

PostgreSQL安定運用のコツ

～ I/Oを制する者はRDBMSを制す ～

日本PostgreSQLユーザ会

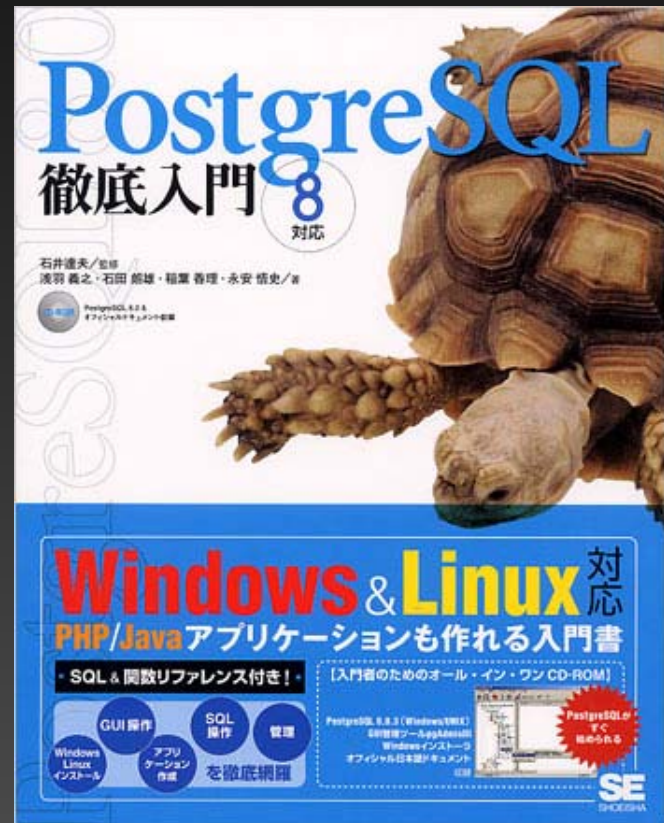
永安 悟史

自己紹介

- 永安 悟史
 - 日本PostgreSQLユーザ会 広報・渉外担当
 - (株)NTTデータ オープンソース開発センタ
- OSS/データベース屋さん(のはず)
 - 検証をしたり、技術支援をしたり、
 - 開発したり、パッチを書いたり、
 - 情報提供をしたり、
 - イベントをしたり、

著書

- PostgreSQL徹底入門 ～ 8対応 (翔泳社)
 - 共著です。。。



WEB+DB PRESS連載

- 連載「PostgreSQL安定運用のコツ」
WEB+DB PRESS vol.32 ~ 37



+



アジェンダ

- パフォーマンスと初期設定
- データベースの監視
- 性能劣化とメンテナンス
- パフォーマンスチューニング (GUC編)
- パフォーマンスチューニング (SQL編)
- バックアップ & リカバリ

(1) パフォーマンスと初期設定

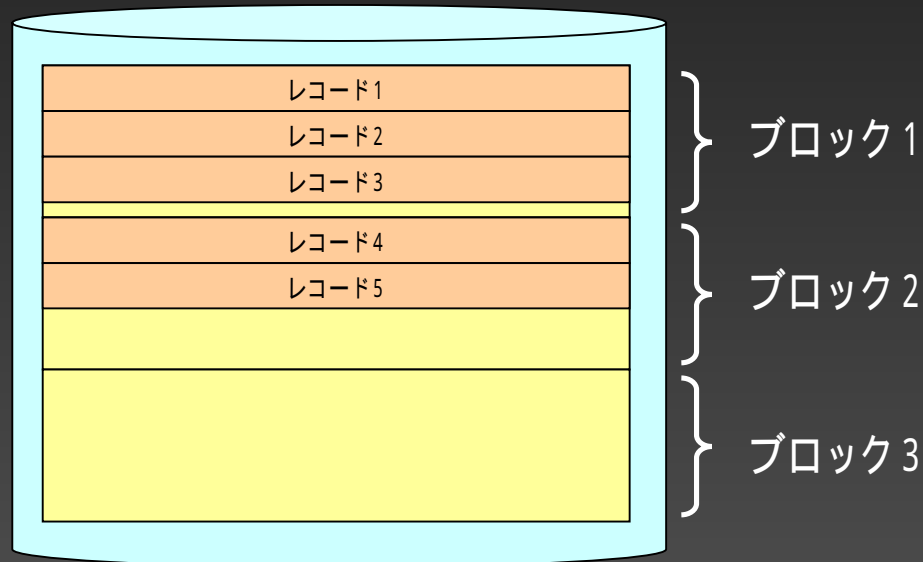


I/Oの種別

- テーブルファイル
 - レコードの実データを保存
 - タプル(行)単位で、テーブルファイルに追記
- インデックスファイル
 - インデックスのキーとレコードへのポインタを保持
 - ツリー構造を持つ(ルート、インターナル、リーフ)
- 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を追記
 - リカバリの際に使用される

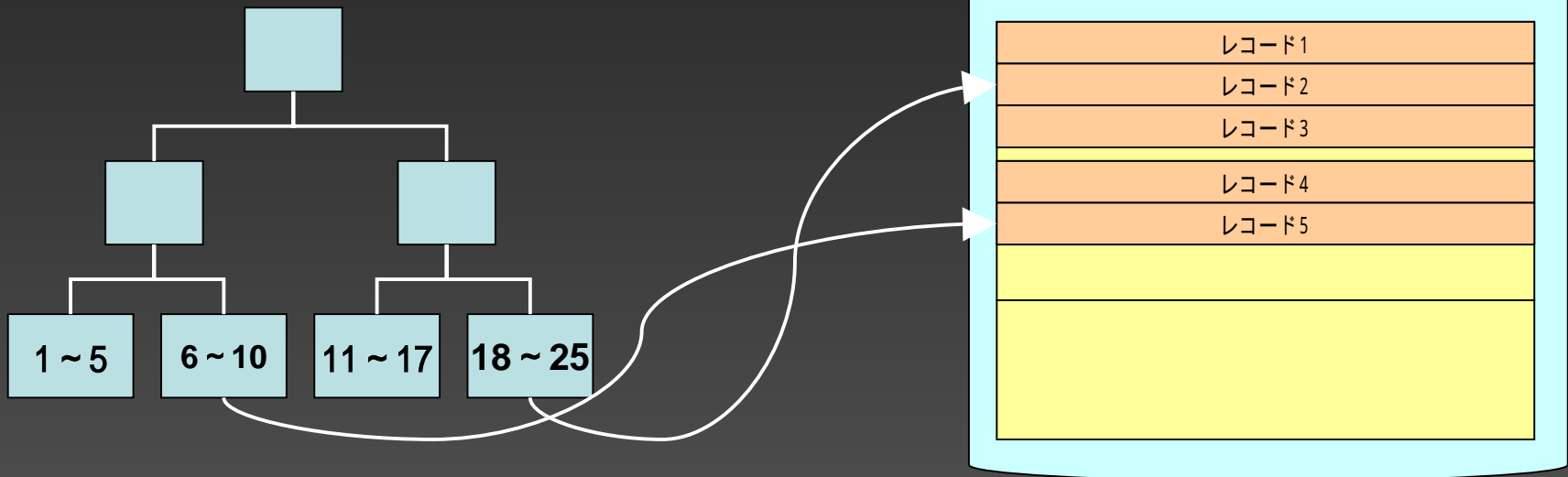
テーブルファイル

- 8kB単位のブロックで構成
 - ブロックの中にタプル(レコード)データを保存
 - 基本的に追記。更新時は、「削除 + 追記」。
 - 削除したら削除マーク。VACUUMで回収。



インデックスファイル

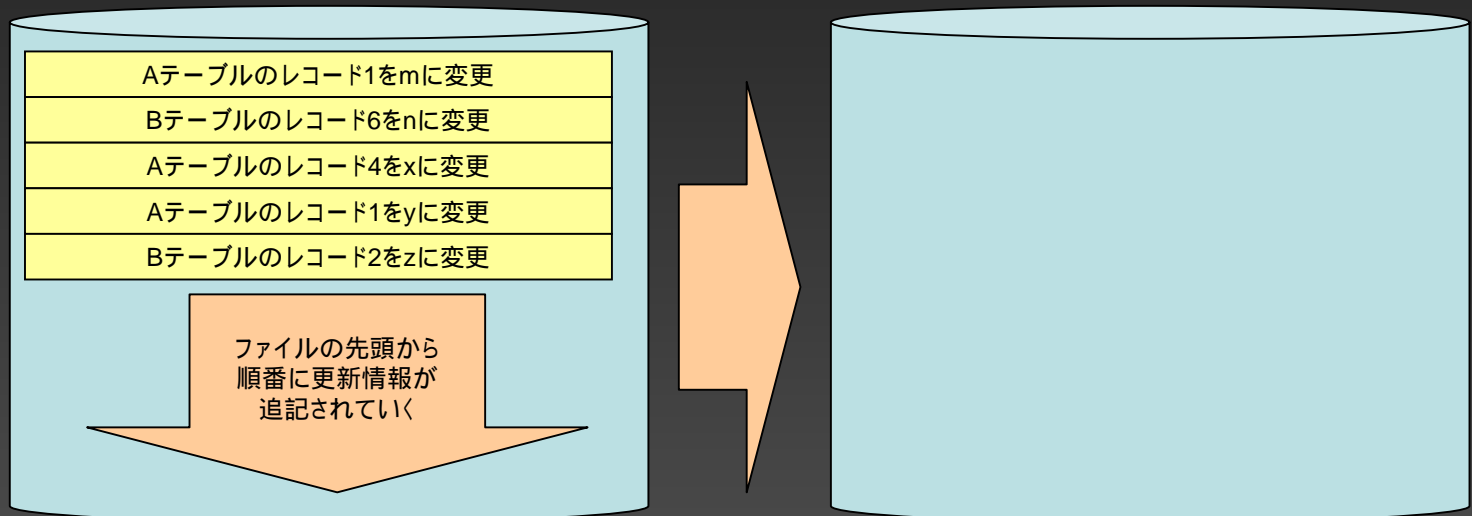
- ツリー構造を持つ
 - 各ノードが、8kBのブロックに相当
 - ルートノードから辿っていく
 - リーフノードは、テーブルファイルの実タプルへのポインタを持つ



トランザクションログ

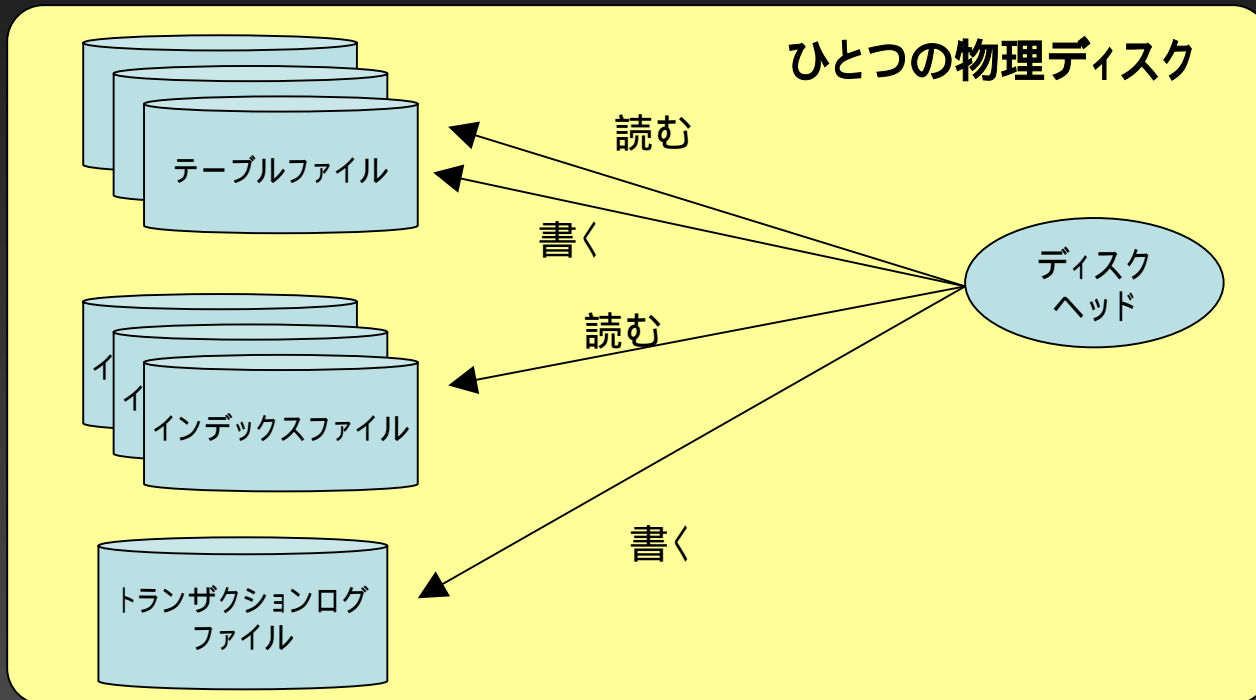
- 更新情報を追記

- リカバリの際に読み込まれる。
- 1セグメント16MB。使い終わると次のファイルへ。
- pg_xlog/ 以下に配置。



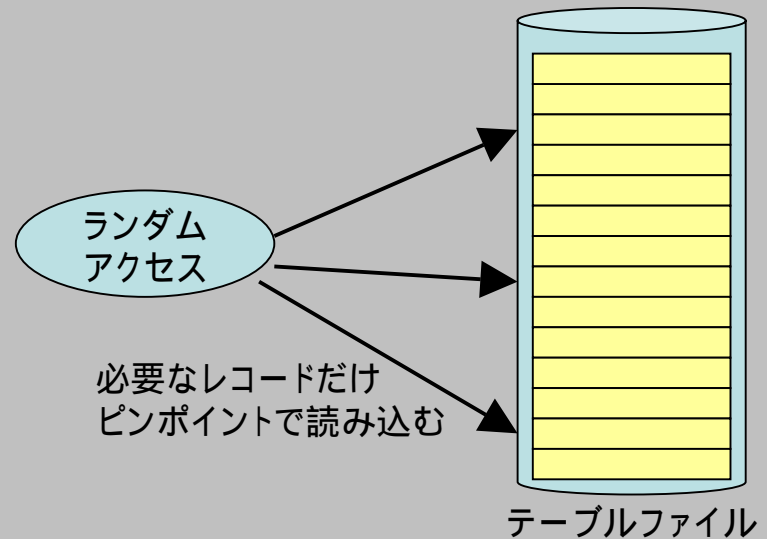
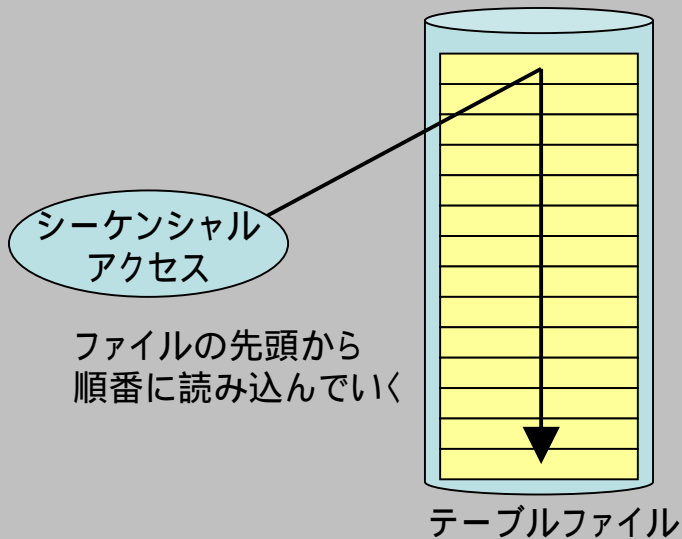
RDBMSで発生するI/O

- RDBMSではさまざまなファイルへさまざまなI/Oが発生する



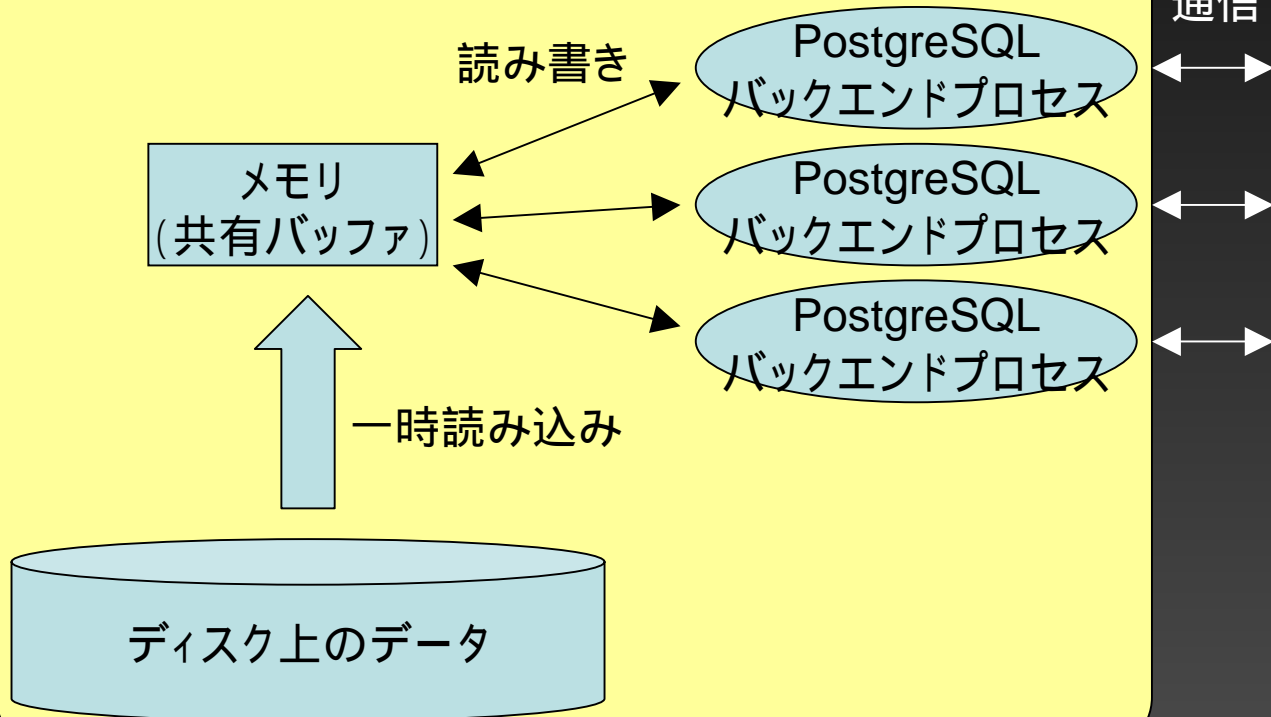
アクセスパターン

- シーケンシャルアクセスとランダムアクセス

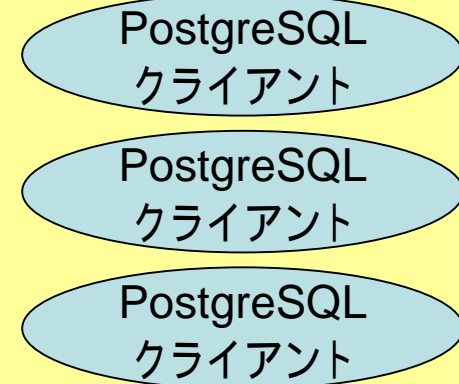


PostgreSQLアーキテクチャ

PostgreSQLバックエンド構造



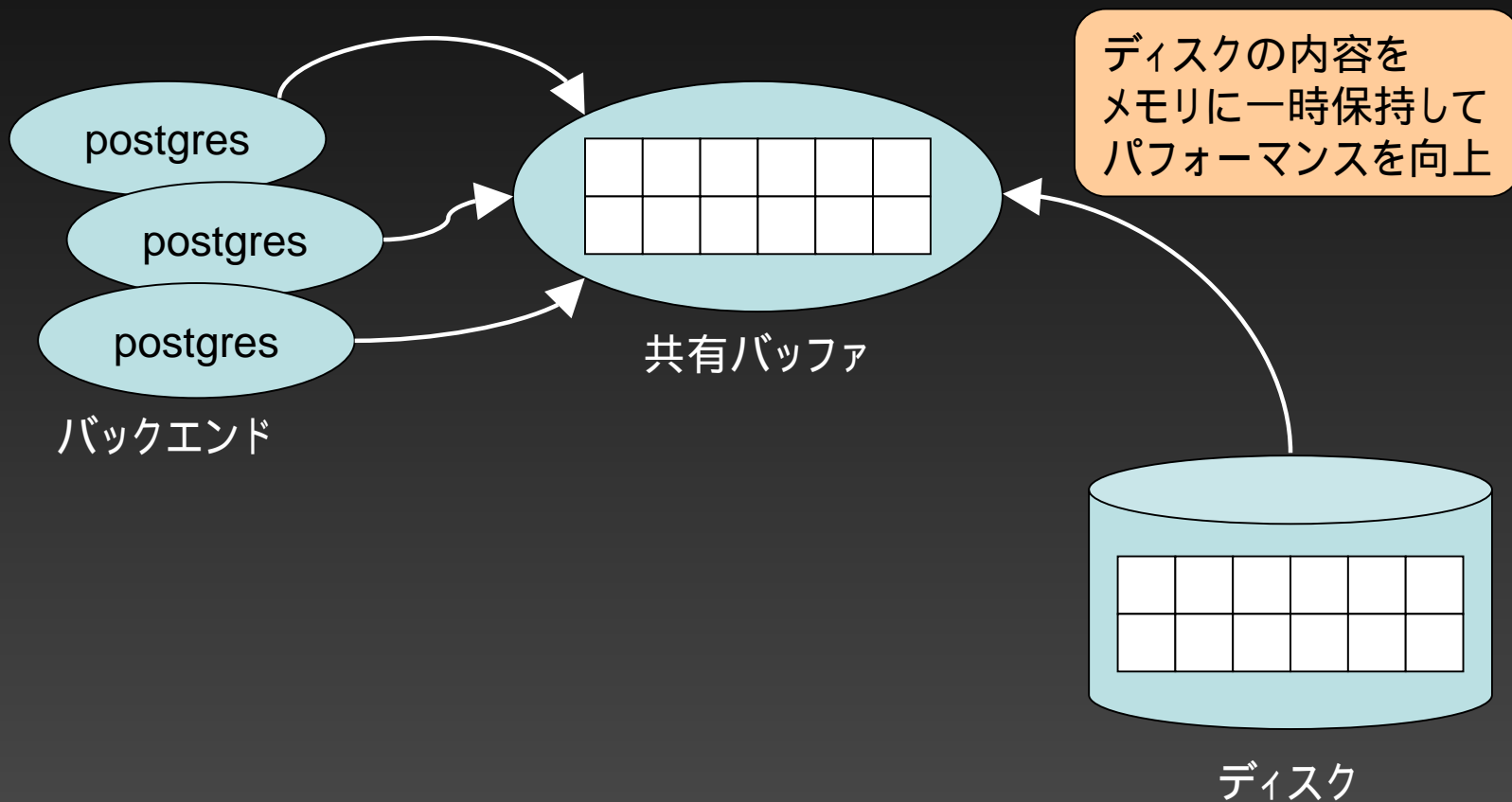
PostgreSQLクライアント



共有バッファとは

- ディスク上のブロックを配置するメモリ領域
 - ディスクI/Oを抑えるための機構
 - すべてのバックエンドプロセスから共有される
 - 8kB単位のブロックで管理される
 - postgresql.conf の shared_buffers でサイズ指定
 - 大きくすれば、より多くのブロックをメモリ中に配置できるが、、、
 - バッファ管理のオーバーヘッドが発生
 - バックエンドプロセスが使えるメモリ領域が減少

共有バッファ



レコード量とデータサイズ

- Pgbenchのaccountsテーブルのサイズ
 - 10万レコードの場合、15MB
 - 100万件レコードの場合、148MB
- 全レコード数を数える
 - SELECT count(*) FROM accounts
 - 10万レコードで107ミリ秒、100万レコードで1,009ミリ秒
- あるカラムの値でソートして、上位3件を取り出す。
 - SELECT * FROM accounts ORDER BY abalance DESC LIMIT 3
 - 10万レコードで831ミリ秒、100万レコードで7,991ミリ秒

初期設定

- 必ず変更すべき項目
 - shared_buffers
 - checkpoint_segments
 - wal_buffers
- 変更を推奨する項目
 - max_connections
 - stats_start_collector, stats_block_level, stats_row_level

(2) データベースの監視



Carmelo Aquilina @ flickr

監視すべき項目とその方法

- データベースサイズ、テーブルサイズ
 - データベースサイズ取得用関数
 - テーブルサイズ取得用関数
- トランザクション量
 - アクセス統計情報ビュー

データベースサイズの監視

- データベース、テーブルサイズ取得用関数
 - `pg_database_size()`
 - `pg_relation_size()`
 - `pg_total_relation_size()`
- 使い方
 - `SELECT pg_database_size('データベース名')`
 - `SELECT pg_relation_size('テーブル名')`

pg_stat_database

```
tpcw=# SELECT current_database(), pg_database_size(current_database());
```

```
current_database | pg_database_size
```

```
-----+-----  
tpcw          |          737104520  
(1 row)
```

```
tpcw=# select * from pg_stat_database;
```

```
datid | datname | numbackends | xact_commit | xact_rollback | blks_read | blks_hit  
-----+-----+-----+-----+-----+-----+-----  
10793 | postgres |          0 |          1 |          0 |          3 |          6  
      1 | template1 |          0 |          0 |          0 |          0 |          0  
10792 | template0 |          0 |          0 |          0 |          0 |          0  
16756 | tpcw      |         47 |         438 |          0 |      286375 |      10303  
(4 rows)
```

```
tpcw=#
```

トランザクションの監視

- アクセス統計情報 (システムビュー)
 - `pg_stat_database`
 - `pg_stat_all_tables`
- 使い方
 - `SELECT * FROM pg_stat_database`
 - `SELECT * FROM pg_stat_all_tables`

pg_stat_user_tables

```
tpcw=# SELECT * from pg_stat_user_tables ;
```

relid	schemaname	relname	seq_scan	seq_tup_read	idx_scan	idx_tup_fetch	n_tup_ins	n_tup_upd	n_tup_del
16766	public	country	78	3791	21	21	0	0	0
16784	public	orders	0	0	787	786	13	0	0
16759	public	author	47	11703	351	18126	0	0	0
16757	public	address	0	0	86	25	4	0	0
16782	public	order_line	0	0	8	822	66	0	0
16768	public	customer	0	0	85	85	61	19	0
16780	public	mm1	35	34965			0	0	0
16773	public	iot_customer	0	0	18	18	61	11	0
16764	public	cc_xacts	0	0	4	4	13	0	0
16775	public	item	203	155844	5583	8391	0	68	0

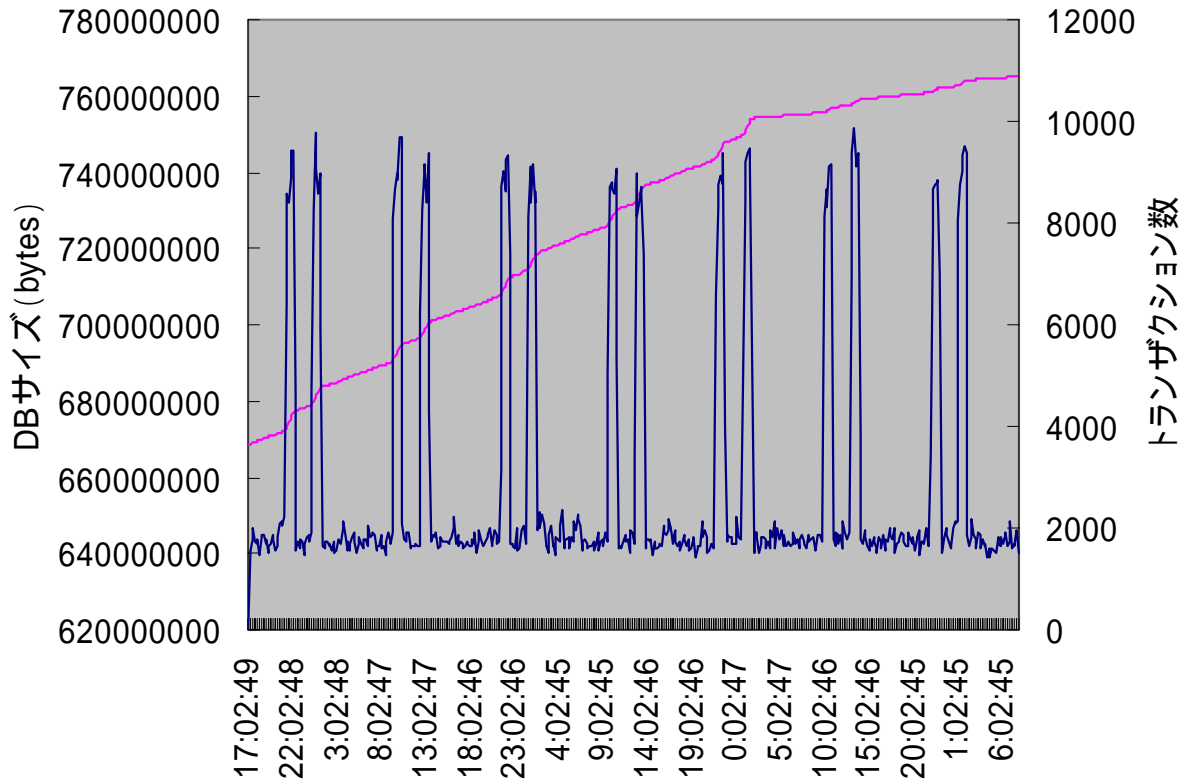
```
(10 rows)
```

```
tpcw=#
```

有効にすべきGUC項目

- stats_start_collector
 - 統計情報の収集を行う
- stats_command_string
 - SQLコマンドの統計を取得する
- stats_block_level
 - ブロック単位の統計を取得する
- stats_row_level
 - 行単位の統計を取得する

監視結果の可視化



(3) 性能劣化とメンテナンス



Michel Craipeau!!! @ flickr

不要領域のできる仕組み

- INSERT
 - テーブルファイルの末尾に追記
- DELETE
 - 削除マークを付けて、以後そのタプル(行)は参照しない。
- UPDATE
 - 変更前のタプルに削除マークを付け、更新後のタプルをINSERTする

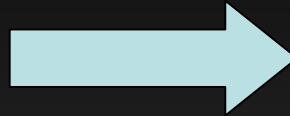
追記型アーキテクチャ

レコード
追加処理

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

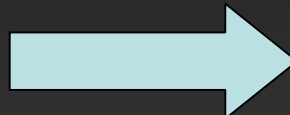


レコード
削除処理

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

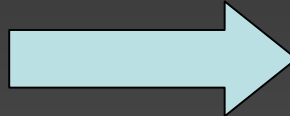
レコード2に削除マークが付けられる

レコード
更新処理

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



データファイル不要領域率の確認

- テーブルの不要領域の確認
 - Pgstattuple関数を使う
- インデックスの不要領域の確認
 - Pgstatindex関数を使う
- contribのpgstattupleモジュールに入っています
 - pgstatindexは8.2から同梱

Pgstattuple使用方法

```
pgbench=# ¥x
```

```
Expanded display is on.
```

```
pgbench=# SELECT * FROM pgstattuple('accounts');
```

```
-[ RECORD 1 ]-----+-----
```

table_len		138739712
tuple_count		1000000
tuple_len		128000000
tuple_percent		92.26
dead_tuple_count		32000
dead_tuple_len		4096000
dead_tuple_percent		2.95
free_space		2109248
free_percent		1.52

pgstatindex使用方法

```
pgbench=# ¥x
```

```
Expanded display is on.
```

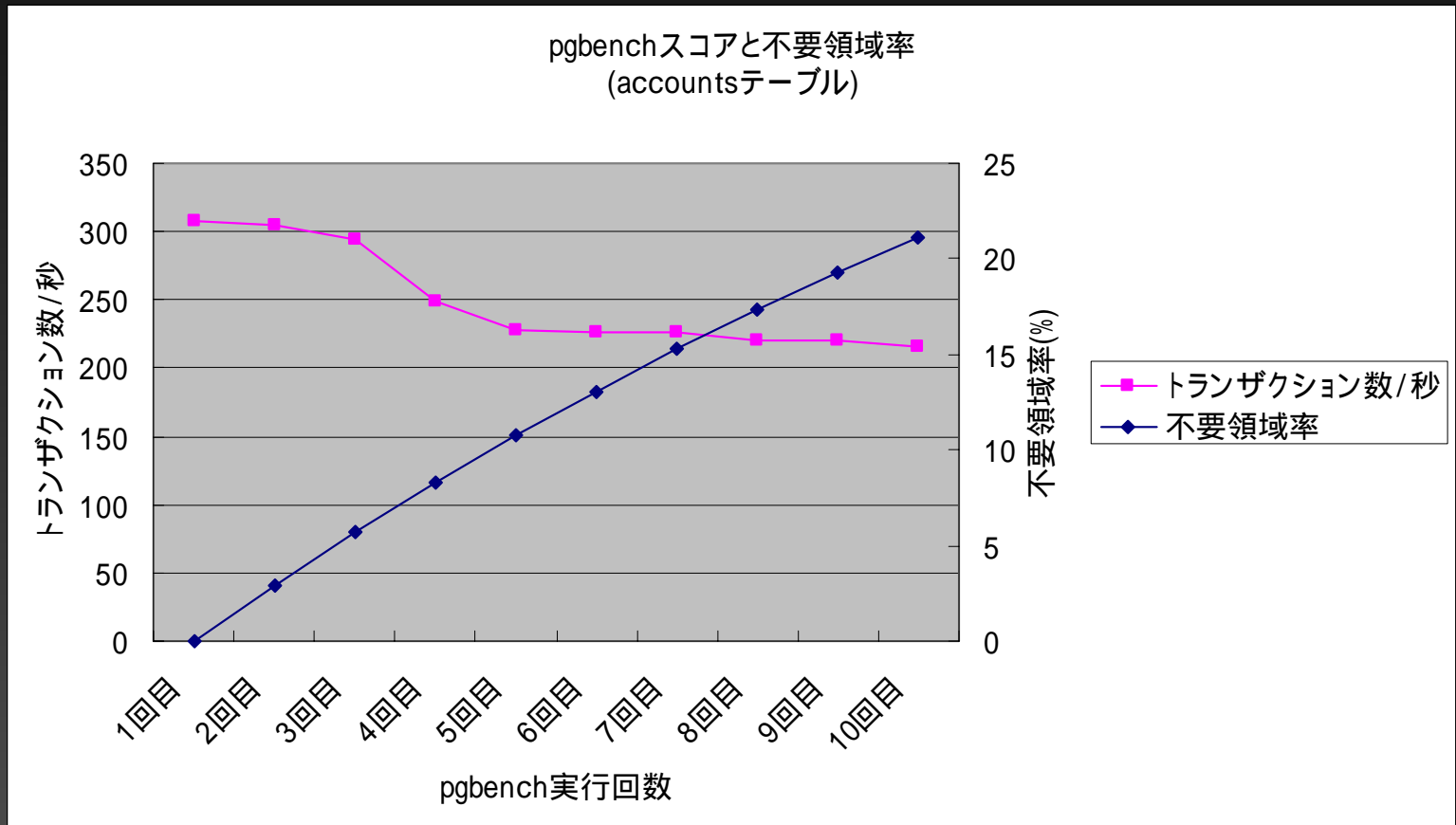
```
pgbench=# SELECT * FROM pgstatindex('accounts_pkey1');
```

```
-[ RECORD 1 ]-----+-----
```

version	2
tree_level	2
index_size	17956864
root_block_no	361
internal_pages	8
leaf_pages	2184
empty_pages	0
deleted_pages	0
avg_leaf_density	90.07
leaf_fragmentation	0

Pgbenchと不要領域率

- 不要領域が増えるとパフォーマンスが落ちる



VACUUM処理

- テーブルの不要領域を回収し、「未使用領域」として記録する。
 - 次の更新(追記)の時から、未使用領域を利用できるようになる。

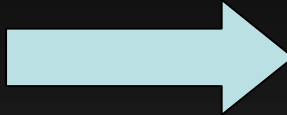
VACUUM処理

VACUUM前

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
付いている

VACUUM処理



VACUUM後

レコード1
空き領域
レコード3
レコード4
レコード2'

レコード2の領域が「空き領域」として
再利用可能になる。

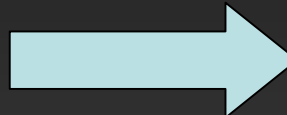
追記前

VACUUM
してあると

レコード1
空き領域
レコード3
レコード4
レコード2'

「空き領域」がある

レコード5を追記



追記後

レコード1
レコード5
レコード3
レコード4
レコード2'

ファイルサイズを変えずに追記できる

追記前

VACUUM
してないと

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の領域が埋まったまま

レコード5を追記



追記後

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5

ファイルサイズが大きくなる

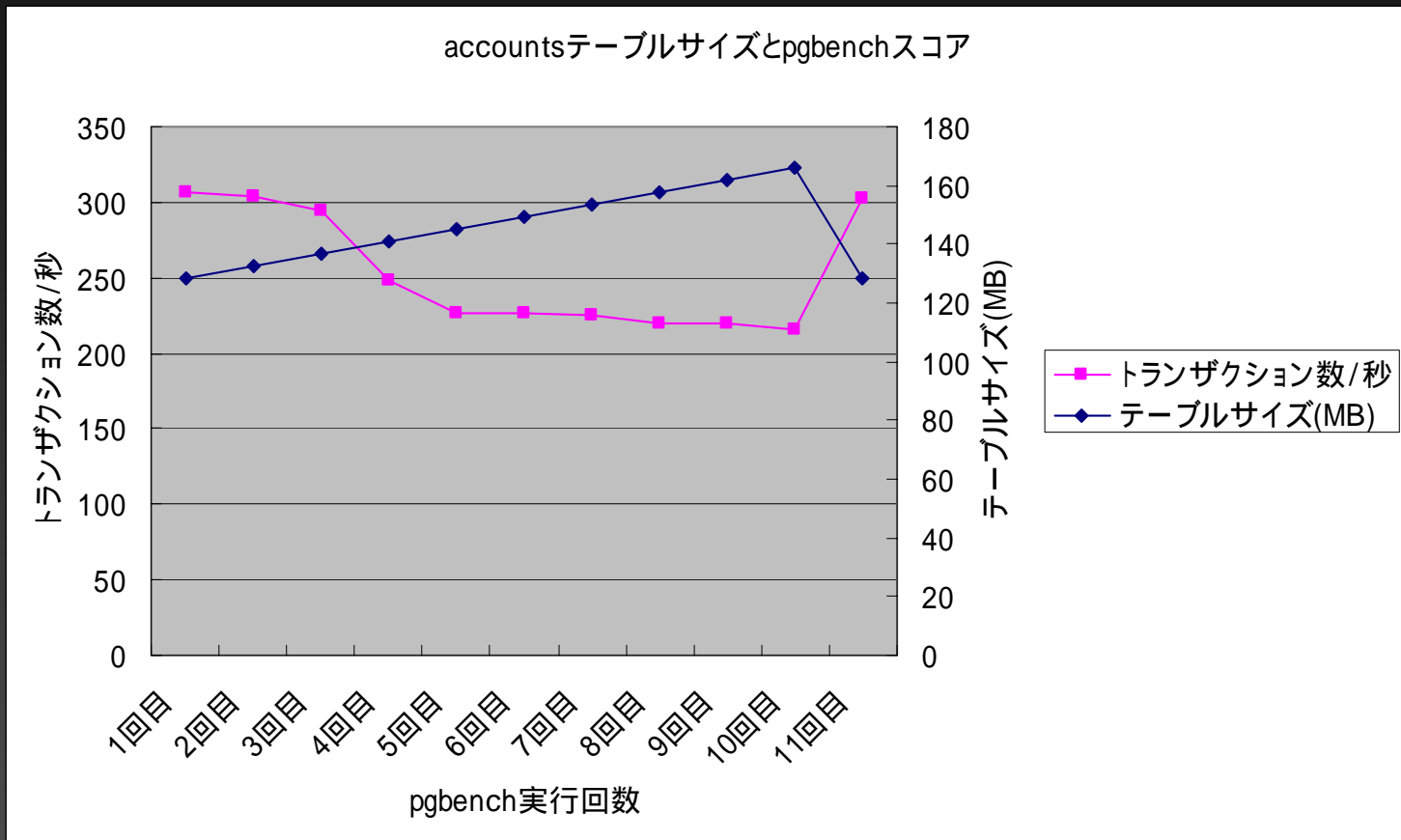


REINDEX処理

- インデックスを再作成する
 - 不要領域のないインデックスが作られる
- REINDEXコマンド
 - REINDEX TABLE <テーブル名>
 - REINDEX DATABASE <データベース名>

メンテナンスの効果

- FULL VACUUMとREINDEXを実行



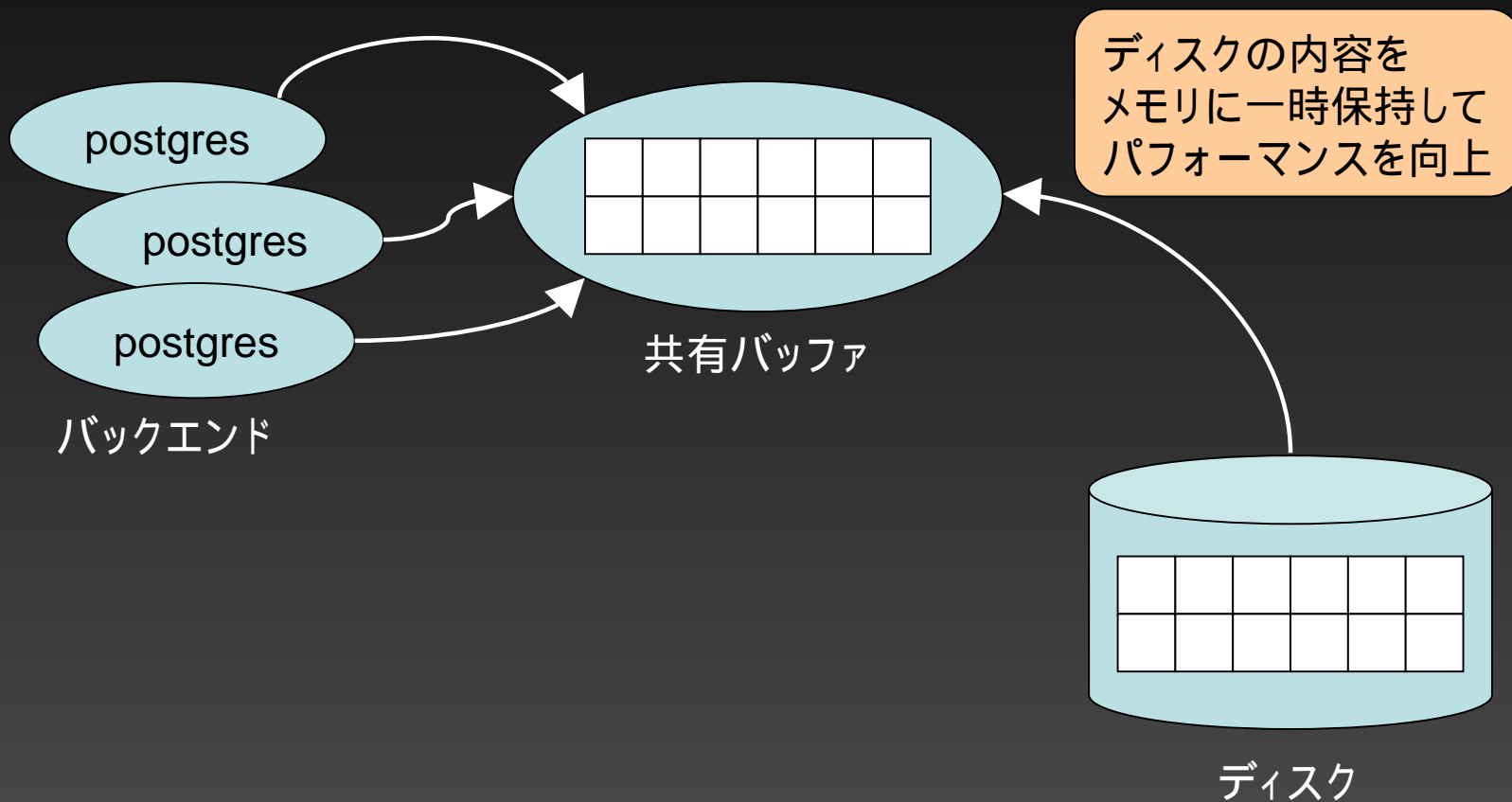
その他の情報

- VACUUMの自動化「autovacuum」機能

(4) パフォーマンスチューニング (GUC編)

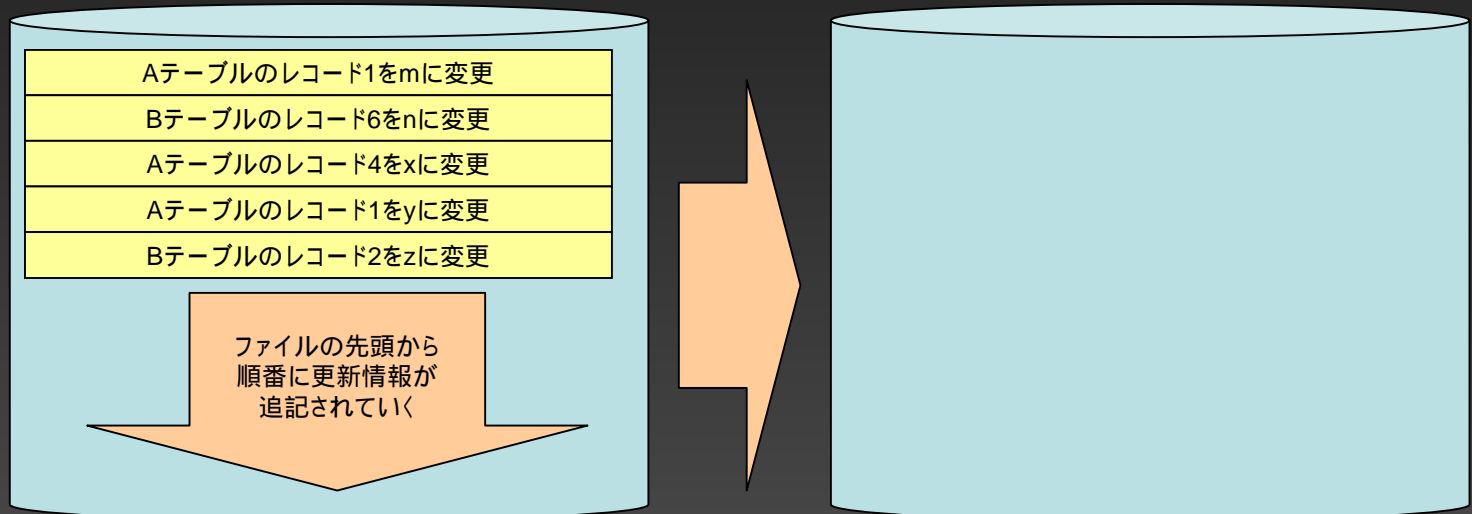


共有バッファ

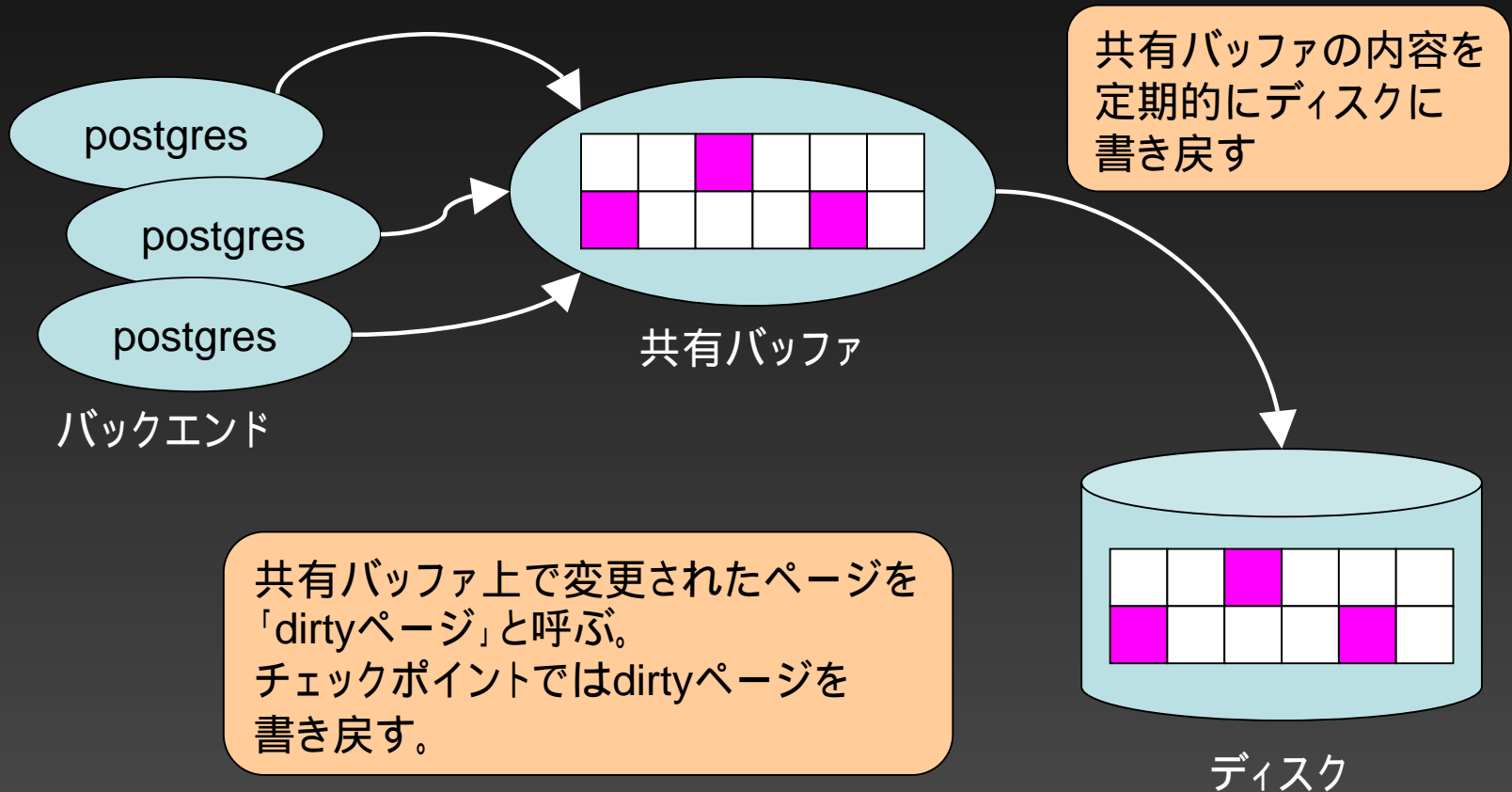


トランザクションログ

- 更新情報を追記していく
 - 1ファイル16MB
 - 16MBを使い終わったら次のファイルに切り替え

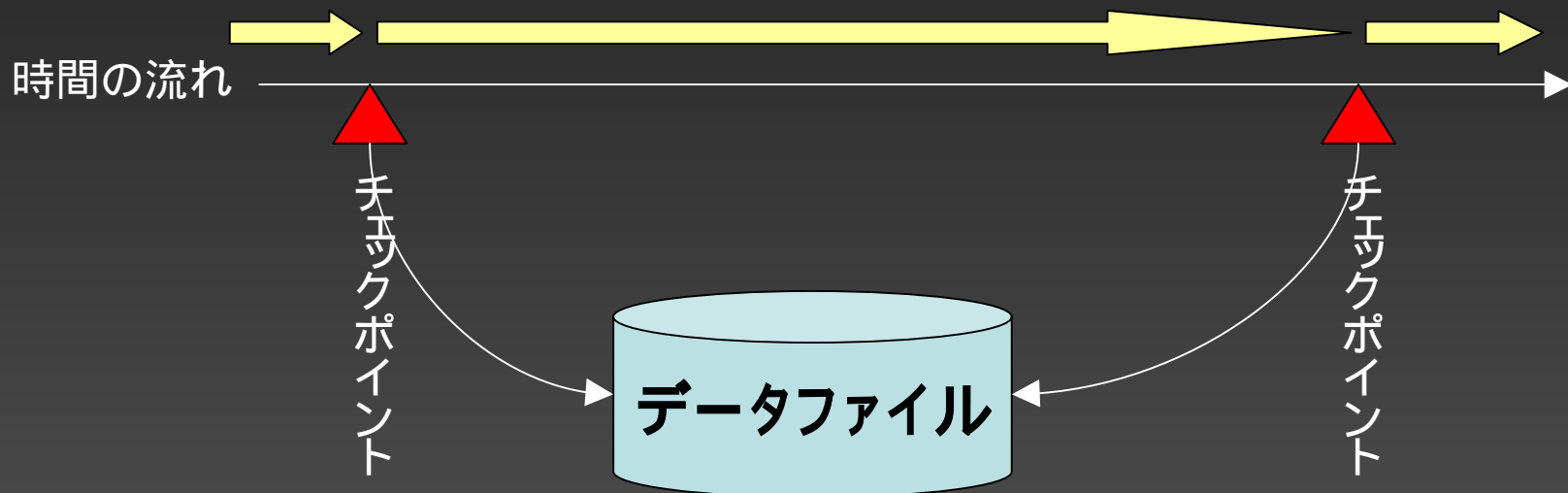


チェックポイント処理



チェックポイント処理

チェックポイントとチェックポイントの間は
共有メモリの内容のみを書き換える。
チェックポイントが発生した時点で、
書き換えられたページ (dirtyページ) を
ディスクに一気に書き戻す。

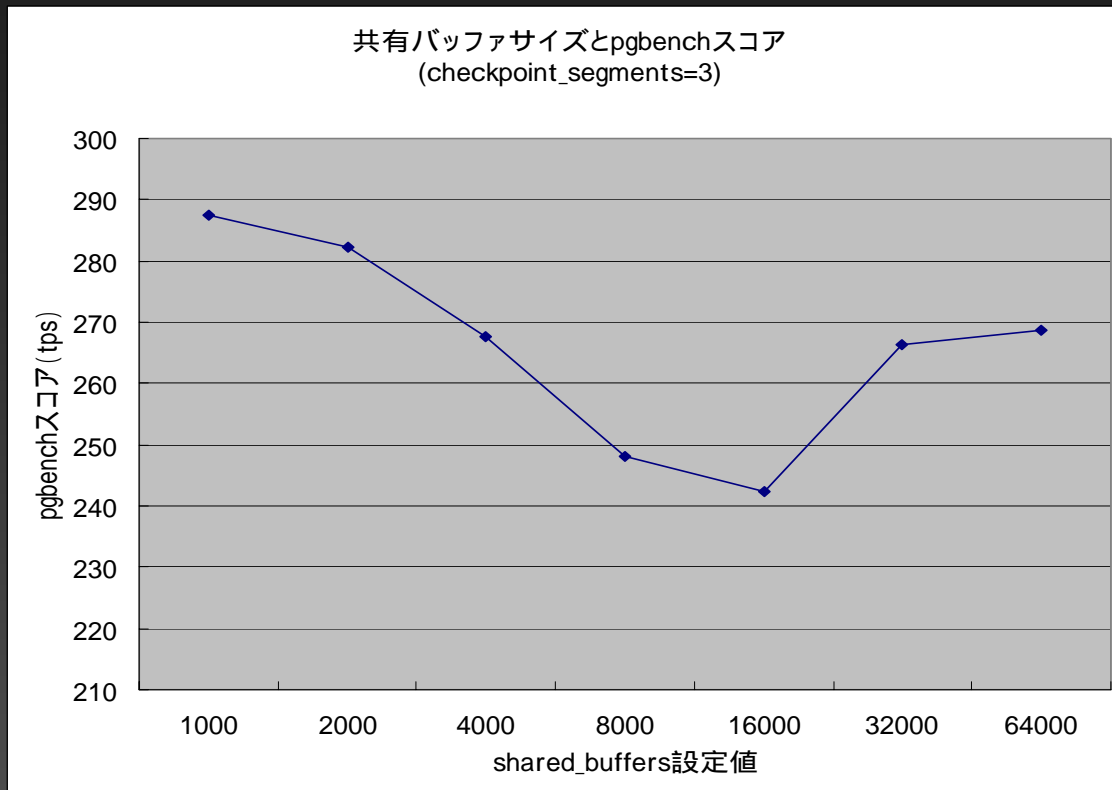


postgresql.conf

- GUCパラメータ
 - shared_buffers
 - checkpoint_segments

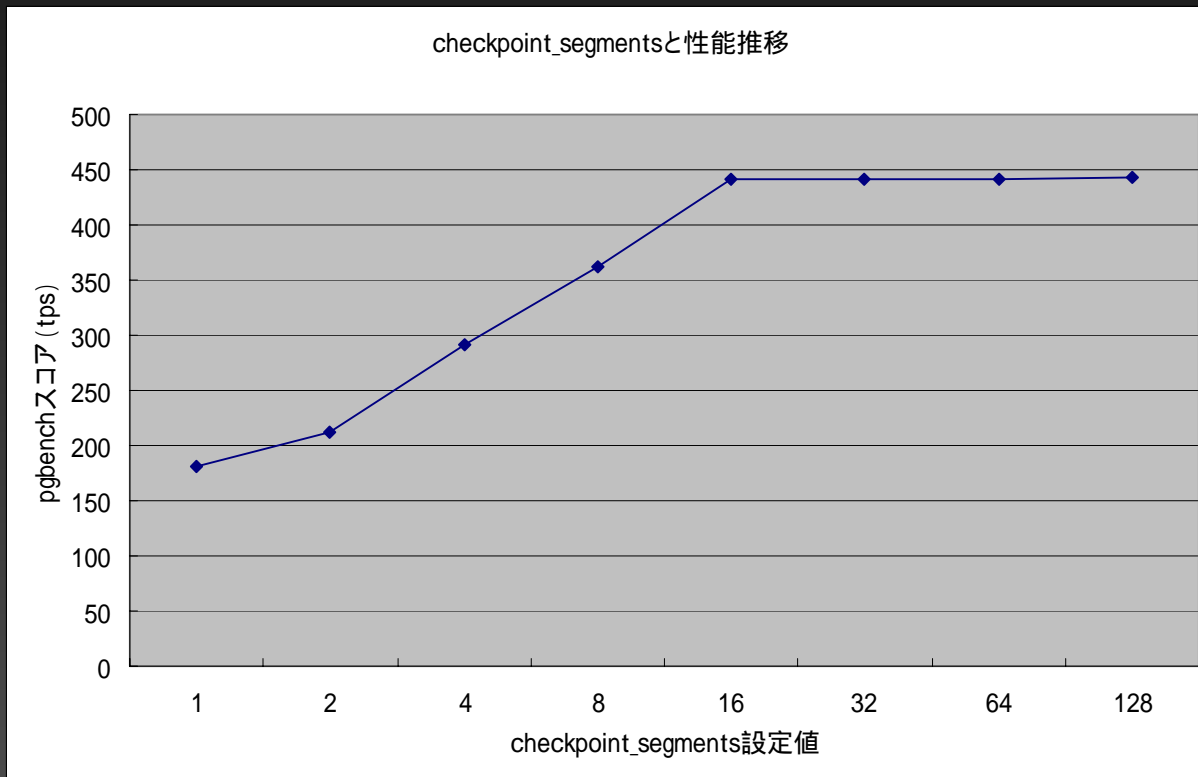
パラメータの変更とパフォーマンスの変化

- shared_buffers
 - checkpoint_segments=3固定



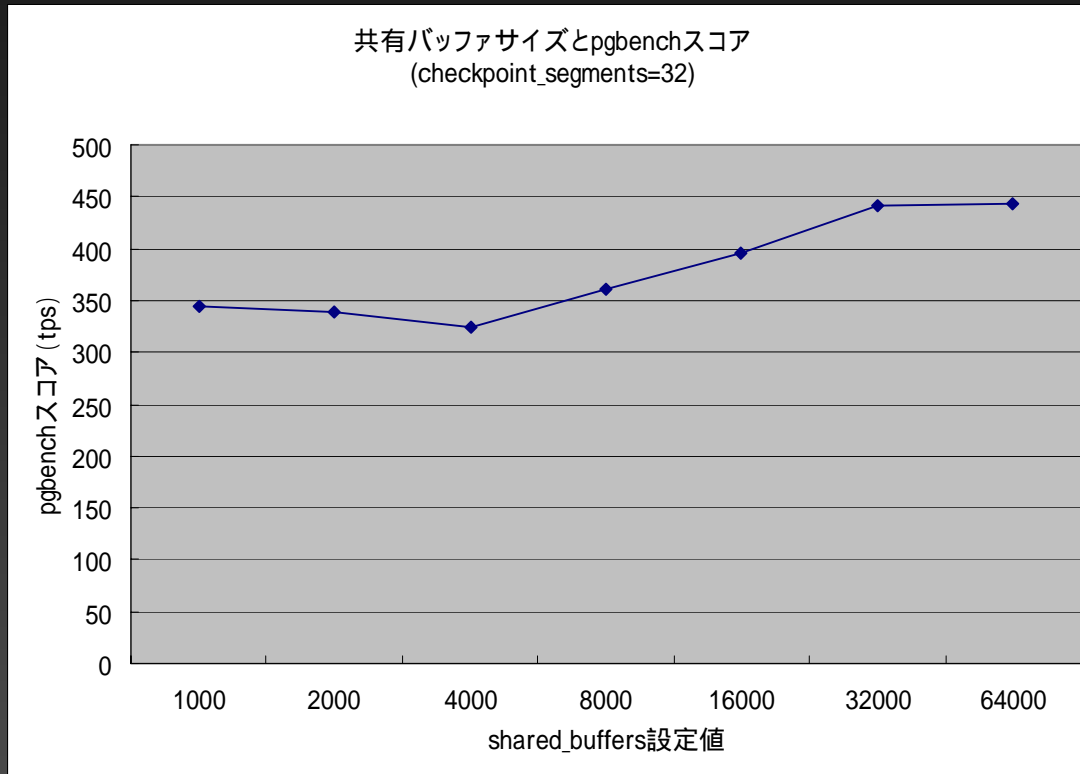
パラメータの変更とパフォーマンスの変化

- checkpoint_segments
 - shared_buffers=32000固定



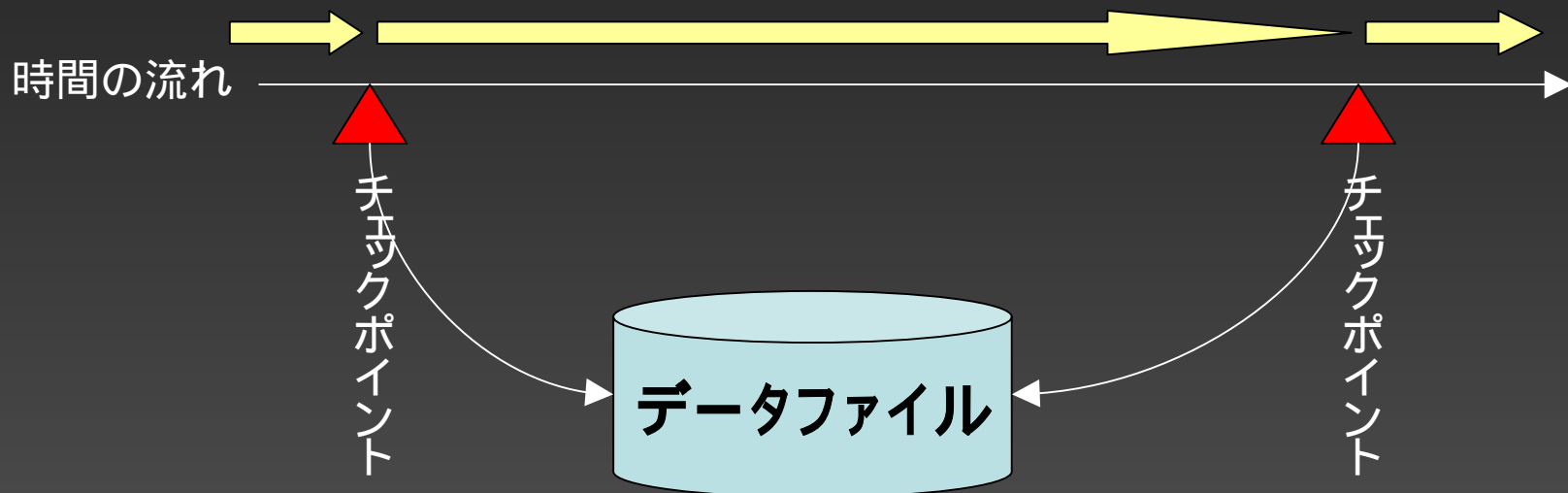
パラメータの変更とパフォーマンスの変化

- shared_buffers
 - checkpoint_segments=32固定

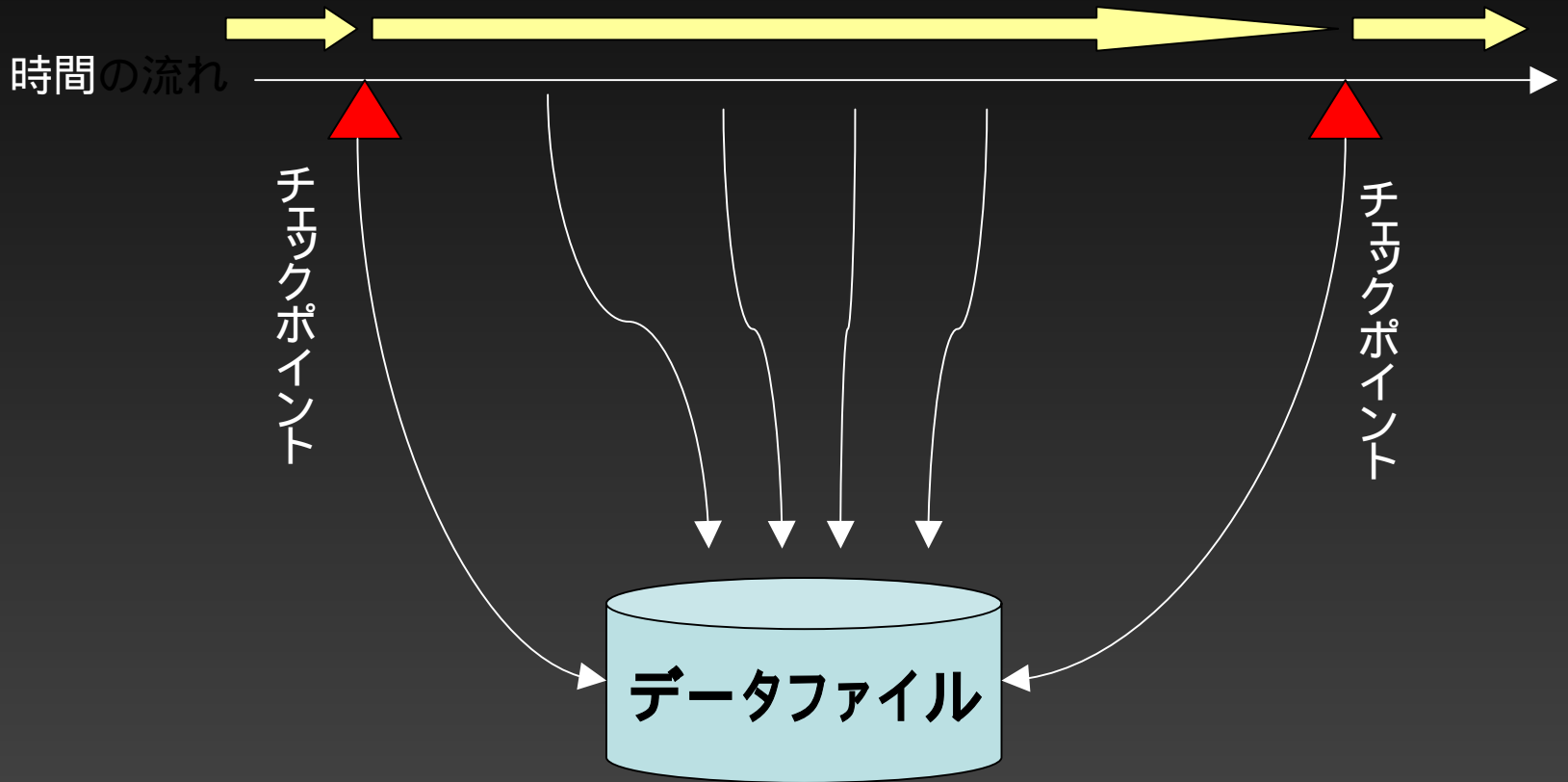


チェックポイント処理

チェックポイントとチェックポイントの間は
共有メモリの内容のみを書き換える。
チェックポイントが発生した時点で、
書き換えられたページ (dirtyページ) を
ディスクに一気に書き戻す。



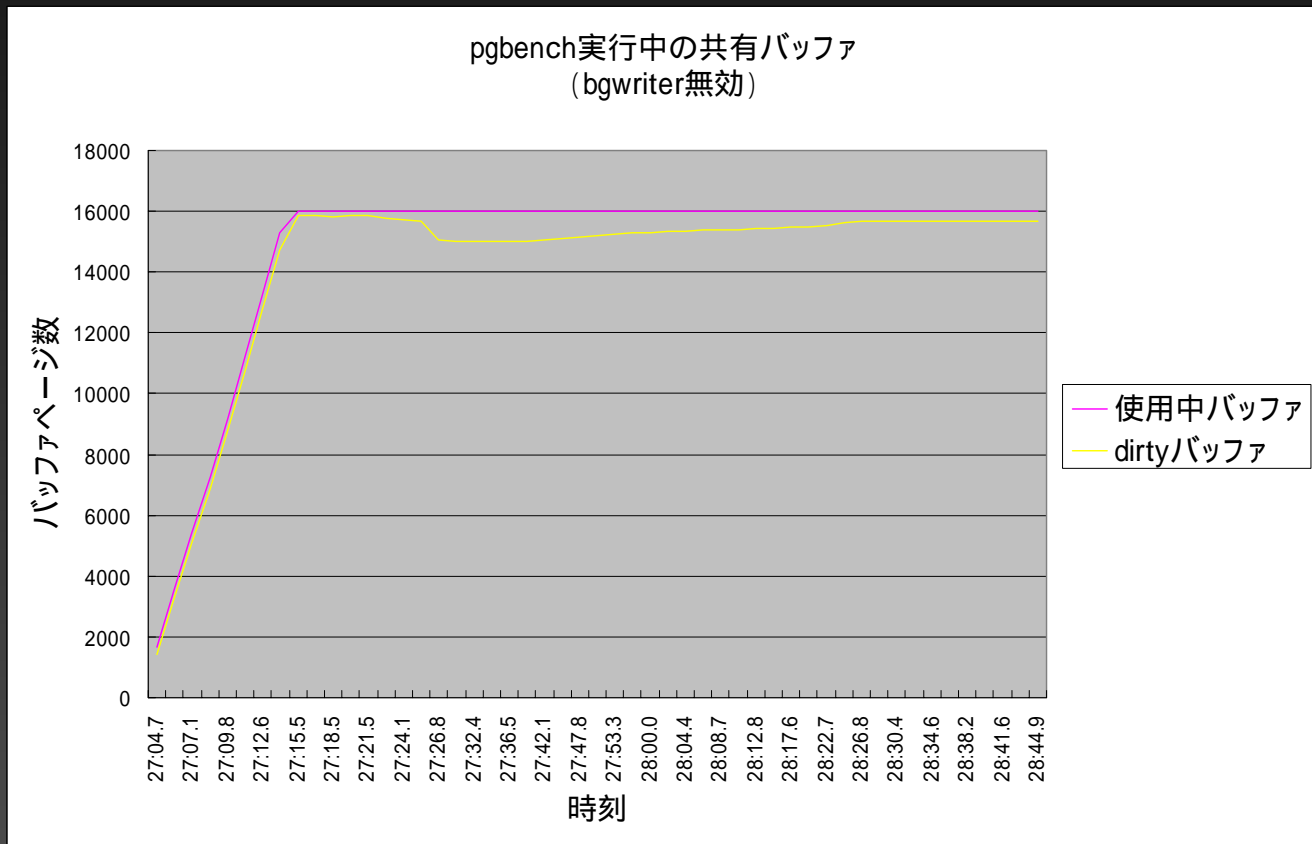
バックグラウンドライタ



チェックポイントでも書き出すが、
チェックポイント以外でも書き出すことによって、
チェックポイント時の負荷を下げる。

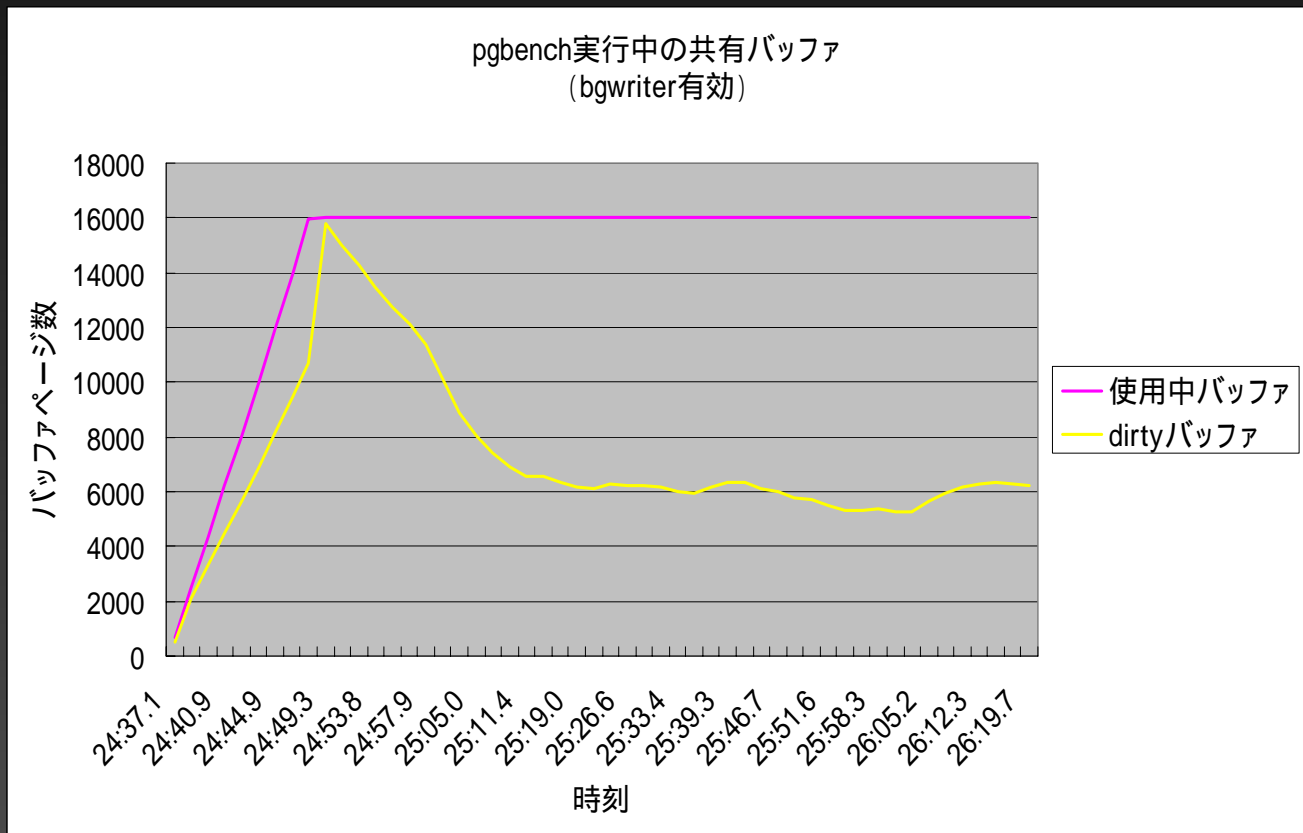
Bgwriterの効果

- ほとんどがDirty (更新済み) なバッファ



Bgwriterの効果

- Dirty (更新済み) のバッファが減っている



その他のパラメータ

- チェックポイントとWAL
- VACUUM関連

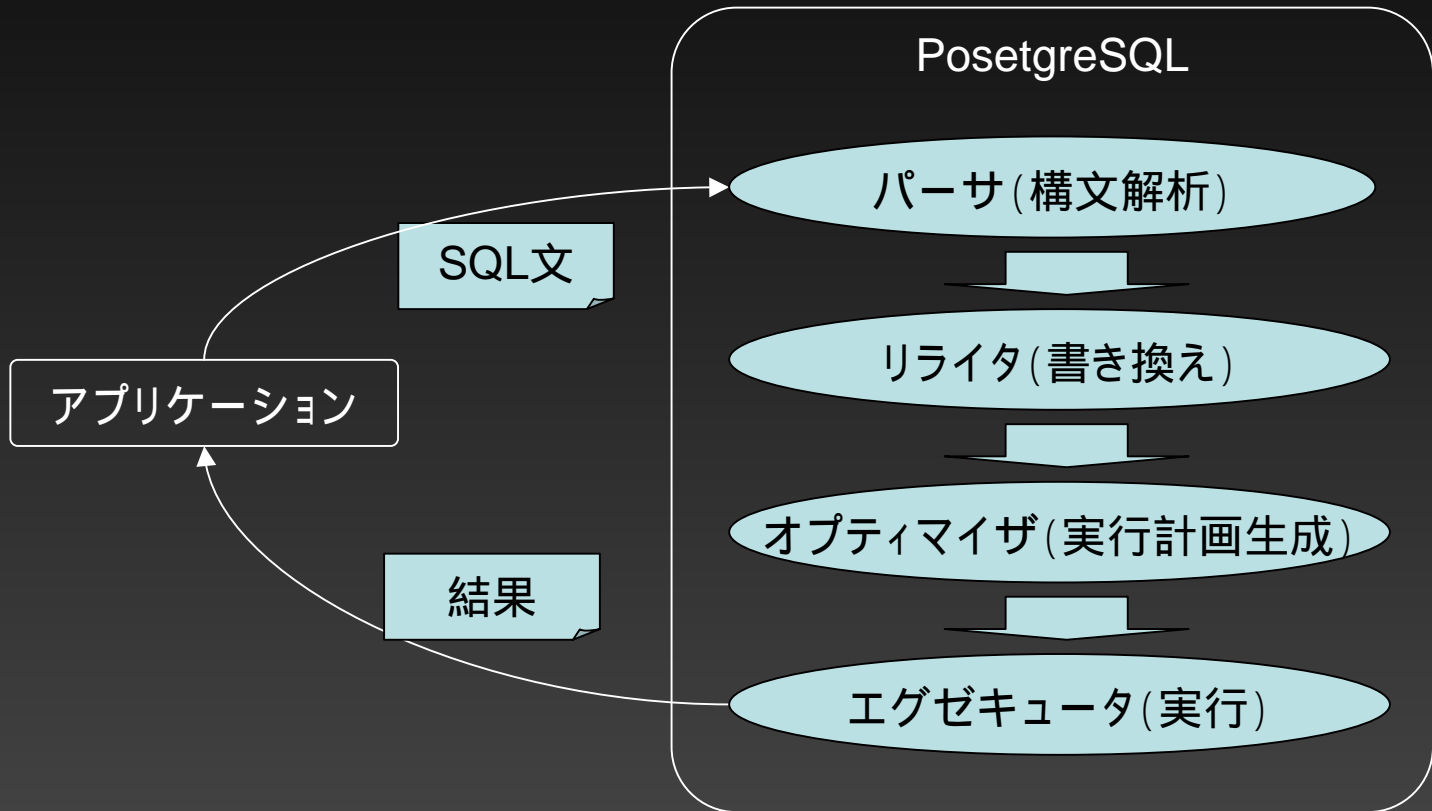
(5) パフォーマンスチューニング (SQL編)



SQLチューニングとは

- 「現状の実行計画を解析しながら、(ある特定の結果を取得するために)より効率の良い実行計画を生成するSQL文を考え出す」

SQL文の処理される流れ



EXPLAINとEXPLAIN ANALYZE

- EXPLAIN

- 「最適である」と判断された「実行計画(アクセスプラン)」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

- EXPLAIN ANALYZE

- 「実行結果」を表示
- 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

EXPLAIN

```
DBT1=# EXPLAIN SELECT count(*) FROM orders;  
QUERY PLAN
```

Aggregate (cost=81909.50..81909.51 rows=1 width=0)

-> Seq Scan on orders (cost=0.00..75427.40
rows=2592840 width=0)

(2 rows)

EXPLAIN ANALYZE

```
DBT1=# EXPLAIN ANALYZE SELECT count(*) FROM orders;  
QUERY PLAN
```

```
-----  
Aggregate (cost=81909.50..81909.51 rows=1 width=0) (actual  
time=14300.250..14300.251 rows=1 loops=1)
```

```
-> Seq Scan on orders (cost=0.00..75427.40 rows=2592840  
width=0) (actual time=0.060..12343.583 rows=2592139 loops=1)
```

```
Total runtime: 14310.862 ms
```

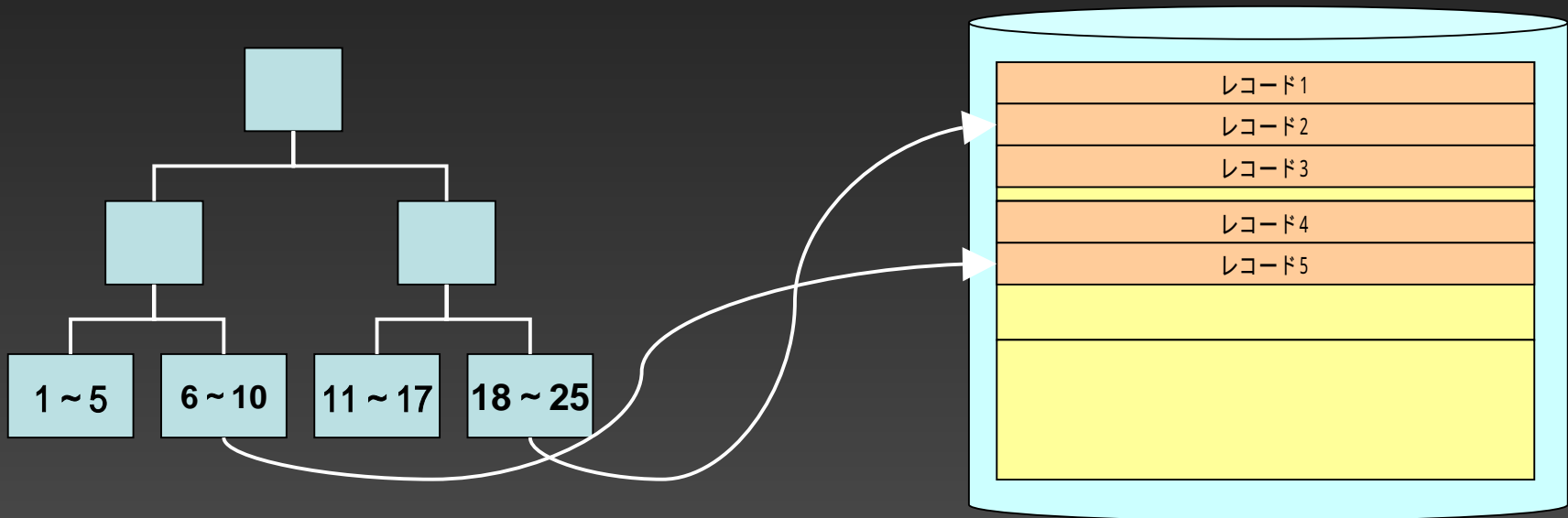
```
(3 rows)
```

チューニングのポイント

- シーケンシャルスキャンを減らす
- ソート処理を減らす
- 中間データを少なくする

シーケンシャルスキャンを減らす

- 極力インデックスを使えるようにする



ソートサイズを小さくする

pgAdmin III Query - DBT1 @ st19:8080 *

ファイル(E) 編集(E) クエリー(Q) ヘルプ(H)

DBT1 @ st19:8080

```
1 SELECT ol_i_id, ol_qty, o_date
2 FROM orders, order_line
3 WHERE ol_o_id=o_id
4 ORDER BY o_date desc
5
```

The diagram illustrates the execution plan for the query. It starts with two tables, 'orders' and 'order_line', each represented by a grid icon. Arrows point from each table to a 'Sort' icon, which consists of two vertical bars with horizontal lines, representing a sort operation. From these two 'Sort' icons, a large arrow points to a 'Merge Join' icon, which is a grid with a curved arrow. Finally, an arrow points from the 'Merge Join' icon to another 'Sort' icon. This sequence indicates that the data from both tables is sorted separately, then joined using a merge join, and finally sorted again.

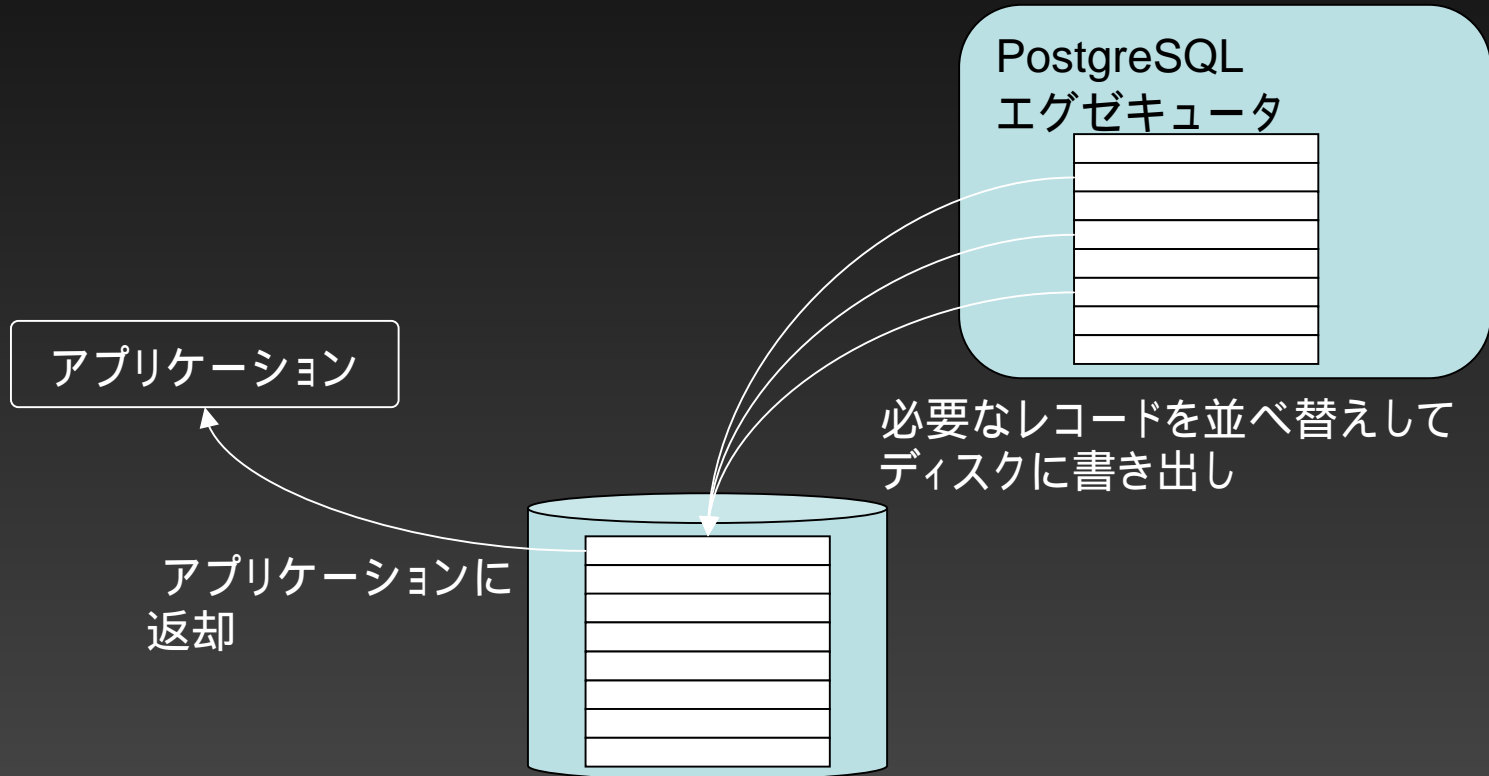
orders → Sort → Merge Join → Sort

order_line → Sort → Merge Join → Sort

データの出力 解釈 メッセージ ヒストリー

OK. 10 行 20 ms

ディスクソートの仕組み



中間データを小さくする

pgAdmin III Query - DBT1 @ st19:8080 *

ファイル(E) 編集(E) クエリー(Q) ヘルプ(H)

DBT1 @ st19:8080

```
1 SELECT count(*) FROM orders o, order_line ol, customer c
2 WHERE o.o_id=ol.ol_o_id
3 AND o.o_c_id=c.c_id
4 AND c.c_fname = 'Tutbrz S'
5 AND c.c_lname='&?05,$9^n';
```

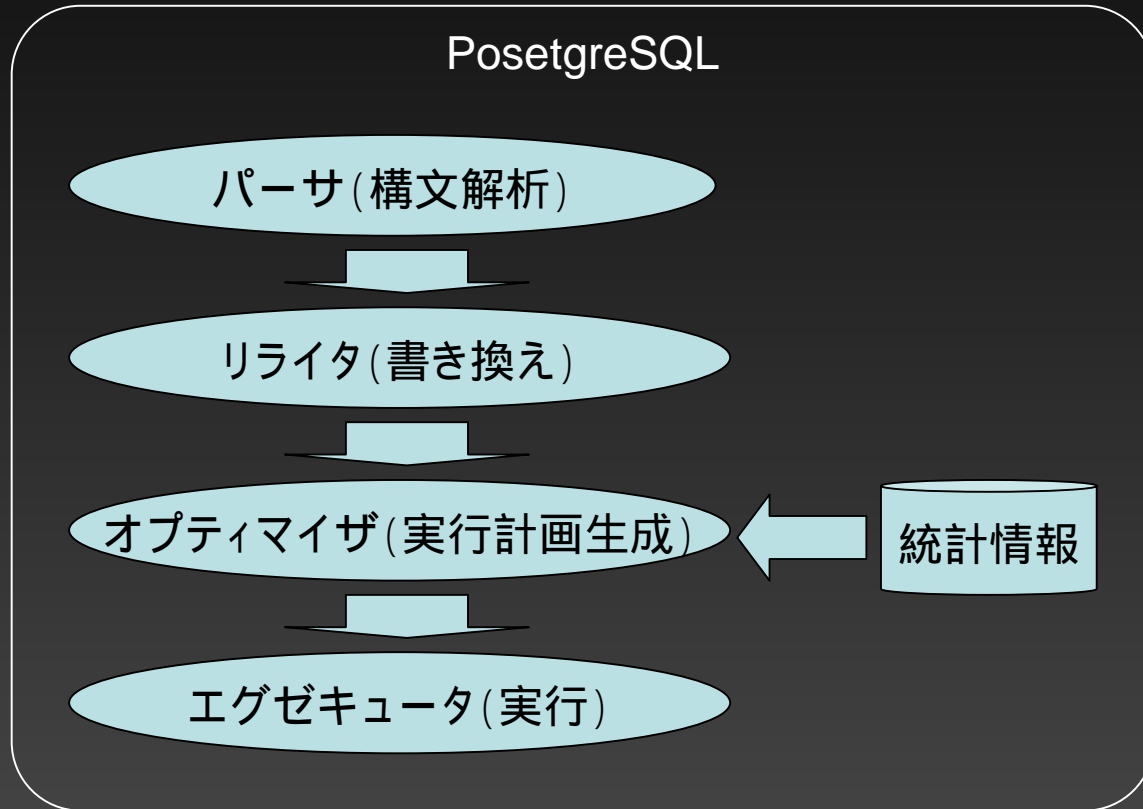
The diagram illustrates the execution plan for the provided SQL query. It shows the following steps:

- orders** table is processed by a **Sort** operator.
- customer** table is processed by a **Sort** operator.
- order_line** table is processed by a **Sort** operator.
- The sorted **orders** and **customer** data are joined by a **Merge Join** operator.
- The result of the first join is then joined with the sorted **order_line** data by a second **Merge Join** operator.
- The final result is processed by an **Aggregate** operator (represented by a sigma symbol Σ).

データの出力 解釈 メッセージ ヒストリー

OK. 17行 20ms

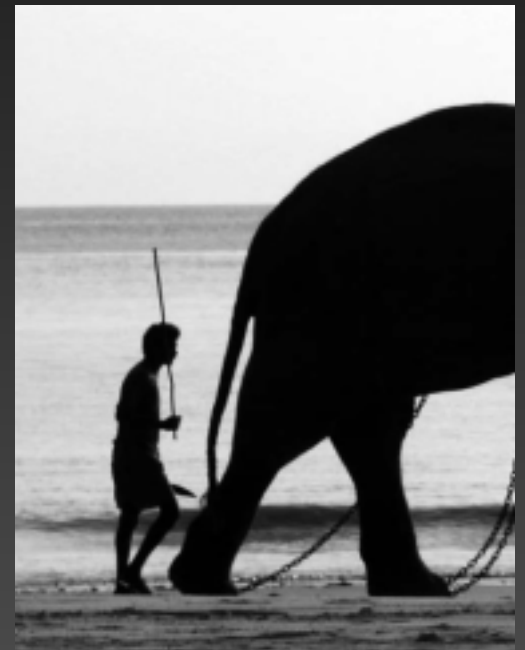
統計情報はオプティマイザの判断材料になる



ANALYZEによるテーブル統計情報 の更新

- ANALYZEコマンド
 - ANALYZE <テーブル名>
 - ANALYZE

(6) バックアップ & リカバリ



athinasmile @ flickr

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

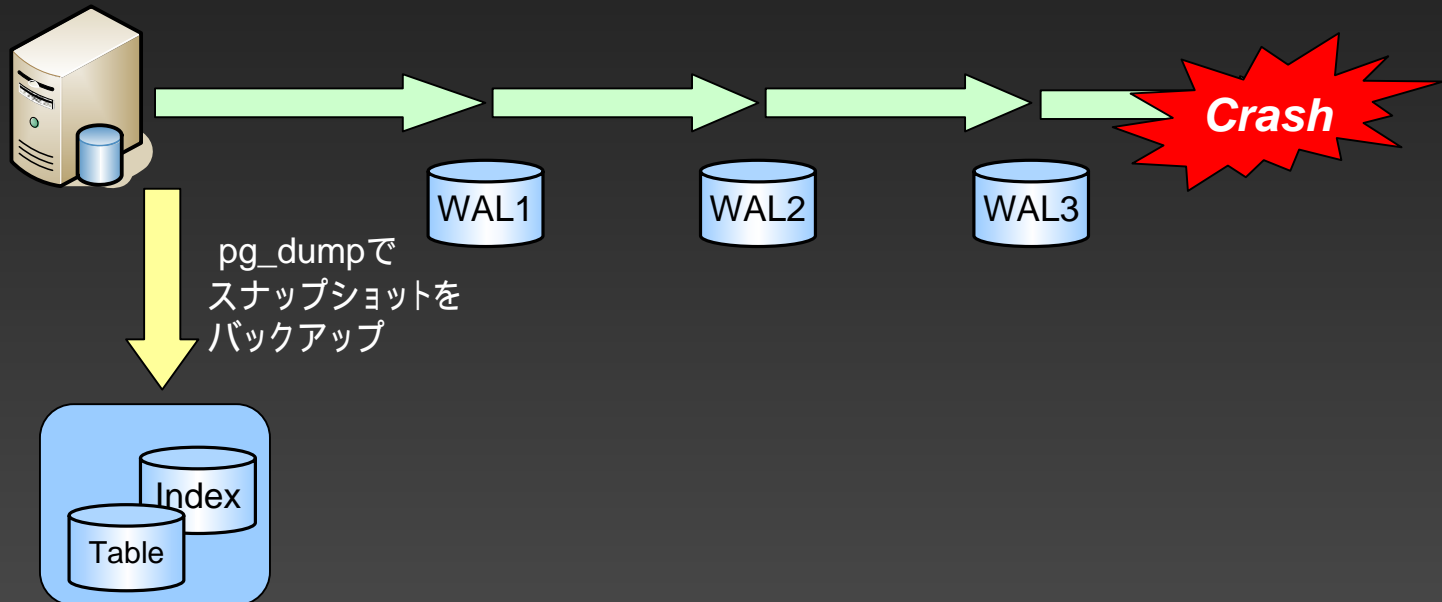
- コールドバックアップ
- ホットバックアップ
- アーカイブログバックアップ

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
- データファイルファイル全体をバックアップ

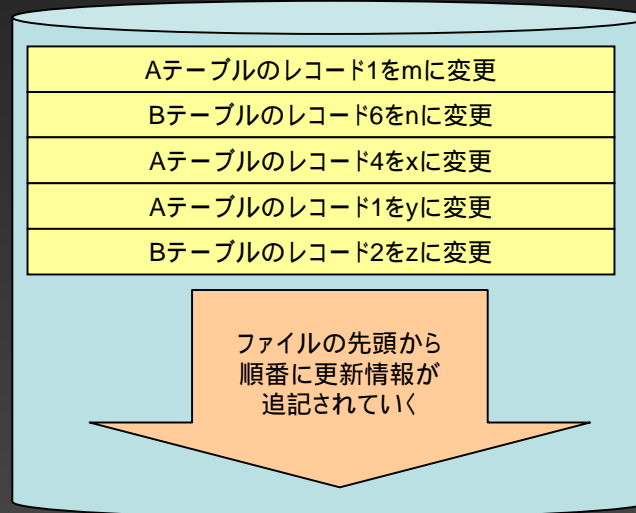
ホットバックアップ (pg_dump)

- 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
 - シンプルで、テーブル単位のバックアップも可



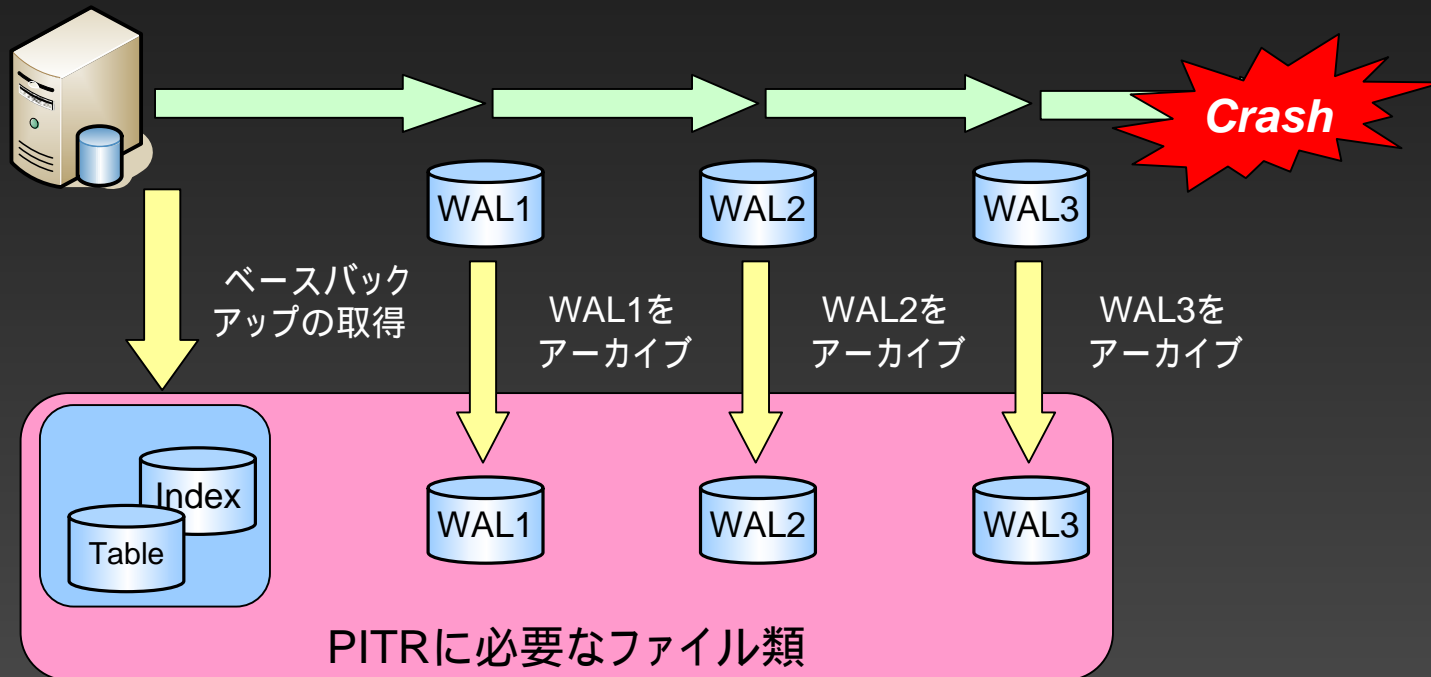
Point-in-time recovery (PITR)

- 更新情報が時系列に保存されているWALを利用



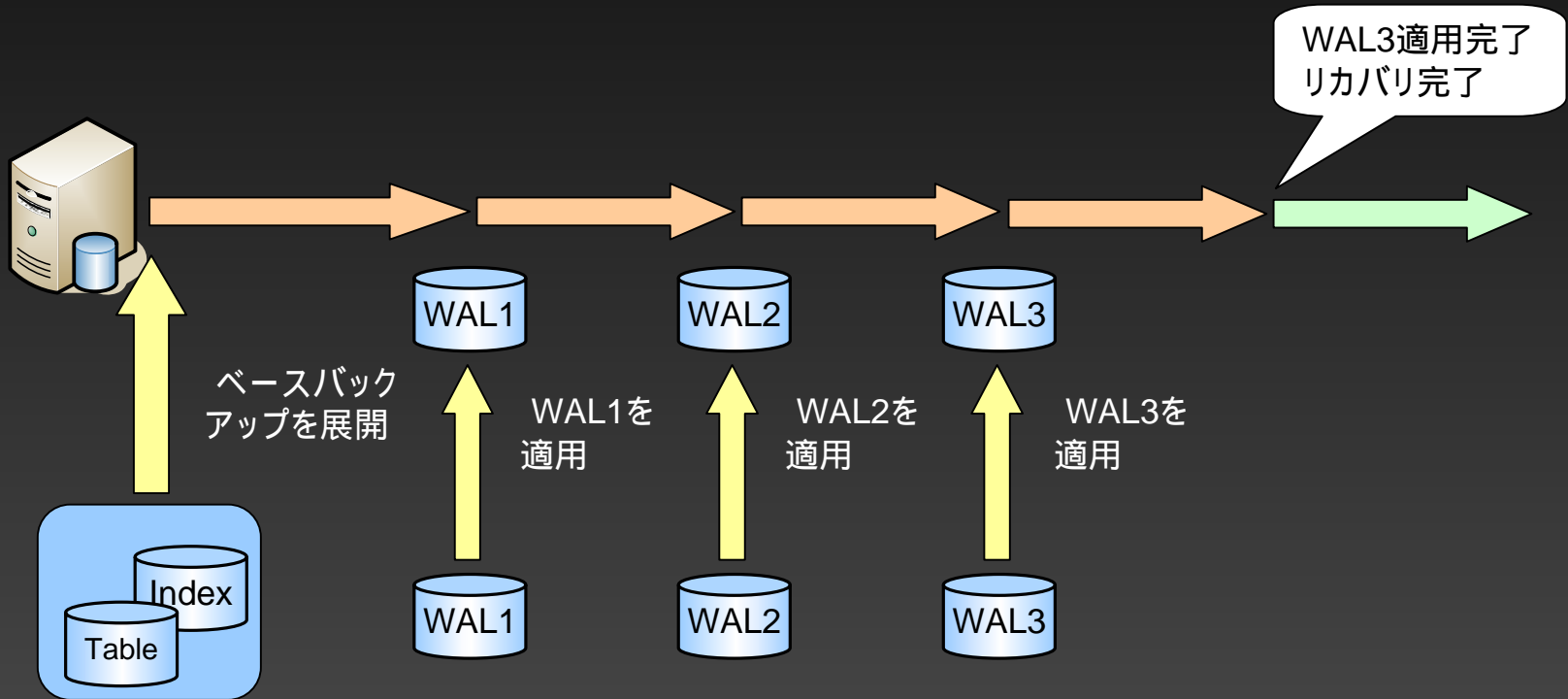
アーカイブログとPITRを用いたバックアップ

- ベースバックアップ + アーカイブログ
– archive_commandで設定



アーカイブログとPITRを用いたり カバリ

- recovery.confで設定を行う



バックアップ & リカバリまとめ

- バックアップ & リカバリの種類は3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 & リカバリはリハーサルをしよう！

今回の話、全部入ってます！

- 連載「PostgreSQL安定運用のコツ」
WEB+DB PRESS vol.32 ~ 37



+



ご清聴ありがとうございました



Fenners1984 @ flickr