

本書は、PostgreSQL の公式ドキュメントを参考に執筆しています。関連する公式ドキュメントへのリンクを適宜掲載しています。

<https://www.postgresql.org/docs/17/index.html>

PostgreSQL Database Management System
(also known as Postgres, formerly known as Postgres95)

Portions Copyright (c) 1996-2026, PostgreSQL Global Development Group

Portions Copyright (c) 1994, The Regents of the University of California

Permission to use, copy, modify, and distribute this software and its documentation for any purpose, without fee, and without a written agreement is hereby granted, provided that the above copyright notice and this paragraph and the following two paragraphs appear in all copies.

IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, INCLUDING LOST PROFITS, ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF THE UNIVERSITY OF CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE SOFTWARE PROVIDED HEREUNDER IS ON AN "AS IS" BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATIONS TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

本書の内容に基づく運用結果について、著者、ソフトウェアの開発元および提供元、株式会社技術評論社は一切の責任を負いかねますので、あらかじめご了承ください。

本書に記載されている情報は、特に断りがない限り、執筆時点（2025年）の情報に基づいています。ご使用時には変更されている可能性がありますのでご注意ください。

本書に記載されている会社名・製品名は、一般に各社の登録商標または商標です。本書中では、™、©、® マークなどは表示していません。

はじめに

PostgreSQL は、世界中で広く利用されているオープンソースの RDBMS (*Relational DataBase Management System*: リレーショナルデータベース管理システム) です。その歴史は 1986 年にカリフォルニア大学バークレイ校で研究用に開発された「Postgres」までさかのぼり、約 40 年にわたって開発されてきました。日本国内においても 1999 年に日本 PostgreSQL ユーザ会が設立されました。この結果、かつては商用ソフトウェアが主流であったエンタープライズ領域やミッションクリティカル領域においても PostgreSQL の利用が進み、また AWS、Google、Microsoft といったパブリッククラウドを提供する各社から PostgreSQL をベースとした RDBMS マネージドサービスが提供されるなど、現在も利用が拡大しています。

本書は、システム開発・運用の現場で PostgreSQL を活用するための解説書です。対象読者として DBA (*DataBase Administrator*: データベース設計・管理者) や DB を使用するアプリケーション開発者を想定しています。現場で使うための基礎的な内容から実践的な使用方法まで解説するほか、その仕組みやパフォーマンスを向上させる方法などについても解説します。

第 1 章では一般的な RDBMS の基礎的な解説から始め、PostgreSQL の歴史やアーキテクチャの概要について解説します。第 2 章ではインストール方法として、Linux でのバイナリパッケージを使ったインストール、ソースコードからビルドしてインストール、Windows でのインストールについて解説します。第 3 章では PostgreSQL サーバの起動・停止方法を解説するとともに、設定したパラメータが有効化されるタイミングの違いなどについて解説します。第 4 章では基礎的な SQL 入門として PostgreSQL の SQL の代表的な構文を解説します。第 5 章ではテーブル設計の際に重要となる、データ型、制約、インデックス、テーブルパーティショニングについて解説します。第 6 章では高度な SQL 機能として関数、プロシージャ、演算子、トリガ、ウィンドウ関数、全文検索について解説します。第 7 章では PostgreSQL が内部で行うクエリ処理の流れ、実行計画の作られ方と読み方を解説します。第 8 章ではデータベース負荷分散や可用性向上のためのレプリケーションを解説します。第 9 章ではデータベースどうしをつな

げて利用できる外部データラップを解説します。第 10 章ではデータベースのユーザーと権限の管理機構について解説します。第 11 章ではデータベースに接続する際のログイン認証、クライアントとサーバ間の通信暗号化、データ暗号化について解説します。第 12 章ではデータベースのバックアップ・リストアについて解説します。第 13 章では PostgreSQL を最適な性能で運用するためのテーブルやインデックスのメンテナンスについて解説します。第 14 章では PostgreSQL を安定的に運用するために重要なモニタリングについて解説します。第 15 章では性能トラブルに対応するためのパフォーマンスチューニングについて解説します。Appendix A では PostgreSQL の主要なパラメータについて設定の基本的な考え方を述べます。

なお本書は執筆時点での PostgreSQL のメジャーバージョンである 17 を前提に解説していますが、2025 年 9 月にリリースされた PostgreSQL 18 についても Appendix B にて新機能を解説しています。

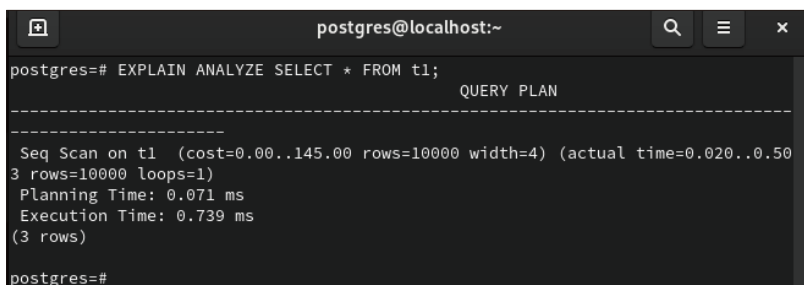
本書の執筆者陣は、NTT オープンソースソフトウェアセンタ（以降、NTT OSS センタ）で PostgreSQL をベースとした研究、PostgreSQL コミュニティでの開発、NTT グループ各社への技術サポートを行ってきたメンバーです。NTT OSS センタは 2006 年に当時の NTT 持株会社の研究開発組織内に設置され、2026 年で 20 周年を迎えます。20 年前はエンタープライズ領域での PostgreSQL の利用はまだ一般的ではありませんでしたが、それが現在は当たり前となり重要性はいっそう増えています。PostgreSQL に関わる人の世代が変わっていく状況もありますが、PostgreSQL コミュニティが継続して発展していくことに、本書を通じて少しでもお役に立てれば幸いです。

読んでいただく際の留意事項

横幅が長い PostgreSQL の実行結果の表示

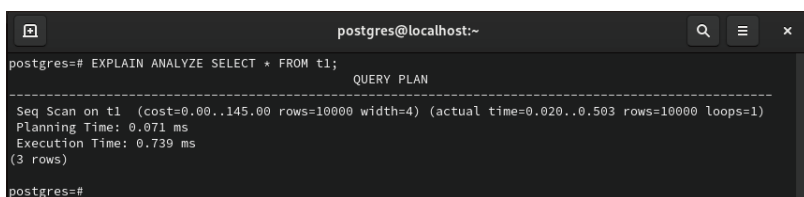
本書では解説の中で、PostgreSQL の実行結果のコンソール画面表示を掲載しています。コンソール画面の横幅はあまり大きくせずに運用することも多いので、コンソール画面での表示に合わせて本書でも 80 文字で折り返しています。

図 1 一般的な横幅にしたコンソールでの表示



```
postgres@localhost:~  
postgres=# EXPLAIN ANALYZE SELECT * FROM t1;  
                                QUERY PLAN  
-----  
Seq Scan on t1 (cost=0.00..145.00 rows=10000 width=4) (actual time=0.020..0.50  
3 rows=10000 loops=1)  
Planning Time: 0.071 ms  
Execution Time: 0.739 ms  
(3 rows)  
postgres=#
```

図 2 画面を横に伸ばした表示



```
postgres@localhost:~  
postgres=# EXPLAIN ANALYZE SELECT * FROM t1;  
                                QUERY PLAN  
-----  
Seq Scan on t1 (cost=0.00..145.00 rows=10000 width=4) (actual time=0.020..0.503 rows=10000 loops=1)  
Planning Time: 0.071 ms  
Execution Time: 0.739 ms  
(3 rows)  
postgres=#
```

PostgreSQL の実行結果は表の形で出力され、表の線が「-----」で出力されます。線の「-」の個数は、その実行結果における横幅の最大文字数と同じです。この「-----」が 2 行にわたって続いている場合は、実行結果がコンソール画面の横幅を超えて折り返されていることを示しています。

PostgreSQL コミュニティの公式ドキュメントへのリンク

本書の解説の中では、公式ドキュメント（英語）へのリンクを記載しています。本書は PostgreSQL 17 を前提としたことから、すべてのリンクはバージョン 17 へのドキュメントのリンクとなっています。最新版（2026 年 1 月時点ではバージョン 18）のドキュメントを参照したい場合は、URL 内の「17」を「current」に置き換えていただくことで最新版のドキュメントを参照できます。

- バージョン 17 の「1. What Is PostgreSQL?」
<https://www.postgresql.org/docs/17/intro-what-is.html>
- 最新版の「1. What Is PostgreSQL?」
<https://www.postgresql.org/docs/current/intro-what-is.html>

また、日本 PostgreSQL ユーザ会による日本語版ドキュメントで参照したい場合は、www.postgresql.org/docs/17 を www.postgresql.jp/document/17/html/ に置き換えることで、日本語版のドキュメントを参照できます。

- 日本語版バージョン 17 の「1. What Is PostgreSQL?」
<https://www.postgresql.jp/document/17/html/intro-what-is.html>

謝辞

今回本書を出版できるのは、PostgreSQL という素晴らしいオープンソースの DBMS が長年にわたって継続的に開発が積み重ねられ、利用され続けてきたおかげです。世界中の PostgreSQL コミュニティに関わるすべての皆様に心より感謝申し上げます。国内においては日本 PostgreSQL ユーザ会の皆様が当初から PostgreSQL の普及促進を担い、牽引し続けていただいたことにもたいへん感謝しています。また企業での利用を促進するため、ミッションクリティカル性の高いエンタープライズ領域への PostgreSQL の適用に向けて取り組んできた PostgreSQL エンタープライズ・コンソーシアムの皆様にも、技術ノウハウ蓄積の協力やさまざまな意見交換もさせていただきたいへん感謝しています。

最後に、今回の執筆の機会を与えていただいた技術評論社の池田様には、いくら感謝してもしきれないほどたいへんお世話になりました。我々執筆陣に対してご親切かつご丁寧なご指導をいただき、たいへん感謝しています。

はじめに.....	iii
読んでいただく際の留意事項.....	v
謝辞.....	vii
目次.....	viii

第1章

PostgreSQL の概要 1

リレーショナルデータベースとは.....	2
データベースとは.....	2
データモデル.....	3
リレーショナルデータベースの役割.....	5
トランザクション管理.....	6
原子性 (Atomicity).....	6
一貫性 (Consistency).....	7
分離性 (Isolation).....	7
耐久性 (Durability).....	9
トランザクション分離レベル.....	9
トランザクション分離レベルにより発生する異常.....	10
ダーティーリード.....	10
ノンリピータブルリード.....	11
ファントムリード.....	12
まとめ.....	12
障害回復——チェックポイントと WAL.....	13
SQL の解釈・実行.....	14
PostgreSQL とは.....	15
PostgreSQL の歴史.....	15
PostgreSQL 開発コミュニティ.....	16
リリーススケジュール.....	16
開発リポジトリ.....	17
メジャーバージョンごとの新機能.....	17
PostgreSQL のアーキテクチャ概要.....	18
プロセス構成.....	18
ファイル、ディレクトリ構成.....	19
ディレクトリ.....	20
ファイル.....	21
システムカタログ—— PostgreSQL の管理情報.....	22
トランザクション——データのー貫性を保つ.....	23
レプリケーション——データベースを複製する.....	25
データベースの可用性 (耐障害性) を上げる.....	25
負荷分散で性能を上げる.....	25
クエリ最適化.....	26
拡張機能.....	27

第2章**インストール**

29

Rocky Linux へのインストール	30
OS のインストールと初期設定.....	30
PostgreSQL 開発コミュニティ提供の RPM リポジトリ.....	30
RPM リポジトリの設定.....	31
Rocky Linux の AppStream に含まれる PostgreSQL モジュールの無効化.....	32
PostgreSQL のインストール.....	32
データベースクラスタの作成.....	34
自動起動の設定.....	35
column データベースクラスタのディレクトリを変更する.....	35
PostgreSQL サーバの起動・停止.....	36
アンインストール.....	38
【参考】 Rocky Linux のモジュールを用いた PostgreSQL のインストール.....	39
ソースコードからビルドしてインストール	42
必要なツールのインストール.....	43
ソースコードの取得.....	45
ビルドとインストール.....	47
スーパーユーザーの作成.....	48
PostgreSQL サーバの起動・停止.....	48
環境変数の設定.....	50
アンインストール.....	50
Windows へのインストール	50
インストーラのダウンロード.....	51
PostgreSQL のインストール.....	51
Application Stack Builder によるアプリケーションのインストール.....	56
環境変数の設定.....	60
アンインストール.....	63

第3章**PostgreSQL の起動・停止と設定パラメータ**

65

initdb —— データベースクラスタの初期化	66
initdb コマンドの実行.....	67
自動設定の内容.....	67
PostgreSQL の起動・停止	71
postgres コマンドの直接起動.....	71
pg_ctl コマンドを用いた起動.....	72
pg_ctl コマンドを用いた停止.....	73
サービスとしての起動.....	74
サーバ設定	75

postgresql.conf 設定ファイル.....	76
設定パラメータ	77
GUC コンテキスト.....	77
設定パラメータがどの状況で変更可能かを確認する方法	77
設定ソース.....	78
設定方法	79
a サーバ起動時に読み込まれる設定	79
1. pg_ctl (postgres) コマンドラインオプション (ソース名: command line)	80
2. SQL コマンド ALTER SYSTEM SET (ソース名: configuration file)	80
3. postgresql.conf (ソース名: configuration file)	80
4. 環境変数 (ソース名: environment variable).....	81
b 設定リロード時に読み込まれる設定	81
c セッション開始時に行われる設定変更	82
1. 接続オプションによるセッション単位の設定変更	82
2. ALTER ROLE (IN DATABASE) SET、ALTER DATABASE SET によるセッション単位の設定変更	84
d セッション内で行う設定変更	85
SET コマンドによる変更	85
関数実行時の変更	85
設定の確認および変更	86
設定の確認方法	86
SHOW コマンド—— 現在の設定値を確認する	86
pg_settings ビュー—— 設定項目の現時点における詳細情報を表示する	87
pg_file_settings ビュー—— 設定ファイル内の情報を表示する	87
pg_hba_file_rules ビュー—— pg_hba.conf ファイル内の情報を表示する	89
起動時およびリロード時に読み込まれる設定の変更	90
a 設定ファイルを直接編集する方法	90
b コマンドを使って設定ファイルを変更する方法	91
設定ファイルの再読み込み (サーバリロード).....	91

第4章

SQL 入門

95

PostgreSQL のデータベースクラスタ構造.....	96
SQL 構文の記法.....	97
データベースの作成・変更・削除.....	97
構文.....	97
データベースの作成.....	97
データベースの変更.....	98
データベースの削除.....	98
パラメータ	98
使用例.....	99
スキーマの操作	100
構文.....	100
スキーマの作成.....	100
スキーマの変更.....	100
スキーマの削除.....	100

パラメータ	101
使用例	101
テーブルの操作	102
構文	102
テーブルの作成	102
テーブルの変更	102
テーブルの削除	103
パラメータ	103
使用例	103
インデックスの操作	104
構文	105
インデックスの作成	105
インデックスの変更	105
インデックスの削除	105
パラメータ	105
使用例	106
テーブルスペースの操作	106
構文	106
テーブルスペースの作成	106
テーブルスペースの変更	107
テーブルスペースの削除	107
パラメータ	107
使用例	108
シーケンスの操作	108
構文	108
シーケンスの作成	108
シーケンスの変更	108
シーケンスの削除	109
パラメータ	109
シーケンス操作関数	109
使用例	110
データの挿入 (INSERT)	111
構文	111
パラメータ	111
使用例	112
データの変更 (UPDATE)	112
構文	113
パラメータ	113
使用例	113
データの削除 (DELETE)	114
構文	114
パラメータ	114
使用例	114
データの参照	114
構文	115
パラメータ	115

DISTINCT 句	115
SELECT リスト	115
FROM 句	115
WHERE 句	116
GROUP BY 句	116
HAVING 句	116
ORDER BY 句	117
LIMIT 句	117
OFFSET 句	117
FOR UPDATE 句	117
使用例	118
ビューの操作	121
構文	121
ビューの作成	121
ビューの変更	121
ビューの削除	122
パラメータ	122
使用例	122
トランザクション	123
構文	123
トランザクションの開始 (BEGIN)	123
トランザクションの完了 (COMMIT)	124
トランザクションの中断 (ROLLBACK)	124
セーブポイント (SAVEPOINT)	124
ロック (LOCK TABLE)	124
パラメータ	125
ISOLATION LEVEL	125
transaction_mode	125
SAVEPOINT	126
TABLE LOCK	126
使用例	126
データコピー (COPY)	127
構文	127
パラメータ	128
使用例	129
テーブルデータを全削除 (TRUNCATE)	130
構文	130
パラメータ	131
使用例	131

第5章**テーブル設計**

133

データ型	134
数値データ型 (int 型、real 型、numeric 型、serial 型など)	134
整数	134
浮動小数	134
任意精度型	135

シーケンスと連番型	136
文字型 (char 型、varchar 型、text 型など)	137
日付/時刻データ型 (timestamp 型、date 型、interval 型など)	138
列挙型 (enum 型)	139
バイナリ列データ型 (bytea 型)	140
JSON 型 (json 型、jsonb 型)	140
配列型	141
範囲型	142
制約——格納するデータを限定する	143
主キー制約——主キーであることを指定する	144
CHECK 制約——特定の条件を満たすデータのみであることを指定する	145
一意性制約——データが一意であることを指定する	145
非 NULL 制約——列が NULL にならないことを指定する	146
外部キー制約——ほかのテーブルに存在する行の値に限定する	146
排他制約——データの重なりがないことを指定する	148
インデックス——クエリ処理の高速化	149
B-tree インデックス	149
B-tree インデックスの構造	150
B-tree インデックスの作成と高速化の確認	151
重複排除	157
Hash インデックス	159
Hash インデックスの構造	159
Hash インデックスの作成と高速化の確認	160
GiST インデックス	161
GiST インデックスを用いた R-tree の構造	161
GiST インデックスの作成と高速化の確認	162
SP-GiST インデックス	164
SP-GiST インデックスを用いた Quad-tree の構造	164
GIN インデックス	165
GIN インデックスを用いた全文検索	165
BRIN インデックス	169
BRIN インデックスの作成と高速化の確認	170
その他の高度なインデックス	174
複数列インデックス	174
一意インデックス	174
式に対するインデックス	175
部分インデックス	176
インデックス設計の注意点	177
テーブルパーティショニング	177
テーブルパーティショニングとは	177
テーブルパーティショニングの特徴	178
パーティションブルーニング——子テーブルのスキャン省略	179
レンジパーティショニング——連続する値でパーティションを分割	180
リストパーティショニング——値を複数指定してパーティションを分割	181
ハッシュパーティショニング——列のハッシュ値をもとにパーティションを分割	181

第6章

高度な SQL 機能

183

関数・プロシージャ・演算子.....	184
PostgreSQL における関数・プロシージャ・演算子の管理の概要.....	184
関数とプロシージャの定義・変更・削除.....	185
演算子の定義・変更・削除.....	188
インデックスアクセスの際の演算子に関する処理の流れ.....	189
演算子を普通に使う場合とインデックスアクセスで使う場合の違い.....	189
演算子クラスを指定したインデックス作成.....	190
インデックス検索の事前処理の例.....	192
PL/pgSQL.....	194
PL/Perl.....	196
インストール.....	196
関数の定義・変更・削除.....	196
trusted / untrusted.....	198
C 言語による関数の定義.....	200
必要な環境.....	200
定義の手順.....	200
関数の属性.....	203
IMMUTABLE/STABLE/VOLATILE 属性.....	203
LEAKPROOF 属性.....	204
トリガ.....	204
トリガの概要.....	204
トリガの使用の流れ.....	205
トリガ関数の作成.....	206
トリガの作成.....	208
トリガ関数およびトリガの変更.....	210
トリガ関数およびトリガの削除.....	211
グルーピングセットウィンドウ関数——データの集計を効率的に行う.....	211
グルーピングセット (GROUPING SETS、ROLLUP、CUBE).....	212
ウィンドウ関数.....	216
ウィンドウ関数の詳細.....	218
ウィンドウが持つ要素.....	218
フレーム定義.....	219
フレームモードの種類.....	220
ウィンドウ関数の利用例.....	225
階層別集計.....	225
移動平均.....	226
汎用ウィンドウ関数.....	228
ユーザー定義集約関数をウィンドウ関数として利用する.....	229
全文検索.....	232
全文検索とは.....	232
tsvector 型を用いた全文検索.....	233
tsvector.....	233
tsquery.....	234
tsvector、tsquery を使った自然言語演算.....	235

全文検索インデックス	235
その他全文検索の補助機能	236
pg_trgm	237
pg_bigm を用いた日本語全文検索	239
pg_trgm との違い	240
pg_bigm のインストール	240
pg_bigm によるバイグラム (2 つの文字の組) への変換	241
全文検索インデックスの作成	241
pg_bigm で使用する関数や演算子	243

第7章

クエリ処理

245

クエリ処理の基礎	246
どのようにクエリを処理するか	246
クエリ処理の流れ	246
パーサー—SQL からパースツリーを作る	246
アナライザ—パースツリーからクエリツリーを作る	246
リライタ—クエリツリーを書き換える	247
プランナ/オブティマイザ—クエリツリーからプランツリーを作る	247
エグゼキュータ—クエリを実行する	249
本章内の実行例で使用するテーブルやデータ	249
テーブル作成	249
データ投入	252
テーブル統計情報—実行計画を作るための情報	255
テーブル統計情報の参照—pg_stats ビュー	255
テーブル統計情報の更新に関する留意点	257
実行計画	258
実行計画の取得	258
実行計画の読み方	259
複雑な実行計画	259
スキャン方法の種類と特徴	261
シーケンシャルスキャン	261
インデックススキャン	261
ビットマップスキャン	262
インデックスオンリースキャン	264
Tid スキャン	267
結合方法の種類と特徴	268
ネステッドループ結合	268
マージ結合	270
ハッシュ結合	271
その他のプランノード	273
column プランニングを強制的に変更する	274
クエリ処理の高速化	276
パラレルクエリ—複数 CPU を使ってクエリを並列に実行する	276
JIT コンパイル—クエリ実行時にコンパイルを行う	278
column パラレルクエリに関係するパラメータ	279
column JIT コンパイルに関係するパラメータ	281

第 8 章

レプリケーション

283

レプリケーション——データベースの多重化.....	284
レプリケーションの概要.....	284
データの信頼性制御と処理性能.....	284
非同期レプリケーションと同期レプリケーション.....	285
トランザクション単位での信頼性制御.....	285
転送データ形式の違いによるレプリケーションの種類と特徴.....	286
ストリーミングレプリケーションの概要と特徴.....	287
ストリーミングレプリケーションのしくみ.....	287
ストリーミングレプリケーションの基本的な処理の流れ.....	289
PostgreSQL における同期レプリケーションの動作の詳細.....	290
同期レプリケーションにおけるスタンバイのステータス確認方法の詳細.....	292
複数台のスタンバイの同期レプリケーション設定 (synchronous_standby_names).....	293
ホットスタンバイ構成におけるスタンバイ上での処理の衝突.....	295
ロジカルレプリケーションの概要と特徴.....	296
ロジカルレプリケーションのしくみ.....	298
ロジカルレプリケーションの基本的な処理の流れ.....	300
ロジカルレプリケーションにおける同期レプリケーション.....	302
ストリーミングレプリケーションの構成・運用例.....	302
プライマリの構築・設定.....	303
pg_hba.conf.....	303
postgresql.conf.....	304
スタンバイの構築・設定.....	306
postgresql.conf.....	306
ストリーミングレプリケーション構成の管理・メンテナンス.....	308
監視.....	309
遅延の測定.....	309
競合の監視と解消.....	310
障害発生時のプライマリ・スタンバイ切り替え.....	311
スタンバイのプライマリへの昇格.....	311
旧プライマリの再組込み.....	312
よくある問題と対策.....	313
プライマリサーバで不要領域が回収されない.....	313
レプリケーションに必要な WAL ファイルが削除された.....	314
プライマリ切り替え後、スタンバイの WAL ファイルが削除されない.....	315
ロジカルレプリケーションの構成・運用例.....	316
ロジカルレプリケーションを使ったテーブル単位レプリケーション.....	316
データを送信するサーバ (パブリッシャ) の構築・設定.....	316
データを受信するサーバ (サブスクライバ) の構築・設定.....	317
PUBLICATION、SUBSCRIPTION の作成.....	318
テーブルデータの初期同期.....	319
ロジカルレプリケーションの開始・停止.....	319
ロジカルレプリケーションの運用・監視.....	320
PUBLICATION の情報.....	320
レプリケーションスロットの情報.....	321
SUBSCRIPTION の情報.....	322
レプリケーションの監視.....	324
ロジカルレプリケーションの性能に関するチューニング.....	326
レプリケーション対象テーブルの追加.....	327

テーブルの再同期.....	327
サブスクライバでの競合の解消.....	328
よくある問題と対策.....	330
同時実行した CREATE SUBSCRIPTION が完了しない.....	330
テーブルの同期処理でエラーが発生する.....	330
レプリケーションの同期遅延が大きい.....	331

第9章

外部データラップを使ってデータベースどうしをつなげる 333

外部データラップの概要.....	334
外部データラップとは.....	334
外部データラップのしくみ.....	335
外部データラップの使い方.....	337
例 1：リモートの PostgreSQL との連携.....	338
①準備.....	338
リモートホスト (PGFDW-REMOTE) 側の準備.....	339
ローカルホスト (PGFDW-LOCAL) 側の準備.....	341
① PostgreSQL 用の外部データラップである postgres_fdw をインストールする.....	342
② 外部サーバを作成する.....	342
③ ユーザーマッピングを作成する.....	343
④ 外部テーブルを作成する.....	344
CREATE FOREIGN TABLE による外部テーブルの作成.....	344
IMPORT FOREIGN SCHEMA による外部テーブルの作成.....	347
例 2：CSV ファイルとの連携.....	349
①準備.....	349
① ファイル用の外部データラップである file_fdw をインストールする.....	351
② 外部サーバを作成する.....	351
③ ユーザーマッピングを作成する.....	352
④ 外部テーブルを作成する.....	352
例 3：Oracle との連携.....	354
①準備.....	354
リモートホスト (ORACLE-REMOTE) 側の準備.....	355
ローカルホスト (ORAFDW-LOCAL) 側の準備.....	355
① Oracle Database 用の外部データラップである oracle_fdw をインストールする.....	357
② 外部サーバを作成する.....	357
③ ユーザーマッピングを作成する.....	358
④ 外部テーブルを作成する.....	359
外部テーブルを更新する際の注意事項.....	361
oracle_fdw の便利な機能.....	361
困ったときには？.....	362
その他の注意事項.....	362
例 4：MySQL との連携.....	362
①準備.....	363
リモートホスト (MYSQL-REMOTE) 側の準備.....	364
ローカルホスト (MYSQLFDW-LOCAL) 側の準備.....	364

① MySQL 用の外部データラッパである mysql_fdw をインストールする.....	367
②外部サーバを作成する.....	367
③ユーザーマッピングを作成する.....	368
④外部テーブルを作成する.....	369
外部テーブルを更新する際の注意事項.....	370
外部サーバでの効率的なクエリ実行.....	371
postgres_fdw のプッシュダウン.....	371
oracle_fdw のプッシュダウン.....	372
mysql_fdw のプッシュダウン.....	372
プッシュダウンの実行例.....	372
WHERE 句のプッシュダウン.....	375
ORDER BY によるソート処理のプッシュダウン.....	376
ソート処理に伴う LIMIT のプッシュダウン.....	378
集約関数のプッシュダウン.....	380
SELECT 時の結合処理のプッシュダウン.....	382
UPDATE、DELETE 時の結合処理のプッシュダウン.....	384
外部データラッパの活用例.....	387
複数データベースにまたがる検索.....	387
テーブルパーティショニングを併用したデータベースシャーディング機能の利用.....	392
シャーディング構成の構築.....	392
更新処理の負荷分散.....	394
外部テーブルの参照処理の並列実行.....	396
パーティションワイス結合.....	397
column 今後のシャーディング開発動向.....	400

第 10 章

PostgreSQL におけるユーザーと権限の管理 401

ロール.....	402
ロールの作成・削除・オプション変更.....	402
ロール属性の確認.....	404
ロール間のメンバーシップの設定と確認.....	405
事前定義 (Predefined) ロール.....	407
① pg_read_all_data.....	407
② pg_write_all_data.....	407
③ pg_read_all_settings.....	408
④ pg_read_all_stats.....	408
⑤ pg_stat_scan_tables.....	408
⑥ pg_monitor.....	409
⑦ pg_database_owner.....	409
⑧ pg_signal_backend.....	409
⑨ pg_read_server_files.....	409
⑩ pg_write_server_files.....	410
⑪ pg_execute_server_program.....	410
⑫ pg_checkpoint.....	410
⑬ pg_maintain.....	411
⑭ pg_use_reserved_connections.....	411
⑮ pg_create_subscription.....	411
権限.....	411

ACL の格納先	412
ACL の書式フォーマット	413
データベースオブジェクト別の権限管理	415
カラムの権限に関する注意点	418
ビューの権限に関する注意点	418
関数の権限に対する注意点	418
行レベルセキュリティ	418
行レベルセキュリティ機能の概要	419
パラメータ row_security の設定	419
行レベルセキュリティの有効化	419
ポリシー定義	420
行レベルセキュリティの設定例	421
作成したポリシーの確認	425
より高度な行レベルセキュリティのポリシーを設定する方法	426
行レベルセキュリティの注意点	426
LEAKPROOF 演算子	426
行レベルセキュリティが適用されたテーブルにアクセスするビュー	428

第 11 章

ログイン認証と通信およびデータの暗号化 431

サーバ接続およびログイン認証の流れ	432
通信暗号化をしない場合のサーバ接続およびログイン認証の詳細	435
スタートアップメッセージの送信	435
PostgreSQL サーバへの接続認証方式の設定	436
接続タイプ	436
接続先データベース	436
接続ユーザー	437
接続元アドレス	437
認証方式、認証オプション	437
pg_hba.conf の例	438
各認証方式の解説	438
Trust 認証	439
パスワード認証	439
GSSAPI 認証	440
SSPI 認証	445
Ident 認証	445
Peer 認証	446
LDAP 認証	446
RADIUS 認証	450
証明書認証	452
PAM 認証	453
BSD 認証	453
ユーザー名マッピング——認証の際の識別名の対応規則定義	453
通信暗号化をする場合のサーバ接続およびログイン認証の詳細	455
暗号化リクエストのパラメータ	455
sslmode	456
gssencmode	457
接続タイプ	457

SSL の使用	458
SSL による通信の暗号化.....	458
基本的な設定（通信暗号化のみを使用）.....	458
クライアント認証.....	460
JDBC ドライバの場合.....	461
サーバ認証.....	462
CRL —— 証明書失効リストの運用.....	464
データ暗号化	465
① OS レベルでの暗号化.....	466
② PostgreSQL での透過的暗号化.....	466
③ カラム単位での透過的暗号化.....	466
④ サーバサイドでの暗号化およびアプリケーションによる暗号化.....	467
pgcrypto.....	467
pgp_sym_encrypt()、pgp_sym_decrypt().....	468
pgp_pub_encrypt()、pgp_pub_decrypt().....	469
使用上の注意点.....	469
監査	470
基本機能を利用した監査.....	470
pgAudit を利用した監査.....	472

第 12 章

バックアップ・リストア 475

バックアップの分類	477
論理バックアップ、物理バックアップ.....	477
論理バックアップの利点.....	477
論理バックアップの欠点.....	477
物理バックアップの利点.....	478
物理バックアップの欠点.....	478
オンラインバックアップ、オフラインバックアップ.....	479
フルバックアップ、差分バックアップ、増分バックアップ.....	479
PostgreSQL における論理バックアップ・リストア	480
SQL の COPY コマンドでのバックアップ・リストア.....	480
pg_dump でのバックアップ・リストア.....	481
SQL スクリプトファイル形式での実行例.....	482
アーカイブファイル形式での実行例.....	483
PostgreSQL における物理バックアップ・リストア	485
pg_basebackup を使った物理バックアップ・リストア.....	485
pg_basebackup でのフルバックアップ・リストア.....	486
pg_basebackup での増分バックアップ・リストア.....	489
バックアップの完全性の確認（バックアップマニフェスト）.....	492
バックアップ中の WAL 取得の必要性.....	493
レプリケーション構成時のスタンバイサーバからのバックアップ.....	495
pg_backup_start と pg_backup_stop による物理オンラインバックアップ・リストア.....	497
pg_backup_start と pg_backup_stop によるバックアップ手順.....	497
オフラインバックアップでの物理バックアップの手順.....	500

OS の機能を使った物理バックアップ手順	500
物理バックアップのポイントインタイムリカバリ (PITR) によるリストア	
——データベースを任意の時点まで復元する	501
ポイントインタイムリカバリ (PITR) とは	502
PITR を利用したリストア手順	503
タイムライン	509
PITR のリカバリターゲット	511
リカバリターゲットに関する他のパラメータ	515
column WAL ファイル名の命名規則	516
column archive_command における留意事項	517
pg_rman による増分バックアップの取得	518
pg_rman のインストール	518
pg_rman によるバックアップ取得手順	518
フルバックアップ	519
増分バックアップ	520
pg_rman によるリストア・リカバリ手順	521
スタンバイからのバックアップ	523
column スタンバイのバックアップに関する注意点	524
バックアップのよくある問題と対策	525
pg_backup_stop が完了しない	525
バックアップ中に WAL が無いというエラーで停止する	526

第 13 章

テーブル、インデックスのメンテナンス 527

Vacuum —— ガベージコレクション (不要領域回収) 機能	528
Vacuum とは	528
ゴミタブルの回収	529
可視性マップ	530
空き領域マップ (Free Space Map)	531
トランザクション ID の周回防止	531
column トランザクション ID (XID) のメンテナンス	532
Vacuum の処理概要	533
VACUUM コマンド	534
パラレル Vacuum	535
Failsafe モード	536
Analyze —— テーブルの統計情報を取得する	536
Analyze とは	536
ANALYZE コマンド	537
Analyze の処理概要	537
自動 Vacuum および自動 Analyze	538
自動 Vacuum および自動 Analyze の起動契機	538
自動 Vacuum および自動 Analyze の停止	539
コストに基づく Vacuum 遅延	539
Vacuum 処理の監視 —— pg_stat_progress_vacuum ビュー	539

Analyze の監視—— pg_stat_progress_analyze ビュー	541
よくある問題と対策	542
不要領域が回収されない	542
Vacuum が長時間化している	543
テーブル、インデックスの再構築	543
VACUUM FULL、CLUSTER、REINDEX	544
pg_repack を利用した「止めない」メンテナンス	545

第 14 章

モニタリング 547

モニタリングとは	548
モニタリングは必須？	549
モニタリングに必要な情報	550
モニタリングに必要な情報の取得元	550
情報の保存期間	551
お勧めの情報取得方法	551
モニタリングに必要な情報の取得元の詳細	552
OS の情報	552
OS と PostgreSQL の状況把握のために取得する項目と取得元	552
取得間隔の設定	553
どのように情報を確認すべきか	553
PostgreSQL のサーバログ	554
サーバログのパラメータ	554
取得間隔の設定	554
どのように情報を確認すべきか	554
PostgreSQL の稼働統計情報	555
PostgreSQL 内部の状況	555
取得間隔の設定	557
どのように情報を確認すべきか	557
稼働統計情報	558
稼働統計情報とは	558
稼働統計情報のビューに関する注意事項	559
所持する権限によって表示される情報量が異なる	559
PostgreSQL のバージョンによって異なる場合がある	561
取得できる稼働統計情報と活用例	561
ロングトランザクションの検知と解消	562
ロングトランザクションの検知と解消の例	563
①ターミナル 1 でロングトランザクションを発生させる	563
②ターミナル 2 でロングトランザクションを検知する	564
③ロングトランザクションの解消方法	565
ロック取得状況の確認とロック待ちの解消状況の確認 (pg_locks の活用例)	566
pg_locks ビュー	566
ロック競合とは	567
ロック取得状況の確認とロック待ちの解消の例	567

①ターミナル 1 で実行するロックを取得し保持するクエリ.....	568
②ターミナル 2 で実行するロック待ちを発生させるクエリ.....	568
③ターミナル 3 で実行するロック取得状況を確認するクエリ.....	568
④ターミナル 3 で実行するクエリ (ロック待ちを解消).....	570
pg_statsinfo を用いたモニタリング	571
pg_statsinfo とは.....	571
取得する情報の一覧、スナップショット.....	572
出力可能なレポートの一覧.....	573
pg_statsinfo のインストールと使用方法.....	575
インストール方法.....	575
postgresql.conf への追記内容 (必要最低限の設定).....	575
テキスト形式のレポート生成方法.....	576
基本的な使用方法.....	578
第 15 章	
パフォーマンスチューニング	581
<hr/>	
パフォーマンスチューニングの流れ.....	582
ボトルネックを見つける.....	583
インデックスを活用する.....	584
インデックスを作ることのメリット・デメリット.....	584
インデックス活用によるチューニング例.....	585
items テーブルに対するシーケンシャルスキャンの改善.....	586
orders テーブルに対するシーケンシャルスキャンの改善.....	586
customers テーブルの customer_id によるソート処理の改善.....	587
インデックスによるチューニングの結果.....	588
設定パラメータを最適化する.....	589
shared_buffers (共有バッファ) を調整する.....	589
work_mem (作業用メモリ) を調整する.....	592
テーブルパーティショニングを使ってクエリを高速化する.....	596
パーティションブルーニングを使う.....	597
パーティションワイズ結合を使う.....	599
パーティションワイズアグリゲーションを使う.....	604
拡張統計情報を活用してプランナの見積り誤差を正す.....	609
プランナの見積り誤差の例.....	609
拡張統計情報.....	610
ヒント句を使って実行計画を操作する.....	613
pg_hint_plan のインストール.....	613
ヒント句の記述方法.....	614
ヒント句を使ったチューニング例.....	615

Appendix A**各パラメータの設定値の考え方**

621

接続と認証に関する設定	622
基本設定.....	622
接続維持・通信障害検出.....	623
クライアントタイムアウト.....	623
ロギング.....	624
SSL 関係.....	624
その他.....	624
検索処理に関するリソース使用（ディスク、メモリ、CPU）	625
共有メモリの設定.....	625
ローカルメモリの設定.....	626
ファイルに関する設定.....	626
バックグラウンドワーカの設定.....	627
クエリ処理の際の I/O 使用量の設定.....	627
テーブルデータのライトバックの設定.....	627
データ圧縮.....	628
WAL —— トランザクションログ	628
WAL の基本的な設定.....	628
WAL 出力効率に関する設定.....	629
チェックポイントに関する設定.....	629
リカバリに関する設定.....	630
特殊な設定.....	630
WAL アーカイブ	631
基本的な設定.....	631
アーカイブライブラリ.....	631
レプリケーション	632
WAL の基本的な設定.....	632
スタイバイ遅れの際の接続維持に関する設定（WAL の保存）.....	632
接続タイムアウト.....	632
ステータス更新.....	633
衝突解決.....	633
ロギング.....	633
論理レプリケーション.....	633
サーバログ出力	634
ログ出力先の設定.....	634
ログ出力ファイルの設定.....	634
syslog 出力の設定.....	634
ログ行のフォーマット.....	635
事象ごとのログ出力設定.....	635
ログ出力の間引き設定.....	635
実行統計情報.....	636

コアダンプ.....	636
Vacuum、自動 Vacuum	636
基本的な設定.....	636
実行間隔の調整.....	637
トランザクション ID (XID) 周回処理の調整.....	637
使用メモリ量の調整.....	638
共有バッファ使用量の調整.....	638
I/O 使用量の調整.....	638
並行して Vacuum するテーブル数.....	639
単一のテーブル内の並列処理.....	639
ページのプリフェッチの調整.....	639
稼働統計情報、Analyze 処理	640
実行間隔の調整.....	640
稼働統計情報ビューの一貫性の調整.....	640
SQL 実行計画	640
プラン利用の可否変更.....	640
並列実行の調整.....	640
Prepared 文の実行計画.....	641
JIT コンパイルの設定.....	641
データ統計の粒度調整.....	641
パーティションワイズ結合・集約.....	642
結合探索の抑制.....	642
インデックススキャンの優先度.....	642
ハッシュ結合の優先度.....	643
検索応答の速さの優先度.....	643
制約による除外の設定.....	643
再帰問い合わせの結果サイズ想定.....	643
基本コスト値.....	644
遺伝的計画.....	644
設定項目のバージョン間の差分	644
16 → 17.....	645
15 → 16.....	647
14 → 15.....	648
13 → 14.....	649
12 → 13.....	651
11 → 12.....	652
10 → 11.....	654
9.6 → 10.....	655

Appendix B**PostgreSQL 18 の新機能と設定項目の差分** 657

PostgreSQL 18 の新機能.....	658
非同期 I/O.....	658
メジャーバージョンアップ時の統計情報の引き継ぎ.....	658
複数列 B-tree インデックスにおけるスキップスキャンの追加.....	659
UUIDv7 対応.....	659
仮想生成列.....	659
OAuth 2.0 認可／認証.....	660
RETURNING 句での変更前の値 (OLD) と変更後の値 (NEW) の指定.....	660
時間制約.....	660
設定項目のバージョン間の差分 (17 → 18).....	660
索引.....	663
著者紹介.....	693

第 1 章

PostgreSQL の概要

データベースは情報システムを構成する要素の一つで、さまざまな場面で重要な役割を果たしています。本章では、リレーショナルデータベースが持つ性質や、PostgreSQL のアーキテクチャ・特徴について見ていきましょう。

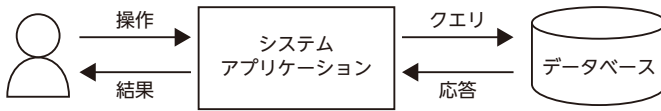
リレーショナルデータベースとは

データベースとは

近年、ネットワーク技術の進歩などにより、日々の活動で収集されるデータの数は膨大なものになりました。そうしたデータを効率的に活用することが求められ、データベースはそれを実現するうえで非常に重要な役割を果たしています。データベースとは、このように蓄積され効率的に検索・活用できるようにされたデータの集合のことを言います。データベースをコンピュータ上で管理するソフトウェアのことを**データベース管理システム (DataBase Management System、DBMS)** と呼びます。単に「データベース」と言った場合、この DBMS のことをしばしば指します。本書でも、DBMS のことを単に「データベース」と称することがあります。

例として、大学における成績管理システムを考えてみましょう。このシステムを利用するのは、教員と学生です。教員は、試験を行うとその結果をシステムに登録します。一方、学生は自分の成績をシステム上で確認するでしょう。システムにデータベースを接続することでこれらの操作を実現できます。**図 1.1** は、成績管理システムの簡単な構成を表したものです。エンドユーザー（教員や学生）がアクセスするのはあくまでも成績管理システム（図中のシステムアプリケーション）の画面であり、データベースを直接操作することはありません。データの取得や更新などは、英語で問い合わせを意味する「クエリ」としてデータベースに対して送信されます。この例では、データベースはミドルウェアとして使用されています。そのため、エンドユーザーはデータベースを使っているということすら意識しないかもしれません。

図 1.1 データベースとアプリケーション



データモデル

データベースでは、現実世界のデータを特定の規則にしたがって整理し、大量のデータから所望するデータを効率的に検索できるようになっています。このとき、どのような規則にしたがってデータをモデル化するかを**データモデル**と呼びます。データモデルのうち、現実世界を抽象化したものを概念データモデル、さらにデータベースとして扱いやすい構造へと詳細化したものを論理データモデルと呼びます。論理データモデルには次のようなものが存在します。

- 階層型データモデル
- ネットワークモデル
- オブジェクト指向モデル
- リレーショナルモデル（関係モデル）

階層型データモデルでは、データを木構造で表現します。このモデルでは、会社などの組織図のような構造を採用します。あるデータは、複数のデータとの間に親子関係を持ちます。データベースの歴史の中でも初期に登場したものです。

ネットワークモデルはその名のとおり網の目のようなネットワークで、データの関係性が表現されるデータモデルです。階層型データモデルでは1つのデータに対して親ノードは1つしか存在しませんが、ネットワークモデルでは複数の親ノードを持つことができます。これによって、データの冗長性を排除できます。

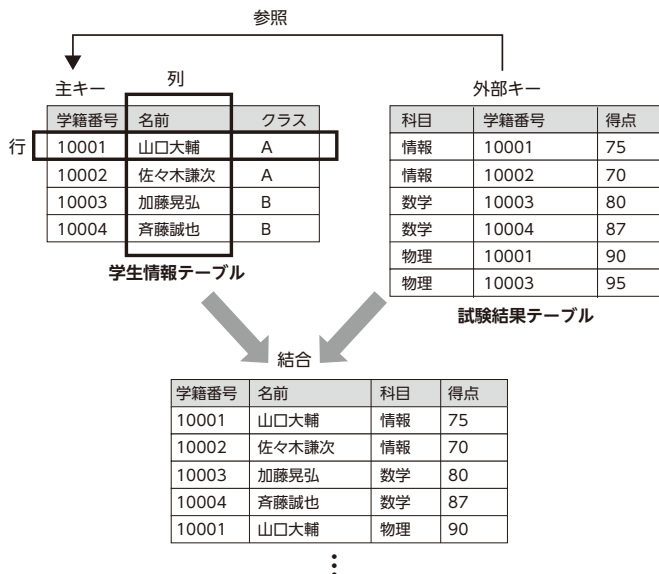
オブジェクト指向モデルは、オブジェクト指向プログラミングを取り入れたデータモデルです。多くのオブジェクト指向プログラミング言語と同様に、データとそれに対する処理をオブジェクトとしてまとめて処理します。この

モデルを採用したデータベースは、オブジェクト指向プログラミング言語と親和性が高くなるよう設計されています。

本書で取り扱う PostgreSQL は、**リレーショナルモデル（関係モデル）**に従ったデータベースです。リレーショナルモデルに基づく DBMS のことを **リレーショナルデータベース管理システム（RDBMS：Relational Database Management System）** と言います。

リレーショナルモデルでは、データをリレーション（関係）で表します。リレーションは、行と列から成る 2 次元のテーブル（表）としてしばしば説明されます。行は「レコード」や「タプル」、列は「カラム」とも呼ばれます。ここでは、先ほどの成績管理システムについて、**図 1.2** に示す 2 つのテーブルを例にとり考えてみましょう。この例は、大学における学生情報を記録したテーブル（学生情報テーブル）と試験の結果を記録したテーブル（試験結果テーブル）から成ります。学生情報テーブルには、列として学籍番号と名前とクラスが存在し、各行は学生一人一人を表しています。テーブルに存在する列やそこに入る値の型（スキーマ）は、テーブルを作成する前に定義されなければなりません。

図 1.2 リレーショナルモデルにおけるテーブルの例



学生情報テーブルでは、学籍番号によって行を一意に識別できます。このような性質を持った列のことを**主キー（プライマリーキー）**と呼びます。同姓同名の学生が存在する可能性を考慮すれば、氏名の列は主キーとして用いることはできません。

一方、試験結果テーブルにも学籍番号の列がありますが、これは学生情報テーブルの各学生を参照するもので、**外部キー**と呼ばれます。図 1.2 の例では、試験結果テーブルには学生の名前は含まれていません。名前と得点を一度に取得するために、両テーブルの主キーと外部キーを結び付けて表を結合 (*Join*) できます。その結果は図 1.2 の下側に示したとおりになります。

このようにリレーショナルデータベースでは、複数のテーブルを関連付けることで高度な検索を実現しています。

リレーショナルデータベースの役割

リレーショナルデータベースは、次のような役割を担います。

- **データを一元的に管理**
アプリケーションやシステムごとにデータを保存する代わりに、データベースで一元的に管理することで、データ管理の効率性を上げることができる。データの物理的格納方法やインデックスの管理などをアプリケーションから切り離すことができる。また、複数のアプリケーションからデータを活用することが可能になる
- **データを検索・追加・削除・更新するための機能を提供**
検索は、データベースが担う役割の中でも最も重要なもの。さらに、データの追加・削除・更新を扱える必要がある。データベースは、このような機能を SQL により簡単かつ高速に提供する
- **データの妥当性や整合性のチェック**
先ほどの成績管理システムの例では、存在しない学生の成績データを追加すると問題が生じる。また、学籍番号は重複してはいけない。テーブルにこのような制約が存在する場合、その制約にしたがったデータのみを受け付け、矛盾のない状態を担保する必要がある。リレーショナルデータベースによって、このような制約が実現される
- **トランザクション処理**
データベースに対する一連の操作が中途半端な形で反映されたり、矛盾が起こったりしないようにする（詳細は後述）

- **同時実行制御**

データベースに対して、複数のユーザーが同時にデータを追加したり更新したりすることがある。このとき、データの整合性を維持しながら、追加・更新リクエストに対応する必要がある。すなわち、複数ユーザーが同時にデータを操作しても、互いに干渉せず独立して処理できるようにする

- **障害回復**

データベースに何らかの障害が発生してデータが消失するようなことがあれば、それを利用するシステムは動作を継続できない。データベースに障害が発生しても障害が発生する前の状態に回復し、データを失わないようにする（詳細は後述）

- **セキュリティ**

データベースには、ユーザーの個人情報など機密性の高い情報が多数格納されることが多くある。それらのデータに対して、想定しないユーザーからアクセスされれば、情報漏洩につながる。そのため、データにアクセスできる権限を指定し、機密性の高いデータが権限のないユーザーによってアクセスされないようにする。データベースによっては、データの暗号化機能を提供しているものも存在する

トランザクション管理

RDBMS は、データの検索・追加・更新・削除の機能を提供します。一般に、アプリケーションはこれらの操作を一つだけ行うのではなく、複数の操作をまとめて実行します。トランザクションとは、データベースに対するこうした一連の操作を一つにまとめたもののことです。トランザクションは、RDBMS において最も重要な要素の一つです。

トランザクションが満たすべき性質として、ACID 特性が存在します。ACID とは、次に示す 4 つの性質の頭文字を取ったものです。

- **原子性 (Atomicity)**

トランザクション内の一連の操作が「すべて成功する」か「すべて失敗する」かの二者択一であり、中途半端に反映されないことを指します。

銀行口座におけるトランザクションを例に見ていきましょう。ここでは、口座 A と口座 B の 2 つの口座が存在するものとします。最初、2 つの口座にはともに 1,000 円が入金されています。ここで「口座 A から口座 B に 200 円送金する」というトランザクションを考えます。このトランザクションは、次のような一連の操作で構成されるでしょう。

- ❶口座 A の残高を取得する。結果は 1,000 円となる
- ❷口座 B の残高を取得する。結果は 1,000 円となる
- ❸口座 A の残高を 800 円に更新する
- ❹口座 B の残高を 1,200 円に更新する

原子性が満たされる場合、このトランザクションがすべて成功する（口座 A の残金が 800 円、口座 B の残金が 1,200 円）か、すべて失敗する（口座の残金はともに 1,000 円である）かのどちらかにしかありません。たとえば、❸までの操作だけが反映されてしまい、口座 A が 800 円、口座 B が 1,000 円という中途半端な状態にならないことを指します（この場合、口座 A の 200 円が失われています）。

トランザクションは、その一連の操作がすべて完了すると、処理を確定させるために最後にコミット（Commit）を行います。一方、途中で何らかのトラブルが発生した場合は、ロールバック（Rollback）することで操作をすべて元に戻し、トランザクション実行前の状態に復帰して、トランザクションはアボート^{注1}されます。

●一貫性 (Consistency)

データベースでは、整合性を担保するために制約を課することができます。トランザクションの前後で整合性が矛盾しないことを一貫性と言います。銀行口座のトランザクションの例で見ると、口座の残高がマイナスにはならないように制約を課されていた場合、実行後に残高がマイナスになるようなトランザクションは失敗することとなります。

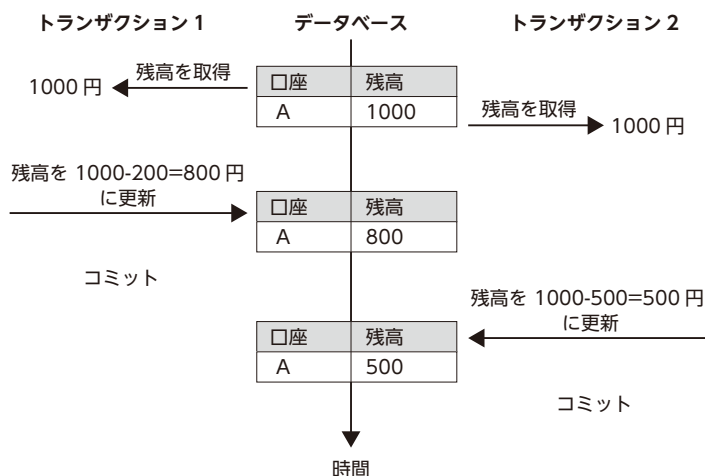
●分離性 (Isolation)

トランザクションが同時に複数のユーザーから実行されているときに、各トランザクションの途中の状態がほかのトランザクションからは見えず、直列で実行された場合と同じ状態になる性質のことを分離性と呼びます。

注1 トランザクションがアボートされると、そのトランザクションが一切実行されなかったかのように、すべての処理が取り消されます。

ここでも銀行口座の例を用いて考えてみましょう。今回存在する銀行口座は 1 つのみで、2 人が同時に引き出しを行うものとします。トランザクション 1 は 200 円、トランザクション 2 は 500 円を引き出します。このトランザクションが正しく実行されない様子を図 1.3 に示します。図 1.3 は、2 つのトランザクションを並行して実行したときのスケジュールの一例で、実際には、これ以外にもさまざまなものが考えられます。

図 1.3 ロストアップデートあり

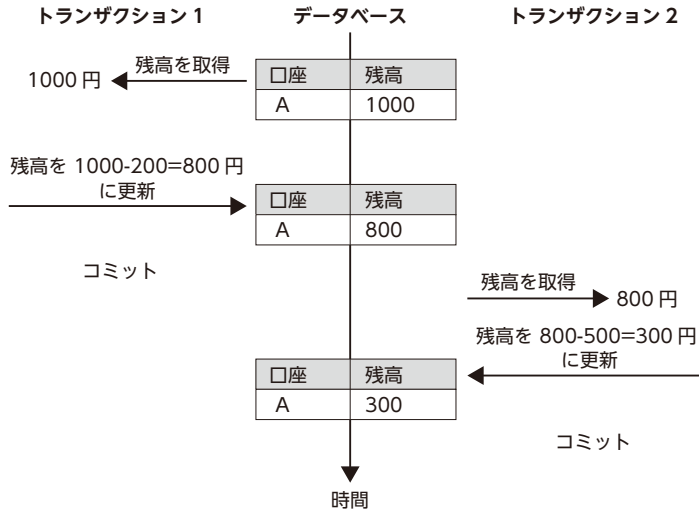


2 つのトランザクションは、口座の残高を取得したうえで、自分が引き出したい額を引いた値をデータベースに書き込みます。図 1.3 のような順序でこれらの操作が実行されたとすると、口座の残高は 500 円となり、トランザクション 1 の更新が失われています。このようにあるトランザクションの書き込みが別のトランザクションによって上書きされることにより生じる異常のことをロストアップデートと呼びます。

分離性は、このような矛盾した操作が発生しないことを指します。すなわち、トランザクションが直列で実行されたと仮定したときに得られる結果と同じ状態になることを意味します。先ほどの例では図 1.4 のとおり、トランザクション 1 のコミット後の残高 800 円から 500 円を引き出し、最終的な口座の残高は 300 円となるべきです。しかし、この制約を完全に課

すとパフォーマンス上の問題となるため、SQL 標準では 4 つの分離レベル (*Isolation level*) を定めています。詳細は後述します。

図 1.4 ロストアップデートなし



● 耐久性 (Durability)

トランザクションがコミットされ成功したという通知を受けると、そのデータは永続化され障害によっても失われない性質のことを耐久性と呼びます。詳細は「障害回復」の項目で後述します。

トランザクション分離レベル

トランザクションの分離性を完全に実現するとパフォーマンス上の問題が生じるため、SQL 標準では 4 つの分離レベル (*Isolation level*) を定めています。最も分離性が低いのが Read uncommitted、最も高いのが Serializable です。Serializable 以外のレベルでは、ファントムリードなどいくつかの異常 (*Anomaly*、後述) が発生することが許容されます。

● Serializable

- すべてのトランザクションが直列で実行されたときと同じ状態になることを保証される

- **Repeatable read**
 - ・ 自分のトランザクションで読み込んだデータをもう一度読み込んでも、同じ値になることが保証される
 - ・ ファントムリードの発生が許容される
- **Read committed**
 - ・ 別のトランザクションによる変更は、それがコミットされるまで見えないことが保証される
 - ・ ファントムリード、ノンリピータブルリードの発生が許容される
- **Read uncommitted**
 - ・ 別のトランザクションによる変更のうち、コミットされていない変更が見える可能性がある
 - ・ ファントムリード、ノンリピータブルリード、ダーティーリードの発生が許容される

トランザクション分離レベルにより発生する異常

ここからは、それぞれの分離レベルで発生が許容されている異常について、重大さの順で説明します。

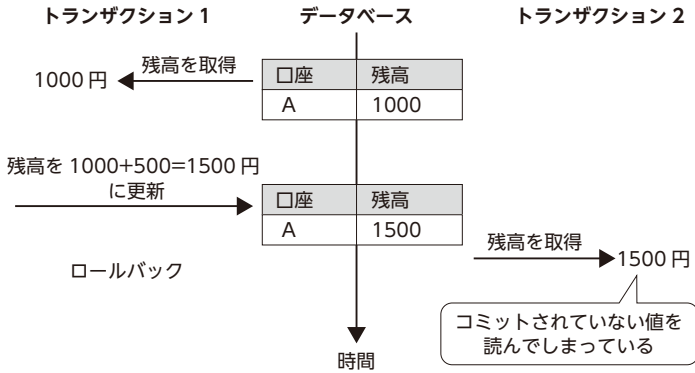
● ダーティーリード

ダーティーリードは、コミットされていないトランザクションの変更を読み取ってしまう異常のことです。

銀行口座の例を使って考えてみましょう。トランザクション 1 は口座に 500 円入金しようとしませんが、途中で何らかの問題が生じロールバック（処理をすべて取り消してトランザクション開始前の状態に戻すこと。前出の「原子性」の説明を参照）したとします。一方、トランザクション 2 は、口座の残高を取得しようとしています。

このトランザクションのスケジュールの一例として、**図 1.5** のようなものが考えられます。この場合、トランザクション 2 は、トランザクション 1 が進行中に口座の残高を参照しようとしています。このとき、トランザクション 1 が更新したもののコミットされていない 1,500 円という値を読み込んでしまいます。この異常をダーティーリードと呼びます。

図 1.5 ダーティリード

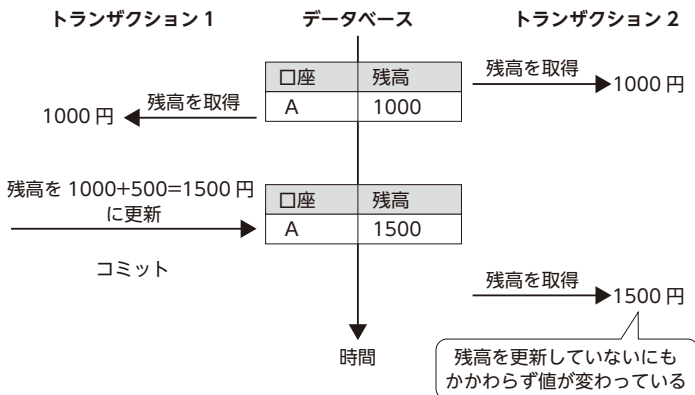


● ノンリピータブルリード

ノンリピータブルリードは、トランザクションが2回同じデータを読み取ったとき、その結果が異なってしまう異常のことです。

ダーティリードのときと同じ銀行口座の例で考えます。トランザクション1は口座に500円入金しようとしています。その前後にトランザクション2は、口座の残高を2回取得します。スケジュールの一例は、図 1.6 のようになります。トランザクション2は、口座Aの残高を一切操作していないにもかかわらず、2回残高を取得すると異なる値を取得してしまいます。このような異常をノンリピータブルリードと呼びます。

図 1.6 ノンリピータブルリード

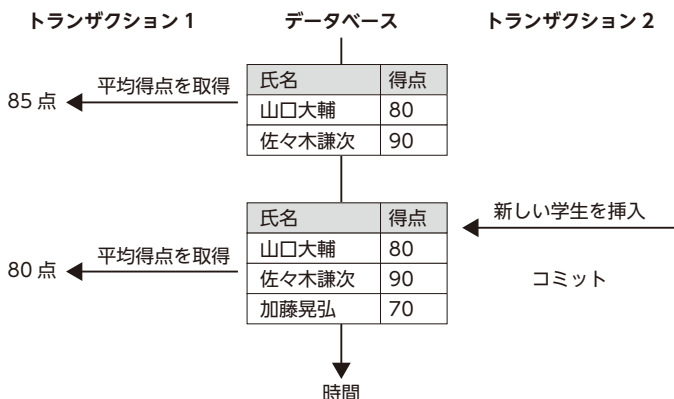


● ファントムリード

ファントムリードは、トランザクションがある条件に合致するデータの一覧を 2 度取得したとき、1 回目には出現しなかったデータを読み取ってしまう異常のことです。

ここでは、本節冒頭に出てきた成績管理システムを例に取り考えてみましょう。説明を簡単にするため、テーブル構造を簡略化し、テーブルは学生の氏名と得点から成るものとします。トランザクション 2 は、新しい学生に関するデータを挿入します。トランザクション 1 は、学生全員の得点の平均値を 2 回計算します。スケジュールの一例は、**図 1.7** のようになります。トランザクション 1 が得点の平均値を 2 回取得しようとする時、その値は異なるものになっています。これは、1 回目にはなかったデータを 2 回目で読み込んでしまうために起きる異常で、これをファントムリードと呼びます。

図 1.7 ファントムリード



● まとめ

分離レベルと発生する異常を**表 1.1**にまとめます。

PostgreSQL には、Read Uncommitted は存在しません。PostgreSQL では、Read Uncommitted は Read Committed として扱われるためです。また、PostgreSQL における Repeatable Read ではファントムリードは発生しませんが、上には示していない異常が発生する可能性があるため Serializable とは区別されます。

表 1.1 分離レベルと発生する異常

分離レベル	ダーティーリード	ノンリピータブル リード	ファントムリード
Read Uncommitted	可能性あり*	可能性あり	可能性あり
Read Committed	なし	可能性あり	可能性あり
Repeatable Read	なし	なし	可能性あり*
Serializable	なし	なし	なし

* PostgreSQL では発生しない

障害回復——チェックポイントと WAL

DBMS が提供する重要な機能の一つとして、障害回復が挙げられます。トランザクションはその性質として耐久性を持つ必要があるため、データベースサーバや OS に障害が発生してもデータを失ってはいけません。データベースは、さまざまな技術を用いてパフォーマンスを維持しながら、障害回復を実現しています。

トランザクションがコミットされるたびにその結果をディスクに同期的に書き込めば、永続性が担保されます。しかし、そのような処理は非常に重いため、パフォーマンス上の問題が存在します。PostgreSQL をはじめとする RDBMS では、キャッシュバッファとログ先行書き込みを活用することで、この問題に対処しています。

RDBMS は、テーブルの一部をキャッシュバッファ（PostgreSQL では共有バッファと呼びます）上にキャッシュしています。これにより、ディスク I/O を抑えて高速化しています。データベースに対する変更は、このキャッシュバッファ上に対して行われます。変更されたもののディスクにそれが反映されていないページのことをダーティーページと呼び、すべてのダーティーページをディスクに書き出すことを**チェックポイント**と呼びます。

まだチェックポイントにより書き出されておらずメモリ上（キャッシュバッファ上）にしか存在しないデータは、RDBMS がクラッシュすると失われてしまいます。そのため、RDBMS は、データベースに対して行われた操作の履歴を、実際に操作を行う前にログファイルに逐次記録することで永続性を担保しています。これを**ログ先行書き込み（WAL：Write Ahead Logging）**と呼びます。ログファイルは、いったんメモリ上の WAL バッファ

に書き込まれ、WAL バッファが一定の量に達したときおよびコミットするときにディスクに書き出されます。書き込みはディスクに対してシーケンシャルに行われるため、ランダムに書き込む場合に比べて性能低下を抑えることができます。障害が発生してデータベースがクラッシュすると、チェックポイント後のデータに対して WAL を用いて再実行することで、障害回復がなされます（ロールフォワード）。

SQL の解釈・実行

ここでは、リレーショナルデータベースが提供する大きな特徴である SQL について説明します。SQL は、RDBMS を操作するために用意されている言語です。SQL では、ユーザーがどのような操作を行いたいかをシンプルな英文の形で記述します。たとえば、先ほど例に挙げた学生情報テーブルと試験結果テーブルを作成する SQL 文は次のようになります。

```
CREATE TABLE students(id INT PRIMARY KEY, name TEXT);
CREATE TABLE scores(course TEXT, student_id INT, score INT, FOREIGN KEY(student_id) REFERENCES students(id));
```

この例では、`students` テーブルと `scores` テーブルについて、スキーマを指定しながら作成していることがわかります。

このテーブルに対してデータを挿入してみましょう。データの挿入は、`INSERT` 文によって行われます。

```
INSERT INTO students VALUES (10001, 'AA BB'), (10002, 'CC DD');
INSERT INTO scores VALUES ('数学', 10001, 70), ('英語', 10002, 80);
```

続いて、先ほどの「生徒の名前と得点を取得する」という結合演算の検索を SQL で実行してみます。SQL で検索を行う際は、`SELECT` 文が用いられます。結合演算は次のような SQL になります。

```
SELECT students.name AS name, scores.score AS score
FROM students JOIN scores ON students.id = scores.student_id;
```

このほか、データの更新は `UPDATE` 文、削除は `DELETE` 文で行います。

このように、検索のために必要な詳細なアルゴリズムを知らずとも、ユーザーは高度な検索処理を簡単に実現できます^{注2}。SQL は標準化されており、異なるリレーショナルデータベースで共通に利用できます^{注3}。SQL の詳細は第4章で解説します。

PostgreSQL とは

PostgreSQL は、オープンソースの RDBMS です。日本では、PostgreSQL は「ポストグレスエスキューレル」「ポストグレス」、または「ポスグレ」などと発音されます。

PostgreSQL は、後述するように開かれたコミュニティ主導で開発が進められるオープンソースのデータベースです。また、高い拡張性や SQL 標準への準拠といった特徴を持つほか、Linux/macOS/Windows で動作するマルチプラットフォームなプログラムです。

PostgreSQL の歴史

PostgreSQL の歴史は、1986 年にカリフォルニア大学バークレイ校で研究用に開発された「Postgres」までさかのぼります。Postgres は、ユーザー定義型などの現在の PostgreSQL に通じる拡張性をすでに兼ね備えていました。その後、コードのメンテナンスとサポートの負荷増大により開発はいったん終了しましたが、カリフォルニア大学バークレイ校の学生であった Andrew Yu と Jolly Chen によって、問い合わせ言語に SQL を用いるなどの改良が施され、Postgres95 としてリリースされました。その後、SQL を処理する能力を持つことを込めて PostgreSQL と名前を変え、現在に至ります。

注2 もっとも、効率的にクエリを実行するにはアルゴリズムに関する知識は欠かせません。詳細は第7章で説明します。

注3 ただし、SQL には方言があるため、完全な互換性はないことに注意してください。

PostgreSQL 開発コミュニティ

PostgreSQL は、特定の企業に依存せずインターネット上のコミュニティ主導でオープンな形で開発が進められています。PostgreSQL の開発者によるメーリングリストやオフラインでの会議を通じて新機能やバグ修正などの議論が進められます。PostgreSQL の最新動向を知りたい方は、メーリングリストに参加するとよいでしょう。PostgreSQL には複数のメーリングリストがありますが、pgsql-hackers^{注4}では、日々新機能などが提案・議論されています。

PostgreSQL 開発の特徴として、コミットフェストというオンラインイベントが挙げられます。PostgreSQL の開発では、メーリングリストにパッチを投稿し、それを誰かがレビューして、コミッターがリポジトリにコミットすることでパッチがコードに取り入れられます。しかし、大量のメールが行き交うメーリングリストでは、パッチは埋もれがちです。コミットフェストでは、投稿されたパッチのレビューを集中的に行い、パッチを適用してコミットしていきます。コミットフェストを通じて開発者は自分が作成したパッチを登録するとともに、ほかの人が投稿したパッチのレビューを行います。これにより、パッチが埋もれてしまうのを防ぎつつ、パッチの品質を高めて PostgreSQL 本体に反映しています。コミットフェストは定期的開催されます。このようなサイクルによって、PostgreSQL は日々進化を続けています。

リリーススケジュール

PostgreSQL は、近年では 1 年に 1 回のメジャーバージョンアップと数回のマイナーバージョンアップが行われます。

バージョン番号は、9.6 以前と 10 以降で異なります。9.6 以前では「x.y.z」という形でバージョンが表され、「x.y」の部分がメジャーバージョン、「z」の部分がマイナーバージョンを示します。10 以降では、メジャーバージョンは整数となり、「x.y」の形で「x」がメジャーバージョン、「y」がマイナー

注4 <https://www.postgresql.org/list/pgsql-hackers/>

バージョンを表すようになりました。2026年1月時点での最新のメジャーバージョンは、PostgreSQL 18です。ただし執筆時点でのメジャーバージョンは PostgreSQL 17であったため、本書ではこのバージョンを前提に解説しています。

開発リポジトリ

PostgreSQL は、公式の Git リポジトリ^{注5} でそのソースコードが管理されています。ここから最新バージョンを取得することで、まだリリースされていない機能を試すことができます。ソースコードを読むことで PostgreSQL への理解を深めることができます。もし、バグと思われる挙動などがあれば、メーリングリストに報告するとよいでしょう。

メジャーバージョンごとの新機能

表 1.2 に、各メジャーバージョンで追加された新機能の一覧を挙げます。近年は、パーティショニングやパラレルクエリなど大規模データを効率的に管理するための機能強化が図られています。

表 1.2 各メジャーバージョンで追加された新機能の一覧

バージョン	リリース	主な機能強化
9.1	2011年9月	レプリケーション、外部テーブル
9.2	2012年9月	Index only scan、カスケードレプリケーション
9.3	2013年9月	マテリアライズドビュー、postgres_fdw
9.4	2014年12月	バイナリ JSON 型
9.5	2016年1月	UPSERT 機能、BRIN インデックス
9.6	2016年9月	パラレルクエリ
10	2017年10月	宣言的パーティショニング、ロジカルレプリケーション
11	2018年10月	JIT (<i>Just In Time</i>) コンパイル、パーティショニング機能改善
12	2019年10月	JSON Path、CTE のインライン化
13	2020年9月	並列 VACUUM、B-tree インデックスの改善、インクリメンタルソート

注5 <https://git.postgresql.org/gitweb/?p=postgresql.git>

14	2021 年 9 月	postgres_fdw の拡張（複数の外部データリソースの同時並列参照など）
15	2022 年 10 月	MERGE 文のサポート
16	2023 年 9 月	JOIN の並列処理強化、スタンバイからのロジカルレプリケーション
17	2024 年 9 月	VACUUM の省メモリ化、JSON 関連機能強化
18	2025 年 9 月	非同期 I/O、バージョンアップグレード高速化

PostgreSQL のアーキテクチャ概要

プロセス構成

PostgreSQL は、マルチプロセスタイプのアーキテクチャです。PostgreSQL を起動するときさまざまなプロセスが生成されます。まずは、それを一覧表示してみましょう。

```
$ ps ux
USER          PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
postgres    66661  0.0  3.5 428528 27776 ?        Ss   05:15   0:00 /usr/pgsql-17/
bin/postgres -D /var/lib/pgsql/17/data/
postgres    66662  0.0  0.8 282152  6572 ?        Ss   05:15   0:00 postgres:
logger
postgres    66663  0.0  0.8 428664  6580 ?        Ss   05:15   0:00 postgres:
checkpointer
postgres    66664  0.0  0.9 428680  7860 ?        Ss   05:15   0:00 postgres:
background writer
postgres    66666  0.0  1.3 428528 10676 ?        Ss   05:15   0:00 postgres:
walwriter
postgres    66667  0.0  1.2 430100  9908 ?        Ss   05:15   0:00 postgres:
autovacuum launcher
postgres    66668  0.0  1.1 430108  9140 ?        Ss   05:15   0:00 postgres:
logical replication launcher
postgres    66709  0.6  0.6 224092  5504 pts/0    S   05:17   0:00 -bash
postgres    66766  0.0  0.4 225480  3456 pts/0    R+  05:17   0:00 ps ux
```

ここに載っていないものも含めて、代表的なものを次に挙げます。

- **postmaster**
PostgreSQL を起動すると最初に立ち上がるプロセス
- **postgres**
クライアントがデータベースサーバに接続すると生成されるプロセス。クライアントからのクエリを受け付けて処理する。このように PostgreSQL では、接続ごとにプロセスが生成される
- **parallel worker**
PostgreSQL では、クエリを複数のプロセスで並列して処理できる。これをパラレルクエリと呼ぶ。parallel worker は、パラレルクエリが実行されている間に起動するプロセス
- **background writer**
共有バッファ上のダーティーページを定期的にディスクに書き出すプロセス。チェックポイント処理では、すべてのダーティーページを書き出すため負荷が高くなる。background writer によってチェックポイント時に書き出すべきデータ量を削減することで、この負荷を軽減できる
- **checkpointer**
チェックポイント処理を自動で行うプロセス
- **walwriter**
WAL バッファに書き込まれた WAL をディスクに書き出すプロセス
- **autovacuum launcher、autovacuum worker**
autovacuum launcher は、autovacuum worker を起動するプロセス。autovacuum worker は、テーブルやインデックスのバキューム (Vacuum) 処理を行う。バキューム処理では、データの更新や削除によって生じたテーブルやインデックスの未使用領域を回収し再利用できるようにする。また、同時にアナライズ (Analyze) 処理も行われる。アナライズでは、テーブルの統計情報を更新する

ファイル、ディレクトリ構成

PostgreSQL で `initdb` コマンドによりデータベースを初期化すると、データベースクラスタ配下にさまざまなファイルやディレクトリが生成されます。以下は、初期化直後のデータベースクラスタのディレクトリの様子を表示したものです。

```
$ ls -la
total 72
drwx-----, 20 postgres postgres 4096 Nov 11 05:15 .
drwx-----,  4 postgres postgres  51 Nov 11 05:14 ..
```

```
drwx-----, 5 postgres postgres 33 Nov 11 05:14 base
-rw-----, 1 postgres postgres 30 Nov 11 05:15 current_logfiles
drwx-----, 2 postgres postgres 4096 Nov 11 05:16 global
drwx-----, 2 postgres postgres 32 Nov 11 05:15 log
drwx-----, 2 postgres postgres 6 Nov 11 05:14 pg_commit_ts
drwx-----, 2 postgres postgres 6 Nov 11 05:14 pg_dynshmem
-rw-----, 1 postgres postgres 5499 Nov 11 05:14 pg_hba.conf
-rw-----, 1 postgres postgres 2640 Nov 11 05:14 pg_ident.conf
drwx-----, 4 postgres postgres 68 Nov 11 05:14 pg_logical
drwx-----, 4 postgres postgres 36 Nov 11 05:14 pg_multixact
drwx-----, 2 postgres postgres 6 Nov 11 05:14 pg_notify
drwx-----, 2 postgres postgres 6 Nov 11 05:14 pg_repslot
drwx-----, 2 postgres postgres 6 Nov 11 05:14 pg_serial
drwx-----, 2 postgres postgres 6 Nov 11 05:14 pg_snapshots
drwx-----, 2 postgres postgres 6 Nov 11 05:15 pg_stat
drwx-----, 2 postgres postgres 6 Nov 11 05:14 pg_stat_tmp
drwx-----, 2 postgres postgres 18 Nov 11 05:14 pg_subtrans
drwx-----, 2 postgres postgres 6 Nov 11 05:14 pg_tblspc
drwx-----, 2 postgres postgres 6 Nov 11 05:14 pg_twophase
-rw-----, 1 postgres postgres 3 Nov 11 05:14 PG_VERSION
drwx-----, 4 postgres postgres 77 Nov 11 05:14 pg_wal
drwx-----, 2 postgres postgres 18 Nov 11 05:14 pg_xact
-rw-----, 1 postgres postgres 88 Nov 11 05:14 postgresql.auto.conf
-rw-----, 1 postgres postgres 30694 Nov 11 05:14 postgresql.conf
-rw-----, 1 postgres postgres 58 Nov 11 05:15 postmaster.opts
-rw-----, 1 postgres postgres 100 Nov 11 05:15 postmaster.pid
```

● ディレクトリ

PostgreSQL のデータベースを構成する主なディレクトリには以下のよう
なものがあります。

- **base**
データベースごとのサブディレクトリを保有するディレクトリ。データベースごと
に識別子（数字）から成るディレクトリが生成され、その中にファイルが配置され
る
- **global**
データベースクラスタで共有するテーブルを保有するディレクトリ。pg_database
などのシステムカタログが含まれる
- **pg_wal**
WAL ファイルを格納するディレクトリ
- **pg_xact**

トランザクションのコミット状態のデータを保有するディレクトリ

● ファイル

主要なファイルは以下のとおりです。

● PG_VERSION

データベースクラスタを初期化した PostgreSQL のメジャーバージョンが記載されたファイル。PostgreSQL は、同じメジャーバージョンであればデータベースクラスタに互換性があるものの、異なるメジャーバージョン間では互換性がない。初期化したときのバージョンと現在実行しているバージョンが違うときに誤って起動してしまうのを防ぐため、このようなファイルが存在する

● postmaster.pid

postmaster プロセスのプロセス番号が記載されたファイル。このファイルは PostgreSQL が起動しているときに生成され、二重起動を抑制している

● postgresql.conf

PostgreSQL に関する設定が記載されたファイル

● テーブルファイル

テーブルデータの実体が格納されているファイルで 8,192 バイトごとのページで構成される。ファイル名はテーブルのファイルノード番号。主に base ディレクトリのデータベースごとのサブディレクトリ内に配置される

● インデックスファイル

インデックスが格納されているファイルでテーブルファイルと同様、8,192 バイトごとのページで構成される。ファイル名はインデックスのファイルノード番号。主に base ディレクトリのデータベースごとのサブディレクトリ内に配置される

● TOAST (過 大 属 性 格 納 技 法 : The Oversized-Attribute Storage Technique) テーブルファイル

テーブルファイルやインデックスファイルは、8,192 バイトごとのページで構成されていた。PostgreSQL では、ページをまたがる行が存在することを許可していない。そのため、行の中のサイズの大きな列を格納する別のテーブルとして TOAST テーブルが作成される。この TOAST テーブルデータの実態が格納されているテーブルファイル。ファイル名は TOAST テーブルのファイルノード番号。主に base ディレクトリのデータベースごとのサブディレクトリ内に配置される

● 可視性マップ (Visibility Map) ファイル

Vacuum 処理によって回収されるべき不要領域がどのページに存在しているかが格納されているファイル。この情報にしたがって適宜 Vacuum をスキップすることで高速化している。ファイル名はテーブルのファイルノード番号に _vm という接尾辞が付いたもの。主に base ディレクトリのデータベースごとのサブディレクトリ内に配置される

- **空き領域マップ (Free Space Map) ファイル**
Vacuum によって回収され再利用可能となった場所の情報が格納されているファイル。ファイル名はテーブルのファイルノード番号に「_fsm」という接尾辞が付いたもの。主に base ディレクトリのデータベースごとのサブディレクトリ内に配置される

システムカタログ—— PostgreSQL の管理情報

システムカタログには、テーブルや列の情報などのスキーマメタデータと内部的な情報が格納されています。クエリ内で要求されたテーブルがデータベース内に存在するかチェックするなど、システムカタログはさまざまな用途で使用されます。システムカタログは通常のテーブルと同じであり変更できますが、データベースの状態を破壊する可能性があるため、通常は手作業で変更することは推奨されません。

PostgreSQL には次のようなシステムカタログが存在します。

- **pg_class**
テーブル、インデックス、シーケンス、ビュー、マテリアライズドビューなどに関する情報が格納されている
- **pg_extension**
PostgreSQL にインストールされた拡張機能（後述）に関する情報が格納されている
- **pg_statistic**
データベースの内容に関する統計情報が格納されている。この情報は、ANALYZE コマンドによって作成され、プランナが実行計画を生成する際に使用される

ここではテーブルを新たに作成して、**pg_class** システムカタログが更新されていることを確認してみましょう。確認するには、**psql** コマンドを使用します。**psql** コマンドは、PostgreSQL の対話的ターミナル型フロントエンドで、SQL などを入力すると入力サーバに送信され、結果を得ることができます。ここでは、次のようにして、**psql** コマンドでテーブルの一覧を **pg_class** システムカタログから取得します。最初の状態では、**public** スキーマにはテーブルは存在しません。

```

postgres=# SELECT pg_class.oid, pg_namespace.nspname, pg_class.relname
postgres-# FROM pg_class INNER JOIN pg_namespace ON pg_class.relnamespace = pg_
namespace.oid
postgres-# WHERE pg_class.relkind = 'r' AND pg_namespace.nspname = 'public'
postgres-# ORDER BY pg_class.relname;
   oid | nspname | relname
-----+-----+-----
(0 rows)

```

ここで新しいテーブルを作成してみましょう。pg_class システムカタログが更新され、新しいテーブルに関する情報が追加されていることが確認できます。

```

postgres=# CREATE TABLE test(id INTEGER);
CREATE TABLE
postgres=# SELECT pg_class.oid, pg_namespace.nspname, pg_class.relname
postgres-# FROM pg_class INNER JOIN pg_namespace ON pg_class.relnamespace = pg_
namespace.oid
postgres-# WHERE pg_class.relkind = 'r' AND pg_namespace.nspname = 'public'
postgres-# ORDER BY pg_class.relname;
   oid | nspname | relname
-----+-----+-----
 16389 | public  | test
(1 row)

```

システムカタログに格納されている情報は、運用上重要な役割を果たします。ほかにどのようなシステムカタログが存在するかは、PostgreSQL の公式ドキュメントを参照してください。

トランザクション——データの一貫性を保つ

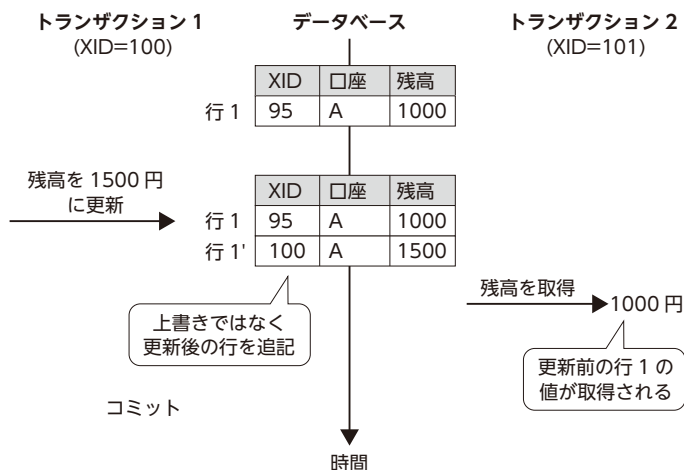
データベースでは、さまざまな検索・更新などが並行して行われます。これらの操作が可能な限り競合することなく、トランザクションを実現させる必要があります。PostgreSQL では、MVCC (*Multi Version Concurrency Control*、多版型同時実行制御) によって同時実行制御を実現しています。PostgreSQL における MVCC では、ある行に対して変更がなされたときに、変更前と変更後のデータとともに保持します。データの更新は実際に行を上書きするのではなく、テーブルの末尾に追記したり空き領域に書き込んだりする形で行われます。これによって、トランザクションごとに異なる値を読

み取ることが可能になります。

以下では、PostgreSQL での MVCC の挙動について説明します。トランザクションを開始すると、トランザクション ID (XID) が払い出されます。XID は 32 ビット整数であり、XID が大きいことはそのトランザクションのほうがあとに開始されていることを示します。行には、その行を挿入・削除したトランザクションの XID が記録されます。トランザクションでデータをスキャンする際は、この XID と自身の XID を比較することで、どの行を読むべきかを判断しています。

図 1.8 を用いてこの様子を見ていきます。ここでは先ほどの銀行口座の例を使用します。トランザクション 1 が口座の残高を 1,500 円に更新すると、PostgreSQL はその行を上書きするのではなく、新たな行 (行 1') としてテーブルの末尾に追記します。ただし、テーブルの中間に空いている行が存在する場合には、新たに行を増やして追記するのではなくその空いている行が利用されます。トランザクション 1 がまだコミットしていない状態でトランザクション 2 が口座の残高を取得したとき、行 1' (1,500 円) は見えず、行 1 (1,000 円) だけを読むこととなります。このようにして、トランザクションごとに異なる時点でのデータを参照できるようになっています。

図 1.8 MVCC



トランザクション 1 とトランザクション 2 のコミット後、行 1 は不要な領域となるためバキューム処理によって回収され、以降のトランザクションで新たに行を挿入するときにその領域が再利用されます。

レプリケーション——データベースを複製する

PostgreSQL では、レプリケーションによってデータベースを複製できます。レプリケーションの詳細は第 8 章で解説します。

レプリケーションでは、1 台のプライマリと 1 台以上のスタンバイが存在します。データの更新はプライマリに対して行われ、スタンバイがその変更を表す WAL を受け取り随時適用していくことで、レプリケーションが行われます。

PostgreSQL のレプリケーションには、大きく分けてストリーミングレプリケーション（物理レプリケーション）とロジカルレプリケーション（論理レプリケーション）が存在します。ストリーミングレプリケーションでは、WAL レコード（物理的な更新情報）をそのままスタンバイへ転送するのに対し、ロジカルレプリケーションでは、テーブルのどの行をどのように更新したかなどのデータの論理的な変更情報を転送するという点に違いがあります。

レプリケーションは、次のような役割を持ちます。

● データベースの可用性（耐障害性）を上げる

レプリケーションの一つの目的は、データベースの可用性（耐障害性）を上げることにあります。プライマリに障害が発生した場合、スタンバイがプライマリに昇格することで、全体として動作を継続できます。

● 負荷分散で性能を上げる

レプリケーションのもう一つの目的に、参照の負荷を分散させることが挙げられます。スタンバイの一つの構成としてホットスタンバイ構成^{注6}を取

注6 PostgreSQL において、プライマリでのデータ更新をスタンバイにも反映させながら、スタンバイのデータ参照をユーザーに許可する構成のことです。

る場合、ホットスタンバイは、データの更新を受け付けることはできないものの、参照処理を行うことができます。そのため、クライアントからの参照クエリの負荷を分散させることで、クラスタ全体の性能を向上できます。

クエリ最適化

SQL を実行する際、ユーザーは結合演算などの処理をどのようなアルゴリズムで実行するかを考える必要がありません。PostgreSQL は、コストベースのクエリ最適化によって、適切なアルゴリズムを選択して効率的にクエリ実行することを可能にしています。

クエリが発行されると、PostgreSQL は実行計画を生成します。この実行計画には、クエリをどのようなアルゴリズムで実行するかなどの情報が含まれています。PostgreSQL では、実行計画のコストを評価して、コストが最小になる実行計画を生成するように努めます。このコストは、テーブルのサイズ・インデックスの有無・アルゴリズムの計算量などから PostgreSQL のプランナによって見積もられます。詳細は第 7 章で説明します。

PostgreSQL が生成した実行計画は、SQL の先頭に `EXPLAIN` と付けることによって表示できます。以下に簡単な SQL の実行計画を示します。ここでは、`integer` 型の列のみを持つテーブル `t1` に対して、10,000 件のデータを挿入したあと、それを全件取得しています。この結果から、PostgreSQL は、テーブル `t1` に対して Seq Scan (シーケンシャルスキャン、順次検索) を行う実行計画を生成したことがわかりました。`EXPLAIN` ではクエリの実行計画を表示するだけで実行はされませんが、`EXPLAIN ANALYZE` とすることで、実際に実行し、実行計画のどのノードでどの程度の時間を要したかなどの詳細な情報を取得できます。

```
postgres=# CREATE TABLE t1(id integer);
CREATE TABLE
postgres=# INSERT INTO t1 SELECT id FROM generate_series(1, 10000) AS id;
INSERT 0 10000
postgres=# EXPLAIN SELECT * FROM t1;
          QUERY PLAN
-----
Seq Scan on t1 (cost=0.00..145.00 rows=10000 width=4)
```

```
(1 row)

postgres=# EXPLAIN ANALYZE SELECT * FROM t1;
                                         QUERY PLAN
-----
Seq Scan on t1 (cost=0.00..145.00 rows=10000 width=4) (actual time=0.020..0.503
rows=10000 loops=1)
  Planning Time: 0.071 ms
  Execution Time: 0.739 ms
(3 rows)
```

拡張機能

PostgreSQL の大きな特徴として、拡張機能の存在が挙げられます。拡張機能はプラグインの形で提供され、PostgreSQL 本体にはない機能を追加することが可能です。PostgreSQL のソースコードには、contrib ディレクトリに拡張機能があらかじめ用意されていますが、それ以外にもさまざまな場所で配布されている拡張機能を利用することが可能です。自分で拡張機能を開発して配布することも可能です。

ここでは、`auto_explain` と呼ばれる拡張機能をソースコードからインストールして試してみましょう。`auto_explain` は、SQL が実行されるとその実行計画をログに出力するもので、PostgreSQL のソースコードの `contrib/auto_explain` に存在します。このディレクトリに移動して、次のコマンドを実行することでコンパイル・インストールします。

```
$ cd ソースコードを展開したディレクトリ / contrib / auto_explain
$ make
$ make install
```

続いて、`postgresql.conf` に次の内容を記載します。

```
shared_preload_libraries = 'auto_explain'
auto_explain.log_min_duration = 0
```

1 行目で `auto_explain` のモジュールを読み込むことを指定しています。2 行目は、`auto_explain` に関する設定項目です。このように設定することで、

すべての SQL について実行計画がログファイルに出力されるようになります。

それでは、この拡張機能を試してみましょう。先ほどの例と同様に、`psql` コマンド上で `SELECT * FROM t1` という SQL を実行します。この SQL を実行すると、ログファイルに対して次のような内容が書き出されます。

```
2024-11-11 13:55:25.414 JST [239] LOG: duration: 1.758 ms plan:  
Query Text: SELECT * FROM t1;  
Seq Scan on t1 (cost=0.00..145.00 rows=10000 width=4)
```

このログから、実行した SQL の実行計画を知ることができます。もし、実際の運用でこのクエリの実行に時間を要しているなどのトラブルが発生した場合は、こうしたログを参照することで問題を解決できる可能性があります。ログファイルは、`postgresql.conf` 内で指定したパスに出力されます。詳細については第 14 章を参照してください。

`auto_explain` で提供されている機能は、PostgreSQL 本体には存在しません。拡張機能を利用することで、PostgreSQL をより便利に利用できるようになります。