PGECons
PostgreSQL Enterprise Consortium

「shared_buffersの適正値」を探る旅

2022年11月11日
PostgreSQL Conference Japan 2022 (B3)
PostgreSQLエンタープライズ・コンソーシアム
ヤマトシステム株式会社(ヤマト運輸所属) 毛呂 良寛

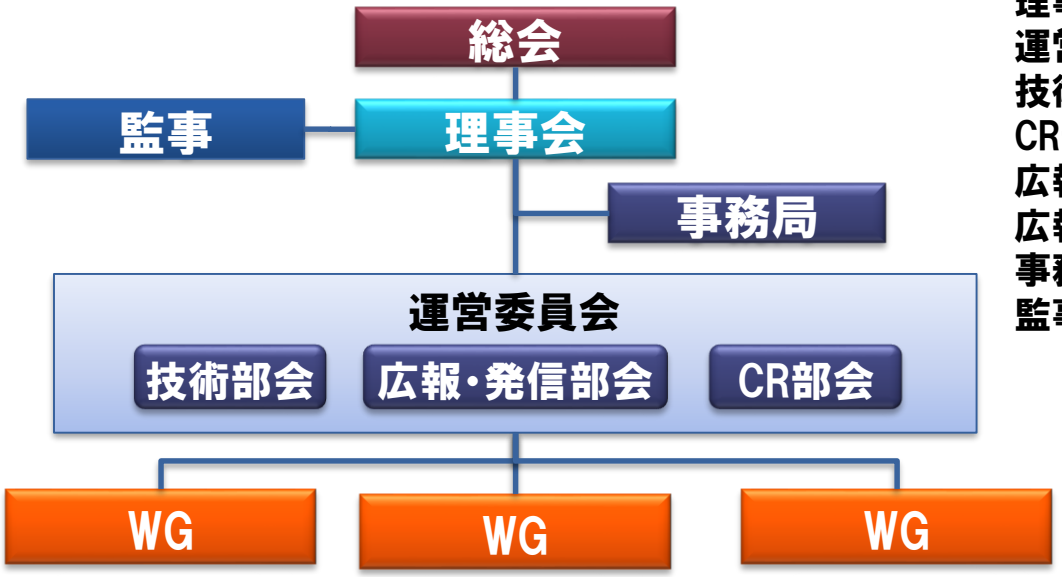
PGECons について

- PGEConsの発足と目的
  - 2012年4月11日発足
  - ミッションクリティカル性の高いエンタープライズ領域へのPostgreSQLの普及を推進するため、各種ツールやPostgreSQL本体に関する利用技術情報の収集と提供および、その整備などの活動を企業ベースで展開する

活動項目	概要
情報発信	会員の導入実績を基に、PostgreSQLおよび周辺ツールに関する情報を集約し、情報発信サイトやセミナ等を通じて提供する
共同検証	エンタープライズ領域への適用に向けて必要となる情報を、実証を通じて充実を図る
開発コミュニティへのフィードバック	よりミッションクリティカル性の高い領域への適用に向けた技術的な課題を集約し、開発コミュニティに要望を発信する
開発プロジェクト支援	会員間での機能拡張に関する連携開発や、必要な周辺ツールの開発プロジェクト支援を進める

2022年度 体制・会員構成

- 会員は法人およびそれに準ずる団体で構成



2022年度体制

理事長	:日本電信電話株式会社
運営委員長	:日本電気株式会社
技術部会長	:富士通株式会社
CR部会長	:SRA OSS LLC
広報・発信部会長	:株式会社 日立製作所
広報・発信副部会長	:株式会社アシスト
事務局長	:SRA OSS LLC
監事	:オープンソース活用研究所

種別		概要	総会議決権
正会員	理事	理事会に参加、理事長および運営委員長は理事のうちから就任する	あり
	運営委員	運営委員会に参加、部会長およびWG長は運営委員から就任する	
		ワーキンググループ(WG)に参加し、活動に貢献	
一般会員		メーリングリストやWebなどから、活動情報を取得することが可能	なし

ワーキンググループ活動について

- 現在3つのワーキンググループにて活動中

新技術検証WG（WG1）

- 新バージョンの性能や新技術の検証を通じて有用性を明確化
- スケールアップ検証、新機能における性能特性調査等

移行WG（WG2）

- 異種DBMSからの移行をテーマに活動
- 商用DBMSからの移行プロセスに伴う技術調査や検証を実施

課題検討WG（WG3）

- データベース管理者やアプリケーション開発者が抱える現場の課題や困り事に対するテーマを設定
- 可用性・運用性・保守性・セキュリティ・接続性が主な課題領域

PGEConsウェブサイト

- https://www.pgecons.org/

会員募集

- 正会員・一般会員を広く募集いたします
  - WG・CR部会で一緒に活動を行っていただける団体様 ⇒ 正会員
  - PostgreSQLのエンタープライズ領域に興味を持っている団体様 ⇒ 一般会員

お問い合わせ先：
PostgreSQLエンタープライズ・コンソーシアム事務局
メール ： jimukyoku@pgecons.org
Web ： http://www.pgecons.org/

Contents

- 1. 背景
- 2. 調査
- 3. 検証結果と考察
- 4. 総括
- 補足資料: 検証結果詳細

責任範囲

- 本資料は、PGEConsが独自に検証した結果であり、
  結果はPGEConsの責任の元、公開しています。

1. 背景

shared_buffers設定値の疑問点

- 2019年度WG3(課題検討ワーキンググループ)活動
  「機械学習を用いたPostgreSQLパラメータのチューニング検証」
  にて得られたshared_buffers最適値はシステムメモリ※の50%(4.2GB)。

  参考: 機械学習を用いたPostgreSQLパラメータのチューニング検証 | 2019年度WG3活動報告書

  ※システムメモリ: OSが使用できる または クラウドサービスのリソースに割り当てられているメモリ

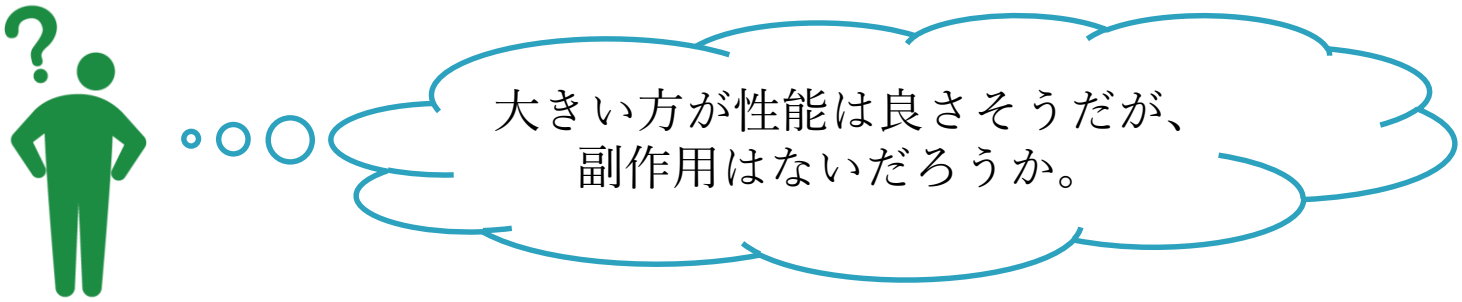
- PostgreSQLマニュアル(13.1)には
  システムメモリに40%以上割り当てても性能は向上しないとある。
  - 「shared_buffersにRAMの40%以上を割り当てても、それより小さい値の
    時より動作が良くなる見込みはありません。」
  - 「妥当な初期値はシステムメモリの25%」

    参考: 19.4. 資源の消費 | PostgreSQL 13.1文書

検証で得られた最適値と
マニュアルの記載内容が異なる？？？

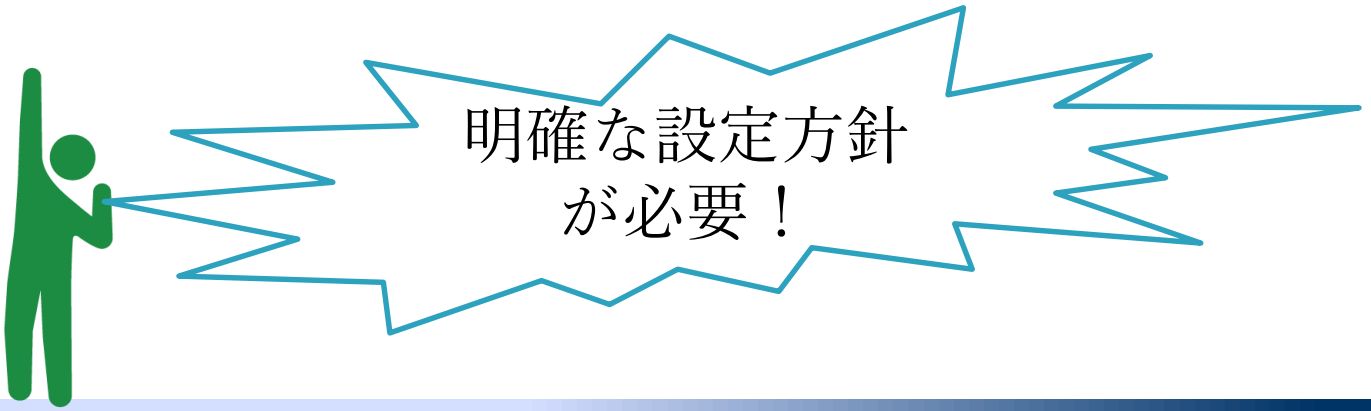
データベースバッファキャッシュの一般論

- **PostgreSQL**
  - PostgreSQLグローバル開発グループが公開するマニュアルでは
    システムメモリの25%が妥当な初期値とされている。
    ※バージョン13.1現在
- **他のDBMS**
  - バッファキャッシュのサイズは可能な限り最大化する。
- **Aurora PostgreSQL**
  - バッファキャッシュサイズのデフォルト値が
    システムメモリの75%とされている。



「25%」の根拠についての調査

- **マニュアル**
  - バージョンによって表現の遷移は見られるが
    明確な根拠と思われる記載はなし。
  - “ワークロードによっては大きくした方が良い”とあるが
    具体的なワークロードについては記載なし。
    - **参考**: 19.4. 資源の消費 | PostgreSQL 13.1文書
- **インターネット上の情報**
  - マニュアルを根拠に25%を推奨する記載が大多数である。
  - 残念ながら根拠となり得る情報は記載されていない。



本検証の目的と方針

- **目的**
  - shared_buffersの設定を変化させた場合の
    処理性能への影響を調査し設定方針を明確にする。
- **方針**
  - 全てのワークロードのパターンを今回の検証でカバーすることは出来ない。
    そのため、性能劣化に影響を与えると思われる内部処理単位で検証を実施することで、検証パターンの網羅性をカバーする。

検証項目概要

- 内部処理の次の点に着目し検証を実施する
  - 読み込み
    - 単一行読み込み
    - テーブル全読み込み
  - 書き込み
    - 行更新
    - チェックポイント
  - 並行実行性
  - メモリ利用効率
    - ページ重複による無駄
    - テーブル全読み込み

検証環境

サーバ	ハードウェア情報	CPU: 2コア4プロセッサ（Intel(R) Core(TM) i5-2415M CPU @ 2.30GHz）
		システムメモリ: 16GB
	ソフトウェア情報	OS: Ubuntu 20.04.3 LTS （GNU/Linux 5.4.0-88-generic x86_64）
		PostgreSQL: 13.4 （検証時の最新版）
クライアント	サーバ上に用意し、同一マシン内で行う（psql, pgbench）	
PostgreSQL の主な設定	shared_buffers = 4GB/8GB/12GB	※今回の検証対象
	checkpoint_timeout = 1d	チェックポイント発生抑制のため
	max_wal_size = 30GB	チェックポイント発生抑制のため
	autovacuum = off	自動バキューム発生抑制のため

2. 調査

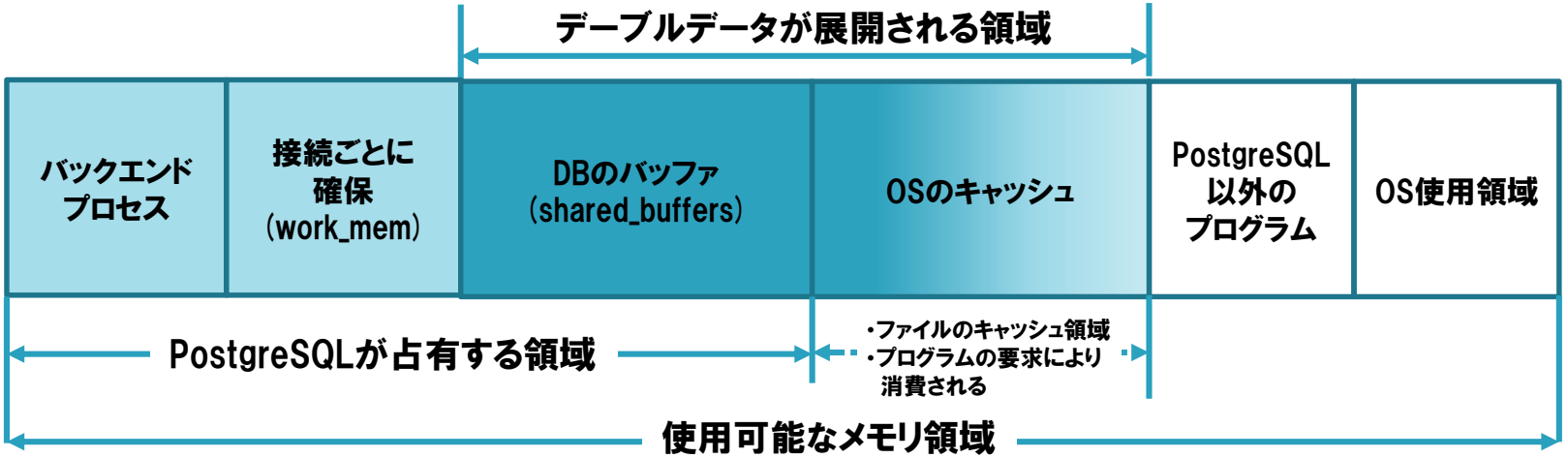
資料中の表記について

- **資料中の呼称について以下の通り統一する。**
  - キャッシュ領域の呼称
    - OSが管理するページキャッシュ ➡ OSのキャッシュ
    - PostgreSQLの共有バッファ(shared_buffersにより確保される領域)
      ➡ DBのバッファ

PostgreSQLのバッファ戦略　1/2

- **OSのキャッシュをDBのバッファの延長として考える**
  - DBのバッファ上に参照予定のデータが残っていない場合でもOSのキャッシュ上にデータが残っていればディスクアクセス無しにデータを参照できる。
  - Aurora PostgreSQLはOSのキャッシュを使わない設計のためshared_buffersを75%に拡大する戦略を取っている。

    参考: Shared_Buffers DB パラメータのデフォルト値と Amazon RDS PostgreSQL および Aurora PostgreSQL の間に差がある理由を理解する | AWS ナレッジセンター



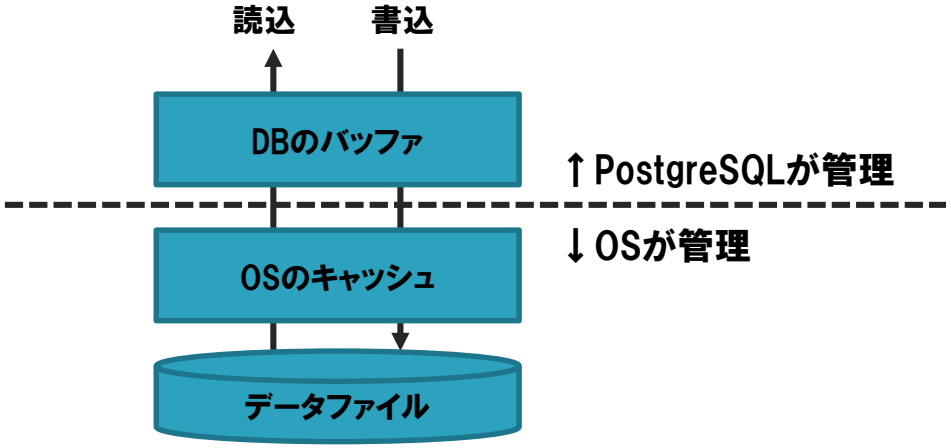
図：PostgreSQL稼働時のメモリの使用イメージ

PostgreSQLのバッファ戦略　2/2

- **OSのキャッシュを活用するメリット**
  - 接続プロセス数の増加や使用メモリ領域の増加に対応できる。
    - メモリが必要になればOSのキャッシュが割り当てられる※。
      ※ OSのキャッシュが少なくPostgreSQLが huge page で起動している時(huge_pagesの設定がtryまたはonの場合) 新たな接続が発生した場合などにPostgreSQLが使用するメモリがスワップアウトされずメモリ不足によるプロセスの異常終了に繋がることに注意が必要。
- **OSのキャッシュを活用するデメリット**
  - 不用意にOSのキャッシュのヒット率が低下し処理性能が劣化する可能性がある。
    - OSのキャッシュは占有されないため、データファイルが載るキャッシュが他プログラムに割り当てられたり別のファイルのキャッシュに利用される。
  - PostgreSQLから正確なキャッシュヒット率を測定できない。
    - 統計情報コレクタの採取対象にはOSのキャッシュへのアクセスが含まれない。

PostgreSQLのデータキャッシュアーキテクチャ 1/2

- **データのキャッシュアーキテクチャの要素として DBのバッファ と OSのキャッシュ が存在する**



図：データキャッシュアーキテクチャ

  - クライアントから要求されたデータは図の上位から検索し見つかれば返却する。
    - 見つからない場合、下位を検索する。
    - 挿入/更新/削除もDBのバッファとOSのキャッシュを経由する。ただしデータファイルへの書き込みは遅延実行により性能を確保する。

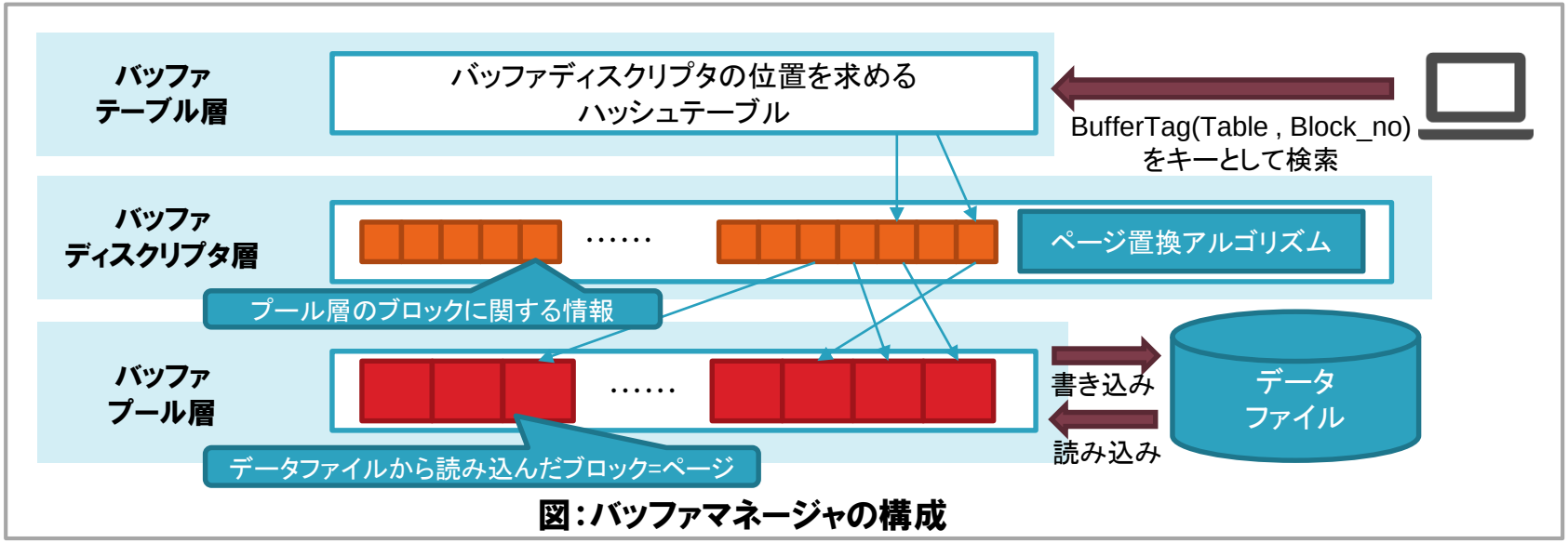
PostgreSQLのデータキャッシュアーキテクチャ 2/2

- **DBのバッファの管理に必要な機能**
  1. バッファから目的のページを探す
  2. バッファに目的のページがない場合、データファイル（OSのキャッシュ）から目的のページを取得する
  3. データファイルからページを取得するがバッファに空きがない場合、データを置き換えるページを特定し、置き換えを行う
- **PostgreSQLでは、バッファマネージャが DBのバッファを管理する**

PostgreSQLのバッファマネージャ

- バッファマネージャは3層にて構成

層の名称	役割
バッファテーブル	テーブルのページ情報をキーにしたハッシュテーブル。バッファディスクリプタの配列位置を保持
バッファディスクリプタ	バッファ数分の固定配列。ロック状態、ダーティーページフラグ、参照回数などを保持
バッファプール	バッファ数分の固定配列、テーブルのページを保持

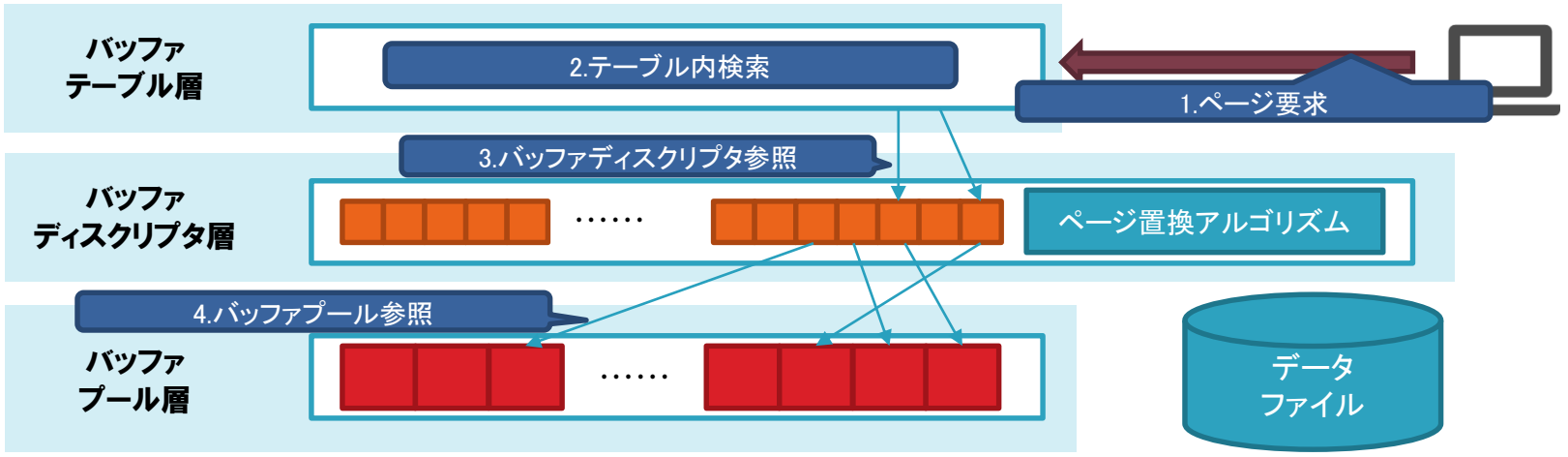


図：バッファマネージャの構成

バッファマネージャの挙動　1/2

■ 要求ページがDBのバッファにある場合

1. ページ要求
2. 要求ページがバッファプールに存在するか、バッファテーブルをチェックする（この場合存在する）
3. バッファテーブルのバッファディスクリプタ配列位置情報よりバッファディスクリプタを参照する
4. バッファディスクリプタのバッファプール配列位置情報よりバッファプールの目的ページを参照する。

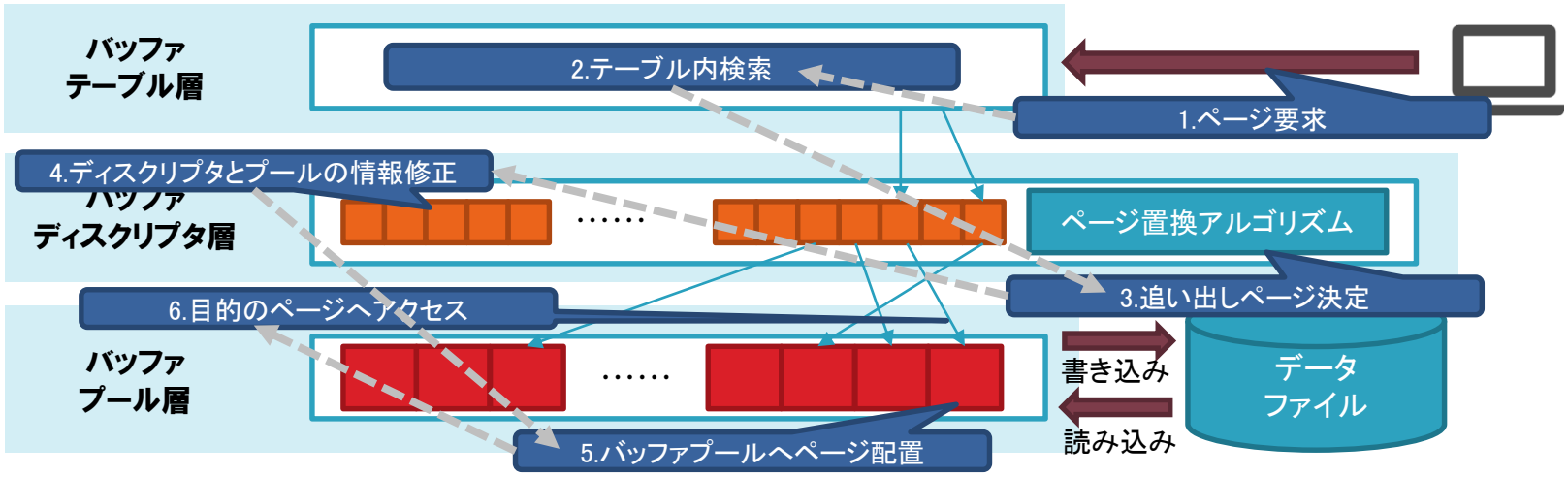


図：要求ページがバッファプールにある場合の動作

バッファマネージャの挙動　2/2

■ 要求ページがDBのバッファにない場合

1. ページ要求
2. 要求ページがバッファプールに存在するか、バッファテーブルをチェックする（この場合、存在しない）
3. バッファプールに空きがない場合、追い出しページを決定する
   ※この処理は必要時に同期で実行（ソースコードより確認）
4. バッファテーブル、バッファディスクリプタの情報を修正する
5. データファイルから目的のページを読み込み、バッファプールに配置する
6. バッファプールの目的ページへアクセスする



図：要求ページがバッファプールにない場合の動作

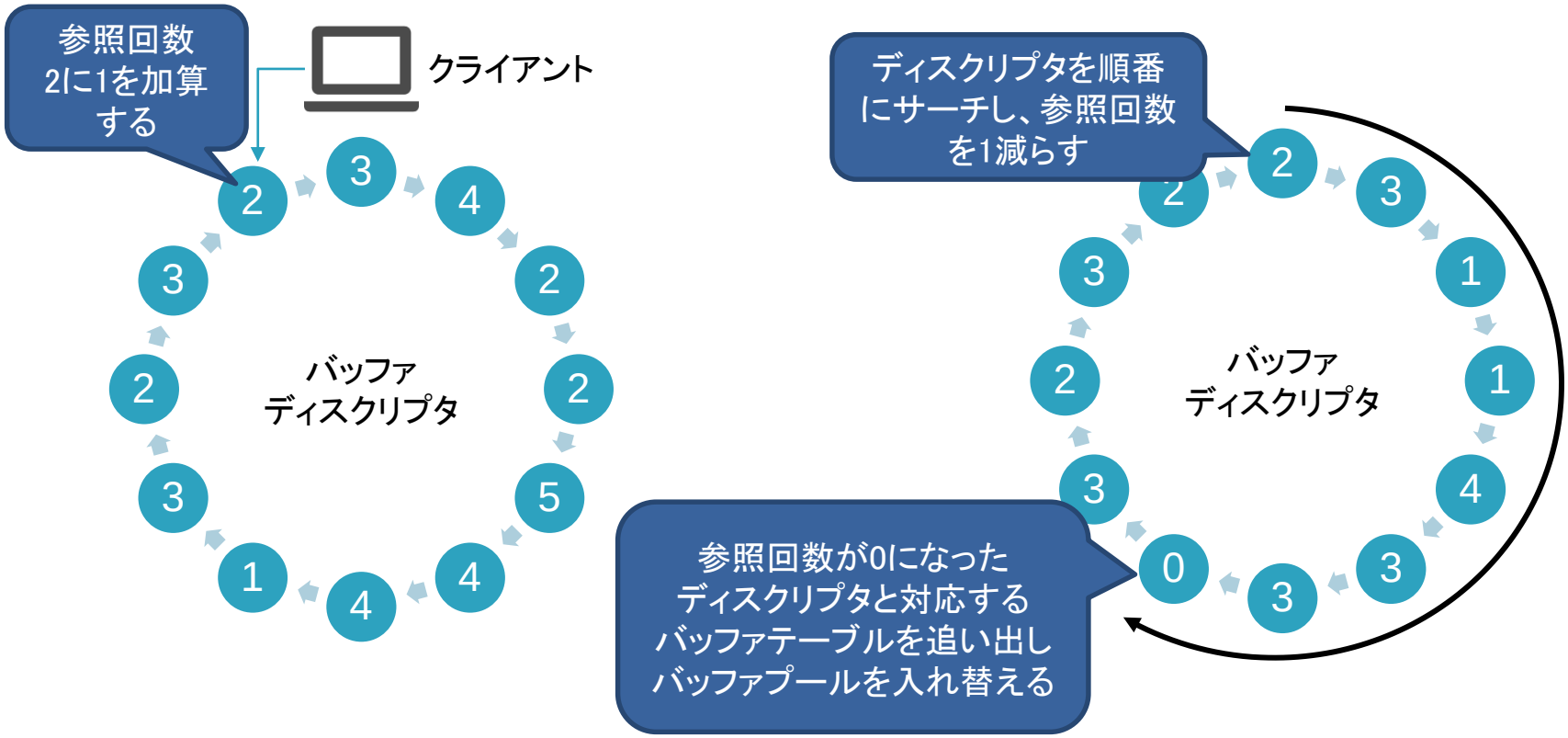
PostgreSQLのページ置換アルゴリズム　1/3

- **Clock Sweepを採用している**
  - 読み出し時
    - バッファディスクリプタ上のページに紐付けされている項目「参照回数(usagecount)」をインクリメントする。
  - 追い出し時
    - バッファディスクリプタ上のページを順番にサーチし追い出しページが見つかるまで繰り返す。
      - 参照回数が１以上であれば、参照回数をデクリメントし次のバッファディスクリプタを参照する。
      - 参照回数が0の場合、そのページをバッファプールから追い出す（データの置き換え対象とする）。

PostgreSQLのページ置換アルゴリズム　2/3

[読み出し時]
アクセスされたプールに対応した
バッファディスクリプタの参照回数に1加算

[追い出し時]
バッファディスクリプタの参照回数を1減算
参照回数が0となったバッファディスクリプタの
データを追い出す



図：Clock Sweep 動作

PostgreSQLのページ置換アルゴリズム　3/3

- **メリット**
  - ページの順序を定期的に入れ替える必要がないことから、LRUなど他のアルゴリズムと比べてオーバーヘッドが少なく、処理負荷が軽い。
- **デメリット**
  - キャッシュヒット率が高い場合、追い出し対象となるページを見つける（参照回数が0になるまでデクリメントする）ために時間がかかることがある。ただし次の理由からその影響は少ない。
    - PostgreSQLではバッファディスクリプタが持つ参照回数は最大5に制限されており、バッファ全体の参照回数は有限である。
    - 全てのページの参照回数が均一になるようなワークロードでなければ、バッファ全体を参照する必要がないため問題にならない。
      OLTPであればページの参照回数は偏るため少数のページの検索で追い出し対象を見つけることができる。

参考: ページ置換アルゴリズム変遷経緯

PostgreSQL バージョン			開発期間	アルゴリズム
	-	7.4	～2004	LRU
8.0.0	-	8.0.1	～2005.1	ARC
8.0.2	-	8.0.36	～2005.2	2Q
8.1	-		2005.11～	Clock Sweep

3. 検証結果と考察

本検証の目的と方針（再掲）

- **目的**
  - shared_buffersの設定を変化させた場合の
    処理性能への影響を調査し設定方針を明確にする。
- **方針**
  - 全てのワークロードのパターンを今回の検証でカバーすることは出来ない。
    そのため、性能劣化に影響を与えると思われる内部処理単位で検証を実施することで、検証パターンの網羅性をカバーする。

検証項目概要（再掲）

- 内部処理の次の点に着目し検証を実施する
  - 読み込み
    - 単一行読み込み
    - テーブル全読み込み
  - 書き込み
    - 行更新
    - チェックポイント
  - 並行実行性
  - メモリ利用効率
    - ページ重複による無駄
    - テーブル全読み込み

検証環境（再掲）

サーバ	ハードウェア情報	CPU: 2コア4プロセッサ（Intel(R) Core(TM) i5-2415M CPU @ 2.30GHz）
		システムメモリ: 16GB
	ソフトウェア情報	OS: Ubuntu 20.04.3 LTS （GNU/Linux 5.4.0-88-generic x86_64）
		PostgreSQL: 13.4 （検証時の最新版）
クライアント	サーバ上に用意し、同一マシン内で行う（psql, pgbench）	
PostgreSQL の主な設定	shared_buffers = 4GB/8GB/12GB	※今回の検証対象
	checkpoint_timeout = 1d	チェックポイント発生抑制のため
	max_wal_size = 30GB	チェックポイント発生抑制のため
	autovacuum = off	自動バキューム発生抑制のため

検証で設定するDBのバッファについて

- 検証で設定するDBバッファの割合、サイズと本資料での呼称は以下の通り。

サイズの呼称	システムメモリに対する割合	検証環境（16GB）のサイズ	備考
小	25%	4GB	マニュアル推奨値
中	50%	8GB	
大	75%	12GB	

検証結果サマリ

DBバッファのサイズ(大・中・小)によるバッファマネージャ処理時間への影響

カテゴリ	検証観点	実行時間への影響	備考
読み込み	単一行読み込み	影響小	サイズが大きいほど性能が向上
	テーブル全読み込み	影響なし	
書き込み	行更新	影響小	サイズが大きいほど性能が向上
	チェックポイント	影響小	サイズが小さいほど性能が向上
並行実行性	—	影響なし	
メモリ利用効率	データの重複による無駄	影響大	利用効率と性能はDBのバッファのサイズが 大 > 小 > 中 の順で低下する
	テーブル全読み込み後の影響	影響大	サイズが大きいほど性能が向上

※ バッファマネージャの動作調査から、絶対的なサイズの違いによって結果の傾向が変わることはないと判断し、システムメモリは固定値(16GB)として検証を実施。

読み込み(単一行読み込み) 1/4

- **バッファマネージャ動作手順**
  1. 目的のページを特定する。
  2. 特定したページがDBのバッファにあるか検索する。
  3. DBのバッファにない場合データファイルから取得する。
     - 3-1. DBのバッファに空きがない場合、DBのバッファを空ける。
     - 3-2. 配置先のバッファに対してロックを取得する。
     - 3-3. ディスクからファイルを読み込む。
     - 3-4. 読み込んだページをDBのバッファに配置する。
     - 3-5. DBのバッファのページロックを解除する。
  4. DBのバッファから目的のページを取得する。

赤枠の項目がDBのバッファのサイズが影響を与えると考えれられる箇所。
これらについて以降検証を行う。

読み込み(単一行読み込み) 2/4

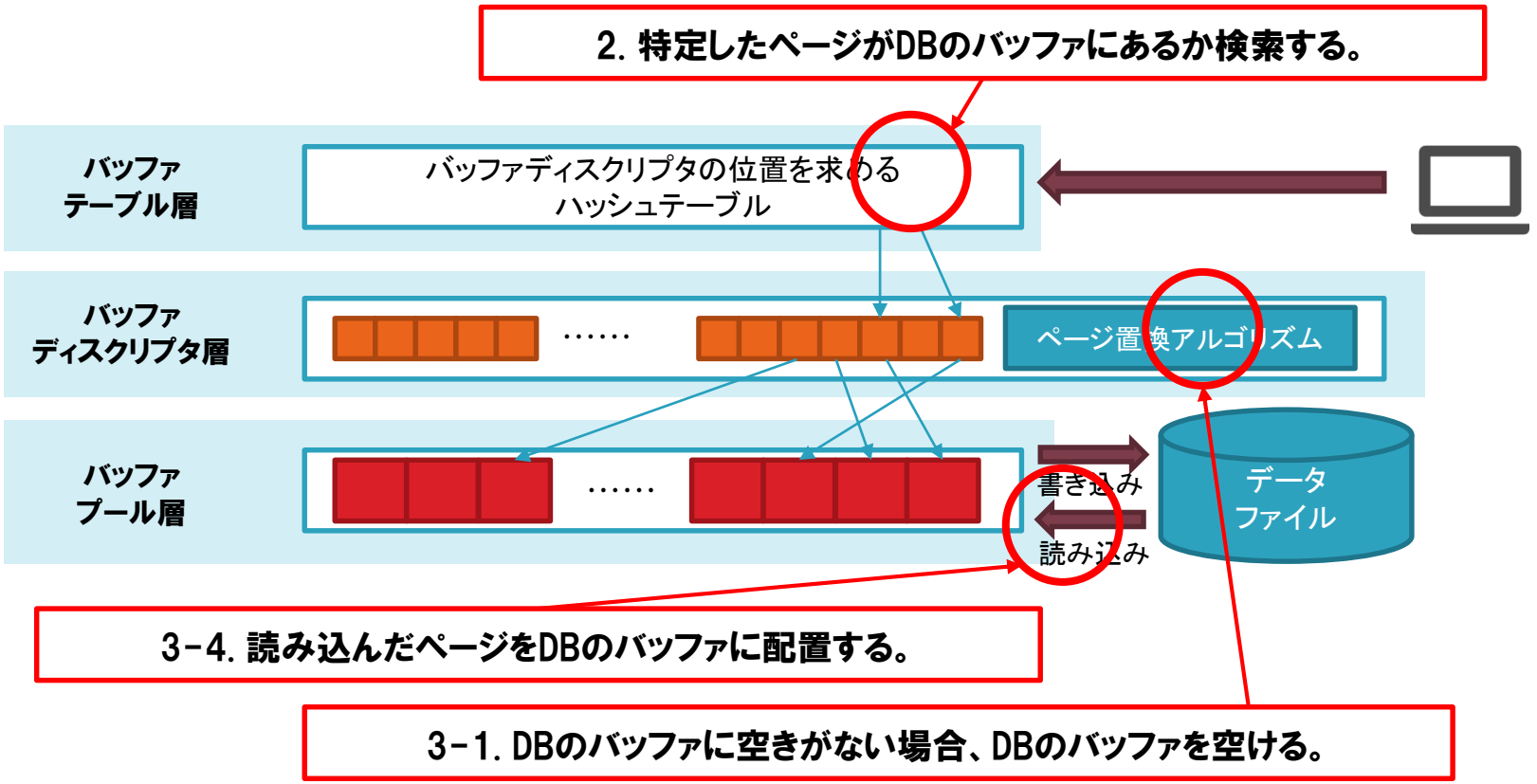


図:バッファサイズの大小により影響が想定される箇所

読み込み(単一行読み込み) 3/4

- **検証箇所**
  - **「2. 特定したページがDBのバッファにあるか検索する。」の部分**
    検証#1　DBのバッファからの読み込み速度比較
    結果　DBのバッファの大きさによる影響はない。
  - **「3-1. DBのバッファに空きがない場合、DBのバッファを空ける。」の部分**
    検証#2　Clock Sweepの速度比較
    結果　DBのバッファ小の方が DBのバッファ大より30ms程度早い
    （1Gバイトあたり3.8ms程度）
  - **「3-4. 読み込んだページをDBのバッファに配置する。」の部分**
    検証#3　「DBのバッファ＋OSのキャッシュ（バッファ小）」と
    「DBのバッファのみ（バッファ大）」の読み込み速度比較
    結果　DBのバッファのみ（バッファ大）の方が0.038ms程度早い

読み込み(単一行読み込み） 4/4

- **考察:影響小 - DBのバッファ大が有利**
  - DBのバッファのサイズによる、バッファプールへのアクセス時間の差は発生しない。
  - DBのバッファが小さいとOSのキャッシュからの読み込みの頻度が高くなり、以下の理由から読み込み時間が増加する（0.038msec程度）。
    ただし絶対的な増加時間は小さいため、影響は小さいと考える。
    - OSのキャッシュからDBのバッファへロードするためのオーバーヘッドが発生
    - Clock Sweepの発生

読み込み(テーブル全読み込み) 1/2

- **前提**
  - DBのバッファサイズの1/4を超えるテーブルサイズの全読み込みの場合、リングバッファによる読み込みを行う
    - 1/4以下の場合は単一行読み込みと同じ動作となる
- **バッファマネージャ動作手順**
  1. 共有メモリ上に256KBのリングバッファ領域を確保する
  2. データファイルから読み込んだデータをリングバッファに配置する
  3. リングバッファ上のページよりレコードを取得する

リングバッファ

リングバッファはDBのバッファ上に確保される固定長の領域。
大きなサイズのテーブルでもリングバッファ内を循環させるようして読み込むことでDBのバッファ全体を使用せずに読み込みを行う。そのため、DBのバッファが大きなテーブルのデータで埋め尽くされる事態を回避できる。

読み込み(テーブル全読み込み) 2/2

- **DBのバッファのサイズによる影響箇所**
  - なし
    - リングバッファを使用することでDBのバッファ全体を使用せずに読み込みを行う。リングバッファのサイズはDBのバッファサイズに依らず固定であるため、処理手順に影響を与える箇所はない

- **考察:影響なし**
  - リングバッファを使用した場合、処理時間はDBのバッファのサイズに影響を受けない。

書き込み(行更新) 1/2

- **バッファマネージャ動作手順**
  1. 操作対象ページの特定・読み込み
     - 1-A. [INSERT] 空きページを特定する
     - 1-B. [UPDATE/DELETE] 目的ページの特定・読み込み
  2. 更新前レコードを更新する
     - 2-A. [INSERT] 処理なし
     - 2-B. [UPDATE/DELETE] 更新前レコードに現在のトランザクション情報を書き込む
  3. 更新後レコードを書き込む
     - 3-A. [INSERT] 操作対象ページに書き込む
     - 3-B. [UPDATE] 更新前レコードの存在するページに空きが
       - 3-B-A. ある：同一ページに書き込む
       - 3-B-B. ない
         - 3-B-B-1. 空ページの特定・読み込み
         - 3-B-B-2. 特定したページに書き込む
     - 3-C. [DELETE] 処理なし
  4. 更新・書き込んだ内容をWALに書き込む
  5. COMMIT時、WALをディスクに書き出す

赤枠の項目がDBのバッファのサイズが影響を与えると考えれられる箇所。
ただしこれらの内部処理は
「読み込み(単一行読み込み)」
と同じ内容となる。

書き込み(行更新) 2/2

- **検証箇所**
  - **バッファサイズが影響を与える箇所の処理内容が「読み込み(単一行読み込み)」と同じであるため、同様の傾向となることを確認する**

    検証#4 コミットまでの速度比較

    結果　　UPDATEによる検証の結果、読み込み（単一行読み込み）と同じくバッファサイズ大の方が若干処理時間が小さい傾向となった

表:UPDATE時の処理速度比較 (msec)

バッファサイズ	UPDATE	(参考)COMMIT
小	0.170	8.025
大	0.165	8.099

- **考察:影響小 - DBのバッファサイズ大が有利**
  - 読み込み（単一行読み込み）と同一

書き込み(チェックポイント)　1/3

- **バッファマネージャ動作手順**
  1. チェックポイント発生
  2. ダーティーページの特定
  3. ダーティーページをディスクに書き出し

赤枠の項目がDBのバッファのサイズが影響を与えると考えれられる箇所。
これらについて以降検証を行う。

書き込み(チェックポイント)　2/3

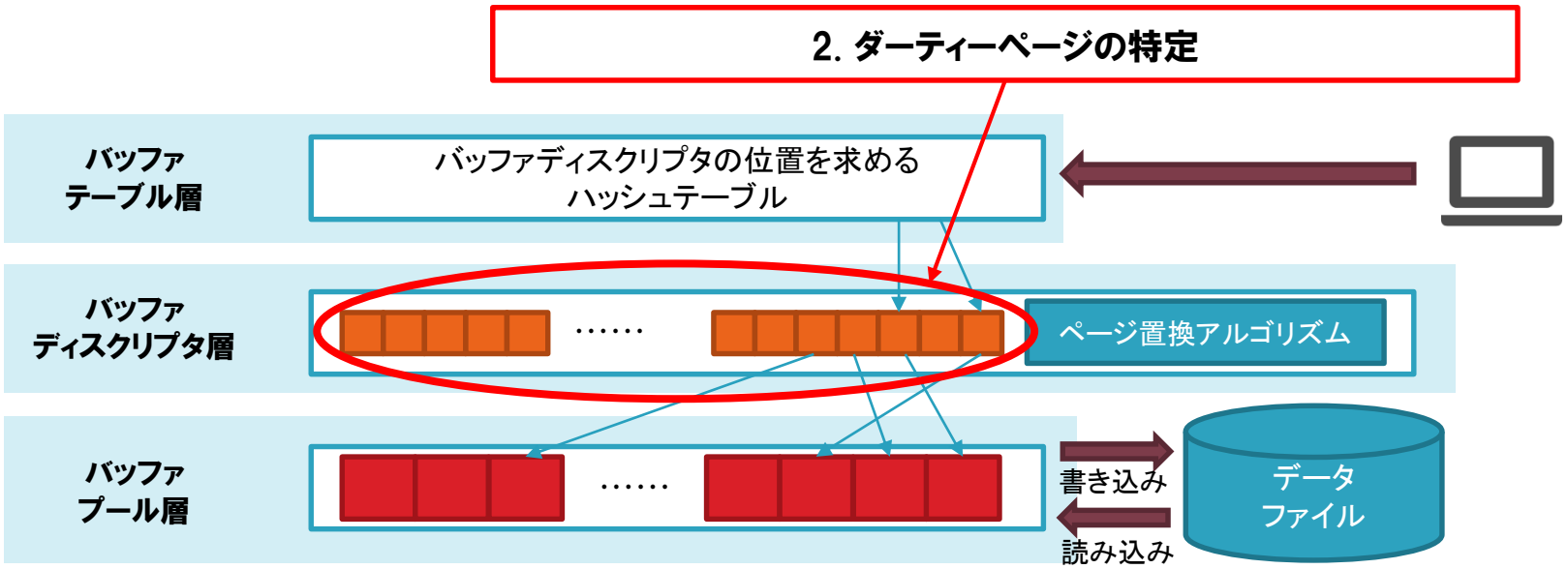


図:バッファサイズの大小により影響が想定される箇所

書き込み(チェックポイント)　3/3

- **検証箇所**
  - **「2. ダーティページの特定」の部分**

    検証#5 バッファディスクリプタ全走査時間の確認
    - チェックポイント処理は『ダーティページの特定』と『ディスクへの書き出し』が行われる。このうち『ディスクへの書き出し』は更新量によるためDBのバッファサイズとは関係がない。一方『ダーティページの特定』はバッファディスクリプタ全体を走査する必要があるため、DBのバッファサイズが処理時間に影響を与える。そのためバッファディスクリプタ全走査時間を確認する。

    結果　　全検索はDBのバッファ 1GB あたり 3.8ms

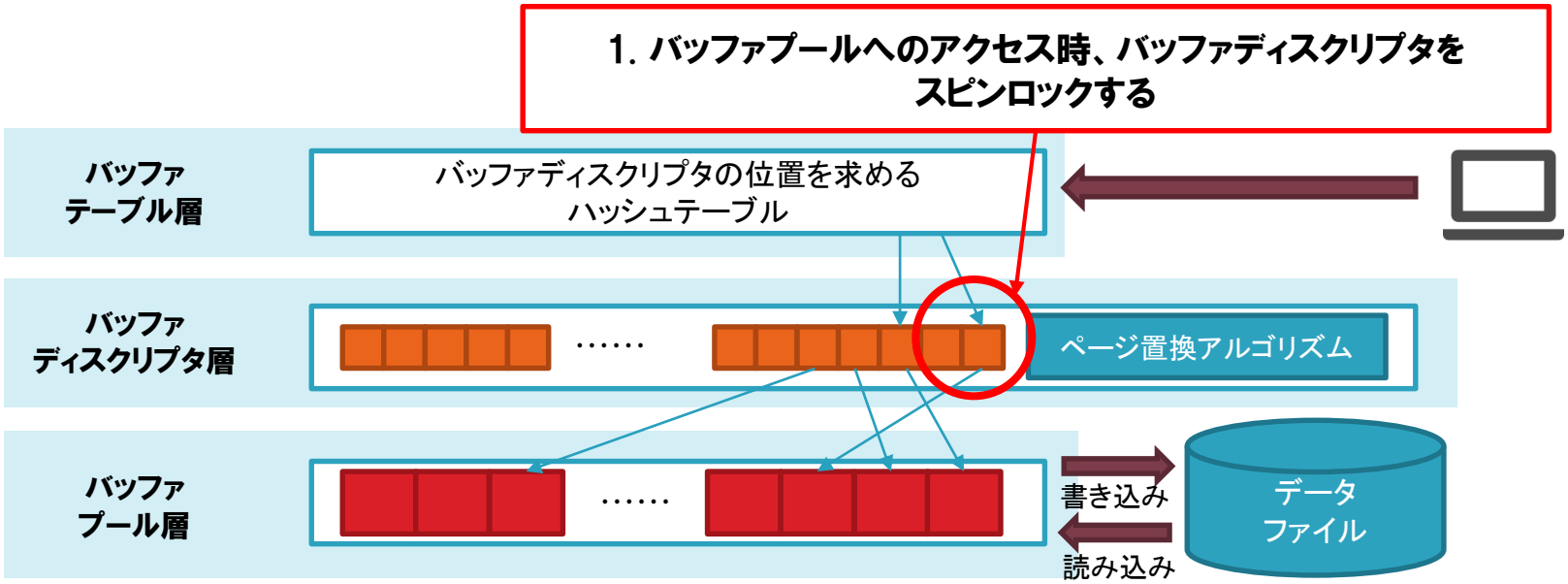
- **考察:影響小 – DBのバッファサイズ小が有利**
  - バッファディスクリプタ全走査時間はDBのバッファサイズに比例するがディスクへの書き込み時間と比べて相対的に小さい値となるため影響は少ない。

並行実行性　1/4

- **確認事項**
  - 並行実行数を増やすと、ある実行数を境に処理時間は長くなり、スループットは頭打ちとなり、以降減少する。
    この傾向や変化の度合いが、DBのバッファのサイズにより変化するか確認する。
- **バッファマネージャ動作手順**
  1. バッファプールへのアクセス時、バッファディスクリプタをスピンロックする

赤枠の項目がDBのバッファのサイズが影響を与えると考えれられる箇所。これらについて以降検証を行う。

並行実行性　2/4



図：バッファサイズの大小により影響が想定される箇所

並行実行性　3/4

- **検証箇所**
  - **「1. バッファプールへのアクセス時、バッファディスクリプタをスピンロックする」の部分**

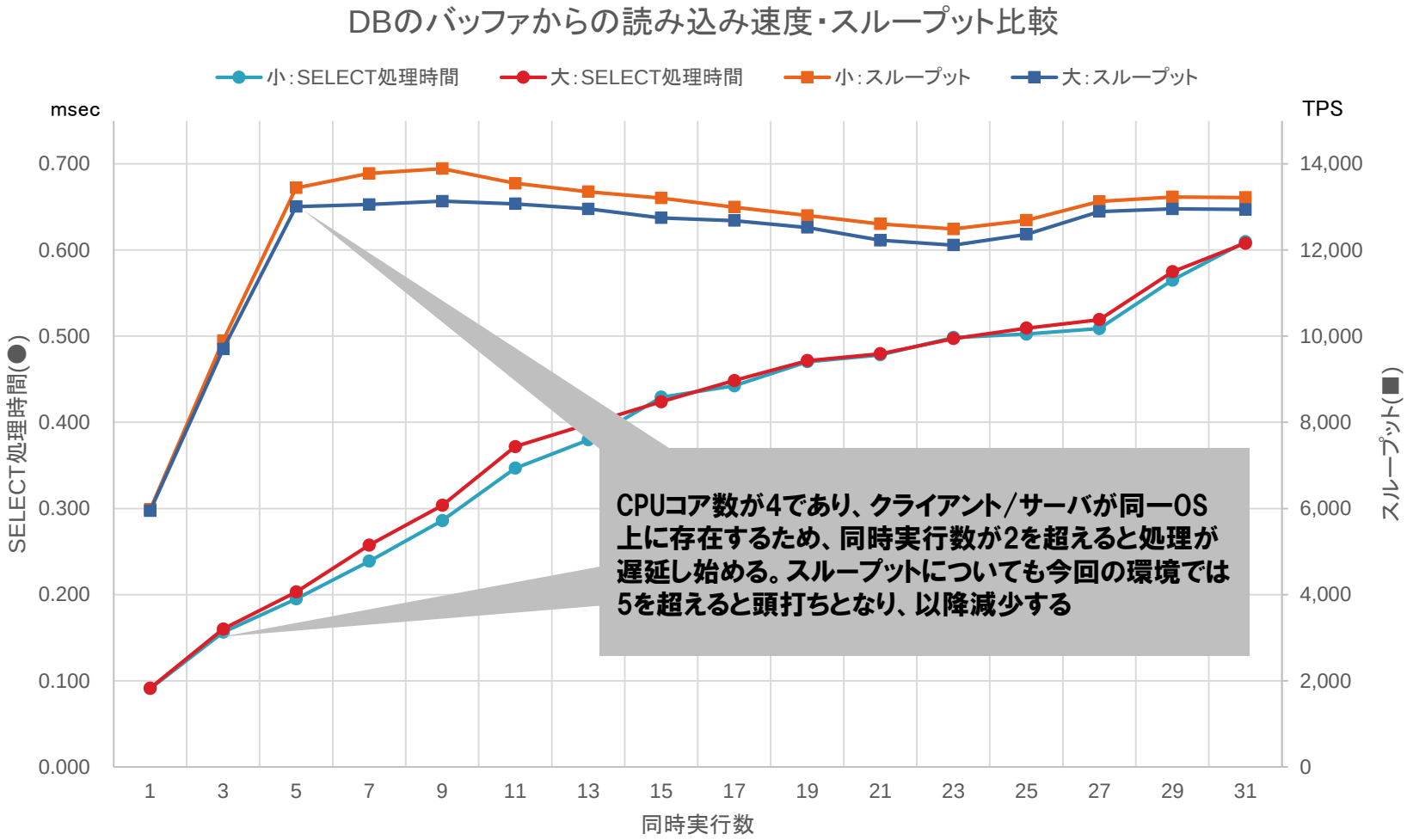
    検証#6 並行実行時の影響

    - DBのバッファサイズに依らず並行実行時でもスピンロックの競合頻度は増えず処理時間とスループットの傾向に影響が出ないことを確認する。
    - pgbenchを使用し検証を実施
      - 独自テーブルと独自スクリプトを使用
      - 独自テーブル：DBのバッファ内で処理を完結させるため、DBのバッファの90%のサイズを持つテーブルを用意し、事前にDBのバッファに展開しておく。検証時には1テーブルだけDBのバッファに展開された状態とする
      - 独自スクリプト：同一ページを同時に参照することによる影響を極力避けるために、ランダムに1ページを参照するSELECT文を使用。ページの指定にはctidを条件に使用（今回作成したテーブルは1ページ内に複数タプルが存在するため）
      - 同時実行数を1から31まで変化させる

    結果　処理時間とスループットの推移はDBのバッファサイズに影響されない（次ページ参照）

- **考察：影響なし**

並行実行性　4/4



同時実行数を増やした時の処理時間の増加量、スループットが頭打ちとなるタイミングがDBのバッファサイズに依らないことが確認できた。

メモリ利用効率(ページ重複による無駄) 1/5

- **前提**
  - PostgreSQLは OSのキャッシュとDBのバッファ の両方を活用する設計となっている。
  - ファイルからテーブルデータを参照する際、まずOSのキャッシュへデータが展開され、その後DBのバッファへと展開される。
    そのためメモリ上にデータが重複して存在することになり、利用効率の観点では無駄となる。
- **処理性能への影響**
  - メモリ上により多くのデータが重複なく格納出来れば、それだけキャッシュヒット率も上昇し、処理性能も向上する。
  - そのため、OSのキャッシュも含め、どれだけの領域がバッファキャッシュとして利用できるか（メモリの利用効率）を把握する必要がある。
  - メモリの利用効率が低いと、ディスクへのアクセスが増加するため、処理性能が低下する。

メモリ利用効率(ページ重複による無駄) 2/5

- DBのバッファサイズによる利用効率の違い
  - 以下はデータロード後の「データが展開されるメモリ領域」の状況を表している。この領域の利用効率という観点では、DBのバッファサイズが中の場合が最も効率が悪い。

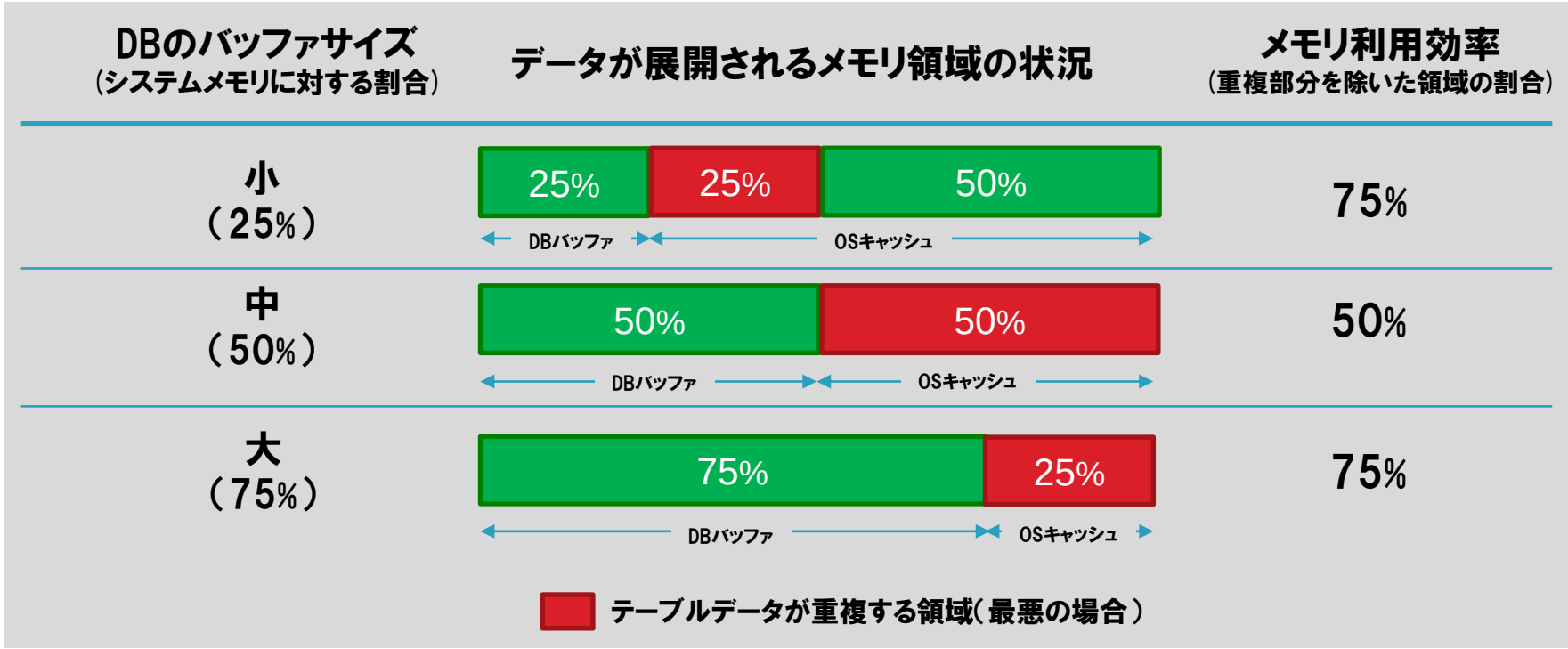


図:DBのバッファサイズごとのメモリ利用状況

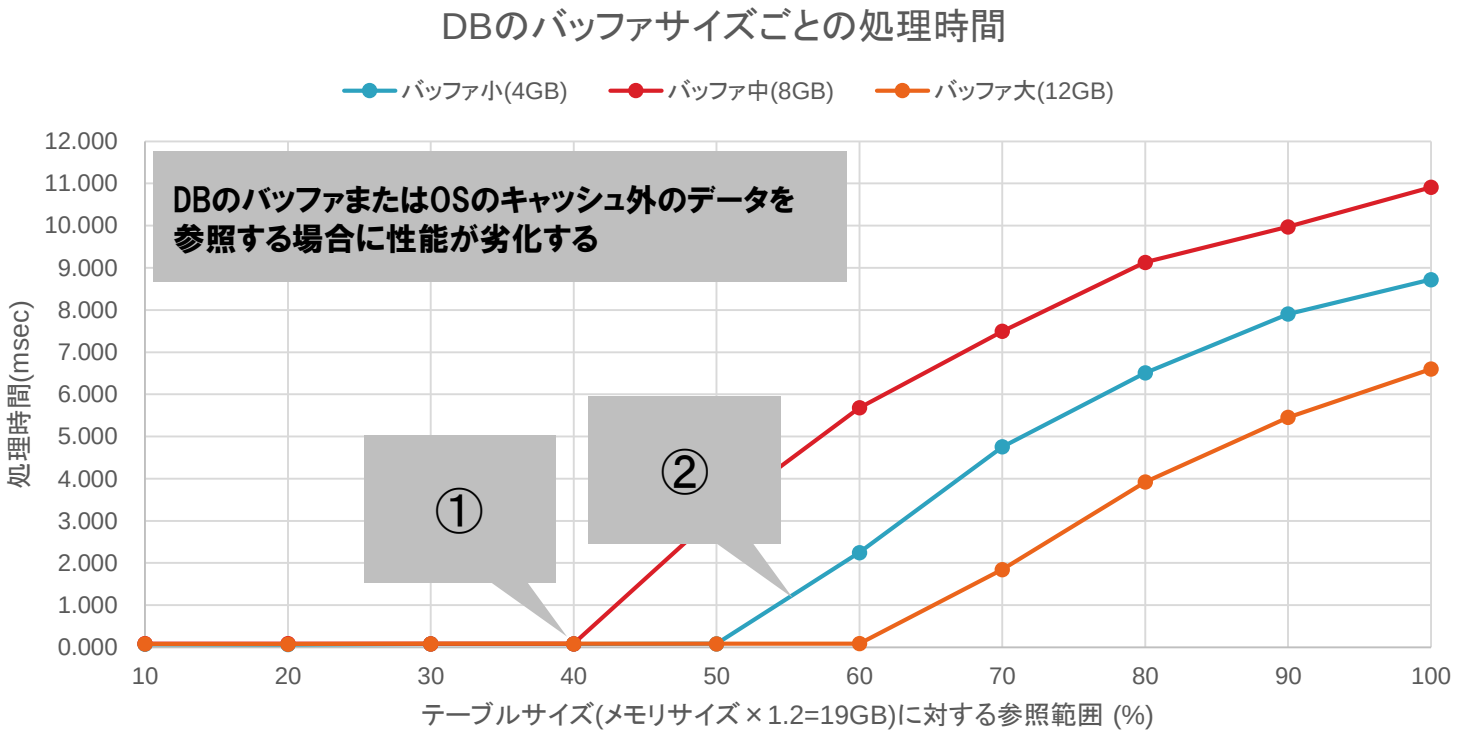
メモリ利用効率(ページ重複による無駄) 3/5

- **検証箇所**
  - **前提にある「ディスクからテーブルデータを参照する際、まずOSのキャッシュへデータが展開され、その後DBのバッファへと展開される。」の部分**

    検証#7 ページ重複による利用効率の変化を確認する
    - OSのキャッシュとDBのバッファのサイズによりメモリの利用効率が変化し処理時間に影響することを確認する。
    - テーブルデータの参照範囲を10%～100%（10%刻み）で変化させワークロードによる影響を確認する。

    結果　メモリ利用効率は 大、小、中 の順で低下する。
    参照レコード数が多いワークロードの場合、処理時間は
    メモリ利用効率の順で低下した（次ページ参照）。

メモリ利用効率(ページ重複による無駄) 4/5



① 『バッファ中』が最もメモリ効率が悪いため、テーブルの参照範囲を広げると最も早く性能が劣化する。

② 『バッファ小』は『バッファ中』よりもメモリ効率が良いが、『バッファ大』と比べてDBのバッファの割合が少なく必要なデータがOSのキャッシュから追い出される可能性が高い。そのため性能の劣化が『バッファ中』と『バッファ大』の間で発生する。

メモリ利用効率(ページ重複による無駄) 5/5

- **考察:影響大 – メモリの利用効率はDBのバッファのサイズが大→小→中の順に低くなる**
  - 参照するデータ量を増やしていくと、メモリの利用効率と処理時間はDBのバッファサイズ『大⇒小⇒中』の順で劣化する。
    『大』と『小』は見かけのメモリ利用効率は同じだが、DBのバッファとOSのキャッシュではデータの追い出しアルゴリズムが異なり、DBのバッファの方が効率よくデータを管理できるため、実際には『小』よりも『大』の方がメモリ利用効率が高く処理時間も有利であることが確認できた。
    特にワークロードの参照データ量が多い場合、DBのバッファが大きい方が処理時間の面で有利となることが確認できた。

メモリ利用効率(テーブル全読み込み) 1/4

- **前提**
  - DBのバッファサイズの1/4を超えるテーブルサイズの全読み込みを行いリングバッファによる読み込みを行う
    - DBのバッファ上には対象テーブルデータが存在しない状態を想定する
    - 1/4以下の場合は単一行読み込みと同じ動作となる
- **バッファマネージャ動作**
  1. DBのバッファ上にリングバッファを確保
  2. ディスク（OSのバッファ）から読み込んだデータをリングバッファ上へ配置する
  3. リングバッファ上のページよりレコードを取得

赤枠の項目がDBのバッファのサイズが影響を与えると考えれられる箇所。これらについて以降検証を行う。

メモリ利用効率(テーブル全読み込み) 2/4

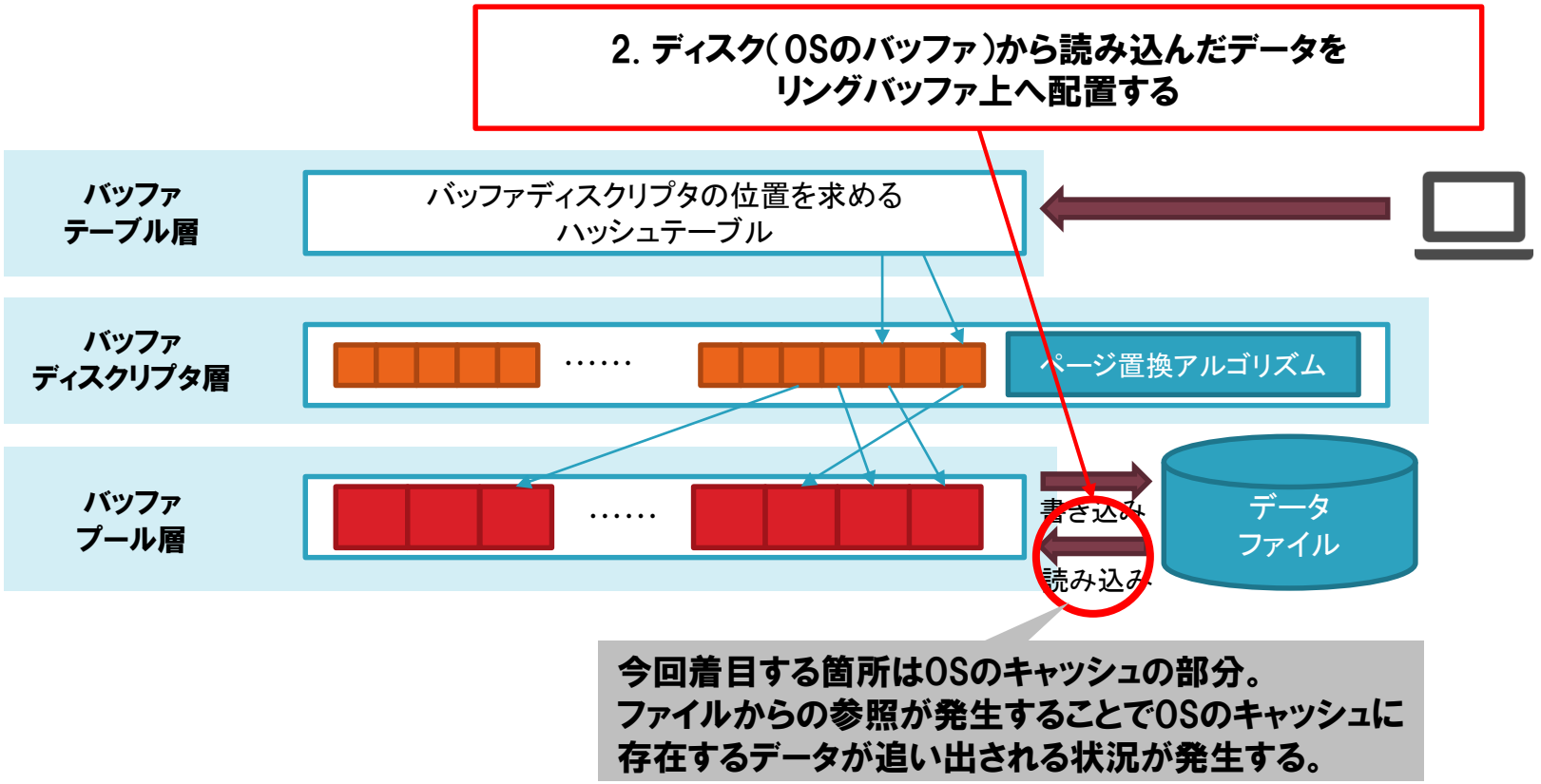


図:バッファサイズの大小により影響が想定される箇所

メモリ利用効率(テーブル全読み込み) 3/4

- 「2. ディスクから読み込んだデータをリングバッファ上へ配置する」の部分

検証#8　テーブル全読み込み後のOSのキャッシュヒット率の比較（全読み込み対象のテーブルサイズ：8GB）

結果　DBのバッファ小の場合、期待しているOSのキャッシュがテーブル全読み込みにより追い出され、DBのバッファ外の参照が全てディスクアクセスとなり処理時間が劣化している事を確認

表:テーブル全読み込み後の処理時間とOSのキャッシュヒット率の比較

バッファサイズ	平均処理時間	DBのバッファヒット率	OSのキャッシュヒット率
小(4GB)	9.958msec	51.1%※	0.0%
大(12GB)	0.076msec	100.0%	OSキャッシュを参照していない

※DBのバッファキャッシュヒット率が50%でないのは、DBのバッファ上に存在する対象テーブルのブロックの割合がテーブル全体の51.18%のため

メモリ利用効率(テーブル全読み込み) 4/4

- **考察:影響大 - DBのバッファ大が有利**
  - DBのバッファには、巨大テーブルのフルスキャン時にリングバッファにより
    DBのバッファ上のデータが追い出されない仕組みが実装されている。
    一方でOSのキャッシュはリングバッファのような工夫がないため
    テーブル全読み込みのような巨大なファイルの読み込みが発生すると
    OSのキャッシュデータが置き換わってしまう。
    そのためDBのバッファ小の場合は、テーブル全件読込みが発生した後の
    OSのキャッシュヒット率低下による一時的な性能低下に注意が必要となる。

4. 総括

総括　1/2

- DBのバッファサイズ別メリット・デメリット

サイズ	メリット	デメリット
小	柔軟なOSのキャッシュを活用するため、綿密なメモリ設計・管理が不要。 突発的な大量のメモリ使用が発生してもOSのキャッシュの利用で耐えられる可能性がある。	OSのキャッシュのデータ管理方法がDB向けでないため、必要なデータがOSのキャッシュ上に残らず、性能が十分に発揮されない場合がある。
中	特になし	メモリの利用効率が最も低く、性能劣化に影響する。
大	メモリを最大限に活用でき、性能を最大化できる。 DBバッファのキャッシュヒット率が信頼できる値となり、問題発生時の稼働情報として活用できる。	同時接続数や負荷状況を基に、入念なパラメータ設計、メモリ管理(監視と見直し)を行わなければ、メモリ枯渇によりサービスがダウンする危険性がある。

総括　2/2

- DBのバッファサイズ別適用推奨パターン
  - 小 – 柔軟なメモリ運用が優先される場合に推奨
    - PostgreSQLのメモリ占有量が小さく、OSの稼動状況に合わせて柔軟なメモリ運用が行える。そのため、同一サーバで複数サービスが稼働するような環境で特に効果的。
    - 詳細な設計や検証の実施が案件の都合により難しい、または将来の運用状況が予測できないような場合には、多少の性能劣化を許容してでも最低限の設計で済ませることが可能。
  - 大 – 処理性能が優先される場合に推奨
    - 内部処理の大半は、DBのバッファを大きくすることで処理性能が向上する。
      そのため、処理性能を優先するならば大とすることを推奨。
    - ただし、メモリ枯渇によるプロセスダウンが発生する危険性があるため、入念な設計、テストおよび運用監視が求められる。
  - 中 – 非推奨
    - 小と大のメリットの良い所取りとはならず、推奨されない。
      特にメモリの利用効率が小/大と比べると悪く、大きな性能劣化を引き起こしかねない。

補足資料: 検証結果詳細

検証環境

サーバ	ハードウェア情報	CPU: 2コア4プロセッサ（Intel(R) Core(TM) i5-2415M CPU @ 2.30GHz）
		システムメモリ: 16GB
	ソフトウェア情報	OS: Ubuntu 20.04.3 LTS （GNU/Linux 5.4.0-88-generic x86_64）
		PostgreSQL: 13.4 （検証時の最新版）
クライアント	サーバ上に用意し、同一マシン内で行う（psql, pgbench）	
PostgreSQL の主な設定	shared_buffers = 4GB/8GB/12GB	※今回の検証項目
	checkpoint_timeout = 1d	チェックポイント発生抑制
	max_wal_size = 30GB	チェックポイント発生抑制
	autovacuum = off	自動バキューム発生抑制

検証手順

- **各試行開始時のDBバッファ/OSキャッシュの状況が同一となる手順とする**
- **1試行の手順**
  1. PostgreSQLが起動していれば停止
  2. OSキャッシュクリア
  3. PostgreSQL起動
  4. 検証で必要なデータのロード
  5. 検証実施

  [pgbenchの場合] ログ出力なし、指定秒数継続

  例: pgbench -c <同時実行数> -j <同時実行数> -T <指定秒数> -f <検証クエリ>

  [psqlの場合] EXPLAIN (ANALYE, BUFFERS) 付きで、指定回数実行

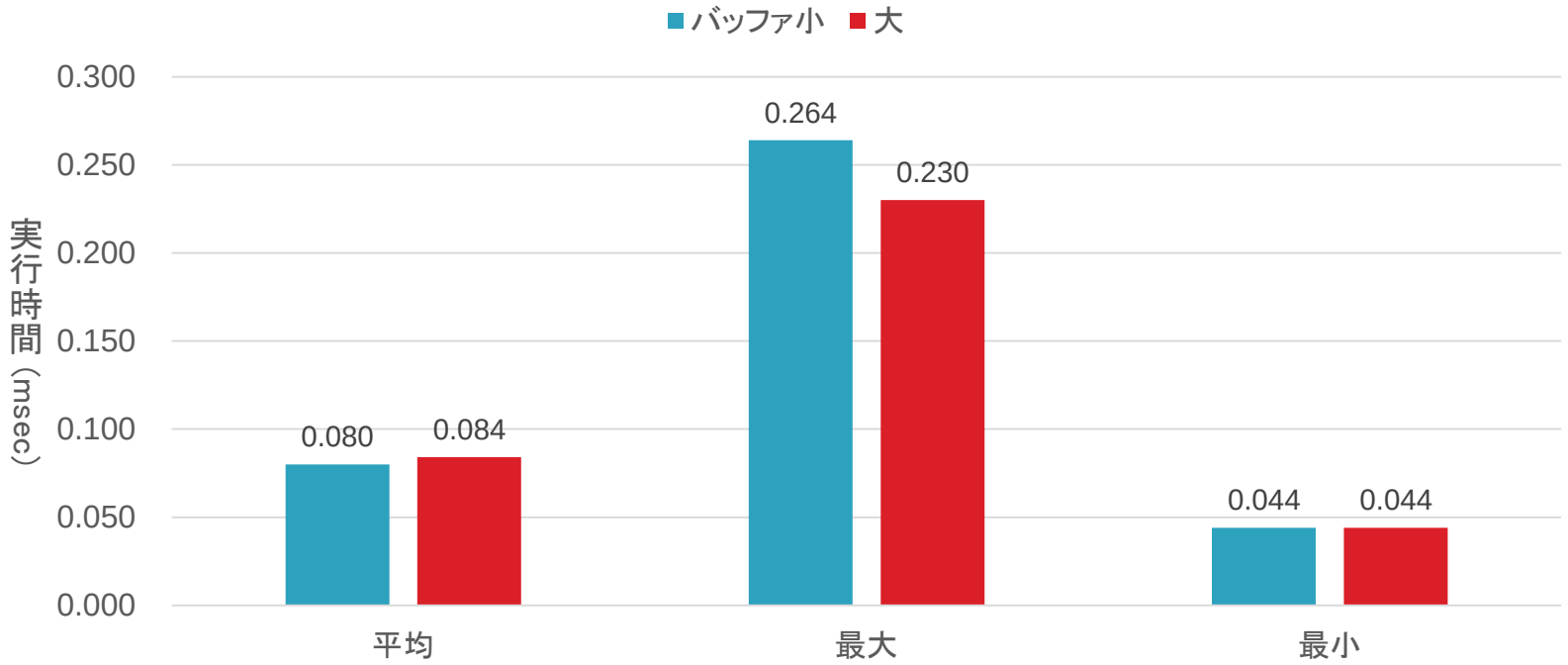
  例: for i in {1..<指定実行回数>} ; do
  psql -f <検証クエリ (EXPLAN (ANALYZE, BUFFERS) を含む) >
  done
- **5回試行しその平均処理時間を評価対象とする**

検証#1 DBのバッファからの読み込み速度比較　1/2

- **目的**
  - DBのバッファの大きさによる、
    DBのバッファから1ページを読み込む速度を確認する。
- **計測内容**
  - DBのバッファに対して90%のデータ容量を持つテーブルを用意する。
    - バッファ小：バッファサイズ 4.0GB、テーブルサイズ 3.6GB
    - バッファ大：バッファサイズ 12.0GB、テーブルサイズ 10.8GB
  - ディスクアクセスとならないよう、検証前にテーブルデータをバッファにロードする。
  - テーブルデータ全域に対して、1ページのランダムアクセスを行う。
    - 実行するSQL（SELECT文）の条件としてTIDを指定する
    - 実行ごとに参照するTIDの値をランダムに変更する（bash の RANDOM 変数を使用）
  - 1回の試行で100回のSQLを実行。
    - SQL実行には psql を使用する。
    - EXPLAIN (ANALYZE, BUFFERS) を用い Execution Time を計測対象とする。

検証#1 DBのバッファからの読み込み速度比較　2/2

■ 計測結果



- DBのバッファサイズの大小に関わらず、読み込み速度は一定である。
- 最大値にて 0.034ms の差があるが実行時のCPU状況によるものと判断。

検証#2 Clock Sweepの速度比較　1/4

- **目的**
  - DBのバッファの大きさによる、Clock Sweepの速度差を確認する。
- **計測内容**
  - DBのバッファ全体にデータをロードしておく。
    - テーブルの全データはusagecountの最大値である5回参照しておく。
  - ロードしたテーブルとは別のテーブルを1ページずつ参照しClock Sweepを発生させる。
    - 実行するSQL（SELECT文）の条件としてTIDを指定する。
    - 実行ごとに参照するTIDの値をインクリメントする。
    - SQL実行には psql を使用する。
    - EXPLAIN (ANALYZE, BUFFERS) を用い Execution Time を計測対象とする。
  - SQLを1試行500回実行し、実行ごとにClock Sweep発生を確認する。
    - pg_buffercacheを用いSQL実行前後の状況（usagecountごとのページ数）を取得する。
    - usagecountごとのページ数分布に大きく変動があった場合に Clock Sweep が発生と判断する。
  - DBのバッファサイズ大/小、Clock Sweepの発生有/無 の組み合わせの4パターンで処理時間を比較する。

検証#2 Clock Sweepの速度比較　2/4

- usagecount 状況による Clock Sweep 発生の判断

表：SQL実行後の usagecount ごとのページ数

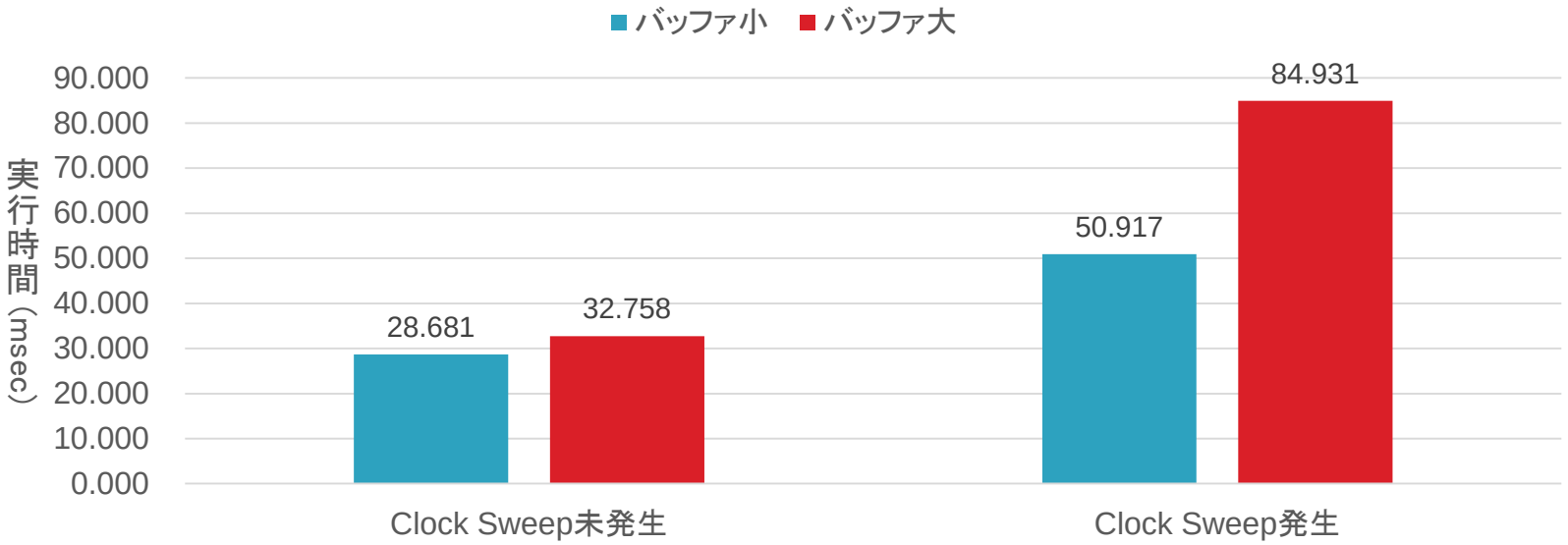
SQL 実行回数	SQL実行後のusagecount						
	空	0	1	2	3	4	5
102	3	0	103	4	0	1	524,177
103	2	0	104	4	0	1	524,177
104	1	0	105	4	0	1	524,177
105	0	0	106	4	0	1	524,177
106	0	108	2	1	37	524,071	69

① SQL実行回数が105回までは未使用のバッファが使用される
105回目の実行後、未使用のバッファが枯渇している

② 106回目のSQL実行で Clock Sweep が発生し、
usagecount ごとのページ数の分布が左に移動している
（500回のSQL実行で3回程度 Clock Sweep が発生）

検証#2 Clock Sweepの速度比較　3/4

- 計測結果



- 参照するデータは全てディスクからの読み込みとなるため20ミリ秒を超える実行時間となり、ディスク装置の状況やデータの存在位置によるブレも比較的大きく発生している。
- Clock Sweep発生時は、DBバッファ全体を参照するため
  未発生時と比べて処理時間が増加している。

検証#2 Clock Sweepの速度比較　4/4

- **考察**
  - バッファ小/大における Clock Sweep 発生による差分の比率は2.34であり、バッファサイズ（小：4GB、大：12GB）に比例している。
  - 検証では全ページを均一に参照しているため特定のusagecountにページ数が集中している。そのため追い出し対象のページを見つけ出すためにはDBバッファ全体を参照する必要があり、これはClock Sweepの影響が最大化するパターンとなる。
  - OLTPシステムにおいては高いusagecountよりも低いusagecountの方のページ数が少ない傾向がある。そのため、Clock SweepによりDBバッファ全体を参照するケースは少なく、処理時間に与える影響も小さいと考えられる。

検証#3 「DBのバッファ＋OSのキャッシュ」と「DBのバッファのみ」の読み込み速度比較 1/2

目的

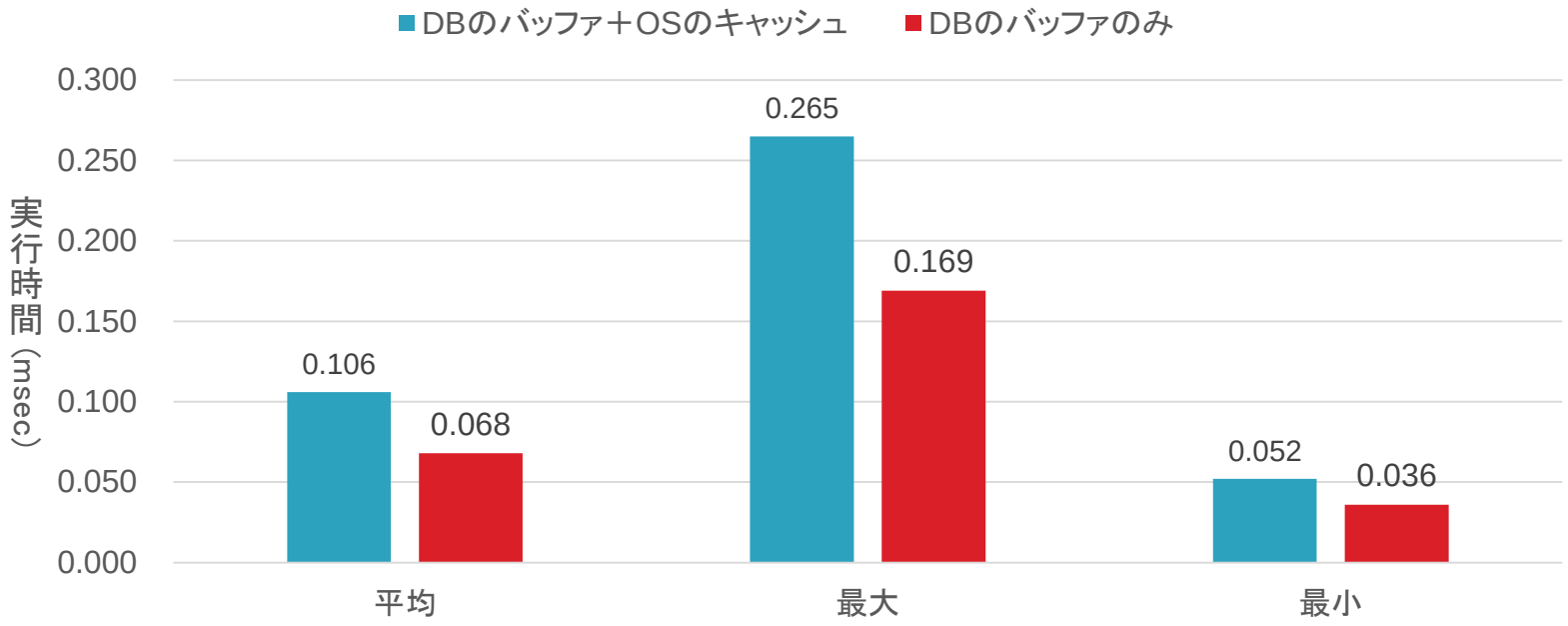
- OSのキャッシュからデータを参照する場合、DBのバッファから参照する場合と比べてどの程度速度が低下するか確認する。

計測内容

- 同一サイズのテーブルを次の状態になるようロードする。
  - バッファ小：DBのバッファとOSのキャッシュにそれぞれ50%
  - バッファ大：DBのバッファに全て
- ロードしたテーブルを1ページずつ参照する。
  - バッファ小ではOSのキャッシュに載ったデータを参照対象とする
  - バッファ大ではDBのバッファに乗ったデータを参照対象とする
  - 実行するSQL（SELECT文）の条件としてTIDを指定する
  - 実行ごとに参照するTIDの値をインクリメントする
  - SQL実行には psql を使用する。
  - EXPLAIN (ANALYZE, BUFFERS) を用い Execution Time を計測対象とする。

検証#3 「DBのバッファ＋OSのキャッシュ」と「DBのバッファのみ」の読み込み速度比較 2/2

■ 計測結果



■ 考察

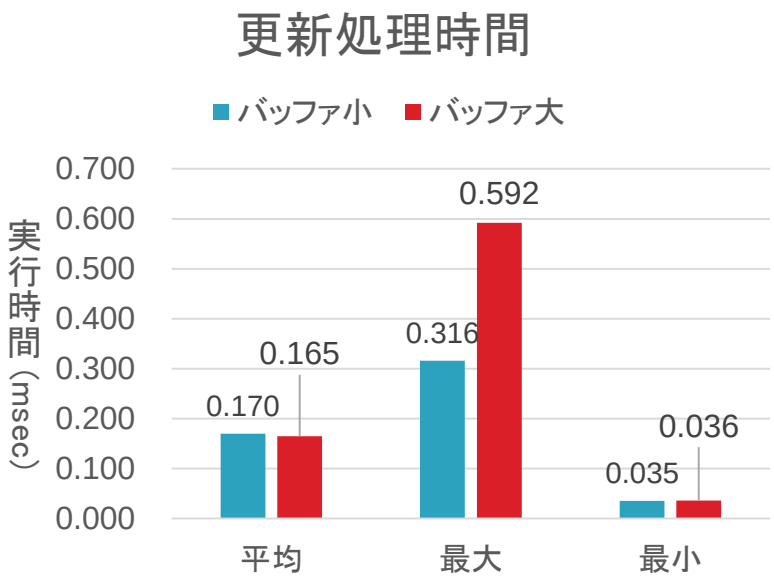
- 平均値で比較するとDBのバッファのみの方が0.038ミリ秒速い。
  OSのキャッシュからデータを参照する場合、Clock Sweep およびDBのバッファへの展開が発生するため、DBのバッファのみと比較すると遅くなるためである。
  しかしその差は非常に小さい（0.038ミリ秒）ため影響も小さいと考えられる。

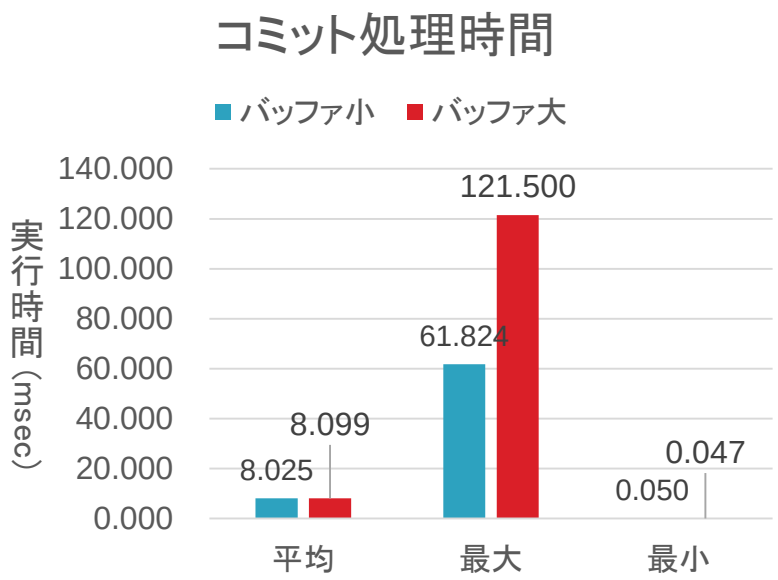
検証#4 コミットまでの速度の確認　1/2

- **目的**
  - DBのバッファサイズによって、
    コミットまでの処理時間に差が無いことを確認する。
- **計測内容**
  - 更新先ページが参照先ページと同じ更新を行う。
    - テーブルのfillfactorを50%とする。
    - 更新対象ページはDBのバッファに載せておく。
  - 更新時間とコミット処理時間を計測する。
    - 実行するSQL（UPDATE文）の条件としてTIDを指定する
    - 実行ごとに参照するTIDの値をインクリメントする
    - SQL実行には psql を使用する。
      - UPDATE 文には EXPLAIN (ANALYZE, BUFFERS) を用い Execution Time を計測対象とする。
      - COMMIT には ¥timing を使用し処理時間を計測する。

検証#4 コミットまでの速度の確認　2/2

■ 計測結果





■ 考察

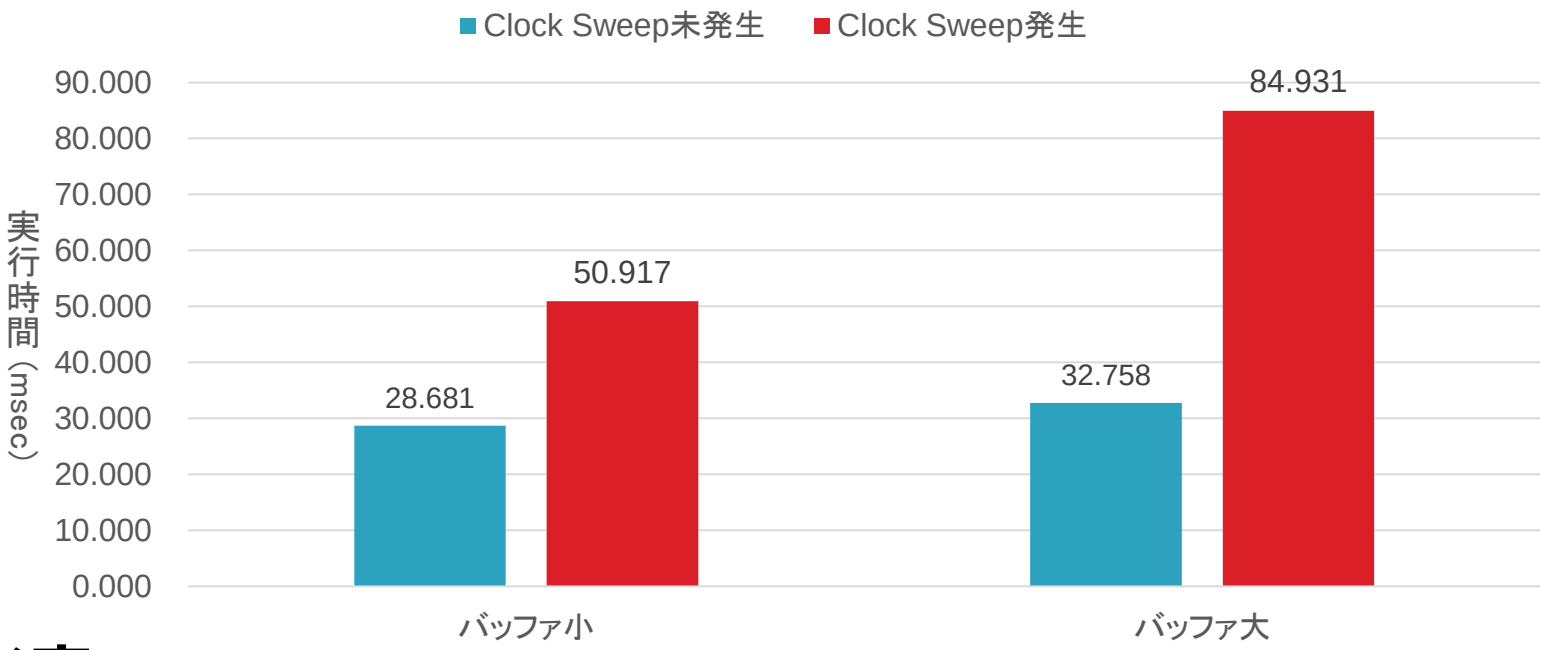
- 平均値で比較すると、更新/コミットの両方でほとんど差は見られない。
  一方でコミット処理時間の最大値に大きく差がみられる。これはWAL書き込みが必要となるためディスクの状況に依存することが原因と考えれる。
  そのため、コミットまでの処理時間はDBのバッファサイズに依存しないと言える。

検証#5 コミット以降の速度の確認　1/2

- **目的**
  - ダーティーページの内容をディスクに保存する処理（チェックポイント）のうち、ダーティーページを見つけ出す処理時間がDBのバッファのサイズの影響を受けることを確認する。
- **計測内容**
  - チェックポイント処理のうちDBバッファの参照にかかる時間を計測する。これは「検証２」の「Clock Sweep発生」と「Clock Sweep未発生」の差分に相当する

検証#5 コミット以降の速度の確認　2/2

■ 計測結果



■ 考察

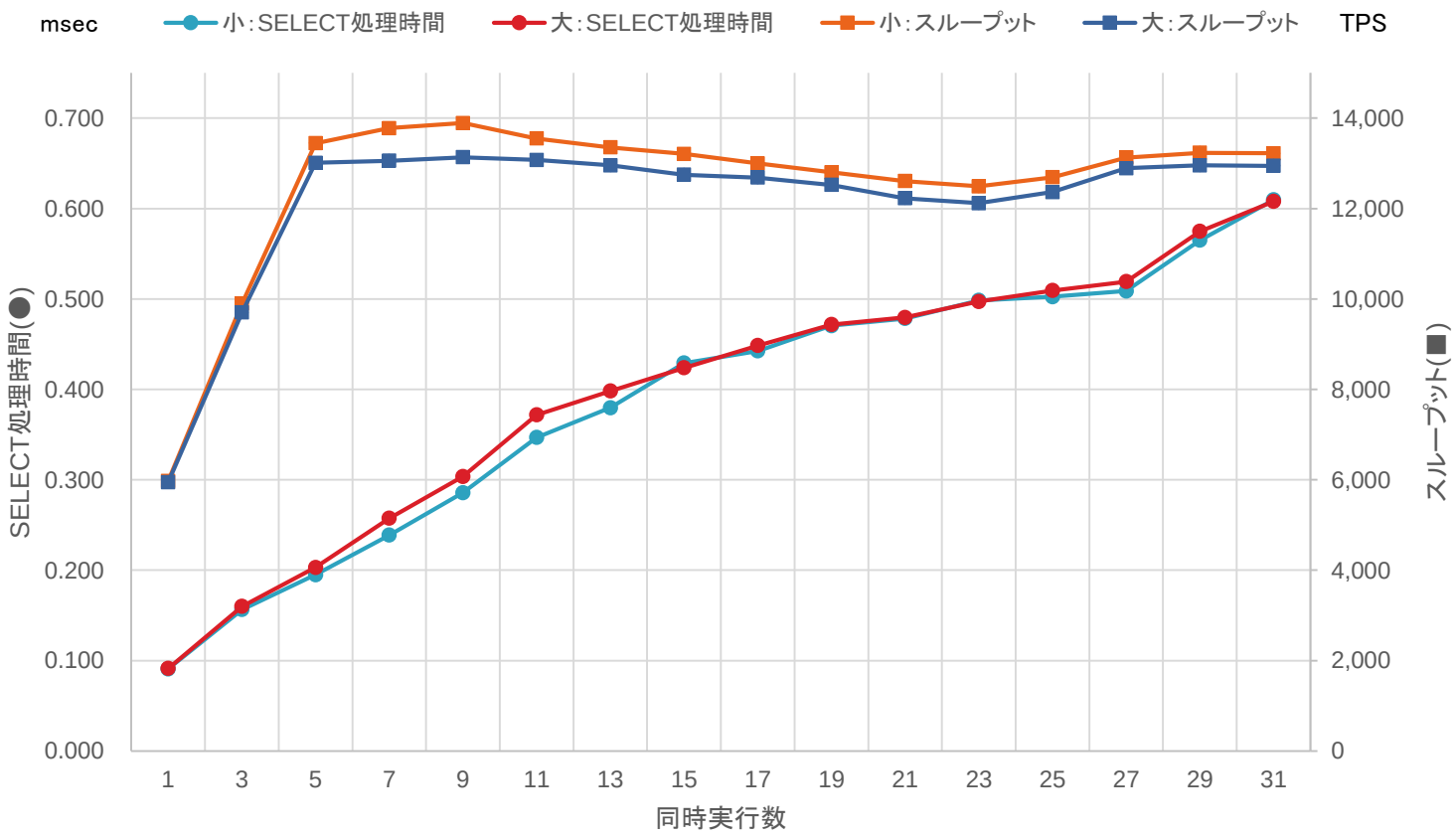
- Clock Sweep により DB のバッファ全体を参照する時間は DB のバッファサイズに比例する
- チェックポイントではディスクへの書き込みが発生し、書き込み処理が全体のほとんどを占めるため、影響は少ない

検証#6 並行実行時の影響　1/4

- **目的**
  - 読み込み処理を並行実行した場合に、DBのバッファサイズが処理時間とスループットに影響を与えないことを確認する。
- **計測内容**
  - 『検証#1-DBのバッファからの読込み速度比較』と『検証#2-「DBのバッファ＋OSのキャッシュ」と「DBのバッファのみ」の読込み速度比較』を同時実行数を増やして実施する。
  - 実施にはpgbenchを使用する。
    - 独自テーブルと独自スクリプトを使用
    - 独自テーブル：DBのバッファ内で処理を完結させるため、DBのバッファの90%のサイズを持つテーブルを用意し、事前にDBのバッファに展開しておく。検証時には1テーブルだけDBのバッファに展開された状態とする
    - 独自スクリプト：参同一ページを同時に参照することによる影響を極力避けるため、ランダムに1ページを参照するSELECT文を使用。ページの指定にはctidを条件に使用（今回作成したテーブルは1ページ内に複数タプルが存在するため）
    - 同時実行数を1から31まで変化させる

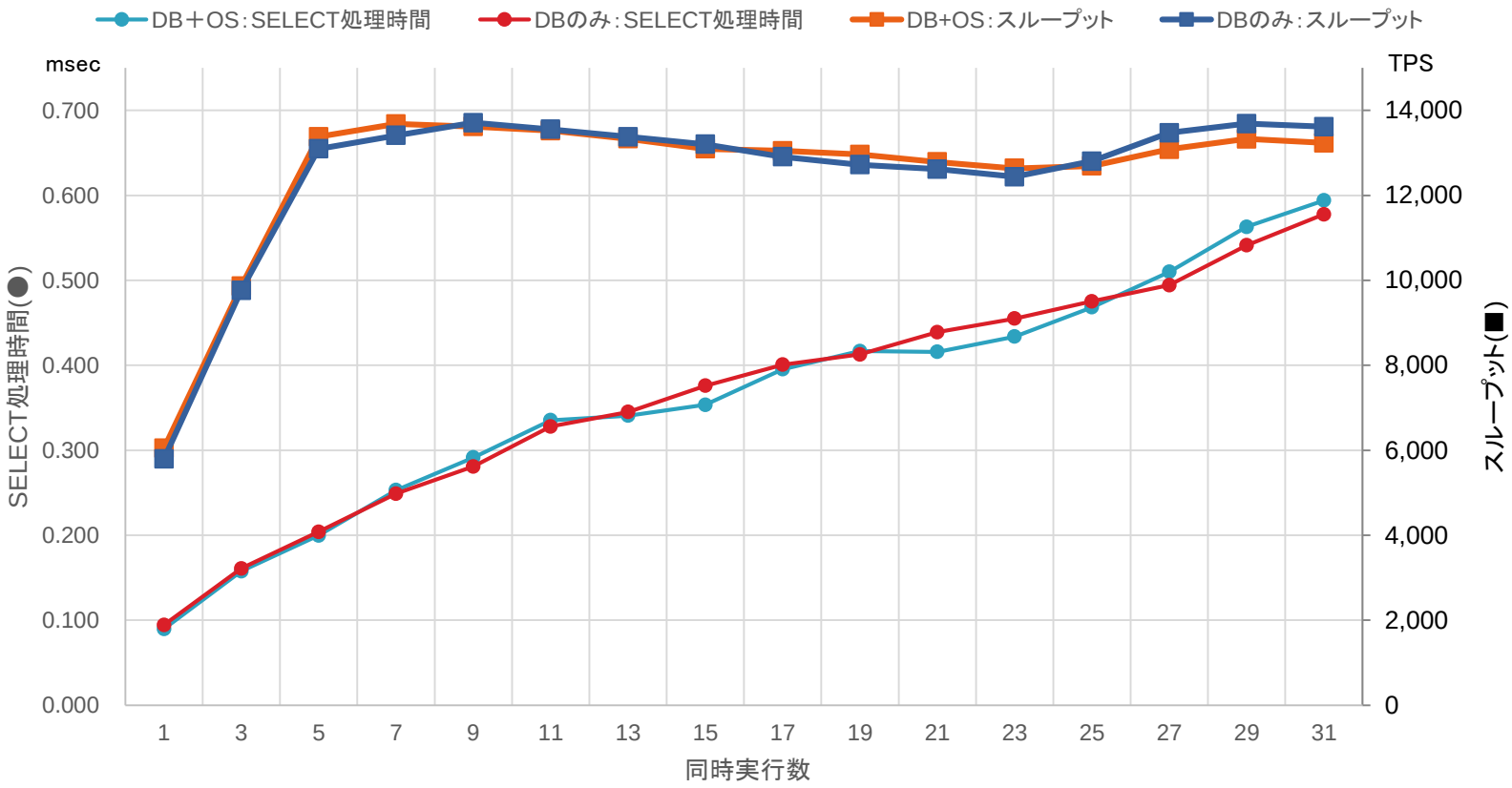
検証#6 並行実行時の影響　2/4

- 計測結果
  - 検証#1 DBのバッファからの読み込み速度比較



検証#6 並行実行時の影響　3/4

- 計測結果
  - 検証#2「DBのバッファ＋OSのキャッシュ」と「DBのバッファのみ」の読み込み速度比較



検証#6 並行実行時の影響　4/4

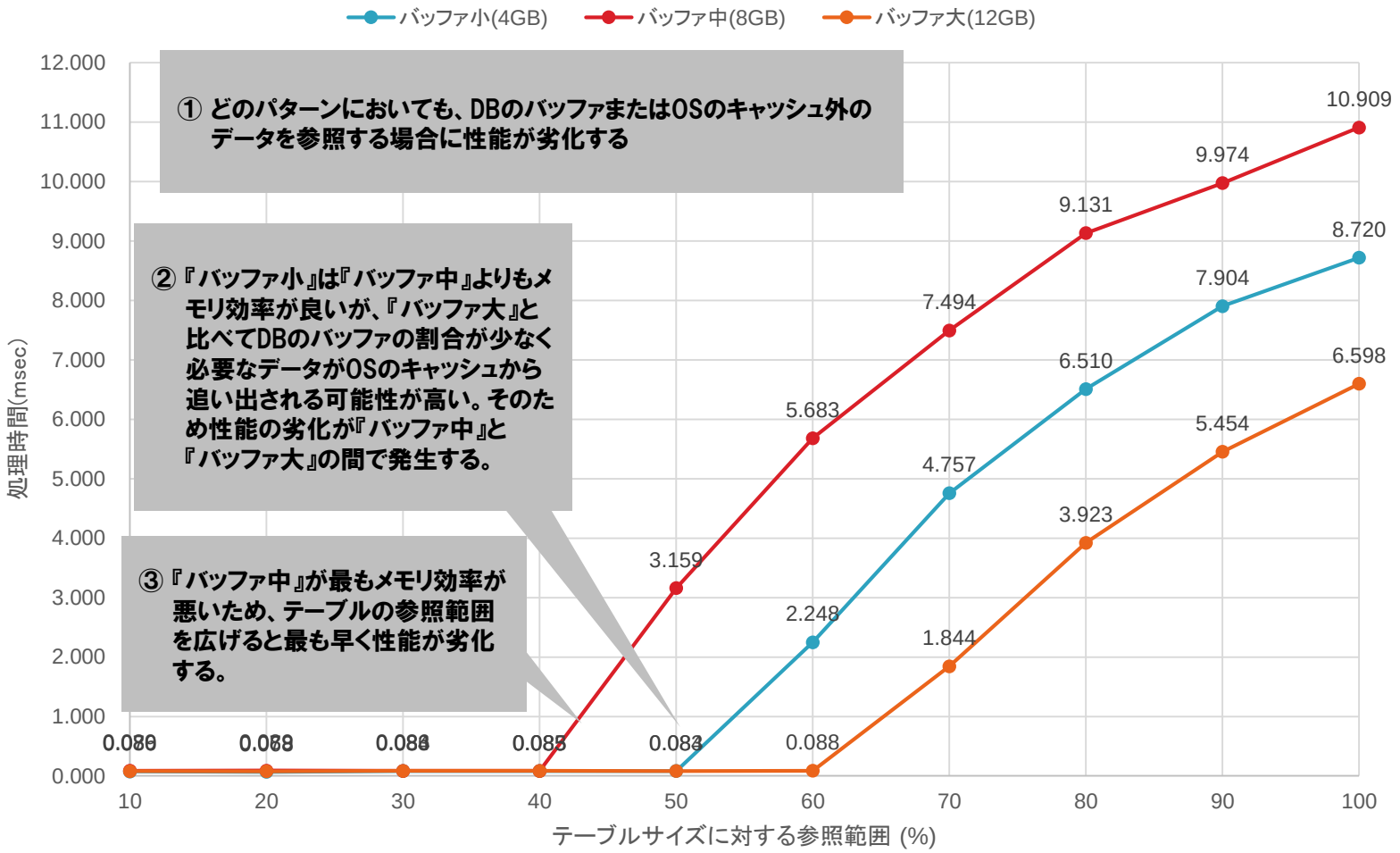
- 考察
  - 検証#1・検証#2の両方とも、DBバッファの大・小（「DBのバッファのみ」が大、「DBのバッファ＋OSのキャッシュ」がバッファのサイズ小）で並行実行数を増やした場合の処理時間とスループットの変化は同じであった。
    - コア数4の環境でクライアント・サーバが同一マシン上に存在するため同時実行数2を超えると処理時間が延びる

検証#7 ページ重複による無駄の確認　1/4

- **目的**
  - DBのバッファサイズによりメモリの利用効率が異なり、効率の差が性能差となることを確認する。
- **計測内容**
  - システムメモリを超えるサイズ（メモリサイズの1.2倍=19.2GB）のテーブルを、サイズ小/中/大のDBのバッファにロードする。
  - ロードしたテーブルの固定範囲内のデータに対して、ランダムに1ページずつアクセスする。
    - 固定する範囲は、10%～100%の間で10%毎に増加させる。
    - 実行するSQL（SELECT文）の条件としてTIDを指定する
    - 実行ごとに参照するTIDの値をランダムに変更する（bash の RANDOM 変数を使用）
  - 1回の試行で1,000回のSQLを実行。
    - SQL実行には psql を使用する。
    - EXPLAIN (ANALYZE, BUFFERS) を用い Execution Time を計測対象とする。

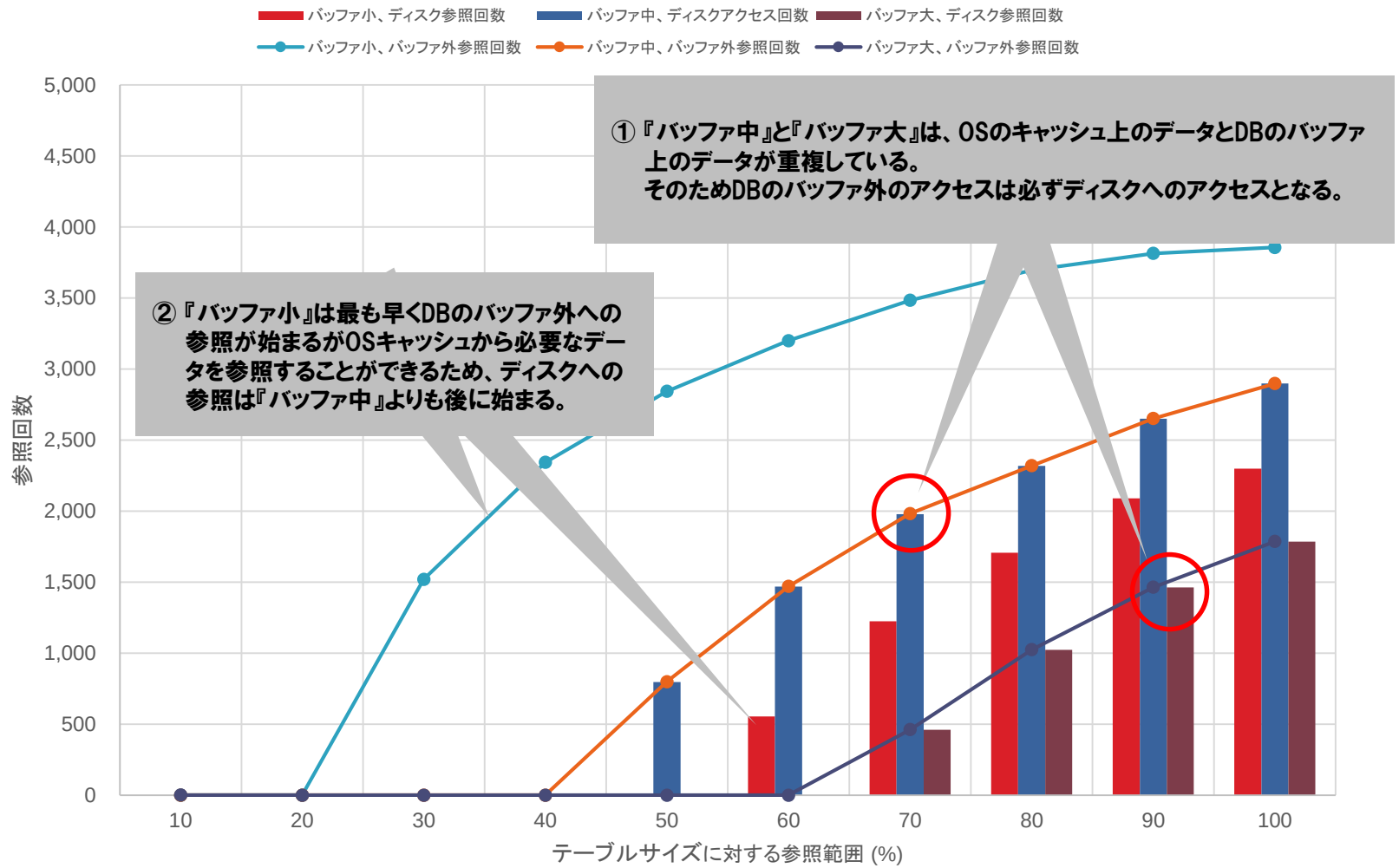
検証#7 ページ重複による無駄の確認　2/4

- 計測結果：平均処理時間



検証#7 ページ重複による無駄の確認　3/4

- 計測結果：DBのバッファ外とディスク参照回数



検証#7 ページ重複による無駄の確認　4/4

- 考察
  - ワークロードの参照範囲がメモリの利用効率内であれば
    DBのバッファサイズの違いによる処理時間に差はない。
    一方で利用効率を超える広範囲を参照対象とするワークロードの場合には、DBのバッファサイズの違いが処理時間に影響した。

    参照範囲が広範囲に及ぶワークロードの場合にはメモリの利用効率が処理時間に影響することに気を付ける必要があるが、特にPostgreSQLのバッファ戦略では OS のキャッシュを利用するため、メモリの利用効率はDBのバッファサイズ が

    『中』＜『小』＜『大』

    の順で高くなり、サイズ順にならないことに注意が必要である。

検証#8 テーブル全読み込み後のキャッシュヒット率の比較　1/3

目的

- DBのバッファサイズごとに、リングバッファを用いたテーブル全読み込みの影響を確認する。

計測内容

- テーブルを2つ用意する。
  - テーブル１：サイズはDBのバッファ小よりも大きくかつDBのバッファ小時のOSのキャッシュに全てのデータが載り、かつDBのバッファ大よりも小さい（8GB）
  - テーブル２：DBのバッファ小時のOSのキャッシュよりも大きい（5GB）
- テーブル１をロードし、DBのバッファ小・大のそれぞれを次の状態とする。
  - DBのバッファ小：DBのバッファにテーブルの半分が載り、OSのバッファに全てのデータが載る
  - DBのバッファ大：DBのバッファに全てのデータが載る
- テーブル１をロード後、テーブル２を Seq Scan で参照する
  - テーブル２はリングバッファによる参照となる
  - DBのバッファに存在するテーブル1のデータはほとんど追い出されない
  - OSのキャッシュは、テーブル1のデータが追い出され、テーブル２のデータのみとなる
- テーブル１のデータ全域を1ページずつ参照する。
  - 実行するSQL（SELECT文）の条件としてTIDを指定する。
  - 実行ごとに参照するTIDの値をインクリメントする。
  - SQL実行には psql を使用する。
  - EXPLAIN (ANALYZE, BUFFERS) を用い Execution Time を計測対象とする。

検証#8 テーブル全読み込み後のキャッシュヒット率の比較　2/3

- 計測結果

DBのバッファのサイズ	小	大
DBのバッファヒット率（%）	51.100	100.000
OSのキャッシュヒット率（%）	0.000	-
平均処理時間（msec）	9.966	0.076
DBのバッファアクセス時の平均処理時間（msec）	0.077	0.076
DBのバッファ外アクセス時の平均処理時間（msec）	20.303	-

- 処理時間が1ms以上の場合はディスクへのアクセスと判断している
  - OSのキャッシュを参照した場合1ms未満となることがこれまでの検証でわかっているため
- サイズ小のDBバッファヒット率が50%を超えているのはDBバッファ上のデータブロック数がテーブル全体のデータブロック数を少しだけ上回っていたため
  - 全体ブロック数: 1,024,000、DBバッファ上のブロック数: 524,131(51.18%)

検証#8 テーブル全読み込み後のキャッシュヒット率の比較　3/3

- 考察
  - DBのバッファ大ではディスクアクセスが発生しなかったが、バッファ小ではDBのバッファ外のデータ参照時にディスクアクセスが発生した。
  - DBのバッファでは、リングバッファによって大きなサイズのテーブル読み込みが発生した場合でも他のテーブルデータを追い出さない仕組みになっている。そのためデータの大部分をDBのバッファに依存する設定（DBのバッファサイズ大）ではその後の処理時間への影響は小さい。
    しかしOSのキャッシュにはそのような機能がないため、データの大部分をOSのキャッシュに依存する設定（DBのバッファサイズ小）ではテーブルデータの追い出しが発生し、その後のOSのキャッシュヒット率が小さくなる。その結果処理時間も劣化することとなる。
    そのためDBのバッファサイズ小の場合はテーブル全読み込み後の性能劣化に注意する必要がある。

参考資料

- PostgreSQL 13.1文書
  https://www.postgresql.jp/document/13/html/index.html
- 機械学習を用いたPostgreSQLパラメータのチューニング検証
  https://pgecons-sec-tech.github.io/tech-report/html_wg3_ml_tuning/wg3_ml_tuning.html
- Shared_Buffers DB パラメータのデフォルト値と Amazon RDS PostgreSQL および Aurora PostgreSQL の間に差がある理由を理解する
  https://aws.amazon.com/jp/premiumsupport/knowledge-center/rds-aurora-postgresql-shared-buffers/

PGECons

PostgreSQL Enterprise Consortium