PostgreSQLエンタープライズ・コンソーシアム 技術部会 WG#1

2018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	2019/03/29	新規作成 定点観測(スケールアップ検証)、及びWindows検証を公開
1.1	2019/04/05	JITコンパイル検証、及びパラレルクエリ検証を公開
1.2	2019/11/26	EDB版インストーラ問題に関するEDB社の調査結果を追記

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)
  - 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、2018年10月にリリースされた PostgreSQL 11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. 2018年度の活動テーマ

2018年度は、2018年10月18日にリリースされた PostgreSQL 11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
- JITコンパイル
- パラレルクエリ

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。参照系クエリの性能は 10 と 11 で差は見られませんでした。しかし、同検証で副次的に集約計算性能の向上を観察することができました。更新系クエリの性能は 11 で性能向上が見られました。これは clog に関する処理の変更が主な原因でした。また、昨年度発覚した 10 の更新系クエリの性能低下について再検証したところ、推測していた通りセマフォの変更が原因であることが確認できました。

3.3.3. Windows版PostgreSQL性能

昨年に引き続き、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、参照系はCPU数を超えない範囲の同時接続数、つまり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。更新系は、同時接続数が32を超える場合に、Linux版と比較してWindows版のtpsが大きく劣るという結果となりました。

3.3.4. JITコンパイル

PostgreSQL11で追加されたJITコンパイルについて、有効なクエリ種の調査と実行時間の削減効果の検証を行いました。四則演算と集約関数を含むクエリに有効であることがわかりました。また、カラムのデータ型の違いによっても削減効果の違いがあることがわかりました。OLAP系の大規模なクエリを実行した場合には、削減効果は見られたもののクエリの内容によって効果に差があることがわかりました。

3.3.5. パラレルクエリ

PostgreSQL 9.6にてメニーコアを有効活用するパラレルクエリが実装され、バージョン10以降で継続的に強化されています。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Star Schema Benchmark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9.5～11で、パラレルクエリが実装されたことによりどのように性能が変化するのかを検証しました。また、バージョン11にて実装された、B-Treeインデックスのパラレル作成について、パラメータによって、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、テーブルサイズによって決定される並列度までは性能が向上しましたが、実行計画の変化に起因して性能が頭打ちになる場合があることがわかりました。性能が頭打ちになる現象については、並列度を決定するロジックが不完全であることが原因とわかりました。

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. 2017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、2016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。

2017年度は、2017年10月にリリースされたPostgreSQL 10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 10で強化されたパラレルクエリの性能検証、新機能であるロジカルレプリケーションを利用した負荷分散の効果検証、同期・非同期レプリケーション構成や混在環境における性能検証、FDW機能利用時の性能検証を行いました。

3.5. 実施体制

2018年7月19日に開催された2018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 2018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
日本電信電話株式会社
富士通株式会社

この中で、NECソリューションイノベータ株式会社は、「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

2018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	2018年7月19日
実施計画策定	2018年7月～12月
検証実施	2019年1月～2019年2月
2018年度WG1活動報告書作成	2019年2月～2019年3月
総会と成果報告会	2019年4月25日(東京)、2019年5月10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、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。2018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 11 と前バージョンの 10 との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: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 10
number of transactions actually processed: 10/10
latency average = 4.433 ms
tps = 225.580586 (including connections establishing)
tps = 241.413390 (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

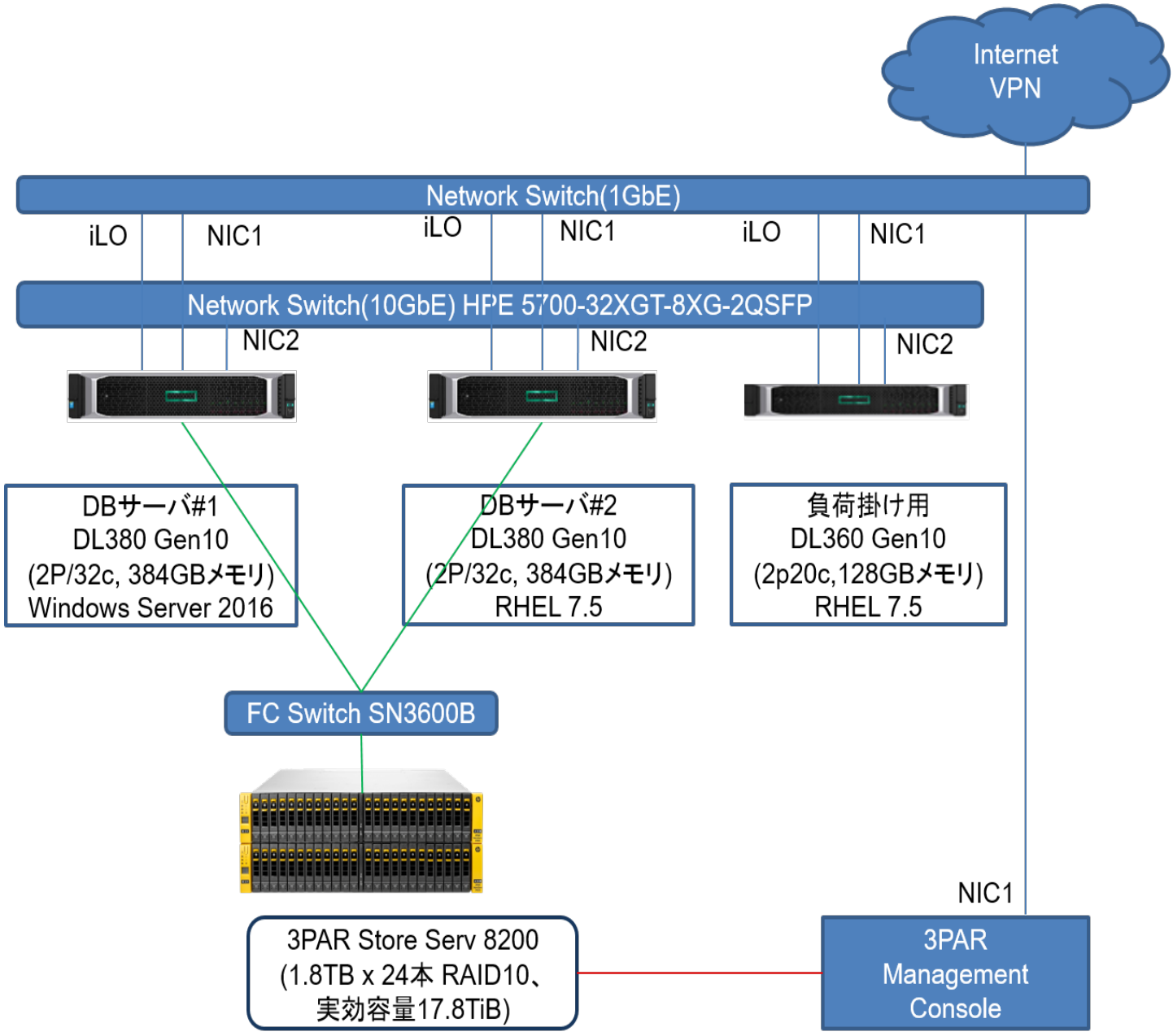


図 4.1 検証ハードウェア構成

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 7.5
PostgreSQL	10.6, 11.1

表 4.8 負荷掛け用

OS	Red Hat Enterprise Linux 7.5
pgbench	11.1

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
$ createdb [dbname]
```

ここからは、参照系性能と更新系性能についてそれぞれ検証していきます。

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 4 16 32 48 64 80 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。TPSは接続確立にかかった時間を無視した値(excluding connections establishing)を用いています。(以降同様)

10, 11 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。11 のTPSは平均 5% ほど 10 を上回る結果となりました。

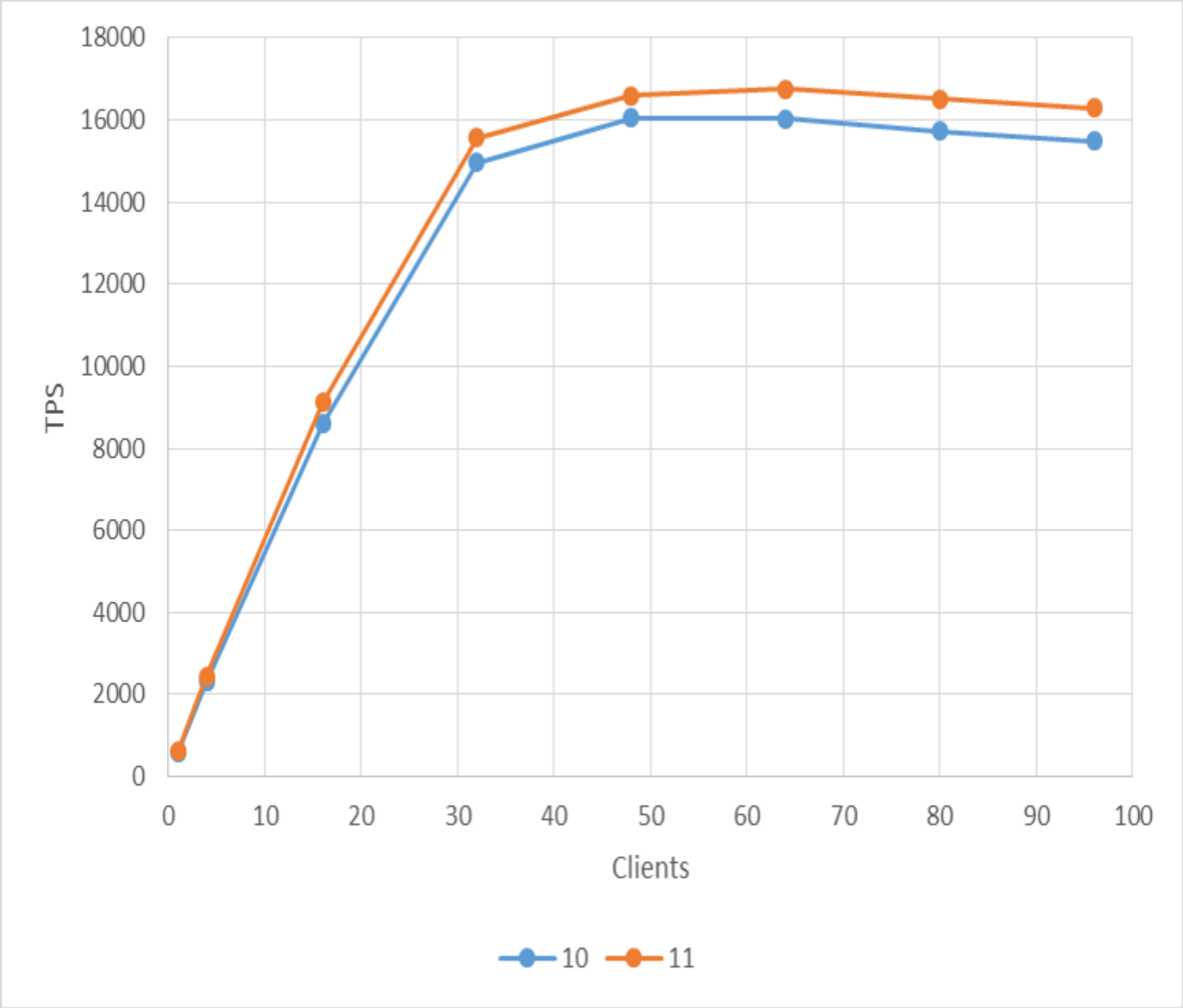


図 4.2 各クライアント数に対する TPS (参照系)

続いて、バージョン間の性能差異をもたらした原因を追求するため追加検証を行いました。

4.6. 追加検証(参照系)

上記の結果から 11 は 10 よりも参照系性能が向上したように見えますが、実は 11 の リリースノート には以下の記述があります。

Improve the speed of aggregate computations (Andres Freund)

「集約計算の速度が改善されました。」

このため、例年利用している以下のクエリには集約関数の count() が含まれているため、上記の性能向上は集約計算の性能向上のみによるものであった可能性があります。

そこで、検証に用いるクエリを、以下の集約計算を使用しないクエリに変更して 10, 11 のパフォーマンス比較を再度行いました。

```
SELECT abalance FROM pgbench_accounts WHERE aid BETWEEN :aid and :aid + :row_count offset
:row_count limit 1;
```

結果は以下の通りです。

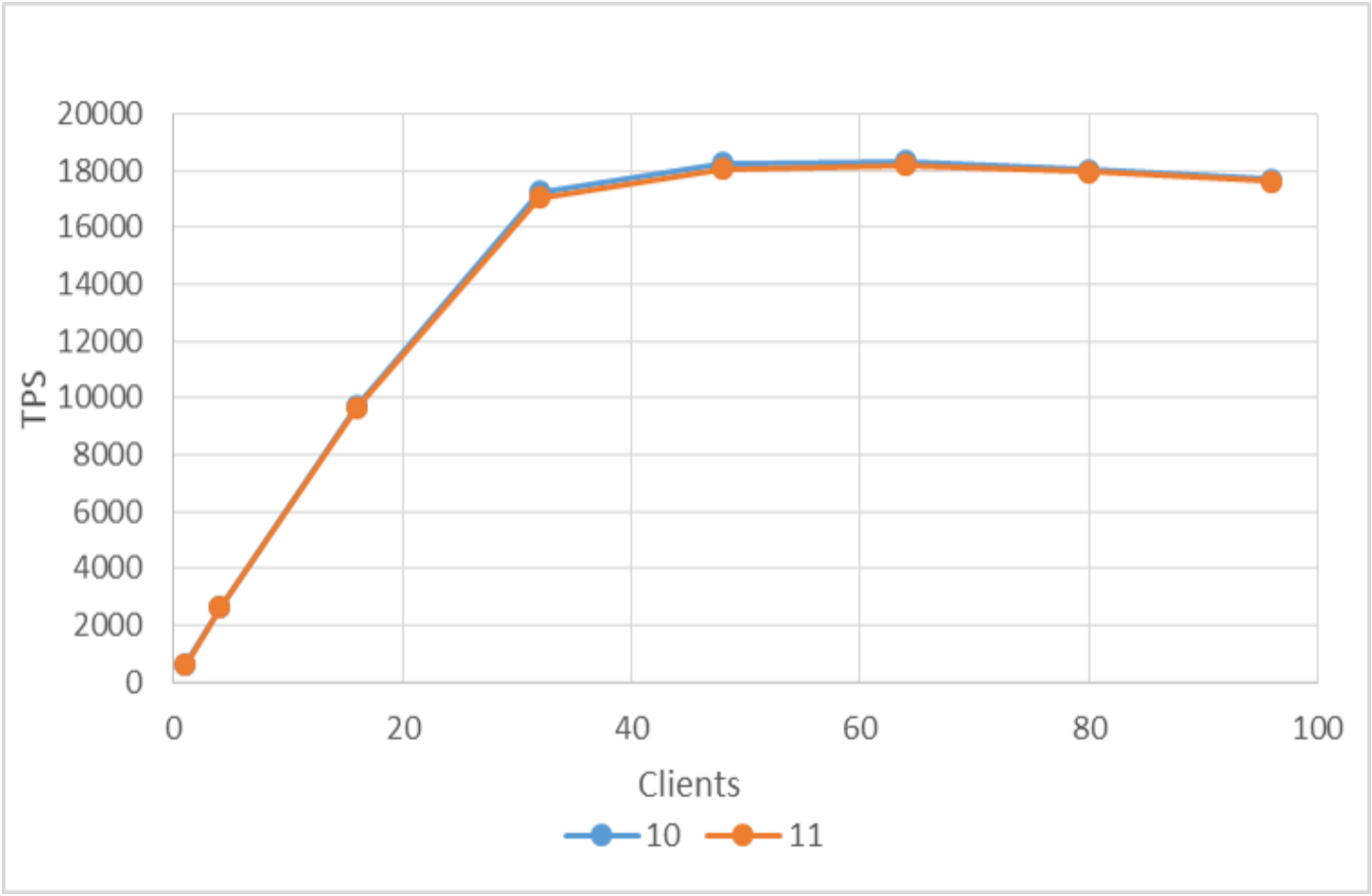


図 4.3 各クライアント数に対する TPS (参照系、集約計算無しクエリ)

さらに、集約計算を使用している元のクエリと集約計算を使用しない新しいクエリについてバージョン 10 から 11 に対しての性能向上率を比較したものが以下の通りです。

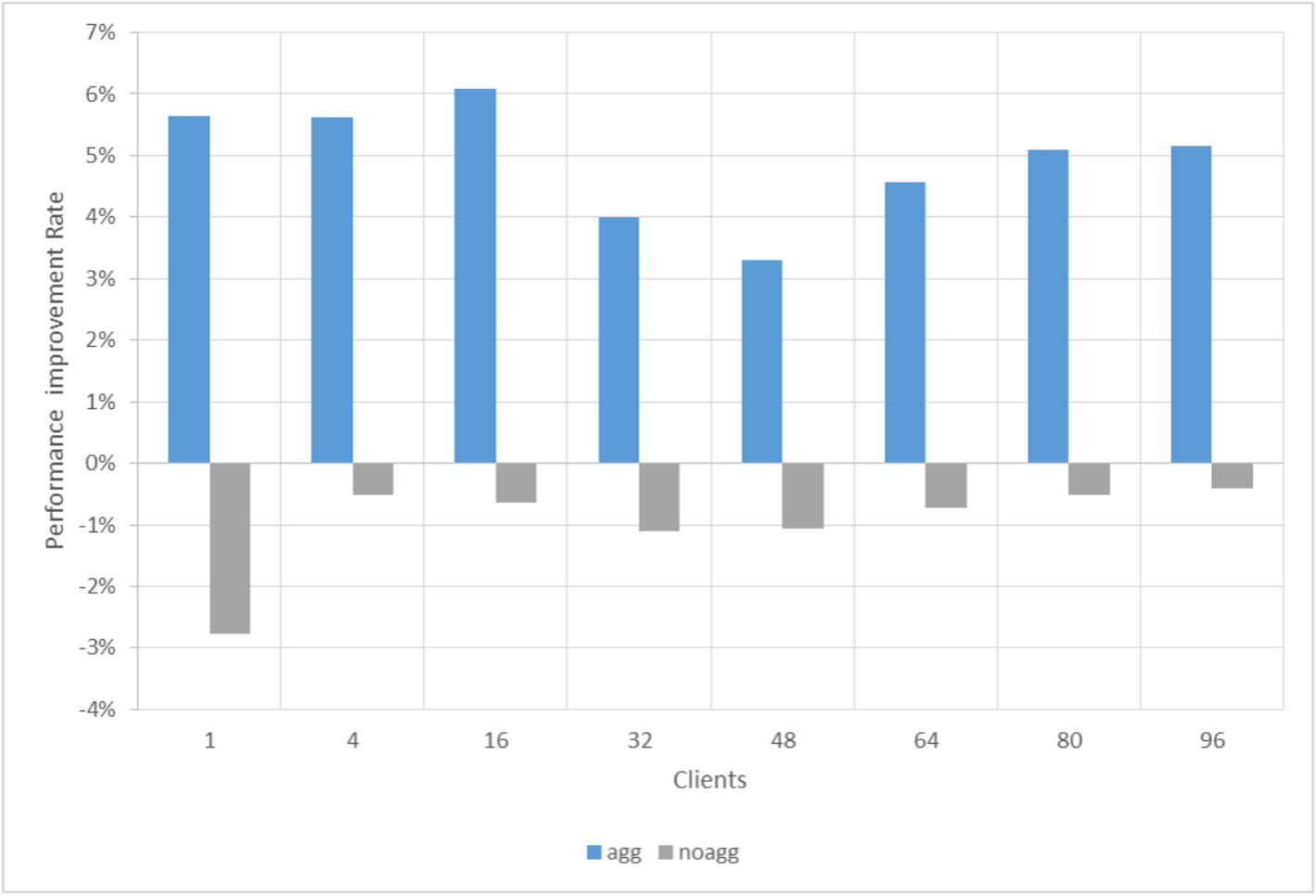


図 4.4 性能向上率比較 (agg:集約計算有り, noagg:集約計算無し)

集約計算を含んだクエリは各クライアント数で平均 5% 程度の性能向上しているのに対して、集約計算無しクエリは全く性能向上していませんでした。これらの結果から、最初の検証クエリの性能向上は集約計算の箇所が大きく影響していたことが示唆されます。よって、バージョン 10, 11 間では一般的な参照性能の向上はありませんでした。また、副次的ではありますが本検証では集約計算(ここではcount())における性能向上を観察することができました。

続いて、更新性能について検証します。

4.7. 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 4 16 32 48 64 80 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。

4.8. 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。10, 11 ともにクライアント数増加につれてTPSが増加しましたが、よりクライアント数が多いほど 11 のTPSが 10 のTPSを上回る結果となりました。96 クライアントに至っては 10% の向上が見られました。

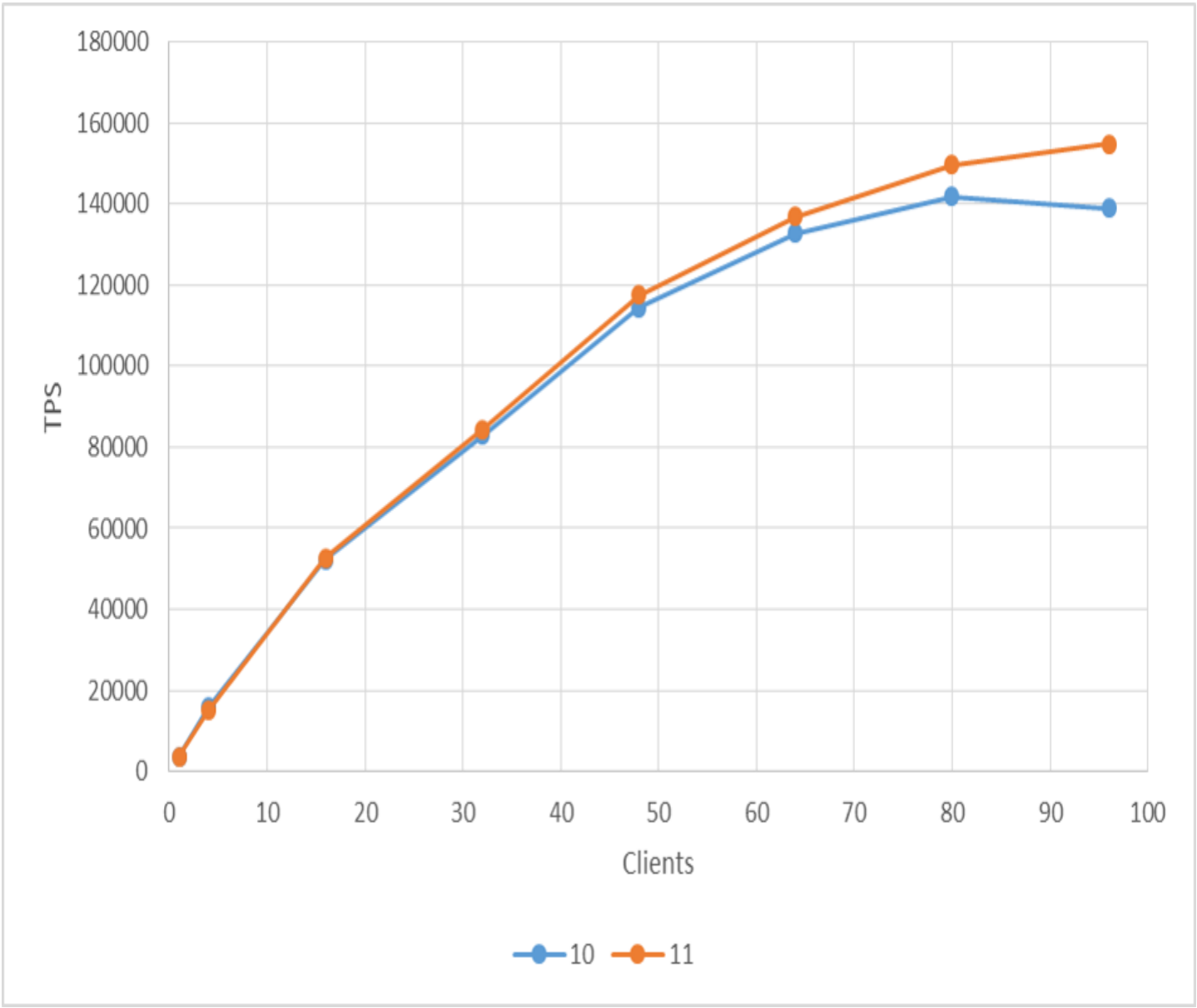


図 4.5 各クライアント数に対する TPS (更新系)

11 の更新性能向上の原因を解明するために別途、追加の検証を行いました。

4.9. 追加検証(更新系)

さて、ここで 11 の リリースノート を参照すると、

> Improve performance of committing multiple concurrent transactions (Amit Kapila)

「同時実行トランザクションのコミットの性能が改善されました。」

という記述があります。上記の記述にあたるソースコードコミットは [baaf272ac] であり、このコミットでは clog 更新処理をグループ化する実装が行われています。

clog 更新処理のグループ化とはつまり、複数のトランザクションによりトランザクションステータスの更新処理が競合している際に単一のリーダープロセスが複数の更新処理をまとめて行い、トランザクションステータス更新時に発生する CLogControlLock の回数を減らすための処理です。この処理自体の wait_event は ClogGroupUpdate として表されます。

追加検証では [baaf272ac] の実装が実際に性能に影響していたかどうかを確認します。そのために追加検証では [baaf272ac] の実装箇所を実装前の状態に戻したビルドを用います。

「バージョン10ビルド」、「バージョン11ビルド」、「バージョン11から[baaf272ac]の実装を外したビルド」で性能比較した結果は以下の通りです。このとき、10 と 11 で性能差が一番大きかった 96 クライアントで比較しています。

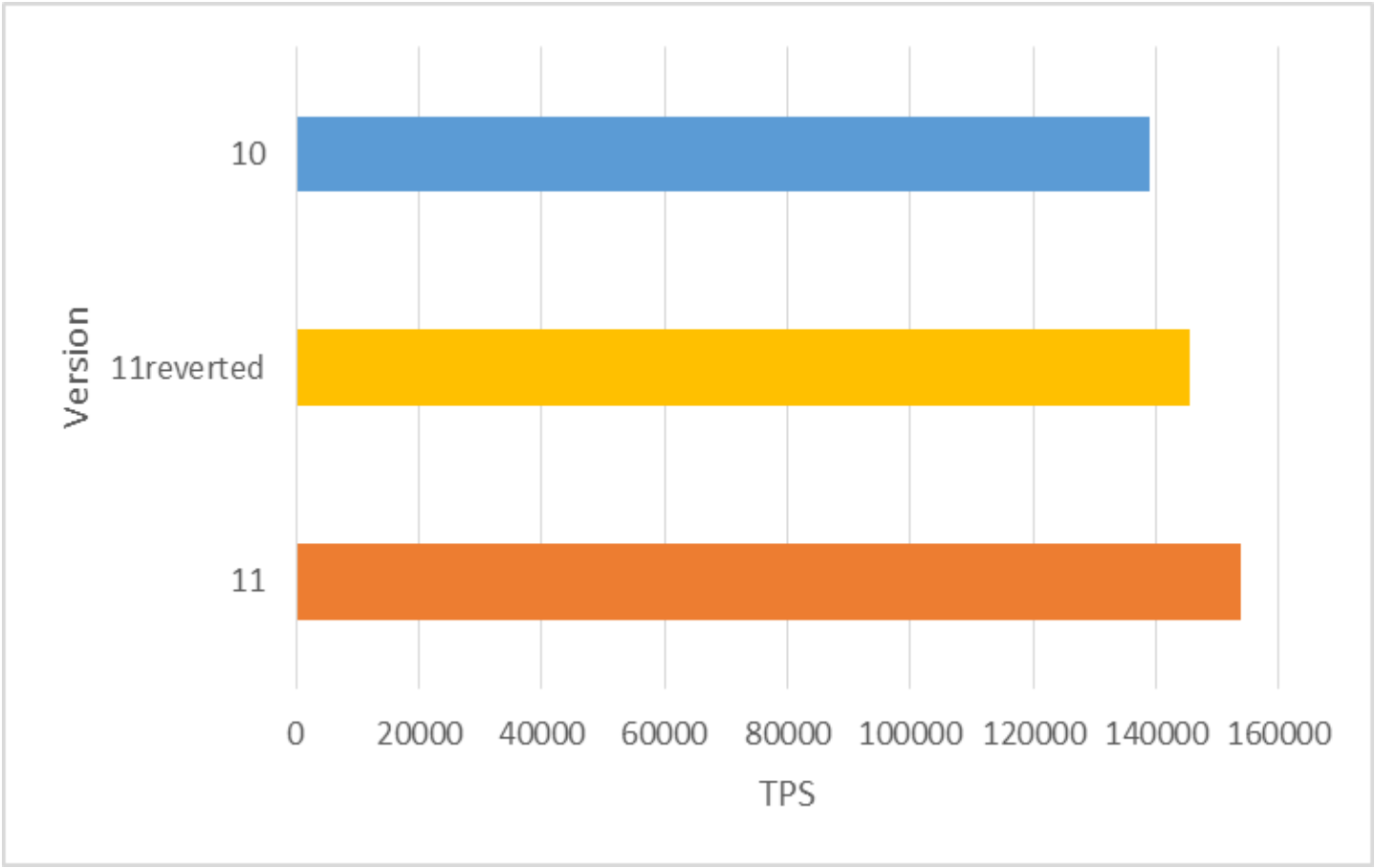


図 4.6 clog 更新のグループ化実装を外したビルドと比較（クライアント数 96）

図の「11reverted」が「バージョン11から[baaf272ac]の実装を外したビルド」です。

以上の結果から、10% の性能向上のうち 5% 以上は [baaf272ac] の修正が更新性能向上に寄与していることが確認できました。ただ残念ながら、残りの 5% 未満の性能向上については検証期間の関係から原因を突き止めるには至りませんでした。

4.10. 2017年度検証の残課題

昨年度はバージョン 9.6 と比較した 10 の更新系性能が大きく低下していることが発覚し、その原因をデフォルトセマフォの変更(System V → POSIX)によるものであると推測するところまで行いました。今回これを立証するために検証を行いました。

検証に用いたビルドのバージョンは昨年度検証と同じ 9.6.6, 10.1 および旧デフォルトセマフォの System V を指定してビルドした 10.1 を用いました。

バージョン 10 以降はソースコードの src/template/linux の以下の箇所を削除することで、利用するセマフォを System V に変更することができます。

```
# Prefer unnamed POSIX semaphores if available, unless user overrides choice
if test x"$PREFERRED_SEMAPHORES" = x"" ; then
  PREFERRED_SEMAPHORES=UNNAMED_POSIX
fi
```

セマフォの設定が変わったどうかは configure 実行後の config.log から確認できます。(以下は10.6の例です)

[POSIX セマフォが設定されている場合]

```
$ cat postgresql-10.6/config.log | grep -A 1 semaphore
configure:15720: checking which semaphore API to use
configure:15743: result: unnamed POSIX
```

[System V セマフォが設定されている場合]

```
$ cat postgresql-10.6x/config.log | grep -A 1 semaphore
configure:15720: checking which semaphore API to use
configure:15743: result: System V
```

System V セマフォを用いた 9.6.6, POSIX セマフォを用いた 10.1, SYSV セマフォを用いた 10.1 の各パフォーマンスを比較した結果は以下の通りです。

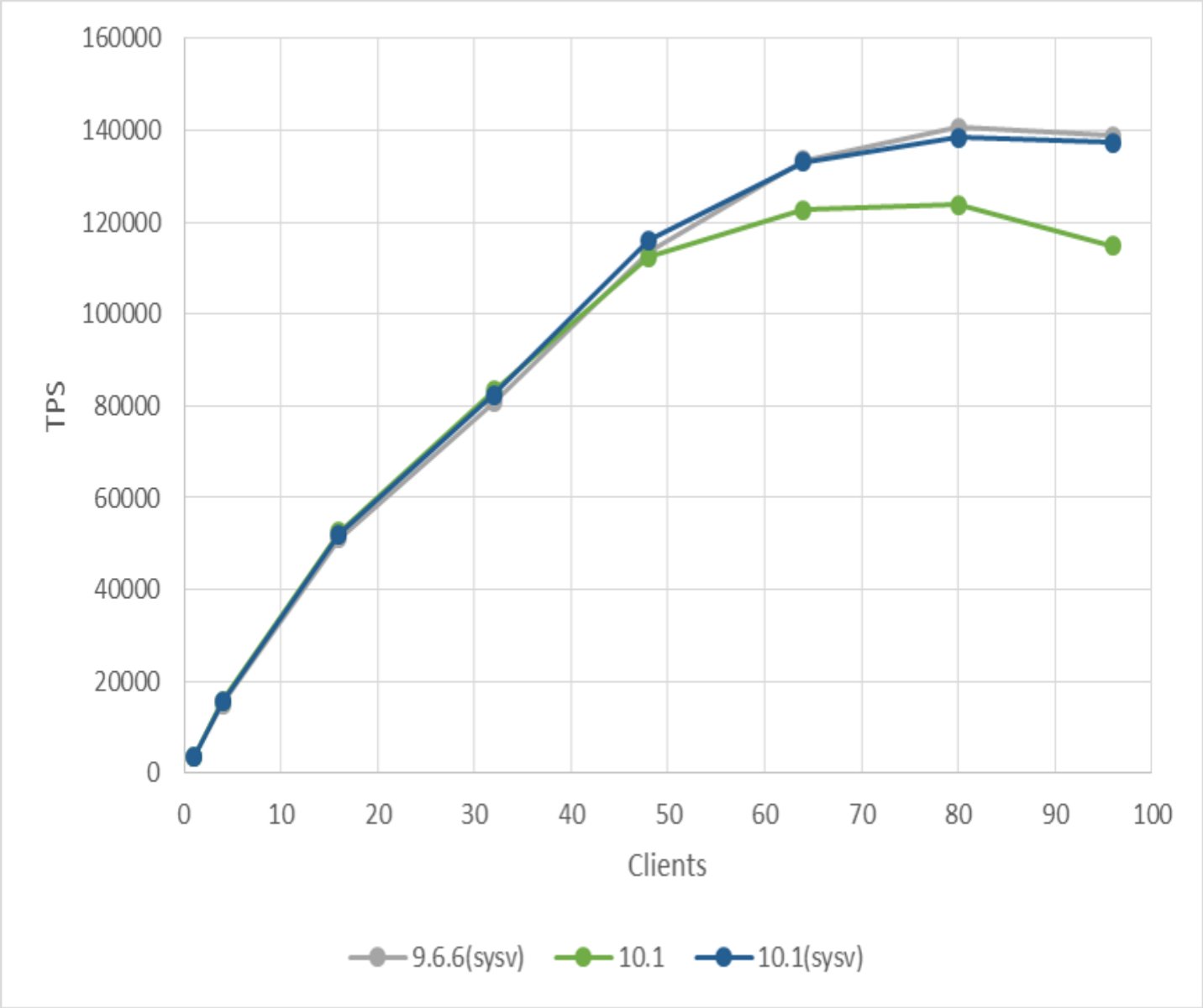


図 4.7 セマフォとバージョンの組み合わせごとのパフォーマンス比較

この結果から、同じバージョン 10 でも使用するセマフォを POSIX から 9.6 と同じ System V に戻せばパフォーマンスも同程度に戻ることが確認できました。

ところで、10.5 の リリースノート には以下のような記述があります。

> Pad arrays of unnamed POSIX semaphores to reduce cache line sharing (Thomas Munro) This reduces contention on many-

CPU systems, fixing a performance regression (compared to previous releases) on Linux and FreeBSD.

「キャッシュライン共用を減らすために、無名POSIXセマフォの配列を埋めこむようになりました。これは多CPUシステムでの競合を減らし、Linuxと FreeBSD での旧リリースと比較しての性能退行を修正します。」

実は、この修正によりバージョン 10.5 以降は POSIX セマフォを利用していても 9.6 のパフォーマンスを実現できるようになっています。この修正について実際に確認してみたものが以下の結果です。（検証で使用したのは10.6です）

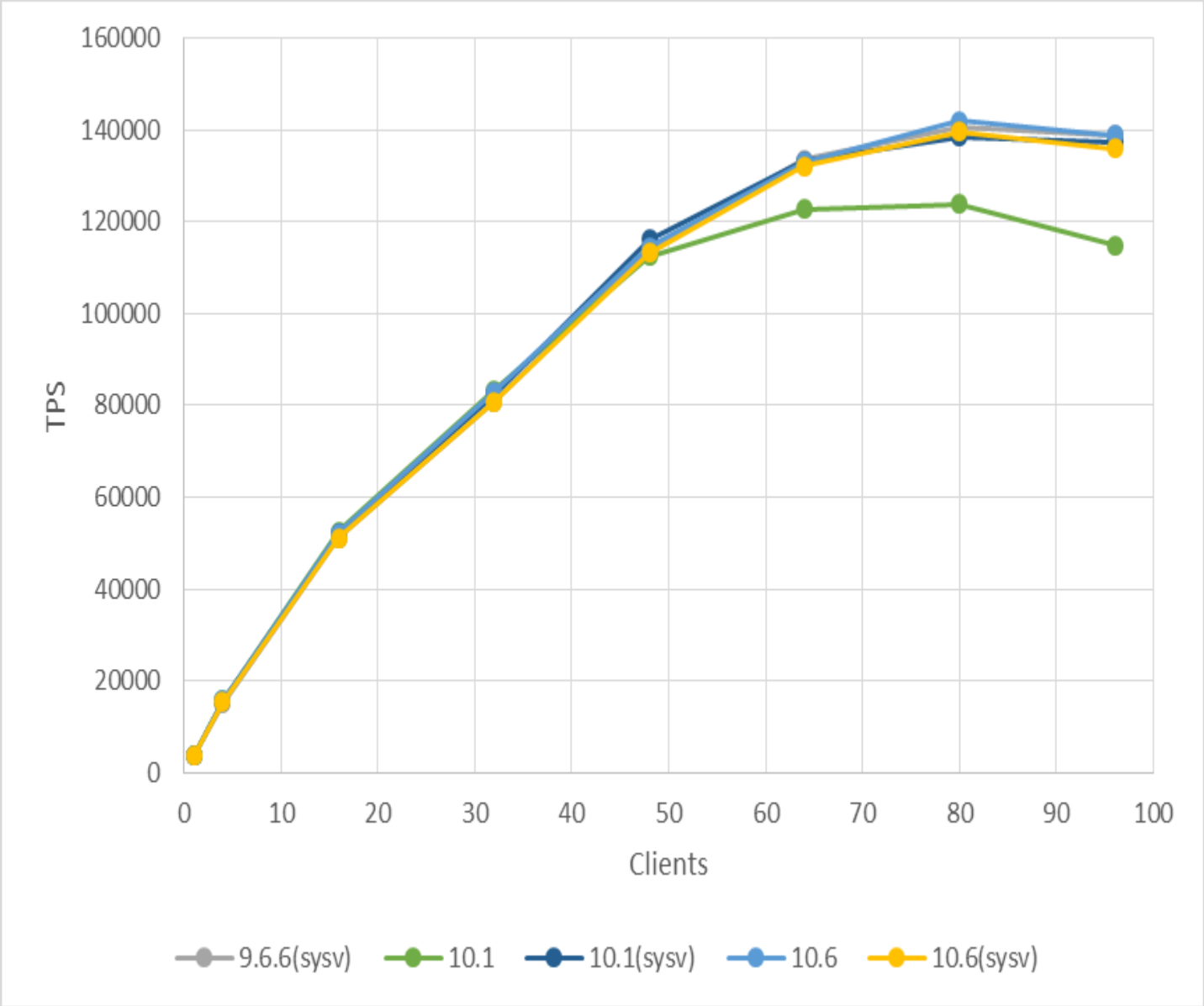


図 4.8 セマフォとバージョンの組み合わせごとのパフォーマンス比較

ご覧の通り、POSIX セマフォを用いた 10.1 のみ性能が低く、それ以外のビルドは同程度の性能を発揮しています。これにより、10.0当初のデフォルトセマフォの変更による性能低下は後のマイナーバージョンで修正されていることが確認できました。

4.11. まとめ

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。

参照系については、例年利用しているクエリでは性能向上が見られましたが、原因を突き詰めてみると、それは集約計算を含むことが原因でした。集約計算を外したクエリにおいては新旧での性能差は見られませんでした。

更新系については、11 の性能が 10 から向上している様子が確認できました。また、性能向上の大きな要因が clog の更新のグループ化によることを証明できました。

2017年度検証の残課題については、10 の更新性能低下の原因が、当時の予想通りデフォルトセマフォの変更によるものであることを証明することができました。また、デフォルトセマフォが変更されたバージョン 10 においても 10.5 以降のマイナーバージョンで性能修正がなされていることが確認できました。

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。

昨年度の成果報告として、Windows版PostgreSQLを用いた性能検証は実施したところ多くの反響をいただいたため、今年度のPGEConsでも継続して、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 11とWindows版PostgreSQL 11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

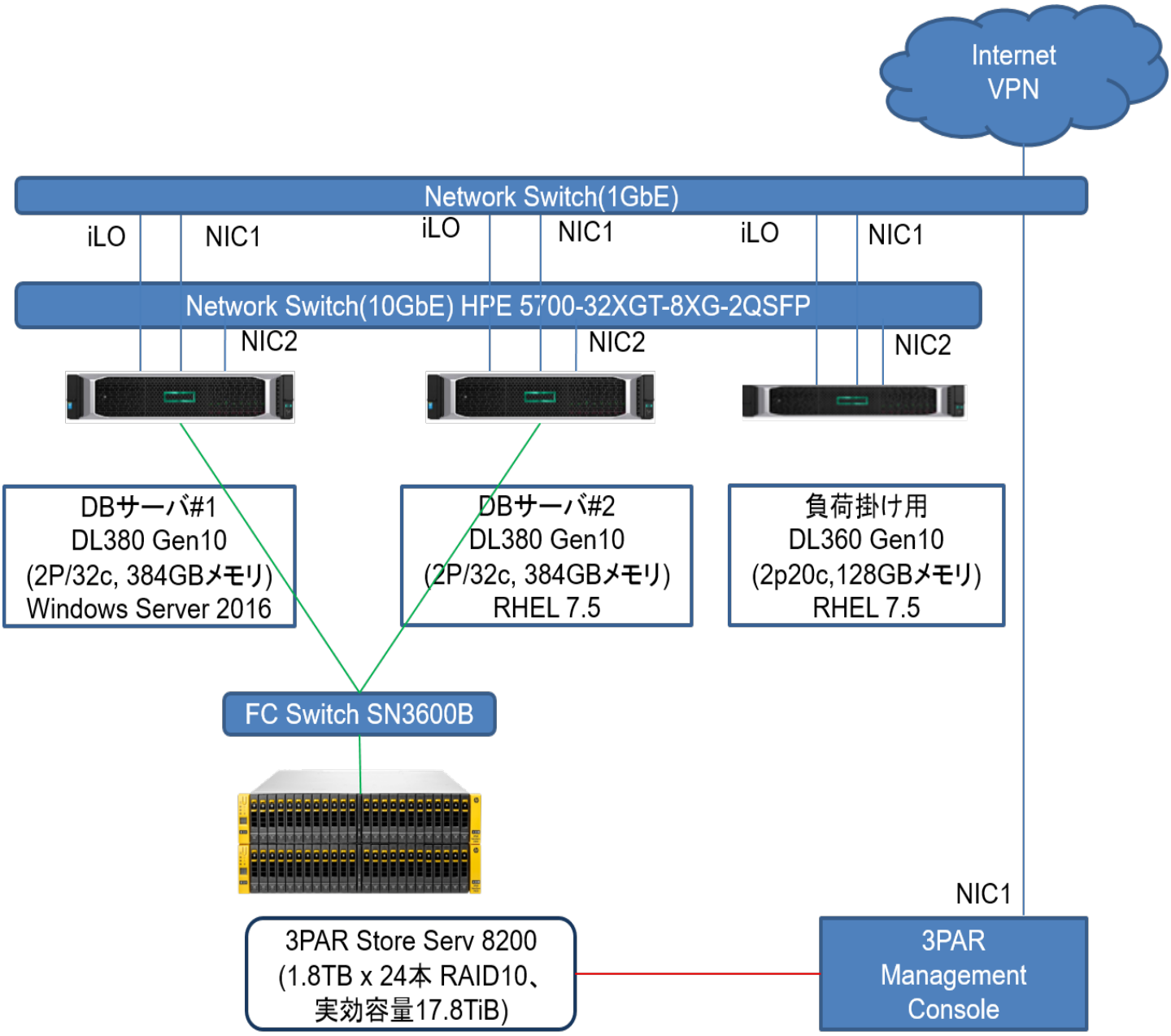


図 5.1 検証ハードウェア構成

本検証では上図構成のうち「DBサーバ#1」と「負荷掛け用」を用いました。

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 Server 2012 R2 Standard Edition
PostgreSQL	11.1 (EDB installer版/BigSQL installer版)

表 5.2 負荷掛け用

OS	Red Hat Red Hat Enterprise Linux 7.5
PostgreSQL	11.1 (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。検証モデルの内容については、「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。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. 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。

5.3.2.1. 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=2000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. 仮測定

本測定の前に、インストールしたPostgreSQLが正常に動作することを確認するため、仮測定を行いました。ここで、参照系の性能結果に異常が見られたため本検証ではEDB installer版ではなく、BigSQL installer版を用いることとしました。

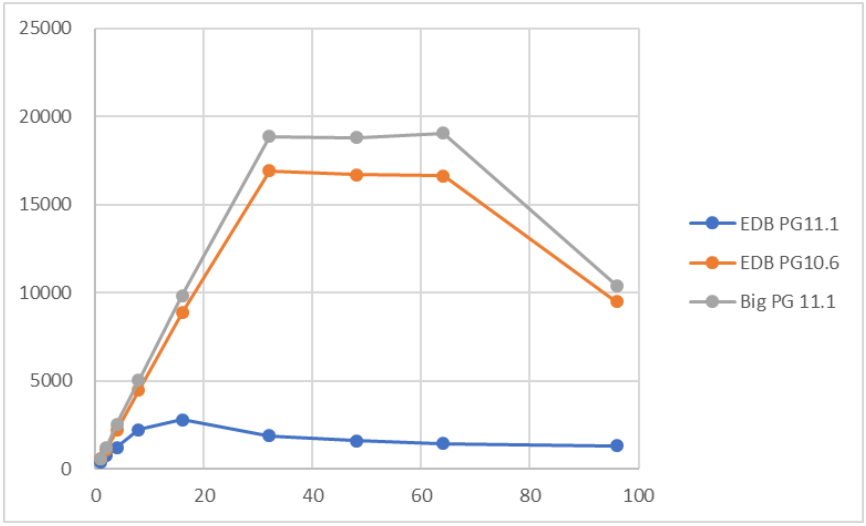
5.3.3.1. 参照系仮測定

走行時間は60秒を指定しました。同時実行数として1, 2, 4, 8, 16, 32, 48, 64, 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;
```

仮測定時の参照系検証結果(tps)を以下に示します。



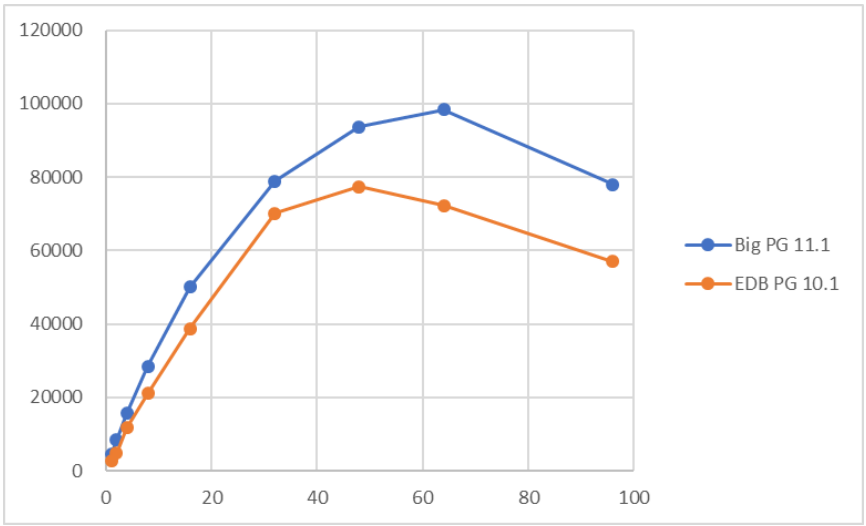
上記の通り、EDB installer版PostgreSQL11.1では思ったような負荷がかかっておらず、バージョン間の性能観測を行うことは困難と判断しました。代替としてBigSQL installer版のPostgreSQL11.1ではEDB installer版PostgreSQL11.1と同等（もしくはそれ以上）の性能が出ることが確認できたため、以降の検証のPostgreSQLバージョンとしては、BigSQL installer版PostgreSQL 11を利用することとしました。

5.3.3.2. 更新系仮測定

走行時間は60秒を指定しました。同時実行数として1, 2, 4, 8, 16, 32, 48, 64, 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;
```

仮測定時の更新系検証結果(tps)を以下に示します。



上記の仮測定の結果を確認し、BigSQL installer版を用いて、参照系・更新系ともにLinux版PostgreSQLとの比較を行う方針としました。

5.3.3.3. バグレポート

EDB版インストーラでインストールしたバイナリを用いた参照系測定を実施したときに、スループットが低くなる事象については、PostgreSQL開発コミュニティに、バグレポートとして報告しました。https://www.postgresql.org/message-id/15610-f5b6c1e204660ace%40postgresql.org このバグレポートに関する議論はあり、翌月のマイナーバージョンアップに伴い新しい版のEDB版インストーラもリリースされましたが、その版に対応したインストーラがこの問題に対応していたのかはPGECons側では未確認です。

追記：本資料発表後、EDB installer版PostgreSQL11.3では性能に問題ないことがEnterpriseDB社により確認済み。

5.3.4. 本測定

本測定では測定時間を300秒として、仮測定と同様の測定を実行しました。

5.3.5. 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 = 16384 ... Windows版ではshared_buffersに大きな値を設定しても意味がないため
work_mem = 1GB
maitenance_work_mem = ... Windows版PostgreSQLでは上限値がLinux版よりも小さいため、20GBではなく6GBを設定した。
checkpoint_timeout = 60min
max_wal_size = 160GB
```

5.3.6. OSリソース情報の収集

Linuxで使われているvmstat/sarといったOSのリソース情報の取得コマンドはWindowsにはありません。代わりにtypeperfコマンドでOSのリソース情報を取得します。リソースがどのように利用されているかを記録するためtypeperfコマンドを参照系・更新系それぞれで以下の通り実行しました。今回の検証では、typeperfコマンドを以下のように実行することで、CPU使用率、メモリ使用量、ディスクへの秒あたりの書き込み量を取得するようにしました。

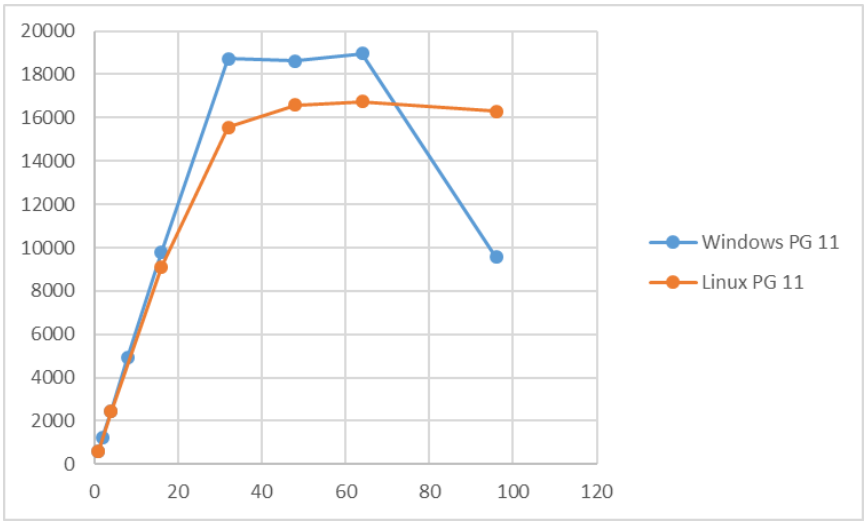
```
typeperf -sc 900 -si 10  "\processor information(_Total)\% Processor Time" "\Memory\Committed
Bytes" "\LogicalDisk(C:)\Avg. Disk sec/Read" "\LogicalDisk(C:)\Avg. Disk sec/Write" >
select-res.txt
typeperf -sc 2300 -si 10  "\processor information(_Total)\% Processor Time" "\Memory\Committed
Bytes" "\LogicalDisk(C:)\Avg. Disk sec/Read" "\LogicalDisk(C:)\Avg. Disk sec/Write" >
update-res.txt
```

5.4. 検証結果

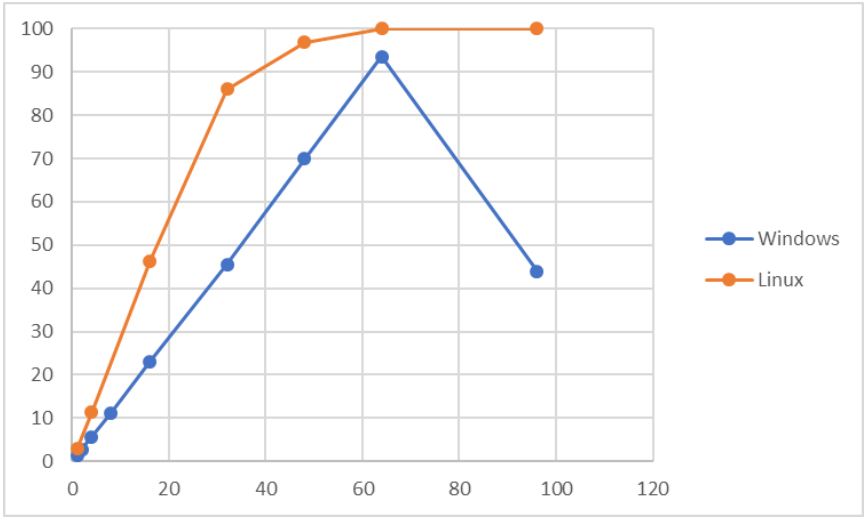
本節では、Windows版PostgreSQL検証結果を示します。

5.4.1. 参照系

参照系検証結果(tps)を以下に示します。



参照系検証時の平均CPU使用率を以下に示します。



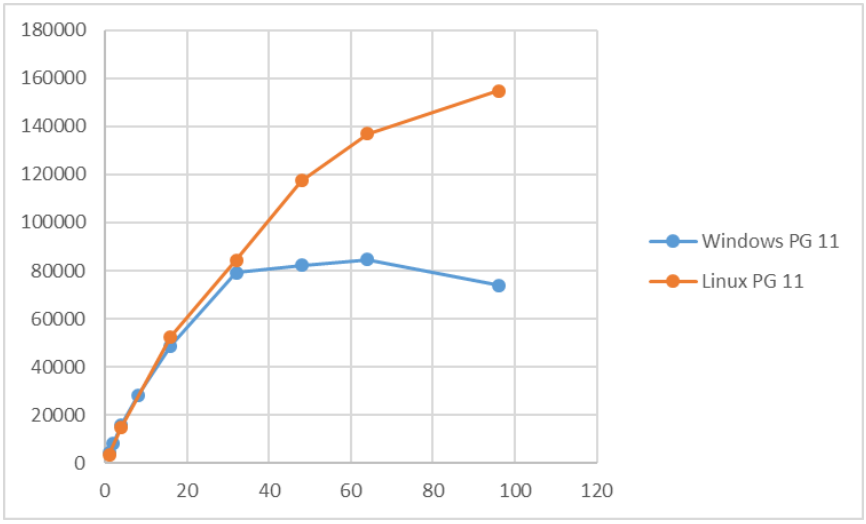
参照系については同時接続数32までは、同時接続数を上げることでほぼ線形にtpsが増加していきます。同時接続数32～64の区間では、tpsは変わらず、レイテンシが増加するという傾向になりました。この区間では、Linux版PostgreSQL 11よりも高いtpsとなっています。この区間で、Windows版PostgreSQLのtpsがLinux版PostgreSQLよりも高いtpsになっている原因については、現時点では不明であり、今後の課題と考えています。Linux版PostgreSQLとWindows版PostgreSQLのCPUの使用状況を比較してみると、Linux版PostgreSQLでもCPUは十分に使われており、CPUの使用状況が原因ではないと考えられます。今後、どういった情報を収集してLinux版PostgreSQLとの挙動を比較すれば良いかも含めて検討が必要です。

同時実行数が64を超えると急激にtpsが低下し、またCPU使用率も大きく減少しています。このとき、Linux版PostgreSQL 11よりもtpsは悪化し、またCPU使用率についてもLinux版PostgreSQLと比較すると大きく低下するという結果になっています。

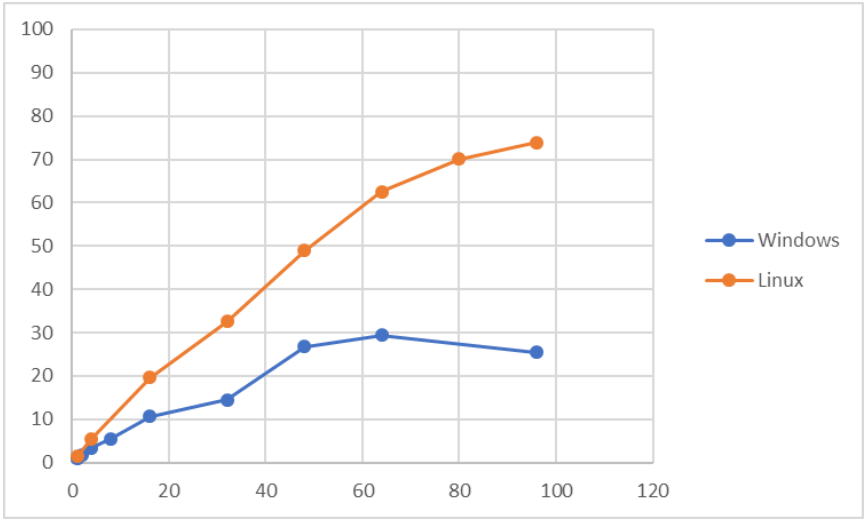
過負荷状態では、CPUネックとは別の原因がネックになっていると考えられます。プロセス間の排他待ちの影響がLinux版と比較すると大きくなっていると推測していますが、今年度の測定では、それを示す情報の取得はできませんでした。次年度の測定では、同時接続数が過大になったときのtpsおよびCPU使用率の低下原因の追求が必要と考えます。

5.4.2. 更新系

更新系検証結果(tps)を以下に示します。



更新系検証時の平均CPU使用率を以下に示します。



更新系に関しては、同時接続数32でtpsのピーク値となり、以降はほぼtpsは横ばいとなりました。Linux版と比較すると、同時接続数48以降は大きくtpsが小さくなるという結果になっています。更新系の平均CPU使用率を見ると、参照系とは異なり、同時接続数64をピークとして最大でも30%程度の使用率となっています。また、Linux版更新系の平均使用率と比較すると、全般にCPU使用率が低い傾向にあります。このことから、Windows環境では更新時のI/O負荷がLinuxと比較して高いことが推測されます。

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。

参照系の場合、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、CPU数を超えない範囲の同時接続数であれば、Linux版と同等あるいは上回る性能となりました。ただし、CPU数を超えた同時接続数になると急速に性能が劣化します。この原因が検証に使用したバイナリ(EDB版/BigSQL版)の違いによるものか、PostgreSQLバージョンによるものか、その違いを調査することも今後の課題と考えます。

更新系の場合、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、同時接続数が32を超える場合に、Linux版と比較してWindows版のtpsが大きく劣るという結果となりました。この原因については、I/O性能の差が原因と想定していますが、それを示すOSリソース値の取得が今後の課題と考えます。また、今回はデータ格納用のディスクと更新ログ格納用のディスクを同居した構成にしていましたが、データ格納用のディスクと更新ログ格納用のディスクを分離した場合の測定も、次年度以降検討したいと考えます。

6. JITコンパイル　性能検証

6.1. 検証概要

PostgreSQL 11に実行時(Just-In-Time; JIT)コンパイルが実装されました。JITコンパイルの目的はクエリの実行速度を向上させることです。クエリプランの一部分に対してJITコンパイルを実行することで実現されます。本検証では、JITコンパイルが有効に働くクエリ種を調査するとともに、それによって実行時間がどれだけ削減されるか検証します。

検証用データベースに Star Schema Benchmark(SSB) を採用し、以下の検証と考察を実施しました。

- 演算、関数ごとのJITコンパイルの有効性の検証
- データ型によるJITコンパイルの有効性の差異の検証
- SSBクエリを用いたJITコンパイルの有効性の検証

6.2. JITコンパイルとは

6.2.1. JITコンパイル

JITコンパイルとは、ソフトウェアの実行時にコードのコンパイルを行い実行速度の向上を図る手法です。反復される処理を、インタプリタ形式からコンパイルしてバイナリを実行する形式に変更することで高速化を実現しています。

6.2.2. PostgreSQLでのJITコンパイル

PostgreSQLでJITコンパイルを利用可能にするためには、以下のパッケージが必要になります。

表 6.1 追加パッケージ

パッケージ名	最低限必要なバージョン
llvmライブラリ	3.9
clang（clang互換のコンパイラ）	3.9
C++コンパイラ	なし

上記パッケージがインストールされた状態でPostgreSQLのビルド時に"--with-llvm"を指定することにより、LLVMを使ったJITコンパイルが組み込まれます。具体的な方法は、SRA OSS社が公開している PostgreSQL11検証報告書 をご参照ください。

高速化の対象として、WHERE句やターゲットリスト、集約関数などといった式の評価やディスク上のタプルをインメモリ形式に変換するタプル変形、そして関数のインライン化があります。インライン化はクエリ中の関数呼び出しや演算子をバイナリに変換し、関数呼び出しのオーバーヘッドを削減します。JITコンパイルが有効に働く状況は、CPU使用率の高い状態が長時間継続するようなクエリを実行している状態とされています。つまり、式表現が複雑であり、参照する行数が多い解析系のクエリで有効であるといえます。逆に、短時間で完了するクエリではJITのオーバーヘッドの方が大きくなってしまいます。（下記の実行計画例を参照）

JITコンパイルを有効にするには、postgresql.confファイルのjitパラメータ（後述）をonに設定する必要があります。さらに、JITコンパイルが実行されるか否かは、クエリのコストに依存します。コストはjit_above_costパラメータ（後述）で与えられる閾値と比較され、コストの方が大きければJITコンパイルが実行されます。JITコンパイルが実行される場合、jit_inline_above_costパラメータ（後述）の閾値およびjit_optimize_above_costパラメータ（後述）の閾値と比較され、コストの方が大きければ関数のインライン化または最適化がそれぞれ追加で実行されます。

JITコンパイルの実行状況は、実行計画から確認できます。

```
                                        QUERY PLAN
-------------------------------------------------------------------------------------------------------------
Aggregate  (cost=16.27..16.29 rows=1 width=8) (actual time=6.049..6.049 rows=1 loops=1)
  ->  Seq Scan on pg_class  (cost=0.00..15.42 rows=342 width=4) (actual time=0.019..0.052 rows=356 loops=1)
Planning Time: 0.133 ms
JIT:
  Functions: 3
  Options: Inlining false, Optimization false, Expressions true, Deforming true
  Timing: Generation 1.259 ms, Inlining 0.000 ms, Optimization 0.797 ms, Emission 5.048 ms, Total 7.104 ms
Execution Time: 7.416 ms
```

「JIT:」以下に実行したJITコンパイルの詳細が表示されます。

表 6.2 実行計画に表示されるJITコンパイルの実行状態

項目	説明
Functions	JITコンパイルによって処理された関数の数
Option	実行されたJIT処理（bool型で表示） Inlining : インライン化 Optimization : 最適化 Expressions : 式の評価 Deforming : タプルの変換
Time	一連の処理で要した時間 Generation : JITコンパイルの所要時間 Inlining : インライン化所要時間 Optimization : 最適化所要時間 Emission : JITコード出力所要時間 Total : 上記の合計時間

6.2.3. JITコンパイルに関するパラメータ

JITコンパイル時に関連する主なパラメータには以下があります。

表 6.3 JITコンパイルに関連する主なパラメータ

パラメータ	デフォルト値	説明
jit	off	JITコンパイルを使用するかどうか
jit_above_cost	100000	JITコンパイルを使用するコストの閾値
jit_inline_above_cost	500000	関数のインライン化を使用するコストの閾値
jit_optimize_above_cost	500000	積極的な最適化を使用するコストの閾値

6.3. 検証構成

6.3.1. 検証ハードウェア環境

検証環境のハードウェア構成の概略を示します。

表 6.4 検証用DBサーバ構成

機器	項目	仕様
検証サーバ（PRIMERGY RX2540M4）	CPU	インテル Xeon Gold 6140@2.30GHz 18コア x2 合計 36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	HDD 700GB

6.3.2. 検証ソフトウェア環境

検証環境のソフトウェア環境を示します。

表 6.5 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 7.5
Database	PostgreSQL 11.1

6.4. 検証方法

6.4.1. 検証項目

JITコンパイル検証では以下の3点を検証します。

6.4.1.1. 検証1

検証1では、単一の演算、関数を含むクエリを用いた際のJITコンパイルの有効性を確認します。使用する演算、関数は以下とします。

表 6.6 検証1で使用する演算と関数

演算の種類	使用する演算
四則演算	乗算(×)、減算(-)
集約関数	sum, avg, count, max
統計処理用の集約関数	stddev, var_samp

四則演算は、パラレルクエリ検証でも用いられたSSBのクエリに含まれるものを抜粋しました。集約関数については、JITコンパイルは集約に効果があるとの記載 [1] から検証に使用しています。また、集約関数の分け方はPostgreSQL文書に基づいたものです。これにより、単純な演算を含むクエリでのJITコンパイルによってどれほど実行時間が削減されるかを明らかにします。

[1] PostgreSQL 11 文書 第32章 実行時コンパイル(JIT) 32.1.1. JITにより高速化される処理

6.4.1.2. 検証2

検証2では、データ型によってJITコンパイルの効果が変化するか検証します。使用するデータ型は、integer型、numeric型、real型、double precision型です。これにより、データ型の違いでJITコンパイルによる実行時間の削減量がどれほど変わるのかを明らかにします。

6.4.1.3. 検証3

検証3では、SSBのクエリを用いて、JITコンパイルを用いた場合と用いない場合でどれほどの差が出るか検証します。これにより、OLAP系の実用的なクエリを実行した際にJITコンパイルによって実行時間がどれほど削減される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、Star Schema Benchmark(SSB)に準拠します。データの生成プログラムは、こちらのGitHubリポジトリ から入手したものを使用しました。また、テーブル定義は こちら から入手したものを使用しました。

```
# zipファイルを解凍する

$ bzip2 -d ssb.tar.bz2
$ tar xf ssb.tar
$
$ make

# データ生成(-sはスケールファクタ(後述)を指定するオプション、-Tはテーブルの種類を指定するオプション)

$ ./dbgen -s 1 -T c #customerテーブル
$ ./dbgen -s 1 -T d #dateテーブル
$ ./dbgen -s 1 -T l #lineorderテーブル
$ ./dbgen -s 1 -T p #partテーブル
$ ./dbgen -s 1 -T s #supplierテーブル

# テーブルを定義する (コマンドは省略)

# テーブルにデータを挿入する(以下のコマンドを各テーブルごとに実行する)

$ sed -e 's/|$//' customer.tbl | psql -d jit-test -c "copy customer from STDIN ( delimiter '|');"
```

SSBはその名前の通り、スター・スキーマ型のデータモデルを採用しています。スタースキーマ型のデータモデルは、1つのファクトテーブルと複数のディメンジョンテーブルで構成されています。ファクトテーブルとは分析対象となる実績値が格納される明細データであり、データ件数は億単位(本検証では約6億件)になります。ディメンジョンテーブルとはビジネス上の分析の軸になるマスターデータであり、データ件数は少ないのが特徴です。SSBではlineorderテーブルがファクトテーブルであり、商品の購入履歴が時系列に蓄積されます。そしてSSBのディメンジョンテーブルにはcustomerテーブル、supplierテーブル、partテーブル、dateテーブルがあり、顧客やサプライヤなどの識別キーや名称などが格納されます。

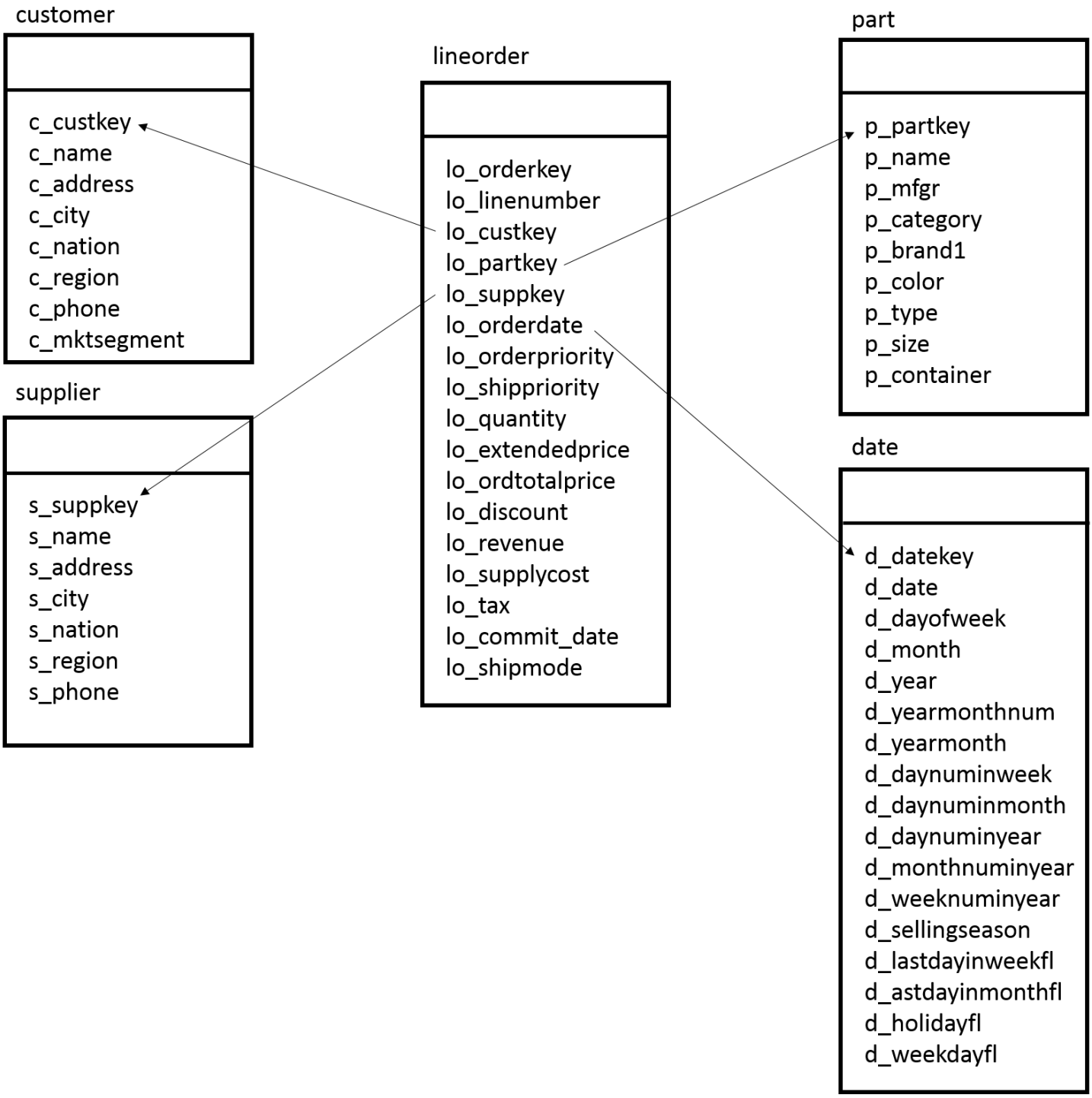


図 6.1 Star Schema Benchmarkのテーブル構成

なお、Scale Factor=100GBとしています。Scale Factor(SF)とは、ベンチマーク全体のテーブル規模を示しています。SSBで使用されるテーブルの行数はSFによって以下のように計算できます。dateテーブルは7年分の日付情報を持つため、常に一定の大きさになっています。

表 6.7 検証に使用するテーブルの規模

relname	rows
customer	SF × 30000
date	2556
lineorder	SF × 6000000
part	200000 × (1+log2(SF)) (小数点切り捨て)
supplier	SF × 10000

本検証における各テーブルのサイズは以下の通りです。

```
     relname          |    rows    |    bytes
----------------------+------------+-------------
customer              | 3000000    | 374030336
date                  | 2315       | 311296
lineorder             | 599985000  | 63557517312
part                  | 1399570    | 164265984
supplier              | 999604     | 114376704
```

6.4.3. 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。本検証ではJITコンパイルの性能を確認するため、パラレルクエリとして実行されないようにmax_parallel_workers_per_gatherパラメータの値を0としています。

表 6.8 デフォルトから変更したパラメータ

パラメータ	設定値
shared_buffers	64GB
work_mem	2GB
maintenance_work_mem	32GB
max_parallel_workers_per_gather	0
wal_level	replica
max_wal_size	10GB
min_wal_size	800MB

6.4.4. 測定手順

測定に際しては、SQLの実行前にpg_prewarmを利用して、テーブルのデータをディスクから一度読込み、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'
);
```

6.4.4.1. 検証1測定方法

以下のSQLを、psqlを用いて実行します。

- 乗算

```
select lo_extendedprice * lo_discount
from lineorder;
```

- 減算

```
select lo_revenue - lo_supplycost
from lineorder;
```

- sum

```
select sum(lo_extendedprice)
from lineorder;
```

- avg

```
select avg(lo_extendedprice)
from lineorder;
```

- count

```
select count(lo_extendedprice)
from lineorder;
```

- count(*)

```
select count(*)
from lineorder;
```

- max

```
select max(lo_extendedprice)
from lineorder;
```

- stddev

```
select stddev(lo_extendedprice)
from lineorder;
```

- var_samp

```
select var_samp(lo_extendedprice)
from lineorder;
```

6.4.4.2. 検証2測定方法

以下のSQLを、psqlを使用して実行します。本検証で用いるテーブルlineorder_n, lineorder_r, lineorder_dは、lineorderテーブルのlo_extendedpriceカラムのデータ型をinteger型からnumeric型、real型、double precision型にそれぞれ変更したものです。

- numeric型

```
select sum(lo_extendedprice)
from lineorder_n;
```

- real型

```
select sum(lo_extendedprice)
from lineorder_r;
```

- double precision型

```
select sum(lo_extendedprice)
from lineorder_d;
```

6.4.4.3. 検証3測定方法

以下のSSBのクエリを、psqlを使用して実行します。

6.4.4.3.1. No.1

No.1のクエリは、ある年の商品出荷において割引を廃止した場合に発生する収益の増加量を求めます。

- No.1-1

```
select  sum(lo_extendedprice * lo_discount) as revenue
from lineorder, date
where lo_orderdate = d_date_key and
      d_year = 1993 and
      lo_discount between 1 and 3 and
      lo_quantity < 25;
```

- No.1-2

```
select  sum(lo_extendedprice * lo_discount) as revenue
from lineorder, date
where lo_orderdate = d_date_key and
      d_yearmonthnum = 199401 and
      lo_discount between 4 and 6 and
      lo_quantity between 26 and 35;
```

- No.1-3

```
select  sum(lo_extendedprice * lo_discount) as revenue
from lineorder, date
where lo_orderdate = d_date_key and
      d_weeknuminyear = 6 and
      d_year = 1994 and
      lo_discount between 5 and 7 and
      lo_quantity between 26 and 35;
```

6.4.4.3.2. No.2

No.2のクエリは、納入元の地域、商品カテゴリの観点から発注年と商品分類でグループ化した収益を求めます。

- No.2-1

```
select sum(lo_revenue), d_year, p_brand1
from lineorder, date, part, supplier
where lo_orderdate = d_datekey and
      lo_partkey = p_partkey and
      lo_suppkey = s_suppkey and
      p_category = 'MFGR#12' and
      s_region = 'AMERICA'
group by d_year, p_brand1
order by d_year, p_brand1;
```

- No.2-2

```
select sum(lo_revenue), d_year, p_brand1
from lineorder, date, part, supplier
where lo_orderdate = d_datekey and
      lo_partkey = p_partkey and
      lo_suppkey = s_suppkey and
      p_brand1 between 'MFGR#2221' and 'MFGR#2228' and
      s_region = 'ASIA'
group by d_year, p_brand1
order by d_year, p_brand1;
```

- No.2-3

```
select sum(lo_revenue), d_year, p_brand1
from lineorder, date, part, supplier
where lo_orderdate = d_datekey and
      lo_partkey = p_partkey and
      lo_suppkey = s_suppkey and
      p_brand1 = 'MFGR#2221' and
      s_region = 'EUROPE'
group by d_year, p_brand1
order by d_year, p_brand1;
```

6.4.4.3.3. No.3

No.3のクエリは、ある特定の期間と地域における顧客の国、納入元の国、発注年でグループ化した収益を求めます。

- No.3-1

```
select c_nation, s_nation, d_year, sum(lo_revenue) as revenue
from customer, lineorder, supplier, date
where lo_custkey = c_custkey and
      lo_suppkey = s_suppkey and
      lo_orderdate = d_datekey and
      c_region = 'ASIA' and
      s_region = 'ASIA' and
      d_year between 1992 and 1997
group by c_nation, s_nation, d_year
order by d_year asc, revenue desc;
```

- No.3-2

```
select c_city, s_city, d_year, sum(lo_revenue) as revenue
from customer, lineorder, supplier, date
where lo_custkey = c_custkey and
      lo_suppkey = s_suppkey and
      lo_orderdate = d_datekey and
      c_nation = 'UNITED STATES' and
      s_nation = 'UNITED STATES' and
      d_year between 1992 and 1997
group by c_city, s_city, d_year
order by d_year asc, revenue desc;
```

- No.3-3

```
select c_city, s_city, d_year, sum(lo_revenue) as revenue
from customer, lineorder, supplier, date
where lo_custkey = c_custkey and
      lo_suppkey = s_suppkey and
      lo_orderdate = d_datekey and
      ( c_city = 'UNITED KI1' or
        c_city = 'UNITED KI5') and
      ( s_city = 'UNITED KI1' or
        s_city = 'UNITED KI5') and
      d_year between 1992 and 1997
group by c_city, s_city, d_year
order by d_year asc, revenue desc;
```

- No.3-4

```
select c_city, s_city, d_year, sum(lo_revenue) as revenue
from customer, lineorder, supplier, date
where lo_custkey = c_custkey and
      lo_suppkey = s_suppkey and
      lo_orderdate = d_datekey and
      ( c_city = 'UNITED KI1' or
        c_city = 'UNITED KI5') and
      ( s_city = 'UNITED KI1' or
        s_city = 'UNITED KI5') and
      d_yearmonth = 'Dec1997'
group by c_city, s_city, d_year
order by d_year asc, revenue desc;
```

6.4.4.3.4. No.4

No.4のクエリは、5つのテーブルすべてを利用し、総計の利益を求めます。

- No.4-1

```
select d_year, c_nation, sum(lo_revenue - lo_supplycost) as profit
from date, customer, supplier, part, lineorder
where lo_custkey = c_custkey and
      lo_suppkey = s_suppkey and
      lo_partkey = p_partkey and
      lo_orderdate = d_datekey and
      c_region = 'AMERICA' and
      s_region = 'AMERICA' and
      (p_mfgr = 'MFGR#1' or
       p_mfgr = 'MFGR#2')
group by d_year, c_nation
order by d_year, c_nation;
```

- No.4-2

```
select d_year, s_nation, p_category, sum(lo_revenue - lo_supplycost) as profit
from date, customer, supplier, part, lineorder
where lo_custkey = c_custkey and
      lo_suppkey = s_suppkey and
      lo_partkey = p_partkey and
      lo_orderdate = d_datekey and
      c_region = 'AMERICA' and
      s_region = 'AMERICA' and
      ( d_year = 1997 or
        d_year = 1998) and
      (p_mfgr = 'MFGR#1' or
       p_mfgr = 'MFGR#2')
group by d_year, s_nation, p_category
order by d_year, s_nation, p_category;
```

- No.4-3

```
select d_year, s_city, p_brand1, sum(lo_revenue - lo_supplycost) as profit
from date, customer, supplier, part, lineorder
where lo_custkey = c_custkey and
      lo_suppkey = s_suppkey and
      lo_partkey = p_partkey and
      lo_orderdate = d_datekey and
      c_region = 'AMERICA' and
      s_nation = 'UNITED STATES' and
      ( d_year = 1997 or
        d_year = 1998) and
      p_category = 'MFGR#14'
group by d_year, s_city, p_brand1
order by d_year, s_city, p_brand1;
```

6.5. 検証結果

6.5.1. 検証1

検証1での結果を以下に示します。

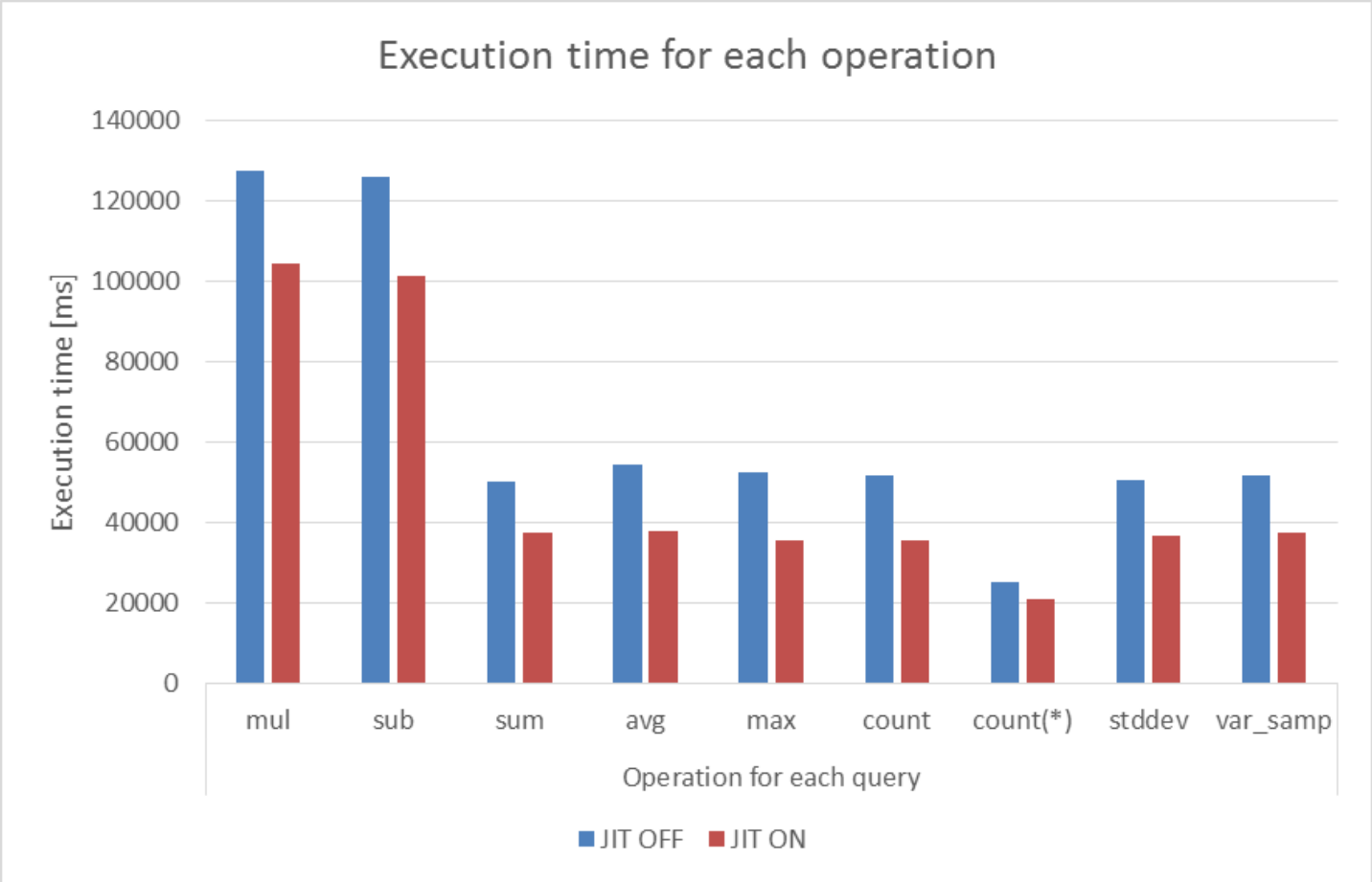


図 6.2 JITコンパイルON/OFFでの実行時間の比較

表 6.9 JITコンパイルON/OFFでの実行時間（秒）

演算の種類	JIT:OFF	JIT:ON	削減率(%)
乗算	127.560	104.274	18.254
減算	125.733	101.229	19.489
sum	50.017	37.535	24.955
avg	54.466	37.946	30.331
max	52.533	35.495	32.433
count	51.519	35.673	30.758
count(*)	25.142	20.983	16.541
stddev	50.681	36.609	27.764
var_samp	51.837	37.588	27.489

削減率は、100×(t_off-t_on)/(t_off)で求められます。t_offはJITコンパイルがOFFの場合の実行時間、t_onはJITコンパイルがONの場合の実行時間です。いずれの演算を含むクエリにおいても約20%～30%実行時間が削減され、JITコンパイルを実行したときの方が実行時間が短くなることがわかります。乗算と減算を含むクエリの実行時間がそのほかの演算を含むクエリの実行時間よりも長いのは、lineorderテーブルのもつ約6億行のレコードごとに演算を実行した結果、約6億行の演算結果が返却されるためと考えられます。

6.5.2. 検証2

検証2での結果を以下に示します。

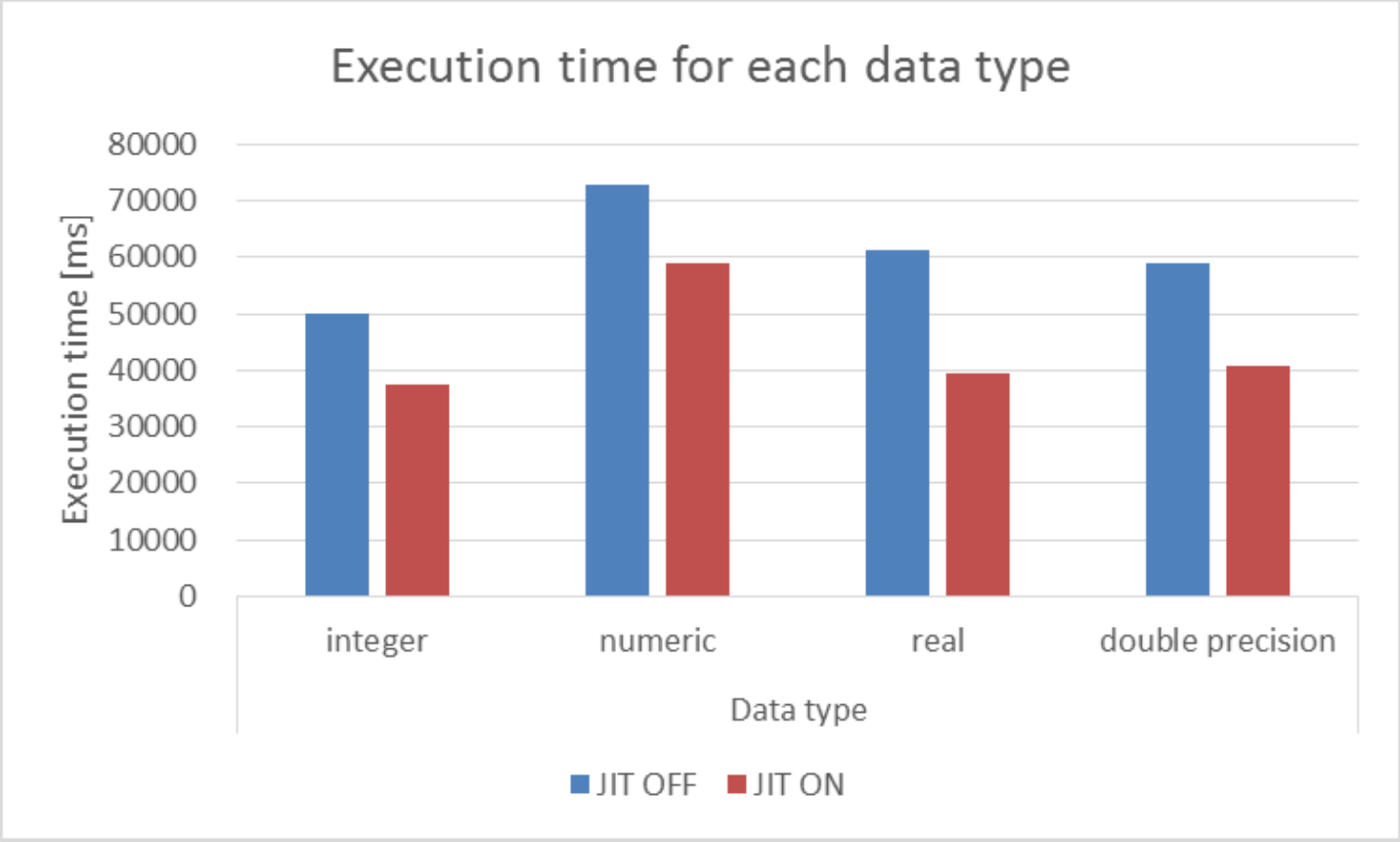


図 6.3 JITコンパイルON/OFFでの実行時間の比較

表 6.10 JITコンパイルON/OFFでの実行時間（秒）

カラムのデータ型	JIT:OFF	JIT:ON	削減率(%)
integer型	50.017	37.535	24.955
numeric型	72.863	58.913	19.134
real型	61.197	39.329	35.733
double precision型	58.886	40.946	30.466

データ型によってJITコンパイルの効果に差が出ることがわかります。特に、numeric型は他のデータ型に比べて実行時間が長くなっています。これはnumeric型が可変長サイズであるためと考えられます。

6.5.3. 検証3

検証3での結果を以下に示します。

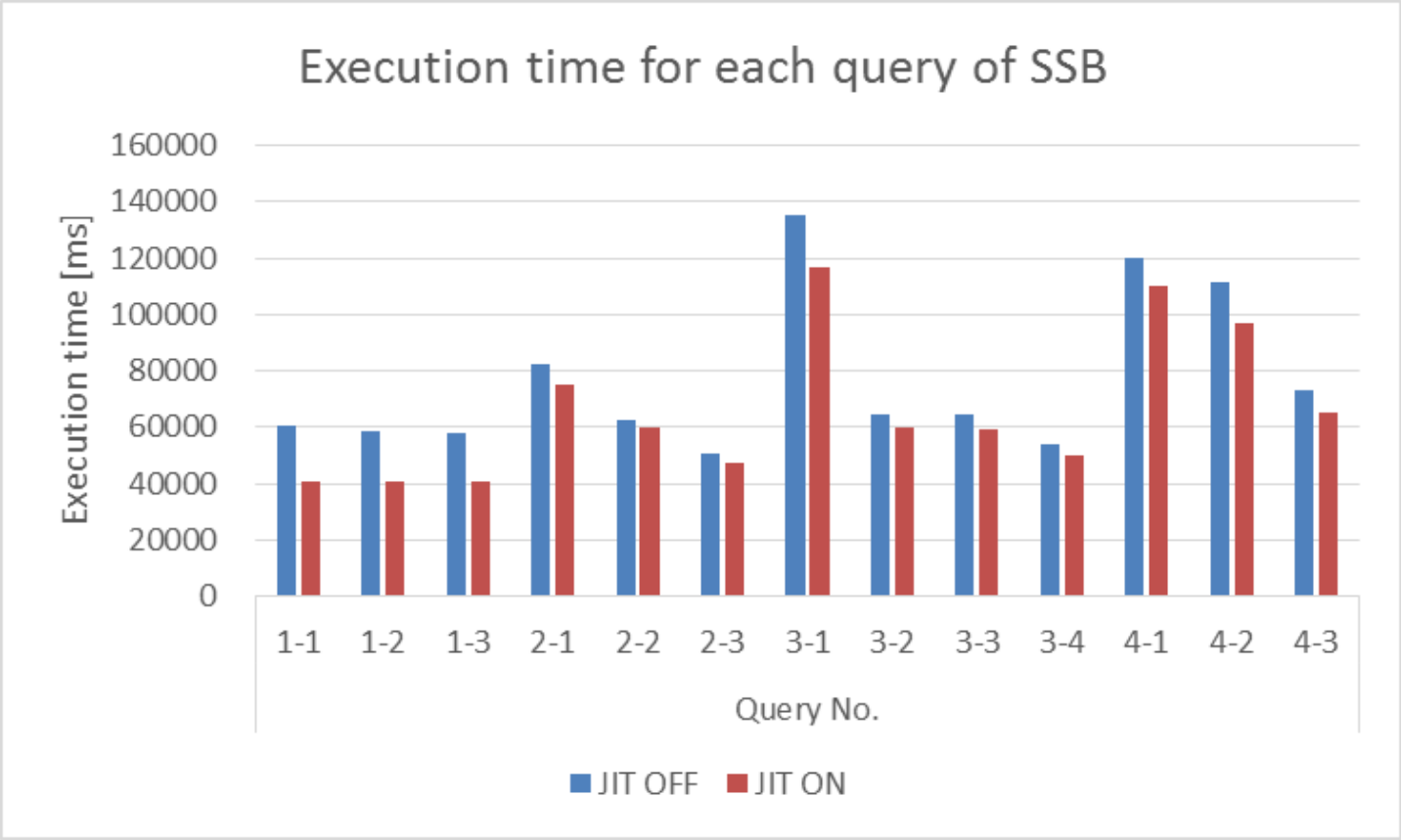


図 6.4 JITコンパイルON/OFFでの実行時間

表 6.11 JITコンパイルON/OFFでの実行時間（秒）

クエリNo.	JIT:OFF	JIT:ON	削減率(%)
1-1	60.434	41.016	32.131
1-2	58.386	40.568	30.517
1-3	57.747	40.737	29.456
2-1	82.116	75.149	8.485
2-2	62.754	60.174	4.112
2-3	50.340	47.039	6.557
3-1	135.593	117.087	13.648
3-2	64.237	59.919	6.721
3-3	64.236	59.318	7.657
3-4	53.743	50.173	6.643
4-1	119.962	109.932	8.361
4-2	111.415	97.129	12.823
4-3	73.247	65.111	11.108

OLAP系の実用的なクエリを実行する際にも、JITコンパイルを実行することで実行時間が短くなることが確認できました。また、クエリによって効果に差が見られました。

6.6. 考察

検証1、検証2、検証3で用いたクエリから実行計画を取得し、JITコンパイルによるオーバヘッド部分と高速化した処理を確認します。さらに、検証3では、削減率の低いクエリがある理由について追加調査しました。

6.6.1. 検証1

検証1においては、乗算を含むクエリとsum関数の含むクエリ2種類の実行計画を確認します。

6.6.1.1. 乗算を含むクエリ

- JITコンパイルがONの場合

```
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
Seq Scan on lineorder  (cost=0.00..15259254.00 rows=600061440 width=4) (actual time=13.632..49875.825 rows=600037902 loops=1)
  Buffers: shared hit=7758486
Planning Time: 0.040 ms
JIT:
  Functions: 2
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 0.309 ms, Inlining 0.957 ms, Optimization 8.485 ms, Emission 4.106 ms, Total 13.857 ms
Execution Time: 66032.955 ms
```

- JITコンパイルがOFFの場合

```
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
Seq Scan on lineorder  (cost=0.00..15259254.00 rows=600061440 width=4) (actual time=0.006..71122.610 rows=600037902 loops=1)
  Buffers: shared hit=7758486
Planning Time: 0.043 ms
Execution Time: 87332.431 ms
```

JITコンパイルがONの場合では、シーケンシャルスキャンノードにおけるactual timeの左側の数値（1行目の結果を受け取るのに要した時間）がJITコンパイル処理の所要時間にほぼ一致していることがわかります。このことから、オーバーヘッド部分を確認できました。また、actual timeからシーケンシャルスキャンの実行にかかる時間が短縮されたことでクエリの実行時間が削減されたこともわかります。

6.6.1.2. sum関数を含むクエリ

- JITコンパイルがONの場合

```
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
Aggregate  (cost=15259254.00..15259254.01 rows=1 width=8) (actual time=70981.559..70981.559 rows=1 loops=1)
  Buffers: shared hit=7758486
  ->  Seq Scan on lineorder  (cost=0.00..13759100.40 rows=600061440 width=4) (actual time=0.004..27183.683 rows=600037902 loops=1)
        Buffers: shared hit=7758486
Planning Time: 0.038 ms
JIT:
  Functions: 3
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 0.565 ms, Inlining 1.459 ms, Optimization 20.451 ms, Emission 12.119 ms, Total 34.594 ms
Execution Time: 70982.215 ms
```

- JITコンパイルがOFFの場合

```
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
Aggregate  (cost=15259254.00..15259254.01 rows=1 width=8) (actual time=90474.587..90474.587 rows=1 loops=1)
  Buffers: shared hit=7758486
  ->  Seq Scan on lineorder  (cost=0.00..13759100.40 rows=600061440 width=4) (actual time=0.004..28574.686 rows=600037902 loops=1)
        Buffers: shared hit=7758486
Planning Time: 0.047 ms
Execution Time: 90474.615 ms
```

乗算を含むクエリとは異なり、オーバーヘッドになっている部分を特定することはできませんでした。しかし、シーケンシャルスキャンにかかる時間はJITコンパイルのON/OFFによらずほぼ同じだったものの、Aggregate処理による出力が開始/完了する時間はJITコンパイルがONの場合の方が早くなっていました。このことから、Aggregate処理におけるJITコンパイルの高速化度合いが、JITコンパイル自身のオーバーヘッドを上回ったと考えられます。

6.6.2. 検証2

検証2においては、整数型であるinteger型とnumeric型を設定した際の実行計画を取得しました。integer型の実行計画は検証1の考察（sum関数を含むクエリ）で示しているため、ここではnumeric型の実行計画のみを示します。

- JITコンパイルがONの場合

```
                                                      QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------
Aggregate  (cost=15530386.20..15530386.21 rows=1 width=32) (actual time=86556.379..86556.380 rows=1 loops=1)
  Buffers: shared hit=8029987
  ->  Seq Scan on lineorder_n  (cost=0.00..14030306.36 rows=600031936 width=7) (actual time=0.005..28851.551 rows=600037902 loops=1)
        Buffers: shared hit=8029987
Planning Time: 0.051 ms
JIT:
  Functions: 3
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 0.557 ms, Inlining 2.549 ms, Optimization 20.533 ms, Emission 12.185 ms, Total 35.824 ms
Execution Time: 86556.987 ms
```

- JITコンパイルがOFFの場合

```
                                                      QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------
Aggregate  (cost=15530386.20..15530386.21 rows=1 width=32) (actual time=107847.607..107847.607 rows=1 loops=1)
  Buffers: shared hit=8029987
  ->  Seq Scan on lineorder_n  (cost=0.00..14030306.36 rows=600031936 width=7) (actual time=0.004..28785.690 rows=600037902 loops=1)
        Buffers: shared hit=8029987
Planning Time: 0.048 ms
Execution Time: 107847.637 ms
```

integer型のときの実行計画と同様に、オーバーヘッドになっている部分を特定することはできませんでしたが、シーケンシャルスキャンにかかる時間はJITコンパイルのON/OFFによらずほぼ同じであり、Aggregate処理による出力が開始/完了する時間はJITコンパイルがONの場合の方が早くなっていました。integer型とnumeric型の間での差は、Aggregate処理における高速化度合いの差にあることがわかります。

6.6.3. 検証3

検証3においては、削減率が最も高かったNo.1-1のクエリと、最も低かったNo.2-2のクエリから実行計画を取得しました。

6.6.3.1. No.1-1

- JITコンパイルがONの場合

```
                                                      QUERY PLAN
----------------------------------------------------------------------------------------------------------------------------------------
Aggregate  (cost=18719494.34..18719494.35 rows=1 width=8) (actual time=44864.530..44864.530 rows=1 loops=1)
  Buffers: shared hit=7758524
  ->  Hash Join  (cost=74.19..18663739.86 rows=11150896 width=8) (actual time=112.756..44343.859 rows=11913972 loops=1)
        Hash Cond: (lineorder.lo_orderdate = date.d_datekey)
        Buffers: shared hit=7758524
        ->  Seq Scan on lineorder  (cost=0.00..18259561.20 rows=78025470 width=12) (actual time=112.507..39017.385 rows=78550619 loops=1)
              Filter: ((lo_discount >= 1) AND (lo_discount <= 3) AND (lo_quantity < 25))
              Rows Removed by Filter: 521487283
              Buffers: shared hit=7758486
        ->  Hash  (cost=69.66..69.66 rows=362 width=4) (actual time=0.225..0.226 rows=365 loops=1)
              Buckets: 1024  Batches: 1  Memory Usage: 21kB
              Buffers: shared hit=38
              ->  Seq Scan on date  (cost=0.00..69.66 rows=362 width=4) (actual time=0.043..0.188 rows=365 loops=1)
                    Filter: (d_year = 1993)
                    Rows Removed by Filter: 2191
                    Buffers: shared hit=38
Planning Time: 0.101 ms
JIT:
  Functions: 18
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 1.511 ms, Inlining 1.941 ms, Optimization 72.381 ms, Emission 38.003 ms, Total 113.835 ms
Execution Time: 44866.132 ms
```

- JITコンパイルがOFFの場合

```
                                                                  QUERY PLAN
----------------------------------------------------------------------------------------------------------------------------------------
Aggregate  (cost=18719494.34..18719494.35 rows=1 width=8) (actual time=64377.092..64377.093 rows=1 loops=1)
  Buffers: shared hit=7758524
  ->  Hash Join  (cost=74.19..18663739.86 rows=11150896 width=8) (actual time=0.242..63766.864 rows=11913972 loops=1)
        Hash Cond: (lineorder.lo_orderdate = date.d_datekey)
        Buffers: shared hit=7758524
        ->  Seq Scan on lineorder  (cost=0.00..18259561.20 rows=78025470 width=12) (actual time=0.006..58183.140 rows=78550619 loops=1)
              Filter: ((lo_discount >= 1) AND (lo_discount <= 3) AND (lo_quantity < 25))
              Rows Removed by Filter: 521487283
              Buffers: shared hit=7758486
        ->  Hash  (cost=69.66..69.66 rows=362 width=4) (actual time=0.232..0.232 rows=365 loops=1)
              Buckets: 1024  Batches: 1  Memory Usage: 21kB
              Buffers: shared hit=38
              ->  Seq Scan on date  (cost=0.00..69.66 rows=362 width=4) (actual time=0.033..0.202 rows=365 loops=1)
                    Filter: (d_year = 1993)
                    Rows Removed by Filter: 2191
                    Buffers: shared hit=38
Planning Time: 0.102 ms
Execution Time: 64377.128 ms
```

No.1-1のクエリでは、lineorderテーブルにおけるシーケンシャルスキャン処理部分にJITコンパイルによるオーバーヘッドが発生していますが、高速化されていることもわかります。また、JITコンパイルによるオーバーヘッドはある1つの処理（No.1-1のクエリではlineorderテーブルにおけるシーケンシャルスキャン処理）にのみ発生していることもわかります。

6.6.3.2. No.2-2

- JITコンパイルがONの場合

```
                                                                           QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------
------------------
GroupAggregate  (cost=16129050.94..16131822.99 rows=6713 width=21) (actual time=88508.083..88720.456 rows=56 loops=1)
  Group Key: date.d_year, part.p_brand1
  Buffers: shared hit=7792538
  ->  Sort  (cost=16129050.94..16129727.17 rows=270492 width=17) (actual time=88505.310..88627.316 rows=961749 loops=1)
        Sort Key: date.d_year, part.p_brand1
        Sort Method: quicksort  Memory: 99713kB
        Buffers: shared hit=7792538
        ->  Hash Join  (cost=70125.25..16104645.50 rows=270492 width=17) (actual time=1050.155..87659.761 rows=961749 loops=1)
              Hash Cond: (lineorder.lo_orderdate = date.d_datekey)
              Buffers: shared hit=7792538
              ->  Hash Join  (cost=70030.26..16100831.24 rows=270492 width=17) (actual time=750.644..87139.416 rows=961749 loops=1)
                    Hash Cond: (lineorder.lo_suppkey = supplier.s_suppkey)
                    Buffers: shared hit=7792500
                    ->  Hash Join  (cost=41091.26..16064069.52 rows=1364746 width=21) (actual time=622.302..84999.357 rows=4813031 loops=1)
                          Hash Cond: (lineorder.lo_partkey = part.p_partkey)
                          Buffers: shared hit=7778538
                          ->  Seq Scan on lineorder  (cost=0.00..13759100.40 rows=600061440 width=16) (actual time=0.014..30166.398 rows=60
0037902 loops=1)
                                Buffers: shared hit=7758486
                          ->  Hash  (cost=41051.46..41051.46 rows=3184 width=13) (actual time=622.161..622.161 rows=11270 loops=1)
                                Buckets: 16384 (originally 4096)  Batches: 1 (originally 1)  Memory Usage: 657kB
                                Buffers: shared hit=20052
                                ->  Seq Scan on part  (cost=0.00..41051.46 rows=3184 width=13) (actual time=0.051..620.122 rows=11270 loops
=1)
                                      Filter: (((p_brand1)::text >= 'MFGR#2221'::text) AND ((p_brand1)::text <= 'MFGR#2228'::text))
                                      Rows Removed by Filter: 1388730
                                      Buffers: shared hit=20052
                    ->  Hash  (cost=26461.59..26461.59 rows=198193 width=4) (actual time=127.785..127.785 rows=199889 loops=1)
                          Buckets: 262144  Batches: 1  Memory Usage: 9076kB
                          Buffers: shared hit=13962
                          ->  Seq Scan on supplier  (cost=0.00..26461.59 rows=198193 width=4) (actual time=0.016..102.187 rows=199889 loops
=1)
                                Filter: ((s_region)::text = 'ASIA'::text)
                                Rows Removed by Filter: 800111
                                Buffers: shared hit=13962
              ->  Hash  (cost=63.33..63.33 rows=2533 width=8) (actual time=299.477..299.477 rows=2556 loops=1)
                    Buckets: 4096  Batches: 1  Memory Usage: 132kB
                    Buffers: shared hit=38
                    ->  Seq Scan on date  (cost=0.00..63.33 rows=2533 width=8) (actual time=298.936..299.241 rows=2556 loops=1)
                          Buffers: shared hit=38
Planning Time: 0.190 ms
JIT:
  Functions: 46
  Options: Inlining true, Optimization true, Expressions true, Deforming true
  Timing: Generation 2.996 ms, Inlining 3.770 ms, Optimization 194.606 ms, Emission 100.255 ms, Total 301.627 ms
Execution Time: 88729.503 ms
```

● JITコンパイルがOFFの場合

```
                                                                    QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------------------------
GroupAggregate  (cost=16129050.94..16131822.99 rows=6713 width=21) (actual time=94027.557..94265.541 rows=56 loops=1)
  Group Key: date.d_year, part.p_brand1
  Buffers: shared hit=7792538
  ->  Sort  (cost=16129050.94..16129727.17 rows=270492 width=17) (actual time=94024.314..94150.612 rows=961749 loops=1)
        Sort Key: date.d_year, part.p_brand1
        Sort Method: quicksort  Memory: 99713kB
        Buffers: shared hit=7792538
        ->  Hash Join  (cost=70125.25..16104645.50 rows=270492 width=17) (actual time=779.793..93198.298 rows=961749 loops=1)
              Hash Cond: (lineorder.lo_orderdate = date.d_datekey)
              Buffers: shared hit=7792538
              ->  Hash Join  (cost=70030.26..16100831.24 rows=270492 width=17) (actual time=779.182..92970.569 rows=961749 loops=1)
                    Hash Cond: (lineorder.lo_suppkey = supplier.s_suppkey)
                    Buffers: shared hit=7792500
                    ->  Hash Join  (cost=41091.26..16064069.52 rows=1364746 width=21) (actual time=636.921..90822.476 rows=4813031 loops=1)
                          Hash Cond: (lineorder.lo_partkey = part.p_partkey)
                          Buffers: shared hit=7778538
                          ->  Seq Scan on lineorder  (cost=0.00..13759100.40 rows=600061440 width=16) (actual time=0.006..31115.221 rows=60
0037902 loops=1)
                                Buffers: shared hit=7758486
                          ->  Hash  (cost=41051.46..41051.46 rows=3184 width=13) (actual time=636.823..636.824 rows=11270 loops=1)
                                Buckets: 16384 (originally 4096)  Batches: 1 (originally 1)  Memory Usage: 657kB
                                Buffers: shared hit=20052
                                ->  Seq Scan on part  (cost=0.00..41051.46 rows=3184 width=13) (actual time=0.020..634.619 rows=11270 loops
=1)
                                      Filter: (((p_brand1)::text >= 'MFGR#2221'::text) AND ((p_brand1)::text <= 'MFGR#2228'::text))
                                      Rows Removed by Filter: 1388730
                                      Buffers: shared hit=20052
                    ->  Hash  (cost=26461.59..26461.59 rows=198193 width=4) (actual time=141.636..141.636 rows=199889 loops=1)
                          Buckets: 262144  Batches: 1  Memory Usage: 9076kB
                          Buffers: shared hit=13962
                          ->  Seq Scan on supplier  (cost=0.00..26461.59 rows=198193 width=4) (actual time=0.010..116.053 rows=199889 loops
=1)
                                Filter: ((s_region)::text = 'ASIA'::text)
                                Rows Removed by Filter: 800111
                                Buffers: shared hit=13962
              ->  Hash  (cost=63.33..63.33 rows=2533 width=8) (actual time=0.592..0.592 rows=2556 loops=1)
                    Buckets: 4096  Batches: 1  Memory Usage: 132kB
                    Buffers: shared hit=38
                    ->  Seq Scan on date  (cost=0.00..63.33 rows=2533 width=8) (actual time=0.008..0.335 rows=2556 loops=1)
                          Buffers: shared hit=38
Planning Time: 0.197 ms
Execution Time: 94271.519 ms
```

No.2-2では、No.1-1とは異なり、lineorderテーブルにおけるシーケンシャルスキャン処理部分にJITコンパイルによるオーバヘッドは発生しておりませんが、代わりにdateテーブルにおけるシーケンシャルスキャン処理部分に発生しています。つまり、No.2-2のクエリにおいてもオーバーヘッドは1か所のノードでのみ発生していることがわかります。そして、各テーブルのシーケンシャルスキャンではそれほど高速化していませんが、Hash Joinノードにおける処理が高速化されていることがわかります。

6.6.3.3. 削減率が低い原因の考察

削減率が低いクエリについて条件を変更して再検証しました。具体的には、shared_buffersパラメータを64GBから100GBに変更、固定し、work_memパラメータを1,2,4GB、maintenance_work_memパラメータを8,16,32GBと変化させた際に削減率が変わるかを検証しました。

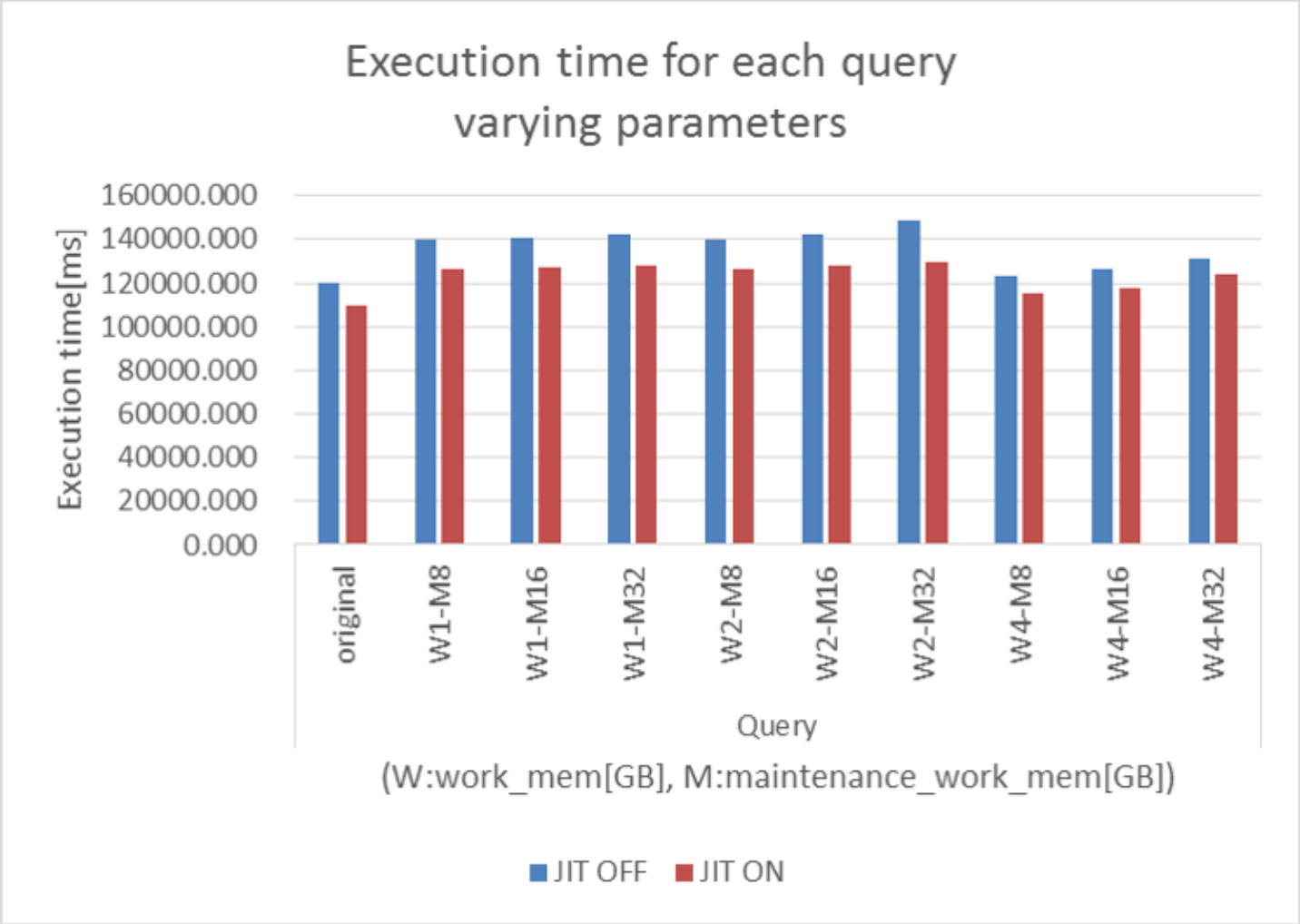


図 6.5 JITコンパイルON/OFFでの実行時間

表 6.12 JITコンパイルON/OFFでの実行時間（秒）

クエリNo.	JIT:OFF	JIT:ON	削減率(%)
original	119.962	109.932	8.361
W1-M8	139.948	126.646	9.505
W1-M16	141.113	127.229	9.839
W1-M32	142.326	127.772	10.225
W2-M8	140.028	126.374	9.751
W2-M16	142.223	127.922	10.055
W2-M32	148.642	129.573	12.829
W4-M8	123.327	115.550	6.306
W4-M16	126.472	117.974	6.719
W4-M32	130.843	123.784	5.395

結果としては、削減率が向上する場合はあったものの、実行時間が本検証よりも長くなってしまいました。ここから、パラメータ設定ではなく、クエリの構造を紐解くことで原因がわかるのではないかと考えられます。

6.7. まとめ

今回の検証では、JITコンパイルが有効に働くクエリ種を調査するとともに、それによって実行時間がどれだけ削減されるか検証しました。結果として、四則演算や集約関数へのJITコンパイルの適用は有効であることを示すことができました。また、テーブルのデータ型によってJITコンパイルによる実行時間の削減率が変化することもわかりました。そして、SSBクエリを用いた検証では、OLAP系の大規模な分析クエリへのJITコンパイルを適用することで実行時間を削減することができることを示すことができました。しかし、クエリの種類によっては削減率が1割以下に留まるものもありました。

今回はマニュアルに記載がある、JITコンパイルの効果が見込まれる演算、関数のうち、自作の関数について検証は行うことができませんでした。実際のシステムにおいて、独自に定義した関数は多く使用されているため、今後検証が必要だと思われます。今回のSSBクエリを用いた検証では、JITコンパイルの効果を見るためにインデックスを

設定していません。実際のシステムではインデックスを設定している場合が多いので、次回の検証ではインデックスを設定した場合においてJITコンパイルの効果を検証する必要があると思われます。また、考察で突き止めることができなかった、削減率が低いクエリの構造を紐解き、削減率が低い原因を解明することも今後行うべきと考えております。JITコンパイルはPostgreSQL 11に実装された新しいものであるため、今後機能の強化などが望まれます。

7. パラレルクエリ 性能検証

7.1. 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以降で継続して強化されている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Star Schema Benchmarkで定義されているクエリを題材として使用します。

7.1.1. Star Schema Benchmarkとは

Star Schema Benchmarkとは、Star Schema Benchmarkの論文 によって公表されている、TPC-HをもとにBIで用いられるものを模したファクトテーブル、ディメンジョンテーブル、クエリが設計されているベンチマークツールです。

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。

Star Schema Benchmarkでは、4種類/13本のクエリが定義されており、そのすべてについて検証を行いました。

7.2. パラレルクエリとは

7.2.1. 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。

7.2.2. PostgreSQLでのパラレルクエリ

2018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、B-Treeインデックス作成の四種類です。

前者三種類については、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 7.1 テーブルサイズによるワーカー数

ブロック数 [1]	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
...	...	...	...

[1] 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。

7.2.3. 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以降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 7.2 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)	デフォルト値(11)	説明
max_worker_processes [2]	8	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 [2]	0	2	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 [2] [3]	—	8	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。
max_parallel_maintenance_workers [4]	—	—	2	メンテナンス一回につき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	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	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	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	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。

[2] (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。

[3] PostgreSQL 10以降では、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。

[4] PostgreSQL 11で追加されました。max_parallel_maintenance_workers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、パラレルクエリ等が同時に実行されている場合、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。

7.3. 検証構成

7.3.1. 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。検証では下記構成の「DBサーバ#2」を使用しました。

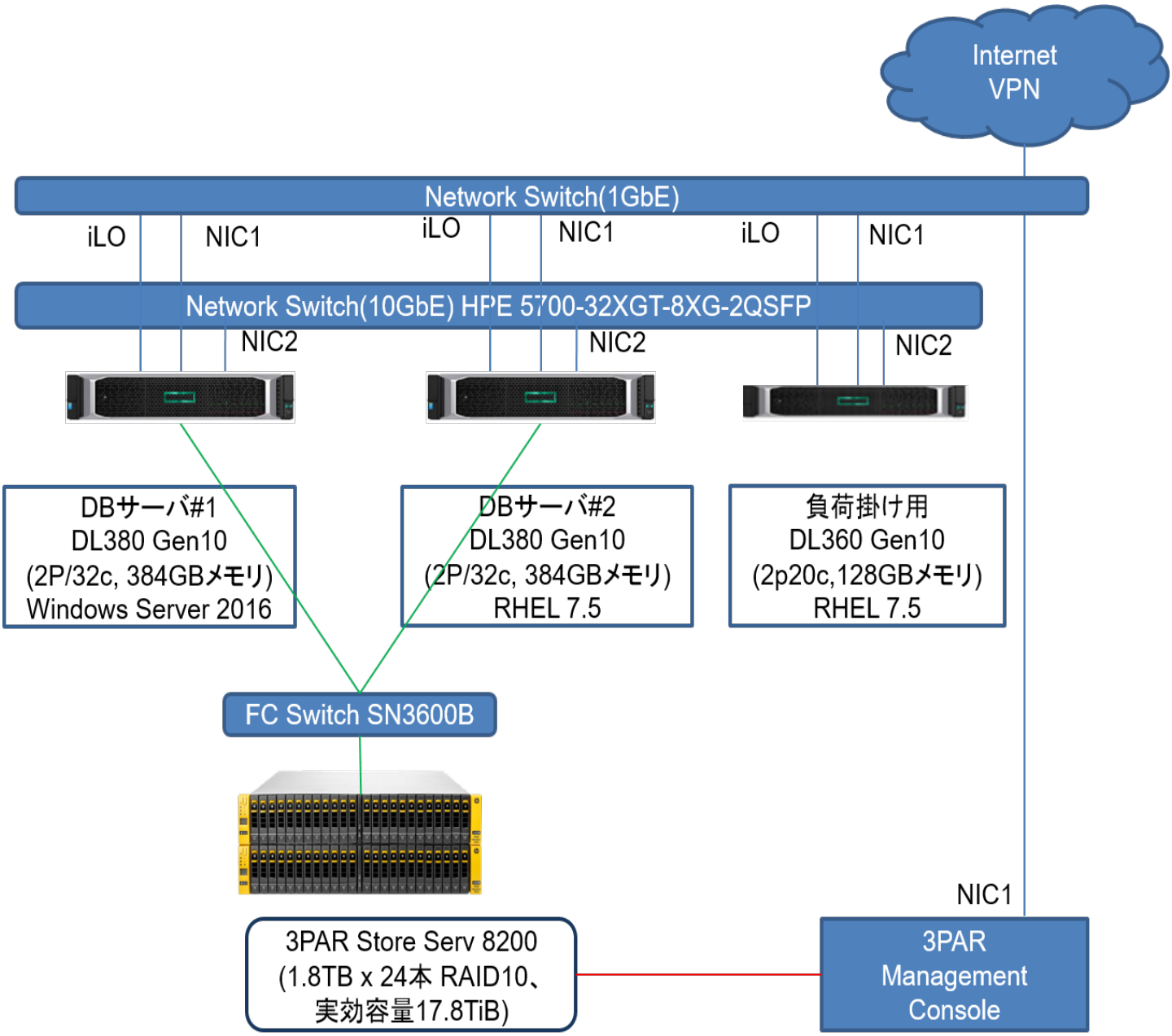


図 7.1 検証ハードウェア構成

表 7.3 検証用DBサーバ構成

機器	項目	仕様
検証サーバ (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
	外付けストレージ装置 (3PAR 8200)	HDD 1.8TB SAS 10K x 24

7.3.2. 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 7.4 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 7.5
Database	PostgreSQL 9.5.15 PostgreSQL 9.6.11 PostgreSQL 10.6 PostgreSQL 11.1

7.4. 検証方法

7.4.1. 検証項目

パラレルクエリ検証では以下の3点を検証します。

7.4.1.1. 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、単にパラレルクエリを有効化した状態で、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。これにより、コメントアウトを外すだけという簡単な作業のみで、クエリに対する応答性能がどのように変動するのかが明らかになります。(特に、バージョン10、バージョン11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、比較対象として、パラレルクエリが実装される前のバージョン9.5での性能も検証することで、9.5～11の各バージョンのデフォルト設定のままで、OLAP用途におけるクエリに対する性能が、どのように変化するのかを明らかにします。

7.4.1.2. 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(バージョン10以降)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。

7.4.1.3. 検証C

検証Cでは、バージョン11において新たに追加された、B-Treeインデックスのパラレル作成について、並列度による処理性能の改善状況を確認します。

max_worker_processes、max_parallel_maintenance_workers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、B-Treeインデックスの作成時間が、どの程度改善する余地があるのかを明らかにします。

7.4.1.4. 対照群

対照群として、バージョン9.6、バージョン10、バージョン11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、検証A、検証Bにおいて、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。

7.4.2. データモデル

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

Scale Factorは100GB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
-----------+-----------+-------------
 part      |   1400000 |   164265984
 supplier  |   1000000 |   114376704
 customer  |   2999990 |   374030336
 date      |      2556 |      311296
 lineorder | 600113000 | 63557664768
```

7.4.3. 使用クエリについて

本検証では、Star Schema Benchmarkで定義される全クエリを使用します。

これは Star Schema Benchmarkの論文 の"3.1 Query Definition"に掲載されているものであり、JITコンパイル　性能検証 で使用したものと同一です。

7.4.4. 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 7.5 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	150GB
work_mem	1GB
maintenance_work_mem	20GB
autovacuum	off
max_worker_processes [5]	8
max_parallel_workers_per_gather [5]	2
max_parallel_workers [5] [6]	8
max_parallel_maintenance_workers [5] [7]	2
log_line_prefix	'%t [%p-%l]: '

[5] (1, 2, 3, 4) 測定により値を変更して実施します。

[6] PostgreSQL 10以降

[7] PostgreSQL 11のみ

7.4.5. 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'
);
```

7.4.5.1. 検証A測定方法

Star Schema Benchmarkで定義される全クエリを、psqlを用いて実行します。

7.4.5.2. 検証B測定方法

Star Schema Benchmarkで定義される全クエリを、psqlを用いて実行します。

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 7.6 検証時設定

パラメータ	gather = 2, workers = 2 [8]	gather = 2, workers = 8 [9]	gather = 4, workers = 4 [8]	gather = 4, workers = 16 [9]	gather = 6, workers = 6 [8]	gather = 6, workers = 24 [9]	gather = 8, workers = 8 [8]	gather = 8, workers = 32 [9]	gather = 10, workers = 10 [8]	gather = 10, workers = 40 [9]
max_parallel_workers_per_gether	2	2	4	4	6	6	8	8	10	10
max_worker_processes	2	8	4	16	6	24	8	32	10	40
max_parallel_workers	2	8	4	16	6	24	8	32	10	40

[8] (1, 2, 3, 4, 5) workersの値がgatherの値と同値であるパターンで、以下worker=gatherと表記します。

[9] (1, 2, 3, 4, 5) workersの値がgatherの値の4倍であるパターンで、以下worker=gather*4と表記します。

7.4.5.3. 検証C測定方法

Star Schema Benchmarkで定義されるlineorderテーブルに対し、以下の列を対象としてB-Treeインデックスを作成します。

表 7.7 複合インデックスの作成対象列

複合数	対象列
1	lo_orderkey
2	lo_orderkey lo_custkey
3	lo_orderkey lo_custkey lo_partkey
4	lo_orderkey lo_custkey lo_partkey lo_suppkey

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、測定後はDROP INDEXによって、インデックス削除しています。

表 7.8 検証時設定

パラメータ	workers = 0	workers = 2	workers = 4	workers = 6	workers = 8	workers = 10
max_parallel_maintenance_workers	0	2	4	6	8	10

7.5. 検証結果

7.5.1. 検証A(パラレルクエリを有効にしたことによる効果)

表 7.9 検証条件

ストレージ	SQL文	保持データ量
外付ストレージ装置	Star Schema Benchmarkで定義される全クエリ	Scale Factor = 100GB

上記の条件で検証を行った結果、バージョン9.6以降では、全クエリでパラレルクエリが動作しました。

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。

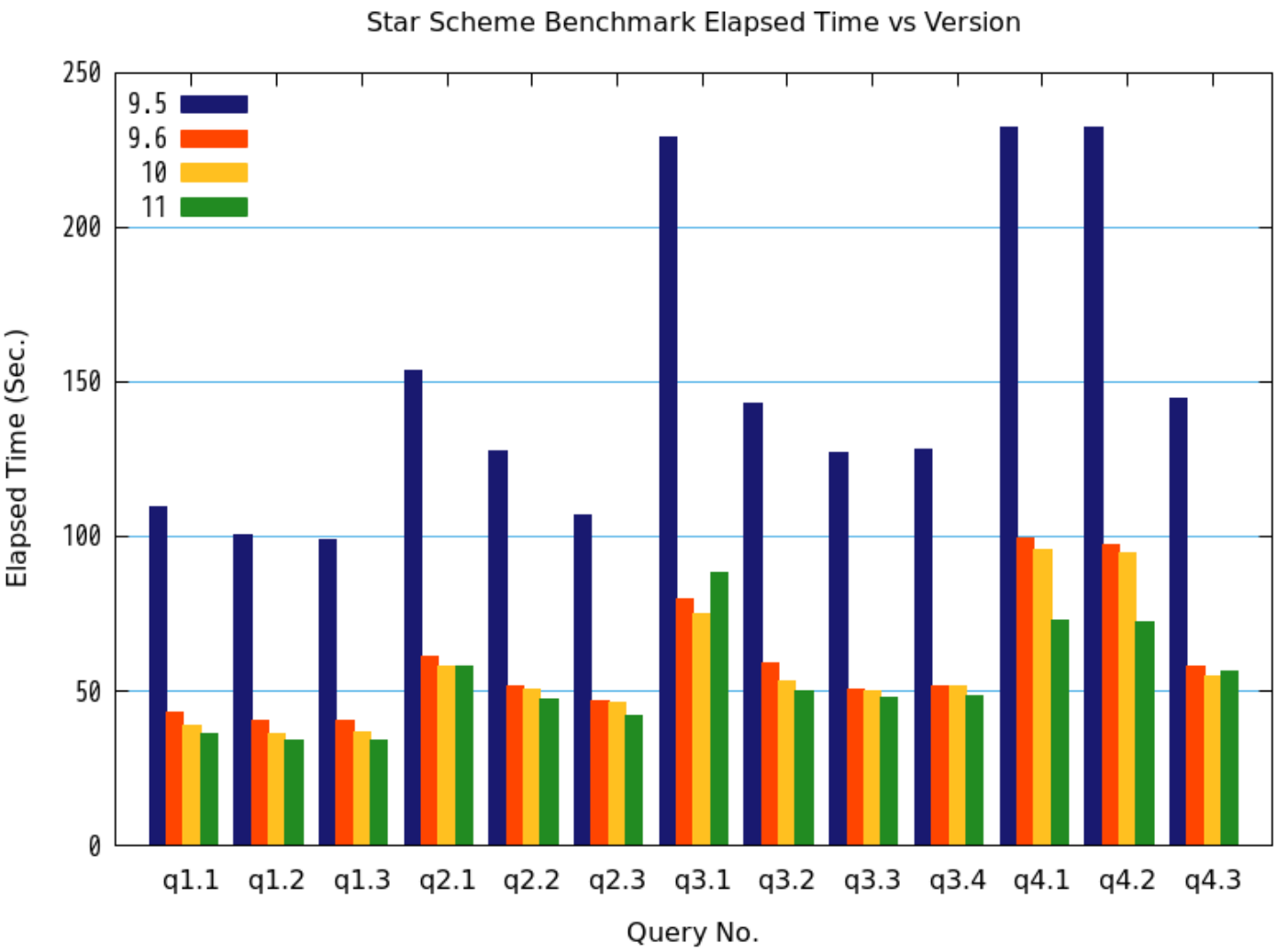


図 7.2 バージョンと処理時間の比較

表 7.10 バージョンと処理時間

クエリ	9.5 処理時間 (秒)	9.6 処理時間 (秒)	9.6 性能向上比 (倍) [10]	10 処理時間 (秒)	10 性能向上比 (倍) [10]	11 処理時間 (秒)	11 性能向上比(倍) [10]
q1.1	109.6738	43.0780	2.5459	38.5922	2.8419	36.0279	3.0441
q1.2	100.5875	40.4387	2.4874	36.4137	2.7624	34.0107	2.9575
q1.3	99.0719	40.2400	2.4620	36.4419	2.7186	34.0967	2.9056
q2.1	153.6779	61.2717	2.5081	57.8688	2.6556	58.2329	2.6390
q2.2	127.4209	51.7503	2.4622	50.4701	2.5247	47.1534	2.7023
q2.3	106.7450	46.5650	2.2924	46.3889	2.3011	41.9794	2.5428
q3.1	229.1734	79.8066	2.8716	75.1554	3.0493	88.3712	2.5933
q3.2	143.0648	58.9300	2.4277	53.4046	2.6789	49.8938	2.8674
q3.3	127.0212	50.5020	2.5152	50.1682	2.5319	48.0444	2.6438
q3.4	128.4246	51.4737	2.4950	51.7527	2.4815	48.2142	2.6636
q4.1	232.5540	99.3121	2.3416	95.9063	2.4248	72.7343	3.1973
q4.2	232.1840	97.3383	2.3853	94.5822	2.4548	72.4182	3.2062
q4.3	144.7047	58.0662	2.4921	55.0490	2.6287	56.3806	2.5666

[10] (1, 2, 3) 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。

検証の結果、クエリの処理時間について、バージョン9.5との比較で、バージョン9.6で2.34倍(q4.1)〜2.87倍(q3.1)、バージョン10で2.42倍(q4.1)〜3.04倍(q3.1)、バー

ジョン11で2.54倍(q2.3)〜3.20倍(q4.2)の高速化が認められました。

7.5.2. 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)

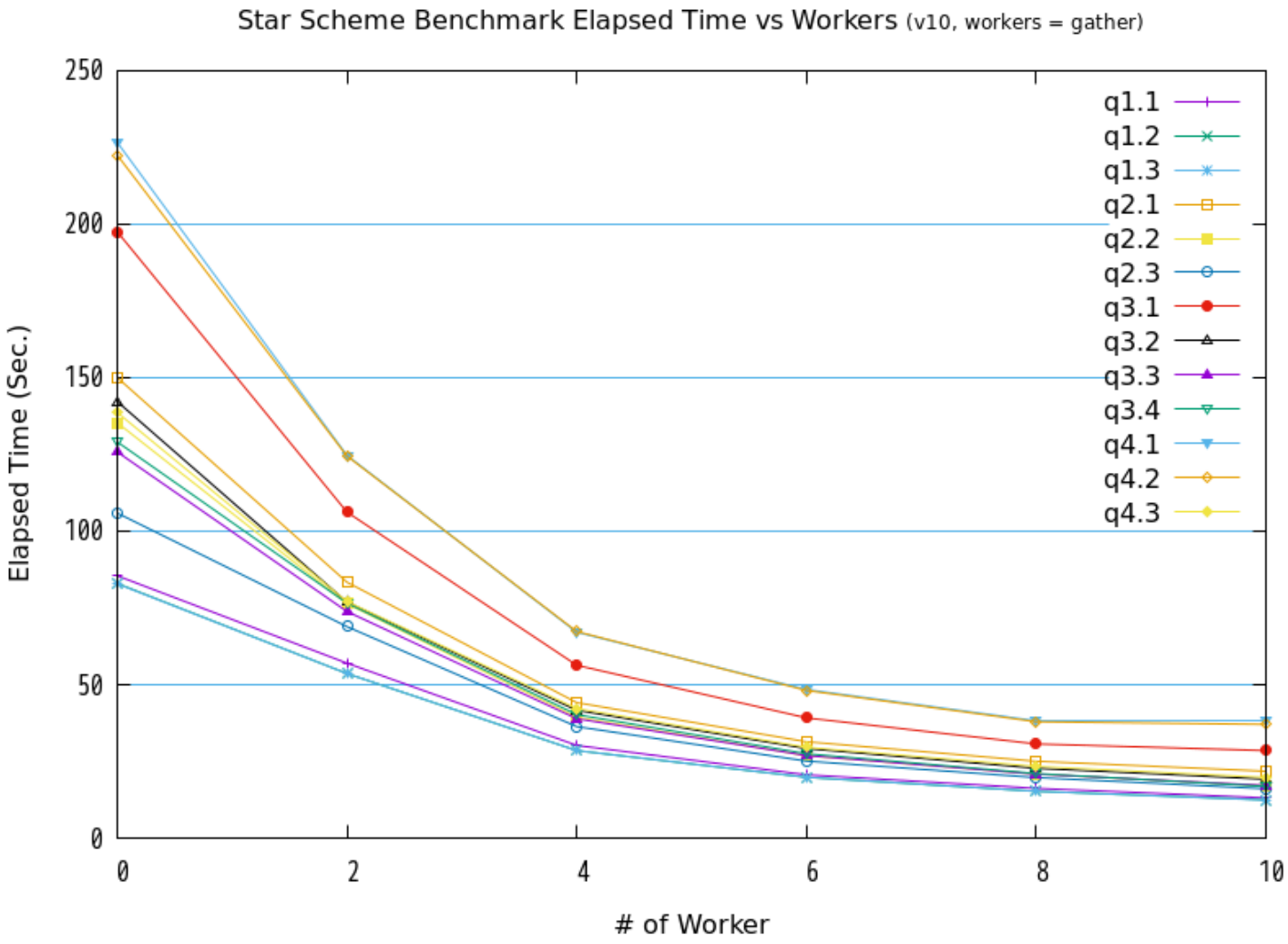


図 7.3 max_parallel_workersの値と処理時間の比較 (バージョン10 / workers = gather)

表 7.11 並列処理数と処理時間 (バージョン10 / workers = gather)

クエリ	0	2	4	6	8	10
q1.1	85.4489	56.9939	30.2153	20.6437	16.2143	13.1179
q1.2	82.9543	53.5944	28.5039	19.8002	15.2736	12.4638
q1.3	82.8403	53.5943	28.5369	19.8255	15.2616	12.3869
q2.1	150.0331	83.3708	44.1458	31.4149	25.0433	21.8037
q2.2	135.3275	76.3389	39.4012	26.8608	20.8826	17.0109
q2.3	105.8896	68.9048	36.328	25.0901	19.726	16.0949
q3.1	197.5059	106.1948	56.3073	39.1798	30.6809	28.5272
q3.2	142.1985	76.8667	41.6347	29.2016	22.7531	19.2159
q3.3	125.7963	73.8228	38.8993	26.8994	20.8419	17.1459
q3.4	129.0627	76.5403	40.2315	27.5024	21.0395	16.859
q4.1	226.6286	124.6846	67.0518	48.4096	38.2259	38.2996
q4.2	222.4871	124.4402	67.2911	48.1134	37.9252	37.0782
q4.3	138.6454	77.0612	42.2118	29.6051	23.37	19.7866

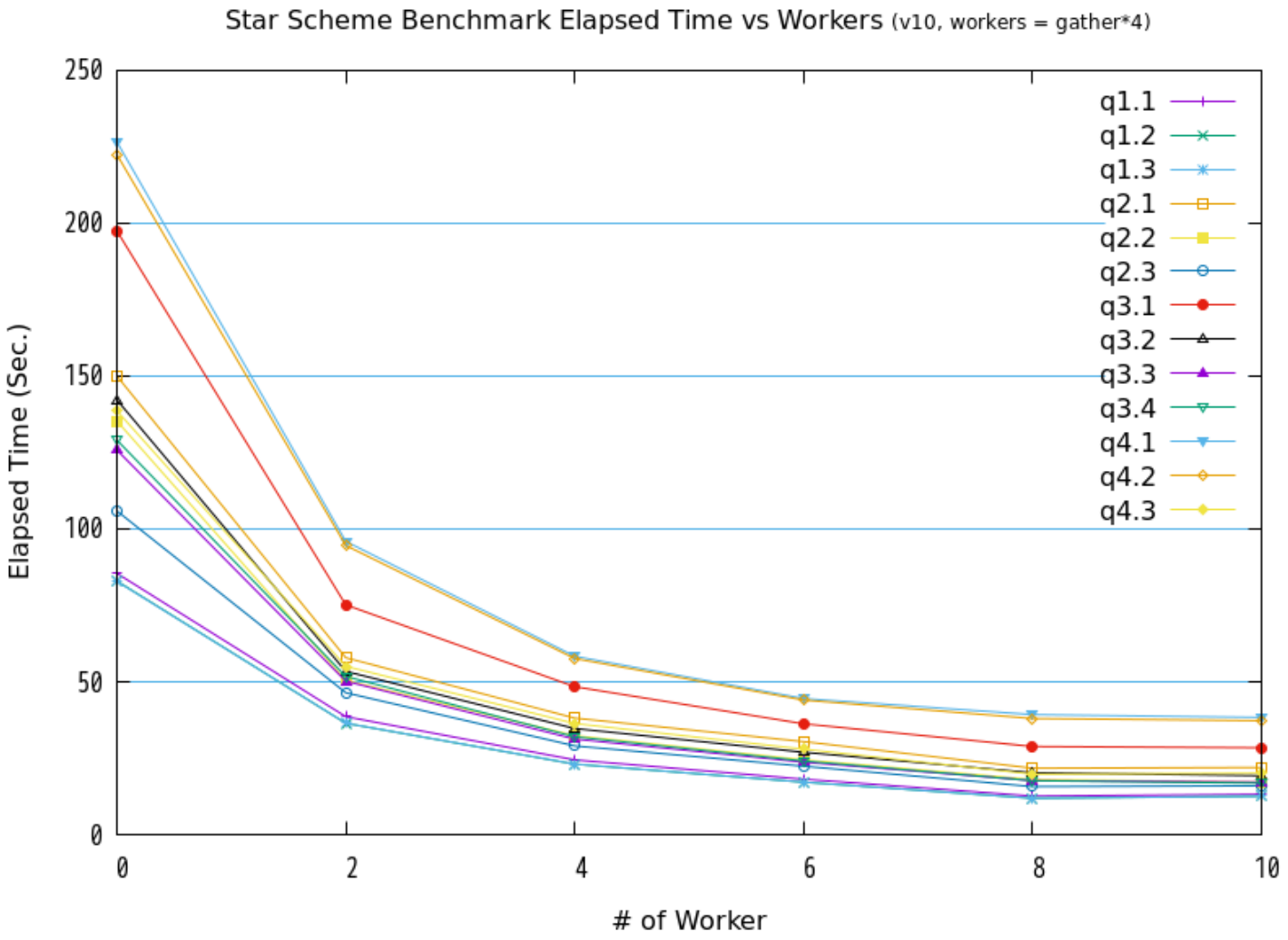


図 7.4 max_parallel_workersの値と処理時間の比較 (バージョン10 / workers = gather*4)

表 7.12 並列処理数と処理時間 (バージョン10 / workers = gather*4)

クエリ	0	2	4	6	8	10
q1.1	85.4489	38.5922	24.4377	18.1406	12.6109	13.1812
q1.2	82.9543	36.4137	23.034	17.1541	11.8751	12.5499
q1.3	82.8403	36.4419	23.0721	17.1504	11.8915	12.5125
q2.1	150.0331	57.8688	38.0912	30.4211	21.7118	21.9554
q2.2	135.3275	50.4701	32.349	24.4605	18.0272	17.1562
q2.3	105.8896	46.3889	29.0172	22.3617	15.7164	15.9769
q3.1	197.5059	75.1554	48.3587	36.2399	28.8156	28.4356
q3.2	142.1985	53.4046	34.6972	26.9351	20.1988	19.3084
q3.3	125.7963	50.1682	31.2504	23.6949	17.654	17.156
q3.4	129.0627	51.7527	31.9982	24.0661	17.6214	16.8232
q4.1	226.6286	95.9063	58.3261	44.51	39.2568	38.3434
q4.2	222.4871	94.5822	57.6029	44.0192	37.9737	37.281
q4.3	138.6454	55.049	36.3353	27.9577	19.7588	20.0958

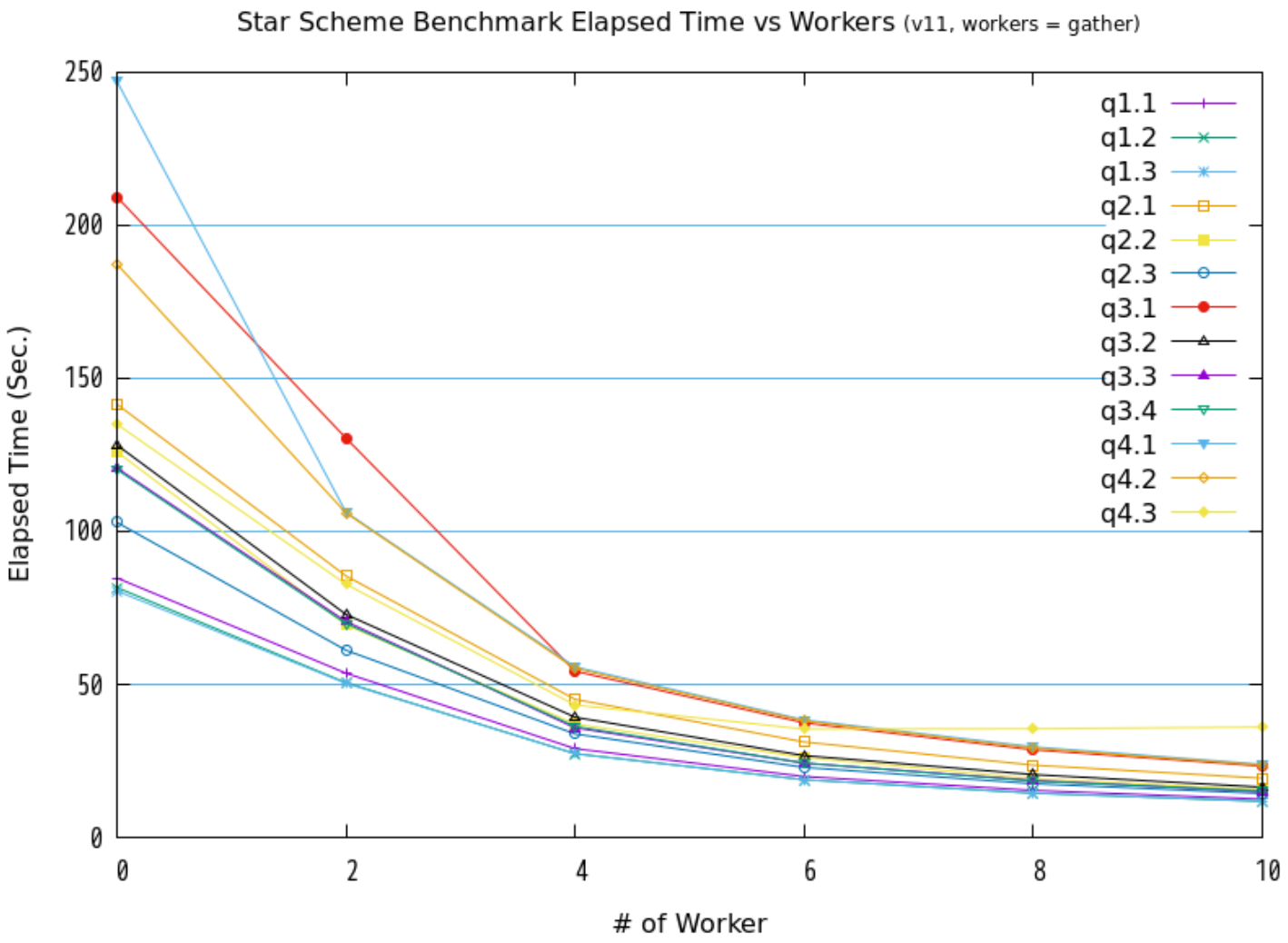


図 7.5 max_parallel_workersの値と処理時間の比較 (バージョン11 / workers = gather)

表 7.13 並列処理数と処理時間 (バージョン11 / workers = gather)

クエリ	0	2	4	6	8	10
q1.1	84.7647	53.6562	29.0181	19.9105	15.3618	12.5352
q1.2	81.6414	50.5466	27.3301	18.8776	14.4657	11.8407
q1.3	80.5189	50.3723	27.4255	18.8421	14.5301	11.8443
q2.1	141.5359	85.4431	45.136	31.1805	23.6265	19.3543
q2.2	126.016	69.4636	37.0795	26.1183	19.2126	15.5833
q2.3	102.9223	61.1581	33.8343	22.9064	17.5493	14.3799
q3.1	209.2735	130.4551	54.3009	37.4342	28.6454	23.2988
q3.2	128.2047	72.8591	39.271	26.7486	20.5554	16.4219
q3.3	120.6891	70.5463	35.8179	24.2731	18.641	14.9511
q3.4	120.2251	69.9018	36.2683	24.3294	18.4329	15.19
q4.1	247.059	106.2362	55.7347	38.3949	29.6088	23.8907
q4.2	187.3207	106.0171	55.2027	38.0867	29.1774	23.5996
q4.3	135.0164	82.7183	43.2652	35.4254	35.5206	36.0839

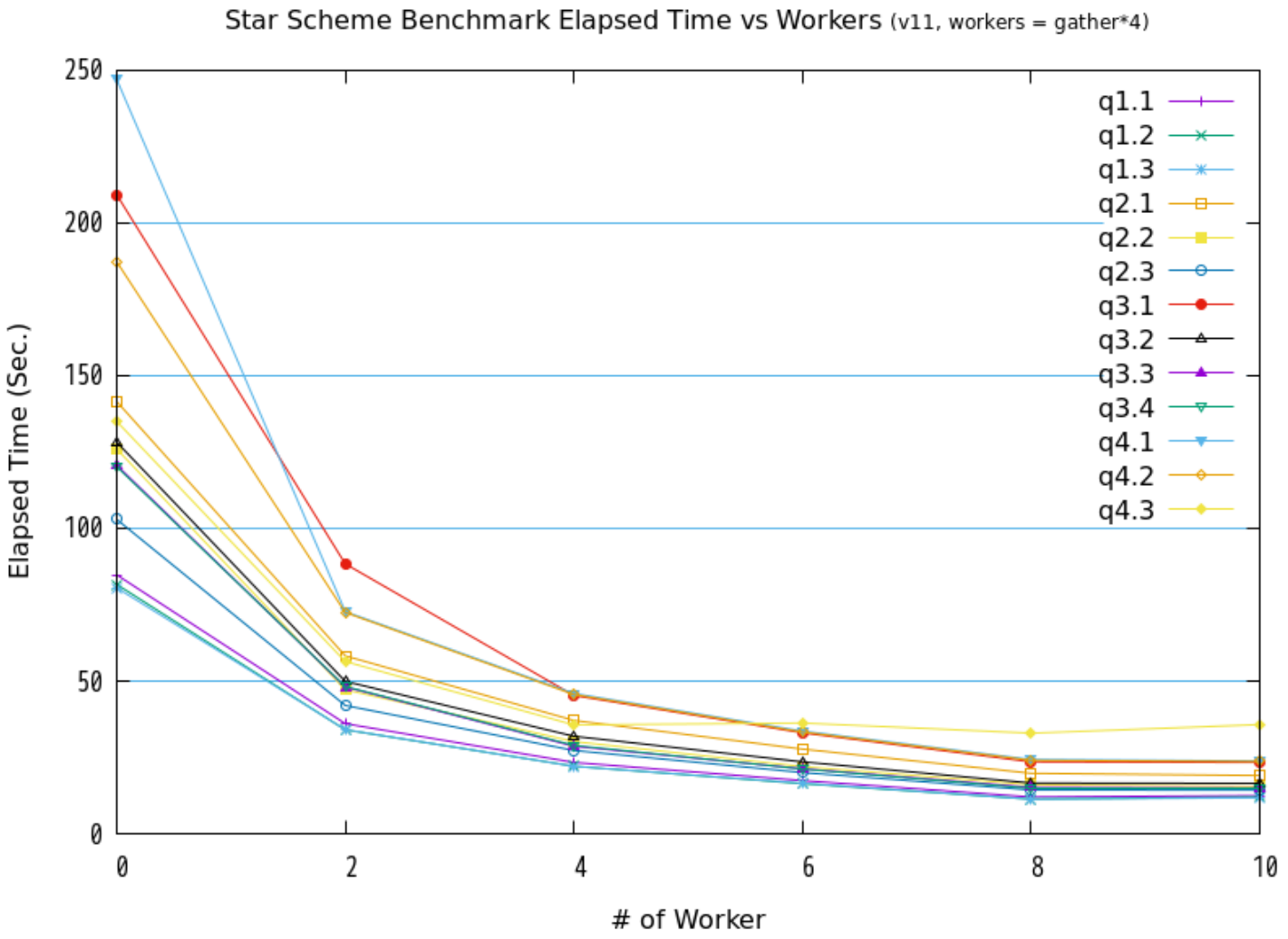


図 7.6 max_parallel_workersの値と処理時間の比較 (バージョン11 / workers = gather*4)

表 7.14 並列処理数と処理時間 (バージョン11 / workers = gather*4)

クエリ	0	2	4	6	8	10
q1.1	84.7647	36.0279	23.3482	17.3823	12.0182	12.4706
q1.2	81.6414	34.0107	22.093	16.3843	11.3385	12.0085
q1.3	80.5189	34.0967	22.0687	16.5403	11.3078	11.7771
q2.1	141.5359	58.2329	37.088	27.7059	19.7786	19.0712
q2.2	126.016	47.1534	30.1746	21.9779	16.0026	15.4808
q2.3	102.9223	41.9794	27.2528	20.0144	14.335	14.2886
q3.1	209.2735	88.3712	45.2504	33.0227	23.5292	23.3375
q3.2	128.2047	49.8938	31.8979	23.5351	16.5862	16.5175
q3.3	120.6891	48.0444	28.6483	21.2713	15.1452	15.001
q3.4	120.2251	48.2142	28.8943	21.1918	14.9976	15.0031
q4.1	247.059	72.7343	45.9153	33.6155	24.295	23.8475
q4.2	187.3207	72.4182	45.5837	33.3028	24.0262	23.658
q4.3	135.0164	56.3806	35.6238	36.233	32.9085	35.7456

検証の結果、全クエリが、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、処理時間と並列処理数の間には、反比例に近い関係があると推測できます。

また、workers = gatherとworkers = gather*4を比較すると、バージョン10とバージョン11では、並列処理数が2から8の間では、gather*4の方が高速である傾向が確認できました。

ただ、バージョン11のq4.3は、並列処理数4以上の性能が頭打ちとなっており、バージョン11特有の問題が発生している可能性があります。

7.5.3. 検証C

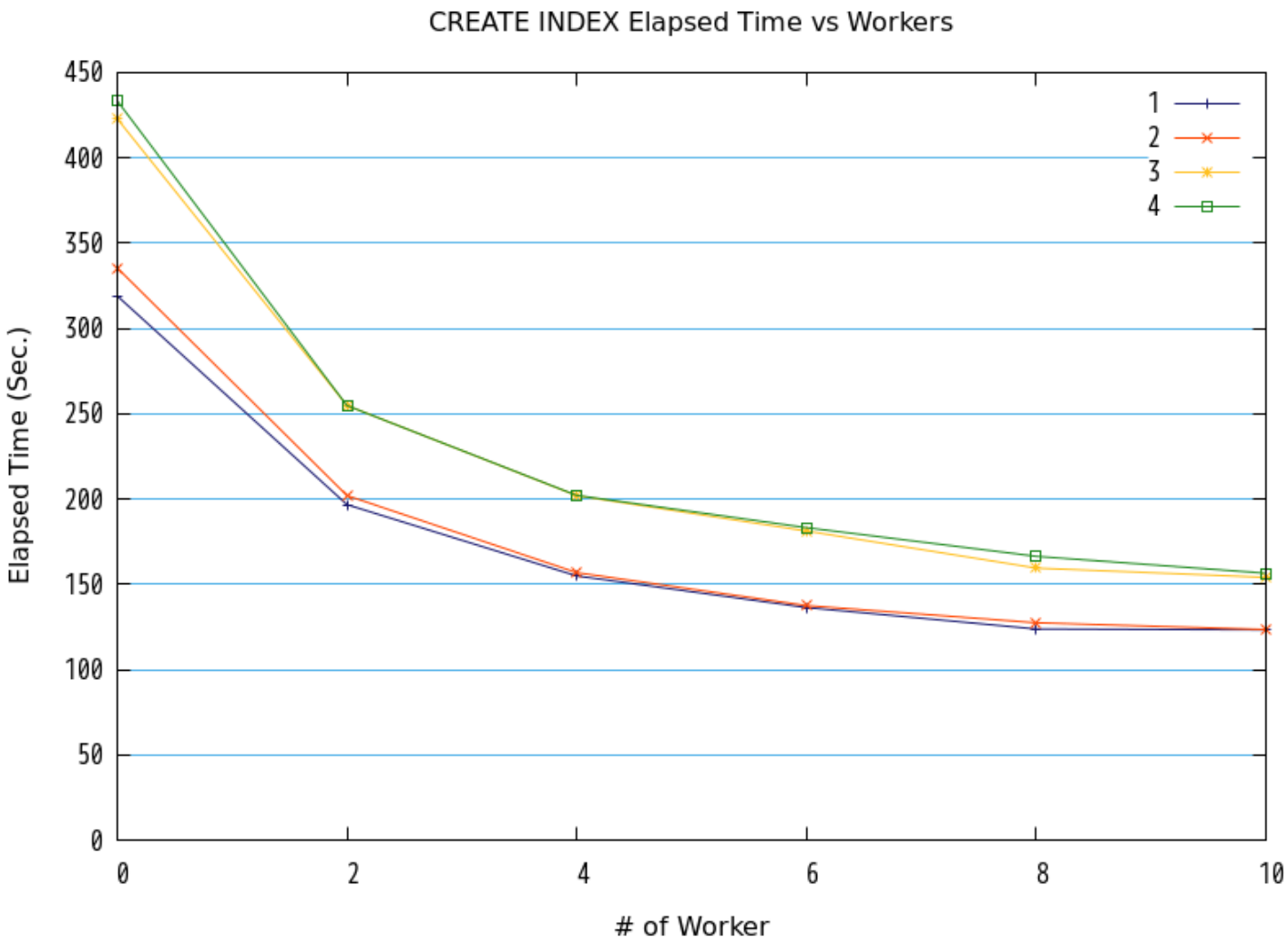


図 7.7 max_parallel_maintenance_workersの値と処理時間の比較

表 7.15 並列処理数と処理時間

ワーカー数	1列 (秒)	2列 (秒)	3列 (秒)	4列 (秒)
0	318.9848	335.4180	422.8608	433.4036
2	196.6795	201.9619	254.7677	254.7121
4	154.8794	156.7898	201.9349	202.0619
6	136.3506	137.5311	181.1933	183.1736
8	123.7943	127.3894	159.4948	166.3534
10	123.2796	123.4265	154.0125	156.4379

検証の結果、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、処理時間と並列処理数の間には、反比例に近い関係があると推測できます。

7.5.4. 対照群 (パラレルクエリoff)

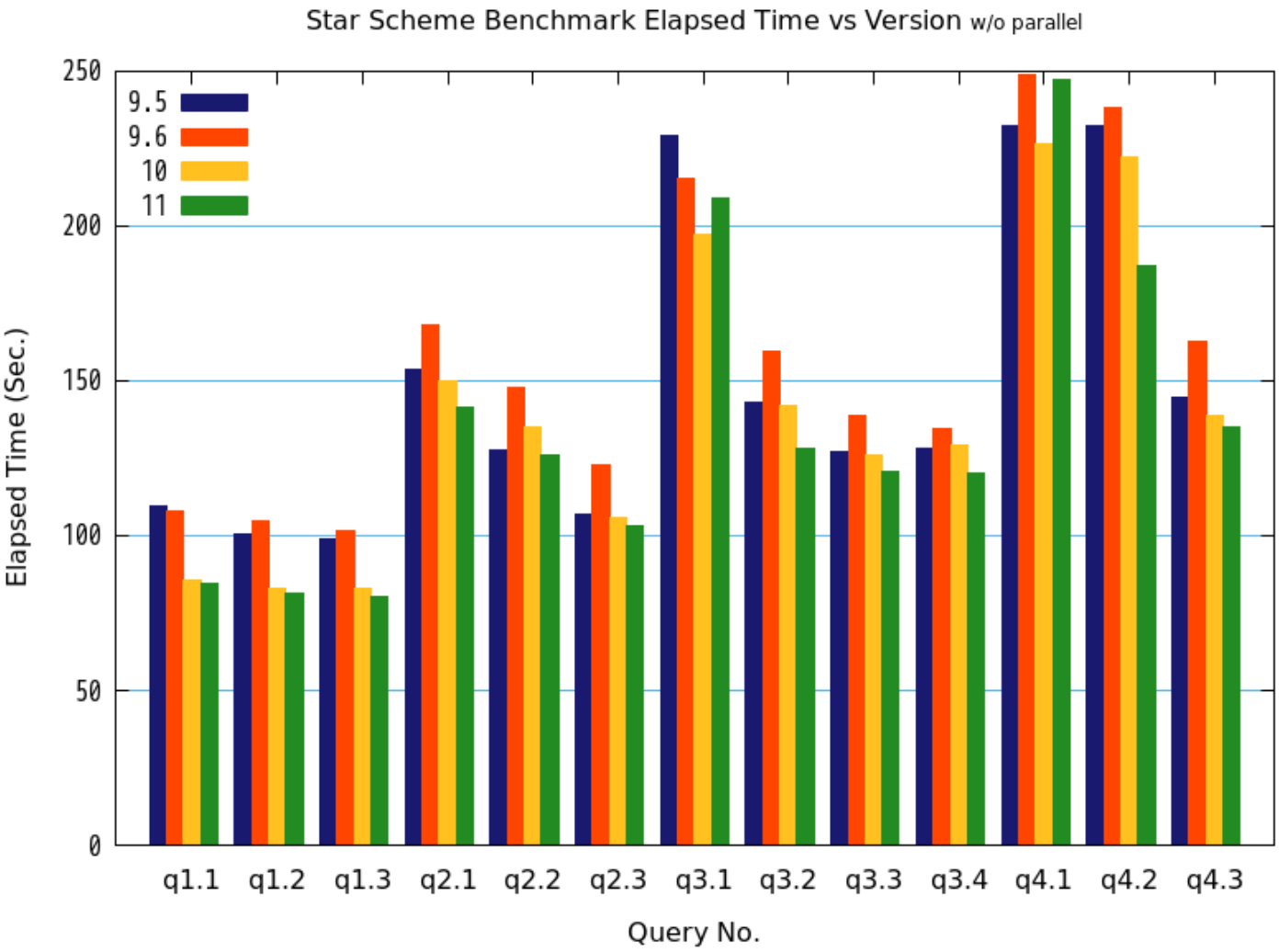


図 7.8 バージョンと処理時間の比較

表 7.16 バージョンと処理時間

クエリ	9.5 処理時間(秒)	9.6 処理時間(秒)	9.6 性能向上比(倍)	10 処理時間(秒)	10 性能向上比(倍)	11 処理時間(秒)	11 性能向上比(倍)
q1.1	109.6738	107.8582	1.0168	85.4489	1.2835	84.7647	1.2939
q1.2	100.5875	105.0022	0.9580	82.9543	1.2126	81.6414	1.2321
q1.3	99.0719	101.5858	0.9753	82.8403	1.1959	80.5189	1.2304
q2.1	153.6779	168.0892	0.9143	150.0331	1.0243	141.5359	1.0858
q2.2	127.4209	147.6302	0.8631	135.3275	0.9416	126.0160	1.0111
q2.3	106.7450	122.7834	0.8694	105.8896	1.0081	102.9223	1.0371
q3.1	229.1734	215.4955	1.0635	197.5059	1.1603	209.2735	1.0951
q3.2	143.0648	159.3785	0.8976	142.1985	1.0061	128.2047	1.1159
q3.3	127.0212	138.7790	0.9153	125.7963	1.0097	120.6891	1.0525
q3.4	128.4246	134.3616	0.9558	129.0627	0.9951	120.2251	1.0682
q4.1	232.5540	248.7920	0.9347	226.6286	1.0261	247.0590	0.9413
q4.2	232.1840	238.1739	0.9749	222.4871	1.0436	187.3207	1.2395
q4.3	144.7047	162.9800	0.8879	138.6454	1.0437	135.0164	1.0718

検証の結果、クエリの処理時間について、パラレルクエリoffの状態で、バージョン9.5との比較でバージョン9.6では0.91倍(q2.1)〜1.01倍(q1.1)、バージョン10では0.94倍(q2.2)〜1.21倍(q1.2)、バージョン11では0.94倍(q4.1)〜1.29倍(q1.1)の高速化が認められました。

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、バージョン10とバージョン11の間において改善が行われ、大部分が高速化したことが伺えます。

7.6. 考察

7.6.1. 検証A

検証Aでは、パラレルクエリのないバージョン9.5よりバージョン9.6の方が高速であり、バージョン9.6よりもバージョン10、バージョン11の方が概ね高速である傾向が見られました。ただし、バージョン10とバージョン11の間の差は、概ねバージョン11の方が高速であるものの、q3.1ではバージョン10が最も早く、q4.1とq4.2ではバージョン11が大幅に高速でした。

q3.1では、バージョン10とバージョン11で結合順が変化しており、これが原因となってバージョン11で速度が低下した可能性があります。

q4.1とq4.2では結合順は変化しておらず、ディメンジョンテーブルであるpart, supplier, dateの各テーブルのSeqScanに要する時間が、バージョン10と比較して1/3となっていることが確認できました。

パラレルクエリでのJoinノードにおける右側(Inner)のテーブルの検索処理は、バージョン10までは各パラレルワーカーがそれぞれのテーブルを全検索する関係で、検索時間が各パラレルワーカーでの所要時間分だけ増加していました。バージョン11での高速化は、新機能であるハッシュテーブルの共有化により、テーブルスキャンが1回になった結果であると考えます。SSBでのディメンジョンテーブルは、テーブルサイズが数百MBとなるものもあるため、ハッシュテーブルの共有化によるスキャン回数低減が大きな効果をもたらしたものと言えます。

以上から、バージョン11からの新機能であるハッシュテーブルの共有化は、複数のテーブルの結合操作時に、テーブルの大きさが大きいほど効果が出やすいと考えます。

なお、2016年度のパラレルクエリ検証 で使用した検証用クエリを使用しての検証も行いましたが、こちらの結果 からはバージョン10とバージョン11の間の差はほとんどなく、この2バージョンのパラレルクエリの基本性能は互角と言って差し支えないと考えます。

7.6.2. 検証B

7.6.2.1. max_worker_processes、max_parallel_workers、max_parallel_workers_per_gather 間の関係

バージョン10とバージョン11では、ほとんどのクエリで max_worker_processes、max_parallel_workers を max_parallel_workers_per_gather の4倍の値としたパターンの方が、max_worker_processes、max_parallel_workers と max_parallel_workers_per_gatherを同値とした場合よりも高速に動作しました。以下に、SSB q3.1におけるworkers = 0 (パラレルクエリなし) と 特に顕著な差を認めた workers = 2 (起動予定ワーカー数2) の性能向上比を示します。

表 7.17 SSB q3.1におけるパターンと性能向上比

パターン	workers=0	workers=2	性能向上比(倍)
バージョン10 workers = gather	197.5059	106.1948	1.8598
バージョン11 workers = gather	209.2735	130.4551	1.6042
バージョン10 workers = gather*4	197.5059	75.1554	2.6280
バージョン11 workers = gather*4	209.2735	88.3712	2.3681

誤差とは言えない大きさの差がついているため起動ワーカー数が正しくないのではないか、との仮説を立て、SSB q3.1のworkers = 2を対象に、EXPLAIN ANALYZE中の起動ワーカー数を確認しました。その確認結果を以下に示します。

表 7.18 バージョンと起動ワーカー数

バージョン	workers = gather	workers = gather*4
10	1	2
11	1	2

実際の並列度は起動ワーカー数+1であるため、workers = gatherのパターンでは並列度2、workers = gather*4のパターンでは並列度3となります。パターンと性能向上比の関係が、並列度とほぼ一致していることがわかります。

これは、バージョン10以降では、パラレルクエリのためのワーカーとは別に、バックグラウンドワーカーが1つ起動していることを示唆します。このため、max_worker_processesの上限に達してしまい、パラレルクエリ用に起動予定のワーカーを一部立ち上げられなかったものと推測します。

バージョン10以降のバックグラウンドワーカーは、パラレルクエリ以外にはロジカルレプリケーション用のものが挙げられます。パラレルクエリ関連のパラメータ設定を行うにあたっては、他にバックグラウンドワーカーを使用する(パラレルクエリの動作と競合する)機能との整合を十分に考慮する必要があると考えます。

7.6.2.2. バージョン11における SSB q4.3 の性能

バージョン11において、SSBのq4.3の性能が並列処理数4で頭打ちとなりました。

バージョン9.6とバージョン10ではこの現象は起きておらず、バージョン11固有の現象であることがわかります。今回の検証で使用したテーブルのサイズからは、並列処理数の理論値は9であるため、並列処理数9付近まで性能が改善することを予想していました。

そこで、実行計画上の並列処理数を確認しました。その結果を以下に示します。実行計画上の並列処理数は、パラレルクエリ使用時に現れる Gather ノード内の"Workers Planned"の値として、EXPLAINの実行結果内に示されます。

表 7.19 SSB q4.3における並列処理数の設定と実行計画での値

max_parallel_workers_per_gather	Workers Planned
0	—
2	2
4	4
6	4
8	4
10	4

以上から、実行計画として並列処理数が4より大きい値にならないことがわかりました。次に、現象が発生していないバージョン10と現象が発生するバージョン11の実行計画を比較します

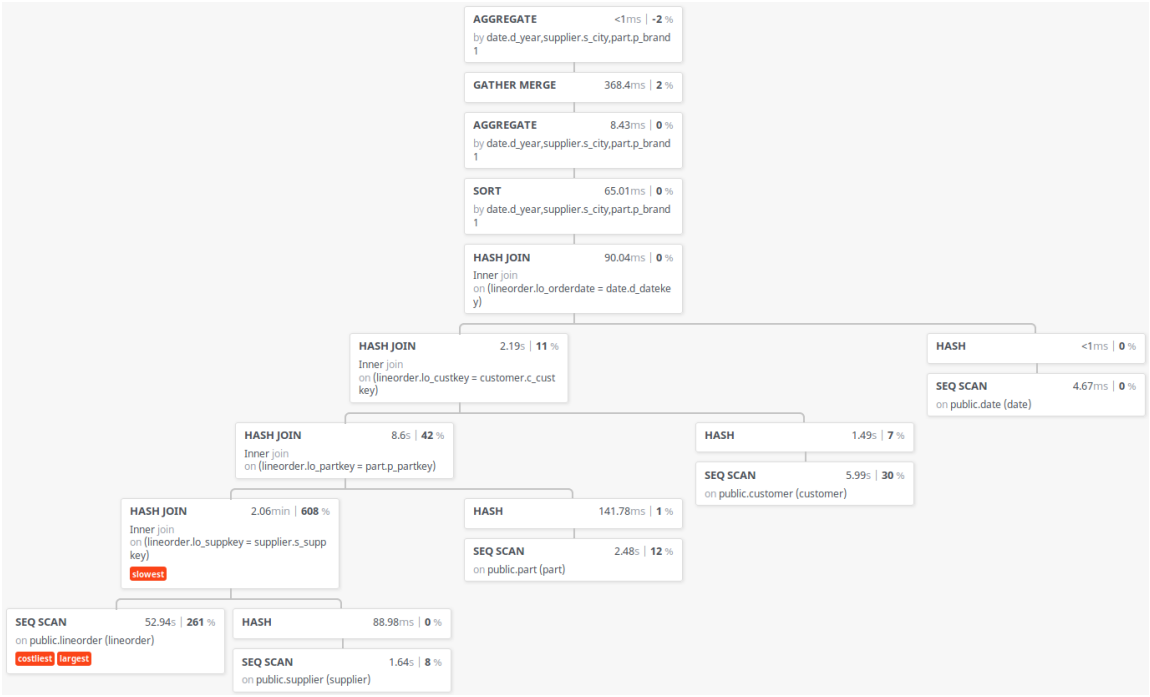


図 7.9 Star Schema Benchmark q4.3の実行計画(バージョン10)

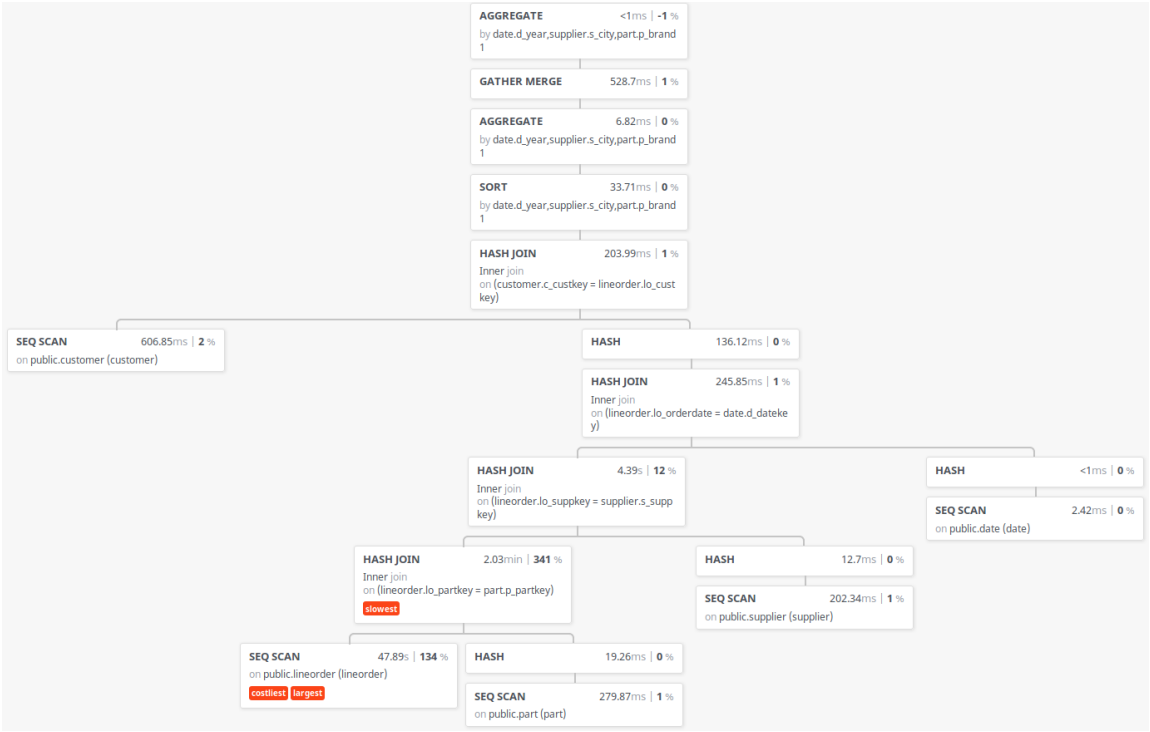


図 7.10 Star Schema Benchmark q4.3の実行計画(バージョン11)

バージョン10とバージョン11の間で、実行計画の形が変化していることがわかります。ここで、各テーブルの物理サイズと起動予定ワーカー数を調べました。

表 7.20 SSBテーブル(SF = 100GB)におけるテーブルサイズと起動予定ワーカー数

テーブル名	テーブルの物理サイズ	起動予定ワーカー数
lineorder	59 GB	9
customer	357 MB	4
date	336 kB	0
supplier	109 MB	3
part	157 MB	3

上記の中で、起動予定ワーカー数が9となるのは lineorder テーブルのみであり、一方で起動予定ワーカー数が4となるのは customer テーブルのみです。ここで、上記のバージョン11の実行計画を確認すると、バージョン11の実行計画は、最上位にあるのHash Joinの左側(Outer)にcustomerテーブルのSeqScanが存在します。[11] 一方で、問題が発生していないバージョン10の実行計画では、lineorderテーブルのSeqScanが最上位のHash Joinの左側(Outer)から、左に辿っていった末端に存在します。

[11] 実行計画を出力するツールの制限でパラレルクエリ関連の情報が出力されていませんが、上記バージョン11の実行計画中のSeqScan、Hash Joinはそれぞれ Parallel Seq Scan、Parallel Hash Joinです。

このため、このクエリ全体の起動予定ワーカー数を決定するものは、最上位のJoinノードにおける左側(Outer)側の起動予定ワーカー数ではないか、という仮説を立て、バージョン11のソースコードを確認しました。この結果、実行計画の最適化機構(Optimizer)内の結合方法を検討する部分のコードに、以下のコードがあることを確認しました。

```
/* This is a foolish way to estimate parallel_workers, but for now... */
pathnode->jpath.path.parallel_workers = outer_path->parallel_workers;
```

以上のコードがもととなり、クエリ全体の起動予定ワーカー数が customer テーブル由来の起動予定ワーカー数で規定されたものと考えます。なお、このコードはバージョン9.6から存在しており、本件と同様の問題が、バージョン9.6以降のすべてのバージョンで起こりうると考えます。

本件については、PostgreSQLの開発コミュニティに報告の上、ロジックが改良されるよう働きかけていきます。

7.6.3. 検証C

B-Treeインデックスをパラレルで作成する処理において、複合インデックスの対象列が3以上となった場合に、2以下の場合と比較して、およそ1.25倍～1.35倍の時間がか

かるようになりました。

インデックスの作成は、作成対象のテーブルにアクセスする必要があるため、対象列の増加に伴って処理量が増えたためであると想定しました。この場合、対象列と作成時間の差に関して、比例の関係となることになります。そこで、対象列と作成時間の差を比較しました。結果を以下に示します。

表 7.21 CREATE INDEXにおける複合列数と対象列数1との差

ワーカー数	列数2と列数1の差分	列数3と列数1の差分	列数4と列数1の差分
0	16.4333	103.876	114.4189
2	5.2824	58.0882	58.0326
4	1.9104	47.0555	47.1825
6	1.1805	44.8427	46.823
8	3.5951	35.7005	42.5591
10	0.1469	30.7329	33.1583

以上から、対象列1と対象列2、対象列3と対象列4で、作成時間にほとんど差が見られず、比例関係にはないことがわかり、したがって、対象列数の増加に伴うものではないことがわかります。

作成時間に影響を与える要因としては、TOASTへのアクセスが発生する場合です。TOASTではデータ圧縮されている他、テーブルデータの一部のみがTOASTに配置される場合もあり、この場合TOASTアクセスに伴うオーバヘッドが発生します。しかし、TOASTにデータが配置されるのは、テーブルの1行のサイズが 1物理ページ(デフォルト8kB)を 超える場合のみであり、今回の lineorder テーブルは、1行が94Bにとどまっているため、当てはまりません。

他の要因としては、インデックスのサイズが大きくなる場合です。B-Treeインデックスの大きさは、複合列にまたがるインデックスの場合、各列に格納されている値の一意な組み合わせの数に依存します。そこで、今回検証した列の組み合わせにおいて、組み合わせ数を算出しました。

表 7.22 複合インデックスの対象列と組み合わせ数

複合数	対象列	組み合わせ数
1	lo_orderkey	150000000
2	lo_orderkey lo_custkey	150000000
3	lo_orderkey lo_custkey lo_partkey	600036916
4	lo_orderkey lo_custkey lo_partkey lo_suppkey	600038145

以上から、複合インデックスの対象列が1の場合と2の場合の間で組み合わせ数が変わっておらず、2の場合と3の場合の間で大幅に増加、3の場合と4の場合の間で微増にとどまっていることがわかります。

これは、列数2以上の各複合インデックス作成時間と列数1のインデックス作成時間の差の傾向と一致します。ここから、対象列数が3以上となった場合にインデックス作成時間が増加したのは、対象列における一意な組み合わせの数が増加したことによるものと考えます。

7.7. 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。実際に検証を行った結果、昨年度と比較して、注意点は依然として多いもののパラレルクエリは実用に耐えうるものになったと思います。

特に、テーブルの検索やテーブルの結合、ソート、集約演算といった基本操作のパラレル対応はもちろんのこと、大きいテーブルの結合操作に効果を発揮するハッシュテーブルの共有化といった新機能で、大量のデータ内から目的の情報を探索・導出するといった作業を効率的に行えるようになったものと確信します。

以上から、事前の十分な評価は必要ですが、現状のパラレルクエリの実装でOLAP用途での使用は可能である、という結果となりました。

ただ、バグと見られる挙動も見られることから、より不安なくパラレルクエリを使用するためにも、常に最新バージョンでの使用を推奨するとともに、パラレルクエリ関連のさらなる進化と、これを司るプランナ・エグゼキュータのさらなる深化を働きかけていきます。

7.8. Appendix 2016年度検証モデルでの結果

検証A で述べたとおり、2016年度のパラレルクエリ検証 で使用した検証用クエリを使用しての検証も行いました。

2016年度検証データモデルでは、以下に示す条件で検証を行った結果、バージョン9.6以降では、全クエリでパラレルクエリが動作しました。2016年度の検証においてはパーティション構成ありとなしの両方で検証を行いましたが、本検証においてはパーティション構成を採用していません。

表 7.23 検証条件

ストレージ	SQL文	保持データ量
外付けストレージ装置	384万行(1日分) [12] を集計するSQL	1年分(384万×366)

[12] Star Schema BenchmarkにおけるScale Factorとデータサイズを一致させる目的で、1日あたりの件数を2016年度のパラレルクエリ検証 での設定値の半分(384万件)に設定しています。この設定により、データサイズは110GB程度となります。

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。

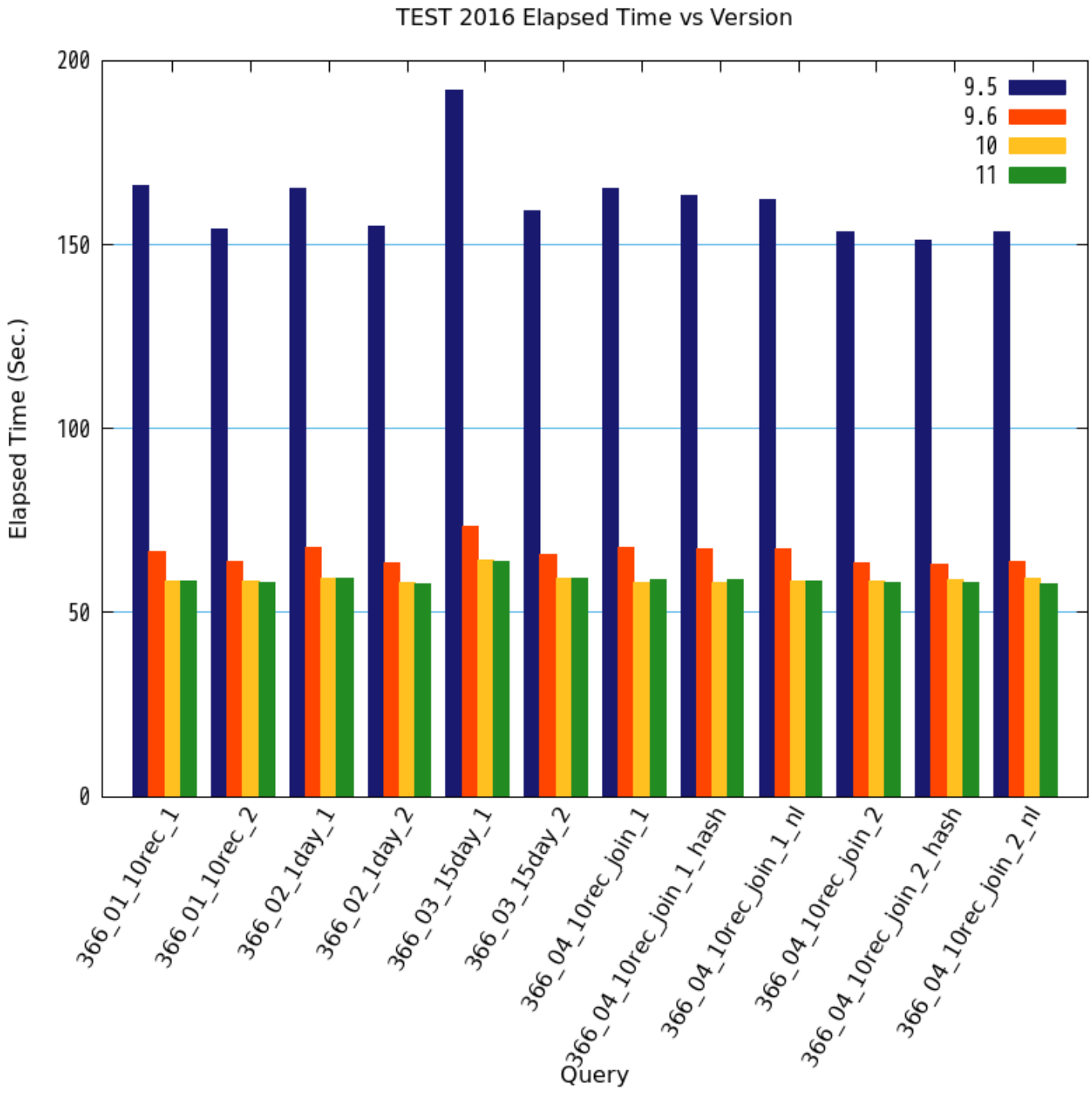


図 7.11 バージョンと処理時間の比較

表 7.24 バージョンと処理時間

クエリ	9.5 処理時間(秒)	9.6 処理時間(秒)	9.6 性能向上比(倍)	10 処理時間(秒)	10 性能向上比(倍)	11 処理時間(秒)	11 性能向上比(倍)
366_01_10rec_1	166.1090	66.6200	2.4934	58.5179	2.8386	58.4063	2.8440
366_01_10rec_2	154.2194	63.7848	2.4178	58.7123	2.6267	58.0952	2.6546
366_02_1day_1	165.4775	67.7755	2.4416	59.2671	2.7921	59.1916	2.7956
366_02_1day_2	154.8390	63.5417	2.4368	58.1420	2.6631	57.7205	2.6826
366_03_15day_1	192.0502	73.4120	2.6161	64.3524	2.9844	63.9317	3.0040
366_03_15day_2	159.1187	65.9643	2.4122	59.3359	2.6817	59.4298	2.6774
366_04_10rec_join_1	165.4620	67.5307	2.4502	58.1532	2.8453	59.0345	2.8028
366_04_10rec_join_1_hash	163.4051	67.4919	2.4211	58.0703	2.8139	58.8882	2.7748
366_04_10rec_join_1_nl	162.3903	67.3878	2.4098	58.5880	2.7717	58.3832	2.7815
366_04_10rec_join_2	153.5317	63.6836	2.4109	58.6402	2.6182	58.0241	2.6460
366_04_10rec_join_2_hash	151.2344	63.3228	2.3883	58.9818	2.5641	58.1867	2.5991
366_04_10rec_join_2_nl	153.5228	63.8352	2.4050	59.2450	2.5913	57.9361	2.6499

検証の結果、クエリの処理時間について、バージョン9.5との比較で、バージョン9.6で2.38倍(366_04_10rec_join_2_hash)〜2.61倍(366_03_15day_1)、バージョン10で2.56倍(366_04_10rec_join_2_hash)〜2.98倍(366_03_15day_1)、バージョン11で2.64倍(366_04_10rec_join_2)〜3.00倍(366_03_15day_1)の高速化が認められました。こちらの結果が示すとおり、バージョン10とバージョン11の間の差はわずかであり、この2バージョンのパラレルクエリの基本性能は互角と言って差し支えないと考えます。

8. おわりに

今年度(2018 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 7 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 7 期目となりました。本 WG では 2016 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 4 つのテーマの検証結果を報告しました。

1. PostgreSQL 11および前バージョン10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 11で強化されたパラレルクエリ検証
4. PostgreSQL 11で実装されたJITコンパイル機能の検証

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、昨年度から引き続き、定点観測と同様の検証モデルを用いてWindows版PostgreSQLの性能検証を行っています。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11となった今回は強化されたパラレルクエリの他に、JITコンパイルが検証対象に選定されました。

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 2017 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)

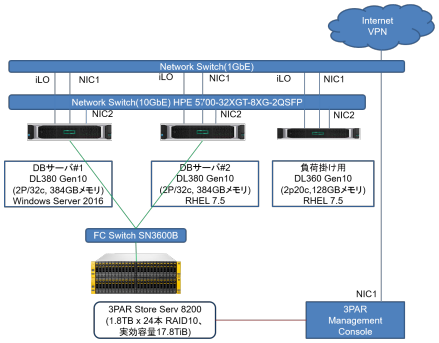
9. 著者

（企業・団体名順）

版	所属企業・団体名	部署名	氏名
第1.0版 (2018年度WG1)	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
	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	小澤 怜
	SRA OSS, Inc. 日本支社	OSS事業本部	近藤 雄太
	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
	NTTテクノクロス株式会社	IoTイノベーション事業部	山本 育
	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	ミドルウェア事業本部 データマネジメント・ミドルウェア事業部 グローバル技術部	並木 達郎

10. 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：富士通株式会社
- JITコンパイル検証での使用機器/設備

項目	仕様
機種	PRIMERGY RX2540M4
CPU	Intel(R) Xeon Gold 6140 (18コア 36スレッド) ×2
メモリ	256GB
内蔵ストレージ	SSD 700GB