PostgreSQL
SQL チューニングの基礎

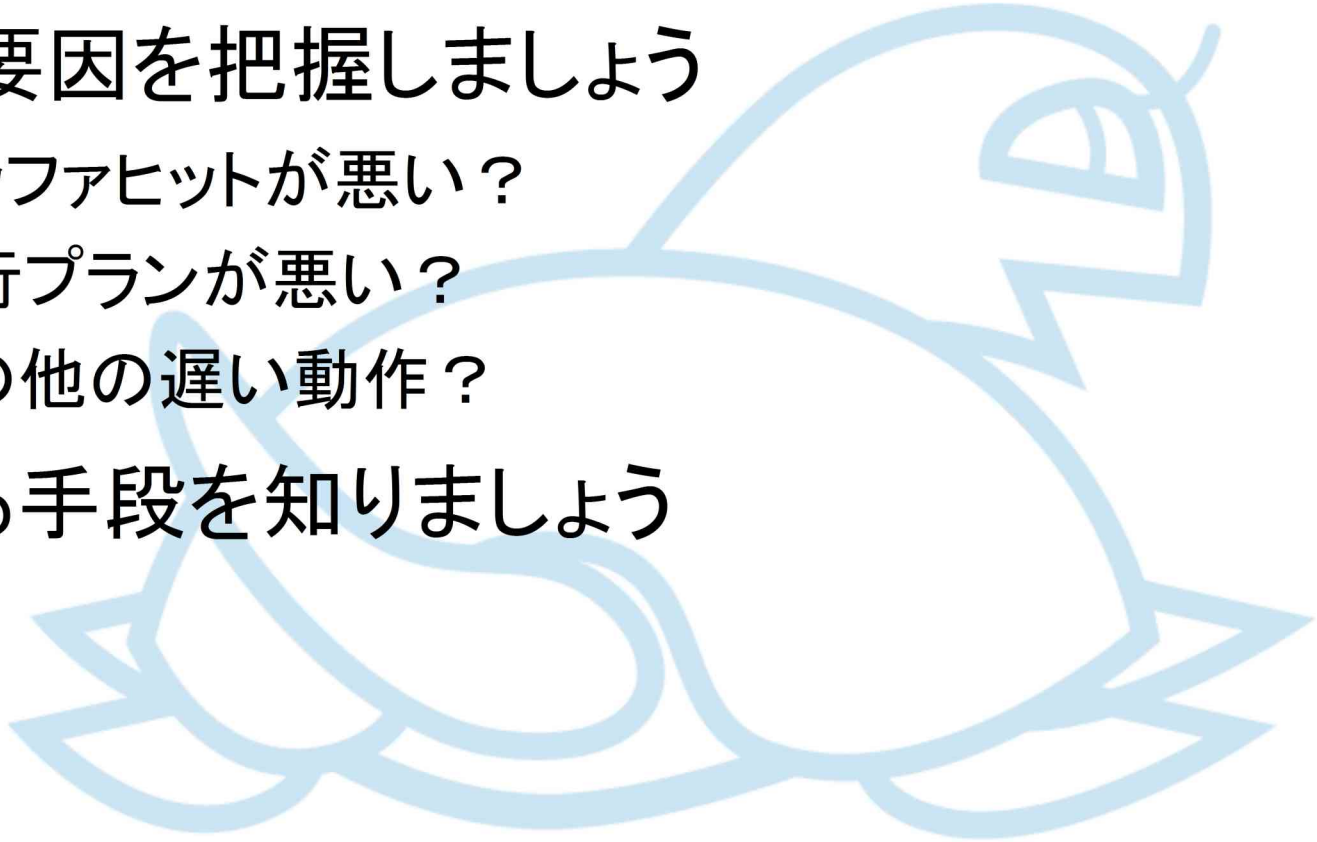
PostgrSQL カンファレンス
チュートリアルセッション T3
2018-11-22

日本 PostgreSQL ユーザ会 理事長 高塚 遥
harukat@postgresql.jp

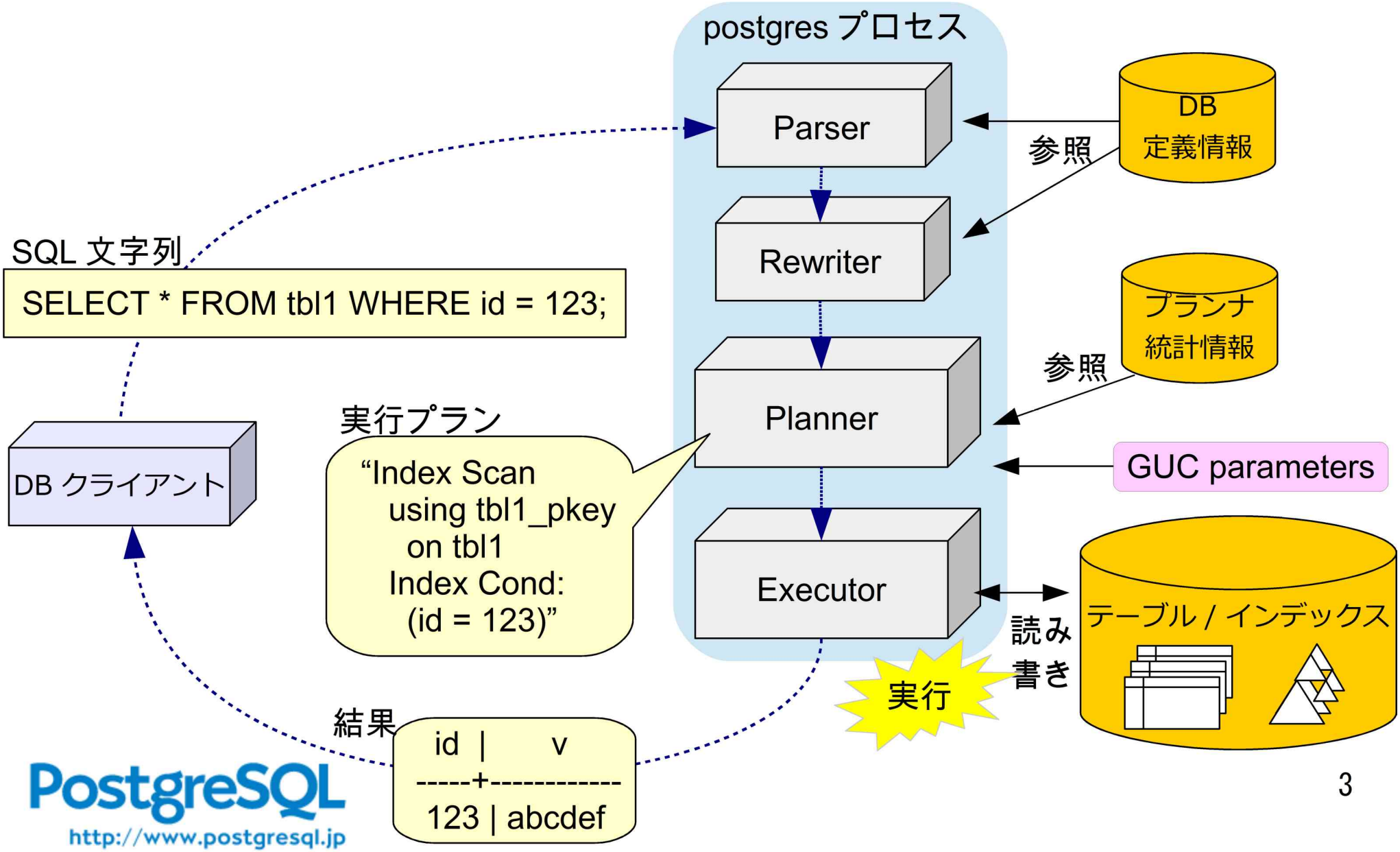
PostgreSQL
http://www.postgresql.jp

「 SQL 応答が遅い」に立ち向かうには

- SQL が実行される基本的な仕組みを知りましょう
- 遅い要因を把握しましょう
  - バッファヒットが悪い？
  - 実行プランが悪い？
  - その他の遅い動作？
- できる手段を知りましょう

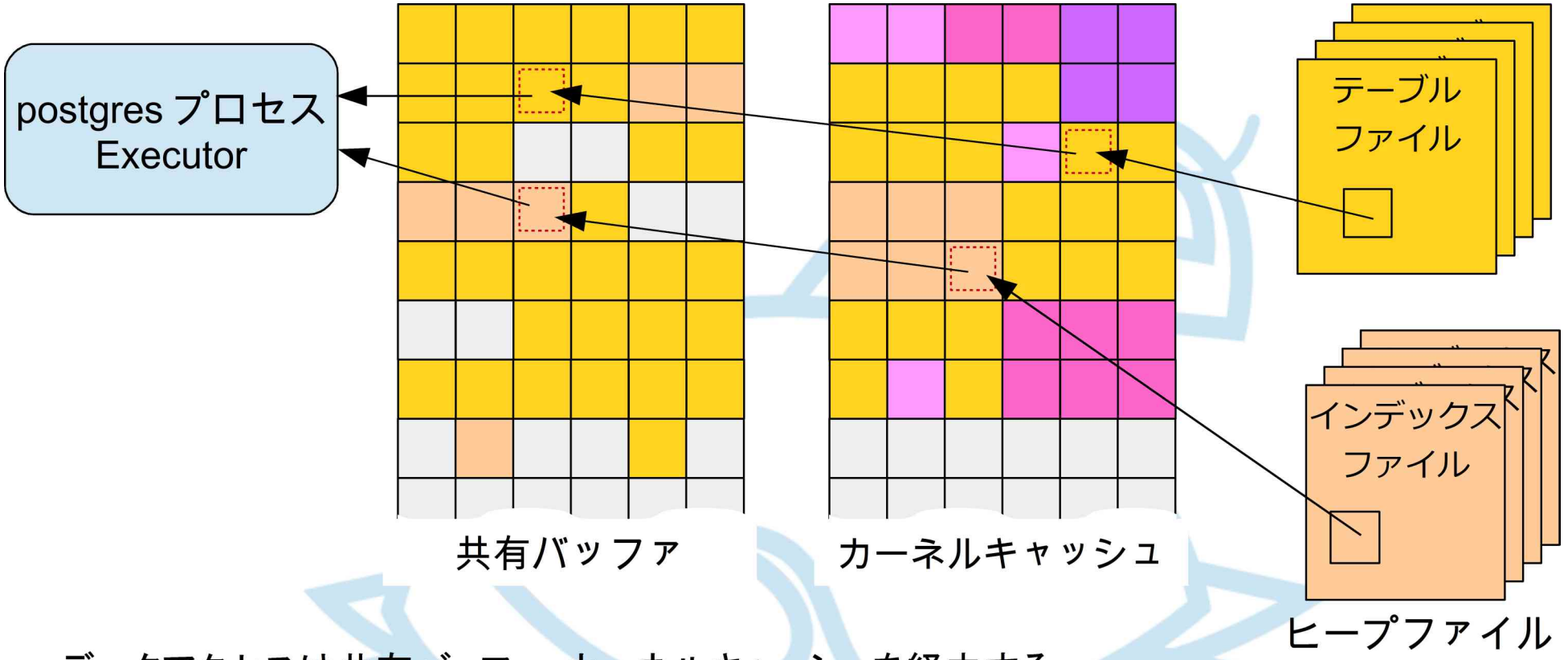


SQL 実行の仕組み - プランナと Executor



PostgreSQL
http://www.postgresql.jp

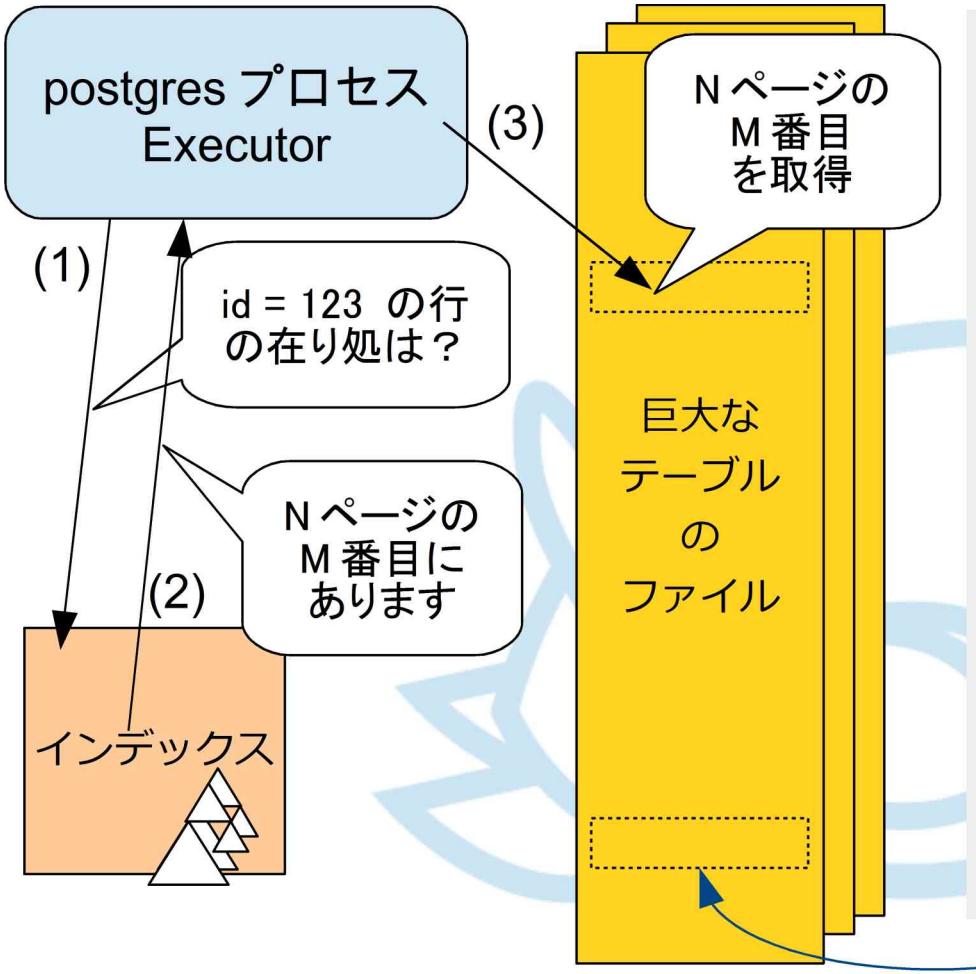
SQL 実行の仕組み - データ読み取り

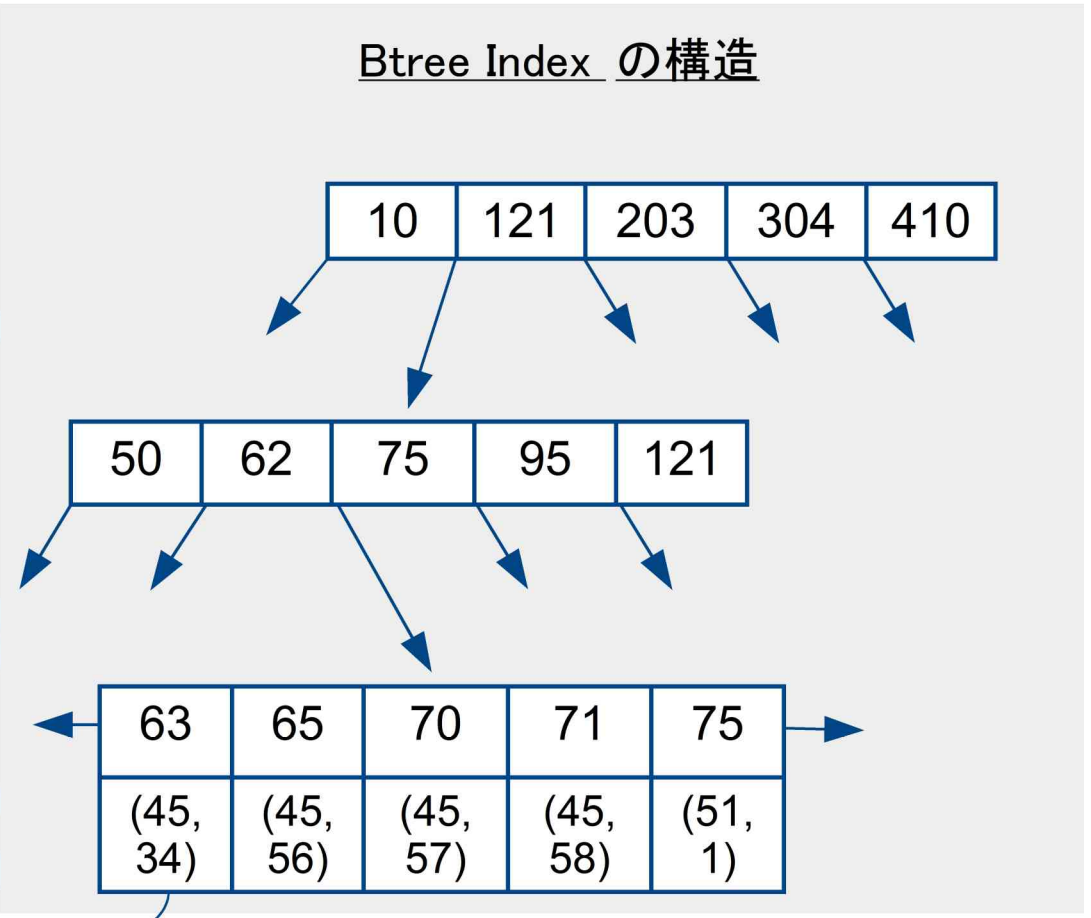


- データアクセスは共有バッファ、カーネルキャッシュを経由する
- おおむね、空きメモリ＝カーネルキャッシュとなる
- 共有バッファ(shared_buffers)が小さくても、カーネルキャッシュにヒットするなら参照についてはストレージ I/O を減らすことができる

インデックスの利用

Index Scan



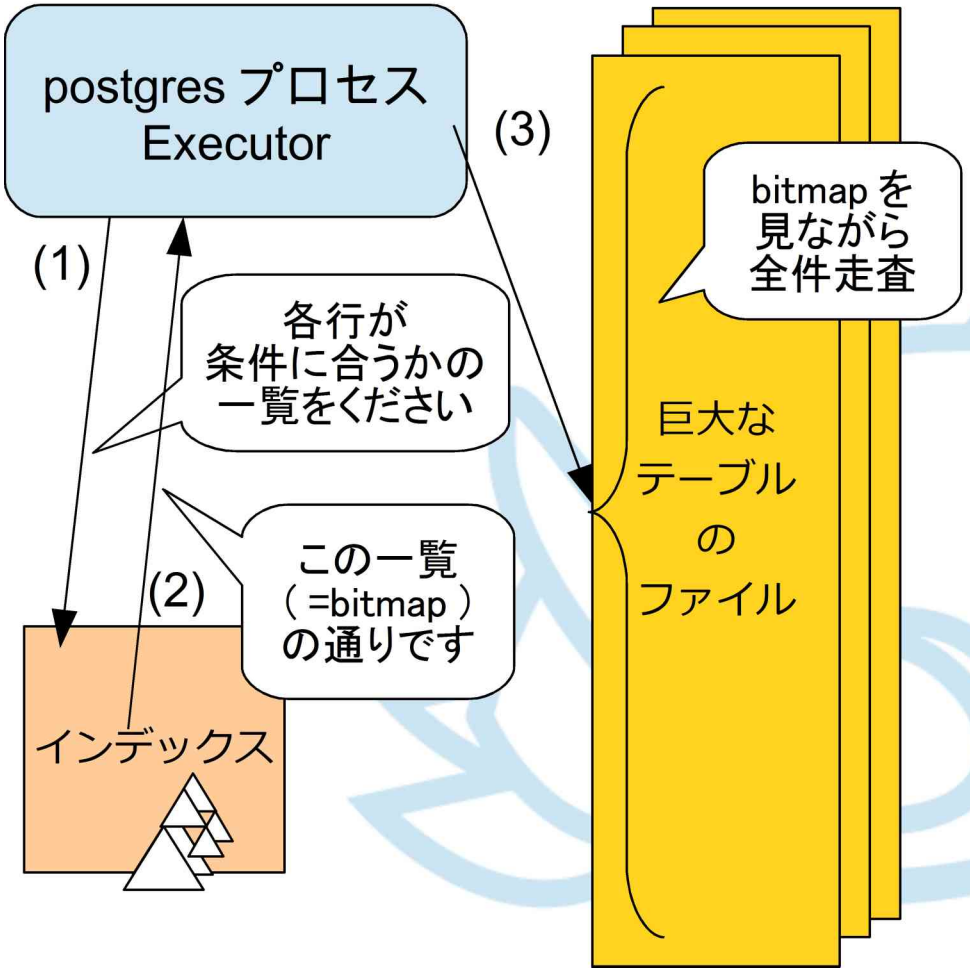


- インデックスで条件にあうデータの在り処を調べることができる
- インデックスを使わないなら、テーブル全件走査しかない

PostgreSQL
http://www.postgresql.jp

インデックスの利用

Bitmap Index Scan + Bitmap Heap Scan



インデックスの種類

- Btree
  - 通常これを使う
  - 一致条件、大小比較、最大最小
- Hash
  - 長い文字列データに使う
  - 一致条件
- GIN
  - 配列や JSON 、テキスト検索にて「～を含む」の条件
- GiST / SP-GiST
  - テキスト検索、範囲データ、幾何データの「重なり」の条件など
- BRIN
  - 履歴データの日付や連番に

実行プランを調べる

```
db1=# explain (analyze, buffers)
      SELECT * FROM t1 JOIN t3 ON (t1.id = t3.id1) WHERE t3.ts < '2017-11-24 9:00;

                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Nested Loop  (cost=0.29..578.65 rows=341 width=89)
              (actual time=0.063..3.781 rows=340 loops=1)
   Buffers: shared hit=1028
   -> Seq Scan on t3  (cost=0.00..19.50 rows=341 width=52)
                      (actual time=0.027..0.579 rows=340 loops=1)
         Filter: (ts < '2017-11-24 09:00:00'::timestamp without time zone)
         Rows Removed by Filter: 660
         Buffers: shared hit=7
   -> Index Scan using t1_pkey on t1  (cost=0.29..1.64 rows=1 width=37)
                                      (actual time=0.007..0.007 rows=1 loops=340)
         Index Cond: (id = t3.id1)
         Buffers: shared hit=1021
 Planning time: 0.468 ms
 Execution time: 4.077 ms
(11 rows)
```

プラン要素

見積のコストと行数、
実際の所要時間と行数

バッファヒット

0.007ms x 340 回

PostgreSQL
http://www.postgresql.jp

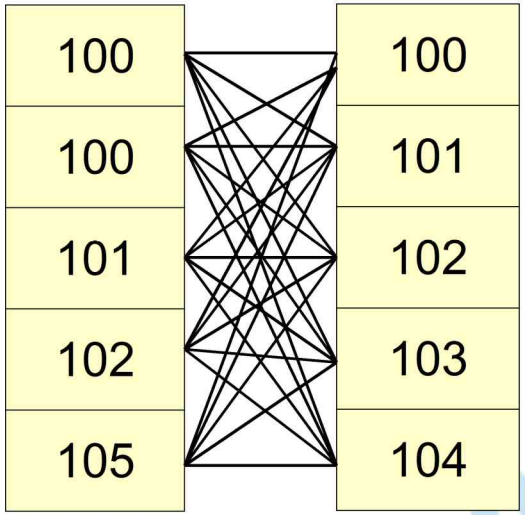
同じ目的に
複数の手段が
あるケースに注意

主な計画ノード

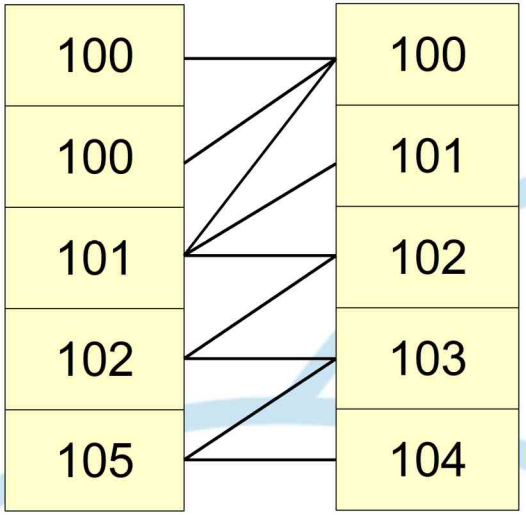
計画ノード名	動作
Seq Scan	テーブルの 全件読み出し
Index Scan	インデックスを 使った読み取り
Index Only Scan	インデックスだけ 読み取る
Bitmap Index Scan / Bitmap Heap Scan	インデックスから ビットマップを作っ たうえで、テーブル から読み出し
Nested Loop	二重ループによる テーブル結合
Hash Join / Hash	ハッシュを使う テーブル結合
Merge Join	ソート済データを 使うテーブル結合

計画ノード名	動作
Sort	ORDER BY 等に対応
Aggregate	集約関数に対応
WindowAgg	ウィンドウ関数に対応
Hash Aggregate / Hash	GROUP BY に対応、 ハッシュを使用する方式
Group Aggregate / Sort	GROUP BY に対応、 ソート済データを使う方式
Append	UNION ALL やパーティ ションテーブル参照に対応
Unique	DISTINCT 等に対応
Limit	LIMIT に対応
Materialize	結果を憶えて再利用する WITH 句などで

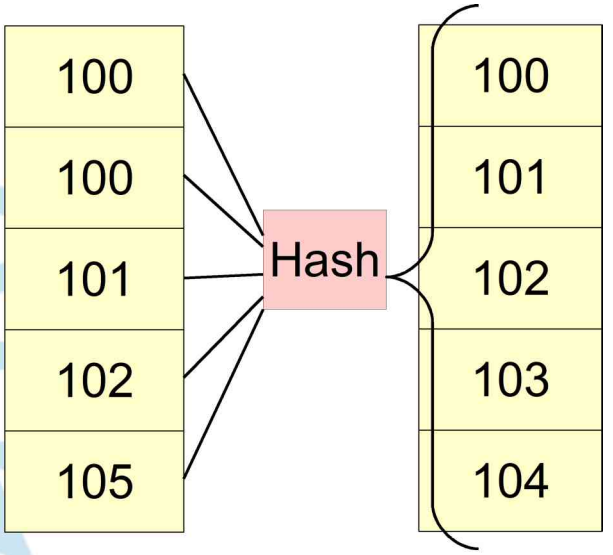
結合方式



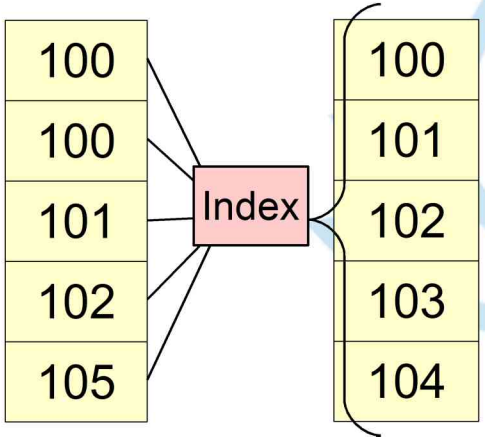
Nested Loop



Merge Join



Hash Join

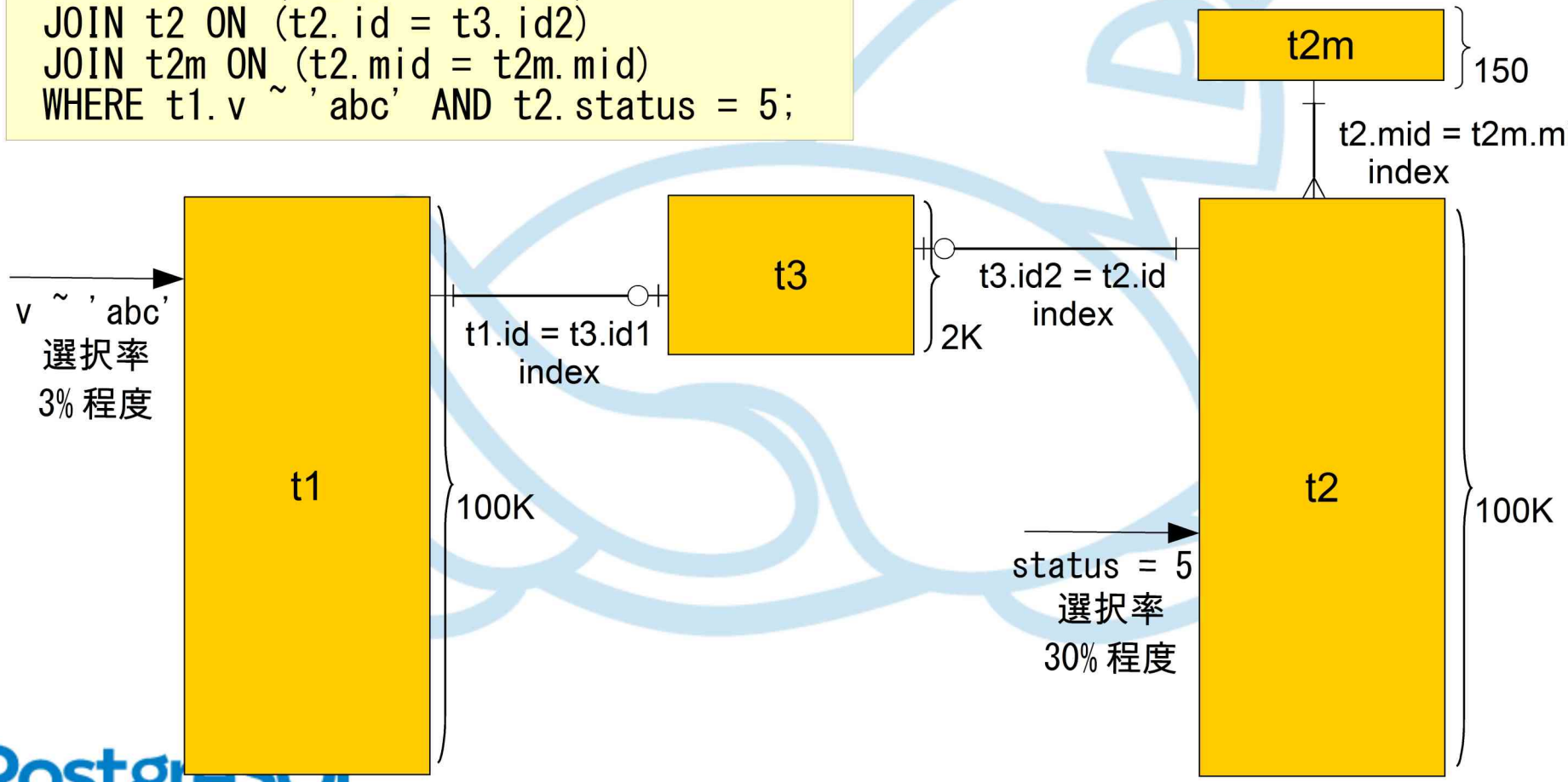


PostgreSQL
http://www.postgresql.jp

現状が悪いプランなのか？

- SQL に対して人力で実行プランを考えて検討する必要がある

```
SELECT t1.v, t2m.v, t2.v, t3.ts FROM t1
 JOIN t3 ON (t1.id = t3.id1)
 JOIN t2 ON (t2.id = t3.id2)
 JOIN t2m ON (t2.mid = t2m.mid)
 WHERE t1.v ~ 'abc' AND t2.status = 5;
```



プランナ統計情報

- プランナ統計情報の採取状況
  - 各テーブルの VACUUM/ANALYZE の最終実行タイムスタンプ確認

```
SELECT schemaname, relname,
  last_vacuum, last_autovacuum,
  last_analyze, last_autoanalyze FROM pg_stat_all_tables;
```

  - 統計情報の内容確認

```
SELECT relname, reltuples, relpages FROM pg_class;

SELECT * FROM pg_stats;
```

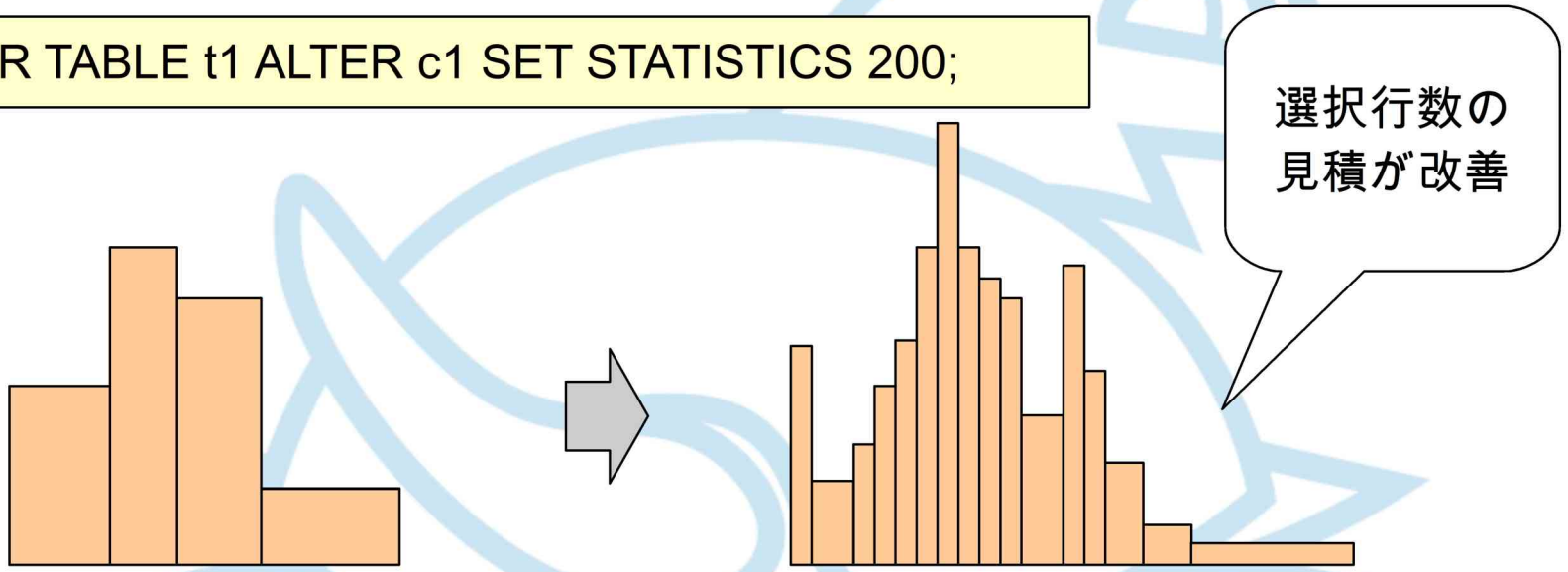
- VACUUM でテーブル件数と物理サイズが採取
- ANALYZE で各列値の値の分布、NULL 率、物理配置と値の相関、値の平均長、がサンプル採取
  - 大小比較が効かない値は分布の統計が取れない

プランナ統計情報の粒度を上げる

- ヒストグラムと最頻出現値の粒度が調整可能

```
default_statistics_target = 200   #  default 100
```

```
ALTER TABLE t1 ALTER c1 SET STATISTICS 200;
```



Value:	100	50	0	15
Retio:	0.10	0.05	0.03	0.03

Value:	100	50	0	15	20	70	65	90
Retio:	0.10	0.05	0.03	0.03	0.02	0.02	0.01	0.01

プランナに指示する設定

- パラメータを postgresql.conf や SET コマンドで指定

パラメータ	使い方
enable_bitmapscan　enable_hashagg enable_hashjoin　enable_indexscan enable_indexonlyscan　enable_material enable_mergejoin　enable_nestloop enable_seqscan　enable_sort	・デフォルトは全て on ・ SET コマンドで一時的に off にして、選択されるプランを限定する。
seq_page_cost = 1.0 random_page_cost = 4.0 → 2.0 or 1.0 cpu_tuple_cost = 0.01 cpu_index_tuple_cost = 0.005 cpu_operator_cost = 0.0025	・プランナのコスト計算基準値を指定 ・速い RAID 装置や SSD なら、random_page_cost は減らす ・全体的なインデックス選好傾向付与に
effective_cache_size = 4GB	・共有バッファサイズ ＋ カーネルキャッシュとして使われそうなサイズ を指定 ・インデックス利用選択時の参考情報
cursor_tuple_fraction = 0.1	・カーソルの場合に先頭行応答を優先 ・ 1.0 以外の場合、カーソルではプランが変わることに注意

SQL を書き換える

- 指定の結合順序で実行させる
  - 以下パラメータを与え、先に結合させたい箇所を括弧でくくったり、サブクエリ化したりする

パラメータ	使い方
from_collapse_limit = 8 → 1 join_collapse_limit = 8 → 1	・プランナがテーブルの結合順序を決める上限 ・１にするとプランナは何もせず、SQL に書いたとおりの順になる

- 様々な工夫
  - WHERE 条件の インデックスが使われる書き方 ⇔ 使われない書き方
  - 検索条件をサブクエリ内に明示的に重複して記載してみる
  - WITH句 ⇔ サブクエリ で選択プランの傾向が違う

… WHERE c1 + 1 = 100 + 1

パラレル動作

- PostgreSQL 9.6 ～ 11 で各種処理がパラレル動作できるようになった
  - パラレルの推奨と抑止を試行して比較する

パラメータ	使い方
max_worker_processes max_parallel_workers_per_gather max_parallel_workers	パラレル処理をするとして、 最大何並列するかの設定
parallel_tuple_cost parallel_setup_cost	パラレル処理のコスト値。 減らしてパラレル処理をさせる、 増やしてパラレル処理を抑止する
min_parallel_table_scan_size 、 min_parallel_index_scan_size	パラレル処理対象とするテーブル、 インデックスの最小サイズ

一時ファイルの使用

- ソート処理、ハッシュ作成処理で発生
  - ストレージ I/O が生じて遅延の要因となる
- 判別方法
  - $PGDATA/base/pgsql_tmp/ 以下にファイルが作られる
  - log_temp_files = '1MB'
  - EXPLAIN (analyze)
  - SELECT datname, temp_files FROM pg_stat_database;
- 回避方法
  - work_mem を大きくする

ソートについては、以下パラメータの調整も試行価値あり:

replacement_sort_tuples = 150000

効果有無はデータ物理配置に依存
(PG11 で廃止のパラメータ)

ロック待ちの可能性

- 実はロック待ちで遅いのかもしれない
- 判別方法
  - log_lock_wait = on 、 deadlock_timeout = 10s
    → 10 秒以上のロック待ちをログ報告
  - SELECT * FROM pg_stat_activity;
    → 今現在のロック待ち状態
    - state = 'active' で、wait_event_type と wait_event に何か値がある
    - (PostgreSQL 9.5.x 以前なら) waiting = 't'

拡張モジュール

- pg_hint_plan
  - SQL にヒント句が使える
- pg_dbms_stats
  - プランナ統計情報を固定化できる

- パブリッククラウドのデータベース AS サービスでは、運営者がその拡張モジュールを採用してくれないと使えない
- Windows で使うにはハードルが高い

まとめ

- チューニングに取り掛かる前に、
  SQL実行の仕組みを想像できるようになりましょう
- EXPLAIN 出力、ログ、システムテーブル、OS から
  情報を集めましょう
- 各種の手段を試行しましょう