

# PostgreSQLアーキテクチャ入門

アップタイム・テクノロジーズ合同会社

永安 悟史

2011.7.15

# PostgreSQLアーキテクチャ入門

## アジェンダ

- アーキテクチャ概要
  - PostgreSQLの構成要素
  - PostgreSQLの基本的なアーキテクチャ
  - プロセス
  - メモリ
  - ファイル
  - 発生する3種類のI/O
- クエリの処理
  - SQL文の処理される流れ
  - クエリとクエリプラン
  - クエリプランの詳細
  - クエリプランの確認方法
  - データアクセスのパターン
  - テーブルスキャン
  - インデックスアクセス
  - 結合
- I/O処理の詳細
  - テーブルに対する更新処理
  - タプルの更新とインデックスの更新
  - テーブルに対する参照処理
  - VACUUM処理
- パフォーマンス管理
  - パフォーマンスは何で決まるか？
  - データベースを構成するハードウェアリソース
  - パフォーマンス改善の基本手順
  - 全体の傾向を可視化する
  - SQLパフォーマンス分析
- 可用性管理
  - バックアップとレストア/リカバリ
  - コールドバックアップ
  - ホットバックアップ
  - アーカイブログとPITRを用いたバックアップ
  - アーカイブログとPITRを用いたリカバリ
  - 冗長化方式の選定
- 参考文献

※本資料の最新版は以下に掲載されています。  
<http://www.uptime.jp/>  
(ホーム→リソース→技術情報)

# 本セッションのねらい

- PostgreSQL でシステムを構築して実運用をするためには、データベース管理者 (DBA) として ある程度内部構造を理解しておく必要があります。
- 本講演では、開発や運用において必要とされる技術的知識について、PostgreSQL の基本的な仕組みからバックアップ & リカバリ、レプリケーションまで、PostgreSQL の動作原理を俯瞰して解説を行います。
- 主に PostgreSQL 初級者から中級者向けの内容です。
- 特に以下のような方にオススメです。
  - データベースの特に運用管理・パフォーマンス管理に詳しくなりたい方。
  - コンピュータアーキテクチャに詳しくなりたい方。
  - コンピュータエンジニアリングの基礎を知りたい方。
  - 他のRDBMSを利用して、PostgreSQLについて知りたい方。

# 自己紹介

- 氏名

- 永安 悟史 (ながやす さとし)

- 略歴

- 2004/4-2007/9 (3年6ヵ月)
  - ・ 株式会社NTTデータ入社。
  - ・ PostgreSQLによる並列分散RDBMSの研究開発。
  - ・ SIプロジェクトの技術支援、並列分散PostgreSQLミドルウェアの製品サポートおよび保守。
- 2007/10-2008/9 (1年)
  - ・ データセンタ企画部門にて、次世代ITプラットフォームサービスの企画・開発。
- 2008/10-2009/10 (1年1ヵ月)
  - ・ データセンタ運用部門にて、OSS系システムの基盤保守・運用、および運用チームの統括。
  - ・ 株式会社NTTデータ退職。
- 2009/11-
  - ・ アップタイム・テクノロジーズ創業(共同創業者兼CEO)。

- 専門分野

- データベースシステム、並列分散システム、クラスタシステム
- オープンソース・インフラ技術
- ITサービスマネジメント(ITIL)、ITインフラ運用管理(運用設計～運用)

- 執筆等

- 翔泳社「PostgreSQL徹底入門～8対応」(共著)
- 技術評論社「PostgreSQL安定運用のコツ」(WEB+DB PRESS vol.32～27連載)、他



# WEB+DB PRESS vol.63

- 特集: Web開発の「べし」「べからず」



## 第3章 データベース編

性能を最大限に引き出すための設計・開発・運用

アップタイム・テクノロジーズ合同会社  
永安 悟史 NAGAYASU Satoshi  
Mail snaga@uptime.jp

### アーキテクチャから理解するデータベース

データベースの技術は長い歴史の蓄積があり、その実装であるデータベース製品は非常に複雑なソフトウェアとなっています。そのため、深く理解するにはそれなりの学習期間と経験が必要とします。

本章では、データベースの中でも特にリレーショナルデータベース(RDBMS)について、理解するためのポイント/観点と、具体的に開発/運用時に注意すべき点について解説します。

### データベースの理解は「立体的」に

データベースの挙動をきちんと理解するためには、データベースを構成する要素を「立体的」、つまり複数の角度/軸で理解する必要があります。

### ■処理の流れを理解する

データベースの挙動を理解する第1の軸は「処理の流れ」です。図1に示すような一連の流れで処理を行います。まずはこの流れに沿って、データがどのように処理されていくのかを理解する必要があります。

### ■処理の層(レイヤ)を理解する

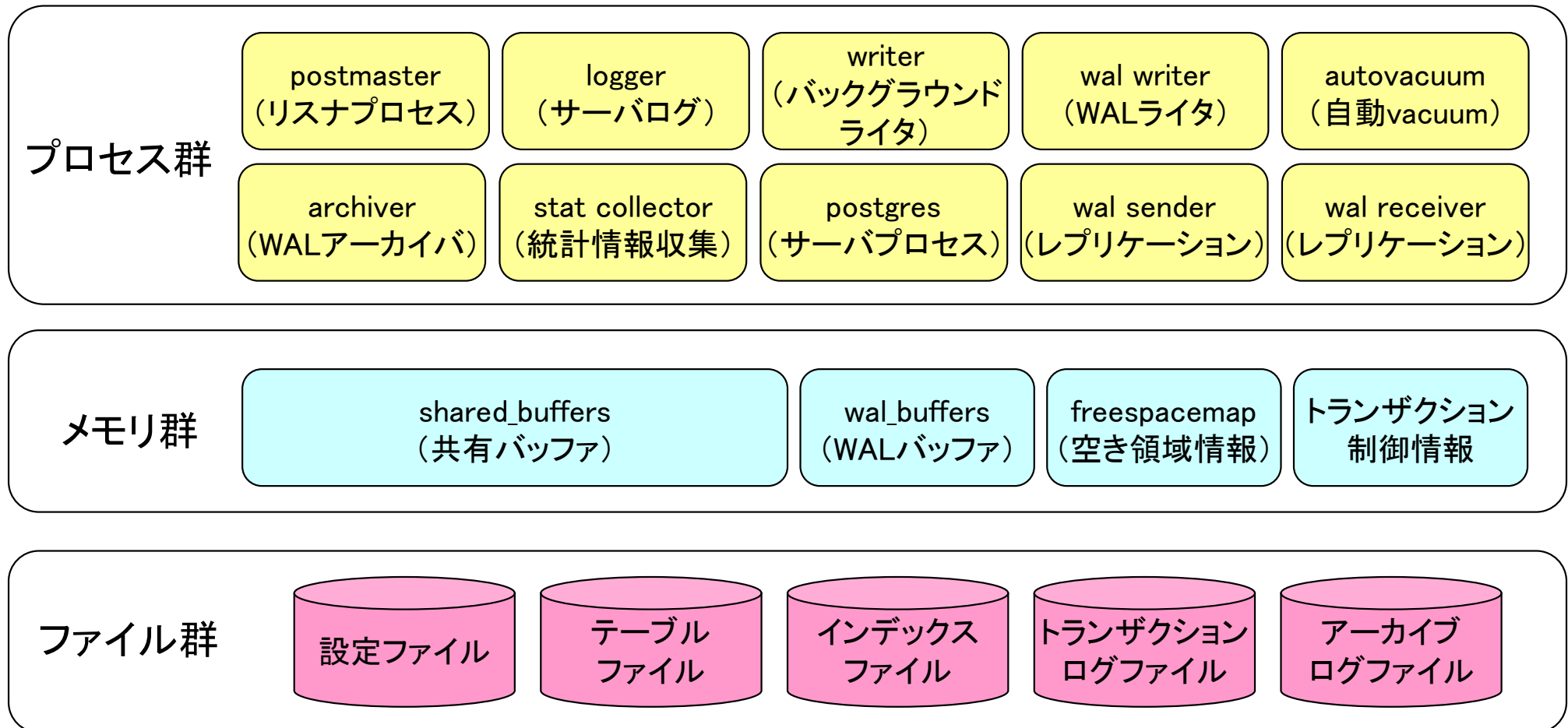
データベースを理解するもう一つの軸は「層(レイヤ)」です。

データベース製品が動作している環境には、CPU、メモリ、ディスク、ネットワークインタフェースカードなどのハードウェアコンポーネントがあります。また、メモリ内には、実行プロセスのメモリ、共有メモリ、ディスクキャッシュなどのメモリ空間があります(図2)。これらのコンポーネントが、それぞれ重要な役割を果たしつつ相互に連携しながら動作しており、個々のコンポーネントの動作が全

## (1)アーキテクチャ概要

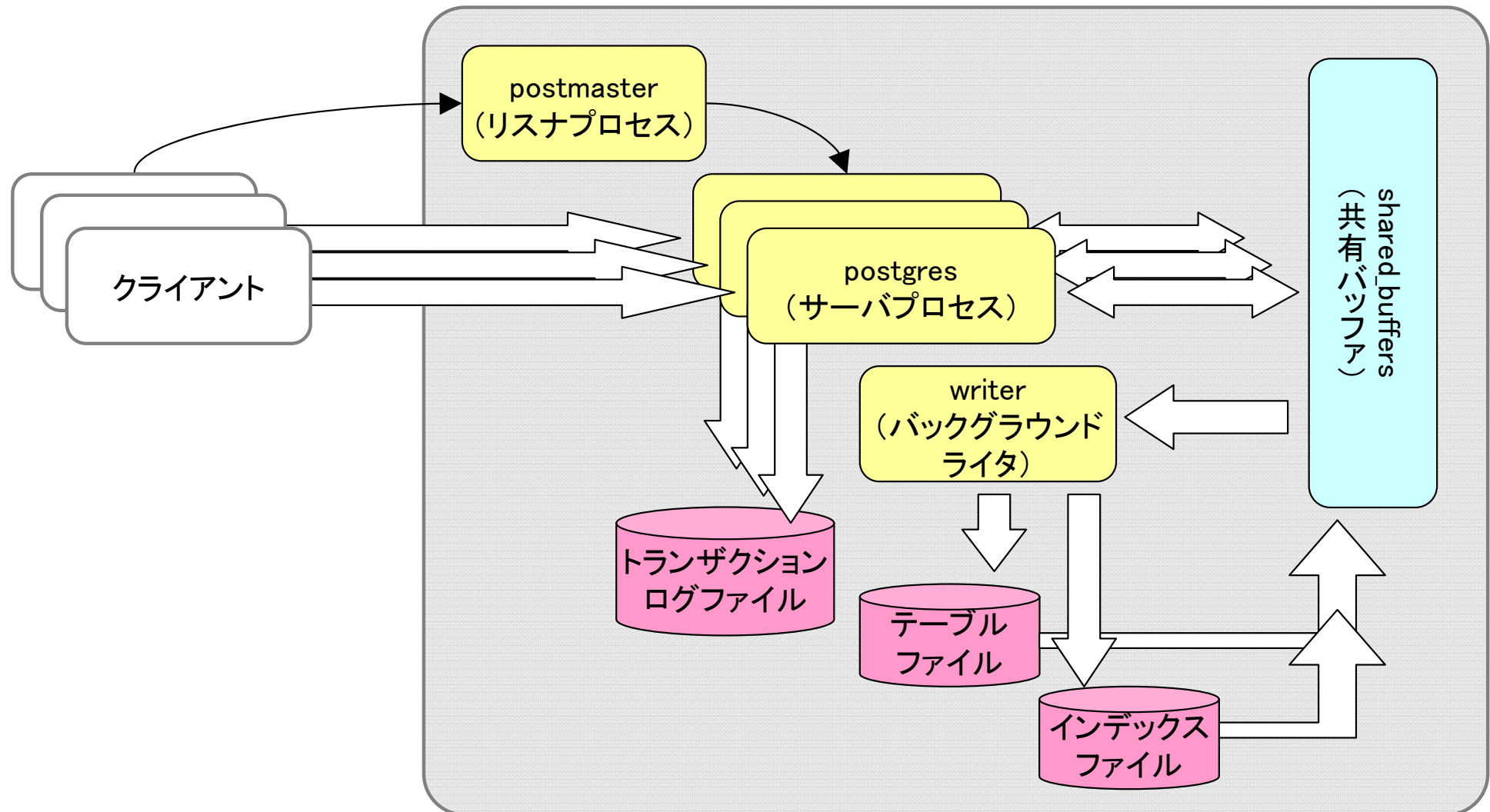
# PostgreSQLの構成要素

PostgreSQLは、さまざまなプロセス・メモリ領域・ファイルによって構成されている。



# PostgreSQLの基本的なアーキテクチャ

共有バッファを中心として、複数のプロセス間で連携しながら処理を行うマルチプロセス構造。

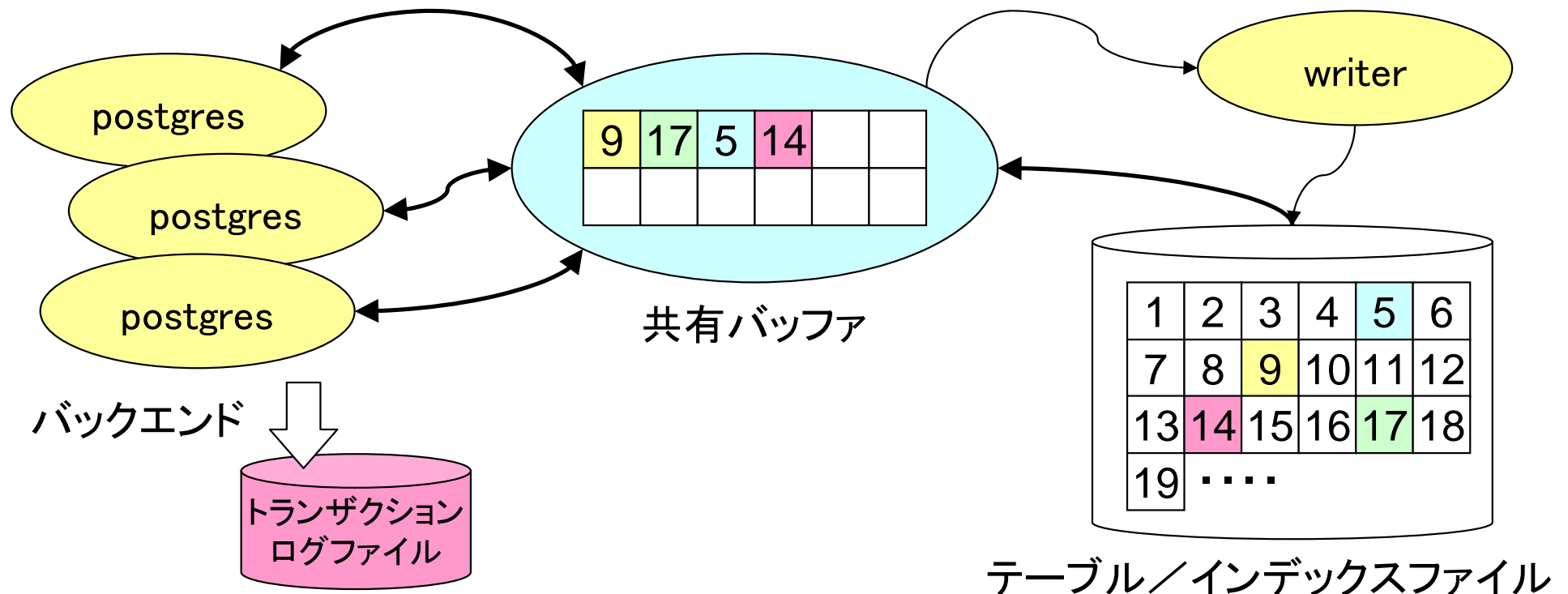


# プロセス

- Postmasterプロセス
  - PostgreSQLを起動すると最初に開始されるプロセス。
  - クライアントからの接続を受け付け、認証処理を行う。
  - 認証されたクライアントに対して、Postgresプロセスを生成(fork)して処理を引き渡す。
- Postgresプロセス
  - クライアントに対して1対1で存在する。
  - クライアントからSQL文を受け付け、構文解析、最適化、実行、結果返却を行う。
  - 共有バッファを介してデータを読み書きし、トランザクションログを書く。
- Writerプロセス
  - 共有バッファの内容をディスク(テーブルファイル、インデックスファイル)に非同期的に書き戻す。バックグラウンドライターとも呼ばれる。

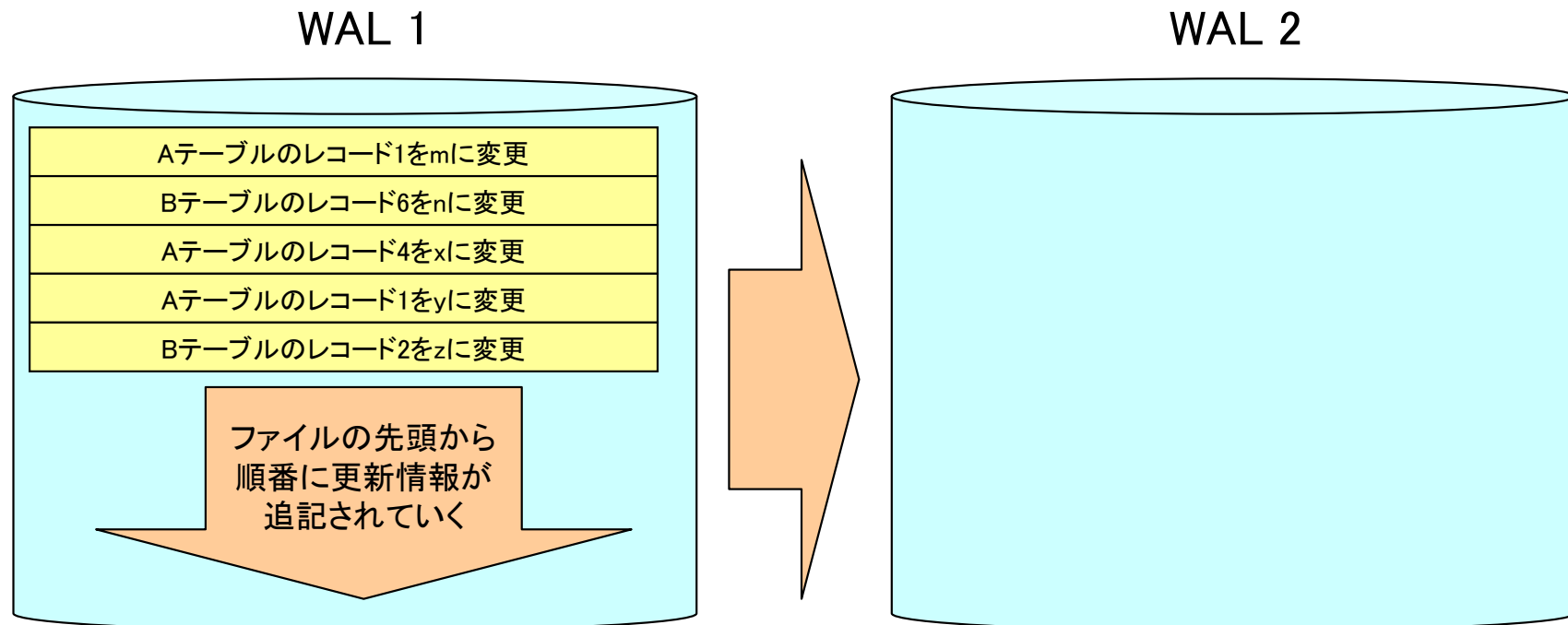
# メモリ(共有バッファ)

- ・ ディスク上のブロックをキャッシュするメモリ領域
  - ディスク上のブロックのうち、アクセスするものだけを読み込む
  - すべてのバックエンドプロセスで共有
- ・ キャッシュすることで、ディスクI/Oを抑えて高速化
  - 更新の永続性はトランザクションログで担保する
  - メモリ上で変更されたブロックは、ライタプロセス(非同期)またはチェックポイント(同期)がテーブル/インデックスファイルに書き戻す



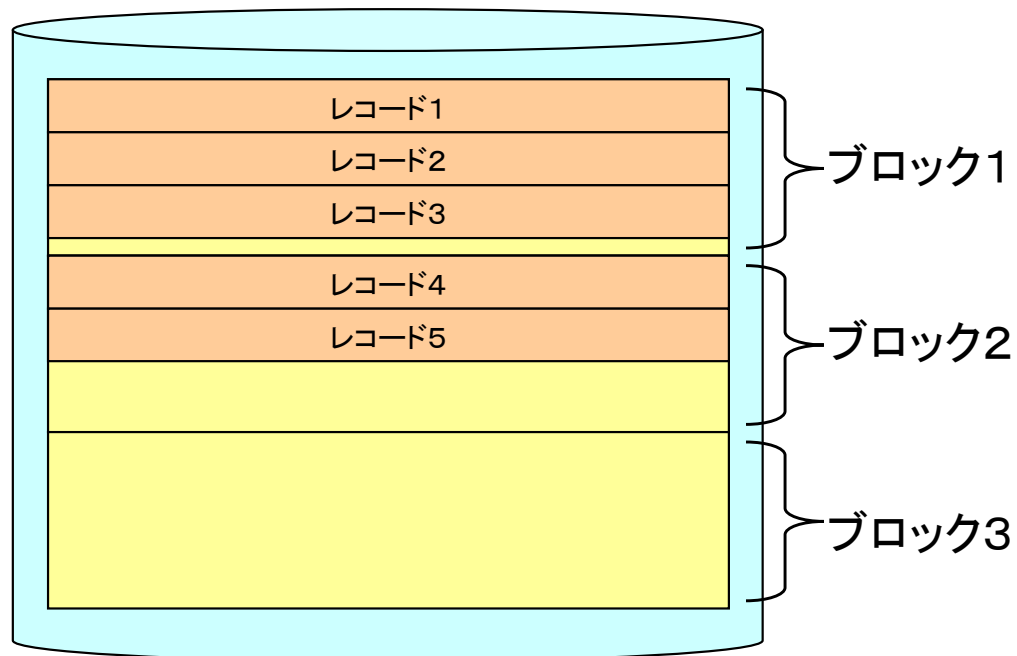
# トランザクションログ

- テーブルやインデックスの更新情報が記録(追記)される
  - 共有バッファのデータを更新する「前」に記録(Write-ahead)
  - 16MBずつのセグメント(ファイル)に分割されている。
  - リカバリの際に読み込まれる (pg\_xlog/ 以下に配置)
  - アーカイブされて、PITRのバックアップ/リカバリで使われる(アーカイブログ)



# テーブルファイル

- ・ 8kB単位のブロック単位で管理
- ・ ブロックの中に実データのレコード(タプル)を配置
  - 基本的に追記のみ
  - 削除したら削除マークを付加する(VACUUMで回収)
  - レコード更新時は「削除+追記」を行う。



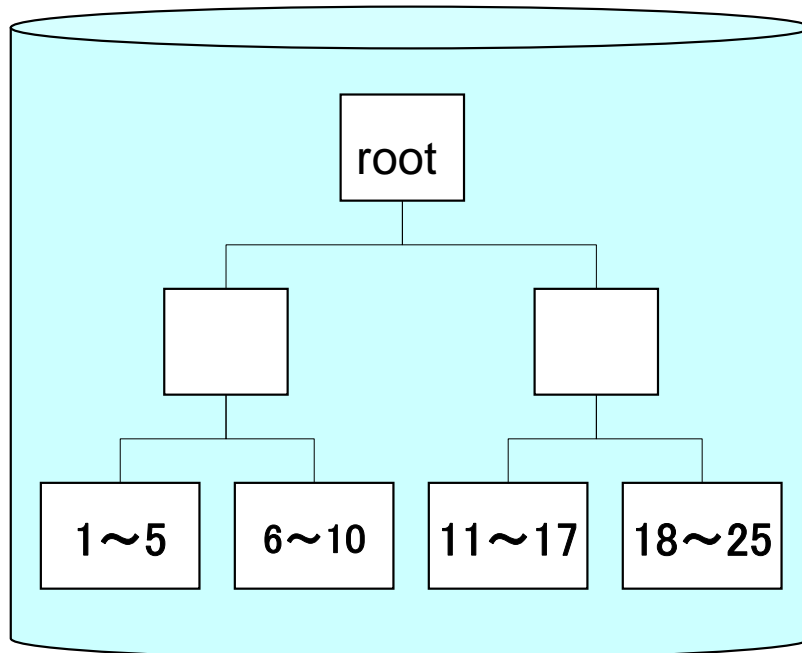
```
DBT1=# SELECT * FROM
pgstattuple('customer');
-[ RECORD 1 ]-----+-----
table_len      | 1754857472
tuple_count    | 3456656
tuple_len      | 1703225491
tuple_percent  | 97.06
dead_tuple_count | 695
dead_tuple_len | 350038
dead_tuple_percent | 0.02
free_space     | 31391624
free_percent   | 1.79
```

```
DBT1=#
```

# インデックス (B-Tree) ファイル

- ブロック(8kB単位)をノードとする論理的なツリー構造を持つ
  - ルート、インターナル、リーフの各ノードから構成
  - ルートノードから辿っていく
  - リーフノードは、インデックスのキーとレコードへのポインタを持つ

インデックスファイル

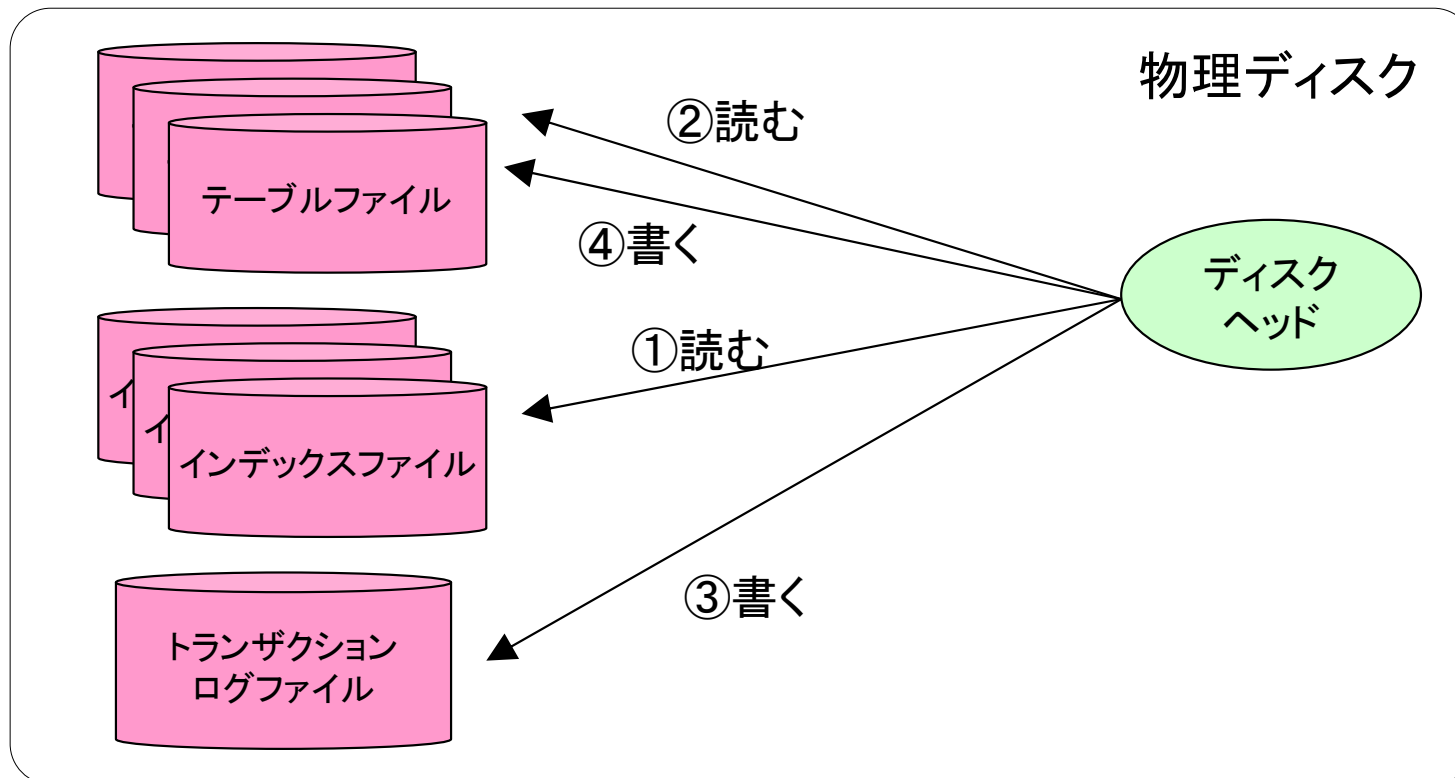


```
DBT1=# SELECT * FROM
pgstatindex('customer_pkey');
-[ RECORD 1 ]-----+-----
version          | 2
tree_level       | 2
index_size       | 108953600
root_block_no    | 217
internal_pages   | 66
leaf_pages       | 13233
empty_pages      | 0
deleted_pages    | 0
avg_leaf_density | 90.2
leaf_fragmentation | 0
```

```
DBT1=#
```

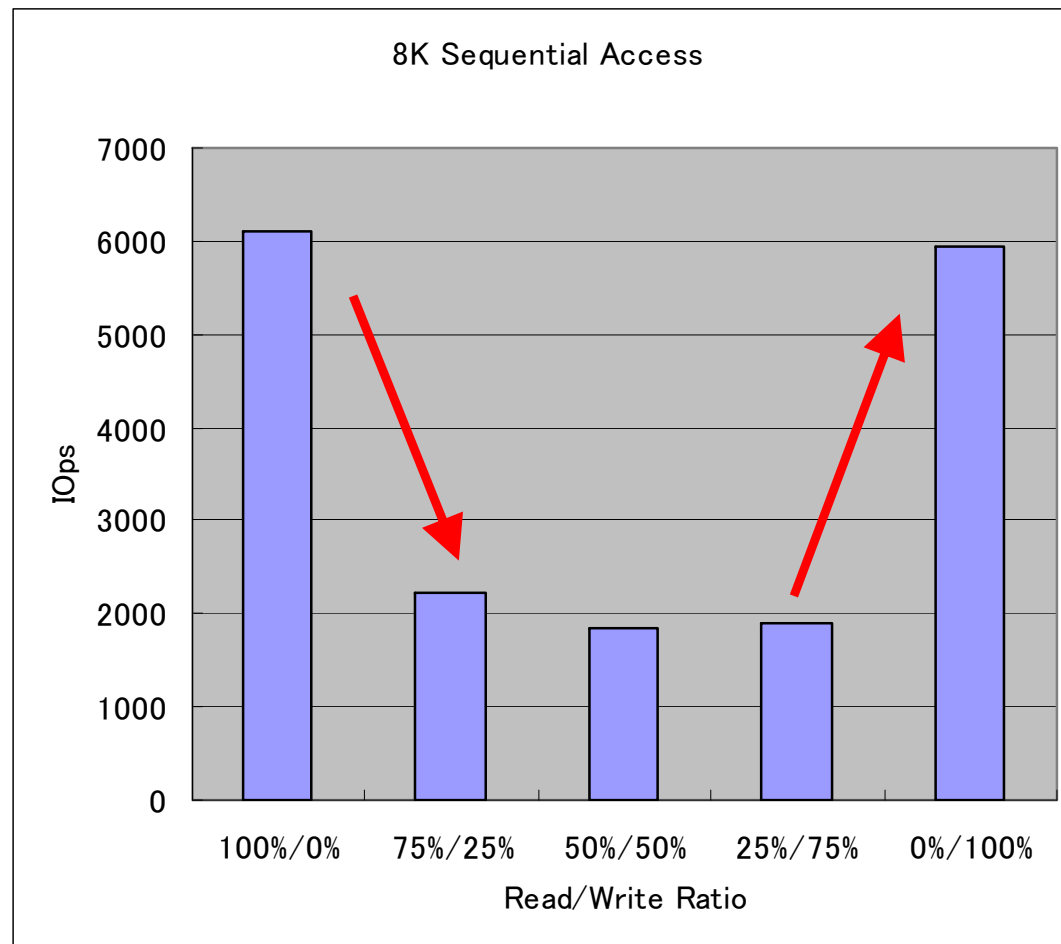
## 発生する3種類のI/O

- 例えば、主キーで検索して該当レコードを更新する場合
  - プライマリーキーでインデックスエントリを探す
  - インデックスのポインタを元に、テーブル内のレコードを探す
  - テーブルレコードを更新前にトランザクションログに記録する
  - テーブルファイルを更新する



## (参考) I/Oパターンの混在

- 複数のアクセスパターンが単一のHDD上に混在すると、パフォーマンスは低下する。
  - 物理ディスクをどのように配置するかがストレージ戦略。



※SATA接続のHDD 1台に対してiometerを用いて8kBブロック単位のI/Oで計測。

## (2) クエリの処理

# SQL文の処理される流れ



# クエリとクエリプラン

The screenshot shows a SQL query editor window titled "Query - DBT1 @ snaga@10.0.2.12:5432". The query text is:

```
SELECT count(*) FROM orders o, order_line ol, customer c
WHERE o.o_id=ol.ol_o_id
AND o.o_c_id=c.c_id
AND c.c_fname='ouE kqP*)/0'
AND c.c_lname='Cckj0eh[byh';
```

The execution plan below the query shows the following steps:

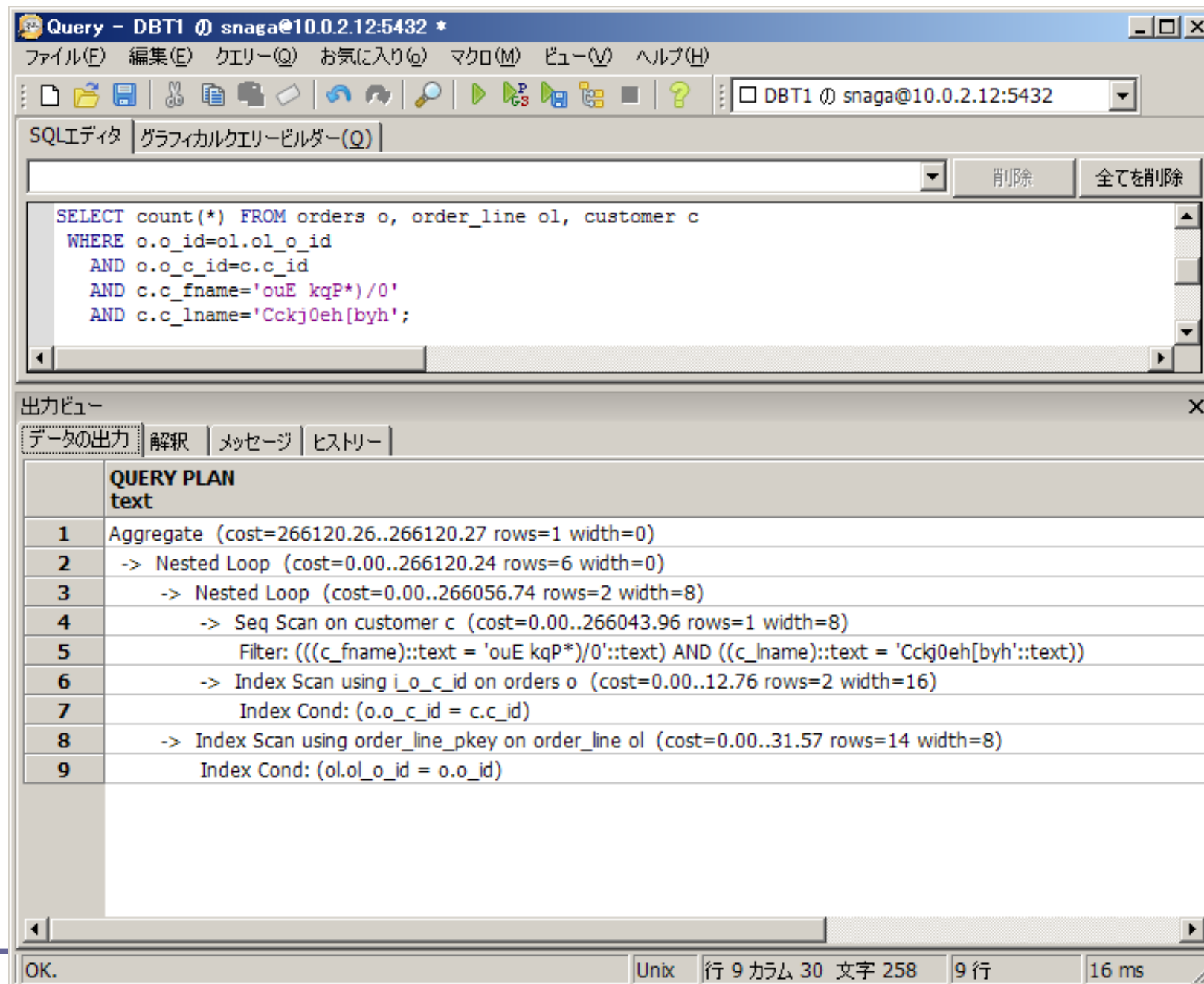
- customer (Table Scan)
- order\_line\_pkey (Index Scan)
- Nested Loop (Join)
- Nested Loop (Join)
- Aggregate (Aggregation)

Annotations in yellow callouts explain the plan:

- "テーブルスキャン" (Table Scan) points to the customer table.
- "インデックススキャン" (Index Scan) points to the order\_line\_pkey index.
- "ネステッドループジョイン" (Nested Loop Join) points to the two Nested Loop operators.
- "集約 count()" (Aggregation count()) points to the Aggregate operator.

The status bar at the bottom indicates: OK. Unix 行 9 カラム 30 文字 258 9 行 16 ms

# クエリプランの詳細



The screenshot shows a SQL query editor window titled "Query - DBT1 @ snaga@10.0.2.12:5432". The query is as follows:

```
SELECT count(*) FROM orders o, order_line ol, customer c
WHERE o.o_id=ol.ol_o_id
AND o.o_c_id=c.c_id
AND c.c_fname='ouE kqP*)/0'
AND c.c_lname='Cckj0eh[byh';
```

The execution plan is displayed in the "出力ビュー" (Output View) section, showing the following steps:

Step	QUERY PLAN text
1	Aggregate (cost=266120.26..266120.27 rows=1 width=0)
2	-> Nested Loop (cost=0.00..266120.24 rows=6 width=0)
3	-> Nested Loop (cost=0.00..266056.74 rows=2 width=8)
4	-> Seq Scan on customer c (cost=0.00..266043.96 rows=1 width=8)
5	Filter: (((c_fname)::text = 'ouE kqP*)/0')::text) AND ((c_lname)::text = 'Cckj0eh[byh']::text))
6	-> Index Scan using i_o_c_id on orders o (cost=0.00..12.76 rows=2 width=16)
7	Index Cond: (o.o_c_id = c.c_id)
8	-> Index Scan using order_line_pkey on order_line ol (cost=0.00..31.57 rows=14 width=8)
9	Index Cond: (ol.ol_o_id = o.o_id)

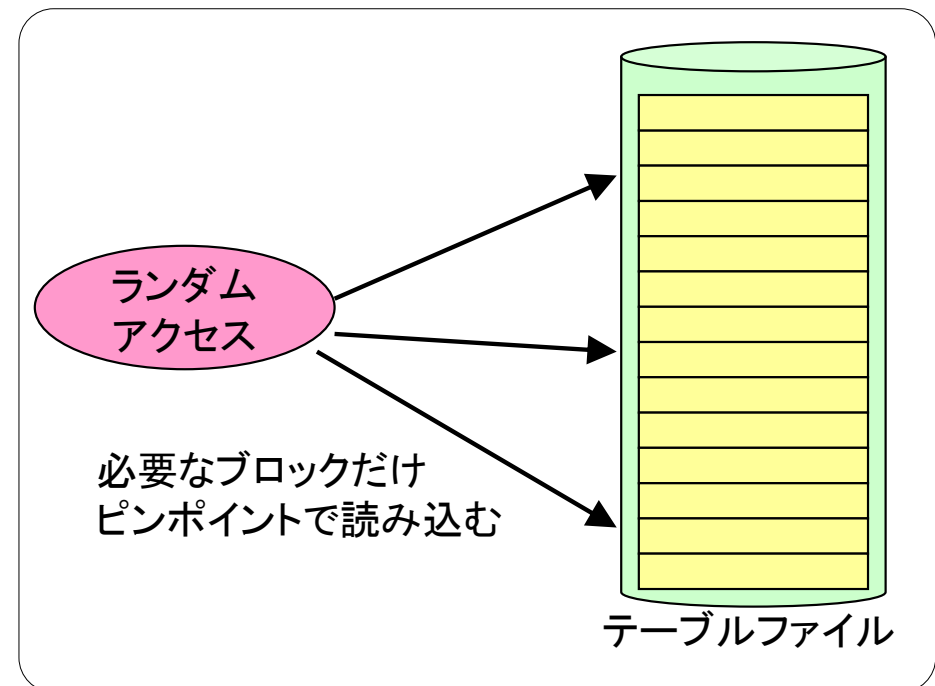
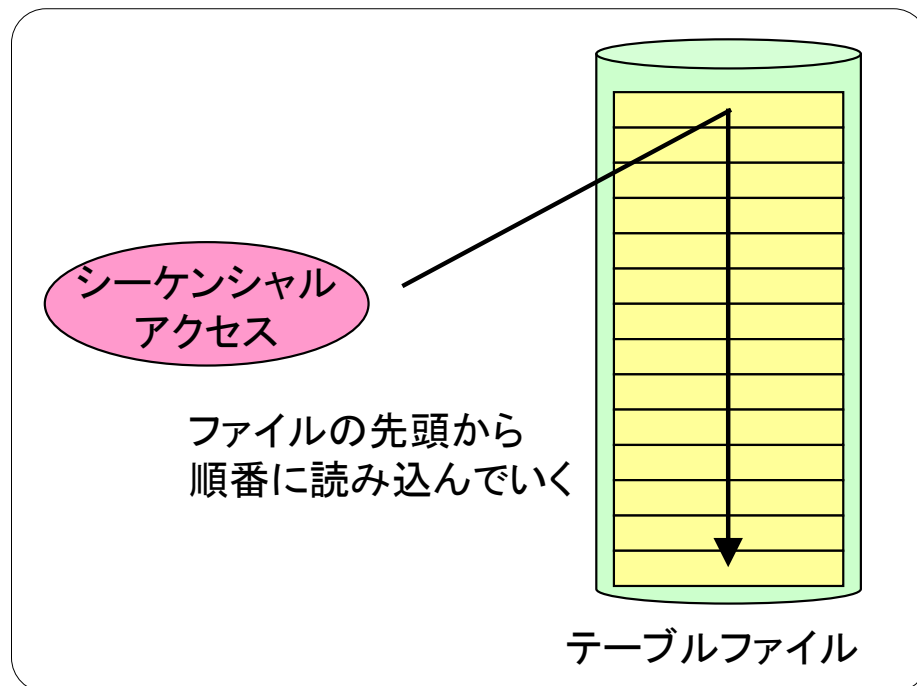
The status bar at the bottom indicates "OK.", "Unix", "行 9 カラム 30 文字 258", "9 行", and "16 ms".

# クエリプランの確認方法

- EXPLAINコマンド
  - 最適であると判断された「クエリプラン」を表示。
  - 入力されたSQL文を、PostgreSQLがどのように解釈して処理しようとしているのかを表示。
- EXPLAIN ANALYZEコマンド
  - 「クエリプラン」に加えて、「実行結果」を表示。
  - 実際に、どのアクセスにどの程度の時間がかかっているのか、何件のレコードを処理したのか、などを表示。
- GUIツールで確認する方法 (pgAdminIII)
  - 「クエリー解釈」=EXPLAIN
  - 「アナライズ解釈」=EXPLAIN ANALYZE

# データアクセスのパターン

- ・ シーケンシャルアクセス
  - 全レコード、または多くのレコードを処理する必要がある場合
  - 集約処理、LIKE文の中間一致など
- ・ ランダムアクセス
  - 特定のレコード(を含むブロック)だけにアクセスする必要がある場合
  - 主にインデックスを用いたアクセス



# テーブルスキャン

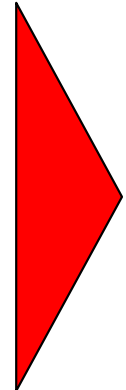
SELECT count(\*) FROM customer;

```
snaga@devsv02:~/pgsql/postgres...
DBT1=# SELECT * FROM pg_statio_user_tables WHERE relname='customer'; SELECT * FROM pg_statio_user_indexes WHERE relname='customer';
-[ RECORD 1 ]-----
relid          | 18062
schemaname     | public
relname        | customer
heap_blks_read | 0
heap_blks_hit  | 0
idx_blks_read  | 0
idx_blks_hit   | 0
toast_blks_read| 0
toast_blks_hit | 0
tidx_blks_read | 0
tidx_blks_hit  | 0

-[ RECORD 1 ]-----
relid          | 18062
indexrelid     | 18065
schemaname     | public
relname        | customer
indexrelname   | customer_pkey
idx_blks_read  | 0
idx_blks_hit   | 0

-[ RECORD 2 ]-----
relid          | 18062
indexrelid     | 18119
schemaname     | public
relname        | customer
indexrelname   | i_c_uname
idx_blks_read  | 0
idx_blks_hit   | 0

DBT1=#
```



```
snaga@devsv02:~/pgsql/postgres...
DBT1=# SELECT * FROM pg_statio_user_tables WHERE relname='customer'; SELECT * FROM pg_statio_user_indexes WHERE relname='customer';
-[ RECORD 1 ]-----
relid          | 18062
schemaname     | public
relname        | customer
heap_blks_read | 0
heap_blks_hit  | 214216
idx_blks_read  | 0
idx_blks_hit   | 0
toast_blks_read| 0
toast_blks_hit | 0
tidx_blks_read | 0
tidx_blks_hit  | 0

-[ RECORD 1 ]-----
relid          | 18062
indexrelid     | 18065
schemaname     | public
relname        | customer
indexrelname   | customer_pkey
idx_blks_read  | 0
idx_blks_hit   | 0

-[ RECORD 2 ]-----
relid          | 18062
indexrelid     | 18119
schemaname     | public
relname        | customer
indexrelname   | i_c_uname
idx_blks_read  | 0
idx_blks_hit   | 0

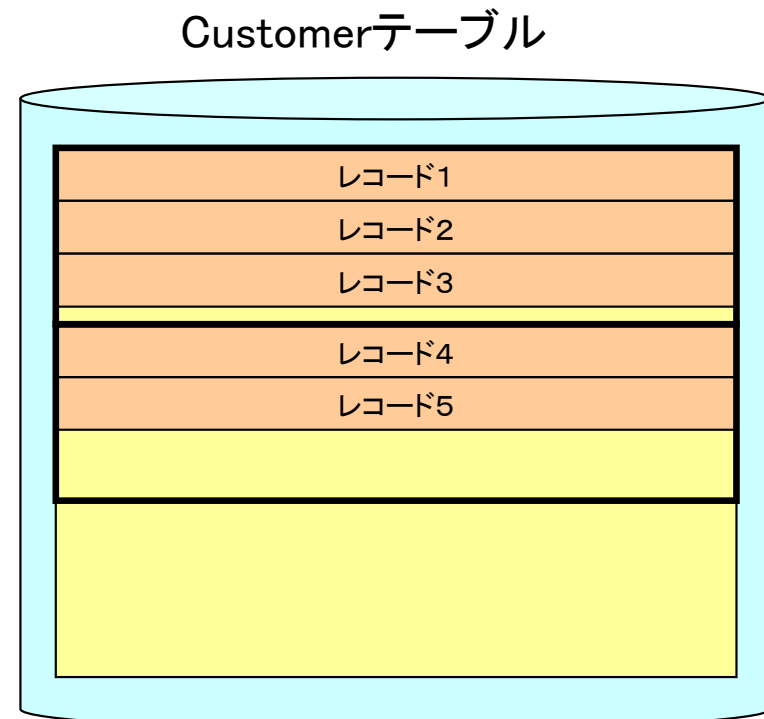
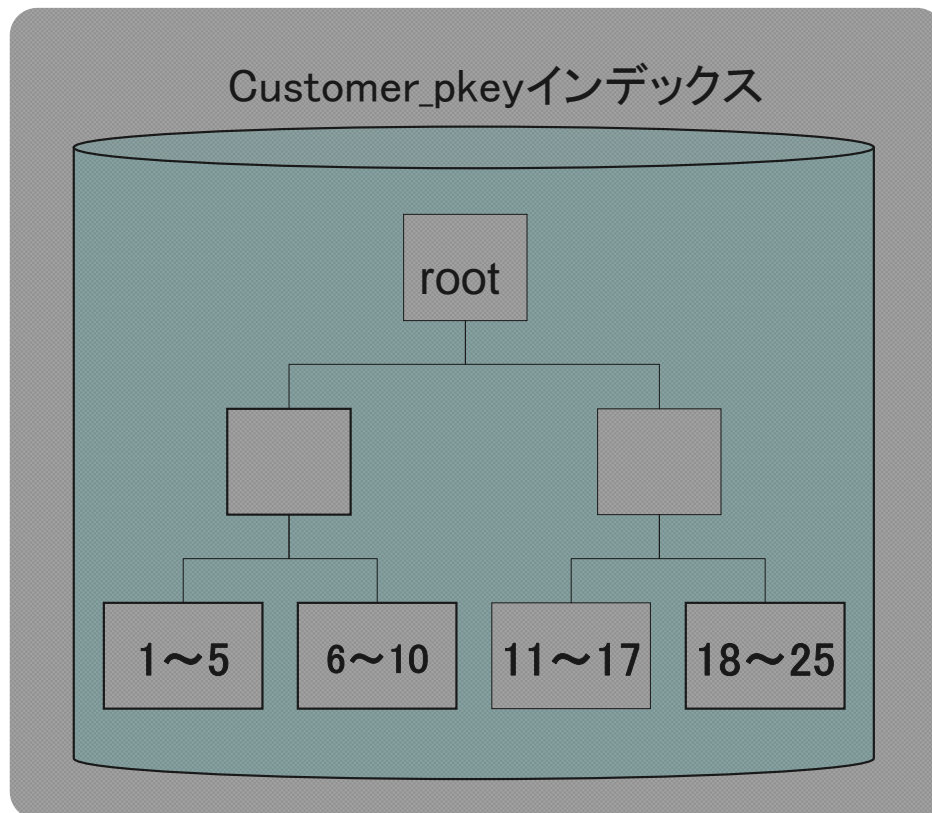
DBT1=#
```

Customer  
テーブルからの  
ブロック読込  
× 214,216

Customer\_pkey  
インデックスの  
ブロック読込 × 0

## テーブルスキャン cont'd

- すべてのデータを確認する必要があるため、customerテーブルファイルを構成するブロックを先頭から読み込む
  - よって、データが増えれば増えるほど時間がかかるようになる。
  - この例では、214,216 ブロック(約1.7GB)を読んでいる。



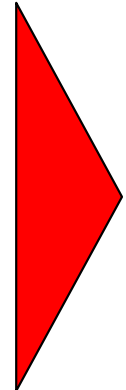
# インデックスアクセス

SELECT \* FROM customer c WHERE c.c\_id=7;

```
snaga@devsv02:~/pgsql/postgres...
DBT1=# SELECT * FROM pg_statio_user_tables WHERE relname='customer'; SELECT * FROM pg_statio_user_indexes WHERE relname='customer';
-[ RECORD 1 ]-----
relid          | 18062
schemaname     | public
relname        | customer
heap_blks_read | 0
heap_blks_hit  | 0
idx_blks_read  | 0
idx_blks_hit   | 0
toast_blks_read |
toast_blks_hit |
tidx_blks_read |
tidx_blks_hit  |

-[ RECORD 1 ]-----
relid          | 18062
indexrelid     | 18065
schemaname     | public
relname        | customer
indexrelname   | customer_pkey
idx_blks_read  | 0
idx_blks_hit   | 0

-[ RECORD 2 ]-----
relid          | 18062
indexrelid     | 18119
schemaname     | public
relname        | customer
indexrelname   | i_c_uname
idx_blks_read  | 0
idx_blks_hit   | 0
DBT1=#
```



```
snaga@devsv02:~/pgsql/postgres...
DBT1=# SELECT * FROM pg_statio_user_tables WHERE relname='customer'; SELECT * FROM pg_statio_user_indexes WHERE relname='customer';
-[ RECORD 1 ]-----
relid          | 18062
schemaname     | public
relname        | customer
heap_blks_read | 0
heap_blks_hit  | 1
idx_blks_read  | 0
idx_blks_hit   | 3
toast_blks_read |
toast_blks_hit |
tidx_blks_read |
tidx_blks_hit  |

-[ RECORD 1 ]-----
relid          | 18062
indexrelid     | 18065
schemaname     | public
relname        | customer
indexrelname   | customer_pkey
idx_blks_read  | 0
idx_blks_hit   | 3

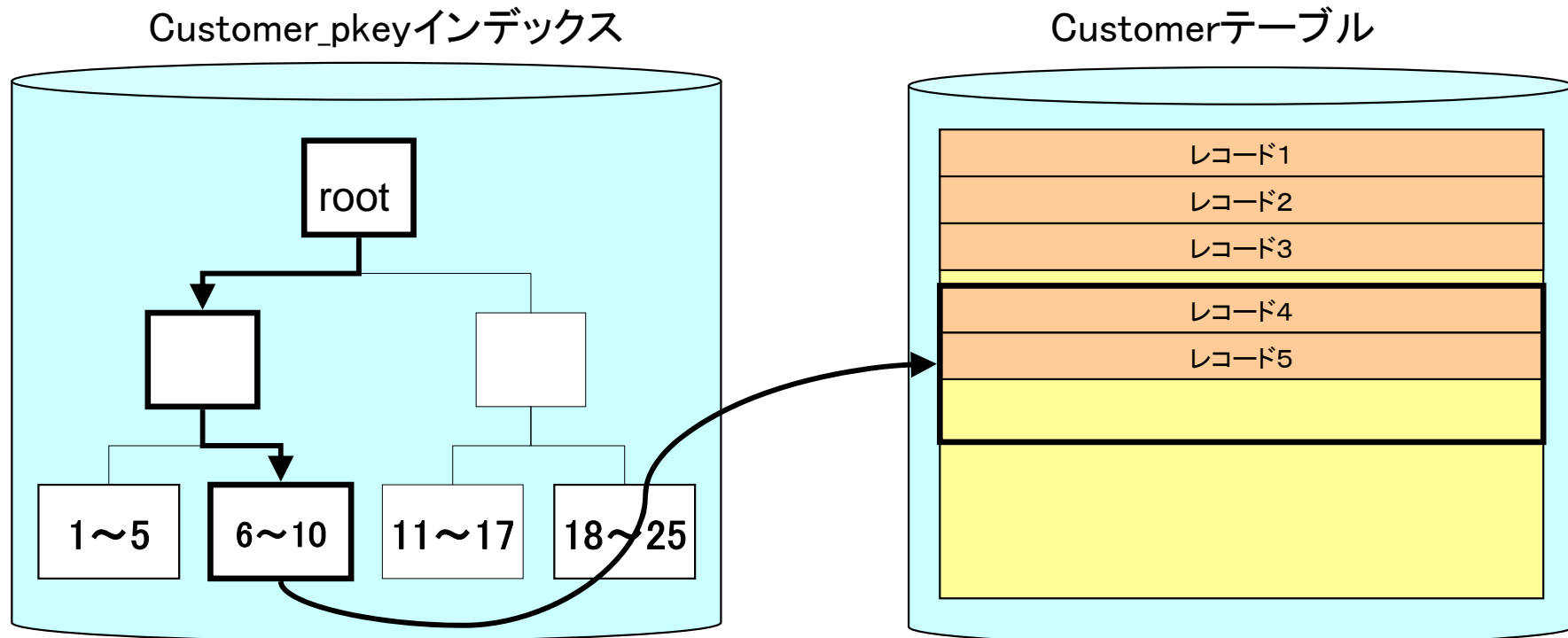
-[ RECORD 2 ]-----
relid          | 18062
indexrelid     | 18119
schemaname     | public
relname        | customer
indexrelname   | i_c_uname
idx_blks_read  | 0
idx_blks_hit   | 0
DBT1=#
```

Customer  
テーブルからの  
ブロック読込 × 1

Customer\_pkey  
インデックスの  
ブロック読込 × 3

## インデックスアクセス cont'd

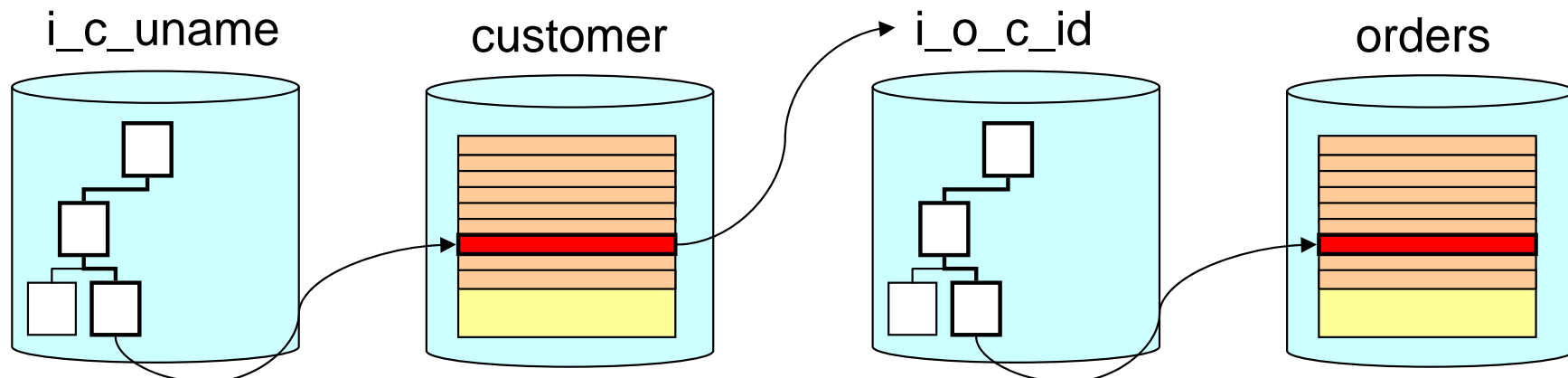
- “c\_id=7” レコードの位置を探すため、customer\_pkeyを辿ってポインタを見つけ、レコードを含むテーブルファイルのブロックを読み込む。
  - この例では、customer\_pkeyインデックスから3ブロック、customerテーブルから1ブロックを読んでいる。
  - レコードの量とディスクアクセス量が比例しない。



## 結合 (Nested Loop Join)

- `SELECT count(*) FROM orders o, customer c WHERE o.o_c_id=c.c_id AND c.c_uname='UL';`
  - customer を `c_uname='UL'` でインデックススキャン
  - customer のレコードの `c_id` を使って orders をインデックススキャン

```
snaga@devsv02:~/pgsql/postgresql-9.0.1/contrib/pgstattuple
DBT1=# explain SELECT count(*) FROM orders o, customer c WHERE o.o_c_id=c.c_id AND c.c_uname='UL';
          QUERY PLAN
-----
Aggregate  (cost=21.60..21.61 rows=1 width=0)
  -> Nested Loop  (cost=0.00..21.59 rows=2 width=0)
    -> Index Scan using i_c_uname on customer c  (cost=0.00..8.80 rows=1 width=8)
        Index Cond: ((c_uname)::text = 'UL'::text)
    -> Index Scan using i_o_c_id on orders o  (cost=0.00..12.76 rows=2 width=8)
        Index Cond: (o.o_c_id = c.c_id)
(6 rows)
DBT1=#
```



## (3) I/O処理詳細

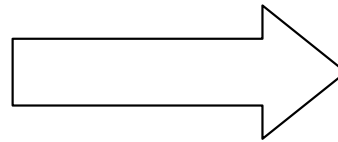
# テーブルに対する更新処理

レコード  
追加処理  
(INSERT)

レコード1
レコード2
レコード3
レコード4

ファイル中に4件のレコードが  
順番に並んでいる

「レコード5」を追加



レコード1
レコード2
レコード3
レコード4
レコード5

レコード5がファイル末尾に追加され、  
ファイルサイズが増える

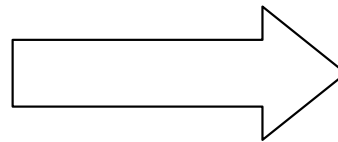


レコード  
削除処理  
(DELETE)

レコード1
レコード2
レコード3
レコード4

ファイル中に4件のレコードが  
順番に並んでいる

「レコード2」を削除



レコード1
(レコード2)
レコード3
レコード4

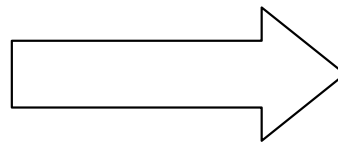
レコード2に削除マークが付けられる

レコード  
更新処理  
(UPDATE)

レコード1
レコード2
レコード3
レコード4

ファイル中に4件のレコードが  
順番に並んでいる

「レコード2」を  
「レコード2'」として更新



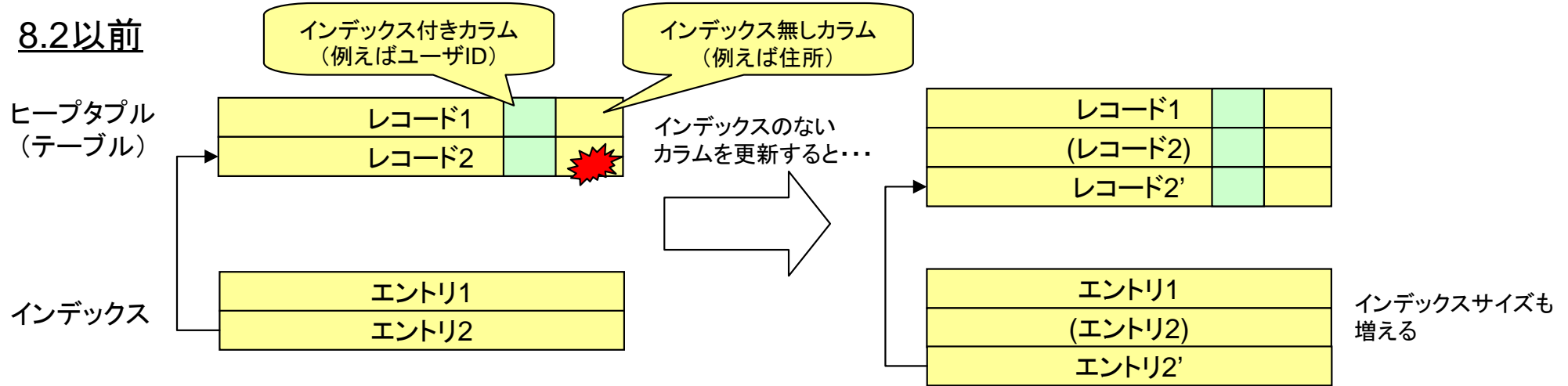
レコード1
(レコード2)
レコード3
レコード4
レコード2'

レコード2に削除マークが付けられ、  
レコード2'が新たに追加、ファイルサイズ増加

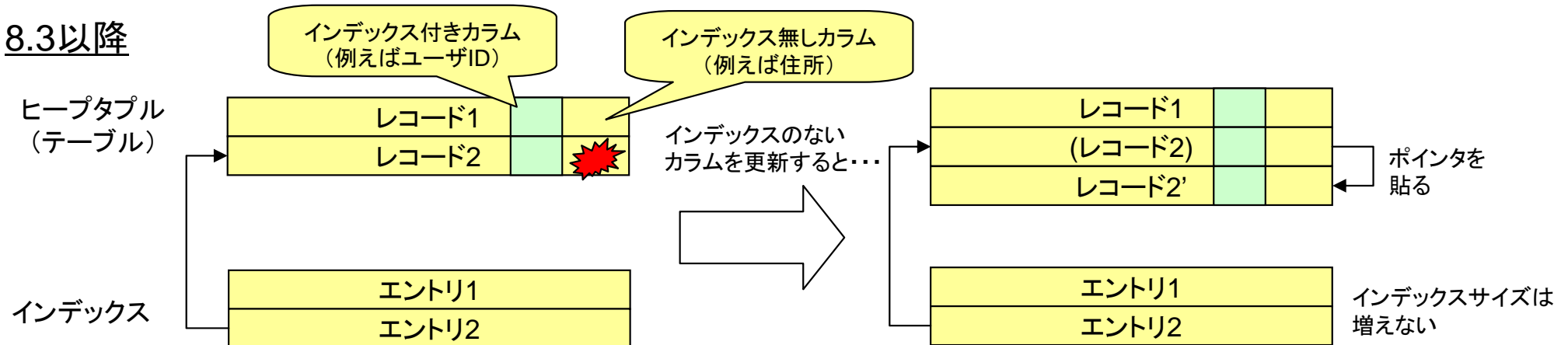


# タプルの更新とインデックスの更新

## 8.2以前



## 8.3以降



インデックスの張られていないカラムを更新すると、  
ヒープのみの(インデックスエントリが無い)カラムができ、  
インデックスの増加が抑制される。

これが、HOT (Heap Only Tuple)

# テーブルに対する参照処理(MVCC)

- 作成・削除したトランザクションID(XID)を参照しながら、「読み飛ばすレコード」を決める。

作成 XID	作成 CID	削除 XID	削除 CID	レコードデータ
-----------	-----------	-----------	-----------	---------

作成XID ... レコードを作成したトランザクションのID

作成CID ... レコードを作成したコマンドID(トランザクションの内部での番号)

削除XID ... レコードを削除したトランザクションID

削除CID ... レコードを削除したコマンドID

テーブル例

101	2	-	-	レコードデータ1
101	3	102	6	レコードデータ2
102	7	-	-	レコードデータ3
102	9	-	-	レコードデータ4



トランザクション101 ... トランザクション開始。レコード1とレコード2を作成。

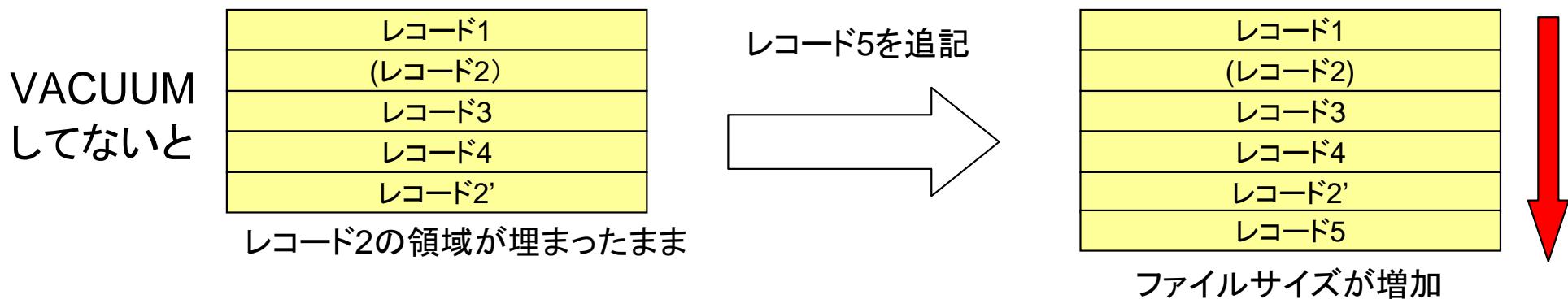
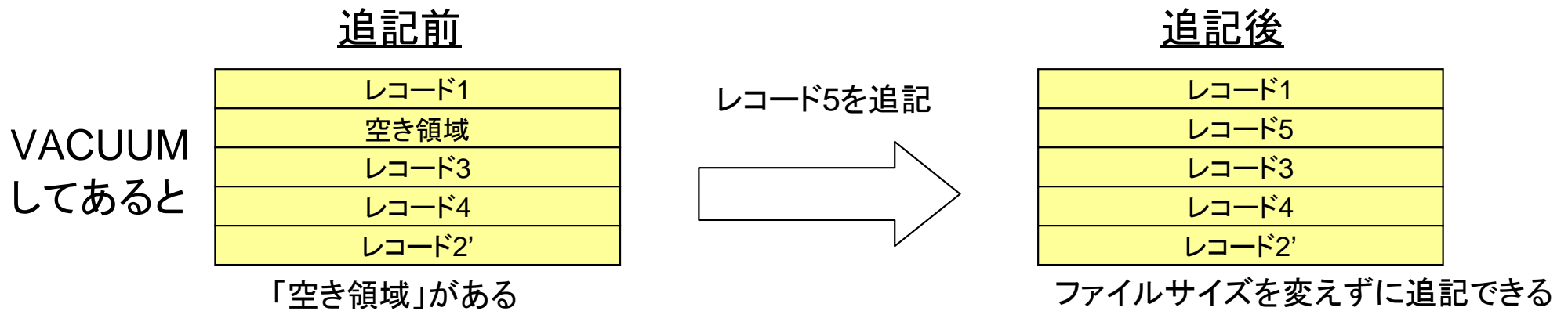
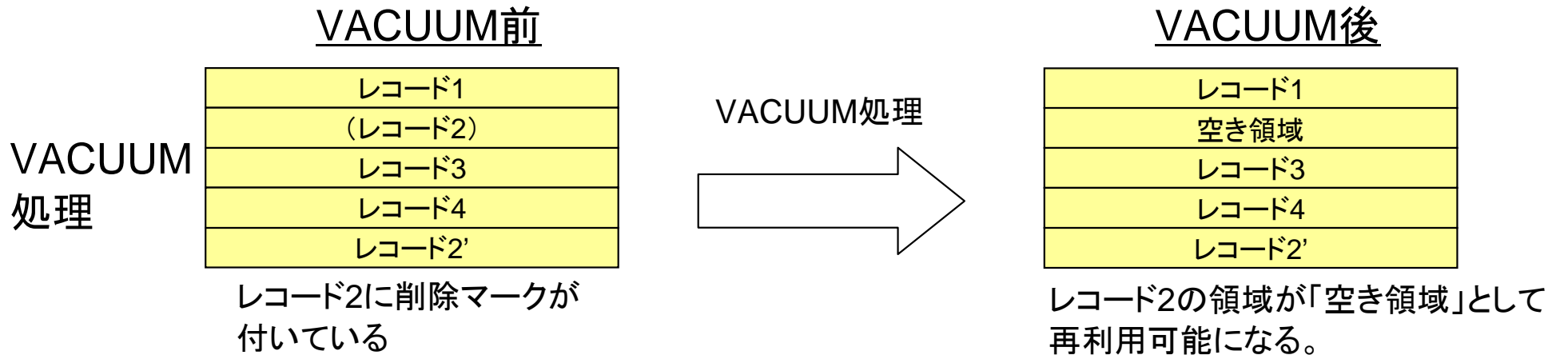
トランザクション102 ... トランザクション開始。

レコード2を削除して、レコード3、レコード4を作成。

トランザクション103 ... トランザクション開始。

レコード3、レコード4は参照可、レコード2は参照不可。

# VACUUM処理



# (参考)pgestimate

- テーブル／インデックスファイルサイズ見積もりツール

pgestimate - PostgreSQL Database Size Estimator - Mozilla Firefox

http://www.uptime.jp/tools/pgestimate/

## pgestimate - PostgreSQL Database Size Estimator

Tweet 2 Like Satoshi Nagayasu and 6 others like this.

### How to use pgestimate

1. Input a column name you need. (up to 10 columns)
2. Select a column type from the drop-down list.
3. Input a column size if it's variable.
4. Check 'index 1' ~ 'index 3' if you want to add the column to index. (up to 3 indexes)
5. Input a number of rows of the table.
6. Press the 'Estimate' button.

### Table and index definitions

column #	name	type	size	index 1	index 2	index 3
0	uid	integer		<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
1	fullname	varchar	64	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
2	email	varchar	64	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
3	address	varchar	256	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
4		-		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
5		-		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
6		-		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
7		-		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
8		-		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
9		-		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

rows: 10000

Estimate Reset

### Estimated object size

- Table table1: 4317184
- Index table1\_uid\_ind: 237568
- Database db1: 4554752

<http://www.uptime.jp/go/pgestimate>

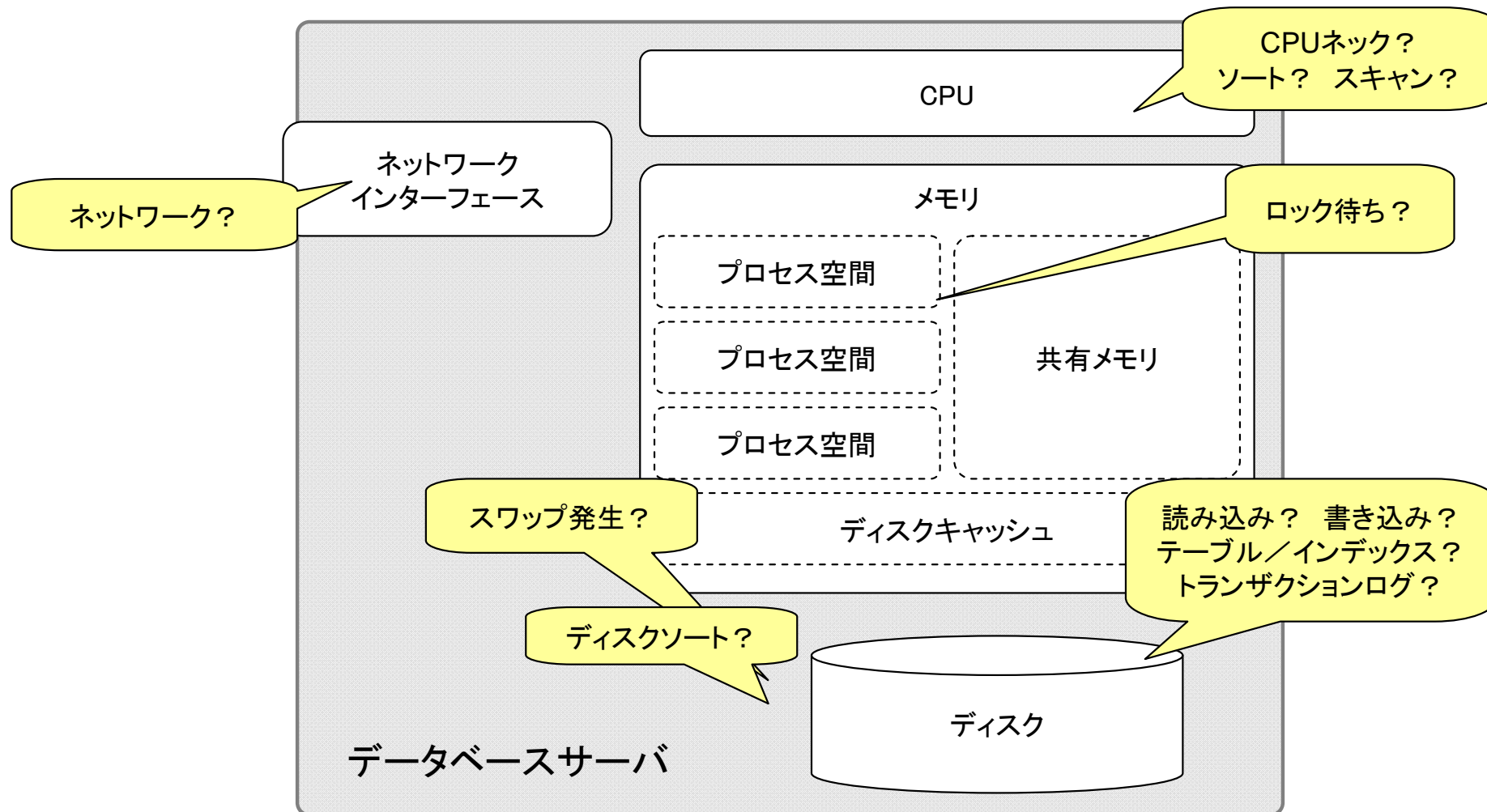
## (4) パフォーマンス管理

# パフォーマンスは何で決まるか？

- 「単一クエリのレスポンス × クエリの同時実行数」
  - 単一クエリのレスポンス
    - サーバ・クライアント間通信(ネットワーク)
    - SQLの構文解析、最適化(CPU処理)
    - ロックの競合(ロック待ち、デッドロックの発生)
    - テーブル、インデックス、ログへのI/O量(ディスクI/O)
    - ソート、結合などの演算処理(CPU処理、ディスクI/O)
  - クエリの同時実行数
    - 接続クライアント数(いわゆるWebユーザ数)
    - コネクションプール接続数
- 全体としてハードウェアのキャパシティの範囲内であるか？
  - ネットワーク、ディスクI/O、メモリ、CPUなどがボトルネックとなり得る。
  - ただし、ボトルネック自体は「結果」であり、「原因」ではない。
  - 「なぜ、それがボトルネックになっているのか？」が重要。
    - テーブル設計？ SQL文？ 同時接続数？ HW？ 設定パラメータ？...

# データベースを構成するハードウェアリソース

- 複雑な構造を持つRDBMSでは、ボトルネックはいたるところに発生し得るため、まずはきちんと切り分けることが重要。
  - いきなりパラメータチューニングとかを始めない。

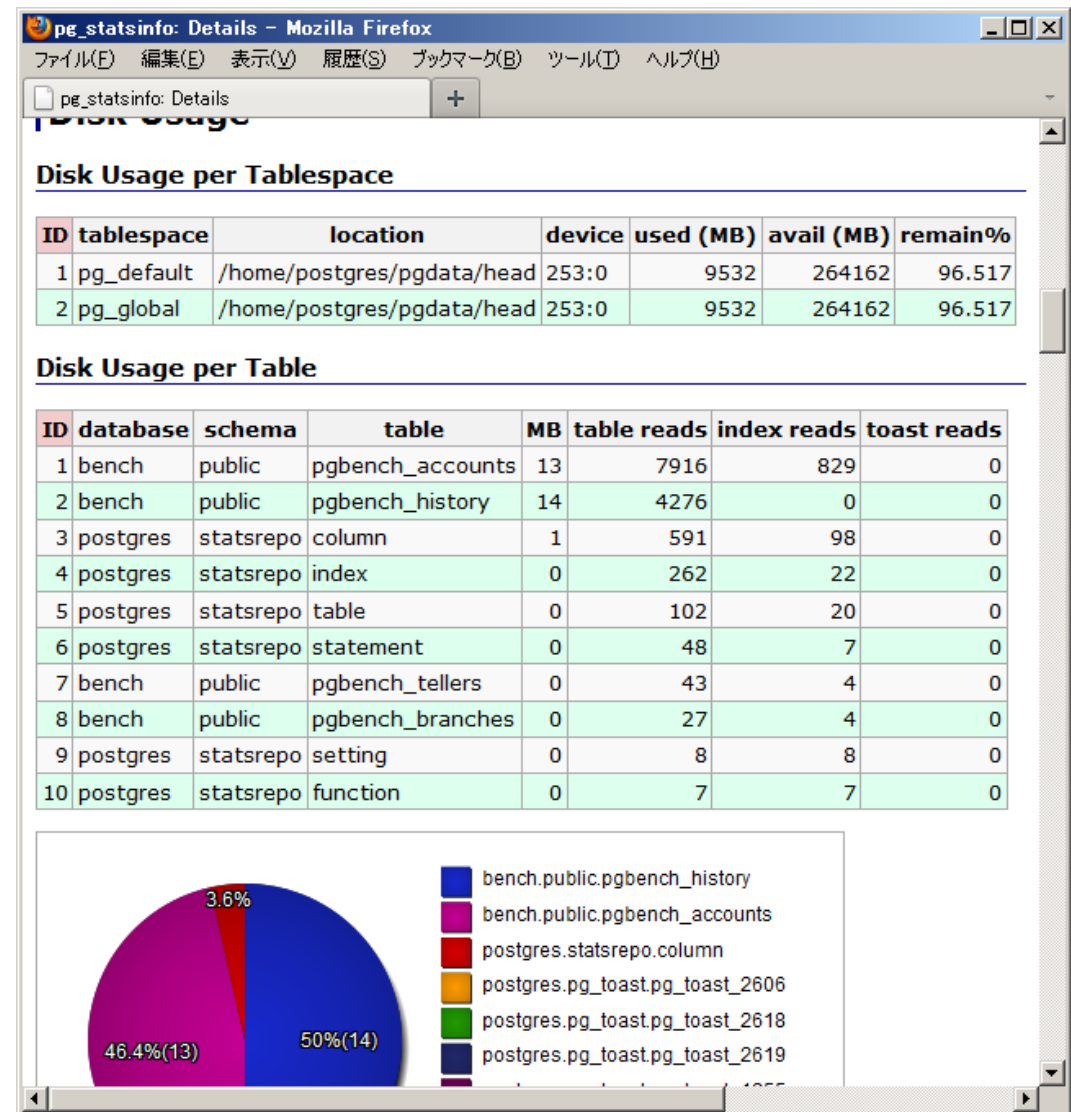


# パフォーマンス改善の基本手順

- 全体のパフォーマンスの傾向をつかむ
  - どのデータベース、テーブルへのアクセスか？ HWの利用状況はどうか？
- 遅いSQL文を特定する or 実行回数の多いSQLを特定する
  - log\_min\_durationオプション
  - pgFouine
- 特定のSQLだけが遅い場合・・・
  - SQLのクエリプランおよび実行状況を確認する(EXPLAIN)
- 遅いSQLが特定されない(偏りが無い)場合・・・
  - ハードウェアリソースのボトルネックを探す
- 対策を実施する
  - SQL文を書き換える、インデックスを張る、テーブル設計を修正する
  - アプリケーションを修正する
  - ハードウェアを増強する
  - 他・・・

# 全体の傾向を可視化する

- pg\_statinfo/pg\_reporterを使って、アクセス統計情報を可視化する。
  - データベース統計情報
  - ディスク使用状況
  - テーブル統計情報
  - チェックポイント情報
  - Autovacuum実行状況
  - SQL文実行状況
  - 等...



# SQLパフォーマンス分析

- pgFouineによる問題SQL文の抽出、ランキング作成
  - 総実行時間 = レスポンスタイム (実行時間) × 実行回数
  - 最長レスポンスタイム
  - 他...

pgFouine: PostgreSQL log analysis report

Overall statistics | Queries by type | Queries that took up the most time (N) | Slowest queries | Most frequent queries (N) | Slowest queries (N)

Normalized reports are marked with a "(N)".

- Generated on 2011-01-13 16:19
- Parsed /home/snaga/pgsql/pgfouine-1.2/dbt1-001.log (7,000 lines) in 7s
- Log from 2011-01-13 16:14:42 to 2011-01-13 16:16:39
- Executed on devwa02.uptime.jp

Overall statistics ^

- Number of unique normalized queries: 14
- Number of queries: 3,506 (identified: 3,493)
- Total query duration: 25m27s (identified: 24m49s)
- First query: 2011-01-13 16:14:42
- Last query: 2011-01-13 16:16:39
- Query peak: 49 queries/s at 2011-01-13 16:16:01

Queries by type ^

Type	Count	Percentage
SELECT	3,493	100.0

Queries that took up the most time (N) ^

Rank	Total duration	Times executed	Av. duration (s)	Query
1	14m	239	3.52	<pre>SELECT * FROM search_results_author( ", 0 ) AS l(i_related1 numeric(0),i_related2 numeric(0),i_related3 numeric(0),i_related4 numeric(0),i_related5 numeric(0),i_thumbnail1 numeric(0),i_thumbnail2 numeric(0),i_thumbnail3 numeric(0),i_thumbnail4 numeric(0),i_thumbnail5 numeric(0),items numeric(0),i_id1 numeric(0),i_title1 VAR VARCHAR(0),a_lname1 VARCHAR(0),i_id2 numeric(0),i_title2 VARCHAR(U),a_rname2</pre>

完了

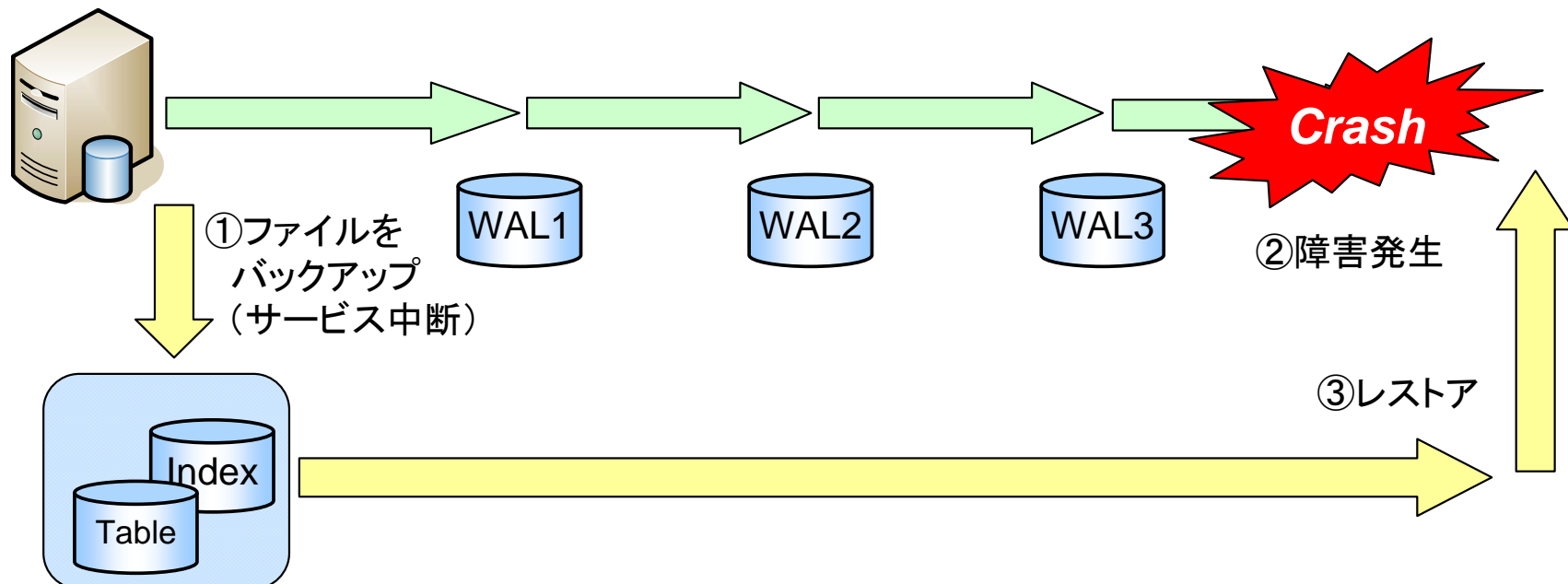
## (5) 可用性管理

# バックアップとレストア／リカバリ

- バックアップの難しさ
  - データはファイルの中にだけあるのではない！
  - 通常は、共有バッファの内容が最新
  - ファイルだけバックアップを取ってもダメ
  - ミリ秒単位で処理が進む中、すべてを一貫性を保った状態で
- バックアップの種類
  - コールドバックアップ
  - ホットバックアップ
  - アーカイブログバックアップ
- バックアップ&レストア／リカバリはリハーサルをしよう！
  - 簡単な試験や手順書を作るだけで満足してはいけない・・・

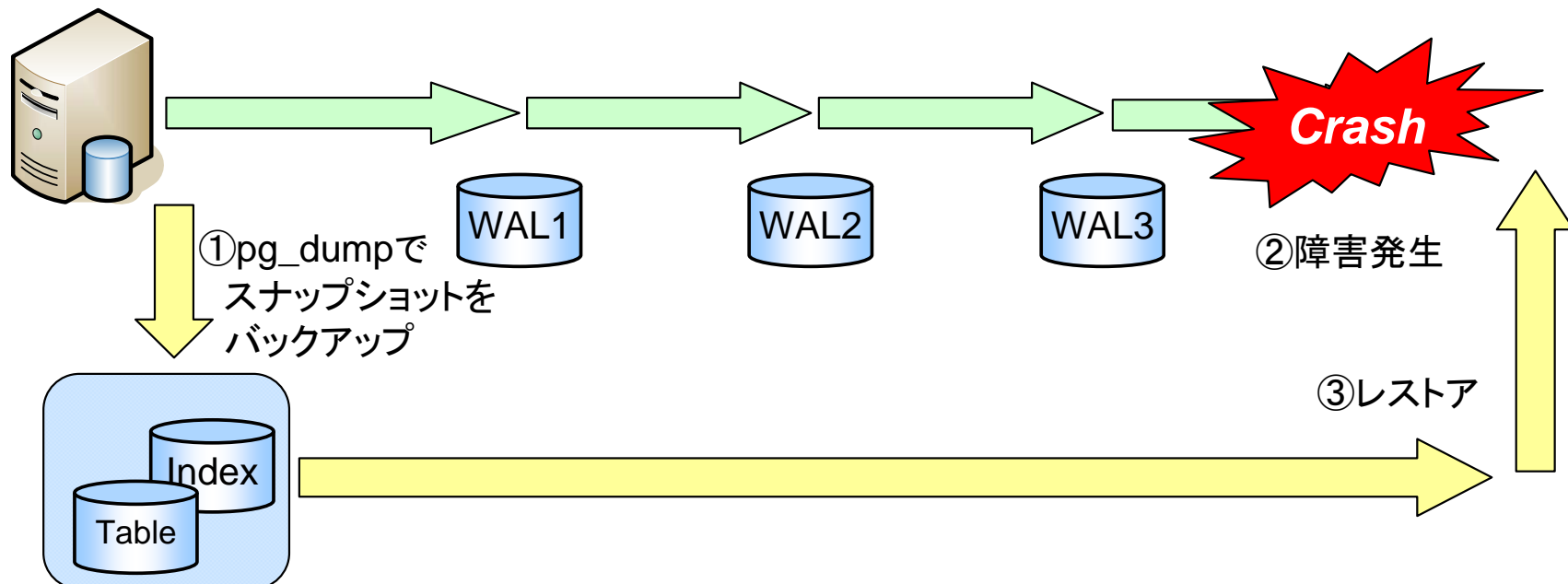
# コールドバックアップ

- サーバプロセスをすべてシャットダウンしてデータファイル全体をバックアップ
  - バックアップの間、サービス停止が発生する。
  - リカバリの際には、バックアップ時のデータに戻る。
- 向いているケース
  - 前回バックアップ以降の更新データを、アプリログなどから復旧できる場合。
  - ストレージスナップショットが一般化した今、案外現実的。
- 向いていないケース
  - サービスを停止させられない場合。
  - 障害発生直前までの更新データが必要で、DB以外から復旧できない場合。



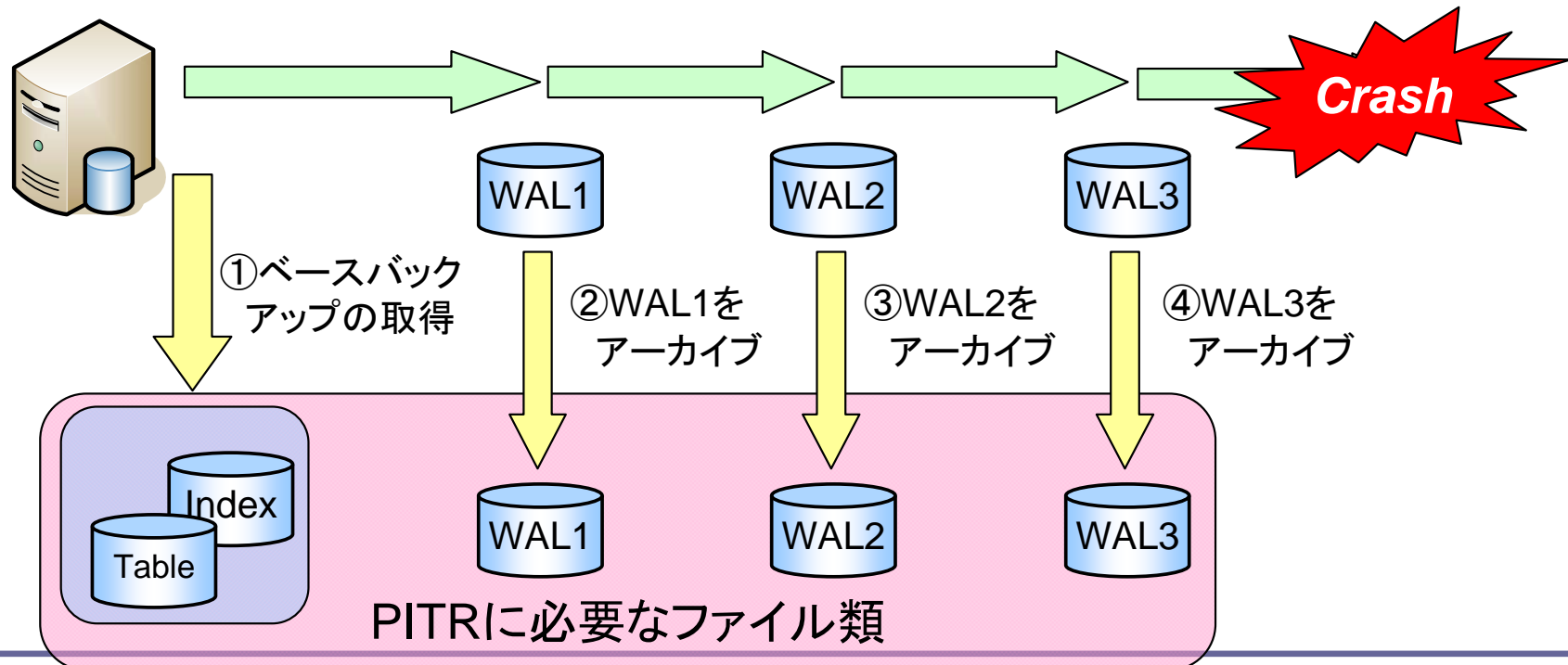
# ホットバックアップ (pg\_dump)

- あるタイミングでデータの一貫性を保ちつつバックアップ
  - シンプルかつ柔軟(テーブル単位のバックアップも可)
  - バックアップ時にサービス停止は起こらない。
  - リカバリの際には、バックアップ時のデータに戻る。
- 向いているケース
  - 前回バックアップ以降の更新データを、アプリログなどから復旧できる場合。
  - データベース単位、テーブル単位でバックアップを取りたい場合。
  - 論理バックアップが必要な場合(メジャーバージョンアップなど)
- 向いていないケース
  - 障害発生直前までの更新データが必要で、DB以外から復旧できない場合。



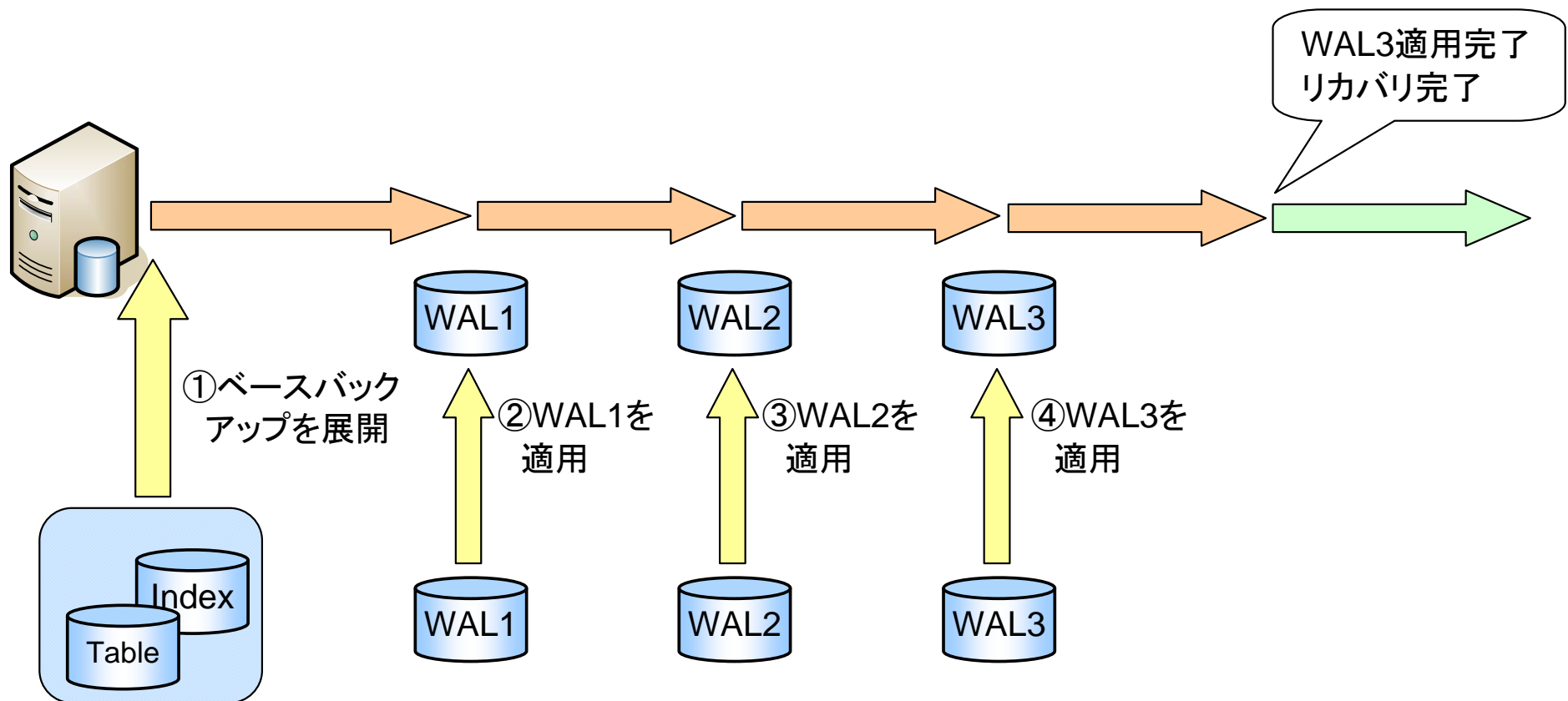
# アーカイブログとPITRを用いたバックアップ

- ベースバックアップ(基準点) + アーカイブログ(更新差分)
  - サービスを継続したままベースバックアップを取得可能
  - クラッシュ直前のWALの内容まで復旧することが可能
- 向いているケース
  - データベースクラスタ全体の完全なバックアップを取りたい場合。
  - クラッシュ直前の更新まで復旧させる必要がある場合。
- 向いていないケース
  - データベース単位、テーブル単位などでバックアップを取得したい場合。



# アーカイブログとPITRを用いたリカバリ

- ベースバックアップ(基準点) + アーカイブログ(更新差分)
  - 前回のベースバックアップ以降、長期間が経過しているとアーカイブログが多くなり、リカバリの時間が長くなる。
  - ベースバックアップストア時間 + アーカイブログ適用時間 × アーカイブログ数



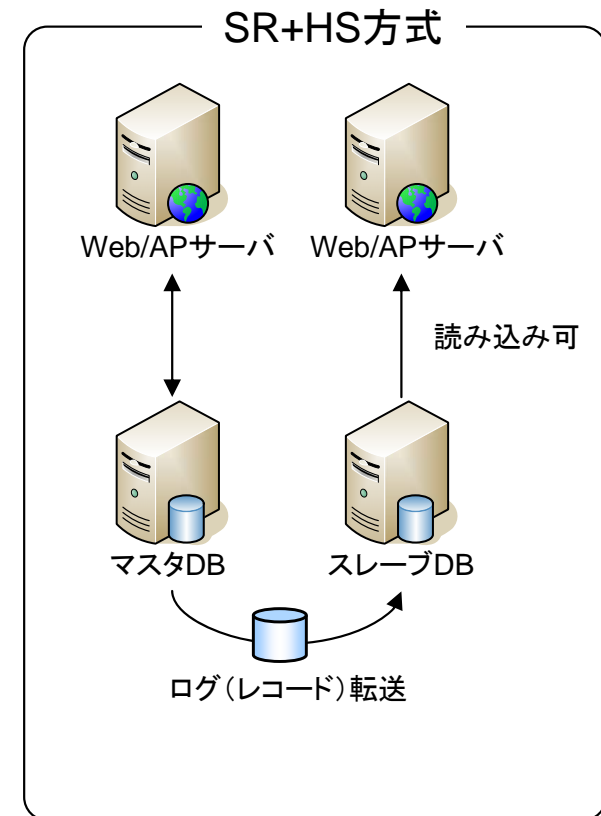
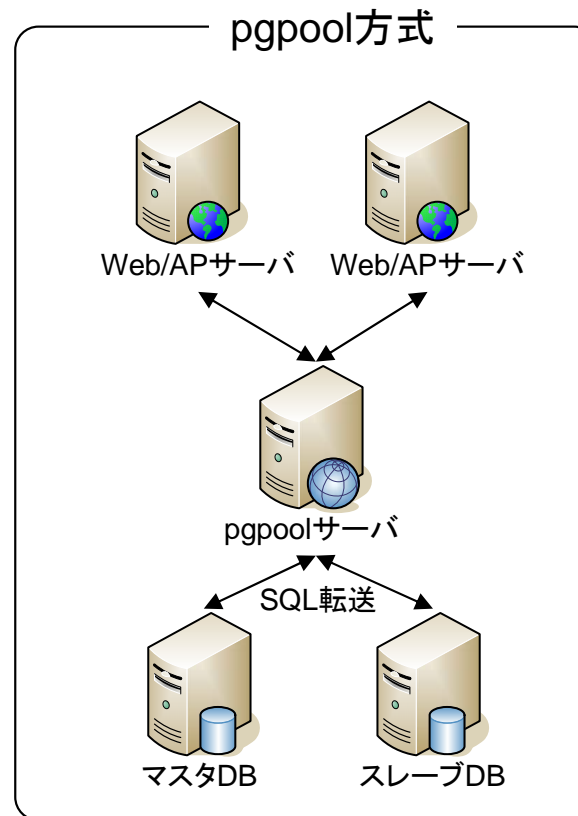
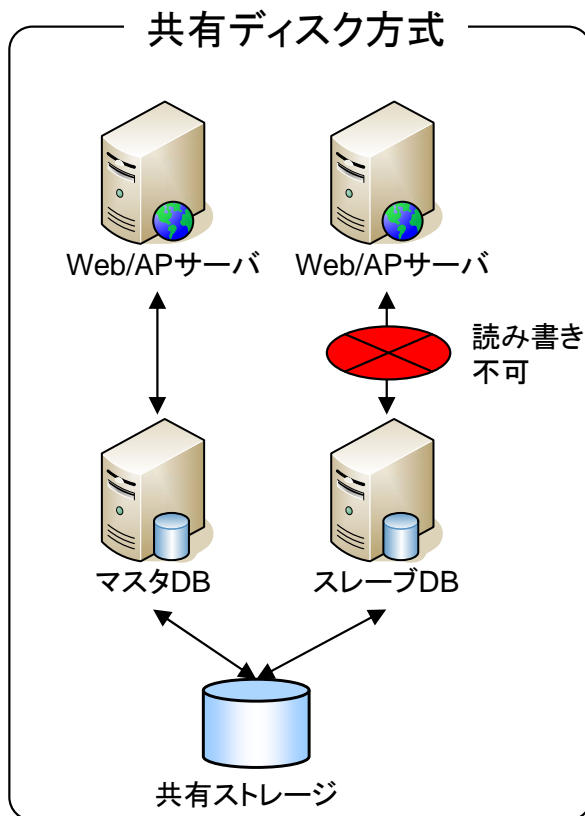
# 冗長化方式の選定

- 実現方式を評価するに当たって特に重視すべき点
  - 負荷分散の必要性の有無。
  - 単一障害点 (Single Point of Failure、SPoF) の有無。
  - 運用が容易であるかどうか (運用の作業負荷、ノウハウの蓄積)。
  - データ一貫性の厳密性 (レプリケーション遅延) の程度。

実現方式	アーキテクチャ	負荷分散	同期遅延	運用性	備考
アーカイブログ転送	アクティブ/スタンバイ	×	数十秒 ～数分	◎	ウォームスタンバイ方式。
DRBDディスク同期	アクティブ/スタンバイ	×	なし	△	要DRBD運用ノウハウ。
共有ディスク方式	アクティブ/スタンバイ	×	なし	△	共有ディスクが高価でSPOF。
Slony-Iレプリケーション	アクティブ/アクティブ、 マスター/スレーブ	○	数秒	△	公開されているSlony-Iの運用ノウハウが少ない。バージョン混在可。
pgpool-II	アクティブ/アクティブ、 マスター/スレーブ	○	なし	○	pgpoolサーバがSPOF (冗長化可)。一部、APへの影響有り(now()等)。
ストリーミング・レプリケーション (9.0～)	アクティブ/アクティブ、 マスター/スレーブ	○	数百ms ～なし	△	公開されている運用ノウハウが少ない。遅延なしは9.1以降。

## 冗長化方式の選定 cont'd

- PostgreSQLの代表的な冗長化方式の構成は以下の通り。
  - シンプルな冗長化のみで良い場合は共有ディスク方式。
  - スケールアウトが必要な場合は pgpool か Slony-I。
  - 9.0以降はストリーミングレプリケーション(SR+HS)構成が可能。



Q&A?

# 参考文献

- 書籍・雑誌
  - WEB+DB PRESS vol.24、25「徒然PostgreSQL散策」(技術評論社)
  - WEB+DB PRESS vol.32～37「PostgreSQL安定運用のコツ」(技術評論社)
  - WEB+DB PRESS vol.63「Web開発の『べし』『べからず』」(技術評論社)
  - データベースパフォーマンスアップの教科書 基本原理編(翔泳社)
- オンラインドキュメント類
  - PostgreSQL 9.0.4文書  
<http://www.postgresql.jp/document/pg904doc/html/index.html>
  - Explaining Explain ～ PostgreSQLの実行計画を読む ～ (PDF版)  
[http://lets.postgresql.jp/documents/technical/query\\_tuning/explaining\\_explain\\_ja.pdf](http://lets.postgresql.jp/documents/technical/query_tuning/explaining_explain_ja.pdf)
  - HOTの仕組み (1) - Let's Postgres  
[http://lets.postgresql.jp/documents/tutorial/hot\\_2/](http://lets.postgresql.jp/documents/tutorial/hot_2/)
  - ソーシャルゲームのためのデータベース設計  
<http://www.slideshare.net/matsunobu/ss-6584540>
  - 高信頼システム構築標準教科書 ー仮想化と高可用性ー  
<http://www.lpi.or.jp/linuxtext/system.shtml>
  - MVCC in PostgreSQL  
[http://chesnok.com/talks/mvcc\\_couchcamp.pdf](http://chesnok.com/talks/mvcc_couchcamp.pdf)