PostgreSQLエンタープライズ・コンソーシアム 技術部会 WG#1

2017年度WG1活動報告書

目次

1. 改訂履歴

版	改訂日	変更内容
1.0	2018/04/12	新規作成

2. ライセンス

3. はじめに

3.1. PostgreSQLエンタープライズコンソーシアムとWG1について

PostgreSQLエンタープライズコンソーシアム(略称 PGECons) は、PostgreSQL本体および各種ツールの情報収集と提供、整備などの活動を通じて、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の普及を推進することを目的として、2012年4月に設立された団体です。

3.1.1. 各WGとWG1の活動テーマ

PGECons 技術部会ではPostgreSQLの普及を促進する技術的な課題の解決をテーマとし、「大規模基幹業務に向けたPostgreSQLの適用領域の明確化」を大きな目標に活動しております(2012年7月6日開催のPGEConsセミナーより)。

この目標に向けて技術部会では課題を以下の課題領域に分類し、三つのワーキング・グループ(WG)で分担することとしました。

表 3.1 PGEConsにおける課題領域

課題領域	概要	担当WG
性能	性能評価手法、性能向上手法、チューニングなど	1
互換性	データ、スキーマ、SQL、ストアドプロシージャの互換性	2
可用性	高可用クラスタ、BCP	3
保守性	保守サポート、トレーサビリティ	3
運用性	監視運用、バックアップ運用	3
セキュリティ	監査	3
接続性	他ソフトウェアとの連携	3

- WG1(新技術検証ワーキンググループ)
  - 新しい技術の実検証から有用性や課題を検討
  - PostgreSQLの最新バージョンの性能傾向の調査・検討
- WG2(移行ワーキンググループ)
  - ストアドプロシージャの移行技術に対する強化
  - 利用者観点で整備した「移行フレームワーク」に基づき、成果物のユーザビリティ向上、内容の最新化などを実施
- WG3(課題検討ワーキンググループ)
  - 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、現場の課題に向けた課題に対するテーマを設定

3.2. WG1の活動テーマ

WG1では、PostgreSQLの新技術に対する技術検証、および新バージョンにおける性能検証を進めるにあたり、以下のような観点から個々の検証テーマをたてて、参加企業が集まってチームを作って具体的な検討を進めています。

表 3.2 WG1の検証テーマ

検証観点	具体的なテーマの例
新機能の技術検証	文字列データやJSONなどの特定データ型に対する処理性能や利便性、VACUUMに代表される保守コマンドの改善など
性能向上機能	パラレルクエリ、パーティショニング、クエリキャッシュ、高速ロードなど
定点観測	最新バージョンの PostgreSQL に対して、前バージョンとの性能傾向の差異を把握する
スケールアップ	マルチコアCPUでのスケールアップ性検証。定点観測で実施する
スケールアウト	負荷分散クラスタでのスケールアウト性検証
性能検証手法	オンラインやバッチなどの業務別性能モデル、サイジング手法
性能チューニング	チューニングノウハウの整備、実行計画の制御手法

3.3. 本資料の概要と目的

本資料は、2017年10月にリリースされた PostgreSQL 10のスケール性、新技術・機能向上、Windows版PostgreSQL性能、および近年のPostgreSQLに関連する技術について調査と性能検証を実施し、その方法と結果を報告します。

3.3.1. 2017年度の活動テーマ

2017年度は、2017年10月5日にリリースされた PostgreSQL 10を対象として以下のようなテーマで検証を実施しました。

- 定点観測（スケールアップ）
- Windows版PostgreSQL性能
- パラレルクエリ
- 全文検索
- レプリケーション
- FDW

以下、各検証テーマごとの概要を紹介します。

3.3.2. 定点観測（スケールアップ）

例年通り、PostgreSQL 新旧バージョンにおけるスケール性能を比較する検証を行いました。参照系クエリのスケール性能は 9.6 と 10 で大きな差は見られませんでした。しかし、更新系クエリのスケール性能は 9.6 に比べ、10 では大きな性能低下が見られました。この結果を踏まえ。10における更新系クエリのスケール性能低下の原因を重点的に調査、検証しました。今回の結論として、原因は10で変更されたセマフォによるものと見ています。

3.3.3. Windows版PostgreSQL性能

Linux上で実施していた定点観測と同様の測定をWindows版PostgreSQLでも実施し、Linux版PostgreSQLと、Windows版PostgreSQLの性能の傾向を評価しました。同一スペックのサーバ上でLinux/WindowsのOSの差異による測定を実施した結果、過負荷状態でなければ、Windows版PostgreSQLは性能面ではLinux版と遜色はないと考えられます。また、特定のPostgreSQLパラメータについて、Windows版固有のチューニングが必要か検証しました。shared_buffersについては、Linux版のチューニング方針とは異なり、参照主体のモデルでは、非常に小さな値を設定しても性能上の問題はないという結果となりました。wal_sync_methodについては、open_datasyncを設定することが妥当であるという結果となりました。

3.3.4. パラレルクエリ

PostgreSQL 9.5までは、メニーコア環境であっても1つのクエリでは1つのコアしか使用されませんでした。商用ソフトウェアでは以前から実装されていて、PostgreSQLでも長年実装が待ち望まれてきたメニーコアを有効活用するパラレルクエリが、バージョン9.6で実装され、バージョン10で強化されました。今回は、BIのようなOLAPにおいて使用される形式のクエリを模したベンチマークテストである TPC-DSで定義されているクエリを題材とし、OLAPにおいてバージョン9.5～10で、パラレルクエリによりどのように性能が変化するのかを検証しました。検証の結果、扱うクエリによってパラレルクエリのパフォーマンスは、パラレルクエリの外側にある要因に左右される可能性があることが明らかになりました。特に、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しないかの確認、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し、効果が十分に得られるかどうかの確認が必要であるものと考えます。

3.3.5. 全文検索

近年の全文検索のユースケースでは、静的なデータを検索するだけではなく、データが継続して変更されているなかで検索を行う事が多くなっています。このユースケースでは、同一のDBに対して更新と検索が同時に発生するため、通常では十分なパフォーマンスが得られないと予想されることから、今回PostgreSQL10から実装されたロジカルレプリケーションを使用した負荷分散を行い、十分なパフォーマンスが得られるかどうか検証しました。検証の結果、MasterとSlaveに別々のインデックスを設定できるロジカルレプリケーションを利用した負荷分散構成をつくることで、全文検索用のインデックスによるデータ更新性能の劣化を起こさずに、全文検索用のインデックスを利用した高速な全文検索が可能になることが明らかになりました。

3.3.6. レプリケーション

PostgreSQLでは、9.0からレプリケーション機能を実装しており、継続的に機能強化が行われています。9.6からは、2台以上の複製先に対してデータを同期しながら更新が行える、複数同期レプリケーションが利用可能となりました。本テーマでは、PostgreSQLにおけるレプリケーション機能について、同期レプリケーション・非同期レプリケーション、またそれらが混在する構成において、どのように性能が変化するのかを検証しました。本検証結果から、同期のみの構成は非同期のみの構成に比べ約1/2の性能となること、同期・非同期混在の構成では総台数が同じでも同期の台数を増やすことで性能の低下が見られること、同期・非同期の台数を変化させることである程度一定の割合で性能値も変化することが確認できました。

3.3.7. Foreign Data Wrapper

PostgreSQLから外部のデータへアクセスする為の仕組みであるFDW（Foreign Data Wrapper）機能は点在する様々なデータをPostgreSQLから一括して管理する事ができる為、企業のデータ活用に非常に有望視されていますが、一方で性能的なオーバヘッドに対する懸念もあります。本検証では、PostgreSQL 10系のFDW機能に注目して、外部テーブルの有無を条件として参照系、更新系における性能オーバヘッドを明らかにしました。今回の検証では、FDWを利用しない場合と比較してFDWを利用する場合は参照系では最低で4%、更新系では最低で66%のスループットとなりました。

3.4. 成果の公開

本資料も含めて、活動成果は報告書の形にまとめて、PGEConsのWebサイト で公開しています。また、過去の活動成果も含めてテーマ毎に報告書を検索できるように成果物総索引 も用意しています。

3.4.1. 2016年度までの活動テーマの紹介

2012年度は、企業システムで使われる機器構成で、PostgreSQLのスケールアップ、スケールアウトによる性能特性、性能限界を検証しました。企業システムへのPostgreSQL採用や、システム構成を検討するための、一つの指針として「2012年度WG1活動報告書」として情報を公開しています。

2013年度は、2012年度に引き続き2013年9月9日にリリースされたPostgreSQL 9.3を対象とした スケールアップの定点観測を実施、PostgreSQL 9.3新機能による性能影響も合わせて評価することとしました。また、更新スケールアウト構成が可能なPostgres-XCの測定パターンを変えた再測定により、最適な利用指針を探る評価を

実施することとしました。さらに、2013年度の新たな取り組みとしてデータベースの性能向上に着目、データベースのI/O負荷分散機能であるパーティショニングや、ハードウェアを活用した性能向上の検証を実施しました。

2014年度は、2013年度に引き続き2014年12月7日にリリースされたPostgreSQL 9.4を対象とした スケールアップの定点観測を実施、そして9.4新機能のWAL改善を評価するために 更新系処理を新たに評価することとしました。また、新たな取り組みとして物理環境以外の環境におけるデータベースの性能評価に着目、KVMを使った仮想化環境と、LinuxコンテナのDocker環境の検証を実施しました。

2015年度は、2016年1月7日にリリースされた PostgreSQL 9.5を対象とした、スケールアップの定点観測を実施しました。これは昨年度に引き続き、参照系・更新系の双方で性能を調査するものです。PostgreSQL 9.5での2つの新機能、BRINインデックスとParallel Vacuumについて、その利用ノウハウが得られるような検証を行い ました。最後に、基盤となるLinux OSの主要なディストリビューションの一つである Red Hat Enterprise Linux 6と7とでPostgreSQLの性能を比較しました。

2016年度は、2017年9月にリリースされた PostgreSQL 9.6を対象とした、スケールアップの定点観測を実施しました。これは例年通り、参照系・更新系の双方で性能を調査するものです PostgreSQL 9.6の新機能のパラレルクエリや改善が行われたVACUUMについて効果を検証しました。また、Pgpool-IIのスケールアウト性能、JSON/JSONB データ型に関する性能評価を行いました。

3.5. 実施体制

2017年7月20日に開催された2017年度第1回技術部会より、以下の体制で実施しています(企業名順)。

表 3.3 2017年度WG1参加企業

SRA OSS, Inc. 日本支社
NECソリューションイノベータ株式会社
NTTテクノクロス株式会社
株式会社クリアコード
日本電信電話株式会社
富士通株式会社

この中で、SRA OSS, Inc. 日本支社は、「主査」としてWG1の取りまとめ役を担当することになりました。

3.6. 実施スケジュール

2017年度は、下記スケジュールで活動しました。

表 3.4 実施スケジュール

活動概要	スケジュール
WG1スタート	2017年7月20日
実施計画策定	2017年7月～12月
検証実施	2018年1月～2018年2月
2017年度WG1活動報告書作成	2018年2月～2018年4月
総会と成果報告会	2018年5月8日(東京)、2018年5月23日(大阪/報告会のみ実施)

3.7. 謝辞

検証用の機器を日本ヒューレット・パッカード株式会社、日本電気株式会社および富士通株式会社(敬称略)よりご提供いただきました。この場を借りて厚く御礼を申し上げます。

4. 定点観測(スケールアップ検証)

4.1. 検証概要

WG1では、PostgreSQLの新バージョン・新リリースにあわせて、新旧バージョンの性能比較やスケールアップ特性の検証を主な目的にした定点観測を2012年度から実施してきました。2014年度からは、それまでの参照処理に加えて更新処理についても検証を実施し、検証結果の公開を行うようになりました。2017年度は32コアのCPU（Xeon Gold 6142）、メモリ384GBといったスペックのサーバで、最新のPostgreSQLバージョン 10 と前バージョンの 9.6 とのスケールアップ参照性能の比較および更新性能の比較を行いました。

4.2. pgbench とは

本検証では、pgbench というベンチマークツールを使用しました。

pgbenchはPostgreSQLに付属する簡易なベンチマークツールです(バージョン9.5より前はcontribに付属)。標準ベンチマークTPC-B（銀行口座、銀行支店、銀行窓口担当者などの業務をモデル化）を参考にしたシナリオに基づくベンチマークの実行のほか、検索クエリのみを実行するシナリオも搭載されています。また、カスタムスクリプトを用意することで、独自のシナリオでベンチマークを実行することも可能です。

pgbenchでベンチマークを実行すると、以下のように1秒あたりで実行されたトラザクションの数(TPS: Transactions Per Second)が出力されます。なお、「including connections establishing」はPostgreSQLへの接続に要した時間を含んだTPSを、「excluding connections establishing」はこれを含まないTPSを示します。

```
transaction type: TPC-B (sort of)
scaling factor: 10
query mode: simple
number of clients: 10
number of threads: 1
number of transactions per client: 1000
number of transactions actually processed: 10000/10000
tps = 85.184871 (including connections establishing)
tps = 85.296346 (excluding connections establishing)
```

pgbenchには「スケールファクタ」という概念があり、データベースの初期化モードでpgbenchを起動することにより、任意のサイズのテスト用のテーブルを作成できます。デフォルトのスケールファクタは1で、このとき「銀行口座」に対応する「pgbench_accounts」というテーブルで10万件のデータ、約15MBのデータベースが作成されます。

以下に、各スケールファクタに対応するデータベースサイズを示します。

表 4.1 スケールファクタに対するデータベースサイズ

スケールファクタ	データベースサイズ
1	15MB
10	150MB
100	1.5GB
1000	15GB
5000	75GB

初期化モードではpgbench_accounts の他にもテーブルが作成されます。作成されるテーブルのリストを以下に示します。

表 4.2 pgbench_accounts(口座)

列名	データ型	コメント
aid	integer	アカウント番号(主キー)
bid	integer	支店番号
abalance	integer	口座の金額
filler	character(84)	備考

表 4.3 pgbench_branches(支店)

列名	データ型	コメント
bid	integer	支店番号
bbalance	integer	口座の金額
filler	character(84)	備考

表 4.4 pgbench_tellers(窓口担当者)

列名	データ型	コメント
tid	integer	担当者番号
bid	integer	支店番号
tbalance	integer	口座の金額
filler	character(84)	備考

スケールファクタが 1 の時、pgbench_accounts は 10 万件、pgbench_branches は 1 件、pgbench_tellers は 10 件のデータが作成されます。スケールファクタを増やすとこれに比例して各テーブルのデータが増えます。

pgbench には、様々なオプションがあります。詳細は PostgreSQL文書 をご覧ください。ここでは、本検証で 使用している主なオプションのみを説明します。

表 4.5 ベンチマークテーブル初期化

オプション	説明
-i	ベンチマークテーブルの初期化
-s	スケールファクタ
-F	作成するテーブルのフィルファクタ

表 4.6 ベンチマークの実行

オプション	説明
-c	同時接続クライアント数
-j	pgbench 内のワーカスレッド数
-T	ベンチマークを実行する秒数
-n	実行前にバキュームを行わない
-P	指定した秒数ごとのレポートを表示
-r	実行クエリの平均レイテンシを表示

4.3. 検証構成

4.3.1. ハードウェア構成

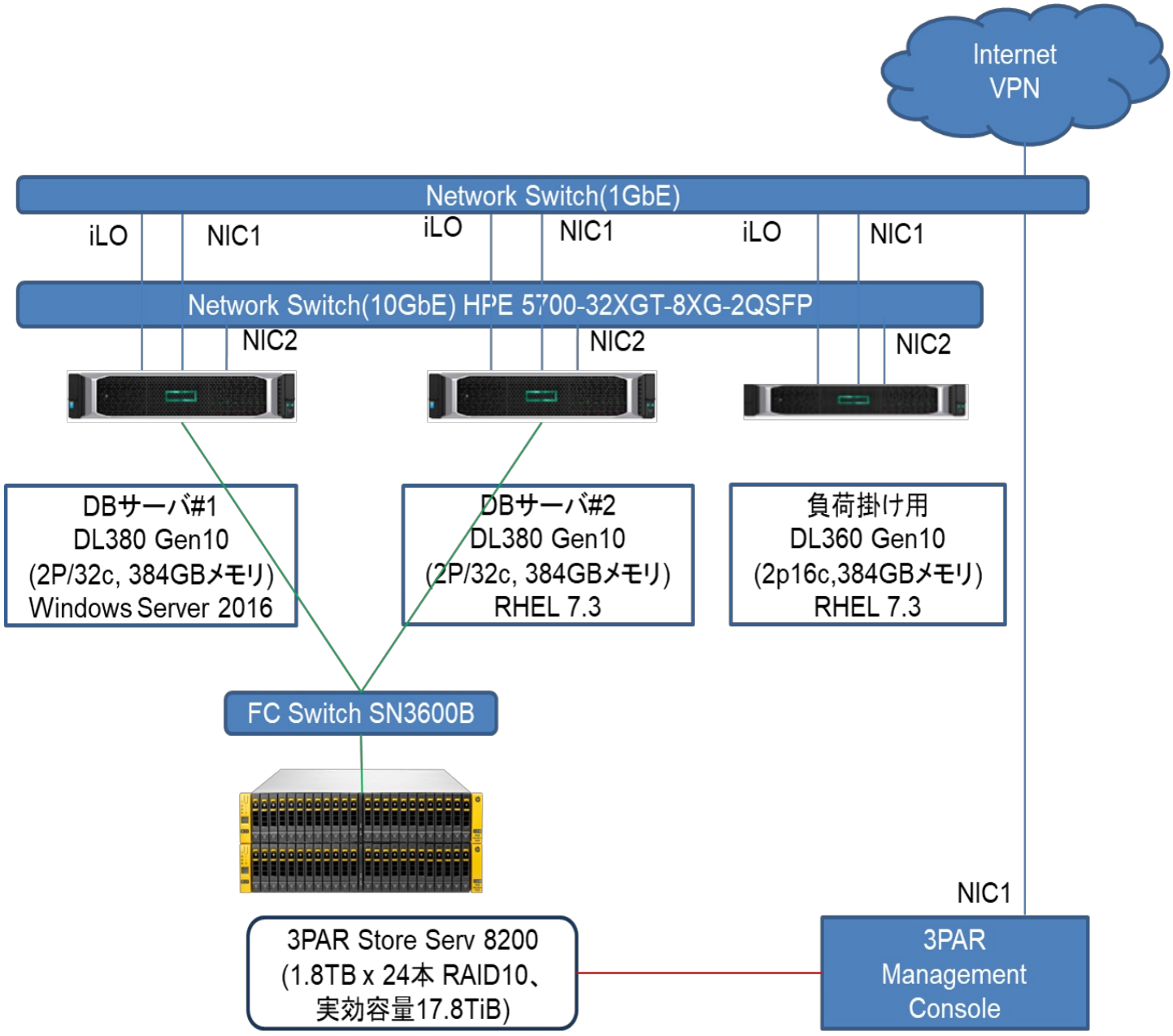


図 4.1 検証ハードウェア構成

本検証では上図構成のうち「DBサーバ#2」と「負荷掛け用」を用いました。

4.3.2. ソフトウェア構成

検証環境のソフトウェア構成を示します。

表 4.7 DBサーバ#2

OS	Red Hat Enterprise Linux 7.3
PostgreSQL	9.6.6, 10.1

表 4.8 負荷掛け用

OS	Red Hat Enterprise Linux 7.3
pgbench	10.1

4.3.3. カーネル設定

- OS デフォルト
- ハイパースレッディングは利用しない

4.3.4. PostgreSQL 設定

PostgreSQL設定ファイル postgresql.conf での設定は以下の通りです。これらは参照系検証と更新系検証で共通です。

```
listen_addresses = '*' ... 負荷掛け用マシンからの接続用
max_connections = 500 ... 多めに設定
shared_buffers = 40GB
work_mem = 1GB
maitenance_work_mem = 20GB
checkpoint_timeout = 60min
max_wal_size = 160GB
```

4.3.5. 環境

以下の手順で、データベースクラスタを作成しました。

initdbでデータディレクトリを作成し、上記に示した設定をpostgresql.confに記述します。

```
$ initdb --no-locale --encoding=utf-8
$ vi $PGDATA/postgresql.conf
```

PostgreSQLを起動してベンチマーク用のデータベースを作成します。

```
$ pg_ctl start -w
$ createdb [dbname]
```

4.4. 検証方法(参照系)

pgbenchコマンドを用いて、ベンチマーク用データベースをスケールファクタ2000で初期化します。

```
$ pgbench -i -s 2000 [dbname]
```

本検証では pg_prewarm モジュールを用います。

pg_prewarm はバッファキャッシュにテーブルデータを読み込むためのモジュールで、バッファキャッシュがクリア されているデータベース起動直後の性能低下状態を解消するために用いることができます。

まず、測定スクリプト実行前に pg_prewarm を実行します。これによりテーブルデータはすべてバッファキャッシュに格納されます。

```
=# SELECT pg_prewarm('pgbench_accounts');
```

以下のスクリプトを参照系カスタムスクリプトとして作成して、適度な負荷がかかるようにしました。これは、pgbench の標準 シナリオ（pgbench -S）ではCPUに充分な負荷がかからないためです。具体的には、ランダムに10000行を取得 しています。

```
\set naccounts 100000 * :scale
\set row_count 10000
\set aid_max :naccounts - :row_count
\setrandom aid 1 :aid_max

SELECT count(abalance) FROM pgbench_accounts WHERE aid BETWEEN :aid and :aid + :row_count;
```

これを、クライアント用検証機から

```
$ pgbench -n -h [host] -p [port] -c [clients] -j [threads] -f [参照系カスタムスクリプト] -T 300 -s 2000 -P 1
-r [dbname]
```

として実行しました。SELECTのみであるためVACUUMを実行せず(-n)、pgbench クライアント数(-c)とスレッド数(-j)を変動させながら、300秒ずつ(-T)実行しています。スレッド数はクライアント数の半分としています。スケールファクタ(-s)にはデータベース初期化時と同じ2000を指定します。また、1秒毎の進捗レポート取得(-P)と各クエリの平均レイテンシ(-r)も取得する指定にしました。(ただし、-P, -r の指定によって得られた結果は今回の考察に直接用いることはありませんでした)

計測はクライアント数ごとにそれぞれ3回ずつ実行し、その中央値を結果とします。また、変動させるクライアント数は{1 2 4 8 16 32 48 64 96} です。

4.5. 検証結果(参照系)

検証結果のTPSのグラフを以下に示します。9.6, 10 はともにクライアント数増加につれてTPSが増加し、コア数を超えた辺りのクライアント数でTPSは頭打ちとなっています。10のTPSはほとんどクライアント数においてわずかに9.6を上回る結果となりました。

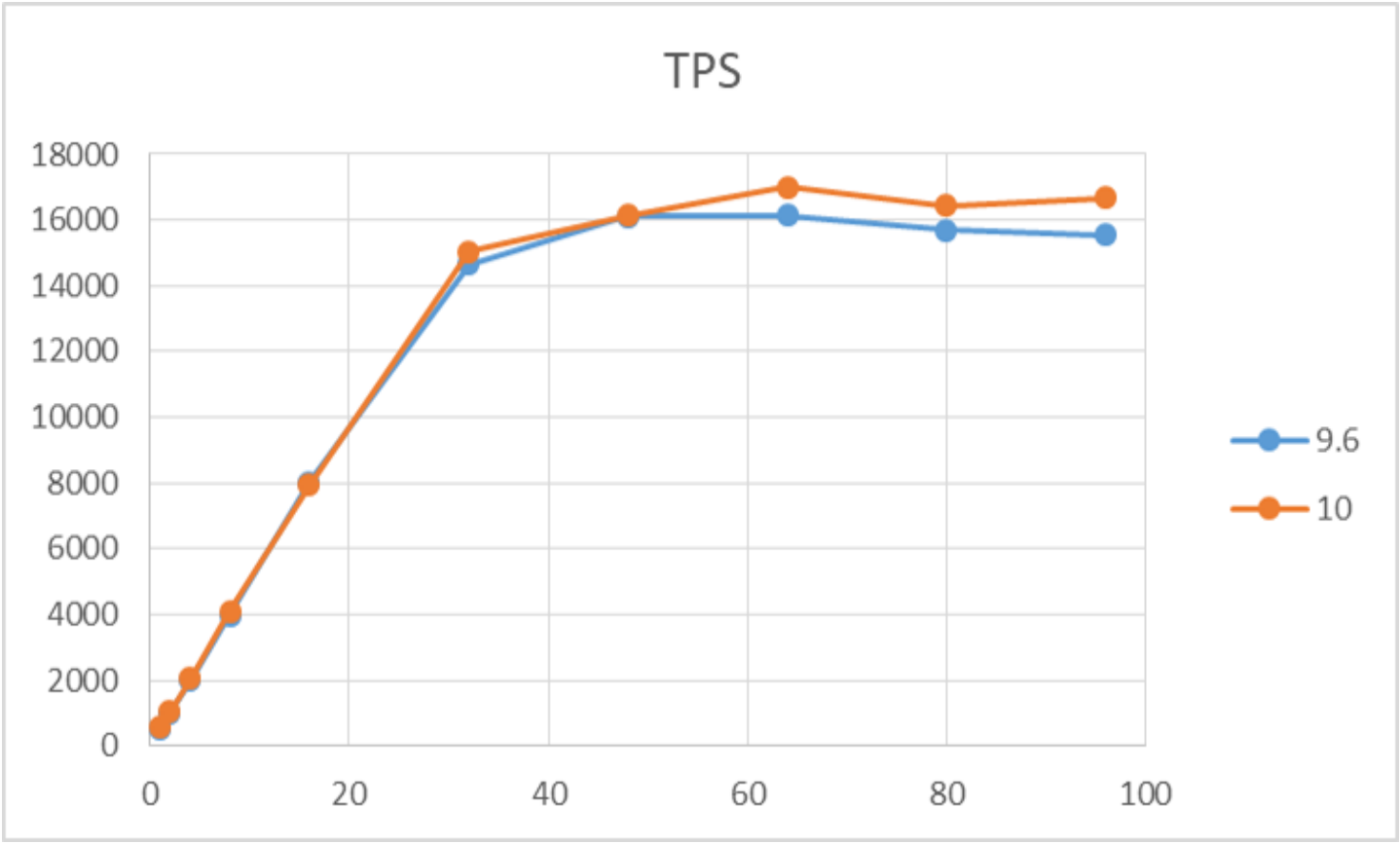


図 4.2 各クライアント数に対する TPS (参照系)

4.6. 検証方法(更新系)

pgbenchコマンドを用いて、ベンチマーク用データベースをスケールファクタ2000で初期化します。このとき、フィルファクタは 80 とします。

```
$ pgbench -i -s 2000 [dbname] -F 80
```

更新系検証でも、pg_prewarm モジュールを用います。測定スクリプト実行前に pg_prewarm を実行します。これによりテーブルデータはすべてバッファキャッシュに格納されます。

```
=# SELECT pg_prewarm('pgbench_accounts');
```

以下のスクリプトを更新系カスタムスクリプトとして作成して、適度な負荷がかかるようにしました。

```
\set naccounts 100000 * :scale
\set aid_val random(1, :naccounts)
UPDATE pgbench_accounts SET filler=repeat(md5(current_timestamp::text),2) WHERE aid = :aid_val;
```

これを、クライアント用検証機から

```
$ pgbench -n -h [host] -p [port] -c [clients] -j [threads] -f [更新系カスタムスクリプト] -T 300 -s 2000 -P 1
-r [dbname]
```

として実行しました。VACUUMを実行せず(-n)、pgbench クライアント数(-c)とスレッド数(-j)を変動させながら、300秒ずつ(-T)実行しています。スレッド数はクライアント数の半分としています。スケールファクタ(-s)にはデータベース初期化時と同じ2000を指定します。また、1秒毎の進捗レポート取得(-P)と各クエリの平均レイテンシ(-r)も取得する指定にしました。(ただし、-P, -r の指定によって得られた結果は今回の考察に直接用いることはありませんでした)

計測はクライアント数ごとにそれぞれ3回ずつ実行し、その中央値を結果とします。また、変動させるクライアント数は{1 2 4 8 16 32 48 64 96} です。このとき、更新系検証の場合は実行後のテーブルを使い回さず、毎回初期化しています。

4.7. 検証結果(更新系)

検証結果のTPSのグラフを以下に示します。9.6, 10 はともにクライアント数増加につれてTPSが増加しますが、コア数を超えた辺りのクライアント数から差が現れ、最大TPSは9.6が10を大きく上回る結果となりました。

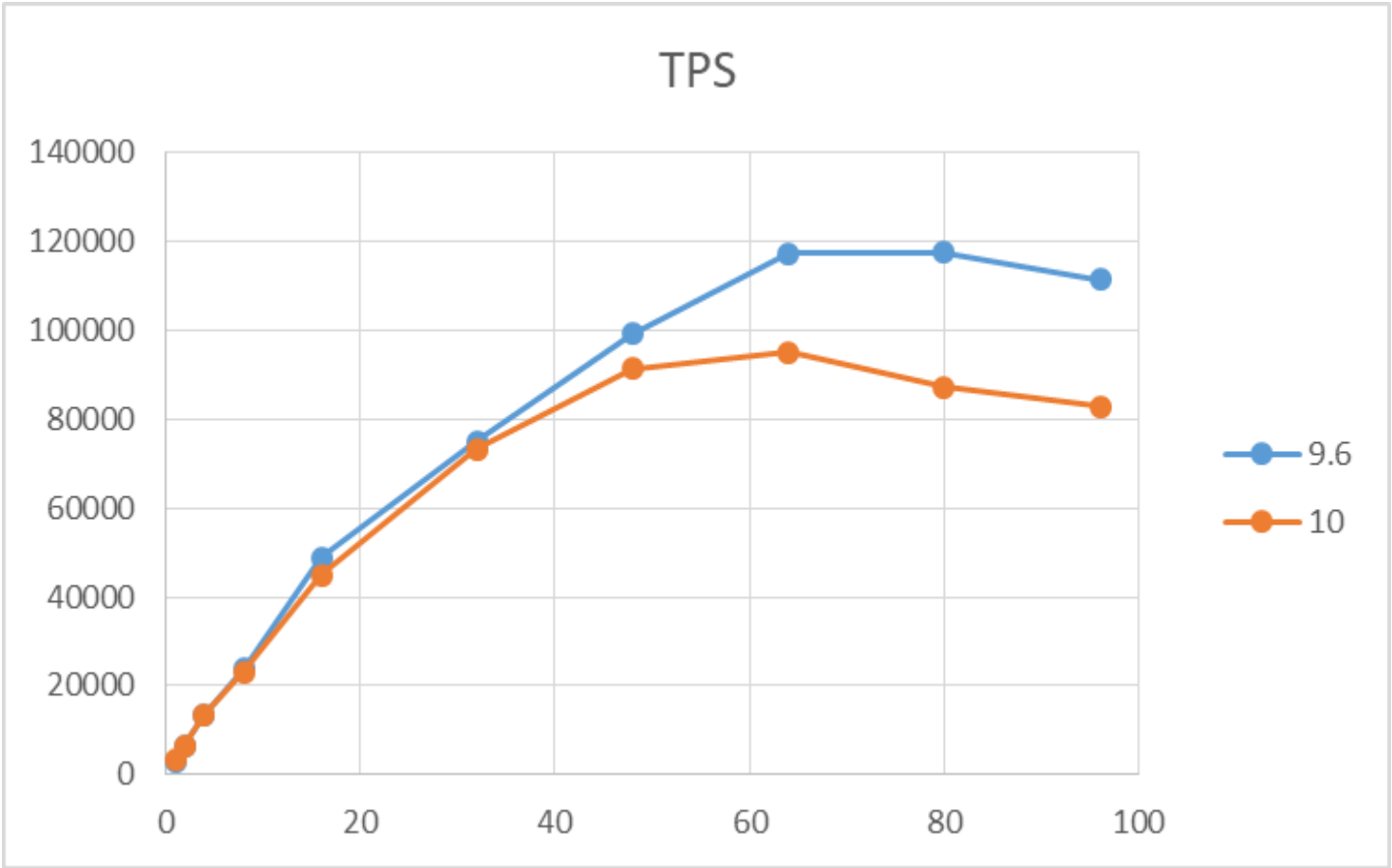


図 4.3 各クライアント数に対する TPS (更新系)

10の更新性能劣化を解明するために別途、追加の検証を行いました。

4.8. 追加検証

4.8.1. システムコール調査

各バージョンの postgres プロセスが実行するシステムコールを調査しました。strace コマンドを用いることで、実行したシステムコールを取得できます。strace コマンドはデフォルトでは実行されたシステムコールの詳細を示しますが、-c オプションを用いることでシステムコールの集計を行うことが可能です。

また、-f オプションを用いることで集計の対象に全子プロセスを含めています。（ただし、バックエンドプロセス以外も拾ってしまっています）

9.6 と 10 で大きく差の出た 80 クライアント接続指定の pgbench を対象に、pgbench 実行開始直後にシステムコール情報取得を開始しています。

```
$ cat ${PGDATA}/postmaster.pid | head -n1 | xargs -I@ strace -p @ -f -c -o《出力先ファイル》
```

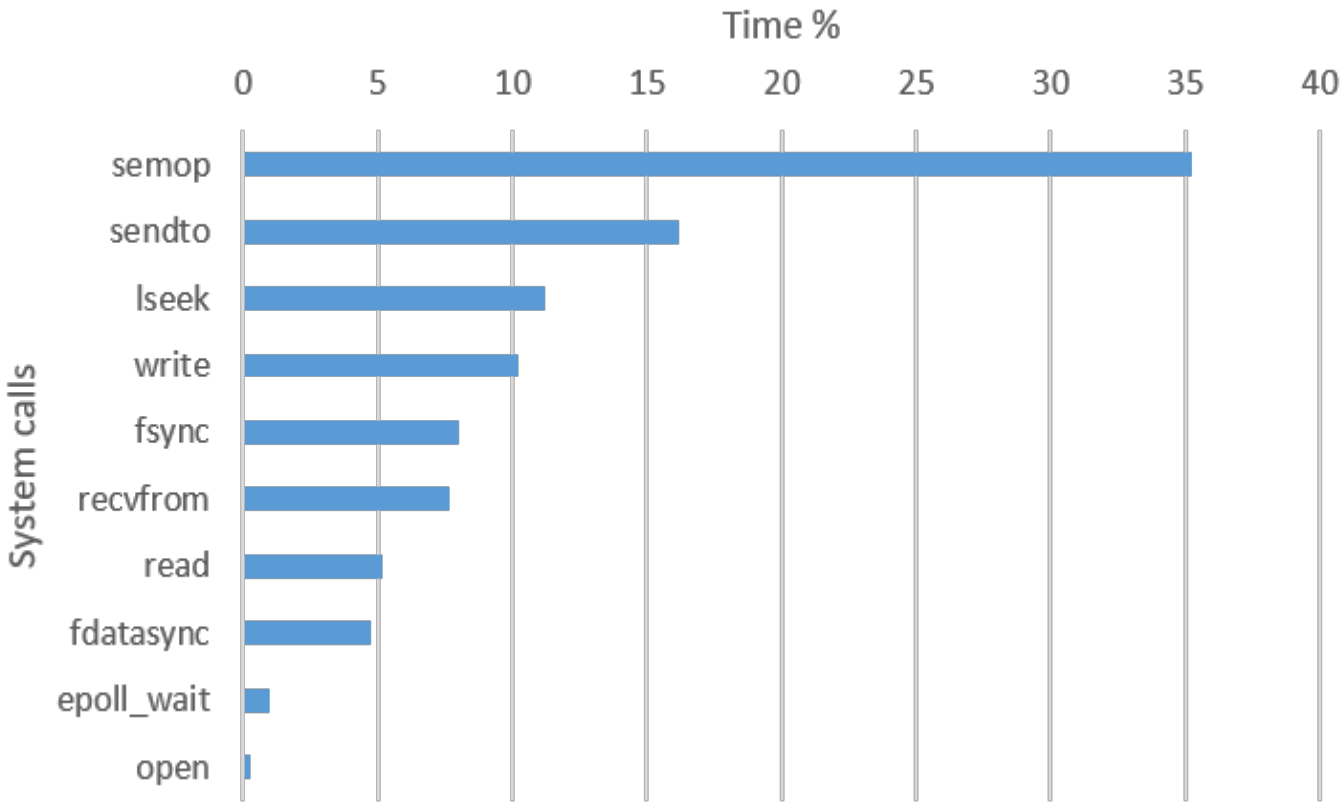


図 4.4 9.6 の postgres プロセスが使用するシステムコール

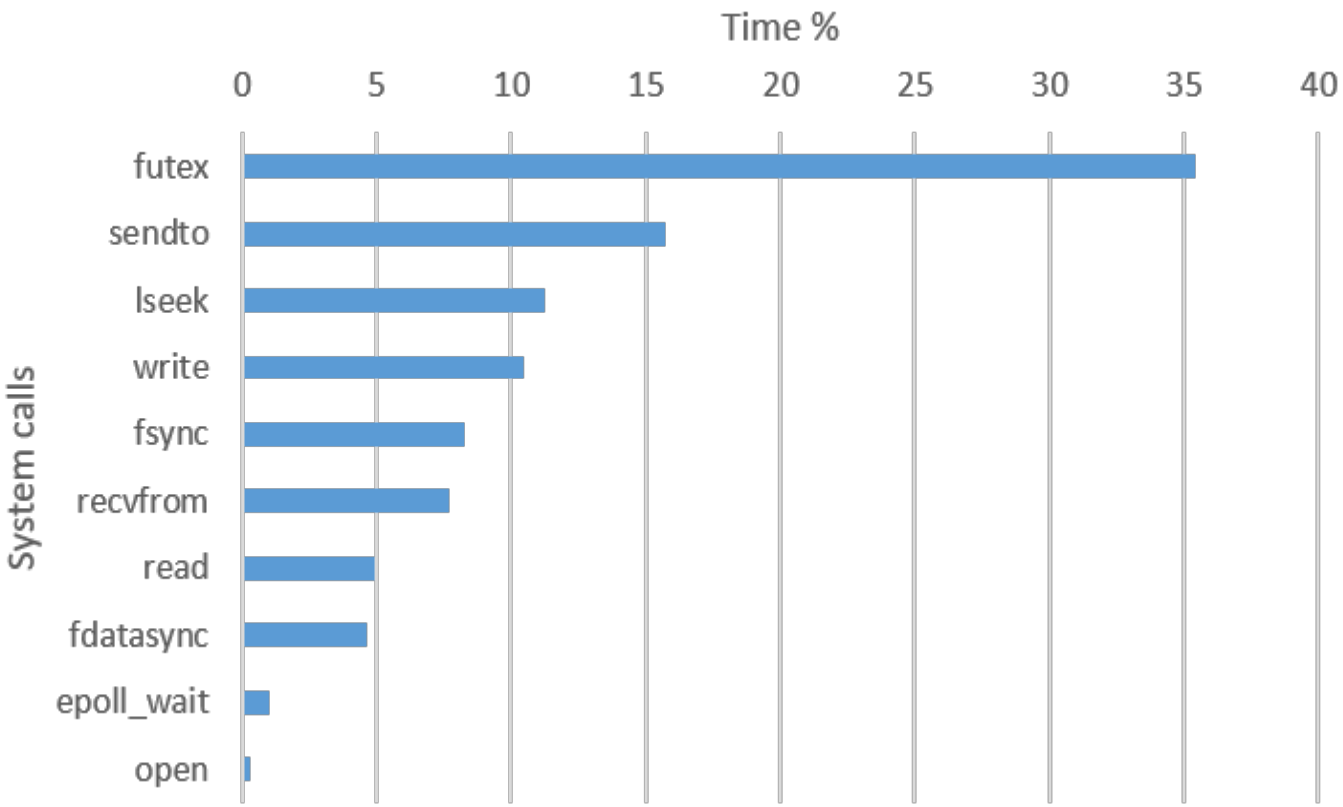


図 4.5 10 の postgres プロセスが使用するシステムコール

図ではそれぞれ実行時間割合の大きい10個のシステムコールを示しています。利用されているシステムコールはほぼ同じですが、実行時間割合が一番高いシステムコールが異なっていました。9.6 は semop, 10 は futex システムコールが最も用いられています。

また、実行されたシステムコールの情報は実行回数とエラー回数も同時に得られます。9.6 の semop, 10 の futex の実行回数とエラー回数を示したのが次の図です。

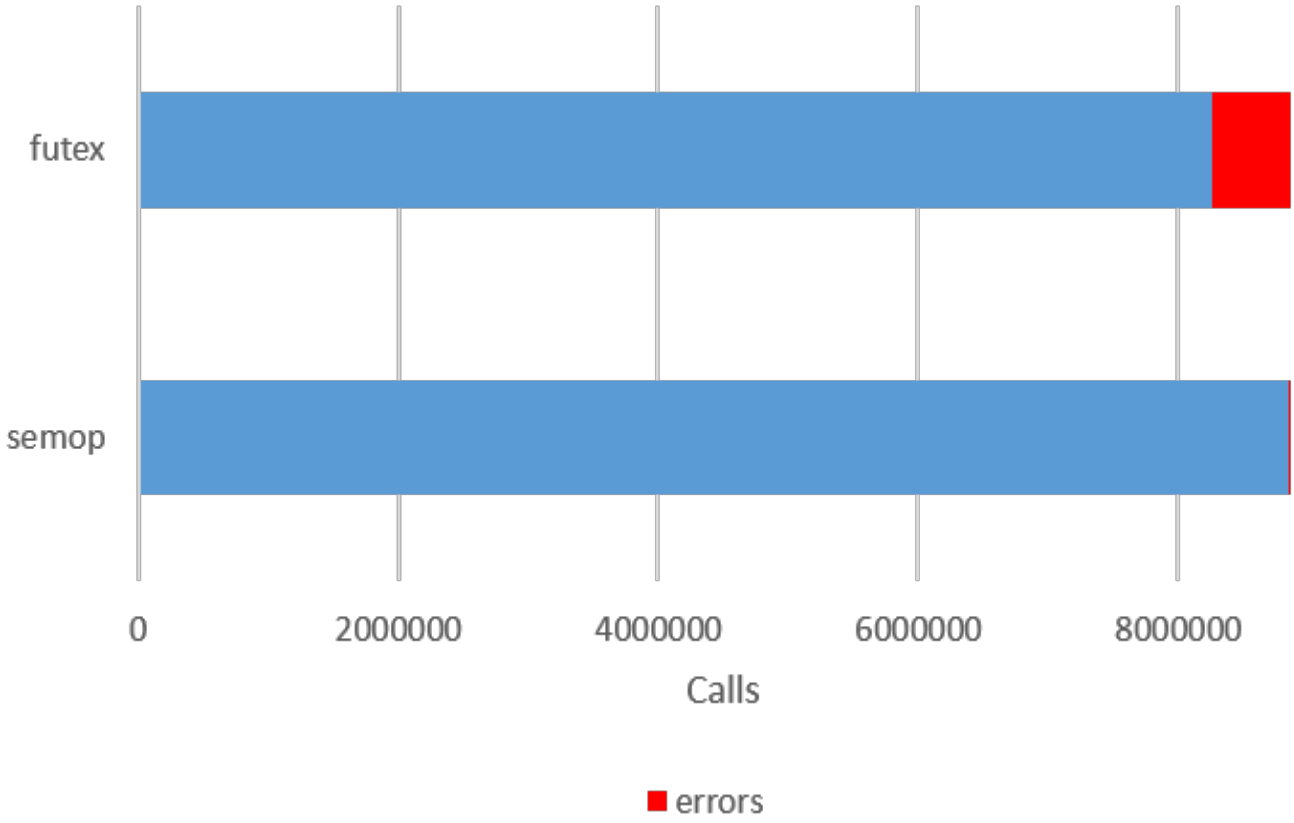


図 4.6 9.6 の semop, 10 の futex の実行回数とエラー回数

futex のシステムコールは semop に対してエラー回数が多く、エラー割合の大きさがパフォーマンス低下につながったのではないかと予想されます。

さらに、これらのシステムコールがどの関数から利用されているかを確認しました。(ここから先は検証期間の関係上、検証者の環境で行っています)

システムコールと呼び出し側のスタックトレース採取は strace の -k オプションを用いました。

```
(9.6)
./strace -ff -o《出力先》 -p《postmaster PID》-k -e semop
(10)
./strace -ff -o《出力先》 -p《postmaster PID》-k -e futex
```

9.6 で semop が用いられていたケースは以下の2通りでした。

```
semop(5472262, [{3, -1, 0}], 1)         = 0
 > /usr/lib64/libc-2.17.so(semop+0x7) [0xf8f37]
 > /usr/local/pgsql/9.6.6/bin/postgres(PGSemaphoreLock+0x41) [0x24f441]
 > /usr/local/pgsql/9.6.6/bin/postgres(LWLockAcquireOrWait+0xfc) [0x2ab60c]
 > /usr/local/pgsql/9.6.6/bin/postgres(XLogFlush+0xcd) [0xdb75d]
 > /usr/local/pgsql/9.6.6/bin/postgres(CommitTransaction+0x4a7) [0xd2f17]
 > /usr/local/pgsql/9.6.6/bin/postgres(CommitTransactionCommand+0xc5) [0xd39f5]
 > /usr/local/pgsql/9.6.6/bin/postgres(finish_xact_command.part.4+0x35) [0x2b5fe5]
 > /usr/local/pgsql/9.6.6/bin/postgres(PostgresMain+0x1fb2) [0x2b98c2]
 > /usr/local/pgsql/9.6.6/bin/postgres(ServerLoop+0x964) [0x6b42a]
 > /usr/local/pgsql/9.6.6/bin/postgres(PostmasterMain+0xe6a) [0x26018a]
 > /usr/local/pgsql/9.6.6/bin/postgres(main+0x3e0) [0x6bf10]
 > /usr/lib64/libc-2.17.so(__libc_start_main+0xf5) [0x21b35]
 > /usr/local/pgsql/9.6.6/bin/postgres(_start+0x29) [0x6bf77]

semop(5505031, [{1, 1, 0}], 1)          = 0
 > /usr/lib64/libc-2.17.so(semop+0x7) [0xf8f37]
 > /usr/local/pgsql/9.6.6/bin/postgres(PGSemaphoreUnlock+0x41) [0x24f4c1]
 > /usr/local/pgsql/9.6.6/bin/postgres(LWLockRelease+0x264) [0x2abc84]
 > /usr/local/pgsql/9.6.6/bin/postgres(XLogFlush+0x138) [0xdb7c8]
 > /usr/local/pgsql/9.6.6/bin/postgres(CommitTransaction+0x4a7) [0xd2f17]
 > /usr/local/pgsql/9.6.6/bin/postgres(CommitTransactionCommand+0xc5) [0xd39f5]
 > /usr/local/pgsql/9.6.6/bin/postgres(finish_xact_command.part.4+0x35) [0x2b5fe5]
 > /usr/local/pgsql/9.6.6/bin/postgres(PostgresMain+0x1fb2) [0x2b98c2]
 > /usr/local/pgsql/9.6.6/bin/postgres(ServerLoop+0x964) [0x6b42a]
 > /usr/local/pgsql/9.6.6/bin/postgres(PostmasterMain+0xe6a) [0x26018a]
 > /usr/local/pgsql/9.6.6/bin/postgres(main+0x3e0) [0x6bf10]
 > /usr/lib64/libc-2.17.so(__libc_start_main+0xf5) [0x21b35]
 > /usr/local/pgsql/9.6.6/bin/postgres(_start+0x29) [0x6bf77]
```

10 で futex が用いられていたケースは以下の2通りでした。

```
futex(0x7f7f17ddfe58, FUTEX_WAKE, 1)    = 1
 > /usr/lib64/libpthread-2.17.so(sem_post+0x3d) [0xdb7d]
 > /usr/local/pgsql/10.1/bin/postgres(PGSemaphoreUnlock+0x22) [0x284ef2]
 > /usr/local/pgsql/10.1/bin/postgres(LWLockRelease+0x324) [0x2ed664]
 > /usr/local/pgsql/10.1/bin/postgres(XLogFlush+0x138) [0xedac8]
 > /usr/local/pgsql/10.1/bin/postgres(CommitTransaction+0x4c1) [0xe4ca1]
 > /usr/local/pgsql/10.1/bin/postgres(CommitTransactionCommand+0xc5) [0xe57b5]
 > /usr/local/pgsql/10.1/bin/postgres(finish_xact_command.part.4+0x15) [0x2f82d5]
 > /usr/local/pgsql/10.1/bin/postgres(exec_simple_query+0x392) [0x2fa0a2]
 > /usr/local/pgsql/10.1/bin/postgres(PostgresMain+0xf5c) [0x2fb28c]
 > /usr/local/pgsql/10.1/bin/postgres(ServerLoop+0xbd7) [0x75c6d]
 > /usr/local/pgsql/10.1/bin/postgres(PostmasterMain+0xe89) [0x296609]
 > /usr/local/pgsql/10.1/bin/postgres(main+0x3f1) [0x76681]
 > /usr/lib64/libc-2.17.so(__libc_start_main+0xf5) [0x21b35]
 > /usr/local/pgsql/10.1/bin/postgres(_start+0x29) [0x766e8]

futex(0x7f7f17ddfc98, FUTEX_WAIT, 0, NULL) = 0
 > /usr/lib64/libpthread-2.17.so(do_futex_wait.constprop.1+0x2b) [0xd79b]
 > /usr/lib64/libpthread-2.17.so(__new_sem_wait_slow.constprop.0+0x4f) [0xd82f]
 > /usr/lib64/libpthread-2.17.so(sem_wait+0x2b) [0xd8cb]
 > /usr/local/pgsql/10.1/bin/postgres(PGSemaphoreLock+0x22) [0x284e92]
 > /usr/local/pgsql/10.1/bin/postgres(LWLockAcquireOrWait+0xbd) [0x2ece6d]
 > /usr/local/pgsql/10.1/bin/postgres(XLogFlush+0xcd) [0xeda5d]
 > /usr/local/pgsql/10.1/bin/postgres(CommitTransaction+0x4c1) [0xe4ca1]
 > /usr/local/pgsql/10.1/bin/postgres(CommitTransactionCommand+0xc5) [0xe57b5]
 > /usr/local/pgsql/10.1/bin/postgres(finish_xact_command.part.4+0x15) [0x2f82d5]
 > /usr/local/pgsql/10.1/bin/postgres(exec_simple_query+0x392) [0x2fa0a2]
 > /usr/local/pgsql/10.1/bin/postgres(PostgresMain+0xf5c) [0x2fb28c]
 > /usr/local/pgsql/10.1/bin/postgres(ServerLoop+0xbd7) [0x75c6d]
 > /usr/local/pgsql/10.1/bin/postgres(PostmasterMain+0xe89) [0x296609]
 > /usr/local/pgsql/10.1/bin/postgres(main+0x3f1) [0x76681]
 > /usr/lib64/libc-2.17.so(__libc_start_main+0xf5) [0x21b35]
 > /usr/local/pgsql/10.1/bin/postgres(_start+0x29) [0x766e8]
```

semop も futex もいずれも PGSemaphoreLock, PGSemaphoreUnlock 関数で利用されていました。つまり、いずれもセマフォを操作する関数から用いられるシステムコールでした。

4.9. 考察

本検証では例年通り、PostgreSQL最新バージョンと旧バージョンとのスケールアップ性能比較検証を行いました。

さて今回、10のリリースノートにはスケール性能向上を謳った項目が見受けられなかったので、検証前は参照系更新系いずれも性能は大きく変化しないものと予想していました。

実際、参照系においては新旧バージョンで性能は大きく変わりませんでしたが、しかし更新系においては多数の同時接続時において新バージョンの明らかな性能低下が確認されました。

更新系について、システムコールの面から追加検証を行いました。追加検証により、各バージョンで実行時間が一番長いシステムコールが異なること、それらのシステムコールはセマフォ操作の関数から呼び出されていること、10のシステムコールのエラー率が高いことを確認できました。

また、各バージョンで実行時間が一番長いシステムコールが異なるのは、10において「Linux 上で利用されるデフォルトのセマフォが System V セマフォから POSIX セマフォに変更」されたことに依るものでした。本変更点はリリースノートからも確認できました。

状況証拠ではありますが、これらの結果より更新性能低下の原因はセマフォが変更されたことが大きいとみられます。今回、検証期間をオーバーしてしまったため叶いませんでしたが、実際にセマフォがパフォーマンス低下の原因かどうかは10において System V セマフォの使用を明示的に指定することで確認可能と思われます。

ところで、本検証と同時期(2018-01-23)に pgsql-hackers メーリングリストへ9.6と10の更新性能比較検証報告(※3)が投稿されています。こちらの結果も本検証と同様に10の性能低下を示す結果でした。その後のコメントでやはりセマフォの変更による性能低下が示唆されておりました。

参考:

1. PostgreSQL 10 リリースノート (日本語)
   https://www.postgresql.jp/document/10/html/release-10.html

2. PostgreSQL 10 リリースノート (英語)
   https://www.postgresql.org/docs/10/static/release-10.html

3. pgsql-hackers メーリングリストに投稿された 9.6 と 10 の更新性能比較検証
   https://www.postgresql.org/message-id/CAD__OugYDM3O%2BdyZnnZSbJprSfsGFJcQ1R%3De59T3hcLmDug4_w%40mail.gmail.com

5. Windows版PostgreSQL性能検証

5.1. 検証概要

本章では、Windows版PostgreSQLの性能を測定し、Linux版PostgreSQLとの性能の傾向を比較した結果を示します。

国内のPostgrreSQLユーザの中にはWindows版PostgreSQLを業務に使っているユーザも少なからずいます。しかし、これまでPGEConsでは、Windows版PostgreSQLを用いた性能検証は実施していませんでした。今年度のPGEConsでは、Linux上で実施していた定点観測と同様の測定をWindows版PostgreSQLでも実施し、Linux版PostgreSQLと、Windows版PostgreSQLの性能の傾向を評価しました。

5.2. 検証構成

本節では、Windows版PostgreSQL検証の環境を説明します。

5.2.1. ハードウェア構成

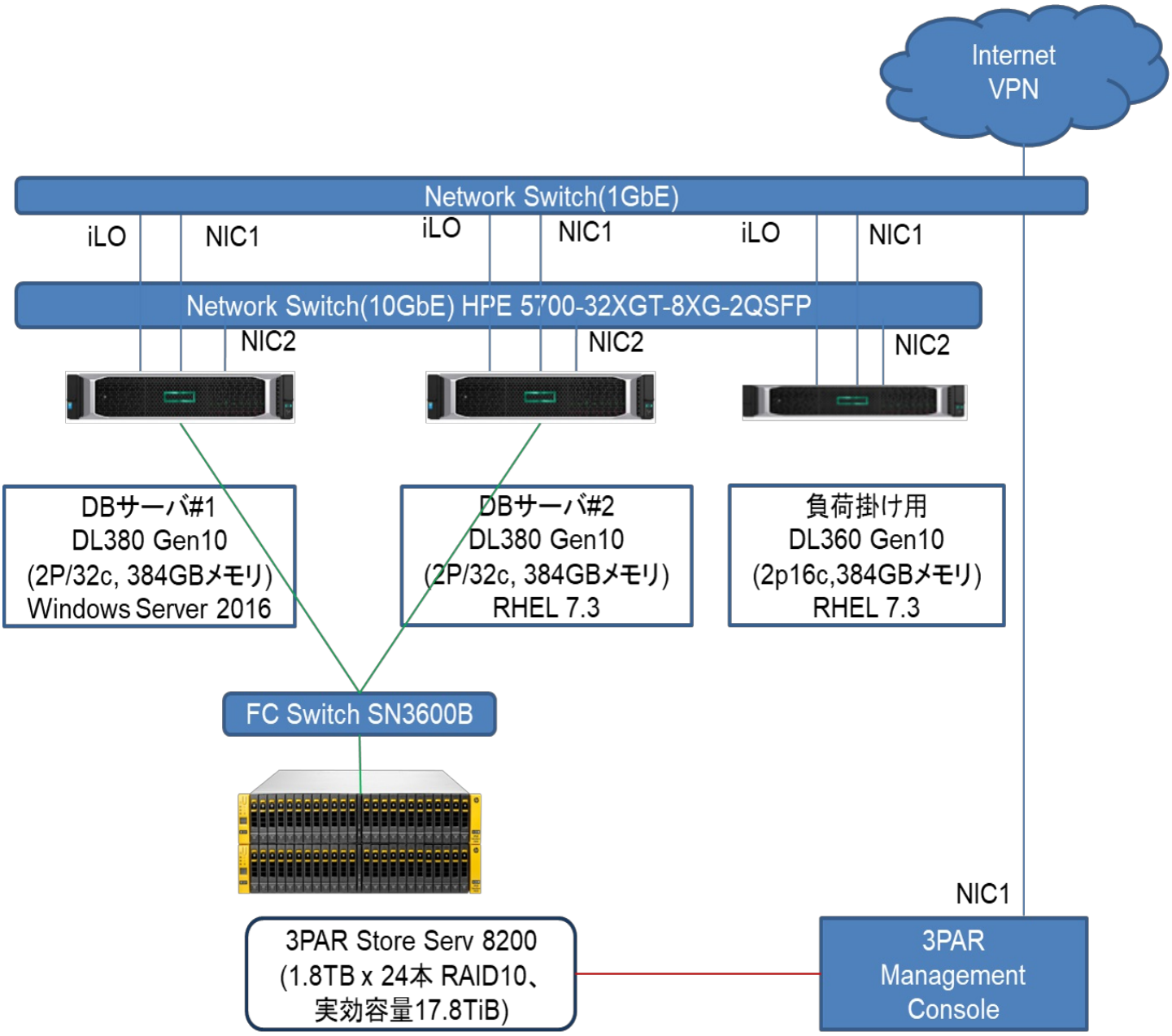


図 5.1 検証ハードウェア構成

5.2.2. ソフトウェア構成

検証環境のソフトウェア構成を示します。

表 5.1 DBサーバ#1

OS	Windows Server2016
PostgreSQL	10.1 (EDB installer版)

表 5.2 負荷掛け用

OS	Red Hat Enterprise Linux 7.3
PostgreSQL	10.1 (ソースビルド版)

5.3. 検証モデル

本節では、Windows版PostgreSQL検証のモデルを説明します。検証モデルの概要については、「定点観測(スケールアップ検証)」を参考にしてください。

5.3.1. データベースクラスタ構成

Windows版PostgreSQLをインストーラでインストールすると、インストール中にデフォルトのデータベースクラスタを作成します。このデータベースクラスタは、デフォルトでは

```
C:\Program Files\PostgreSQL\10\data
```

の下に作成されます。今回の性能検証では、データベースクラスタを外部ストレージには配置せず、またデータベース格納領域とWAL格納領域の分離は行わない構成で実施しています。

5.3.2. PostgreSQLの設定

PostgreSQLの設定として設定ファイルを以下のように変更して測定を実施しました。

```
pgbench=# SELECT * FROM pg_file_settings ;
                    sourcefile                     | sourceline | seqno |            name
|      setting       | applied | error
---------------------------------------------------+------------+-------+-------------------------
----+--------------------+---------+-------
 C:/Program Files/PostgreSQL/10/data/postgresql.conf |         59 |     1 | listen_addresses
| *                  | t       |
 C:/Program Files/PostgreSQL/10/data/postgresql.conf |         63 |     2 | port
| 5432               | t       |
 C:/Program Files/PostgreSQL/10/data/postgresql.conf |        127 |     3 |
dynamic_shared_memory_type | windows            | t       |
 C:/Program Files/PostgreSQL/10/data/postgresql.conf |        344 |     4 | log_destination
| stderr             | t       |
 C:/Program Files/PostgreSQL/10/data/postgresql.conf |        350 |     5 | logging_collector
| on                 | f       |
 C:/Program Files/PostgreSQL/10/data/postgresql.conf |        472 |     6 | log_timezone
| Asia/Tokyo         | t       |
 C:/Program Files/PostgreSQL/10/data/postgresql.conf |        566 |     7 | datestyle
| iso, mdy           | t       |
 C:/Program Files/PostgreSQL/10/data/postgresql.conf |        568 |     8 | timezone
| Asia/Tokyo         | t       |
 C:/Program Files/PostgreSQL/10/data/postgresql.conf |        581 |     9 | lc_messages
| C                  | t       |
 C:/Program Files/PostgreSQL/10/data/postgresql.conf |        583 |    10 | lc_monetary
| C                  | t       |
 C:/Program Files/PostgreSQL/10/data/postgresql.conf |        584 |    11 | lc_numeric
| C                  | t       |
 C:/Program Files/PostgreSQL/10/data/postgresql.conf |        585 |    12 | lc_time
| C                  | t       |
 C:/Program Files/PostgreSQL/10/data/postgresql.conf |        588 |    13 |
default_text_search_config | pg_catalog.english | t       |
 C:/Program Files/PostgreSQL/10/data/postgresql.conf |        662 |    14 | max_connections
| 510                | t       |
 C:/Program Files/PostgreSQL/10/data/postgresql.conf |        663 |    15 | shared_buffers
| 38GB               | t       |
 C:/Program Files/PostgreSQL/10/data/postgresql.conf |        664 |    16 | work_mem
| 1GB                | t       |
 C:/Program Files/PostgreSQL/10/data/postgresql.conf |        665 |    17 | checkpoint_timeout
| 30min              | t       |
 C:/Program Files/PostgreSQL/10/data/postgresql.conf |        666 |    18 | max_wal_size
| 160GB              | t       |
 C:/Program Files/PostgreSQL/10/data/postgresql.conf |        667 |    19 | logging_collector
| on                 | t       |
 C:/Program Files/PostgreSQL/10/data/postgresql.conf |        668 |    20 | log_line_prefix
| %t [%p-&l]         | t       |
(20 rows)
```

5.3.3. 検証用ベンチマーク

検証用のベンチマークとして、PostgreSQL標準のベンチマークツールpgbenchを使用しました。（pgbenchの詳細については、「定点観測(スケールアップ検証)」を参考にしてください。

5.3.3.1. 初期データ登録

初期データの登録は、pgbenchの-iオプションでスケールファクタ=1000のサイズを指定しました。fillfactorの設定はデフォルト(テーブルのfillfactor=100)、unlogged tableの指定はなしとしました。

5.3.3.2. 参照系測定

走行時間は300秒を指定しました。同時実行数として1, 2, 4, 8, 16, 32, 48, 72, 80, 96を与えて実行しました。参照系測定のために、以下のカスタムクエリをpgbenchに与えて実行しました。

```
\set naccounts 100000 * :scale
\set row_count 10000
\set aid_max :naccounts - :row_count
\set aid random(1, :aid_max)

SELECT count(abalance) FROM pgbench_accounts WHERE aid BETWEEN :aid and :aid + :row_count;
```

それぞれの同時実行数において、3回の測定を実施し、その測定結果(tps)の中間値を測定値としています。

5.3.3.3. 更新系測定

走行時間は300秒を指定しました。同時実行数として1, 2, 4, 8, 16, 32, 48, 72, 80, 96を与えて実行しました。更新系測定のために、以下のカスタムクエリをpgbenchに与えて実行しました。

```
\set naccounts 100000 * :scale
\set aid_val random(1, :naccounts)
UPDATE pgbench_accounts SET filler=repeat(md5(current_timestamp::text),2) WHERE aid = :aid_val;
```

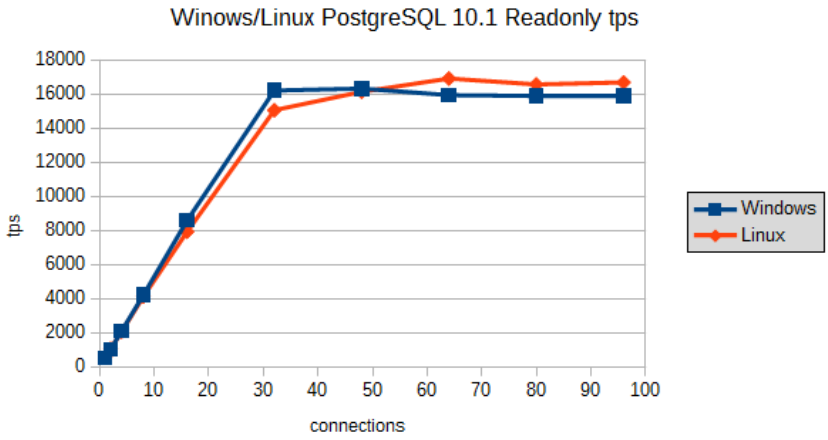
それぞれの同時実行数において、3回の測定を実施し、その測定結果(tps)の中間値を測定値としています。

5.4. 検証結果

本節では、Windows版PostgreSQL検証結果を示します。

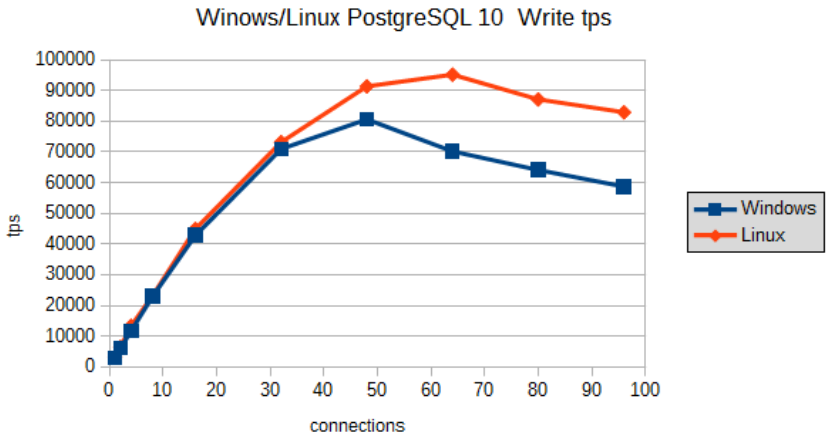
5.4.1. 参照系

参照系ベンチマークの結果を以下に示します。



参照系に関しては、Linux版PostgreSQLと同等の傾向かつ性能になっていると判断できます。またCPU使用率とtps/latencyの関係を見ると、CPU使用率が100%近くになるまでは、同時接続数に対してほぼスケールしており、CPU使用率が100%近くになると、tpsは横ばいとなり、latencyが増加していくという傾向があります。このモデルでは、CPUネックになるという結果になりました。tpsの傾向がLinux版とほぼ同等の傾向であることから、CPU使用率の傾向についてもLinux版と大きな差はないと考えられます。

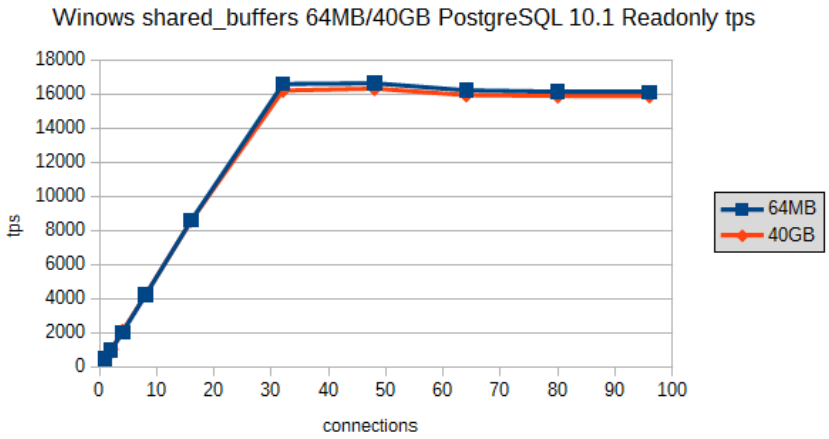
5.4.2. 更新系



更新系に関しては、tpsがピーク付近になると、Linux版と比較してtpsが低下するという傾向になりました。接続数がピークを超えると、Windows版もLinux版もtpsが低下するという傾向にありますが、tpsの劣化度合いが、Linux版よりもWindows版のほうが大きくなっています。CPU使用率についてはtpsのピークおよび、それ以降になってもそれほど大きくなっておらず、I/Oネックになっていることが推測されます。

5.4.3. Windows版 shared_buffersチューニング

shared_buffersを定点観測用の設定値(実メモリの約10%＝約40GB)と、非常に小さな設定値(64MB)で参照系モデルを測定した結果を示します。



Windows版PostgreSQLの場合、shared_buffersが小さくてもほとんど変わらない結果になっています。接続数が大きくなった場合には小さな設定値のほうが僅かに高い結果となっています。Windows版PostgreSQLの場合、PostgreSQL文書のshared_buffersの説明に、「Windowsシステムでのshared_buffersの範囲は一般的に64MBから512MBです。」という記述がありますが、その記述どおりの結果が出たと考えられます。

5.4.4. Windows版 wal_sync_methodチューニング

wal_sync_methodチューニングは、以下の2ステップで実施しました。

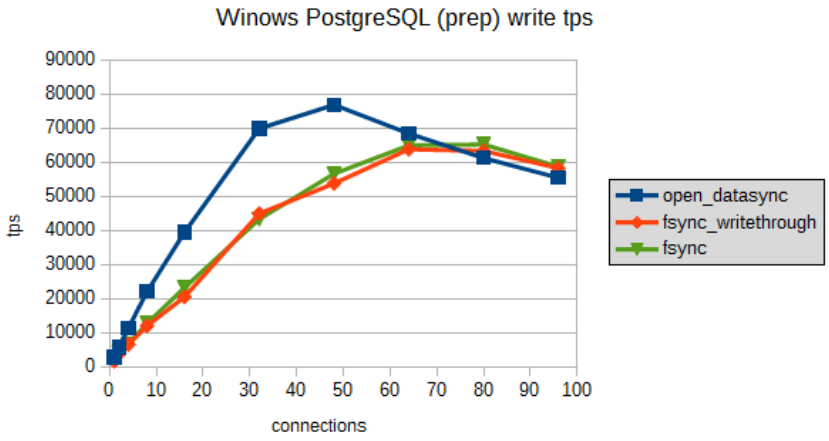
- pg_test_fsyncによる測定
- pgbenchによる測定

pg_test_fsyncとは、PostgreSQLを動作させる場合に、最も高速なwal_sync_methodを決定するためのツールです。pg_test_fsyncによる測定を実施した結果を以下に示します。

表 5.3 pg_fsyncによる測定結果

テストパターン	open_datasync	fdatasync	fsync_writethrough	fsync	open_sync
one 8kB write (ops/sec)	105161.390	n/a	5288.632	5406.255	n/a
one 8kB write (ops/sec)	105161.390	n/a	5288.632	5406.255	n/a

次に更新系ベンチマークで実施した測定を、wal_sync_methodをopen_datasync,fsync_writethrough,fsyncに設定して実施した結果を以下に示します。



wal_sync_methodについては、pg_test_fsyncユーティリティを使った事前検証と、実際にpgbenchを用いた測定の両方において、open_datasyncを設定することで、pgbenchでも最も高いtpsを出せるという結果となりました。

5.5. まとめ

定点観測モデルを用いて、Windows版PostgreSQLとLinux版PostgreSQLの性能を比較しました。同一スペックのサーバ上でLinux/WindowsのOSの差異による測定を実施した結果、参照のみの場合は、Linux版と同等の性能、更新系の場合には、tpsがピークに到達する前には同等の性能でしたが、ピークを超えた後は、Linuxよりもtpsが低下する傾向があるのを確認しました。この結果から、Windows版PostgreSQLでは、大量の書き込みが発生する場合には、Linux版PostgreSQLと比較すると性能が低い可能性があると推測されます。しかし、過負荷状態でなければ、Linux版と同等の性能にはなっており、Windows版PostgreSQLは性能面ではLinux版と遜色はないと考えられます。

特定のPostgreSQLパラメータについて、Windows版固有のチューニングが必要か検証しました。shared_buffersについては、Linux版のチューニング方針とは異なり、参照主体のモデルでは、非常に小さな値を設定しても性能上の問題はないという結果となりました。wal_sync_methodについては、open_datasyncを設定することが妥当であるという結果となりました。

5.6. 参考

1. PostgreSQL文書 pg_test_fsync
   https://www.postgresql.jp/document/10/html/pgtestfsync.html

2. PostgreSQL文書 資源の消費
   https://www.postgresql.jp/document/10/html/runtime-config-resource.html#runtime-config-resource-memory

6. パラレルクエリ 性能検証

6.1. 検証概要

近年、ビッグデータ活用の流れから、データベースに大量のデータを投入して検索や集計を行い、新たな知見を得ようとする試みが増えています。例えば、店舗における売上データなど、経営に関わるデータを組み合わせて多面的に解析するBI(Business Intelligence)が典型的です。このような使用方法においては、長大かつ高負荷となるクエリを処理することとなり、多数の短小なクエリを大量に処理するような使用方法であるOLTPに対して、OLAPと呼ばれています。一般的に、OLAPではパラレルクエリやパーティショニングの恩恵を受けやすいとされています。

PostgreSQLをOLAP用途で使用したいという要望は、以前から根強くありましたが、PostgreSQL 9.5まではパラレルクエリが実装されていなかったため、OLAP用途で実用的に使用できない状態でした。

そこで、PostgreSQL 9.6で実装され、PostgreSQL 10で強化されたパラレルクエリによって、PostgreSQLがOLAP用途で実用的に使用できるようになったのかを検証します。本検証では、OLAPにおいて使用される形式のクエリを模したベンチマークテストであるTPC-DSで定義されているクエリを題材として使用します。

6.1.1. TPC-DSとは

TPC-DSは、TPC-Hの後継となるベンチマークテストです。BIでよく用いられるスタースキーマをもとに、TPC-Hよりさらにテーブル間の関連を複雑にし、1回のクエリ発行で多数のテーブルジョインや集約演算が行われるように クエリが工夫されています。

大規模なデータを取り扱うので、I/O周りはもちろんのこと、多数のジョイン操作や集約演算が行われることから、通常のOLTPよりもCPUの処理性能と、CPUをどの程度うまく使えているかがベンチマーク結果を大きく左右します。

TPC-DSでは、99種類/103本のクエリが定義されていますが、本検証では、そのうちのNo.10とN0.11の2種類/2本のクエリについて検証を行いました。

これら2種類/2本を選定したのは、本検証に先立って行った事前検証の結果、No.10に関してはパラレルクエリ実行を行うことで劇的な性能改善が見られたこと、No.11に関してはPostgreSQL 10でのみパラレルクエリ実行が行われており [1] 、PostgreSQL 10で新規にパラレルクエリ対応したパターンにマッチしたものと判断したことが理由です。

[1] No.11以外にはNo.4が該当しますが、事前検証の結果から、No.4は検証対象としたバージョン9.5～10のすべてで実行時間が長大(6時間強)となることが見込まれたため、検証期間の関係から採用を見送りました。

6.2. パラレルクエリとは

6.2.1. パラレルクエリ

パラレルクエリとは、一つのクエリを複数のプロセスで分担して並列処理することで、単一プロセスでの処理よりも高速にクエリを実行する処理方式です。

例えばSELECT文で1000万レコードを全件検索する場合、検索で使用するプロセスが1個だけだと1プロセスが1000万レコードをすべて検索する必要があります。それに対して、例えば、検索で使用するプロセスが10プロセスあって、それらで分担することができれば、1プロセス当たりの検索対象は100万レコードと10分の1に減ります。複数のプロセスで分担することで高速化を図るため、メニーコア環境での使用で、効果を発揮します。

複数のプロセスで検索処理を分担して実行することで、1プロセスあたりの検索自体にかかる実行時間は短くなりますが、ある処理を複数に分割する配分の制御や受け渡し、そして複数で処理した結果を集めて一つにまとめる処理などの実行時間が長くなります。複数のプロセスで短縮できる時間があまり少ないと、パラレルクエリに変更することでむしろ余計に実行時間がかかってしまうこともあります。

そのため、プランナは多くの実行プランの中から、パラレルクエリのほうが効率的に処理できると判断した場合のみパラレルクエリの実行プランを選択します。

6.2.2. PostgreSQLでのパラレルクエリ

2017年現在、PostgreSQLで実装されているパラレルクエリに関する処理は、パラレルスキャン、パラレルジョイン、パラレル集約の三種類です。

プランナが処理を分担したほうが効率的だと判断した場合、パラレルクエリのプランを採用します。

explain (analyze , verbose)で確認したパラレルクエリ プラン例

```
                                                         QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------------------
Gather  (cost=1000.00..112816.10 rows=38400 width=52) (actual time=0.198..224.017 rows=99740 loops=1)
  Output: log_id, date, "time", product_id, place_id, machine_id, app_id, access_time_second, access_count, err_code
  Workers Planned: 4
  Workers Launched: 4
  ->  Parallel Seq Scan on public.access_log  (cost=0.00..107976.10 rows=9600 width=52) (actual time=0.017..209.797 rows=19948 loops=5)
        Output: log_id, date, "time", product_id, place_id, machine_id, app_id, access_time_second, access_count, err_code
        Filter: ((access_log.log_id % '77'::bigint) = 0)
        Rows Removed by Filter: 1516052
        Worker 0: actual time=0.016..214.828 rows=20088 loops=1
        Worker 1: actual time=0.016..214.518 rows=19967 loops=1
        Worker 2: actual time=0.020..218.061 rows=20817 loops=1
        Worker 3: actual time=0.015..213.654 rows=20054 loops=1
```

パラレルクエリでは、親プロセスに加えワーカーがバックグラウンドワーカープロセスとして起動されて処理を実行するため、検索プロセス数は親プロセスとワーカーの合計となります。そして、各プロセスで処理した結果をGatherノードで集約します。

また、パラレルクエリで起動されるワーカー数は、テーブルサイズによりデフォルトで起動する数が決まっています。

表 6.1 テーブルサイズによるワーカー数

ブロック数 [2]	サイズ(MB)	起動ワーカー数	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
746,496	5,832	7	8
2,239,488	17,496	8	9
6,718,464	52,488	9	10
20,155,392	157,464	10	11
...	...	...	...

[2] ブロック数はブロックサイズがデフォルト(8KB)の場合です。

6.2.3. パラレルクエリに関連するパラメータ

パラレルクエリ時に関連する主なパラメータには以下のパラメータがあります。なお、max_parallel_workers_per_gatherパラメータのデフォルト値がバージョン9.6では0のため、デフォルト設定ではパラレルクエリは実行されません。バージョン10では同パラメータのデフォルト値は2であるため、デフォルト設定でもパラレルクエリが実行されます。

表 6.2 パラレルクエリに関連する主なパラメータ

パラメータ	デフォルト値(9.6)	デフォルト値(10)	説明
max_worker_processes [3]	8	8	システムがサポートするバックグラウンドワーカープロセスの最大数を指定します。
max_parallel_workers_per_gather [3]	0	2	一つのGatherノードに対して起動できるワーカー数の最大値を指定します。
max_parallel_workers [3] [4]	—	8	一度に有効となるパラレルワーカープロセスの最大数を指定します。
min_parallel_relation_size	8MB	—	パラレル処理を行うテーブルの最小サイズを指定します。
min_parallel_table_scan_size	—	8MB	パラレル処理を行うテーブルの最小サイズを指定します。
min_parallel_index_scan_size	—	512KB	パラレル処理を行うインデックスの最小サイズを指定します。
parallel_setup_cost	1000	1000	ワーカーを起動するためのコストに対するプランナの推測値を指定します。
parallel_tuple_cost	0.1	0.1	あるワーカーから、1行を他のプロセスに転送するためのコストに対するプランナの推測値を指定します。

[3] (1, 2, 3) max_parallel_workers_per_gatherで指定した値がmax_worker_processesで指定した値を超える場合、起動されるワーカー数はmax_worker_processesの値に制限されます。また、複数のパラレルクエリが同時に行われる場合、起動されるワーカー数の合計がmax_worker_processesで指定した値に制限されます。このため、プラン作成時のワーカー数よりも実際に起動されるワーカー数が少ない場合があります。

[4] PostgreSQL 10では、max_parallel_workersもあり、パラレルクエリの実行に使用されるワーカー数は、基本的にはこちらに指定した値に制限されます。ただし、この値がmax_worker_processesで指定した値を超える場合は、起動されるワーカー数はmax_worker_processesの値に制限されます。

6.3. 検証構成

6.3.1. 検証ハードウェア構成

検証環境のハードウェア構成の概略を示します。

本パラレルクエリ検証ではDBサーバ上でSQLを直接実行する形式としました。

またコア数やメモリサイズの違いによるパラレルクエリの効果の違いを比較するために、検証ではコア数とメモリサイズの構成が異なる下記のDBサーバを使用しました。

表 6.3 検証用DBサーバ構成

機器	項目	仕様
検証サーバ1 (DL380 Gen10)	CPU	インテル Xeon Gold 6142@2.60GHz 16コアx2 合計 32コア ハイパースレッディング:off
	搭載メモリ	384GB
	内蔵ストレージ	HDD 1.2TB SAS 10K x 16
検証サーバ2 (Express5800/R120f-1E)	CPU	インテル Xeon E5-2630 v4@2.20GHz 10コアx2 合計 20コア ハイパースレッディング:off
	搭載メモリ	256GB
	内蔵ストレージ	SSD 400GB x 8 (HW-RAIDカード, RAID 5構成)

6.3.2. 検証ソフトウェア構成

検証環境のソフトウェア構成を示します。

表 6.4 ソフトウェア構成

種類	ソフトウェア名およびバージョン
OS	Red Hat Enterprise Linux 7.2
Database	PostgreSQL 9.5.10 PostgreSQL 9.6.6 PostgreSQL 10.1

6.4. 検証方法

6.4.1. 検証項目

パラレルクエリ検証では以下の2点を検証します。

6.4.1.1. 検証A

検証Aでは、postgresql.confで、デフォルトではコメントアウトされているパラレルクエリ関連のパラメータの コメントアウトを単に外した(有効化した)だけの状態で、パラレルクエリでの処理性能の改善状況を 確認します。

本検証によって、コメントアウトを外すだけという作業のみで、OLAP用途におけるクエリに対する性能が、どの程度改善するのか、もしくは悪化するのかを明らかにします。(特に、PostgreSQL 10においては、デフォルトの設定そのものの性能を検証することになります。)

また、パラレルクエリを有効化した場合に作成される実行計画が、有効化前からどのように変化し、それが適切なものであるかどうかを検証します。

6.4.1.2. 検証B

検証Bでは、パラレルクエリの並列度による処理性能の改善状況を確認します。

検証Aにおけるパラメータに加えて、max_worker_processes、max_parallel_workers_per_gather、max_parallel_workers(PostgreSQL 10のみ) の各パラメータの値を変更し、並列度の違いによる処理時間を比較します。

本検証によって、パラレルクエリ関連パラメータ、特に並列度のチューニングによって、OLAP用途におけるクエリに対する性能が、検証Aとの比較でさらに改善する余地があるのかどうかを明らかにします。

6.4.1.3. 対照群

対照群として、バージョン9.6、バージョン10において、max_parallel_workers_per_gather=0に明示的に設定を行った状態で 処理時間を比較します。

これにより、パラレルクエリ部分によらない性能改善がどの程度あったのかを明らかにします。

6.4.2. データモデル

本検証で使用するデータモデルは、TPC-DSにおいて定義されているデータモデルに準拠します。

なお、Scale Factorは2GBを採用しました。

各テーブルのサイズは、以下のとおりです。

```
       relname         |   rows   |   bytes
-----------------------+----------+------------
 call_center           |        8 |       8192
 catalog_page          |    11718 |    2318336
 catalog_returns       |   288491 |   68222976
 catalog_sales         |  2880108 |  838221824
 customer              |   144000 |   39616512
 customer_address      |    72000 |   13123584
 customer_demographics |  1920800 |  224788480
 date_dim              |    73049 |   16179200
 dsdgen_version        |        1 |       8192
 household_demographics |     7200 |     557056
 income_band           |       20 |       8192
 inventory             | 16966362 | 1017929728
 item                  |    26000 |   15671296
 promotion             |      322 |      81920
 reason                |       36 |       8192
 ship_mode             |       20 |       8192
 store                 |       22 |       8192
 store_returns         |   575285 |  102301696
 store_sales           |  5760759 | 1109213184
 time_dim              |    86400 |   13533184
 warehouse             |        5 |       8192
 web_page              |       74 |      16384
 web_returns           |   143629 |   29876224
 web_sales             |  1439236 |  420954112
 web_site              |       30 |      16384
```

```
          relname           |   rows   |   bytes
----------------------------+----------+-----------
 call_center_pkey           |        8 |     16384
 catalog_page_pkey          |    11718 |    286720
 catalog_returns_pkey       |   288491 |   9125888
 catalog_sales_pkey         |  2880108 |  90865664
 customer_address_pkey      |    72000 |   1630208
 customer_demographics_pkey |  1920800 |  43171840
 customer_pkey              |   144000 |   3260416
 date_dim_pkey              |    73049 |   1654784
 household_demographics_pkey |     7200 |    180224
 income_band_pkey           |       20 |     16384
 inventory_pkey             | 16966362 | 689053696
 item_pkey                  |    26000 |    606208
 promotion_pkey             |      322 |     16384
 reason_pkey                |       36 |     16384
 ship_mode_pkey             |       20 |     16384
 store_pkey                 |       22 |     16384
 store_returns_pkey         |   575285 |  18169856
 store_sales_pkey           |  5760759 | 181723136
 time_dim_pkey              |    86400 |   1957888
 warehouse_pkey             |        5 |     16384
 web_page_pkey              |       74 |     16384
 web_returns_pkey           |   143629 |   4554752
 web_sales_pkey             |  1439236 |  45424640
 web_site_pkey              |       30 |     16384
```

6.4.3. 使用クエリについて

本検証では、TPC-DSで定義されるNo.10とNo.11のクエリを使用します。それぞれのクエリは、データモデルに対し、以下のデータの抽出を指示します。

6.4.3.1. No.10

(参考訳) ある一定の地方に住み、実店舗と実店舗以外 [5] での流通経路の両方で購入したことのある顧客のうち、性別・配偶者有無・最終学歴・購入額予測・信用状況・被扶養者数・就業している被扶養者数・大学生の被扶養者数が 同一のものの人数を数えよ。

[5] 実店舗以外には、Web(ECサイト)通販とカタログ通販の2種類がある想定。

6.4.3.2. No.11

(参考訳) 昨年と比較して、今年の実店舗における使用金額よりWeb(ECサイト)通販での使用金額が大きくなった 顧客を抽出せよ。

6.4.4. DB設定

性能測定にあたり、デフォルトから変更したパラメータは以下のとおりです。

表 6.5 デフォルトから変更したパラメータ

パラメータ	検証サーバ1
listen_addresses	*
shared_buffers	40GB
work_mem	1GB
maintenance_work_mem	20GB
autovacuum	off
max_worker_processes [6]	8
max_parallel_workers_per_gather [6]	2
max_parallel_workers [6] [7]	8
log_line_prefix	'%t [%p-%l]: '

[6] (1, 2, 3) 測定により値を変更して実施します。

[7] PostgreSQL 10のみ

6.4.5. 測定手順

測定に際しては、SQLの実行前にpg_prewarmを利用して、テーブル・インデックスのデータをディスクから一度読み込み、PostgreSQLのバッファ上、もしくはOSのディスクバッファ上に常駐させた状態にします。

```
SELECT relname, pg_prewarm(oid)
FROM pg_class
WHERE relnamespace in (
  SELECT oid
  FROM pg_namespace
  WHERE nspname='public'
) AND (
  relkind='r' OR relkind='i'
);
```

6.4.5.1. 検証A測定方法

No.10とNo.11のSQLを、psqlを使用して実行します。

- No.10

```
select
  cd_gender,
  cd_marital_status,
  cd_education_status,
  count(*) cnt1,
  cd_purchase_estimate,
  count(*) cnt2,
  cd_credit_rating,
  count(*) cnt3,
  cd_dep_count,
  count(*) cnt4,
  cd_dep_employed_count,
  count(*) cnt5,
  cd_dep_college_count,
  count(*) cnt6
 from
  customer c,customer_address ca,customer_demographics
 where
  c.c_current_addr_sk = ca.ca_address_sk and
  ca_county in ('Walker County','Richland County','Gaines County','Douglas County','Dona Ana
County') and
  cd_demo_sk = c.c_current_cdemo_sk and
  exists (select *
          from store_sales,date_dim
          where c.c_customer_sk = ss_customer_sk and
                ss_sold_date_sk = d_date_sk and
                d_year = 2002 and
                d_moy between 4 and 4+3) and
   (exists (select *
            from web_sales,date_dim
            where c.c_customer_sk = ws_bill_customer_sk and
                  ws_sold_date_sk = d_date_sk and
                  d_year = 2002 and
                  d_moy between 4 ANd 4+3) or
    exists (select *
            from catalog_sales,date_dim
            where c.c_customer_sk = cs_ship_customer_sk and
                  cs_sold_date_sk = d_date_sk and
                  d_year = 2002 and
                  d_moy between 4 and 4+3))
 group by cd_gender,
          cd_marital_status,
          cd_education_status,
          cd_purchase_estimate,
          cd_credit_rating,
          cd_dep_count,
          cd_dep_employed_count,
          cd_dep_college_count
 order by cd_gender,
          cd_marital_status,
          cd_education_status,
          cd_purchase_estimate,
          cd_credit_rating,
          cd_dep_count,
          cd_dep_employed_count,
          cd_dep_college_count
limit 100;
```

- No.11

```
with year_total as (
 select c_customer_id customer_id
       ,c_first_name customer_first_name
       ,c_last_name customer_last_name
       ,c_preferred_cust_flag customer_preferred_cust_flag
```

```
      ,c_birth_country customer_birth_country
      ,c_login customer_login
      ,c_email_address customer_email_address
      ,d_year dyear
      ,sum(ss_ext_list_price-ss_ext_discount_amt) year_total
      ,'s' sale_type
from customer
    ,store_sales
    ,date_dim
where c_customer_sk = ss_customer_sk
  and ss_sold_date_sk = d_date_sk
group by c_customer_id
        ,c_first_name
        ,c_last_name
        ,c_preferred_cust_flag
        ,c_birth_country
        ,c_login
        ,c_email_address
        ,d_year
union all
select c_customer_id customer_id
      ,c_first_name customer_first_name
      ,c_last_name customer_last_name
      ,c_preferred_cust_flag customer_preferred_cust_flag
      ,c_birth_country customer_birth_country
      ,c_login customer_login
      ,c_email_address customer_email_address
      ,d_year dyear
      ,sum(ws_ext_list_price-ws_ext_discount_amt) year_total
      ,'w' sale_type
from customer
    ,web_sales
    ,date_dim
where c_customer_sk = ws_bill_customer_sk
  and ws_sold_date_sk = d_date_sk
group by c_customer_id
        ,c_first_name
        ,c_last_name
        ,c_preferred_cust_flag
        ,c_birth_country
        ,c_login
        ,c_email_address
        ,d_year
        )
 select
                t_s_secyear.customer_id
               ,t_s_secyear.customer_first_name
               ,t_s_secyear.customer_last_name
               ,t_s_secyear.customer_email_address
from year_total t_s_firstyear
    ,year_total t_s_secyear
    ,year_total t_w_firstyear
    ,year_total t_w_secyear
where t_s_secyear.customer_id = t_s_firstyear.customer_id
        and t_s_firstyear.customer_id = t_w_secyear.customer_id
        and t_s_firstyear.customer_id = t_w_firstyear.customer_id
        and t_s_firstyear.sale_type = 's'
        and t_w_firstyear.sale_type = 'w'
        and t_s_secyear.sale_type = 's'
        and t_w_secyear.sale_type = 'w'
        and t_s_firstyear.dyear = 2001
        and t_s_secyear.dyear = 2001+1
        and t_w_firstyear.dyear = 2001
        and t_w_secyear.dyear = 2001+1
        and t_s_firstyear.year_total > 0
        and t_w_firstyear.year_total > 0
        and case when t_w_firstyear.year_total > 0 then t_w_secyear.year_total /
```

```
t_w_firstyear.year_total else 0.0 end
            > case when t_s_firstyear.year_total > 0 then t_s_secyear.year_total /
t_s_firstyear.year_total else 0.0 end
 order by t_s_secyear.customer_id
         ,t_s_secyear.customer_first_name
         ,t_s_secyear.customer_last_name
         ,t_s_secyear.customer_email_address
limit 100;
```

6.4.5.2. 検証B測定方法

No.10とNo.11のSQLを、psqlを使用して実行します。

測定に際しては、postgresql.confに以下の設定を行い、再起動を行うことで設定の反映を行っています。また、測定を行う前にはpg_prewarmを用いて、メモリ上にデータを常駐させた状態にしています。

表 6.6 検証時設定

パラメータ	workers = 2	workers = 4	workers = 6	workers = 8
max_parallel_workers_per_gether	2	4	6	8
max_worker_processes	8	16	24	32
max_parallel_workers	8	16	24	32

6.5. 検証結果

6.5.1. 検証A(パラレルクエリを有効にしたことによる効果)

表 6.7 検証条件

DBサーバ	ストレージ	SQL文	保持データ量
検証サーバ1	内蔵ストレージ	TPC-DSで定義されるNo.10とNo.11の各クエリ	Scale Factor = 2GB
検証サーバ2	内蔵ストレージ	TPC-DSで定義されるNo.10とNo.11の各クエリ	Scale Factor = 2GB

バージョン毎の処理時間のグラフと表を以下に示します。

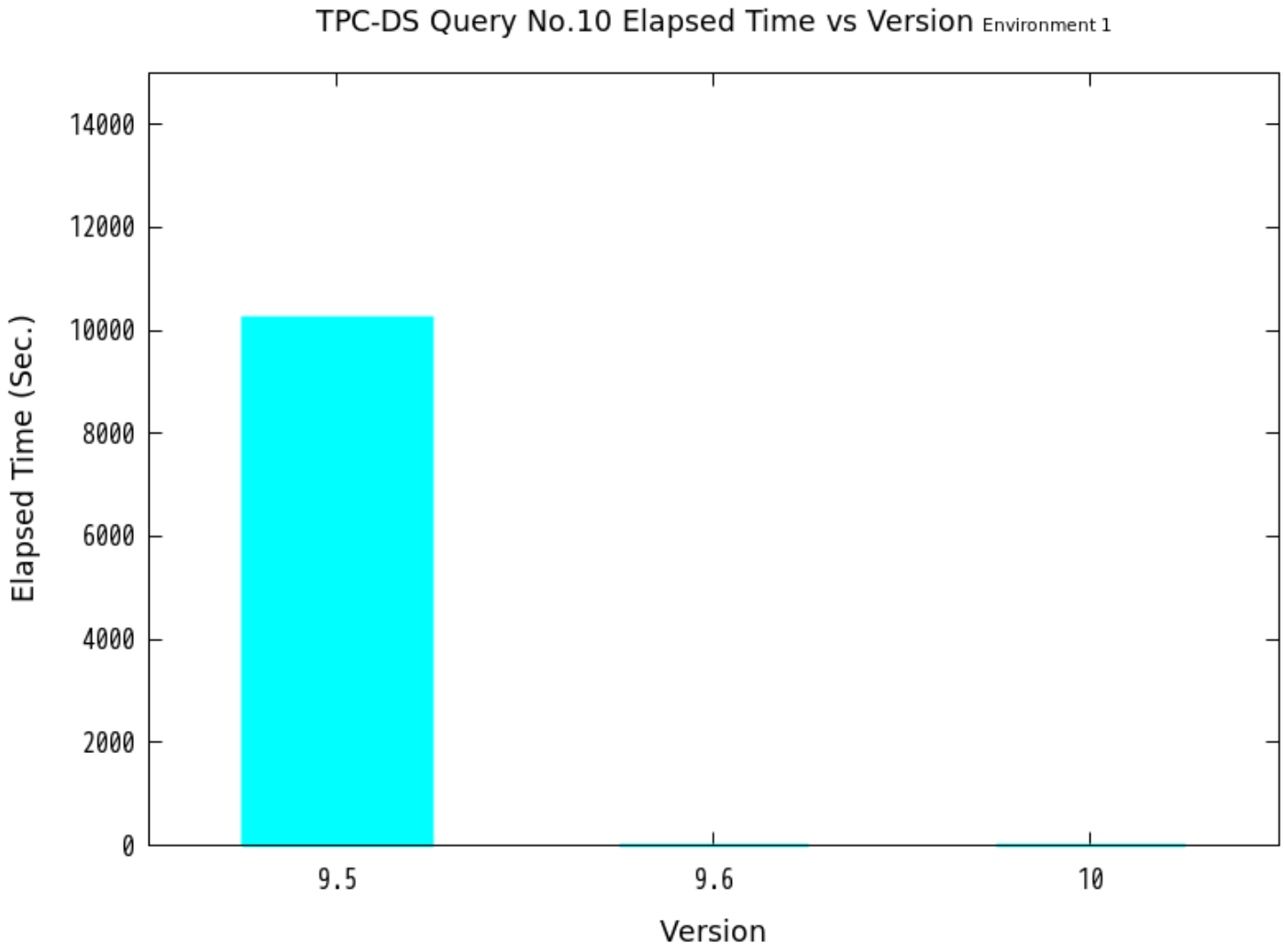


図 6.1 バージョンと処理時間の比較 (No.10/検証サーバ1)

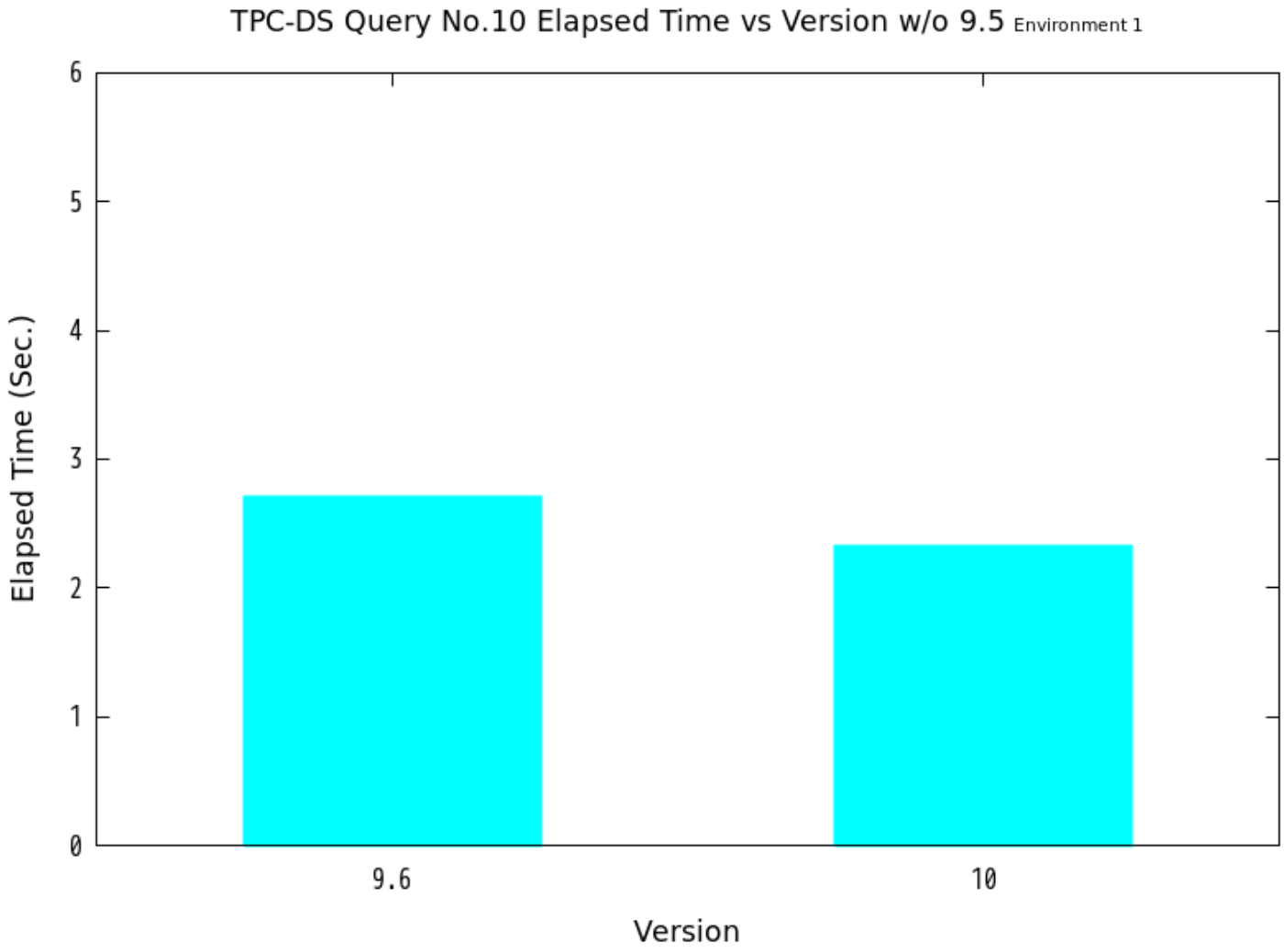


図 6.2 バージョンと処理時間の比較 (No.10/検証サーバ1 9.6と10を抜粋)

表 6.8 バージョンと処理時間 (No.10/検証サーバ1)

バージョン	処理時間(秒)	性能向上比(倍) [8]	パラレルクエリ選択有無
9.5	10247	1.00	—
9.6	2.7083	3783.6	○
10	2.3205	4415.9	○

[8] 性能向上比は、バージョン9.5の結果を、比較対象の各バージョンでの結果で除したものです。以下の性能向上比も、すべてこの方法で計算しています。

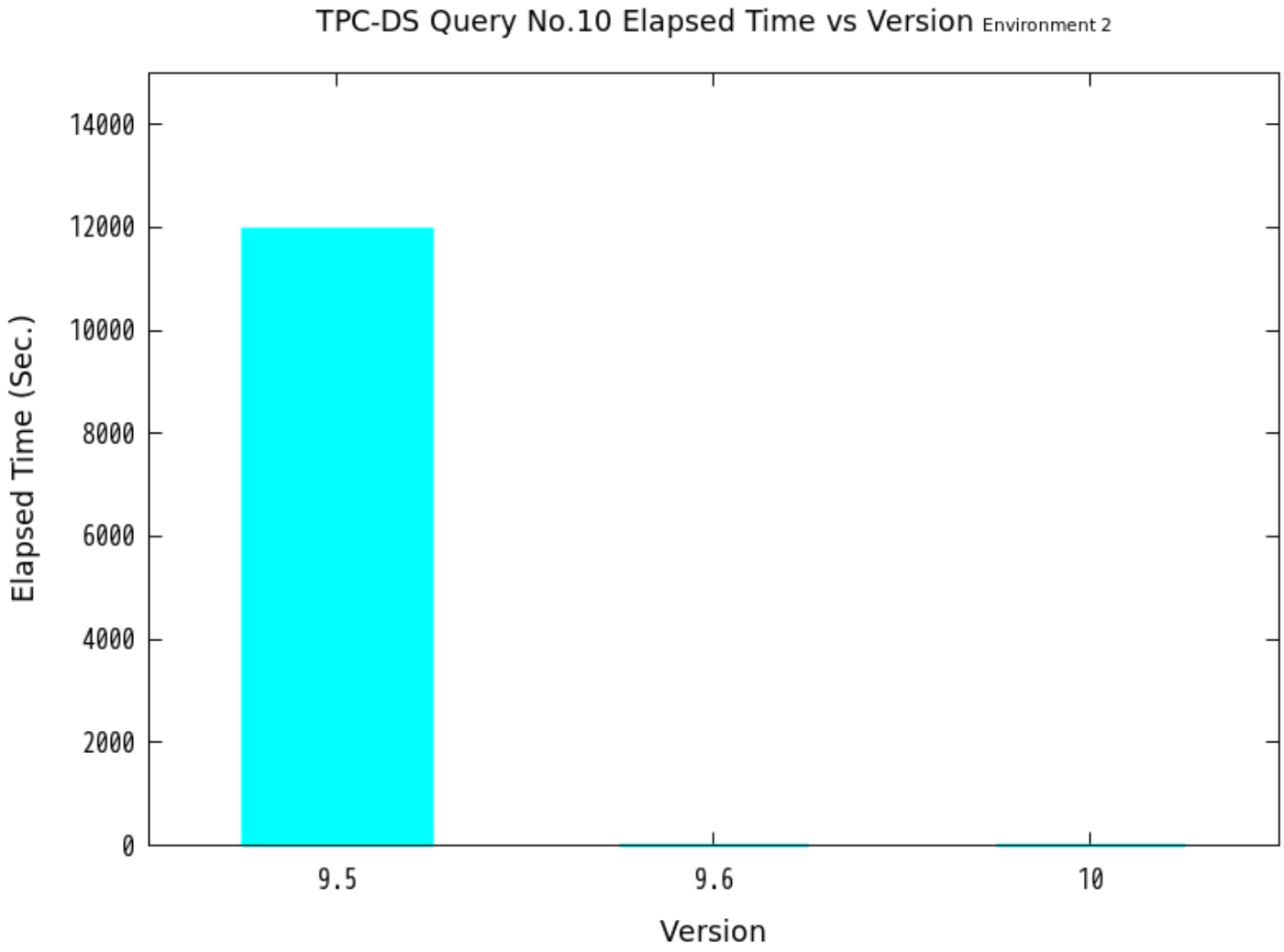


図 6.3 バージョンと処理時間の比較 (No.10/検証サーバ2)

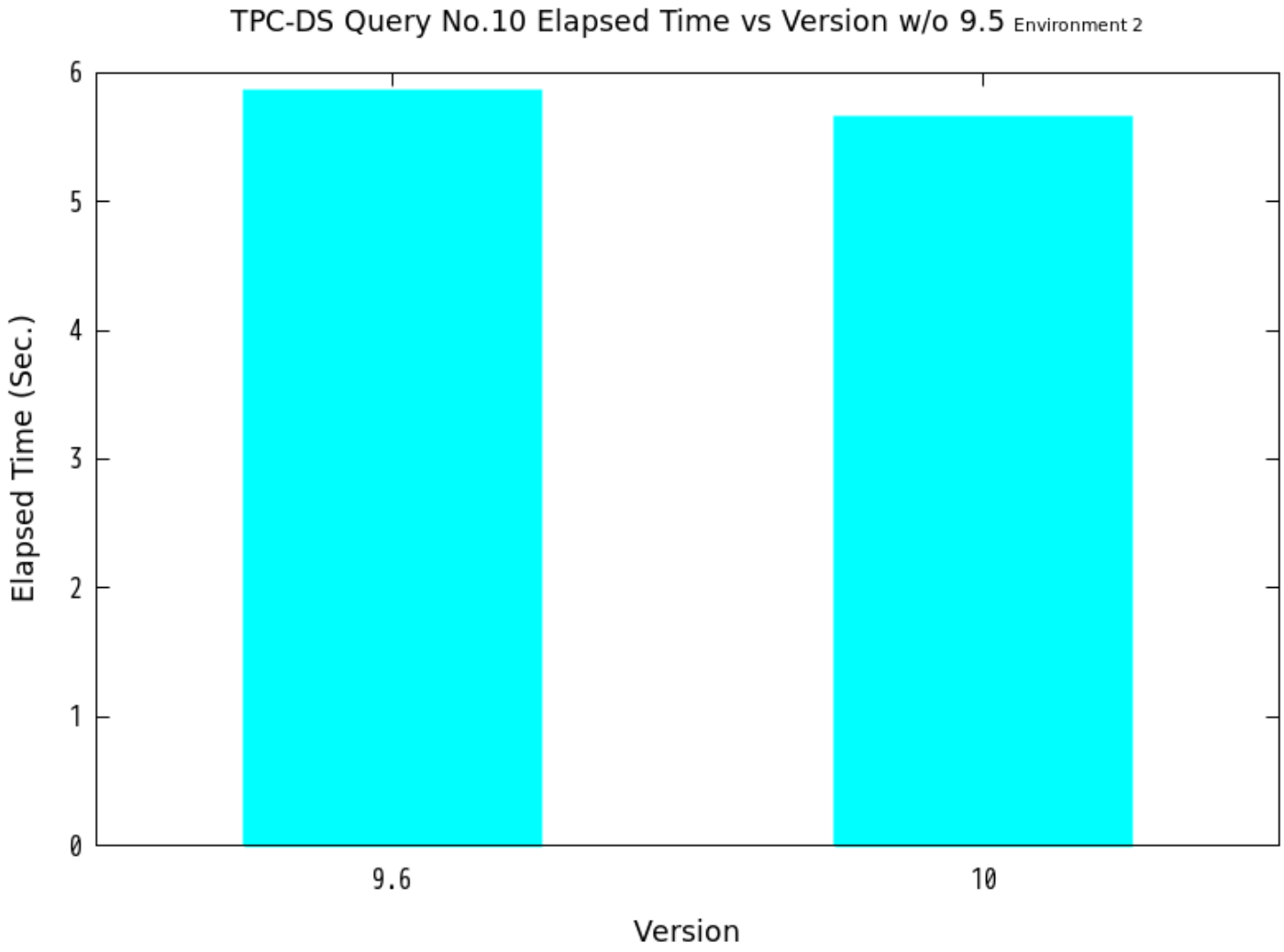


図 6.4 バージョンと処理時間の比較 (No.10/検証サーバ2 9.6と10を抜粋)

表 6.9 バージョンと処理時間 (No.10/検証サーバ2)

バージョン	処理時間(秒)	性能向上比(倍)	パラレルクエリ選択有無
9.5	11968	1.00	—
9.6	5.8563	2043.6	○
10	5.6491	2118.6	○

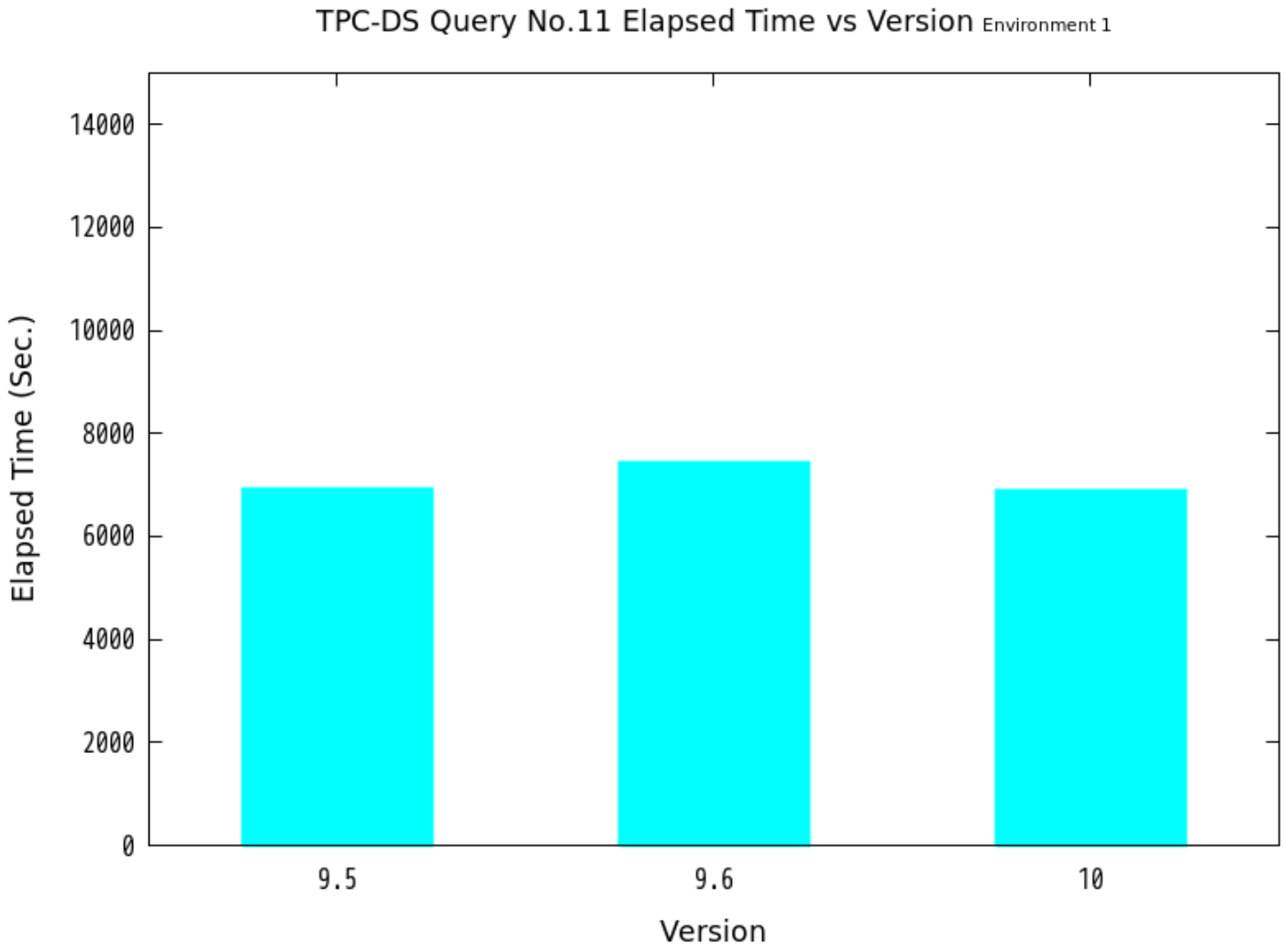


図 6.5 バージョンと処理時間の比較 (No.11/検証サーバ1)

表 6.10 バージョンと処理時間 (No.11/検証サーバ1)

バージョン	処理時間(秒)	性能向上比(倍)	パラレルクエリ選択有無
9.5	6913.6	1.00	—
9.6	7420.9	0.93164	×
10	6899.5	1.0020	○

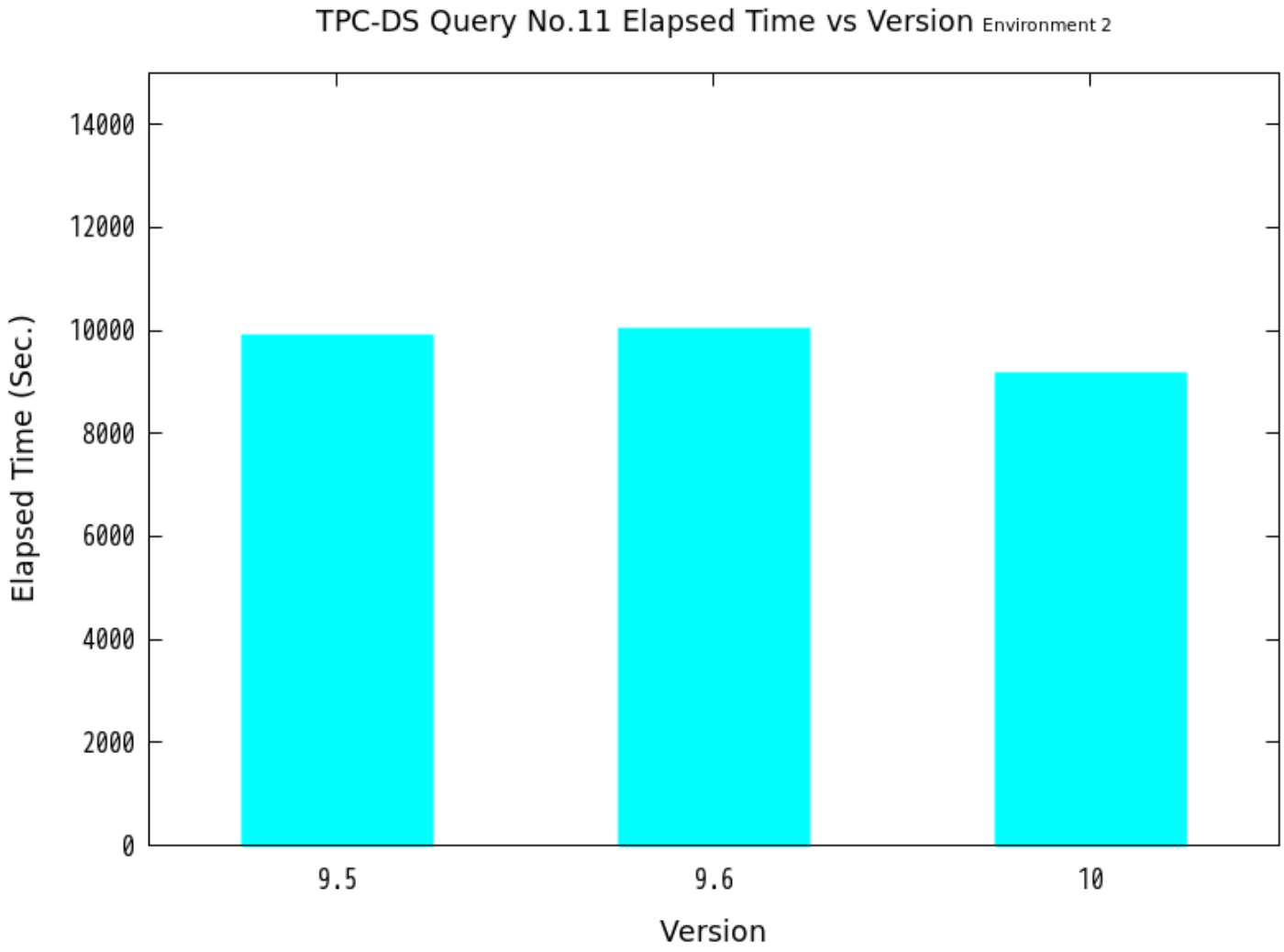


図 6.6 バージョンと処理時間の比較 (No.11/検証サーバ2)

表 6.11 バージョンと処理時間 (No.11/検証サーバ2)

バージョン	処理時間(秒)	性能向上比(倍)	パラレルクエリ選択有無
9.5	9882.6	1.00	—
9.6	10013	0.98698	×
10	9167.4	1.0780	○

検証の結果、No.10はバージョン9.6・バージョン10ともにパラレルクエリの実行が行われ、バージョン9.5と比較して、劇的に高速化したことがわかります。一方、No.11ではバージョン10のみパラレルクエリの実行が行われましたが、バージョン9.5と比較してあまり高速化しませんでした。

なお、No.11のクエリにおいて、バージョン10のみパラレルクエリ実行となったのは、本クエリがバージョン10のパラレルクエリ機能強化において対応した、『非相関サブクエリのパラレル実行』に該当したものと考えます。

6.5.2. 検証B(並列処理数(起動するワーカーの数)の違いによる検証結果)

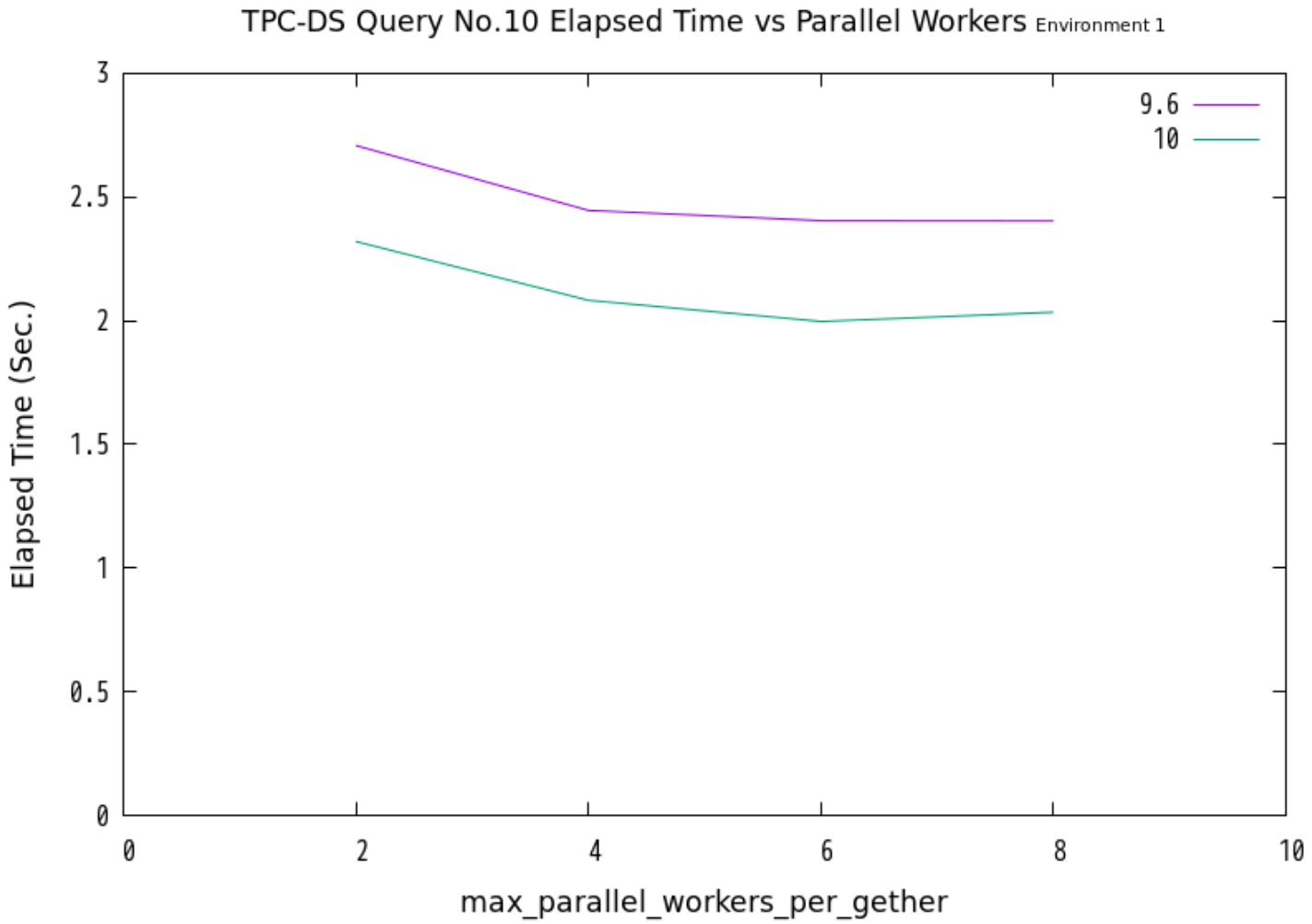


図 6.7 max_parallel_workersの値と処理時間の比較 (No.10/検証サーバ1)

表 6.12 並列処理数と処理時間 (No.10/検証サーバ1)

ワーカー数	9.6 (秒)	10 (秒)
2	2.7083	2.3205
4	2.4459	2.0827
6	2.4047	1.9970
8	2.4036	2.0346

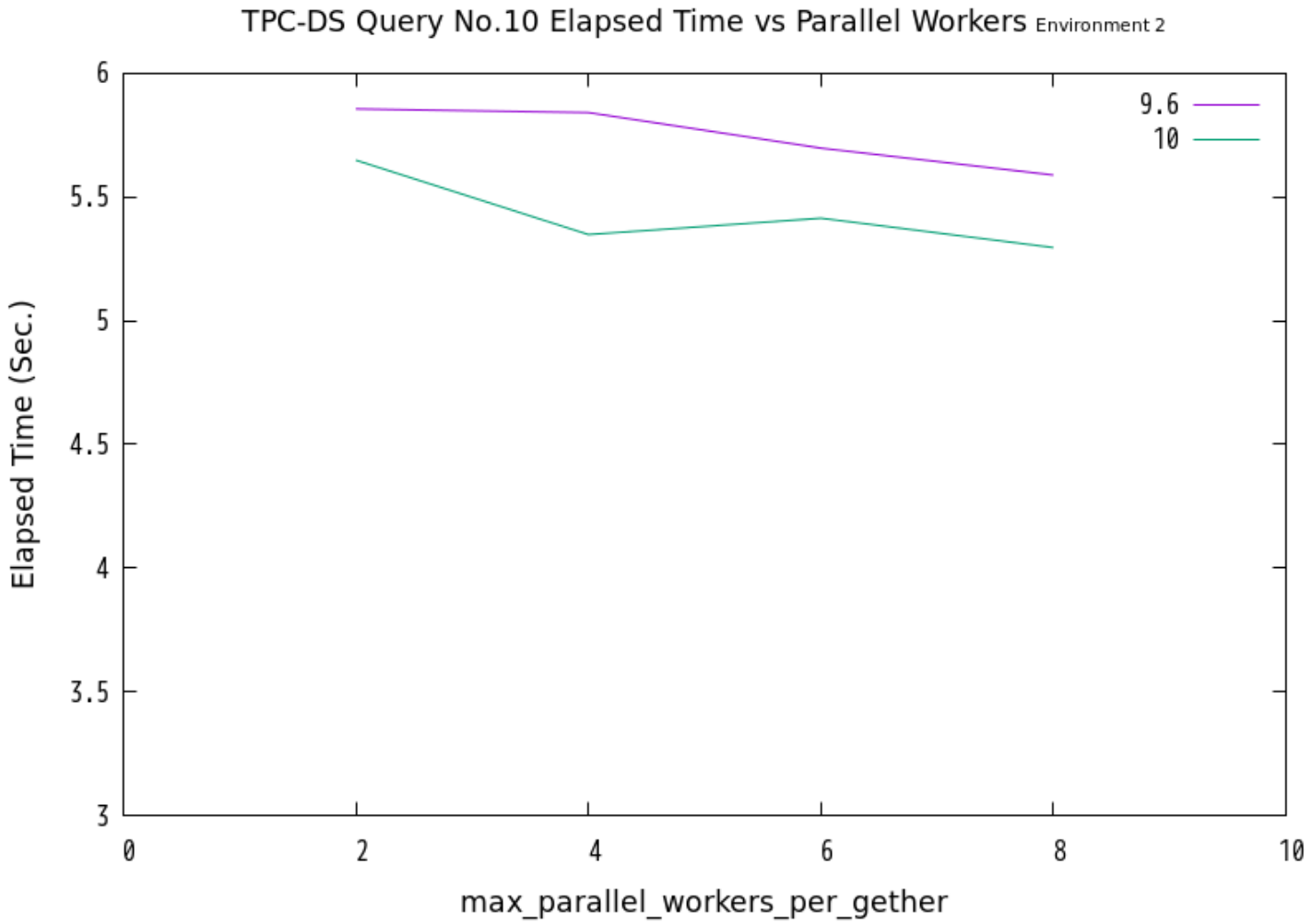


図 6.8 max_parallel_workersの値と処理時間の比較 (No.10/検証サーバ2)

表 6.13 並列処理数と処理時間 (No.10/検証サーバ2)

ワーカー数	9.6 (秒)	10 (秒)
2	5.8563	5.6491
4	5.8415	5.3810
6	5.6977	5.4467
8	5.5888	5.2956

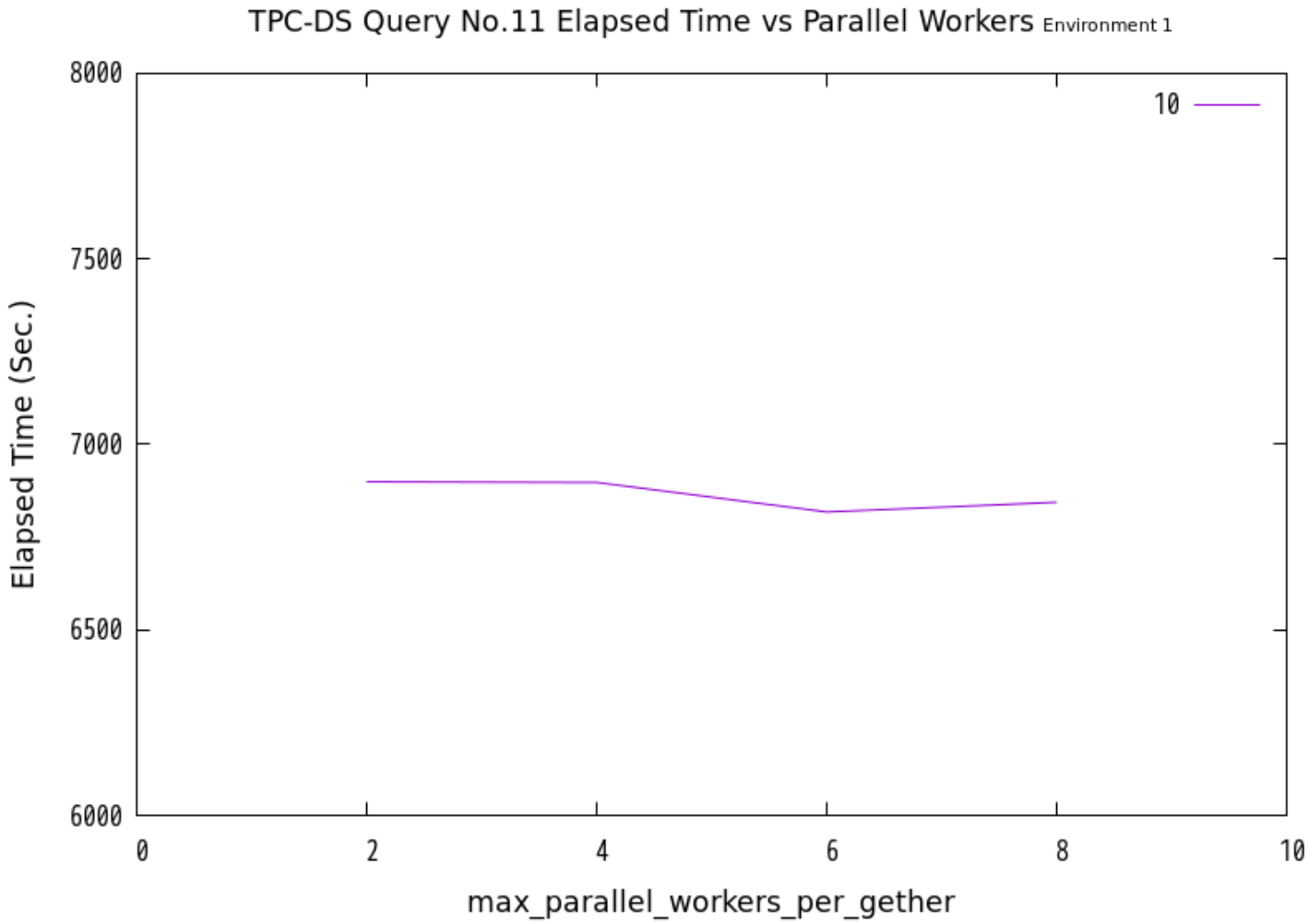


図 6.9 max_parallel_workersの値と処理時間の比較 (No.11/検証サーバ1)

表 6.14 並列処理数と処理時間 (No.11/検証サーバ1)

ワーカー数	10 (秒)
2	6899.5
4	6897.4
6	6818.1
8	6844.0

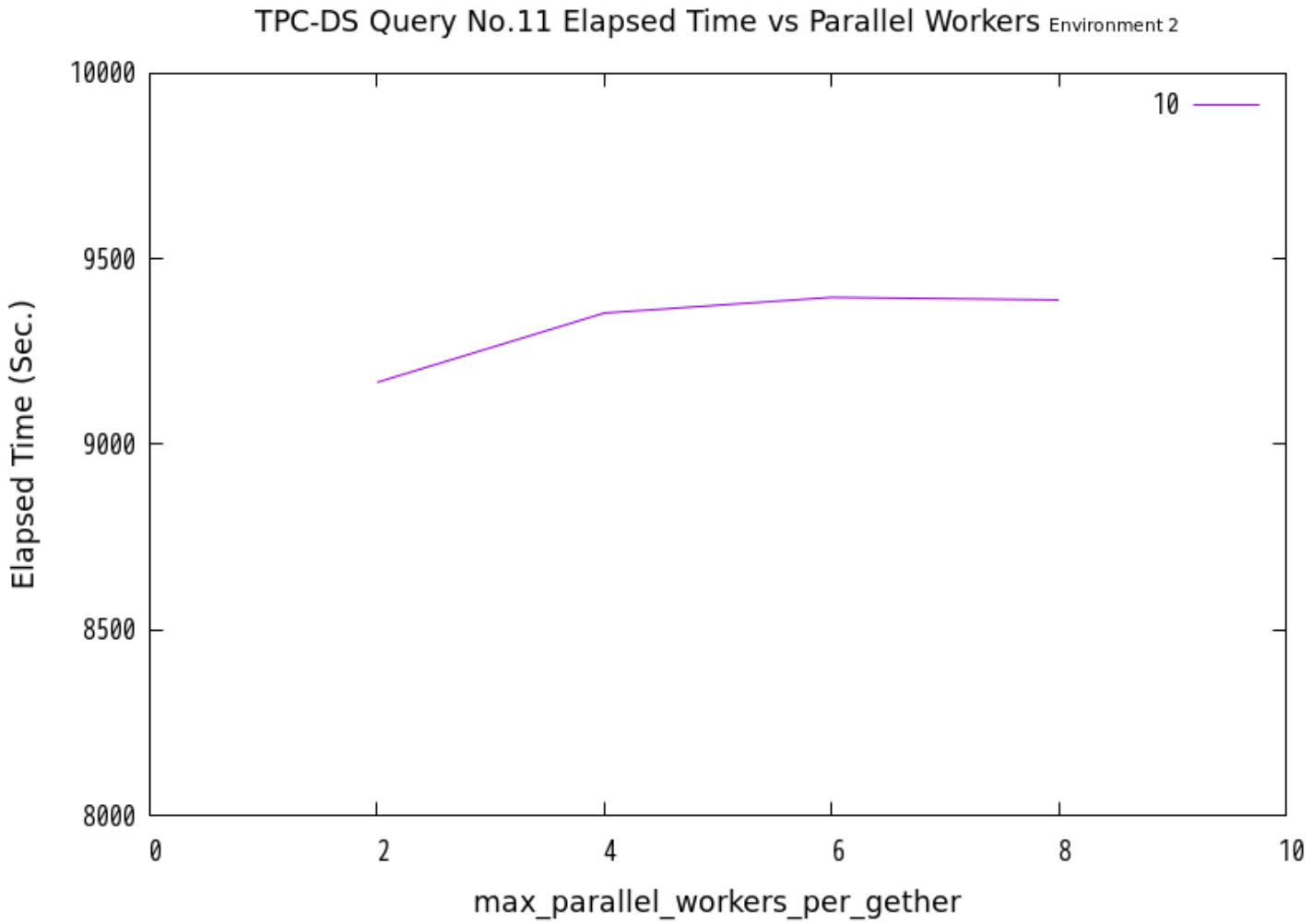


図 6.10 max_parallel_workersの値と処理時間の比較 (No.11/検証サーバ2)

表 6.15 並列処理数と処理時間 (No.11/検証サーバ2)

ワーカー数	10 (秒)
2	9167.4
4	9354.1
6	9396.5
8	9389.6

検証の結果、No.10は、並列処理数の増加にともなって、処理時間が減少する傾向にありました。一方、No.11では並列処理数の増加に伴って処理時間が減少せず、特に検証サーバ2においては処理時間が増加する傾向が見られました。

6.5.2.1. 検証B補足 パラレルで動作した部分の比較(検証サーバ2)

パラレルクエリ実行した部分の処理時間は、理論上、並列処理数に反比例する関係がありますが、検証Bのクエリの処理時間全体での比較データでは反比例の関係にあることを推測できるものとはなりませんでした。

ここでは、クエリ全体の処理に要した時間ではなく、パラレルクエリ実行となった部分のみの時間を抜き出して、理論と合致する傾向にあるかどうかを検証しました。

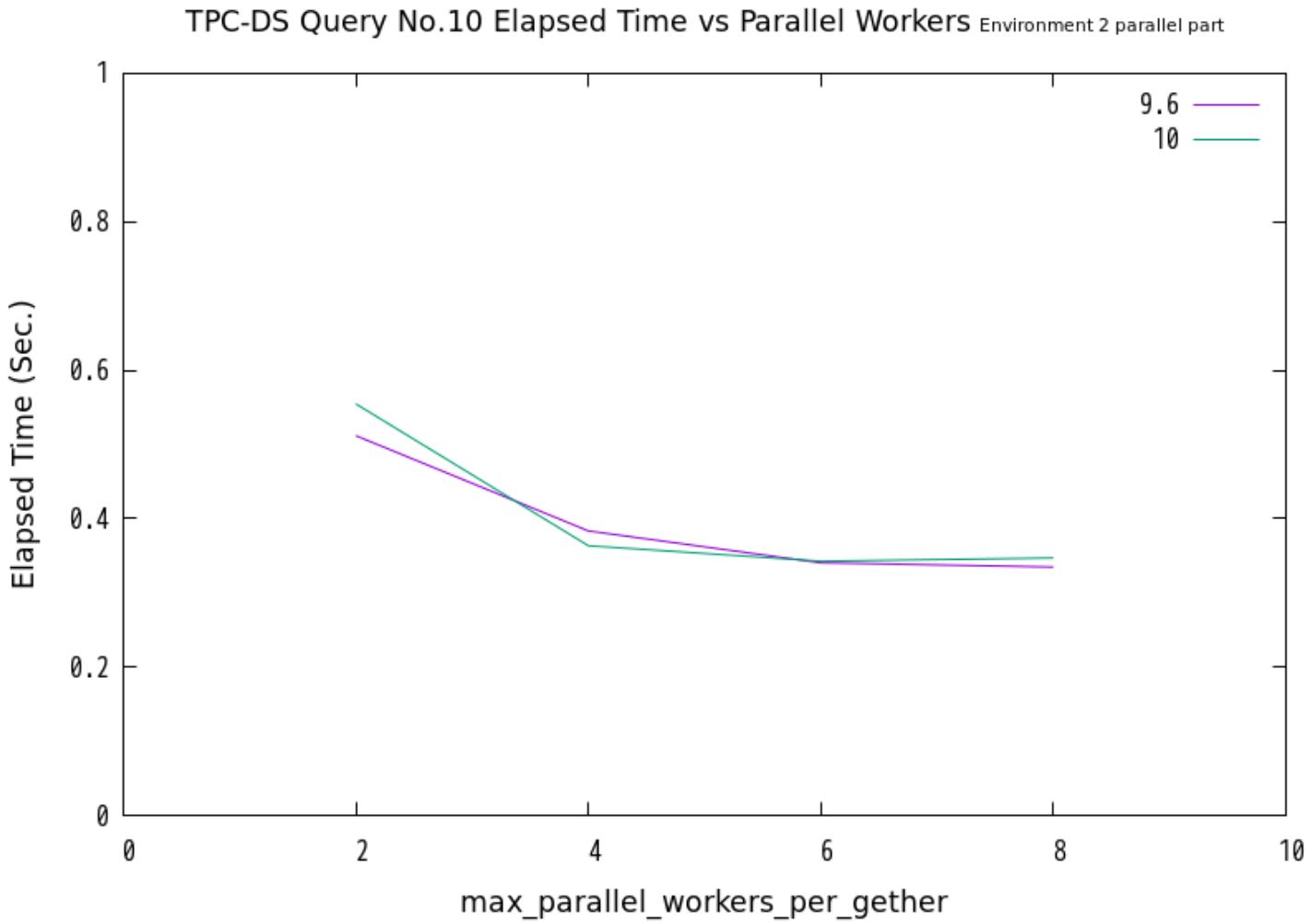


図 6.11 max_parallel_workersの値と処理時間の比較 (No.10/検証サーバ2 パラレル実行部分のみ)

表 6.16 並列処理数と処理時間 (No.10/検証サーバ2)

ワーカー数	9.6 (秒)	10 (秒)
2	0.5117	0.5546
4	0.3834	0.3635
6	0.3404	0.3426
8	0.3348	0.3469

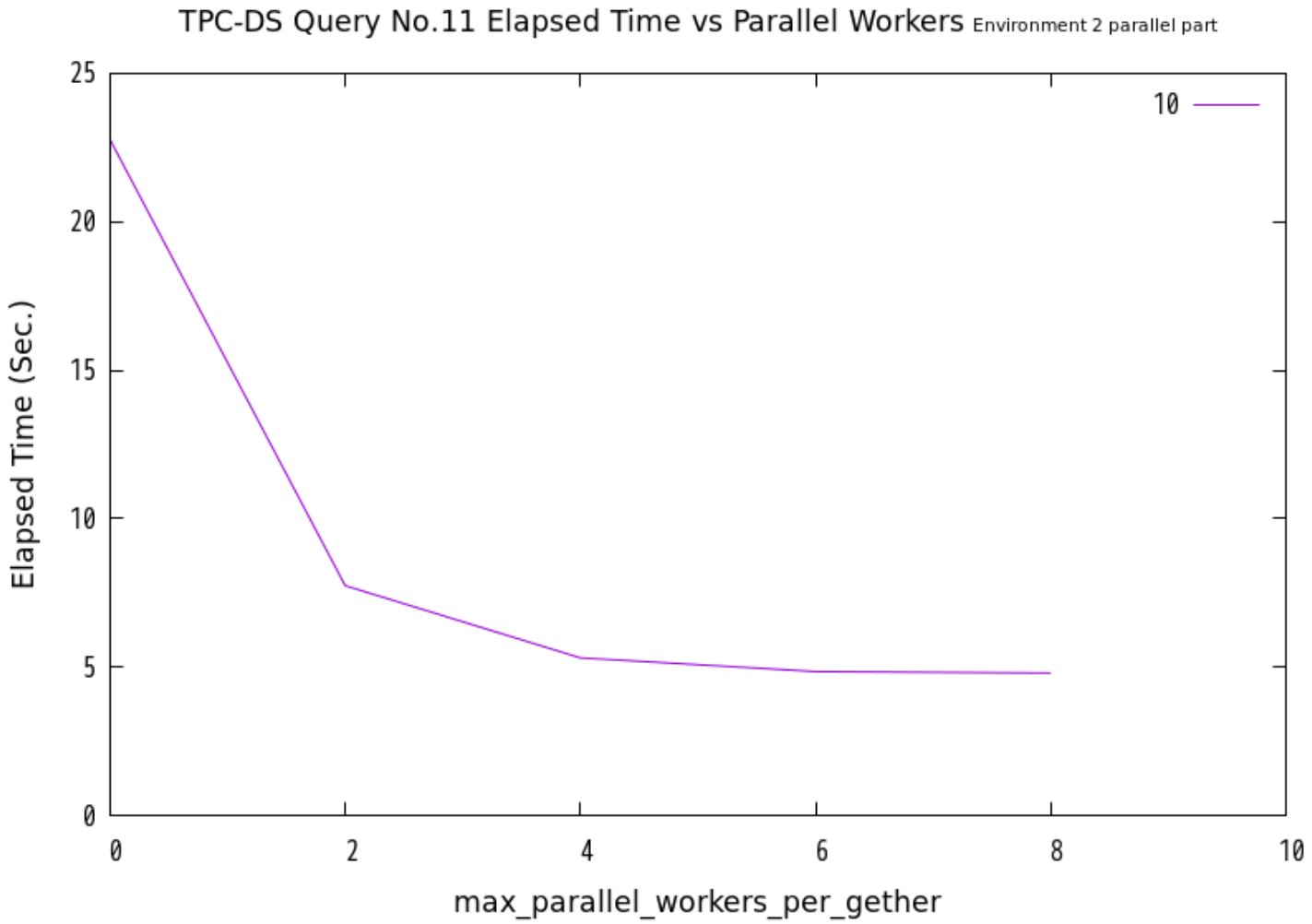


図 6.12 max_parallel_workersの値と処理時間の比較 (No.11/検証サーバ2 パラレル実行部分のみ) [9]

[9] max_parallel_workers_per_gather=0 は、パラレル実行offを意味します。

表 6.17 並列処理数と処理時間 (No.11/検証サーバ2)

ワーカー数	10 (秒)
0	22.77
2	7.74
4	5.31
6	4.85
8	4.80

No.10、No.11のクエリにおいて、パラレルクエリ実行となった部分のみを抜き出して比較したところ、両クエリともに、並列処理数の増加にともなって、処理時間が減少する傾向にありました。グラフの形状からは、パラレルクエリ実行された部分の処理時間と並列処理数の間には、反比例に近い関係がありそうであると推測できます。

6.5.3. 対照群 (パラレルクエリoff)

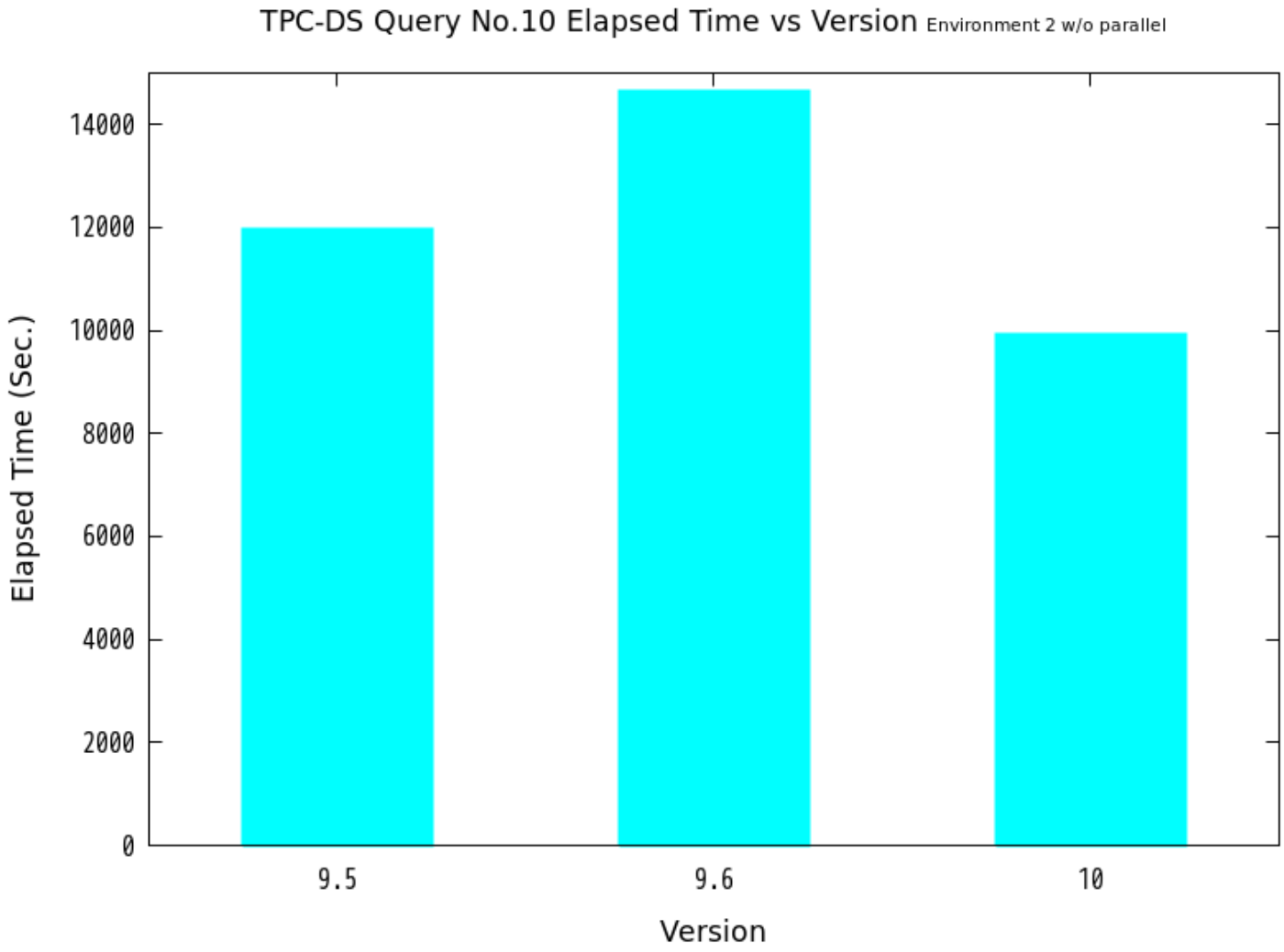


図 6.13 バージョンと処理時間の比較 (No.10/検証サーバ2 パラレル実行なし)

表 6.18 バージョンと処理時間 (No.10/検証サーバ2)

バージョン	処理時間(秒)	性能向上比(倍)
9.5	11968	1.00
9.6	14658	0.81648
10	9930.2	1.2052

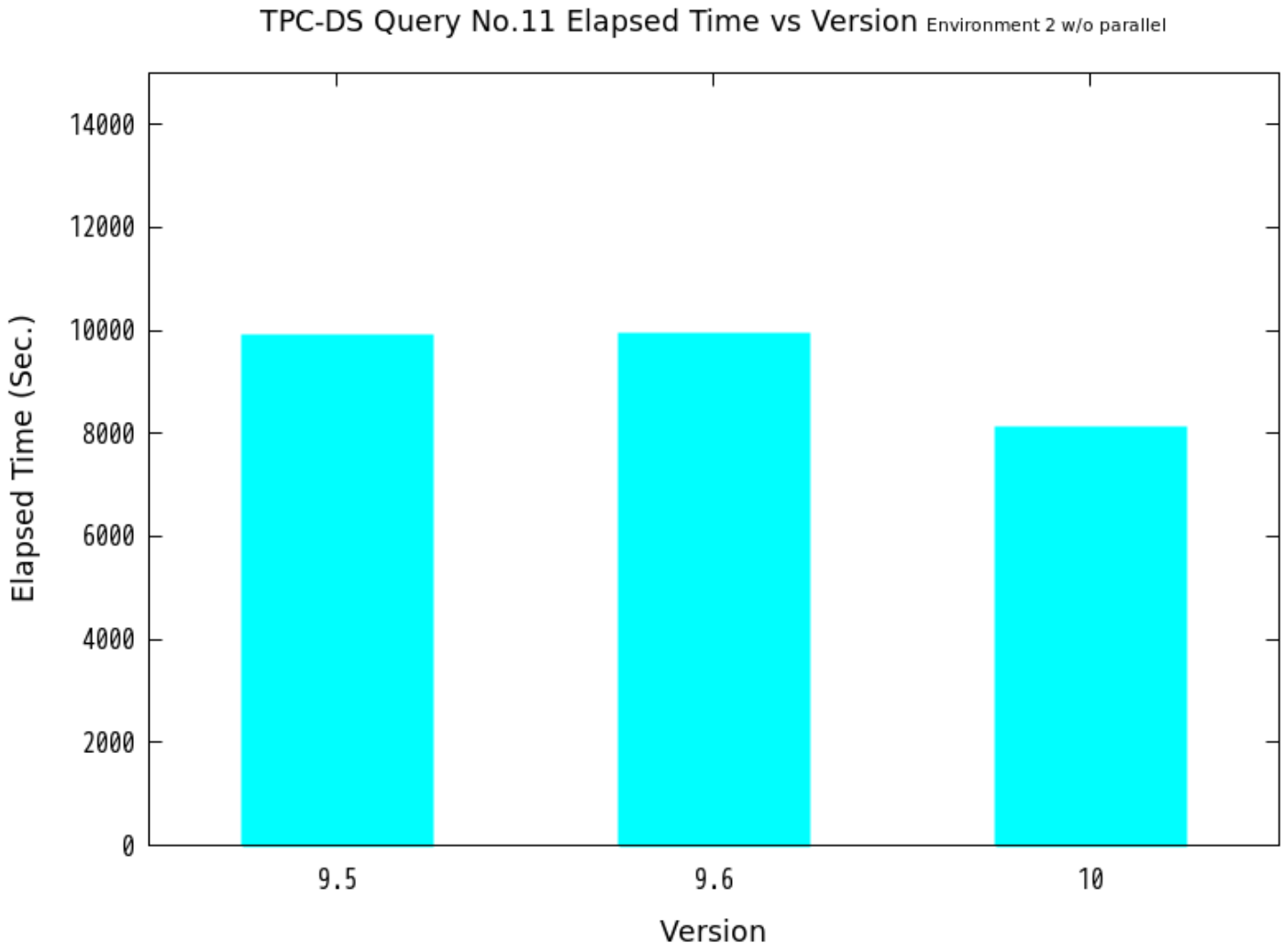


図 6.14 バージョンと処理時間の比較 (No.11/検証サーバ2 パラレル実行なし)

表 6.19 バージョンと処理時間 (No.11/検証サーバ2)

バージョン	処理時間(秒)	性能向上比(倍)
9.5	9882.6	1.00
9.6	9911.1	0.99712
10	8105.3	1.2193

検証の結果、パラレルクエリoffの状態で、バージョン9.5比でバージョン9.6はNo.10で20%、No.11で1%の性能低下が認められ、バージョン10はNo.10、No.11ともに概ね20%の性能向上が認められました。

ここから、パラレルクエリ以外の部分でバージョン9.6とバージョン10の間において改善が行われ、高速化したことが伺えます。

6.6. 考察

6.6.1. 検証A

6.6.1.1. No.10

No.10では、検証サーバ1、検証サーバ2ともに、パラレルクエリのないバージョン9.5よりも、パラレルクエリのあるバージョン9.6、およびバージョン10で劇的に高速化しました。また、バージョン9.6とバージョン10を比較すると、10のほうがより高速化しています。

ここで、検証サーバ2で採取した、バージョン9.5とバージョン9.6のEXPLAIN ANALYZEの結果を比較します。

- バージョン9.5

```
QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------
 Limit  (cost=1920510150.45..1920510152.68 rows=47 width=68) (actual time=11967560.428..11967560.439 rows=19 loops=1)
   ->  GroupAggregate  (cost=1920510150.45..1920510152.68 rows=47 width=68) (actual time=11967560.427..11967560.437 rows=19 loops=1)
         Group Key: customer_demographics.cd_gender, customer_demographics.cd_marital_status, customer_demographics.cd_education_status, cu
stomer_demographics.cd_purchase_estimate, customer_demographics.cd_credit_rating, customer_demographics.cd_dep_count, customer_demographics
.cd_dep_employed_count, customer_demographics.cd_dep_college_count
         ->  Sort  (cost=1920510150.45..1920510150.56 rows=47 width=68) (actual time=11967560.409..11967560.410 rows=19 loops=1)
               Sort Key: customer_demographics.cd_gender, customer_demographics.cd_marital_status, customer_demographics.cd_education_statu
s, customer_demographics.cd_purchase_estimate, customer_demographics.cd_credit_rating, customer_demographics.cd_dep_count, customer_demogra
phics.cd_dep_employed_count, customer_demographics.cd_dep_college_count
               Sort Method: quicksort  Memory: 27kB
               ->  Nested Loop  (cost=217984.67..1920510149.14 rows=47 width=68) (actual time=11963323.535..11967560.289 rows=19 loops=1)
                     Join Filter: (c.c_current_cdemo_sk = customer_demographics.cd_demo_sk)
                     Rows Removed by Join Filter: 36495181
                     ->  Seq Scan on customer_demographics  (cost=0.00..46648.00 rows=1920800 width=76) (actual time=0.005..202.805 rows=19
20800 loops=1)
                     ->  Materialize  (cost=217984.67..1919051713.26 rows=49 width=8) (actual time=0.114..6.229 rows=19 loops=1920800)
                           ->  Nested Loop  (cost=217984.67..1919051713.02 rows=49 width=8) (actual time=219831.780..11962737.942 rows=19 l
oops=1)
                                 ->  Nested Loop  (cost=217984.37..1919048998.10 rows=6909 width=16) (actual time=13693.957..11962695.074 r
ows=1912 loops=1)
                                       ->  HashAggregate  (cost=217983.95..218076.07 rows=9212 width=8) (actual time=1408.577..1502.045 row
s=17305 loops=1)
                                             Group Key: store_sales.ss_customer_sk
                                             ->  Hash Join  (cost=3254.88..217960.92 rows=9212 width=8) (actual time=21.141..1376.343 rows=
211451 loops=1)
                                                   Hash Cond: (store_sales.ss_sold_date_sk = date_dim.d_date_sk)
                                                   ->  Seq Scan on store_sales  (cost=0.00..193010.67 rows=5760867 width=16) (actual time=0
.011..830.192 rows=5760749 loops=1)
                                                   ->  Hash  (cost=3253.36..3253.36 rows=122 width=8) (actual time=21.083..21.083 rows=122
loops=1)
                                                         Buckets: 1024  Batches: 1  Memory Usage: 13kB
                                                         ->  Seq Scan on date_dim  (cost=0.00..3253.36 rows=122 width=8) (actual time=10.94
7..21.057 rows=122 loops=1)
                                                               Filter: ((d_moy >= 4) AND (d_moy <= 7) AND (d_year = 2002))
                                                               Rows Removed by Filter: 72927
                                       ->  Index Scan using customer_pkey on customer c  (cost=0.42..208296.88 rows=1 width=24) (actual tim
e=691.194..691.194 rows=0 loops=17305)
                                             Index Cond: (c_customer_sk = store_sales.ss_customer_sk)
                                             Filter: ((alternatives: SubPlan 1 or hashed SubPlan 2) OR (alternatives: SubPlan 3 or hashed S
ubPlan 4))
                                             Rows Removed by Filter: 1
                                             SubPlan 1
                                               ->  Nested Loop  (cost=0.29..69605.82 rows=1 width=0) (actual time=246.900..246.900 rows=0 l
oops=17304)
                                                     ->  Seq Scan on web_sales  (cost=0.00..69376.65 rows=28 width=8) (actual time=169.745.
.246.871 rows=9 loops=17304)
                                                           Filter: (c.c_customer_sk = ws_bill_customer_sk)
                                                           Rows Removed by Filter: 1416796
                                                     ->  Index Scan using date_dim_pkey on date_dim date_dim_1  (cost=0.29..8.17 rows=1 wid
th=8) (actual time=0.002..0.002 rows=0 loops=163513)
                                                           Index Cond: (d_date_sk = web_sales.ws_sold_date_sk)
                                                           Filter: ((d_moy >= 4) AND (d_moy <= 7) AND (d_year = 2002))
                                                           Rows Removed by Filter: 1
                                          ★1 SubPlan 2
                                               ->  Hash Join  (cost=3254.88..74454.63 rows=2403 width=8) (never executed)
                                                     Hash Cond: (web_sales_1.ws_sold_date_sk = date_dim_2.d_date_sk)
                                                     ->  Seq Scan on web_sales web_sales_1  (cost=0.00..65778.52 rows=1439252 width=16) (ne
ver executed)
                                                     ->  Hash  (cost=3253.36..3253.36 rows=122 width=8) (never executed)
                                                           ->  Seq Scan on date_dim date_dim_2  (cost=0.00..3253.36 rows=122 width=8) (neve
r executed)
                                                                 Filter: ((d_moy >= 4) AND (d_moy <= 7) AND (d_year = 2002))
                                             SubPlan 3
                                               ->  Nested Loop  (cost=0.29..138690.29 rows=1 width=0) (actual time=458.050..458.050 rows=0
loops=16785)
                                                     ->  Seq Scan on catalog_sales  (cost=0.00..138323.23 rows=46 width=8) (actual time=196
.035..457.985 rows=19 loops=16785)
                                                           Filter: (c.c_customer_sk = cs_ship_customer_sk)
                                                           Rows Removed by Filter: 2842597
                                                     ->  Index Scan using date_dim_pkey on date_dim date_dim_3  (cost=0.29..7.97 rows=1 wid
th=8) (actual time=0.002..0.002 rows=0 loops=319521)
                                                           Index Cond: (d_date_sk = catalog_sales.cs_sold_date_sk)
                                                           Filter: ((d_moy >= 4) AND (d_moy <= 7) AND (d_year = 2002))
                                                           Rows Removed by Filter: 1
                                          ★2 SubPlan 4
                                               ->  Hash Join  (cost=3254.88..145226.12 rows=4789 width=8) (never executed)
                                                     Hash Cond: (catalog_sales_1.cs_sold_date_sk = date_dim_4.d_date_sk)
                                                     ->  Seq Scan on catalog_sales catalog_sales_1  (cost=0.00..131122.98 rows=2880098 widt
```

```
h=16) (never executed)
                                                  ->  Hash  (cost=3253.36..3253.36 rows=122 width=8) (never executed)
                                                        ->  Seq Scan on date_dim date_dim_4  (cost=0.00..3253.36 rows=122 width=8) (neve
r executed)
                                                              Filter: ((d_moy >= 4) AND (d_moy <= 7) AND (d_year = 2002))
                                  ->  Index Scan using customer_address_pkey on customer_address ca  (cost=0.29..0.38 rows=1 width=8) (actua
l time=0.017..0.017 rows=0 loops=1912)
                                        Index Cond: (ca_address_sk = c.c_current_addr_sk)
                                        Filter: ((ca_county)::text = ANY ('{"Walker County","Richland County","Gaines County","Douglas Count
y","Dona Ana County"}'::text[]))
                                        Rows Removed by Filter: 1
 Planning time: 2.273 ms
 Execution time: 11967560.997 ms
(65 rows)
```

- バージョン9.6

```
QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------
 Limit  (cost=1989435197.83..1989435200.20 rows=50 width=116) (actual time=5837.880..5837.890 rows=19 loops=1)
   ->  GroupAggregate  (cost=1989435197.83..1989435200.20 rows=50 width=116) (actual time=5837.878..5837.886 rows=19 loops=1)
         Group Key: customer_demographics.cd_gender, customer_demographics.cd_marital_status, customer_demographics.cd_education_status, cu
stomer_demographics.cd_purchase_estimate, customer_demographics.cd_credit_rating, customer_demographics.cd_dep_count, customer_demographics
.cd_dep_employed_count, customer_demographics.cd_dep_college_count
         ->  Sort  (cost=1989435197.83..1989435197.95 rows=50 width=68) (actual time=5837.864..5837.865 rows=19 loops=1)
               Sort Key: customer_demographics.cd_gender, customer_demographics.cd_marital_status, customer_demographics.cd_education_statu
s, customer_demographics.cd_purchase_estimate, customer_demographics.cd_credit_rating, customer_demographics.cd_dep_count, customer_demogra
phics.cd_dep_employed_count, customer_demographics.cd_dep_college_count
               Sort Method: quicksort  Memory: 27kB
               ->  Nested Loop  (cost=173679.20..1989435196.42 rows=50 width=68) (actual time=1725.947..5837.733 rows=19 loops=1)
                     Join Filter: (c.c_current_cdemo_sk = customer_demographics.cd_demo_sk)
                     Rows Removed by Join Filter: 36495181
                     ->  Seq Scan on customer_demographics  (cost=0.00..46648.00 rows=1920800 width=76) (actual time=0.007..243.870 rows=19
20800 loops=1)
                     ->  Materialize  (cost=173679.20..1987947948.54 rows=50 width=8) (actual time=0.001..0.001 rows=19 loops=1920800)
                           ->  Nested Loop  (cost=173679.20..1987947948.29 rows=50 width=8) (actual time=1099.634..1160.304 rows=19 loops=1
)
                                 ->  Nested Loop  (cost=173678.90..1987945135.92 rows=7157 width=16) (actual time=1097.034..1152.777 rows=1
912 loops=1)
                                       ->  HashAggregate  (cost=173678.48..173773.91 rows=9543 width=8) (actual time=557.000..561.240 rows=
17305 loops=1)
                                             Group Key: store_sales.ss_customer_sk
                                             ->  Gather  (cost=4254.87..173654.63 rows=9543 width=8) (actual time=22.603..525.341 rows=2114
51 loops=1)
                                                   Workers Planned: 2
                                                   Workers Launched: 2
                                                   ->  Hash Join  (cost=3254.87..171700.33 rows=3976 width=8) (actual time=20.033..514.916
rows=70484 loops=3)
                                                         Hash Cond: (store_sales.ss_sold_date_sk = date_dim.d_date_sk)
                                                         ->  Parallel Seq Scan on store_sales  (cost=0.00..159405.96 rows=2400396 width=16)
(actual time=0.013..322.422 rows=1920250 loops=3)
                                                         ->  Hash  (cost=3253.36..3253.36 rows=121 width=8) (actual time=19.853..19.853 row
s=122 loops=3)
                                                               Buckets: 1024  Batches: 1  Memory Usage: 13kB
                                                               ->  Seq Scan on date_dim  (cost=0.00..3253.36 rows=121 width=8) (actual time
=9.799..19.836 rows=122 loops=3)
                                                                     Filter: ((d_moy >= 4) AND (d_moy <= 7) AND (d_year = 2002))
                                                                     Rows Removed by Filter: 72927
                                       ->  Index Scan using customer_pkey on customer c  (cost=0.42..208296.27 rows=1 width=24) (actual tim
e=0.034..0.034 rows=0 loops=17305)
                                             Index Cond: (c_customer_sk = store_sales.ss_customer_sk)
                                             Filter: ((alternatives: SubPlan 1 or hashed SubPlan 2) OR (alternatives: SubPlan 3 or hashed S
ubPlan 4))
                                             Rows Removed by Filter: 1
                                           ★3 SubPlan 1
                                             ->  Nested Loop  (cost=0.29..69605.71 rows=1 width=0) (never executed)
                                                   ->  Seq Scan on web_sales  (cost=0.00..69376.54 rows=28 width=8) (never executed)
                                                         Filter: (c.c_customer_sk = ws_bill_customer_sk)
                                                   ->  Index Scan using date_dim_pkey on date_dim date_dim_1  (cost=0.29..8.17 rows=1 wid
th=8) (never executed)
                                                         Index Cond: (d_date_sk = web_sales.ws_sold_date_sk)
                                                         Filter: ((d_moy >= 4) AND (d_moy <= 7) AND (d_year = 2002))
                                             SubPlan 2
                                               ->  Gather  (cost=4254.87..64134.86 rows=2384 width=8) (actual time=19.795..181.225 rows=552
59 loops=1)
                                                     Workers Planned: 2
                                                     Workers Launched: 2
                                                     ->  Hash Join  (cost=3254.87..62896.46 rows=993 width=8) (actual time=19.257..176.775
rows=18420 loops=3)
```

```
                                                  Hash Cond: (web_sales_1.ws_sold_date_sk = date_dim_2.d_date_sk)
                                                  ->  Parallel Seq Scan on web_sales web_sales_1  (cost=0.00..57382.85 rows=599685
width=16) (actual time=0.009..107.567 rows=479749 loops=3)
                                                  ->  Hash  (cost=3253.36..3253.36 rows=121 width=8) (actual time=19.055..19.055 r
ows=122 loops=3)
                                                        Buckets: 1024  Batches: 1  Memory Usage: 13kB
                                                        ->  Seq Scan on date_dim date_dim_2  (cost=0.00..3253.36 rows=121 width=8)
(actual time=9.966..19.026 rows=122 loops=3)
                                                              Filter: ((d_moy >= 4) AND (d_moy <= 7) AND (d_year = 2002))
                                                              Rows Removed by Filter: 72927
                                        ★4 SubPlan 3
                                              ->  Nested Loop  (cost=0.29..138689.82 rows=1 width=0) (never executed)
                                                    ->  Seq Scan on catalog_sales  (cost=0.00..138322.75 rows=46 width=8) (never executed)
                                                          Filter: (c.c_customer_sk = cs_ship_customer_sk)
                                                    ->  Index Scan using date_dim_pkey on date_dim date_dim_3  (cost=0.29..7.97 rows=1 wid
th=8) (never executed)
                                                          Index Cond: (d_date_sk = catalog_sales.cs_sold_date_sk)
                                                          Filter: ((d_moy >= 4) AND (d_moy <= 7) AND (d_year = 2002))
                                              SubPlan 4
                                                ->  Gather  (cost=4254.87..123574.10 rows=4771 width=8) (actual time=251.298..330.496 rows=1
10495 loops=1)
                                                      Workers Planned: 2
                                                      Workers Launched: 2
                                                      ->  Hash Join  (cost=3254.87..122097.00 rows=1988 width=8) (actual time=248.959..310.3
83 rows=36832 loops=3)
                                                            Hash Cond: (catalog_sales_1.cs_sold_date_sk = date_dim_4.d_date_sk)
                                                            ->  Parallel Seq Scan on catalog_sales catalog_sales_1  (cost=0.00..114322.25 ro
ws=1200025 width=16) (actual time=0.009..200.903 rows=960019 loops=3)
                                                            ->  Hash  (cost=3253.36..3253.36 rows=121 width=8) (actual time=17.051..17.051 r
ows=122 loops=3)
                                                                  Buckets: 1024  Batches: 1  Memory Usage: 13kB
                                                                  ->  Seq Scan on date_dim date_dim_4  (cost=0.00..3253.36 rows=121 width=8)
(actual time=8.867..17.027 rows=122 loops=3)
                                                                        Filter: ((d_moy >= 4) AND (d_moy <= 7) AND (d_year = 2002))
                                                                        Rows Removed by Filter: 72927
                                    ->  Index Scan using customer_address_pkey on customer_address ca  (cost=0.29..0.38 rows=1 width=8) (actua
l time=0.004..0.004 rows=0 loops=1912)
                                          Index Cond: (ca_address_sk = c.c_current_addr_sk)
                                          Filter: ((ca_county)::text = ANY ('{"Walker County","Richland County","Gaines County","Douglas Count
y","Dona Ana County"}'::text[]))
                                          Rows Removed by Filter: 1
 Planning time: 2.310 ms
 Execution time: 5838.753 ms
(74 rows)
```

ここで注目するのは、never executedとなっている部分 [10] です。バージョン9.5ではSubPlan 2 (上記バージョン9.5のEXPLAIN ANALYZE ★1印)とSubPlan 4 (上記バージョン9.5のEXPLAIN ANALYZE ★2印)がnever executedとなっているのに対し、バージョン9.6ではSubPlan 1 (上記バージョン9.6のEXPLAIN ANALYZE ★3印)とSubPlan 3 (上記バージョン9.6のEXPLAIN ANALYZE ★4印)がnever executedとなっています。

[10] 実行計画としてはあったが、実際の実行時に何らか(ジョイン対象のテーブル検索の結果が0件であった、など)の理由で、実行不要と判断され、実行されなかったことを示します。

ここから、実行計画としてはパラレルクエリ実行部分以外での差異はないものの、実際のテーブルに対する検索・ジョインの操作が行われた内容が異なっていることになります。

特に、バージョン9.5の場合、ジョイン操作の中では、データ量が多い場合に最も非効率とされているNested LoopをSubPlan 1では17304回、SubPlan 3では16785回繰り返しており、実行時間の長大化に拍車がかかった可能性があります。

対して、バージョン9.6の場合、ジョイン操作の中では、データ量が多い場合にNested Loopよりも効率的とされているHash JoinをSubPlan 2、SubPlan 4ともに1回のみ実行しており、実行時間の数千倍高速化という、劇的な短縮に寄与した可能性があります。

しかし、対照群(パラレルクエリoff) の結果から、バージョン9.5とパラレルクエリoffのバージョン9.6、および、バージョン10の性能差は、バージョン9.6がバージョン9.5比で-20%程度、バージョン10でも9.5比で+20%倍程度のスピードであり、上記の数千倍からは大きく乖離しています。

これは、パラレルクエリを有効にすることで、実行計画上の差異ではわからない差異が実際には発生し、実行時間がまったく異なる可能性がある、ということを示しています。No.10のクエリでは高速化しましたが、実際のクエリでは逆に大幅に実行時間が長くなることもあり得るため、注意が必要であると考えます。

なお、バージョン9.6とバージョン10の間のEXPLAIN ANALYZEの結果も同様でしたが、バージョン9.6とバージョン10ではバージョン10のほうが高速化しています。これは、対照群(パラレルクエリoff) での測定で示されたとおり、パラレルクエリ以外での改善によるものと考えられます。

6.6.1.2. No.11

No.11では、検証サーバ1、検証サーバ2ともに、パラレルクエリのないバージョン9.5、パラレルクエリを選択しなかったバージョン9.6よりも、パラレルクエリを選択したバージョン10で高速化しました。

ここで、検証サーバ2で採取した、バージョン9.5とバージョン10のEXPLAIN ANALYZEの結果を比較します。

- バージョン9.5

```
QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------
 Limit  (cost=2098571.40..2098571.41 rows=4 width=132) (actual time=9882417.331..9882417.344 rows=100 loops=1)
   CTE year_total
     ->  Append  (cost=944045.46..1393256.05 rows=6715350 width=220) (actual time=31347.496..39832.048 rows=461362 loops=1)
           ->  GroupAggregate  (cost=944045.46..1155118.18 rows=5276818 width=166) (actual time=31347.494..37126.071 rows=350725 loops=1)
                 Group Key: customer.c_customer_id, customer.c_first_name, customer.c_last_name, customer.c_preferred_cust_flag, customer.c
_birth_country, customer.c_login, customer.c_email_address, date_dim.d_year
                 ->  Sort  (cost=944045.46..957237.51 rows=5276818 width=166) (actual time=31347.437..33488.240 rows=5371601 loops=1)
                       Sort Key: customer.c_customer_id, customer.c_first_name, customer.c_last_name, customer.c_preferred_cust_flag, custo
mer.c_birth_country, customer.c_login, customer.c_email_address, date_dim.d_year
                    ▲ Sort Method: external merge  Disk: 883896kB
                       ->  Hash Join  (cost=11694.60..354856.10 rows=5276818 width=166) (actual time=139.594..4694.187 rows=5371601 loops=1
)
                             Hash Cond: (store_sales.ss_sold_date_sk = date_dim.d_date_sk)
                             ->  Hash Join  (cost=8076.00..277802.21 rows=5511229 width=166) (actual time=104.472..3431.795 rows=5501511 lo
ops=1)
                                   Hash Cond: (store_sales.ss_customer_sk = customer.c_customer_sk)
                                   ->  Seq Scan on store_sales  (cost=0.00..193010.67 rows=5760867 width=26) (actual time=0.013..982.918 ro
ws=5760749 loops=1)
                                   ->  Hash  (cost=6276.00..6276.00 rows=144000 width=156) (actual time=103.472..103.472 rows=144000 loops=
1)
                                         Buckets: 262144  Batches: 1  Memory Usage: 26182kB
                                         ->  Seq Scan on customer  (cost=0.00..6276.00 rows=144000 width=156) (actual time=0.006..59.612 ro
ws=144000 loops=1)
                             ->  Hash  (cost=2705.49..2705.49 rows=73049 width=16) (actual time=34.605..34.605 rows=73049 loops=1)
                                   Buckets: 131072  Batches: 1  Memory Usage: 4449kB
                                   ->  Seq Scan on date_dim  (cost=0.00..2705.49 rows=73049 width=16) (actual time=0.011..23.222 rows=73049
loops=1)
           ->  HashAggregate  (cost=153002.71..170984.36 rows=1438532 width=169) (actual time=2596.201..2677.001 rows=110637 loops=1)
                 Group Key: customer_1.c_customer_id, customer_1.c_first_name, customer_1.c_last_name, customer_1.c_preferred_cust_flag, cu
stomer_1.c_birth_country, customer_1.c_login, customer_1.c_email_address, date_dim_1.d_year
                 ->  Hash Join  (cost=11694.60..117039.41 rows=1438532 width=169) (actual time=110.878..1271.497 rows=1438731 loops=1)
                       Hash Cond: (web_sales.ws_sold_date_sk = date_dim_1.d_date_sk)
                       ->  Hash Join  (cost=8076.00..93639.92 rows=1438820 width=169) (actual time=85.471..936.814 rows=1438929 loops=1)
                             Hash Cond: (web_sales.ws_bill_customer_sk = customer_1.c_customer_sk)
                             ->  Seq Scan on web_sales  (cost=0.00..65778.52 rows=1439252 width=29) (actual time=0.009..249.180 rows=143924
7 loops=1)
                             ->  Hash  (cost=6276.00..6276.00 rows=144000 width=156) (actual time=84.549..84.549 rows=144000 loops=1)
                                   Buckets: 262144  Batches: 1  Memory Usage: 26182kB
                                   ->  Seq Scan on customer customer_1  (cost=0.00..6276.00 rows=144000 width=156) (actual time=0.008..42.5
86 rows=144000 loops=1)
                       ->  Hash  (cost=2705.49..2705.49 rows=73049 width=16) (actual time=24.930..24.930 rows=73049 loops=1)
                             Buckets: 131072  Batches: 1  Memory Usage: 4449kB
                             ->  Seq Scan on date_dim date_dim_1  (cost=0.00..2705.49 rows=73049 width=16) (actual time=0.019..14.193 rows=
73049 loops=1)
   ->  Sort  (cost=705315.35..705315.36 rows=4 width=132) (actual time=9882417.328..9882417.334 rows=100 loops=1)
         Sort Key: t_s_secyear.customer_id, t_s_secyear.customer_first_name, t_s_secyear.customer_last_name, t_s_secyear.customer_email_add
ress
         Sort Method: top-N heapsort  Memory: 51kB
         ->  Nested Loop  (cost=0.00..705315.31 rows=4 width=132) (actual time=65298.027..9882415.035 rows=382 loops=1)
               Join Filter: ((t_s_secyear.customer_id = t_w_secyear.customer_id) AND (CASE WHEN (t_w_firstyear.year_total > '0'::numeric) T
HEN (t_w_secyear.year_total / t_w_firstyear.year_total) ELSE 0.0 END > CASE WHEN (t_s_firstyear.year_total > '0'::numeric) THEN (t_s_secyea
r.year_total / t_s_firstyear.year_total) ELSE 0.0 END))
               Rows Removed by Join Filter: 116789448
               ->  Nested Loop  (cost=0.00..537356.80 rows=13 width=268) (actual time=51185.761..9440691.635 rows=5287 loops=1)
                     Join Filter: (t_s_firstyear.customer_id = t_s_secyear.customer_id)
                     Rows Removed by Join Filter: 757357814
                     ->  Nested Loop  (cost=0.00..369414.25 rows=16 width=104) (actual time=41605.749..8383949.375 rows=10841 loops=1)
                           Join Filter: (t_s_firstyear.customer_id = t_w_firstyear.customer_id)
                           Rows Removed by Join Filter: 1554943027
                           ->  CTE Scan on year_total t_s_firstyear  (cost=0.00..184672.13 rows=56 width=52) (actual time=31347.529..31679.
091 rows=69732 loops=1)
                                 Filter: ((year_total > '0'::numeric) AND (sale_type = 's'::text) AND (dyear = 2001))
                                 Rows Removed by Filter: 391630
                           ->  CTE Scan on year_total t_w_firstyear  (cost=0.00..184672.13 rows=56 width=52) (actual time=86.506..117.877 r
ows=22299 loops=69732)
                                 Filter: ((year_total > '0'::numeric) AND (sale_type = 'w'::text) AND (dyear = 2001))
                                 Rows Removed by Filter: 439063
                     ->  CTE Scan on year_total t_s_secyear  (cost=0.00..167883.75 rows=168 width=164) (actual time=0.002..91.634 rows=6986
1 loops=10841)
                           Filter: ((sale_type = 's'::text) AND (dyear = 2002))
                           Rows Removed by Filter: 391501
               ->  CTE Scan on year_total t_w_secyear  (cost=0.00..167883.75 rows=168 width=52) (actual time=58.814..81.687 rows=22090 loop
s=5287)
                     Filter: ((sale_type = 'w'::text) AND (dyear = 2002))
                     Rows Removed by Filter: 439272
 Planning time: 1.732 ms
```

```
 Execution time: 9882579.077 ms
(58 rows)
```

- バージョン10

```
QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=2325263.07..2325263.08 rows=5 width=132) (actual time=9167300.012..9167300.025 rows=100 loops=1)
   CTE year_total
     ->  Append  (cost=487078.35..1569013.03 rows=7199995 width=220) (actual time=5368.720..10511.778 rows=461362 loops=1)
           ->  Finalize GroupAggregate  (cost=487078.35..1329228.33 rows=5760759 width=220) (actual time=5368.718..7740.506 rows=350725 loops=1)
                 Group Key: customer.c_customer_id, customer.c_first_name, customer.c_last_name, customer.c_preferred_cust_flag, customer.c_birth_country, customer.c_login, customer.c_email_address, date_dim.d_year
               ★ ->  Gather Merge  (cost=487078.35..1137203.04 rows=4800632 width=188) (actual time=5368.659..7284.982 rows=526909 loops=1)
                       Workers Planned: 2
                       Workers Launched: 2
                       ->  Partial GroupAggregate  (cost=486078.33..582090.97 rows=2400316 width=188) (actual time=5297.614..6795.760 rows=175636 loops=3)
                             Group Key: customer.c_customer_id, customer.c_first_name, customer.c_last_name, customer.c_preferred_cust_flag, customer.c_birth_country, customer.c_login, customer.c_email_address, date_dim.d_year
                             ->  Sort  (cost=486078.33..492079.12 rows=2400316 width=166) (actual time=5297.576..5509.034 rows=1790534 loops=3)
                                   Sort Key: customer.c_customer_id, customer.c_first_name, customer.c_last_name, customer.c_preferred_cust_flag, customer.c_birth_country, customer.c_login, customer.c_email_address, date_dim.d_year
                                 ▲ Sort Method: quicksort  Memory: 518574kB
                                   ->  Hash Join  (cost=11694.60..231707.33 rows=2400316 width=166) (actual time=129.090..1662.490 rows=1790534 loops=3)
                                         Hash Cond: (store_sales.ss_sold_date_sk = date_dim.d_date_sk)
                                         ->  Hash Join  (cost=8076.00..197785.01 rows=2400316 width=166) (actual time=97.578..1187.631 rows=1833837 loops=3)
                                               Hash Cond: (store_sales.ss_customer_sk = customer.c_customer_sk)
                                               ->  Parallel Seq Scan on store_sales  (cost=0.00..159405.16 rows=2400316 width=26) (actual time=0.014..341.113 rows=1920250 loops=3)
                                               ->  Hash  (cost=6276.00..6276.00 rows=144000 width=156) (actual time=96.616..96.616 rows=144000 loops=3)
                                                     Buckets: 262144  Batches: 1  Memory Usage: 26182kB
                                                     ->  Seq Scan on customer  (cost=0.00..6276.00 rows=144000 width=156) (actual time=0.015..52.952 rows=144000 loops=3)
                                         ->  Hash  (cost=2705.49..2705.49 rows=73049 width=16) (actual time=30.976..30.976 rows=73049 loops=3)
                                               Buckets: 131072  Batches: 1  Memory Usage: 4449kB
                                               ->  Seq Scan on date_dim  (cost=0.00..2705.49 rows=73049 width=16) (actual time=0.019..19.521 rows=73049 loops=3)
           ->  HashAggregate  (cost=149794.31..167784.76 rows=1439236 width=220) (actual time=2659.957..2741.482 rows=110637 loops=1)
                 Group Key: customer_1.c_customer_id, customer_1.c_first_name, customer_1.c_last_name, customer_1.c_preferred_cust_flag, customer_1.c_birth_country, customer_1.c_login, customer_1.c_email_address, date_dim_1.d_year
                 ->  Hash Join  (cost=11694.60..113813.41 rows=1439236 width=169) (actual time=114.108..1395.779 rows=1438731 loops=1)
                       Hash Cond: (web_sales.ws_sold_date_sk = date_dim_1.d_date_sk)
                       ->  Hash Join  (cost=8076.00..92024.63 rows=1439236 width=169) (actual time=86.674..1025.239 rows=1438929 loops=1)
                             Hash Cond: (web_sales.ws_bill_customer_sk = customer_1.c_customer_sk)
                             ->  Seq Scan on web_sales  (cost=0.00..65778.36 rows=1439236 width=29) (actual time=0.015..287.207 rows=1439247 loops=1)
                             ->  Hash  (cost=6276.00..6276.00 rows=144000 width=156) (actual time=85.756..85.756 rows=144000 loops=1)
                                   Buckets: 262144  Batches: 1  Memory Usage: 26182kB
                                   ->  Seq Scan on customer customer_1  (cost=0.00..6276.00 rows=144000 width=156) (actual time=0.008..43.611 rows=144000 loops=1)
                       ->  Hash  (cost=2705.49..2705.49 rows=73049 width=16) (actual time=26.996..26.996 rows=73049 loops=1)
                             Buckets: 131072  Batches: 1  Memory Usage: 4449kB
                             ->  Seq Scan on date_dim date_dim_1  (cost=0.00..2705.49 rows=73049 width=16) (actual time=0.023..15.640 rows=73049 loops=1)
   ->  Sort  (cost=756250.03..756250.05 rows=5 width=132) (actual time=9167300.007..9167300.013 rows=100 loops=1)
         Sort Key: t_s_secyear.customer_id, t_s_secyear.customer_first_name, t_s_secyear.customer_last_name, t_s_secyear.customer_email_address
         Sort Method: top-N heapsort  Memory: 51kB
         ->  Nested Loop  (cost=0.00..756249.97 rows=5 width=132) (actual time=34279.533..9167297.564 rows=382 loops=1)
               Join Filter: ((t_s_secyear.customer_id = t_w_secyear.customer_id) AND (CASE WHEN (t_w_firstyear.year_total > '0'::numeric) THEN (t_w_secyear.year_total / t_w_firstyear.year_total) ELSE 0.0 END > CASE WHEN (t_s_firstyear.year_total > '0'::numeric) THEN (t_s_secyear.year_total / t_s_firstyear.year_total) ELSE 0.0 END))
               Rows Removed by Join Filter: 116789448
               ->  Nested Loop  (cost=0.00..576151.10 rows=16 width=268) (actual time=21125.197..8715422.593 rows=5287 loops=1)
                     Join Filter: (t_s_firstyear.customer_id = t_s_secyear.customer_id)
                     Rows Removed by Join Filter: 757357814
                     ->  Nested Loop  (cost=0.00..396080.13 rows=18 width=104) (actual time=12131.633..7708400.851 rows=10841 loops=1)
                           Join Filter: (t_s_firstyear.customer_id = t_w_firstyear.customer_id)
                           Rows Removed by Join Filter: 1554943027
                           ->  CTE Scan on year_total t_s_firstyear  (cost=0.00..197999.86 rows=60 width=52) (actual time=5368.741..5627.443 rows=69732 loops=1)
                                 Filter: ((year_total > '0'::numeric) AND (sale_type = 's'::text) AND (dyear = 2001))
```

```
                                Rows Removed by Filter: 391630
                           ->  CTE Scan on year_total t_w_firstyear  (cost=0.00..197999.86 rows=60 width=52) (actual time=83.745..108.815 r
ows=22299 loops=69732)
                                Filter: ((year_total > '0'::numeric) AND (sale_type = 'w'::text) AND (dyear = 2001))
                                Rows Removed by Filter: 439063
                     ->  CTE Scan on year_total t_s_secyear  (cost=0.00..179999.88 rows=180 width=164) (actual time=0.002..87.811 rows=6986
1 loops=10841)
                          Filter: ((sale_type = 's'::text) AND (dyear = 2002))
                          Rows Removed by Filter: 391501
               ->  CTE Scan on year_total t_w_secyear  (cost=0.00..179999.88 rows=180 width=52) (actual time=65.129..83.847 rows=22090 loop
s=5287)
                    Filter: ((sale_type = 'w'::text) AND (dyear = 2002))
                    Rows Removed by Filter: 439272
 Planning time: 2.155 ms
 Execution time: 9167346.764 ms
(63 rows)
```

変化した部分は、パラレルクエリ実行を示すGatherの有無(上記バージョン10のEXPLAIN ANALYZE ★印の行)と、Sort Method(上記バージョン9.6のEXPLAIN ANALYZE ▲印 と 上記バージョン10のEXPLAIN ANALYZE ▲印)の違いになります。

Sort Methodがバージョン10では効率が良いインメモリ処理のQuick Sortであるのに対し、バージョン9.5では若干効率の劣るMerge Sortで、ディスクを作業領域に使用して処理されています。

一般的に、ソートの方式としてQuick Sortが選択されない原因は、実行計画作成時に予想される作業メモリ量がwork_memパラメータで指定した値を超えてしまうことにあります。パラレルクエリを用いない場合は、ソート対象となるデータ量が多くなると予想され、メモリの確保を断念したものと考えます。一方、パラレルクエリを用いる場合は、ソートもパラレルクエリ実行の対象に含まれるため、本ケースの場合は、1ワーカーが担当するソート対象のデータ量が3分の1となり、メモリ確保可能と判断したと考えます。

検証サーバ2はディスクがSSDであるため、ディスクが作業領域として使用されても、大きな性能劣化は発生しにくいと考えますが、通常のハードディスクが作業領域として使用される場合は、性能が大きく劣化することが予想され、パラレルクエリ実行によってQuick Sortを選択可能となっただけでも性能の改善が見込まれます。

データ量が処理方式を左右する操作(ソート、ジョイン、集約)の場合は、パラレルクエリ実行に伴って、より効率的な処理方式を選択する可能性があるものと考えます。

6.6.2. 検証B

6.6.2.1. No.10

No.10では、検証サーバ1、および、検証サーバ2ともに、並列処理数の増加にともなって、高速化する傾向にありました。

理論上、並列処理数とパラレルクエリで実行された部分の処理時間は反比例の関係にありますが、並列処理数と処理時間の間において、正確な反比例の関係が認められませんでした。特に、検証サーバ2では、グラフの形状からは反比例の関係を推測できる形状になっていませんでした。また、ワーカー数2とワーカー数8の間における、性能の変化はワーカー数2に対してワーカー数8が、検証サーバ1で12.3%改善、検証サーバ2で6.3%改善と小さく、No.10における処理時間の大半はパラレルクエリ実行できない部分が 占めていることが示唆されます。

ここで、No.10において実行計画を確認すると、パラレルクエリ実行を有効にした結果、パラレルクエリ実行の対象となっていないジョイン操作(考察 検証A No.10 バージョン9.6のEXPLAIN ANALYZE 9行目のNested Loop)が 全体5.86秒に対して4.11秒強と73%を占めていました。また、他のワーカー数(4, 6, 8)における実行計画を確認したところ、当該部分はそれぞれ4.33秒強、4.28秒弱、4.20秒弱となり、4.11秒から見て、2%～5%程度のばらつきが見られました。ここからは、No.10のクエリの実行時間において、パラレルクエリ実行に要した時間は支配的ではなかったということ、パラレルクエリ外に要した時間について無視できない大きさのばらつきが含まれていたことが示唆されます。

また、パラレルクエリ実行された部分のみの実行時間で比較すると、検証B補足 パラレルで動作した部分の比較(検証サーバ2) より、検証サーバ2では、グラフの形状が反比例の関係に近い高速化の傾向にあったことが確認できますが、反比例の曲線が本来存在するべきであろう位置から、x軸からy軸の+の方向に0.3秒～0.4秒弱移動しているような グラフとなっています。ここからは、パラレルクエリ実行において、0.3秒～0.4秒弱の固定的な時間を追加で要していることが示唆されます。

以上から、ワーカー数による性能差は、パラレルクエリ実行による改善分とパラレルクエリ実行に伴うオーバヘッド分、パラレルクエリ外の要因の合成となり、オーバヘッド分よりも、パラレルクエリ外に要した時間が無視できない大きさでばらついたことから、グラフが複雑な形状となった可能性があると考えます。

6.6.2.2. No.11

No.11では、検証サーバ1では並列処理数の増加に伴って、高速化する傾向を確認できましたが、ワーカー数2とワーカー数4では0.03%程度の改善とほとんど効果が見られませんでした。これは、ワーカー数2とワーカー数4の間でのパラレルクエリ実行に関するオーバヘッドが増加したが、その増加分をパラレルクエリ実行で埋められなかったため、と考えます。

また、ワーカー数6とワーカー数8との比較では、ワーカー数8が0.38%の低速化が認められました。こちらに関しては、実行計画を確認したところ、No.10と同様にパラレルクエリ実行における並列処理数は、今回の条件では5が最大であったので、ワーカー数6とワーカー数8では実行計画が同一となり、処理時間が0.38%微増と、ほとんど変化していないことから、概ね妥当と考えます。

検証サーバ2においては、No.11では逆に低速化する傾向が認められました。ただし、検証B補足 パラレルで動作した部分の比較(検証サーバ2) に示したとおり、パラレルクエリで実行された部分に関しては、理論通りの高速化傾向を示している点、クエリ実行全体の処理時間9000秒強に対して、パラレルクエリ実行部分の処理時間が

5秒弱～23秒弱程度(0.06%～0.25%)と比率が極めて小さい点から、低速化した原因は、パラレルクエリ実行におけるオーバヘッド、もしくはパラレルクエリ実行の部分以外での要因がもととなっているものと考えます。

以上の結果から、パラレルクエリ実行に伴うオーバヘッドの大きさが、パラレルクエリ実行による高速化分を上回る場合がありうると考えます。また、パラレルクエリでの実行が選択されても、処理時間全体に占めるパラレルクエリ実行部分の比率からは、大きな改善が見込めない場合も存在することが示されたと考えます。

6.6.3. 総括

PostgreSQL 9.6においてパラレルクエリが実装されたことによって、大規模エンタープライズ用途向けの機能の一つがPostgreSQLに新たに加わり、PostgreSQLは新たな時代に突入したと感じていましたが、実際に検証を行った結果、パラレルクエリの使用には細心の注意が必要であることが明らかになりました。

特に、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しないかの確認、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し、効果が十分に得られるかどうかの確認が必要であるものと考えます。

以上から、現状のパラレルクエリの実装においては、OLAP用途での使用には不安が残る結果となりました。

より不安なくパラレルクエリを使用できるよう、次期バージョンでのパラレルクエリのさらなる進化と、これを司るプランナ・エグゼキュータのさらなる深化に期待したいです。

7. 全文検索

7.1. 検証概要

近年の全文検索のユースケースでは、静的なデータを検索するだけではなく、データが継続して変更されている なかで検索を行う事が多くなっています。例えば...

- ロコミサイト(ロコミは継続的に更新される、ロコミが更新されている最中もロコミは検索される)
- 動画共有サイト(動画は継続的に投稿され更新されている、動画が更新されている最中も動画は検索される)

上記のようなケースでは、同一のDBに対して更新と検索が同時に発生するため、十分なパフォーマンスが 得られないと予想されます。

そこで、PostgreSQL10から実装されたロジカルレプリケーションを使用した負荷分散を行い、上記のようなケースで十分なパフォーマンスが得られるかを検証します。

ロジカルレプリケーションを用いた負荷分散は以下のような構成で行います。

- Masterをデータの更新専用とし、全文検索用のインデックスを設定しません。
- Slaveをデータ検索専用とし、全文検索用のインデックスを設定します。

全文検索は、PostgreSQLに以下の全文検索用のモジュールを使います。

- pg_bigm(ピージーバイグラム)
- PGroonga(ピージールンガ)

7.2. 検証構成

7.2.1. マシン構成

本検証に用いたハードウェア構成を 表. 7.1 、ソフトウェア構成を 表. 7.2 に示します。

表 7.1 ハードウェア構成

項目	仕様
機種	Amazon Web Service(AWS) m4.2xlarge
CPU	Intel(R) Xeon(R) CPU E5-2686 v4 @ 2.30GHz (8コア)
メモリ	32GB
内蔵ストレージ	SSD 256GB

表 7.2 ソフトウェア構成

ソフトウェア	バージョン	用途	備考
CentOS	7.4(64bit)	OS	
PostgreSQL	10.1	データベース	
pg_bigm	1.2	インデックス	
PGroonga	2.0.3	インデックス	
Groonga	7.1.0	インデックス	PGroongaの動作に必要

7.2.2. 全文検索用モジュール

本検証で利用した全文検索用モジュールを 表. 7.3 に示します。

表 7.3 検証対象の全文検索用モジュール

名称	説明	ライセンス	解析方法
pg_bigm	日本語に対応した全文検索機能を提供するモジュール	PostgreSQL License	2-gram
PGroonga	日本語に対応した全文検索機能として実績の豊富な GroongaをPostgreSQLから利用可能にするモジュール	PostgreSQL License	N-gram

7.2.3. postgresql.confの設定値

postgresql.confの設定値を 表. 7.4 のように変更しています。今回は大量のデータに一度にインデックスを作成するため、maintenance_work_memを多めに確保し

ています。

表 7.4 postgresql.confの設定値

パラメータ	設定値	考え方
max_connections	100	
shared_bufferes	8GB	搭載メモリの1/4を割り当て
max_wal_size	1GB	
maintenance_work_mem	4GB	インデックス作成の際に必要となる領域のため、多めに確保

7.3. 検証方法

7.3.1. 検証構成

検証構成を 表. 7.5 に示します。

表 7.5 検証構成と測定項目

検証構成	測定項目
レプリケーション構成なし	データ更新性能 * インデックスなしの時のデータ更新性能 * pg_bigmのインデックスありの時のデータ更新性能 * PGroongaのインデックスありの時のデータ更新性能 全文検索性能 * インデックスなしの時の全文検索性能 * pg_bigmのインデックスありの時の全文検索性能 * PGroongaのインデックスありの時の全文検索性能
ロジカルレプリケーション構成	データ更新性能 * インデックスなしの時のデータ更新性能 * pg_bigmのインデックスありの時のデータ更新性能 * PGroongaのインデックスありの時のデータ更新性能 全文検索性能 * インデックスなしの時の全文検索性能 * pg_bigmのインデックスありの時の全文検索性能 * PGroongaのインデックスありの時の全文検索性能

レプリケーション構成なしと、ロジカルレプリケーション構成時のデータ更新性能、全文検索性能を比較します。
ロジカルレプリケーション構成時は、Master(データ更新用)のDBには全文検索用のインデックスを設定しないため、インデックスを設定するレプリケーション構成なしの時のデータ更新性能と比べることで、どの程度更新性能が向上したかを示すことができます。

7.3.2. 全文検索対象のデータ

本検証の検索対象は、国立情報学研究所のダウンロードサービスにより株式会社ドワンゴから提供を受けた「 ニコニコ動画コメント等データ 」内の動画のメタデータをPostgreSQLへjsonb型として格納したものです。 表. 7.6 にテーブル構造とデータの例を示します。

表 7.6 テーブル名「nikoniko」

tag（Text型）	record（jsonb型）	備考
1	{"tags": ["料理", "カレー"], "title": "テスト", "length": 578, "category": "料理", "size_low": 29606343, "video_id": "smXXXXXXX", "file_type": "flv", "size_high": 31095758, "watch_num": XXXX, "mylist_num": 22, "comment_num": 290, "description": "テスト", "upload_time": XXXXXXXXXX}	表中の *X* には、任意の数値が入ります。

7.3.3. インデックスの作成方法

作成したテーブルの「record」カラムに対し、CREATE INDEXコマンドでインデックスを作成します。インデックスは、良く検索されるであろう、動画のタイトル(title)と動画の詳細(description)に対してインデックスを設定します。全文検索のインデックスは以下の5パターンの設定を行います。

- インデックスなし

- pg_bigmのインデックス2種類

```
CREATE EXTENSION pg_bigm;
CREATE INDEX pgbigm_title_index ON nikoniko USING gin((record->>'title') gin_bigm_ops);

CREATE EXTENSION pg_bigm;
CREATE INDEX pgbigm_description_index ON nikoniko USING gin((record->>'description') gin_bigm_ops);
```

- PGroongaのインデックス2種類

```
CREATE EXTENSION pgroonga;
CREATE INDEX pgroonga_title_index ON nikoniko USING pgroonga((record->>'title'));

CREATE EXTENSION pgroonga;
CREATE INDEX pgroonga_description_index ON nikoniko USING pgroonga((record->>'description'));
```

7.3.4. 背景負荷

背景負荷は、断続的にINSERTを行うことで実現します。背景負荷のスループットは以下の通りです。

- 背景負荷(約1200[record/s])

INSERTするデータは以下のようなJSONデータです。

```
{
  "tags": ["xxx", "xxxx", "xxxx"],
  "title": "xxxx",
  "length": 208,
  "category": "ゲーム",
  "size_low": 9274707,
  "video_id": "smXXXXX",
  "file_type": "flv",
  "size_high": 10298881,
  "watch_num": 201902,
  "mylist_num": 3112,
  "comment_num": 3788,
  "description": "xxxxxxxxx",
  "upload_time": XXXXXXXXXX
}
```

7.3.5. 検索方法

動画検索で検索されやすい、動画のタイトル、動画の詳細を対象に検索します。検索キーワードは、ヒット数が0にならない一般的な用語とします。

検索に使うクエリーは以下の通りです。

- PGroongaのインデックス使用時
- 動画タイトルでの検索

```
SELECT COUNT(*) FROM fluentd WHERE (record ->> 'title') &@ 'カレー';
```

- 動画詳細での検索

```
SELECT COUNT(*) FROM fluentd WHERE (record ->> 'description') &@ 'カレー';
```

- pg_bigmのインデックス使用時、インデックスなしの時
- 動画タイトルでの検索

```
SELECT COUNT(*) FROM fluentd WHERE (record->>'title') LIKE '%カレー%';
```

- 動画詳細での検索

```
SELECT COUNT(*) FROM fluentd WHERE (record->>'description') LIKE '%カレー%';
```

7.4. 検証結果

7.4.1. レプリケーション構成なし

7.4.1.1. データ更新性能

表. 7.7 にデータ更新性能を記載します。データ更新性能は、1秒間あたりにINSERTしたレコード数を測定しており、表. 7.7 に記載されている数値が大きいほど 高速にデータを更新していることを表しています。

以下の通り、インデックスなしのときが最もデータ更新が速く、PGroongaとpg_bigmはインデックスを設定する キーによって、更新性能が異なります。

descriptionキーのように、長いテキストが多いカラムにインデックスを設定した場合:
更新性能（良） インデックスなし ＞ PGronnga ＞ pg_bigm 更新性能（悪）

titleキーのように、短いテキストが多いカラムにインデックスを設定した場合:
更新性能（良） インデックスなし ＞ pg_bigm ＞ PGronnga 更新性能（悪）

表 7.7 データ更新性能

	インデックスなし	pg_bigmのインデックスあり	PGroongaのインデックスあり
データ更新性能[insert/s](descriptionキーにインデックスを設定)	1305[insert/s]	944[insert/s]	998[insert/s]
データ更新性能[insert/s](titleキーにインデックスを設定)	1305[insert/s]	1202[insert/s]	1152[insert/s]

7.4.1.2. 全文検索性能

表. 7.8、表. 7.9 に 全文検索性能を記載します。全文検索性能は、平均レイテンシを測定しており、表. 7.8、表. 7.9 に記載されている数値が小さいほど、高速に検索出来ることを表します。検索に1件もヒットしない状態にならないように、事前に203317レコードを登録済みの状態で計測します。背景負荷がある場合は、INSERTとSELECTを別々のインスタンスから実行し、全文検索性能を測定します。

表 7.8 全文検索性能 動画タイトルでの検索

背景負荷	インデックスなし	pg_bigmインデックスあり	PGroongaインデックスあり
なし	62.48[ms]	0.46[ms]	0.88[ms]
あり	121.44[ms]	1.63[ms]	0.95[ms]

表 7.9 全文検索性能 動画の詳細での検索

背景負荷	インデックスなし	pg_bigmインデックスあり	PGroongaインデックスあり
なし	138.66[ms]	1.62[ms]	1.12[ms]
あり	330.07[ms]	2.09[ms]	1.30[ms]

7.4.2. ロジカルレプリケーション構成

7.4.2.1. データ更新性能

表. 7.10 にデータ更新性能を記載します。データ更新性能は、1秒間あたりにINSERTしたレコード数を測定しており、表. 7.10 に記載されている数値が大きいほど 高速にデータを更新していることを表しています。

ロジカルレプリケーション構成の場合は、データ更新専用にデータベースを用意するので、全文検索用のインデックスを設定しません。そのため、結果はインデックスなしの時のデータ更新性能のみとなります。

表 7.10 データ更新性能

	インデックスなし
データ更新性能[insert/s]	1292[insert/s]

7.4.2.2. 全文検索性能

表. 7.11、表. 7.12 に 全文検索性能を記載します。全文検索性能は、平均レイテンシを測定しており、表. 7.11、表. 7.12 に記載されている数値が小さいほど、高速

に検索出来ることを表します。

検索に1件もヒットしない状態にならないように、事前に203317レコードを登録済みの状態で計測します。背景負荷がある場合は、INSERTとSELECTを別々のインスタンスから実行し、全文検索性能を測定します。

表 7.11 全文検索性能 動画タイトルでの検索

背景負荷	インデックスなし	pg_bigmインデックスあり	PGroongaインデックスあり
なし	60.26[ms]	1.93[ms]	0.88[ms]
あり	118.94[ms]	2.15[ms]	0.92[ms]

表 7.12 全文検索性能 動画の詳細での検索

背景負荷	インデックスなし	pg_bigmインデックスあり	PGroongaインデックスあり
なし	138.44[ms]	1.92[ms]	1.11[ms]
あり	262.53[ms]	2.11[ms]	1.16[ms]

7.4.3. 結果比較

7.4.3.1. データ更新性能

図. 7.1 に、レプリケーション構成なしとロジカルレプリケーション 構成のデータ更新性能の比較結果を示します。以下のグラフは左から、レプリケーション構成なしでインデックスなし、pg_bigmのインデックスあり、PGroongaのインデックスあり、ロジカルレプリケーション構成のMasterのデータ更新性能を示しています。

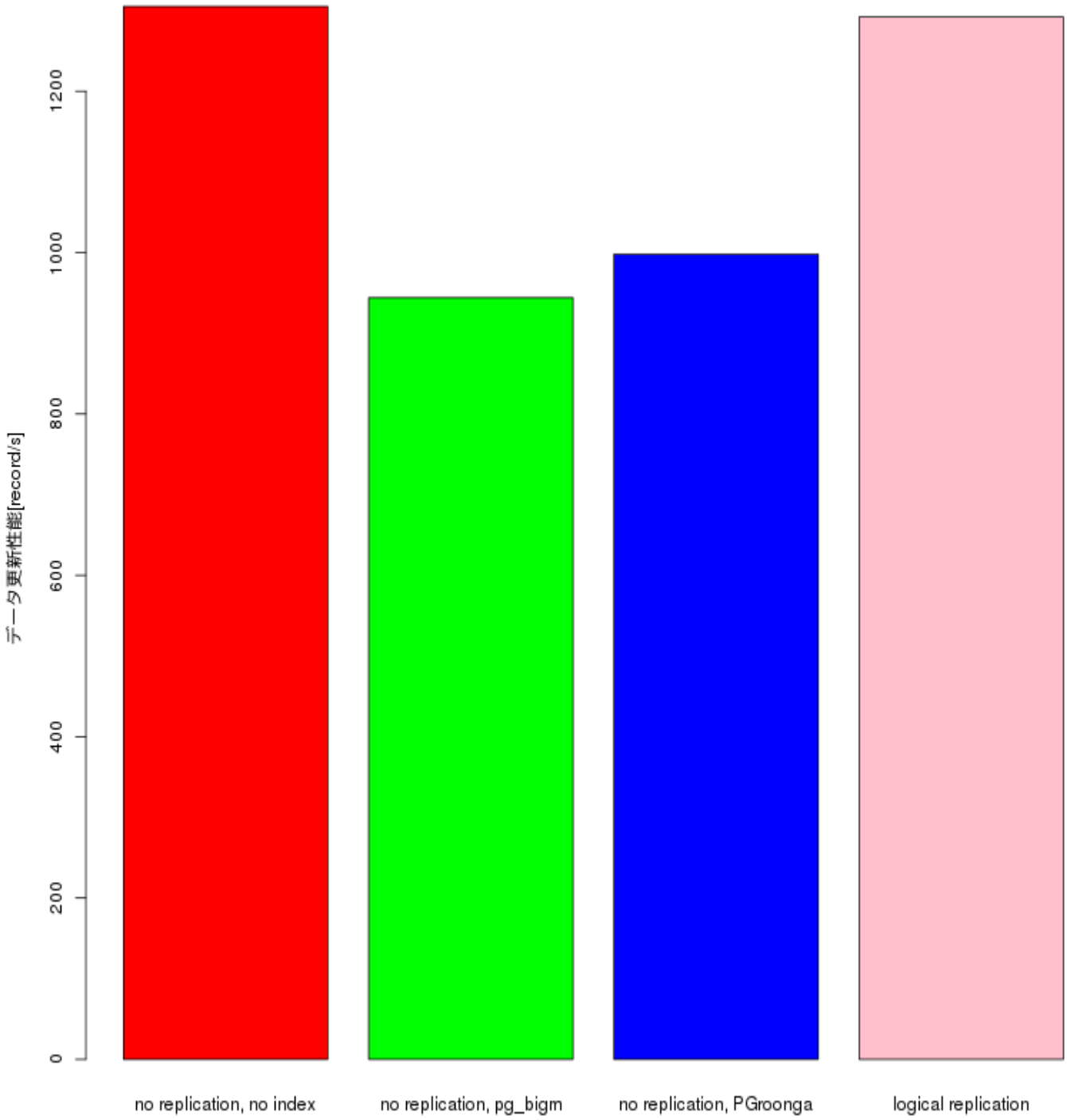


図 7.1 データ更新性能比較

上記グラフから、レプリケーション構成なしでインデックスなしの時のデータ更新性能と ロジカルレプリケーション構成のデータ更新性能は、ほぼ同じ性能であることがわかります。これは、ロジカルレプリケーション構成のMasterデータベースはデータ更新専用に運用されるため、全文検索用のインデックスを設定していないためです。

レプリケーション構成のない、単一のデータベースで全文検索のインデックスを設定する場合は、上記グラフの左から2,3番目の値が示すとおり、インデックスなしよりデータ更新性能が落ちますが、ロジカルレプリケーション構成の場合は、Slaveに全文検索のインデックスを設定するため、データ更新性能を落とさず、全文検索の高速化が可能です。

7.4.3.2. 全文検索性能

図. 7.2 に、レプリケーション構成なしの時の全文検索性能と ロジカルレプリケーション構成時の全文検索性能の比較結果を示します。背景負荷がある状態での全文検索性能を比較しています。

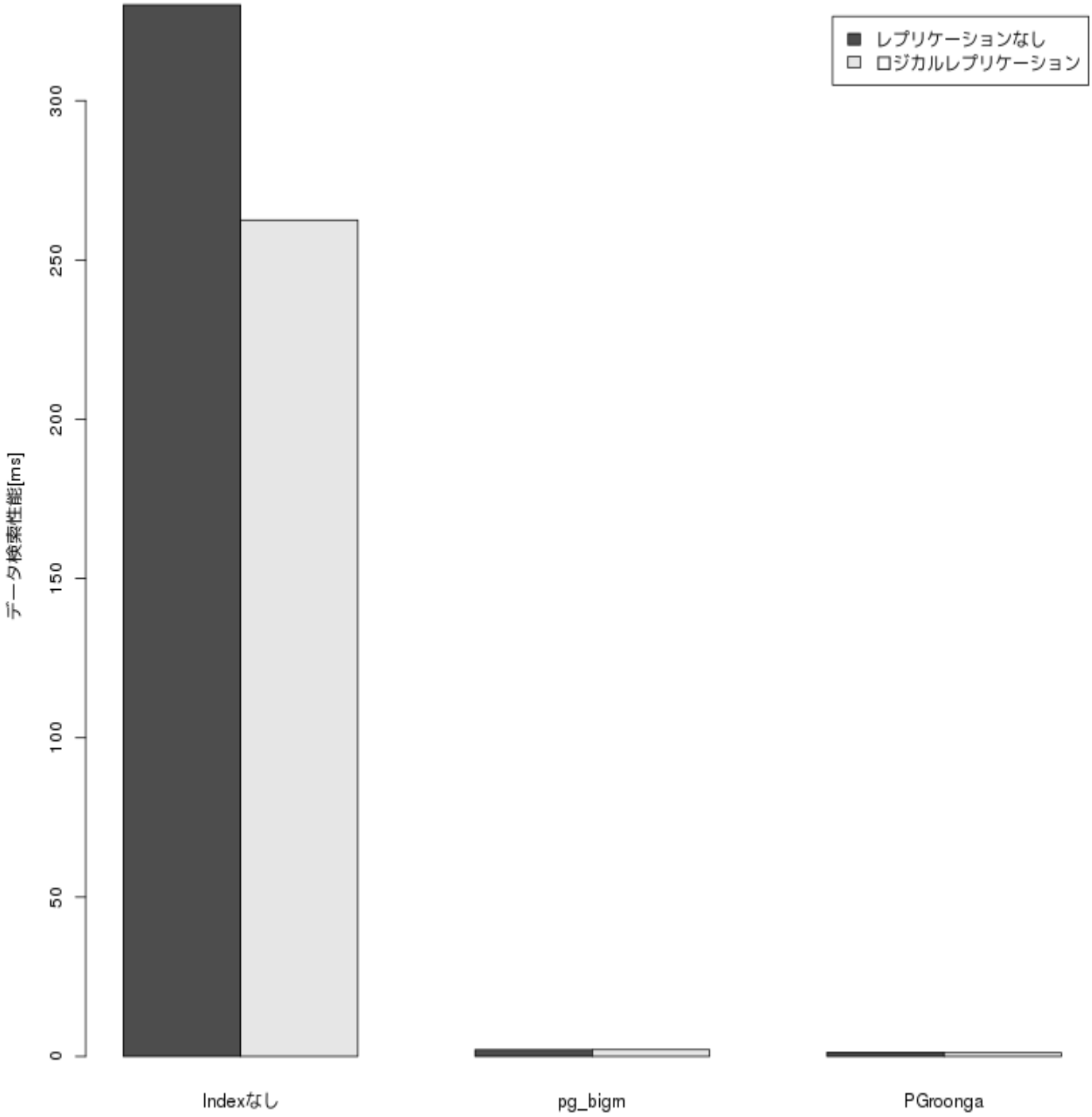


図 7.2 全文検索性能

インデックスなしの場合は、ロジカルレプリケーション構成の方がデータ更新の影響を受けないため、全文検索性能が向上しています。pg_bigmとPGroongaのインデックスが設定されている場合は、レプリケーション構成なしの時の全文検索性能とあまり変化はありません。

上記2つの比較結果から、ロジカルレプリケーション構成にすることで、データの更新性能を落とさずに、全文検索性能を向上させることができます。

7.5. 考察

全文検索用のインデックスを設定すると、全文検索性能が大幅に向上しますが、単一のデータベースで 更新も全文検索も実施する場合は、全文検索用のインデックスが設定されているとデータ更新性能 が落ちてしまいます。

今回の検証では、ロジカルレプリケーションを用いて、データ更新を行うデータベースには全文検索用の インデックスを設定せず、検索用のデータベースを用意し、検索用の

データベースにのみインデックスを 設定する構成でのパフォーマンスを検証しました。また、ロジカルレプリケーションを非同期で実施することによって、データ更新用のデータベースへの 性能影響を抑えています。ロジカルレプリケーションが非同期で実施されることにより、最新の更新がSlaveへ反映されませんが、 今回のモデルのような検索システムでは、最新状態であることは必須ではないため、現実の運用としては、 問題ないと判断しました。

データ更新専用のデータベースを用意することで、データ更新用のデータベースには全文検索用のインデックス を設定しなくても良くなります。検証の結果から、このような負荷分散を行うことで、全文検索用のインデックス が設定されていることによるデータ更新の劣化を防ぐことができる事がわかりました。

また、ロジカルレプリケーション構成時の全文検索性能の結果から、この検索専用のデータベースに全文検索用の インデックスを設定した場合の検索性能とレプリケーション構成なしでの全文検索性能がほぼ変わらないことから 今回の検証のような負荷分散を行った構成でも、通常の構成と同様の全文検索性能が発揮できることがわかりました。

上記から、MasterとSlaveに別々のインデックスを設定できるロジカルレプリケーションを利用した負荷分散 構成をつくることで、全文検索用のインデックスによるデータ更新性能の劣化を起こさずに、全文検索用の インデックスを利用した高速な全文検索が可能になります。

8. レプリケーション検証

8.1. 検証概要

PostgreSQLでは、9.0からレプリケーション機能を実装しており、継続的に機能強化が行われています。9.6からは、2台以上の複製先に対してデータを同期しながら更新が行える、複数同期レプリケーションが利用可能となりました。これにより、負荷分散構成でのデータの信頼性向上などが期待されます。

しかしながら、従来まで可能であった非同期での複数台レプリケーションに比べ、同期処理にかかるオーバヘッドによってスループットが低下することが予想されます。また、同期の台数を増加させることによる性能的なオーバヘッドや、同期・非同期を混在させた環境での動作については懸念も存在します。

本章では、PostgreSQLにおけるレプリケーション機能について、同期レプリケーション・非同期レプリケーション、またそれらが混在する構成において、どのように性能が変化するのかを検証しました。

8.1.1. レプリケーションとは

データベースにおいて、レプリケーションとは複数のデータベースサーバの間で、データの一貫性を保ちながら、その内容を複製する手法を指します。複製を作成することで、障害発生時のバックアップや、負荷分散等の冗長化構成を取ることが可能となることから、データベースの可用性や性能向上を目的として利用される機能です。

PostgreSQLにおいては、Master側のデータベースに対する操作が記録されたWAL(Write Ahead Logging)ファイルを Slave側に転送し、反映することでレプリケーションが実現されます。転送時の処理方法を設定することにより、同期方式を選択することが可能となっています。

また、転送されたWALファイルのSlave側の処理の違いにより、PostgreSQLにおけるレプリケーションには「同期レプリケーション」と「非同期レプリケーション」の2種類の形態が存在します。

8.1.1.1. 同期方式

PostgreSQLでは、同期中のWALファイルの処理を設定することにより、同期のレベルを設定することが出来ます。同期のレベルを高くすると、Slave側への同期が保証されるデータが増えますが、その分の処理によるオーバヘッドがかかるため、パフォーマンスが低下することになります。

同期方式は、postgresql.confのパラメータ'synchronous_commit'により設定可能であり、以下の5つから選択することが可能です。本検証では、デフォルトの設定である'on'に設定し、測定を行っています。

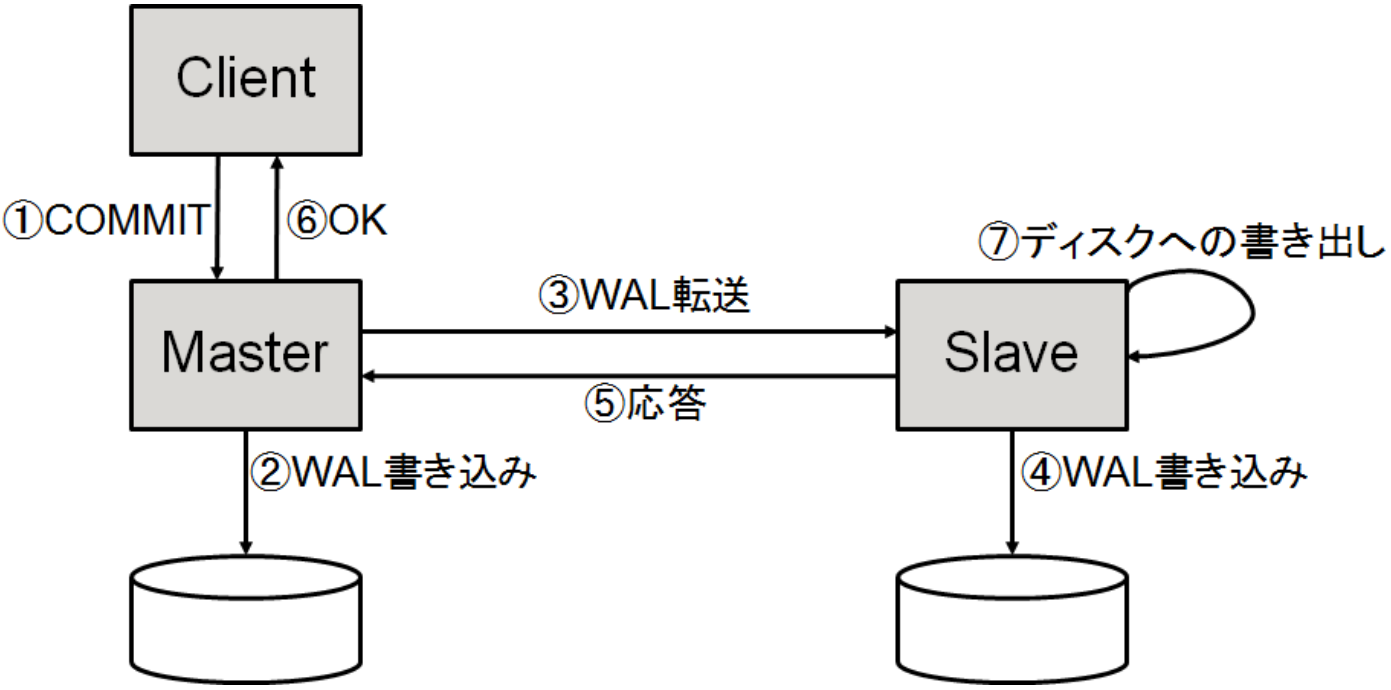
表 8.1 同期方式の比較

設定	内容
remote_apply	完全同期。WAL適用まで保証。SlaveからMasterへの昇格時の時間短縮と参照負荷分散の拡充が目的。
on	同期。WAL転送（ディスク書き込み）まで保証。データ保護が目的。
remote_write	準同期。WAL転送（メモリ書き込み）まで保証。データ保護とパフォーマンスのバランスが目的。
local	非同期。ローカルのWAL書き込みまで保証。パフォーマンス優先が目的。
off	完全非同期。ローカルのWAL書き込みも保証しない。最も高パフォーマンスだが、非現実的。

8.1.1.2. 同期レプリケーション

同期レプリケーションとは、MasterからSlaveへWALファイルを転送した際に、MasterがSlaveへの反映完了を待つレプリケーション方式です。MasterはWALファイルを転送後、Slaveの応答を待ってから次の処理に移ります。

処理の流れは以下のようになり、①から順に処理が実行されます。Clientが⑥OKを受け取るためには、Slaveでの④WAL書き込みの処理が完了し、⑤応答を受け取る必要があります。

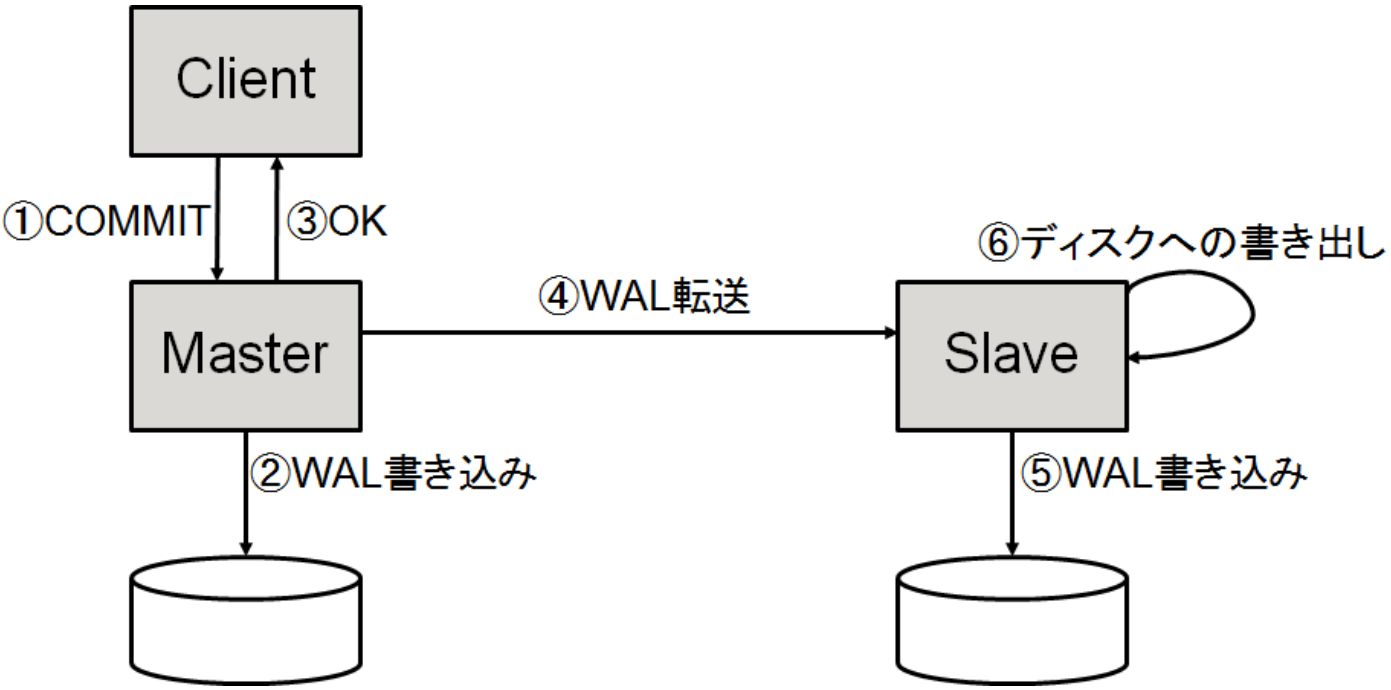


このため、Masterでコミットが成功した場合、Slaveにもデータが反映されていることが保証されます。ただし、コミットの完了にはSlaveからの応答を待つ必要があるため、単位時間当たりの処理性能は低くなります。

8.1.1.3. 非同期レプリケーション

非同期レプリケーションは、同期レプリケーションと異なり、MasterがSlaveの反映完了を待たないレプリケーション方式です。MasterはWALファイルをSlaveへ転送した後、すぐに次の処理に移ります。

処理の流れは以下のようになり、①から順に処理が実行されます。同期の場合と異なり、Slaveでの⑤WAL書き込みの処理の前に、Clientには③OKが返されています。



このため、MasterとSlaveのデータは必ずしも一致するとは言えません。一方で、コミットの完了にはSlaveからの応答を待つ必要がないため、単位時間の処理性能は高くなります。

表. 8.2 に、それぞれの方式の特長をまとめます。

表 8.2 レプリケーション方式の比較

	同期レプリケーション	非同期レプリケーション
Masterのコミット時	Slaveの反映完了を待つ	Slaveの反映完了を待たない
コミットされているデータの内容	MasterとSlaveで一致する	MasterとSlaveで一致するとは限らない
処理性能	比較的低い	比較的高い

8.2. 検証構成

8.2.1. 検証サーバ

本検証を実施するにあたり、各検証サーバには仮想マシン(KVM)を用いました。負荷かけ用のClientマシン1台、レプリケーション元となるMasterマシン1台、レプリケーション先となるSlaveマシン3台、計5台の構成です。

表. 8.3 にホストマシン、表. 8.4 にゲストマシンの構成を示します。

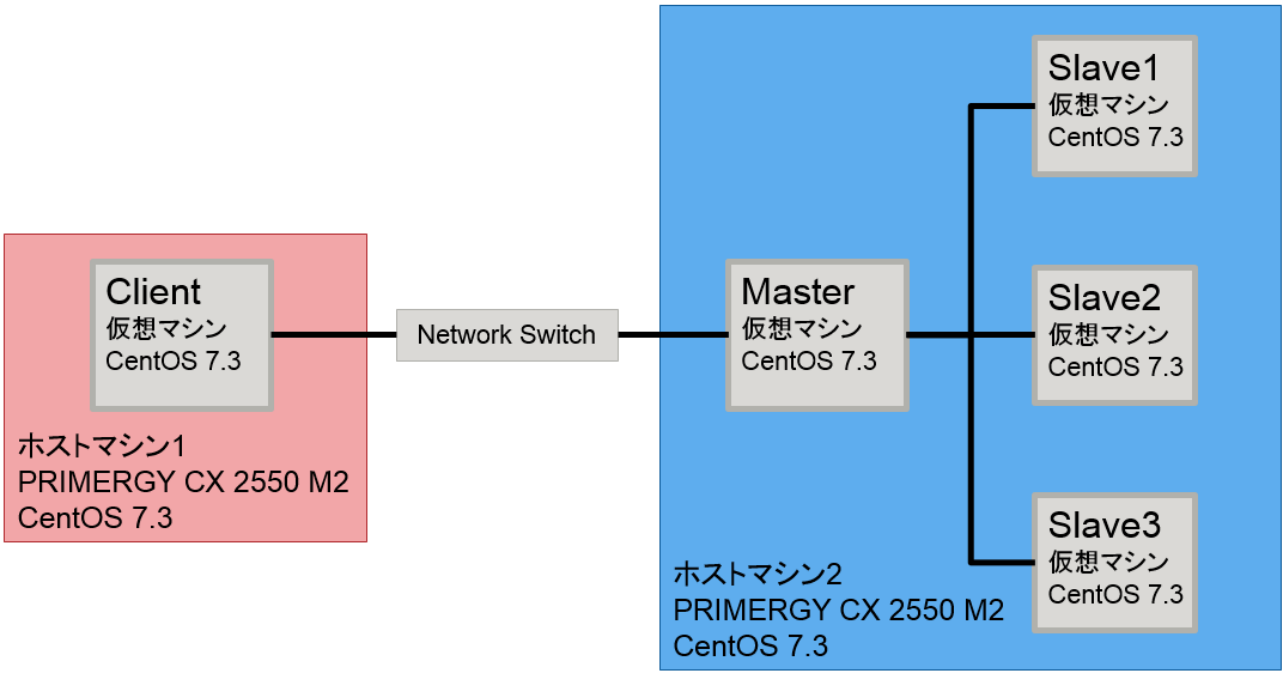
表 8.3 ホストマシン構成

ホストマシン×2	
機種	PRIMERGY CX 2550 M2
CPU	インテル Xeon プロセッサ E5-2650 v4 48コア 2.2GHz
メモリ	256GB
ストレージ	内蔵 2.5 インチSSD−400GB ×24
OS	CentOS 7.3

表 8.4 ゲストマシン構成

	Client	Master	Slave*3台
OS	CentOS 7.3	CentOS 7.3	CentOS 7.3
vCPU	4	4	4
メモリ	48GB	48GB	48GB
ディスク	200GB	200GB	200GB
DBMS	PostgreSQL 10.1	PostgreSQL 10.1	PostgreSQL 10.1

また、それぞれの仮想マシンは以下の様にネットワークを設定しました。



8.2.2. データベース設定

レプリケーション検証を行うための、PostgreSQLの各種設定を以下に示します。

8.2.2.1. postgresql.confの設定

Masterのpostgresql.confの設定を 表. 8.5 に、Slaveのpostgresql.confの設定を 表. 8.6 に示します。これらはデフォルトから変更した値のみとなります。

表 8.5 Masterのpostgresql.conf設定

パラメータ	設定値
shared_buffres	16GB
wal_level	replica
synchronous_commit	on
max_wal_size	8GB
max_wal_senders	4
wal_keep_segment	16
#synchronous_standby_names	"X(< 同期とするSlaveサーバのapplication_name>)" #X= 同期とするサーバ数
autovacuum	off

表 8.6 Slaveのpostgresql.confの設定

パラメータ	設定値
shared_buffres	16GB
wal_level	replica
synchronous_commit	on
max_wal_size	8GB
max_wal_senders	4
wal_keep_segment	16
autovacuum	off
hot_standby	on

8.2.2.2. pg_hba.confの設定

Master側での認証のために、pg_hba.confの設定を以下の様にします。

```
host    all             all             (負荷掛けマシンのIP)/24          trust
host    replication     repl_user       (Slaveマシン1のIP)/24            trust
host    replication     repl_user       (Slaveマシン2のIP)/24            trust
host    replication     repl_user       (Slaveマシン3のIP)/24            trust
```

8.3. 検証方法

8.3.1. 検証パターン

本検証では、以下の2パターンで性能を比較します。

8.3.2. 検証A(Slaveの台数を増加させた場合)

検証Aでは、Slave台数を1台～3台に変更し、Slave台数の増加による 性能影響を同期レプリケーション、非同期レプリケーションそれぞれで明らかにします。

8.3.3. 検証B(同期・非同期混在の場合)

検証Bでは、同期・非同期が混在する構成での検証を行います。Slaveの合計台数が3台となるように、以下の4パターンで性能を比較します。

表 8.7 検証Bのパターン

パターン	同期	非同期
非同期のみ	0台	3台
同期・非同期混在①	1台	2台
同期・非同期混在②	2台	1台
同期のみ	3台	0台

8.3.4. 環境

8.3.4.1. データベースの初期設定

initdbコマンドにてデータベースクラスタを作成します。

```
initdb --no-locale -E UTF8
```

先述した通りに、pg_hba.confを編集します。

```
vi pg_hba.conf
```

postgresql.confを先に述べた設定に編集し、データベースを起動します。

```
vi postgresql.conf
pg_ctl -w start
```

8.3.4.2. Master側の設定

検証用のデータベースを作成します。pgbenchの初期化オプションを用いて、スケールファクタ=1000のおおよそ15GBのデータベースを作成します。

```
createdb pgbench
pgbench -i -s 1000 pgbench
```

レプリケーション用のユーザを作成します。

```
psql -c "CREATE ROLE repl_user LOGIN REPLICATION PASSWORD 'password'"
CREATE ROLE
```

8.3.4.3. 同期・非同期の設定

- 同期・非同期で共通の設定

  Slaveにて、Masterのデータベースのバックアップ(複製)を取ります。

```
pg_basebackup -h <MasterサーバのIPアドレス> -D $PGDATA -R --progress -U repl_user
```

  postgresql.confを設定し、データベースサーバを起動します。

```
vi postgresql.conf
pg_ctl -w start
```

  Slaveのrecovery.confへapplication_nameの記述を追加します。下記ではSlave2を例としています。

```
vi recovery.conf
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# primary_conninfoの末尾にapplication_name=slave2を追加
```

- 同期の場合の設定

同期とする場合には、Masterのpostgres.conf内、synchronous_standby_namesを下記のように、適した値に設定する必要があります。下記では一例として、同期台数2台(slave1とslave2)としています。

```
vi postgresql.conf
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
synchronous_standby_names='2(slave1,slave2)'
```

8.3.4.4. 各データベースサーバ再起動

同期・非同期の設定後、MasterとSlaveそれぞれのPostgreSQLを再起動します。

```
pg_ctl -w restart
```

8.3.5. 測定

性能測定には、PostgreSQLに標準で同梱されているpgbenchを使用しました。pgbenchでは、ベンチマーク用のスクリプトをユーザにてカスタマイズすることが可能です。本検証では、以下に示す簡単な更新処理を行うカスタムスクリプトで測定を行いました。

```
\set naccounts 100000 * :scale
\set aid random(1, :naccounts)
UPDATE pgbench_accounts
  SET filler = repeat(md5(clock_timestamp()::text),2)
  WHERE aid = :aid;
```

また、キャッシュによる測定値の影響を排除するため、以下の処理の順で測定を行ないました。

1. VACUUM処理の実施
2. CHECKPOINT処理の実施
3. 各マシンのキャッシュクリア
4. pg_prewarmによるデータのロード
5. pgbenchによるTPS値の測定

これらを1パターンにつき3回実施し、各パターンの平均値を結果としました。以下にその処理を行ったスクリプトを示します。

```
#!/bin/bash
mkdir /data/$1
for i in `seq 1 3`; do

  #VACUUM処理の実施
  vacuumdb -h <MasterのIP> -a -z

  #CHECKPOINT処理の実施
  psql -h <MasterのIP> pgbench -c "CHECKPOINT;"

  #各マシンのキャッシュクリア
  echo "password" | sudo -S ssh <MasterのIP> "sync"
  echo "password" | sudo -S ssh <MasterのIP> "echo 3 > /proc/sys/vm/drop_caches"
  echo "password" | sudo -S ssh <Slave1のIP> "sync"
  echo "password" | sudo -S ssh <Slave1のIP> "echo 3 > /proc/sys/vm/drop_caches"
  echo "password" | sudo -S ssh <Slave2のIP> "sync"
  echo "password" | sudo -S ssh <Slave2のIP> "echo 3 > /proc/sys/vm/drop_caches"
  echo "password" | sudo -S ssh <Slave3のIP> "sync"
  echo "password" | sudo -S ssh <Slave3のIP> "echo 3 > /proc/sys/vm/drop_caches"

  #pg_prewarmによるデータのロード
  psql -h <MasterのIP> pgbench -c "SELECT * FROM pg_prewarm('pgbench_accounts')"

  #pgbenchによるTPS値の測定
  pgbench -n -c 64 -T 300 -f script.sh -s 1000 -h <MasterのIP> pgbench >
/data/$1/bench_data_${i}.txt

done
```

8.4. 検証結果

8.4.1. 検証A(Slaveの台数を増加させた場合)

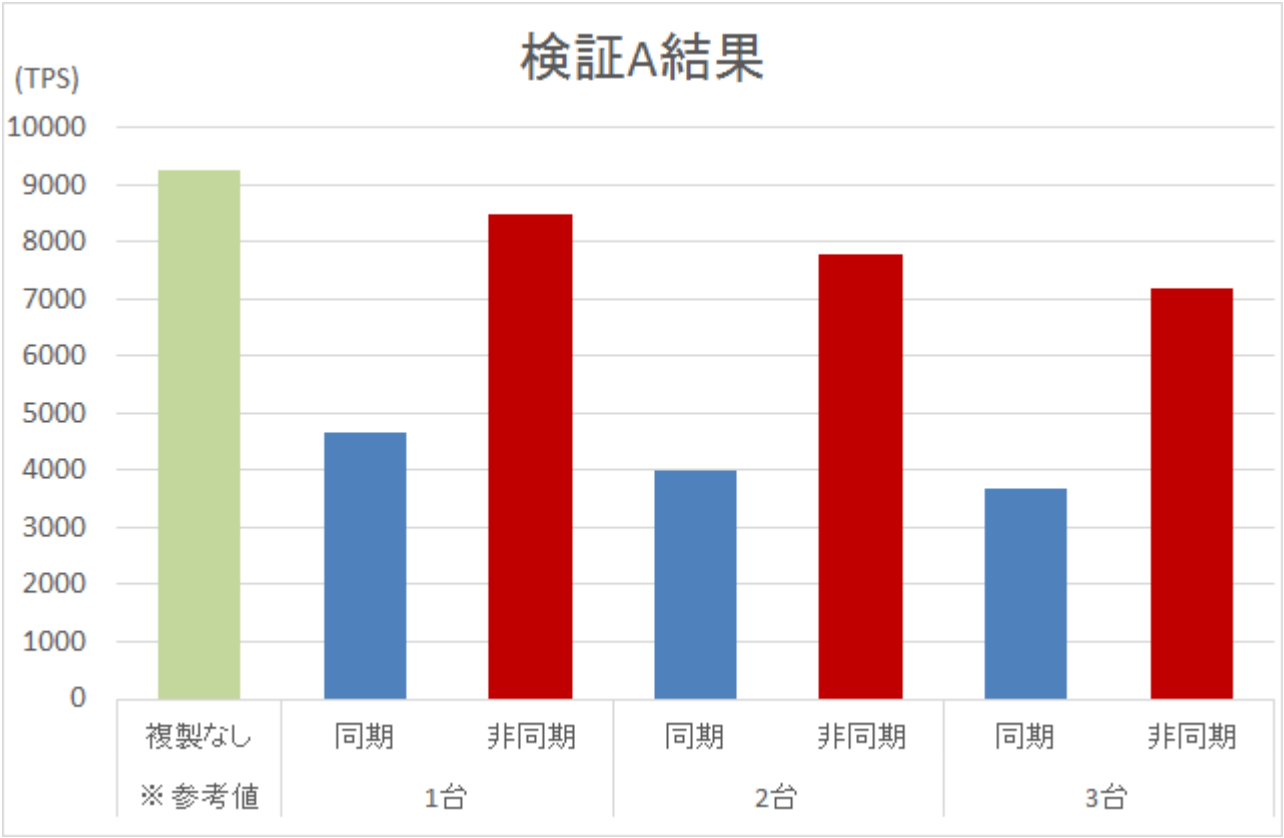
検証Aでの結果は下記となります。

表 8.8 同期レプリケーションの結果

同期台数	TPS値	レプリケーションなしのTPS値を100%としたときの割合
(参考)レプリケーションなし	9254.0	(100.0%)
1台	4668.6	50.4%
2台	4014.4	43.4%
3台	3686.3	39.8%

表 8.9 非同期レプリケーションの結果

非同期台数	TPS値	レプリケーションなしのTPS値を100%としたときの割合
(参考)レプリケーションなし	9254.0	(100.0%)
1台	8495.9	91.8%
2台	7786.9	84.1%
3台	7205.3	77.9%



8.4.2. 検証B(同期・非同期混在の場合)

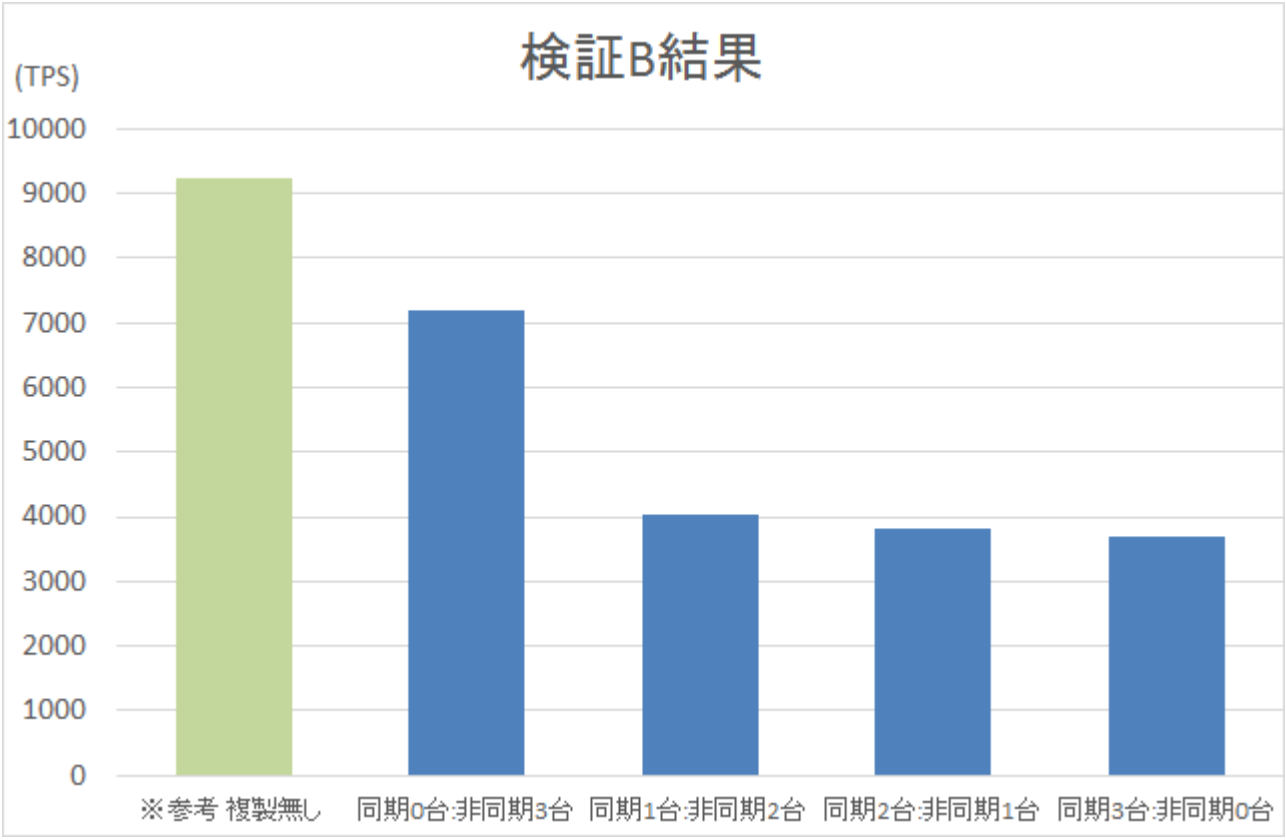
検証Bでの結果は以下となります。

表 8.10 検証B結果

パターン	同期	非同期	TPS値	レプリケーションなしのTPS値を100%としたときの割合
(参考)レプリケーションなし	0台	0台	9254.0	(100.0%)
非同期のみ	0台	3台	7205.3	77.9%
同期・非同期混在①	1台	2台	4043.6	43.7% [1]
同期・非同期混在②	2台	1台	3822.9	41.3% [2]
同期のみ	3台	0台	3686.3	39.8%

[1] 同期1台のみは50.4%、非同期2台のみは84.1%のため、単純に掛け合わせると42.4%となるが、上回っている。

[2] 同期2台のみは43.4%、非同期1台のみは91.8%のため、単純に掛け合わせると39.8%となるが、上回っている。



8.5. 考察

今回の構成では、MasterとSlaveは同じホストマシン上で稼働しており、ネットワークとストレージは物理的には同一のものを使用しています。そのため、Slaveのノード数が多くなるほど、物理的なリソースの競合により性能に影響が出てきたのではないかと考えられます。今回の検証では詳細な原因の特定には至りませんでしたが、2012年度にPGEConsで実施した「更新系・複数台レプリケーション検証」[3] でも 同様の傾向が見られており、非同期レプリケーションであっても必ずしもMasterの更新性能への影響が少ないわけではなく、スケールアウトによるボトルネックが出てきてしまうケースがあると言えます。

[3] PGECons 2012年度WG1活動報告 p.41「(4) ネットワークやストレージ構成について」

8.5.1. 検証Aについて

同期のみの構成の場合は、非同期のみの構成の場合に比べ約1/2の性能値となりました。同期の場合、一つの処理の完了にはMasterとSlave双方の処理の完了を待つ必要があります。一方で非同期の場合は、Masterだけの処理が完了すれば次の処理が開始されます。このことから、同期の場合は非同期に比べ、処理の完了を待つ対象が2倍になるため、TPS値が約1/2になったものと考えられます。

また、同期・非同期ともに、サーバ台数を増やすとTPS値が下がっていく傾向が見られました。特に、Slave側の処理を待たない非同期の場合でも、サーバ台数を増やすことでTPS値の低下が見られる結果となりました。TPS値の低下の度合いを見てみると、サーバ台数を増やすごとにTPS値が約10%ずつ減少していることが分かり、複製先が4台以上となっても同じように性能は一定の割合で逓減していくことが考えられます。

8.5.2. 検証Bについて

表. 8.10 で見たように、同じ台数の構成であっても、同期の台数を増やすほど性能値が低下していく傾向を見ることが出来ました。また、非同期を同期に1台変更するごとに約5%ずつTPS値が減少しており、検証Aと同様、台数を変化させる毎に性能は一定の割合で逓減していくことが考えられます。

非同期のみの構成と比較すると、同期を1台でも追加すると性能値が急落するということも確認できました。これは検証Aと同様に、同期ではSlave側の処理を待つ時間が増えるためと考えらえます。また、同期ではSlave側の処理の完了時にMaster側への応答が返ってくるため、ネットワークの使用量が増加します。そのため、検証Aの場合と同様に、ネットワークのリソース競合により性能が低下することが考えらえます。

8.5.3. まとめ

本章では、PostgreSQL 9.6より利用可能となった複数同期レプリケーションを利用した構成での性能測定を行い、非同期のみの構成、同期・非同期が混在する構成との比較検証を行いました。

検証結果から、今回の検証環境下では以下のような傾向を見ることができました。

1. 同期のみの構成は、非同期のみの構成に比べ約1/2の性能となる。
2. 同期・非同期混在の構成では、総台数が同じでも同期の台数を増やすことで性能の低下が見られる。
3. 同期・非同期の台数を変化させることで、ある程度一定の割合で性能値も変化している。

特に3. に関しては、今回検証を行った環境よりSlave数の多い構成でも サイジングの基礎値として利用可能な値が取れたのではないかと考えられます。

また、今回の検証では原因の特定までには至りませんでしたが、性能に影響を与える要因として、ネットワークやストレージのリソースを共有していることによるオーバーヘッドが考えられます。レプリケーション構成を考える際は、PostgreSQLの設定だけではなく、サーバの配置やマシンの構成も 考慮する必要があると言えそうです。

9. FDW検証

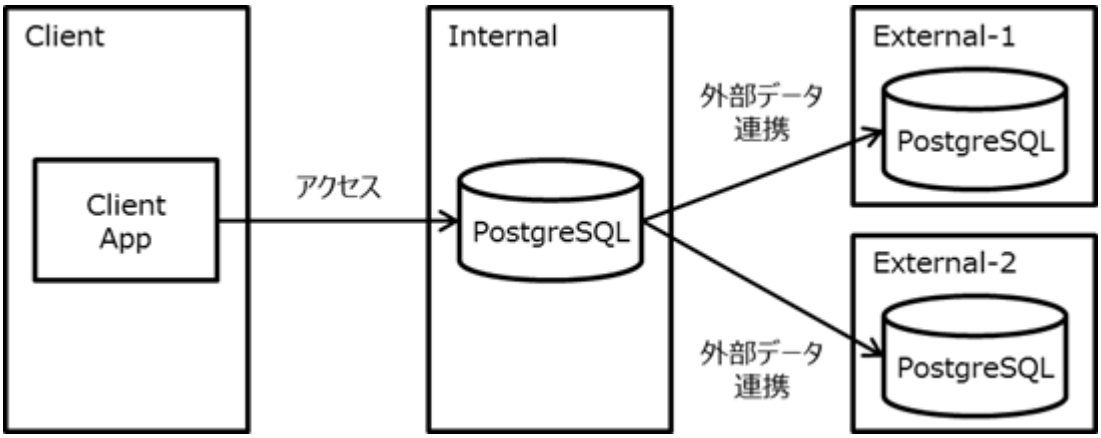
9.1. 検証概要

PostgreSQLから外部のデータへアクセスするための仕組みであるFDW(Foreign Data Wrapper)は近年PostgreSQLが最も力を入れて強化している機能の一つです。この機能を用いることで、点在する様々な外部データをPostgreSQLから一括して管理する事ができるため、企業のデータ活用に非常に有望視されています。しかしながら、性能的なオーバヘッドに対する懸念も存在します。そこで本検証では、PostgreSQL 10系のFDW機能に注目して、参照系、更新系における性能オーバヘッドを明らかにします。

検証用データベースにPostgreSQLに標準で同梱されているpgbenchから生成したものに加え、CREATE TABLEによりカスタムに作成したテーブルを組み合わせ使用しています。詳細は後述します。また、FDWモジュールにはPostgreSQL9.3より実装されているpostgres_fdwを使用しました。このモジュールを用いることで、外部に存在するPostgreSQLのテーブルを内部に存在するテーブルと同様に扱うことができます。なお、FDWモジュールとしてはPostgreSQLだけでなく、MySQL、OracleDB、CSVファイル、MongoDB等、様々なデータソースへアクセス可能なものが提供されています。詳細は公式wikiに記載されています（https://wiki.postgresql.org/wiki/Foreign_data_wrappers）。

9.2. 検証構成

本検証では、負荷クライアント(以降「Client」)からデータベースサーバ(以降、「Internal」)のいくつかのテーブルにアクセスします。「Internal」のテーブルに格納されるデータの実体は、「Internal」の中に存在する場合もあるし、FDW機能によって接続された2つの外部データベースサーバ(以降、「External-1」「External-2」)に存在する場合もあります。



9.2.1. ハードウェア構成

ハードウェア構成を 表. 9.1 に示します。なお、今回、データベースサーバはすべて仮想環境としています。本検証は、FDW機能を利用する場合と利用しない場合でのオーバヘッドを測定する事が目的であるため、仮想環境である点は性能面で考慮不要と考えました。

表 9.1 ハードウェア構成

	種別	CPU	メモリ	ディスク
Internal	仮想	2.40GHz 2cores	4 GB	20 GB
External-1	仮想	2.40GHz 2cores	4 GB	20 GB
External-2	仮想	2.40GHz 2cores	4 GB	20 GB
Client	物理	Intel(R) Xeon(R) E5620 2.40GHz 16cores	24 GB	

9.2.2. ソフトウェア構成

ソフトウェア構成を 表. 9.2 に示します。

表 9.2 ソフトウェア構成

	OS	ソフトウェア
Internal	CentOS 7.4(1708)	PostgreSQL 10.3
External-1	CentOS 7.4(1708)	PostgreSQL 10.3
External-2	CentOS 7.4(1708)	PostgreSQL 10.3
Client	CentOS 7.4(1708)	JDBC Runner 1.2 OpenJDK 1.8.0_161 postgresql-jdbc 42.2.2-1

9.2.3. データベース構成

9.2.3.1. テーブル構成

本検証では、簡単な銀行の口座処理を模擬しています。この銀行のデータベースは、顧客情報(addrbook)、口座情報 (pgbench_accounts)、取引履歴(pgbench_history)の3つのテーブルを持っています。なお、今回は1人の顧客が1つの口座を持つ(顧客のPRIMARY KEYであるIDと口座のPRIMARY KEYであるIDが1対1に紐付く)ことを想定としています。なお、口座情報(pgbench_accounts)、取引履歴(pgbench_history)については、PostgreSQLの標準ベンチマークツールであるpgbenchを利用して作成し、

顧客情報(addrbook)テーブルについては下記のCREATE TABLEにより個別なテーブルを作成しました。

- 顧客情報(addrbook)

氏名、メールアドレス、性別、年齢などの情報を持つ顧客テーブルを以下の様に作成しました。なお、顧客情報テーブルには、「なんちゃって個人情報(http://kazina.com/dummy/index.html)」を利用して作成した10万件のデータを格納しています。

```
CREATE SEQUENCE addrbook_id_seq;
CREATE TABLE addrbook (
  id          bigint PRIMARY KEY DEFAULT nextval('addrbook_id_seq'::regclass),
  name        character varying(32),
  kana        character varying(32),
  email       character varying(64),
  gender      character(1),
  age         integer,
  birthday    date,
  marriage    character(2),
  blood_type  character(3),
  division    character varying(4),
  phone_no    character varying(12),
  mobile_no   character varying(13),
  carrier     character varying(10),
  note        text
);
```

- 講座情報(pgbench_accounts)、取引履歴(pgbench_history)

講座情報(pgbench_accounts)、取引履歴(pgbench_history)テーブルは、pgbenchにてScaling Factor = 1で生成しました。下記にテーブル生成時のコマンドを示します。

```
$ /usr/pgsql-10/bin/pgbench -i -s 1 fdwdb
```

9.2.3.2. PostgreSQLの設定

今回の検証環境においてデフォルトより変更した設定について以下に示します。

- postgresql.confの設定

```
listen_addresses = '*'
shared_buffers = 1024MB
work_mem = 8MB
effective_cache_size = 1GB
max_connections = 110
```

- pg_hba.confの設定

```
host    all       all     (Client's IP Address)/24        trust
```

9.3. 検証方法

検証は参照・更新・挿入のそれぞれの処理を外部データの有無のパターンで測定し、内部テーブルのみの場合を基準値とした場合において、外部テーブルの変化/増加に伴いどの程度の性能オーバーヘッドが発生するかを検証していきます。

9.3.1. 測定方法

以下の方法で測定を行いました。なお、測定には「JDBC Runner」を用いています。

- 共通設定

  ・走行開始後10秒は計測を行わず、10秒経過後に60秒間のスループットを計測します。

  ・ 各パターン3回測定して、中央値を採用します。

- 参照(SELECT)での設定

  ・ 同時接続数は1、10、20、30、40、50、60、70、80、90、100の11パターン。

- 更新(UPDATE)、挿入(INSERT)での設定

  ・ 同時接続数は1のみ(注)

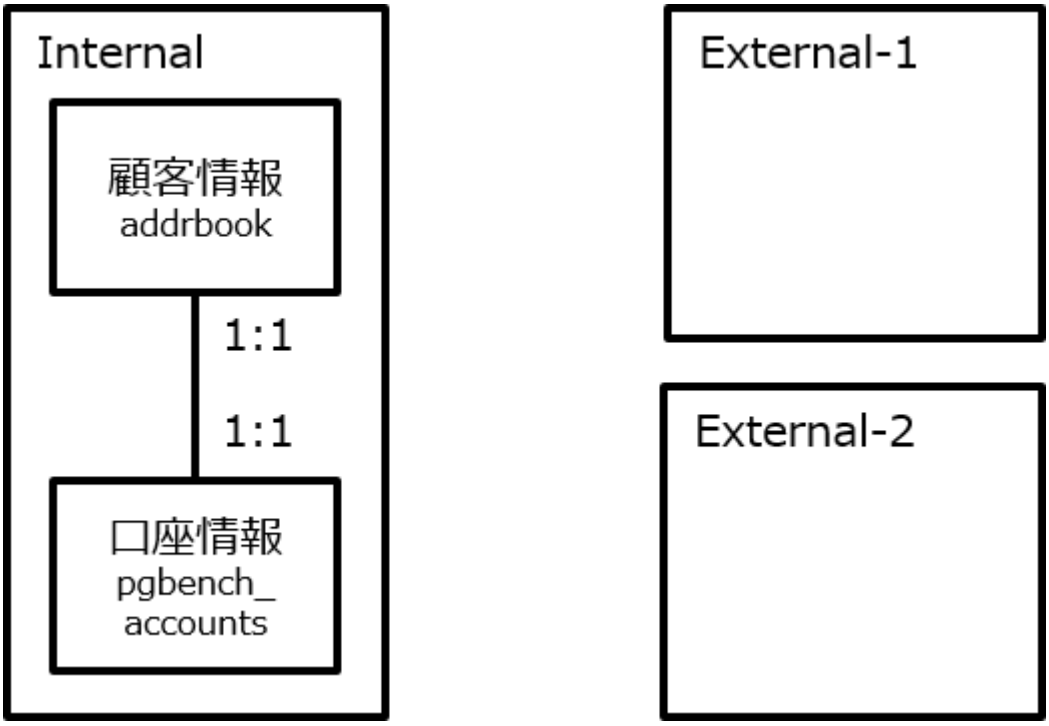
(注)FDWを使用した場合、外部データベース側のPostgreSQLでは、トランザクション隔離レベルが、SERIALIZABLEかREPEATABLE READしか使用できず、多重度が増すとUPDATE時に「could not serialize access due to concurrent update」エラーが頻発するため、更新、挿入では、同時接続数1のみ実施しています。

9.3.2. 参照(SELECT)

9.3.2.1. 検証パターン

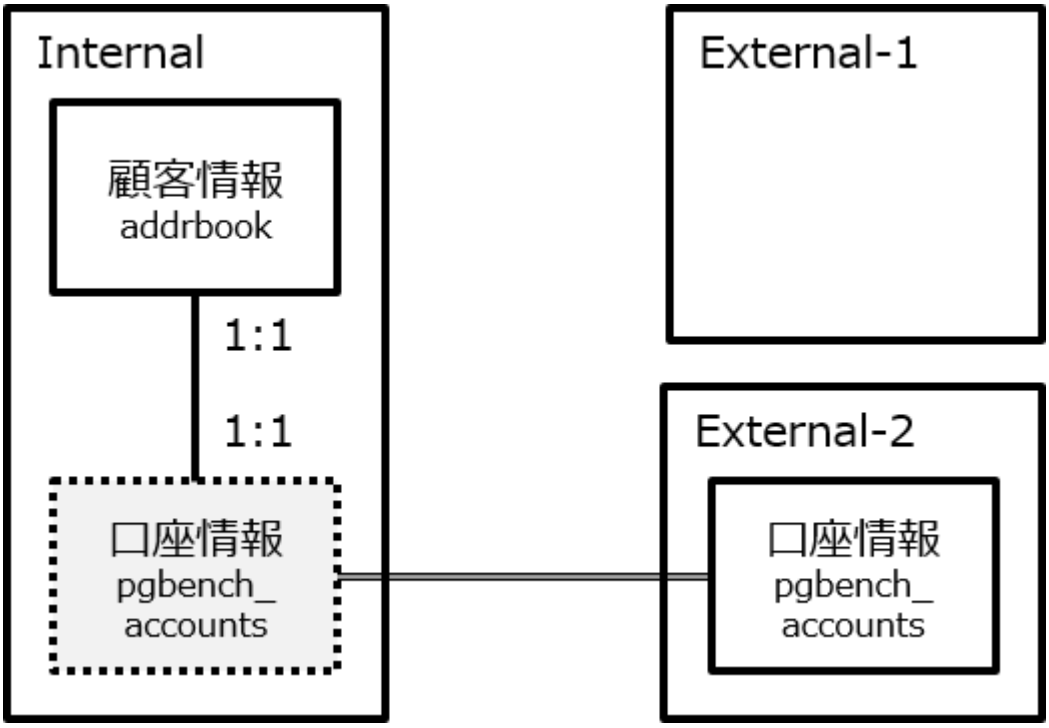
- パターン1

  顧客情報(addrbook)、口座情報(pgbench_accounts)共に内部テーブルである場合。



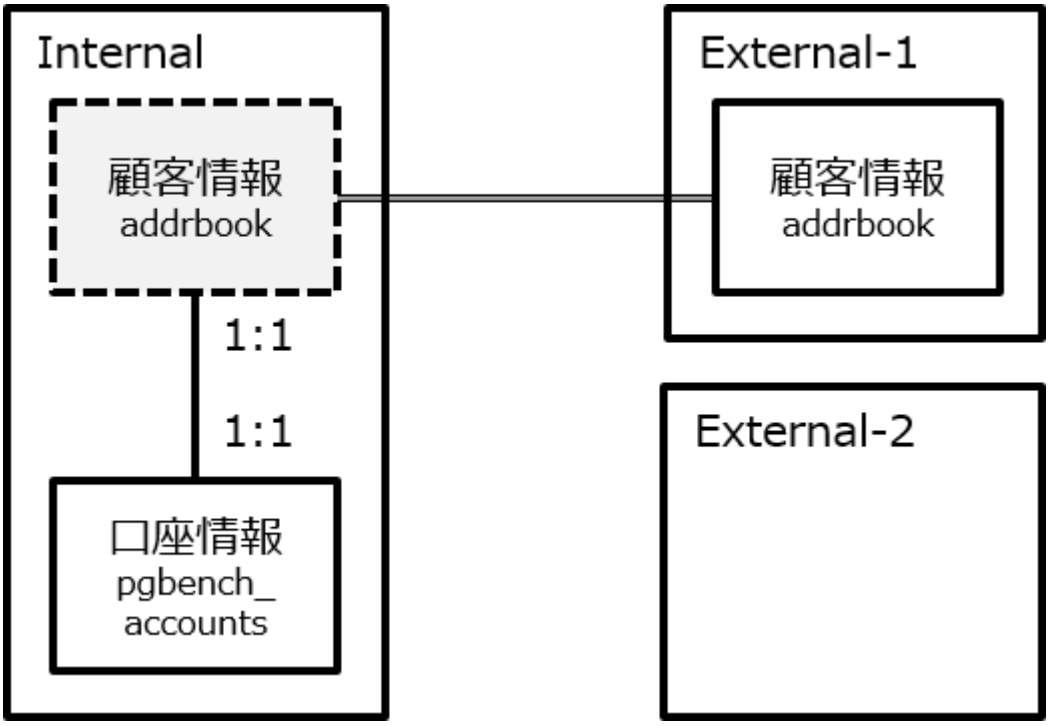
- パターン2

  顧客情報(addrbook)が内部テーブル、口座情報(pgbench_accounts)が外部テーブルである場合。



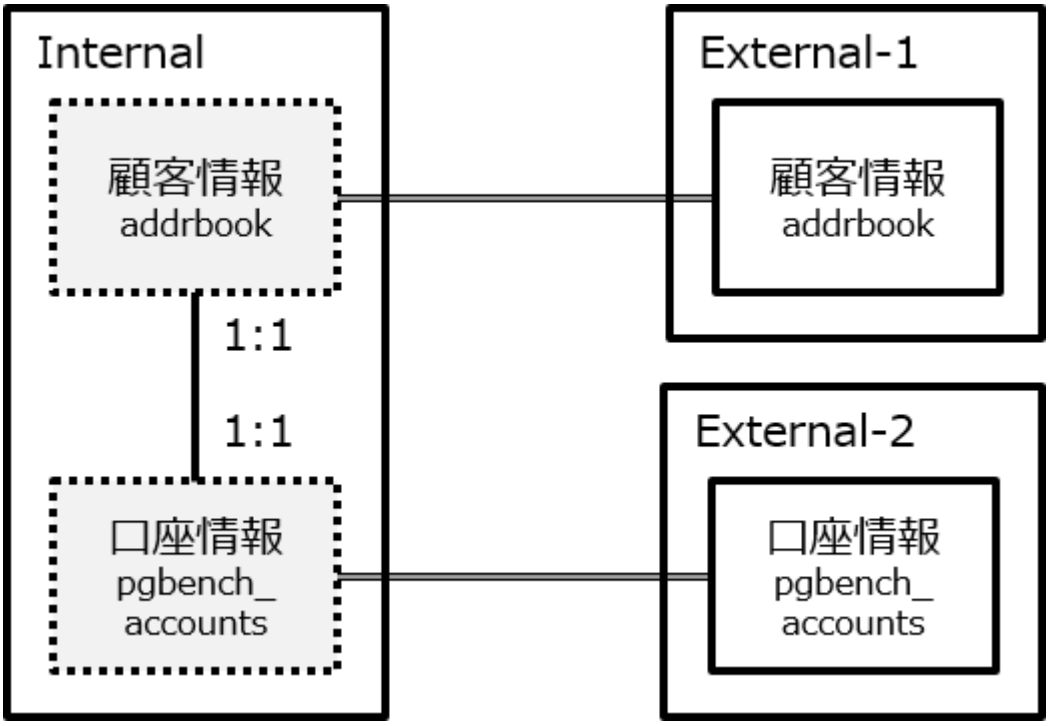
- パターン3

  顧客情報(addrbook)が外部テーブル、口座情報(pgbench_accounts)が内部テーブルである場合。



- パターン4

  顧客情報(addrbook)、口座情報(pgbench_accounts)共に外部テーブルである場合。



9.3.2.2. データベースへのアクセス

ある顧客に対して、顧客情報(addrbook)から氏名(name)、年齢(age)、口座情報(pgbench_account)から残高(abalance)を抽出します。

```
SELECT ad.name, ad.age, ac.abalance
FROM addrbook ad, pgbench_accounts ac
WHERE ad.id = $int AND ad.id = ac.aid
```

9.3.3. 更新(UPDATE)

9.3.3.1. 検証パターン

更新対象のテーブル(今回は口座情報である、pgbench_accnounts)が内部データの場合と外部データの場合で測定します。

9.3.3.2. データベースへのアクセス

ある口座(pgbench_accounts)の残高(abalance)を増加させます。

```
UPDATE pgbench_accounts
SET abalance = abalance + $int_a WHERE aid = $int_b
```

$int_a = -999999 〜 999999 の整数(乱数)
$int_b = 1 〜 100000 の整数(乱数)

9.3.4. 挿入(INSERT)

9.3.4.1. 検証パターン

データの挿入対象のテーブル(今回は取引履歴であるpgbench_history)が内部データの場合と外部データの場合で測定します。

9.3.4.2. データベースへのアクセス

取引履歴(pgbench_history)に適当な履歴を挿入します。

```
INSERT INTO pgbench_history (tid, bid, aid, delta)
VALUES ($int_a, 1, $int_b, $int_c)
```

$int_a = 1 〜 10 の整数(乱数)
$int_b = 1 〜 100000 の整数(乱数)
$int_c = -999999 〜 999999 の整数(乱数)

9.4. 検証結果

9.4.1. 参照(SELECT)

外部テーブル参照のあり/無しで大きく性能が異なりました。下記図と 表. 9.3 にその結果を示します。

またパターン2、3、4にはパターン1を基準とした時の性能(Throughput)劣化率を記載しています。

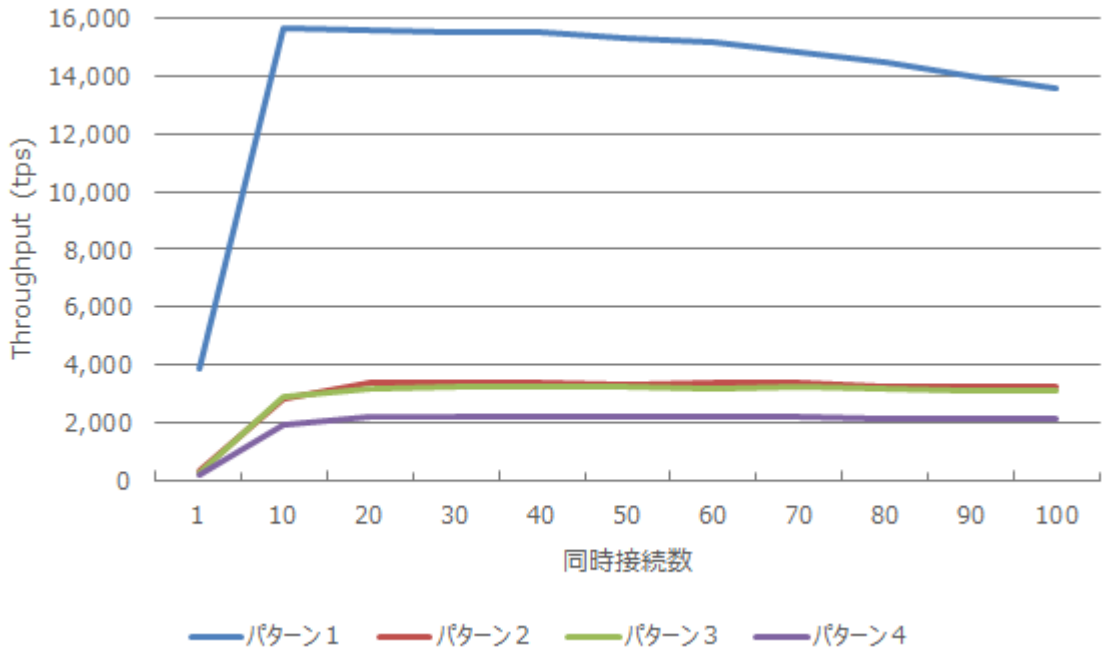


表 9.3 結果一覧

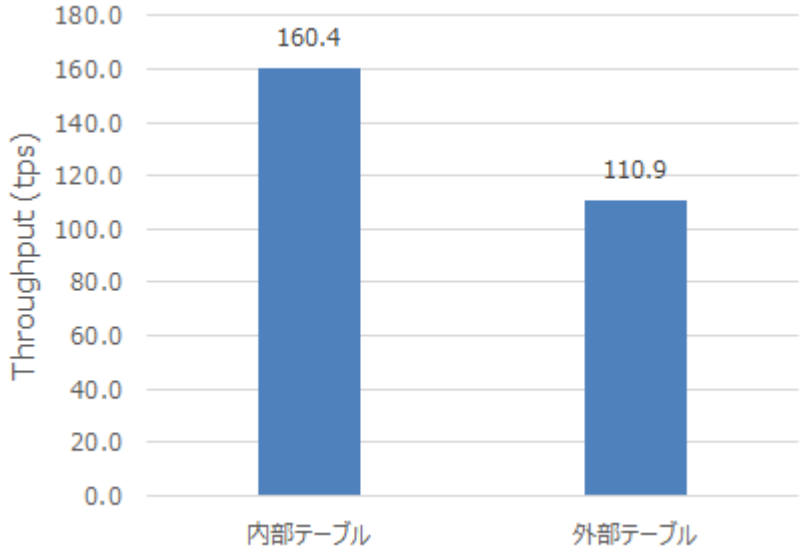
	1	10	20	30	40	50	60	70	80	90	100
パターン1	3869.2	15675.2	15593.4	15561.3	15517.1	15350.5	15164.8	14872.3	14492.1	14016.6	13605.8
パターン2	296.4 ▲92.3%	2852.0 ▲81.8%	3347.2 ▲78.5%	3379.8 ▲78.23%	3353.4 ▲78.4%	3340.5 ▲78.2%	3350.8 ▲77.9%	3350.1 ▲77.5%	3222.5 ▲77.8%	3269.2 ▲76.7%	3222.8 ▲76.3%
パターン3	276.6 ▲92.9%	2916.8 ▲81.4%	3181.6 ▲79.6%	3240.4 ▲79.2%	3253.6 ▲79.0%	3229.8 ▲79.0%	3199.2 ▲78.9%	3222.1 ▲78.3%	3145.7 ▲78.3%	3119.1 ▲77.7%	3108.5 ▲77.2%
パターン4	161.7 ▲95.8%	1905.5 ▲87.8%	2204.5 ▲85.9%	2214.0 ▲85.8%	2231.1 ▲85.6%	2184.2 ▲85.8%	2208.6 ▲85.4%	2166.6 ▲85.4%	2125.3 ▲85.3%	2118.9 ▲84.9%	2112.7 ▲84.5%

パターン1：顧客情報(addrbook)、口座情報(pgbench_accounts)共に内部テーブル
パターン2：顧客情報(addrbook)が内部テーブル、口座情報(pgbench_accounts)が外部テーブル
パターン3：顧客情報(addrbook)が外部テーブル、口座情報(pgbench_accounts)が内部テーブル
パターン4：顧客情報(addrbook)、口座情報(pgbench_accounts)共に外部テーブル

同時接続数1の場合では、片方が外部テーブルとなるパターン2/パターン3において▲92～93%、両方が外部テーブルとなるパターン4において▲96%の性能(Throughput)劣化となりました。同時接続数10～100において大きく傾向は変わりませんが、パターン2/パターン3において▲76～81%、パターン4において▲84～87%劣化していました。

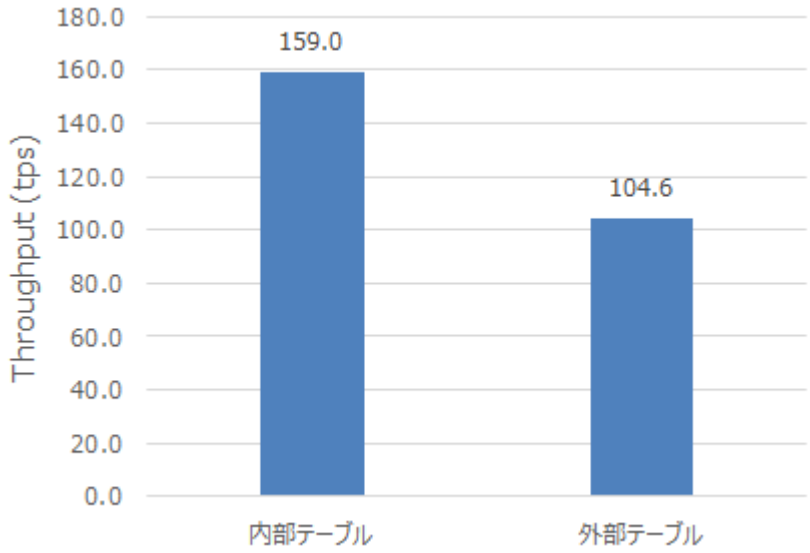
9.4.2. 更新(UPDATE)

今回の環境で外部テーブルを更新した場合(FDW機能あり)、内部テーブルの更新(FDW機能なし)に比べて、約31%性能(Throughput)が劣化しています。



9.4.3. 挿入 (INSERT)

今回の環境で外部テーブルにデータを挿入した場合(FDW 機能あり)、内部テーブルへの挿入(FDW 機能なし)に比べて、約34%性能(Throughput)が劣化しました。



9.5. 考察

参照においては、測定条件によって異なりますが▲76%～▲91%、更新・挿入においては▲35%前後の性能低下が見られました。

個々のアクセスのレスポンス時間を比較すると、同時接続数100の場合、内部テーブルのみで平均14ms、すべて外部テーブルの場合でも平均70msとなっています。今回、非常に単純なアクセスパターンであったため、内部テーブルのみのレスポンス時間が非常に短く、外部テーブルアクセスのオーバヘッドの影響が大きく出る結果となったと想定されます。

更新・挿入においてもレスポンス時間は全パターンで10ms前後であり、外部テーブルアクセスによるオーバヘッドの大きさは同様の傾向であると考えられます。

来年度以降の測定においては、複数のテーブルの結合など、内部テーブルのみであったとしても時間のかかる処理を対象として測定することで、外部テーブル処理のオーバヘッドをより明らかにできるのではないかと考えます。

10. おわりに

今年度(2017 年度)は、PostgreSQL エンタープライズ・コンソーシアム(PGECons)の発足から数えて 6 年目となり、わたくしども技術部会ワーキンググループ 1(以下、本 WG)も、今回で 6 期目となりました。本 WG では昨年度にPGEConsのワーキンググループの位置づけを見直してから、性能に加えて新技術の検証も行う「新技術検証ワーキンググループ」として活動しました。本報告書では、以下の 6 つのテーマの検証結果を報告しました。

1. PostgreSQL 10および前バージョン9.6と比較したスケール性の検証
2. Windows版PostgreSQLとLinux版PostgreSQLの性能比較検証
3. PostgreSQL 10で強化されたパラレルクエリ検証
4. PostgreSQL 10で実装されたロジカルレプリケーションを利用した全文検索検証
5. PostgreSQL 9.6で実装された複数同期レプリケーション検証
6. Foreign Data Wrapper検証

以下では、今年度の検証活動を振り返って、具体的な進め方をご紹介することで、報告書のあとがきとしたいと思います。まず、テーマの選定にあたっては、参加メンバからテーマを募るとともに、PGECons の成果報告会や、オープンソースカンファレンスなどに参加されたお客様に記入いただくアンケートを参考に進めました。

例年通りの定点観測を行うことはもちろんのことながら、さらに今年度は定点観測と同様の検証モデルを用いて今までWG1で触れてこなかったWindows版PostgreSQLの性能検証を行うこととなりました。以前から、WindowsでもPostgreSQLサーバを立てるケースが増えるに従い、Windows版PostgreSQLの実際の性能、特にLinux版PostgreSQLと比較した際の性能を知りたいという要望が多くなってきました。今回のWindows版PostgreSQL検証ではLinux版との性能比較に加えてWindows版固有のチューニング要素も検証対象となっており、Windows版PostgreSQL利用者にとって有用な情報を提供できることが期待されます。また、検証テーマを考慮する上でPostgreSQL最新バージョンの新機能は外せません。バージョンが10となった今回は強化されたパラレルクエリ、ロジカルレプリケーションが検証対象となりました。さらに、複数同期レプリケーション検証やForeign Data Wrapperの検証といったテーマは過去に検証しきれなかったテーマを改めて今回選定しています。

各テーマごとの検証活動は担当する企業が主体となって進めます。各担当企業が作成した検証計画は、定例の会合で議論しました。提出された検証計画に対して「どのような観点で・何を計測することで、検証の目的が満たされるのか」検討します。その中で不足している観点や冗長な測定パターンが明らかになることもあります。こうして検証計画が固まると、約2週間から1ヶ月程度、実機上で測定作業をします。得られたデータは整理されて会合に提出されます。「検証の目的に適ったデータが得られているか」確認し、「そのデータの意味するところは何か」考察して、少しでも質の高い結果が得られるように議論を重ねます。

こうした作業を積み重ねることで、単にベンチマークプログラムの測定値に留まらない、有用な結果が得られます。これらは、日常の業務で PostgreSQL に深く係わり、その挙動やソースコードを熟知したメンバが集まって初めて達成できた成果です。PGECons の会合は、異なる企業に属するメンバが集まって、専門的な意見を交わす貴重な場となっており、メンバの技術力の向上にも寄与しているものと思います。さらに今年度から議論のためにチャットツールを導入したところ、以前よりも圧倒的に意見のやりとりの回数が増えました。これはメンバー間の連帯感向上、ひいては各検証における考察の質向上に寄与しました。

今回の報告書が皆様の PostgreSQL の活用のご参考になれば幸いです。報告書へのご意見がございましたら、PGECons までお寄せください。また、PGECons の活動にご関心をお持ちの方は、是非、参加をご検討ください(ご連絡にはこちらのWebサイト をご利用ください)。

(終わり)

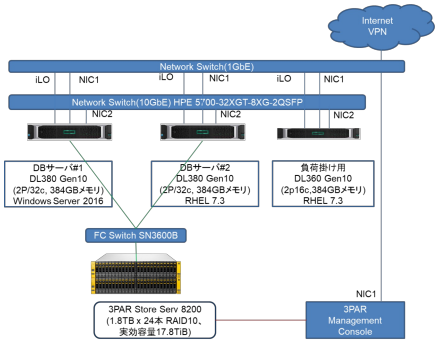
11. 著者

（企業・団体名順）

版	所属企業・団体名	部署名	氏名
第1.0版 (2017年度WG1)	SRA OSS, Inc. 日本支社	OSS事業本部	近藤 雄太
	NECソリューションイノベータ株式会社	サポートサービス事業部	近藤 太樹
	NTTテクノクロス株式会社	IoTイノベーション事業部	原田 登志
	NTTテクノクロス株式会社	IoTイノベーション事業部	勝俣 智成
	株式会社クリアコード		須藤 功平
	株式会社クリアコード		堀本 泰弘
	日本電信電話株式会社	オープンソースソフトウェアセンタ	坂田 哲夫
	富士通株式会社	共通ソフトウェア開発技術本部 ソフトウェア開発技術統括部 OSS技術センター	野山 孝太郎
	富士通株式会社	共通ソフトウェア開発技術本部 ソフトウェア開発技術統括部 OSS技術センター	白石 明大
	富士通株式会社	共通ソフトウェア開発技術本部 ソフトウェア開発技術統括部 OSS技術センター	高澤 亮平

12. Appendix 検証環境

- 提供：日本ヒューレット・パッカード株式会社
- 定点観測（スケールアップ)/Windows 検証/パラレルクエリ検証での使用機器/設備



- 提供：日本電気株式会社
- パラレルクエリ検証での使用機器/設備

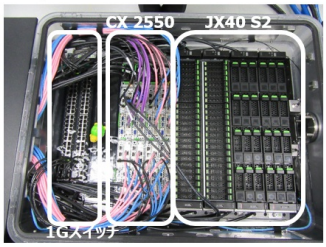
型番	Express5800/R120f-1E
CPU	インテル Xeon E5-2630 v4@2.20GHz
メモリ	256GB
内蔵ストレージ	SSD 400GB x 8 (RAID 5構成)
OS	Red Hat Enterprise Linux 7.2

- 提供：富士通株式会社
- 複数同期レプリケーション検証/FDW 検証での使用機器/設備

■ 液浸サーバ

■ サーバなどの機器を丸ごと液体の冷媒に浸す「液浸冷却技術」を使用

機種	FUJITSU Server PRIMERGY CX2550 M2
CPU	インテル Xeon E5-2650 v4@2.2GHz（48Core）
メモリ	256GB
ストレージ	2.5インチSSD 400GB×24（キャビネット：JX40 S2）
OS	CentOS 7.3



- Amazon Web Service
- 全文検索検証での使用機器/設備

項目	仕様
機種	Amazon Web Service(AWS) m4.2xlarge
CPU	Intel(R) Xeon(R) CPU E5-2686 v4 @ 2.30GHz (8コア)
メモリ	32GB
内蔵ストレージ	SSD 256GB