERT INTO orders(order_id,order_date,customer_id,prod_id,qty) VALUES
    (5,CURRENT_TIMESTAMP,3,2,4);
```

以下は実行例です。

```
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）
```

2　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。第 2 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。

2.1　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。

2.2　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。

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。postgres ユーザーとしてログインするか、admin ユーザーでログインした後、su コマンドで postgres ユーザーに切り替えます。

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）を付けて実行するようにしてください。

```
[admin@host1 ~]$ su - postgres
[postgres@host1 ~]$
```

2.3　データベース一覧を表示する

psql に-l オプションを付けて実行します。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@host1 ~]$ psql -l
                                        データベース一覧
   名前     |  所有者  | エンコーディング |  照合順序   | Ctype(変換演算子) |
         アクセス権限
-----------+----------+------------------+-------------+-------------------+-----
------------------
 ossdb     | postgres | UTF8             | ja_JP.UTF-8 | ja_JP.UTF-8       |
 postgres  | postgres | UTF8             | ja_JP.UTF-8 | ja_JP.UTF-8       |
 template0 | postgres | UTF8             | ja_JP.UTF-8 | ja_JP.UTF-8       |
     =c/postgres          +
           |          |                  |             |                   |
               postgres=CTc/postgres
```

```
 template1 | postgres | UTF8              | ja_JP.UTF-8 | ja_JP.UTF-8       |
     =c/postgres          +
           |          |                   |             |                   |
               postgres=CTc/postgres
(4 行)
```

2.4 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。第 1 章で作成した ossdb データベースに接続します。

```
[postgres@host1 ~]$ psql ossdb
psql (13.14)
"help"でヘルプを表示します。

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、postgres ユーザーで psql を実行しているかを確認してください。

2.5 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
PostgreSQL へのコマンド ライン インターフェイス、psql を使用しています。
ヒント: \copyright とタイプすると、配布条件を表示します。
       \h とタイプすると、SQLコマンドのヘルプを表示します。
       \? とタイプすると、psqlコマンドのヘルプを表示します。
       \g と打つかセミコロンで閉じると、問い合わせを実行します。
       \q で終了します。
```

2.6 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。

2.6.1 ヘルプの確認（\h）

利用できる SQL のヘルプが確認できます。

```
ossdb=# \h
利用可能なヘルプ:
  ABORT                            CREATE FOREIGN DATA WRAPPER      DROP ROUTINE
  ALTER AGGREGATE                  CREATE FOREIGN TABLE             DROP RULE
  ALTER COLLATION                  CREATE FUNCTION                  DROP SCHEMA
（以下略）
```

2.6.2 SQL コマンドのヘルプの確認（\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
コマンド:     DELETE
説明: テーブルの行を削除します。
書式:
[ WITH [ RECURSIVE ] WITH問い合わせ [, ...] ]
DELETE FROM [ ONLY ] テーブル名 [ * ] [ [ AS ] エイリアス ]
    [ USING FROM項目 [, ...] ]
    [ WHERE 条件 | WHERE CURRENT OF カーソル名 ]
    [ RETURNING * | 出力表現 [ [ AS ] 出力名 ] [, ...] ]

URL: https://www.postgresql.org/docs/13/sql-delete.html
```

ページ制御が行われて左下に「-続きます-」と表示されている場合には、カーソルキーの上下やスペースキーで表示を送ることができます。最後まで表示するとページ制御が行われなくなります。途中でページ制御を止めたい場合には q キーを入力します。

2.6.3 psql メタコマンドのヘルプを確認（\?）

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=# \?
一般
  \copyright             PostgreSQL の使い方と配布条件を表示
  \crosstabview [列]     問い合わせを実行し、結果をクロス表形式で出力
  \errverbose            最後のエラーメッセージを最大の冗長性で表示
（以下略）
```

2.6.4 psql の終了（\q）

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@host1 ~]$
```

2.7 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
              リレーション一覧
 スキーマ |   名前   |  タイプ  |  所有者
----------+----------+----------+----------
 public   | customer | テーブル | postgres
 public   | orders   | テーブル | postgres
 public   | prod     | テーブル | postgres
(3 行)
```

2.8 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
                    テーブル"public.customer"
      列       | タイプ  | 照合順序 | Null 値を許容 | デフォルト
---------------+---------+----------+---------------+------------
 customer_id   | integer |          |               |
 customer_name | text    |          |               |
```

```
ossdb=# \d orders
                              テーブル"public.orders"
     列      |           タイプ            | 照合順序 | Null 値を許容 |
         デフォルト
-------------+------------------------------+----------+---------------+----------
 order_id    | integer                      |          |               |
 order_date  | timestamp without time zone |          |               |
 customer_id | integer                      |          |               |
 prod_id     | integer                      |          |               |
 qty         | integer                      |          |               |

ossdb=# \d prod
                    テーブル"public.prod"
    列     | タイプ  | 照合順序 | Null 値を許容 | デフォルト
-----------+---------+----------+---------------+------------
 prod_id   | integer |          |               |
 prod_name | text    |          |               |
 price     | integer |          |               |
```

2.9 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「リレーション一覧」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。

2.10 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。改行しても複数行に渡って入力を受け付けます。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。

2.11 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。

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:  "1" またはその近辺で構文エラー
行 1: 1st line
      ^
ossdb=# \unset PROMPT2
```

```
ossdb=# 1st line
2nd line
3rd line;
ERROR:  "1" または その 近辺 で 構文 エラー
行 1: 1st line
      ^
```

本書では、複数行に渡る SQL を電子版 PDF や EPUB からコピー&ペーストで簡単に実行できるように 2 行目以降のプロンプトが無い実行例を掲載しています。実習時には、デフォルトでは 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。

2.12 テキストファイルからの読み込み実行

psql に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。

ファイルを読み込ませる場合の psql の構文は以下の通りです。

```
$ 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@host1 ~]$ cat test.sql
\d
[postgres@host1 ~]$ 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)
```

2.13 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。

SELECT 文の基本構文は以下の通りです。

```
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]
```

2.13.1 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。

全件全項目検索の SELECT 文は以下の通りです。

```
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 行)

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 行)

ossdb=# SELECT * FROM orders;
 order_id |          order_date          | customer_id | prod_id | qty
----------+------------------------------+-------------+---------+-----
        1 | 2024-04-06 14:55:30.607262 |           1 |       1 |  10
        2 | 2024-04-06 14:55:30.612462 |           2 |       2 |   5
        3 | 2024-04-06 14:55:30.6152   |           3 |       3 |   8
        4 | 2024-04-06 14:55:30.616348 |           2 |       1 |   3
        5 | 2024-04-06 14:55:30.617621 |           3 |       2 |   4
(5 行)
```

2.13.2 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。

SELECT 項目リストを使った SELECT 文の構文は以下の通りです。

```
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 行)
```

2.13.3 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。

WHERE 句の構文は以下の通りです。

```
WHERE 列名 条件式 条件値
```

2.13.4 主な条件式

WHERE 句で指定する条件式には以下のものがあります。

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。

2.13.5 等号、不等号

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 行)
```

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 行)
```

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 行)
```

2.13.6 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 行)
```

2.13.7 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 行)
```

2.13.8 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 行)
```

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。2 つの条件の両方を満たすよう AND 演算子を指定しています。

```
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 行)
```

2.14 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。

条件の指定には、以下の記号が使えます。

記号	マッチング内容
%	0 文字以上の文字列
_	1 文字の文字列

2.14.1 指定した値がデータのどこかに一致（中間一致検索）

文字列中に指定した文字列を含むものを検索します。検索したい文字の前後を%で挟んでいます。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。「みかん」は後ろに指定した%が 0 文字の文字列にマッチしたことになります。「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 行)
```

2.14.2 前方一致検索、後方一致検索

文字列の先頭、あるいは最後に指定した文字を含むものを検索します。中間一致検索同様%と組み合わせますが、文字列の先頭や最後であることを表すために、%は前後 1 つだけ指定します。

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 行)
```

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。たとえば、’り%’と指定すると「りんご」が該当します。

2.14.3 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。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 行)
ossdb=# SELECT * FROM prod WHERE prod_name LIKE '_ん';
 prod_id | prod_name | price
---------+-----------+-------
(0 行)
```

2.14.4 指定した文字列を「含まない」検索

一致 (=) に対して不一致 (<>) があったように、LIKE 演算子に対してその文字列を含まないことを表す NOT LIKE 演算子が用意されています。以下の例では、NOT LIKE 演算子を使って「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 行)
```

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 行)
```

2.15 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。

ORDER BY 句の構文は以下の通りです。

```
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 行)

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 行)
```

2.16 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。

2.16.1 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。

JOIN 句の構文は以下の通りです。

```
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 句のあとに表名を指定します。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。

2.16.2 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 行)
```

この検索結果のうち、customer_id 列と prod_id 列はこのままでは何を表しているのか分かりません。それぞれ customer 表、prod 表から対応する値を持ってきて置き換えることにします。

2.16.3 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。ON 句で結合条件を指定しますが、orders 表の customer_id 列と、customer 表の customer_id 列の値が同じものを結合します。

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
        4 |           2 | 鈴木物産
        3 |           3 | 高橋商店
        5 |           3 | 高橋商店
(5 行)
```

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。

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

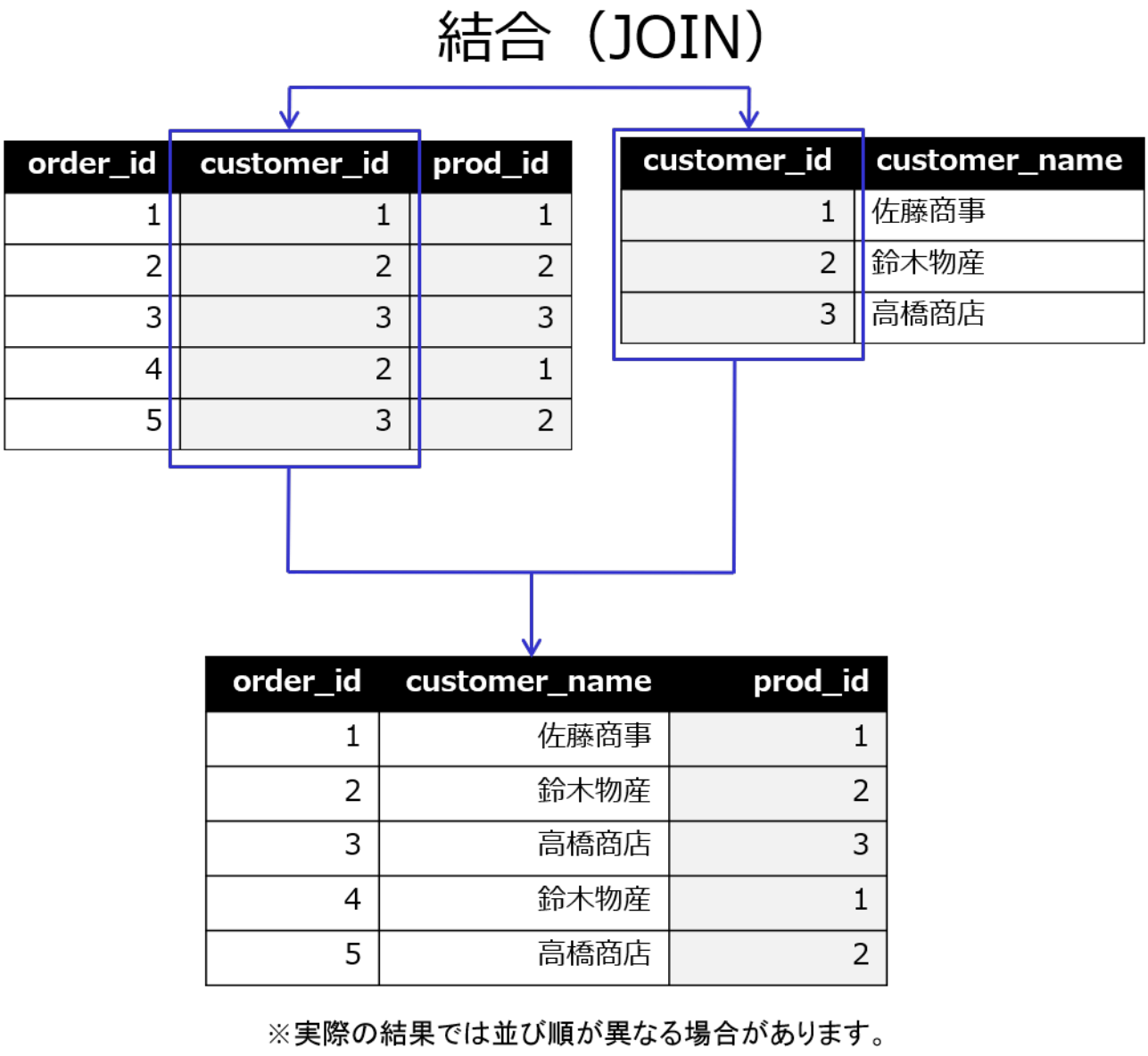

図 6: orders 表と customer 表の JOIN

2.16.4 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。ON 句で結合条件を指定しますが、orders 表の prod_id 列と、prod 表の prod_id 列の値が同じものを結合します。

```
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
         4 | みかん
         2 | りんご
         5 | りんご
         3 | メロン
(5 行)
```

2.16.5 3 つの表を結合

2 つの結合を 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
         4 | 鈴木物産      | みかん    |   3
         2 | 鈴木物産      | りんご    |   5
         5 | 高橋商店      | りんご    |   4
         3 | 高橋商店      | メロン    |   8
(5 行)
```

customer_id を customer 表から取得した顧客名、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。

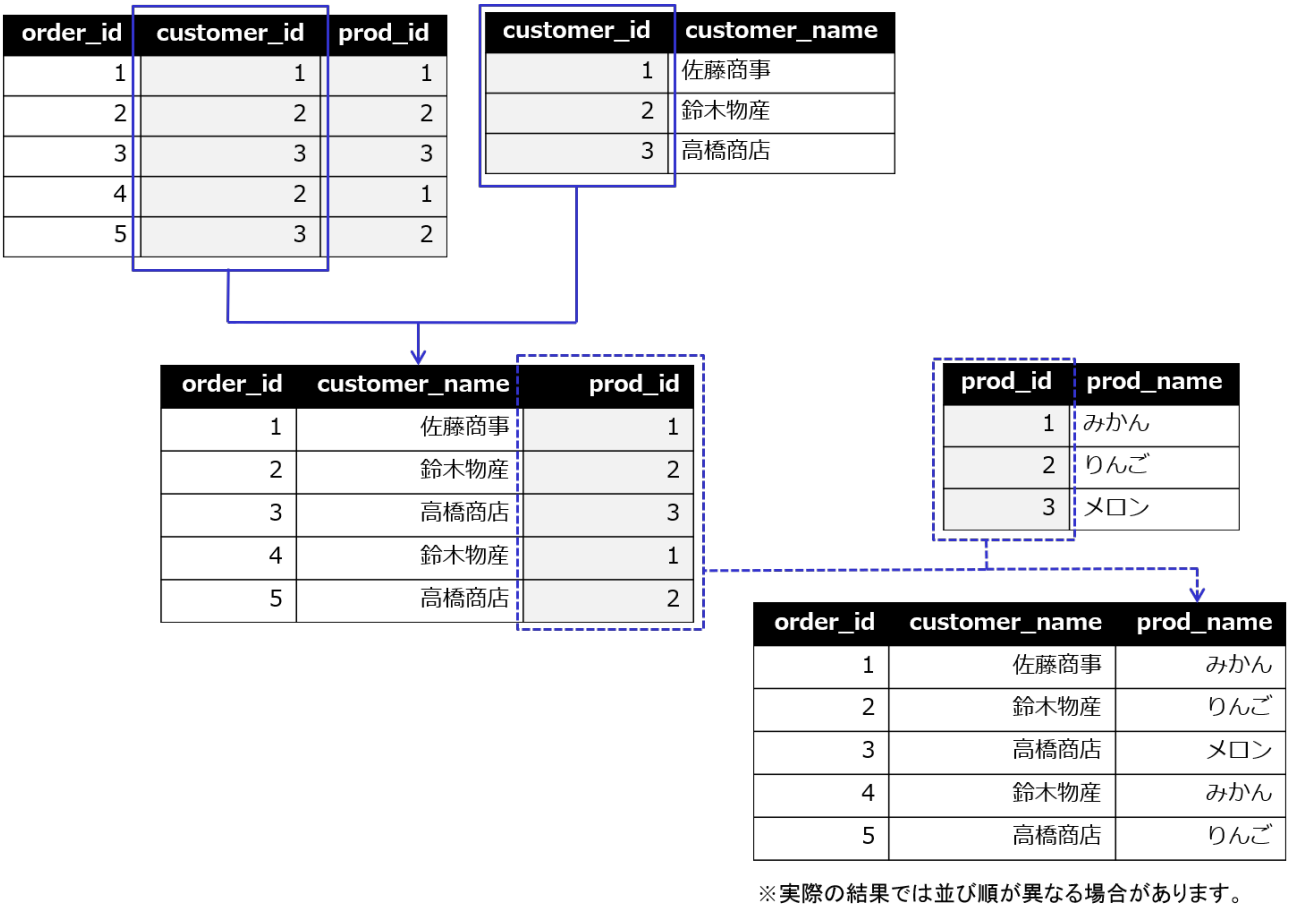

図 7: 3 つの表の JOIN

2.16.6 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。

表別名の指定方法は以下の通りです。

```
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。orders 表は o、cusstomer 表は c、prod 表は p と表別名を付けています。

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

2.17 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。

INSERT 文の構文は以下の通りです。

```
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 行)
```

2.18 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。

UPDATE 文の構文は以下の通りです。

```
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 の行データの 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 行)
```

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、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 行)

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

2.19 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。

DELETE 文の構文は以下の通りです。

```
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 行)
```

3 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。第 3 章では、データ型について解説します。

3.1 サンプルデータベースで使用しているデータ型

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

3.2 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。

3.2.1 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 行)

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 行)
```

3.2.2 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-9999.99 から 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
                テーブル"public.numeric_test"
```

```
 列 |    タイプ     | 照合順序 | Null 値を許容 | デフォルト
----+--------------+----------+---------------+------------
 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フィールドのオーバーフロー
DETAIL:  精度6、位取り2を持つフィールドは、10^4より小さな絶対値に丸められます。
```

3.2.3 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。

以下のような数値型が用意されています。

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

3.3 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。文字数の上限や、可変長か固定長かなどによって異なるデータ型があります。

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

3.3.1 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:  値は型character varying(3)としては長すぎます
ossdb=# INSERT INTO varchar_test VALUES ('あいう');
INSERT 0 1
```

```
ossdb=# INSERT INTO varchar_test VALUES ('AIUEO');
ERROR:  値は型character varying(3)としては長すぎます
ossdb=# SELECT * FROM varchar_test;
 varstring
-----------
 ABC
 あいう
(2 行)
```

3.3.2 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います。以下の例では、LIKE 演算子で文字数を 1 文字から 3 文字まで検索すると、3 文字の場合のみ検索できています。

```
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 '_';
 string
--------
(0 行)

ossdb=# SELECT * FROM char_test WHERE string LIKE '___';
 string
--------
 あ
(1 行)
```

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:  値は型character(3)としては長すぎます
ossdb=# INSERT INTO char_test VALUES ('あいう');
INSERT 0 1
ossdb=# INSERT INTO char_test VALUES ('AIUEO');
ERROR:  値は型character(3)としては長すぎます
ossdb=# SELECT * FROM char_test;
 string
--------
 あ
 ABC
 あいう
(3 行)
```

3.3.3 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。

サンプルデータベースの customer 表や prod 表で使用されています。

```
ossdb=# \d customer
                  テーブル"public.customer"
      列       | タイプ  | 照合順序 | Null 値を許容 | デフォルト
---------------+---------+----------+---------------+------------
 customer_id   | integer |          |               |
 customer_name | text    |          |               |
```

3.4 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。

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。それぞれに現在の日付と時刻を挿入すると、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。CURRENT_TIMESTAMP 関数は timestamp 型の現在の日付と時刻を返します。

```
ossdb=# CREATE TABLE date_test ( d_test  date,
                                 t_test  time,
                                 ts_test timestamp);
CREATE TABLE
ossdb=# \d date_test
                        テーブル"public.date_test"
   列    |            タイプ             | 照合順序 | Null 値を許容 | デフォルト
---------+-------------------------------+----------+---------------+------------
 d_test  | date                          |          |               |
 t_test  | time without time zone        |          |               |
 ts_test | timestamp without time zone |          |               |

ossdb=# INSERT INTO date_test VALUES
    (CURRENT_TIMESTAMP,CURRENT_TIMESTAMP,CURRENT_TIMESTAMP);
INSERT 0 1
ossdb=# SELECT * FROM date_test;
   d_test   |     t_test      |          ts_test
------------+-----------------+---------------------------
 2024-04-07 | 22:58:38.353556 | 2024-04-07 22:58:38.353556
(1 行)
```

4 表

第 4 章では、リレーショナルデータベースの基本的なデータ格納の仕組みである表について解説します。

4.1 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。

表の作成は、CREATE TABLE 文を使用します。

CREATE TABLE 文の構文は以下の通りです。

```
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 staff 表を作成します。

	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
               リレーション一覧
 スキーマ |     名前     |  タイプ  |  所有者
----------+--------------+----------+----------
 public   | char_test    | テーブル | postgres
 public   | customer     | テーブル | postgres
 public   | date_test    | テーブル | postgres
 public   | numeric_test | テーブル | postgres
 public   | orders       | テーブル | postgres
 public   | prod         | テーブル | postgres
 public   | staff        | テーブル | postgres
 public   | varchar_test | テーブル | postgres
(8 行)

ossdb=# \d staff
                  テーブル"public.staff"
    列    | タイプ  | 照合順序 | Null 値を許容 | デフォルト
----------+---------+----------+---------------+------------
 id       | integer |          |               |
 name     | text    |          |               |
 birthday | date    |          |               |
```

4.2 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 行)
```

4.3 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。

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
コマンド:     ALTER TABLE
説明: テーブルの定義を変更します。
書式:
ALTER TABLE [ IF EXISTS ] [ ONLY ] 名前 [ * ]
    アクション [, ... ]

（略）

アクションは以下のいずれかです:

    ADD [ COLUMN ] [ IF NOT EXISTS ] 列名 データ型 [ COLLATE 照合順序 ] [
        カラム制約 [ ... ] ]
    DROP [ COLUMN ] [ IF EXISTS ] 列名 [ RESTRICT | CASCADE ]
    ALTER [ COLUMN ] 列名 [ SET DATA ] TYPE データ型 [ COLLATE 照合順序 ] [
        USING 評価式
（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 表に所属部署を表す dept_id 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_id integer;
ALTER TABLE
ossdb=# \d staff
                  テーブル"public.staff"
    列    | タイプ  | 照合順序 | Null 値を許容 | デフォルト
----------+---------+----------+---------------+------------
 id       | integer |          |               |
 name     | text    |          |               |
 birthday | date    |          |               |
 dept_id  | integer |          |               |

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

```
（1 行）
```

4.4 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。

4.4.1 正しい設計になっているか

例えば、上記では staff 表に dept_id 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。

所属部門表の例は以下の通りです。意味の列は実際のデータベースの表には含めません。

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）

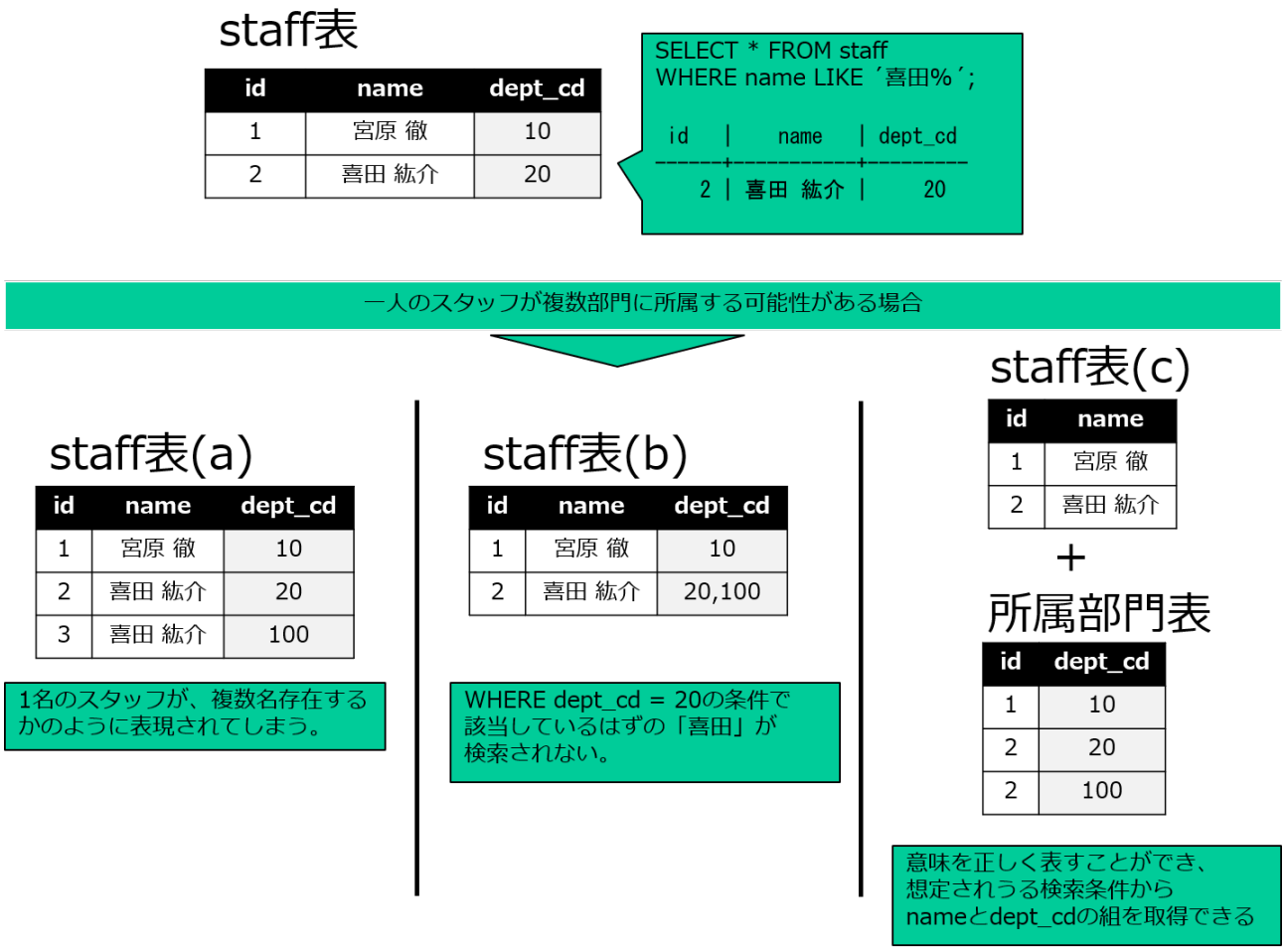

図 8: ALTER TABLE は行うべきか

4.4.2 表定義の変更作業そのものが及ぼす影響

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。

小規模な表であれば短時間で終わるため問題は少ないですが、たとえば数百万人のデータを収めたウェブサービスの会員表ならばどうでしょうか。利用者がログインするたびにこの表が参照されているような場合、表の変更作業中は参照できなくなるため、誰もログインできなくなります。実際のデータベース製品の実装では、表定義の変更中の参照が許されているものもあります。しかし、データの更新などは行えなくなるので、なんらかの影響は出てしまいます。

影響する範囲や時間を鑑みて、表の定義を変更するのではなく上記のような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を保持することができます。

4.4.3 短期の対策と中長期の対策

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による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中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。

4.5 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
                リレーション一覧
 スキーマ |     名前     |  タイプ  |  所有者
----------+--------------+----------+----------
 public   | char_test    | テーブル | postgres
 public   | customer     | テーブル | postgres
 public   | date_test    | テーブル | postgres
 public   | numeric_test | テーブル | postgres
 public   | orders       | テーブル | postgres
 public   | prod         | テーブル | postgres
 public   | staff        | テーブル | postgres
 public   | varchar_test | テーブル | postgres
(8 行)

ossdb=# DROP TABLE staff;
DROP TABLE
ossdb=# \d
                リレーション一覧
 スキーマ |     名前     |  タイプ  |  所有者
----------+--------------+----------+----------
 public   | char_test    | テーブル | postgres
 public   | customer     | テーブル | postgres
 public   | date_test    | テーブル | postgres
 public   | numeric_test | テーブル | postgres
 public   | orders       | テーブル | postgres
 public   | prod         | テーブル | postgres
 public   | varchar_test | テーブル | postgres
(7 行)
```

4.6 TRUNCATE によるデータ削除

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、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 TRUNCATE 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います。表を再作成する場合、これらも再定義する必要があるため時間がかかるという問題が発生します。このような問題が起きないよう、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。

TRUNCATE 文の構文は以下の通りです。

```
TRUNCATE 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 行)

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

4.7 データのセーブ・ロード

psql を使って、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。

方法には、SQL である COPY 文を使う方法と、psql メタコマンドである\copy を使う方法があります。両者は似ていますが、仕組みが若干違います。それぞれについて解説します。

4.7.1 COPY 文は SQL として実行される

COPY 文は SQL として実行され、PostgreSQL が動作している OS のファイルに直接データを書き出します。そのため、以下の制限があります。

- PostgreSQL のスーパーユーザーでのみ実行することができます。この実習では postgres ユーザーがスーパーユーザーにあたります。
- リモートに書き出すことはできません。データは PostgreSQL が動作しているサーバーのローカルファイルとして書き出されます。リモートにデータが欲しい場合には、書き出されたファイルをリモートにネットワーク経由で転送する必要があります。

4.7.2 COPY TO 文による行データのセーブ

COPY TO 文でデータをファイルにセーブできます。

COPY TO 文によるデータのセーブの構文は以下の通りです。FORMAT 句で csv を指定することで、CSV 形式のファイルとしてセーブができます。

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

COPY TO 文は、ファイルの保存先によっては OS のセキュリティを制御している SELinux によって拒否される場合があります。確認方法については後述します。

4.7.3 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。

COPY 文によるデータのロードの構文は以下の通りです。FORMAT 句で csv を指定することで、CSV 形式のファイルからロードできます。

```
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 行)

ossdb=# COPY customer FROM '/tmp/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 行)
```

4.7.4 \copy メタコマンドは psql で処理される

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。それに対して、\copy メタコマンド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

具体的な使用例はこの後の演習で解説します。

4.7.5 参考：PostgreSQL と SELinux の関係

COPY 文は PostgreSQL が OS 上にファイルを作成してデータをセーブします。このファイル作成が、OS 側のセキュリティを制御している SELinux によって拒否される場合があります。

以下は、postgres ユーザーのホームディレクトリに CSV ファイルを作成しようとする例です。SELinux によりホームディレクトリへのファイル作成が拒否されて、エラーが発生しています。

```
ossdb=# COPY customer TO '/home/postgres/customer.csv' (FORMAT csv);
ERROR:
    ファイル"/home/postgres/customer.csv"を書き込み用にオープンできませんでした:
    許可がありません
HINT:  COPY
    TOによってPostgreSQLサーバプロセスはファイルの書き込みを行います。psqlの
    \copy のようなクライアント側の仕組みが必要かもしれません
```

SELinux のログは/var/log/audit/audit.log に記録されています。このログファイルから、customer.csv の文字列を探してみると、どのような処理が拒否されたのかが分かります。

postgresql_t コンテキストによる、user_home_dir_t コンテキストに対しての、add_name などの OS のファイル操作に関わる処理が拒否されています。

```
[admin@host1 ~]$ sudo grep customer.csv /var/log/audit/audit.log
type=AVC msg=audit(1712711439.220:626): avc:  denied  { add_name } for
    pid=24821 comm="postmaster" name="customer.csv"
    scontext=system_u:system_r:postgresql_t:s0
    tcontext=unconfined_u:object_r:user_home_dir_t:s0 tclass=dir permissive=1
type=AVC msg=audit(1712711439.220:626): avc:  denied  { create } for  pid=24821
    comm="postmaster" name="customer.csv"
    scontext=system_u:system_r:postgresql_t:s0
    tcontext=system_u:object_r:user_home_dir_t:s0 tclass=file permissive=1
type=AVC msg=audit(1712711439.220:626): avc:  denied  { write open } for
    pid=24821 comm="postmaster" path="/home/postgres/customer.csv" dev="dm-0"
    ino=18700809 scontext=system_u:system_r:postgresql_t:s0
    tcontext=system_u:object_r:user_home_dir_t:s0 tclass=file permissive=1
type=AVC msg=audit(1712711439.220:627): avc:  denied  { getattr } for
    pid=24821 comm="postmaster" path="/home/postgres/customer.csv" dev="dm-0"
    ino=18700809 scontext=system_u:system_r:postgresql_t:s0
    tcontext=system_u:object_r:user_home_dir_t:s0 tclass=file permissive=1
```

行いたい処理が SELinux によって拒否される場合には、/tmp ディレクトリのように拒否されない場所にファイルを作成する処理に変更するか、SELinux の設定を適切に変更してください。

SELinux が原因かどうかを確認するために、一時的に SELinux を Permissive で動作させることもできます。ただし、Permissive への変更は確認作業を行う目的のみとし、問題に対する対処が終わったら Enforcing に戻す必要があります。

以下の例では、SELinux が Enforcing の状態から Permissive に変更しています。

```
[root@host1 ~]# getenforce
Enforcing
[root@host1 ~]# setenforce 0
[root@host1 ~]# getenforce
Permissive
```

再度、COPY 文を実行すると、今度は成功します。

```
ossdb=# COPY customer TO '/home/postgres/customer.csv' (FORMAT csv);
COPY 3
```

原因が SELinux と分かったら、Enforicing に戻し、適切な対処を行ってください。

5　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。

演習 1：データ操作

以下の操作を SQL で行ってみましょう。

1. prod 表のすべての商品の価格を 10%アップします
2. prod 表の価格が 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セーブします
4. prod 表を削除します
5. prod 表を再度作成します
6. prod 表にデータをファイルからロードします

表の定義はあらかじめ確認しておきましょう。また、第 1 章の表を作成するための CREATE TABLE 文の例も参考にしてください。

演習 1-1：prod 表のすべての商品の価格を 10%アップします

prod 表の price 列を指定して UPDATE を実行する。

```
ossdb=# \d prod
                    テーブル"public.prod"
    列     | タイプ  | 照合順序 | Null 値を許容 | デフォルト
-----------+---------+----------+---------------+------------
 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 行)

ossdb=# UPDATE prod SET price = price * 1.1;
UPDATE 4
ossdb=# SELECT * FROM prod;
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 行)
```

演習 1-2：prod 表の価格が 100 以上の商品の価格を元に戻します

price 表の price の値が 100 以上の商品を指定して UPDATE を実行する。

```
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 行)
```

演習 **1-3**：**prod** 表のデータをファイルにセーブします

COPY TO 文でデータをファイルにセーブします。

```
ossdb=# COPY prod TO '/tmp/prod.csv' (FORMAT csv);
COPY 4
ossdb=# \\! cat /tmp/prod.csv
1,みかん,55
2,りんご,77
4,バナナ,34
3,メロン,100
```

演習 **1-4**：**prod** 表を削除します

DROP TABLE 文で prod 表を削除します。

```
ossdb=# DROP TABLE prod;
DROP TABLE
ossdb=# SELECT * FROM prod;
ERROR:  リレーション"prod"は存在しません
行 1: SELECT * FROM prod;
                    ^
```

演習 **1-5**：**prod** 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します。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 型を指定します。

```
ossdb=# CREATE TABLE prod ( prod_id     integer,
                            prod_name   text,
                            price       numeric );
CREATE TABLE
ossdb=# SELECT * FROM prod;
 prod_id | prod_name | price
---------+-----------+-------
(0 行)
```

演習 **1-6**：**prod** 表にデータをファイルからロードします

COPY FROM 文を使用してデータをファイルからロードします。

```
ossdb=# COPY prod FROM '/tmp/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 行)
```

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。

演習 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。

```
https://www.post.japanpost.jp/zipcode/download.html
```

データは様々な形式のものが配布されています。「住所の郵便番号（1 レコード 1 行、UTF-8 形式）（CSV 形式）」にある「住所の郵便番号（1 レコード 1 行、UTF-8 形式）」のリンクをクリックします。

「データのダウンロード」にある「最新データのダウンロード」のリンクから、ZIP 形式でアーカイブされた CSV ファイルがダウンロードできます。ダウンロードはサーバーで行うか、ダウンロードしたファイルをサーバーにコピーする必要があります。

```
https://www.post.japanpost.jp/zipcode/dl/utf/zip/utf_ken_all.zip
```

以下の例は、サーバーで 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@host1 ~]$ wget
    https://www.post.japanpost.jp/zipcode/dl/utf/zip/utf_ken_all.zip
--2024-04-10 15:23:02--
    https://www.post.japanpost.jp/zipcode/dl/utf/zip/utf_ken_all.zip
www.post.japanpost.jp (www.post.japanpost.jp) をDNSに問いあわせています...
    43.253.212.144
www.post.japanpost.jp (www.post.japanpost.jp)|43.253.212.144|:443
    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: 2183402 (2.1M) [application/zip]
`utf_ken_all.zip' に保存中

utf_ken_all.zip     100%[===================>]   2.08M  9.69MB/s 時間 0.2s

2024-04-10 15:23:02 (9.69 MB/s) - `utf_ken_all.zip' へ保存完了 [2183402/2183402]

[postgres@host1 ~]$ ls
utf_ken_all.zip
[postgres@host1 ~]$ unzip utf_ken_all.zip
Archive:  utf_ken_all.zip
  inflating: utf_ken_all.csv
[postgres@host1 ~]$ ls -l utf_ken_all.csv
-rw-r--r--. 1 postgres postgres 18335216  3月 22 10:40 utf_ken_all.csv
```

演習 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。

郵便番号データのデータ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)

内容	備考
市区町村名	半角カタカナ (コード順に掲載)
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	

このデータ項目に基づいて、表を作成します。

以下の例は、固定長の文字列データを char 型、固定長ではない文字列データを 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
```

演習 2-3：郵便番号データの内容確認

データは以下のような内容です。

```
[postgres@host1 ~]$ head -3 utf_ken_all.csv
01101,"060  ","0600000","ホッカイドウ","サッポロシチュウオウク",
"イカニケイサイガナイバアイ","北海道","札幌市中央区","以下に掲載がない場合",
0,0,0,0,0,0
01101,"064  ","0640941","ホッカイドウ","サッポロシチュウオウク",
"アサヒガオカ","北海道","札幌市中央区","旭ケ丘",0,0,1,0,0,0
01101,"060  ","0600041","ホッカイドウ","サッポロシチュウオウク",
"オオドオリヒガシ","北海道","札幌市中央区","大通東",0,0,1,0,0,0
```

演習 2-4：データのロード

psql を使って、CSV ファイルをロードします。

以下の例では、\copy メタコマンドを使って CSV ファイルからデータをロードしています。

```
ossdb=# \copy zip from utf_ken_all.csv (format csv)
COPY 124370
```

演習 2-5：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 |
     choume | smallarea | change | reason
--------+--------+---------+--------------+----------+--------------------------
---+--------+--------+-----------------------+----------+-------+--------+-----
------+--------+--------
 13113  | 150    | 1500002 | トウキョウト | シブヤク |
     シブヤ（ツギノビルヲノゾク） | 東京都 | 渋谷区 | 渋谷（次のビルを除く） |
            0 |     0 |      1 |         0 |      0 |      0
(1 行)
```

6 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。

6.1 prod 表を再作成

以降の実習は、prod 表を再作成した状態で進めます。

prod 表を削除し、再作成と新しい行データの insert 文を実行してください。

```
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 行)
```

6.2 演算子

SELECT 文の中で条件を指定する際に、複数の条件を指定する AND/OR 演算子や、部分一致条件を指定する LIKE 演算子、値の範囲を指定する BETWEEN 演算子があります。既に SQL の基本の解説でも出てきていますが、あらためて詳細を解説します。

6.2.1 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 行)
```

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 行)
```

6.2.2 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。

条件の指定には、ワイルドカードが使用できます。

ワイルドカード	内容
_	1 文字
%	0 文字以上の文字列

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 行)
```

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 行)
```

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。

6.2.3 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 行)
```

6.3 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。

主な集約関数は以下の通りです。

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

6.3.1 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 行)
```

6.3.2 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 行)
```

6.3.3 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 行)
```

6.3.4 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 行)
```

6.3.5 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 行)
```

6.3.6 参考：文字列データの最大/最小

文字列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、convert_to 関数で「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。ORDER BY DESC 句は指定された列のデータを逆順にソートします。

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 行)
```

6.4 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 行)
```

prod 表では、prod_id が 1 のデータは「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。

6.4.1 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 行)
```

```
ossdb=# SELECT prod_id,sum(qty) FROM orders
GROUP BY prod_id
HAVING sum(qty) < 10;
 prod_id | sum
---------+-----
       3 |   8
       2 |   9
(2 行)
```

6.4.2 集約関数における 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。このグループに対する集約関数の演算結果に対して HAVING 句が絞り込みを行います。

6.5 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。

6.5.1 EXISTS 演算子

EXISTS 演算子は、副問い合わせが結果を 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、まず主問い合わせを実行して返された行データの値を、1 行ずつ副問い合わせに渡して実行します。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 行)

ossdb=# SELECT * FROM orders;
  order_id |          order_date          | customer_id | prod_id | qty
----------+------------------------------+-------------+---------+-----
         1 | 2024-04-06 14:55:30.607262 |           1 |       1 |  10
         2 | 2024-04-06 14:55:30.612462 |           2 |       2 |   5
         3 | 2024-04-06 14:55:30.6152   |           3 |       3 |   8
         4 | 2024-04-06 14:55:30.616348 |           2 |       1 |   3
         5 | 2024-04-06 14:55:30.617621 |           3 |       2 |   4
(5 行)

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 行)
```

6.5.2 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 行)

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 行)
```

6.6 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。

6.6.1 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 YMD、つまり年月日のスタイルであることが分かります。この場合、西暦を 2 桁で表すという日付は「24-04-14」は年-月-日と解釈するので「2024 年 4 月 14 日」と扱われています。

```
ossdb=# SHOW DATESTYLE;
 DateStyle
-----------
 ISO, YMD
(1 行)

ossdb=# SELECT '24-04-14'::date;
    date
------------
 2024-04-14
(1 行)
```

DATESTYLE を変更することもできます。たとえばアメリカ式の月-日-年に変更します。同じ「24-04-14」という文字列は、今度は「2014 年 24 月 4 日」と解釈されてしまい、エラーになります。

```
ossdb=# set DATESTYLE to 'ISO, MDY';
SET
ossdb=# SHOW DATESTYLE;
 DateStyle
```

```
-----------
 ISO, MDY
(1 行)

ossdb=# SELECT '24-04-14'::date;
ERROR:  日付時刻のフィールドが範囲外です: "24-04-14"
行 1: SELECT '24-04-14'::date;
             ^
HINT:  他の"datestyle"設定が必要かもしれません。

ossdb=# SELECT '04-14-24'::date;
    date
------------
 2024-04-14
(1 行)
```

このように日付書式は国毎の習慣や、使用している環境の設定で異なるので、必ず確認して必要に応じて設定を変更してください。また、年を 2 桁で指定するのは解釈の違いが起きるため望ましくないので、できるだけ 4 桁で指定するようにしましょう。「2024-04-14」という形式は必ず「2024 年 4 月 14 日」として扱われる、推奨される書式です。

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

ossdb=# SELECT '2024-04-14'::date;
    date
------------
 2024-04-14
(1 行)
```

6.7 現在の日付や時刻を取得する関数

行データの挿入時などに、現在の日付や時刻を取得してデータにしたい場合に使用できる関数があります。

6.7.1 CURRENT_DATE/CURRENT_TIME/CURRENT_TIMESTAMP 関数

CURRENT_DATE/CURRENT_TIME/CURRENT_TIMESTAMP 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。

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 CURRENT_DATE;
 current_date
--------------
 2024-04-14
(1 行)

ossdb=# SELECT CURRENT_TIME;
    current_time
--------------------
 16:31:41.243898+09
(1 行)

ossdb=# SELECT CURRENT_TIMESTAMP;
       current_timestamp
-------------------------------
 2024-04-14 16:31:47.848579+09
```

```
(1 行)
```

6.7.2 now() 関数

now() 関数は、現在の日付と時刻を取得する関数です。結果は CURRENT_TIMESTAMP 関数と同じ TIMESTAMP 型で返ってきます。

```
ossdb=# SELECT now();
              now
-------------------------------
 2024-04-14 16:52:24.355623+09
(1 行)
```

6.8 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。

6.8.1 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 行)

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
 鈴木物産      |       1 |   3
 高橋商店      |       3 |   8
 高橋商店      |       2 |   4
(5 行)
```

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。

下の例では、customer 表を JOIN 句の左側にした 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ります。

```
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
 鈴木物産      |       1 |   3
 高橋商店      |       3 |   8
 高橋商店      |       2 |   4
 藤原流通      |         |
(6 行)
```

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
---------------+-----------+-----
 佐藤商事      | みかん    |  10
 鈴木物産      | みかん    |   3
 鈴木物産      | りんご    |   5
 高橋商店      | りんご    |   4
 高橋商店      | メロン    |   8
 藤原流通      |           |
(6 行)
```

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

6.8.2 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ます。

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

```
(16 行)
```

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
 佐藤商事      | りんご
 佐藤商事      | メロン
 佐藤商事      | バナナ
 鈴木物産      | みかん
 鈴木物産      | りんご
 鈴木物産      | メロン
 鈴木物産      | バナナ
 高橋商店      | みかん
 高橋商店      | りんご
 高橋商店      | メロン
 高橋商店      | バナナ
 藤原流通      | みかん
 藤原流通      | りんご
 藤原流通      | メロン
 藤原流通      | バナナ
(16 行)
```

6.8.3 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 行)
```

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 行)
```

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
 りんご    |    70 | りんご    |    70
 りんご    |    70 | メロン    |   100
 りんご    |    70 | バナナ    |    30
 メロン    |   100 | みかん    |    50
 メロン    |   100 | りんご    |    70
 メロン    |   100 | メロン    |   100
 メロン    |   100 | バナナ    |    30
 バナナ    |    30 | みかん    |    50
 バナナ    |    30 | りんご    |    70
 バナナ    |    30 | メロン    |   100
 バナナ    |    30 | バナナ    |    30
(16 行)
```

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
 りんご    |    70 | りんご    |    70 |  140
 りんご    |    70 | メロン    |   100 |  170
 りんご    |    70 | バナナ    |    30 |  100
 メロン    |   100 | みかん    |    50 |  150
 メロン    |   100 | りんご    |    70 |  170
 メロン    |   100 | メロン    |   100 |  200
 メロン    |   100 | バナナ    |    30 |  130
 バナナ    |    30 | みかん    |    50 |   80
 バナナ    |    30 | りんご    |    70 |  100
 バナナ    |    30 | メロン    |   100 |  130
 バナナ    |    30 | バナナ    |    30 |   60
(16 行)
```

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 行)
```

あとは SELECT リストに指定するものを必要なものに絞れば最初に実行した SQL 文の結果が得られます。

6.9 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。

6.9.1 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 | 2024-04-06 14:55:30.607262 |           1 |       1 |  10
        2 | 2024-04-06 14:55:30.612462 |           2 |       2 |   5
        3 | 2024-04-06 14:55:30.6152   |           3 |       3 |   8
        4 | 2024-04-06 14:55:30.616348 |           2 |       1 |   3
        5 | 2024-04-06 14:55:30.617621 |           3 |       2 |   4
(5 行)

ossdb=# SELECT * FROM orders ORDER BY order_id LIMIT 3;
 order_id |         order_date         | customer_id | prod_id | qty
----------+----------------------------+-------------+---------+-----
        1 | 2024-04-06 14:55:30.607262 |           1 |       1 |  10
        2 | 2024-04-06 14:55:30.612462 |           2 |       2 |   5
        3 | 2024-04-06 14:55:30.6152   |           3 |       3 |   8
(3 行)
```

6.9.2 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 | 2024-04-06 14:55:30.612462 |           2 |       2 |   5
        3 | 2024-04-06 14:55:30.6152   |           3 |       3 |   8
        4 | 2024-04-06 14:55:30.616348 |           2 |       1 |   3
(3 行)
```

7 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。

7.1 主キー

主キーは、表のデータを「一意」(UNIQUE) に識別できる 1 つ以上の列のことです。一意とは、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 | 2024-04-06 14:55:30.607262 |           1 |       1 |  10
(1 行)

ossdb=# SELECT * FROM orders WHERE customer_id = 2;

 order_id |         order_date         | customer_id | prod_id | qty
----------+----------------------------+-------------+---------+-----
        2 | 2024-04-06 14:55:30.612462 |           2 |       2 |   5
        4 | 2024-04-06 14:55:30.616348 |           2 |       1 |   3
(2 行)
```

7.1.1 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。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。また、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。

主キーを指定する ALTER TABLE 文の構文は以下の通りです。

```
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。

```
ossdb=# ALTER TABLE prod ADD PRIMARY KEY(prod_id);
ALTER TABLE
ossdb=# \d prod
                  テーブル"public.prod"
    列     | タイプ  | 照合順序 | Null 値を許容 | デフォルト
-----------+---------+----------+---------------+------------
 prod_id   | integer |          | not null      |
 prod_name | text    |          |               |
 price     | numeric |          |               |
インデックス:
    "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
```

7.1.2　主キーの動作を確認する

主キーを指定すると、「一意」(UNIQUE) であり、かつ「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:  リレーション"prod"の列"prod_id"のNULL値が非NULL制約に違反しています
DETAIL:  失敗した行は(null, すいか, 60)を含みます

ossdb=# INSERT INTO prod (prod_id,prod_name,price) VALUES (4,'すいか',60);
ERROR:  重複したキー値は一意性制約"prod_pkey"違反となります
DETAIL:  キー (prod_id)=(4) はすでに存在します。

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 行)
```

7.1.3　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
                テーブル"public.student"
  列   | タイプ  | 照合順序 | Null 値を許容 | デフォルト
-------+---------+----------+---------------+------------
 class | text    |          | not null      |
 no    | integer |          | not null      |
 name  | text    |          |               |
インデックス:
    "student_pkey" PRIMARY KEY, btree (class, no)
```

7.2　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。

7.2.1 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。

7.2.2 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。

外部キーを指定する ALTER TABLE 文の構文は以下の通りです。

```
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
                          テーブル"public.orders"
     列      |           タイプ            | 照合順序 | Null 値を許容 |
         デフォルト
-------------+-----------------------------+----------+---------------+----------
 order_id    | integer                     |          | not null      |
 order_date  | timestamp without time zone |          |               |
 customer_id | integer                     |          |               |
 prod_id     | integer                     |          |               |
 qty         | integer                     |          |               |
インデックス:
    "orders_pkey" PRIMARY KEY, btree (order_id)
外部キー制約:
    "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
                  テーブル"public.customer"
      列       | タイプ  | 照合順序 | Null 値を許容 | デフォルト
---------------+---------+----------+---------------+------------
 customer_id   | integer |          | not null      |
 customer_name | text    |          |               |
インデックス:
    "customer_pkey" PRIMARY KEY, btree (customer_id)
参照元:
    TABLE "orders" CONSTRAINT "orders_customer_id_fkey" FOREIGN KEY
        (customer_id) REFERENCES customer(customer_id)
```

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,CURRENT_TIMESTAMP,4,6,6);
```

```
ERROR:
    テーブル"orders"への挿入、更新は外部キー制約"orders_prod_id_fkey"に違反して
います
DETAIL:  テーブル"prod"にキー(prod_id)=(6)がありません

ossdb=# INSERT INTO orders(order_id,order_date,customer_id,prod_id,qty)
VALUES (6,CURRENT_TIMESTAMP,3,6,6);
ERROR:
    テーブル"orders"への挿入、更新は外部キー制約"orders_prod_id_fkey"に違反して
います
DETAIL:  テーブル"prod"にキー(prod_id)=(6)がありません
```

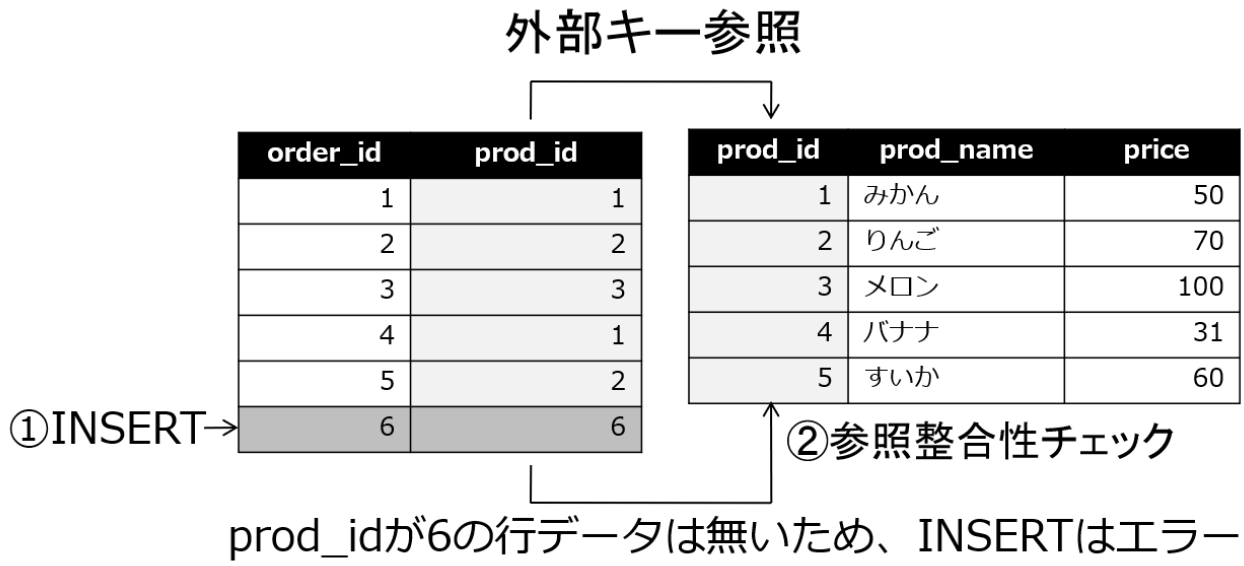

図 9: 外部キー制約違反

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,CURRENT_TIMESTAMP,3,5,6);
INSERT 0 1
ossdb=# SELECT * FROM orders WHERE order_id = 6;
 order_id |         order_date         | customer_id | prod_id | qty
----------+----------------------------+-------------+---------+-----
        6 | 2024-04-17 11:11:11.840835 |           3 |       5 |   6
(1 行)

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 行)
```

7.2.3　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
```

作成した表に行データを再度作成しておきます。

```
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

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);
```

7.2.4 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。

7.3 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。

7.4　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。

7.4.1　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
                    テーブル"public.customer"
      列       | タイプ  | 照合順序 | Null 値を許容 | デフォルト
---------------+---------+----------+---------------+------------
 customer_id   | integer |          | not null      |
 customer_name | text    |          |               |
インデックス:
    "customer_pkey" PRIMARY KEY, btree (customer_id)
参照元:
    TABLE "orders" CONSTRAINT "orders_customer_id_fkey" FOREIGN KEY
        (customer_id) REFERENCES customer(customer_id)
```

7.4.2　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 (6,'ぶどう',NULL);
ossdb=# SELECT * FROM prod;
 prod_id | prod_name | price
---------+-----------+-------
       1 | みかん    |    50
       2 | りんご    |    70
       3 | メロン    |   100
       6 | ぶどう    |
(4 行)

ossdb=# SELECT * FROM prod WHERE price IS NULL;
 prod_id | prod_name | price
---------+-----------+-------
       6 | ぶどう    |
(1 行)

ossdb=# SELECT * FROM prod WHERE price IS NOT NULL;
 prod_id | prod_name | price
---------+-----------+-------
       1 | みかん    |    50
       2 | りんご    |    70
       3 | メロン    |   100
(3 行)
```

7.4.3　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
-------
     4
(1 行)

ossdb=# SELECT count(price) FROM prod;
 count
-------
     3
(1 行)
```

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。

```
ossdb=# SELECT sum(price),count(price),avg(price) FROM prod;
 sum | count |         avg
-----+-------+---------------------
 220 |     3 | 73.3333333333333333
(1 行)
```

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、「220 円 ÷ 3 行 = 73.33…円」という結果になりました。

7.4.4　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 VALUES (7,'',NULL);
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
       6 | ぶどう    |
       7 |           |
(5 行)

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

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

```
(2 行)
```

7.5　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。

7.5.1　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。

シーケンスの作成の構文は以下の通りです。

```
CREATE SEQUENCE シーケンス名;
```

作成時に値を何も指定しなかったシーケンスのデフォルト値は以下の通りです。

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)

7.5.2　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。

currval() 関数は、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。

nextval() 関数は、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。

以下の例では、order_id_seq シーケンスを作成し、値を取り出しています。シーケンスを使用していないと currval() 関数はエラーになりますが、nextval() 関数を使用するとエラーが出なくなります。

```
ossdb=# CREATE SEQUENCE order_id_seq;
CREATE SEQUENCE

ossdb=# SELECT currval('order_id_seq');
ERROR:  本セッションでシーケンス"order_id_seq"のcurrvalはまだ定義されていません

ossdb=# SELECT nextval('order_id_seq');
 nextval
---------
       1
(1 行)

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

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

7.5.3　シーケンスの値を設定する

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

以下の例では、シーケンスの値を設定しています。

```
ossdb=# SELECT setval('order_id_seq',0);
ERROR:  setval:
    値0はシーケンス"order_id_seq"の範囲(1..9223372036854775807)外です"

ossdb=# SELECT setval('order_id_seq',100);
 setval
--------
    100
(1 行)

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

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

7.5.4　シーケンスを SQL 文で使用する

シーケンスは、INSERT 文に組み込んで使用できます。

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 行)

ossdb=# INSERT INTO orders(order_id,order_date,customer_id,prod_id,qty)
VALUES (nextval('order_id_seq'),CURRENT_TIMESTAMP,2,1,7);
INSERT 0 1
ossdb=# SELECT * FROM orders;
 order_id |         order_date          | customer_id | prod_id | qty
----------+-----------------------------+-------------+---------+-----
      101 | 2024-04-17 14:38:11.97433 |           2 |       1 |   7
(1 行)
```

7.5.5　シーケンスをテーブル定義で使用する

シーケンスをテーブル定義で指定することで、INSERT 時に自動的にシーケンスから取得した値を挿入することができます。

方法としては、列のデータ型として SERIAL を指定する方法と、作成済みのシーケンスをデフォルトとして指定する方法があります。

以下の例は、テーブル作成時に id 列に SERIAL 型を定義しています。INSERT 時に memo 列の値だけ指定することで、自動的に id 列にシーケンス値が挿入されます。

```
ossdb=# CREATE TABLE serial_test(
    id  SERIAL,
    memo    TEXT);
CREATE TABLE
ossdb=# insert into serial_test(memo) values('SERIALのテスト');
INSERT 0 1
ossdb=# SELECT * FROM serial_test;
 id |      memo
----+----------------
  1 | SERIALのテスト
(1 行)
```

7.5.6 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:
    テーブル"orders"への挿入、更新は外部キー制約"orders_customer_id_fkey"に違反
しています
DETAIL:  テーブル"customer"にキー(customer_id)=(10)がありません
ossdb=# SELECT currval('order_id_seq');
 currval
---------
     102
(1 行)

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 | 2024-04-17 14:38:11.97433  |           2 |       1 |   7
      103 | 2024-04-17 14:43:04.379357 |           1 |       2 |   5
(2 行)
```

8 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。この章ではマルチユーザーによるデータベースの利用について解説します。

8.1 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。

```
ossdb=# CREATE USER sato PASSWORD 'sato';
CREATE ROLE
ossdb=# \du
                                             ロール一覧
 ロール名 |                                      属性
                                        | 所属グループ
----------+-------------------------------------------------------------------------
    -----+--------------
 postgres | スーパユーザ, ロール作成可, DB作成可, レプリケーション可, RLS
     のバイパス | {}
 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@host1 ~]$ 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@host1 ~]$ psql ossdb
ossdb=# \du
                                             ロール一覧
 ロール名 |                                      属性
                                        | 所属グループ
----------+-------------------------------------------------------------------------
    -----+--------------
 postgres | スーパユーザ, ロール作成可, DB作成可, レプリケーション可, RLS
     のバイパス | {}
 sato     |
                                                                                 |
     {}
 suzuki   |
                                                                                 |
     {}
```

8.1.1 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。

8.1.2 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。

8.2 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。

8.2.1 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@host1 ~]$ cat /var/lib/pgsql/data/pg_hba.conf
（略）
# TYPE  DATABASE        USER            ADDRESS                 METHOD

# "local" is for Unix domain socket connections only
local   all             all                                     peer
# IPv4 local connections:
host    all             all             127.0.0.1/32            ident
# IPv6 local connections:
host    all             all             ::1/128                 ident
（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。

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。
- クライアントのアドレス（ADDRESS）接続を許可するクライアントのアドレスを指定します。指定しないとすべてのクライアントが対象となります。
- 認証方法（METHOD）
  認証方式を指定します。

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

実習環境の設定では、ローカルホストでの接続（local）で、すべてのデータベース、ユーザーに対して Peer 認証を行うことが設定されています。Peer 認証は、OS のユーザー名とデータベースのユーザー名が一致していることを確認する認証です。

8.2.2 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コマンドを実行した OS ユーザー postgres が指定されているのが分かります。

```
[postgres@host1 ~]$ id
uid=1001(postgres) gid=1001(postgres) groups=1001(postgres)
    context=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
[postgres@ host1 ~]$ psql ossdb
psql (13.14)
"help"でヘルプを表示します。

ossdb=# \set
※不要な設定値は削除しています
DBNAME = 'ossdb'
USER = 'postgres'
```

次に、ユーザー sato で接続した場合の例です。接続で使用するユーザー名は psql の 2 番目の引数として指定します。Peer 認証なので、接続に失敗します。

```
[postgres@host1 ~]$ psql ossdb sato
psql: エラー: FATAL:  ユーザ"sato"で対向(peer)認証に失敗しました
```

8.3 パスワード認証の設定

ユーザー名を変えてデータベースに接続できるよう、パスワード認証の設定を行ってみましょう。

8.3.1 ユーザーのパスワードの設定

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、まず先にユーザーのパスワードを設定しておきます。

既存のユーザーに対しては 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、通常は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は以下の通りです。

```
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。

```
ossdb=# ALTER USER postgres PASSWORD 'postgres';
ALTER ROLE
```

8.3.2　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 の設定で認証方式を peer から md5 に変更します。設定の変更は OS ユーザーを postgres にして行います。

```
[postgres@host1 ~]$ vi /var/lib/pgsql/data/pg_hba.conf
（省略）
# TYPE  DATABASE        USER            ADDRESS                 METHOD

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

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。パスワードの設定を行わないまま本設定を読み込むと、Peer 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ので注意してください。

8.3.3　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、OS の管理ユーザー admin で、systemctl コマンドで PostgreSQL の設定を再読み込みしています。OS のユーザーが postgres ではない点に注意してください。

```
[admin@host1 ~]$ sudo systemctl reload postgresql
```

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、設定の再読み込み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。

8.3.4　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 での接続を確認しています。

```
[postgres@host1 ~]$ psql ossdb
ユーザ postgres のパスワード: ※postgresと入力
psql (13.14)
"help"でヘルプを表示します。
```

```
ossdb=#
```

8.3.5　その他のユーザー

その他のユーザーにもパスワードを設定して接続できるようにします。

以下の例では、ユーザー sato にパスワードを設定して、接続ユーザーを sato に切り替えています。

```
ossdb=# ALTER USER sato PASSWORD 'postgres';
ALTER ROLE
ossdb=# \q
[postgres@host1 ~]$ psql ossdb sato
ユーザ sato のパスワード: ※postgresと入力
psql (13.14)
"help"でヘルプを表示します。

ossdb=>
```

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。

8.4　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。

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@host1 ~]$ vi /var/lib/pgsql/data/postgresql.conf
（略）
#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'; use '*' for
                                            all
                                        # (change requires restart)
#port = 5432                            # (change requires restart)
（略）
```

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ローカルループバックアドレス（127.0.0.1/32・::1/128）からの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 postgresql.conf も変更していますので、PostgreSQL の再起動で設定を反映させる必要があります。

```
[postgres@host1 ~]$ vi /var/lib/pgsql/data/pg_hba.conf
（略）
# 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
（略）
```

8.4.1 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。

以下の例は、OS の管理ユーザー admin で、systemctl コマンドで PostgreSQL の設定を再起動しています。OS のユーザーが postgres ではない点に注意してください。

```
[admin@host1 ~]$ sudo systemctl restart postgresql
```

8.4.2 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、-U(大文字) でユーザ名を指定します。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。

```
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、サーバーのローカルループバックアドレス（127.0.0.1）に対してネットワーク経由接続を行っています。これまでのローカル接続（UNIX ドメインソケット接続）と、ローカルループバックアドレス（TCP/IP 接続）は接続経路が異なります。

```
[postgres@host1 ~]$ psql -h localhost -p 5432 -U postgres ossdb
ユーザ postgres のパスワード: ※postgresと入力
psql (13.14)
"help"でヘルプを表示します。

ossdb=#
```

8.5 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。

8.5.1 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。

GRANT 文の構文は以下の通りです。

```
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。

```
ossdb=# \dt prod
          リレーション一覧
 スキーマ | 名前 |  タイプ  |  所有者
----------+------+----------+----------
 public   | prod | テーブル | postgres
(1 行)
```

```
ossdb=# GRANT all ON prod TO sato;
GRANT
```

8.5.2 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
                                アクセス権限
 スキーマ | 名前 |  タイプ  |       アクセス権限         | 列の権限 | ポリシー
----------+------+----------+----------------------------+----------+----------
 public   | prod | テーブル | postgres=arwdDxt/postgres+|          |
          |      |          | sato=arwdDxt/postgres     |          |
(1 行)
```

アクセス権の表記は以下の通りです。

```
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「/postgres」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。

8.5.3 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。

REVOKE 文の構文は以下の通りです。

```
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。

```
ossdb=# REVOKE all ON prod FROM sato;
REVOKE
ossdb=# \dp prod
                                アクセス権限
 スキーマ | 名前 |  タイプ  |       アクセス権限         | 列の権限 | ポリシー
----------+------+----------+----------------------------+----------+----------
 public   | prod | テーブル | postgres=arwdDxt/postgres |          |
(1 行)
```

8.6 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 行)

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

トランザクションが ROLLBACK されて、INSERT が取り消されているのが確認できました。

以下の例ではトランザクションを COMMIT します。

```
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 行)
```

トランザクションが COMMIT されて、INSERT が確定しました。

8.6.1 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後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 WHERE prod_id = 3;
 prod_id | prod_name | price
---------+-----------+-------
```

```
       3 | メロン     |   100
(1 行)

ossdb=# BEGIN;
BEGIN
ossdb=# UPDATE prod SET price = 120 WHERE prod_id = 3;
UPDATE 1
ossdb=# SELECT * FROM prod WHERE prod_id = 3;
 prod_id | prod_name | price
---------+-----------+-------
       3 | メロン     |   120
(1 行)
```

COMMIT していない状態で、別のターミナルを起動してデータベースに接続し、検索を行ってみます。

```
[postgres@host1 ~]$ psql ossdb
ユーザ postgres のパスワード:
psql (13.14)
"help"でヘルプを表示します。

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

COMMIT 前の行データが検索されます。

トランザクションを実行していたターミナルに戻り、トランザクションを確定させます。

```
ossdb=# COMMIT;
COMMIT
```

再度、新しい方のターミナルに戻り、検索を行ってみます。

```
※以下は前の実行結果
ossdb=# SELECT * FROM prod WHERE prod_id = 3;
 prod_id | prod_name | price
---------+-----------+-------
       3 | メロン     |   100
(1 行)
※以下は別ターミナルでCOMMITされた後の実行結果
ossdb=# SELECT * FROM prod WHERE prod_id = 3;
 prod_id | prod_name | price
---------+-----------+-------
       3 | メロン     |   120
(1 行)
```


prod_id
prod_name
price
1
みかん
50
2
りんご
70
3
メロン
100
3
メロン
120
①BEGIN
②UPDATE→
③SELECT
更新後の値が追記され、
未確定の状態
①BEGINのトランザクションがCOMMITされていないので、
②UPDATE後の行データは
③SELECTで読み取られない
prod_id
prod_name
price
1
みかん
50
2
りんご
70
3
メロン
100
3
メロン
120
④COMMIT→
⑤SELECT
更新前の行は不要なものとして
読み飛ばす
更新後の値が確定された状態
④更新が確定された後は、
⑤SELECTで更新後のデータを読み取ることができる
（更新前の古いデータは読み飛ばす）

図 10: 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。

8.6.2　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 行)
```

別のターミナルで prod 表の同じ行データを更新します。

```
ossdb=# SELECT * FROM prod WHERE prod_id =1;
 prod_id | prod_name | price
---------+-----------+-------
       1 | みかん    |    50
(1 行)

ossdb=# BEGIN;
BEGIN
ossdb=# UPDATE prod SET price = price * 1.2 WHERE prod_id = 1;
※結果が返らず、制御も戻りません
```

図 11: 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。

```
※以下は前の実行結果
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)
※以下でROLLBACKを実行
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。

```
※以下は前の実行結果
ossdb=# BEGIN;
BEGIN
ossdb=# UPDATE prod SET price = price * 1.2 WHERE prod_id = 1;
※結果が返らず、制御も戻らない状態
※別ターミナルでROLLBACKが実行されるとUPDATEが完了する
UPDATE 1
ossdb=*# SELECT * FROM prod WHERE prod_id =1;
 prod_id | prod_name | price
---------+-----------+-------
       1 | みかん    |    60
(1 行)

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。

8.6.3 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。

以下の例では、2 つのトランザクションでデッドロックを発生させています。

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、別のターミナルで実行した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。

```
※以下は前の実行結果
ossdb=# BEGIN;
BEGIN
ossdb=# UPDATE prod SET price = price * 1.1 WHERE prod_id = 1;
UPDATE 1
※更新の競合が発生するUPDATEを実行
ossdb=# UPDATE prod SET price = price * 1.2 WHERE prod_id = 2;
※結果が返らず、制御も戻りません
```

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。

```
※以下は前の実行結果
ossdb=# BEGIN;
BEGIN
ossdb=# UPDATE prod SET price = price * 1.1 WHERE prod_id = 2;
UPDATE 1
※デッドロックが発生するUPDATEを実行
ossdb=# UPDATE prod SET price = price * 1.2 WHERE prod_id = 1;
ERROR:  デッドロックを検出しました
DETAIL:  プロセス 43233 は ShareLock を トランザクション 599
    で待機していましたが、プロセス 43040 でブロックされました
プロセス 43040 は ShareLock を トランザクション 600
    で待機していましたが、プロセス 43233 でブロックされました
HINT:  問い合わせの詳細はサーバログを参照してください
CONTEXT:  リレーション"prod"のタプル(0,8)の更新中
```

デッドロックを検出し、トランザクションが強制的にロールバックされると、更新の競合が発生していた UPDATE は実行されます。

```
ossdb=# BEGIN;
BEGIN
ossdb=*# UPDATE prod SET price = price * 1.1 WHERE prod_id = 1;
UPDATE 1
ossdb=*# UPDATE prod SET price = price * 1.2 WHERE prod_id = 2;
UPDATE 1
```

図 12: 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。

9 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。この章ではパフォーマンスチューニングについて解説します。

9.1 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。

9.1.1 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、主キーを定義していると自動的にインデックスが作成されます。

```
ossdb=# \d prod
                    テーブル"public.prod"
    列     | タイプ  | 照合順序 | Null 値を許容 | デフォルト
-----------+---------+----------+---------------+------------
 prod_id   | integer |          | not null      |
 prod_name | text    |          |               |
 price     | integer |          |               |
インデックス:
    "prod_pkey" PRIMARY KEY, btree (prod_id)
参照元:
    TABLE "orders" CONSTRAINT "orders_prod_id_fkey" FOREIGN KEY (prod_id)
        REFERENCES prod(prod_id)
```

9.1.2 インデックスの作成

手動でインデックスを作成するには、CREATE INDEX 文を使用します。

インデックスの作成の構文は以下の通りです。

```
CREATE INDEX インデックス名 ON 表名(列名[,...]);
```

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
                              テーブル"public.orders"
     列      |            タイプ             | 照合順序 | Null 値を許容 |
          デフォルト
-------------+-------------------------------+----------+---------------+----------
 order_id    | integer                       |          | not null      |
 order_date  | timestamp without time zone |          |               |
```

```
 customer_id | integer                     |          |               |
 prod_id     | integer                     |          |               |
 qty         | integer                     |          |               |
インデックス:
    "orders_pkey" PRIMARY KEY, btree (order_id)
    "orders_customer_id_idx" btree (customer_id)
外部キー制約:
    "orders_customer_id_fkey" FOREIGN KEY (customer_id) REFERENCES
        customer(customer_id)
    "orders_prod_id_fkey" FOREIGN KEY (prod_id) REFERENCES prod(prod_id)
```

9.1.3 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。

インデックスの削除の構文は以下の通りです。

```
DROP INDEX インデックス名;
```

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
                          テーブル"public.orders"
     列      |           タイプ            | 照合順序 | Null 値を許容 |
         デフォルト
-------------+-----------------------------+----------+---------------+----------
 order_id    | integer                     |          | not null      |
 order_date  | timestamp without time zone |          |               |
 customer_id | integer                     |          |               |
 prod_id     | integer                     |          |               |
 qty         | integer                     |          |               |
インデックス:
    "orders_pkey" PRIMARY KEY, btree (order_id)
外部キー制約:
    "orders_customer_id_fkey" FOREIGN KEY (customer_id) REFERENCES
        customer(customer_id)
    "orders_prod_id_fkey" FOREIGN KEY (prod_id) REFERENCES prod(prod_id)
```

9.1.4 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 SQL 実行プランの分析などを行って確認する必要があるでしょう。

9.2 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。

9.2.1 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..4226.62 rows=1 width=140)
   Filter: (newzip = '1500002'::bpchar)
(2 行)
```

9.2.2 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=140)
   Index Cond: (newzip = '1500002'::bpchar)
(2 行)
```

9.2.3 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。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。

```
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..4226.62 rows=121597 width=140)
   Filter: (largearea = 0)
(2 行)

ossdb=# EXPLAIN SELECT * FROM zip WHERE largearea = 1;
                                   QUERY PLAN
--------------------------------------------------------------------------------
 Index Scan using zip_largearea on zip  (cost=0.29..719.21 rows=2773 width=140)
   Index Cond: (largearea = 1)
(2 行)

ossdb=# SELECT largearea,count(*) FROM zip GROUP BY largearea;
 largearea | count
-----------+--------
         0 | 121714
         1 |   2656
(2 行)
```

この例はたとえば書籍の索引と同じで、PostgreSQL に関する技術書で「PostgreSQL」という頻出する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か、異なる単語で目次や索引を見るほうがはるかに効率的だからです。

9.3 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。

バキューム処理は自動的に行われるように設定されていますが、手動で実行することもできます。

9.3.1 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。

9.3.2 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。

9.3.3 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
```

9.3.4 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@host1 ~]$ ps ax | grep autovacuum
  37358 ?        Ss     0:10 postgres: autovacuum launcher
  60901 pts/1    S+     0:00 grep --color=auto autovacuum
```

9.4 データベースのクラスタ化によるデータの再配置

データベースのクラスタ化は、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。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:  テーブル"orders"には事前にクラスタ化されたインデックスはありません
ossdb=# CLUSTER orders USING orders_pkey;
CLUSTER
ossdb=# \d orders
                              テーブル"public.orders"
     列      |            タイプ             | 照合順序 | Null 値を許容 |
         デフォルト
-------------+-------------------------------+----------+---------------+----------
 order_id    | integer                       |          | not null      |
 order_date  | timestamp without time zone |          |               |
 customer_id | integer                       |          |               |
 prod_id     | integer                       |          |               |
 qty         | integer                       |          |               |
インデックス:
    "orders_pkey" PRIMARY KEY, btree (order_id) CLUSTER
外部キー制約:
    "orders_customer_id_fkey" FOREIGN KEY (customer_id) REFERENCES
        customer(customer_id)
    "orders_prod_id_fkey" FOREIGN KEY (prod_id) REFERENCES prod(prod_id)
```

10　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。この章ではバックアップとリストアについて解説します。

10.1　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。

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。システム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。

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。

10.2　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、OS ユーザー admin で PostgreSQL を停止した後、OS ユーザー postgres で 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以下をアーカイブしています。

```
[admin@host1 ~]$ sudo systemctl stop postgresql
[admin@host1 ~]$ sudo systemctl status postgresql
```

```
○ postgresql.service - PostgreSQL database server
     Loaded: loaded (/usr/lib/systemd/system/postgresql.service; disabled;
         preset: disabled)
     Active: inactive (dead)

[admin@host1 ~]$ su - postgres
パスワード:
[postgres@host1 ~]$ tar cvf backup.tar /var/lib/pgsql/data
tar: メンバ名から先頭の `/' を取り除きます
/var/lib/pgsql/data/
/var/lib/pgsql/data/pg_wal/
/var/lib/pgsql/data/pg_wal/archive_status/
（略）
/var/lib/pgsql/data/postgresql.conf
/var/lib/pgsql/data/pg_hba.conf
/var/lib/pgsql/data/current_logfiles
[postgres@host1 ~]$ ls -l backup.tar
-rw-r--r--. 1 postgres postgres 94341120  4月 21 13:27 backup.tar
```

10.3 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。オプションなどを指定しないと、テキストの SQL 文が出力されます。

```
[postgres@host1 ~]$ pg_dump ossdb > backup.sql
パスワード: ※postgresと入力
[postgres@host1 ~]$ head backup.sql
--
-- PostgreSQL database dump
--

-- Dumped from database version 13.14
-- Dumped by pg_dump version 13.14

SET statement_timeout = 0;
SET lock_timeout = 0;
SET idle_in_transaction_session_timeout = 0;
[postgres@host1 ~]$ tail backup.sql
--

ALTER TABLE ONLY public.orders
    ADD CONSTRAINT orders_prod_id_fkey FOREIGN KEY (prod_id) REFERENCES
        public.prod(prod_id);


--
-- PostgreSQL database dump complete
--
```

pg_dump は実行時オプションとして、出力形式やファイルでの出力先の指定、処理の並列化などを指定できます。詳しくはマニュアルを参照してください。

10.4　psql 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、psql コマンドにリダイレクトで読み込ませることでリストアすることができます。リストアをするには、テンプレートデータベース template0 から新しいデータベースを作成し、そのデータベースにリストアを行います。テンプレートデータベースは新規にデータベースを作成する際に雛形となるデータベースです。通常のデータベース作成では template1 が雛形として使用されますが、pg_dump コマンドのバックアップからのリストアの際には template0 を使用します。

以下の例では、バックアップ用に新たにデータベース ossdb2 を作成し、リストアを行っています。

```
[postgres@host1 ~]$ createdb -T template0 ossdb2
パスワード:
[postgres@host1 ~]$ psql ossdb2 < backup.sql
ユーザ postgres のパスワード:
SET
SET
SET
（略）
CREATE INDEX
ALTER TABLE
ALTER TABLE
```

リストアできたデータベースに接続して、リストアされたことを確認します。

```
[postgres@host1 ~]$ psql ossdb2
ユーザ postgres のパスワード:
psql (13.14)
"help"でヘルプを表示します。

ossdb2=# \d
                リレーション一覧
 スキーマ |     名前     |   タイプ   |  所有者
----------+--------------+------------+----------
 public   | char_test    | テーブル   | postgres
 public   | customer     | テーブル   | postgres
 public   | date_test    | テーブル   | postgres
 public   | numeric_test | テーブル   | postgres
 public   | order_id_seq | シーケンス | postgres
 public   | orders       | テーブル   | postgres
 public   | prod         | テーブル   | postgres
 public   | student      | テーブル   | postgres
 public   | varchar_test | テーブル   | postgres
 public   | zip          | テーブル   | postgres
(10 行)

ossdb2=# SELECT count(*) FROM zip;
 count
--------
 124370
(1 行)
```

オープンソースデータベース標準教科書

2025 年 5 月 1 日　Ver.3.0.1

発行元　LPI-Japan