PostgreSQLエンタープライズ・コンソーシアム 技術部会 WG#1

2020年度WG1活動報告書

目次

目次 2
1. 改訂履歴 5
2. ライセンス 6
3. はじめに 7
3.1. PostgreSQLエンタープライズコンソーシアムとWG1について 7
3.1.1. 各WGとWG1の活動テーマ 7
3.2. WG1の活動テーマ 7
3.3. 本資料の概要と目的 7
3.3.1. 2020年度の活動テーマ 7
3.3.2. 定点観測（スケールアップ） 8
3.3.3. パラレルクエリ 8
3.4. 成果の公開 8
3.4.1. 2019年度までの活動テーマの紹介 8
3.5. 実施体制 8
3.6. 実施スケジュール 9
3.7. 謝辞 9
4. 定点観測(スケールアップ検証) 10
4.1. 検証概要 10
4.2. pgbenchとは 10
4.3. 検証構成 11
4.3.1. マシン構成 11
4.3.2. ソフトウェア構成 12
4.3.3. カーネル設定 12
4.3.4. PostgreSQL 設定 12
4.3.5. 環境 12
4.4. 検証方法(参照系) 13
4.5. 検証結果(参照系) 14
4.6. 検証方法(更新系) 15
4.7. 検証結果(更新系) 16
4.8. まとめ 17
5. Windows版PostgreSQL性能検証 18
5.1. 検証概要 18
5.2. 検証構成 18
5.2.1. ハードウェア構成 18
5.2.2. ソフトウェア構成 18
5.3. 検証モデル 19
5.3.1. データベースクラスタ構成 19
5.3.2. 検証用ベンチマーク 19
5.3.2.1. 初期データ登録 19
5.3.3. 仮測定 19
5.3.3.1. 参照系仮測定 19
5.3.3.2. 更新系仮測定 20
5.3.4. 本測定 21
5.3.4.1. 参照系測定フロー 21
5.3.4.2. 更新系測定フロー 21
5.3.5. PostgreSQL 設定 21
5.4. 検証結果 21
5.4.1. 参照系 22
5.4.2. 更新系 22
5.5. Linux WindowsOS 比較 23
5.5.1. 参照系 23
5.5.2. 更新系 24
5.6. まとめ 25
6. パラレルクエリ 性能検証 26

6.1. パラレルクエリとは 26
6.1.1. パラレルクエリ 26
6.1.2. PostgreSQLでのパラレルクエリ 26
6.1.3. パラレルクエリに関連するパラメータ 27
6.2. Star Schema Benchmarkとは 28
6.2.1. データモデル 28
6.2.2. クエリについて 30
6.3. パラレルインデックススキャン性能検証 33
6.3.1. 検証概要 33
6.3.2. 検証構成 33
6.3.2.1. マシン構成 33
6.3.2.2. SW構成 33
6.3.2.3. DB設定 34
6.3.3. 検証内容 34
6.3.3.1. Count クエリ 34
6.3.3.2. Star Schema Benchmark 34
6.3.3.2.1. Index 付与 34
6.3.3.2.2. クエリ 35
6.3.4. 検証結果 35
6.3.4.1. Count クエリ 35
6.3.4.2. Star Schema Benchmark 37
6.3.4.2.1. Index あり状態の結果 37
6.3.4.2.2. Merge Join の強制によるパラレルインデックスキャンの利用 38
6.3.5. 補足 - Star Schema Benchmark クエリでの INDEX 有無による性能比 39
6.3.6. まとめ 41
6.3.6.1. パラレルインデックススキャンの効果 41
6.3.6.2. 注意点 41
6.3.6.2.1. Parallel Index Only Scan 利用時の注意点 : VACUUM の必要性 41
6.3.6.2.2. パラレルインデックススキャンとインデックススキャンのパラレル実行の違い 41
6.4. Windowsでのパラレルクエリ性能検証 42
6.4.1. 検証概要 42
6.4.2. 検証構成 42
6.4.2.1. マシン構成 42
6.4.2.2. SW構成 42
6.4.2.3. DB設定 43
6.4.2.3.1. 測定手順 43
6.4.2.4. データモデル 43
6.4.2.5. クエリ 43
6.4.3. 結論 43
6.4.4. 測定結果 43
6.4.4.1. Star Schema Benchmarkの結果 43
6.4.4.2. Windows版での性能差について 44
6.5. パラレルクエリ/テーブル・パーティショニング性能検証 45
6.5.1. 検証概要 45
6.5.2. テーブル・パーティショニングとは 45
6.5.2.1. テーブル・パーティショニング 45
6.5.2.2. PostgreSQLでのパーティショニング 46
6.5.2.3. テーブル・パーティショニングに関連するパラメータ 46
6.5.3. 検証構成 46
6.5.3.1. 検証ハードウェア構成 46
6.5.3.2. 検証ソフトウェア構成 47
6.5.4. 検証方法 47
6.5.4.1. 検証項目 47
6.5.4.1.1. 検証A(パラレルクエリの効果確認) 47
6.5.4.1.2. 検証B(テーブル・パーティショニングの効果確認) 47
6.5.4.1.3. 検証C(パラレルクエリとテーブル・パーティショニングの組み合わせによる効果確認) 47

6.5.4.2. データモデル 47
6.5.4.3. クエリ 47
6.5.4.4. DB設定 47
6.5.4.5. 測定手順 48
6.5.4.5.1. 検証A(パラレルクエリの効果確認) 48
6.5.4.5.2. 検証B測定方法(テーブル・パーティショニングの効果確認) 48
6.5.4.5.3. 検証C測定方法(パラレルとテーブル・パーティショニングの組み合わせによる効果確認) 49
6.5.5. 検証結果・考察 49
6.5.5.1. 検証A(パラレルクエリの効果確認) 49
6.5.5.1.1. 検証結果 49
6.5.5.1.2. 考察 50
6.5.5.1.3. 検証Aまとめ 51
6.5.5.2. 検証B(テーブル・パーティショニングの効果確認) 51
6.5.5.2.1. 検証結果 51
6.5.5.2.2. 考察 52
6.5.5.2.3. 検証Bまとめ 59
6.5.5.3. 検証C(パラレルとテーブル・パーティショニングの組み合わせによる効果確認) 60
6.5.5.3.1. 検証結果 60
6.5.5.3.2. 考察 61
6.5.5.3.3. 追加検証 62
6.5.5.3.4. 追加検証結果 63
6.5.5.3.5. 追加検証考察 63
6.5.5.3.6. 検証Cまとめ 64
6.5.6. まとめ 64
7. おわりに 66
8. 著者 67

1. 改訂履歴

版	改訂日	変更内容
1.0	2021/05/25	新規作成

2. ライセンス

本作品はCC-BYライセンスによって許諾されています。 ライセンスの内容を知りたい方は こちら でご確認ください。 文書の内容、表記に関する誤り、ご要望、感想等につきましては、PGEConsのサイト を通じてお寄せいただきますようお願いいたします。

- Eclipseは、Eclipse Foundation,Inc.の米国、およびその他の国における商標もしくは登録商標です。
- IBMおよびDb2は、世界の多くの国で登録されたInternational Business Machines Corporationの商標です。
- Intel、インテルおよびXeonは、米国およびその他の国における Intel Corporation の商標です。
- Javaは、Oracle Corporation 及びその子会社、関連会社の米国及びその他の国における登録商標です。 文中の社名、商品名等は各社の商標または登録商標である場合があります。
- Linux は、Linus Torvalds 氏の日本およびその他の国における登録商標または商標です。
- Red HatおよびShadowman logoは、米国およびその他の国におけるRed Hat,Inc.の商標または登録商標です。
- Microsoft、Windows Server、SQL Server、米国 Microsoft Corporationの米国及びその他の国における登録商標または商標です。
- MySQLは、Oracle Corporation 及びその子会社、関連会社の米国及びその他の国における登録商標です。 文中の社名、商品名等は各社の商標または登録商標である場合があります。
- Oracleは、Oracle Corporation 及びその子会社、関連会社の米国及びその他の国における登録商標です。 文中の社名、商品名等は各社の商標または登録商標である場合があります。
- PostgreSQLは、PostgreSQL Community Association of Canadaのカナダにおける登録商標およびその他の国における商標です。
- Windows は米国 Microsoft Corporation の米国およびその他の国における登録商標です。
- TPC, TPC Benchmark, TPC-B, TPC-C, TPC-E, tpmC, TPC-H, TPC-DS, QphHは米国Transaction Processing Performance Councilの商標です。
- その他、本資料に記載されている社名及び商品名はそれぞれ各社が 商標または登録商標として使用している場合があります 。

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、2020年9月にリリースされた PostgreSQL 13のスケール性、新技術・機能向上、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. 2020年度の活動テーマ

2020年度は、2020年9月24日にリリースされた PostgreSQL 13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）

- 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。 バージョン13と12の比較においては、参照系/更新系ともに性能差は認められませんでした。 WindowsOSとLinuxOSで比較したところ、Linuxの性能優位が認められました。

3.3.3. 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3つの観点で、パラレルクエリの性能を検証しました。 1つ目はパラレルインデックススキャンの性能について、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。 2つ目はWindows環境におけるパラレルクエリの性能について、デフォルトのパラメータ設定における性能をLinux環境の測定結果と比較・検証しました。 3つ目はLinux環境において、パラレルクエリのみを使用した場合の測定結果と、テーブル・パーティショニングのみを用いた場合の測定結果、さらにそれらを併用した場合の測定結果を比較し、それぞれの性能特性について比較・検証しました。

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. 2019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。

2018年度は、2018年10月にリリースされたPostgreSQL 11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。 Windows版PostgreSQLについても定点観測と同じモデルによる測定を実施しました。 PostgreSQL 10で強化されたパラレルクエリの性能検証、PostgreSQL 11からサポートされたJITコンパイルの性能検証を実施しました。

2019年度は、2019年10月にリリースされたPostgreSQL 12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 12の新機能であるPluggable Storage機構の調査および実験実装中の2つのアクセスメソッドの検証測定を実施しました。 パラレルクエリ、JITコンパイルをメニーコア環境で動作させたときの挙動および性能に関する検証を実施しました。

3.5. 実施体制

2020年10月15日に開催された2020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 2019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
日本電気株式会社
株式会社富士通ソーシアルサイエンスラボラトリ

この中で、NTTテクノクロス株式会社は、「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

2020年度は、下記スケジュールで活動しました。 2020年度の活動では、コロナウィルス禍により、対面での打ち合わせは実施せず、すべてオンライン上で議論を進めました。

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	2020年10月15日
実施計画策定	2020年10月～2020年12月
検証実施	2020年12月～2021年2月
WG1活動報告書作成	2021年2月～2021年4月
総会と成果報告会	2021年5月27日(オンライン)

3.7. 謝辞

検証用の機器を日本電気株式会社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目的とした定点観測を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。2020年度はアマゾン ウェブ サービス（AWS）上の仮想マシン（vCPU: 32、メモリ: 128GB）で、最新のPostgreSQLバージョン13と前バージョンの12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. マシン構成

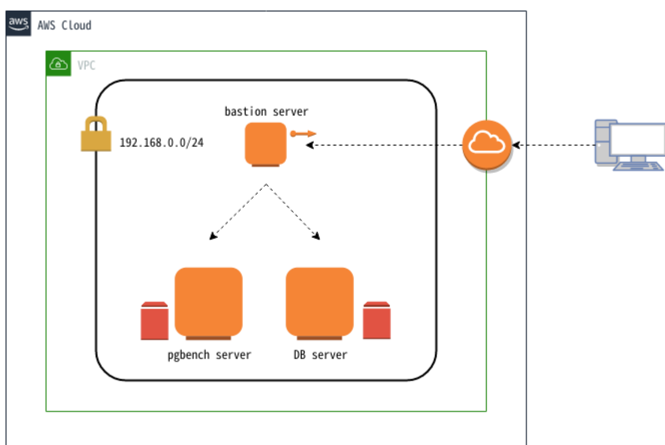


図 4.1 検証マシン構成

本検証では上図構成の通り、AWS上の仮想ネットワーク内に3台の仮想マシンを作成しました。各仮想マシンの用途は、bastion serverがAWS環境へのアクセスに用いる踏み台サーバ、pgbench serverがpgbenchを実行するためのクライアント用サーバ、DB serverがPostgreSQLを稼働させ

るサーバでした。また、pgbench serverとDB serverには検証作業で作成されるデータを格納するために、ストレージを追加しました。追加ストレージは標準的な用途で使用される汎用SSDタイプでした。DB serverに追加したストレージはIOPSをデフォルトの100 ではなく600を設定しました。各仮想マシンのスペックは以下の通りです。

表 4.7 仮想マシンのスペック

名前	インスタンスタイプ	vCPU	メモリ（GiB）	ルートデバイスサイズ（GiB）/IOPS	追加ストレージサイズ（GiB）/IOPS
bastion server	t2.micro	1	1	10/100	N/A
pgbench server	m5a.8xlarge	32	128	20/100	20/100
DB server	m5a.8xlarge	32	128	20/100	200/600

4.3.2. ソフトウェア構成

検証環境の主要なソフトウェア構成を示します。

表 4.8 bastion server

OS	Red Hat Enterprise Linux 7.9

表 4.9 pgbench server

OS	Red Hat Enterprise Linux 7.9
pgbench	13.2

表 4.10 DB server

OS	Red Hat Enterprise Linux 7.9
PostgreSQL	12.6, 13.2

上記に加え、pgbench serverとDB serverに以下のパッケージをインストールしました。

表 4.11 インストールしたパッケージ

パッケージ名	用途
Development tools（グループパッケージ）, readline-devel, zlib-devel	PostgreSQLのビルド
chrony	時刻同期
perf	検証作業中の性能分析
screen	作業セッションの保持
sysstat	検証作業中のsar取得

4.3.3. カーネル設定

- OS デフォルト

4.3.4. PostgreSQL 設定

PostgreSQL設定ファイル(postgresql.conf)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 = '*' # クライアント用サーバからの接続用
max_connections = 500  # 多めに設定
shared_buffers = 40GB  # pgbenchのスケールファクタに合わせてデータがすべてメモリに載るように設定
work_mem = 1GB
maintenance_work_mem = 20GB
checkpoint_timeout = 60min  # 試験中にチェックポイントを発生させない
max_wal_size = 160GB        # 試験中にチェックポイントを発生させない
logging_collector = on
log_checkpoints = on
log_lock_waits = on
autovacuum = off # 試験中にI/O処理を発生させない
```

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]
```

以上の構成で、クライアント接続数を変動させて実行したpgbenchのTPSが、PostgreSQL新旧バージョン間で差があるか否かを検証しました。検証は参照系性能と更新系性能について行いました。

4.4. 検証方法(参照系)

参照系ベンチマークの手順を以下に示します。

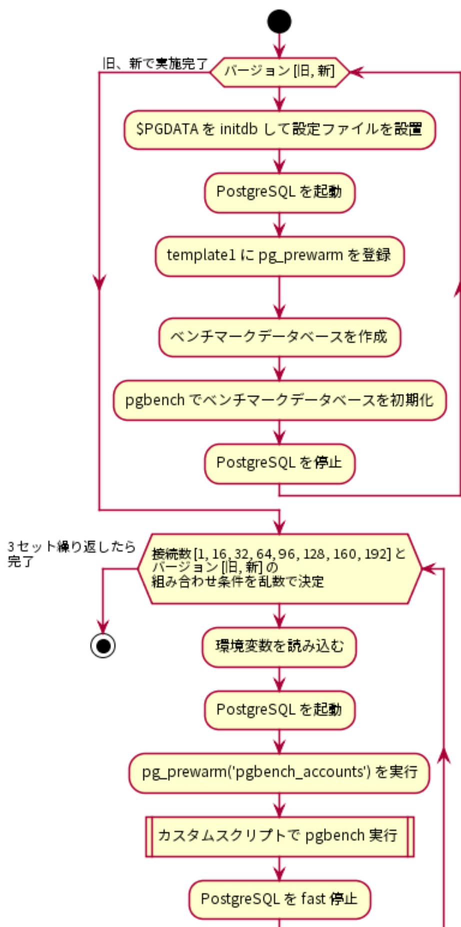


図 4.2 参照系ベンチマークの手順

はじめに、PostgreSQLサーバにデータベースクラスタを作成し、検証構成に従ってpostgresql.confを調整しました。postgresql.confの調整が完了したら、データベースクラスタを起動してtemplate1データベースにpg_prewarmを登録しました。 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。pg_prewarmの登録が完了したら、ベンチマークデータベースを作成し、pgbenchコマンドを使ってベンチマーク用データベースを初期化しました。ベンチマーク用データベースはスケールファクタ2000で初期化しました。

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

以上の作業を、PostgreSQL新旧バージョンで行いました。

その後、クライアント用サーバからベンチマークを実施しました。参照系ベンチマークではクライアント接続数を1、16、32、64、96、128、160、192の8条件とし、PostgreSQL新旧バージョンのそれぞれでTPSを取得しました。各ベンチマーク試行で決定されるクライアント接続数とPostgreSQL新旧バージョンはランダマイズしました。これは各要因における順序効果を打ち消すことを意図していました。ベンチマーク試行条件に当てはまる環境変数を読み込んで、PostgreSQLを起動したら、測定スクリプト実行前にpg_prewarmを実行しました。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');
```

pg_prewarmを実行したら、以下の参照系カスタムスクリプトをpgbenchで実行し、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;
```

クライアント用サーバで実行したpgbenchコマンドは以下の通りです。

```
$ 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]
```

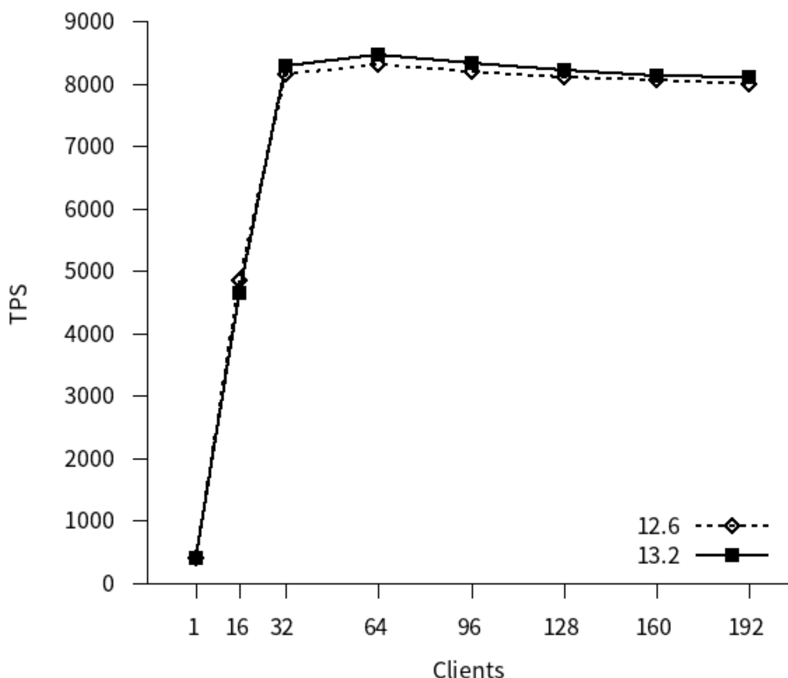
SELECTのみであるためVACUUMを実行せず(-n)、pgbench クライアント数(-c)とスレッド数(-j)を変動させながら、300秒ずつ(-T)実行しています。スレッド数はクライアント数の半分としています。スケールファクタ(-s)にはデータベース初期化時と同じ2000を指定します。また、1秒毎の進捗レポート取得(-P)と各クエリの平均レイテンシ(-r)も取得する指定にしました。ただし、-P, -r の指定によって得られた結果は今回の考察に直接用いることはありませんでした。

クライアント用サーバからのベンチマークはここまでを1セットとし、3セット実施して得られたTPSの中央値を結果としました。

4.5. 検証結果(参照系)

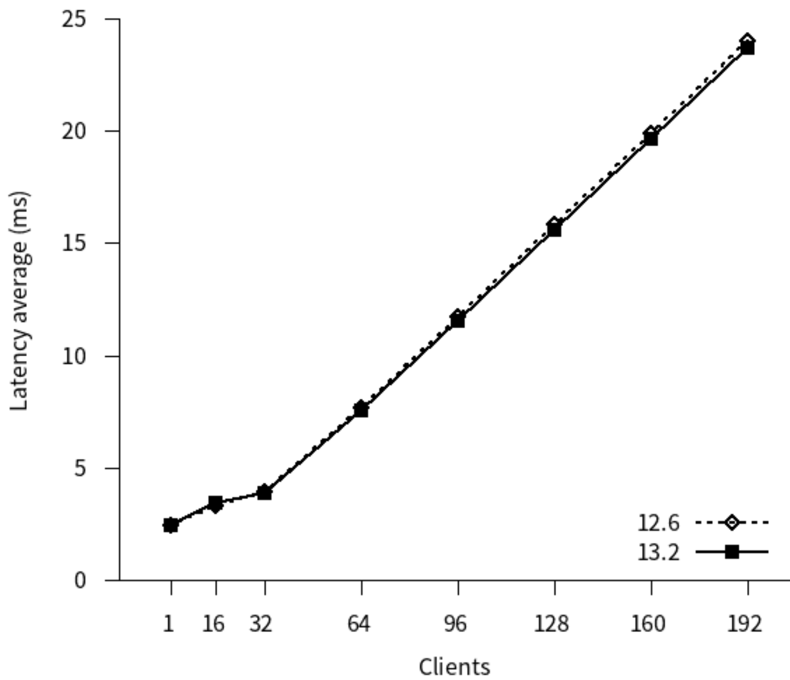
参照系ベンチマークのTPSのグラフを以下に示します。TPSは接続確立にかかった時間を無視した値(excluding connections establishing)を用いています。(以降同様)

12, 13 はともにクライアント接続数の増加につれて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。また、新旧バージョン間でTPSに差はありませんでした。



参照系ベンチマークのレイテンシのグラフを以下に示します。

12, 13 はともにクライアント接続数の増加につれてレイテンシがほぼ直線的に増加しています。また、新旧バージョン間でレイテンシに差はありませんでした。



4.6. 検証方法(更新系)

更新系ベンチマークの手順を以下に示します。

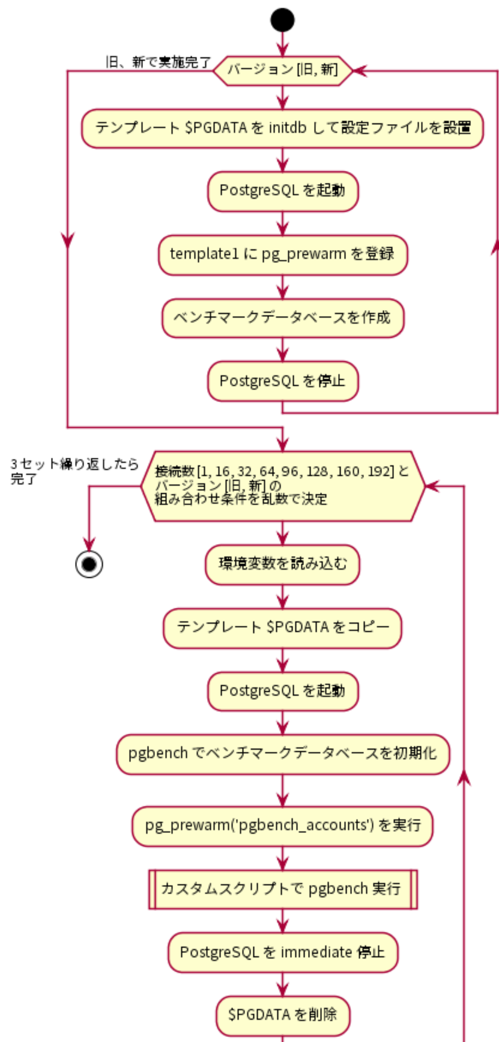


図 4.3 更新系ベンチマークの手順

はじめに、PostgreSQLサーバにデータベースクラスタを作成し、検証構成に従ってpostgresql.confを調整しました。postgresql.confの調整が完了したら、データベースクラスタを起動してtemplate1データベースにpg_prewarmを登録し、ベンチマークデータベースを作成しました。

以上の作業を、PostgreSQL新旧バージョンで行いました。

その後、クライアント用サーバからベンチマークを実施しました。更新系ベンチマークではクライアント接続数を1、16、32、64、96、128、160、192の8条件とし、PostgreSQL新旧バージョンのそれぞれでTPSを取得しました。各ベンチマーク試行で決定されるクライアント接続数とPostgreSQL新旧バージョンはランダマイズしました。これは各要因における順序効果を打ち消すことを意図していました。ベンチマーク試行条件に当てはまる環境変数を読み込んで、予め作成しておいたテンプレートデータベースクラスタをコピーし、PostgreSQLを起動し、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で初期化しました。このとき、フィルファクタは80としました。

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

その後、pg_prewarmを実行し、以下の更新系カスタムスクリプトをpgbenchで実行し、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、クライアント用サーバから

```
$ 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]
```

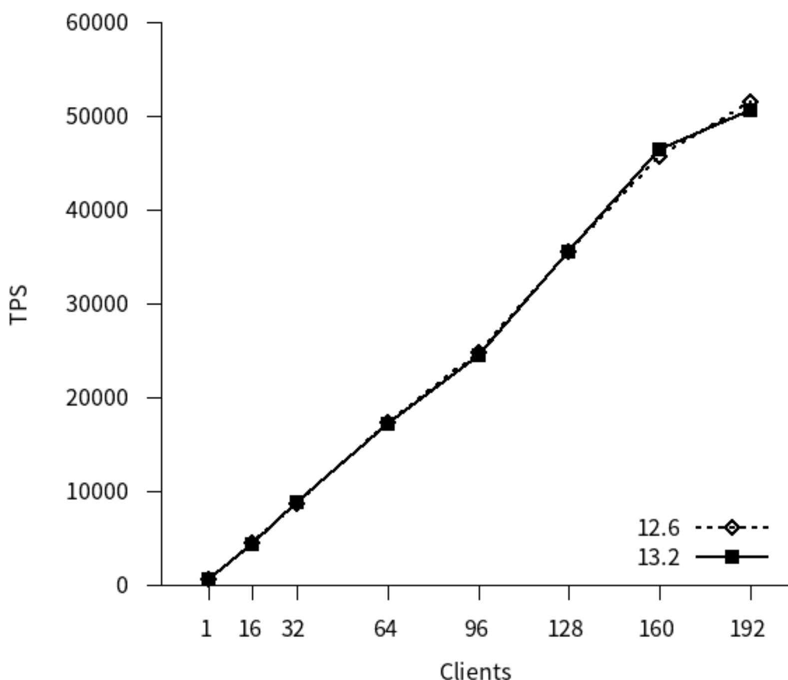
として実行しました。VACUUMを実行せず(-n)、pgbench クライアント数(-c)とスレッド数(-j)を変動させながら、300秒ずつ(-T)実行しています。スレッド数はクライアント数の半分としています。スケールファクタ(-s)にはデータベース初期化時と同じ2000を指定します。また、1秒毎の進捗レポート取得(-P)と各クエリの平均レイテンシ(-r)も取得する指定にしました。（ただし、-P, -r の指定によって得られた結果は今回の考察に直接用いることはありませんでした）

クライアント用サーバからのベンチマークはここまでを1セットとし、3セット実施して得られたTPSの中央値を結果としました。なお、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)

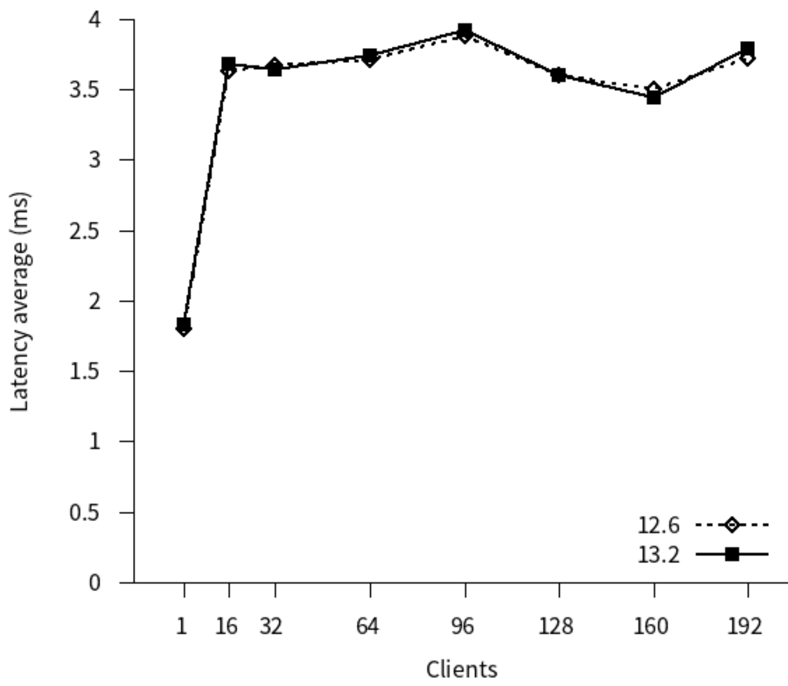
更新系ベンチマークのTPSのグラフを以下に示します。

12, 13 はともにクライアント接続数の増加につれてTPSが増加し、192接続までTPSの増加傾向は均一でした。新旧バージョン間でTPSに差はありませんでした。



更新系ベンチマークのレイテンシのグラフを以下に示します。

12, 13 はともにクライアント接続数 16 の時点からレイテンシがほぼ横ばいになりました。また、新旧バージョン間でレイテンシに差はほとんどありませんでした。



4.8. 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、更新系共に、新旧バージョン間に性能差は認められませんでした。更新性能は192接続時点までほぼ線形にTPSが上昇しましたが、性能限界付近ではTPS値に差があった可能性は残ります。次年度以降は性能限界まで接続数を増やして計測する必要があります。

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、新旧バージョンの性能比較やLinux版PostgreSQLとの性能の傾向を比較した結果を示します。

昨年度の成果報告では一度中断してしまったWindows版PostgreSQLの性能検証について、Linux上で実施していた定点観測と同様の測定を未検証となっていた11と12の比較も兼ねて11、12、13の3バージョンで実施し、併せてLinux版PostgreSQL 13とWindows版PostgreSQL 13の性能の傾向も評価しました。

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

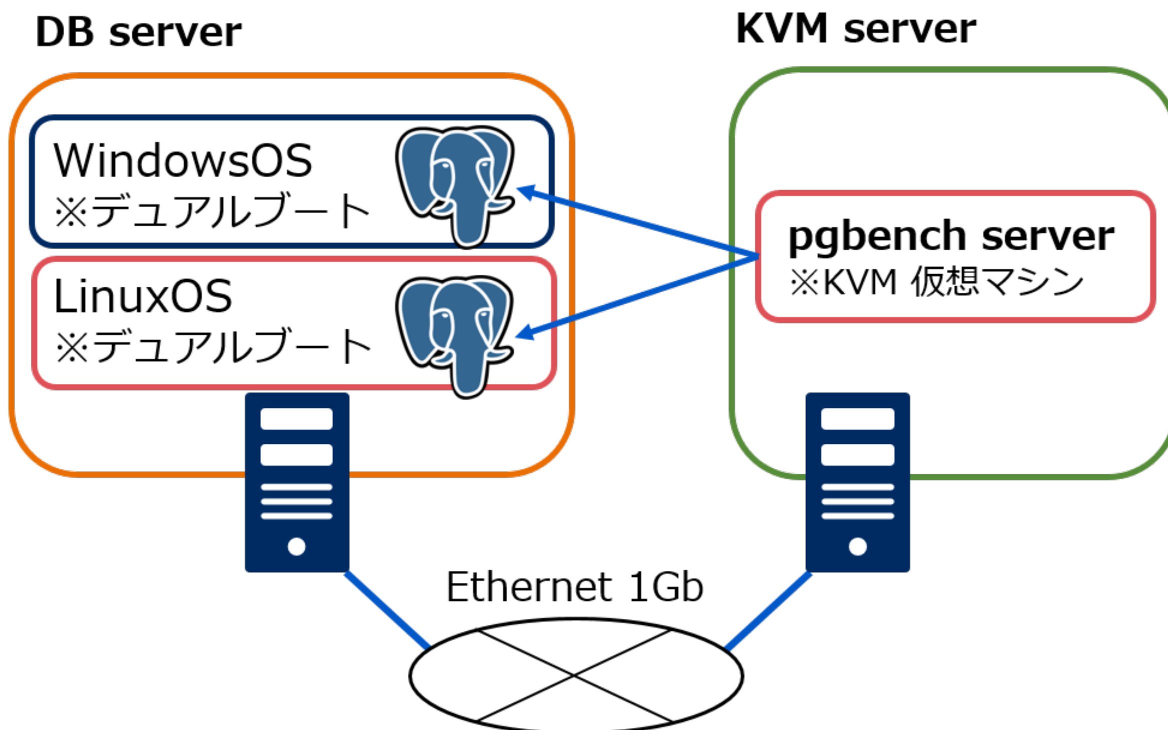


図 5.1 検証ハードウェア構成

本検証では上図構成の通り、物理サーバ1台にPostgreSQLを構築し、負荷掛け用マシンは同じNW内のKVMホスト上に仮想マシンで用意しました。 同一スペックでWindowsOSとLinuxを比較するため、OSをデュアルブートで構築しています。 各サーバ、仮想マシンのスペックは以下の通りです。

表 5.1 マシンスペック

名前	CPU	メモリ	ストレージ
DB server	Xeon E5-2630 v4(10C/20T) x 2	256GB	400GB SSD x 8　パーティションにより各OSで1.2TB設定
KVM server	Xeon E5-2630 v4(10C/20T) x 2	256GB	600GB HDD x 8
pgbench server	4vCPU	4GB	100GB

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 5.2 DB server(Windows版)

WindowsOS	Windows Server 2019 (デュアルブート)
LinuxOS	Red Hat Enterprise Linux release 8.3 (デュアルブート)
PostgreSQL 11	11.11 (EDB installer版)
PostgreSQL 12	12.6 (EDB installer版)
PostgreSQL 13	13.2 (EDB installer版)

表 5.3 DB server(Linux版)

LinuxOS	Red Hat Enterprise Linux release 8.3 (デュアルブート)
PostgreSQL 13	13.2 (ソースビルド版)

表 5.4 pgbench server

OS	Red Hat Enterprise Linux Server release 7.2
PostgreSQL(pgbench)	13.2 (ソースビルド版)

表 5.5 KVM server

OS	Red Hat Enterprise Linux Server release 7.2

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

今回の性能検証では、ディスクをパーティションで区切りデータベース格納領域とWAL格納領域を分離した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、今回用意した検証マシンのスペックでどの程度の同時実行数まで性能向上が期待できるか確認するため、仮測定を行いました。 その結果をもとに本測定では参照系と更新系の同時実行数をそれぞれ変えて測定しています。

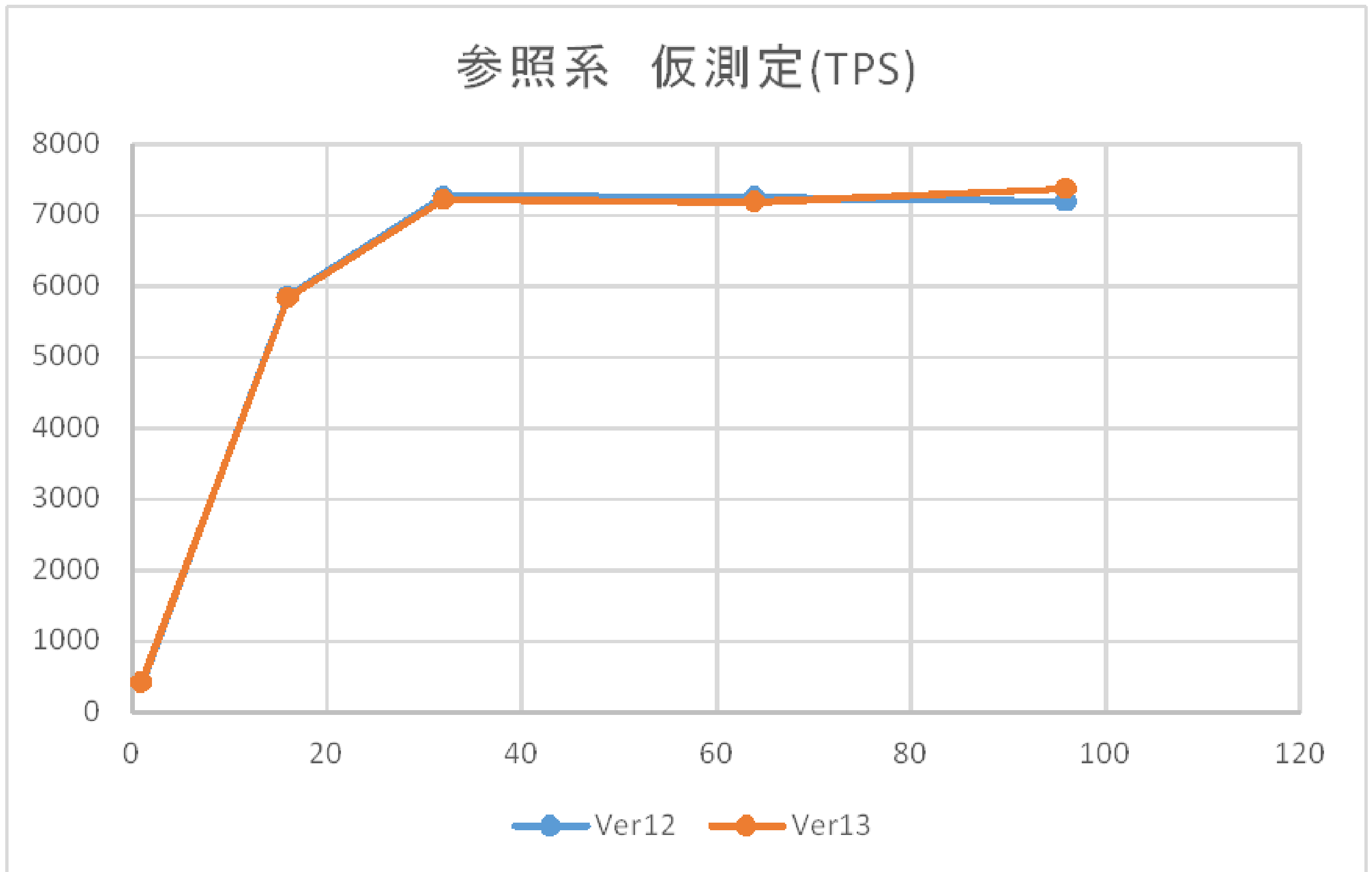
5.3.3.1. 参照系仮測定

WindowsOSにてPostgreSQL 12, 13 の2種類で実施し、 走行時間は300秒を指定しました。 同時実行数として1, 16, 32, 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。 また、測定の初めにselect pg_prewarm('pgbench_accounts')を実施しています。

```
\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。



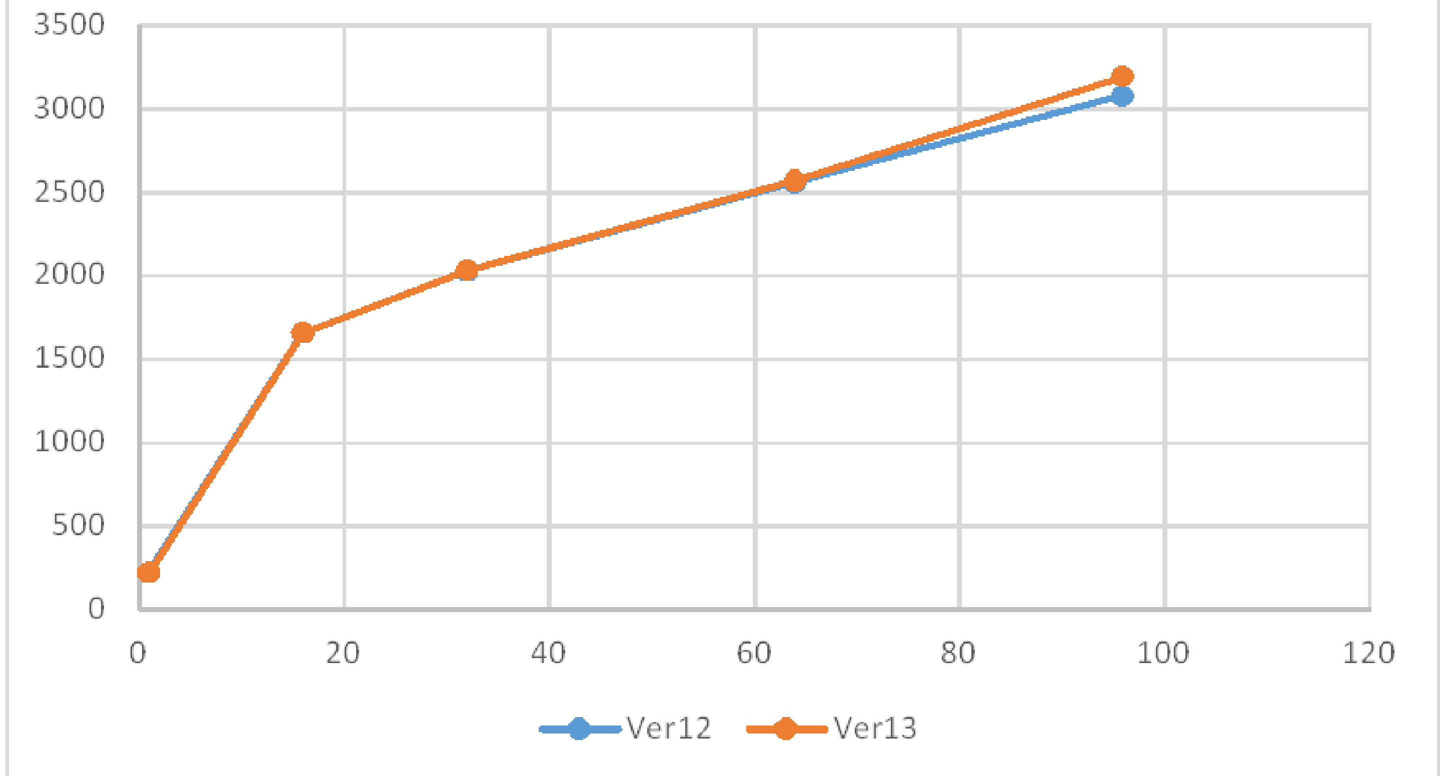
上記の通り、本環境では32接続でほぼ限界となっていることが確認できたため、本測定ではHTで40となるコア数付近の48接続を最大として実施することとしました。また、仮測定ではWindows版においても12と13の差がないことが見受けられます。

5.3.3.2. 更新系仮測定

WindowsOSにてPostgreSQL 12, 13 の2種類で実施し、 走行時間は300秒を指定しました。 同時実行数として1, 16, 32, 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。 また、測定の初めにselect pg_prewarm('pgbench_accounts')を実施しています。

```
\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。



上記の通り、本環境では96接続でもまだ限界を迎えている様子はないため、Linux版の検証と同様192接続を最大として実施することとしました。また、参照系同様に仮測定ではWindows版においても12と13の差がないことが見受けられます。

5.3.4. 本測定

本測定では測定時間を300秒とし、仮測定の結果をもとに 参照系と更新系の接続数をそれぞれ変えて実行しました。

5.3.4.1. 参照系測定フロー

仮測定と同一のカスタムクエリを走行時間は300秒で指定して実行しました。 同時実行数として1, 2, 4, 8, 16, 24, 32, 40, 48とし、この順番で3周分測定し、 その測定結果(tps)の中央値を測定値としています。 また、測定の初めにselect pg_prewarm('pgbench_accounts')を実施しています。

5.3.4.2. 更新系測定フロー

仮測定と同一のカスタムクエリを走行時間は300秒で指定して実行しました。 同時実行数として1, 16, 32, 64, 96, 128, 160, 192とし、この順番で3周分測定し、 その測定結果(tps)の中央値を測定値としています。 各同時接続数における300秒のクエリ実行後にcheck pointとvacuumを実施しており、 1週ごとに検証データの再作成とvacuum full,check point,ANALYZEを実施しています。 また、各周の測定の初めにselect pg_prewarm('pgbench_accounts')を実施しています。

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。

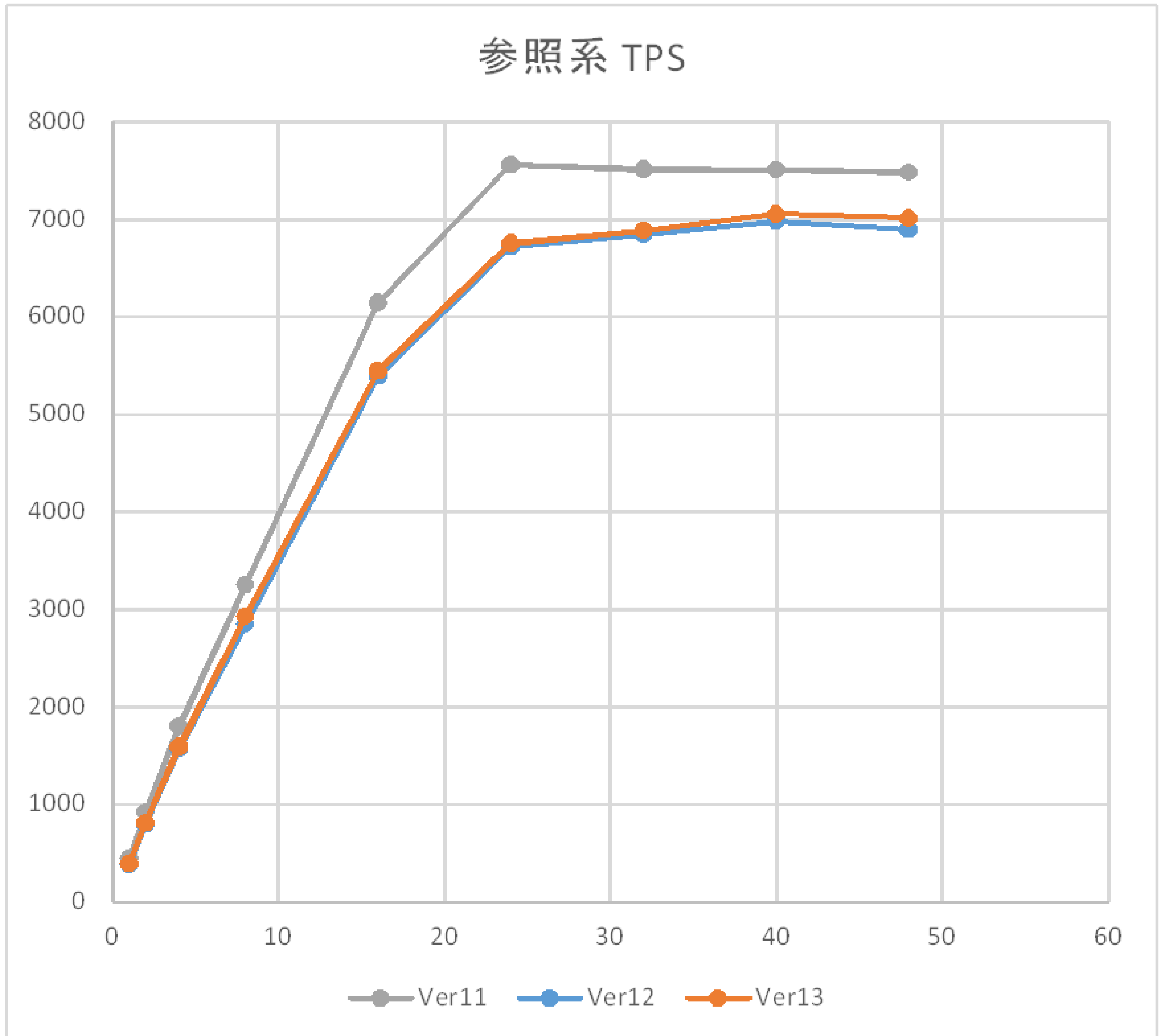
```
listen_addresses = '*' ... 負荷掛け用マシンからの接続用
max_connections = 500 ... 多めに設定
shared_buffers = 64GB ... システムメモリの25%
work_mem = 128MB
maitenance_work_mem = 2047MB ... Windows版PostgreSQLでは上限値がLinux版よりも小さく、測定環境の限界値を設定した。
checkpoint_timeout = 1d
max_wal_size = 100GB
effective_cache_size = 128GB ... システムメモリの50%
autovacuum = off
```

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。



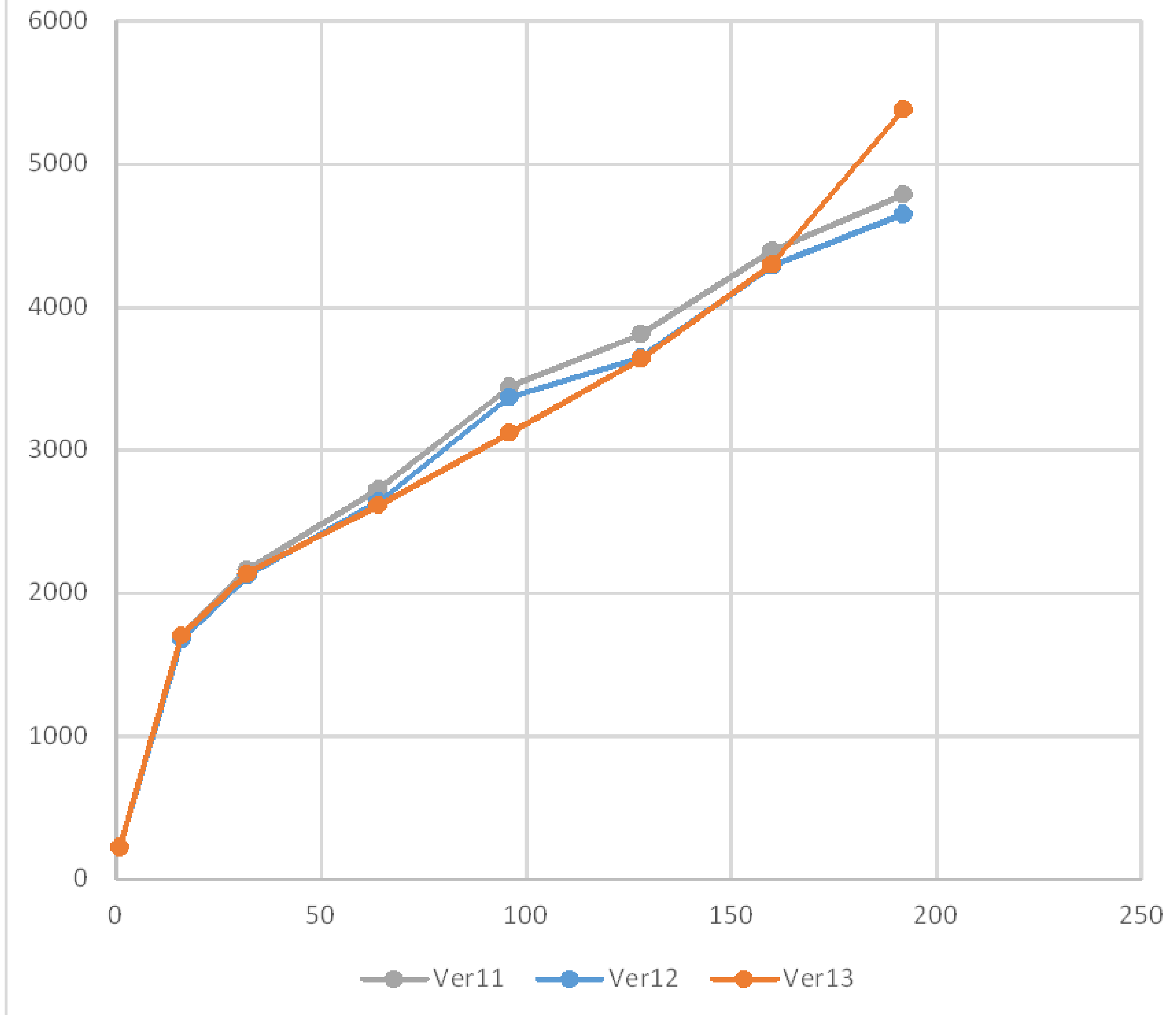
参照系については、同時接続数24までは、同時接続数を上げることでほぼ線形にtpsが増加していきます。 本検証環境は物理コア数が20、HyperThreadingで論理コア数を40としている環境であるため、 線形に増加するのは物理コア数付近、収束するのが論理コア数付近という 1接続1プロセスの動作仕様に沿った傾向を確認できます。 平均CPU使用率についても40接続で100%付近になることを確認できました。

その上で、Ver間比較についてはまず13と12の比較では差が認められず、 Linux版と同じ傾向であることを確認できました。

12と11の比較についても、2019年度検証活動においてLinux定点観測で確認された 12になって少し劣化するという傾向と一致しており、Table Access Methodの導入に伴うオーバヘッドはWindowsOSでも発生していることが改めて確認できました。

5.4.2. 更新系

更新系検証結果(tps)を以下に示します。



更新系に関しては、同時接続数192まではtpsのピークは迎えず、概ね線形に増加する傾向となりました。 11,12,13の各Ver間の差については多少の揺らぎはありつつも明らかな差はなく、 Linux版の定点観測における2019年度、2020年度の検証結果と同じ傾向であることを確認できました。

更新系が収束しない(微増になりつつも頭打ちしない)傾向は2019年度の検証からLinux版で確認できていますが、 2018年度検証ではWindows版はVer 11でも早い段階で頭打ちしていました。 Ver 11において2018年度と2020年度検証の違いはOS(2012と2019)とディスク(HDDとSSD)にあります。

パフォーマンスモニタによるリソース状況ではCPUは全体的に10%未満しか使いきれておらず、 I/Oネックとなっていることが予想されます。 WAL保存領域のAvg. Disk sec/Writeも同時接続数64で頭打ちとなりますが、 CPU使用率は僅かながらあがっていく傾向が見れます。

Linux版も2018年度検証(物理HDD)から2019年度検証(クラウド/SSD)に変わって 収束しづらい傾向が確認できており、更新系はWAL書き込み先のストレージ特性によって 同時接続数増加に対するTPSの増加傾向が変わる可能性が考えられます。

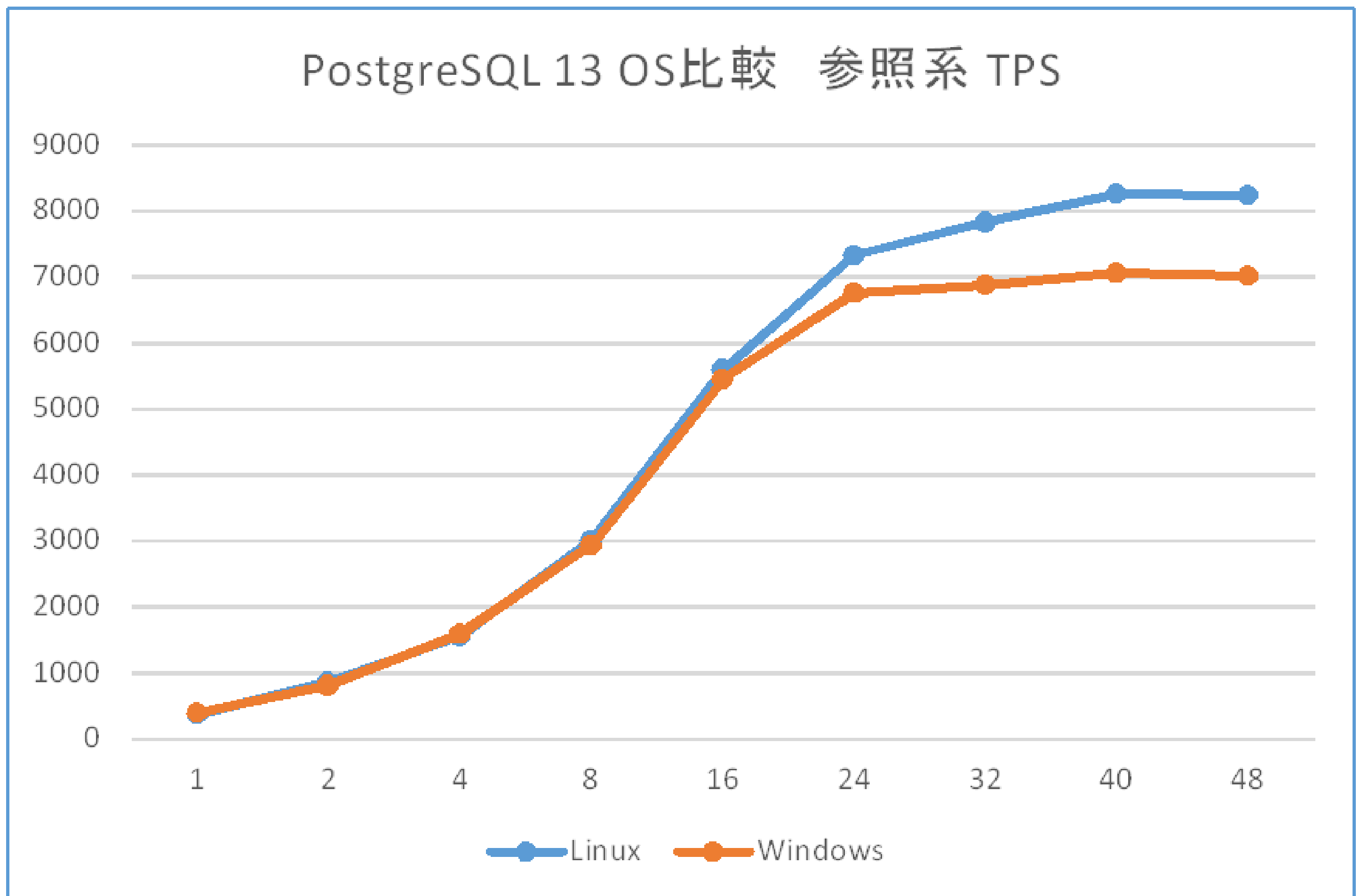
5.5. Linux WindowsOS 比較

本節では、同一サーバ上でのLinux版PostgreSQLとWindows版PostgreSQLの比較検証結果を示します。

DB serverにデュアルブートでインストールしたRHEL 8.3にOSを切り替えて、 データベースクラスタ構成や検証モデルと測定フロー、PostgreSQL設定などすべてWindowsOS版測定と 同条件としたうえでPostgreSQL 13での比較検証を実施しました。

5.5.1. 参照系

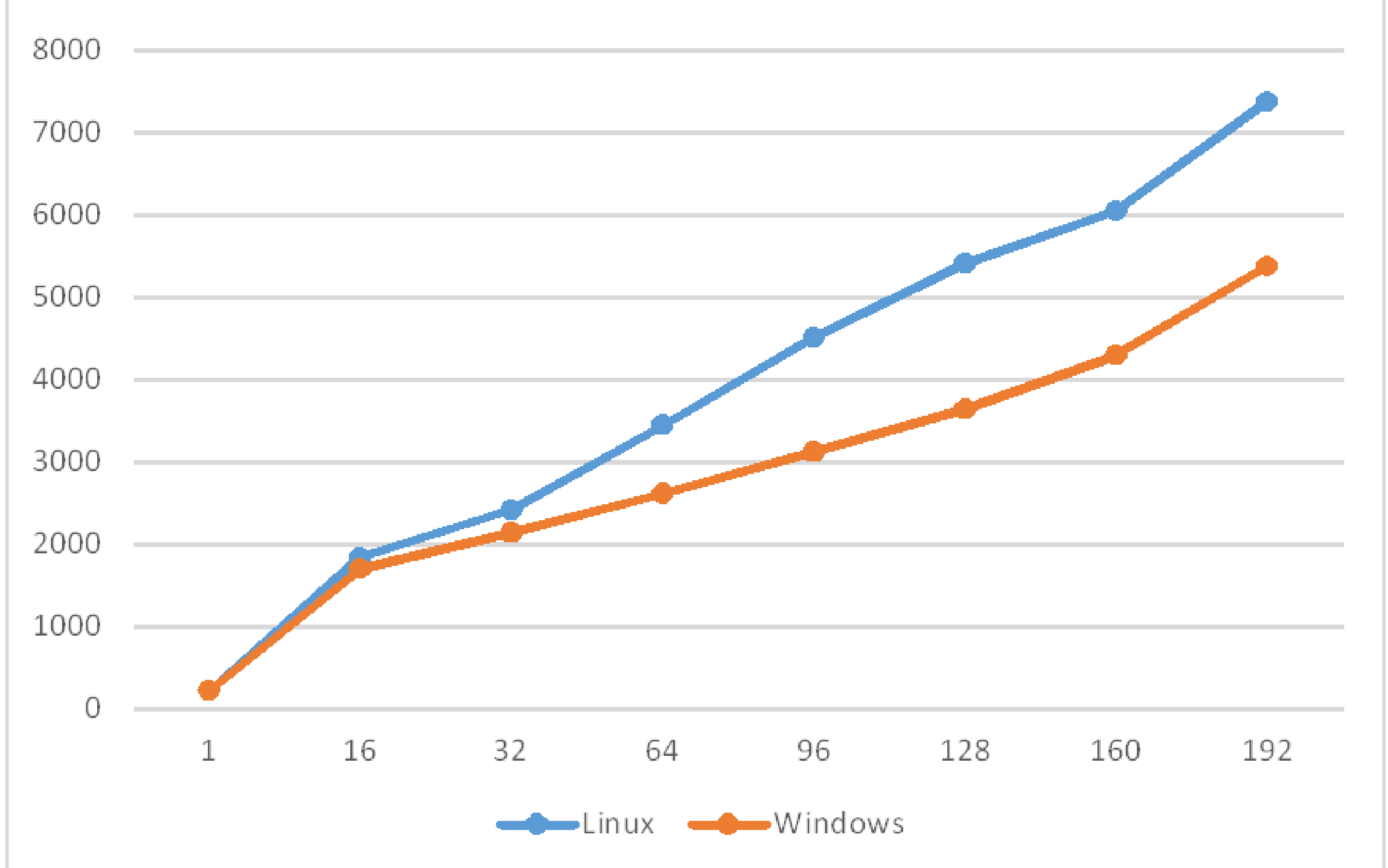
PostgreSQL 13における参照系 比較検証結果(tps)の比較を以下に示します。



同時接続数16までは同等の傾向かつ性能があると判断できますが、24接続を超えたところで Linux版が優位になり始めます。 CPU使用率についてはLinux版でもほぼ同等であり、参照系については事前にprewarmでオンメモリにした上で 測定を実施しているため、メモリの参照や管理になんらかの性能差が出ている可能性が考えられます。

5.5.2. 更新系

PostgreSQL 13における更新系 比較検証結果(tps)の比較を以下に示します。



傾向としては同時接続数192までtpsのピークは迎えず、概ね線形に増加するという点は同様ですが、 32接続から差が出始め、64接続以降は明確にLinuxの優位を確認できます。

WAL保存領域のI/Oについて、sarによる%utilは100%に張り付いておりディスクI/Oがネックであったことはそれぞれ同様ですが、CPU使用率についてはLinux版のほうがより低い傾向にあり(1%〜5%未満)ました。

OSによるファイルシステムの違いが性能差の原因となった可能性など考えられます。 本測定ではデータディレクトリ、WAL保存先ともにWindows版はNTFS、Linux版はxfs(inode64)で設定しました。

5.6. まとめ

定点観測モデルを用いて、Windows版PostgreSQLのVer間比較検証と、Linux版PostgreSQLに対する比較検証をそれぞれ実施しました。

Ver間比較検証について、参照系/更新系ともにLinux版と同じ特性と傾向であることを改めて確認できました。 11と12ではTable Access Methodの機能追加の影響を受けたこと、12と13では性能への影響はないこと、 同時接続数とTPS増加の傾向などもLinux版と変わりはないと判断できます。

対して、同一スペック、同一条件でのLinux版との比較については参照系/更新系ともにLinux版が優位であることが確認されました。 同時接続数増加に対するTPSの増加傾向や参照系における頭打ちのタイミングなど性能特性は似ていますが、 スループットの絶対値については参照/更新ともにLinuxが優位という結果です。

Linuxが優位となる性能傾向については、2020年度のパラレルクエリ検証にてスタースキーマベンチマークの測定結果でも確認されています。 参照系についてはオンメモリでも差が出るためメモリ管理について差があること、 更新系はファイルシステムによる性能の違いがあるのではないかと見込んではいますが、 その仮説を裏付けるためのOSリソース値の取得であったり、メモリとファイルシステムそれぞれに特化したベンチマーク検証の考案が今後の課題と考えます。

6. パラレルクエリ 性能検証

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。 今年度は過去の検証では検証できていなかったパラレルインデックススキャン機能で性能がどのように変化するかを検証します。 また、比較検証として、Windows版PostgreSQLでのパラレルクエリの性能との比較、テーブルパーティショニング構成との比較についても検証をします。

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。

6.1. パラレルクエリとは

6.1.1. 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。

6.1.2. PostgreSQLでのパラレルクエリ

2021年3月現在、PostgreSQL 11、PostgreSQL 12およびPostgreSQL13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 6.1 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。

6.1.3. 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 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)	デフォルト値(11以降)	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	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。

6.2. Star Schema Benchmarkとは

本検証ではStar Schema Benchmarkを利用して、パラレルクエリの性能を評価しています。

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、ディメンションテーブル、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。

6.2.1. 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。 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

# データ生成ツールをビルドする
$ 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 ssb -c "copy customer from STDIN ( delimiter '|');"
```

Star Schema Benchmark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シ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。 ディメンシ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。 Star Schema Benchmark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Star Schema Benchmarkのディメンションテーブルには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。

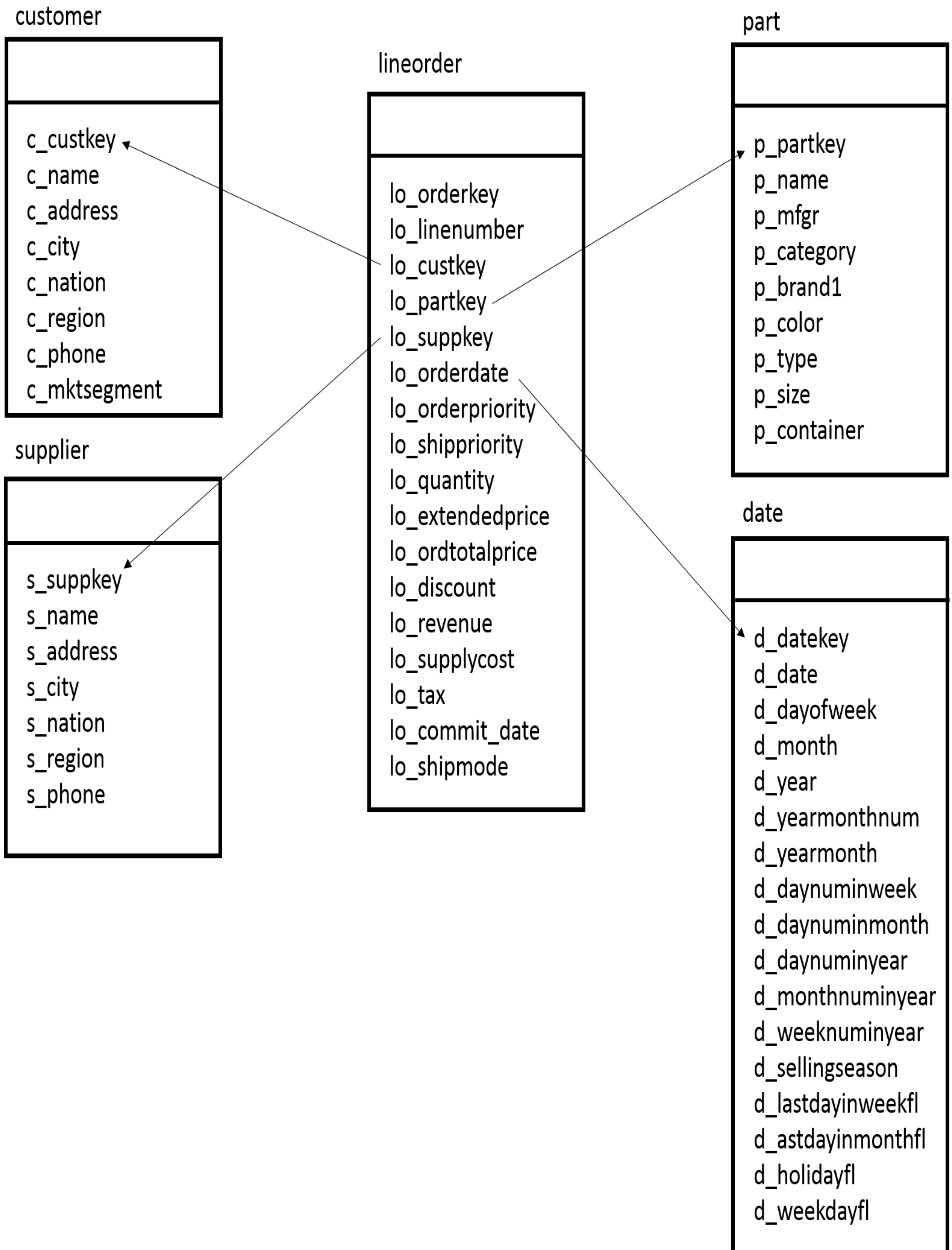


図 6.1 Star Schema Benchmarkのテーブル構成

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。 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。 Star Schema Benchmark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 6.3 検証に使用するテーブルの規模

テーブル名	行数(理論値)	行数(実測値)	データサイズ(実測値) [Bytes]	データサイズ(実測値) [GB]
customer	SF × 30,000	2,999,825	374,030,336	0.348
date	2,556	2,556	311,296	0.00029
lineorder	SF × 6,000,000	600,044,480	63,557,664,768	59.193
part	200,000 × (1+log2(SF)) (小数点切り捨て)	1,400,000	164,265,984	0.153
supplier	SF × 10,000	1,000,000	114,376,704	0.107

6.2.2. クエリについて

本検証では、Star Schema Benchmarkで定義されるクエリを使用しています。

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す。 以下に、Star Schema Benchmarkで定義されるクエリと、クエリの意味を示します。

- 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_datekey 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_datekey 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_datekey and
     d_weeknuminyear = 6 and
     d_year = 1994 and
     lo_discount between 5 and 7 and
     lo_quantity between 26 and 35;
```

- 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;
```

- 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;
```

- 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;
```

6.3. パラレルインデックススキャン性能検証

6.3.1. 検証概要

本検証にて、PostgreSQL のパラレルインデックススキャンの性能について確認を行います。

インデックスを利用したアクセス(実行計画)となるクエリについて Worker 起動数を変化させることにより下記を確認することを目的とします。

1. Worker 数により性能向上すること
2. Worker 起動数の要素

max_worker_processes を想定起動 Worker 数以上の 16 で固定し、max_parallel_workers, max_parallel_workers_per_gather を変化させて Worker 起動数を変動させ、クエリ実行性能を確認します。 max_parallel_workers, max_parallel_workers_per_gather を 0 に設定した場合パラレルクエリ未使用となります。

6.3.2. 検証構成

6.3.2.1. マシン構成

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 (NEC Express 5800)	CPU	インテル Xeon E5-2630 v4 @ 2.20GHz 10コアx2 合計 20コア
	搭載メモリ	256GB
	内蔵ストレージ装置	SSD 400GB SATA x 8 (RAID 5)

6.3.2.2. SW構成

検証環境のソフトウェア構成を示します。

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 8.3
Database	PostgreSQL 13.1

6.3.2.3. 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。

表 6.6 デフォルトから変更したパラメータ

パラメータ	設定値
max_connections	20
shared_buffers	100GB
effective_cache_size	100GB
work_mem	104857kB
maintenance_work_mem	2GB
wal_buffers	16MB
max_wal_size	16GB
min_wal_size	4GB
effective_io_concurrency	200
checkpoint_completion_target	0.9
random_page_cost	1.1
default_statistics_target	500
max_worker_processes	16
max_parallel_workers [5]	16
max_parallel_workers_per_gather [5]	16
max_parallel_maintenance_workers	4

[5] (1, 2) 測定により値を変更して実施します。

6.3.3. 検証内容

測定に際しては、SQL の実行前に pg_prewarm を利用してディスクからデータを一度読み込み、PostgreSQL のバッファ上に常駐させた状態にします。 また、VACUUM ANALYZE により不要領域を回収して Visibility Map を all-visible に設定するとともに、統計情報を更新した状態にします。

6.3.3.1. Count クエリ

Worker 数の変化によるパラレルインデックススキャン性能向上を確認するため、単純なクエリとして Star Schema Benchmark のファクト表である lineorder の件数カウントを計測します。

```
SELECT COUNT(*) FROM LINEORDER
```

6.3.3.2. Star Schema Benchmark

パラレルインデックスの応用として、実トランザクションでの性能向上度を確認するため Star Schema Benchmark データに Index を作成して計測します。

6.3.3.2.1. Index 付与

Scale Factor = 100GB で作成したデータセットに対し、ファクト表とディメンション表の結合列(外部キー)となっている列に B-Tree 索引を作成します。

表 6.7 SF=100GB の Star Schema Benchmark データ Table/Index サイズ

Table	Index	MB
lineorder		60,613.3
	idx_lo_pkey	12,853.5
	idx_lo_partkey	4,042.3
	idx_lo_custkey	4,068.4
	idx_lo_suppkey	4,042.7
	idx_lo_orderdate	3,966.5
customer		356.7
	idx_c_custkey	64.3
part		156.7
	idx_p_partkey	30.0
supplier		109.1
	idx_s_suppkey	21.4
date		0.3
	idx_d_datekey	0.1

6.3.3.2.2. クエリ

本検証では、Star Schema Benchmarkで定義されているクエリを利用しています。

6.3.4. 検証結果

6.3.4.1. Count クエリ

Star Schema Benchmark の lineorder 表のカウント結果で、Worker 数の増加により性能向上する様子が確認できました。

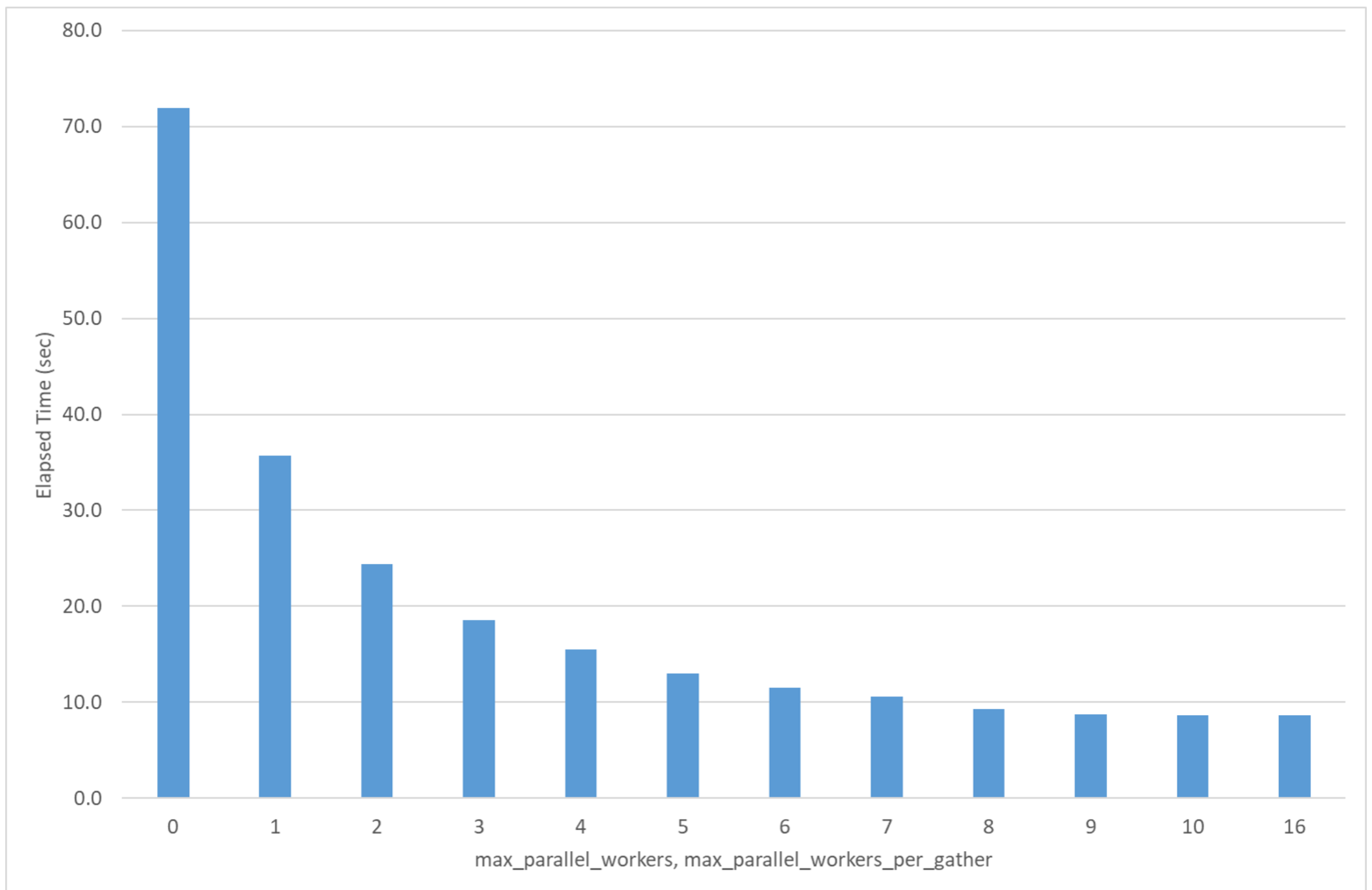


図 6.2 処理時間の計測結果

表 6.8 各 Worker 数での処理時間と性能向上比率

max_parallel_workers max_parallel_workers_per_gather	Elapsed Time (sec)	性能向上比率
0	71.9	1.0
1	35.7	2.0
2	24.4	2.9
3	18.6	3.9
4	15.5	4.6
5	13.0	5.5
6	11.5	6.2
7	10.6	6.8
8	9.3	7.7
9	8.7	8.3
10	8.6	8.4
16	8.6	8.4

また、Worker はインデックスサイズによって要求される起動数が決定されていることが確認できました。

lineorder の件数カウントに利用される idx_lo_orderdate はサイズが 3,966.5MB であり、 想定通り 9プロセスが起動されることを確認しています。

- max_parallel_workers=10, max_parallel_workers_per_gather=10 での実行計画

```
QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=4393067.24..4393067.25 rows=1 width=8)
   ->  Gather  (cost=4393066.31..4393067.22 rows=9 width=8)
         Workers Planned: 9
         ->  Partial Aggregate  (cost=4392066.31..4392066.32 rows=1 width=8)
               ->  Parallel Index Only Scan using idx_lo_orderdate on lineorder  (cost=0.57..4225389.04 rows=66670905 width=0)
```

- Woker 起動数

表 6.9 インデックスサイズによるワーカー数

サイズ(MB)	起動ワーカー数	検索プロセス数
0.5	1	2
1.5	2	3
4.5	3	4
13.5	4	5
40.5	5	6
121.5	6	7
364.5	7	8
1,093.5	8	9
3,280.5	9	10
9,841.5	10	11

6.3.4.2. Star Schema Benchmark

6.3.4.2.1. Index あり状態の結果

Star Schema Benchmark にインデックスを付与すると、各クエリはインデックスを利用した実行計画とはなりますが、Nested Loop Join の内部表として利用されるため、パラレルインデックススキャンとはなりませんでした。 パラレルクエリとしては動くため、Worker 数は Nested Loop Join の外部表のとして選択されたテーブルサイズに依存して上限が決定されています。

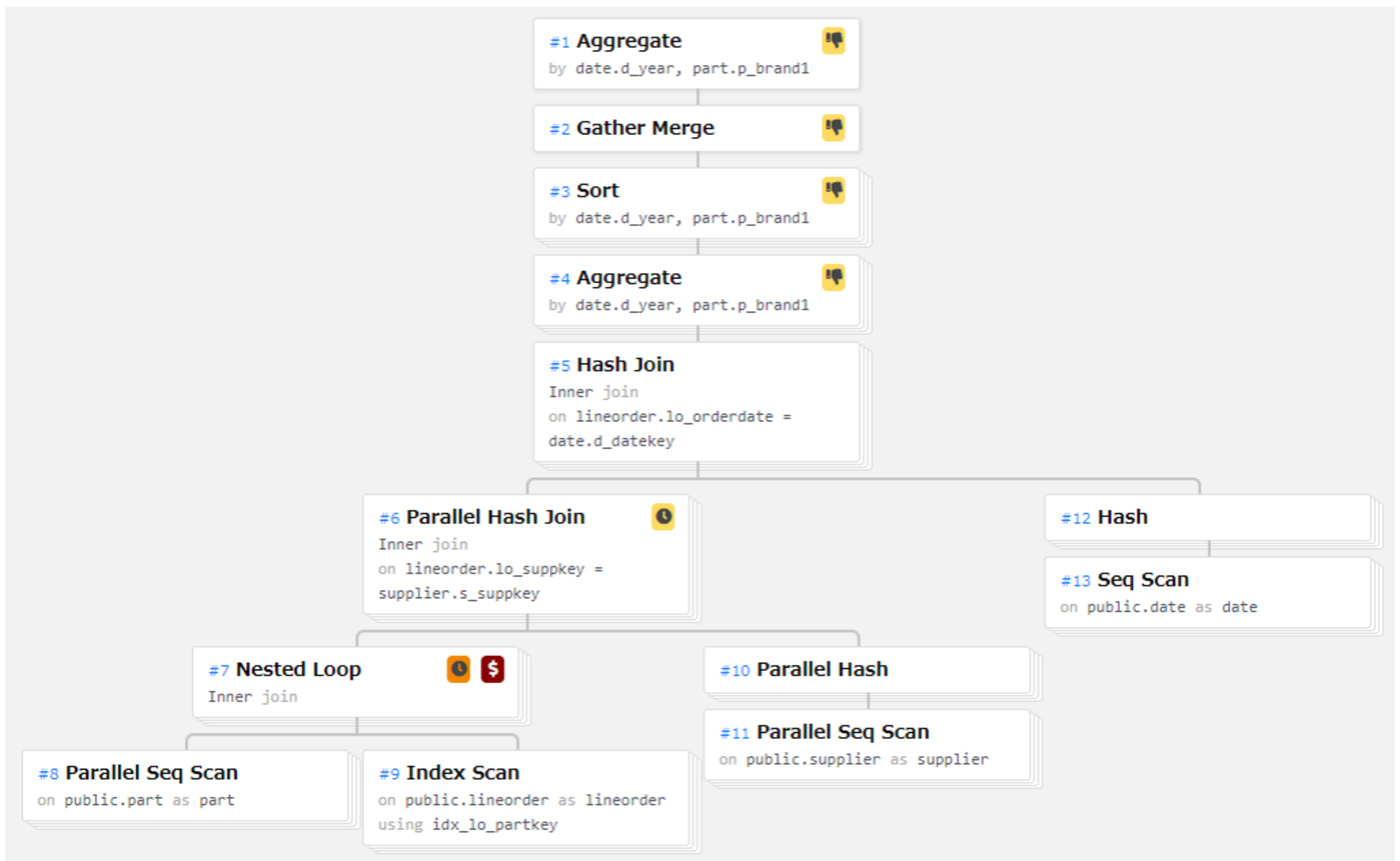


図 6.3 実行計画の例(q2.1)

6.3.4.2.2. Merge Join の強制によるパラレルインデックスキャンの利用

パラレルインデックススキャンが選択された場合の挙動を確認するため、Nested Loop Join / Hash Join が選択されない状態に設定し Merge Join を強制した場合にはパラレルインデックススキャンが選択されることを確認しました。

ただし、Merge Join を強制しない通常の実行計画と比較すると処理実行速度は遅い結果となっています。

DB に以下パラメータを追加設定しています。

- enable_nestloop = off
- enable_hashjoin = off

Merge Join が選択された実行計画では、パラレルインデックススキャンが実行されている様子が確認できます。

#1 Aggregate
by date.d_year, part.p_brand1
#2 Gather Merge
#3 Aggregate
by date.d_year, part.p_brand1
#4 Sort
by date.d_year, part.p_brand1
#5 Merge Join
Inner join
#6 Sort
by lineorder.lo_orderdate
#7 Merge Join
Inner join
#8 Sort
by lineorder.lo_suppkey
#9 Merge Join
Inner join
#10 Parallel Index Scan
on public.lineorder as lineorder
using idx_lo_partkey
#11 Materialize
#12 Index Scan
on public.part as part
using idx_p_partkey
#13 Sort
by supplier.s_suppkey
#14 Seq Scan
on public.supplier as supplier
#15 Sort
by date.d_datekey
#16 Seq Scan
on public.date as date

図 6.4 Merge Join を強制した場合の実行計画の例(q2.1)

各クエリともインデックスのサイズに応じで Worker 数が決定され、起動数に応じて性能向上が見られました。

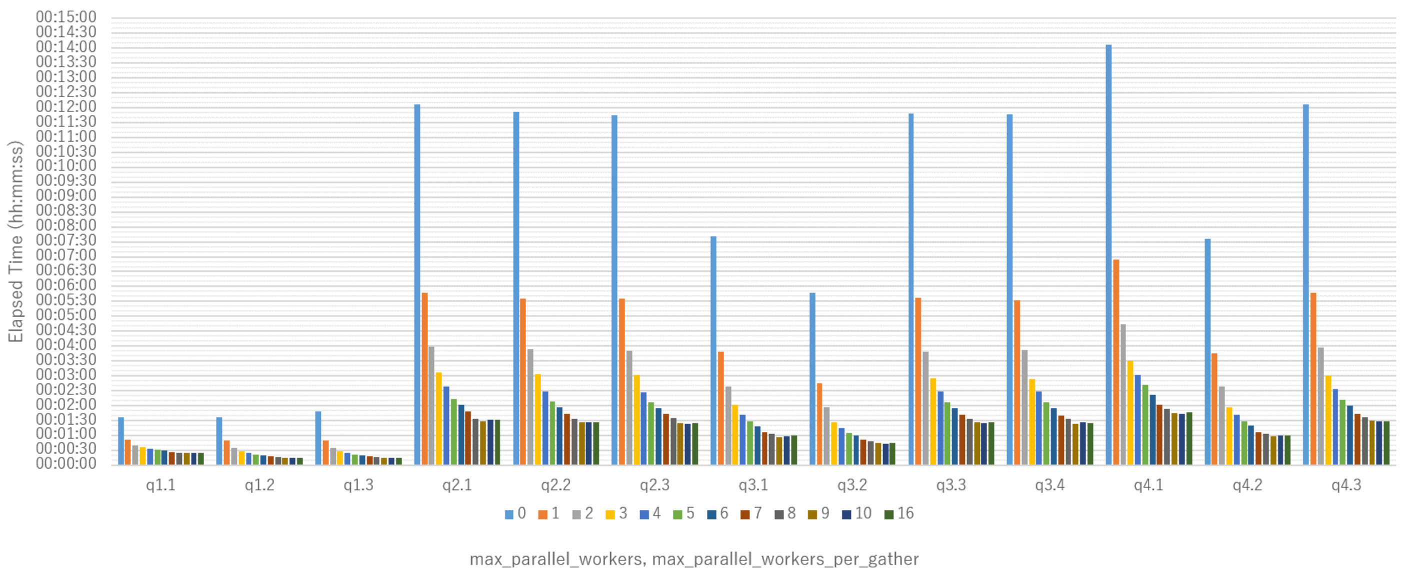


図 6.5 処理時間の計測結果

6.3.5. 補足 - Star Schema Benchmark クエリでの INDEX 有無による性能比

Star Schema Benchmark にインデックスを付与した場合の性能差を比較します。

パラレルクエリ未使用の場合(Worker 数が 0 の場合)は、各クエリともインデックスありの場合の方が処理実行速度は速く、インデックスの有効性が確認できました。

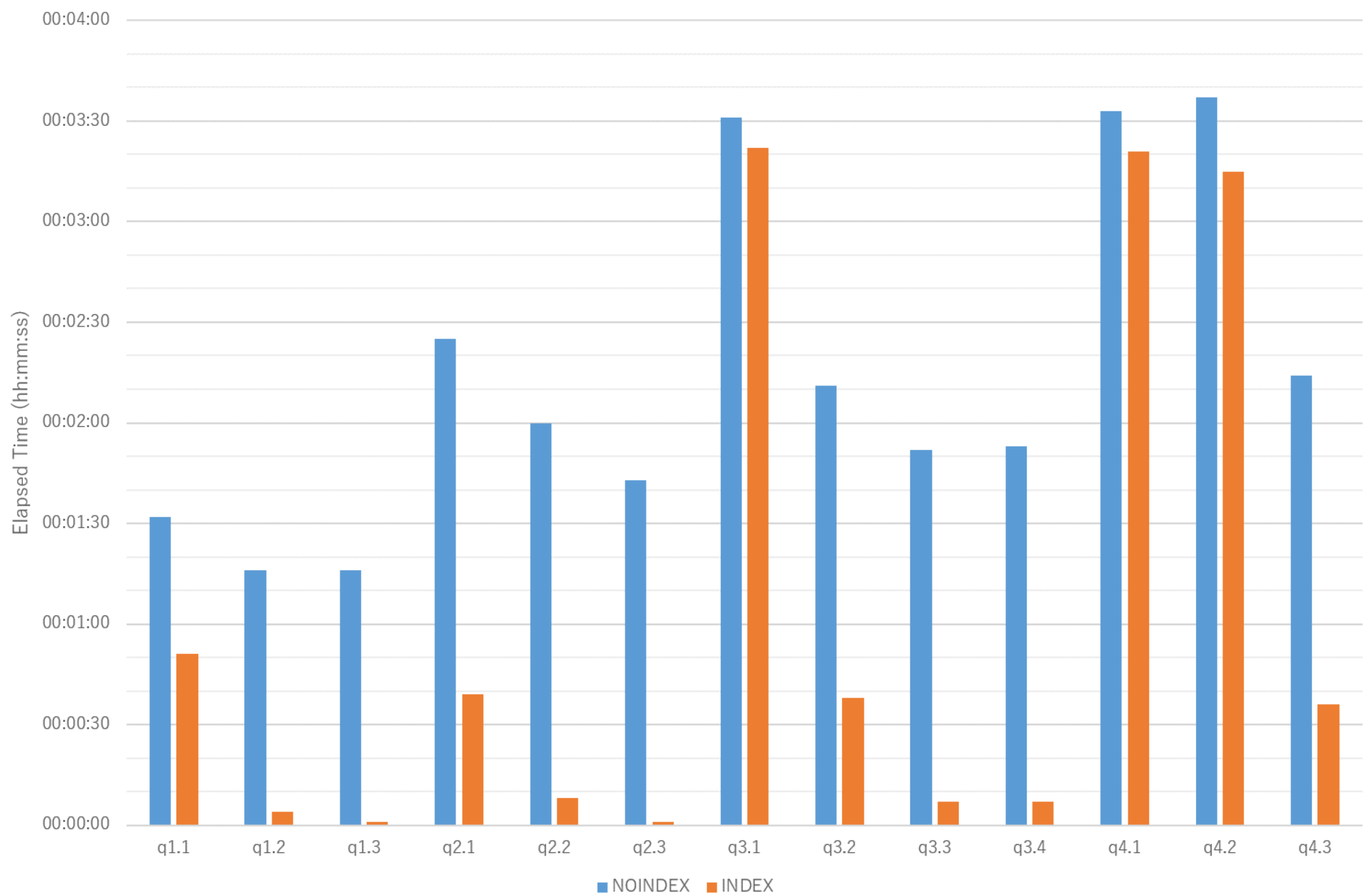


図 6.6 処理時間の計測結果(No parallel query)

ただし、パラレルクエリを使用した場合、クエリによってはインデックスありの場合の方が処理速度が遅いケースが見られました(q3.1, q4.1, q4.2)。

これは、選択された実行計画で最初にアクセスするテーブルやインデックスのサイズに依存して Worker 数が決定されるため、 インデックスなしの場合よりもサイズの小さなテーブルやインデックスからアクセスする実行計画となった際には Worker 数の上限が低くなり、 処理速度の向上も頭打ちとなってしまうためと考えられます。

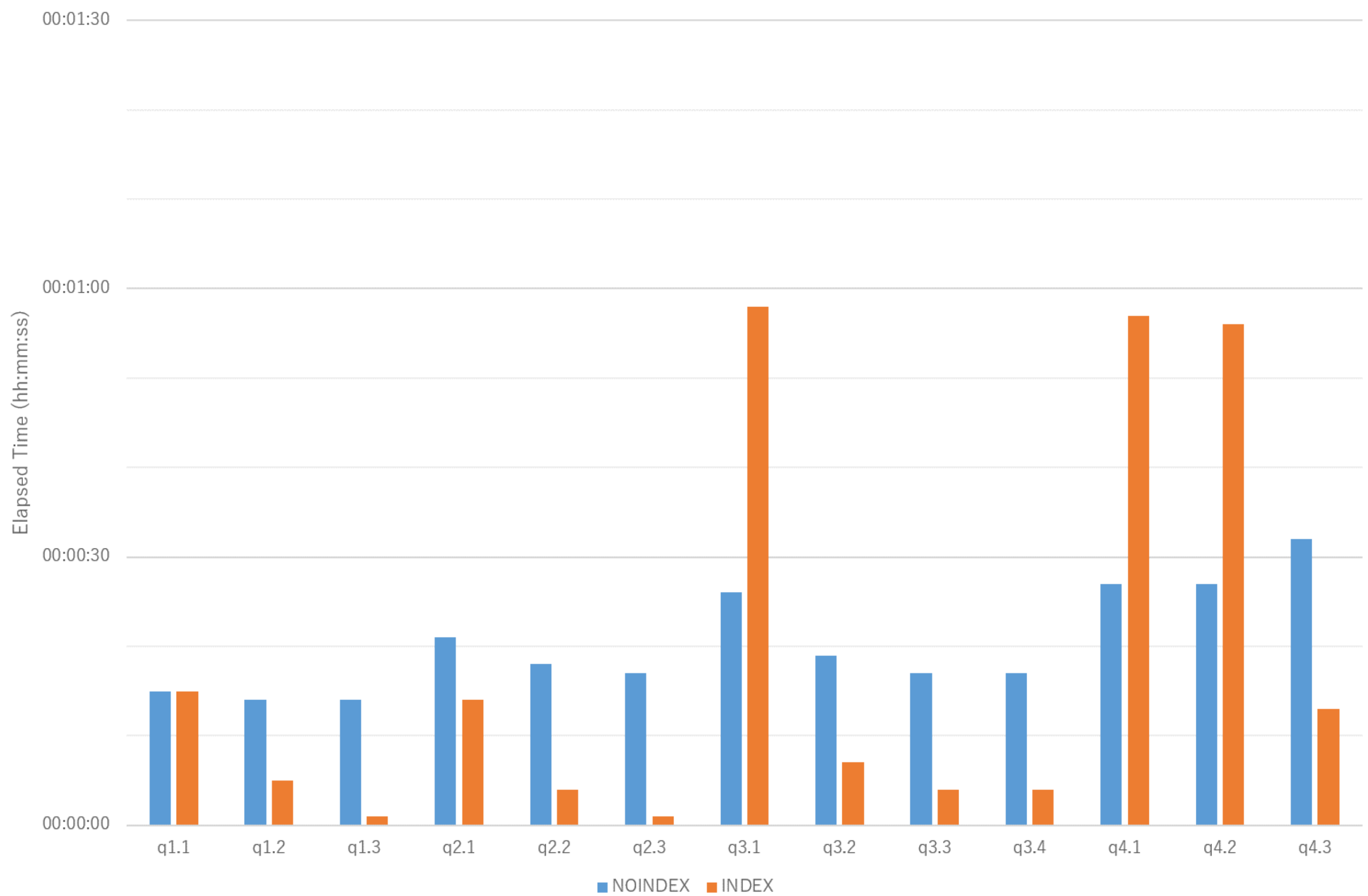


図 6.7 処理時間の計測結果(parallel query)

6.3.6. まとめ

6.3.6.1. パラレルインデックススキャンの効果

実行計画としてパラレルインデックススキャンが選択されるクエリについて、Worker 数に応じて性能向上する点、インデックスのサイズにより起動する Worker 数が決定される点が確認できました。

本検証では、パラレルクエリが有効な場合、インデックスを利用しない方が高速となるケースもあったため、Worker 起動数とともにインデックス要否の判断がチューニングのポイントとなります。

6.3.6.2. 注意点

6.3.6.2.1. Parallel Index Only Scan 利用時の注意点：VACUUM の必要性

Index Only Scan は参照するページに変更されたデータがあると利用できません。 クエリ実行前に VACUUM ANALYZE により Visibility Map(VM:可視性マップ)が all-visible となっている必要があります。

本検証で確認した Parallel Index Only Scan も、Visibility Map が all-visible の状態ではなく、不可視ページがある場合にはテーブルへの参照が発生し性能劣化する可能性がある点に注意が必要です。

6.3.6.2.2. パラレルインデックススキャンとインデックススキャンのパラレル実行の違い

本検証ではパラレルインデックススキャンの性能を目的として確認しましたが、これは、1回のインデックス走査を Worker で分担する動作となります。 一方、Nested Loop Join の内部表としてインデックスが利用される場合、インデックススキャンのパラレル実行が行われますが、これは、何度も発生するインデックス走査を Worker で分担する動作となります。 いずれもパラレルクエリであり、性能向上が見込まれるため、必ずしもパラレルインデックススキャン(Parallel Index Scan または Parallel Index Only Scan)が選択されなければインデックスアクセスが高速化されないわけではありません。

- パラレルインデックススキャン(Count クエリ)のケースでの実行計画抜粋

```
-> Parallel Index Only Scan using idx_lo_orderdate on public.lineorder
     (cost=0.57..4225389.04 rows=66670905width=0) (actual time=0.053..26609.216 rows=60003814 loops=10)
   Output: lo_orderdate
   Heap Fetches: 0
   Worker 0:  actual time=0.046..26816.741 rows=61381559 loops=1
   Worker 1:  actual time=0.046..26698.880 rows=59206752 loops=1
   Worker 2:  actual time=0.050..26714.891 rows=60687709 loops=1
   Worker 3:  actual time=0.041..26774.976 rows=59185760 loops=1
   Worker 4:  actual time=0.066..26203.609 rows=59597155 loops=1
   Worker 5:  actual time=0.045..26471.542 rows=59663385 loops=1
   Worker 6:  actual time=0.044..26637.349 rows=60829885 loops=1
   Worker 7:  actual time=0.046..26254.571 rows=60829602 loops=1
   Worker 8:  actual time=0.044..26772.008 rows=57950139 loops=1
```

idx_lo_partkey へのアクセスで 60,003,814行のデータをカウントしているが、Worker + 1 の10プロセスで分担して各1回ずつアクセスとなっている。

- インデックススキャンのパラレル実行(Star Schema Benchmark で Nested Loop Join が選択されるクエリ)のケースでの実行計画抜粋

```
-> Nested Loop  (cost=0.44..331528.46 rows=774285 width=21) (actual time=46.857..2666072.754 rows=598740 loops=4)
   Output: lineorder.lo_revenue, lineorder.lo_orderdate, lineorder.lo_suppkey, part.p_brand1
   Worker 0:  actual time=89.215..2666065.374 rows=600791 loops=1
   Worker 1:  actual time=36.373..2666055.700 rows=610096 loops=1
   Worker 2:  actual time=26.764..2666099.598 rows=572545 loops=1
   -> Parallel Seq Scan on public.part  (cost=0.00..14684.81 rows=10326 width=13) (actual time=19.352..430.546 rows=7970 loops=4)
      Output: part.p_partkey, part.p_name, part.p_mfgr, part.p_category, part.p_brand1, part.p_color, part.p_type, part.p_size, part.p_container
      Filter: ((part.p_category)::text = 'MFGR#12'::text)
      Rows Removed by Filter: 192030
      Worker 0:  actual time=19.202..410.377 rows=8180 loops=1
      Worker 1:  actual time=19.367..551.952 rows=8054 loops=1
      Worker 2:  actual time=19.326..424.537 rows=7544 loops=1
   -> Index Scan using idx_lo_partkey on public.lineorder  (cost=0.44..29.68 rows=100 width=16) (actual time=5.296..334.383 rows=75 loops=31
      Output: lineorder.lo_orderkey, lineorder.lo_linenumber, lineorder.lo_custkey, lineorder.lo_partkey, lineorder.lo_suppkey, ...
      Index Cond: (lineorder.lo_partkey = part.p_partkey)
      Worker 0:  actual time=5.128..325.821 rows=73 loops=8180
      Worker 1:  actual time=5.180..330.898 rows=76 loops=8054
      Worker 2:  actual time=5.572..353.293 rows=76 loops=7544
```

idx_lo_partkey へのアクセスが 31,882回実行されており、Worker + 1 の4プロセスで分担して各 8,000回前後ずつアクセスしている。 また、Parallel Seq Scan の各 Worker の rows の数値と、Index Scan の各Worker の loops の値が一致しており、SeqScan のヒット1行に対して、IndexScanが1回呼ばれている。

6.4. Windowsでのパラレルクエリ性能検証

6.4.1. 検証概要

本検証では、Windows版のPostgreSQLのパラレルクエリの性能の確認を行います。

Star Schema Benchmarkを使用し、同条件のLinux版での結果と比較を行い、性能差の有無を確認します。 比較は、 *EXPLAIN (ANALYZE, COSTS, VERBOSE, BUFFERS, FORMAT JSON)* での実行結果について評価を行います。

max_worker_processes を想定起動 Worker 数以上の 16 で固定し、PostgreSQLから要求される数の Worker を起動した状態でのクエリ実行性能を確認します。

6.4.2. 検証構成

6.4.2.1. マシン構成

検証環境のハードウェア構成の概略を示します。

表 6.10 ハードウェア構成

機器	インスタンスタイプ	vCPU	搭載メモリ	ディスク
Azure Virtual Machine	E16s_v4	16	128GiB	200GiB(Premium SSD)

6.4.2.2. SW構成

検証環境のソフトウェア構成を示します。

表 6.11 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	CentOS-based 8.2, Windows Server 2019
Database	PostgreSQL 13.1

6.4.2.3. 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。

Windows版では JITコンパイルは使用できないため、Linux版では無効にした状態で測定を行っています。

表 6.12 デフォルトから変更したパラメータ

パラメータ	設定値
max_connections	20
shared_buffers	96GB
effective_cache_size	96GB
maintenance_work_mem	2GB
checkpoint_completion_target	0.9
wal_buffers	16MB
random_page_cost	1.0
work_mem	1GB
min_wal_size	4GB
max_wal_size	16GB
max_worker_processes	16
max_parallel_workers_per_gather	16
max_parallel_workers	16
max_parallel_maintenance_workers	0
jit	off

6.4.2.3.1. 測定手順

Star Schema Benchmarkで定義される全クエリを、測定用のPythonスクリプトを用いて実行します。 測定は、各測定パターンにおいて、3回ずつ実施しました。データベースへの同時接続数は 1 とし、複数のクライアントからはクエリを実行していません。

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ています。

6.4.2.4. データモデル

本検証では、Star Schema Benchmarkで定義されているデータモデルを利用しています。

6.4.2.5. クエリ

本検証では、Star Schema Benchmarkで定義されているクエリを利用しています。

6.4.3. 結論

Windows版とLinux版のPostgreSQLにおいて、パラレルクエリ自体には性能差がないことを確認しました。

6.4.4. 測定結果

以下、本検証での測定結果について説明します。

6.4.4.1. Star Schema Benchmarkの結果

q4.3を除いて、全ての結果においてLinux版が性能が良いという結果になりました。 q4.3については、Windows版で起動したパラレルワーカーの要求数が少なかったために生じた差分であることを確認しています。 本事象は 2018年度の成果物 で報告されたものと同様と考えられます。

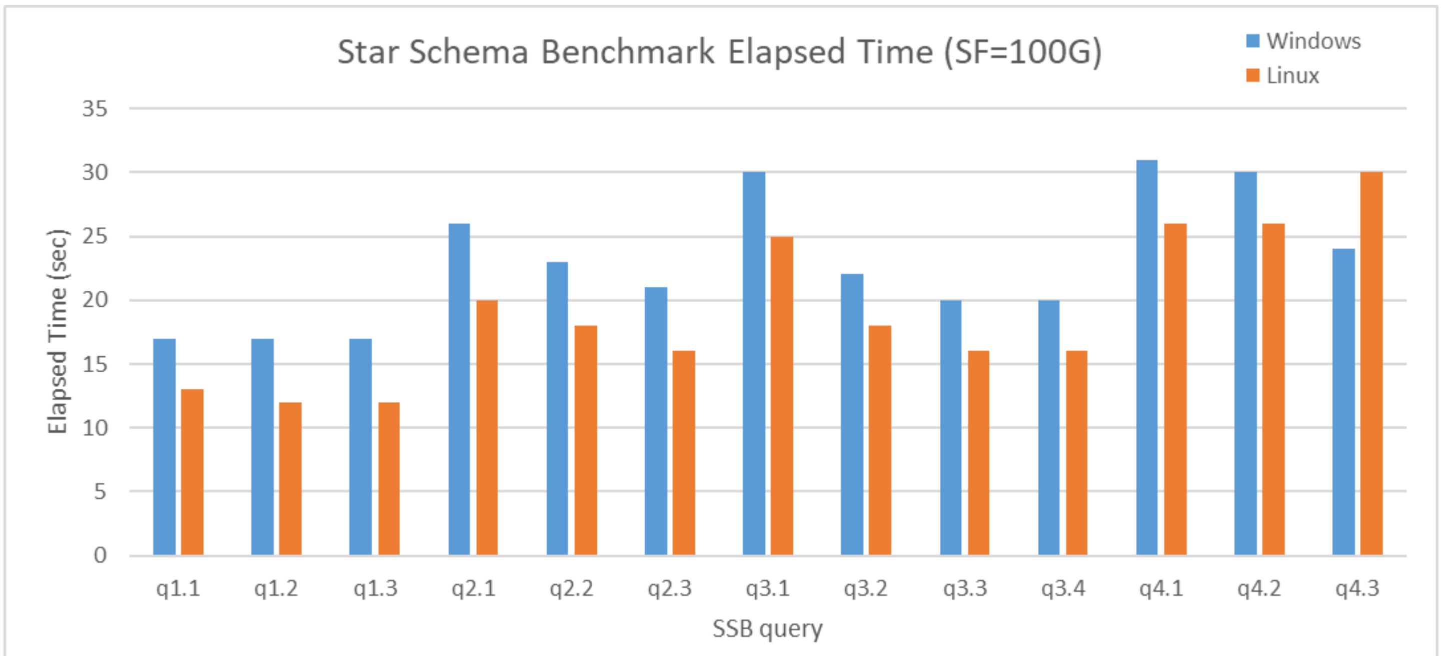


図 6.8 Execution time on Star Schema Benchmark [sec]

なお、全体的にLinux版のほうが性能が良いのは、パラレルクエリ機能を無効にしても同様の傾向であることが確認できました。 パラレルクエリを無効化した設定（max_parallel_workers_per_gather = 0）でStar Schema Benchmarkを実行した結果は以下のとおりです。

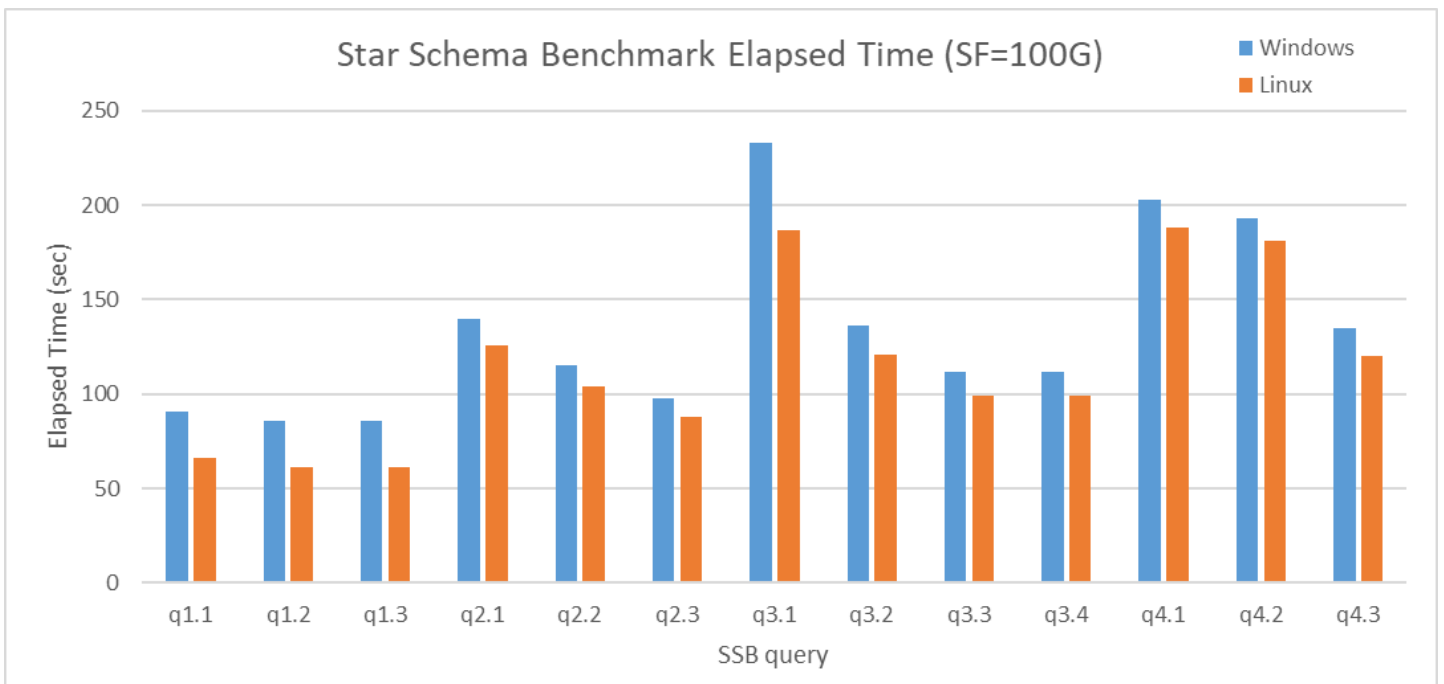


図 6.9 Execution time on Star Schema Benchmark (No parallel query) [sec]

パラレルクエリ有効時と無効時で各クエリにおいて、およそ30%程度の性能差となっていることから、Windows版とLinux版においてパラレルクエリの性能差はないものと考えられます。

6.4.4.2. Windows版での性能差について

Windows版、Linux版の実行計画の比較から、SeqScanの処理時間に差があることがわかりました。

以下は、パラレルクエリを有効にした状態でのq1.1の実行計画ですが、Windows版は、Linuxに比べて30%程度遅い結果となりました。 なお、この性能差はパラレルクエリを無効化した状態でも同等の差分（30%の劣化）となりました。

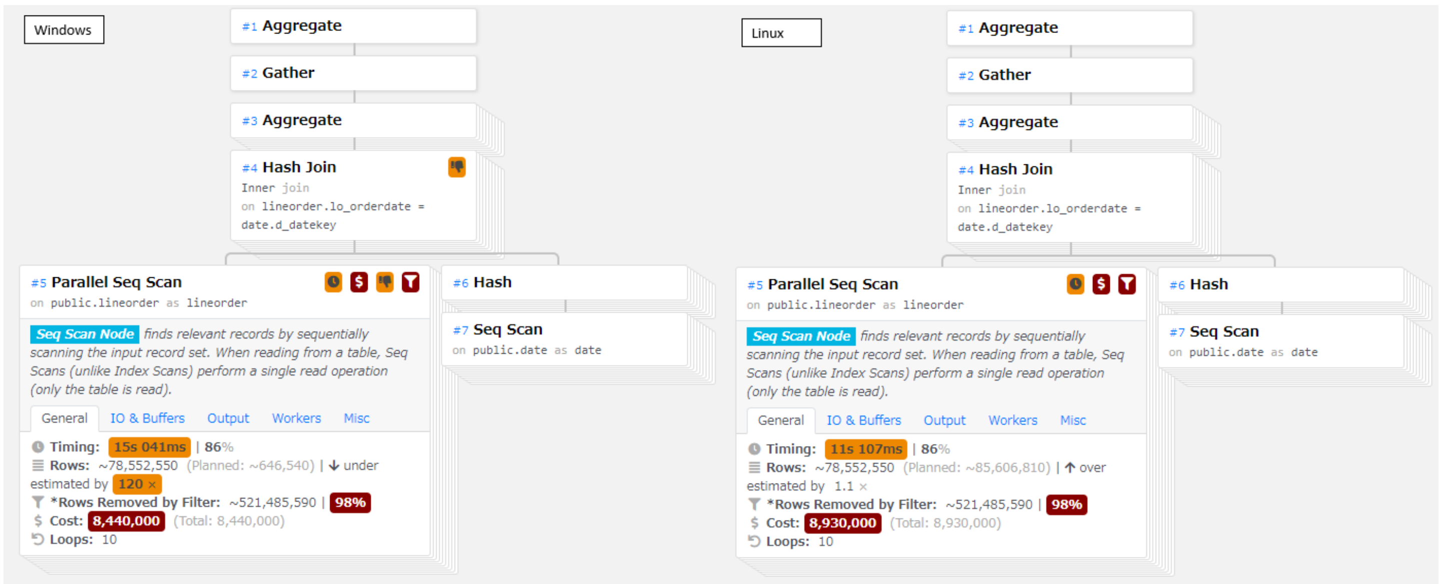


図 6.10 The execution plan of q1.1 on SSB (L: Windows R:Linux)

本検証は、オンメモリとなる状態で測定を行っていることから、Windows版での性能差については、WindowsとLinuxでのメモリの扱い方の違いによって生じたものと考えられます。

6.5. パラレルクエリ/テーブル・パーティショニング性能検証

6.5.1. 検証概要

データベースに蓄積された大容量のデータに対して複雑な集計・分析を行うクエリは、 パラレルクエリおよびテーブル・パーティショニングによる性能改善の恩恵を受けやすいとされています。

本検証では複雑な集計・分析が必要なOLAPで使用される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、 下記を確認することを目的としました。

1. パラレルクエリ、テーブル・パーティショニングそれぞれの性能への影響
2. パラレルクエリとテーブル・パーティショニングを組み合わせた場合の性能への影響および留意点

6.5.2. テーブル・パーティショニングとは

6.5.2.1. テーブル・パーティショニング

テーブル・パーティショニングとは、巨大なテーブルを複数のパーティションと呼称されるテーブルに分割して格納することです。

複数のパーティションで構成されたテーブルは1つのテーブルと同じように扱え、 PostgreSQLのクライアントはテーブルがどのように分割されているかを意識することなく利用することが可能です。

テーブル・パーティショニングの利点を下表に記載します。

表 6.13 テーブル・パーティショニングの主な利点

利点	概要
検索範囲の絞り込みによる処理性能向上	クエリの検索条件がテーブル・パーティショニングの分割条件と一致する場合、検索条件に合わないパーティションを処理対象から除外することで、無駄な処理が行われず、結果としてクエリの処理時間が短縮します。
メモリキャッシュの有効活用	検索条件に合わないパーティションへのスキャンを回避することで頻繁に検索されるパーティション内のデータをメモリに保持しやすくなります
一括挿入・削除の効率化	テーブル内のデータを分割して保存することで一括でのデータロードや削除が、特定のパーティションの追加や削除で行える場合あります。 例えば、1ヶ月のデータを削除する場合には、1つの大規模なテーブルでは、DELETE文による負荷とそれに伴うVACUUMによる負荷が発生します。テーブル・パーティショニングの場合、1か月分のデータが格納されパーティションの削除(DROP TABLEなど)でDELETEやVACUUMの負荷をかけず高速に大量データを削除することが可能です。

テーブル・パーティショニングにおける典型的なデータの分割方法は「レンジ (範囲)」「リスト」「ハッシュ」の3種類です。

表 6.14 データの分割方法

データの分割方法	概要	利用例
レンジ (範囲)	データ値の範囲毎に分割する方法。パーティションそれぞれが互いに重ならないような範囲を持ち、範囲内のデータが各パーティションに格納されます。	売り上げ履歴テーブルを年単位、月単位に分割
リスト	データを一定の選択肢 (リスト) から値を選ぶ列がある場合に、その値に基づいて分割を行います。	売り上げ履歴テーブルを製品ID毎に分割
ハッシュ	列のハッシュ値に基づいて各パーティションに分配します。	商品テーブルで一意な列(商品ID)のハッシュ値でパーティション毎に均等にデータを分散

6.5.2.2. PostgreSQLでのパーティショニング

PostgreSQL9.6以前のバージョンでは、テーブル・パーティショニングはテーブル継承、CHECK 制約、トリガー関数などの複数の機能を組み合わせて実現していました。 上記方法はテーブル・パーティショニングの実装に手間がかかる、かつトリガー関数でデータを振り分けているため、データ挿入に時間がかかる等の課題がありました。

PostgreSQL10でパーティショニング機能が正式サポートされました。これによりCREATE TABLE構文によるテーブル・パーティショニング実装が可能になりました。 この機能は宣言的パーティショニングと呼称され、宣言的パーティショニングではトリガー関数を利用していないためデータ挿入にかかる時間が改善されています。

本検証では、PostgreSQL10でサポートされた宣言的パーティショニングを利用しています。

宣言的パーティショニングでは、データの分割方法として「レンジ (範囲)」「リスト」「ハッシュ」の3種類をサポートしています。

レンジ、リストはPostgreSQL10、ハッシュはPostgreSQL11からサポートされています。

また、テーブル・パーティショニングに対するクエリの処理時間を短縮するため、PostgreSQLには下記の機能が実装されております。

表 6.15 テーブル・パーティショニングへのクエリ処理時間を短縮する主な機能

機能	概要
パーティションプルーニング	検索条件に合致しないパーティションに対するスキャンを省略する機能。検索に不要なパーティションへのスキャンを省略することで処理時間を短縮します。
パーティションワイズ結合	同じパーティションを持つテーブル同士をパーティションキーを利用して等価結合する際に、対応するパーティション同士を結合する機能です。パーティション同士を結合することで処理時間を短縮します。
パーティションワイズ集約	テーブル・パーティショニングに対する集約処理において、パーティションごとに集約処理を行い、最後にその結果を統合する機能です。パーティション単位に集約処理を行うことで処理時間を短縮します。

6.5.2.3. テーブル・パーティショニングに関連するパラメータ

テーブル・パーティショニングに関連する主なパラメータは下記の通りです。

表 6.16 テーブル・パーティショニングに関連する主なパラメータ

パラメータ	デフォルト値(10)	デフォルト値(11)	デフォルト値(12)	デフォルト値(13)	説明
enable_partition_pruning	—	on	on	on	SQL実行時のパーティションプルーニングを有効にするか否か
enable_partitionwise_join	—	—	off	off	パーティションワイズ結合を有効にするか否か
enable_partitionwise_aggregate	—	—	off	off	パーティションワイズ集約を有効にするか否か

6.5.3. 検証構成

6.5.3.1. 検証ハードウェア構成

Microsoft Azure上の仮想マシンで検証を実施しました。

クライアント用サーバとDBサーバは分けずに1台のみの構成で検証を実施しました。

仮想マシンのスペックは以下の通りです。

表 6.17 仮想マシンのスペック

名前	仮想マシンサイズ	vCPU	メモリ（GiB）	ストレージサイズ（GiB）最大/IOPS
DB server	Standard E16s_v4	16	128	200/1100

6.5.3.2. 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 6.18 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	CentOS Linux release 8.1.1911 (Core)
Database	PostgreSQL 13.1

6.5.4. 検証方法

6.5.4.1. 検証項目

パラレルクエリ/テーブル・パーティショニング検証では以下の3点を検証しました。

6.5.4.1.1. 検証A(パラレルクエリの効果確認)

検証Aではパラレルクエリによる性能効果を確認しました。

具体的には、パラレルクエリ使用時、未使用時のぞれぞれの状態でクエリを実行し処理時間の比較を実施しました。

6.5.4.1.2. 検証B(テーブル・パーティショニングの効果確認)

検証Bではテーブル・パーティショニングによる性能効果を確認します。

具体的には、ファクトテーブルであるlineorderテーブルのlo_orderdate列および、 ディメンションテーブルであるdateテーブルのd_datekey列をパーティションキーとし、 レンジパーティショニングでそれぞれ月毎のパーティションを作成した状態でクエリを実行し処理時間を測定しました。

また検証Aと検証Bで測定した処理時間を比較することでテーブル・パーティショニング、パラレルクエリそれぞれの性能影響について確認を行いました。

6.5.4.1.3. 検証C(パラレルクエリとテーブル・パーティショニングの組み合わせによる効果確認)

検証Cではパラレルクエリとテーブル・パーティショニングを組み合わせた場合の性能影響を確認します。

具体的には、月毎のパーティションを作成した状態でパラレルクエリを有効化し、クエリを実行し処理時間を測定しました。

また検証Aと検証Cで処理時間を比較することでテーブル・パーティショニング、パラレルクエリを組み合わせた場合の性能影響について確認を行いました。

6.5.4.2. データモデル

本検証では、Star Schema Benchmarkで定義されているデータモデルを利用しています。

6.5.4.3. クエリ

本検証では、Star Schema Benchmarkで定義されているクエリを利用しています。

6.5.4.4. 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。 パラレルワーカープロセスの最大数は16(仮想マシンのvCPU数)になるよう設定を行っています。

表 6.19 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	変更理由
shared_buffers	96GB	SSBの全データをshared_buffers上に乗せられるよう96GBを指定
work_mem	1GB	クエリで実行されるソート処理等でディスクへの書き込みが発生しないように1GBを指定
max_worker_processes	16	バックグランドワーカープロセスの最大数は、仮想マシンのvCPU数(16)を指定
max_parallel_workers	16	パラレルワーカープロセスの最大数は、仮想マシンのvCPU数(16)を指定
max_parallel_workers_per_gather	16	一つのGatherノードに対して起動できるワーカー数は、仮想マシンのvCPU数(16)を指定
max_parallel_maintenance_workers	0	影響する測定項目がないため、0を設定

6.5.4.5. 測定手順

Star Schema Benchmarkで定義される全クエリを、測定用のPythonスクリプトを用いて実行します。 測定は、各測定パターンにおいて、5回ずつ実施しました。 データベースへの同時接続数は1とし、複数のクライアントからはクエリを実行していません。

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ております。

```
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'
);
```

6.5.4.5.1. 検証A(パラレルクエリの効果確認)

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、PostgreSQLのバッファ上、もしくはOSのディスクバッファ上にデータを常駐させた状態にしています。

表 6.20 検証時設定(検証A)

パラメータ	パラレルクエリ未使用	パラレルクエリ使用
max_parallel_workers_per_gether	0	16

6.5.4.5.2. 検証B測定方法(テーブル・パーティショニングの効果確認)

下表のテーブルにおいて月毎のパーティションを作成した状態でデータを投入しました。

lineorderテーブル,dateテーブル(1992年1月から1998年12月)に日付情報を持つデータが格納されており、 各テーブルに84個(12ヶ月 * 7年)のパーティションを作成しています。

表 6.21 テーブル・パーティショニング(検証B)

パーティション諸元	lineorderテーブル(ファクトテーブル)	dateテーブル(ディメンションテーブル)
分割方式	レンジ(月分割)	レンジ(月分割)
パーティションキー	lo_orderdate列	d_datekey列
パーティション数	84	84

テーブル・パーティショニングを定義するSQLは下記の通りです。

```
CREATE TABLE date(d_datekey INTEGER NOT NULL, ...) PARTITION BY RANGE (d_datekey);

CREATE TABLE date_199201 PARTITION OF date FOR VALUES FROM ('19920101') TO ('19920201');
CREATE TABLE date_199202 PARTITION OF date FOR VALUES FROM ('19920201') TO ('19920301');
:
CREATE TABLE date_199812 PARTITION OF date FOR VALUES FROM ('19981201') TO ('19990101');
```

```
CREATE TABLE lineorder (..., lo_orderdate INTEGER NOT NULL,...) PARTITION BY RANGE (lo_orderdate);
CREATE TABLE lineorder_199201 PARTITION OF lineorder FOR VALUES FROM ('19920101') TO ('19920201');
CREATE TABLE lineorder_199202 PARTITION OF lineorder FOR VALUES FROM ('19920201') TO ('19920301');
:
CREATE TABLE lineorder_199812 PARTITION OF lineorder FOR VALUES FROM ('19981201') TO ('19990101');
```

lineorderテーブルを構成するパーティションの行数およびサイズを一部抜粋し記載します。 各パーティションにおよそ720万〜780件程度のレコードが格納される形になります。

表 6.22 検証に使用するテーブル・パーティショニングの規模(一部抜粋)

テーブル名	行数(実測値)	データサイズ(実測値) [Bytes]	データサイズ(実測値) [MB]
lineorder_199201	7,733,051	819,101,696	781.156
lineorder_199202	7,233,211	766,181,376	730.688
lineorder_199203	7,721,819	817,913,856	780.023

dateテーブルを構成するパーティションの行数およびサイズを一部抜粋し記載します。 各パーティションは1ケ月の日数分のレコードが格納される形になります。

表 6.23 検証に使用するテーブル・パーティショニングの規模(一部抜粋)

テーブル名	行数(実測値)	データサイズ(実測値) [Bytes]
date_199201	31	8192
date_199202	29	8192
date_199203	31	8192

測定に際しては、postgresql.confにパラレルクエリを使用しないための下表の設定を行い、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、PostgreSQLのバッファ上、もしくはOSのディスクバッファ上にデータを常駐させた状態にしています。

表 6.24 検証時設定(検証B)

パラメータ	テーブル・パーティショニング利用(パラレルクエリ未使用)
max_parallel_workers_per_gether	0

6.5.4.5.3. 検証C測定方法(パラレルとテーブル・パーティショニングの組み合わせによる効果確認)

検証Bと同様に月毎のパーティションを作成した状態でデータを投入しました。

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、PostgreSQLのバッファ上、もしくはOSのディスクバッファ上にデータを常駐させた状態にしています。

表 6.25 検証時設定(検証C)

パラメータ	テーブル・パーティショニング利用(パラレルクエリ使用)
max_parallel_workers_per_gether	16
enable_partition_pruning	on
enable_partitionwise_join	on
enable_partitionwise_aggregate	on

6.5.5. 検証結果・考察

6.5.5.1. 検証A(パラレルクエリの効果確認)

6.5.5.1.1. 検証結果

パラレルクエリ使用時と未使用時の各クエリの処理時間(5回測定、中央値)をグラフした結果は下図の通りです。

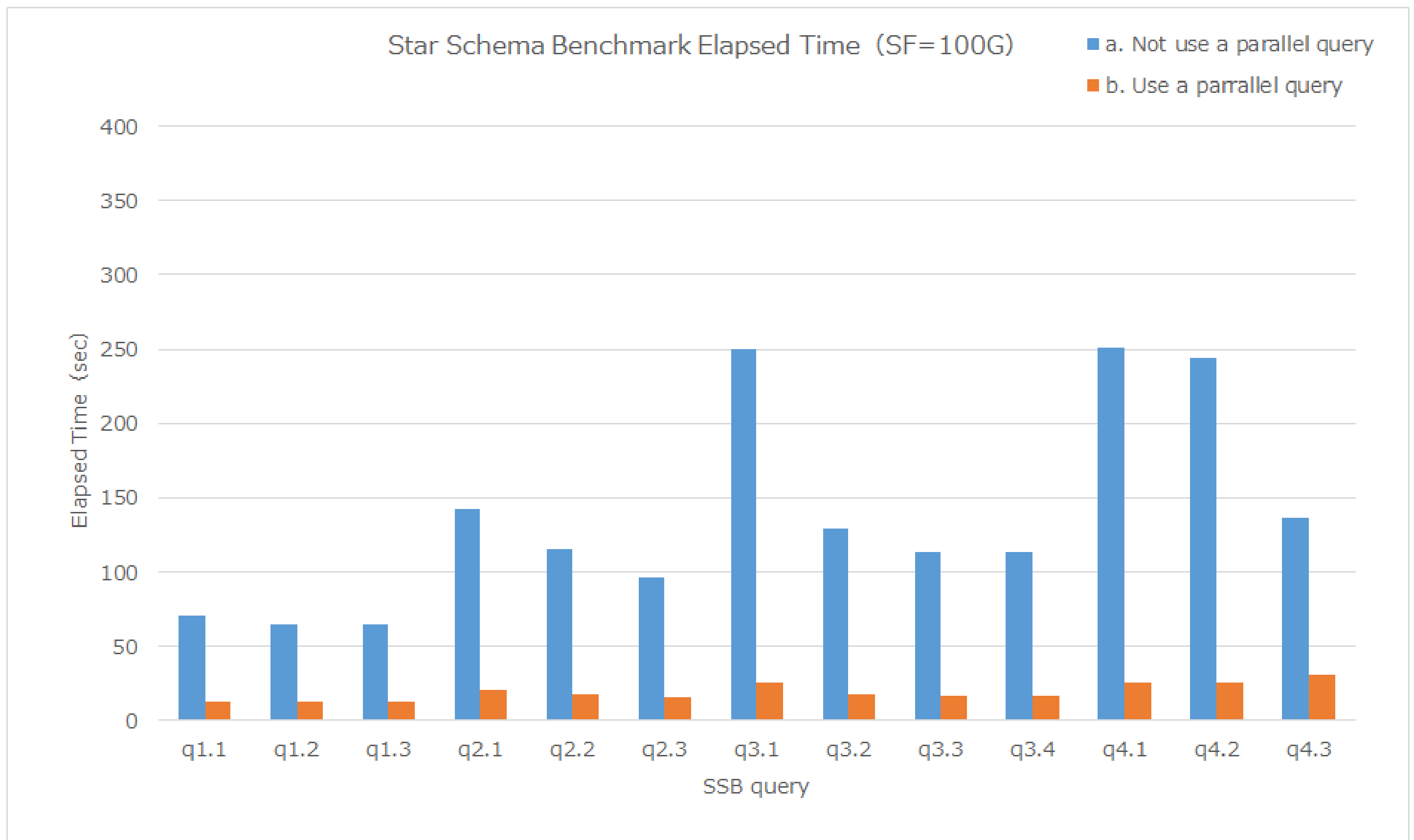


図 6.11 パラレルクエリ使用時/未使用時の処理時間比較

パラレルクエリ使用時と未使用時の各クエリの処理時間(5回測定、中央値)は下記の通りです。

表 6.26 パラレルクエリ使用時の性能向上比

	パラレルクエリ未使用(秒)	パラレルクエリ利用(秒)	性能向上比
q1.1	70.153	12.812	5.48
q1.2	64.541	12.118	5.33
q1.3	64.586	11.979	5.39
q2.1	142.623	20.355	7.01
q2.2	115.156	17.924	6.42
q2.3	95.849	15.898	6.03
q3.1	249.763	25.239	9.90
q3.2	129.444	17.747	7.29
q3.3	113.373	16.335	6.94
q3.4	113.252	16.243	6.97
q4.1	250.72	25.762	9.73
q4.2	243.713	25.532	9.55
q4.3	136.262	30.004	4.54

6.5.5.1.2. 考察

全てのクエリにおいてパラレルクエリが動作し、4.5〜9.9倍の性能改善が確認できました。 大容量のデータに対して複雑な集計・分析を行うクエリではパラレル・クエリによる性能改善の効果が大きいことを確認することができました。

各クエリで起動したパラレルクエリのワーカー数は下記の通りです。

表 6.27 パラレルクエリのワーカー起動数

	ワーカー数
q1.1	9
q1.2	9
q1.3	9
q2.1	9
q2.2	9
q2.3	9
q3.1	9
q3.2	9
q3.3	9
q3.4	9
q4.1	9
q4.2	9
q4.3	4

パラレルクエリq4.3のみワーカー起動数が4になっております。 本事象は 2018年度WG1活動報告書 の「7.6.2.2. バージョン11における SSB q4.3の性能」にて報告されている事象と同一のものです。

6.5.5.1.3. 検証Aまとめ

- Star Schema Benchmarkで定義される複雑な分析を行うOLTPを模したクエリでは、大規模なデータの中から特定の条件に合ったデータを集計する必要があるため、大規模データの集計が行われるクエリにはパラレルクエリの利用が効果的で、大幅な性能改善が見込める

6.5.5.2. 検証B(テーブル・パーティショニングの効果確認)

6.5.5.2.1. 検証結果

テーブル・パーティショニング使用時と未使用の各クエリの処理時間(5回測定、中央値)をグラフした結果は下図の通りです。

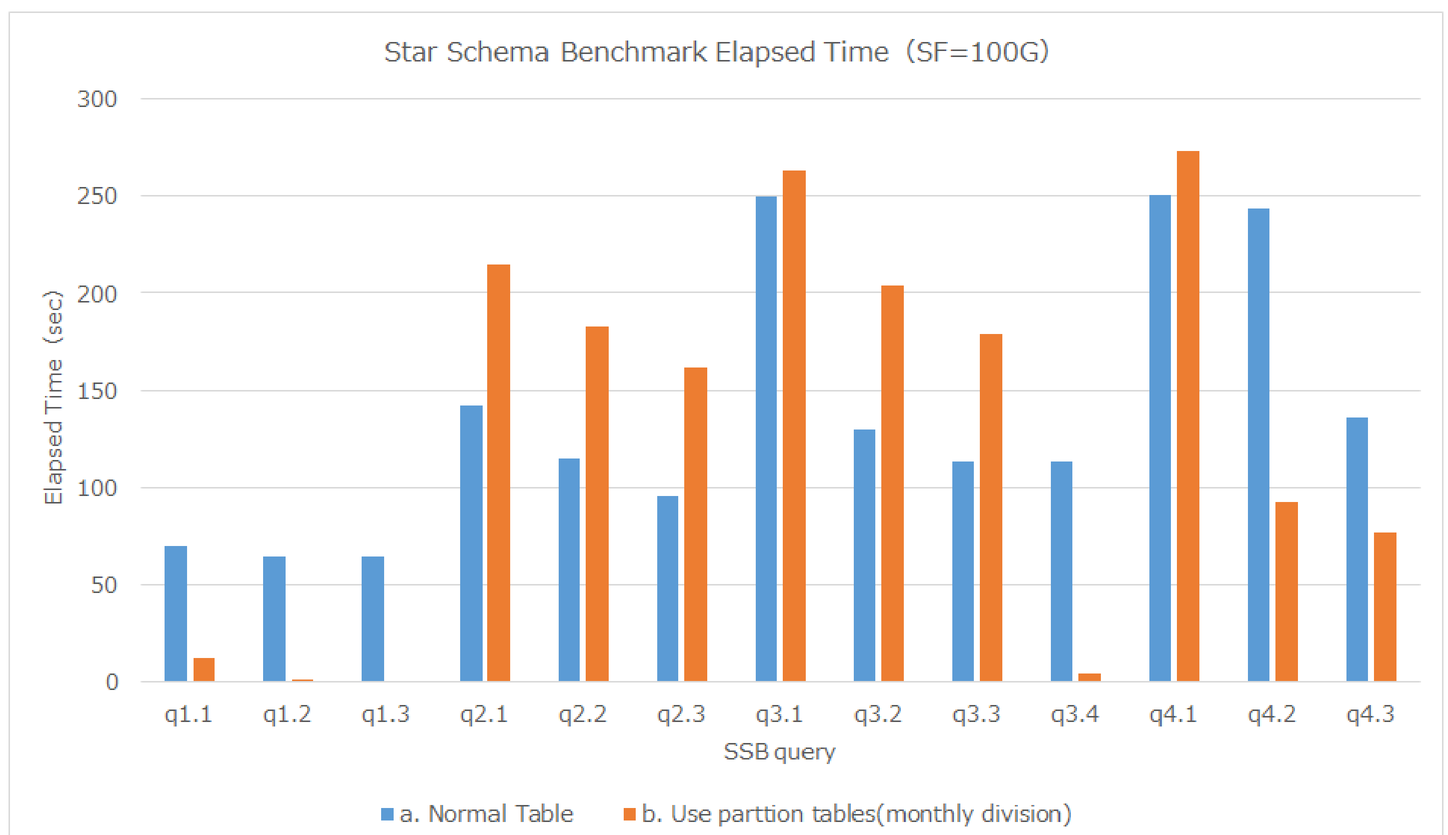


図 6.12 テーブル・パーティショニング使用時/未使用の処理時間比較

テーブル・パーティショニング使用時とテーブル・パーティショニング未使用の各クエリの処理時間(5回測定、中央値)は下記の通りです。

表 6.28 テーブル・パーティショニング使用時の性能向上比

	テーブル・パーティショニングなし	テーブル・パーティショニング	性能向上比
q1.1	70.153	12.338	5.69
q1.2	64.541	0.945	68.30
q1.3	64.586	0.819	78.86
q2.1	142.623	214.432	0.67
q2.2	115.156	183.1	0.63
q2.3	95.849	162.104	0.59
q3.1	249.763	263.195	0.95
q3.2	129.444	203.9	0.63
q3.3	113.373	179.16	0.63
q3.4	113.252	4.163	27.20
q4.1	250.72	273.148	0.92
q4.2	243.713	92.129	2.65
q4.3	136.262	76.521	1.78

6.5.5.2.2. 考察

下表に各クエリでのパーティションキーとした日付での絞り込み条件とテーブル・パーティショニングによる性能向上比を記載します。

表 6.29 lineorder,dateテーブルへの絞り込み条件と性能向上比

クエリ	日付での絞り込み条件	テーブル・パーティショニングによる性能向上比
q1.1	1年分のデータ	5.69
q1.2	1ヶ月分のデータ	68.30
q1.3	1週間分のデータ	78.86
q2.1	(無し)	0.67
q2.2	(無し)	0.63
q2.3	(無し)	0.59
q3.1	6年分	0.95
q3.2	6年分	0.63
q3.3	6年分	0.63
q3.4	1ヶ月分のデータ	27.20
q4.1	(無し)	0.92
q4.2	2年分	2.65
q4.3	2年分	1.78

月毎に分割したテーブル・パーティショニングを利用した場合、 クエリ実行に必要なデータが1週間分のq1.3では78.86倍、 必要なデータが1か月分のq1.2,q3.4ではそれぞれ68.30倍、27.20倍の性能改善が確認できました。

必要なデータが2年分のq4.2,q4,3では2.65倍、1.78倍の性能改善となっております。

・ テーブル・パーティショニングによる性能改善の要因調査

テーブル・パーティショニングによる性能改善の要因を確認するため、68.30倍の改善が確認されたq1.2の実行計画の確認を行いました。

テーブル・パーティショニング未使用時のq1.2の実行計画は下図の通りです。

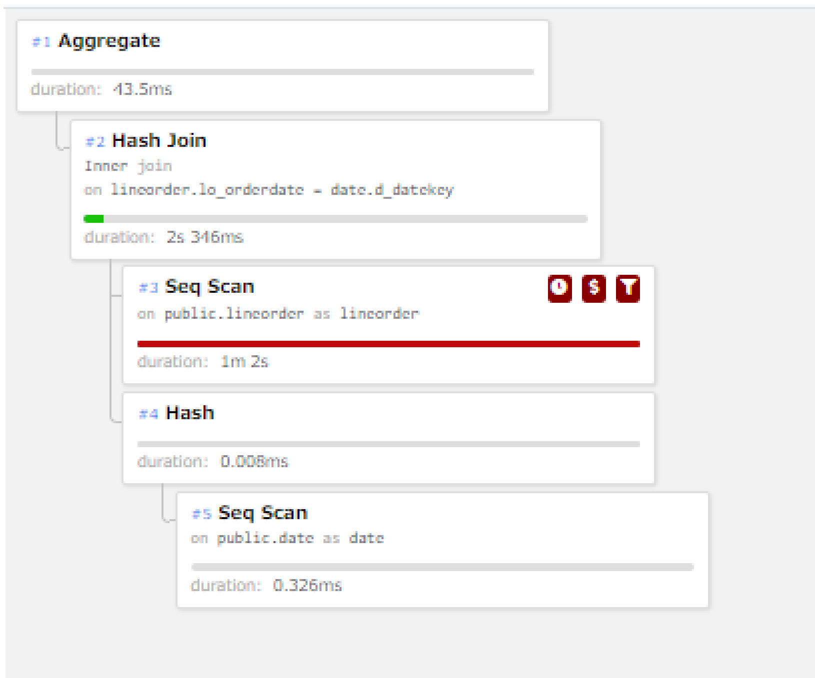


図 6.13 q1.2の実行計画(テーブル・パーティショニング未使用時)

上図の実行計画ではlineorderテーブルにSeq Scanが実行され、dateテーブルとHash Joinでテーブル結合が実施されています。

テーブル・パーティショニング使用時のは下図の通りです。（各パーティションへの処理は見易さのため、大部分を省略しています。）

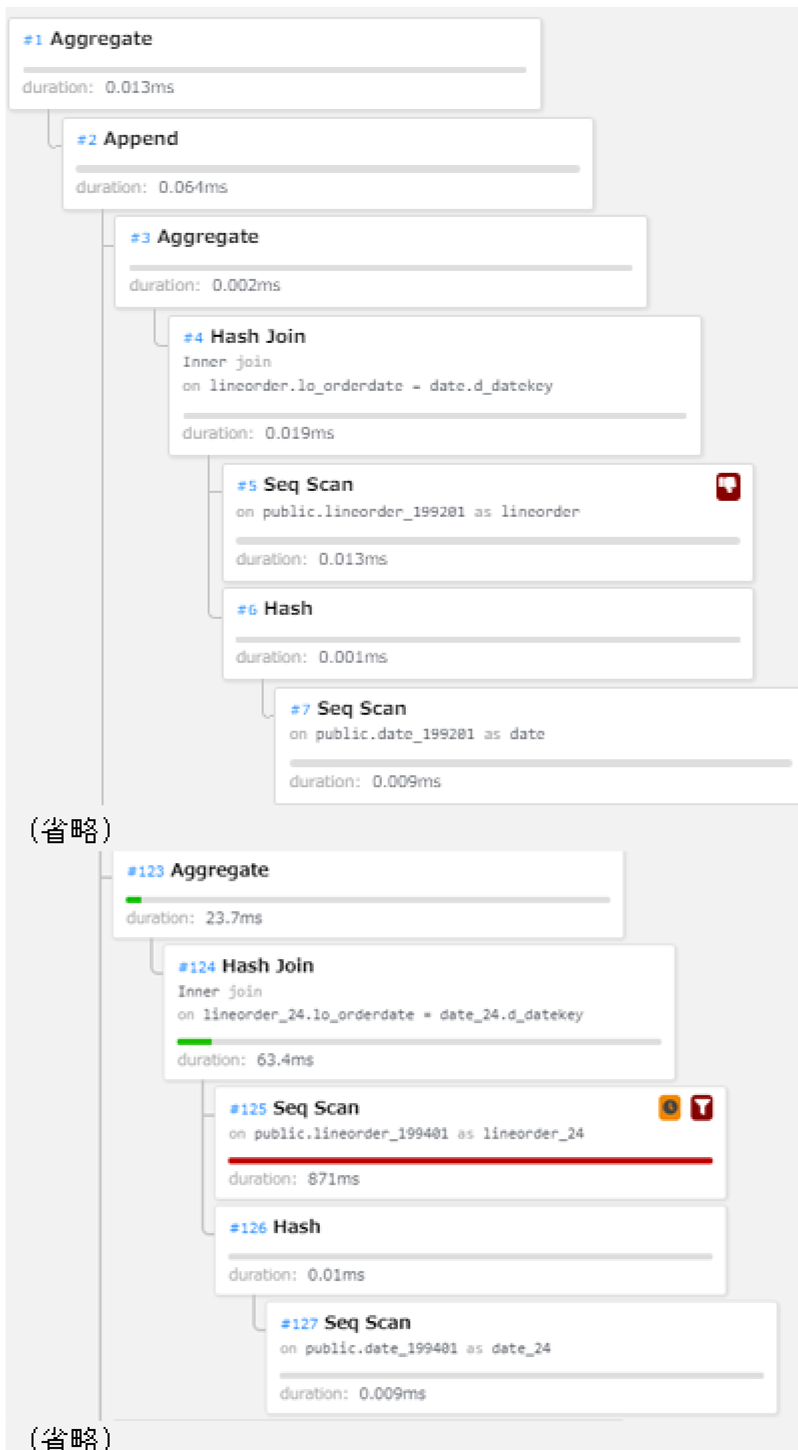


図 6.14 q1.2の実行計画(テーブル・パーティショニング使用時)

上図の実行計画ではパーティションワイズ結合により、同じ範囲を持つパーティション同士で結合(Hash Join)が実行されております。 また、q1.2のクエリの検索条件(d_yearmonthnum = 199401)に合致するデータがdateパーティションに存在しない場合にはlineorderパーティションへのスキャンが省略されています。

q1.2におけるテーブル・パーティショニング使用時・未使用時の性能差の要因を確認するため、計画ノードの比較を行いました。 下表にテーブル・パーティショニング未使用時の計画ノードに関する情報を記載します。

表 6.30 テーブル・パーティショニング未用時の計画ノード統計(q1.2)

計画ノード種別	実行回数	時間(割合)
Seq Scan	2	1m 2s(96%)
Hash Join	1	2s346ms(4%)
Aggregate	1	43.5ms(0%)
Hash	1	0.008ms(0%)

下表にテーブル・パーティショニング使用時の計画ノードに関する情報を記載します。

表 6.31 テーブル・パーティショニング使用時の計画ノード統計(q1.2)

計画ノード種別	実行回数	時間(割合)
Seq Scan	168	872ms(90%)
Hash Join	80	64.6ms(7%)
Aggregate	85	23.8ms(2%)
Hash	80	0.071ms(0%)
Append	1	0.064ms(0%)
Nested Loop	4	0.004ms(0%)

上表からテーブル・パーティショニング利用時は各パーティションにSeq Scan(シーケンシャル・スキャン)が実行され、Seq Scanの実行回数が増加していることが確認できます。 ただし、パーティションにq1.2のクエリの検索条件に合致するレコードがdateパーティションに存在しない場合には、 lineorderパーティションへのスキャンが省略されており、テーブル・パーティショニング未用時と比較するとSeq Scanの処理時間が大幅に短縮しています。

また、テーブル・パーティショニング使用時・未用時でq1.2実行時にアクセスしたlineorderテーブルのブロック数は以下となり、 パーティション・プルーニングにより、アクセスするデータ量を削減できています。

```
アクセスしたブロック数(テーブル・パーティショニング未使用) : 7,758,504
アクセスしたブロック数(テーブル・パーティショニング使用) : 100,079
```

・ テーブル・パーティショニングによる性能劣化の要因調査

6年分のデータが必要なq3.1,q3.2,q3,3および全てのデータが必要なq2.1,q2.2,q2,3では、 テーブル・パーティショニングのオーバヘッドによる性能劣化が発生しております。

テーブル・パーティショニングによる性能劣化が確認されたq2.1の実行計画の確認を行ました。 テーブル・パーティショニング未使用時のq2.1の実行計画は下図の通りです。

#1 Sort
by date.d_year, part.p_brand1
duration: 0.169ms

#2 Aggregate
by date.d_year, part.p_brand1
duration: 1s 358ms

#3 Hash Join
Inner join
on lineorder.lo_orderdate = date.d_datekey
duration: 1s 568ms

#4 Hash Join
Inner join
on lineorder.lo_suppkey = supplier.s_suppkey
duration: 8s 193ms

#5 Hash Join
Inner join
on lineorder.lo_partkey = part.p_partkey
duration: 1m 30s

#6 Seq Scan
on public.lineorder as lineorder
duration: 39s 671ms

#7 Hash
duration: 11.7ms

#8 Seq Scan
on public.part as part
duration: 136ms

#9 Hash
duration: 35.3ms

#10 Seq Scan
on public.supplier as supplier
duration: 122ms

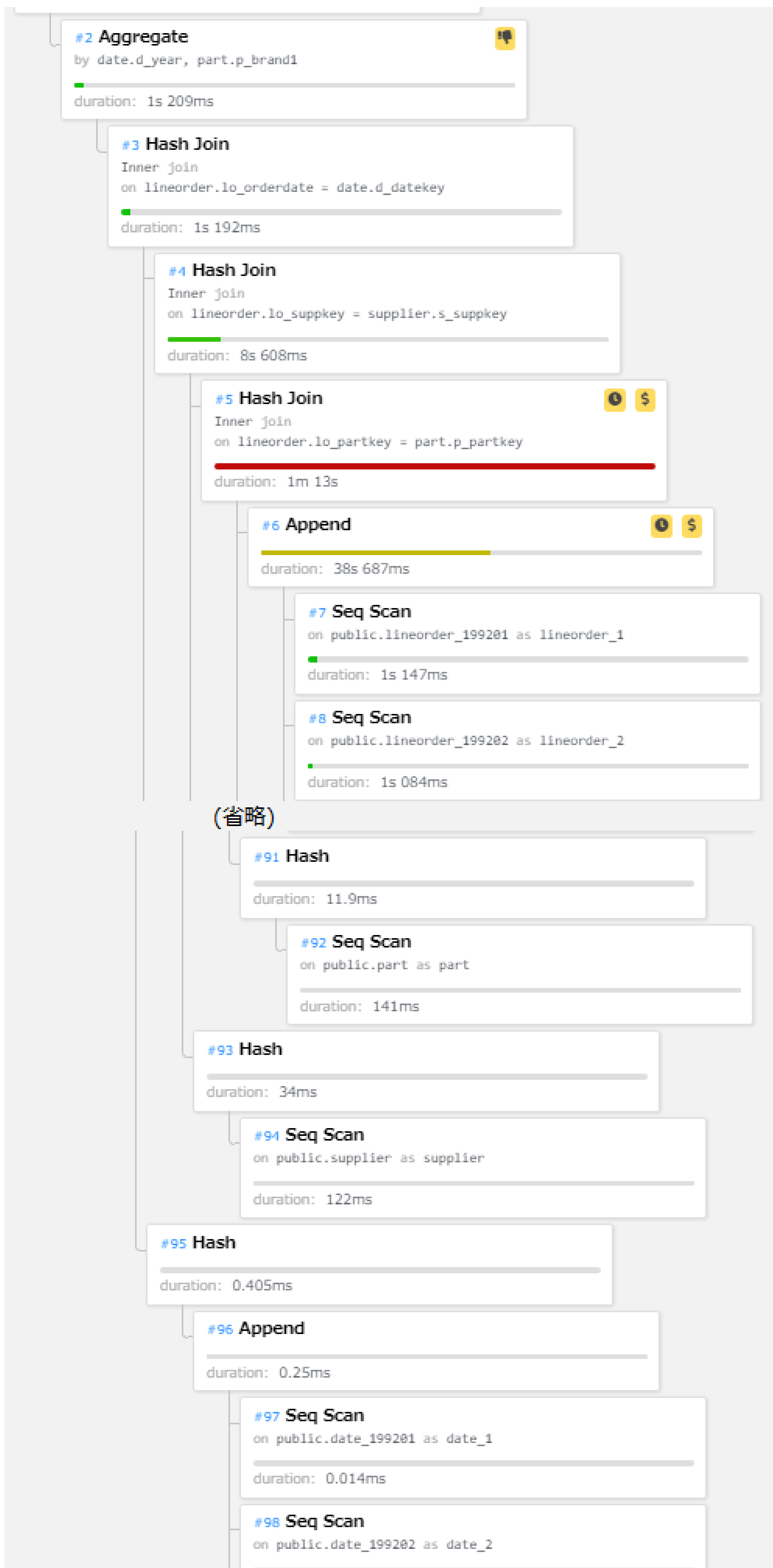
#11 Hash
duration: 0.313ms

#12 Seq Scan
on public.date as date
duration: 0.304ms

図 6.15 q2.1の実行計画(テーブル・パーティショニング未使用時)

上図の実行計画ではlineorderテーブルにSeq Scanが実行され、partテーブルとHash Joinでテーブル結合が実行されています。

テーブル・パーティショニング利用時のq2.1の実行計画は下図の通りです。


#2 Aggregate
by date.d_year, part.p_brand1
duration: 1s 209ms
#3 Hash Join
Inner join
on lineorder.lo_orderdate = date.d_datekey
duration: 1s 192ms
#4 Hash Join
Inner join
on lineorder.lo_suppkey = supplier.s_suppkey
duration: 8s 608ms
#5 Hash Join
$
Inner join
on lineorder.lo_partkey = part.p_partkey
duration: 1m 13s
#6 Append
$
duration: 38s 687ms
#7 Seq Scan
on public.lineorder_199201 as lineorder_1
duration: 1s 147ms
#8 Seq Scan
on public.lineorder_199202 as lineorder_2
duration: 1s 084ms
(省略)
#91 Hash
duration: 11.9ms
#92 Seq Scan
on public.part as part
duration: 141ms
#93 Hash
duration: 34ms
#94 Seq Scan
on public.supplier as supplier
duration: 122ms
#95 Hash
duration: 0.405ms
#96 Append
duration: 0.25ms
#97 Seq Scan
on public.date_199201 as date_1
duration: 0.014ms
#98 Seq Scan
on public.date_199202 as date_2

duration: 0.007ms

(省略)

図 6.16 q2.1の実行計画(テーブル・パーティショニング使用時)

上図の実行計画ではlineorderテーブルを構成する各パーティションに対してSeq Scanが実行され、 取得したデータをAppend(UNION)する処理が実施されています。 Append処理完了後に、partテーブルとHash Joinでテーブル結合が実行されています。

q2.1におけるテーブル・パーティショニング使用時・未使用時の性能差の要因を確認するため、計画ノードの比較を行いました。 下表にテーブル・パーティショニング未使用時の計画ノードに関する情報を記載します。

表 6.32 テーブル・パーティショニング未用時の計画ノード統計(q2.1)

計画ノード種別	実行回数	時間(割合)
Hash Join	3	1m 40s(71%)
Seq Scan	4	39s 930ms(28%)
Aggregate	1	47.3ms(0%)
Sort	1	0.169ms(0%)

下表にテーブル・パーティショニング使用時の計画ノードに関する情報を記載します。

表 6.33 テーブル・パーティショニング使用時の計画ノード統計(q2.1)

計画ノード種別	実行回数	時間(割合)
Seq Scan	170	1m 30s(42%)
Hash Join	3	1m 23s(39%)
Append	2	38s 687ms(18%)
Aggregate	1	1s 209ms(1%)
Hash	3	46.3ms(0%)
Sort	1	0.154ms(0%)

上表からテーブル・パーティショニング使用時は各ワーカーがシーケンシャルスキャンで取得したデータのAppend(UNION)する処理で時間(38s 687ms)がかかっていることがわかります。 また、シーケンシャルスキャンにかかる時間も39s 930msから1m 30sに伸びており、テーブル・パーティショニングのオーバヘッドにより性能劣化が発生しております。

・パラレルクエリ使用時とテーブル・パーティショニング使用時の各クエリの処理時間比較

パラレルクエリ使用時とテーブル・パーティショニング使用時の各クエリの処理時間(5回測定、中央値)をグラフした結果は下図の通りです。

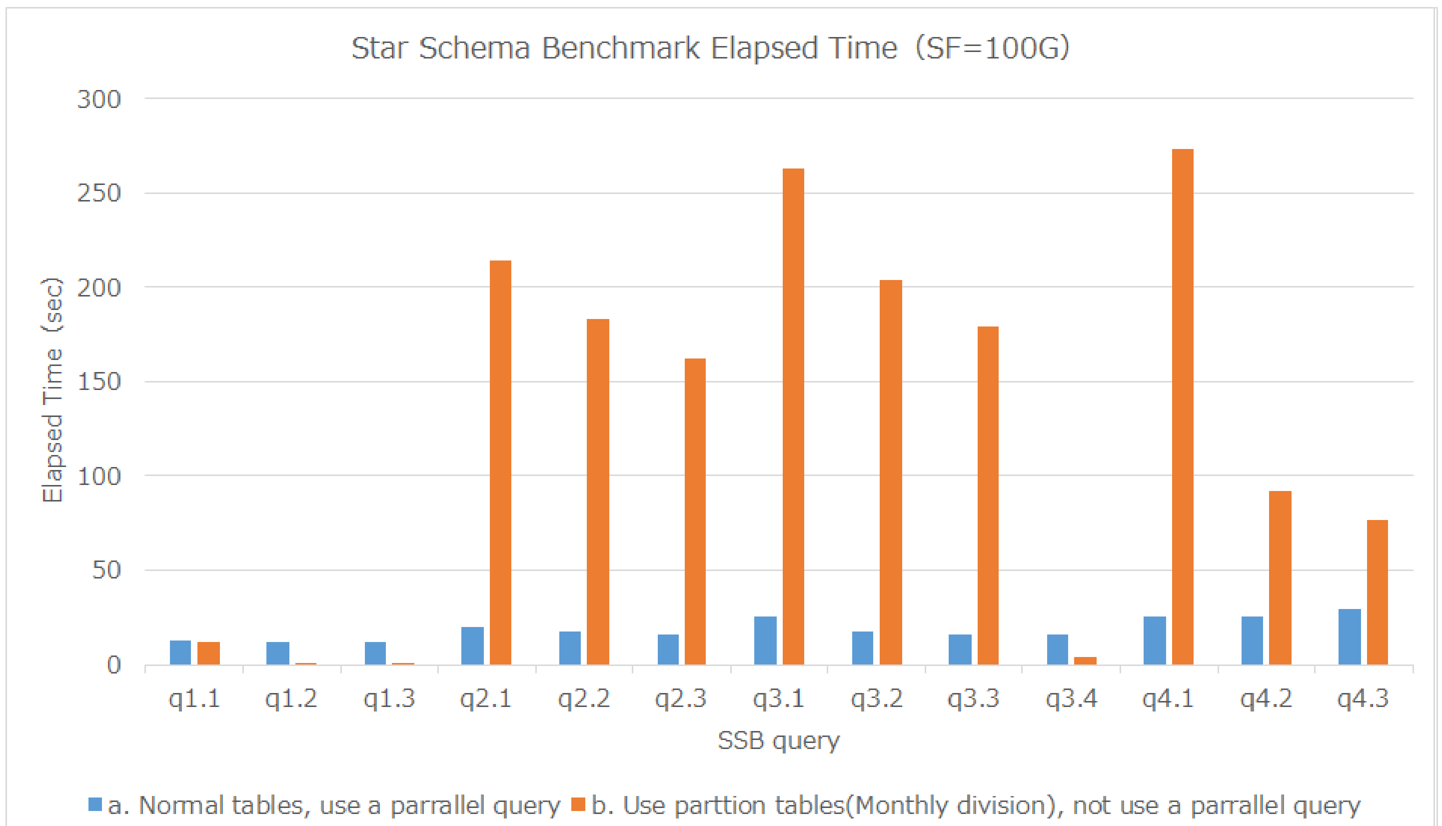


図 6.17 パラレル使用時/パーティション使用の処理時間比較

パラレルクエリ使用時とテーブル・パーティショニング使用時の各クエリの処理時間(5回測定、中央値)は下記の通りです。

表 6.34 テーブル・パーティショニング使用時の性能向上比(パラレルクエリとの比較)

	パラレルクエリ	テーブル・パーティショニング	性能向上比
q1.1	12.812	12.338	1.04
q1.2	12.118	0.945	12.82
q1.3	11.979	0.819	14.63
q2.1	20.355	214.432	0.09
q2.2	17.924	183.1	0.10
q2.3	15.898	162.104	0.10
q3.1	25.239	263.195	0.10
q3.2	17.747	203.9	0.09
q3.3	16.335	179.16	0.09
q3.4	16.243	4.163	3.90
q4.1	25.762	273.148	0.09
q4.2	25.532	92.129	0.28
q4.3	30.004	76.521	0.39

クエリ実行に必要なデータが1週間分のq1.3ではパラレルクエリと比較して、14.6倍、 必要なデータが1か月分のq1.2,q3.4ではそれぞれ12.8倍,3.9倍高速です。

一方、必要なデータが2年分のq4.2,q4,3、6年分のデータが必要なq3.1,q3.2,q3,3、 全てのデータが必要なq2.1,q2.2,q2,3では、パラレルクエリ使用時と比較すると処理時間が伸びています。

6.5.5.2.3. 検証Bまとめ

- 日付での絞り込みを行っているクエリでは、検索条件に合致するデータがdateパーティションに存在しない場合にはプルーニングにより、lineorderパーティションへのスキャンが省略され、性能改善に繋がっている
- 絞り込みの期間が短い場合はパラレルクエリよりもプルーニングによる性能改善効果が高く、期間が長くなる程性能改善効果が低くなっている
- 日付での絞り込みの期間が長いクエリや日付での絞り込みを行っていないクエリでは、各パーティションへのスキャンとスキャン結果の

集約(Append)による処理時間のオーバヘッドの影響が大きく、パーティショニングを利用しない場合よりも性能が劣化する原因となっている

6.5.5.3. 検証C(パラレルとテーブル・パーティショニングの組み合わせによる効果確認)

6.5.5.3.1. 検証結果

パラレルクエリ使用時とテーブル・パーティショニングおよびパラレルクエリを組み合わせた場合の各クエリの処理時間(5回測定、中央値)をグラフした結果は下図の通りです。

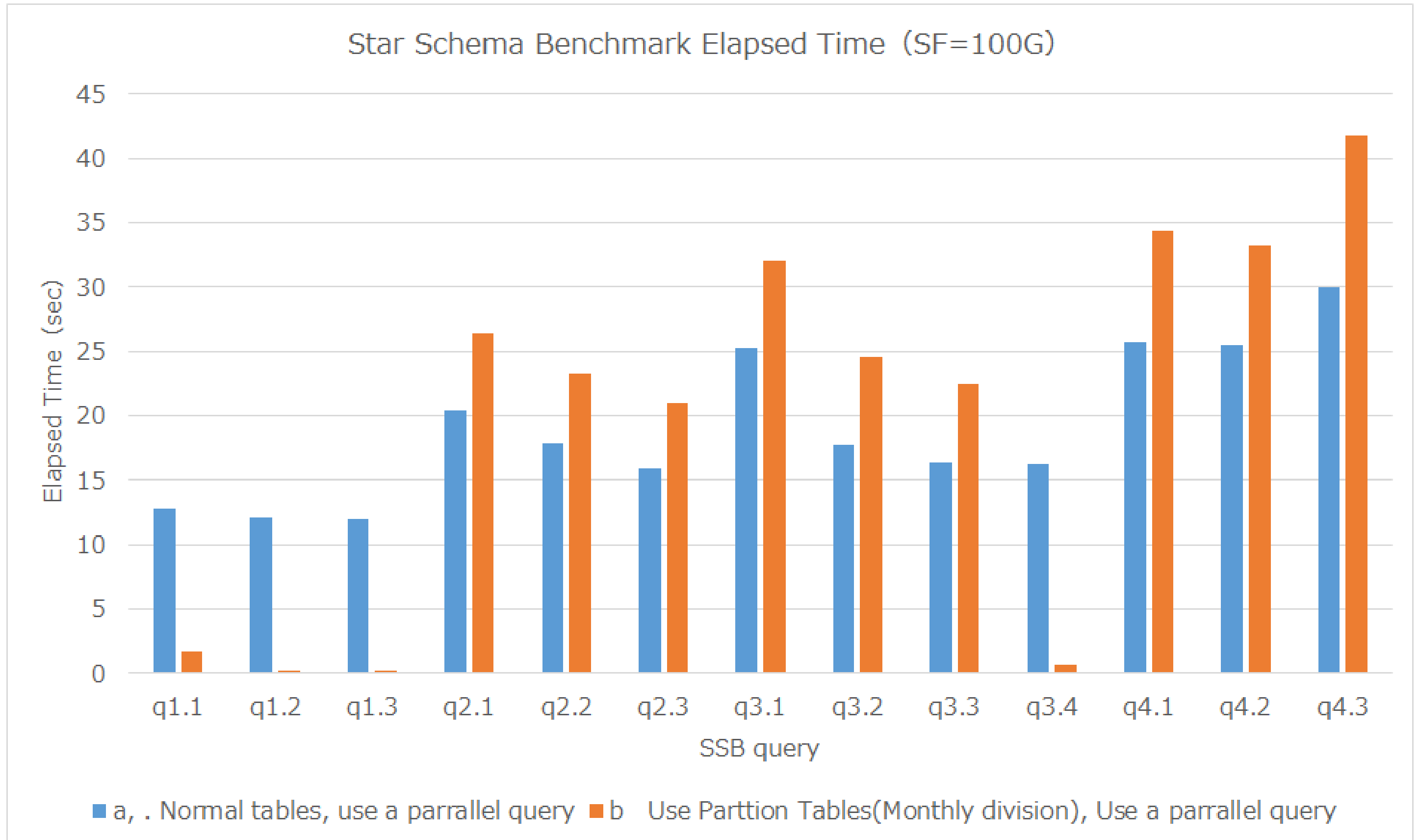


図 6.18 パラレルクエリ使用時とパーテーショニング・テーブルおよびパラレルクエリを組み合わせた場合の処理時間比較

パラレルクエリ使用時とパーテーショニング・テーブルおよびパラレルクエリを組み合わせた場合の各クエリの処理時間(5回測定、中央値)は下記の通りです。

表 6.35 パラレルクエリとテーブル・パーティショニングを組み合わせた場合の性能向上比(パラレルクエリとの比較)

	パラレルクエリ	パラレルクエリとテーブル・パーティショニング組み合わせ	性能向上比
q1.1	12.812	1.721	7.44
q1.2	12.118	0.232	52.23
q1.3	11.979	0.211	56.77
q2.1	20.355	26.431	0.77
q2.2	17.924	23.242	0.77
q2.3	15.898	21.005	0.76
q3.1	25.239	32.035	0.79
q3.2	17.747	24.557	0.72
q3.3	16.335	22.451	0.73
q3.4	16.243	0.642	25.30
q4.1	25.762	34.359	0.75
q4.2	25.532	33.25	0.77
q4.3	30.004	41.703	0.72

6.5.5.3.2. 考察

クエリ実行に必要なデータが1週間分のq1.3ではパラレルクエリと比較して、56.77倍、 必要なデータが1か月分のq1.2,q3.4ではそれぞれ52.23倍,25.30倍高速です。

一方、必要なデータが2年分のq4.2,q4,3、6年分のデータが必要なq3.1,q3.2,q3,3、 全てのデータが必要なq2.1,q2.2,q2,3では、パラレルクエリ使用時と比較すると処理時間が伸びています(20%〜30%の性能劣化)。

・ パラレルクエリとテーブル・パーティショニング組み合わせによる性能劣化要因調査

テーブル・パーティショニング利用により性能劣化が確認されたパラレルクエリ使用時のq2.1の実行計画およびlineorderテーブルへのレコード取得にかかった時間は下図の通りです。

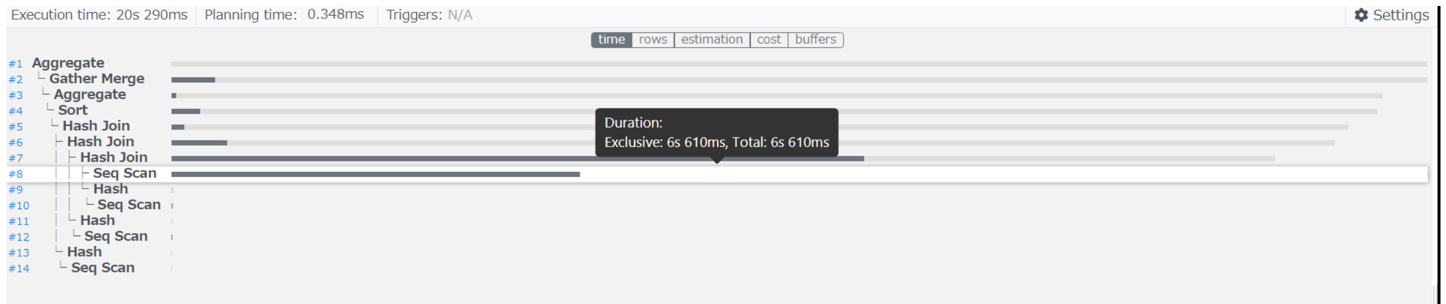


図 6.19 パラレルクエリ使用時のq2.1の実行計画

テーブル・パーティショニングとパラレルクエリを組み合わせた場合の実行計画およびlineorderテーブルへのレコード取得にかかった時間は下図の通りです。

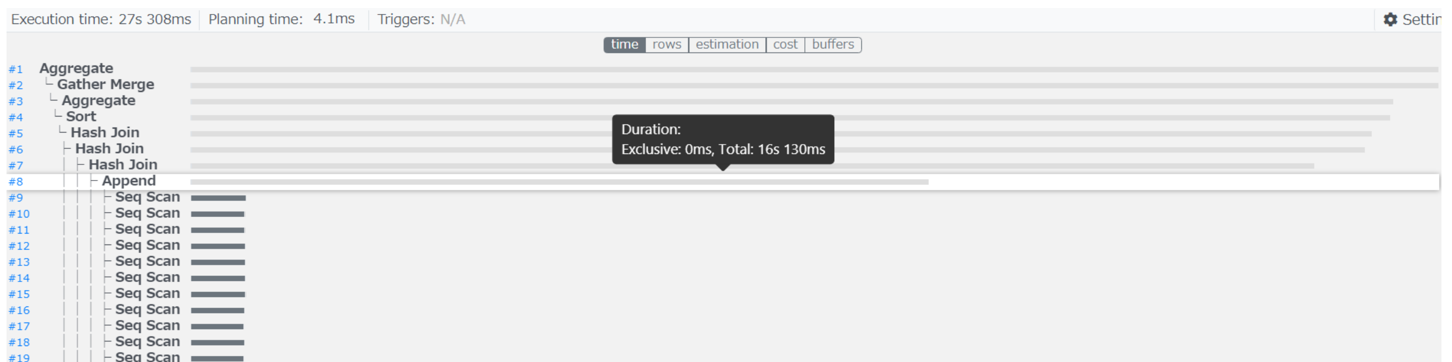


図 6.20 テーブル・パーティショニングとパラレルクエリを組み合わせた場合のq2.1の実行計画

上図よりパラレルクエリのみを使用した場合、lineorderテーブルへのSeq Scanは6s610msで完了していることが確認できます。

一方、テーブル・パーティショニングとパラレルクエリを組み合わせ場合、lineorderテーブルへの各パーティションへのSeq Scanおよび、Append処理が完了するまで16s130msかかっております。

Append処理自体はExclusive(排他処理時間)が0msとなっており、この処理が遅延原因となっておらず、 各パーティションへのSeq Scanに時間を費やしております。

テーブル・パーティショニングとパラレルクエリを組み合わせ状態で、各クエリで起動したパラレルクエリのワーカー数は下記の通りです。

パラレルクエリのワーカー数が通常のテーブルでは9となっておりますが、テーブル・パーティショニングを利用した場合、7に減少しています。

表 6.36 テーブル・パーティショニング利用時のパラレルクエリのワーカー起動数

	ワーカー数
q1.1	7
q1.2	7
q1.3	7
q2.1	7
q2.2	7
q2.3	7
q3.1	7
q3.2	7
q3.3	7
q3.4	7
q4.1	7
q4.2	7
q4.3	4

テーブル・パーティショニングにおけるワーカー起動数の決定を行うロジックをPostgresSQL13のソースコードにて確認しました。

```
/*
 * If the use of parallel append is permitted, always request at least
 * log2(# of children) workers.  We assume it can be useful to have
 * extra workers in this case because they will be spread out across
 * the children.  The precise formula is just a guess, but we don't
 * want to end up with a radically different answer for a table with N
 * partitions vs. an unpartitioned table with the same data, so the
 * use of some kind of log-scaling here seems to make some sense.
 */
if (enable_parallel_append)
{
    parallel_workers = Max(parallel_workers,
                           fls(list_length(live_childrels)));
    parallel_workers = Min(parallel_workers,
                           max_parallel_workers_per_gather);
}
Assert(parallel_workers > 0);
```

上記コードより、パラレル集約が実行可能な場合にはパーティション数を基にワーカーの数を決定しており、

(parallel_workers = Max(parallel_workers,fls(list_length(live_childrels)));の箇所)

fls関数は一番左側のビットを見付けて、その位置を返すため、 本検証のテーブルのパーティション数が84(01010100)の条件ではワーカー数は7と決定されます。

テーブル・パーティショニング利用時のパラレルクエリクエリ起動数は、「少数点以下切り上げ(log2(パーティション数))」で算出することが可能です。

6.5.5.3.3. 追加検証

テーブル・パーティショニング利用時におけるパラレルクエリのワーカー起動数が7となり、 通常テーブル利用時はワーカー起動数が9であったため、ワーカ起動数が処理性能に影響しているのではないかと考えました。

そのため、クエリヒント句でパラレルクエリのワーカー数を制御可能な pg_hint_plan を利用してテーブル・パーティショニング利用時のワーカー起動数を9に制御し、q2.1の処理性能を測定しました。

下図のようにヒント句を用いて、強制的にワーカー数を9に変更しています。

```
/*+ Parallel(lineorder 9 hard) */
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;
```

6.5.5.3.4. 追加検証結果

クエリq2.1の処理時間(5回測定、中央値)は下記の通りです。

表 6.37 パラレルクエリとテーブル・パーティショニングを組み合わせた場合の処理性能(追加検証)

	パラレルクエリ(ワーカー数:9)	パラレルクエリとテーブル・パーティショニングの組み合わせ (ワーカー数:7)	パラレルクエリとテーブル・パーティショニングの組み合わせ (ワーカー数:9)
q2.1	20.355s	26.431s	25.564s

6.5.5.3.5. 追加検証考察

テーブル・パーティショニングおよびパラレルクエリを組み合わせた状態で、 ワーカー数を9に変更した場合でもパラレルクエリのみを使用する場合と比較すると性能劣化が発生しています。

・ テーブル・パーティショニングによる性能劣化要因の調査

通常テーブルでパラレルクエリ使用を利用した場合、lineorderテーブルに対して9個のワーカーを用いて並列にSeqScanが実行されます。

一方、テーブル・パーティショニングを利用した場合、lineorderテーブルを構成する各パーティションに対するそれぞれのSeq Scanを、 9個のワーカーを用いて並列に処理しようと試みています。

しかし、lineorderテーブルを構成するパーティションは84個(12ヶ月 * 7年)定義されているため、 全てのパーティションへのスキャンに9個のワーカーを割り当てる場合、84パーティション * 9ワーカー必要になりますが、 ワーカーは9しか起動しないため、全てのスキャンに9個のワーカーを割り当てることができません。

パラレルクエリとテーブル・パーティショニング組み合わせ(ワーカー数:9)時の実行計画を下図に示します。

```
->  Parallel Append  (cost=0.00..0.00 rows=66678639 width=16) (actual time=0.018..15260.222 rows=60003814 loops=10)
      Worker 0:  actual time=0.020..14775.131 rows=72975180 loops=1
      Worker 1:  actual time=0.016..15985.096 rows=41250157 loops=1
      Worker 2:  actual time=0.012..14788.680 rows=72555040 loops=1
      Worker 3:  actual time=0.030..16000.167 rows=41689661 loops=1
      Worker 4:  actual time=0.008..14768.869 rows=72786323 loops=1
      Worker 5:  actual time=0.024..14692.645 rows=72109601 loops=1
      Worker 6:  actual time=0.028..16062.315 rows=41518687 loops=1
      Worker 7:  actual time=0.022..14784.564 rows=71853808 loops=1
      Worker 8:  actual time=0.011..14714.544 rows=72372799 loops=1
      ->  Parallel Seq Scan on public.lineorder_199201 lineorder_1  (cost=0.00..0.00 rows=859244 width=16) (actual time=0.006..1119.096 rows=77
            Output: lineorder_1.lo_revenue, lineorder_1.lo_orderdate, lineorder_1.lo_partkey, lineorder_1.lo_suppkey
            Worker 4:  actual time=0.006..1119.096 rows=7733051 loops=1
      :
      ->  Parallel Seq Scan on public.lineorder_199711 lineorder_71  (cost=0.00..0.00 rows=831702 width=16) (actual time=0.019..341.257 rows=14
            Output: lineorder_71.lo_revenue, lineorder_71.lo_orderdate, lineorder_71.lo_partkey, lineorder_71.lo_suppkey
            Worker 1:  actual time=0.020..7.313 rows=21991 loops=1
            Worker 3:  actual time=0.022..1145.522 rows=3761844 loops=1
            Worker 4:  actual time=0.017..6.210 rows=38656 loops=1
            Worker 5:  actual time=0.024..532.416 rows=3568783 loops=1
            Worker 7:  actual time=0.012..14.826 rows=94593 loops=1
       :
       ->  Parallel Seq Scan on public.lineorder_199804 lineorder_76  (cost=0.00..0.00 rows=830465 width=16) (actual time=0.018..2233.708 rows=
            Output: lineorder_76.lo_revenue, lineorder_76.lo_orderdate, lineorder_76.lo_partkey, lineorder_76.lo_suppkey
```

上図に示した実行計画のようにlineorderテーブルを構成する各パーティションへのSeq Scanに利用されるワーカー数はそれぞれの異なります。

下表に利用されたワーカー数と該当数のワーカーでSeq Scanが実行されたパーティション数および各パーティションのSeq Scanにかかった平均時間を記載しています。

表 6.38 ワーカー数と該当数のワーカーでSeq Scanが実行されたパーティション数(追加検証)

ワーカー数	該当数のワーカーでSeq Scanが実行されたパーティション数	各パーティションのSeq Scanにかかった平均時間(ms)
9	1	176.2
8	0	(-)
7	0	(-)
6	0	(-)
5	0	341.2
4	0	(-)
3	1	578.6
2	5	549
1	66	1368.2
0	5	2314.2
0(データが少ないパーティション)	1	163.4
0(データが存在しないパーティション)	4	0

上記のようにパーティションへのSeq Scanに割り当てられたワーカー数が増加することで、パーティションにおけるSeq Scanの時間は短縮されています。

ただし、ワーカーが割り当てられなかった(ワーカー数:0)Seq Scanは並列に実行されることがなく、順次実行となるためワーカー数0の処理が全体のボトルネットとなると考えられます。 ワーカーが割り当てられなかった(ワーカー数)Seq Scanにかかるは下記で計算できるため、およそ11.7s(11734.4ms)となり、この処理がクエリの遅延要因と考えます。

```
ワーカが割り当てられなかった(ワーカー数:0)Seq Scanにかかる時間
= 各パーティションSeq Scanにかかった平均時間(ワーカー数:0) * パーティション数
= 2314.2ms * 5 + 163.4ms * 1
= 11734.4ms
```

ワーカーが割り当てられなかった(ワーカー数:0)パーティションへのSeq Scanが順次実行され、 性能のボトルネックになる事象への対処方法は現在調査中です。

テーブル・パーティショニングを構成する各パーティションへのスキャンに割り当てるワーカー数の決定ロジックなどを調査し、 改善可能な手法がないか検討を進めていきます。

6.5.5.3.6. 検証Cまとめ

- テーブル・パーティショニングとパラレルクエリを組み合わせた場合、検証Bと同様にプルーニングによる性能改善効果の高かったクエリはパラレルクエリによってさらに性能が改善するような傾向
- テーブル・パーティショニングを利用した場合のパラレルワーカーの起動数は以下の算出式のようにパーティション数に依存して決定されており、通常のデータサイズに依存したパラレルワーカー数とは異なる
  ワーカー起動数 = 少数点以下切り上げ(log2(パーティション数)
- パラレルクエリとテーブル・パーティショニングを組み合わせて利用した場合でも性能劣化が発生(各パーティションに対してパラレルスキャンの処理にて、ワーカーが割り当てられなかったパーティションのスキャン処理が順次実行されていることが原因と考察する)

6.5.6. まとめ

今年度実施したパラレルクエリおよびテーブル・パーティショニングの性能検証の活動において、 OLAPで使用される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に次のことを確認することができました。

■パラレルクエリ利用時の性能影響

- Star Schema Benchmarkで定義される複雑な分析を行うOLAPを模したクエリではパラレルクエリによる性能改善の効果が大きい

■テーブル・パーティショニング利用時の性能影響

- パーティションキーを検索条件として指定し、パーティションキーを用いた絞り込みによりアクセスするパーティションの大部分を省略できるクエリではよい処理性能が得られる
- テーブルを構成するパーティションの大部分にアクセスする場合には各パーティションへのスキャンとスキャン結果の集約(Append)で処

理時間のオーバヘッドが発生し性能が劣化する
- テーブル・パーティショニングはパーティションキーを検索条件として指定されているクエリのみ実行されるようなワークロードでは有効であるが、検索条件が不定なクエリは性能劣化が発生するため注意が必要

■パラレルクエリ利用時とテーブル・パーティショニング利用時の比較

- OLAPのワークロードにおいては、パラレルクエリと比較するとテーブル・パーティショニングで性能改善できるクエリは限られている
- パーティションキーが検索条件に指定されていない場合にはオーバヘッドによる性能劣化が発生するため、検索条件が不定なOLAPで実行されるクエリではテーブル・パーティショニングによる性能劣化の影響を受けやすい

■パラレルクエリとテーブル・パーティショニングを組み合わせた場合の性能への影響および留意点

- パーティション・プルーニングの効果があるクエリ(パーティションキーを検索条件として指定し、パーティションキーを用いた絞り込みによりアクセスするパーティションの大部分を省略できるクエリ)では、パラレルクエリと組み合わせることで更に処理性能が改善される可能性がある
- テーブル・パーティショニングを利用した場合、パラレルワーカーの起動数がパーティション数に依存して決定される(ワーカー起動数 = 少数点以下切り上げ(log2(パーティション数))
- テーブル・パーティショニングを利用した場合、各パーティションに対してパラレルスキャンが実行されるが、ワーカーが割り当てられないパーティションのスキャンがボトルネックになり性能劣化が発生(本検証では20%〜30%)

上記結果より現時点ではOLAPにおける各機能の利用方針は下記となると考えております。

■パラレルクエリ

- 性能改善の効果が大きいため利用を推奨

■テーブル・パーティショニング

- パーティションキーを検索条件に指定したクエリの性能改善や、パーティションを用いたデータの一括削除・登録が運用面での改善が必要な場合に利用を検討
- OLAPのような検索条件が不定なクエリで利用する場合、パラレルクエリクエリと組み合わせて利用した場合においてもパーティショニングによる性能上のオーバヘッドが発生するため、運用面と性能面のトレードオフとなる

7. おわりに

今年度(2020 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 9 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 9 期目となりました。

本報告書では、以下の 2 つのテーマの検証結果を報告しました。

1. PostgreSQL 13および前バージョン12と比較したスケール性の検証
2. PostgreSQLにおけるパラレルクエリの継続的な性能検証

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。

2020年度検証活動は2019年度の活動におけるスケジュールの影響を受け、例年よりも短い活動期間となりました。 そんな中で、テーマの選定にあたっては参加メンバの興味関心が高く、すぐにでも取り掛かれるものを考慮したことで上記の2テーマが選定されました。

最初に、シンプルながらも聴講者からの人気が高く、新Verリリース時は誰もが気になるテーマである定点観測を例年通り行うことに決めました。 もう一つは9.6のリリース当初から継続して検証し続けているパラレルクエリを選定しましたが、これはパラレルクエリに造詣が深く毎年必ず検証を実施している参加メンバが今年も新たな検証観点を立案し、他のメンバを引っ張ってくれたことに起因しています。

今年度の活動では昨年度の1テーマ1社専任による分散体制から趣向を変え、1テーマに必ず2社以上の企業が参画し、絞り込んだテーマに戦力を集中することで従来より短い活動期間、オンライン主体の会合、議論の制限などを乗り切り検証を完了させることができました。

活動期間の短さについては検証メンバの頑張りやテーマ絞り込みで乗り切ることはできましたが、やはりコロナウィルス流行でニューノーマルとなったオンラインのみの議論、レビュー活動、コミュニケーションのあり方にはまだまだ改善の余地があり、如何にオフラインの会合と同じ空気で各メンバの発言を自然に促し議論を活発なものとできるかを課題として感じています。

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)

8. 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 (2020年度 WG1)	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	上原 一樹
	SRA OSS, Inc. 日本支社	OSS事業本部	正野 裕大
	SRA OSS, Inc. 日本支社	OSS事業本部	北山 貴広
	NECソリューションイノベータ株式会社	第一PFソフトウェア事業部	近藤 太樹
	NECソリューションイノベータ株式会社	第一PFソフトウェア事業部	湯村 昇平
	日本電気株式会社	AIプラットフォーム事業部	坪井 新治
	株式会社富士通ソーシアルサイエンスラボラトリ	基盤システム事業本部 第五システム部	小山田 政紀