PostgreSQLエンタープライズ・コンソーシアム 技術部会 WG#2

WG2活動報告書
移行ガイドブック

目次

1. 改訂履歴

版	改訂日	変更内容
1.0	2019/03/25	新規作成
1.1	2019/05/20	「6.2.4 データベース・オブジェクトの違いについて」の、表6.2の内容を修正
1.2	2020/03/23	「6.2. アプリケーション移行」: 標準規格I/F対応の追加と全体構成の変更

2. ライセンス

3. はじめに

3.1. 本資料の概要と目的

本資料は異種DBMSからPostgreSQLへの移行を検討される方の参考にしていただくことを目的に、 PostgreSQLエンタープライズ・コンソーシアム(以下PGECons)が作成・公開をしています。

PGEConsは、PostgreSQL本体および各種ツールの情報収集と提供、整備などの活動を通じて、 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の普及を 推進することを目的として設立された団体です。

PGEConsの技術部会では、PostgreSQLの普及に対する課題の検討を通じて活動テーマを挙げ、 そのテーマごとにワーキンググループを立ち上げて活動しています。その中の一つである WG2(移行ワーキンググループ)では、「異種DBMSからPostgreSQLへの移行」をテーマとして 調査・検証を行い、収集した技術ノウハウを成果として取り纏めた資料を公開してきました。

しかし、WG2の活動でこれまでに積み上げてきた資料は膨大なページ数になっており、 移行の検討をする初期段階に参考にするのは難しいのではないかという課題が挙がりました。 そこで、PostgreSQLへの移行を検討する方がはじめに読むにあたり、移行の全体像を つかむことができるような資料として、これまでに公開した資料の要素に加え、あまり 触れられていなかった運用面でのポイントを整理し、移行ガイドブックという形でまとめました。

本資料が皆様のPostgreSQL採用検討の一助になれば幸いです。

3.2. 本資料の構成

本資料の本文の構成は下表の通りです。

表 3.1 本資料の構成

章	概要
3 はじめに	本資料の位置づけと概要説明
4 PostgreSQL紹介	PostgreSQLの簡単な紹介
5 データベース移行作業とは	データベース移行の考え方や工程の概要について説明
6 アセスメント	データベース移行のアセスメントに必要となる観点について説明
7 移行作業	実際にデータベース移行開発を行うにあたり発生する作業について説明
8 運用	PostgreSQLの運用に必要となる作業の概要について説明

3.3. 本資料で扱うRDBMS

本資料ではデータベース移行作業の概要を5章で示し、6章以降の具体的な内容についてはOracleからPostgreSQLへの移行を題材として記載しています。 各RDBMSの情報については下表のバージョンをベースに記載をしていますが、極力バージョンに左右されない基本的・共通的な内容を中心にしています。 より新しいバージョンに関する内容については、本文中で補足をしています。

表 3.2 本書で扱うRDBMS

RDBMS名称	バージョン
PostgreSQL	10
Oracle Database	11gR2

4. PostgreSQL紹介

本章ではPostgreSQLについて簡単にご紹介します。

4.1. 概要

PostgreSQL（Postgres）はオープンソースソフトウェア（OSS）として開発されているRDBMSです。 年1回のペースで新しいバージョンをリリースし、年々機能強化・性能改善を行い進化を続けています。 最近のバージョンでは、パーティショニングやパラレルクエリなど、大規模データベース向けの機能強化が進んでいます。 他にも多くの機能が実装されており、機能面では商用データベースに対しても遜色のないものになってきています。

PostgreSQLはThe PostgreSQL Global Development Group（PGDG）というコミュニティにより開発されています。 このコミュニティには世界各国の開発者が参加しており、メーリングリストなどで活発な議論が行われています。 コミュニティとしての活動のため、特定の企業に依存するものではなく、開発の方向性は開発者による議論の中で決められていきます。

また、PostgreSQLのコミュニティとしては、世界各国にユーザコミュニティが存在しています。 日本では早くから日本PostgreSQLユーザ会（JPUG）が設立され、PostgreSQL関連のセミナーを開催したり、 日本語版マニュアルの翻訳・公開、Let's Postgresというポータルの運営などの活動をしています。

日本語マニュアルをはじめ、PostgreSQLについては日本語の情報が多く公開されており、 日本では比較的PostgreSQLの普及が進んでいる傾向にありました。 近年、企業によるOSS採用が伸びていくにあたり、PostgreSQLの利用はさらに広がってきています。

4.2. PostgreSQLライセンス

PostgreSQLは「PostgreSQLライセンス」というライセンスの下に公開されています。 PostgreSQLライセンスはBSDやMITライセンスに似たオープンソース・ライセンスであり、単純に使用する分には無償であることはもちろん、 著作権や免責事項などのライセンス条件の表記の複製を添付することで再頒布・再利用をすることが可能です。 PostgreSQLを改造して組み込み利用するようなケースにおいても、ソースコードを公開する必要はありません。

開発コミュニティはライセンスについて、今後変更する予定はないことを明言しています。 ライセンス条件が変わる可能性を意識せずに、長期的に使用することができます。

また、PostgreSQLは無償で利用することができますが、商用データベースのように保守サポートが提供されるわけではありません。 安心して運用を続けるためには保守サポートについて別途検討する必要があります。 PostgreSQLに対しては、多くの企業が保守サポートをはじめとした様々なサービスを提供しています。

4.3. バージョン

PostgreSQLのバージョンは X.Y 形式（※）で表記されます。 Xがメジャーバージョンを、Yがマイナーバージョンを表し、2019年4月現在の最新メジャーバージョンは11です。

※ バージョン10より前のバージョンでは X.Y.Z 形式で表記され、X.Yの部分がメジャーバージョンを示していました。

PostgreSQLでは概ね年に1回、新しいメジャーバージョンがリリースされます。 メジャーバージョンのリリースでは、新機能の追加や性能改善などが主に実施され、互換性に影響のある内容を含みます。 メジャーバージョンアップを行う際には、旧バージョンのデータをそのまま使用することはできません。 バックアップリストア、もしくはツール（pg_upgrade）を使用して移行をする必要があります。

また、PostgreSQLは3か月に1回、計画的に新しいマイナーバージョンのリリースをしています。 それ以外にも、脆弱性などの重大な問題があれば、スケジュール外のリリースが行われる可能性があります。 マイナーバージョンのリリースでは主にバグの修正が取り込まれており、原則として互換性に影響のある修正は行われません。

コミュニティでは各メジャーバージョンについて、リリースされてから約5年間サポートを継続しています。 5年後に最終リリースを迎えたバージョンについては、原則新しいリリースが行われることはありません。 （非常に影響の大きい問題の場合は対応される可能性はあります。） サポートの終了に合わせて、計画的にバージョンアップを行うことを推奨します。

4.4. 導入方法

PostgreSQLは主要なCPUアーキテクチャ、およびOSでの動作をサポートしています。 主なプラットフォームに関しては、ビルドファームという検証用のサーバ群にて日々動作検証が行われています。

PostgreSQLのソースおよびバイナリパッケージは、下記PostgreSQLのWebサイトから入手することができます。

- https://www.postgresql.org/download/

Linuxディストリビューション向けには、RPMなどのインストール用パッケージが提供されています。 Yumなどのパッケージ管理システム用のリポジトリも提供されているため、そちらを指定してインストールすることも可能です。 Windowsについてもインストーラが提供されており、PostgreSQLのインストール方法は非常に簡単です。

また、PostgreSQLはOSSであり、ソースコードが公開されているため、自らビルドをして導入することができます。 各プラットフォームにおいてバイナリパッケージが用意されているため、そちらを使用するのがシンプルではありますが、 ブロックサイズなどビルド時のみ変更可能なパラメータも存在するため、そのようなケースではソースからの導入が必要になります。

4.5. 周辺ツール

PostgreSQL本体以外にも、PostgreSQLの機能を補完する周辺ツールがOSSとして開発・提供されています。 PostgreSQLに組み込んで機能を拡張するものや、PostgreSQLと連携して動作するものなど、多岐にわたるツールが存在しています。 RDBMSの移行という点では、Oracleと同じような機能や運用性を提供するツール、Oracleからの移行作業を補助するツールなどが有用です。

PostgreSQL向けに提供されている周辺ツールの一部について、下表にてご紹介します。 ただし、Windowsではサポートされていないものが多いためご注意ください。

表 4.1 PostgreSQLの周辺ツール

分類	ツール名	概要
移行	ora2pg	OracleからPostgreSQLへの移行ツール。移行評価レポート、定義・SQLの変換、データの移行などの機能を提供。
	Orafce	Oracleが提供するパッケージや組み込み関数をPostgreSQL上で代替する拡張機能。
性能	pg_dbms_stats	PostgreSQLの統計情報を固定する拡張機能。通常ANALYZEで採取する統計情報をダミー情報で固定化することで、予期しない実行計画の変更を防ぐことが可能。
	pg_hint_plan	PostgreSQLにヒントを実装する拡張機能。ヒント句を使用してスキャンや結合の方式を指定することで、SQLの実行計画を制御することが可能。
運用・監視	pgBadger	PostgreSQLのログファイルを解析して、SQL実行状況などのHTMLレポートを生成。
	pg_monz	ZabbixによるPostgreSQL監視のためのテンプレートを提供。
	pg_repack	通常は排他ロックを必要とするテーブルやインデックスの再編成を、排他ロックをかけずに実行可能とする拡張機能。
	pg_rman	PostgreSQLのバックアップ・リカバリ実行の簡易化、バックアップの世代管理など、バックアップ運用を補助するツール。
	pg_statsinfo	PostgreSQLの稼働統計情報のスナップショットを定期的に収集・蓄積し、データベースの処理状況、性能傾向などの確認に利用可能。レポート出力機能も提供。
その他	pg_bigm	PostgreSQLに日本語対応の全文検索機能を提供する拡張機能。
	pg_bulkload	PostgreSQLに対して大量データの高速ロードを可能とするツール。
	pgpool-II	PostgreSQLのサーバ・クライアント間で動作するミドルウェア。コネクションプール、レプリケーション、負荷分散などの機能を提供。
	PostGIS	PostgreSQL上で地理情報データを取り扱うための拡張機能。

5. データベース移行作業とは

5.1. データベース移行の考え方

異種データベース間の移行ではデータベースの優劣を比較分析するのではなく、データベースを利用している業務システムの利用目線で「As-Is（現状）」と「To-Be（あるべき姿）」に対するFit&Gap分析をして、課題を明らかにします。

そして課題に対してどう対応するのか、つまりTo-Beに近づくために何をすれば良いのかを考えることになりますが、To-Beはあくまでも理想形であり、理想を追求して膨大な修正費用がかかっては意味がありません。
対応が必要な機能・作業に優先順位をつけ、現実的な落としどころ「Can-Be（現実的解）」を見つけることが重要です。

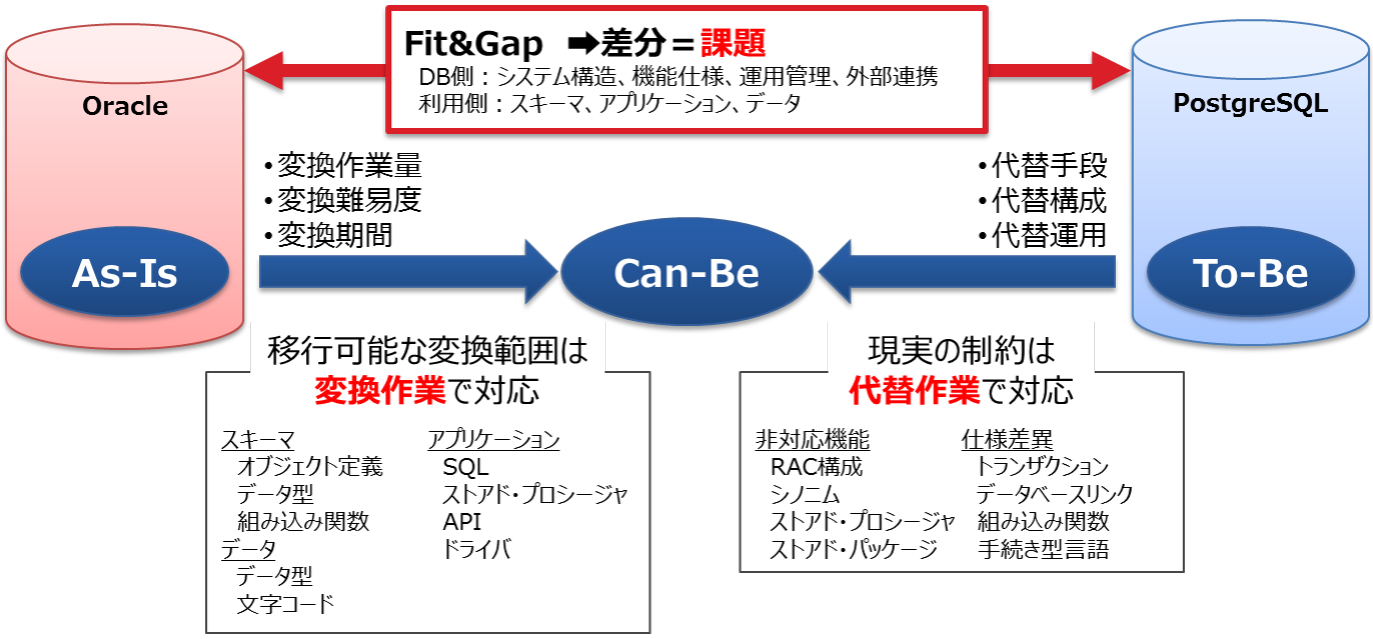


図 5.1 DB移行における課題と対応の考え方

5.2. データベース移行の進め方

データベース移行作業の大まかな流れは以下の図の通りです。

実際の移行作業に取り掛かる前に、前述の考え方に従って移行時の作業内容、作業量を調査する「アセスメント」を実施する必要があります。アセスメントの結果をもとに移行を行うかどうか判断し、実際の移行作業に取り掛かります。
移行作業は、移行対象であるスキーマ、アプリケーション、データに対する変換作業や代替作業などを実施し、その後移行検証、運用の流れで進めます。

図 5.2 データベース移行フロー

移行フローの各工程の作業目的と、主要な作業項目は下表の通りです。

表 5.1 移行フローと作業項目

移行工程	目的	主要な作業項目
アセスメント	移行目的の達成可能性を確認し、データベース移行の可否を判断する	・システム品質要求の適合性の把握 ・移行難易度の把握 ・移行コストの見積り ・移行可否判定
スキーマ移行	テーブル、ビュー、インデックス、ストアド・プロシージャなどのデータベース・オブジェクトを移行する	・データベースの構築 ・データベースオブジェクトの定義移行 ・他DB間の連携（データベースリンク）
アプリケーション移行	データベース変更によるAPI、SQL文等の差異を解消する	・データベース接続、ドライバの変更 ・SQL、組み込み関数、ストアド・プロシージャの改修 ・コマンド、API、ツールの付け替え
データ移行	移行元DB上のデータを移行する	・データ型、文字コードの変換 ・データクレンジング ・データの抽出、投入方法検討
移行検証	データベース変更による影響確認を行う	・機能テスト、非機能テスト ・パフォーマンスチューニング
運用	データベース変更によるシステム運用保守を新たに構成する	・運用保守設計の修正（運用処理、稼働監視）

5.3. 開発工程におけるデータベース移行

移行フローの各工程において、データベース移行作業をどのように進めるか、一般的な開発工程にマッピングした例を以下に示します。
アプリケーションの修正がなくSQLの非互換修正のみであっても、テスト工程は実施する必要があります。

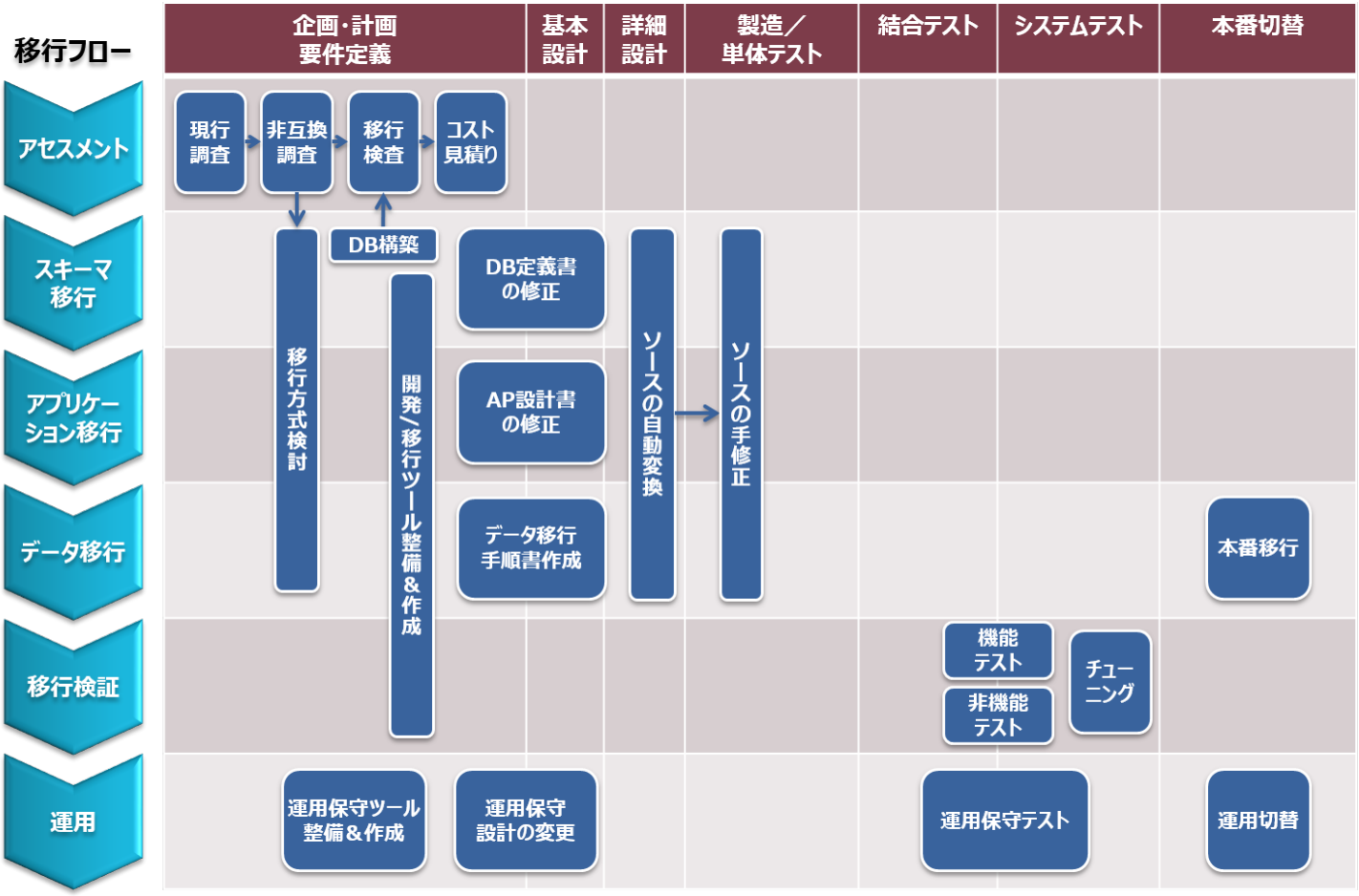


図 5.3 開発工程におけるデータベース移行作業

6. アセスメント

6.1. アセスメントとは

アセスメントとは、現行環境のSQLやアプリケーションに対して、 移行が必要な個所がどこなのか、どういった非互換があるのかを調査し、 移行の作業量や難易度を算出する作業を指します。

次の観点でアセスメントを実施します。

- システム品質要求に適合するか
    - 可用性
    - 運用、保守性
    - 性能、拡張性
    - セキュリティ など
- 移行難易度はどの程度か
    - スキーマ
    - アプリケーション
    - データ
- 移行コストはどのくらいかかるか
    - 工数

また、調査対象の例は次のとおりです。

- 要件定義書
- 各種設計書
- DDLやPL/SQL、アプリケーションなどのソース

具体的な調査内容については次節を参考にしてください。
調査時にはデータベース部分だけではなく、OSや周辺ミドルウェアなどについても合わせて調査を行ってください。

そして、ここで実施したアセスメントの結果を基に、データベース移行を実施するかどうかの判断を行います。

6.2. 非互換調査(製品比較)

アセスメントをするにあたって、RDBMS間の違いを把握しておく必要があります。
以降では以下の主要な違いについて説明します。

- アーキテクチャ
- スキーマ
- ユーザ
- データベース・オブジェクト
- SQL (データ型やSQL文、組み込み関数など)
- SQL手続き言語
- トランザクション

6.2.1. アーキテクチャの違い

アーキテクチャは各RDBMSごとに異なっており、それぞれ特色があります。 移行にあたり、その差異を理解しておく必要があります。本節では主なものをピックアップして説明します。

■プロセス構成

PostgreSQL、Oracle共にクライアント・サーバ型、マルチプロセス構成です。
細かなプロセス構成に違いはありますが、移行に影響することは特にありません。

■メモリ構成

PostgreSQLは、共有メモリ領域とプロセス毎のメモリ領域から構成されます。メモリ管理は手動管理で、データベースクラスタの制御ファイル(postgresql.conf)のパラメータに指定した値が用いられます。
Oracleは、インスタンス内で共有されるメモリ領域(SGA)とプロセス毎に固有のメモリ領域(PGA)から構成されます。メモリ管理は自動管理と手動管理の2種類です。

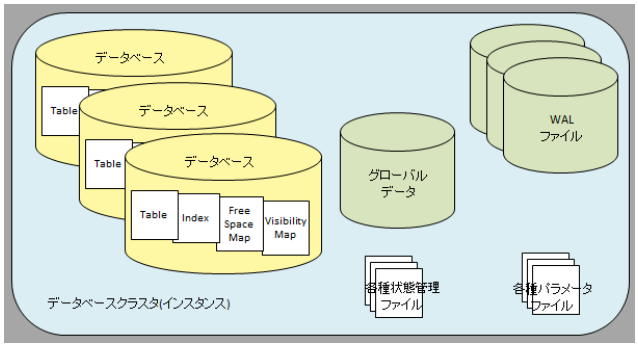
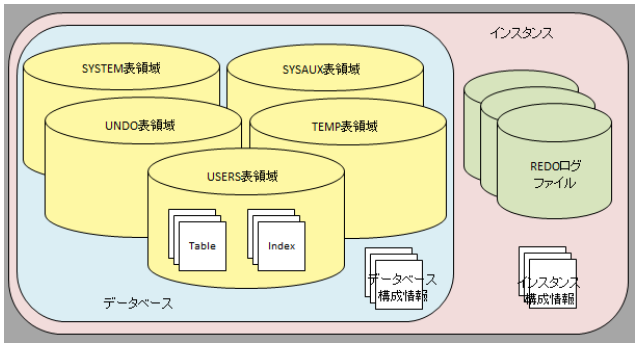
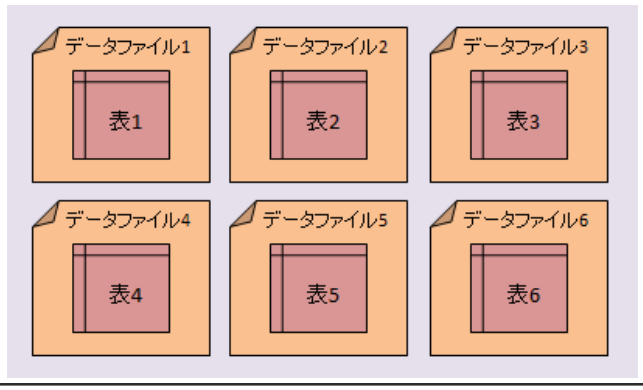
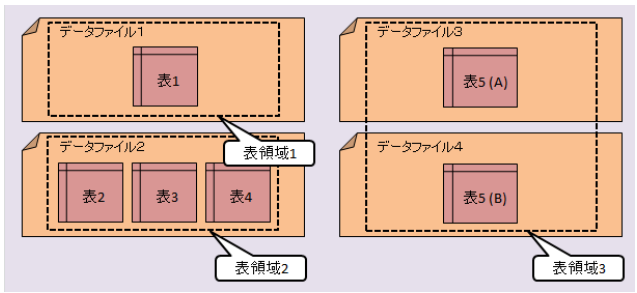
メモリの管理方法は異なりますが、メモリ構成の違いが移行に影響することはありません。

■データ構成

PostgreSQLとOracleでは、「データベース」、「インスタンス」の考え方が異なります。また図に示すように、データ構成も異なります。
データ構成の違いは移行の難易度を大きく左右するものではありませんが、データ移行の設計に関わるため、その差異を理解しておく必要があります。

- PostgreSQL
  - データベースの集合を「データベースクラスタ」として管理します。これはOracleで「インスタンス」と呼ばれるものに相当します。
  - 1つのデータベースクラスタは複数のデータベースを管理することが可能なため、データベースクラスタ(インスタンス)とデータベースは1対Nの関係になります。
  - バックグラウンドプロセス、WAL、設定ファイルなどはデータベース間で共有されます。
- Oracle
  - インスタンスとデータベースは1対1の関係で、1つのインスタンスにつきインスタンス構成情報とデータベース構成情報を1つずつ持ちます。
  - [Oracle12c以降] マルチテナントの場合、1つのCDB(インスタンスに対応)は複数のPDBを管理することが可能です。バックグラウンドプロセス、REDOログ、制御ファイルなどはPDB間で共有されます。

表 6.1 データ構成比較

	PostgreSQL	Oracle
データの構成		
データファイルの構造		

6.2.2. スキーマの違い

PostgreSQLとOracleはともにスキーマを持ちますが、考え方などに違いがあります。
本節では主要な違いをピックアップして説明します。

■スキーマの考え方

表などのオブジェクトはスキーマに所属する形で管理されますが、OracleとPostgreSQLで違いがあります。
Oracleではユーザとスキーマが1対1で対応していますが、PostgreSQLではユーザとスキーマが対応づいていません。

■オブジェクト作成時に割り当てられるスキーマ

Oracleでは"ユーザ名"と同じ名前のスキーマが自動的に作成され、割り当てられます。
PostgreSQLでは"ユーザ名"のスキーマが自動的に作成されることはありません。CREATE SCHEMA文により明示的に定義することは可能です。

■検索時のスキーマの優先順位

Oracleでは"ユーザ名"のスキーマを検索します。
PostgreSQLではsearch_pathパラメータで設定します。デフォルトでは"ユーザ名"、"public"スキーマの順です。スキーマ名を指定していない場合、search_pathに指定された順にオブジェクトの検索を行います。

6.2.3. ユーザの違いについて

PostgreSQLとOracleは共にユーザを作成して利用しますが、考え方などに違いがあります。
本節では主要な違いをピックアップして説明します。

■ユーザの考え方

PostgreSQLとOracleではユーザの考え方が異なります。

PostgreSQLではユーザはロールの一種で、LOGIN属性を持つロールを指します。 Oracleと同様に、LOGIN属性を持つロール（ユーザ）を別のロールに所属させることも可能です。

■事前定義されるユーザ・アカウント

Oracleでは、インストール時に事前定義済のアカウントが複数作成されますが、 PostgreSQLではデータベース作成時に指定したアカウントのみが管理者として作成されます。 また、データベースサーバを監視するロールを簡単に設定できるデフォルトロールが定義されています。

PostgreSQLで管理者以外のユーザ・アカウントが必要な場合は、CREATE ROLE文またはCREATE USER文で作成します。

6.2.4. データベース・オブジェクトの違いについて

表や索引などの基本的なオブジェクトはPostgreSQLでも使用可能です。 一方、シノニムやデータベース・リンクなどのオブジェクトはPostgreSQLにありません。 これらのPostgreSQLにないオブジェクトを使用している場合は、移行方法の事前検討や移行工数を多く見積もるなどの対応を行う必要があります。

データベース・オブジェクトの対応については以下のとおりです。

表 6.2 DBオブジェクトの対応表

Oracle	PostgreSQL	補足
テーブル	○	グローバルー時表は非対応。パーティショニングは、レンジ/リスト相当のみ。
インデックス	○	逆キー索引、ビットマップ、ドメイン、クラスタ、索引構成表は非対応。
ビュー	○	
マテリアライズドビュー	○	参照のみ。自動リフレッシュ非対応。
シノニム	×	ビューで代用可能。
シーケンス	○	
トリガー	○	
データベースリンク	×	FDW(Foreign Data Wrapper)またはdblink関数で代替可能。
ストアドプロシージャ	○	手続き型言語の仕様相違の対処が必要。バージョン11よりサポート。
ストアドファンクション	○	トランザクション制御と手続き型言語の仕様相違の対処が必要。
パッケージ	×	代替方法はスキーマを使用して関数群をまとめる。パッケージレベルの変数はセッションごとの状態を一時テーブル内部に保存する。
ユーザ	○	ユーザはロールに包括されている。
ロール	○	ロール権限とオブジェクト権限があり、一部を除き非互換のために対処が必要。

○：存在する　×：存在しない

6.2.5. SQLの違い

PostgreSQLもOracleもどちらも標準SQL(Core SQL)に完全または部分的に準拠しています。 しかし、独自機能などの非互換があるため、調査を行う必要があります。

以降では詳細について説明します。

6.2.5.1. データ型

データ型についてはそのまま使用できるものもありますが、移行が必要なものも多くあります。 また、同じ名前のデータ型が存在する場合で

も、有効桁数などに違いがある場合があるため注意が必要です。

主要なデータ型の対応については以下のとおりです。

表 6.3 データ型の対応表

属性	Oracle	PostgreSQL	PostgreSQLのデータ型についての説明
文字	VARCHAR2	varchar	上限付き可変長
	CHAR	char	空白で埋められた固定長
	CLOB	text	可変長（最大 1 GB）
真数	NUMBER	decimal	小数点前までは131072桁、小数点以降は16383桁
		numeric	小数点前までは131072桁、小数点以降は16383桁
		integer	整数(-2147483648～+2147483647)
概数	NUMBER	real	6桁精度
		double precision	15桁精度
	FLOAT	float	精度(2進数53桁)
日時	DATE	timestamp	日付と時刻の両方（時間帯なし） 4713BC～294276AD（１μ秒、１４桁）
		date	日付のみの場合
	TIMESTAMP	timestamp	日付と時刻の両方（時間帯なし） 4713BC～294276AD（１μ秒、１４桁）
	TIMESTAMP WITH TIMEZONE	timestamp [(p)] with time zone	日付と時刻の両方、時間帯付き、 4713BC～294276AD（１μ秒、１４桁）
	INTERVAL YEAR TO MONTH	interval [fields] [(p)]	-178000000年～+178000000年（１μ秒、１４桁）
	INTERVAL DAY TO SECOND	interval [fields] [(p)]	-178000000年～+178000000年（１μ秒、１４桁）
バイナリ	BLOB	bytea	可変長のバイナリ列（最大 1 GB）
その他	ROWID	対応なし	

6.2.5.2. 組み込み関数

組み込み関数についてはそのまま使用できるものもありますが、移行が必要なものも多くあります。 また、同じ名前・機能の組み込み関数が存在する場合でも、 パラメータや結果のデータ型などに違いがある場合があるため注意が必要です。

以下に、主要な関数の対応例を示します。

表 6.4 組み込み関数の対応表

Oracle	互換性	PostgreSQLでの対応方法
ABS(n)	○	
MOD(m,n)	○	
ROUND(数値)	○	
TRUNC(数値)	○	
CHR(n)	○	
CONCAT(char1,char2)	○	
LOWER(char1)	○	
REGEXP_REPLACE(string,pattern[,replace[,pos[,occurrence[,match]]]])	△	regexp_replace(string text,pattern text,replacement text[,flags text])
REGEXP_SUBSTR(source_char,pattern[,postion[,occurrence[,match_param[,subexpr]]]])	△	regexp_matches(string text,pattern text[,flags text])
REPLACE(char,search_string,replacement_string)	○	
SUBSTR(char,m,n)	○	
TRIM([LEADING|TRAILING|BOTH] [trim_character] FROM trim_source)	○	

Oracle	互換性	PostgreSQLでの対応方法
UPPER(char)	○	
ASCII(char)	○	
INSTR(string,substring)	△	strpos(string,substring)
LENGTH(char)	○	
ADD_MONTHS(date,integer)	△	+演算子を使って書換え可能 例: select date '2018-03-22' + interval '1 months'
CURRENT_DATE	○	
CURRENT_TIMESTAMP	○	
SYSDATE	△	current_date、current_timestamp、clock_timestamp
SYSTIMESTAMP	△	systimestamp current_timestamp、clock_timestamp
CAST(expr AS type_name)	○	
CONVERT(char,dest_char_set,source_char_set)	○	convert(string bytea,src_encoding name,dest_encoding name)
TO_CHAR(d,fmt)	○	
TO_CHAR(n,fmt)	○	
TO_DATE(char,fmt)	○	
TO_NUMBER(char,fmt)	○	
TO_TIMESTAMP(char,fmt)	○	
DECODE(expr,search,result)	△	case式で置き換える
NVL(expr1,expr2)	△	coalesce(expr1,expr2)
AVG(expr)	○	
COUNT(expr)	○	
MAX(expr)	○	
MIN(expr)	○	
RANK() OVER (ODER_BY_clause)	△	rank()
SUM(expr)	○	

○：あり、△：別の方法で代替可能

6.2.5.3. SQL文

1.データ定義言語(DDL)

PostgreSQLにも存在するオブジェクトについては、 CREATE文やDROP文、ALTER文などを使用する点はOracleと同様です。 ただし、一部の構文やオプションに違いがあるため、 実際のシステムで使用している構文を確認し、非互換があるかどうか調査を行う必要があります。

表 6.5 DDL対応表

Oracle	文の有無	備考
ALTER	○	PostgreSQLに存在するオブジェクトに対するもののみ
ANALYZE	△	構文に違いあり
ASSOCIATE STATISTICS	×	
AUDIT	×	
COMMENT	○	
CREATE ...	○	PostgreSQLに存在するオブジェクトに対するもののみ
DISASSOCIATE STATISTICS	×	
DROP ...	○	PostgreSQLに存在するオブジェクトに対するもののみ
FLASHBACK ...	×	
GRANT	△	システム権限が対象外など付与可能な権限に違いあり
NOAUDIT	×	
PURGE	×	
RENAME	×	
REVOKE	△	システム権限が対象外など取り消し可能な権限に違いあり
TRUNCATE	○	省略可能な構文に違いあり

○：あり、△：文はあるが書き換えが必要もしくは一部機能がない、×：なし

2.データ操作言語(DML)

基本的なSELECT文、INSERT文、UPDATE文、DELETE文はPostgreSQLでも使用可能です。 ただし、DDL同様に一部の構文やオプションに違いがあるため、 実際のシステムで使用している構文を確認し、非互換があるかどうか調査を行う必要があります。

表 6.6 DML対応表

Oracle	文の有無	備考
SELECT	○	UNIQUE句がないなど構文に違いあり
INSERT	○	ALL INTO句がないなど構文に違いあり
UPDATE	○	ONLY句がないなど構文に違いあり
DELETE	○	FROM句の省略不可など構文に違いあり
MERGE	×	INSERT ON CONFLICTで代替するなどの対処が必要
CALL	×	PostgreSQL11で追加
EXPLAIN PLAN	△	EXPLAIN文を使用する
LOCK TABLE	○	PostgreSQLではLOCK文

○：あり、△：文はあるが書き換えが必要もしくは一部機能がない、×：なし

3.その他制御文

■トランザクション制御文

COMMIT文、ROLLBACK文、SAVEPOINT文についてはPostgreSQLでも使用可能です。 ただし、使用できない句があるなど違いがあるため、調査が必要です。

表 6.7 トランザクション制御文対応表

Oracle	文の有無	備考
COMMIT	○	COMMENT句がないなど構文に違いあり
ROLLBACK	○	FORCE句がないなど構文に違いあり
SAVEPOINT	○	同じ名前のセーブポイントを作成した場合の動作に違いあり
SET TRANSACTION	△	ISOLATION LEVEL句の場合のみ
SET CONSTRAINT	○	SET CONSTRAINTSのみ使用可能

○：あり、△：文はあるが書き換えが必要もしくは一部機能がない、×：なし

■セッション制御文

そのままでは使用できません。そのため、実現したい処理に応じた移行法を検討する必要があります。

■システム制御文

そのままでは使用できません。そのため、実現したい処理に応じた移行法を検討する必要があります。

6.2.5.4. 演算子・条件・結合・疑似列

演算子や条件、結合、疑似列について説明します。

そのまま使用できる演算子や条件は多くありますが、 書き換えが必要なものや動作に違いのあるものも存在するため、 実際にどういったものを使用しているか調査を行います。

以下にそれぞれの例を示します。

■そのまま使用できる例

表 6.8 そのまま使用できる例

項目	例
演算子	+, -, *, / (算術演算子) UNION, UNION ALL (集合演算子)
条件	- =, <, >, <=, =>, !=, <>, ANY, SOME, ALL (比較条件) NOT, AND, OR (論理条件) LIKE (パターン一致条件) IS NULL, IS NOT NULL (NULL条件) BETWEEN, EXISTS, IN
結合	JOIN, INNER JOIN, LEFT [OUTER] JOIN, RIGHT [OUTER] JOIN, FULL OUTER JOIN

■書き換えが必要な例

表 6.9 書き換えが必要な例

項目	例	PostgreSQLでの書き換え例
演算子	MINUS (集合演算子)	EXCEPTに書き換え
条件	^= (比較条件)	<>条件に書き換え
結合	(+) (外部結合演算子)	JOINに書き換え
疑似列	ROWNUM	LIMIT、OFFSETに書き換え

■動作に違いのある例

表 6.10 動作に違いのある例

項目	例	差異
演算子	|| (連結演算子)	NULLが含まれる場合の結果に違いあり

6.2.5.5. その他

その他にもPostgreSQLとOracleには以下のような違いがあります。

1. NULLと空文字列

Oracleでは空の文字列はNULLと同値として扱われますが、これは標準SQLに準拠したものではありません。
PostgreSQLでは変換されず空文字列のまま格納されます。
PostgreSQLでOracleと同じように空文字列をNULLと見なしたい場合には、NULLIF関数を使って変換する方法があります。

・空文字列をNULLとみなす検索例

```
SELECT * FROM staff WHERE NULLIF(name, '') IS NOT NULL;
```

2. REGEXP_LIKE条件による正規表現マッチング

Oracleで、POSIX正規表現規格のマッチングを行う場合、REGEXP_LIKE条件を用います。
PostgreSQLでPOSIX正規表現のマッチングを行う場合には正規表現マッチ演算子を用います。

表 6.11 【正規表現マッチ演算子】

演算子	説明	例 （結果はすべて真)
~	正規表現に一致、大文字小文字の区別あり	'thomas' ~ '.*thomas.*'
~*	正規表現に一致、大文字小文字の区別なし	'thomas' ~* '.*Thomas.*'
!~	正規表現に一致しない、大文字小文字の区別あり	'thomas' !~ '.*Thomas.*'
!~*	正規表現に一致しない、大文字小文字の区別なし	'thomas' !~* '.*vadim.*'

POSIX正規表現を使って p で始まるか e が２回現れる名前を検索する例は以下のとおりです。

・Oracleの例

```
SELECT * FROM staff WHERE REGEXP_LIKE(lower(name), '^p|(e.*){2}');
```

・PostgreSQLの例

```
SELECT * FROM staff WHERE lower(name) ~ '^p|(e.*){2}';
```

3. 除算を含む計算

PostgreSQLでは乗除算は前方より順に行われます。このため割り切れない除算は数値型の精度に依存し、丸め処理が行われた後に後方の計算が行われるため、Oracle と異なる結果になります。
PostgreSQL内では除算を含む計算をさせず、結果のみを格納するような変更が必要となります。

・Oracleの例

```
SQL> SELECT 1/3*3 AS result FROM dual;
   RESULT
----------
         1

SQL> SELECT (1/3 + 1/3 +1/3) AS result FROM dual;
   RESULT
----------
         1
```

・PostgreSQLの例

```
postgres=# SELECT 1/3*3 AS result;
 result
--------
      0

 postgres=# SELECT 1.0/3.0*3.0 AS result;
         result
------------------------
 0.99999999999999999990

 postgres=# SELECT (1/3 + 1/3 + 1/3) AS result;
 result
--------
      0

 postgres=# SELECT (1.0/3.0 + 1.0/3.0 + 1.0/3.0) AS result;
 result
------------------------
 0.9999999999999999999
```

4. SEQUENCEキャッシュ動作

PostgreSQLのSEQUENCEキャッシュは、セッション単位にあらかじめ番号を割り当て、メモリに格納することでアクセスを速くする仕組みで

す。そのために、複数のセッションで同時に使用すると連番にはなりません。
これを回避するためには、キャッシュを使用しないようにキャッシュサイズを 1 にします。

5. 複合一意制約のNULL動作

複合一意制約を設定した場合に、これらの 1 つの列がNULLとなるデータを重複して登録しようとした場合、Oracleは一意制約違反となります。
PostgreSQLでは重複して登録されます。これはNULLは未知の値を表し、2つの未知の値が等しいかどうかを判断することはできないからです。

6. 暗黙的な型変換

PostgreSQLは基本的に暗黙的な型変換をしないので、明示的にCASTする必要があります。

以下にCASTが必要な例を示します。

・NGの例

```
postgres=# SELECT 1 + '1.0' AS result;
 ERROR:  invalid input syntax for integer: "1.0"
 LINE 1: SELECT 1 + '1.0' AS result;
```

・OKの例

```
postgres=# SELECT 1 + CAST('1.0' AS numeric) AS result;
 result
 --------
   2.0
```

6.2.6. SQL手続き言語の違い

OracleではSQLの手続き型拡張機能であるPL/SQLがよく使用されますが、PostgreSQLではPL/pgSQLを使用します。
PL/SQLとPL/pgSQLは似ている点も多いですが、以下の例のように大きな違いがあります。そのため、これらのPL/pgSQLにない機能を使用していないか、使用していた場合は代替処理が行えるのか検討が必要です。

- PL/SQLとPL/pgSQLの違い(例)
    - パッケージを作成できない
    - Oracle社が提供するPL/SQLパッケージが使用できない
    - トランザクション制御ができない (PostgreSQL10まで)
    - プロシージャが作成できない (PostgreSQL10まで)
    - コレクション型などの一部データ型がない
    - エラー・コードやエラー・メッセージが違い、関数ではなく変数から取得する

6.2.7. トランザクションの違い

PostgreSQLとOracleでは、読み取り一貫性保証の実現方法やエラー発生時の振る舞いなどに違いがあります。
本節では主要な違いをピックアップして説明します。

■同時実行性

PostgreSQL・Oracleともに、複数ユーザがデータに対して同時アクセスすることを保証するために、MVCC(多版型同時実行制御)を使用して管理します。 そのため、表が同時に問合せおよび更新された際に、新旧の複数バージョンのデータを保持して、読み取り一貫性を保証します。

読み取り一貫性を維持する方法は、PostgreSQLとOracleで下記の通り異なります。

- PostgreSQL
  更新処理・削除処理が実行された場合、旧バージョンのデータを無効にして削除せず、新たにデータを追加します。
- Oracle
  更新処理・削除処理が実行された場合、旧バージョンのデータをUNDOデータとして保持します。

■COMMITの実行タイミング

PostgreSQL・Oracleともに、COMMITの実行により、トランザクションが実行したすべての更新を確定します。 COMMITの実行タイミングは、PostgreSQLとOracleで下記の通り異なります。

- PostgreSQL
    - COMMIT文を明示的に実行したとき
    - 下記の場合で、暗黙的にCOMMITが実行されたとき

        - BEGINを明示的に実行せず、SQL文を実行したとき
- Oracle
    - COMMIT文を明示的に実行したとき
    - 下記の場合で、暗黙的にCOMMITが実行されたとき
        - CREATE、DROP等のDDL文を実行したとき
        - ほとんどのOracle Databaseユーティリティおよびツールを正常に終了するとき

■トランザクション中のエラー処理

トランザクション中にエラーが発生した場合、破棄される処理の対象がPostgreSQLとOracleで異なります。

- PostgreSQL
  エラーの発生以前に実行した同一トランザクション内のSQL文は、全て破棄されます。
- Oracle
  エラーが発生した実行SQL文のみが破棄されます。

6.3. コスト見積もり

技術的に移行が可能であっても、莫大なコストがかかり移行を行うことが現実的ではない場合があります。 そのため、技術的な実現性だけではなく、移行コストを見積もり、移行の目的に見合うかどうか判断する必要があります。

主要な移行コストの例は以下のとおりです。 単純な移行費用だけでなく、PostgreSQLの開発・運用技術の教育費用などについても考慮することが必要です。

表 6.12 主要な移行コスト例

項目	例	コスト大
導入費用	・新DB基盤構築[開発/検証/本番] ・新DB開発/運用ツール導入[初期費用/ライセンス費用]	
移行費用	・アセスメント ・スキーマ/アプリケーション/データ移行 ・本番移行 ・運用切替	○
運用費用	・ハードウェア、ソフトウェア保守費用 ・サーバ/クラウド利用料やリース・レンタル料 ・監視サービス費用 ・障害対応 ・チューニング ・定期メンテナンス ・OSアップデート	
教育費用	・開発技術トレーニング[設計/SQL/DBA] ・運用技術トレーニング[導入/運用管理/クラスタ構築]	○

○：DB移行により特にコストが高くなる項目

7. 移行作業

7.1. スキーマ移行

スキーマとはデータベースの内部構造を定義する用語であり、データの構造、性質やほかのデータとの関連、データベースを操作するときのルールや表現法などを定義したもののことです。
一般的なRDBMSでは、データベース内のオブジェクトは「スキーマ」と呼ばれるセットに編成されていて、そのスキーマの中にテーブル、ビューなどのオブジェクト群で構成されています。
つまり、スキーマ移行とは、スキーマ内のオブジェクト定義を移行することです。

7.1.1. スキーマ移行の全体像

異なるRDBMS間の移行では、機能差異や仕様差異があるので、各オブジェクトの移行可否の対応関係を理解することが重要です。
また、移行元DBのシステム情報を参照するオブジェクトが存在すれば、移行先DBのシステム情報と同等のものに置換するか、新規作成が必要となります。

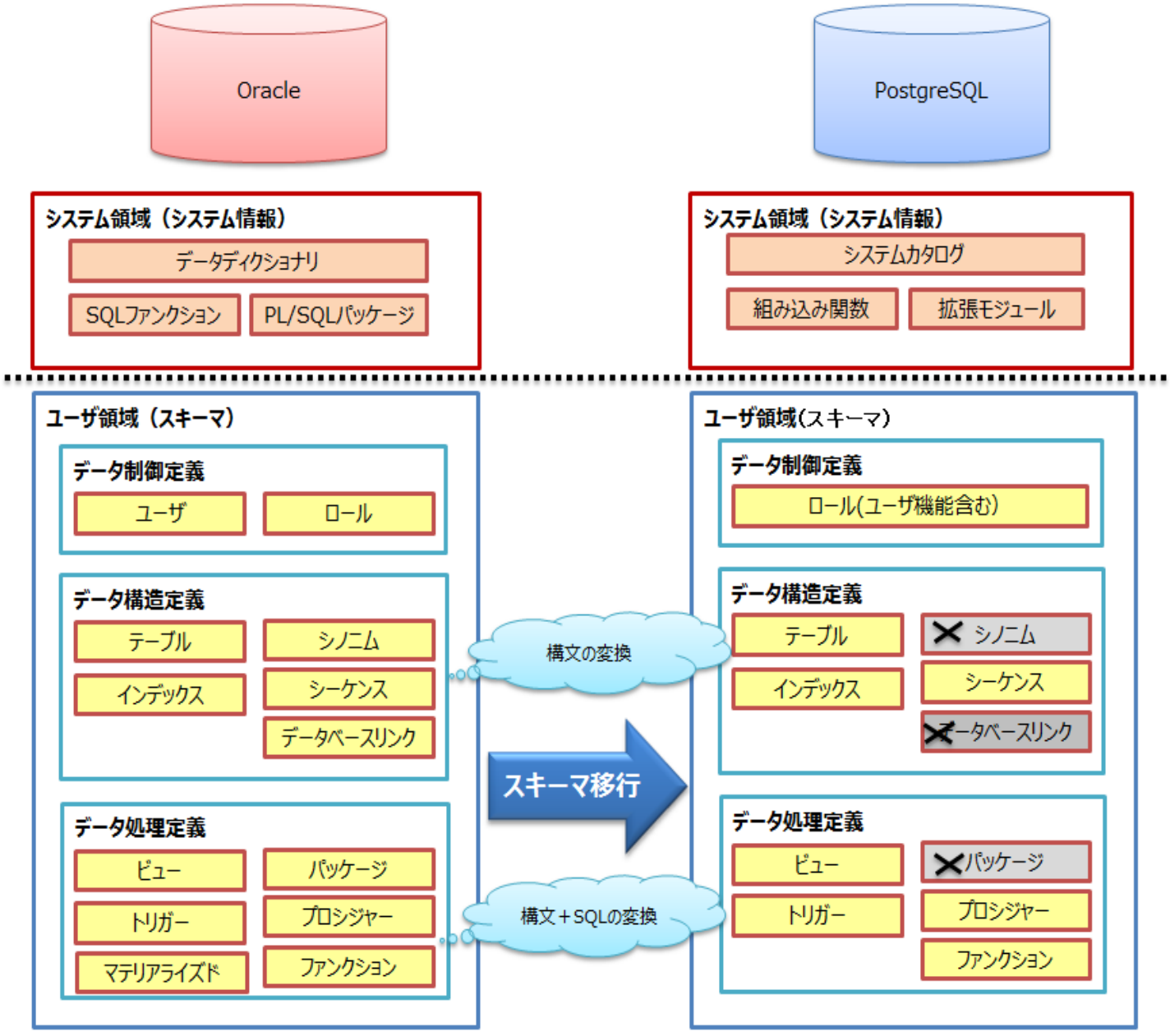


図 7.1 スキーマ移行の全体像

7.1.2. スキーマ移行方式

スキーマ移行方式は、移行元DBからデータベースオブジェクト定義を抽出し、移行先DB用に定義の変換・作成・確認までが基本的な作業フローとなります。

- 移行元のデータベース・オブジェクト定義(DDL,DCL)を準備し、移行先の定義に合うように自動または手動による変換作業を行います。
- その後に移行先DBで変換済みの定義を実行し、システム情報を取得して、データベース・オブジェクトの移行漏れがないかを確認します。

※考慮事項
移行元のデータベース・オブジェクト定義の管理については、以下のような様々な状況が考えられます。

- 仕様書がない、または最新化されていない。
- 定義情報(DDL,DCL)がない、または最新化されていない。
- 仕様書・定義情報とDB上のオブジェクトが不一致となっている。
- DB上のオブジェクトに用途不明なものが存在している。
- 本番環境と開発環境のDB上のオブジェクトが不一致となっている。

既存の仕様書や定義情報だけを鵜呑みにして作業すると、間違った古い定義を移行したり、移行漏れなどのリスクがあります。
DB上のオブジェクト情報を利用すれば、上記のリスクは軽減されます。ただし、本来不要なオブジェクも移行してしまい作業量に影響することもあるので、
この機会に上記の不整合な状況を解消することも念頭に入れて、以下の一般的な作業フローをベースに手直しすると良いでしょう。

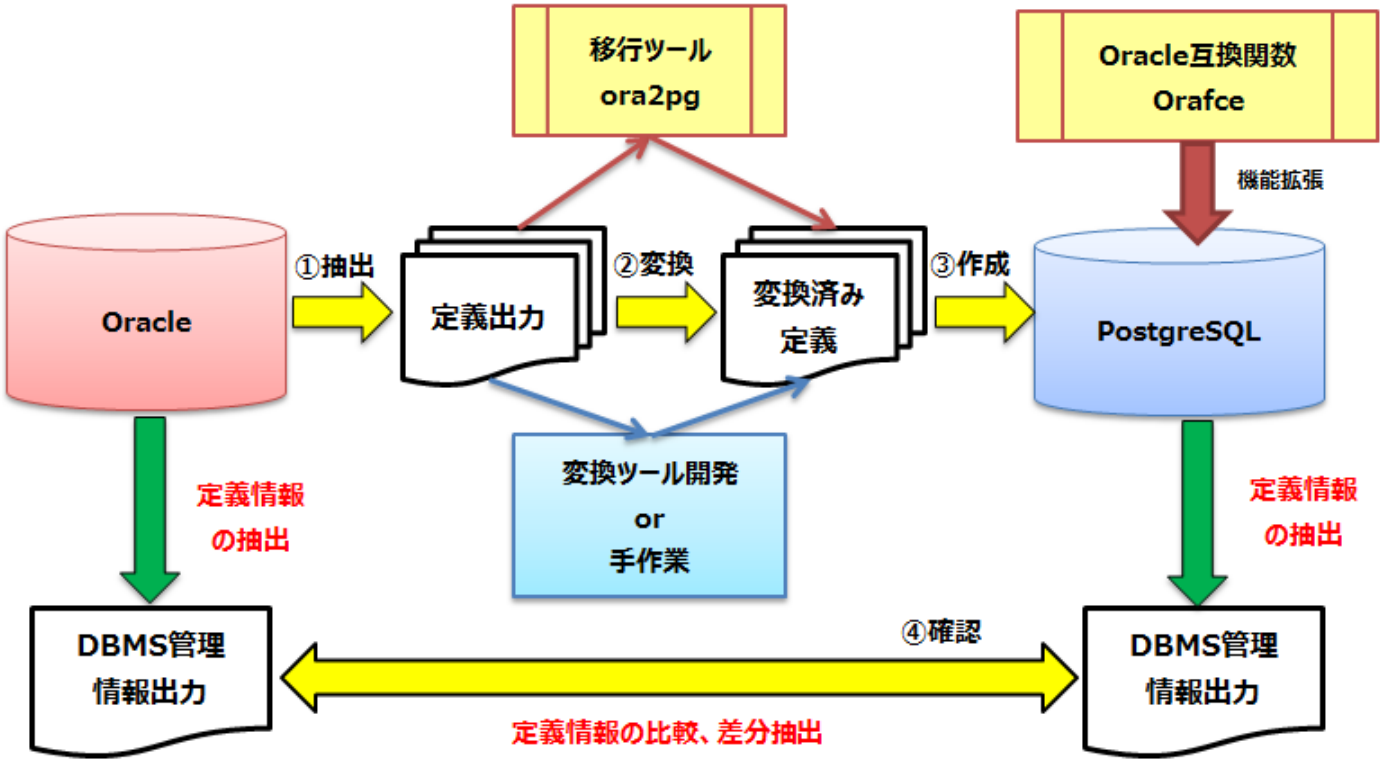


図 7.2 スキーマ移行移行方式

1. 抽出
定義情報(DDL,DCL)をDBのシステム情報から抽出します。
なお、個別に定義情報をバージョン管理している場合は、抽出作業は本来は不要ですが、抽出した結果と差異が無いかを確認はします。

2. 変換
抽出した移行元のデータベース・オブジェクト定義(DDL,DCL)を移行先の定義に合わせて変換します。
この場合は、OracleからPostgreSQLへの移行ツール「ora2pg」を利用すると、DDLの自動変換が可能です。
ただし、PostgreSQLが有している範囲の自動変換だけであるために、それ以外は変換ツールの開発や手作業による対処が必要です。

※データモデリングツールによる一部変換
データモデリングツールなどでテーブル、インデックスなどを管理しているケースでは、ツール上のデータモデル定義を移行先に合わせたデータ型、 桁数、名称に変更し、
移行先用のデータベース・オブジェクト定義を出力します。この場合は、ツールがPostgreSQLに対応していることが前提となります。

3. 作成
オブジェクト間の依存関係を考慮して定義をPostgreSQLで順次実行します。
もし、依存関係を考慮できない場合には、オブジェクト定義を連続実行すると依存オブジェクトが存在しないためのエラーが多発しますが、何回も実行することで依存オブジェクトが徐々に作成されていき、エラーが無くなっていきます。

表 7.1 代用的なオブジェクト依存関係

オブジェクト	依存オブジェクト
テーブル	外部キー制約/シーケンス
インデックス	テーブル
ビュー	テーブル/ビュー
トリガー	テーブル
ストアドプロシージャ	テーブル/ビュー/ストアドプロシージャ/ストアドファンクション
ストアドファンクション	テーブル/ビュー/ストアドプロシージャ/ストアドファンクション

※Oracleの場合は、PL/SQLパッケージなどの固有のシステム関数が多くあり、それらを利用するオブジェクトはエラーとなってしまいます。

4. 確認
移行漏れの確認のために、DBで管理しているオブジェクト情報を取得します。

- オブジェクト単位の件数
- オブジェクト種類と名称

表 7.2 オブジェクト情報の取得方法

抽出情報	データディクショナリ	システムカタログ	情報スキーマ
テーブル	ALL_TABLES	pg_class、pg_tables	information_schema.tables
ビュー	ALL_VIEWS	pg_class、pg_views	information_schema.tables
列	ALL_TAB_COLUMNS	pg_attribute	information_schema.columns
制約定義	ALL_CONSTRAINTS	pg_constraint	information_schema.table_constraints
インデックス定義	ALL_INDEXES	pg_class、pg_indexes	information_schema.key_column_usage
インデックスの列情報	ALL_IND_COLUMNS	pg_attribute	information_schema.key_column_usage
シーケンス	ALL_SEQUENCES	pg_class、pg_sequences	information_schema.sequences
プロシージャ・ファンクション	ALL_SOURCE	pg_proc	information_schema.routines
トリガー	ALL_TRIGGERS	pg_trigger	information_schema.triggers
マテリアライズドビュー	ALL_MVIEWS	pg_matviews	information_schema.tables

※システムカタログとは、RDBMSが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を格納する場所のことです。
※情報スキーマは、標準SQLで定義されている用語で、DBで定義されたオブジェクトについての情報を持つビューの集合から構成されます。

7.1.3. オブジェクト定義の変更

データベースオブジェクト定義の主要な変更箇所は以下のとおりです。
1. 命名規則
オブジェクトに付与できる名称に制約があるので、制約に該当する箇所を変更します。

- 使用可能文字と組み合わせ
- 使用禁止キーワード（予約語）
- 最長文字数

2. データ型
サポートしているのデータ型に変更します。

- 数値の精度による数値型
- 文字型の種類と範囲
- 日付型の種類と精度

3. 構文
基本的な構文はほぼ同じなので、オプションの仕様差部分は代替変更、または削除(利用中止）をします。

4. SQL
標準SQLの記述に従っていれば軽度な変更で済みますが、DB独自要素の記述箇所を変更します。

- 独自の外部結合式
- 演算子の一部
- 除算を含む計算
- 別名の付与
- 独自の仮想表
- 独自の擬似列
- NULLと空文字列の扱い

5. システム関数
SQLで利用しているシステム関数を代替変更します。代替するものが存在しない場合には新規作成が必要となります。

- データディクショナリのビューをシステムカタログのビューで代替する。
- SQLファンクションを組込み関数で代替する。
- PL/SQLパッケージの代替は存在しないが、「Orafce」を導入することで一部を代替する。

6. 手続き型言語
ストアドプロシージャの実装で利用するSQL手続き言語として、Oracleは「PL/SQL」であり、PostgreSQLは「PL/pgSQL」があります。
構文は似通っていますが、上記の対処に加えて、更に下記のような仕様差が多くあるために、手作業による書き換えが必要です。

- サブルーチン構成の分離
- トランザクション制御の相違
- 例外制御の相違
- エラーコードの相違

7.2. アプリケーション移行

ここで言うアプリケーションとは、業務用プログラムの事だけではなく、シェルスクリプトやバッチファイル、DB専用コマンド、支援ツールなどのDB操作を行っている外部プログラム全般を指しています。
これらのアプリケーションからのDB操作には、多種多様な利用形態で実装されています。
つまり、アプリケーション移行とは、アプリケーションからのDB利用形態を移行先DBの仕様に合わせて変更することです。

7.2.1. アプリケーション移行の全体像

アプリケーションからのDB利用形態をおおまかに大別すると、「アプリケーションの実行形式」「アプリケーションの開発言語」「開発/保守運用支援ツール」となります。
これらを移行先DBが提供するインタフェースへ切り替えていくのが基本となります。

- 汎用的なDBインタフェースを利用している場合には、変更範囲は比較的軽度となります。
- DB固有のAPIを利用している場合には、原則再作成となります。ただし、同程度のAPIの提供があり、それに置換することで流用可能な部分が増える可能性はあります。
- SQLの記述は、移行先DBの仕様差異に合わせて変更します。詳細は「スキーマ移行」の章を参照してください。

アプリケーションからのDB利用形態としては、以下となります。

- アプリケーションの実行形式は、各種プログラム言語、コマンド等からDB接続インタフェースを介して、DB接続を確立し、SQL文を実行している。
- アプリケーションの開発言語は、JDBCなどのAPIを利用する形式、ソースコードにSQLを記述した(埋め込みSQL)を使用している形式があります。
  特に埋め込みSQLを使用している場合には、プログラム言語専用のプリプロセッサでDB接続処理のソースコードを生成する必要があります。
- 開発/保守運用支援ツールからDB接続インタフェースを介して、DB接続を確立し、SQL文を実行しています。

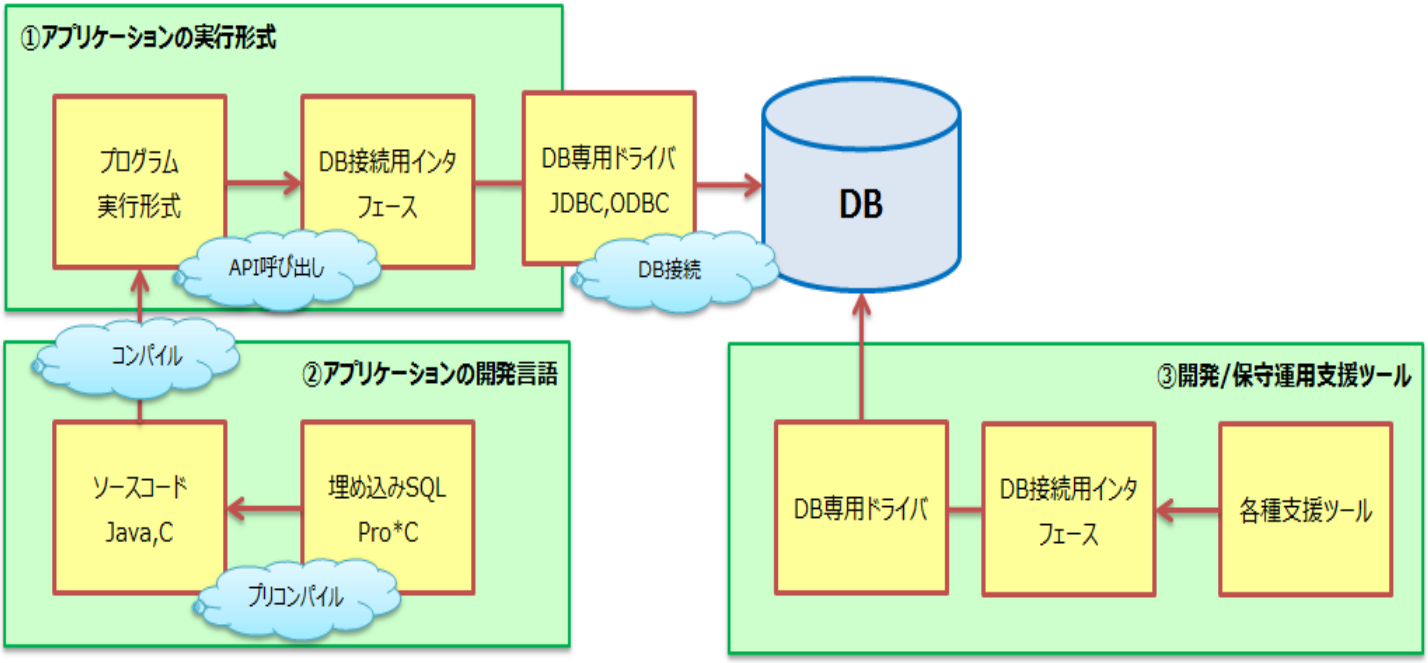


図 7.3 アプリケーション移行の全体像

7.2.2. アプリケーション移行方式

アプリケーション移行方式は、DBの利用形態が多種多様なために一律のパターンは存在しません。
移行パターンとしては、以下に一例を示します。

表 7.3 プログラム言語

言語	Oracle	PostgreSQL
Java	JDBC -Type2	ー
	JDBC -Type4	JDBC -Type4
C	Oracle Call Interface (OCI)	libpq
C++	Oracle C++ Call Interface（OCCI）	libpq++/Pgfe
.NET	Oracle Data Provider for .NET（ODP.NET）	Npgsql
COM	Oracle Provider for OLE DB	PSQL OLE DB
	Oracle Objects for OLE(OO4O)	ー
ODBC	Oracle ODBC	psqlODBC
PHP	php-oci8 PHP Data Objects (PDO)	php-pgsql PHP Data Objects (PDO)
Perl	DBI、DBD::Oracle	DBI、DBD::Pg
Ruby	ruby-oci8 DBI	ruby-pg DBI
Python	cx_Oracle DB-API2	psycopg DB-API2

表 7.4 コマンド

Oracle	PostgreSQL
SQL*Plus	psql
SQL*Loader	copy/pg_bulkload
exp	pg_dump
imp	pg_restore

表 7.5 プリプロセッサ

Oracle	PostgreSQL
Pro*C/C++	ECPG
Pro*COBOL	Open Cobol ESQL

7.2.3. 開発＆実行環境の変更

コンパイル、DB実行に必要なライブラリを入手して置換する必要があります。

- DBドライバの置換、依存ライブラリのインストールをします。
- プリコンパイラを置換します。
- 導入済み開発/運用ツールのDB接続先情報を変更します。
  または、開発/運用ツールを新規導入します。
- 開発環境のビルドやコンパイルのライブラリパスや参照設定を変更します。

7.2.4. ソースコードの変更

1. DBの接続情報
   DBの接続形式と接続先情報を変更します。
   導入したドライバの種類やDBのバージョン、DBライブラリに応じて記述内容が異なります。

2. SQL文
   SQLの構文、データ型、組込み関数などの仕様差異を変更します。
   プリコンパイラの仕様差異を変更します。
   DDLもトランザクション扱いになるので、ROLLBACKさせない場合には明示的なCOMMITを追加します。

3. エラーの判定
   エラーコードやエラーメッセージによる判定処理があれば変更します。

4. 例外ハンドリング
   互換性のない例外による制御処理があれば変更します。
   SQL実行途中にエラーが発生すると、後続のSQL実行はすべてエラーとなりROLLBACKするため、 エラー時に継続したい場合は、トランザクションを分離します。

5.DB実行コマンド
   DB実行コマンドを変更します。

7.2.5. 標準規格インタフェースの対応

DB接続インタフェースには、プログラム言語専用のもの、共通的なオブジェクトやAPIなどの多種多様に存在しています。
ここでは、一般的に利用頻度の高い標準規格のインタフェースを利用した場合の変更点を取り上げます。

7.2.5.1. JDBC系

JDBCはJava言語専用のインタフェースです。このJDBCを前提としたAPIが言語仕様として標準で組み込まれているために
DBの接続方式の種類ごとに提供されているJDBCドライバを用意すれば、各種RDBMSに接続することができます。

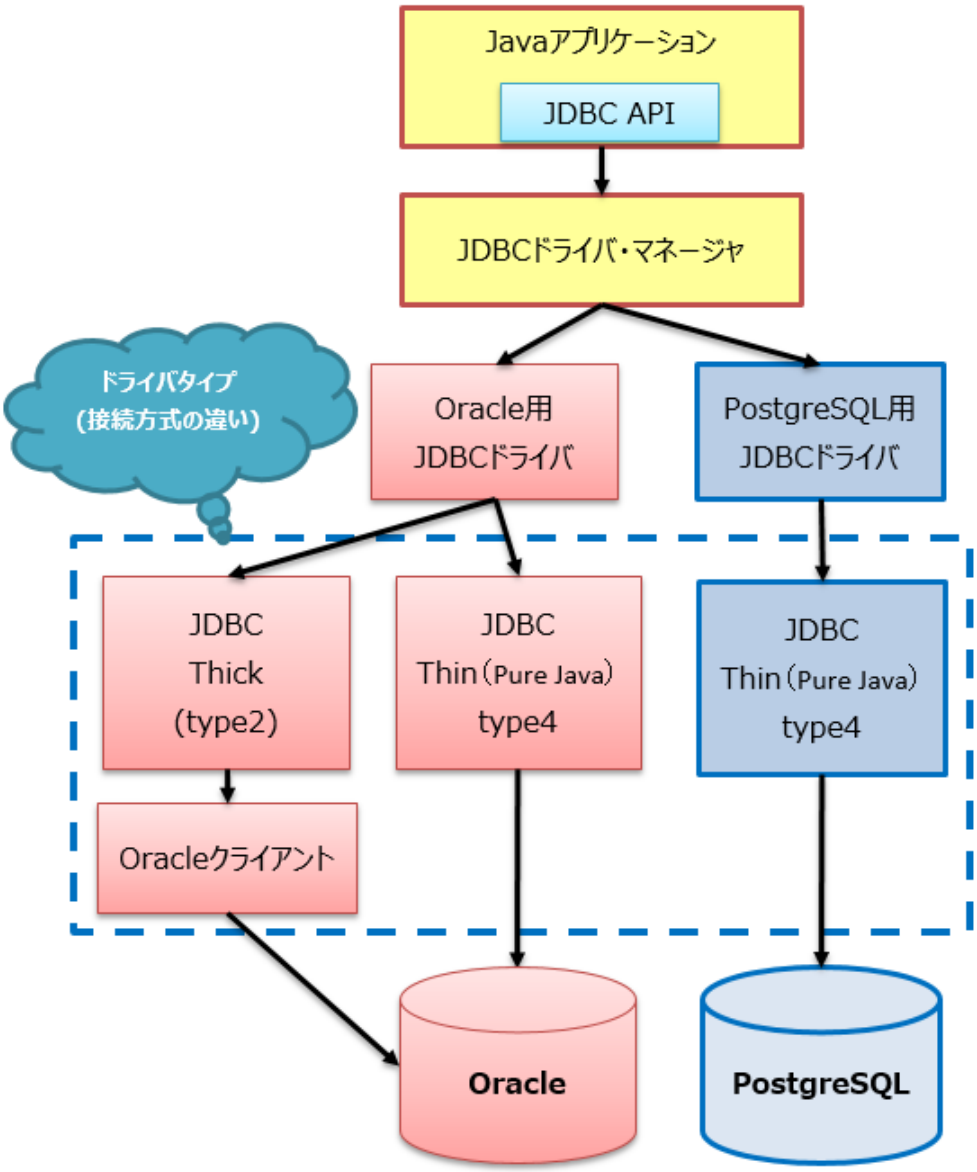


図 7.4 JDBCの通信方式イメージ

■変更範囲
アプリケーションが標準のJDBC API([java.sql]、[java×.sql]パッケージ)を利用している場合には、軽度な変更で対応できます。
JDBCドライバの指定、DB接続情報、エラー値判定、例外ハンドリングの箇所を変更する必要が最低限はあります。
DBベンダー製のSQL例外サブクラスで制御している場合には、SQLExceptionまたはPSQLExceptionで例外を捕捉してエラー判定処理を追加になります。

■変更例
1. JDBCドライバのロード
[Oracle]
```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

[PostgreSQL]
```
Class.forName(“org.postgresql.Driver”);
```

2. 接続文字列
[Oracle]
```
String path = "jdbc:oracle:thin:@//192.168.100.10:1521/orclpdb1.localdomain"; //接続パス
String id = “hr"; //ログインID
String pw = “hrpw"; //ログインパスワード
Connection conn = DriverManager.getConnection(path, id, pw);
```

[PostgreSQL]
```
String path = "jdbc:postgresql://192.168.100.10:5432/db01"; //接続パス
String id = “hr"; //ログインID
```

```
  String pw = “hrpw"; //ログインパスワード
  Connection conn = DriverManager.getConnection(path, id, pw);
```

3. エラー判定
[Oracle]
```
  if (ex.getSQLState().equals("23000")) {    //重複エラー
  if (ex.getSQLState().equals("61000")) {    //リソースエラー
```

[PostgreSQL]
```
  if (ex.getSQLState().equals("23505")) {
  if (ex.getSQLState().equals("55P03")) {
```

4. 例外ハンドリング
[Oracle]
```
  } catch (SQLIntegrityConstraintViolationException ex) {    //整合性制約エラー
```

[PostgreSQL]
```
  } catch (SQLException ex) {
      if (ex.getSQLState().equals("55P03")) {
```

7.2.5.2. Windows系

Windows向けの開発言語からアクセス可能なCOMオブジェクト、ActiveXオブジェクト、.NETオブジェクトがこれまで提供されています。
これらのオブジェクトのDB接続インタフェースとしてODBC/OLEDB/.NET Data Providerなどがあります。
このDB接続インタフェースに対応した各RDBMS専用のものを用意すれば、各種RDBMSに接続することができます。

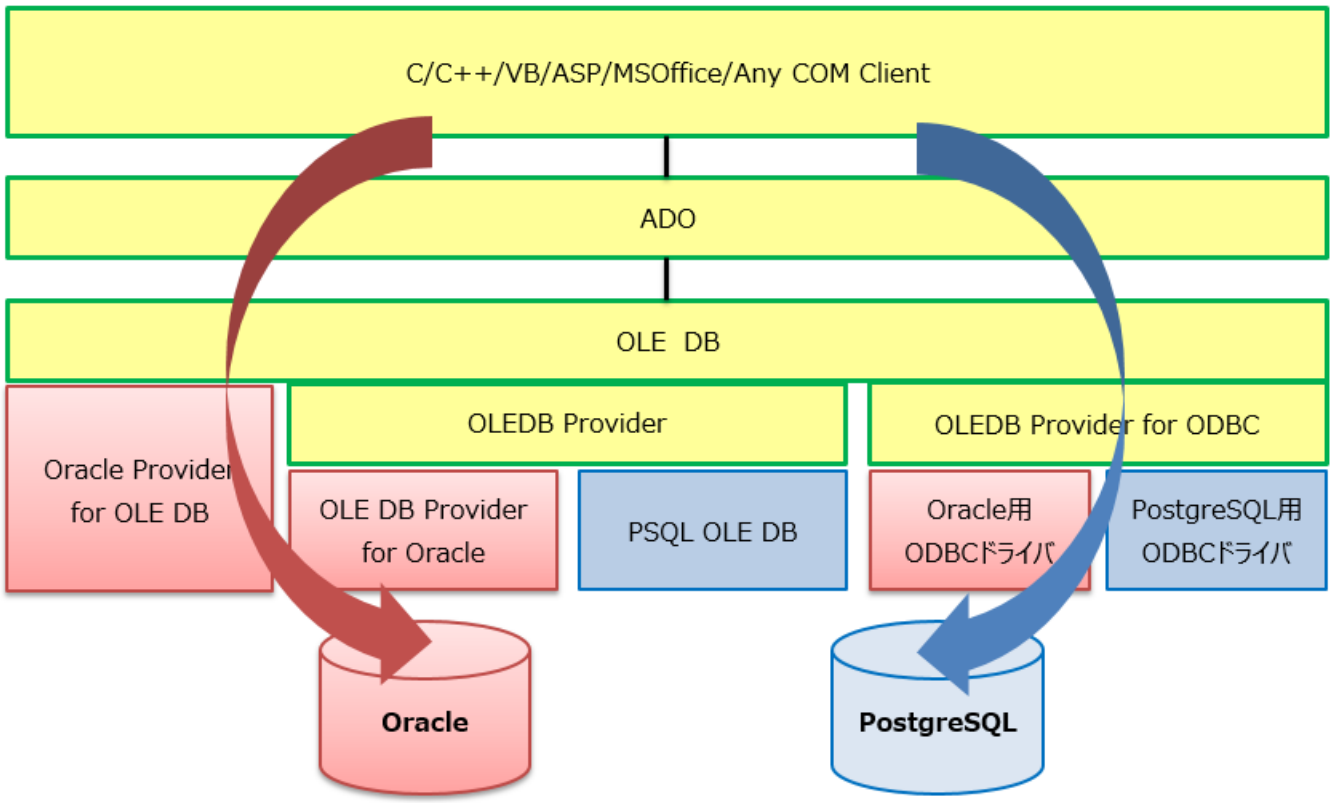


図 7.5 Windows系のDBオブジェクトと通信方式イメージ①

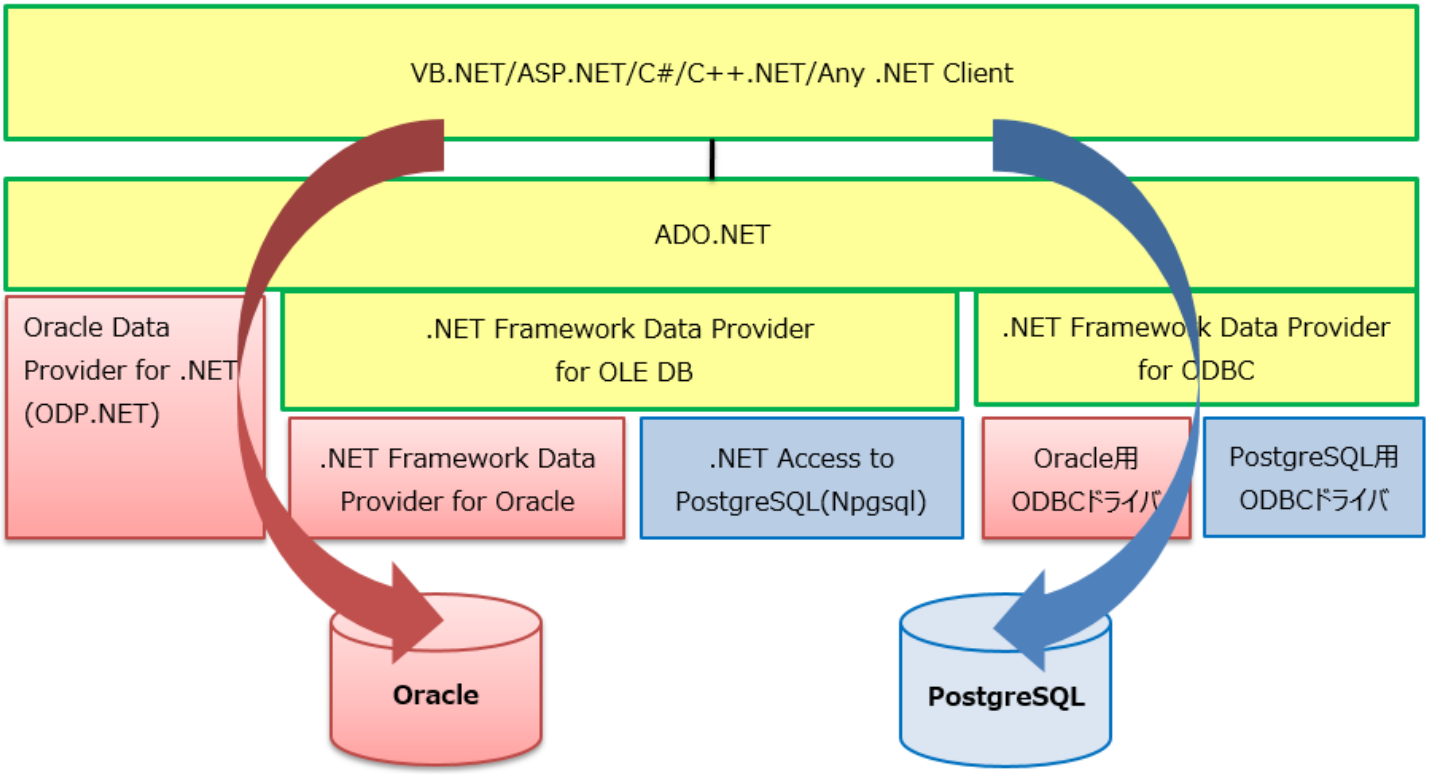


図 7.6 Windows系のDBオブジェクトと通信方式イメージ②

■変更範囲
アプリケーションが利用しているオブジェクトの種類により、変更範囲が変わってきます。
ODBC系のオブジェクトは、DB接続情報、エラー判定の箇所を変更する必要が最低限はあります。
OLEDB系のオブジェクトは、DBベンダー製のオブジェクトを利用することが多く、OLEDBのAPIに基本的には準拠していますが、
部分的には仕様が異なる点や独自に拡張した部分もあり、実装の仕方により影響範囲が異なります。

■変更例
① ADO/VBSの場合　　※ODBC Provider利用
1. 接続文字列
[Oracle]
connect.Open("Driver={Oracle in OraClient12home1};DBQ=ORCL;UID=hr;PWD=hrpw;")

※DBQ：ネットサービス名 「tnsnames.ora」ファイルの内容
ORCL =(DESCRIPTION = (ADDRESS = (PROTOCOL = TCP)(HOST = 192.168.100.10)
(PORT = 1521))(CONNECT_DATA=(SERVICE_NAME=orclpdb1.localdomain)))

[PostgreSQL]
connect.Open("Driver=PostgreSQL Unicode(x64); Server=192.168.100.10;Port=5432;
Database=db01;UID=hr;PWD=hrpw")

2. エラー判定
Errオブジェクトのエラー番号の判定は影響なし
If Err.Number <> 0 Then
WScript.Echo(Err.Number)
WScript.Echo(Err.Source)
WScript.Echo(Err.Description)

② ADO.NET/C#の場合　　※OLEDB Data Provider(ODP.NET/Npgsql) 利用
1. パッケージのインポート
[Oracle]
using Oracle.DataAccess.Client;
using Oracle.DataAccess.Types;
または
using Oracle.ManagedDataAccess.Client;
[PostgreSQL]
using Npgsql;

2. 接続文字列

```
  [Oracle]
    using (var con = new OracleConnection("User ID=hr; Password=hrpw; Data Source=
           (DESCRIPTION = (ADDRESS = (PROTOCOL = TCP)(HOST = 192.168.100.10)(PORT = 1521))
           (CONNECT_DATA=(SERVICE_NAME=orclpdb1.localdomain)));") {

  [PostgreSQL]
    using (var con = new NpgsqlConnection("Server=192.168.100.10;Port=5432;
                User Id=hr;Password=hrpw;Database=db01")) {
```

3. SQLオブジェクト

```
  [Oracle]
    using (OracleConnection cnn = new OracleConnection(connectstring))
         {
           // 接続を開く
           cnn.Open();
           string sql = ""UPDATE EMPLOYEES SET FIRST_NAME = :sei , LAST_NAME = :mei WHERE EMPLOYEE_ID = :id";
           OracleCommand cmd = new OracleCommand(sql, cnn);
           // パラメーターに値をバインドする
           cmd.BindByName = true;
           cmd.Parameters.Add(new OracleParameter(":id", OracleDbType.Int32, 204, ParameterDirection.Input));
           cmd.Parameters.Add(new OracleParameter(":sei", OracleDbType.Varchar2, "織田", ParameterDirection.Input));
           cmd.Parameters.Add(new OracleParameter(":mei", OracleDbType.Varchar2, "信長", ParameterDirection.Input));
           // クエリの実行
           cmd.ExecuteNonQuery();
           // 接続を閉じる
           cnn.Close();
         }

  [PostgreSQL]
    using (NpgsqlConnection conn = new NpgsqlConnection(conn_str))
         {
           // 接続を開く
           cnn.Open();
           string sql = ""UPDATE EMPLOYEES SET FIRST_NAME = :sei , LAST_NAME = :mei WHERE EMPLOYEE_ID = :id";
           NpgsqlCommand cmd = new NpgsqlCommand(sql, cnn);
           // パラメーターに値をバインドする
           cmd.Parameters.Add(new NpgsqlParameter("id", DbType.Int32) { Value = 204 });
           cmd.Parameters.Add(new NpgsqlParameter("sei", DbType.String) { Value = "織田" });
           cmd.Parameters.Add(new NpgsqlParameter("mei", DbType.String) { Value = "信長" });
           // クエリの実行
           cmd.ExecuteNonQuery();
           // 接続を閉じる
           cnn.Close();
         }
```

4. エラー判定

```
  [Oracle]
    switch (ex.Number)
    {
      case 23000:
        MessageBox.Show("重複エラー");
        break;
      case 61000:
        MessageBox.Show("リソース・エラー");
        break;
       default:
        MessageBox.Show("Database error: " + ex.Message.ToString());
        break;
    }

  [PostgreSQL]
    switch (ex.Code)
    {
      case "23505":
        MessageBox.Show("重複エラー");
        break;
      case "55P03":
```

```
            MessageBox.Show("リソース・エラー");
            break;
          default:
            MessageBox.Show("Database error: " + ex.Message.ToString());
            break;
        }
```

5. 例外ハンドリング
   [Oracle]
   catch (OracleException ex) {

   [PostgreSQL]
   catch (NpgsqlException ex) {

7.2.6. 分析支援ツールの利用

「db_syntax_diff」を利用し、JavaやCのソースコード、及びSQLファイルに記述されているSQL文の変更箇所をレポート出力します。
レポート結果は、おおよその影響可能性を示唆するレベルなので、あくまで参考値としての活用に留めてください。

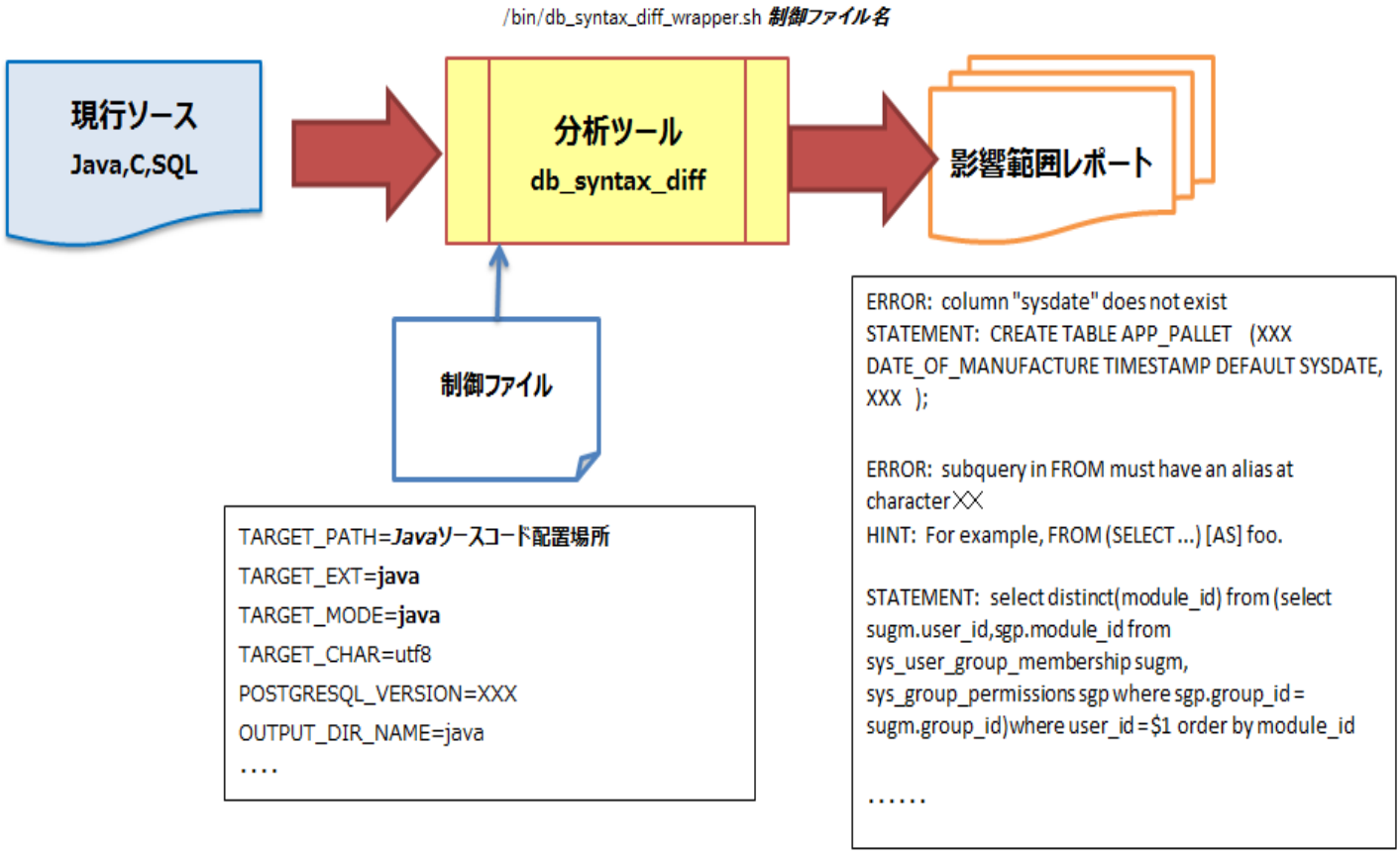


図 7.7 分析ツール利用イメージ

7.3. データ移行

データ移行とは、移行元DBに格納されているデータを、移行先DBに取込むことです。

異なるDB間では、データ型の仕様やサイズ、使用する文字コードなどに差異があります。 したがいまして、データ移行では必要に応じて移行先DBで取り込めるデータへの変換も行います。

7.3.1. データ移行の全体像

異なるDB間の移行では、移行データとして特有のバイナリ形式であるバックアップデータを利用することができません。
標準的な移行では、一旦、移行元DBと移行先DBの両方で使用できるCSV(カンマ・セパレーテッド・バリュー)等のファイルにデータを抽出します。
そして、データ型や文字コードなどの変換を行い、移行先DBへ取り込みます。

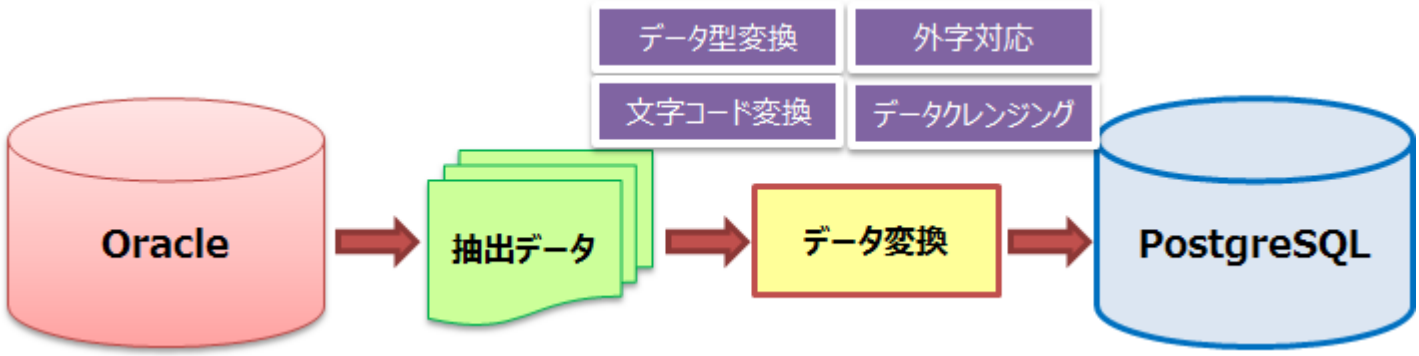


図 7.8 データ移行の全体像

「データ変換」の一部は状況によっては、移行元DB側や移行先DB側の移行前後で行うケースもあります。
また、ora2pgのようなデータ移行ツールでは、同時に両方のDBに接続して、データ型を変換しつつ投入を行うモードもありますが、
意図したとおりに動作しない場合もありますので、一度中間ファイルに出力する方法をお勧めします。

7.3.2. データ移行方式

データ移行で実施する作業について説明します。

7.3.2.1. データ移行フロー

1. 標準的なデータ移行

   標準的なデータ移行は、移行元DBからデータを抽出し、移行先DB用にデータの変換と、取込・確認を行うまでが基本的な作業フローとなります。

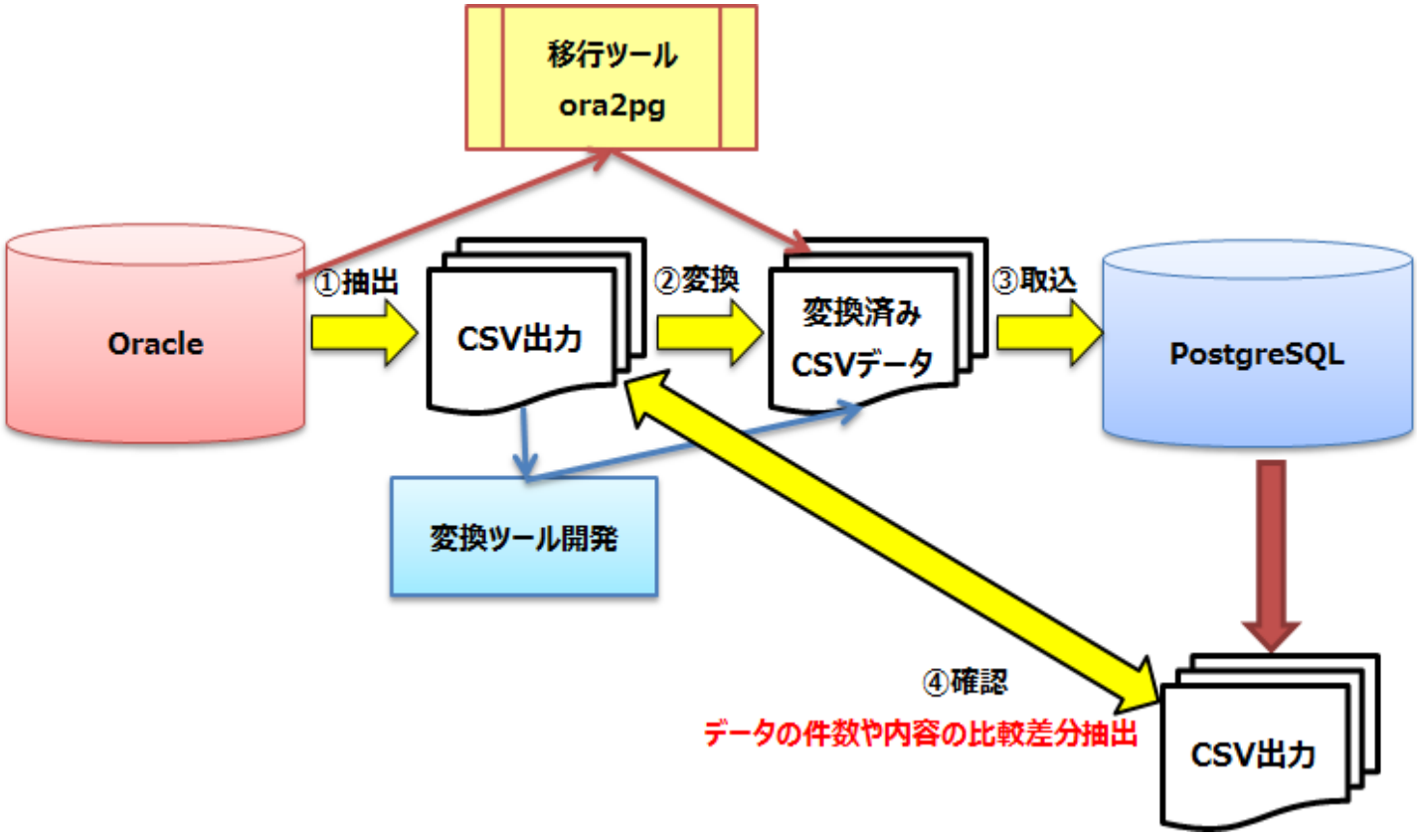


図 7.9 データ移行フロー

2. 大量データの場合

   データ件数やデータサイズが非常に大きい場合、移行作業に時間がかかるうえ、既存システムへの影響も非常に大きくなる可能性あります。
   したがいまして、移行作業における要件に応じて、移行の手段を選択する必要があります。

   - 一括移行

     移行元のシステム停止から新システムの開始までに十分な移行期間を確保することが出来、

データ移行に必要なリソースをを確保できる場合には、移行作業に入ってからデータの抽出、加工、投入で問題ありません。

- 一部データの事前移行

  移行すべきデータを選別し、更新される可能性が無いデータは事前に移行し、更新がかかるデータの移行を移行作業期間中に行います。
  記事など蓄積型のデータは後で更新されることが少ないと考えられ、移行作業中に移行するデータ量を削減することにより作業を短期間で完了させることができます。
  ただし、事前にデータを抽出する際に既存システムへの負荷がかかる可能性があるため、性能への影響を考慮する必要があります。
  また、事前に移行するデータと移行作業期間中のデータが明確に分かれているか、何らかのチェックポイントを設定して未反映のデータが判断できるようにしておく必要があるため、移行作業における設計が重要となります。

- データ連携

  移行元システムのDB と移行先のPostgreSQLを使用する新システムを並行運用したり、更新データを随時新システムに反映することにより、データの移行期間を最小限にすることができます。
  移行元システムのDBからデータを抽出、加工しPostgreSQLに投入するため、移行元のシステムにかかる負荷が課題となる可能性があるほか、
  ETL／データ連携ツールや専用アプリケーションを構築する必要があるため、移行費用が大きくなる可能性があります。

7.3.2.2. データ抽出

主なデータ抽出方法としては、以下のような方法があります。

表 7.6 データ抽出方式

抽出方式	形式	導入	説明
SPOOLコマンド	CSV	不要	SQL*Plusで実行した問合せの結果をファイルに格納する
ora2pg	テキスト(DML)	必要	psqlなどで実行できるDMLとして出力される

7.3.2.3. データ変換

データ変換では、移行元DBと移行先DBとの差異により、データ型や文字コードの変換や外字への対応などを行います。
また、アプリケーションの変更などがあった場合は、データクレンジング(データの整形)が必要となる場合があります。
主な変換に以下のようなものがあります。

表 7.7 データ変換

項目	内容
データ型	以下のようなものを適切な値に変換する。 ・0バイト文字列のように、DBによって格納する値の意味が異なるもの ・同じデータ型でも格納可能な限界値に差異があるもの
文字コード	PostgreSQLへ投入するテキストファイルがPostgreSQLに対応していない場合や、 pg_bulkloadを用いてデータ投入を行う場合、文字エンコーディングの変換を行う必要がある。
外字	移行元DBで外字が使用されていた場合、その外字を移行先DBで使用できる文字に変換する。 もしくは、移行先DBで外字を使用できるようにシステムに登録する。 注意：外字が使用できるようにシステムに登録されており、かつ、クライアントエンコーディングと データベースエンコーディングとが異なる場合、 自動エンコーディング変換を利用するためには、 外字のマッピングを定義し、マッピングを再登録する必要がある。
データクレンジング	移行と同時にデータフォーマットが変わるようなアプリケーションの変更が発生している場合、 変更にあわせたデータの変換を行う。

7.3.2.4. データ取込

主なデータ取込方法としては、以下のような方法があります。

表 7.8 データ取込方式

取込方式	データ形式	速度	導入	制限事項
SQL	テキスト(DML)	低	不要	なし
COPY	CSV	中	不要	なし
pg_bulkload	CSV	高	必要	クラッシュ時にリカバリが必要。レプリケーション構成は再取得や再作成が必要。

7.3.2.5. データ確認

データが正しく移行されたことを確認するために、以下のような作業を行います。

表 7.9 データ確認

項目	内容
データロード時のエラー等の確認	実行したコマンドのエラーメッセージの有無を確認する。 エラーが表示された場合は、その問題点を確認し、対処してから再度データをロードする。 pg_bulkloadではロールバックされないので、データロードを再実行する際は、一旦テーブルのTRUNCATEが必要。
移行対象オブジェクト数および各オブジェクトの行数の確認	データロード後のPostgreSQLデータベースのオブジェクト数と各オブジェクトの行数が、 移行元データベースの状態と一致するかどうか確認する。
CSV出力の結果照合	DBに登録されたデータをCSVファイルとして抽出し、データ取込において使用したCSVファイルと、 内容や行数が一致するかどうかを確認する。
アプリケーションテスト	実際にアプリケーションから接続して一連の処理を行い、想定通りの動きをするかどうか、 文字化けが発生していないか等を確認する。

データ取り込み後には、その他に以下のような作業を行います。

- 制約・索引の作成
- ユーザのシステム権限およびオブジェクト権限の確認
- VACUUMとANALYZEの実行
- 初期状態でのテーブルサイズの確認

8. 運用

8.1. 運用の概要

PostgreSQLも他のRDBMS同様に安定的なデータベースの運用のためにいくつかの運用上のタスクが必要になります。データベース移行時にもPostgreSQLの特性に合致した運用タスクの移行が必要であり、移行元DBと同様な処理であっても機能差やコマンドの仕様差により修正または新規に構築する必要があります。運用要件によって必要なタスクは変わりますが、以下は共通的に実施するものになります。

- 日常のタスク
  - バックアップ
  - 索引の保守
  - データベースのVACUUM
  - ログの管理
  - データベースの監視
  - パフォーマンス診断

- 不定期に発生するタスク
  - アップグレード

8.2. バックアップ

データベースのバックアップは主に以下の２つの目的で使用されます。

- ディスク障害によるデータ破損や操作ミスによるデータロストなど不測の事態が発生した場合に復元可能とするため
- バージョンアップやシステムの変更の検証などのためにデータの移出・移入を行うため

Oracleとはバックアップの方式、手順が異なるため移行においてはこれらを要件毎に新規に実装します。
公式の「PostgreSQL文書」ではデータをバックアップする手法として”SQLによるダンプ”、”ファイルシステムレベルのバックアップ”、”継続的アーカイブ”と３つの方式が説明されています。本項ではそれぞれの方式の特徴を移行元のOracleとの比較も合わせて簡潔に説明します。

- バックアップの方式
  以下ではそれぞれの方式と対応するOracleツールを表にしたものです。

表 8.1 バックアップ方式の一覧

方式	方法	バックアップ対象	リカバリ範囲	DB停止	対応するOracleツール
SQLによるダンプ	pg_dumpツール	"データベースクラスタ，データベース, スキーマ, テーブル"	バックアップ取得時点	不要	DataPump
ファイルシステムレベルのバックアップ	OSコマンド	データベースクラスタ	バックアップ取得時点	要	ー(OSコマンド)
継続的アーカイブ	・ベースバックアップ（全体バックアップ）+アーカイブWAL+未アーカイブWAL ・pg_rman	データベースクラスタ	ベースバックアップ～最新状態	不要	・Begin BackupとEnd Backup+OSコマンドバックアップ+アーカイブログ ・RMAN(リカバリ・マネージャー)

- バックアップ要件と対応する方式
  バックアップの要件と上記の方式と対応を表にしたものです。

表 8.2 バックアップ要件と対応方式

要件	選択する方式	想定するユースケース
最新のトランザクションまたは任意の時点の状態まで復元	継続的アーカイブ	ディスク障害等不測の事態に対する対応
バックアップ時点に復元	全ての方式で可能	定期保守
異なるバージョンへの移行	ＳＱＬによるダンプ	移行
スキーマ単位、テーブル単位など部分的なバックアップアップと復元	ＳＱＬによるダンプ	システム変更などの検証

- バックアップ方式毎の特徴
  - SQLによるダンプ
    メリット：唯一、テーブルやスキーマ単位で取得可能なバックアップであること。OracleのDataPumpをイメージすると分かりやすいです。また、コマンドはシンプルで習得も容易です。
    デメリット：バックアップ時点までしか戻せないことです。

  - ファイルシステムレベルのバックアップ
    メリット：OSファイルコピーのため高速にバックアップ、リストアができること。
    デメリット：DBを停止する必要があり、バックアップ単位もデータベースクラスタ全体のみになります。
    データベースクラスタが小規模で単純な構成かつバックアップ時にはDBが停止できるなど、限定的なケースでのみ選択したほうがよいでしょう。

  - 継続的アーカイブ:
    メリット：未アーカイブのWALが確保できれば障害発生時点のトランザクションまで復元できる唯一のバックアップです。
    デメリット：設計および手順が他より複雑であるため運用には十分な知識と訓練が必要になります。具体的には以下の全てが該当します。
    - 運用の前提としてWALのアーカイブ設定が必要であること。
    - WALアーカイブ先の空き領域監視、アーカイブされたWALの整理が必要であること。
    - リストア時間はベースバックアップの復元およびWALの適用時間となり、適用するWALが多ければ多いほど時間がかかること。
    - バックアップ設計は想定されるリストア時間からベースバックアップの実行間隔を決定する必要があること。
    - バックアップ、リストアの手順が複雑であること。

    ただし、pg_basebackupを使うことでバックアップ時の手順の簡略化は可能です。LinuxOSであればpg_rmanを導入することでバックアップ、リストアの手順の大幅な簡略化、さらに差分バックアップ、世代管理、圧縮バックアップが可能となりバックアップ設計の自由度も向上可能です。

8.3. 索引の保守

索引はデータベースのパフォーマンスに強く影響を与えるもののひとつです。PostgreSQLの索引はいくつかの種類があり、最もよく使われるのがB-tree索引です。本項でもB-tree索引の保守について取り上げます。

テーブルの特定の行にアクセスする場合、テーブル全体をアクセスするより、索引を用いて必要なブロックのみアクセスすることで少ないアクセスで目的の行を取得できます。
反面、テーブル本体以外にも索引をアクセスする必要があるため、索引が断片化していたりすると索引そのもののアクセスが多くなりパフォーマンスを劣化させてしまいます。また、テーブルの更新が発生すると索引もあわせて更新されるため更新処理の負荷を増加させます。（HOT機能を利用することで軽減可能）

このため、運用での索引の保守は「不要な索引の検出と削除」、「断片化された索引の再構築」の対応が必要になります。

- 不要な索引の検出と削除
  索引毎の使用状況は情報スキーマのpg_stat_XXX_indexes,pg_statio_XXX_indexes（XXXはuserまたはall)により、使用回数、読み込みブロック数などを確認することができます。定期的に監視を行い一定期間、使用回数等の増加がない索引がある場合は削除を検討します。

- 断片化された索引の再構築
  - 断片化の監視
    索引の断片化は追加モジュールのpgstattupleをインストールすることでpgstatindex関数が使用可能になります。この関数でtree_levelが深すぎないか、leaf_fragmentationが大きすぎないかなど再構築の必要性を確認します。ただし、索引の監視自体も運用コストの増加になります。監視を行わずに定期的な再構築を行うという選択肢もあります。

  - 索引の再構築
    索引の再構築にはいくつかの方法があります。それぞれにメリット、デメリットがありますので運用要件に応じて選択します。

表 8.3 索引の再構築の方式

NO	方式	メリット	デメリット
1	reindex	コマンドが容易かつDB単位、スキーマ単位、テーブル単位で実行可能	処理中にテーブルの書き込みロックと対象の索引を使用する読み込みにブロックがかかる
2	drop index+create index	1の方式よりロック影響は少ない	create indexでテーブルに書き込みロックがかかる
3	create index concurrentry+drop index+alter index rename	運用中のDBへのロックの影響が最も小さい	手順が多い。concurrentryオプションはテーブルを２回スキャンするため時間がかかる
4	pg_repack	3の方式を1コマンドで実施可能	周辺ツールのインストールが必要。実行時に途中で停止した場合はリカバリ操作が必要

- 索引の再構築の組み込み基準
  索引の再構築を運用に組み込む場合、小規模なDBであればDB全体の索引を再構築しても短時間で完了するため業務停止時間にreindexで実施してしまうことが運用コストを最も小さくできます。また、大規模なDBであれば再構築時間も長くかかることから運用の影響を小さくする配慮も必要となります。

  以下に運用保守のバッチ処理内に索引再構築を組み込む基準例を示します。表中の”推奨する方式”は前表の”NO”を表します。

表 8.4 索引再構築の方式選択例

DBの規模	保守時間が確保できるか	性能要求	推奨する方式（NO)
小	できる	—	1(DBまたはスキーマ単位)
小	できない	高い	3or4
小	できない	低い	再構築しない
中	できる	—	1(スキーマ単位またはテーブル単位)
中	できない	高い	3or4
大	できるが全て再構築する時間はない	—	1(テーブル単位）で曜日毎に対象テーブルを分ける
大	できない	—	3or4で曜日毎に対象テーブルを分ける

8.4. データベースのVACUUM

PostgreSQLのデータベースはデータの更新に追記型のアーキテクチャを採用しているため、物理的にデータ量は拡大し続けます。このため、更新や削除によって不要となった領域の整理を行う必要があり、そのためにVACUUM処理が用意されています。

VACUUM処理はパラメタを指定しなければ不要となった領域の回収のみを行い、パラメタにFULLを指定することで完全なVACUUMを行います。VACUUM FULLは拡大したデータ領域を縮小することができますが、処理中に表への排他ロックがかかるなど制限もあります。

実行方法は下記表のように自動で実行する方法と手動で実行する方法があります。DBクラスタの作成ではパラメタファイルの初期状態で自動的に実行されるように調整されており、ほとんどのケースではそのままで充分です。

表 8.5 VACUUM処理の実行方法

実行方法	実行範囲	備考
自動VACUUM	VACUUM+ANALYZE	パラメタファイルで自動実行を指定する(デフォルトで自動実行)。ANALYZEの閾値は別に指定可能
手動VACUUM	コマンドで指定したパラメタによる	コマンドで実行

- 運用におけるVACUUM処理の調整
  基本として自動VACUUMを使用し、運用上不都合がある場合に手動VACUUMを使用します。

  VACUUM処理はPostgreSQLにとって負荷の高い処理のひとつです。例えば、大きな表で前回VACUUM処理から多くの行が更新・削除された場合での自動VACUUM処理が動作すると、優先すべき他の処理のパフォーマンスに悪影響を与える可能性があります。そのような状況を回避する必要がある場合は、以下のような調整を行う必要があります。

  - 自動VACUUMの実行タイミングを調整
    自動VACUUMの実行タイミングは、テーブル毎に更新・削除により発生した不要行が

autovacuum_vacuum_threshold + autovacuum_vacuum_scale_factor * テーブル行数 > 不要行（dead_tuple数）

を超えたときに実行されます。例えばautovacuum_vacuum_thresholdが0、autovacuum_vacuum_scale_factorが0.2であったとすると１億件の表は2000万件更新・削除されないと自動VACUUMされません。大量件数分のVACUUM処理の負荷が高い場合、これらのパラメタを調整することで１回当たりの自動VACUUMの負荷を下げて他の処理への影響を小さくすることが可能です。この調整はDBインスタンス全体で行うか、以下のようにテーブル単位で行うことも可能です。

例１　全体の10%の不要行が発生したら自動VACUUMする
alter table 表名 set (autovacuum_vacuum_threshold = 0);
alter table 表名 set (autovacuum_vacuum_scale_factor = 0.1);

例２　1,000件の不要行が発生したら自動VACUUMする
alter table 表名 set (autovacuum_vacuum_threshold = 1000);
alter table 表名 set (autovacuum_vacuum_scale_factor = 0.0);

- 特定の表に自動VACUUMを停止し任意に手動VACUUMを行う
  特定の表のみ自動VACUUMを停止することも可能です。その場合、表には以下のような変更を実施します。

  alter table 表名 set (autovacuum_enabled = false);

  自動VACUUMを停止するとその後の更新により、データ領域に再利用可能な領域がなければデータ領域は拡大し続けますので、保守時間や業務負荷の低い時間帯に手動でVACUUMを実行する必要があります。（ほとんどの場合、バッチ処理にVACUUMを組み込み、スケジュール実行となります）

- VACUUMとVACUUM FULLの使い分け
  基本的にはVACUUMを使用します。VACUUM FULLを使用することはほとんどありません。しかし、次のようなケースが発生する場合はVACUUM FULLを検討する価値があります。

  - 表全体の更新が行われ、かつ全件検索が多発する表の場合
    追記型のアーキテクチャであるために１つカラムが表の”全件”更新されるだけでも、表全体の領域の使用率は５０％以下になります（再利用されない不要領域があると割合はさらに低下する）。この表の検索パターンが多く全件検索が多発する場合、表全体の領域をアクセスするために最良の状態に比較し、2倍以上の読み込みが発生することになります。この過剰な読み込みがもたらす性能劣化を許容できない場合はVACUUM FULLを検討します。

  - 業務アプリケーションのメンテナンス時
    通常運用によるデータ更新と異なり、通常は発生しない表の全件更新を伴うケースもあります。この場合も表全体の領域の使用率を必要以上に悪化させる場合は、メンテナンスにVACUUM FULLを行う手順を組み込んでおきます。

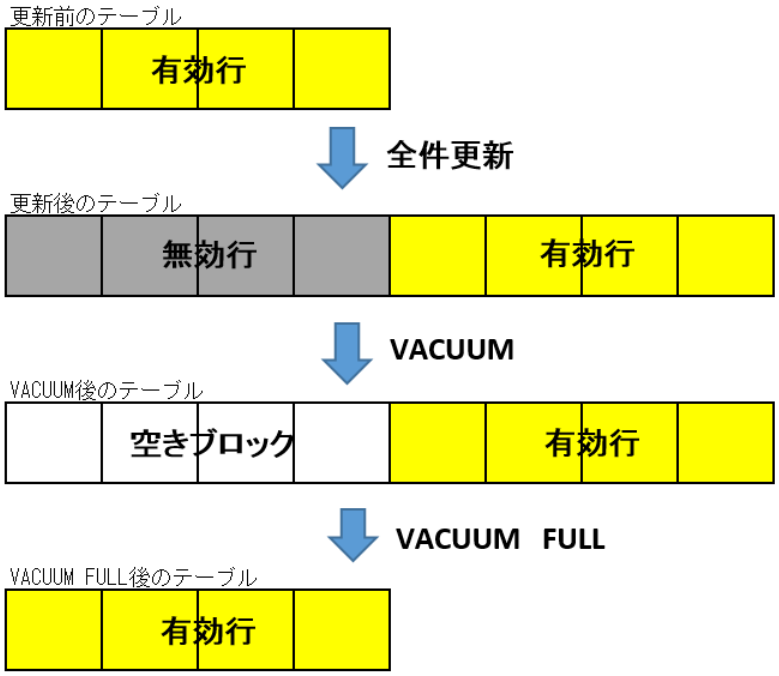


図 8.1 表全体の更新とVACUUM（FULL）のイメージ

なお、表の領域の使用率は、pgstattupleのtuple_percentで確認できます。

- まとめ
  VACUUM処理の運用は以下の表のように調整します。

表 8.6 VACUUMの使い分け

大規模テーブルの有無	全件更新	性能要求	VACUUM処理
なし	なし	高い	自動VACUUM(調整)
なし	あり	高い	自動VACUUM(調整)+VACUUM FULLを検討
あり	なし	高い	自動VACUUM(調整)+テーブル毎の調整 or 手動VACUUM
あり	あり	高い	自動VACUUM(調整)+テーブル毎の調整 or 手動VACUUM+VACUUM FULLを検討
上記以外	←	←	自動VACUUM(デフォルト)

8.5. ログの管理

本項でのログは、データベースの運用に伴い発生する情報やエラー等を含む情報を記録するサーバログになります。ログの出力設定を適切に行うことで以下のようなメリットがあります。

- 障害の発生を検知できる
- 性能問題の発生原因となる情報を取得することができる

ただし、詳細に出力すると運用によっては 1 日で数十キガバイト以上の量を出力する可能性もあります。ここでは一般的な業務システムのログの出力設定例を示します。

- ログファイルの出力先等
  ログファイルの出力指定になります。出力先はsyslogも指定できますが、syslogを出力するjounald,rsyslogプロセスのデフォルトで記録制限をかけているため短時間での大量ログは破棄される恐れがあることから、以下ではstderrとしています。

表 8.7 ログファイルの出力先指定

パラメタ名	設定例	備考
log_destination	stderr	標準エラーを指定
logging_collector	on	onを指定しログ収集を有効にする。offにするとログは出力されない
log_directory	/var/log/postgresなど	絶対パスで指定しない場合はデータベースディレクトリ以下に相対パスとして作成される
log_filename	postgresql-%Y%m%d.log	ログのファイル名。日付毎の出力が運用しやすい
log_rotation_age	1d	1日でログを切替え
log_rotation_size	0	ログファイルの最大サイズ。0はサイズ指定無効

- ログファイルに出力する情報
  業務システムの特性を考慮し設定値を調整する必要があります。性能要件が高い場合、性能欄が"○"の情報は出力を推奨します。

表 8.8 ログファイル出力情報

パラメタ名	設定例	性能	備考
log_min_error_statement	ERROR	ー	エラー条件の原因となったSQL文を記録するか制御する。ERRORがデフォルト
log_min_duration_statement	3s(値は例)	○	指定した時間以上に時間のかかった処理を記録する。問題のある処理の特定に役立つ。0を指定すると全ての文を出力（大量に出力されるのでログの肥大化に注意が必要）
log_checkpoints	on	○	checkpoint starting: xlogが頻発する場合はWAL関連のパラメタを見直す
log_connections	on	○	onでクライアント認証の記録の残す。大量の接続が発生する場合、ログの肥大化に注意が必要
log_line_prefix	%t %u[%p:%l]	ー	各ログ行の先頭に出力する。%tの時刻と%pのプロセス番号は必須
log_lock_wait	on	○	ロック待ちが発生しているか確認できる。あわせてdeadlock_timeoutを指定する
deadlock_timeout	3s(値は例)	○	デッドロック検査前の待ち時間。ロック待ちログメッセージの待機時間にも使われる
log_temp_files	10MB(値は例)	○	最初は0として一時ファイルを使ったら全て出力させてwork_memのサイズを調整する目安にする。調整後は例のような一定サイズに変更する。運用中は大量の一時ファイルを使うSQL(=遅い）を特定できる
log_autovacuum_min_duration	3s(値は例)	○	実行頻度、時間帯、実行時間を確認し、autovacuumの実行を調整する目安にする。調整しても性能要件を満たせない場合は手動VACUUMを検討

- ログファイルの整理
  ログファイル名を曜日としての出力するなど、設定の方法によってはPostgreSQL自身で周期的に古いログファイルを上書きしてログファイルの増加を防ぐことができます。上記の”ログファイルの出力先指定”のように日付でログファイル名を指定した場合は、定期的に保存期間を超えたログファイルを削除するバッチ処理を組み込む必要があります。

8.6. データベースの監視

データベースの運用中に発生した問題に迅速に解決するため、または今後の問題発生の防止のため、リアルタイムおよび定期的にデータベースの監視を行います。

- リアルタイムな監視
  データベースの異常に対する即時対応のために実施します。

- 定期監視
  データベースのピーク接続数やディスク所要量などの変化を経年的に監視することで将来的に不足するかなどの目安にします。

監視も運用コストの増加になりますので運用するシステムの重要性に応じて、どこまで実施するか検討します。以下に代表的な監視項目を列挙します。

- ログの監視
  ログの監視は「8.5.ログの管理」により出力されたログを対象に監視するものとします。
  ログ内のERROR,FATAL,PANIC（syslogではWARNING,ERR,CRIT、イベントログではERROR）の発生を監視し、データベースに異常が発生している場合、発生原因に応じて対策を実施します。

- ディスクの空き領域監視
  運用においてデータ量の拡大等によってディスク領域を不足させてしまうことがあります。発生時はデータベースの停止（＝業務停止）を伴う可能性が高いため、不足が発生しないように監視を行います。

    - データ格納先ディレクトリ、表領域格納先ディレクトリ
      長期間のデータベース利用でデータ増加に伴い不足する可能性があります。不足する場合、不要なテーブルの削除、新規の表領域格納先ディレクトリを用意して移動、索引の再構築、VACUUM FULLなど対応が必要です。

    - トランザクションログ格納先ディレクトリ
      通常は一定量以上は拡大しませんが、アーカイブログ運用時にアーカイブ出来ない状況やレプリケーション運用でレプリケーション出来ない状態が発生するとトランザクションログは増え続けます。原因を取り除き領域を開放する必要があります。

    - アーカイブログ格納先ディレクトリ（アーカイブログ運用時）
      アーカイブログは何もしなければ増え続けます。アーカイブログの格納領域が不足すると書き出せないトランザクションログがトランザクション格納領域に残るため、この領域の増加が発生します。通常、バックアップ計画に伴い、不要なアーカイブログを削除するようにします。緊急時はOSコマンドでアーカイブログを別の領域に移動するなどして空き領域を確保します。また、

pg_rmanなどの利用で自動で整理することもできます。

- 接続数の監視
  利用者数の拡大や業務アプリケーションの追加、システム運用上の特性などにより接続数が増減する可能性があります。監視は統計情報ビューの"pg_stat_activity"の件数をカウントして行います。なお、0件も異常です。少なくともバックグラウンドプロセスの接続があるためです。監視は一定の閾値（max_connectionsの90%など）を超えた時点でアラートを出力するようにします。アラートが出力される場合はmax_connectionsの増加とあわせ、リソースが不足する場合はリソース強化等の対応を行います。

- プロセス監視
  接続数の監視を実施する場合は、データベースへのアクセス（統計情報ビュー）の応答が行われていることからプロセスの監視は不要です（新規の接続が可能なことも合わせて確認できます）。接続数の監視を実施しない場合、postgresのマスタープロセスが起動していることをOSコマンドで監視します。postgresの子プロセス（loggerやcheckpointerなど複数存在）の監視は不要です。子プロセスが起動できない場合はマスタープロセスも異常終了するためです。

上記の他にもシステム要件によっては監査ログ相当のログ（誰が、どこから、いつ、どのデータを取得したか）が必要な場合もあります。この場合はpgauditとログ分析用ツールの導入も検討します。

- まとめ
  以下にデータベース監視について表にまとめます。リアルタイム監視、定期監視の"○"がついている部分が監視が必要な項目になります。

表 8.9 データベースの監視

監視項目	リアルタイム監視	定期監視	定期監視での分析観点
ログの監視	○	ー	ー
ディスクの空き領域監視	○	○	空き領域の減少具合から、いつ不足するか予測
接続数の監視	○	○	接続数の変動を経年変化で観測しパラメタ変更、チューニング、リソース増強などの対策要否の検討
プロセス監視	○	ー	ー
アクセス監査（必要に応じて）	ー	○	不正アクセスが発生していないか。pgauditを導入

8.7. パフォーマンス診断

データベースのパフォーマンスが正常かそうでないかを判断するには、ある一時点での評価ではなく「正常な状態」と「遅くなっている状態」の比較をすることで分析します。このため、パフォーマンス診断は継続的な診断情報の取得と定期的な分析が必要になります。また、定期的な診断を行うことで、将来発生しうる性能問題を事前に対策することも可能となります。

Oracleから移行する場合、以下の場合は移行先のPostgreSQLでもパフォーマンス診断を行う必要があります。

- Oracleのエディションがエンタープライズ・エディションでかつ、オプション製品のDiagnostics Packがある場合
- StatsPackがインストールされ、かつスナップショットの記録が行われている場合

Oracleの場合、上記で取得される情報よりそれぞれAWRレポート、StatsPackレポートを出力し比較・分析をします。PostgreSQLの場合は周辺ツールの利用も検討し同様な分析ができるようにします。

- PostgreSQLでのデータベース診断
  以下にデータベースの診断と対応する周辺ツールを表にします。

表 8.10 データベースの診断と周辺ツール

周辺ツール等	対応OS	診断情報	分析単位	分析方法	備考
pg_statsinfo	linux	リポジトリDB	任意の期間	テキスト形式の性能レポート、HTML形式の性能レポート（別途pg_stats_reporterが必要）	専用のログ設定が必要
pgBadger	linux	ログファイル	ログ全体、日単位、週単位のサマリ	HTML形式の性能レポート	専用のログ設定が必要。運用によっては数十GB/日のログが出力される
pg_stat_statements	linuxとWindows	pg_stat_statement_reset実行後からのSQLの実行統計を累積	データベース単位	pg_stat_statementsビューの参照。例）クエリ実行時間,実行回数のトップ10調査などSQLで取得	データベースにエクステンションの追加とパラメタ:shared_preload_librariesにpg_stat_statemntを追加
情報スキーマの情報から分析の仕組みを自作	linuxとWindows	情報スキーマや上記のpg_stat_statementsなどの情報を一定間隔でテーブル等に蓄積	任意の期間	蓄積されたテーブルの任意の期間の差分で分析するクエリを自作	仕組みを自作する必要がある

実装にあたってはPostgreSQLの提供する情報だけでなく、OSで取得できる情報はOSで定期取得、アプリケーションで独自に出力するアプリログも存在すれば合わせて取得します。また、ベンダーが提供する有償版のPostgreSQLの場合は専用の診断ツールが提供される場合もあります。これらのツールを含めより正確に現状と過去からの変化を把握し安定的に運用できるように診断を行います。

8.8. アップグレード

PostgreSQLのアップグレードはメジャーバージョンのアップグレードとマイナーバージョンのアップグレードがあります。（バージョンについては4.3.バージョンを参照）
メジャーバージョンのアップグレードでは機能追加、マイナーバージョンのアップグレードでは不具合やセキュリティホールへの対応が含まれます。運用としてのアップグレードはマイナーバージョンアップを対象とし、より新しいマイナーバージョンがリリースされている場合は適用を推奨します。

- マイナーバージョンアップグレード
  DBクラスタの内部格納形式はマイナーバージョン間で変更されることはありません。そのため、アップグレードに伴うバックアップ・リストアは必要ではありません（バックアップは作業ミス等への対応として実施することには意味があります）。アップグレード方法の基本は以下のようになります。

  - データベースクラスタの停止
  - 実行ファイルの置き換え
  - データベースクラスタの起動

- メジャーバージョンアップグレード
  DBクラスタの内部格納形式やパラメータ、PostgreSQLの動作も異なります。移行するメジャーバージョン間で作業が異なりますが、共通作業として新機能の適用検討や動作確認、バージョン間の非互換対応を実施する必要があります。運用の作業ではなくバージョンアップ移行という観点で捉える必要があります。

9. 著者

（企業・団体名順）

版	所属企業・団体名	部署名	氏名
第1.0版 (2018年度WG2)	日本電気株式会社	AIプラットフォーム事業部	黒澤 彰
	日本電子計算株式会社	技術本部	毛塚 賢一
	日本電子計算株式会社	技術本部	高橋 泰之
	富士通株式会社	ミドルウェア事業本部	陶山 香織
	富士通株式会社	ミドルウェア事業本部	豊島 良美
	富士通エフ・アイ・ピー株式会社	ソリューションサービス推進本部	多田　明弘
	三菱電機株式会社	情報技術総合研究所	田中　覚
第1.1版	富士通株式会社	ミドルウェア事業本部	豊島 良美
第1.2版 (2019年度WG2)	日本電子計算株式会社	技術本部	毛塚 賢一
	日本電子計算株式会社	技術本部	高橋 泰之