PGECons 勉強会#2
パラレルクエリ

2018/10/18

NECソリューションイノベータ(株)
近藤　太樹

Agenda

- パラレルクエリとは？
- パラレルクエリはいつ使う？
- パラレルクエリには何が必要？
- パラレルクエリの落とし穴！？
  - リソースが足りない場合
  - 大きいテーブルどうしの結合
  - パラレルクエリを使用したら思わぬ伏兵が!?
- 皆さんからの質問にお答えします!

パラレルクエリの概要

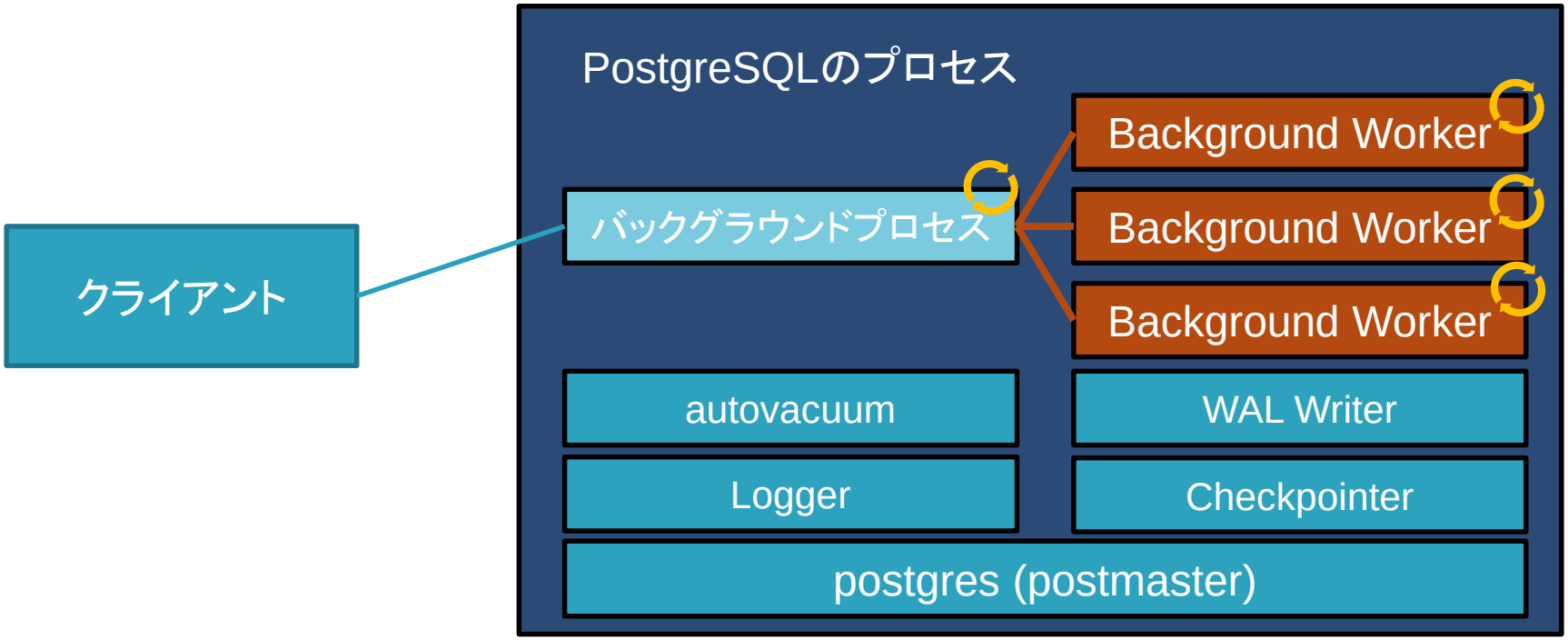
パラレルクエリとは？

PostgreSQLでのパラレルクエリとは？

- 1つのクエリを
  複数のプロセスで分担してパラレル処理すること
  - パラレルクエリで効率的に処理可能と
    判断した場合のみ採用
  - PostgreSQLでは、パラレル処理においては
    スレッドは使用していない

PostgreSQLでのパラレルクエリとは？

- PostgreSQL 9.6で採用
- PostgreSQL 10以降も継続して強化中
- PostgreSQLのパラレルクエリは
  Background Worker Processの一種として実装



PostgreSQLにおけるパラレルクエリの歴史

- パラレルクエリ導入までの道のりは長く、
  基礎作りはPostgreSQL 9.3の頃（2012～2013）から
  - 9.3：Background Worker Process 導入
  - 9.4：Background Worker Processの動的開始・終了
    - max_worker_processesパラメータの導入
  - 9.5：「並列実行基盤を改善しました。」
  - 9.6：祝 パラレルクエリ導入
  - 10：Gatherノードの拡張により、部分集約が可能に
  - 11：SELECT文以外のSQL文でのパラレル対応（後述）

PostgreSQLで対応するパラレルクエリ

- 全てのクエリがパラレルに対応しているわけではない
- パラレル対応しているのは以下の操作
  - PostgreSQL 9.6～
    - 逐次スキャン (SeqScan) 、集約 (Aggregation) 処理、ハッシュ結合、入れ子ループ (Nested Loop) 結合
  - PostgreSQL 10～
    - マージ結合 (Join) 処理、B木 (B-tree) インデックススキャン、集約 (Aggregation) 処理の対応範囲拡大
  - PostgreSQL 11～
    - B木インデックスの作成、ハッシュ結合 (ハッシュテーブルの共有化)

効果が高いクエリについて

パラレルクエリはいつ使う？

パラレルクエリはいつ使う？

- データベースでの処理量の多い、応答に長時間かかるクエリが、パラレルクエリに対応しているとき
  - 大量のデータが登録されているテーブルの検索
  - 大量のデータの集計処理

- このようなクエリをパラレル化することで応答時間の短縮が期待可能
  - BI・DWHなどのOLAP用途でのクエリは、このようなクエリの典型例
  - 同様の理由から、バッチ処理でも効果が期待可能（リソースがあれば）

パラレルクエリはいつ使う？

- パラレルクエリに対応しているSQL文は以下の通り
  - 基本的にはSELECT文
  - 以下は、v11より対応
    - CREATE TABLE … AS SELECT …
    - SELECT INTO …
    - CREATE MATERIALIZED VIEW
    - CREATE INDEX（B木のみ）

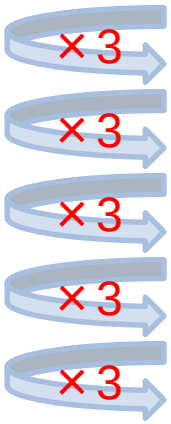
パラレルクエリはいつ使う？

- パラレルクエリを使用できない場合もあるので注意
  - UPDATEやINSERTなどの更新系
  - ロックを取得するようなクエリ
    - SELECT FOR UPDATE
  - 一時テーブルでの検索
    - CREATE TEMPORARY TABLE で作成されたもの
  - 通常のSELECTでも、場合によっては使用されない
    - ただし、かなり複雑なクエリにならないと、陥らない

- 詳細はマニュアルでご確認ください。
  - https://www.postgresql.org/docs/current/static/parallel-safety.html（英語）
  - https://www.postgresql.jp/document/10/html/when-can-parallel-query-be-used.html（v10ベース 日本語訳）

パラレルクエリはいつ使う？

- 大量のデータとはどのくらいか？
  - パラレル度はテーブルのサイズに依存



ブロック数	サイズ(MB)	起動Worker数	検索プロセス数
1,024	8	1	2
3,072	24	2	3
9,216	72	3	4
27,648	216	4	5
82,944	648	5	6
248,832	1,944	6	7
…	…	…	…

- パラレルでない場合の実行計画よりも
  コスト推定値が下回った場合に初めて使用される

必要となるリソース

パラレルクエリには何が必要？

適切なハードウェアリソースが必要

- パラレルクエリを有効活用するには、
  それに応じたハードウェアリソースが必要
  - パラレル実行する（予定の）プロセス数以上の
    物理CPUコアを用意する。
  - パラレル実行する（予定の）テーブルサイズ以上の
    メモリを用意する。

適切なパラメータの投入が必要

- パラレルクエリ関連の設定が行われていること
  - max_worker_processes, max_parallel_workers_per_gather, max_parallel_workers は **全て1以上である必要がある（v9.6は注意）**

パラメータ	デフォルト値 (v9.6)	デフォルト値 (v10)	解説
max_worker_processes	8	8	BgWorkerの最大数
max_parallel_workers_per_gather	0	2	Worker最大数/Gather
max_parallel_workers	—	8	Worker最大数/クエリ
min_parallel_relation_size	8MB	—	パラレル発動最小サイズ(table)
min_parallel_table_scan_size	—	8MB	パラレル発動最小サイズ(table)
min_parallel_index_scan_size	—	512kB	パラレル発動最小サイズ(index)
parallel_setup_cost	1000	1000	コスト推定値/Worker
parallel_tuple_cost	0.1	0.1	コスト推定値/Tuple

この3項目だけは、
最低限 確認が必要

パラレルクエリ使用の際の考慮点

- パラレルクエリ関連のパラメータでは以下に注意
  - max_worker_processes は一種のハードリミットなので使用可能な物理コア数以下に設定
  - max_parallel_workers_per_gather, max_parallel_workers はソフトリミットに近い存在で**チューニングが必要**
    - max_parallel_workersはクエリ実行につき適用されるので、複数クライアントからクエリを実行されると**全体の合計値はこの設定値を超える**
    - max_parallel_workers_per_gatherはGatherノード毎に適用される
      - **1クエリ中に複数のGatherノードが現れる場合がありうる**ので要注意。

パラレルクエリ使用の際の考慮点

- クエリがどの程度のコアを使用するのか？
  - テーブルサイズで（ほぼ）確定する。
  - 正確を期するなら、実行計画を表示する。
    EXPLAIN SELECT …
    - Gatherノードの表示を確認

```
->  Gather  (cost=4254.90..166694.20 rows=9700 width=8)
      Workers Planned: 5
```

上記の場合は、このGatherノードに対して、
Background Workerが5プロセス割り当てられる

- ただし、**パラメータ設定値でクリッピングされる**ので注意

パラレルクエリ使用の際の考慮点

- 適切なメモリを見積もるために
  - psql から以下のようにすれば
    テーブルサイズの概算値が取得可能

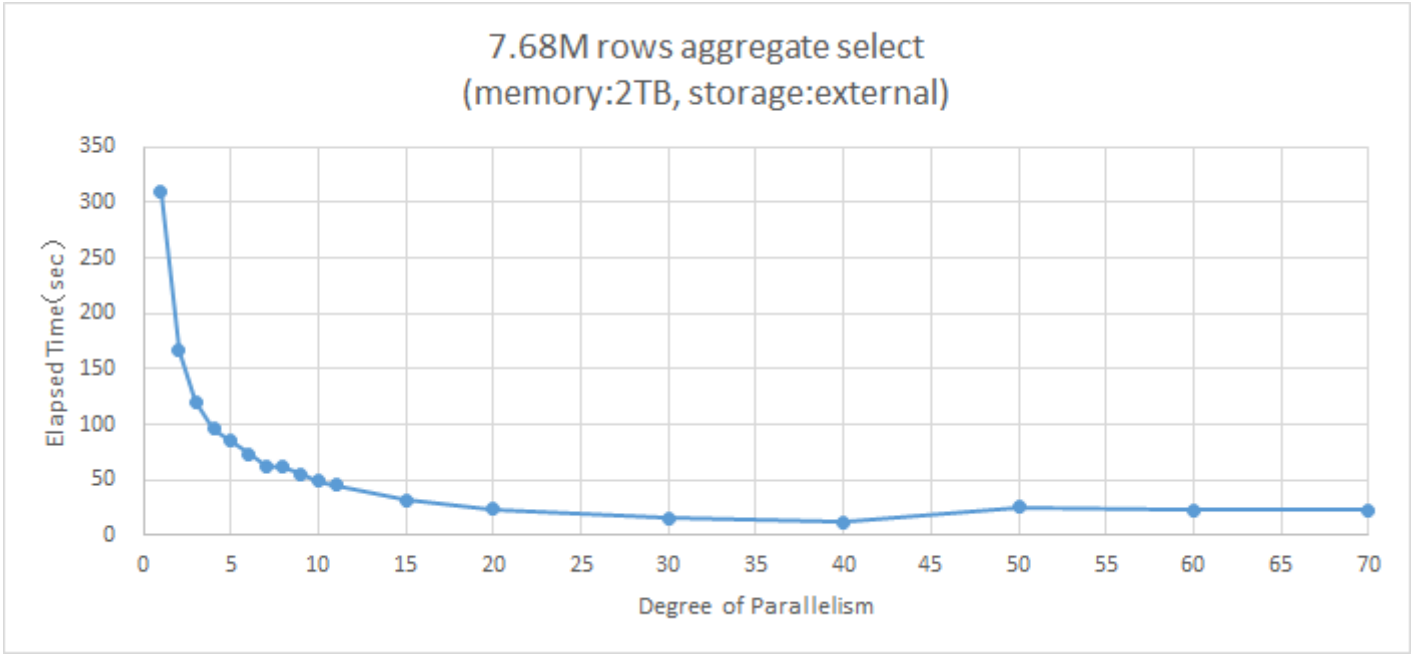
```
=# ¥d+
                                List of relations
 Schema |          Name           | Type  | Owner |    Size    | Description
--------+-------------------------+-------+-------+------------+-------------
 public | call_center             | table | kondo |  8192 bytes |
 public | catalog_page            | table | kondo |  2264 kB    |
 public | catalog_returns         | table | kondo |  65 MB      |
 public | catalog_sales           | table | kondo |  799 MB     |
 public | customer                | table | kondo |  38 MB      |
 public | customer_address        | table | kondo |  13 MB      |
 public | customer_demographics   | table | kondo |  214 MB     |
 public | date_dim                | table | kondo |  15 MB      |
```

パラレルクエリ使用の際の考慮点

- パラレルクエリを使用する際は、
  テーブルのデータがメモリに載っているかが肝心
  - PostgreSQLはOSのディスクキャッシュを活用するが…
  - PostgreSQLの共有バッファに全データを載せることも
    検討する価値が十分にある
    - shared_buffersをテーブルサイズ以上に設定する
    - Contributionパッケージ内のpg_prewarmを使用すれば、
      全データを簡単に共有バッファに載せることが可能

リソースがあれば

- 理論に近い高速化が可能
  - 以下の例では、40並列付近まで高速化が認められる



- 2016年度活動報告書より抜粋
- 物理マシンで測定
- 1テーブルのパラレル検索・集約
- テーブルサイズは230GB程度

利用する際に注意すべき点

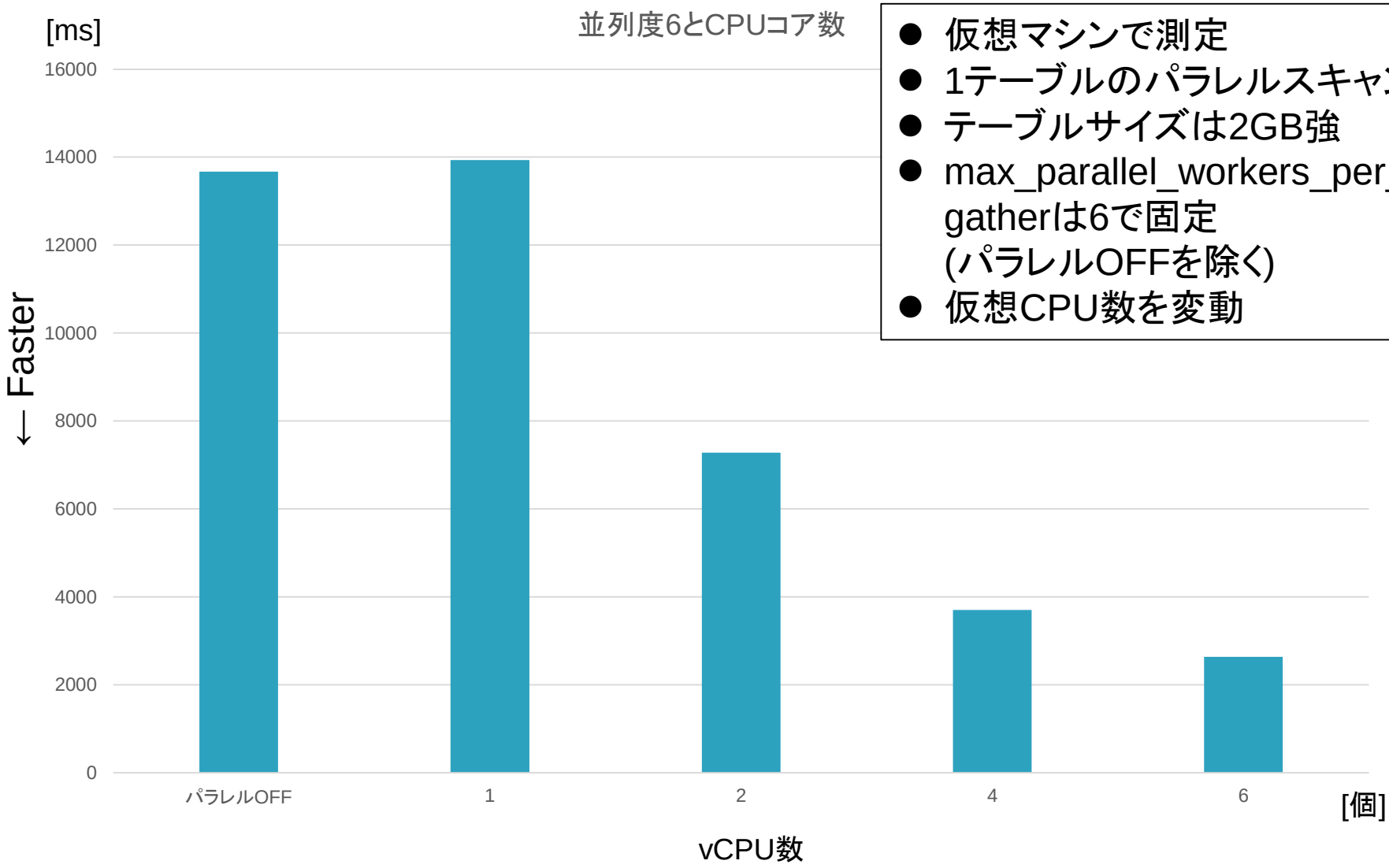
パラレルクエリの落とし穴！？

① リソースが足りない場合

リソースは絶対に確保すること！

- パラレルのプロセス数以上のCPUコアを確保！
  - PostgreSQLはマシンのCPUコア数は見ていない
  - 複数プロセスに処理を割り当てて実行する
    - 一見動いているように見えるがパフォーマンス悪化
    - 複数のパラレルクエリが同時実行される場合も同様
- テーブルサイズ以上のメモリを確保！
  - 共有バッファが足りなければ、データは追い出される
    - OSのキャッシュメモリも同様
  - そして、ディスクにアクセスが集中する
    - I/Oネック発生でパフォーマンス悪化
    - CPU使用率が高くならない

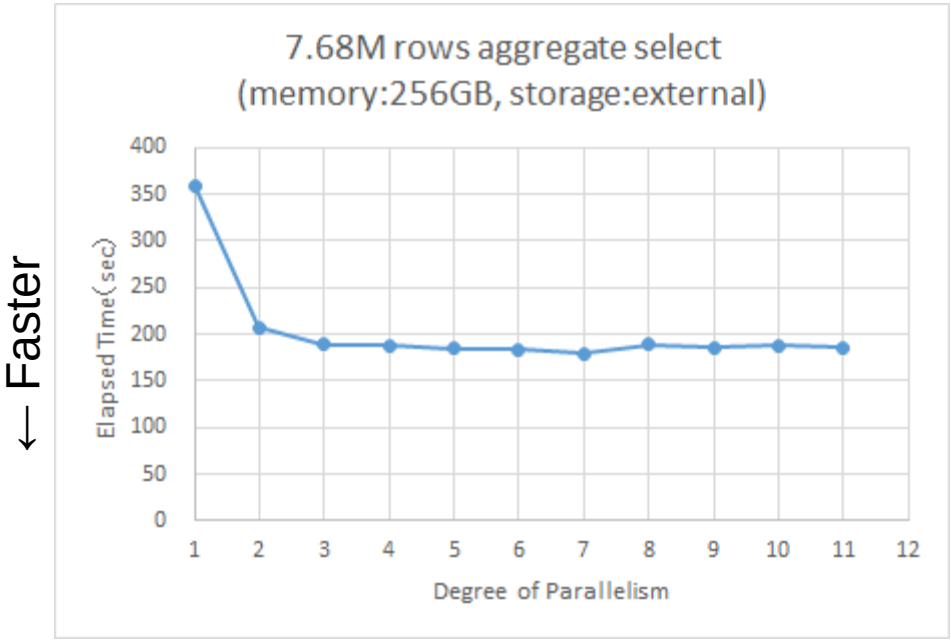
コア数以上のプロセス数になると･･･



- 仮想マシンで測定
- 1テーブルのパラレルスキャン
- テーブルサイズは2GB強
- max_parallel_workers_per_gatherは6で固定
  (パラレルOFFを除く)
- 仮想CPU数を変動

I/Oネックに引っかかると…

- 性能が頭打ちになる
  - 下記の例では、2並列の時点で性能が頭打ちに

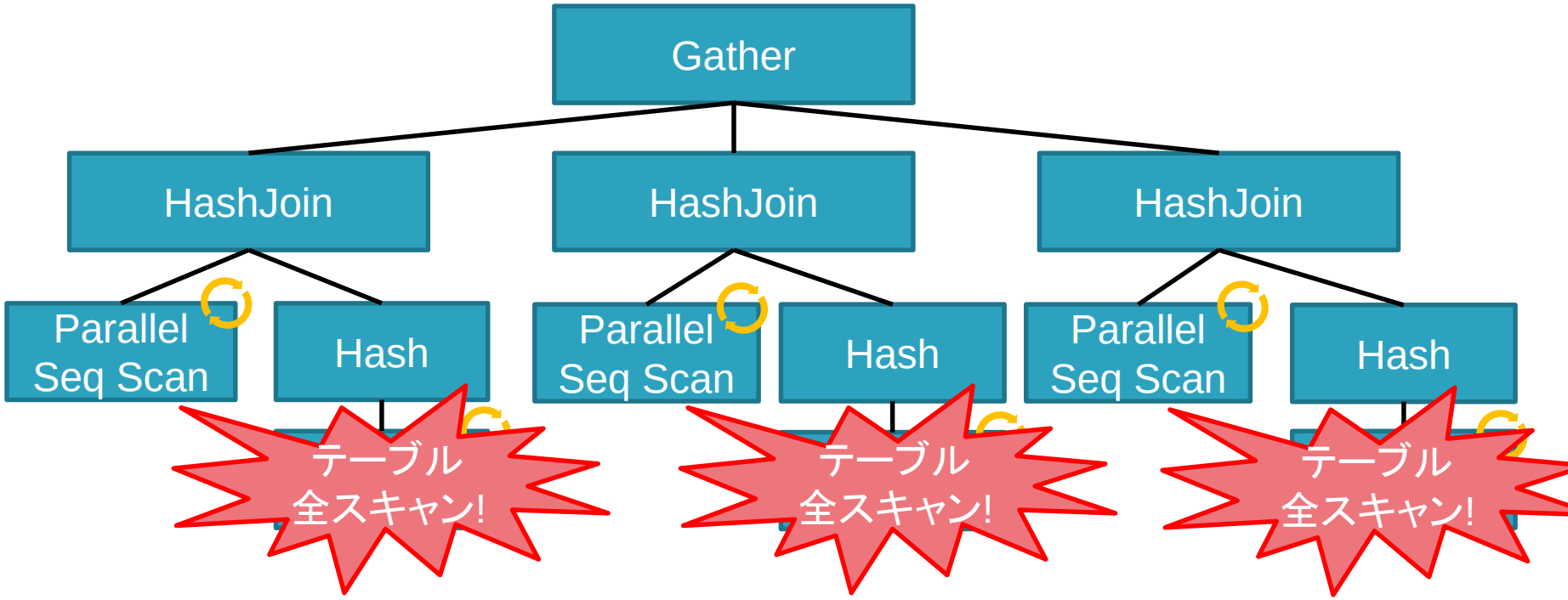


- 2016年度活動報告書より抜粋
- 物理マシンで測定
- 1テーブルのパラレル検索・集約
- テーブルサイズは230GB程度

② 大きいテーブルどうしの結合

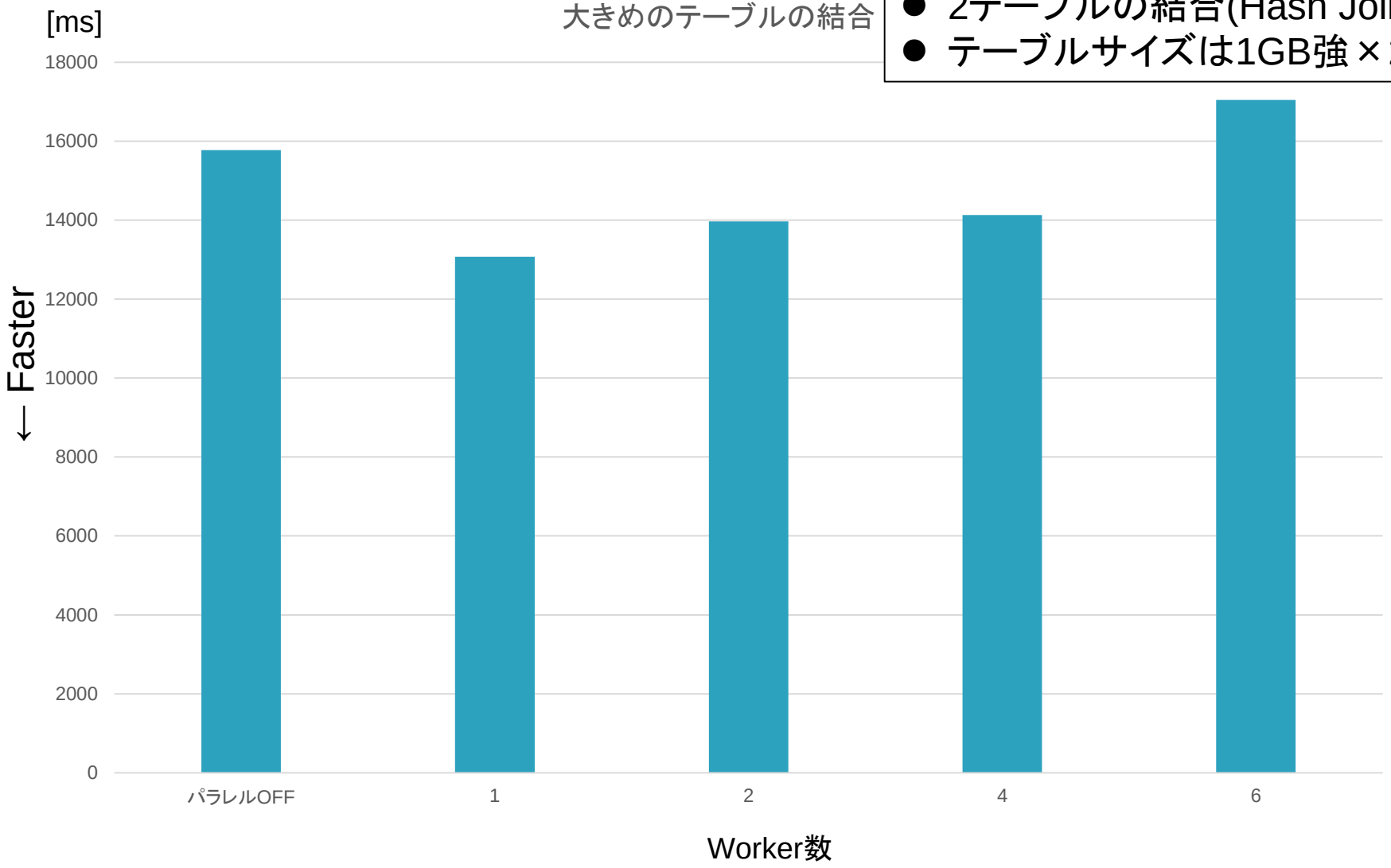
大きいテーブルどうしの結合は避ける

- テーブルの結合にもパラレルクエリは使用される
- しかし、結合対象の2テーブルのうち、
  片側にしかパラレルクエリが用いられない。
  - もう片方のテーブルは必ず全スキャン



大きいテーブルどうしの結合をすると…

- 2テーブルの結合(Hash Join)
- テーブルサイズは1GB強×2



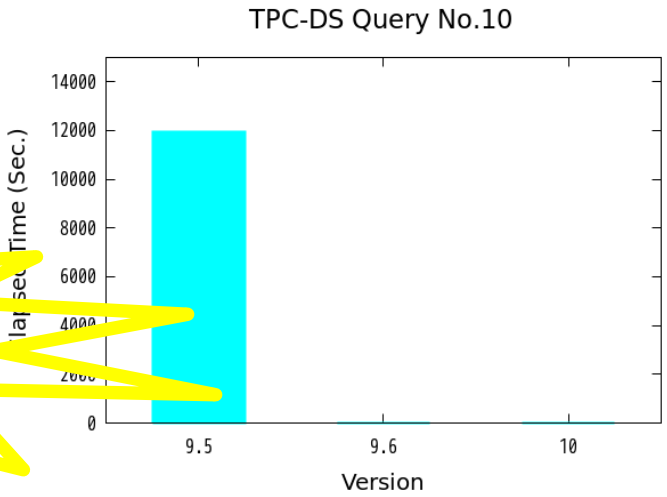
③ パラレルクエリを使用したら思わぬ伏兵が!?

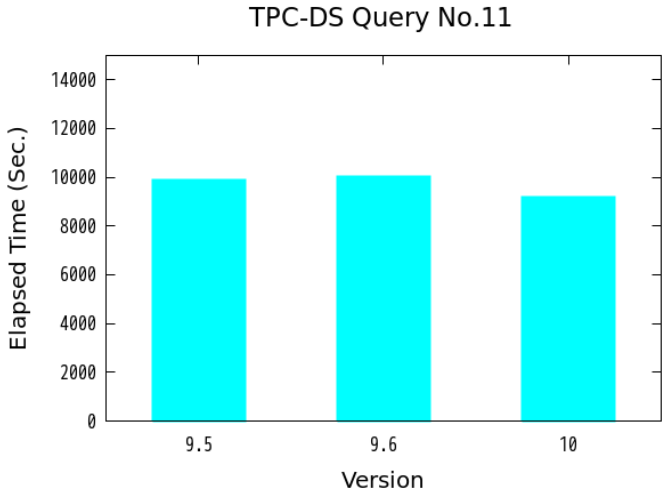
『コスト推定値が…』ということは

- パラレルクエリが選択されるのは…
  - パラレルでない場合の実行計画よりもコスト推定値が下回った場合
  - 同じクエリでも**コスト推定値が異なる**ということを示唆
    - パラレルクエリを選択した部分のコスト値が低下する
  - すると…

検証A パラレルクエリでの処理性能の改善状況

- No.10
  - パラレルクエリの有効化によって劇的（2000倍以上）に高速化
    - ただし、並列処理で速くなったのではなく、実行計画のうち、実際に実行された部分が変化した結果、高速化。
- No.11
  - v9.6ではパラレルクエリを選択せず
  - v10はパラレルクエリを選択し、8%程度の高速化
    - 「非相関サブクエリのパラレル実行」（v10で新規対応）に該当したか？
- パラレルクエリ無効の状態でも、v10がv9.5比で、両クエリともに20%程度高速化していることが判明



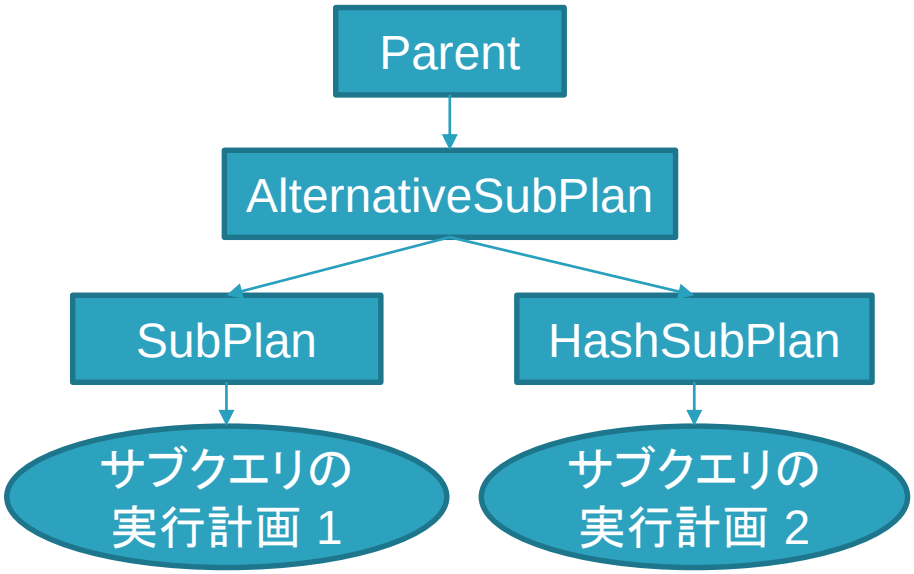


パラレルクエリでNo.10に何が起こったのか？

- 実行計画のうち、実際に実行された部分が変化したのが主要因
  - この変化はパラレルクエリが絡んでいるのか？
    → 結果的には絡んでいた
  - 実行計画に表示されているalternativesの部分が鍵

    (alternatives: SubPlan 1 or hashed SubPlan 2)
  - alternatives とは…
    - exists句の処理を高速化するため、続けて現れるサブクエリの結果を普通に得るか、ハッシュテーブル化するかを実行時に判断
      - ハッシュテーブル化して得るとサブクエリの結果を保持するため大幅に高速化する場合がある
    - Executorのコードの一部である、nodeSubplan.c の ExecInitAlternativeSubPlan関数で判断。
    - なんと、9年前 (v8.4) からあるロジック。(Commit ID : bd3dadda)

alternatives の判定ロジック



- 同一サブクエリに対する実行計画を2種類持つ
- 実行時にExecutorが判断する
  - EXPLAIN ANALYZE時、実行されなかった方は、必ずNever Executed

判定ロジック

SubPlanとHashSubPlanの初期コストをそれぞれCs_1, Cs_2

SubPlanとHashSubPlanの1実行あたりのコストをそれぞれCp_1, Cp_2

Parentにおける出力行数予測値をR、

SubPlanとHashSubPlanの初期コストをそれぞれC_1, C_2とすると、

$$C_1 = Cs_1 + R \times Cp_1\,, \qquad C_2 = Cs_2 + R \times Cp_2$$

$$C_1 < C_2 \rightarrow Do\ SubPlan$$

$$(C_1 \geq C_2 \rightarrow Do\ HashSubPlan)$$

パラレルクエリでNo.10に何が起こったのか？

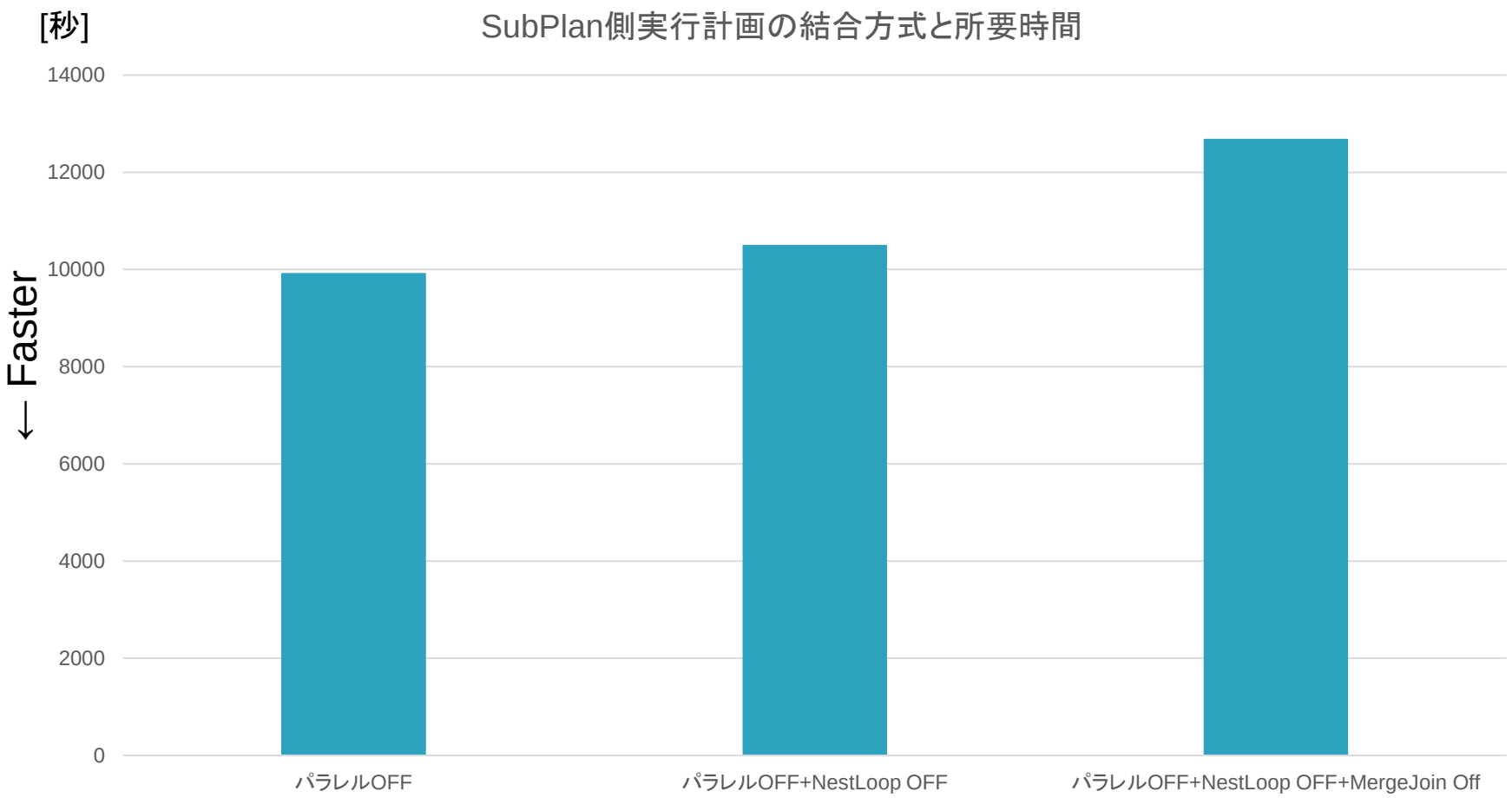
- ずばり、HashSubPlan側のコスト推定値の減少
  - このクエリでは、HashSubPlan側のみ
    パラレルクエリを選択
  - 結果、SubPlan側のコスト推定値以下となり、
    HashSubPlan側を実行

→ では、SubPlan側の実行計画が、
HashSubPlan側と同様となるようすれば、
パラレルクエリなしでも速くなるのでは？

同様：Nested Loop結合ではなく、ハッシュ結合を選択させる

結合方法の制限結果

- 結論：変わらない、または、遅い



パラレルクエリでNo.10に何が起こったのか？

- 2000倍も高速化したのは、“Hash”SubPlanだったことが主要因であった可能性が高い
  - SubPlanでは、結果を保持しておらず、参照されるたびにサブクエリが再実行された
  - HashSubPlanでは、結果をハッシュテーブル化して保持していたため、サブクエリが再実行されることはなかった

まとめ

まとめ

- パラレルクエリは大量のデータを持つテーブルの検索・集約における**応答時間の短縮**が期待可能
- パラレルクエリを有効活用するには**適切なハードウェアリソースとパラメータ**が必要
- **コスト推定値が従来と異なることによる影響**に注意
  - パラレルクエリ実行により、実行計画の取得だけでは判明しない差異が存在
    - EXPLAINではわからない。EXPLAIN ANALYZEが必要。
- パラレルクエリの効果が十分に得られるか、**要確認**
    - パラレルクエリ実行の処理時間と、オーバヘッドやパラレルクエリ部分以外の処理時間の割合を算出する、など

PostgreSQL Enterprise Consortium

POSTGRESQL ENTERPRISE CONSORTIUM