
PGECons
PostgreSQL Enterprise Consortium
Oracle技術者のための
PostgreSQL自習書
第2版
PostgreSQL Enterprise Consortium
WG2(移行WG)
Copyright © PostgreSQL Enterprise Consortium. All Rights Reserved.

はじめに

■ 本書の目的

本書はこれからPostgreSQLによる商用の開発・運用を行う人向けの基礎的な技術文書になります。Oracleデータベース（以下Oracleと略します）経験者がPostgreSQLの概要を理解することを目的とし、一部Oracleとの比較を含め説明しております。

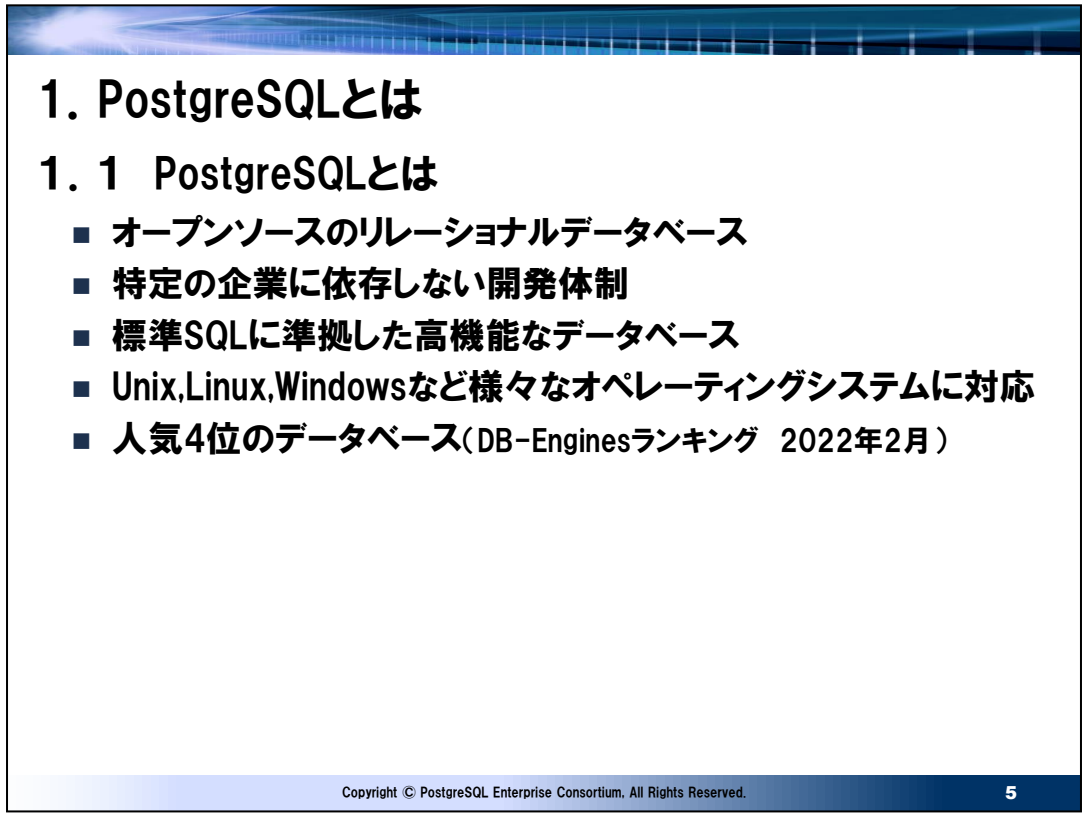
■ 前提条件

RDBMS,SQLの基礎的知識があること

※本書執筆時点のPostgreSQLの最新バージョンは14になります。

目次

1. PostgreSQLとは



- 開発方針

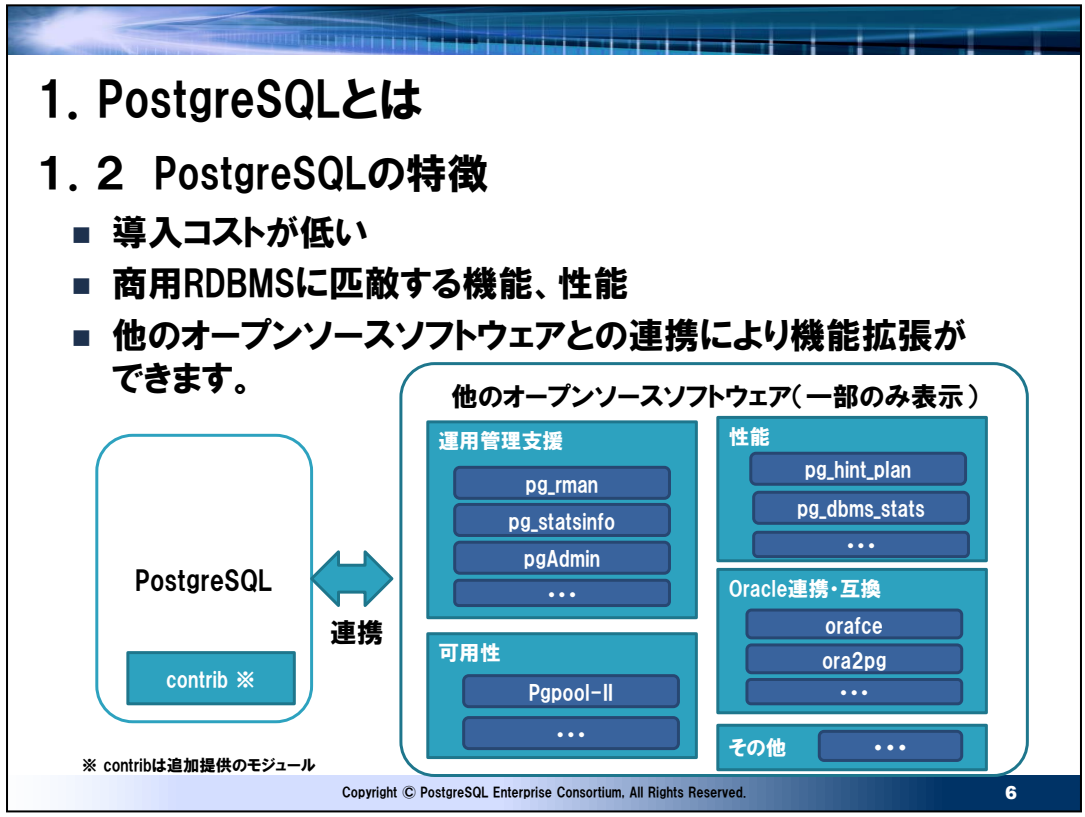
開発コミュニティ（PostgreSQL Global Development Group）による開発方針は合議制であり、特定企業の方針に依存することはありません。

- ライセンス

独自のPostgreSQLライセンスです。これはソースコードを無償で利用可能で、かつソースコードを複製、改変した場合もそのソースコードを公開することなく頒布することのできるライセンスです。

- Oracleデータベースからの移行の親和性

Oracleデータベースからの移行先としてPostgreSQLが検討対象になることが多いです。レプリケーション、パーティショニングテーブル、パラレルクエリー、ストアドプロシージャのサポートなどの機能追加により過去にOracleからの移行で課題とされた部分は解消されつつあり、大規模システムへの採用も増加しています。



● 外部プロジェクト

他のオープンソースソフトウェア（以後、周辺ソフトウェアと呼びます）との連携によりPostgreSQLは機能拡張※１ができます。これはPostgreSQLのマニュアルである「PostgreSQL文書」※２では付録で外部プロジェクトと呼ばれております。これらを商用で継続的に活用するには以下２点の考慮が必要です。

- PostgreSQLのメジャーバージョンに依存するものもあり、これらはPostgreSQLのメジャーバージョンアップに遅れてリリースされます。利用するシステムがメジャーバージョンアップを実施する場合、これらの外部プロジェクトのソフトウェアのバージョンアップも考慮する必要があります。
- 開発が継続される保証はありません。継続されない場合は代替策を検討する必要があります。

※１　PostgreSQLは拡張機能を構築するための基盤が用意されています。そのため必要に応じて作るということも可能であり、「PostgreSQL文書」の「SQLの拡張」の節で説明されています。

※２　以下のURLを参照（XXはメジャーバージョンを入れます）

https://www.postgresql.jp/document/XX/html/

● contrib

「PostgreSQL文書」では付録で説明される追加提供のモジュール群です。使用する場合はpostgresql.confでの指定やエクステンションの追加等が必要になります。有用なモジュールが多く、商用運用では運用管理、性能チューニングの観点からpg_stat_statements,pgstattupleなどが多く利用されます。

● サポート

オープンソースソフトウェアであるため、トラブルは利用者による「自己責任」での解決が必要になります。商用の安定的な運用においてはリスクになりますので、この問題を解決するために、PostgreSQLのサポートサービスが各社より提供されております。以下をご参照ください。

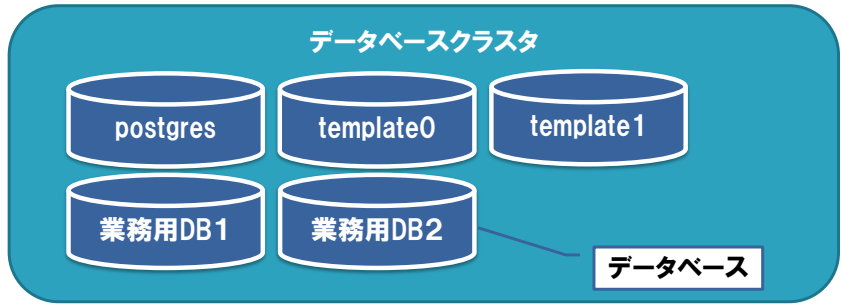
https://www.pgecons.org/postgresql-info/services/

2. データベースの構造

2. データベースの構造

2. 1　データベースクラスタとデータベース

- **データベースクラスタ**
  - PostgreSQLでは、１つのサーバインスタンスにより管理されるデータベースの集合のことを言います。
  - データベースクラスタはinitdbコマンドで作成されます。



- initdbコマンド

実行例

```
initdb --lc-collate="C" --lc-ctype="C" --encoding=UTF8 -k -D データベース格納ディレクトリ
```

ロケール：	照合順と文字セットクラスを指定。“C”または“POSIX”以外はロケールに依存した文字列比較の処理が行われるため性能が劣化します。 このオプションが未指定の場合、環境変数LANGのlocaleが使用されます。
--encoding：	データベースのデフォルト符号化方式。SJISは未サポートです。
-k:	read/write時のデータページのチェックサムを有効化します。有効にしない場合、ページヘッダの破損は検出しますがページデータ部に破損があっても検出できません。他の商用DBでは従来から実装済みでPostgreSQLは9.3以降で指定可能になりました。またPostgreSQL 11以前はデータベースクラスタ作成後に変更することはできません。
-D：	データベース格納ディレクトリ。PGDATA環境変数を指定してれば省略可能。表領域の追加または先行書込みログ（WAL。OracleでのREDOログファイル）用のディレクトリを任意に指定しなければ、すべてのデータベースクラスタの構成資源はこのディレクトリ以下に作成されます。

- その他データベースクラスタの作成方法

パッケージ付属のセットアップスクリプトまたはソフトウェアのインストーラでの自動作成（Windows版）でもデータベースクラスタの作成は可能です。

2. データベースの構造

2. 1 データベースクラスタとデータベース

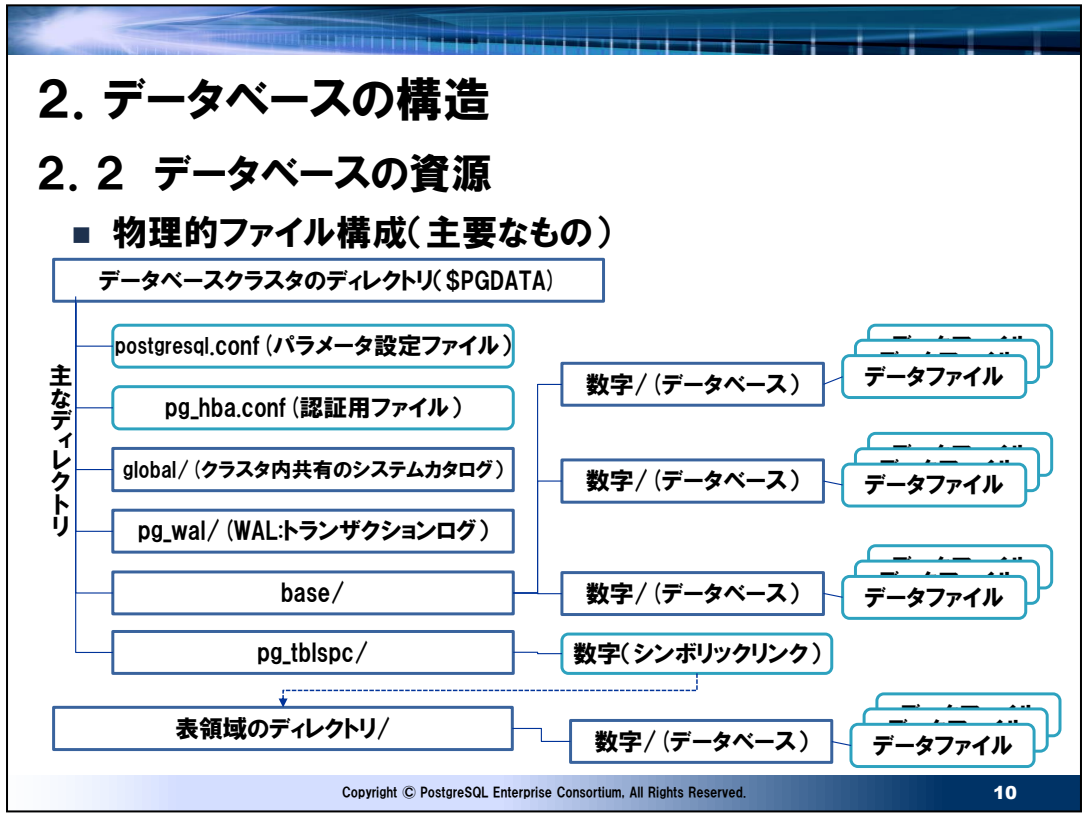
- データベース
  - データベースクラスタを作成時に初期状態で postgres,template0,template1の3つのDBが作成されます。
  - データベースを作成するコマンド(createdb)で業務用のデータベースを追加します。

データベース名	作成コマンド	説明
postgres	initdb (OS)	ユーティリティ、ユーザ、サードパーティ製アプリケーションが使用するデフォルトデータベース
template0	initdb (OS)	Postgresの提供バージョンの標準雛形
template1	initdb (OS)	createdbのデフォルトの雛形。初期はtemplate0と同一
任意の追加DB (業務用)	createdb (OS) create database (SQL)	業務用で使用するオブジェクトを格納するためのデータベース

- 異なるエンコードでのデータベースの追加

PostgreSQLではデータベースクラスタ作成時に指定したエンコードとは異なるエンコードでデータベースを追加できます。このときの注意点として以下のように-Tパラメータでtemplate0データベースを指定します。

createdb 新規に作成するデータベース名　-E デフォルトと異なるエンコード　-T template0



● データベースクラスタの物理的な構成

データベース環境を構成するファイルは、initdbで指定した-Dパラメータ（未指定の場合はPGDATA環境変数）のディレクトリ配下に格納されます。

●pg_wal

先行書込みログ＝WAL（WriteAheadLog)と呼ばれるファイルの格納領域。WALは不要になると自動で削除されます。WALはデータベースクラスタ毎に存在します。（データベース毎ではありません）

●システムカタログ

データベースの定義情報などはシステムカタログに格納されます。システムカタログはデータベース全体で共有するものはglobalに、データベース個別に管理するものは各データベースに格納されます。

●表領域

表領域を追加することで初期のデータベースクラスタと異なるディレクトリにデータベース資源を配置することができ、これによりＩ／Ｏ分散が図れます。データベースクラスタのpg_tblspcディレクトリに追加された表領域ディレクトリのシンボリックリンクが登録されます。ただし、一つのデータファイル（テーブル）を複数の表領域に分散配置することはできません。Oracleの表領域とは異なり、単なるデータベースを格納できる場所と認識すればよいでしょう。

●データベースディレクトリ

ファイルシステム上では数字で表現されます。oid2nameコマンドでデータベース名とこの数字の紐付けを確認することができます。

●データファイル

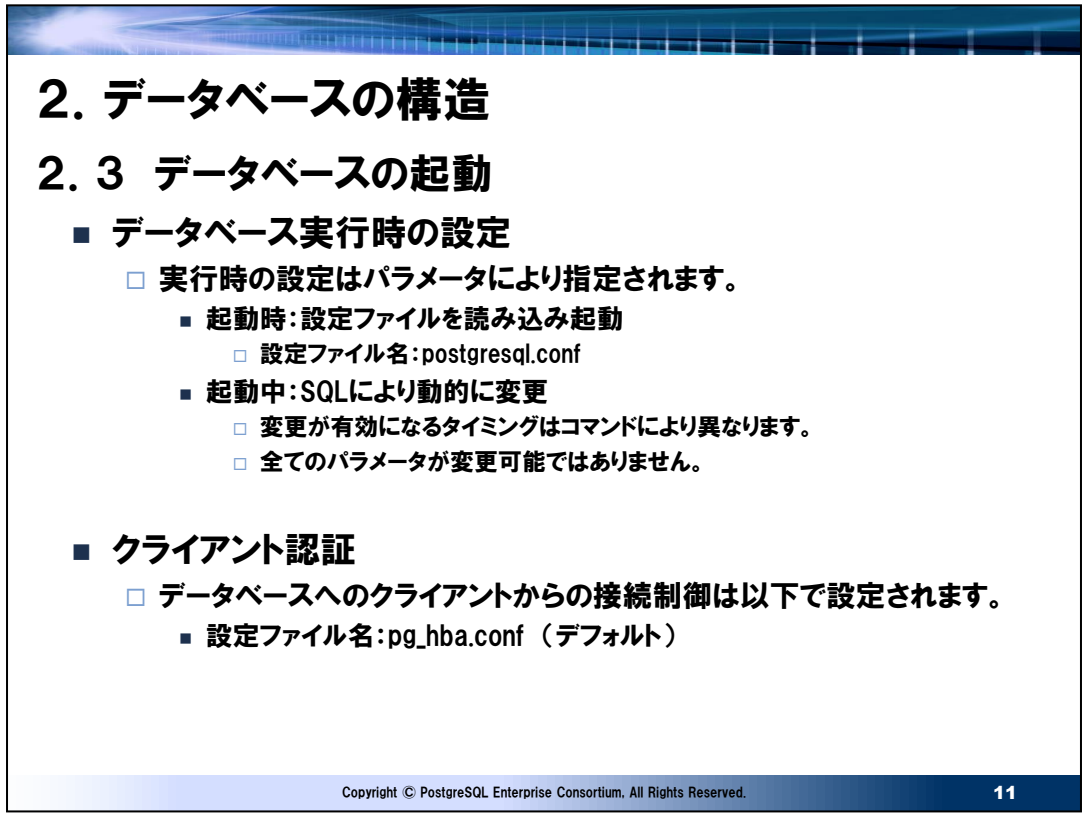
ファイルシステム上では以下の名称で格納されます。

数字のみで表現されるものがテーブルやインデックスの実データ。

数字_fsm：Free Space Map(空き領域マップ)　空き領域がどこにあるか管理しています。

数字_vm:Visibility Map(可視性マップ）不要領域があるページを管理できるようにしています。

数字.1,数字.2,・・：数字のテーブルやインデックスのファイルが１GBを超えると分割されます。これはプラットホーム毎のファイル容量制限の問題を回避するための措置です。

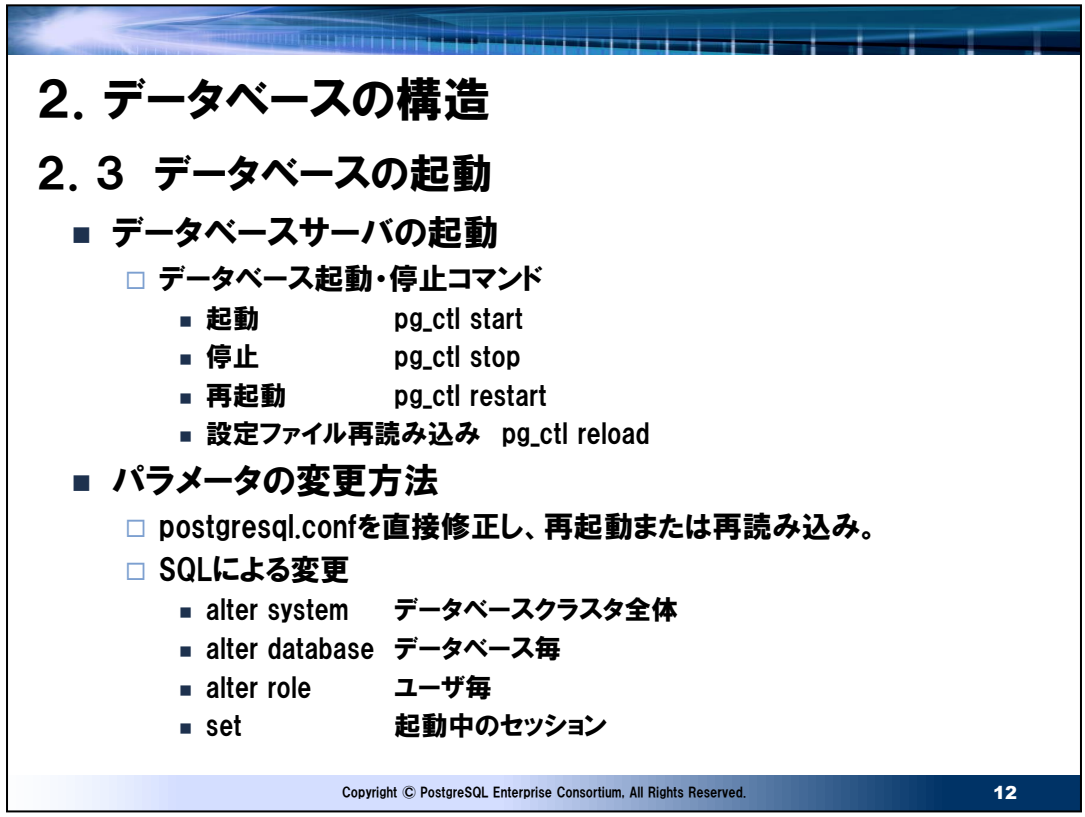


- postgresql.confファイル

postgresql.confファイルはデータベースクラスタを初期化時にデータベースクラスタ格納ディレクトリに作成されるテキストファイルです。パラメータは１行に１つのパラメータを記述し、ファイル中の空白行は無視、また#の後はコメントとして扱われます。パラメータの詳細と指定方法は「PostgreSQL文書」のサーバの設定の章に説明がありますが、PostgreSQLのメジャーバージョンにより若干の差異があるため、使用するバージョンのマニュアルを確認してください。

- pg_hba.confファイル

データベースに接続するための接続先アドレス、ポート、接続数などの全体のルールはパラメータで設定されますが、個別の認証設定はpg_hba.confにより設定します。このファイルはデータベースクラスタを初期化時にデータベースクラスタ格納ディレクトリに作成され、パラメータのhba_fileにより、格納先、ファイル名を変更することができます。このファイルもpostgresql.conf同様に１行で１つの認証方法を指定し、ファイル中の空白行は無視、また#の後はコメントとしてして扱われます。認証パラメータの詳細と指定方法は「PostgreSQL文書」のクライアント認証の章を確認してください。



● データベースの起動・停止

PostgreSQLのデータベースの起動・停止はユーティリティコマンドであるpg_ctlを使用して実施します。pg_ctlはサーバアプリケーションであるpostgresを制御するためのラッパープログラムであり、データベースを安全かつ簡単に起動、停止することができます。オプションの指定等は「PostgreSQL文書」のリファレンスを参照してください。

サーバの起動についてはOracleのようにインスタンスの起動、表領域のマウント、表領域のオープンのように段階的な操作はありません。サーバの停止については-mオプションで以下のモードが用意されています。

- smart　全てのクライアントの切断とオンラインバックアップの終了を待ってサーバを終了させます。
- fast　クライアントの終了を待たず、オンラインバックアップは即時終了させてサーバを終了します。
  これは停止モードのデフォルトの動作になります。
- immediate　クリーンアップ処理なしで即時終了します。Oracleのshutdown abortと同様で、次回起動時にインスタンスリカバリが実行されます。

● パラメータの変更

SQLによるパラメータの変更方法はいくつかあり、alter database,alter roleによる個別のパラメータの上書きは運用の自由度は向上しますが、運用管理を複雑にします。確実に管理できる場合に限り指定したほうが良いでしょう。以下にSQLによる変更方法を列挙します。

- alter system set または reset

  実行するとpostgresql.auto.confファイルに変更するパラメータのエントリが追加、修正、削除（reset時）されます。パラメータの反映は即時ではなく、次回の再起動または設定ファイルの再読み込みのタイミングになります。postgresql.auto.confファイルはpostgresql.confファイルの指定を上書きします。

- alter database データベース名 set または reset

  指定したデータベースでコマンド実行後以降から発生する新規セッションについて有効になります。postgresql.confパラメータの変更より優先されますが、すべてのパラメータを変更できるわけではありません。なお、変更はサーバを再起動しても有効です。この永続的なパラメータ変更の指定はシステムカタログ（本書４．１章参照）のpg_db_role_settingにより確認できます。

- alter role ロール名 [in database データベース名] set または reset

  指定したユーザで、かつデータベース名が指定された場合はそのデータベースを限定にコマンド実行後以降から発生する新規セッションについて有効になります。なお、alter database同様に全てのパラメータを変更できるわけではなく、また変更は永続的であり、変更内容の確認も同様です。

- set

  起動中のセッション内でのみ有効です。バッチ処理内のwork_memの変更など、特異な条件を満たすために使用されます。postgresql.confパラメータより優先されますが、すべてのパラメータを変更できるわけではありません。

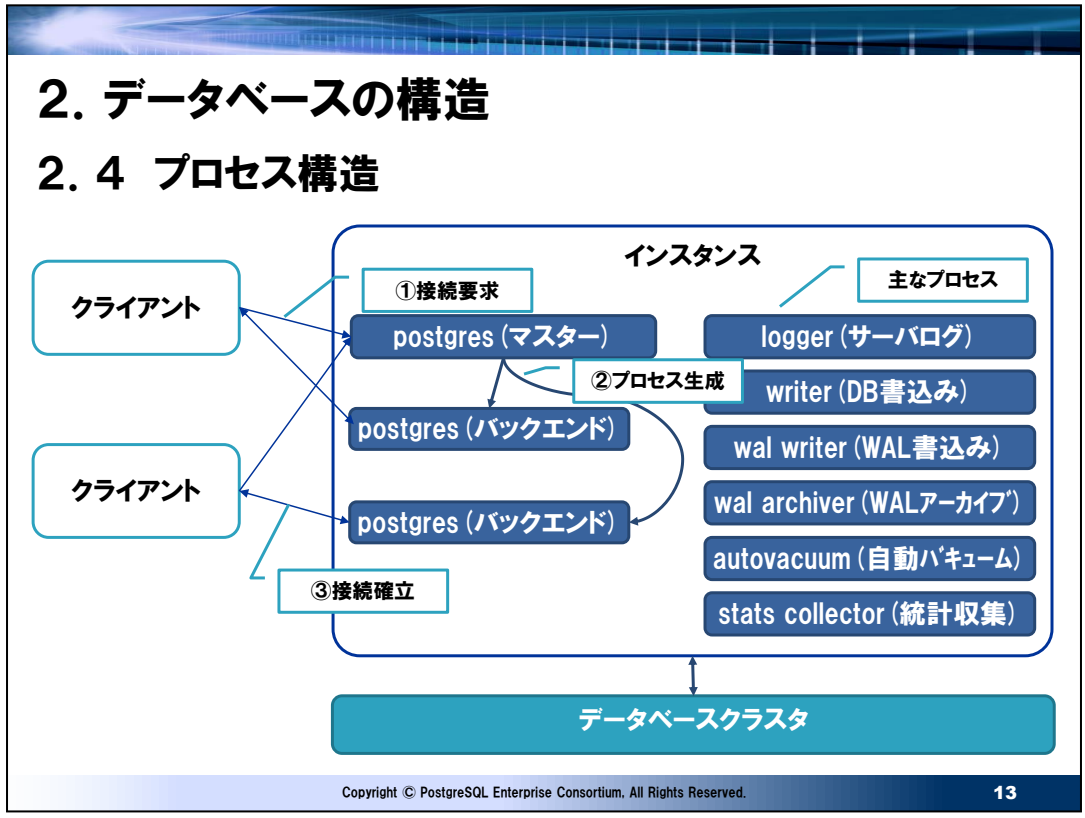
● パラメータ指定の優先度

優先度の高い順にset,alter role ロール名 in database データベース名, alter role ロール名,alter database データベース名,alter systemになります。

● パラメータの確認

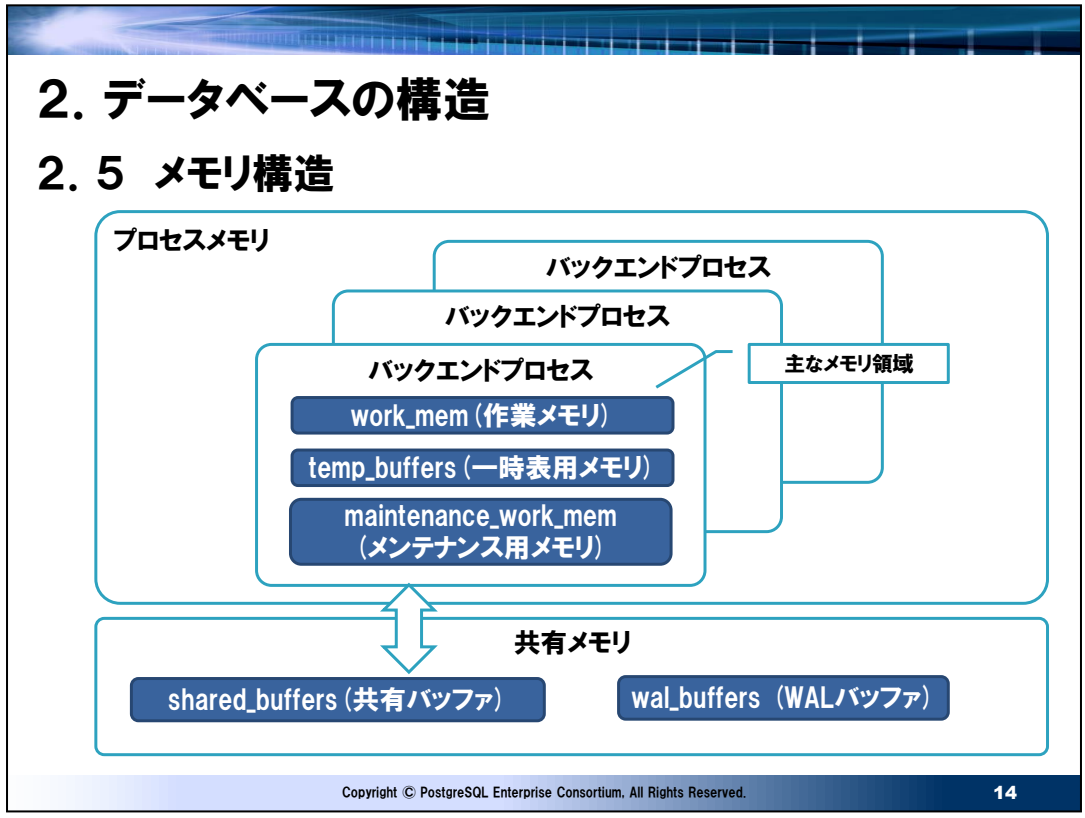
上記のようにパラメータは初期指定、ロールやデータベースの指定、セッション内の指定と柔軟にできます。実行時点でのパラメータはshowコマンドで確認可能で、より多くの様々なパラメータの情報はシステムカタログ（本書４． １章参照）のpg_settingsを参照することで確認できます。以下は確認例です。

```
$ psql postgres
psql (12.4)
Type "help" for help.

postgres=# show work_mem;
 work_mem
----------
 4MB
(1 row)

postgres=# ¥x
Expanded display is on.
postgres=# select * from pg_settings where name = 'work_mem';
-[ RECORD 1 ]---+-----------------------------------------------------------------------------------------------------------------
-----------
name            | work_mem
setting         | 4096
unit            | kB
category        | Resource Usage / Memory
short_desc      | Sets the maximum memory to be used for query workspaces.
extra_desc      | This much memory can be used by each internal sort operation and hash table before switching to temporary
disk files.
context         | user
vartype         | integer
source          | default
min_val         | 64
max_val         | 2147483647
enumvals        |
boot_val        | 4096
reset_val       | 4096
sourcefile      |
sourceline      |
pending_restart | f
```



- PostgreSQLのプロセス

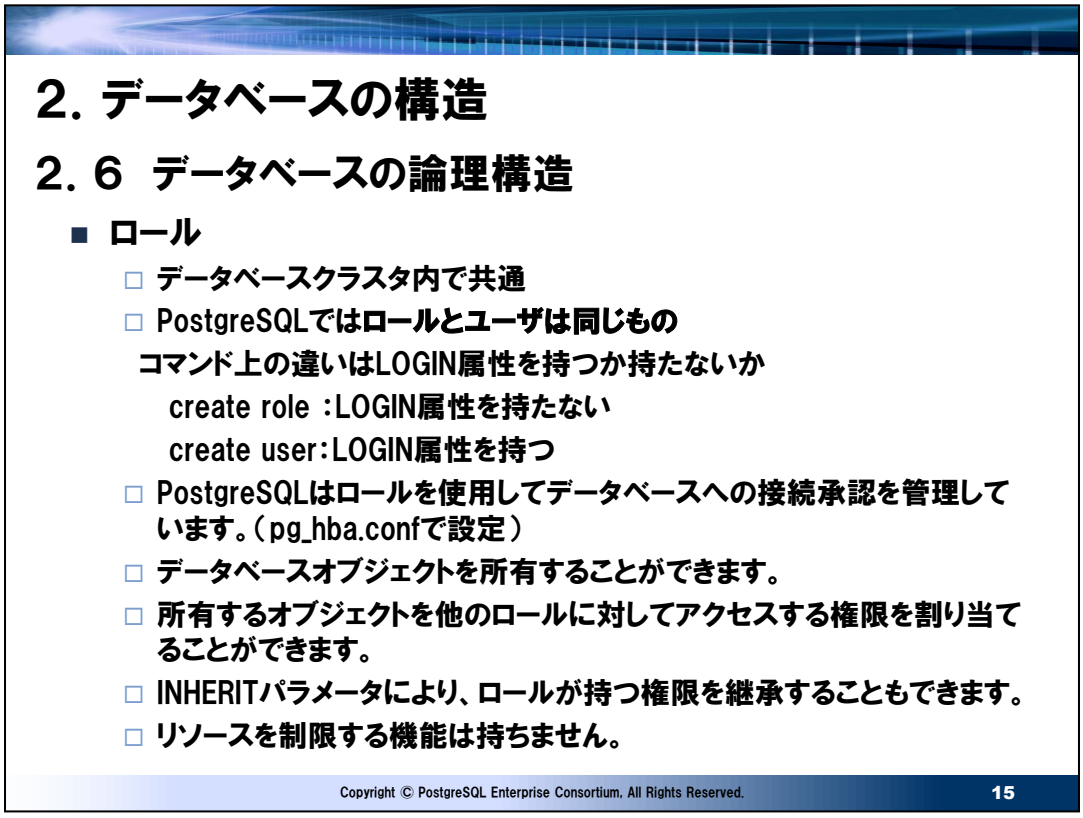
PostgreSQLはスライドの図のようなマルチプロセスの構造となります。マスタープロセスは最初に起動される親プロセスであり、他のプロセスを起動します。アプリケーションからの接続は、マスタープロセスが接続要求を受け、対応するバックエンドプロセスを生成し、そのバックエンドプロセスがアプリケーションとの通信を行います。



- PostgreSQLのメモリ構造

大きくは共有メモリとプロセスメモリに分けることができます。共有メモリはインスタンスに対し作成され、プロセスメモリはバックエンドプロセス毎に作成されます。

- 共有バッファ：テーブルやインデックスのデータをキャシュする領域。OracleのDBバッファ（DB_CASHE_SIZE）と同様です。
- WALバッファ：WALをキャシュする領域。OracleのREDOログバッファ（LOG_BUFFER）と同様です。
- 作業メモリ：SQL実行時のソート、ハッシュに使われる領域。バックエンドプロセス毎に確保されるため同時実行ユーザ数が多い場合は合計のメモリ量は増大しやすくなります。OracleのSORT_AREA_SIZEと同様です。Oracleのように PGA_AGGREGATE_TARGETとしてインスタンス単位での指定はできないため想定されるバックエンドプロセス数を考慮し指定する必要があります。
- 一時表用メモリ：一時表をアクセスするときに使用されるセッション局所バッファ。
- メンテナンス用メモリ：バキューム、索引構築等の保守作業で使用されるメモリです。通常同時に複数のバックエンドプロセスから実行されるものではないため、作業メモリより大きめの値にしてもよいでしょう。ただし自動バキュームが並列で実行されると並列度に応じて増えてしまうためメモリを枯渇させる場合にはautovacuum_work_memを別途指定します。



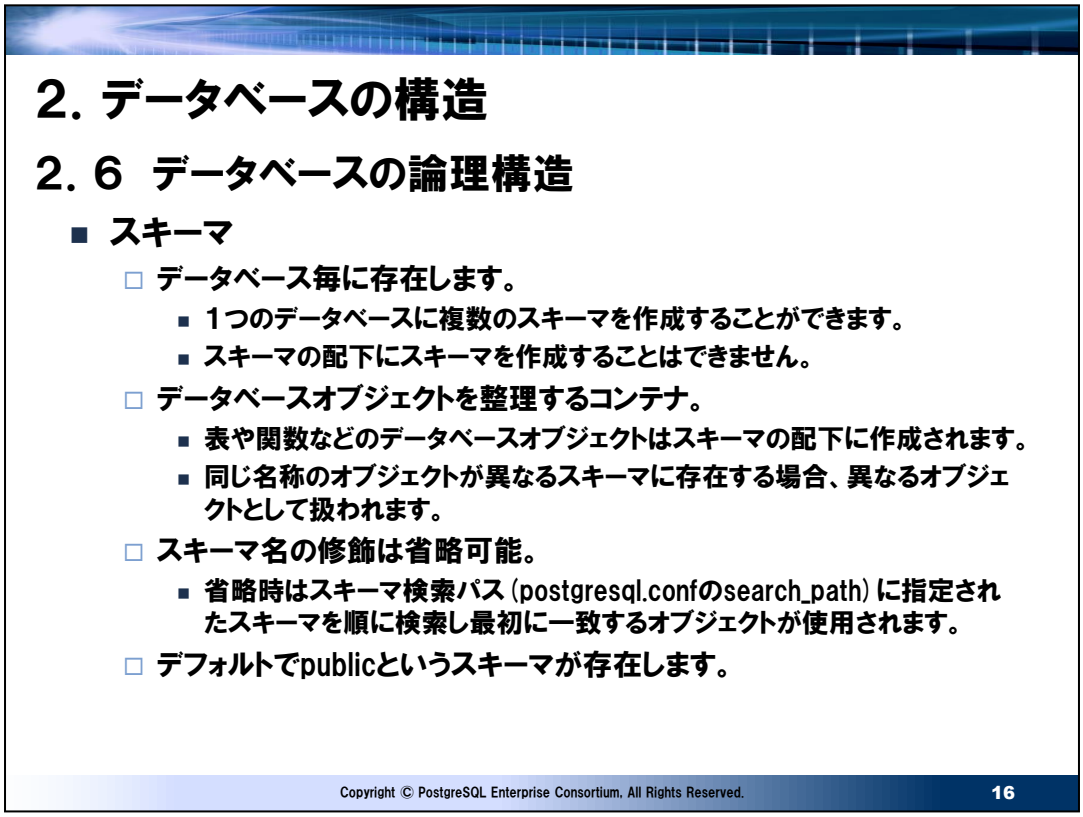
- PostgreSQLでのユーザとロール

PostgreSQLではロールとユーザは同じものです。データベースオブジェクトを所有し、権限を持つことかできます。Oracle同様に権限の継承に使用する場合はINHERITパラメータ（デフォルトで付与される）を用いることで可能です。逆にユーザとして（権限の継承はしない）のみ使用する場合はNOINHERITを指定する必要があります。

- Oracleとのリソース、パスワード制御の違い

Oracleはプロファイルを利用してユーザ毎に細かいリソース制御、パスワードポリシーの指定ができますが、PostgreSQLで可能な制御は以下の通りになります。

- 同時ログインできる接続数
- パスワードの有効期限

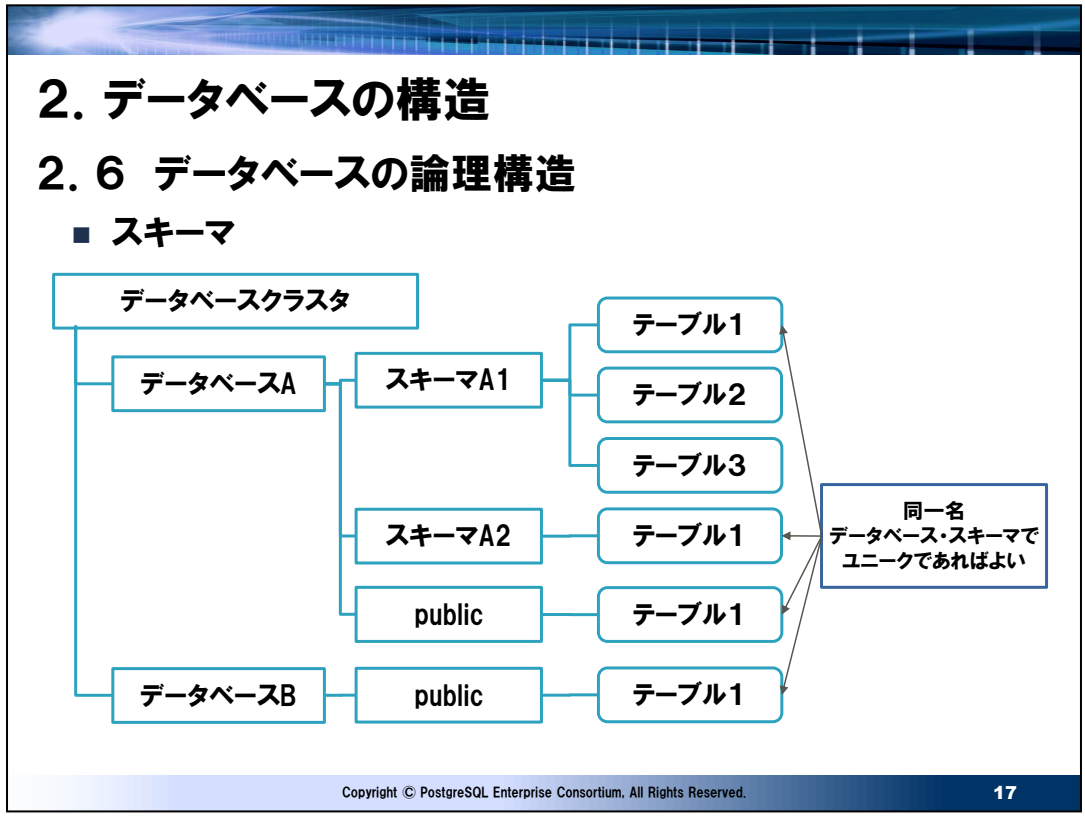


- ロールとスキーマ

Oracleではスキーマはユーザ名と同じ名前のスキーマが自動的に割り当てられます。PostgreSQLではスキーマとロールを関連付けることは必須ではありませんが、スキーマ検索パスのpostgresql.confのデフォルトは以下のようになっており、スキーマ名とロール名（=ユーザ名）を合わせておくと利便性が向上します。

search_path = '"$user", public'　# schema names

（上記ではユーザ名と同名のスキーマ名、publicの順にオブジェクトを検索します）



- 大文字、小文字の同一名について

PostgreSQLでは大文字、小文字を区別します。例えばテーブル名で“PRODUCT”と“product”は別のテーブルとして認識されます。さらにPostgreSQLのディクショナリはダブルクオーテーションでくくらないと小文字で登録されます。

create table product ····· はproductとして登録

create table PRODUCT ····· はproductとして登録

create table “PRODUCT” ····· はPRODUCTとして登録

トラブルを避けるため、開発に際しては記述方法を規約として定義しておくことを推奨します。これはテーブル名だけでなく、関数名、カラム名等も含め同一です。

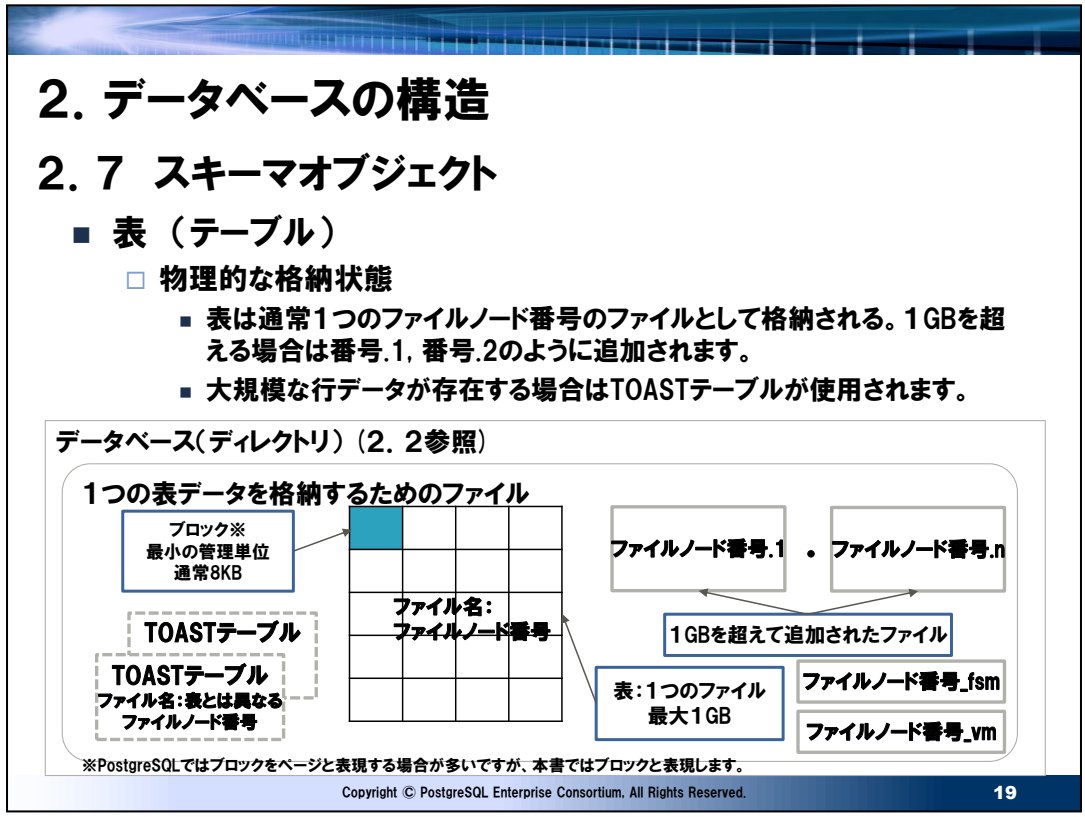
2. データベースの構造

2. 7 スキーマオブジェクト

- 表（テーブル）
  - 概念
    - 行と列で構成されます。
    - 列には名前があり、数と順序は固定されます。
    - 列にはデータ型があり、データ型によって値が制限されます。
    - 行の順序は保証されません。必要に応じて明示的に並び替えを要求する必要があります。

主なデータ型	PostgreSQL	Oracle（比較用）
文字列	char（文字数） varchar（文字数）	char（バイト数） varchar2（バイト数）
整数	integer	integer
固定小数点	numeric（精度,位取り）	number（精度,位取り）
日時	date：精度は年月日 timestamp：精度は1マイクロ秒まで	date：精度は年月日時分秒 timestamp：精度は1ナノ秒まで

- OracleとのPostgreSQLの仕様上の大きな相違
- PostgreSQLの文字列のサイズは文字数になります。
- PostgreSQLは長さ0バイトの文字列とNULLを区別します。



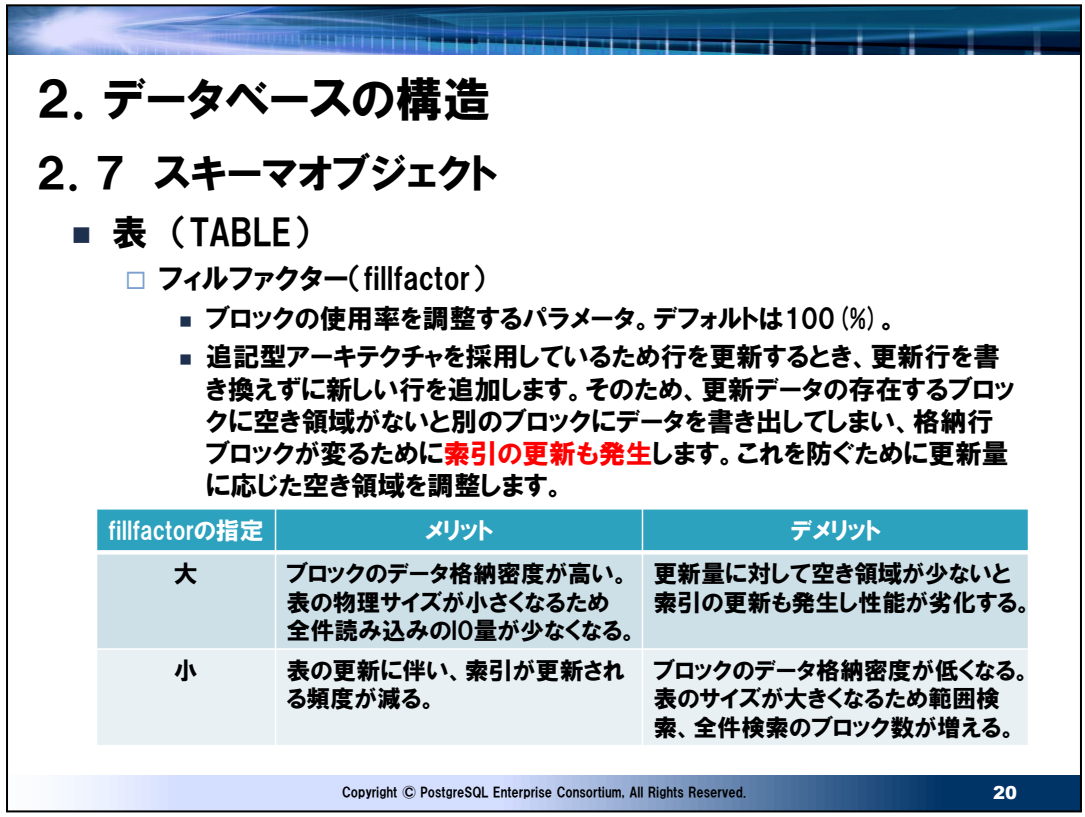
- 表の物理格納先

PostgreSQLのデータの最小の管理単位はブロックであり、ブロックの集合が表となります。通常、１つの表で１つのファイルを構成します。使用条件に応じて複数のファイルに表データは格納されます。

- TOASTテーブル

PostgreSQLは行データがブロックサイズ（通常8KB)を超える場合、格納することができません。このため大規模な可変長フィールド値を持つ場合、圧縮または物理的な分割を行い、この分割された行の格納先がTOASTテーブルという元の表とは異なる別のファイルに格納されます。なおTOASTテーブルは利用者からみれば透過的に発生するため、通常意識する必要はありません。

なお、Oracleは1行のデータを１ブロックに格納しなければならない制限はないのでTOASTのような機能を必要としません。１行が１ブロックで格納できない場合、Oracleは行連鎖が発生することになります。



- fillfactorとpctfree

OracleにもPostgreSQL同様にテーブルのブロック内の空き領域を調整するpctfreeパラメータがあります。OracleはPostgreSQLと異なりUNDOデータによる読み取り一貫性を確保しているため、更新時に行を追加することなく元データを上書きすることが可能です。このため空き領域は元データよりデータ長が大きくなった場合に使用されるためPostgreSQLと比較し少ない空き領域で済みます。ほとんどの場合、このパラメータはデフォルト値の10(%)で問題なく、変更が必須なのは初期に項目がNULLで登録され、更新により項目に値が入るようなケースになります。

また、Oracleは元データが同一ブロックに書き込めない場合は行移行（更新後データは別のブロックに書き込まれ、元のデータブロックには移行先のブロックアドレスが記録されます）、行連鎖が発生するため、索引の更新が発生することはありません。ただし、行移行、行連鎖が発生すると1件の行を読み込むために複数のブロックをアクセスする必要があるため性能は劣化します。この状態を抑止するためにpctfreeを調整します。つまり性能劣化防止の目的は同じですが、PostgreSQLの索引の更新を抑止することとは異なります。

- 運用中にfillfactorを変更したとき

既存の表データそのもののブロック使用率の状態が即座に変更されることはありません。テーブルの定義のみが変更され、以降のデータ更新から定義通りに動作するようになります。変更の状態を表データに反映させたい場合はVACUUM FULLを実行します。

- 索引更新が伴うケース

索引カラムに指定された項目が更新された場合はOracle、PostgreSQL双方とも索引は更新されます。

2. データベースの構造

2. 7 スキーマオブジェクト

- **索引（INDEX）**
  - 表の行へのアクセスを高速化するためのオブジェクト
  - 種類は以下の表の通り。

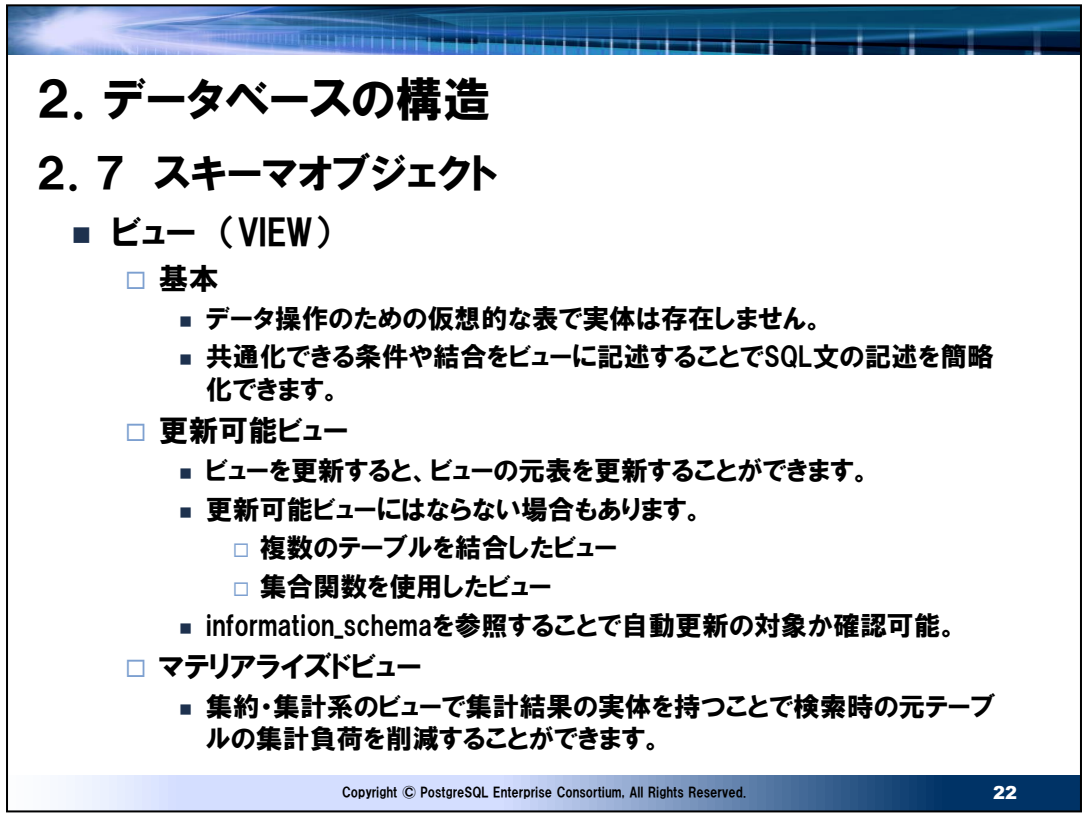
種類	PostgreSQL	Oracle（参考）
B-tree	○	○
Hash	○	－
GiST	○	アプリケーション・ドメイン索引
SP-GiST	○	
GIN	○	
BRIN	○	－
Bitmap	－	○
ファンクション索引	式に対する索引で対応	○
索引構成表	－	○

- PostgreSQLに移行する場合の注意点
- B-tree

  逆キー指定（索引列をビット反転させでキー項目とする）は出来ません。また、OracleではカラムがNULLの場合は索引は作成されませんが、PostgreSQLは索引を作成します。
- ビットマップインデックス

  代替の索引は存在しません。PostgreSQLの実行計画で見られるビットマップスキャンはインデックスから得られた候補行をメモリ上にビットマップ化し、テーブルをスキップしながらシーケンシャルアクセスするスキャン方法。事前にビットマップの索引が存在している訳ではありません。
- 索引構成表

  B-tree索引構造に表データが存在する表です。PostgreSQLには存在しませんので、B-tree索引を作成し対応します。



- Oracleと比較したPostgreSQLのビューの制限
- 参照専用の指定（WITH READ ONLY)ができません。利用者と元表の所有者を分けてオブジェクト権限の付与で対応する必要があります。
- マテリアライズドビューのリフレッシュはPostgreSQL 14時点でサポートされるのは完全リフレッシュのみです。高速リフレッシュ、自動リフレッシュ（ON COMMIT,ON STATEMENT)機能はありません。

3. データベースの操作

3. データベースの操作

3. 1 SQL言語

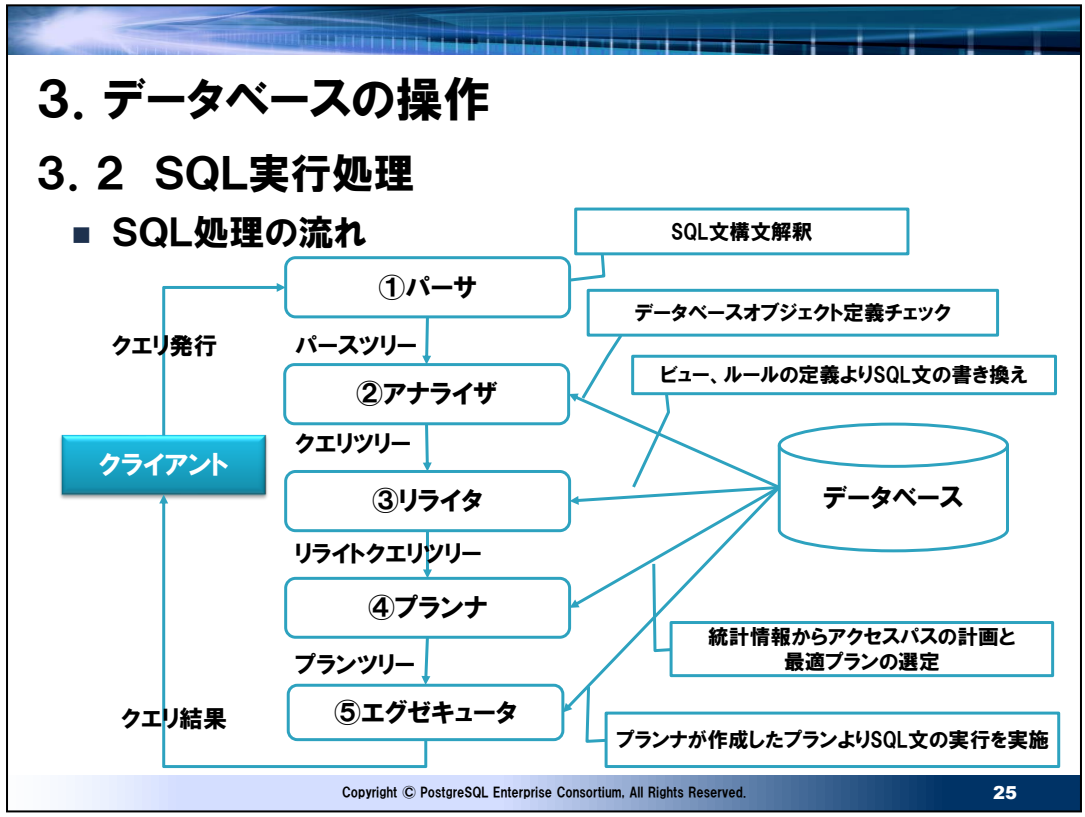
- SQL
  - 標準SQLへの準拠
    - ISO/IEC 9075/2016に準拠
    - 完全なコアの互換性：179の必須機能の内、160個に適合
  - Oracleとの主な相違（移行時に問題となりやすい部分）

	Oracle	PostgreSQLでの対応
SELECT	外部結合演算子（＋）	SQL標準に書き換え
	DUAL表	指定不要
	ヒント句	削除またはpg_hint_planの導入
	MINUS	EXCEPTに書き換え
INSERT	相関名 INSERT INTO TABLE1 T1	相関名は削除
DELETE	FROM句の省略	FROM句を明記
その他	MERGE文	INSERT ON CONFLICT（すべて可能ではない）

- Oracleとの非互換

Oracle,PostgreSQL双方とも標準SQLに準拠しているため、SQLの非互換は多くはありません。詳細な対応はPGEConsの移行WGの成果物である「SQL移行調査編」を参考にしてください。

https://pgecons-sec-tech.github.io/tech-report/pdf/09_SqlMigrationResearch.pdf



- Oracleとの違い

Oracleでは一度実行されたSQL文の解析に使うデータや実行計画は共有プール(shared_pool_size)上にキャッシュされインスタンス内で共有されます。これは同一のSQLであれば既存の実行計画をすべてのセッションで再利用（④までの処理をパス）することが可能になります。PostgreSQLにはセッションを超えてSQLの解析結果を共有する機能はなく、基本的には毎回図の①パーサ～④プランナが実行されることになります。ただし、同一セッション内であればプリペアドステートメントを使用することで実行計画を再利用することは可能です。

- プリペアドステートメントの適用

ループ内で繰り返し実行されるSQLはほとんどの場合、実行計画を取得するまでの時間（Planning Time）は数ミリ秒以下です。

```
test01=# explain analyze select count(*) from pgbench_accounts where bid = 1;
                                                              QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=40019.22..40019.23 rows=1 width=8) (actual time=148.590..148.590 rows=1 loops=1)
 .
 .
Planning Time: 0.264 ms            ←実行計画を取得するまでの時間
 Execution Time: 148.805 ms
(10 行)
```

プリペアドステートメントで効果を得られるケースは毎回、実行計画を取得するまでに数十ミリ秒以上かかるような複雑なSQLが同一セッションで数千回以上実行されるような場合に数十秒の時間短縮ができるというものです。バッチ処理で数万回以上実行されるようなケースで大きな効果を得やすいです。

3. データベースの操作

3. 3　関数

- **組込み関数**
  - **あらかじめ用意されている関数**
    - **標準SQLで規定されていない拡張された関数が多数あります。**
  - **Oracleとの互換性**
    - **同じ関数名でも異なる動作をするものがあります。**
    - **orafceによりOracle互換関数の追加ができます。**
    - **よく使う関数の対応例**

関数	PostgreSQLでの対応	orafce	orafce備考
SUBSTR(char,m,n)	mが0またはマイナスの場合は個別対応	○	
SYSDATE	CURRENR_TIMESTAMP等で対応	○	()は必要(select sysdate();)
ADD_MONTHS(date,integer)	select DATE（CURRENR_TIMESTAMP）+ INTERVAL‘1 months’:のように対応	○	
DECODE(expr,serch,result,・・,result)	CASE文で書き換え	○	
NVL(expr,result)	COALESCE文で書き換え	○	0バイト文字はNULLとして扱わない

- Oracleとの互換性について

Oracleに存在しPostgreSQLに存在しない組み込み関数や同一名の組み込み関数名であっても動作が異なる場合があります。周辺ソフトウェアのorafceを導入することで比較的使用頻度の高い関数は追加もしくはOracleと同様な動作をさせることができます。詳細はPGECons　WG2成果物で「組み込み関数移行調査編」を参照するとよいでしょう。orafceを追加すると、下記のように関数が追加されます。

```
test02=# ¥df oracle.*
                                                関数一覧
 スキーマ |          名前          |       結果のデータ型        |                        引数のデータ型                        | 型
----------+------------------------+-----------------------------+--------------------------------------------------------------+-----
-
 oracle   | add_days_to_timestamp | timestamp without time zone | oracle.date, bigint                                          | 関数
 oracle   | add_days_to_timestamp | timestamp without time zone | oracle.date, integer                                         | 関数
 oracle   | add_days_to_timestamp | timestamp without time zone | oracle.date, numeric                                         | 関数
 oracle   | add_days_to_timestamp | timestamp without time zone | oracle.date, smallint                                        | 関数
 oracle   | add_months            | timestamp without time zone | timestamp with time zone, integer                            | 関数
 oracle   | btrim                 | text                        | character                                                    | 関数
 oracle   | btrim                 | text                        | character, character                                         | 関数
 oracle   | btrim                 | text                        | character, nvarchar2                                         | 関数
 .
 .
```

3. データベースの操作

3. 3 関数

- **ユーザ定義関数**
  - **使用方法は組込み関数と同様**
  - **関数と引数で識別されます。関数名でユニークではありません。**

```
test01=# ¥df tbl*
                                関数一覧
 スキーマ |  名前   | 結果のデータ型 |  引数のデータ型   |  型
----------+---------+----------------+-------------------+------
 public   | tbl_cnt | bigint         | integer           | 関数
 public   | tbl_cnt | bigint         | integer, integer  | 関数
(2 行)
```

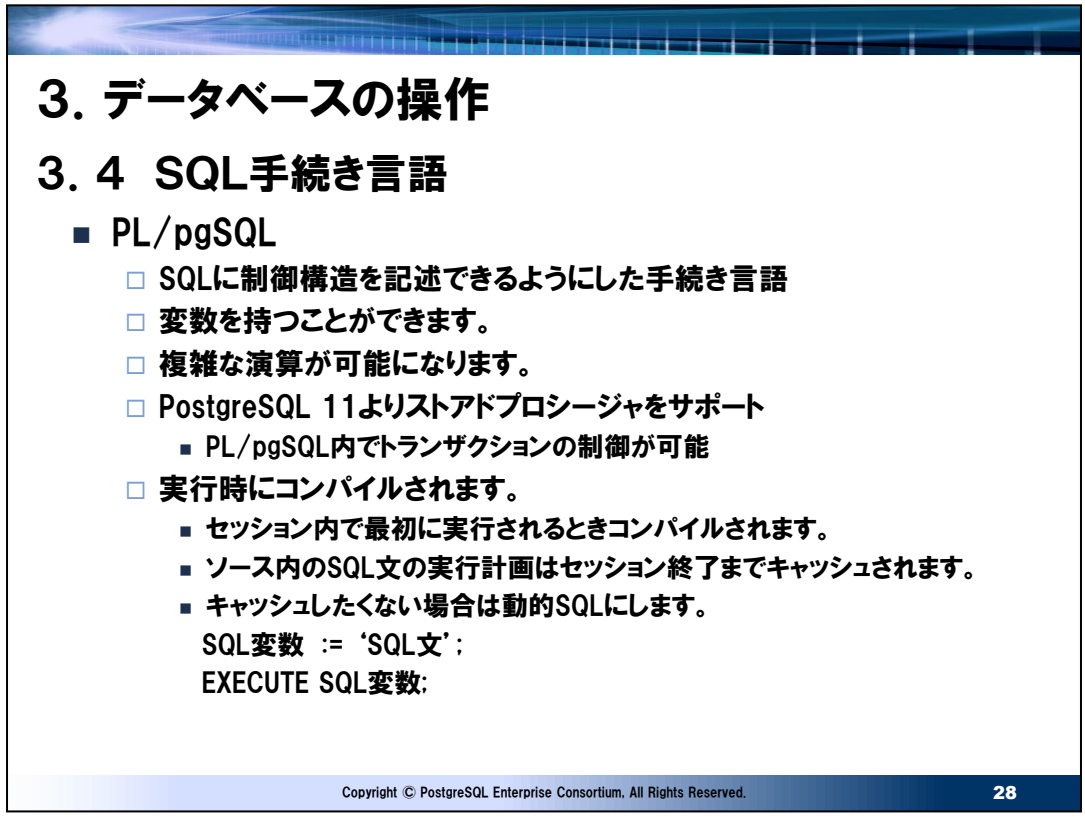
  - **一般的にはSQL、PL/pgSQL（SQLを拡張した手続き言語）が使用されます。**

- 定義方法

  CREATE [OR REPLACE] FUNCTION 関数名
  ([[引数のモード][引数名]引数のデータ型 [,‥]])
  [RETUERNS 戻り値の型]
  {LANGUAGE 言語 オプション AS ‥‥
  }

- 関数と引数

Oracleの関数はスキーマ内において関数名でユニークですが、PostgreSQLでは引数が異なれば重複できます。CREATE OR REPLACE FUNCTIONにより関数定義を変更できますが、引数を変更した場合は新規に別の関数が作成されることに注意する必要があります。



- OracleのPL/SQLとの違い

• パッケージの対応

PostgreSQL 11よりストアドプロシージャをサポートしましたが、パッケージは未サポートです。Oracleはユーザとスキーマを１：１で運用しますがPostgreSQLではそうではないため、スキーマ名をパッケージ名として関連する関数、プロシージャをまとめる方法が取れます。その他、詳細な対応はPGEConsの移行WGの成果物である「ストアドプロシージャ移行調査編」を参考にしてください。

https://pgecons-sec-tech.github.io/tech-report/pdf/wg2_StoredProcedureMigrationResearch_pg11.pdf

・ CREATE文でのエラーチェック

OracleではPL/SQLソースを詳細にチェックできる（実行時エラーとなるのは変数やテーブルのデータが異常な場合）のに比較してPL/pgSQLでは簡単な構文チェックまでしか行いません。例えば、以下のようにテーブルが存在しなくてもエラーにはなりません。

```
test02=# select * from test1;
ERROR:  リレーション"test1"は存在しません
行 1: select * from test1;
                    ^

test02=# CREATE PROCEDURE pg_test1(i_int in int)
test02-# LANGUAGE plpgsql
test02-# AS $$
test02$# BEGIN
test02$#         INSERT INTO test1 (a) VALUES (i_int);     ←存在しないテーブル
test02$#         COMMIT;
test02$# END
test02$# $$;
CREATE PROCEDURE  ←プロシージャが作成された
```

また、PL/pgSQLではセッション内で最初に実行されるときPL/pgSQLインタプリタによって解析されますが、このとき内部のSQL文までは解析されません。SQL文はその文が実行されるときにSPIマネージャによって解析されます。大きなPL/pgSQLで処理内の一部のSQL文しか実行されないケースを考えれば不要な解析オーバーヘッドを減らすメリットがありますが、移行時のテストはPL/pgSQL内部の全てのSQLを網羅するテストが必要になります。

3. データベースの操作

3. 5 トランザクション制御

■ 概要

□ PostgreSQLのトランザクション

- データベースにデータの変更を反映させる単位
- begin文で明示的に指定しない場合は１つのSQL文が１つのトランザクションとなります（デフォルト）。

状態	説明
トランザクション開始	begin文
トランザクション終了	commit文,end文 rollback文
トランザクション内のエラー	トランザクション全体がrollback＊
部分的なrollback	①savepoint セーブポイント名 ②rollback to セーブポイント名
自動コミット	明示的にbegin文を指定しない場合、SQL文の１文が１つのトランザクション

＊:rollback（エラー時はcommitでもrollbackとなる）しなければ次のSQL文を実行できません。

● 自動コミットのデフォルト

PostgreSQLでは自動コミットがデフォルトになります。Oracleと同様にするには自動コミットをオフにします。SQLスクリプトの移行時などは自動コミットをオフにするか、begin~endで明示的に指定します。

自動コミットをオフにする場合は以下のように実施します。この場合は最初のSQL文がトランザクション開始になります。

- psqlの場合　¥set AUTOCOMMIT offと入力する。またはpsql設定ファイル　~/.psqlrc に ¥set AUTOCOMMIT offを記述しておく。
- JDBCの場合　connection.setAutoCommit(false);にする。

● PostgreSQLでのOracleと異なるトランザクションの挙動について

- トランザクション内のエラーはトランザクション“すべて”がrollbackされる。
- トランザクション内でエラーが発生した場合はrollback以外できない。（commitの入力は可能ですが、結果はrollbackと同じ）
- DDLもトランザクション（ロールバックできます）。
- truncateもトランザクション（ロールバックできます）。
- psqlで自動コミットオフでトランザクションを開始しcommitせずに終了した場合はロールバックとなる（SQL*Plusの場合はcommit）。

3. データベースの操作

3. 5 トランザクション制御

■ トランザクション隔離レベル

□ PostgreSQLのトランザクション隔離レベル

- 変更は set transaction isolation level レベル; で可能
- SQL標準規格ではREPETABLE READはファントムリードを許容しますが、PostgreSQLでは発生しません。

レベル	ダーティリード	反復不能読み取り	ファントムリード	直列化異常	PostgreSQLの場合
READ UNCOMMITTED	発生する	発生する	発生する	発生する	指定してもREAD COMMITEDで動作
READ COMMITTED	発生しない	発生する	発生する	発生する	デフォルト
REPETABLE READ	発生しない	発生しない	発生しない	発生する	指定してもファントムリードは発生しない
SERIALIZABLE	発生しない	発生しない	発生しない	発生しない	指定可能

- Oracleのトランザクション隔離レベル

Oracleで指定可能なトランザクション隔離レベルはREAD COMMITTED(デフォルト) とSERIALIZABLEのみになります。どちらもPostgreSQLとは仕様差があり、複数のトランザクションが同時実行される場合に結果が異なるケースが存在します。詳細は 6.2 Oracleとの仕様差 の項を参照してください。

3. データベースの操作

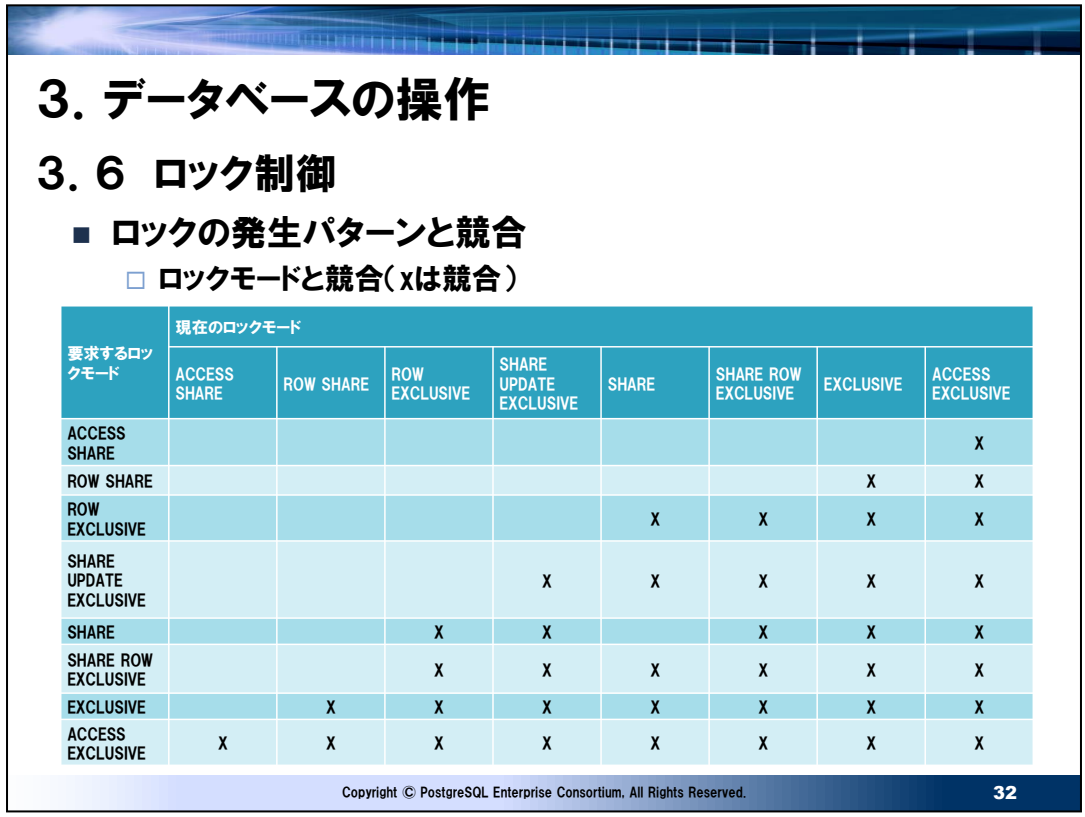
3. 6 ロック制御

- ロック制御
  - トランザクザクションの同時実行制御のために必要
  - 他のトランザクションでロックが獲得されていると解除されるまで待ちます。
- ロックの発生パターンと競合
  - ロックレベルと発生パターン

ロックレベル	発生パターン
ACCESS SHARE	SELECT
ROW SHARE	SELECT FOR UPDATE,SELECT FOR SHARE
ROW EXLUSIVE	UPDATE,DELETE,INSERT
SHARE UPDATE EXLUSIVE	VACUUM,ANALYZE,CREATE INDEX CONCURRENTLY,REINDEX CONCURRENTLY,CREATE STATISTICS, COMMENT ON, ALTER TABLE (一部) ,ALTER INDEX (一部)
SHARE	CREATE INDEX
SHARE ROW EXLUSIVE	CREATE TRIGGER,ALTER TABLE (一部)
EXCLUSIVE	REFRESH MATERIALIZED VIEW CONCURRENTLY
ACCESS EXLUSIVE	DROP TABLE,TRUNCATE,REINDEX,CLUSTER,VACUUM FULL, REFRESH MATERIALIZED VIEW, ALTER TABLE (一部) ,ALTER INDEX (一部) ,LOCK TABLE

- ロックモードについて

ロックレベル名にSHAREとEXCLUSIVEがあります。これはロックモードを示します。
SHARE（共有）：他のトランザクションからの書き換えは禁止するが参照を許可
EXCLUSIVE（占有または排他）：他のトランザクションからの書込みだけでなく参照も禁止



3. データベースの操作

3. 6 ロック制御

■ ロックの発生パターンと競合

□ ロックモードと競合（xは競合）

要求するロックモード	現在のロックモード							
	ACCESS SHARE	ROW SHARE	ROW EXCLUSIVE	SHARE UPDATE EXCLUSIVE	SHARE	SHARE ROW EXCLUSIVE	EXCLUSIVE	ACCESS EXCLUSIVE
ACCESS SHARE								X
ROW SHARE							X	X
ROW EXCLUSIVE					X	X	X	X
SHARE UPDATE EXCLUSIVE				X	X	X	X	X
SHARE			X	X		X	X	X
SHARE ROW EXCLUSIVE			X	X	X	X	X	X
EXCLUSIVE		X	X	X	X	X	X	X
ACCESS EXCLUSIVE	X	X	X	X	X	X	X	X

● ロックの確認方法

ロックはシステムカタログのpg_locksで確認することができます。以下は確認例。

```
test01=# SELECT
test01-#     pg_locks.pid AS "プロセスID"
test01-#   , pg_class.relname AS "ロック対象テーブル"
test01-#   , CASE WHEN pg_locks.granted is true THEN 'ロック中' ELSE 'ロック待ち' END AS "ロック状態"
test01-#   , pg_locks.mode AS "ロックレベル"
test01-#   , pg_stat_activity.state AS "トランザクション状態"
test01-#   , pg_stat_activity.query_start AS "クエリ開始時間"
test01-#   , pg_stat_activity.query AS "クエリ"
test01-# FROM  pg_locks
test01-#       INNER JOIN pg_stat_activity
test01-#             ON pg_locks.pid = pg_stat_activity.pid
test01-#       INNER JOIN pg_class
test01-#             ON pg_locks.relation = pg_class.oid
test01-# WHERE  pg_locks.pid <> pg_backend_pid();
 プロセスID |   ロック対象テーブル   | ロック状態 |   ロックレベル   | トランザクション状態 |        クエリ開始時間
|                        クエリ
------------+------------------------+------------+------------------+----------------------+-------------------------------
-+-----------------------------------------------------------
       1060 | pgbench_accounts2_idx1 | ロック中   | RowExclusiveLock | idle in transaction  | 2021-01-13 11:10:23.498157+09
| update pgbench_accounts2 set filler = '100' where aid = 301;
       1060 | pgbench_accounts2      | ロック中   | RowExclusiveLock | idle in transaction  | 2021-01-13 11:10:23.498157+09
| update pgbench_accounts2 set filler = '100' where aid = 301;
(2 行)
```

● ロックの強制解除

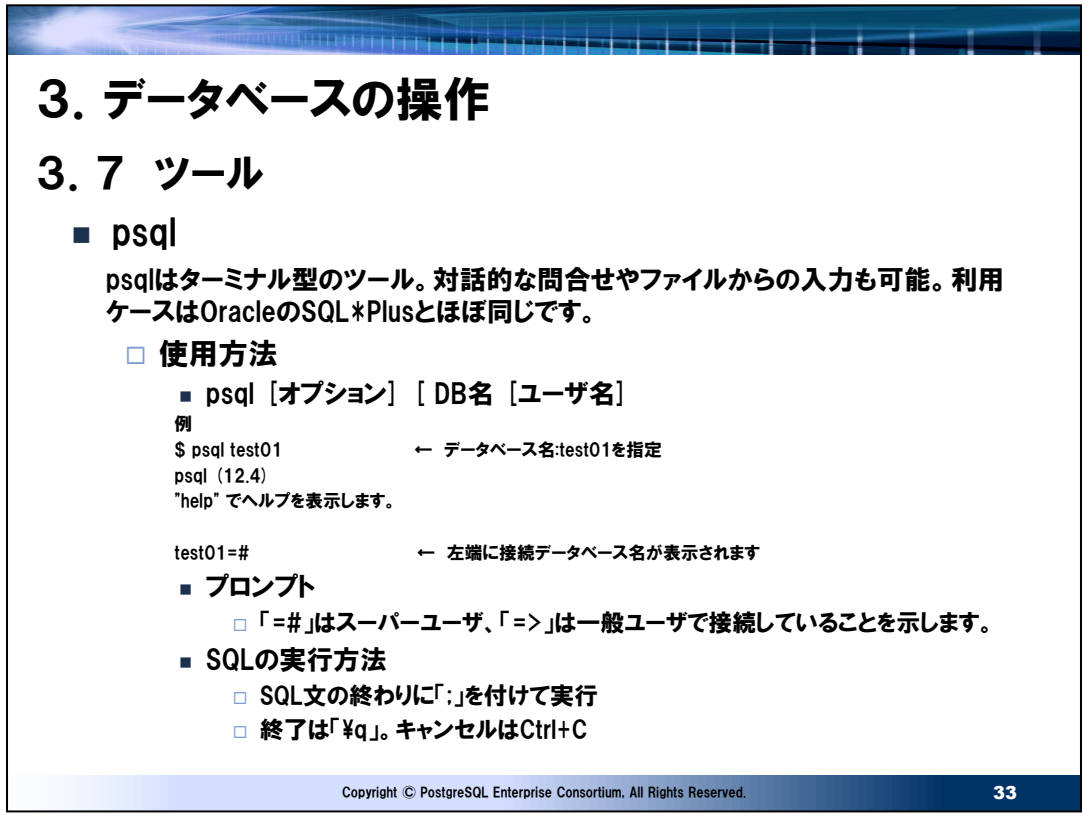
問題が発生した場合など、強制でロックを解除したい場合もあります。以下で強制的に解除することができます。

ロック発生元のクエリが実行中の場合

select pg_cancel_backend (プロセスID);

トランザクションブロック内にクエリがある場合などで上記コマンドが効かない場合

select pg_terminate_backend (プロセスID);



- バッチ的なpsqlの使用例

• スクリプトに変数を与えて実行

実行するスクリプト例

test.sql

```
insert into test values (:id , :test_text);
```

psqlでの実行例

```
psql –d DB名 –U ユーザ名 –f test.sql –v id=1 –v test_text="'テスト'"
```

-f：スクリプトファイル

-v：引数を指定する。

• 表をCSVファイルに出力

```
psql –d DB名 –U ユーザ名 -t -c "select * from pgbench_branches" -A -F , > pgbench_branches.csv
```

-t：ヘッダを省略（タプルのみとなる）

-A：桁揃えなしのテーブル出力モード

-F：フィールドの桁区切り文字

-c：コマンド実行

3. データベースの操作

3. 7 ツール

- psql
  - 使用方法(つづき)
    - メタコマンド

      psqlではSQLコマンドとメタコマンド(バックスラッシュコマンド。日本語ターミナルでは¥マーク)がある。メタコマンドはpsqlが独自に処理を行うもので、コマンドの終わりに「;」は不要です。内部的にはSQL文で実装されています。

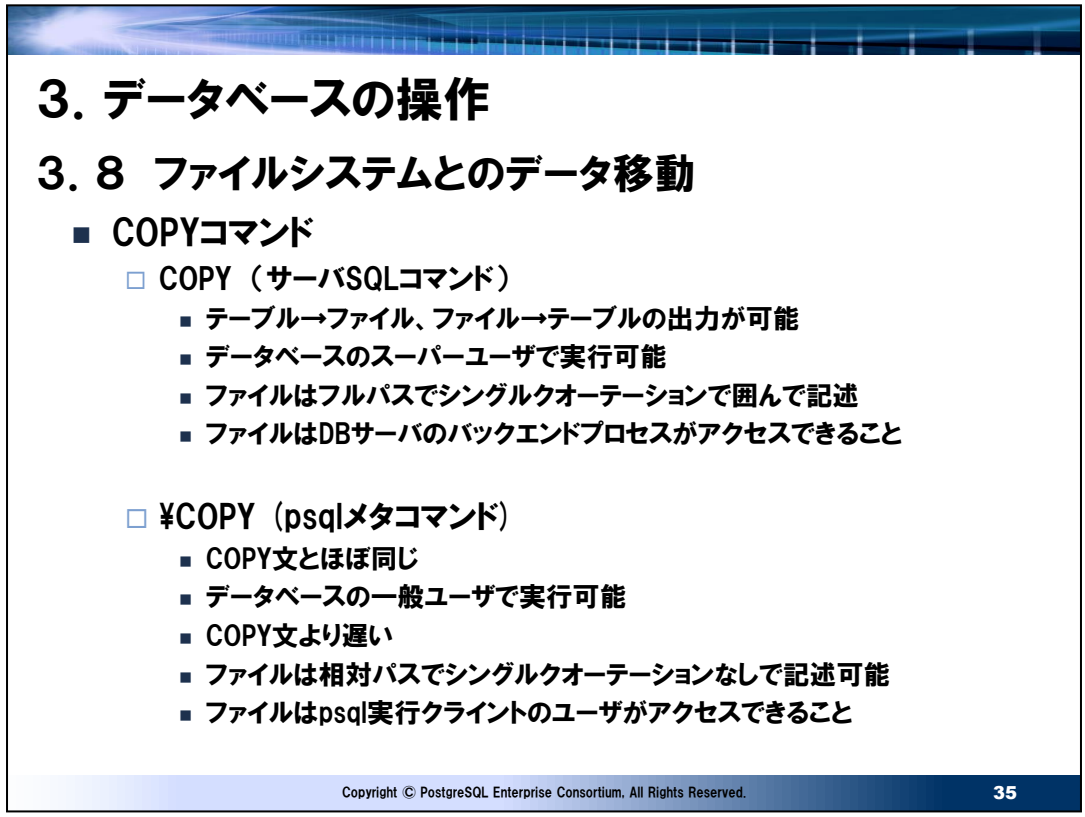
メタコマンド(例)	説明
¥?	コマンドの一覧を表示
¥dt	テーブル一覧を表示
¥d [オブジェクト名]	[ ]内のテーブル、インデックス、Viewを表示
¥d+ [オブジェクト名]	¥dの詳細表示
¥du	ロール(ユーザ)の一覧表示
¥dx	データベースにインストールされたエクステンションの表示
¥x	表示モードの変更。再度¥xで元に戻る

- 実行例

```
test01=# ¥dt+
                          リレーション一覧
 スキーマ |       名前       |    型    | 所有者 | サイズ | 説明
----------+------------------+----------+--------+--------+------
 public   | pgbench_accounts | テーブル | test01 | 131 MB |
 public   | pgbench_branches | テーブル | test01 | 168 kB |
 public   | pgbench_history  | テーブル | test01 | 552 kB |
 public   | pgbench_tellers  | テーブル | test01 | 384 kB |
(4 行)
```

このときの内部的に実行されたSQL文 (サーバログより抽出)

```
LOG:  duration: 17.669 ms  statement: SELECT n.nspname as "Schema",
          c.relname as "Name",
          CASE c.relkind WHEN 'r' THEN 'table' WHEN 'v' THEN 'view' WHEN 'm' THEN 'materialized view' WHEN 'i' THEN 'index'
WHEN 'S' THEN 'sequence' WHEN 's' THEN 'special' WHEN 'f' THEN 'foreign table' WHEN 'p' THEN 'partitioned table' WHEN 'I'
THEN 'partitioned index' END as "Type",
          pg_catalog.pg_get_userbyid(c.relowner) as "Owner",
          CASE c.relpersistence WHEN 'p' THEN 'permanent' WHEN 't' THEN 'temporary' WHEN 'u' THEN 'unlogged' END as
"Persistence",
          pg_catalog.pg_size_pretty(pg_catalog.pg_table_size(c.oid)) as "Size",
          pg_catalog.obj_description(c.oid, 'pg_class') as "Description"
        FROM pg_catalog.pg_class c
             LEFT JOIN pg_catalog.pg_namespace n ON n.oid = c.relnamespace
        WHERE c.relkind IN ('r','p','')
              AND n.nspname <> 'pg_catalog'
              AND n.nspname <> 'information_schema'
              AND n.nspname !~ '^pg_toast'
          AND pg_catalog.pg_table_is_visible(c.oid)
        ORDER BY 1,2;
```



- 実行例

• テーブル→ファイル

```
test01=# copy pgbench_accounts to '/home/postgres/pgbench_accounts.csv' (format csv);
COPY 1000000
test01=# copy (select * from pgbench_accounts where aid < 1000)
test01-# to '/home/postgres/pgbench_accounts2.csv' (format csv);
COPY 999
```

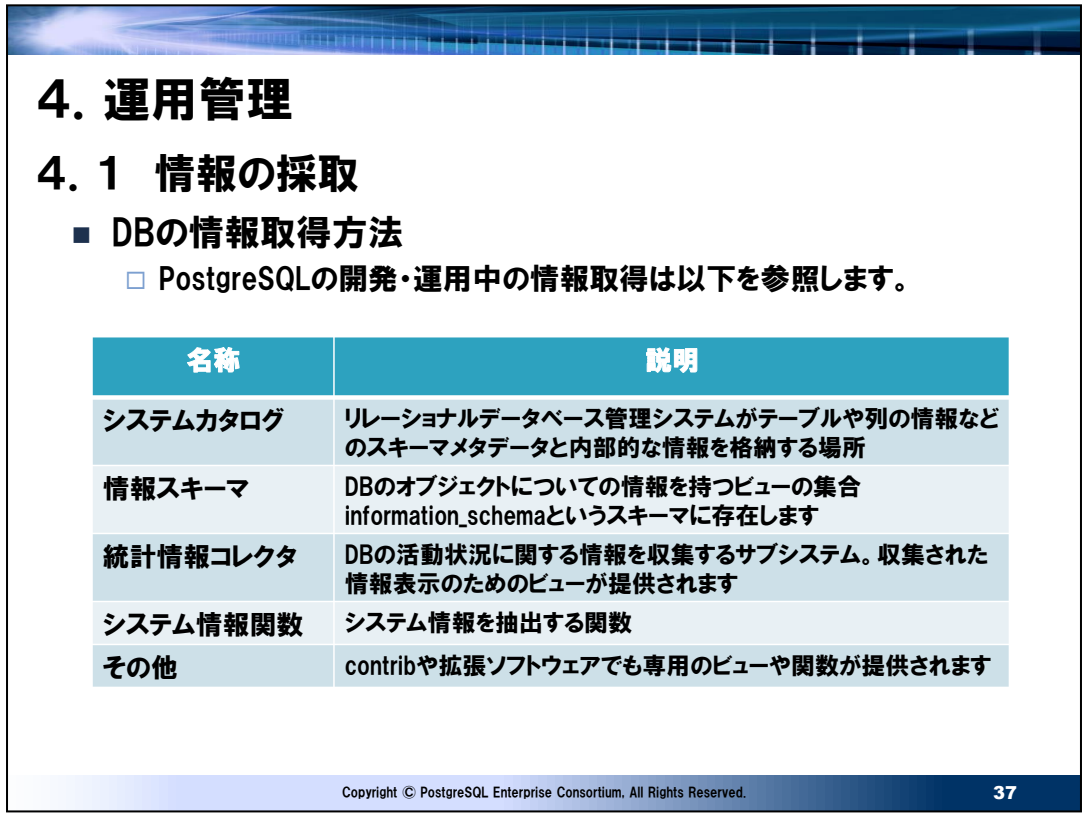
• ファイル→テーブル

```
test01=# copy pgbench_accounts from '/home/postgres/pgbench_accounts.csv' (format csv);
COPY 1000000
```

- エラーが発生する場合

COPYでエラーが発生すると、発生時点で処理は終了します。特にファイルからデータを移入しようとしたとき、OracleのSQL*Loaderのように異常データをbatファイルに隔離して処理を継続することはできません。大量データで多数の異常データが内在する場合は１件毎に異常検出-対処となります。これを回避するには事前に移入元データをクレンジングするか別途、周辺ソフトウェアのpg_bulkloadを導入し対応することになります。

4. 運用管理



● DBの情報取得方法

PostgreSQLはデータベースに関する管理メタデータの情報をいくつかの方法で提供しています。Oracleでのデータディクショナリと同様と理解してよいでしょう。運用面ではこれらを参照することでDBの運用状況を確認できます。それぞれのテーブルやビュー、関数の詳細は利用するバージョンの「PostgreSQL文書」を参照してください。

以下にシステムカタログ、統計情報コレクタの抽出例を示します。

➢ システムカタログ

一部、システムビューとしての提供もありますが、多くは通常のテーブルです。ユーザが手作業で更新してはいけません。

使用例：テーブルの一覧を表示

以下のpg_class,pg_namespaceはシステムカタログです。

```
test01=# select pn.nspname,pc.relname,pc.relkind
test01-# from pg_class pc
test01-#  inner join pg_namespace pn
test01-#  on pc.relnamespace = pn.oid;
      nspname       |                   relname                    | relkind
--------------------+----------------------------------------------+---------
 public             | pg_stat_statements                           | v
 public             | pgbench_accounts                             | r
 public             | pgbench_branches                             | r
 public             | pgbench_tellers                              | r
 public             | pgbench_branches_pkey                        | i
・
```

➢ 統計情報コレクタ

情報の収集の定義はpostgresql.confのパラメータや専用の関数で指定できます。ほとんどの場合、デフォルトのままでも運用できますが、システムの特性に応じて変更します。収集した情報は動的統計情報ビューや収集済み統計情報ビューを利

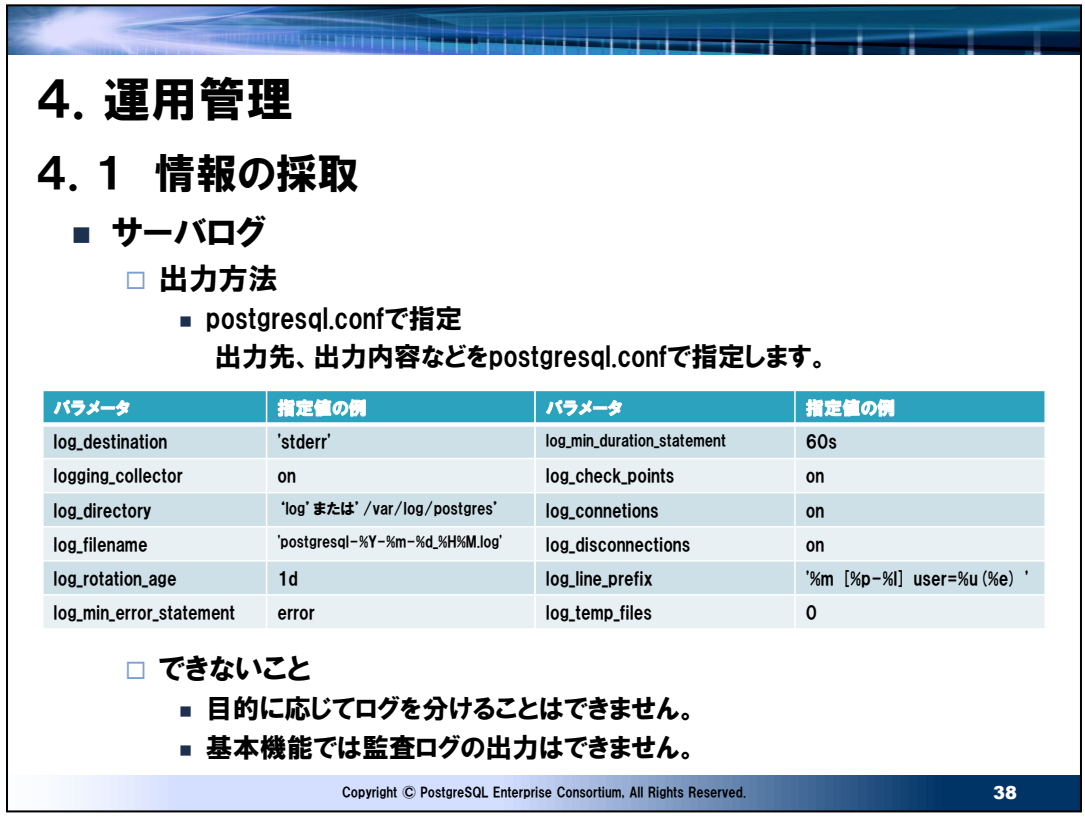
用して参照します。動的統計情報ビューは現在の状態を表示し、収集済み統計情報ビューはリセットしてからの累積値を表示します。累積値のリセットはpg_stat_reset()関数で行います。

使用例１：現在実行中のプロセスを表示
以下のpg_stat_activityは動的統計情報ビューです。

```
test01=# select * from pg_stat_activity
test01-# where state = 'active';
-[ RECORD 1 ]----+-------------------------------
datid            | 16984
datname          | test01
pid              | 1450
usesysid         | 10
usename          | postgres12
application_name | psql
client_addr      |
client_hostname  |
client_port      | -1
backend_start    | 2021-03-23 10:30:03.953349+09
xact_start       | 2021-03-23 15:22:55.082448+09
query_start      | 2021-03-23 15:22:55.082448+09
state_change     | 2021-03-23 15:22:55.082458+09
wait_event_type  |
wait_event       |
state            | active
backend_xid      |
backend_xmin     | 2041355
query            | select * from pg_stat_activity+
                 | where state = 'active';
backend_type     | client backend
・
```

使用例２：索引利用状況の確認（使用回数が多い順）
以下のpg_stat_user_indexesは収集済み統計情報ビューです。

```
test01=# select * from pg_stat_user_indexes
test01-# order by idx_scan desc;
-[ RECORD 1 ]-+----------------------
relid         | 16991
indexrelid    | 17005
schemaname    | public
relname       | pgbench_accounts
indexrelname  | pgbench_accounts_pkey
idx_scan      | 16001
idx_tup_read  | 23127
idx_tup_fetch | 16000
・
```



● サーバログ出力関連のパラメータ

スライドのパラメータは一部です。パラメータとその指定内容の説明は「PostgreSQL文書」を参照してください。以下にスライド上のパラメータを開発・運用面から説明します。

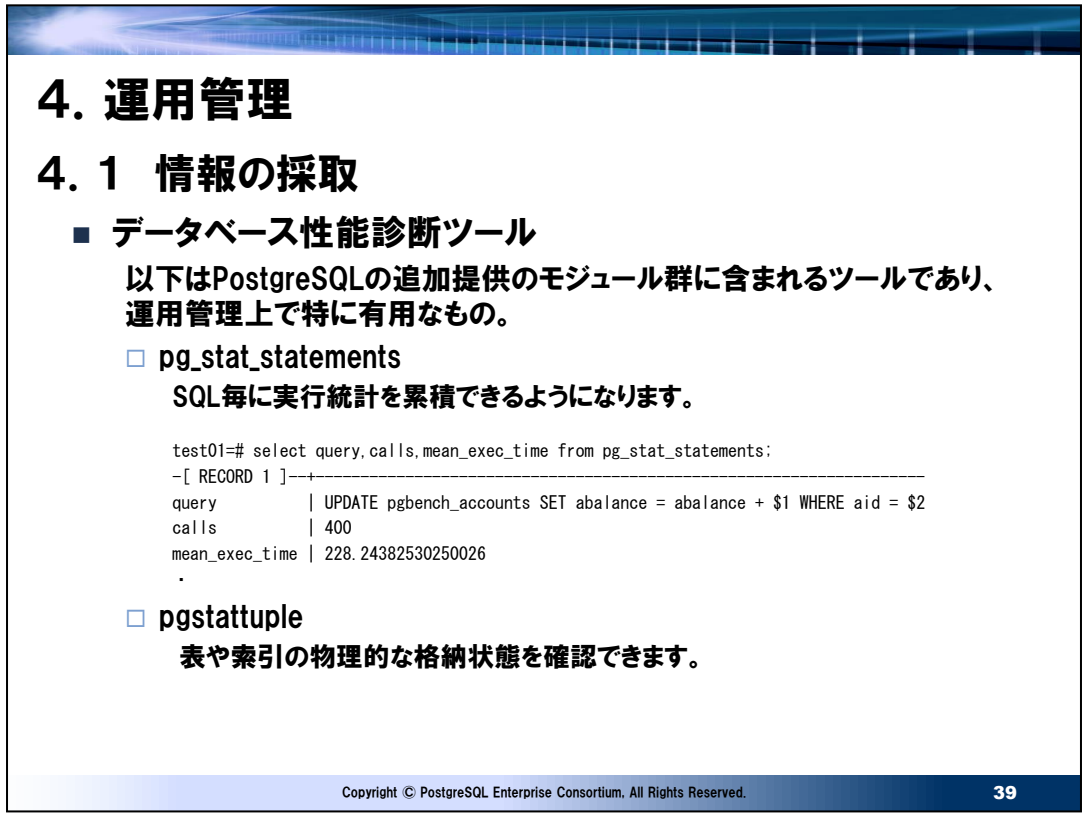
- log_min_duration_statement：実行時間が長いSQLを簡単に把握するときに指定します。対策が必要な場合はcontribのauto_explainを利用しSQL実行時点の実行計画をログに出力することで正確な対処が実施しやすくなります。オンライン処理のように1回あたりの実行時間は短いが、実行回数が多く負荷の高いSQLは同じくcontribのpg_stat_statementsを使用して分析します。
- log_checkpoints：チェックポイントの発生要因（時間、WAL領域のサイズ不足）、書出しバッファ数などの統計情報が出力されます。チューニングの参考情報に使います。
- log_connetions：サーバへの接続試行の証跡を残します。運用によってはログサイズが非常に大きくなる場合もありますが、エラーの証跡など重要な情報を残せます。
- log_disconnections：サーバへの切断情報だけでなく、セッションの接続時間も表示します。時間帯別の平均接続数、平均接続時間など分析できるようになります。運用によってはログサイズが非常に大きくなる場合もあります。
- log_line_prefix：ログの行ヘッダ部分の編集方法を指定します。発生時刻、同時に複数のプロセスが動作しているので区別できるようにプロセスID、エラーコードを確認できるようにSQLSTATEなど、調査を円滑にできるように定義します。
- log_temp_files：ここを0にすることで一時ファイルを使用したらすべてログに記録されます。その結果からwork_memパラメータの調整をします。サーバのメモリを枯渇させないことを前提にオンライン処理であれば基本一時ファイルは使わないように、バッチ処理であれば実行前にセッション個別にwork_memを指定します。

● 監査要件の対応

監査は"いつ"、"誰が"、"どこから"、"どのオブジェクトに対して"、"どんな処理を行い"、その"実行結果"を収集する必要がありますが、基本機能では監査機能がないため周辺ソフトウェアのpgAuditが必要になります。ただし、そのpgAuditも本体のサーバログと監査ログの分離ができないため、さらに別の周辺ソフトウェアであるpgAudit Log to Fileが必要になります。高度な監査要件がある場合は商用版のPostgreSQLや監査ツールを検討したほうが良い場合もあります。

● Oracleのサーバログとの比較

Oracleのログがアラートログ、監査ログ、リスナーログ、トレースファイルなどに使用目的によって分かれているのに対し、PostgreSQLはシンプルに 1 種類のログのみになります。Oracleのように何も指定しなくても必要なログが取得できるわけではありません。適切にログが出力されるようにパラメータで調整する必要があります。



- pg_stat_statements

利用するにはエクステンションの追加とパラメータ：shared_preload_librariesに'pg_stat_statements'を追加しインスタンスを再起動する必要があります。提供される情報は以下の通りであり、OracleのV$SQLと似た機能と考えればよいでしょう。

- ユーザID
- データベースID
- SQL文　バインド変数は$1,$2,,,、定数は?で表示されます。
- 実行回数
- 合計実行時間
- 最短の実行時間
- 最長の実行時間
- 平均実行時間

他

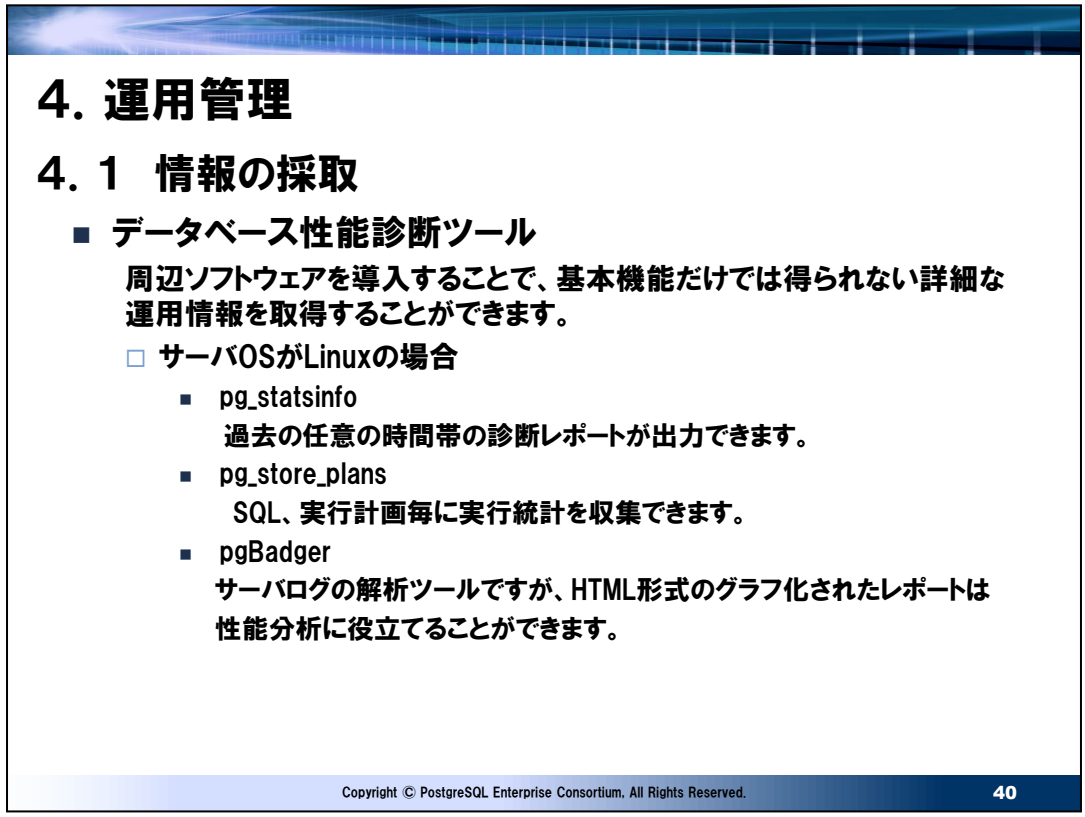
- pgstattuple

pgstattupleエクステンションを追加することで利用可能になります。使用用途として、性能問題が発生している場合に表や索引の物理的な格納状態に問題がないか確認できます。具体的には表のVACUUM FULL、索引の再構築の必要性を分析できます。ただし、情報の取得にはそれぞれのオブジェクトの全領域をアクセスするため、オブジェクトサイズに依存した時間がかかります。このためテーブルのみ可視性マップを利用して高速に推定値を取得することもできます。

実行例

```
test01=# select * from pgstattuple('pgbench_accounts');
-[ RECORD 1 ]------+----------
table_len          | 136708096      ←表のサイズ（バイト）
tuple_count        | 1000000        ←有効行数
tuple_len          | 121000000      ←有効行の合計サイズ（バイト）
tuple_percent      | 88.51          ←表サイズに対する行の割合（%）
dead_tuple_count   | 216            ←無効行数
```

```
dead_tuple_len     | 26136          ←無効行の合計サイズ
dead_tuple_percent | 0.02           ←表に対する無効行の割合（%）
free_space         | 4008952        ←表の空き領域（バイト）
free_percent       | 2.93           ←空き領域の割合（%）

test01=# select * from pgstatindex('pgbench_accounts_pkey');
-[ RECORD 1 ]------+---------
version            | 4
tree_level         | 2              ←B-treeの階層の深さ
index_size         | 22487040       ←索引のサイズ
root_block_no      | 290
internal_pages     | 11             ←ブランチページ（ブロック）の数
leaf_pages         | 2733           ←リーフページ（ブロック）の数
empty_pages        | 0              ←空きページ（ブロック）数
deleted_pages      | 0              ←削除ページ（ブロック）数
avg_leaf_density   | 90.06          ←平均リーフ密度（%）
leaf_fragmentation | 0              ←リーフ断片化率（%）
```



- 運用におけるデータベース性能診断ツール

• pg_statsinfo

性能問題が発生したときなど、正常時と異常時のレポートを取得し比較することで原因の特定が行いやすいツールです。Oracle技術者であればStatsPackを連想すると分かりやすいです。動作もサーバログと定期的に統計情報コレクタやpg_stat_statementsのスナップショットを取得し、そのスナップショット間の差分からレポートを作成します。また、レポートされる内容もStatsPackに近いものがレポートされます。作成されるスナップショットの管理など、運用作業をpostgresql.confに指定することで自動化できるため運用コストを増加させることもありません。詳細は以下マニュアルを確認してください。

http://pgstatsinfo.sourceforge.net/index_ja.html

• pg_store_plans

機能は過去に実行されたSQLの実行計画とその実行計画毎の実行統計を知ることができます。実行計画毎の統計情報はどの実行計画が優れているかを評価できるため、チューニング情報としては非常に有効なものとなります。デメリットとしてはオンライン処理のように短時間に大量のSQLが実行される環境ではpg_store_plansの情報を取得するためのオーバーヘッドも高くなります。対策としてはpostgresql.confで指定するパラメータのpg_store_plans.min_durationにより取得閾値を0より大きな値で調整することで、情報の出力と運用とのバランスを取る必要があります。また、別途周辺ソフトウェアであるpg_statsinfoと併用することで結果をレポートに含めることができるようになります。任意の時間帯でのSQLの実行計画毎の実行統計が確認できるため、SQLで性能悪化した場合の原因が特定しやすくなります。このレポートはpg_stats_reporterを利用することで効率よく確認できるようになります。インストール、指定方法も含め、マニュアルは以下を参照してください。

http://ossc-db.github.io/pg_store_plans/

[分析例]

実行計画の違いで平均応答時間の違いを確認できる例

※pg_store_plans単体ではqueryidのみでSQL文が不明なためpg_stat_statementsと合わせて問い合わせています。

```
test01=# select pss.queryid,pss.query,psp.planid,psp.plan,psp.calls,psp.total_time,psp.mean_time,psp.rows
test01-# from pg_store_plans as psp,pg_stat_statements as pss
test01-# where psp.queryid_stat_statements = pss.queryid
test01-# and pss.queryid = '-4268911728071129898'
test01-# order by total_time desc;
```

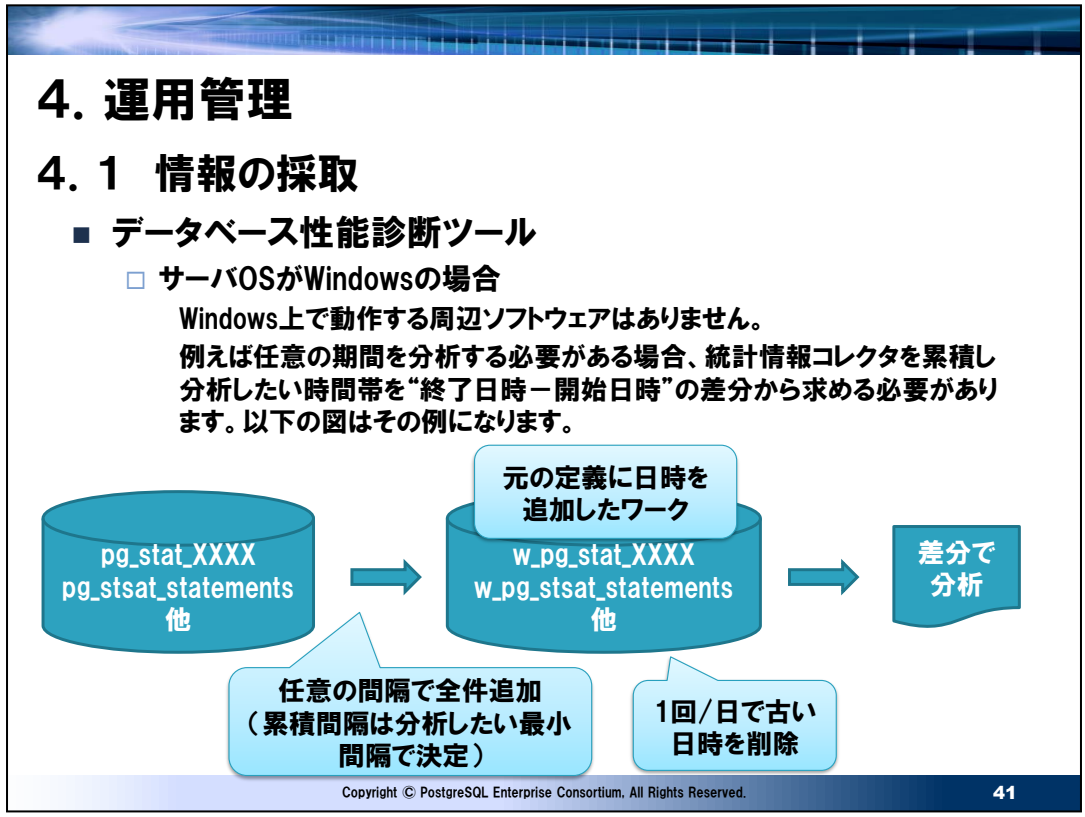
```
-[ RECORD 1 ]----------------------------------------------------------------------------------------------------------
queryid    | -4268911728071129898
query      | UPDATE pgbench_accounts SET abalance = abalance + $1 WHERE aid = $2
planid     | 4152249497
plan       | Update on pgbench_accounts  (cost=0.00..29410.40 rows=1 width=103)                                      +
           |   ->  Seq Scan on pgbench_accounts  (cost=0.00..29410.40 rows=1 width=103)                              +
           |         Filter: (aid = 850018)
calls      | 100
total_time | 90031.83445400001
mean_time  | 900.3183445399999
rows       | 100
-[ RECORD 2 ]----------------------------------------------------------------------------------------------------------
queryid    | -4268911728071129898
query      | UPDATE pgbench_accounts SET abalance = abalance + $1 WHERE aid = $2
planid     | 4250722148
plan       | Update on pgbench_accounts  (cost=0.42..8.45 rows=1 width=103)                                          +
           |   ->  Index Scan using pgbench_accounts_pkey on pgbench_accounts  (cost=0.42..8.45 rows=1 width=103)+
           |         Index Cond: (aid = 444910)
calls      | 300
total_time | 1265.695666999999
mean_time  | 4.218985556666666
rows       | 300
```

上記では同一のSQL文であっても実行計画が異なれば平均実行時間に差があることがわかります。

注意点：calls,rowsはSQL文毎であり、実行計画毎ではないことに注意すること。

- pgBadger

データベースの運用状況を俯瞰するのに便利なツールです。インプットはサーバログになりますので詳細にサーバログを記録することで詳細なレポートを出力できます。例えばSQLの分析を行うにはすべてのSQLの記録が必要となります。このため運用ではログの容量が数GB/日（もしくはそれ以上）になることが多いです。ログファイルが大きいとレポート作成も時間がかかるため一部のログは抑止するなどの検討が必要な場合もあります。ログ出力抑止した場合はpgBadgerのレポートではNO DATASETとなって空のレポートが出力されます。



- Windowsでの性能診断ツール

Windows版PostgreSQLではpg_statsinfoのような過去の任意の期間を分析するツールはありません。統計情報コレクタやpg_stat_statementsはリセットから最新の状態まで累積するので、リセット（分析開始時刻）と最新の時点（分析終了時刻）の分析を繰り返すか、スライドのように統計情報コレクタやpg_stat_statementsを定期的に累積するようなツールを自作し、分析時に終了時刻と開始時刻の統計値の差分からする必要があります。

4. 運用管理

4. 1　情報の採取

■ データベース性能診断ツール

□ サーバOSがWindowsの場合

pg_stat_statementsを累積した場合の分析例

分析開始日時	分析終了日時	ユーザ名	クエリID	クエリ	実行回数	実行時間(ms)	平均実行時間(ms)	影響受けた行数
2019-09-10 18:00:00+09	2019-09-10 19:00:00+09	postgres	831879657	UPDATE pgbench_tellers SET tbalance = tbalance + $1 WHERE tid = $2	5,000	145,593.28	29.12	5,000
2019-09-10 18:00:00+09	2019-09-10 19:00:00+09	fsepuser	831879657	UPDATE pgbench_tellers SET tbalance = tbalance + $1 WHERE tid = $2	5,000	141,064.75	28.21	5,000
2019-09-10 18:00:00+09	2019-09-10 19:00:00+09	postgres	3872086279	UPDATE pgbench_branches SET bbalance = bbalance + $1 WHERE bid = $2	5,000	138,750.83	27.75	5,000
2019-09-10 18:00:00+09	2019-09-10 19:00:00+09	fsepuser	3872086279	UPDATE pgbench_branches SET bbalance = bbalance + $1 WHERE bid = $2	5,000	115,700.57	23.14	5,000
2019-09-10 18:00:00+09	2019-09-10 19:00:00+09	postgres	3371683456	UPDATE pgbench_accounts SET abalance = abalance + $1 WHERE aid = $2	5,000	1,243.99	0.25	5,000
2019-09-10 18:00:00+09	2019-09-10 19:00:00+09	fsepuser	3371683456	UPDATE pgbench_accounts SET abalance = abalance + $1 WHERE aid = $2	5,000	879.24	0.18	5,000

分析開始日時～分析終了日時間での発生量を実行時間が長い順で表示

- 分析例

pg_stat_statementsを１時間毎に累積するw_pg_stat_statementsを作成し、スライドのように開始～終了時刻の間の実行統計を表示するSQLを示します。

```
SELECT
S.time as “分析開始日時”,
E.time as “分析終了日時”,
PA.rolname as "ユーザ名",
E.queryid as "クエリID",
regexp_replace(E.query, '¥r¥n|¥r|¥n', '','g') as "クエリ",
E.calls - COALESCE(S.calls,cast(0 as bigint)) as "実行回数",
E.total_time - COALESCE(S.total_time,cast(0 as double precision)) as "実行時間(ms)",
(E.total_time - COALESCE(S.total_time,cast(0 as double precision)))
/ (E.calls -COALESCE(S.calls,cast(0 as bigint))) as "平均実行時間(ms)",
E.rows - COALESCE(S.rows,cast(0 as bigint)) as "影響受けた行数"
FROM w_pg_stat_statements E
LEFT OUTER JOIN w_pg_stat_statements S
ON TO_CHAR(S.time,'YYYYMMDD HH24') = TO_CHAR(E.time - interval '1 hour' ,'YYYYMMDD HH24') AND
E.queryid = S.queryid AND
E.userid = S.userid
INNER JOIN pg_authid PA
ON E.userid = PA.oid
WHERE
TO_CHAR(E.time, ‘YYYYMMDD HH24’) BETWEEN ‘開始日時’ AND ‘終了日時’ AND
(E.calls -COALESCE(S.calls,cast(0 as bigint))) > 0
ORDER BY E.time,"実行時間(ms)" desc
```

4. 運用管理

4. 2 バックアップ・リストア

- 目的
  - ディスク故障や操作ミスによるデータロストなど不測の事態への対応
  - メジャーバージョンアップ時、移行時のデータの移出・移入
  - テスト環境等へのデータの移出・移入　など
- 方式

方式	方法	バックアップ単位	リカバリ範囲	DB停止
コールドバックアップ	OSのコピーコマンド	データベースクラスタ	バックアップ時点	必要
ホットバックアップ（論理バックアップ）	pg_dump	データベース、スキーマ、テーブル		不要
	pg_dumpall	データベースクラスタ		不要
継続アーカイブ＋PITR (Point In Time Recovery)	pg_basebackup+アーカイブWAL運用	データベースクラスタ	最新の確定したトランザクション＊	不要
	pg_rman (linux版のみ)＋アーカイブWAL運用			

＊：最新のWALが破損していない場合（最新のWALはアーカイブされていないため必要です）

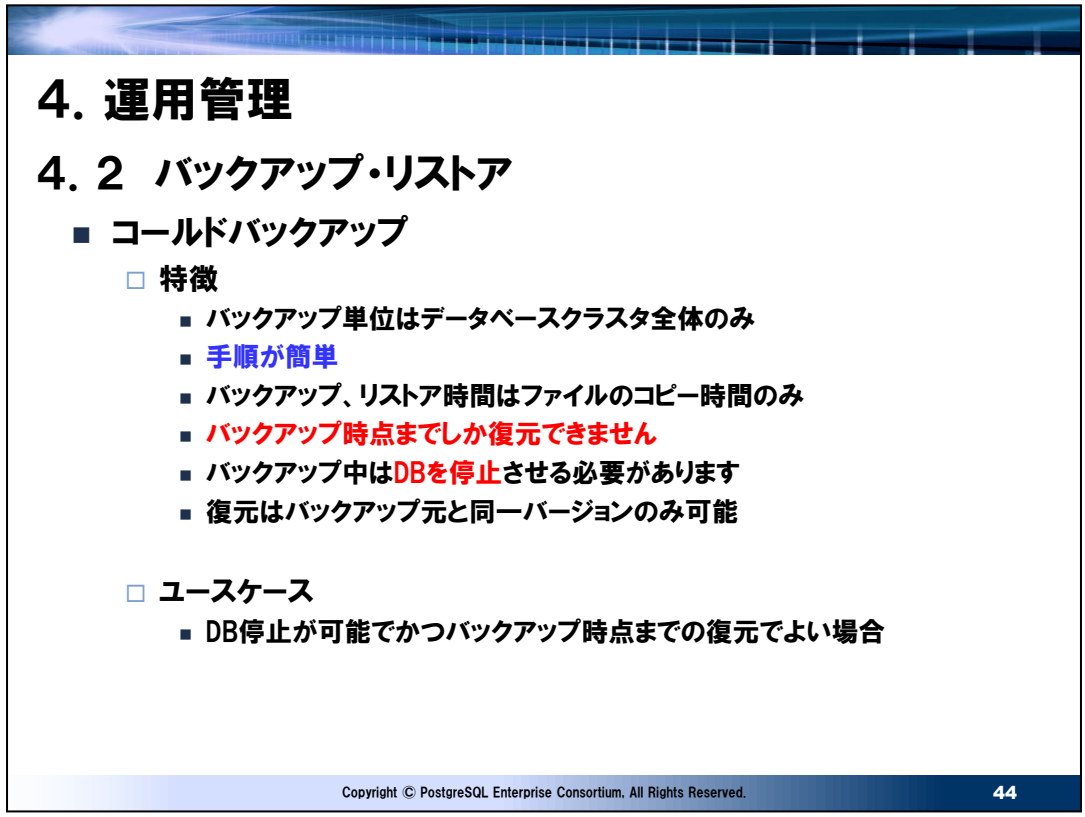
- PostgreSQLのバックアップ

PostgreSQLのデータのバックアップには３つの方式があります。それぞれの方式には特徴がありユースケース含めて後述します。スライドのpg_rmanのみ周辺ソフトウェアの追加が必要になります。

- クラウド上のデータベースサービスのバックアップ

詳細は各クラウドサービスのバックアップを参照してください。基本的にはAWS,Azure共に以下のようになります。

- コールドバックアップは不可
- ホットバックアップは可能。ただし、スーパーユーザ権限を持たないため、ロールに対し権限付与等が必要なケースもある。
- PITRバックアップはそれぞれ独自の方式で用意されています。また、復元可能なRPO（Recovery Point Objective)も異なります。



● コールドバックアップ・リストア例

コールドバックアップは手順が簡単なため実際の商用運用でもよく使われるバックアップです。特にデータベースディレクトリ内で完結するような小規模なデータベースでバックアップ時点までの復元でよく、かつ保守時間を確保できるのであれば採用率は高いです。

➢ バックアップ例

```
$ pg_ctl stop
サーバ停止処理の完了を待っています....完了
サーバは停止しました
$ cp -pr $PGDATA DB_backup_`date "+%Y%m%d"`
```

➢ リストア例

ディスク障害などでデータベースを構成するファイルが破損し、インスタンスが既に停止していると仮定した復旧手順になります。

①破損したデータベースの残骸を削除

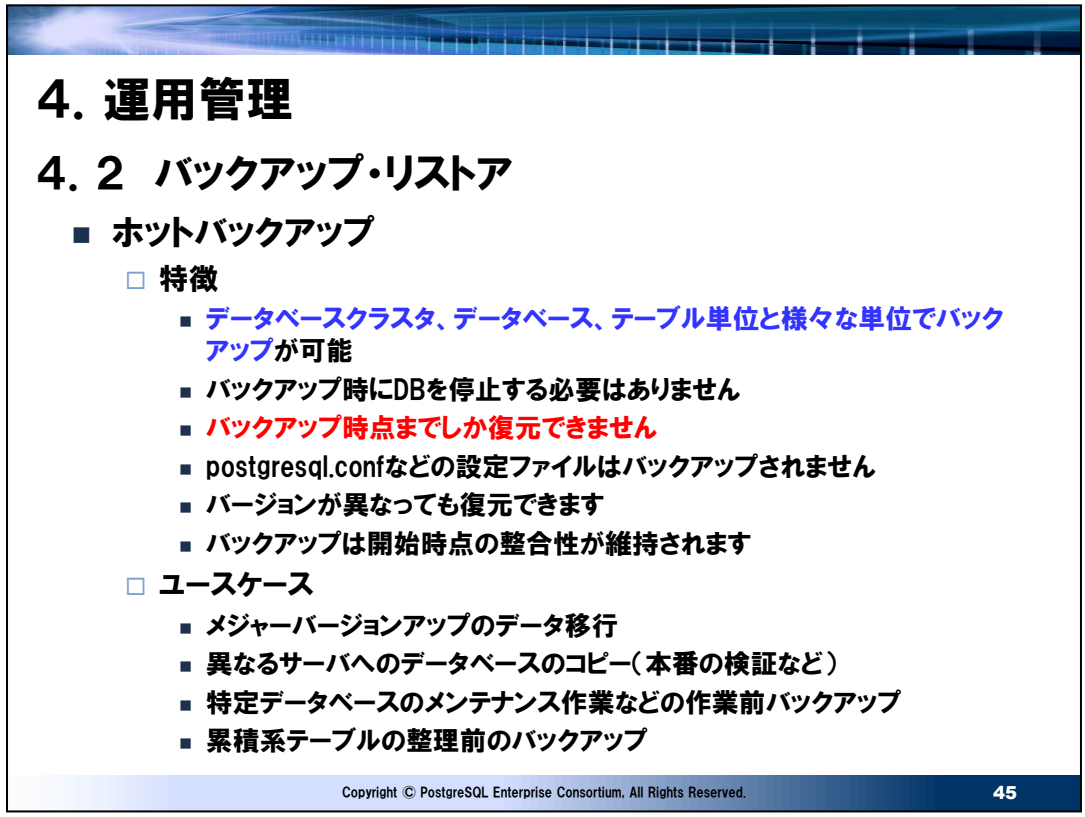
```
$ rm -r $PGDATA
```

②バックアップの復元

```
$ cp -pr DB_backup_20210225 $PGDATA
```

③PostgreSQLの起動

```
$ pg_ctl start
サーバの起動完了を待っています....
・
完了
サーバ起動完了
```



● ホットバックアップ・リストア例

データベースを停止する必要がなく、任意の単位でバックアップ・リストアができることから、保守作業や開発、検証のためのデータ移動など作業系に利用されるバックアップです。データベースクラスタ全体はpg_dumpallコマンド、データベース以下はpg_dumpでバックアップします。ユースケースはOracleのData Pumpと同様と考えてよいでしょう。

➢ バックアップ例

• DBクラスタ全体

```
$ pg_dumpall -c -v -f dumpall.sql > dumpall.log 2>&1
```

上記は-cでバックアップスクリプトにクリーンアップ（削除）コマンドを追加し、-vでメッセージを出力しそれをログファイルにリダイレクトしています。-fはバックアップファイル名です。

• データベースのバックアップ

```
$ pg_dump -Fc -c -d test02 -f test02.dmp
```

上記では-Fcでカスタム形式のフォーマット、-dでバックアップするデータベースを指定しています。

➢ リストア例

• DBクラスタ全体

```
$ psql -f dumpall.sql
```

• データベースのリストア

```
$ pg_restore -c -d test02 -f test02.dmp
```

上記は-cで再作成前にデータベースオブジェクトの整理をしています。（既に対象オブジェクトが存在しない場合は不要なエラーを出力しますが無視してもかまいません。）

• テーブルのリストア

pg_dumpでは上位のバックアップ単位で取得したバックアップから個別のオブジェクトのリストアが可能です。（フォーマットが平文の場合はスクリプトを手動で調整）

以下ではバックアップ例のDBで取得したデータベースのバックアップからテーブルを２つだけ復元しています。

```
[postgres@edu-011 ~]$ pg_restore -d test02 -n public -t pgbench_accounts -t pgbench_history -c -v
test02.dmp
```

上記では-nでスキーマを指定し、-tで復元するテーブルを指定しています。

注意点として、バックアップデータに復元対象のデータが存在しない場合、なにもせずに終了します。pg_restoreはエラーに

なることはありません。

- pg_dumpの出力形式

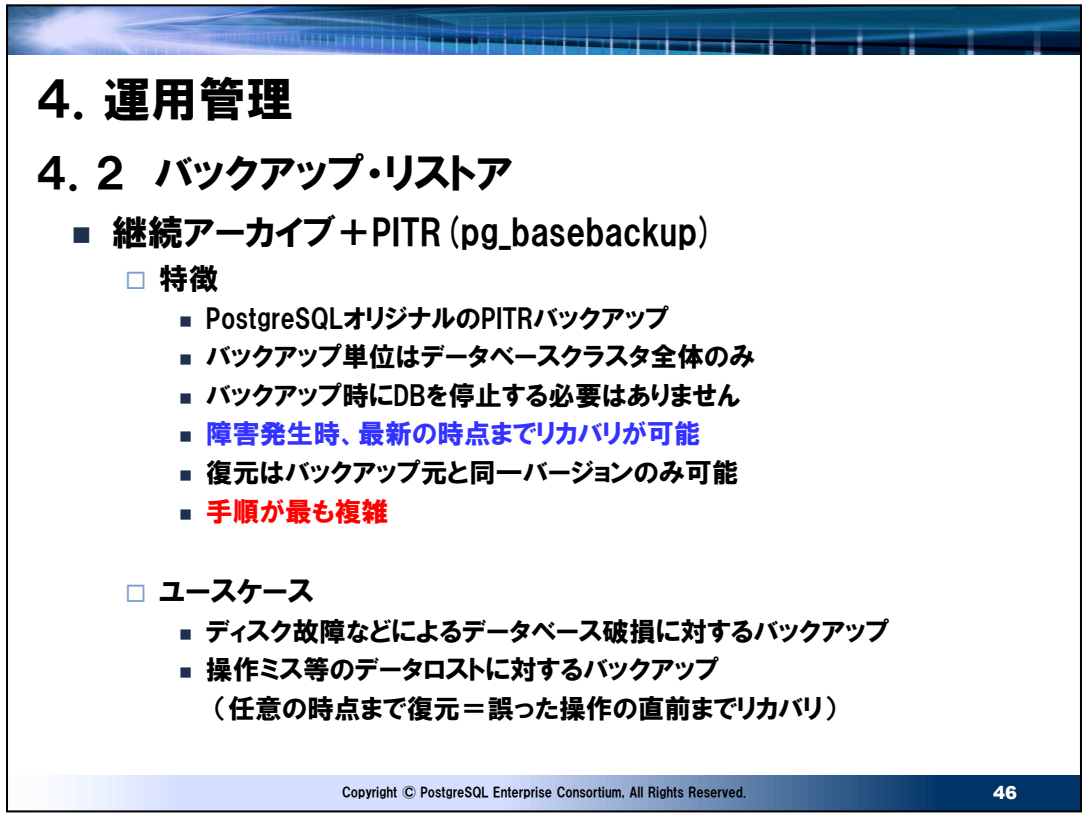
pg_dumpには出力フォーマットが指定することが可能であり、フォーマットによってリストア方法も異なります。

- 平文のSQLファイル
- カスタム形式
- ディレクトリ形式
- tar形式

上記の平文のSQLファイルのみpsqlのスクリプト実行によりリストアしますが、それ以外はpg_restoreコマンドで復元します。
なお、ディレクトリ形式のみバックアップ、リストア時にパラレル実行（１つのオブジェクトの処理をパラレル化するのではなく、複数のオブジェクトを同時に処理を行えるという意味）することができます。

- ホットバックアップのダンプデータ内の整合性

ホットバックアップはデータベースがオンライン中に実施します。そのため、バックアップ中にデータベースの変更が発生したすることもありますが、PostgreSQLではバックアップ開始時点での整合性が維持されることが「PostgreSQL文書」のバックアップとリストアの章に記載されております。OracleのData Pumpユーティリティの場合、ダンプファイルの整合性を維持するためにはDBがアーカイブ・ログモードかつexpdpのオプションでflashback_timeまたはflashback_scnパラメータを使用する手間が必要となります。その点でPostgreSQLのほうが使い勝手は良いと言えます。



● 継続アーカイブ＋PITR例

この方式は前の２つの方式より複雑ですが、最新の状態までリカバリできること、オンラインでバックアップできることから商用運用時のバックアップ方式として最も適しているバックアップ方式と言えます。

➢ バックアップ例

• 継続アーカイブの指定

postgresql.confのパラメータを以下の様に指定しインスタンスを再起動します。

wal_level = replica

archive_mode = on

archive_command = ‘cp “%p” “アーカイブ先ディレクトリ/%f”’

max_wal_senders = 2以上

(%p：walファイル名、%f：アーカイブwalファイル名　が自動でセットされる。)

• ベースバックアップの取得

$ pg_basebackup -D /home/postgres/basebk -Ft -X s -z -P

-D :バックアップ先ディレクトリ。バックアップ先ディレクトリは空でなければなりません。

-Ft：出力フォーマットの指定。tはtar形式。バックアップファイル名はbase.tarとなります。

-X：ベースバックアップ処理中にDBが更新された場合、自動でログをpg_wal.tarにバックアップします。sはstreamであり、このときmax_wal_sendersは2つ必要となります。これは、バックアップ時点の整合性を維持するものであり、アーカイブWALまで破損していたとしてもバックアップ時点までなら整合性を維持したリストアができるようになります。アーカイブWALが存在する場合、pg_wal.tarは使用しません。

-z：gzipで圧縮

-P：進捗の表示。以下のように表示されます。

2156510/3493691 kB (61%), 0/1 テーブル空間

➢ リストア例

最新の状態まで復元する場合、以下の３つが必要になります。

• ベースバックアップ　　pg_basebackupで取得したダンプファイル

- アーカイブWAL　　pg_basebackup後に発生したWALのアーカイブ
- アーカイブ前のWAL　　最新のWAL

［リストア手順］（PostgreSQL 12以降の場合。11以前は異なります）

① アーカイブ前WALのバックアップ

データベースクラスタ格納ディレクトリよりバックアップします。ここが破損している場合、最新の状態まで復元はできません。

$ cp -pr $PGDATA/pg_wal /home/postgres/basebk/

② データベースディレクトリ以下の削除

$ rm -rf $PGDATA/*

③ ベースバックアップの復元

$ cp /home/postgres/basebk/base.tar.gz $PGDATA

$ cd $PGDATA

$ tar xvfz base.tar.gz

④ ①でバックアップしたアーカイブ前WALの復元

$ cp -pr /home/postgres/basebk/pg_wal/* pg_wal/

⑤ postgresql.confのパラメータ変更

restore_command = ‘cp “アーカイブ先ディレクトリ/%f” “%p”’

任意の時刻まで復元したい場合は以下の様に時刻も指定します。何も指定されない場合は最新まで復元を試みます。

recovery_target_time = ‘2021-03-02 12:00:00’

⑥ recovery.signalファイルを作成

$ touch recovery.signal

⑦ インスタンスの起動（アーカイブWAL,アーカイブ前WALの適用）

$ pg_ctl start

ここでアーカイブWAL、アーカイブ前のWALが適用されます。適用中はDBに接続できません。

⑧ 確認作業

ログにはWALの適用状況、最後に完了したトランザクションの時刻などが書き込まれます。これらを確認し最新または任意の時点に復元されていることを確認します。

適用状況のログの例

LOG: ログファイル”000000010000000800000F5”をアーカイブからリストアしました

トランザクションの時刻のログの例

LOG: 最後に完了したトランザクションのログ時刻は2021-03-02 14:08:11.131614+09でした

● タイムライン履歴ファイルについて

PostgreSQLにはタイムラインという概念があり、リストアを実施するとタイムラインIDがカウントアップします。以後の変更は新しいタイムラインのWALで管理されます。このようにすることで、繰り返しリカバリ行えるようにしています。（詳細はマニュアルの「継続的アーカイブとポイントインタイムリカバリ（PITR）」の章を参照）

具体的には“リカバリ後”、または“ストリーミングレプリケーション環境ではスレーブ側を昇格させる”とpg_wal以下に作成される.historyが付くファイルがタイムライン履歴ファイルになります。このファイルはWAL領域に作成されると同時にアーカイブWAL領域にもアーカイブされます。これは小さなテキストファイルであり、マニュアルでは無期限に保管するように記載されています。実際にWAL領域内のWALセグメントはmin_wal_sizeまで自動で整理されますが、このファイルは整理対象外となり削除されることはありません。また、現在のタイムラインはpg_controldataコマンドで確認することができます。

➢ タイムライン履歴ファイルが失われるとどうなるか

タイムラインIDが2以上でかつ最新のタイムライン履歴ファイルが削除されると、以降のpg_basebackupがエラーになります。

エラー例）

```
$ pg_basebackup -D /home/postgres/backup/basebk -Ft -X s -z -P
pg_basebackup: レプリケーションコマンド"TIMELINE_HISTORY"を送信できませんでした: ERROR: could not open file
"pg_wal/00000003.history": No such file or directory
194483/194483 kB (100%), 1/1 テーブル空間
pg_basebackup: 子プロセスが終了コード1で終了しましたpg_basebackup: データディレクトリ
"/home/postgres/backup/basebk"の内容を削除しています
```

何らかの理由で失われた場合はアーカイブWALのディレクトリまたは前回ベースバックアップのpg_walから復元します。ストリーミングレプリケーションを構成する場合でもpg_basebackupを利用するため同様な現象でエラーとなりスレーブ側のDBが作成できなくなります。

- アーカイブWALの整理

PostgreSQLは自動でアーカイブしたWALの整理を行うことはありません。定期的な作業として不要なアーカイブWALを削除する必要があります。以下のコマンドでoldestkeptwalfile（.backupが付いているファイル）以前のアーカイブWALをクリーンアップすることができます。

pg_archivecleanup -d アーカイブディレクトリ oldestkeptwalfile

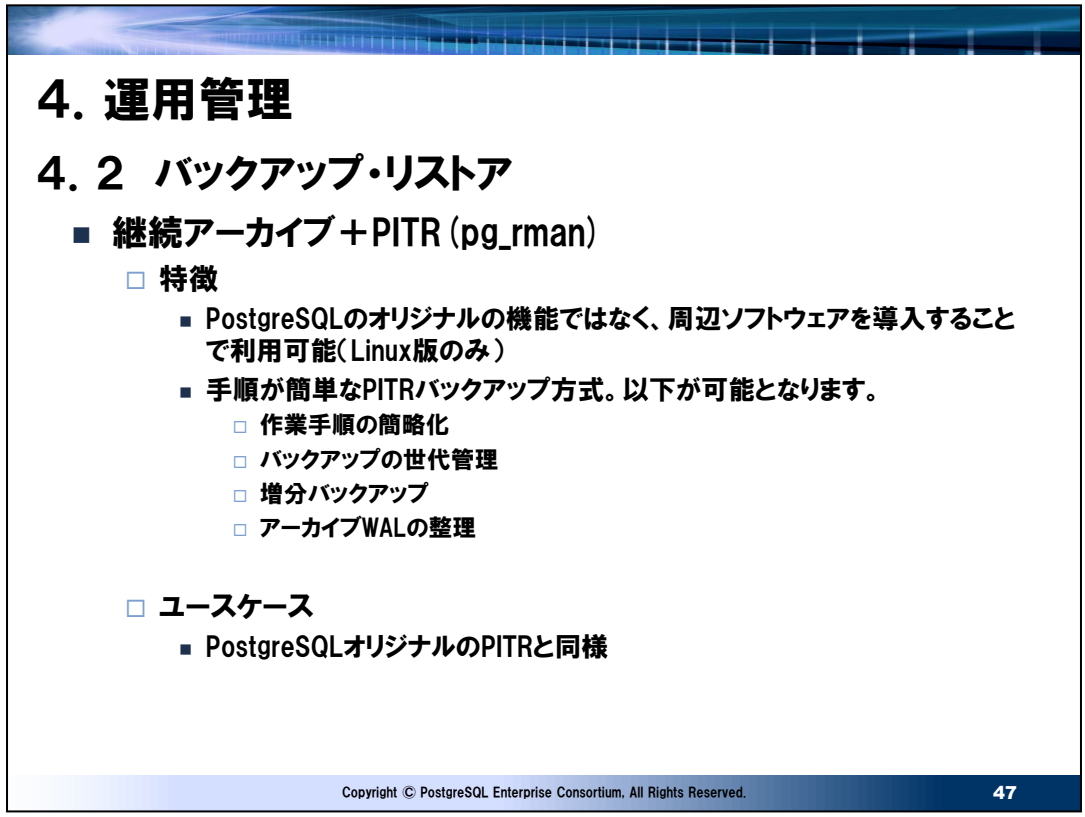
-d:エラー出力にログを表示

- 参考

PostgreSQL 11以降ではデータベースクラスタのデータチェックサムを有効にしている場合、バックアップするデータベースクラスタ中に破損ページがあると以下のようにエラーを検出し終了します。

```
$ pg_basebackup -D /home/postgres/basebk -Ft -z
WARNING:  ファイル"./base/16384/16407"のブロック1でチェックサム検証が失敗しました: 計算されたチェックサムは37A2ですが想定はF7D8です WARNING:  could not verify checksum in file "./base/16384/16407", block 1640: read buffer size 1 and page size 8192 differ pg_basebackup: エラー: チェックサムエラーが発生しました
```

このような場合、正常なベースバックアップからのリカバリが必要になります。また、PostgreSQL 13以降からpg_basebackupで取得されたバックアップの完全性をチェックするためのpg_verifybackupコマンドも用意されています。



- pg_rmanでの運用

pg_rmanはpg_rman.iniを適切に編集することで、手順を簡略化することができます。

➢ 運用例

pg_rman.iniが以下の場合

```
$ cat pg_rman.ini
ARCLOG_PATH='アーカイブWALディレクトリ'          ←pg_rman初期化時に自動で設定されます
SRVLOG_PATH='ログディレクトリ'                   ←pg_rman初期化時に自動で設定されます
BACKUP_MODE=FULL                                ←バックアップモードは全体
COMPRESS_DATA=TRUE                              ←バックアップは圧縮
KEEP_DATA_GENERATIONS=2  ←保管世代は２世代（古いものはバックアップ時に削除）
SMOOTH_CHECKPOINT=YES    ←バックアップ時はチェックポイント処理を早く終わらせる
KEEP_ARCLOG_DAYS=3       ←アーカイブWALの保存日数（バックアップ実行時に保存期間を超えたものは削除される）
```

上記の時、毎日全体バックアップすると仮定すれば通常は以下２つのコマンドだけになります。

```
$ pg_rman backup
$ pg_rman validate
```

古いバックアップが自動で整理されたとしてもその情報は"バックアップ削除済"として残ります。その情報も削除したい場合は以下のコマンドを入力します。

```
$ pg_rman purge
```

障害が発生しリストアが必要になった場合、１つのコマンドのみで完了します。

```
$ pg_rman restore
```

- 参考

pg_basebackupの場合、データベースクラスタのデータチェックサムを有効にしていればデータベースクラスタ中に破損ページがあるとエラーを検出できますが、pg_rmanのバックアップでは検出できません。別途データベースクラスタのデータ破損をチェックを実施するのであれば以下のコマンドで検査します。なお、どちらのコマンドもインスタンスを停止させる必要があります。

- PostgreSQL 11の場合　　pg_verify_checksums
- PostgreSQL 12以降の場合　　pg_checksums

4. 運用管理

4. 3 VACUUM・ANALYZE処理

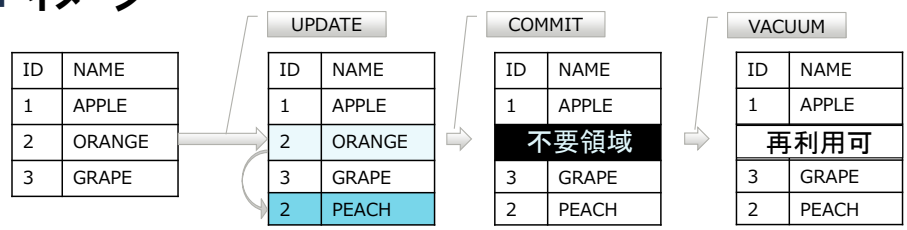
- **VACUUM処理が必要な理由**
  - PostgreSQLはマルチバージョン方式＝MVCC（多版型同時実行制御）で同時実行制御を行います。

  →行を更新するとき、**行を書き換えずに新しい行を追加**します。

  （更新するカラムだけでなく行全体を追加です）

  →トランザクションが確定されると元の行の領域は不要領域になります。

  →不要領域を回収する仕組みが必要です=バキューム処理
- **イメージ**



● バキューム処理が必要な理由

PostgreSQLもOracle同時実行時の読み取り一貫性を確保するためMVCC方式を採用しています。ただし、その実装方式は異なりPostgreSQLは追記型を採用しています。このため、スライドのイメージに示すように不要となった領域を再利用可能にするためのバキューム処理が必要になりますが、代わりにOracleのUNDOセグメントのような領域はありません。知っておくべきはバキューム処理を行わないと更新や削除による不要領域が増大し表の領域は延々と拡大し続けます。実際の例を示すとわかりやすいでしょう。

```
test01=# select table_len,tuple_count,dead_tuple_count from pgstattuple('sample_table');
 table_len | tuple_count | dead_tuple_count
-----------+-------------+------------------
  84459520 |     1000000 |                0
(1 row)

test01=# update sample_table set syor_flg = '1';
UPDATE 1000000
test01=# select table_len,tuple_count,dead_tuple_count from pgstattuple('sample_table');
 table_len | tuple_count | dead_tuple_count
-----------+-------------+------------------
 168910848 |     1000000 |          1000000
(1 row)
```

table_lenは表のサイズです。約80MBの表は全件を１バイト更新するだけでサイズが２倍※の約160MBになっています。元のdead_tuple_countの占める領域は不要領域となります。バキューム処理がなされなければ、今後の更新で不要領域は再利用されず、さらに領域が拡大します。性能面で言えば、全件検索されたとき“不要領域も”含めそのテーブルのすべての領域がアクセスされますので読み込み量は増加します。つまりそれだけレスポンスが悪化します。

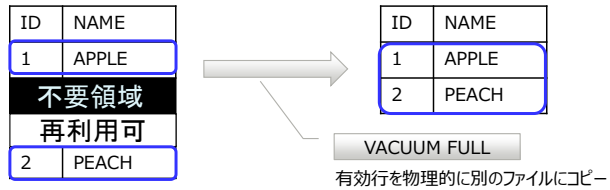
※ 表のfillfactor=100%であるため（詳細は2.6 スキーマオブジェクト – 表 - fillfactorを参照）

4. 運用管理

4. 3　VACUUM・ANALYZE処理

- **VACUUM処理のパターン**
  - VACUUM
    - 不要領域を再利用可能にします。表の領域は変わりません。
  - VACUUM FULL
    - 有効行を別の領域にコピーする。結果として表の領域は圧縮されます。
    - バキューム処理中はACCESS EXCLUSIVEのロックがかかります。

- **VACUUM FULLのイメージ**



● バキューム処理

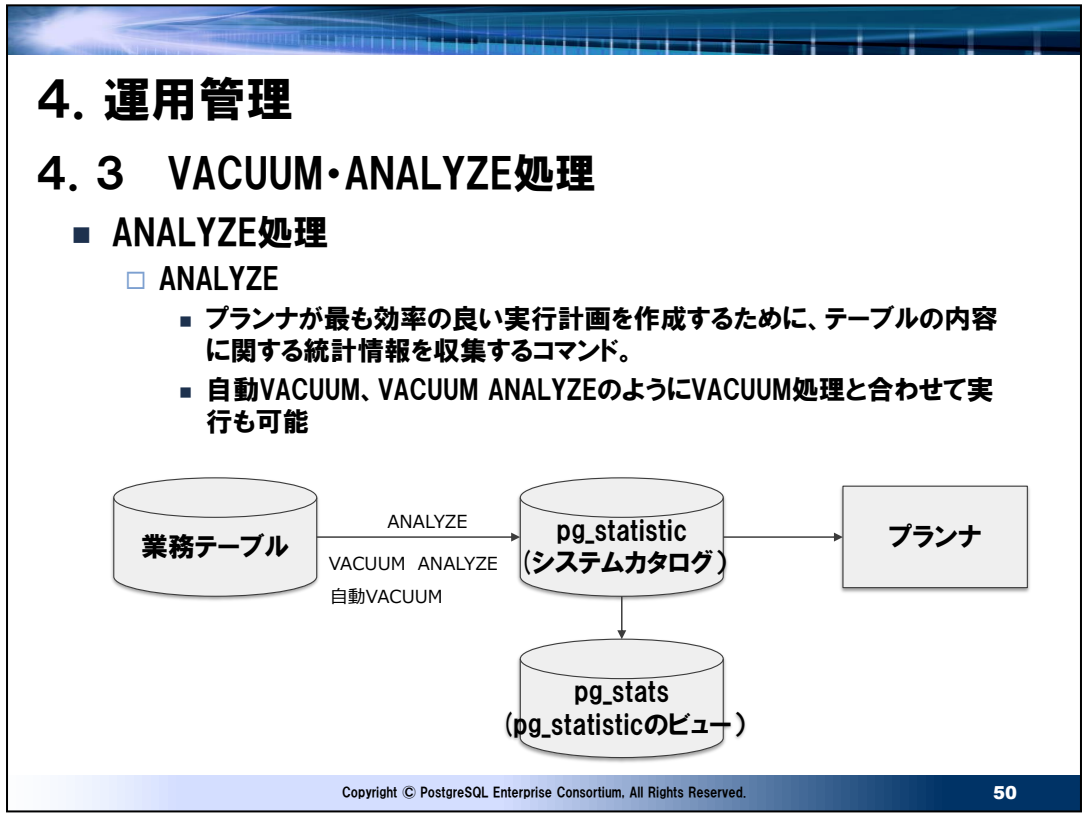
VACUUM、VACUUM FULL共に例を示すとわかりやすいでしょう。前頁の続きからそれぞれのバキューム処理を実施します。

```
test01=# select table_len,tuple_count,dead_tuple_count from pgstattuple('sample_table');
 table_len | tuple_count | dead_tuple_count
-----------+-------------+------------------
 168910848 |     1000000 |          1000000                    ←前頁の劣化した状態。
(1 row)

test01=# vacuum sample_table ;
VACUUM
test01=# select table_len,tuple_count,dead_tuple_count from pgstattuple('sample_table');
 table_len | tuple_count | dead_tuple_count
-----------+-------------+------------------
 168910848 |     1000000 |                0
(1 row)
```

VACUUMすることでdead_tuple_count(不要領域）が解消されたことが確認できます。ただし表サイズはそのままです。

```
test01=# vacuum full sample_table ;
VACUUM
test01=# select table_len,tuple_count,dead_tuple_count from pgstattuple('sample_table');
 table_len | tuple_count | dead_tuple_count
-----------+-------------+------------------
  84459520 |     1000000 |                0
(1 row)
```

VACUUM FULLにより元の80MBに戻っていることが確認できます。ただし、VACUUM FULLはスライドにあるように強力なロックがかかるため、運用中に実施することはできません。実施するには保守時間を設ける必要があります。

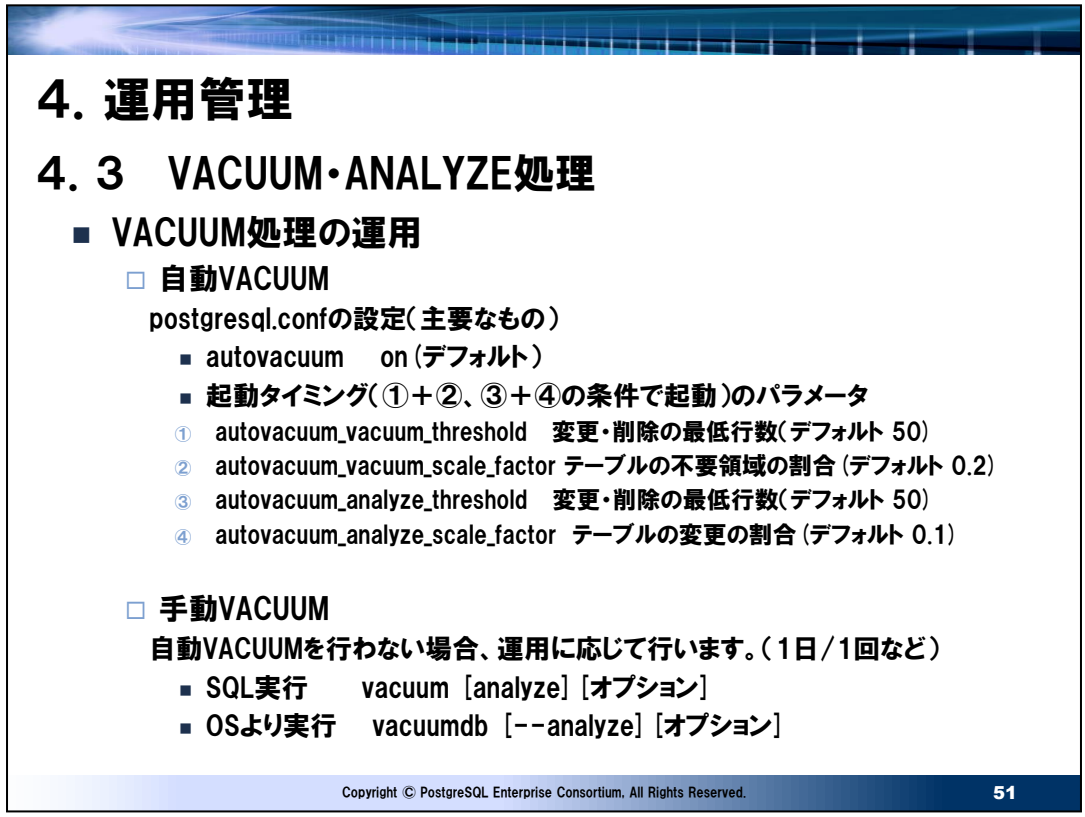


- 統計情報量

ANALYZEで取得する統計の情報量はdefault_statistics_targetパラメータ（デフォルト値：100）により度数分布のバケット数で指定されます。このパラメータにより、テーブルのサンプリング量、システムカタログへの統計情報の量が決まります。パラメータの値を増やせばより正確な情報が取得できます（＝バラツキのあるデータ分布の場合は、より効率の良い実行計画が作成できる可能性が高まる）がANALYZE処理時間の増加、統計情報の容量が増加するというデメリットも発生します。ほとんどの場合はデフォルトままで良いですが大規模なテーブルの索引カラムでカーディナリティ（カラムの値の種類の絶対値）が高い場合は以下のようにデフォルトを変更するとより効果的な実行計画が作成されやすくなります。

例 テーブルの特定カラムのバケット数を200にする。

```
alter table テーブル名 alter column カラム名 set statistics 200;
```



● VACUUMとANALYZE

スライドにあるように自動VACUUMではANALYZE（プランナが使用するための統計情報を更新）も合わせて指定できます。手動ではVACUUM ANALYZEとすることでVACUUMを実行した後にANALYZEも実行するようになります。ANALYZEのみを実行する場合はANALYZEコマンドを使用します。

● 自動VACUUM起動プロセス

①autovacuum_naptime（デフォルト1分）毎にworkerプロセスを起動

②autovacuum_freeze_max_ageの超えた世代のトランザクションIDのテーブルはvacuum

③autovacuum_vacuum_threshold＋autovacuum_vacuum_scale_factor×行数 ＜ 不要領域 のときvacuum

④autovacuum_analyze_threshold＋autovacuum_analyze_scale_factor×行数 ＜ 不要領域 のときanalyze

●自動VACUUMの個別のテーブル毎の指定

大規模テーブルなど個別に自動VACUUM条件を変えたいとき、ALTER TABLE文で変更可能です。具体的にはパラメータのautovacuum_vacuum_scale_factorのデフォルト値は0.2であり1億件ある表の場合、2000万件の不要領域が発生してからVACUUMされます。一度に発生するVACUUM処理量が多くシステム負荷が高くなるため、それを軽減するためにこのパラメータを小さく指定し小刻みにVACUUMを行う場合などです。

例

```
alter table テーブル名
set ( autovacuum_vacuum_scale_factor=0.05, autovacuum_analyze_scale_factor=0.01);
```

● 手動でVACUUMを行うケース

手動VACUUMの検討が必要なケースもあります。例えば、オンライン処理で自動VACUUMが動作してしまうと、レスポンスが遅くなり単純にこの遅延が許容できない、またはレスポンスがバラつくことで利用者のクレームがでる場合などが該当します。保守時間が確保できるようなケースでは手動VACUUMで1回/日のように実行するほうがオンライン処理中でのVACUUMの影響をなくすことができます。保守時間が確保できない場合は、オンライン接続数が少ない影響の少ない時間帯に手動VACUUMを実施するなど工夫します。なお、自動VACUUMを停止するとANALYZEも合わせて停止するため、手動VACUUM時にはANALYZEも実行する必要があります。また、バッチ処理などで一度に大量のテーブルの更新が行われ、直

後（自動VACUUMが起動される前）にそのテーブルをアクセスする場合はバッチ処理中のコマンドとしてANALYZEを追加したほうが効率のよい実行計画でクエリが実行できバッチ処理全体の処理時間を安定させることができます。

- VACUUM・ANALYZEの実行の確認

VACUUMおよびANALYZEの実行はpg_stat_all(またはuser)_tablesビューで確認できます。この過去の実施情報はpg_stat_reset関数が実行されるとクリアされます。

確認例

```
test01=# select last_vacuum,last_autovacuum,last_analyze,last_autoanalyze,
test01-#        vacuum_count,autovacuum_count,analyze_count,autoanalyze_count
test01-# from pg_stat_user_tables
test01-# where relname = 'pgbench_accounts';
-[ RECORD 1 ]-----+------------------------------
last_vacuum       | 2022-02-03 17:49:00.452216+09
last_autovacuum   | 2022-02-21 17:52:50.106337+09
last_analyze      | 2022-02-03 17:49:00.666983+09
last_autoanalyze  | 2022-02-21 17:52:51.124659+09
vacuum_count      | 1
autovacuum_count  | 3
analyze_count     | 1
autoanalyze_count | 3
```

※　上記のlast_vacuum、vacuum_countにはVACUUM FULLの実績は含まれません。

- トランザクション周回問題の回避のためのVACUUM

PostgreSQLのMVCCでは新旧のバージョンの行が存在します。その新旧をトランザクションIDで比較しますが、このサイズは32ビットであり、周回するとトランザクションIDは0に戻ります。このため新旧の比較ができなくなるので強制的に過去のものとする（凍結状態）フラグを立てる処理をバキュームで自動的に実行されます。これはautovacuum を無効にしても “強制的に実行”されます。実行の閾値となるパラメータはautovacuum_freeze_max_ageの超えた世代のトランザクションIDがvacuum対象となります。

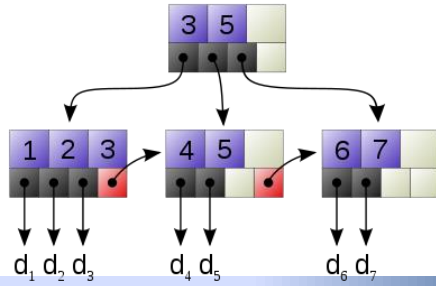
4. 運用管理

4. 4 索引の再構築

■ 索引の再構築が必要な理由

- □ B-treeインデックスは運用とともにブランチの階層は階層数が多くなる、またリーフノードの格納密度が劣化します。
- □ 劣化するとインデックスブロックのアクセス数の増加、ディスク領域の増加が発生します。
- □ PostgreSQLではブロック内全てのキーが削除されたら再利用されます。

→上記だけでは足りないので再構築が必要となります。



出展：ウィキペディア B+木

● 索引が劣化したときの状況

具体的にB-treeの構造と運用中の表データの追加・削除・更新、および索引が劣化した場合の状態を説明します。

① B-treeの構造

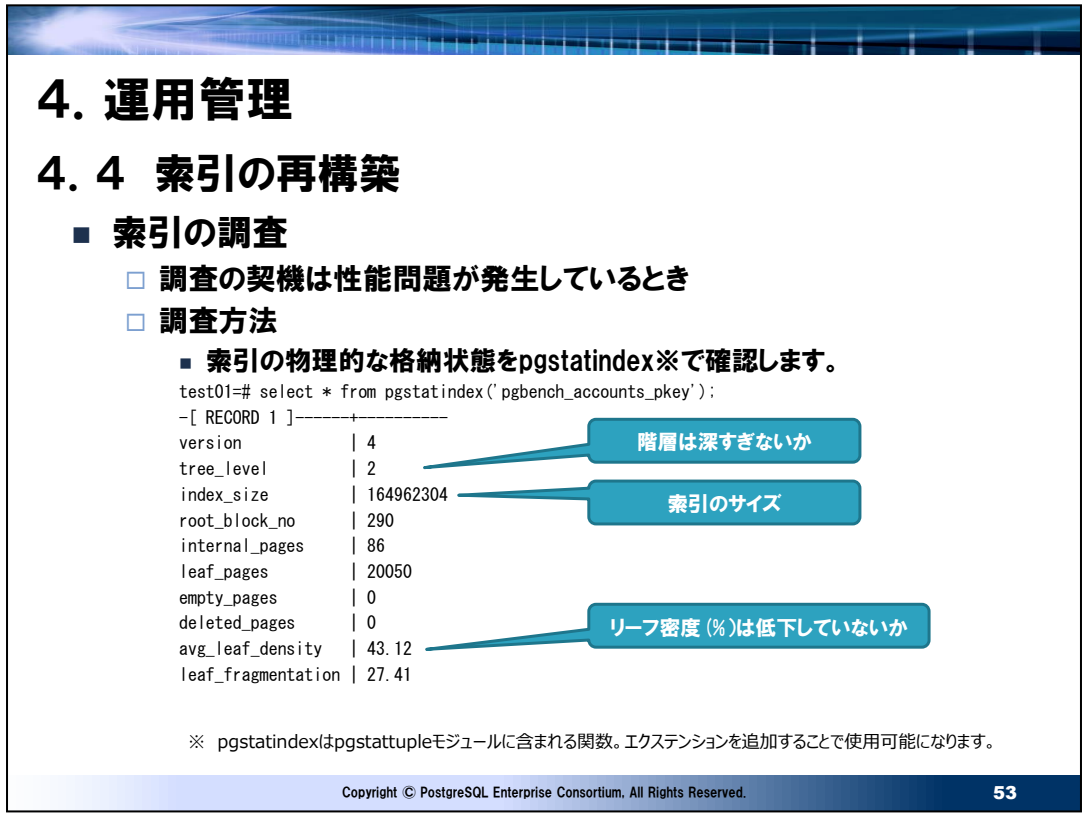
- ルートノードと子ノードはキー値と子ノードのポインタを持ちます。
- リーフノードはキー値とデータへのポインタを持ちます。
- 索引はソートされている必要があります。
- 検索はルートノード、ブランチノード（階層数は0から複数）、リーフノードをたどり目的のデータのポインタを得ます。

② データの追加・更新・削除

- 索引の中間に追加する場合はソート順を維持しつつ、ブロックに空きがあれば追加し、空きがなければブロックを分割します。
- 削除の場合は削除されたフラグが立ちます。
- 索引項目の更新は更新前のキーがリーフノードから削除され、更新後のキーが追加されます。
- PostgreSQLの場合、リーフノードの全てのデータが削除されない限り領域は再利用されません。

③ 索引が劣化したときの状況

- 削除インデックスが大量にある状態で範囲検索を実施する場合、ほとんど空のリーフブロックを大量に読み込むことになります。例えばブロック密度が90%の状態と10%の状態を比較すれば10%は９倍の索引ブロックを読む必要があります。
- 索引検索で１件のデータを取得するとき正常であればルート、ブランチ（ここでは１階層とします。データ件数、索引により最良な状態でも複数の階層を持つケースもあります）、リーフ、対象のデータの4ブロックの読み込みで済むところ劣化が進みブランチブロックが分割され３階層になったとすると全部で6ブロックを読むことになります。正常からすれば1.5倍の読み込み回数になります。



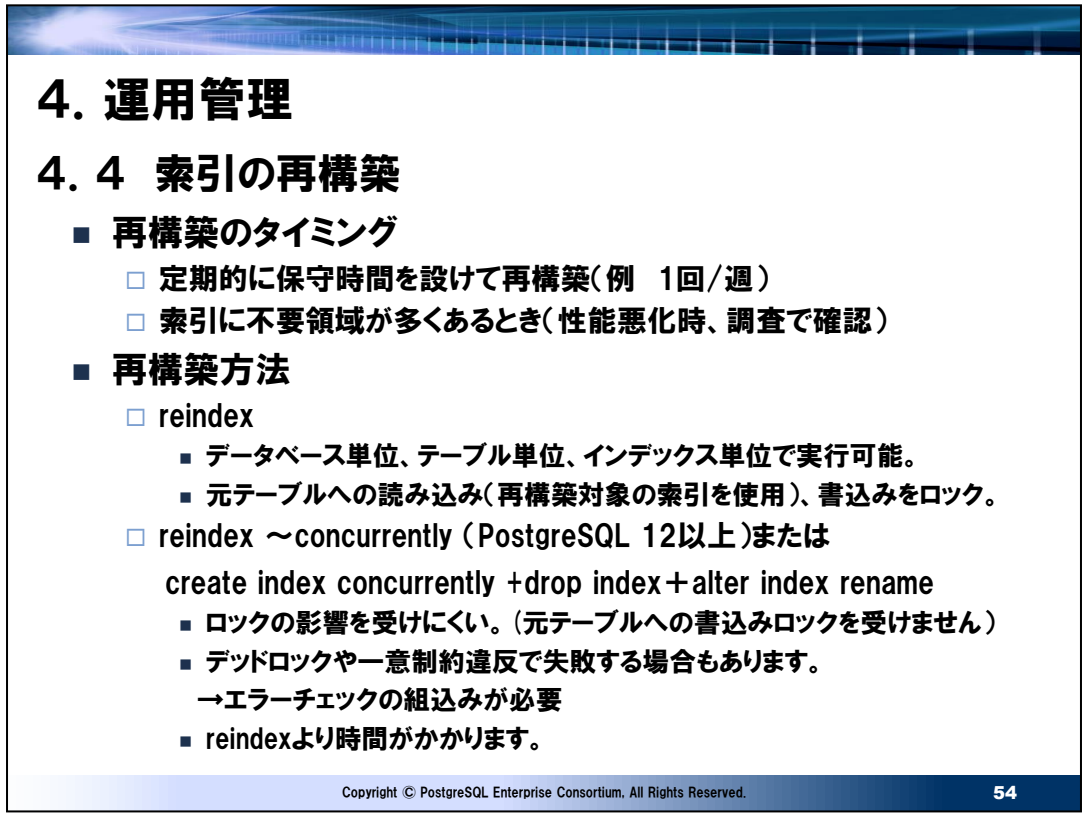
- 調査の契機

性能問題も全体的に遅い場合と特定のSQLが遅い場合があります。全体的に遅い場合は統計情報コレクタのpg_stat_all(またはuser)_indexesのidx_scanなどから、使用頻度の高い索引を特定し調査します。特定のSQLが遅い場合は実行計画を取得することで調査対象の索引を特定します。後者の場合、使用頻度が多くないバッチ処理などが該当します。

- 確認する項目

pgstatindexで索引の物理的な状態を確認できます。作成または再構築直後が最良の状態です。B-tree索引のfillfactorのデフォルトは90%であるためリーフ密度は90%前後になります。性能上問題があり、リーフ密度が著しく低い場合は再構築の対象になります。スライドの索引を再構築すると以下のような状態になります。索引サイズは164MBから76MBと半分以下となっており、これはテーブル件数をカウント（プライマリキーのフルスキャン）する場合など、スライドと比較して半分以下の読み込み量になります。

```
test01=# reindex index pgbench_accounts_pkey;
REINDEX
test01=# select * from pgstatindex('pgbench_accounts_pkey');
-[ RECORD 1 ]------+----------
version            | 4
tree_level         | 2
index_size         | 76390400
root_block_no      | 290
internal_pages     | 34
leaf_pages         | 9290
empty_pages        | 0
deleted_pages      | 0
avg_leaf_density   | 90.08
leaf_fragmentation | 0
```



● 索引再構築のタイミング

B-tree索引は運用中のテーブルの更新、削除により劣化します。性能が悪化してから調査し対象の索引を再構築するより、定期的な保守処理として実施することで性能を安定させることができます。その上で性能問題が発生した場合は、調査を行い特定の索引のみ再構築間隔を短くするなどの対応を行います。

●再構築方法の選択基準

索引再構築時の作業時間、テーブルに対するロックを考慮し、以下の基準で決定します。

- 運用保守時間(業務停止時間)を確保できる場合はreindex
- 運用保守時間(業務停止時間)を確保できない場合かつPostgreSQL 12以上の場合はreindex concurrently
- 運用保守時間(業務停止時間)を確保できない場合かつPostgreSQL 11以下はcreate index concurrently＋drop index+alter index rename

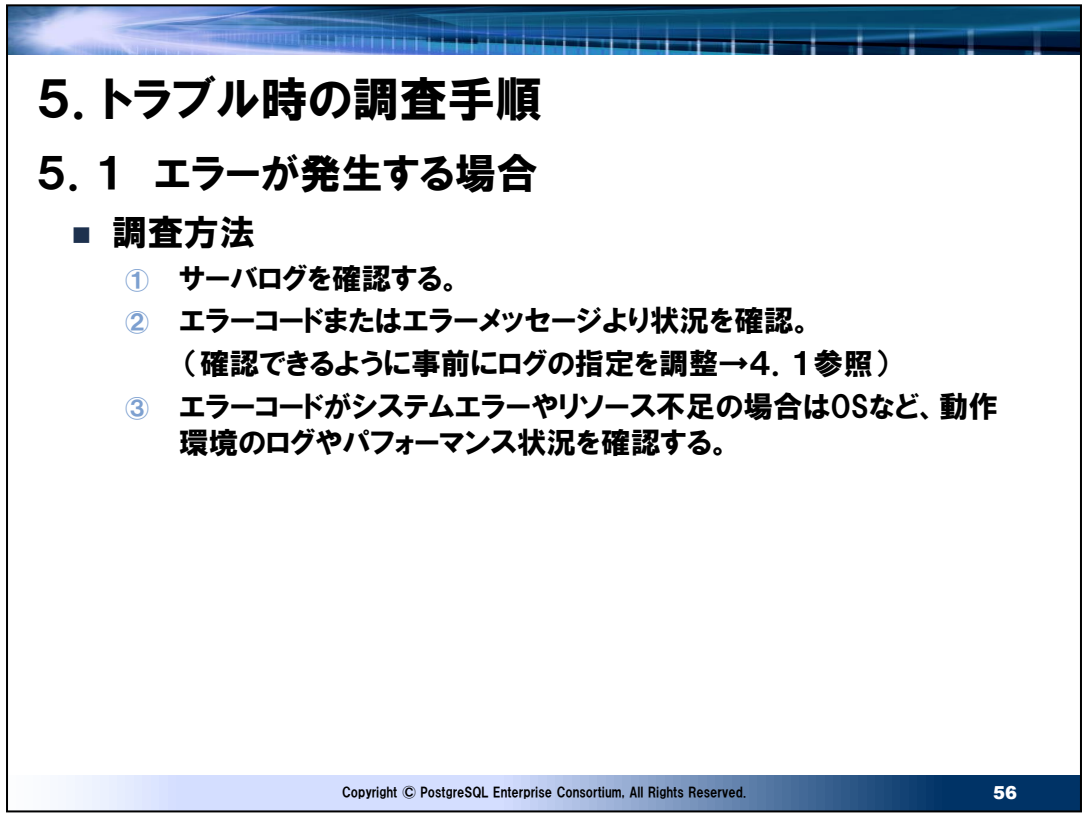
●再構築実施時の注意事項

- reindex, reindex ～ concurrentlyは再構築実施時は元の索引から別の領域に再構築するため最大で元の索引領域分の空き容量が必要となります。
- reindex concurrently, create index concurrentlyはロックの影響が小さい代わりに時間がかかります。テーブルを２回読み込み、再構築中に発生した索引更新を含むトランザクションの修正を待ちます。

●create index concurrently 、reindex concurrentlyで作成失敗した場合

作成に失敗した場合、そのインデックスの残骸が残ります。この場合、システムカタログ（pg_indexのindisvalid(ただしインデックス名はpg_classのrelnameと合わせて確認)）で確認できます。対応は作成に失敗した索引を削除し再作成するか再度reindexを実行します。

5. トラブル時の調査手順



- PostgreSQLのエラーコード

PostgreSQLのエラーコードは標準SQLのSQLSTATEに従います。PostgreSQLのマニュアルである「PostgreSQL文書」の付録部分にエラーコードの説明があります。エラーコードをデータベースログに残す場合はパラメータ：log_line_prefixに%eを指定します。psql上でエラーコードを含め詳細に表示したい場合はクエリ実行前に以下を入力します。

```
test01=# ¥set VERBOSITY verbose
```

- エラーの確認例

以下はエラーが発生した場合とその確認方法になります。パラメータ：log_line_prefixを以下の様にしておきます。

postgresql.confのパラメータ

log_line_prefix = '%m [%e] ‘

以下のオペレーションをします。

```
$ psql test01
psql (14.1)
"help"でヘルプを表示します。

test01=# select * from test;
ERROR:  relation "test" does not exist
行 1: select * from test;
                      ^

test01=# ¥q
```

このときのDBログは以下のようになります。

```
2022-03-03 14:27:12.676 JST [00000] LOG:  connection authorized: user=postgres database=test01
application_name=psql
2022-03-03 14:27:15.095 JST [42P01] ERROR:  relation "test" does not exist at character 15
2022-03-03 14:27:15.095 JST [42P01] STATEMENT:  select * from test;
2022-03-03 14:27:22.016 JST [00000] LOG:  disconnection: session time: 0:00:09.341 user=postgres
database=test01 host=[local]
```

上記では[]内の42P01がエラーコードになります。

- OS等のデータベース以外の情報収集

データベースからのファイルアクセス異常、レプリケーション時のWAL転送異常など、OS、ネットワークなど環境起因で異常となるケースも多いです。合わせてこれらのログやパフォーマンス状況を取得しておくことで早期に原因解決に繋がるケースも多くあります。

- データベースインスタンスの異常停止

データベースインスタンスが異常停止してしまうパターンは主に 2 つあります。“ハード、OSなどのデータベース実行環境の異常”または“PostgreSQL本体のバグ”になります。後者の場合はPostgreSQLのリリースノートを確認し、原因が該当する場合はマイナーバージョンアップを行います。

PostgreSQLのリリースノート

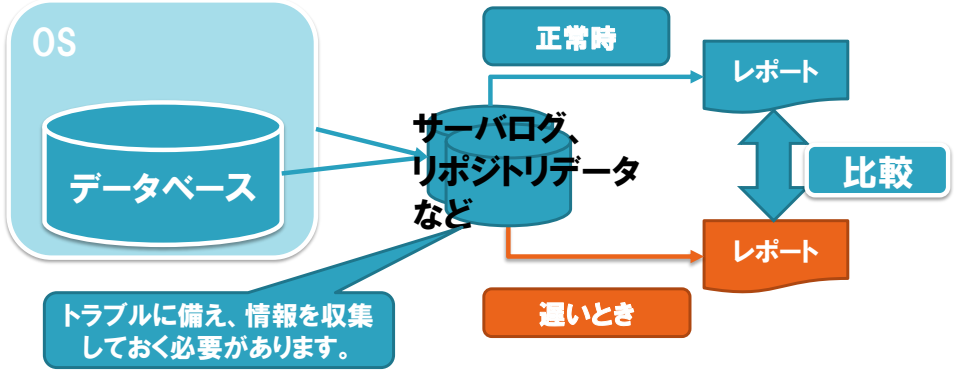
https://www.postgresql.org/docs/release/

5. トラブル時の調査手順

5. 2 性能が劣化する場合

■ 性能劣化と事前準備

性能は相対的な評価であり、正常時と比較してどのように変化し悪影響を与えているかを確認することが基本的手順になります。その事前準備として各種ログやツールのリポジトリデータを収集しておく必要があります。



- 性能劣化の把握

性能劣化は相対的なものであるため、比較するための正常時の基準が必要になります。この基準は何がどのように変化して遅くなっているか原因を特定するためにある程度詳細な情報を時系列的に収集しておくことが重要です。また、性能劣化はデータベース以外の要因で発生している場合もあるため、データベース関連だけでなくOSの稼働統計を含むシステムの運用の状態を把握しておくことも必要になります。（スライド中央の"サーバログ、リポジトリデータなど"の部分）

- 情報取得のオーバーヘッド

情報は詳細に取得すればするほどデータベース、OSへのオーバーヘッドが高くなり性能を悪化させ、ログ等の容量を増加させます。データベース上で運用されるシステムの要件によってこの詳細度を決定してください。例えば、詳細なログを取得していない場合、問題が発生してからその原因を特定できるようにログを調整し状況を再発させて調査するような時間が必要になります。この時間が許されないシステムでは最初から詳細なログを取得するようにすべきであり、そのオーバーヘッド分のリソース増を見込んでおく必要があります。

5. トラブル時の調査手順

5. 2 性能が劣化する場合

- PostgreSQLでの取得可能な情報

  情報の種別と手段をまとめると以下の表になります。

  ○:基本機能 □:周辺ソフトウェア △:その他

情報	手段
現時点の情報 または 現時点までの累積情報	○ システムカタログ ○ 統計情報コレクタ ○ pg_stat_statementsなどの追加モジュール □ pg_store_plans
過去の時系列的な情報	○ データベースログ （何を出力するかはパラメータファイルで調整） □ pg_statsinfo △ 統計情報コレクタ等の累積ツール（Windows版）

- PostgreSQLで時系列に収集しておく情報

上記スライドの手段の詳細は4章にて説明しております。トラブルに対する備えとしてPostgreSQLの基本機能だけでは過去の時系列的な情報はデータベースログのみとなるため、情報としては不足します。このため、必要に応じて周辺ソフトウェア等の利用を検討することを推奨します。

5. トラブル時の調査手順

5. 2 性能が劣化する場合

- **全体的に遅い場合の調査手順（以下は順不同です）**
  - ① **運用上の変化があるか確認。**
    - データベースへの接続数が増加したか
    - データの処理件数が増加したか
  - ② **正常時と異常時の比較。（“性能劣化と事前準備”で収集した情報）**
    - DB:WALサイズ不足でのチェックポイントの発生、一時ファイルの使用など
    - OS:CPU、メモリ、IOでボトルネックが発生していないかなど
    - パフォーマンス診断ツールでの任意の時間帯の比較での差異の確認
  - ③ **PostgreSQLで各種情報の確認**
    - pgstattupleで表の格納状態が劣化していないか
    - pgstatindexで索引の状態が劣化していないか
    - pg_stat_XXX_indexesで想定した索引が使われているか
    - pg_stat_XXX_tablesで想定外にシーケンシャルスキャンが多すぎないか

    など

- 全般的に遅い場合

特定の処理だけでなく、全般的に遅い場合はスライドの情報の複数の部分で現象が現れることが多いです。例えば、データ量が増えれば処理によっては1度のSQLでの読み込み件数も増え、物理IOの増加、work_mem内のメモリソートだけでなく一時ファイルの使用の増加、結果的にIOボトルネックまで併発し全体的に遅くなるなどのケースがあります。多角的に調査することで、正確に原因の因果関係を特定することができ、最小限のコストで対応が可能になります。

5. トラブル時の調査手順

5. 2 性能が劣化する場合

■ 特定の処理が遅い場合

① 遅いSQLの特定と実行計画の確認

	遅いSQLの特定	実行計画の確認
基本機能	• サーバログ: log_min_duration_statement, auto_explain • pg_stat_statements	• サーバログ:auto_explain • explainコマンド
拡張ソフトウェア利用		• pg_store_plans

② 原因の分析

- SQLで使用している表や索引の物理的な状態をpgstattuple,pgstatindexで劣化していないか確認します。
- 遅いSQLとその実行計画が確認できたら、本来想定していた実行計画になっていない原因を分析・調査します。

- 遅いSQL、実行計画の特定方法の向き、不向き

スライドに示した通り、遅いSQLと実行計画は確認できるのですが、それぞれ向き、不向きがあります。

- log_min_duration_statement：バッチ処理向き。オンライン処理の秒レベルの閾値を指定するとサーバログが大量に出力されログが肥大化します。
- auto_explain：log_min_duration_statement と同様。
- pg_stat_statements：バッチ、オンライン処理の分析が可能ですがリセットされてからの累積情報のため、分析前にリセットをするかpg_statsinfoと合わせて利用します。
- explain：開発、SQLチューニング時などSQLの実行計画を確認する場合に使用します。コマンド実行時の実行計画であり、過去の遅かったときの実行計画と同一とは限りません。
- pg_store_plans：オンライン、バッチ処理の分析が可能です。ただし、情報の取得閾値を指定しない場合はオンライン処理のような大量のSQL発行に伴う情報取得のオーバーヘッドが大きくなるため、その負荷が運用に影響するか検証することを推奨します。取得された情報の利用についてはpg_stat_statements同様にリセットされてからの累積情報のため、特定のバッチ処理前など分析前にリセットをするかpg_statsinfo（＋pg_stats_reporterもあるとなお良い）を合わせて利用することで遅い処理、時間帯での分析が可能となります。

6. Oracle移行時の注意点

6. Oracle移行時の注意点

6. 1　Oracleとの機能差

- **機能の有無**

  移行に関連するPostgreSQLの新機能の追加

メジャーバージョン	主な追加機能
9.6	パラレルクエリ
10	宣言的パーティショニング、論理レプリケーション
11	ストアドプロシージャ

- **機能レベル差**

  機能詳細まで踏み込んで、移行が可能か確認する必要があります。

  例）マテリアライズド・ビュー

リフレッシュ・タイプ	Oracle	PostgreSQL 14
完全	○	○
増分（差分）	○	×

- 移行非互換対応

移行時の非互換対応としてはスキーマ移行、SQL移行などにおいて一律の対応が必要な部分もあります。これらの非互換対応方法はPGECons 移行WG（WG2）にノウハウをまとめております。以下のURLを参照してください。

https://pgecons-sec-tech.github.io/tech-report/

- 機能レベル差

Oracleからの移行を検討するとき、機能の有無の確認だけでなく機能の詳細まで踏み込んでどこまでできるか確認する必要があります。スライドに示すようにマテリアライズドビューは機能としてPostgreSQLにもありますが、同期方法に差があり通常一般的に利用されることが多い増分リフレッシュはサポートされません。大規模なマテリアライズドビューを利用する場合など完全リフレッシュには時間もかかるため、現実的には別の方式を検討する必要もあります。なお、個々の機能はバージョンが上がることで詳細な機能レベルや性能も向上しております。例えば、パラレル処理はPostgreSQLのメジャーバージョン毎に機能レベルがアップしています。詳細は以下の資料を参考にしてください。

https://pgecons-sec-tech.github.io/tech-report/html_wg2_Parallel/wg2_Parallel.html

（3.2　パラレル処理の機能差）

6. Oracle移行時の注意点

6. 2 Oracleとの仕様差

■ 仕様差

移行元のOracleでの動作結果と比較した場合、異なる結果となるものの代表例を3点示します。

① 除算を含む計算

Oracle	PostgreSQL

SQL> SELECT 1/3*3 結果 FROM dual
結果
--------- 	
1
 	
SQL> SELECT (1/3 + 1/3 +1/3) 結果 FROM dual
結果
--------- 	
1
	test01=# SELECT 1/3*3;
	?column?
	---------- 	
	0
	test01=# select 1.0/3.0*3.0;
	?column?
	------------------------ 	
	0.99999999999999999990
	test01=# SELECT (1/3 + 1/3 + 1/3)
	?column?
	---------- 	
	0
	test01=# select (1.0/3.0 + 1.0/3.0 + 1.0/3.0);
	?column?
	------------------------ 	
	0.99999999999999999999

- Oracleとの仕様差の例について

スライドの例はPGECons 2017年度成果発表会で報告した例になります。

https://www.pgecons.org/wp-content/uploads/2018/05/PGECons_2017_WG2_Seminar_presentation.pdf

- 除算の仕様差

単純な仕様差ですが、移行に関しては影響の大きい仕様差です。例えば、DB内で除算を行っていればシステム移行の結果を移行前と移行後の現新比較のみで行うような場合、計算結果が異なるため一致することはありません。

6. Oracle移行時の注意点

6. 2 Oracleとの仕様差

- **仕様差**

  ② **トランザクションの違い**

  トランザクション分離レベルはどちらもリードコミッティドですが、トランザクション間の書込み競合が発生した場合の挙動に以下の差異があります。

時系列	Oracle		PostgreSQL	
	トラン１	トラン２	トラン１	トラン２
1	SQL> select * from test_t; COL1 COL2 ---------- ---------- 1 1 2 1 3 2 4 2 5 3 5行が選択されました。	SQL> select * from test_t; COL1 COL2 ---------- ---------- 1 1 2 1 3 2 4 2 5 3 5行が選択されました。	test01=# select * from test_t; col1 | col2 ------+------ 1 | 1 2 | 1 3 | 2 4 | 2 5 | 3 (5 行)	test01=# select * from test_t; col1 | col2 ------+------ 1 | 1 2 | 1 3 | 2 4 | 2 5 | 3 (5 行)

2	SQL> update test_t
	set col2 = col2 + 1;
	
	5行が更新されました。
			test01=# begin;
			BEGIN
			test01=# update test_t set col2 =
			col2 + 1;
			UPDATE 5	

6. Oracle移行時の注意点

6. 2 Oracleとの仕様差

■ 仕様差

② トランザクションの違い（つづき）

トランザクション2側の結果が異なることが確認できます。
Oracle:トラン1の時系列4の結果に対してトラン2の時系列3のUPDATEが実行
PostgreSQL:トラン1で対象が更新されたのでトラン2の時系列5は無視されます

時系列	Oracle		PostgreSQL	
	トラン1	トラン2	トラン1	トラン2

3		SQL> delete from test_t
		where col2 = 3;
				test01=# begin;
				BEGIN
				test01=# delete from test_t
				where col2 = 3;

4	SQL> commit; コミットが完了しました。		test01=# commit; COMMIT	
5		**2行が削除されました。** SQL> commit; コミットが完了しました。		**DELETE 0** test01=# commit; COMMIT

- トランザクションの動作の違い

➢ READ COMMITTEDの仕様差

スライドの「トランザクションの違い」が発生する仕様についてはそれぞれマニュアルで以下の説明があります。

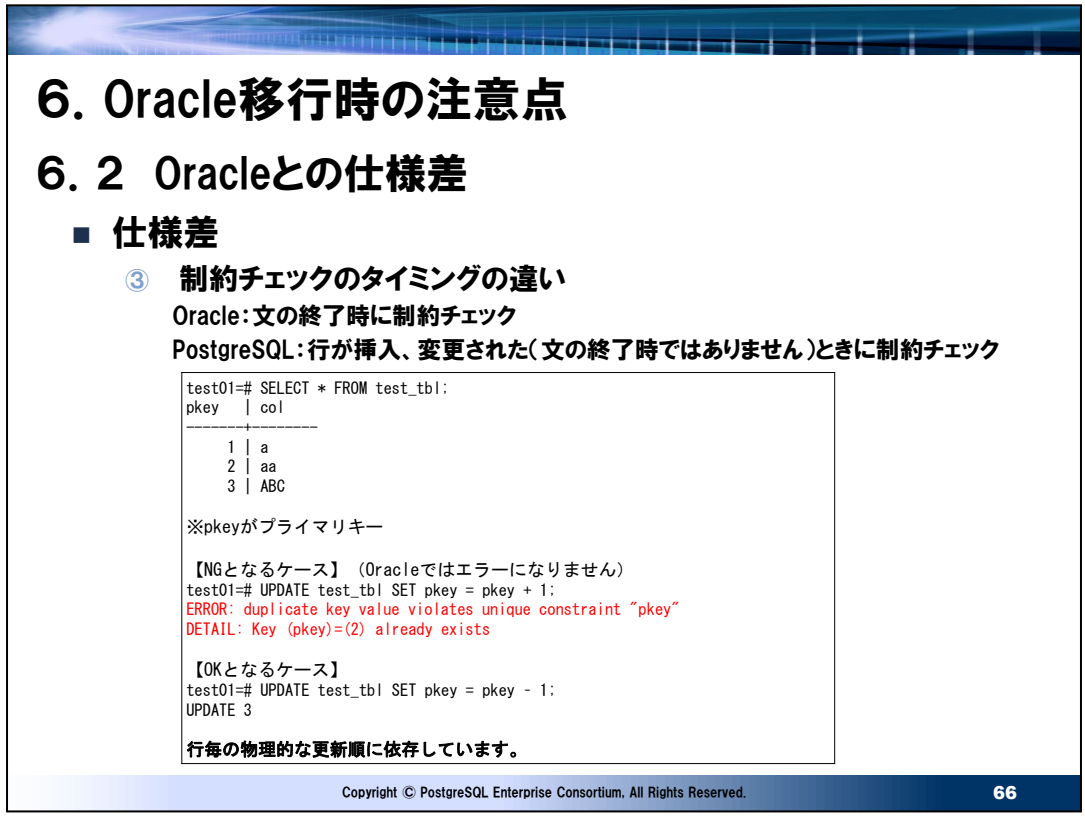
- Oracleの仕様は以下マニュアルの「コミット読取りトランザクション内での書込みの競合」の項
  https://docs.oracle.com/cd/E82638_01/cncpt/data-concurrency-and-consistency.html
- PostgreSQLの仕様は以下マニュアルの「リードコミッティド分離レベル」の項
  https://www.postgresql.jp/document/XX/html/transaction-iso.html
  （XXはバージョン。例　13）

実際には移行元のOracleを利用したシステムでトランザクション中に他のセッションからの更新影響を排除するためにselect ~ for updateなどで書込み競合に対する対策をしている場合が多いと想定しますが、そうでない場合は移行元のOracleの仕様に合わせることは困難になります。

➢ SERIALIZABLEの仕様差

Oracleの場合、以下マニュアルにある通り完全なSERIALIZABLEが提供されてるわけではありません。

https://docs.oracle.com/cd/E82638_01/adfns/sql-processing-for-application-developers.html#GUID-6F888B25-FB84-456F-BF6E-675C955D3E52



● 制約チェックの対応

対応方法は制約検査のタイミングを変更することで可能になります。

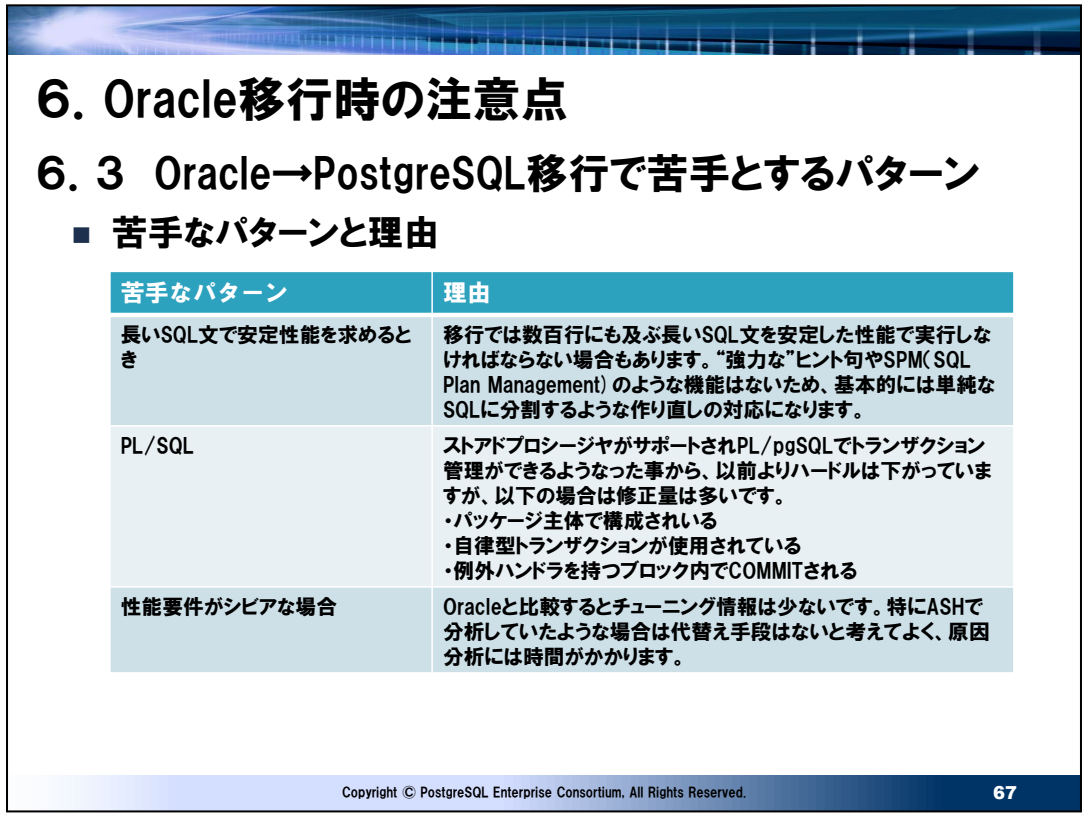
以下はスライドのテーブルを対応したケースになります。

```
test01=# alter table test_tbl drop constraint pkey;
test01=# alter table test_tbl
test01-# add constraint pkey primary key (pkey) deferrable initially immediate;
ALTER TABLE
test01=# UPDATE test_tbl SET pkey = pkey + 1;
UPDATE 3
```

索引を制約チェックをdeferrableにして再作成することでスライドのエラーとなった更新が正常になります。ただし、deferrableにすると以下のようにon conflict句が使えなくなる制限も発生します。

```
test01=# insert into test_tbl values ( 3,'cccccc')
test01-# on conflict ( pkey )
test01-# do nothing;
ERROR:  ON CONFLICT does not support deferrable unique constraints/exclusion constraints as arbiters
```

注意点としてOracleからの移行でmerge文を移行する場合、insert ～ on conflictでの書き換えができなくなるデメリットがあるため、merge文の移行コストが上昇することを見込む必要があります。



- 移行で苦手とするパターン

スライドの表に記載した部分は仕様差以外に、“作り直し”、“仕様変更”、“作業コスト増”を考慮しなければならない部分になります。

- 長いSQL文で安定性能を求めるとき

SQL文に関して言えば、長く利用されたシステムでは複雑に修正の入った、もしくは多くの表を結合し、複数個所にヒント句が使われているような長いSQL文を移行では頻繁に見かけます。結論を言えば、この長く複雑なSQL文を安定した性能で応答させることは困難なため複数の単純なSQL文に書き換える方が、性能の安定性、保守性も向上します。

- PL/SQL

ストアドプロシージャがサポートされたことにより、単純なPL/SQLの移行は容易になりました。ただし、ストアドプロシージャの全てが単純移行可能なわけではなく、パターンによっては大幅な修正が必要になります。この移行方法は以下のPGECons 2019年度成果発表会資料を参照してください。

https://www.pgecons.org/wp-content/uploads/2020/08/0aad0853abb686bac0c25eef9861c4e3.pdf

- 性能要件がシビアな場合

性能チューニングを進めていくと、その対象はインスタンス全体から特定のセッション、SQLへと絞りこまれて行きます。その際、Oracleの性能分析ではASHを利用すればセッションIDやSQL ID単位に対象を絞り込んで分析ができますがPostgreSQLには同等の機能は存在しません。また、長時間かかるSQLをチューニングする場合、実行中のSQLを実行計画中のアクションレベルでの進捗状況の確認したい状況があります。SQLが完了していなくても途中経過で確認できるので想定通りでなければCANCELしてやり方を変えて再実行できます。Oracleにはこのような機能（SQL Developer リアルタイムSQLモニタ）がありますが、PostgreSQLでは実行中のSQLを詳細に分析する機能はありません。このように、分析する手段が限られるため、性能要件の高いシステムではそのチューニングコストがOracleより増加することを考慮する必要があります。

7. 備考

ライセンス

本作品はCC-BYライセンスによって許諾されています。ライセンスの内容を知りたい方はこちらでご確認ください。文書の内容、表記に関する誤り、ご要望、感想等につきましては、PGEConsのサイトを通じてお寄せいただきますようお願いいたします。

- Linux は、Linus Torvalds 氏の日本およびその他の国における登録商標または商標です。
- Microsoft、Windows Server、SQL Server、米国Microsoft Corporationの米国及びその他の国における登録商標または商標です。
- Oracleは、Oracle Corporation 及びその子会社、関連会社の米国及びその他の国における登録商標です。文中の社名、商品名等は各社の商標または登録商標である場合があります。
- PostgreSQLは、PostgreSQL Community Association of Canadaのカナダにおける登録商標およびその他の国における商標です。
- その他、本資料に記載されている社名及び商品名はそれぞれ各社が商標または登録商標として使用している場合があります。

著者

版	所属企業・団体名	部署名	氏名
第1．0版	富士通Japan株式会社	民需ソリューション開発本部	多田　明弘
第2．0版	富士通Japan株式会社	ﾋﾞｼﾞﾈｽｿﾘｭｰｼｮﾝ開発本部	多田　明弘

PGECons

PostgreSQL Enterprise Consortium