アシスト

LPI-JAPAN

OSS-DB Exam Silver
技術解説無料セミナー

2019/6/29

株式会社アシスト　データベース技術本部
NPO法人日本PostgreSQLユーザ会　理事長

喜田紘介

講師プロフィール

- **名前　喜田 紘介(きだ こうすけ) @kkkida_twtr**
- **所属　株式会社 アシスト データベース技術本部**
  NPO法人 日本PostgreSQLユーザ会

■最近のこと

- 本セミナーの受講側で参加したとき、PostgreSQLはVer9.0でした。
  いまや大規模アップデートv10を経てv11へとさらなる進化を続けています。
- そんなPostgreSQL歴≒社会人生活の中で大いに道標にして学ばせていただいたのが、LPI-Japan様が企画するイベントや試験でした。今では一緒にこのような活動ができて大変嬉しく思っています。
- **2019/3 改定に携わった**オープンソースデータベース標準教科書Ver.2.0 **がリリースされました。**
- **2019/6 NPO法人日本PostgreSQLユーザ会の理事長に就任しました。**
  今後もPostgreSQLやDB技術の普及など目指して頑張っていきます！

講師プロフィール

- 名前　喜田 紘介(きだ こうすけ) @kkkida_twtr
- 所属　株式会社 アシスト データベース技術本部
  NPO法人 日本PostgreSQLユーザ会

エンタープライズ領域でのOSS-DB活用を支援

EDB Postgres Advanced Server

Enterprise Deployment and Use
Oracle互換構文、開発インターフェース、
リアルタイムデータ連携

Advanced Server
性能強化、大規模環境用パラメータ、
豊富なチューニング支援機能

PostgreSQL
基本に忠実、拡張豊富なOSSデータベース

Management Tools

増分・並列バックアップ
EDB Backup and Recovery Tool（BART）

クラスタウェア
EDB Failover Manager（EFM）

DB管理 / 監視 / 性能解析
EDB Postgres Enterprise Manager（PEM）

データベース移行ツール
EDB Migration Toolkit（MTK）

異種DB間レプリケーション
EDB Replication Server

オープンソースデータベース（OSS-DB）に関する技術と知識を認定するIT技術者認定

OSS-DB /Silver

データベースシステムの設計・開発・導入・運用ができる技術者

OSS-DB /Gold

大規模データベースシステムの
改善・運用管理・コンサルティングができる技術者

OSS-DB技術者認定資格の必要性

商用/OSSを問わず様々なRDBMSの知識を持ち、データベースの構築、運用ができる、
または顧客に最適なデータベースを提案できる技術者が求められている

OSS-DB /Silver

データベースシステムの設計・開発・導入・運用ができる技術者

- **RDBMSやPostgreSQLの構造の理解**
  - **RDBMSに求められる機能とその実装(高性能・同時実行・耐障害性などを満たす内部構造)**
- **メンテナンスコマンドの理解**
  - **オプションレベルで、何ができるか知っている / コマンドから結果を予測できる**
- **SQL開発**

OSS-DB /Gold

大規模データベースシステムの改善・運用管理・コンサルティングができる技術者

- **PostgreSQLの内部構造の詳細や応用的な構成(たとえば、レプリケーション)**
- **メンテナンスや障害対応の必要性の判断と適切な実施**
- **広い視野でチューニングができる**

■試験概要 > OSS-DB Silver

https://oss-db.jp/outline/silver

データベースをはじめて学習する人には
試験範囲を読み解くハードルが高いかも・・・

■大まかに整理すると・・・

アプリ開発者

・アプリケーションのコードを書く
・データベースに対する処理

SQLを書く、トランザクション
テーブル設計、チューニング基礎
など

インフラ管理者

・システムの安定稼働
・データベースに対する処理

サーバ設計、起動・停止、
運用管理（監視、対処、バックアップ）
など

本講義の内容

■データベースの基本を解説

- データベース技術者としての入門レベルであり、PostgreSQLを扱う上で必須知識であるOSS-DB Silver試験に向けた学習のきっかけに。
- 初級者がPostgreSQLを使用したデータベース学習を進められることを目標とする。

■取り扱う内容

- データベースに求められること
- RDBMSの構造
- SQL開発
- DBA(データベース管理者)のタスク

アジェンダ

■データベースに求められること

データベースに求められる 「高性能」「同時実行性」「耐障害性」などの基本を整理し
これらを実現するRDBMSの重要なキーワードを解説

■RDBMSの構造

前章で挙げたデータベースとしての基本が、PostgreSQLでは
どのような仕組みで実装されているかを解説

■SQL開発

RDBMSの共通言語である 「SQL」 の基本を解説

■DBA(データベース管理者)のタスク

RDBMSの構造から定期的なメンテナンスの必要性を解説し、管理者が実施する
具体的なタスクやその実施方法を解説

データベースに求められること

■データベースに求められる「高性能」「同時実行性」「耐障害性」などの基本を整理し、これらを実現するRDBMSの重要なキーワードを解説

アプリ側で細かいことは考えずとも、

●簡単にデータを記録して取り出したい
●高い性能を維持したい
●お客さんが更新したデータを店舗側でリアルタイムに見たい
●確定したデータは確実に守られていてほしい

アプリ開発者

ありがと、了解！

データベース用意したから（正しく）使ってね！

インフラ管理者

アプリのコードを書く

システムの安定稼働

データベースに求められること

アシスト OSS-DB

■高性能
- 格納された大量のデータから必要なものを高速に検索する

■同時実行
- 同時に多数のユーザがデータを参照し、任意のタイミングでデータを変更する

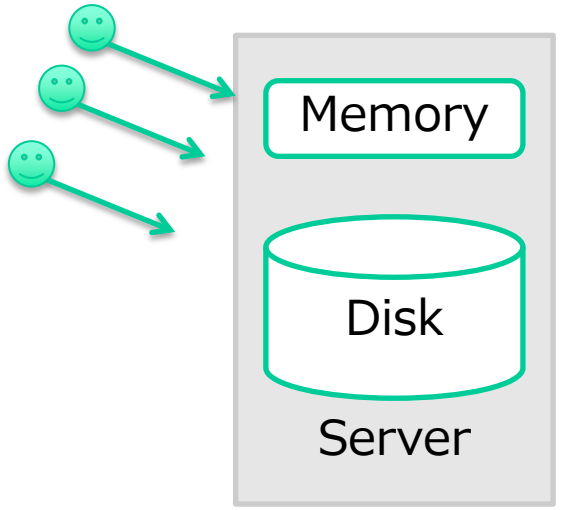
■耐障害性
- データを確実に保護し、万が一の障害時に復旧を可能とする

■その他
- データへのアクセス言語(SQL)
- 各種チューニング/メンテナンス手段
- 性能/障害調査のため内部情報へのアクセス手段
- アクセス制御/暗号化などのセキュリティ機能
- 可用性・負荷分散を実現するレプリケーションなど

■キーワード

- ディスクI/Oの削減
- 共有メモリ
- ログ先行書き込み と チェックポイント
- チューニング

■考え方のポイント



- 性能の観点では、メモリのみで処理を続けることが理想
  ただしデータは永続化したい
- 変更履歴をシーケンシャルI/Oでディスクに保存する事で、
  性能影響を抑えて永続化

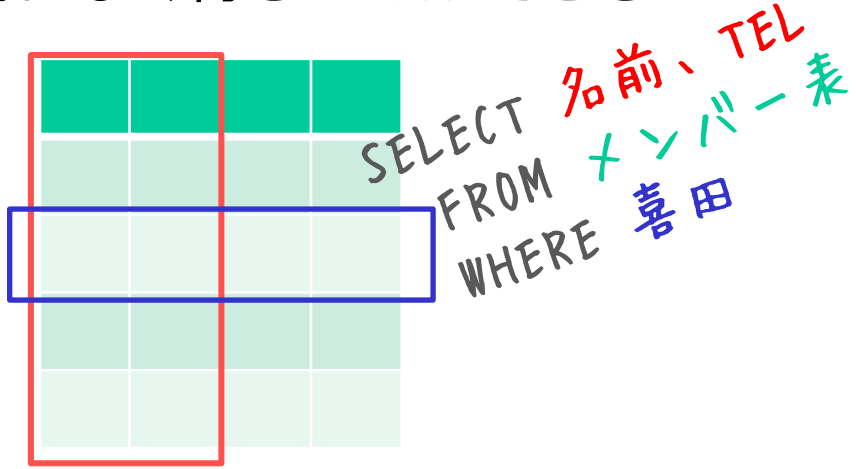
シーケンシャル I/Oの負荷 << ランダム I/Oの負荷

■キーワード

- SQLによる最適なデータの抽出
- 実行計画
- 索引（INDEX）、結合（JOIN）　など

■考え方のポイント

- データの集合から、必要なデータ（条件に合致するデータ）を検索し漏れなく得ることができる



- 集合全体が何行であっても、そこから取り出す結果が何行であっても、それぞれ相応に適したパフォーマンスが得られる

同時実行の実現

キーワード

- トランザクション
- ロック
- 読み取り一貫性

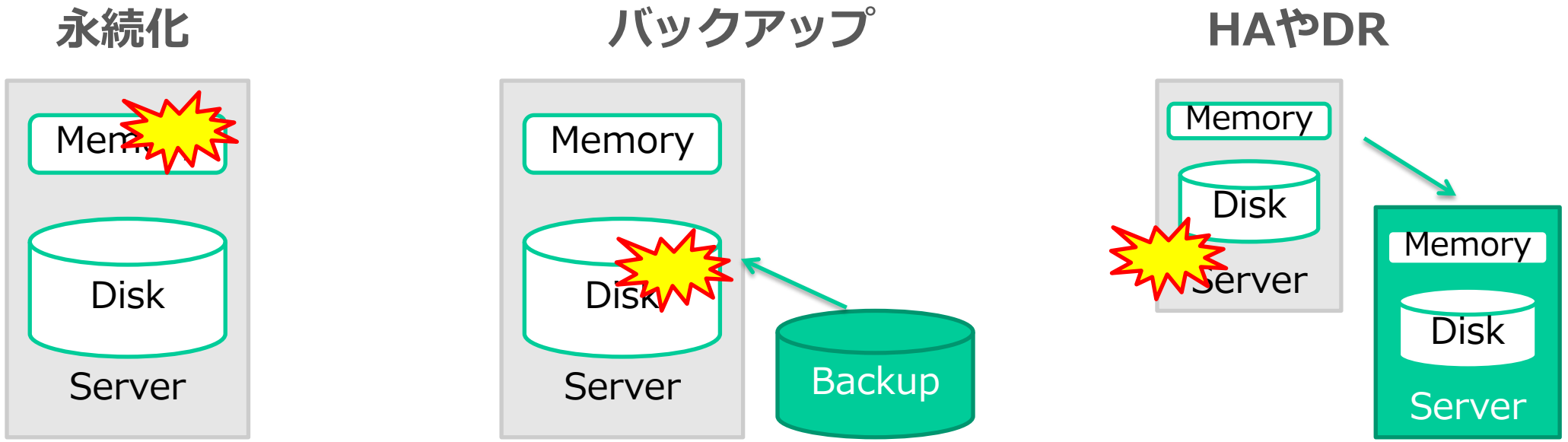
トランザクションとは

- 現実の処理をコンピュータで扱うための考え方
- 適切な**ロック**を獲得する（獲得できない場合は待機する）ことで、同時に同じデータが複数人から更新されることを防ぎ、また、同時に反映されるべきある一連の更新は、**読み取り一貫性**により他者から途中の段階を見られることは無い。

■キーワード

- ログ先行書き込み と チェックポイント
- 障害の種類
- バックアップ・リカバリ

■障害からデータを保護する方法



アジェンダ

■データベースに求められること

データベースに求められる 「高性能」「同時実行性」「耐障害性」などの基本を整理し
これらを実現するRDBMSの重要なキーワードを解説

■RDBMSの構造

前章で挙げたデータベースとしての基本が、PostgreSQLでは
どのような仕組みで実装されているかを解説

■SQL開発

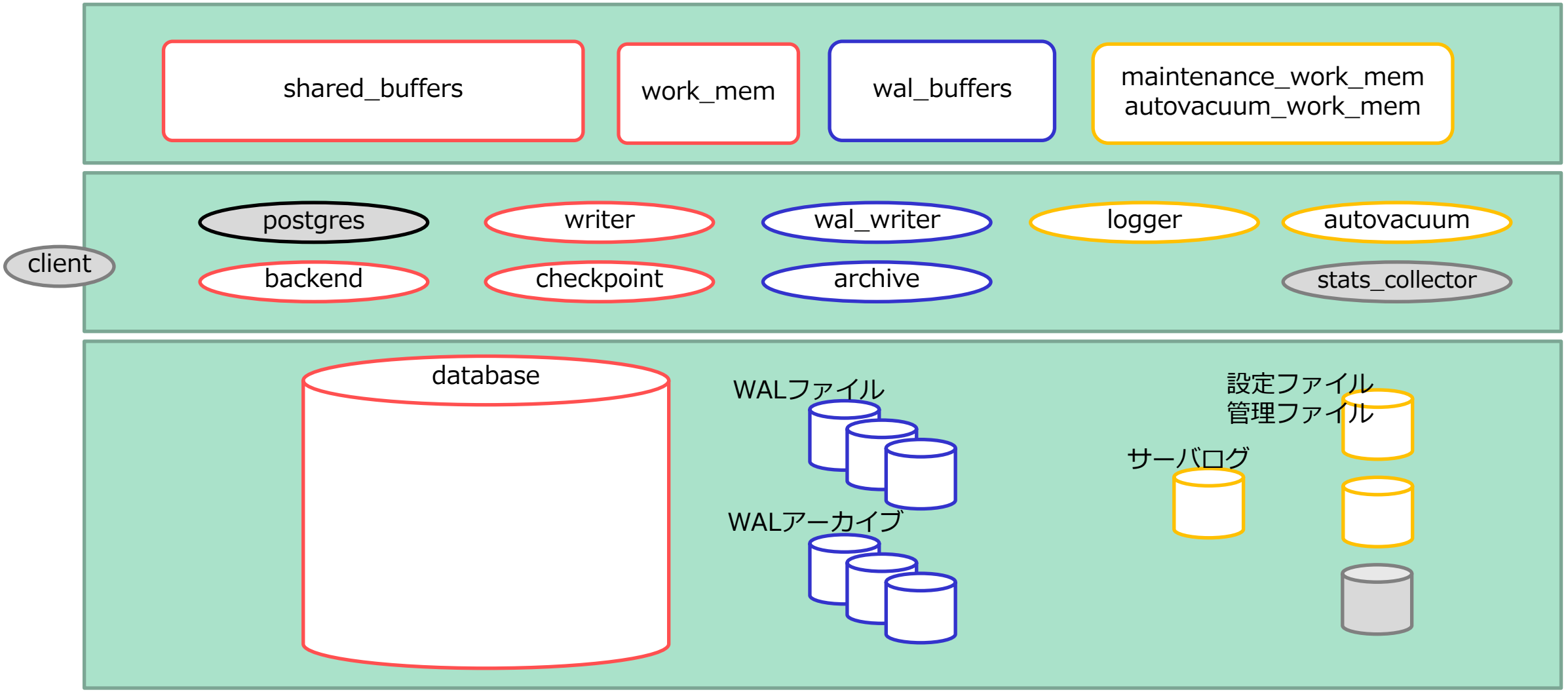
RDBMSの共通言語である 「SQL」 の基本を解説

■DBA(データベース管理者)のタスク

RDBMSの構造から定期的なメンテナンスの必要性を解説し、管理者が実施する
具体的なタスクやその実施方法を解説

PostgreSQLの全体構造

■メモリ・プロセス・ディスク領域からなるDB全体構造を把握



■データベースクラスタ

- 1つ以上のデータベースと、管理情報・設定ファイルが集まったもの
  - PostgreSQLは、データベースクラスタ単位で起動・停止を行う
  - 実体は構築時に指定するPostgreSQL関連の最上位のディレクトリ
    (ディレクトリを指す場合は、「データディレクトリ」と記載される)
  - 環境変数$PGDATAにデータディレクトリのパスを設定しておく

■データベースクラスタの構成要素

内容	指定方法	ディレクトリ(ファイル)名
データディレクトリ	initdb -D	$PGDATA
WALファイル出力先	initdb -X	$PGDATA/**pg_wal**
ユーザデータ格納先	TABLESPACE機能	$PGDATA/**base**
ログファイル出力先	パラメータ	$PGDATA/**log**
アーカイブ退避先	パラメータ	<指定した出力先>
状態管理・設定ファイル群	--	postgresql.conf、pg_hba.conf　他

PostgreSQLのメモリ領域

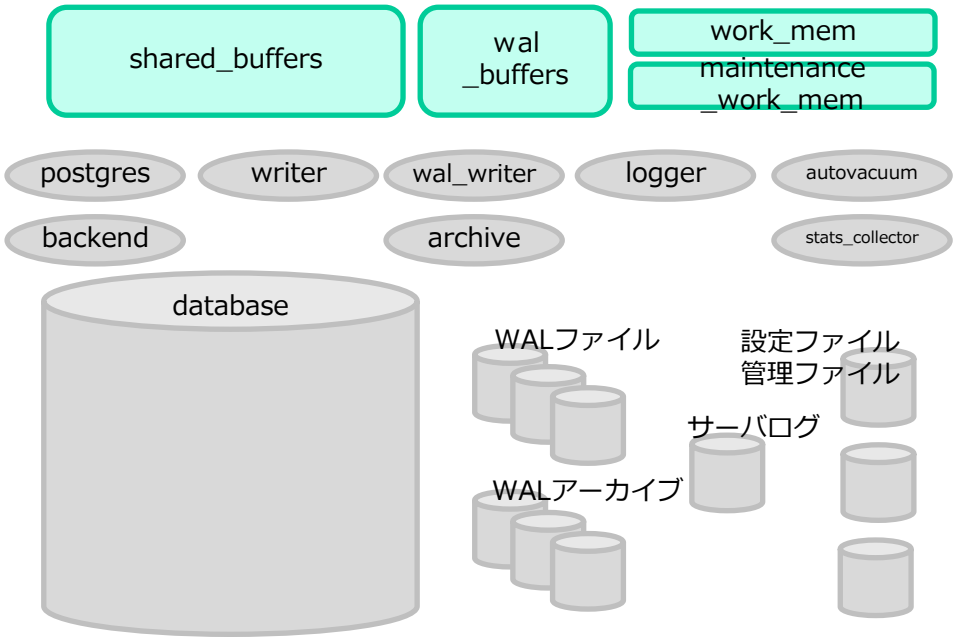
■共有メモリ

- 共有バッファ
  - ディスクから読み取ったデータをキャッシュして、以降のユーザ要求に高速に応答
- WALバッファ
  - ログ先行書き込み(Write Ahead Logging)
  - 耐障害性とパフォーマンスを両立するための仕組み

■セッションメモリ

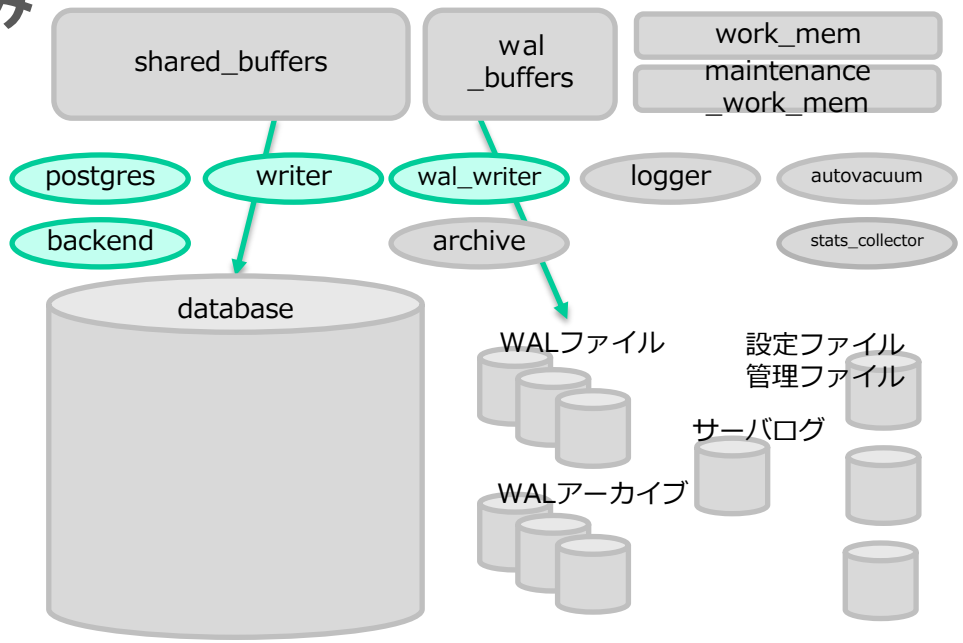
セッション毎に確保される領域

- ワークメモリ
  - ソートやハッシュの一時領域
- メンテナンスワークメモリ
  - メンテナンス操作



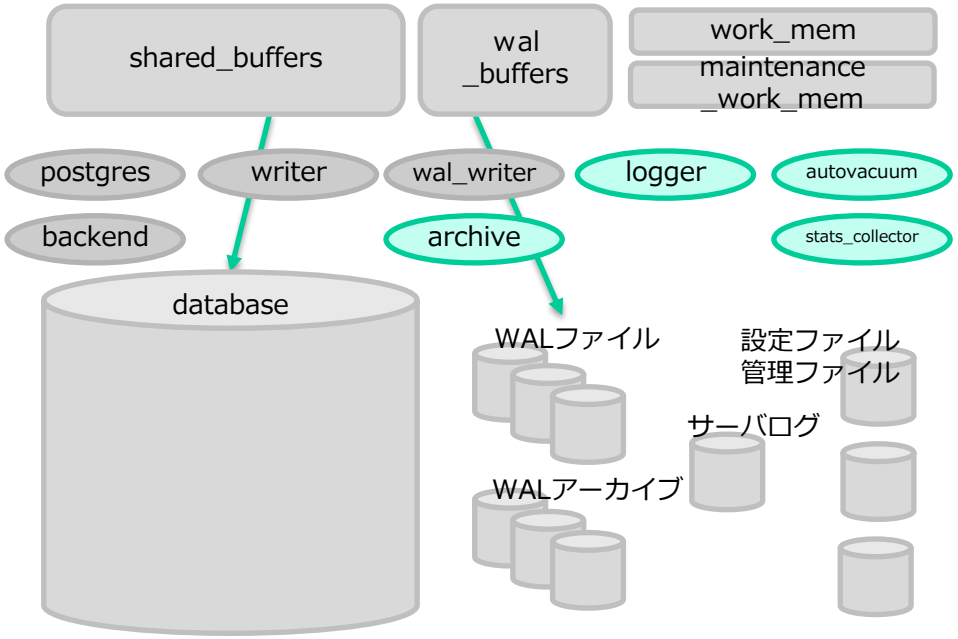
■必須プロセス

- postgres(postmaster)、postgres backend
  - クライアントからの接続を待ち受ける、すべてのプロセスの親プロセス
  - postgresプロセスによって起動され、クライアントからの処理を担当
- writer
  - 共有バッファのデータをディスクに書き込むプロセス
  - チェックポイントやダーティバッファの書き込み
- wal writer
  - データの変更履歴を
    WALファイルに書き込む



パラメータ設定により起動するプロセス

- logger
  - PostgreSQLサーバ実行時のログを記録するプロセス
  - パラメータ設定により有効化し、何をどこに保存するか指定できる
- archive
  - チェックポイント以前の不要なWALをPITRのために別のディスクに退避
- autovacuum launcher/worker
  - 自動VACUUMを実行
- stats collector
  - 実行時統計情報を収集する



1．yum install （最も簡単な方法）

```
# groupadd -g 3001 postgres
# useradd -u 3001 -g 3001 postgres

/* インターネットにアクセス可能、またはOSメディアをyumリポジトリに追加している前提 */
# yum install -y ¥
postgresql11 postgresql11-libs postgresql11-server postgresql11-contrib
```

2．ソースコードからビルド（詳細オプションを指定可能：試験で問われる可能性）

```
/* 任意のディレクトリにあらかじめ postgresql-xx.x.tar.gz を転送しておく */
$ tar zxvf postgresql-xx.x.tar.gz
$ cd postgresql-xx.x
$ ls
COPYRIGHT       HISTORY  Makefile  aclocal.m4  configure     contrib  src ...
$ ./configure --prefix /home/silver/pg_home --with-libxml --with-openssl
$ make world
$ make install-world
```

⇒1．2．いずれの方法とも、この時点で
PostgreSQLの操作コマンドが使用可能になる

インストール後、initdbでデータベースクラスタを初期化

```
$ mkdir silver_data
-----
$ export PGHOME=/usr/pgsql/11
$ export PGDATA=/home/postgres/silver_data
$ export PATH=$PGHOME/bin:.:$PATH
-----
$ env | grep PGDATA
PGDATA=/home/postgres/silver_data
$ initdb -E utf8 --no-locale -D $HOME/silver_data
```

「初期化」＝データディレクトリおよびデータベースクラスタを構成するファイル一式を指定したパスに作成すること。
P.17のオプション指定はこのときに行い、作成するデータベースの用途に応じた設定を行うことができる。

PostgreSQLの構成ファイル一式の実態を確認しておく

```
$ ls -ltr $PGDATA
drwx------. 2 silver postgres  4096  8月 18 09:21 2015 pg_snapshots
drwx------. 2 silver postgres  4096  8月 18 09:21 2015 pg_serial
drwx------. 2 silver postgres  4096  8月 18 09:21 2015 pg_dynshmem
drwx------. 2 silver postgres  4096  8月 18 09:21 2015 pg_twophase
drwx------. 2 silver postgres  4096  8月 18 09:21 2015 pg_replslot
drwx------. 4 silver postgres  4096  8月 18 09:21 2015 pg_multixact
  :
drwx------. 2 silver postgres  4096  8月 18 09:23 2015 pg_notify
drwx------. 2 silver postgres  4096  8月 18 09:39 2015 pg_stat_tmp
-rw-------. 1 silver postgres    34  8月 18 09:23 2015 postmaster.opts
drwx------. 2 silver postgres  4096  8月 18 09:21 2015 pg_clog
-rw-------. 1 silver postgres    88  8月 18 09:21 2015 postgresql.auto.conf
-rw-------. 1 silver postgres  1636  8月 18 09:21 2015 pg_ident.conf
-rw-------. 1 silver postgres  4462  8月 18 09:21 2015 pg_hba.conf
-rw-------. 1 silver postgres 21268  8月 18 09:22 2015 postgresql.conf
drwx------. 3 silver postgres  4096  8月 18 09:21 2015 pg_wal
drwx------. 6 silver postgres  4096  8月 18 09:24 2015 base
drwx------. 2 silver postgres  4096  8月 18 09:39 2015 global
-rw-------. 1 silver postgres    82  8月 18 09:23 2015 postmaster.pid
```

■PostgreSQLサーバーの起動

- データベースクラスタに対してインスタンスが稼働
  - インスタンス＝各役割を担うプロセスが起動し必要なメモリを確保
  - 変更履歴(WAL)やログファイル、各種設定値はデータベース間で共有

■起動・停止

- 起動コマンド

```
/* PostgreSQL実行ユーザにsu */
# su - postgres

/* 起動 */
$ pg_ctl start

/* 停止 */
$ pg_ctl stop
```

・各OS向けにサービスが提供される

```
/* Linux  */
# systemctl start postgresql-11
# systemctl stop postgresql-11

/* Windows */
DOS> net start postgresql-11-x64
DOS> net stop postgresql-11-x64
```

■初期データベースに接続

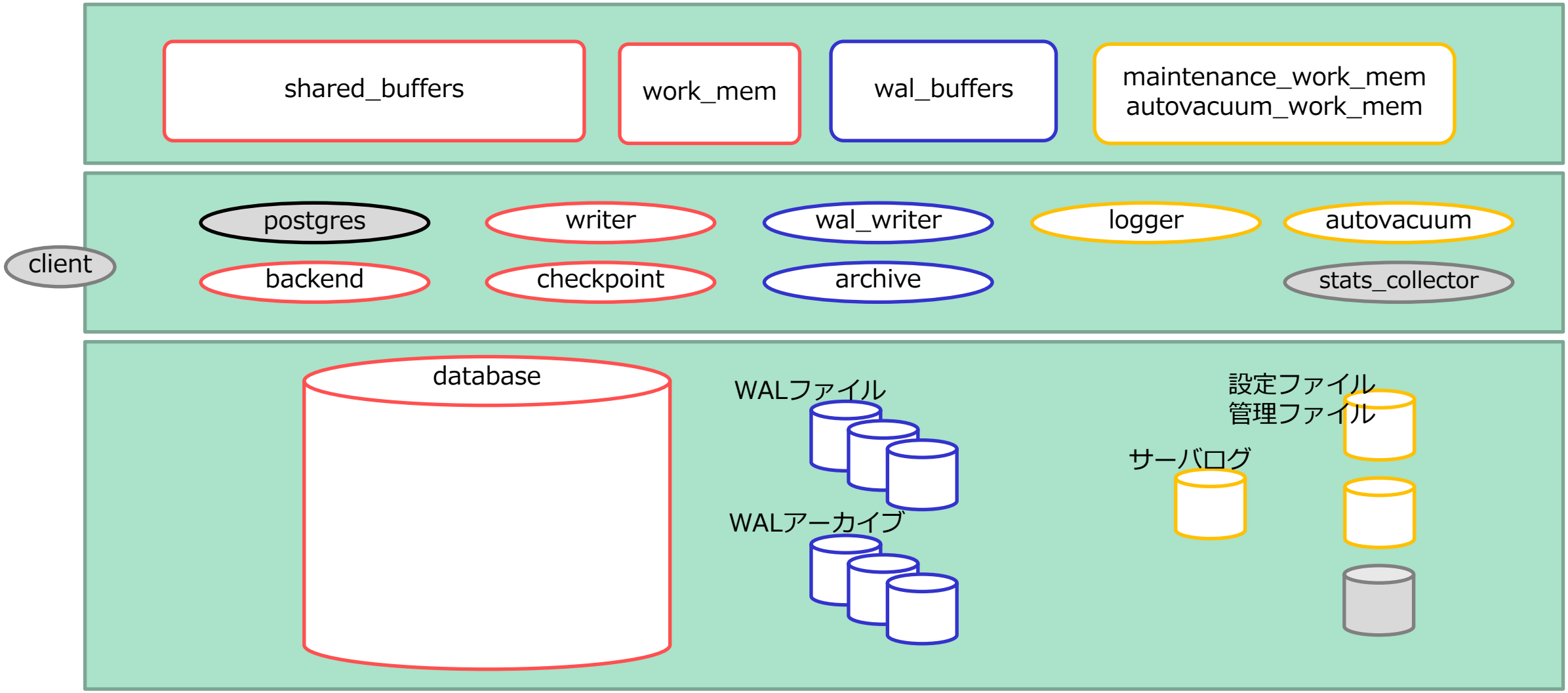
- 初期データベース名：postgres
- 初期ユーザ名　　：postgres（または、initdbを実行したOSユーザ名）

```
$ pg_ctl start   /* データベースクラスタを起動 */
$ psql -h localhost -p 5432 -U postgres -d postgres   /* データベースを指定して接続 */
postgres=#
```

■ユーザーデータベースの作成

```
postgres=# CREATE DATABASE silver OWNER postgres;
postgres=# ¥l
                          List of databases
   Name    | Owner  | Encoding | Collate | Ctype | Access privileges
-----------+--------+----------+---------+-------+-------------------
 postgres  | silver | UTF8     | C       | C     |
 silver    | silver | UTF8     | C       | C     |
 template0 | silver | UTF8     | C       | C     | =c/silver        +
           |        |          |         |       | silver=CTc/silver
 template1 | silver | UTF8     | C       | C     | =c/silver        +
           |        |          |         |       | silver=CTc/silver
```

■データベース構造をもとに、SQL実行時にそれぞれがどう動くか理解する

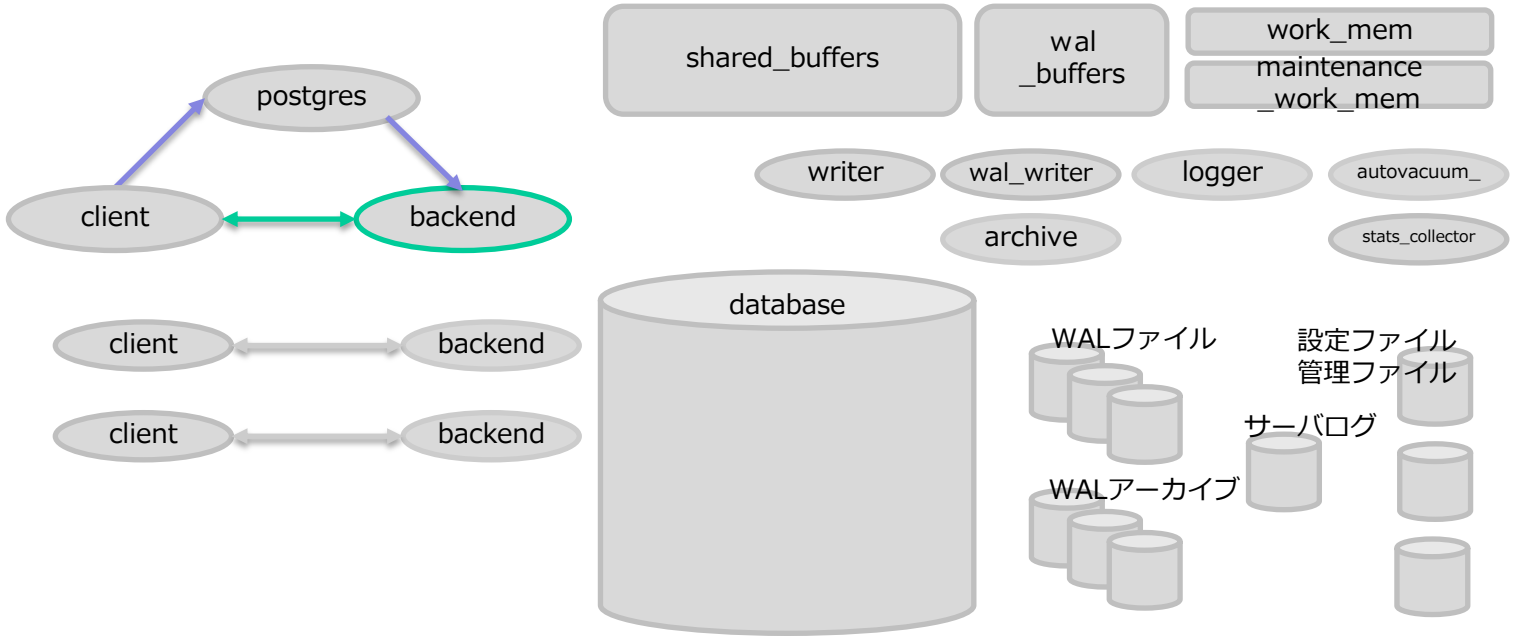


■セッション開始：postgresプロセスが認証を担当

①クライアントから認証要求

②postgresプロセスによる認証後、backendプロセスが起動しセッションを確立

※セッション毎にbackendプロセスが起動され、クライアントと1対1対応する



■参照：共有バッファを利用

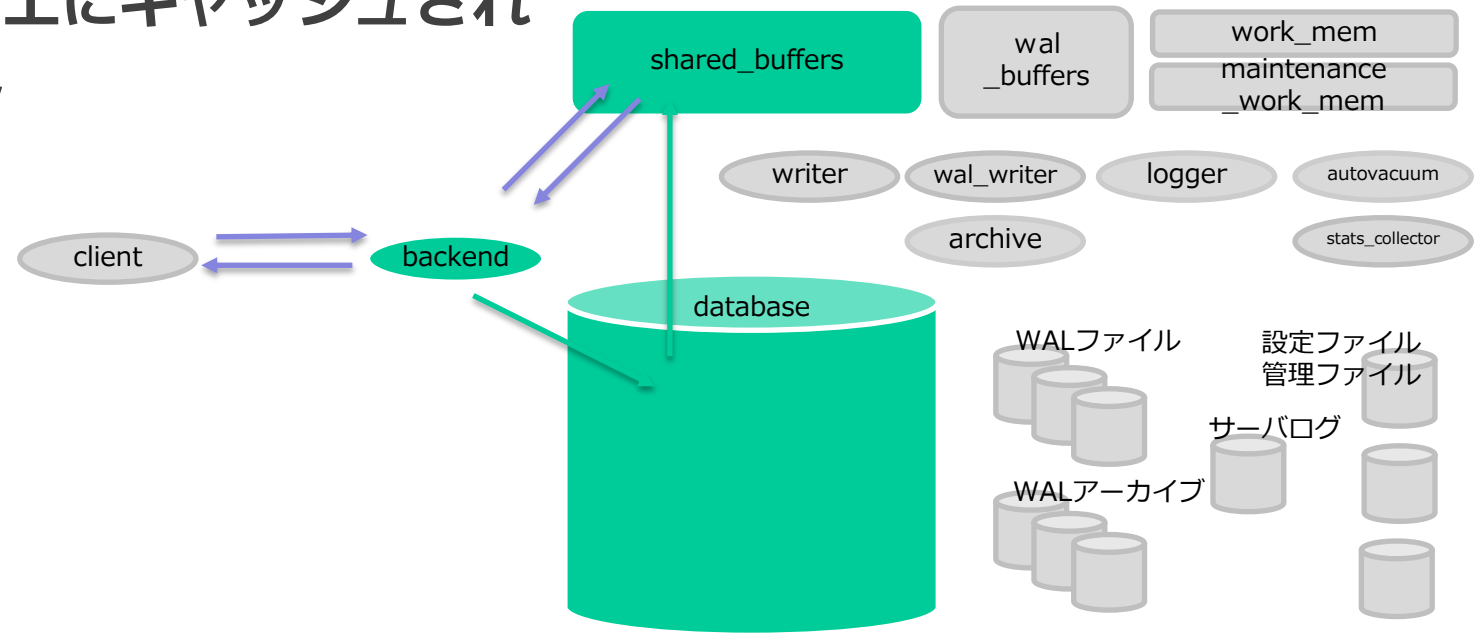
①クライアントからクエリ発行

②backendプロセスが必要なデータを共有バッファから探す

③バッファ上に無い場合は、ディスクの該当ブロックをバッファに載せる

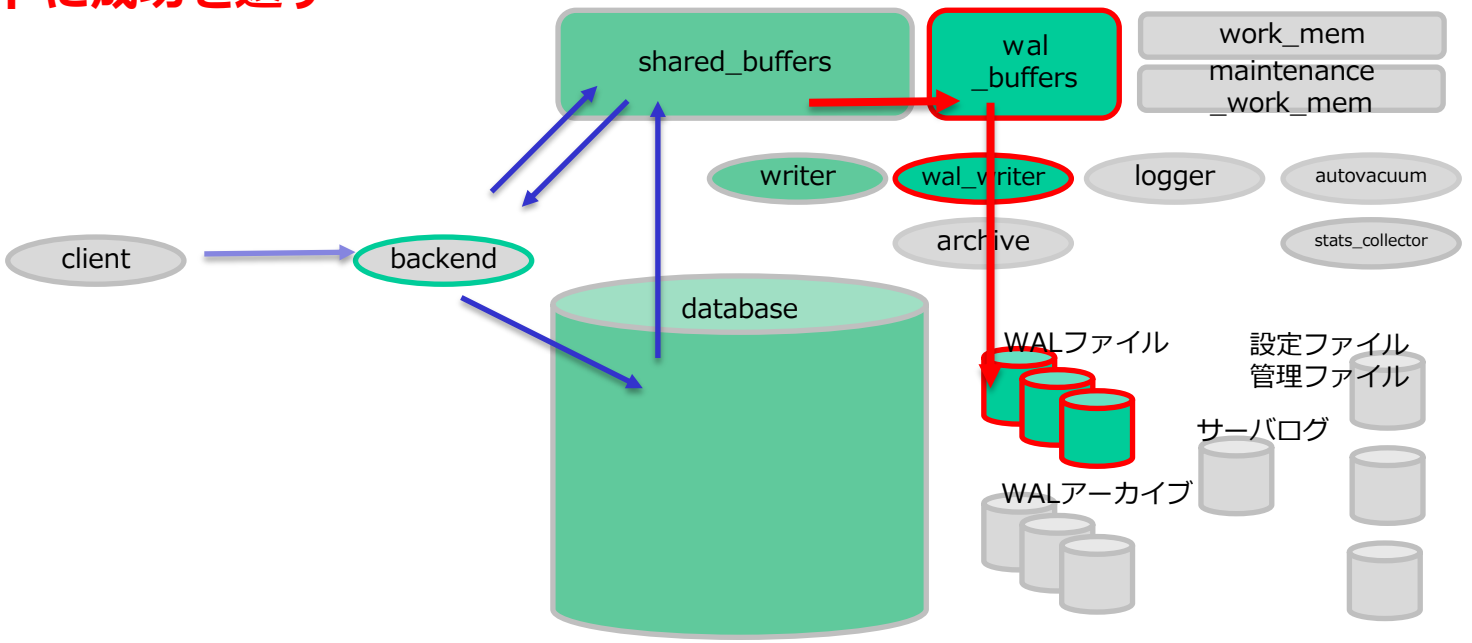
④backendプロセスがクライアントに結果を返却

※アクセスしたデータはバッファ上にキャッシュされ
以降は高速に結果を返す仕組み



■更新：共有バッファ上の更新＋WALによる変更履歴の永続化

①クライアントから更新処理を発行(UPDATE、INSERT、DELETE)
②backendプロセスが必要なデータを共有バッファから探す
③バッファ上に無い場合は、ディスクの該当ブロックをバッファに載せる
④変更内容をWALバッファ上のWALレコードとして作成
⑤共有バッファ上のデータを更新
⑥クライアントが変更を確定(COMMIT)すると、WALレコードをWALファイルに永続化し、
WAL書き込みが成功したらクライアントに成功を返す

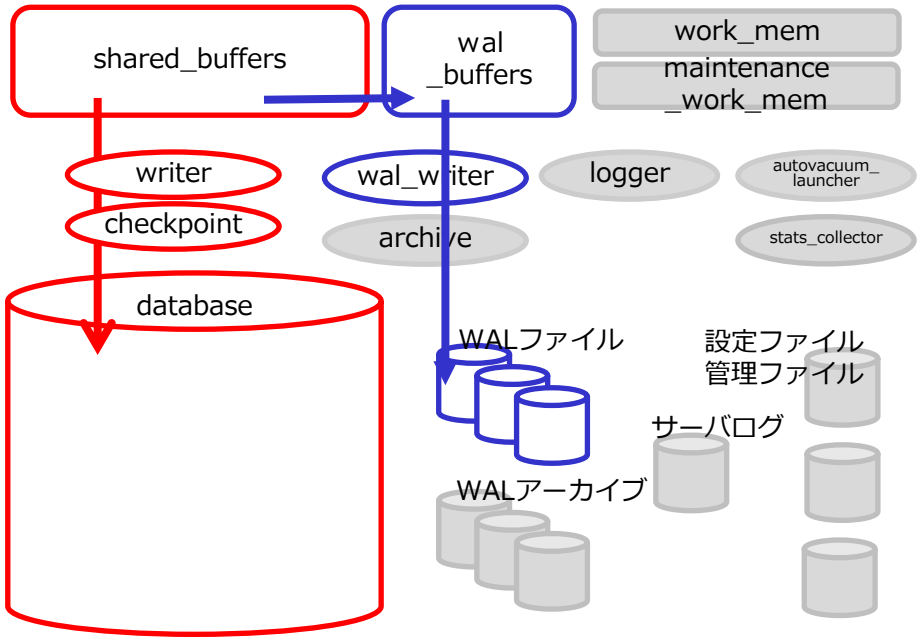


■共有バッファ上のデータをデータファイルに書き込む

- チェックポイントは個々のSQL実行とは非同期に行う
- チェックポイントとWAL
  - チェックポイント以降のWALファイルは非常に重要
    - WAL ⇒ 変更履歴を追跡可能なように都度ディスクに記録
    - チェックポイント以降のWALは、変更履歴を追跡するために必要

ディスク書き込み	特徴
ダーティバッファの書き込み	更新とは非同期に、システムの負荷を極力抑えて実行される。どこまで書くか保証しない。
チェックポイント	ある瞬間のバッファの内容が確実にディスクに反映されたことを保証するタイミング。大量のI/Oが発生する。

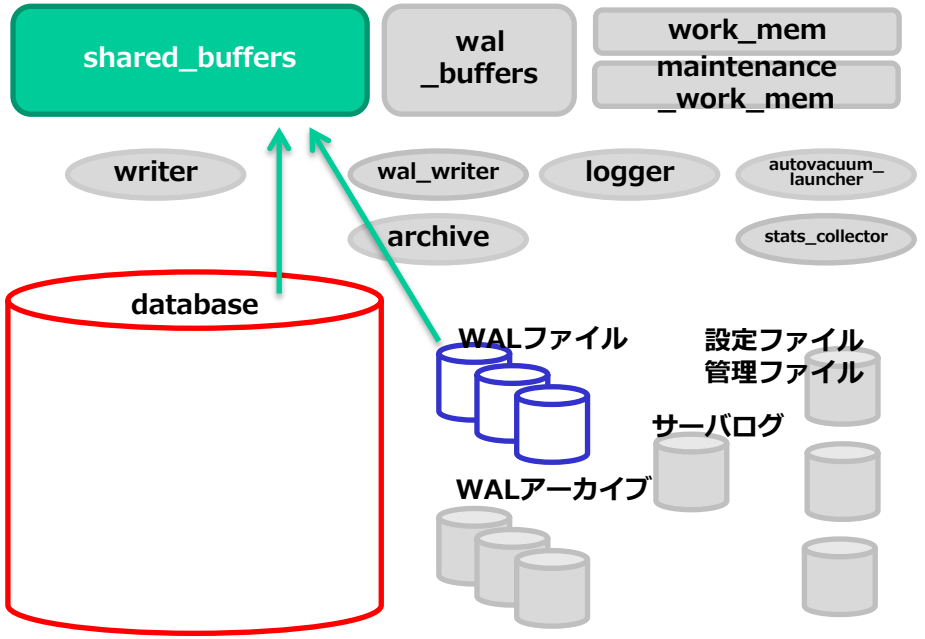
wal writerの動作	特徴
WALファイルへの書き込み	COMMIT時、確実にディスクに書く。一定時間経過やWALバッファが不足する際にも書き込み。



■ディスクに書きこまれた(永続化された)データ
- チェックポイント時点の状態が確実に反映されたデータファイル
- SQLによる変更を刻一刻と記録しているWALファイル

■インスタンス障害、電源障害など
- チェックポイント以降のデータファイル＋WALが残っているため、管理者は特別な操作をせずデータベースの再起動で復旧が可能

■ディスク障害
- データファイルが故障した場合、前回のバックアップにWALを順次適用することで最新状態まで復旧
- WALが故障した場合は、変更履歴を残せないためそれ以上の更新を許さずサーバーダウン
  →WALをリセットしてサーバー再開



RDBMSの共通操作言語

- データの挿入、参照、更新、削除などすべての操作
- 同時実行時のルール設定
- ユーザー定義、権限設定
- データを格納するテーブル定義　など

「SQL標準」で規定された
RDBMSの共通操作言語であり、
どのDB製品でも基本的に同じ

分類	コマンド例	特徴
問合せ Query	SELECT	表名、列名を指定して、条件に合致する行データを取得する。**関連する項目を条件に複数の表を結合**できる。
データ操作 DML	INSERT、UPDATE、DELETE	表を指定して新規行データの挿入、既存行の更新・削除を行う。問合せ同様、条件に合致する行に対する操作であり、複数行をまとめて操作できる。
データ制御 DCL	BEGIN、END、ABORT GRANT、REVOKE	トランザクションを明示的に制御するほか、データへのアクセス制御の管理なども行う。
データ定義 DDL	CREATE、DROP、ALTER	表や索引などのオブジェクトを作成、管理する。

1つ処理≠1回のSQL命令

- トランザクションとは、一連の操作をひとまとまりとする処理単位
- BEGINで開始/COMMITで完了もしくはROLLBACKで破棄するまでを1トランザクションとして扱う
- トランザクション内の命令をまとめて（現実世界での）1つの操作とみなす

トランザクション中の排他制御

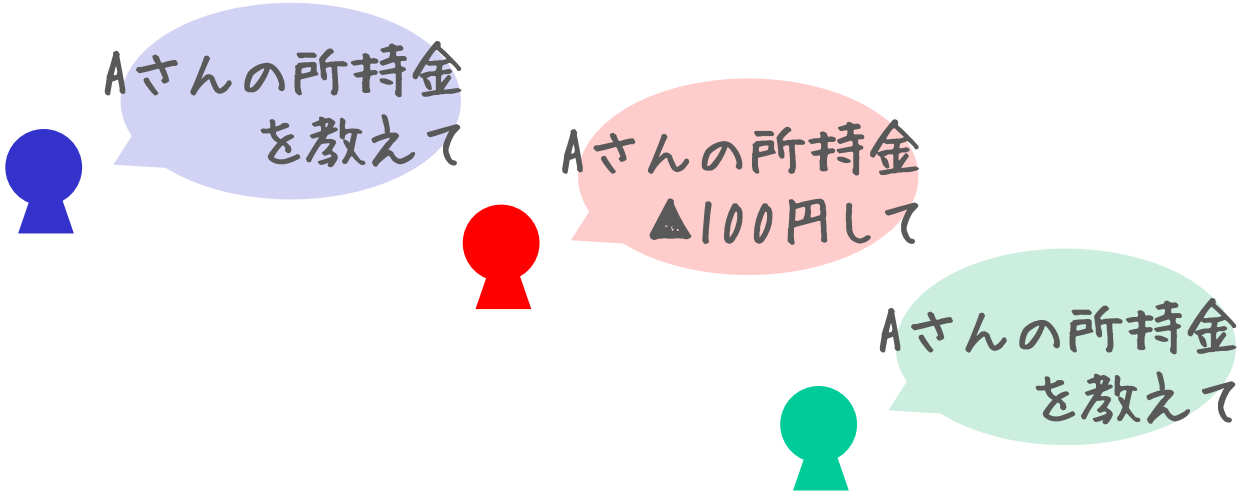
- 複数人が同時にデータベースにアクセスし、自由に更新や参照する
- ある瞬間の「正しいデータ」は何か
  - 変更中データは確定まで他のセッションから不可視　読み取り一貫性
  - 同一の行を他セッションから更新されないよう保護　行ロック

MVCC(Multi Version Concurrency Control)

- 「追記型」とも言われるアーキテクチャ
- とあるAというデータ（Ver.１）と、その行が更新されたA（Ver.2）という2つのバージョンが同時に存在する（更新でも上書きでなく追記している）

	id	value
	A	1000
	B	2000
	C	3000
	A	900

行バージョン管理用の領域



- 変更が確定するまでは、他のセッションからはAが見える
- 変更がコミットされた場合、他のセッションからもAが見える
- 変更がロールバックされた場合、Aは無かったものとし元のAが見える

■VACUUMの必要性

- トランザクションを実現するための「追記型」
  - 1行の更新であっても、繰り返すことでデータが肥大化していく
  - 不要な行を定期的に削除するVACUUM処理が必要

■読み取り一貫性とVACUUM

- 更新後の行であっても、「不要」と判断できるまではVACUUMされない
  ≒更新トランザクション以前の古いトランザクションが残存しないこと

■ロックとVACUUM

- VACUUMは通常の更新を妨げないよう、該当行にのみ弱いロックを必要とする
  - VACUUMと競合する処理が明示された場合、VACUUMがキャンセルされる
  - VACUUMのオプションとして、ロック強度が強く、効果が高いコマンドも存在

アジェンダ

■データベースに求められること

データベースに求められる 「高性能」「同時実行性」「耐障害性」などの基本を整理し
これらを実現するRDBMSの重要なキーワードを解説

■RDBMSの構造

前章で挙げたデータベースとしての基本が、PostgreSQLでは
どのような仕組みで実装されているかを解説

■SQL開発

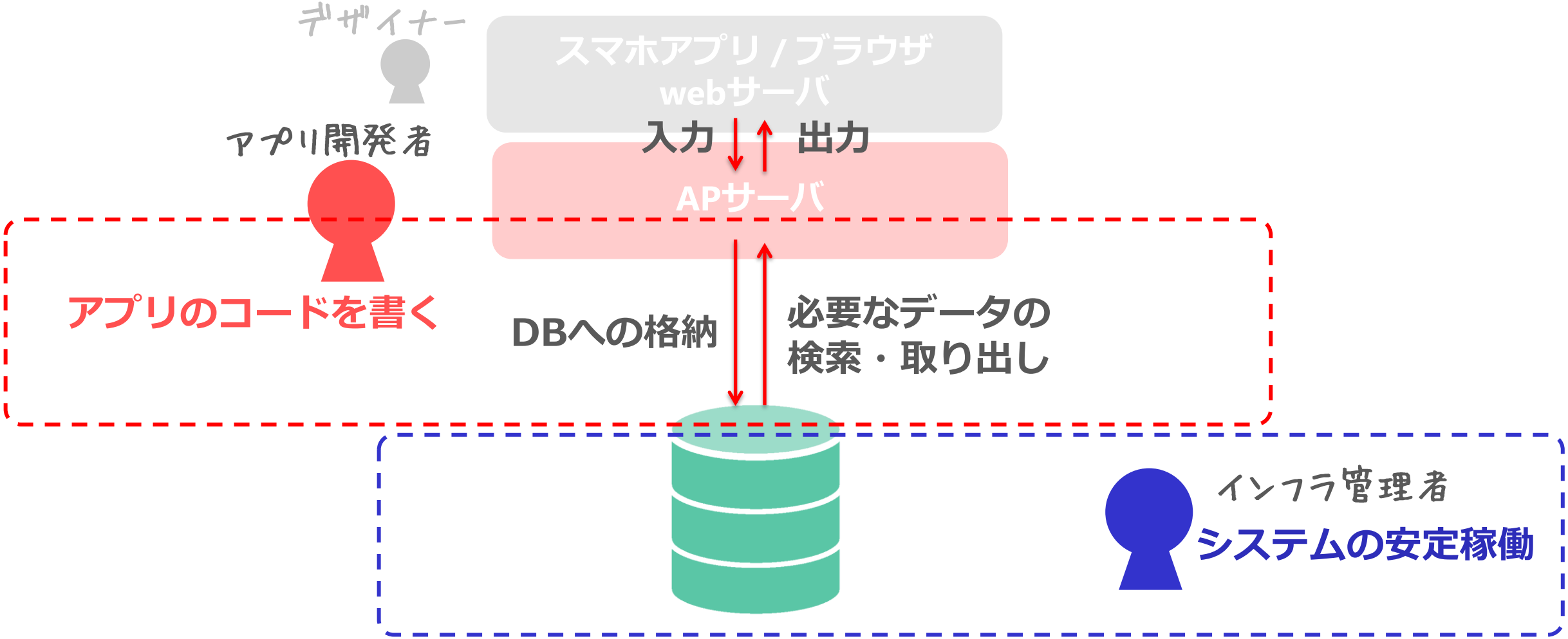
RDBMSの共通言語である 「SQL」 の基本を解説

■DBA(データベース管理者)のタスク

RDBMSの構造から定期的なメンテナンスの必要性を解説し、管理者が実施する
具体的なタスクやその実施方法を解説

■データベースにデータを格納し、取り出す

→ 主にアプリケーション開発者が、SQLを使って必要なデータにアクセスすること



表名や列名を指定し、データへのアクセスを行う

- 列名(選択リスト)、表名(FROM句)の指定は必須
- 条件の指定(WHERE句)により該当する行を選択

dog表から飼い主の住所を検索

```
SELECT name,owner,address
FROM   dog
WHERE  name = 'Poppy';

name   |  owner |  address
-------+--------+-----------
Poppy  | kida   | 千葉県○○市
```

dog表

id	name	kind	owner	address
001	Poppy	Westy	kida	千葉県○○市
002	Mitten	mix	kida	千葉県○○市
003	Pearl	mix	k.kida	東京都△△区
004	Luke	Dachshund	y.kida	神奈川県××市
005	Robbin	Schnauzer	morioka	千葉県○○市
006	Andy	Schnauzer	morioka	千葉県○○市
007	Ace	Jack Russell	sakamoto	東京都△△区

■複数の表を結合して、一つの表のように扱う

- 列名(選択リスト)、結合する全ての表名(FROM句、JOIN句)を指定
- 結合条件(JOIN 表名 ON 条件)を指定

■結合条件　JOIN ONを指定

```
SELECT
d.name,o.o_name,o.o_address
FROM   dog d JOIN owner o
ON     d.owner_cd = o.owner_cd
WHERE  name = 'Poppy';

name   | o_name | o_address
-------+--------+-----------
Poppy  | kida   | 千葉県○○市
```

■dog表

id	name	kind	owner_cd
001	Poppy	Westy	001
002	Mitten	mix	001
003	Pearl	mix	002
004	Luke	Dachshund	003
005	Robbin	Schnauzer	004
006	Andy	Schnauzer	004
007	Ace	Jack Russell	005

■dog表とonwer表を結合

id	name	kind	owner_cd	o_name	o_address
001	Poppy	Westy	001	kida	千葉県○○市
002	Mitten	mix	001	kida	千葉県○○市
003	Pearl	mix	002	k.kida	東京都△△区
004	Luke	Dachshund	003	y.kida	神奈川県××市
005	Robbin	Schnauzer	004	morioka	千葉県○○市
006	Andy	Schnauzer	004	morioka	千葉県○○市
007	Ace	Jack Russell	005	sakamoto	東京都△△区

■owner表

owner_cd	o_name	o_address
001	kida	千葉県○○市
002	k.kida	東京都△△区
003	y.kida	神奈川県××市
004	morioka	千葉県○○市
005	sakamoto	東京都△△区

■結合の種類

- (従来の)結合
  - JOIN句を用いず、WHERE条件で結合
- クロス結合
  - とりうる全ての組み合わせを指す
- 外部結合
  - 片方の表にしかデータが無い場合
  - 例) dog表owner_cd列に「里親募集中」を表すコード「000」が入っている
- 自然結合
  - 結合する表に同じ列名が1つある場合に結合条件を自動で補完
  - ただし、データが持つ意味は考慮されない。同名の列が複数あると使用不可
  - 例) 両者のowner_cd列は同名であり、共通の意味を持つデータを格納

■SQL例

```
-- 従来の結合
SELECT  .name, .o_name, .o_address
FROM
WHERE  d.owner_cd = o.owner_cd
AND    name = 'Poppy';
name   | o_name | o_address
-------+--------+------------
Poppy  | kida   | 千葉県〇〇市

-- クロス結合
SELECT  .name, .o_name, .o_address
FROM
--WHERE  d.owner_cd = o.owner_cd
WHERE  name = 'Poppy';

-- 外部結合
SELECT  .name, .o_name, .o_address
FROM          LEFT OUTER JOIN
ON     d.owner_cd = o.owner_cd;

-- 自然結合
SELECT  .name, .o_name, .o_address
FROM          NATURAL JOIN
WHERE  name = 'Poppy';
```

■検索結果の上位○件を表示

- LIMITで指定した以降のデータの取得を中断し、高速に結果を返す
- OFFSET以降、LIMITまで
- 通常はORDER BY(ソート)と組み合わせ、指定した順位の上位を検索

■dog表(idで降順にソート)

id	name	kind	owner_cd
007	Ace	Jack Russell	005
006	Andy	Schnauzer	004
005	Robbin	Schnauzer	004
004	Luke	Dachshund	003
003	Pearl	mix	002
002	Mitten	mix	001
001	Poppy	Westy	001

```
SELECT * FROM dog
ORDER BY id desc
LIMIT 3 OFFSET 2;

 id |  name  |   kind    | owner_cd
----+--------+-----------+----------
  5 | Robbin | Schnauzer |        4
  4 | Luke   | Duchshund |        3
  3 | Pearl  | mix       |        2
(3 rows)
```

WHERE句の条件に別の問合せ結果を用いる

dog表

id	name	kind	owner_cd
001	Poppy	Westy	001
002	Mitten	mix	001
003	Pearl	mix	002
004	Luke	Dachshund	003
005	Robbin	Schnauzer	004
006	Andy	Schnauzer	004
007	Ace	Jack Russell	005

```
SELECT * FROM dog
WHERE owner_cd = (
  SELECT owner_cd FROM owner
  WHERE  o_name = 'k.kida');
```

```
 id | name  | kind | owner_cd
----+-------+------+----------
  3 | Pearl | mix  |        2
(1 row)
```

owner表

owner_cd	o_name	o_address
001	kida	千葉県〇〇市
002	k.kida	東京都△△区
003	y.kida	神奈川県××市
004	morioka	千葉県〇〇市
005	sakamoto	東京都△△区

■副問合せの結果が1行とは限らない

- LIKEによる曖昧検索

```
SELECT * FROM owner
WHERE  o_name LIKE '%kida%';
 owner_cd | o_name |  o_address
----------+--------+--------------
        1 | kida   | 千葉県○○市
        2 | k.kida | 東京都△△区
        3 | y.kida | 神奈川県××市
```

- WHERE句の演算子を「＝」でなく、「IN」に変更

```
SELECT * FROM dog
WHERE owner_cd =  (SELECT owner_cd FROM owner WHERE o_name LIKE '%kida%');
ERROR:  more than one row returned by a subquery used as an expression

SELECT * FROM dog
WHERE owner_cd IN (SELECT owner_cd FROM owner WHERE o_name LIKE '%kida%');

 id |  name  |   kind    | owner_cd
----+--------+-----------+----------
  1 | Poppy  | Westy     |        1
  2 | Mitten | mix       |        1
  3 | Pearl  | mix       |        2
  4 | Luke   | Duchshund |        3
(4 rows)
```

FROM句に副問合せ結果を用いる

- 結合、ソート、集計済みの結果に対する条件指定をしたい場合
- PostgreSQLでは、インライン・ビューの別名が必須

dog表とonwer表を結合

id	name	kind	owner_cd	o_name	o_address
001	Poppy	Westy	001	kida	千葉県○○市
002	Mitten	mix	001	kida	千葉県○○市
003	Pearl	mix	002	k.kida	東京都△△区
004	Luke	Dachshund	003	y.kida	神奈川県××市
005	Robbin	Schnauzer	004	morioka	千葉県○○市
006	Andy	Schnauzer	004	morioka	千葉県○○市
007	Ace	Jack Russell	005	sakamoto	東京都△△区

```
SELECT * FROM (
  SELECT * FROM dog NATURAL JOIN owner)
  AS dog_with_owner
WHERE o_name = 'k.kida';


 id | name  | kind | owner_cd
----+-------+------+----------
  3 | Pearl | mix  |        2
(1 row)
```

dog表

id	name	kind	owner_cd
001	Poppy	Westy	001
002	Mitten	mix	001
003	Pearl	mix	002
004	Luke	Dachshund	003
005	Robbin	Schnauzer	004
006	Andy	Schnauzer	004
007	Ace	Jack Russell	005

owner表

owner_cd	o_name	o_address
001	kida	千葉県○○市
002	k.kida	東京都△△区
003	y.kida	神奈川県××市
004	morioka	千葉県○○市
005	sakamoto	東京都△△区

■表名や列名を指定し、データへの操作を行う

- 列名(選択リスト)、表名(FROM句)、条件(WHERE)句を指定
  - UPDATE・DELETEは、WEHRE条件が無い場合は列に対する操作
  - INSERTは、列名の指定が無い場合は列の順に挿入する値のリストを記述

■dog表に対する操作

```
-- データのINSERT
INSERT INTO dog
VALUES (008,'Candy','mix','kida','千葉県○○市');

-- データのUPDATE
UPDATE dog SET onwer='a.kida' WHERE id=003;

-- データのDELETE
DELETE FROM dog WHERE id=004;
```

■dog表

id	name	kind	owner	address
001	Poppy	Westy	kida	千葉県○○市
002	Mitten	mix	kida	千葉県○○市
003	Pearl	mix	k.kida	東京都△△区
004	Luke	Dachshund	y.kida	神奈川県××市
005	Robbin	Schnauzer	morioka	千葉県○○市
006	Andy	Schnauzer	morioka	千葉県○○市
007	Ace	Jack Russell	sakamoto	東京都△△区
008	Candy	mix	kida	千葉県○○市
003	Pearl	mix	a.kida	東京都△△区

ロック

- 同じ行に対する更新を防ぐ仕組み
- DMLの対象行はロックされ、別トランザクションの操作を待機させる

デッドロック

- 2つのトランザクションがロックを取り合う状態
  - 片方がエラーになりトランザクション失敗
  - 他方はロック待ちが終わり成功
- デッドロックが発生しないよう、アプリケーション側で考慮

```
-- トランザクションA
BEGIN;
-- データのUPDATE
UPDATE dog SET onwer='a.kida'
WHERE id=003;

-- データのUPDATE
DELETE FROM dog WHERE id=004;
--トランザクションBの確定を待機
```

id	name	kind	owner
001	Poppy	Westy	kida
002	Mitten	mix	kida
003	Pearl	mix	k.kida
004	Luke	Dachshund	y.kida
005	Robbin	Schnauzer	morioka
006	Andy	Schnauzer	morioka
007	Ace	Jack Russell	sakamoto

```
-- トランザクションB
BEGIN;
-- データのUPDATE
UPDATE dog SET onwer='k.kida'
WHERE id=004;

-- データのUPDATE
DELETE FROM dog WHERE id=003;
--トランザクションAの確定を待機
--デッドロックを検知しエラー
```

■表名、列名とデータ型を定義する

- CREATE TABLE文
- テーブル定義からCREATE TABLE文を作成
  または
  テーブル定義からデータのサンプルを想像

■dog表のCREATE TABLE文

```
CREATE TABLE dog
( id        integer
 ,name      text
 ,kind      text
 ,owner_cd integer
);
```

■dog表のテーブル定義

```
dog
----------------------------
id        integer
name      text
kind      text
owner_cd integer
```

■dog表の定義からデータを想像する

id	name	kind	owner_cd
001	Poppy	Westy	001
002	Mitten	mix	001
003	Luke	Dachshund	002
999	xxxxxx	xxx	100

データまで想像するとわかること

- 犬一頭につき1行のデータ
- 飼い主は重複する可能性がある
  ただしdog表とowner表の件数は同等規模
  (常識的に、例えば1000対1となるような超多頭飼いは無い)
- NULL値の可能性を予測
  データの意味を考え、idやnameがNULLの可能性は低い

■列に格納されるデータに対する制約条件を定義する

- CREATE TABLE 時に指定
  ALTER TABLE ALTER COLUMN　などで指定

- PRIMARY KEY制約
- UNIQUE KEY制約
- NOT NULL制約
- CHECK制約
- FOREIGN KEY制約
  (参照整合性制約)

■dog表のテーブル定義(イメージ)

```
dog
----------------------------------------
id        integer  PRIMARY KEY
name      text     NOT NULL
kind      text
owner_cd integer   FOREIGN KEY(owner)
```

■dog表のテーブル定義(確認例)

```
postgres=# ¥d dog
     テーブル "public.dog"
    列    |   型     |  修飾語
----------+---------+----------
 id       | integer | not null
 name     | text    | not null
 kind     | text    |
 owner_cd | integer |
インデックス:
    "dog_pkey" PRIMARY KEY, btree (id)
外部キー制約:
    "dog_owner_cd_fkey" FOREIGN KEY
(owner_cd) REFERENCES owner(owner_cd)
```

【参考】サンプルテーブル

```
DROP TABLE dog;
DROP TABLE owner CASCADE;
CREATE TABLE owner (owner_cd  integer primary key
                   ,o_name    text
                   ,o_address text);
¥d owner
CREATE TABLE dog ( id       integer primary key
                  ,name     text    not null
                  ,kind     text
                  ,owner_cd integer references owner(owner_cd) );
¥d dog

insert into owner values (001,'kida','千葉県○○市');
insert into owner values (002,'k.kida','東京都△△区');
insert into owner values (003,'y.kida','神奈川県××市');
insert into owner values (004,'morioka','千葉県○○市');
insert into owner values (005,'sakamoto','東京都△△区');

insert into dog values (001,'Poppy','Westy',001);
insert into dog values (002,'Mitten','mix',001);
insert into dog values (003,'Pearl','mix',002);
insert into dog values (004,'Luke','Duchshund',003);
insert into dog values (005,'Robbin','Schnauzer',004);
insert into dog values (006,'Andy','Schnauzer',004);
insert into dog values (007,'Ace','Jack Russell',005);

SELECT * FROM dog d NATURAL JOIN owner o;
```

アジェンダ

■データベースに求められること

データベースに求められる 「高性能」「同時実行性」「耐障害性」などの基本を整理し
これらを実現するRDBMSの重要なキーワードを解説

■RDBMSの構造

前章で挙げたデータベースとしての基本が、PostgreSQLでは
どのような仕組みで実装されているかを解説

■SQL開発

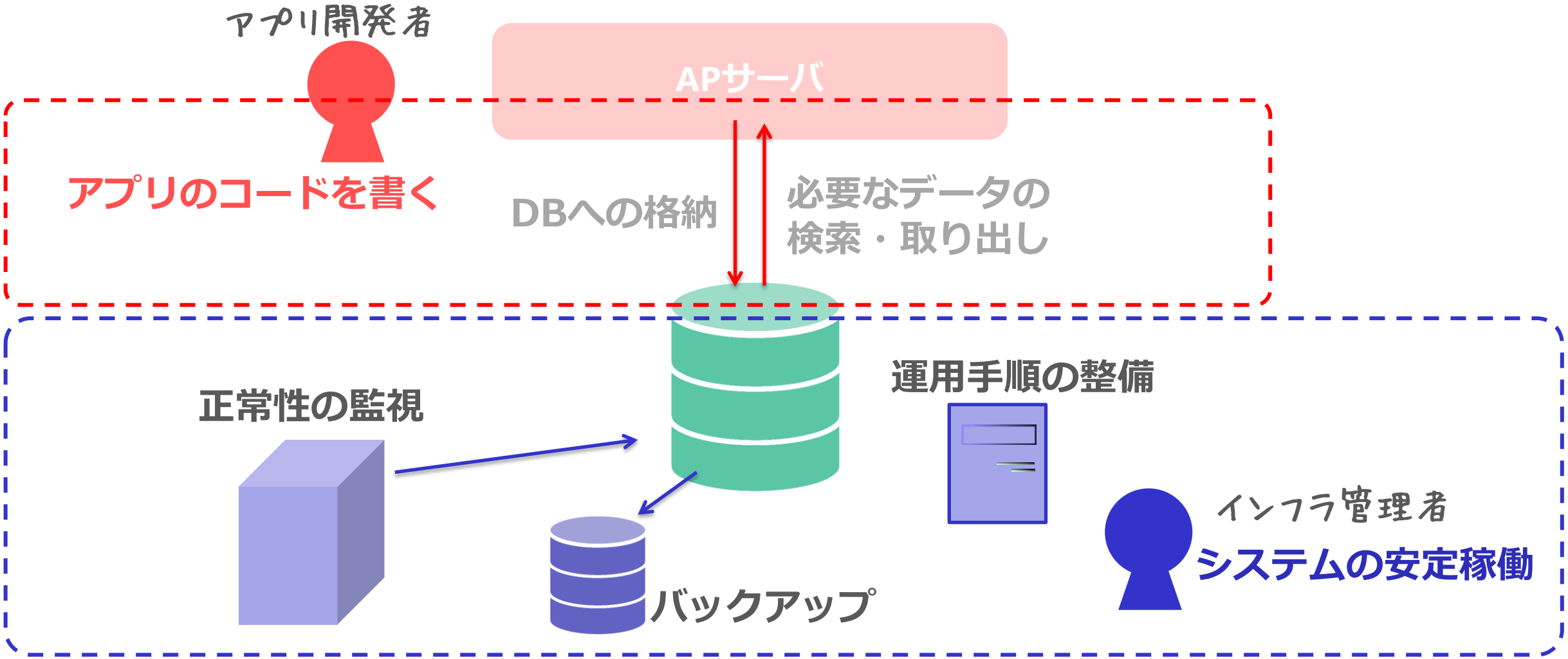
RDBMSの共通言語である 「SQL」 の基本を解説

■DBA(データベース管理者)のタスク

RDBMSの構造から定期的なメンテナンスの必要性を解説し、管理者が実施する
具体的なタスクやその実施方法を解説

データベース管理タスクの位置づけ

■データベースの安定稼働のために正常時から監視や定期メンテナンスを行う



DB管理者の業務

アシスト OSS-DB

分類	タスク	備考
サーバ構築 初期設定	サーバサイジング OS設定、インストール パラメータ設定 セキュリティ設定　など	構築時の初期設定は代表的なパラメータの変更など、ある程度決まった設定で対応可能(**Silver**) 上級では、システム要件から必要なサーバスペックを見積もり、OS設定等を含めた対応が求められる(**Gold**)
監視	死活監視 領域監視 エラー監視 パフォーマンス監視　など	サーバログ出力設定を行い、基礎的なメッセージを理解し対処を行う。 また、正常稼働中のステータス確認やプロセスの状態を知っている。(**Silver**) 各種監視を行い障害を未然に防止する(**Gold**)
メンテナンス	オブジェクトのメンテナンス ユーザのメンテナンス 起動・停止	オブジェクト作成や基本のメンテナンス (**Silver**) 監視情報からメンテナンスの必要性を判断・対処し、障害を未然に防止する(**Gold**)
チューニング	ボトルネックの把握 データベースチューニング SQLチューニング	初期設定時に基本的なチューニングを実施(**Silver**) 監視情報からボトルネックを判断し、適切なチューニングを行う(**Gold**)
障害復旧	バックアップの取得 HA、BCP対策 リストア・リカバリ	標準的なバックアップの手法を理解し、対応可能な障害の種類を整理する(**Silver**) レプリケーション・HA、BCPや環境固有の対策(クラウド機能によるHAなど)を含めた計画を立て、可用性を高く保つ(**Gold**)

正常時の運用管理

- **初期設定**
  - パラメータ設定
  - ユーザ作成 / 認証設定
- **監視**
  - 死活監視 / プロセス監視
  - サーバーログ監視
  - 閾値監視
- **定期メンテナンス**
  - テーブルやオブジェクトのメンテナンス

■ 代表的なメンテナンスコマンド

コマンド	タスク
initdb	データベースクラスタの初期化
pg_ctl pg_isready	データベースの起動・停止 データベース稼働状態の確認 パラメータのリロード
psql	データベースに接続、SQL発行
createdb dropdb	データベースの作成・削除
vacuumdb	データベースまたはテーブルを指定した VACUUMの実施
pg_basebackup	物理バックアップの取得 無停止でデータファイルを複製
pg_dump pg_restore pg_dumpall	論理バックアップの取得 論理バックアップを使用したリストア

■パラメータ設定

- 初期化パラメータpostgresql.confを変更して、サーバ再起動
- 同じくpostgresql.confを変更して、pg_ctl reload
- セッション単位で動的に変更可能などパラメータによって方法が異なる

■代表的なパラメータ

接続関連	port 、listen_addresses、max_connections
メモリ関連	shared_buffers、work_mem、maintenance_work_mem
チェックポイント関連	max_wal_size、checkpoint_timeout
ログ出力関連	logging_collector、log_line_prefix

```
/* pg_settingsビューから現在の設定を参照 */
=# ¥x
=# SELECT name,setting,unit,context FROM pg_settings;

/* pg_settingsビューからパラメータの分類を確認 */
SELECT distinct context FROM pg_settings;
Internal    ・・・変更不可(構築時設定確認用)
postmaster  ・・・サーバ起動時
Sighup      ・・・設定ファイルの再読み込み
Backend     ・・・セッション確立時に決定
Superuser   ・・・スーパユーザ権限で動的変更可能
User        ・・・一般ユーザで動的変更可能
```

■データベースユーザの作成

- 初期ユーザ(一般にpostgresユーザと表記される)はスーパーユーザ
- ログイン属性を持つユーザを作成
  - (標準ツール)createuserコマンド
  - (SQL)CREATE ROLE文

■アクセス制御

- pg_hba.confファイルに記載し、pg_ctl　reload
- どのデータベース/どのユーザへの接続を、どの接続元から許可(拒否)

```
/* pg_hba.confにアクセス制御リストを記述　*/
cd $PGDATA
vi pg_hba.conf
------------------------------------------------------------------
# TYPE  DATABASE        USER            ADDRESS                 METHOD
local   all             all                                     trust
host    silver          kkida         192.168.10.xx/32          md5
host    gold            all           192.168.10.xx/24          reject
------------------------------------------------------------------
/* 設定を読み込み */
pg_ctl reload
```

■サーバログ出力設定

- 初期化パラメータpostgresql.conf の logging_collector = on
- log_line_prefixに時刻やSQL Stateを記録するよう指定(推奨)

■サーバログの何を監視するか

- エラーラベルの監視　log_min_messagesのエラーラベル
  - INFO、NOTICE、WARNING、ERROR、LOG、FATAL、PANIC
  - 重要度の高いものは以下

エラーレベル	内容
PANIC	サーバが停止している
FATAL	セッションが切断されている(他のセッションは正常)
ERROR	該当の処理が失敗し、セッションは残っている

死活監視

サーバの死活監視はプロセス監視またはクライアント接続で確認

- OSコマンド(ps –ef など)で監視
  - postgresプロセスのPIDを確認
  - $PGDATA/postmaster.pidファイルに記録されたPIDと一致
  - 他のプロセスは、postgresプロセスが自動的に再起動する
- SQLによる死活監視
  - 数分間隔で SELECT 1; などの単純なSQLを実行
- 専用コマンドによる死活監視
  - pg_isreadyコマンド
  - 管理コマンドとしてインストールされ、死活監視に利用

正常時	接続不可の場合
$ pg_isready /tmp:5432 – accepting connections $ echo $? 0	$ pg_isready -h localhost -p 5433 localhost:5433 - rejecting connections $ echo $? 1　：起動中などで接続を拒否 2　：無応答 3　：pg_isreadyの実行に失敗

■テーブルの肥大化を抑制する

- VACUUMするとどうなるか
  - 不要な行の位置を記録しておき、次の挿入や更新時に再利用する
  - データファイルを縮小する効果はないので注意
- VACUUMの動作イメージ
  - Visibrity Mapから不要行を検索
  - 使用可能な領域としてFree Space Mapに記録

id	name	kind	owner	address
001	Poppy	Westy	kida	千葉県○○市
002	Mitten	mix	kida	千葉県○○市
003	Pearl	mix	k.kida	東京都△△区
004	Luke	Dachshund	y.kida	神奈川県××市
005	Robbin	Schnauzer	morioka	千葉県○○市
006	Andy	Schnauzer	morioka	千葉県○○市
007	Ace	Jack Russell	sakamoto	東京都△△区
008	Candy	mix	kida	千葉県○○市
003	Pearl	mix	a.kida	東京都△△区

←更新済みの行
←削除済みの行

VM

Visiblity Map
・不要行を追跡可能

FSM

Free Space Map
・不要行の位置を記録
・更新時、FSMから
空き領域を再利用

VACUUMには複数の選択肢がある

種類	内容
コンカレントVACUUM	不要行をFree Space Mapに登録し、再利用可能にする
VACUUM FULL	表の再作成を行い、不要行を詰めて物理ファイルの縮小を行う ※一時的に表サイズの2倍の領域を使用するため、ディスク不足時の領域確保には使えない
CLUSTER	表の再作成を行い、かつ併せて指定したインデックスの並び順にデータを並び替えた状態で格納する
VACUUM FREEZE	トランザクションID周回問題への対処

- VACUUM FULLやCLUSTERはロック強度が強いため、これらを実施しなくても良いようにコンカレントVACUUMを適宜実行する（自動VACUUM）

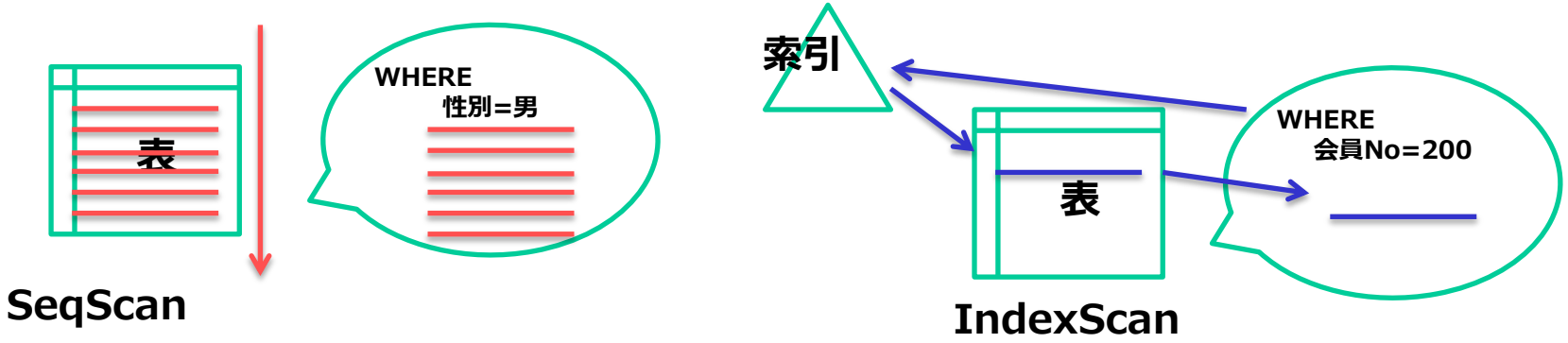
自動VACUUM

- デフォルトでは自動VACUUMが有効
- テーブルに対する更新量を追跡し、一定割合の更新があるとVACUUM実行
- 同じ追跡の仕組みで自動ANALYZEも実行されている

大規模表の場合、
VACUUM実行の閾値見直しが重要

ANALYZEによる列統計の収集

- ANALYZEの必要性
  - 必要なデータを高速に検索する仕組みとして「実行計画」がある
    例)テーブル全体をディスクから読みこむ　SeqScan
    索引を使って必要な行だけ読み込む　IndexScan・・・どっちが高速？
  - PostgreSQLが実行計画の候補を複数作成し、最適なものを実行する



  - ANALYZEで対象列にどのようなデータがどのような分布で格納されているか
    サンプリング。最適(な可能性が高い)実行計画を作る事ができる。

■データベースの基本を解説

- データベース技術者としての入門レベルであり、PostgreSQLを扱う上で必須知識であるOSS-DB Silver試験に向けた学習のきっかけに。
- 初級者がPostgreSQLを使用したデータベース学習を進められることを目標とする。

■取り扱う内容

- データベースに求められること
- RDBMSの構造
- SQL開発
- DBA(データベース管理者)のタスク

■講演資料

- OSS-DB Exam Silver技術解説無料セミナー　2015/05/16
  株式会社メトロシステムズ（当時）　佐藤千佳　氏

■Webサイト

- PostgreSQLマニュアル

https://www.postgresql.jp/document/11/html/index.html

■書籍

- 徹底攻略　OSS-DB Silver 問題集[OSDBS-01]対応
  インプレスジャパン刊　ISBN978-4844331933
- SQL逆引き大全363の極意
  秀和システム刊　ISBN978-4798038520
- これならわかる Oracle 超入門教室 第2版 (DB Magazine SELECTION)
  翔泳社刊　ISBN978-4798114262

OSS-DBの普及

- 現代の契約社会を支えるデータベース技術では、これまで商用製品が圧倒的なシェアを有していたが、近年の製品品質の向上や、国内での情報整備、サービス提供企業の存在から、急速にOSS化が進んでいる。
- 商用/OSSを問わず様々なRDBMSの知識を持ち、データベースの構築、運用ができる、または顧客に最適なデータベースを提案できる技術者が求められている。

OSS-DB資格の重要性

- 体系的な知識を持った技術者の存在は採用する製品を選定する観点で重要視され、ベンダ資格がないPostgreSQLにとっては普及の起爆剤となる。
- データベースに限らずOSSを使うことが一般化してきていることから、本資格取得による個人のキャリアアップの面と、さらなる製品の普及促進の面から非常に重要。

アシスト OSS-DB

ご清聴ありがとうございました。

■お問い合わせ■

株式会社アシスト

喜田 紘介

Mail：kkida@ashisuto.co.jp