PostgreSQLエンタープライズ・コンソーシアム 技術部会 WG#1

2019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. 2019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. Pluggable Storage 8
3.3.4. パラレルクエリ 8
3.4. 成果の公開 8
3.4.1. 2018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 13
4.3.4. PostgreSQL 設定 13
4.3.5. 環境 13
4.4. 検証方法(参照系) 13
4.5. 検証結果(参照系) 15
4.6. 追加検証(参照系) 16
4.7. 追加検証結果(参照系) 16
4.8. 検証方法(更新系) 18
4.9. 検証結果(更新系) 19
4.10. まとめ 20
5. Pluggable Storage 検証結果 21
5.1. 検証概要 21
5.1.1. 調査背景 21
5.2. Pluggable Storageとは 21
5.2.1. 概要 21
5.2.2. FDWとの棲み分けについて 21
5.3. Pluggable Storageの開発状況 21
5.4. 検証で使用したデータモデル概要 22
5.4.1. pgbench 22
5.4.2. star schema benchmark 22
5.5. 検証環境 22
5.5.1. マシン構成 22
5.6. DB設定 22
5.7. zheap 23
5.7.1. zheapの概要 23
5.7.2. テーブル構造 23
5.7.3. UNDOログの構造 24
5.7.4. 構築手順 25
5.7.5. 検証用テーブルの定義 25
5.7.6. バルク挿入性能 25
5.7.7. 参照性能 26

5.7.8. 更新性能 27
5.7.9. ベンチマークモデル比較(pgbench) 29
5.7.10. AM特性の確認 30
5.7.10.1. テーブルサイズ比較 30
5.7.10.2. テーブルサイズの変化 30
5.7.11. 設計、運用に関する確認 31
5.7.11.1. メンテナンス（VACUUM/ANALYZE/CLUSTER/REINDEX） 31
5.7.11.2. バックアップ/リカバリ 35
5.7.11.3. ストリーミングレプリケーション 36
5.7.11.4. ロジカルレプリケーション 37
5.7.11.5. 宣言的パーティション 38
5.7.11.6. パラレルクエリ 38
5.7.11.7. FILLFACTOR 39
5.7.12. 追加モジュール 40
5.8. zheapの検証結果に対する考察 40
5.9. zedstore 40
5.9.1. zedstoreの概要 40
5.9.2. テーブル構造 41
5.9.3. 構築手順 41
5.9.4. 検証用テーブルの定義 42
5.9.5. バルク挿入性能 43
5.9.6. 参照性能 44
5.9.7. 更新性能 46
5.9.8. ベンチマークモデル比較 48
5.9.8.1. pgbench(TPC-B) 48
5.9.8.2. Star Schema Benchmark(OLAP) 49
5.9.9. AM特性の確認 49
5.9.9.1. テーブルサイズ比較 49
5.9.10. 設計、運用に関する確認 50
5.9.10.1. メンテナンス（VACUUM/ANALYZE/CLUSTER/REINDEX） 50
5.9.10.2. バックアップ/リカバリ 54
5.9.10.3. ストリーミングレプリケーション 55
5.9.10.4. ロジカルレプリケーション 56
5.9.10.5. 宣言的パーティション 57
5.9.10.6. パラレルクエリ 57
5.9.10.7. FILLFACTOR 58
5.9.11. 追加モジュール 59
5.10. zedstoreの検証結果に対する考察 59
6. パラレルクエリ・JITコンパイル 性能検証 60
6.1. 検証概要 60
6.1.1. Star Schema Benchmarkとは 60
6.2. パラレルクエリとは 60
6.2.1. パラレルクエリ 60
6.2.2. PostgreSQLでのパラレルクエリ 60
6.2.3. パラレルクエリに関連するパラメータ 61
6.3. JITコンパイルとは 62
6.3.1. JITコンパイル 62
6.3.2. PostgreSQLでのJITコンパイル 62
6.3.3. JITコンパイルに関するパラメータ 63
6.4. 検証構成 63
6.4.1. 検証ハードウェア構成 64
6.4.2. 検証ソフトウェア構成 64
6.5. 検証方法 64
6.5.1. 検証項目 64
6.5.1.1. 検証A 64
6.5.1.2. 検証B 64

6.5.1.3. 対照群 64
6.5.2. データモデル 64
6.5.3. 使用クエリについて 67
6.5.4. DB設定 70
6.5.5. 測定手順 70
6.5.5.1. 検証A測定方法 70
6.5.5.2. 検証B測定方法 71
6.6. 検証結果 71
6.6.1. 検証A(パラレルクエリを複数クライアントから発行した場合の応答時間) 71
6.6.2. 検証B(JITコンパイルを有効にした場合の応答時間) 88
6.6.3. 対照群(パラレルクエリoff) 105
6.7. 考察 107
6.7.1. 検証A 107
6.7.1.1. max_parallel_workersの設定値について 107
6.7.1.2. 実際に使用するパラレルワーカー数の決定方法 107
6.7.2. 検証B 109
6.7.2.1. JITコンパイル有効時の性能について 109
6.8. 総括 110
7. おわりに 111
8. 著者 112

1. 改訂履歴

版	改訂日	変更内容
1.0	2020/09/01	新規作成

2. ライセンス

3. はじめに

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

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

3.1.1. 各WGとWG1の活動テーマ

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

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

表 3.1 PGEConsにおける課題領域

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

- WG1（新技術検証ワーキンググループ）
  - 新しい技術の実検証から有用性や課題を検討
  - PostgreSQLの最新バージョンの性能傾向の調査・検討
- WG2（移行ワーキンググループ）
  - 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、2019年10月にリリースされた PostgreSQL 12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. 2019年度の活動テーマ

2019年度は、2019年10月3日にリリースされた PostgreSQL 12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）

- Pluggable Storage
- 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。 参照系については、性能劣化が見られました。FlameGraphの結果および、12のリリースノートから、アクセスメソッドに関するソースコー>ド上の変更が関与しているものと推察されます。更新系については、性能差は認められませんでした。

3.3.3. Pluggable Storage

本検証では、PostgreSQL 12で実装されたPlugabble Storageの検証として、現在開発中のAccess Methodであるzheapとzedstoreの検証を行いました。 zheapについては、更新操作によるテーブルの肥大化の有無を確認する検証、heapとzheapの参照・更新処理性能を検証しました。zedstoreについては、アクセスするブロック数による参照処理時間の変化、OLAP系のベンチマークStarSchemaBenchmarkによる性能検証を実施しました。

3.3.4. 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2つの観点で、パラレルクエリの性能を検証しました。 1つ目は複数クライアントからのパラレルクエリの性能について、2種類のパラメータ設定における性能を比較・検証しました。 2つ目は1つ目と同じ検証をJITコンパイルと併用することによって、処理性能の変化を検証しました。

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. 2018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、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。 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。

3.5. 実施体制

2019年7月18日に開催された2019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

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

2019年度は、下記スケジュールで活動しました。 2019年度の活動では、コロナウィルス禍により、2020年2月以降は原則、対面での打ち合わせは実施せず、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	2019年7月18日
実施計画策定	2019年7月～2019年11月
検証実施	2019年12月～2020年2月
WG1活動報告書作成	2020年3月～2020年8月
総会と成果報告会	2020年9月3日(オンライン)

3.7. 謝辞

検証用の機器を日本ヒューレット・パッカード株式会社、日本電気株式会社、およびAmazon Web Services(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。2019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 12 と前バージョンの 11 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

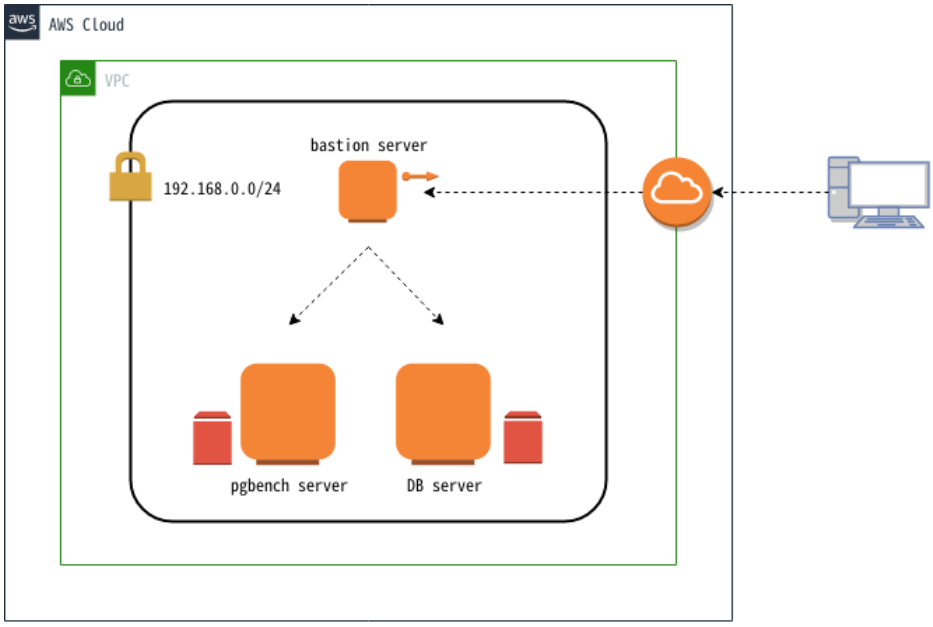


図 4.1 検証マシン構成

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 7.7

表 4.9 pgbench server

OS	Red Hat Enterprise Linux 7.7
pgbench	12.1

表 4.10 DB server

OS	Red Hat Enterprise Linux 7.7
PostgreSQL	11.6, 12.1

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 処理を発生させいない
```

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。

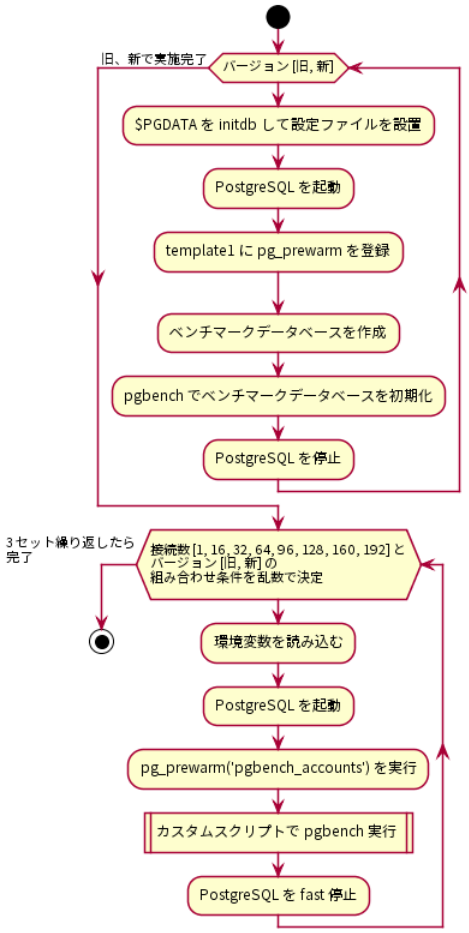


図 4.2 参照系ベンチマークの手順

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みんで、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)

11, 12 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に差がありました。11の平均TPSは7204.12、12の平均TPSは6688.70と、12の平均TPSは11よりも7.15%下回っていました。

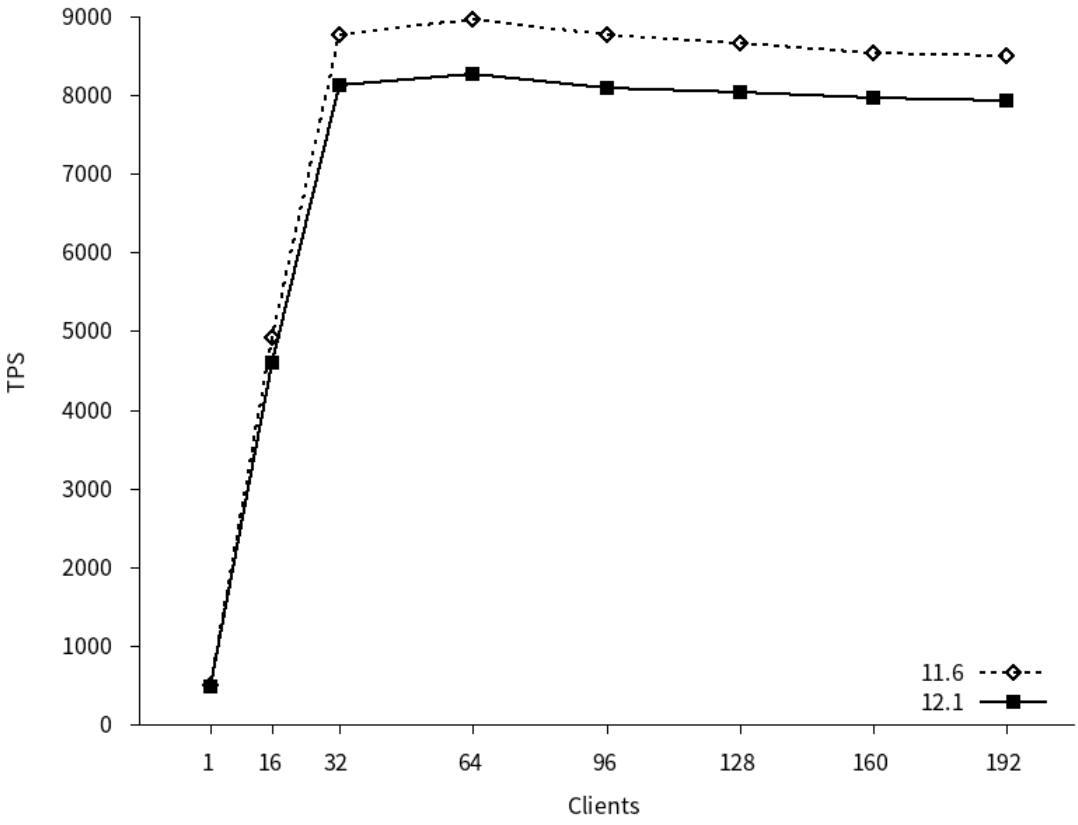


図 4.3 各クライアント数に対するTPS (参照系)

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。

11, 12 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がありました。11の平均レイテンシは10.38、12の平均レイテンシは11.16と、12の平均レイテンシは11よりも7.5%上回っていました。

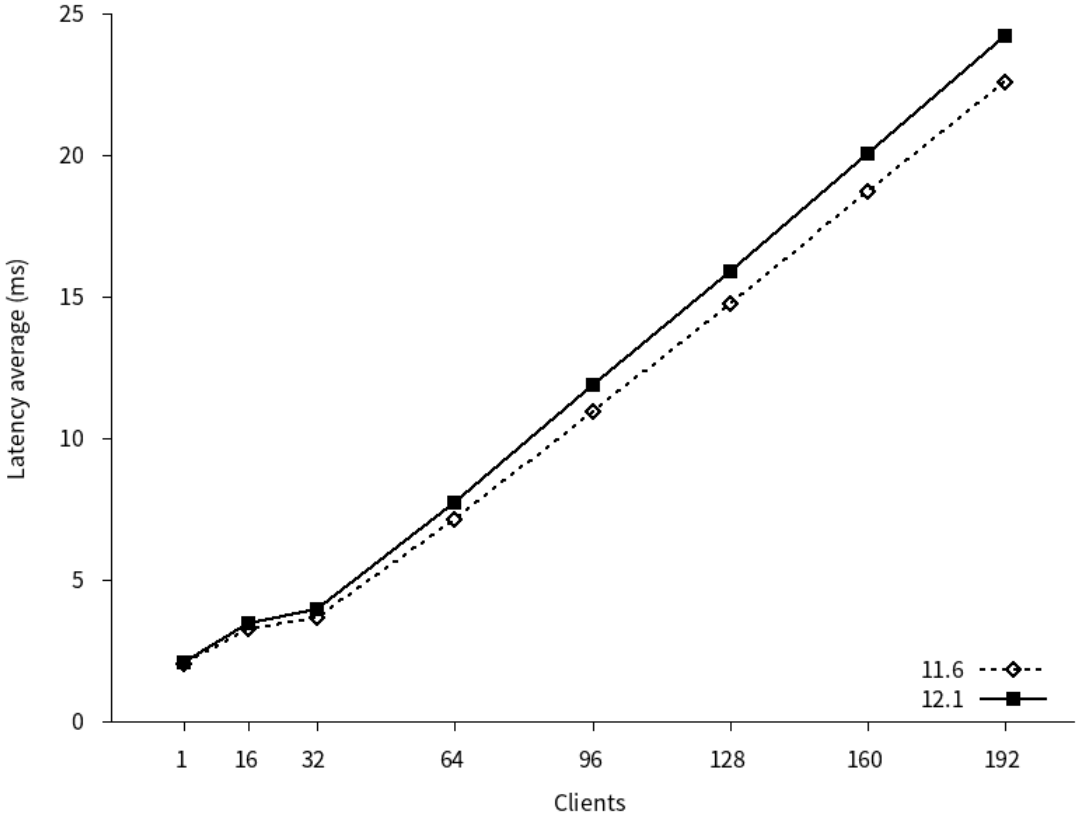


図 4.4 各クライアント数に対するレイテンシ (参照系)

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)

バージョン11と12の参照系検証結果から、12に性能劣化が確認されました。そこで、両バージョンのTPS差が最も大きかかった、クライアント数64の条件でperfを取得し、FlameGraphを生成しました

4.7. 追加検証結果(参照系)

FlameGraphの結果は以下の通りです。

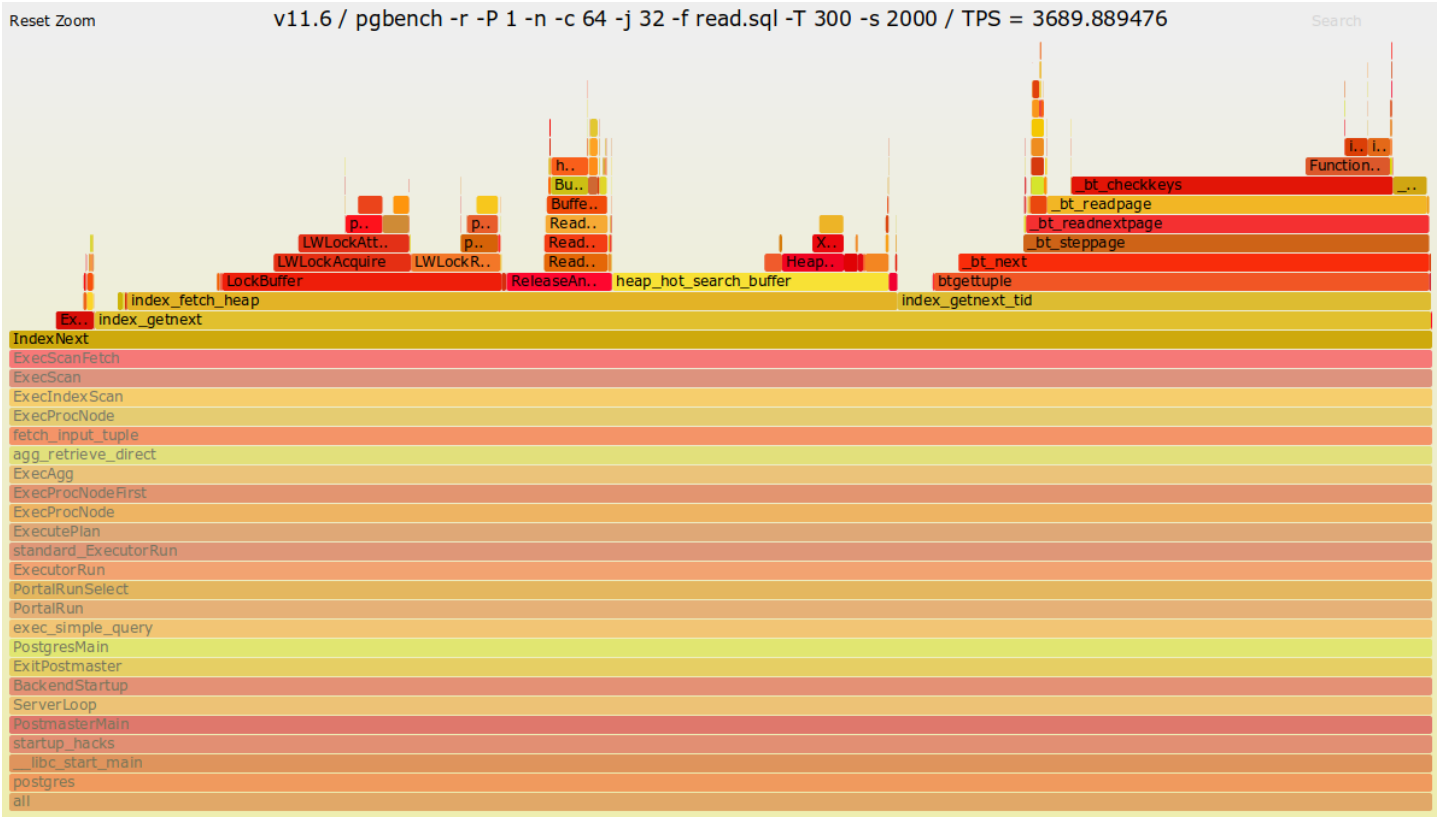


図 4.5 バージョン11.6のFlameGraph

図 4.6 バージョン12.1のFlameGraph

両者のコールグラフに着目すると、IndexNext関数以降に差があります。特に、バージョン12ではindex_fetch_heap関数以降にtable_index_fetch_tuple関数とheapam_index_fetch_tuple関数が追加されています。PostgreSQL 12の リリースノート によると、アクセスメソッドの変更がソースコードに加えられています。

> Add CREATE ACCESS METHOD command to create new table types (Andres Freund, Haribabu Kommi, Álvaro Herrera, Alexander Korotkov, Dmitry Dolgov)
>
> This enables the development of new table access methods, which can optimize storage for different use cases. The existing heap

access method remains the default.

この変更が新旧バージョンのTPSおよびレイテンシに影響を与えたものと推察されます。

続いて、更新性能について検証します。

4.8. 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。

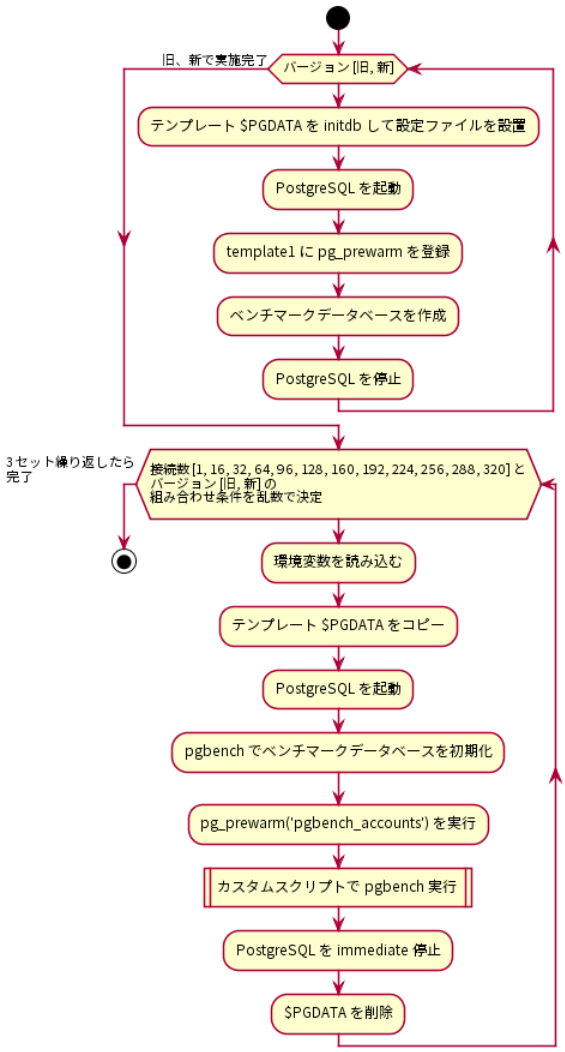


図 4.7 更新系ベンチマークの手順

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, 224, 256, 288, 320 の12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みんで、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。

4.9. 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。

11, 12 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。224接続以降もTPSの増加はやや認められましたが、192接続までの増加傾向よりも緩やかでした。新旧バージョン間でTPSに差はありませんでした。

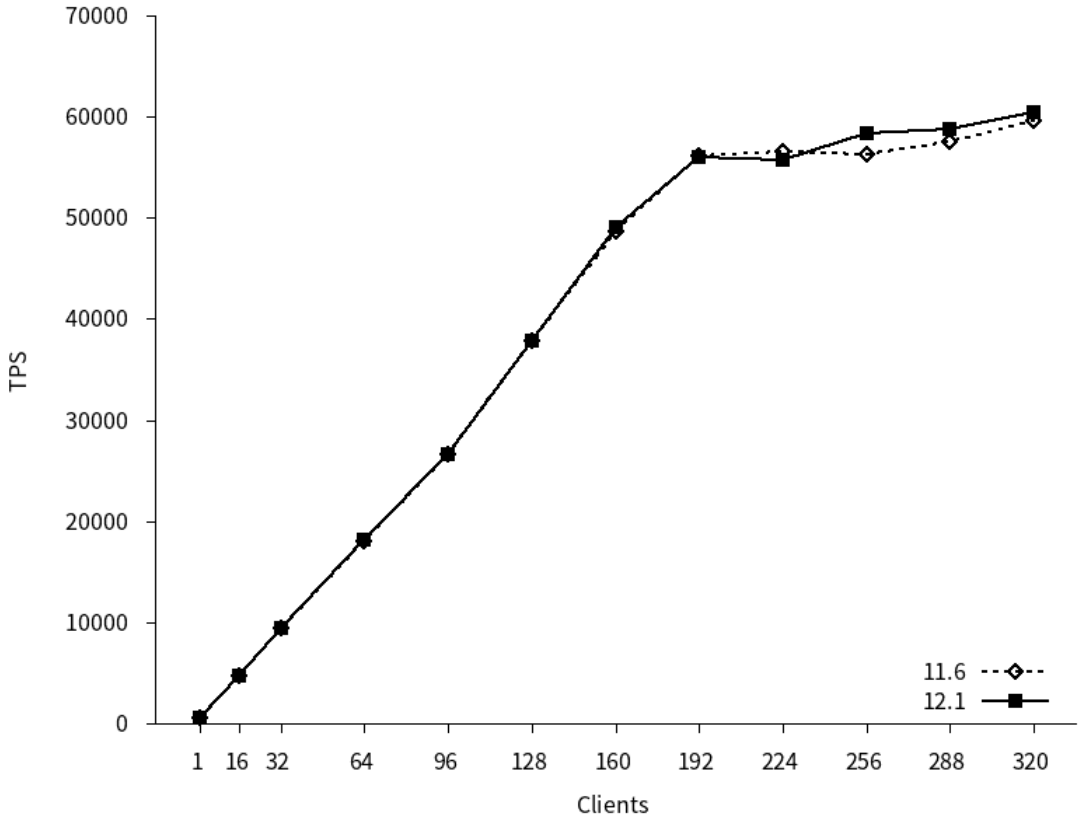


図 4.8 各クライアント数に対するTPS (更新系)

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。

11, 12 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。

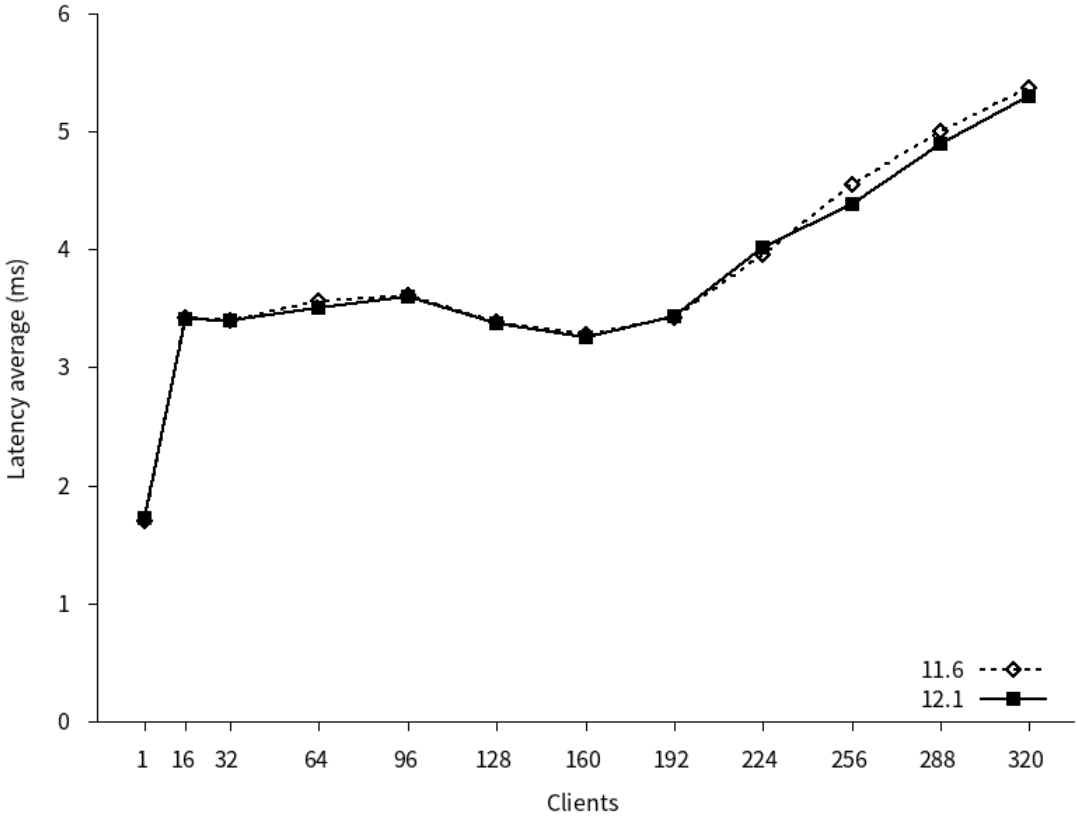


図 4.9 各クライアント数に対するレイテンシ (更新系)

4.10. 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、性能劣化が見られました。FlameGraphの結果および、12のリリースノートから、アクセスメソッドに関するソースコード上の変更が関与しているものと推察されます。

更新系については、性能差は認められませんでした。

5. Pluggable Storage 検証結果

5.1. 検証概要

5.1.1. 調査背景

Pluggable Storageの導入により、PostgreSQLではテーブルへのアクセスが抽象化されました。 PostgreSQL 12では、これまで使用されていたheap形式のAccess Method（以降、AM）しか提供されていませんが、現在、コミュニティより新たなAMの開発が進められています。

今後、PostgreSQLを運用する際、複数のAMを活用する機会が増えることが想定されます。その際、DBAにはシステムに適したAMの選択を含めたDB設計の能力が求められます。

本検証では、現在開発中のAM(zheap、zedstoreの2種類)の動作検証を行います。各AMの特性を確認し、適用領域の検討に向けた情報を整理します。

5.2. Pluggable Storageとは

5.2.1. 概要

PostgreSQLはこれまでheap形式のデータフォーマットのみを提供していました。 PostgreSQL 12では、Pluggable Storage機能としてBufferManager層とやり取りを行う新しいレイヤとして抽象化されたTable Access Methodが追加されました。

データベースはシンプルにデータを格納するだけではなく、大量データの分析に活用する等様々な用途で利用されています。 ただし、用途によってはPostgreSQLでは不向きな処理もありました。 Pluggable Storageの機能によって、テーブルへのアクセスが抽象化されます。 今後、この機能を利用し、用途に適したAMを選択することで、PostgreSQLの適用領域が拡大することが期待できます。

PostgreSQL 12では、従来のheap形式のデータフォーマットに対するAMしか提供されていません。 今後、新たなAMが提供された場合、利用者はAMの特性を理解しておくことで、特定のアプリケーションニーズに特化したAMをテーブルごとに選択可能となり、データベースの最適化が可能になります。

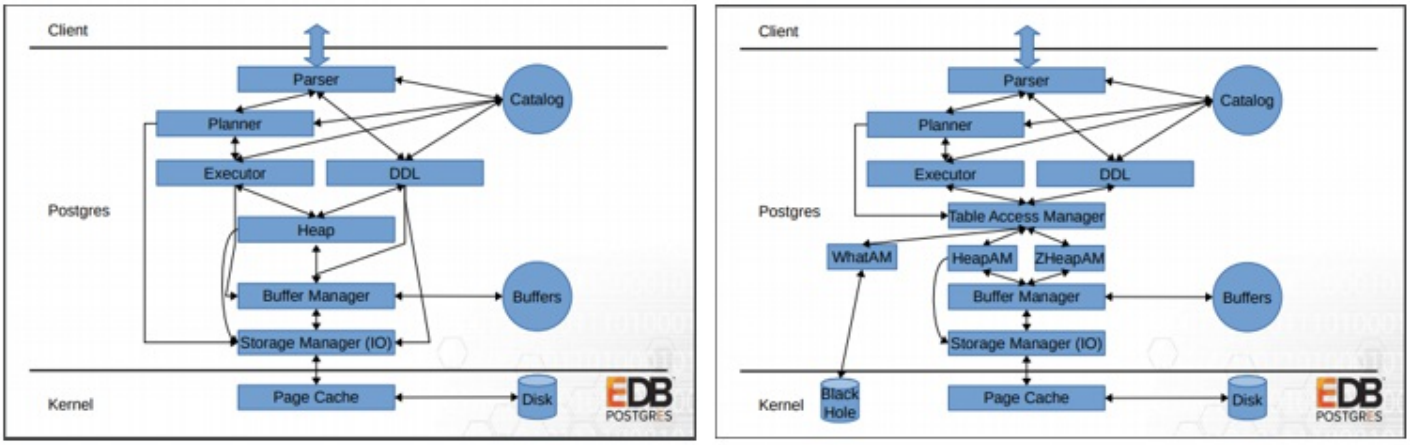


図 5.1 Route to access table data on Disk (Left: PostgreSQL 11 Right: PostgreSQL 12)

https://anarazel.de/talks/2018-10-25-pgconfeu-pluggable-storage/pluggable.pdf

5.2.2. FDWとの棲み分けについて

特性の異なるデータを扱う手段として、PostgreSQLには標準SQLで定義されているSQL/MEDの実装系である、Foreign Data Wrapper（以降、FDW）という仕組みを利用することが可能です。

Plugabble StorageとFDWは特性の異なるデータを扱うことができる点では共通していますが、両者は機能の目的が異なっており、そこから利用するシーンも分かれてきます。

FDWは、外部のPostgreSQLだけでなく異種DB等の外部データとの連携することが目的となっています。対して、Pluggable Storageはローカルのデータを最適なアクセス方法で扱うために利用されます。 Pluggable StorageとFDWでは提供される機能は異なっており、実現できる内容にも違いがあります。PostgreSQLでこれらの機能を利用する場合は両者を混同しないように注意する必要があります。

5.3. Pluggable Storageの開発状況

Access Method	主目的	開発状況
heap	従来のテーブル形式	PostgreSQL 12のコア機能として組み込まれています。
zheap	UNDOログを使用したVACUUM不要なテーブル形式	Enterprise DBが開発中。UNDOログの構成について再検討中のため、開発は2019年9月以降中断しています。(UNDOログの構成の方針が決まり次第、開発を再開する見込みです。)
zedstore	列指向のテーブル形式	13-develをベースにGreenplumが開発を行っています。
TDEforPG v2系	透過的暗号化テーブル形式	NECが開発中。

今回、PGEConsではzheapおよびzedstoreのAccess Methodを調査しました。

5.4. 検証で使用したデータモデル概要

本検証では以下のベンチマークを利用した検証を実施しました。

5.4.1. pgbench

pgbenchとは、PostgreSQLに付属するTPC-Bモデルの性能検査ツールです。

ツールの詳細については 前述の「定点観測（スケールアップ検証）」の説明をご参考ください。

5.4.2. star schema benchmark

Star Schema Benchmarkとは、 Star Schema Benchmarkの論文 によって公表されている古典的なデータウェアハウスモデルであるスタースキーマデータウェアハウスとして設計されています。

巨大なファクトテーブル(LINEORDER)を中心に放射上に情報の少ないディメンションテーブルが配置された構成のベンチマークモデルです。

- ファクトテーブル
  - 商品の購入履歴として詳細なデータが記録されています。
- ディメンションテーブル
  - ファクトテーブルを分析するため基本データとなるマスタデータとして顧客情報、製品情報が格納されています。

5.5. 検証環境

5.5.1. マシン構成

本検証は、Windows10上で動作するVirtualBoxのVM環境で行いました。

クライアント用サーバとDBサーバは分けずに1台のみの構成で行いました。

ホストマシンとVMのスペックは以下のとおりです。

名前	CPU	メモリ	OS	備考
ホストマシン	8	16GB	Windows 10 Pro	VirualBox 6.0.6を使用。
ゲストマシン	3	2GB	CentOS 7.4	

5.6. DB設定

pluggable storageの検証では、postgresql.confに以下の設定を行い、検証に影響を与えなようにチェックポイントの頻度を下げて検証を実施しました。 また、よりシンプルな動作となるようにパラレルクエリは停止しています。

```
shared_buffers = 512MB
checkpoint_timeout = 30min
max_wal_size = 15GB
log_checkpoints = on
max_parallel_workers_per_gather = 0
```

上記で記載したパラメータ以外のものは基本的にデフォルト値を採用しました。なお、zheapとzedstoreはベースとなるメジャーバージョンが異なっている点にご注意ください。

Access Method	ベースとなるメジャーバージョン(2020年2月時点)	備考
zheap	PostgreSQL 12	2019年9月から開発が停止しているため、12がベースとなっている
zedstore	PostgreSQL 13	

5.7. zheap

5.7.1. zheapの概要

PostgreSQLは追記型アーキテクチャを採用しているため、ある程度テーブルの肥大化は発生することは避けられきません。

テーブルの肥大化が発生するとスキャン量の増加に伴うパフォーマンスの劣化等、サービスに影響を与える可能性があります。 また、一度肥大化したテーブルを正常な状態に戻すには、テーブルの再構築等が必要になります。 テーブルの再構築は強力なロックが発生してしまうため、商用の環境では頻繁に実施することが困難です。 PostgreSQLの利用者は肥大化をなるべく抑えるために十分なVACUUMの実施や、FILLFACTORによるHOTを活用できる運用設計を行う必要がありました。

zheapでは、これらの課題を解決するため、新たなStorage Formatを提供しています。 zheapはUNDOログを採用することで追記型アーキテクチャをやめています。なお、PostgreSQL 11まで使用されていた従来のテーブルアクセスメソッド(heap)では、追記型アーキテクチャを採用している点に違いはありません。

MVCCはUNDOログにて実現しており、削除した領域の即時再利用を可能にすることでテーブルの肥大化を抑えています。また、UNDOログの採用により追記型アーキテクチャではなくなったため、zheapを使用したテーブルへのVACUUM処理は不要となっており、負荷の高いFREEZE処理も必要とはしません。これらの変化に伴いzheapでは、タプルヘッダ等の見直し不要なヘッダも行うことで、テーブルへの書き込み量の圧縮、テーブルサイズの圧縮を実現します。

5.7.2. テーブル構造

zheapのタプルヘッダの基本的な構造はheapのものと変わらないですが、heapと比較するといくつか削除された領域があります。

具体的には、タプルがINSERTされたトランザクションIDを記録するもの、DELETE等を行ったトランザクションIDを記録する領域はUNDOログの利用により不要となったため、削除されています。 これらの削減によりタプルヘッダのサイズは23バイトだったのものがzheapでは5バイトに圧縮されています。

heapを使用した従来のタプルヘッダの構造は以下のとおりです。

```
struct HeapTupleHeaderData
{
  union
  {
    HeapTupleFields t_heap;
    DatumTupleFields t_datum;
  } t_choice;
  ItemPointerData t_ctid;         /* current TID of this or newer tuple (or a
                                   * speculative insertion token) */
  uint16          t_infomask2;    /* number of attributes + various flags */
  uint16          t_infomask;             /* various flag bits, see below */
  uint8           t_hoff;                 /* sizeof header incl. bitmap, padding */
  /* ^ - 23 bytes - ^ */
  bits8           t_bits[FLEXIBLE_ARRAY_MEMBER];  /* bitmap of NULLs */
  /* MORE DATA FOLLOWS AT END OF STRUCT */
};
```

zheapを使用したタプルヘッダの構造は以下のとおりです。

```
typedef struct ZHeapTupleHeaderData
{
  uint16   t_infomask2;    /* number of attributes + translot info + various flags */
  uint16   t_infomask;     /* various flag bits, see below */
  uint8    t_hoff;         /* sizeof header incl. bitmap, padding */
  /* ^ - 5 bytes - ^ */
  bits8    t_bits[FLEXIBLE_ARRAY_MEMBER];  /* bitmap of NULLs */
  /* MORE DATA FOLLOWS AT END OF STRUCT */
} ZHeapTupleHeaderData;
```

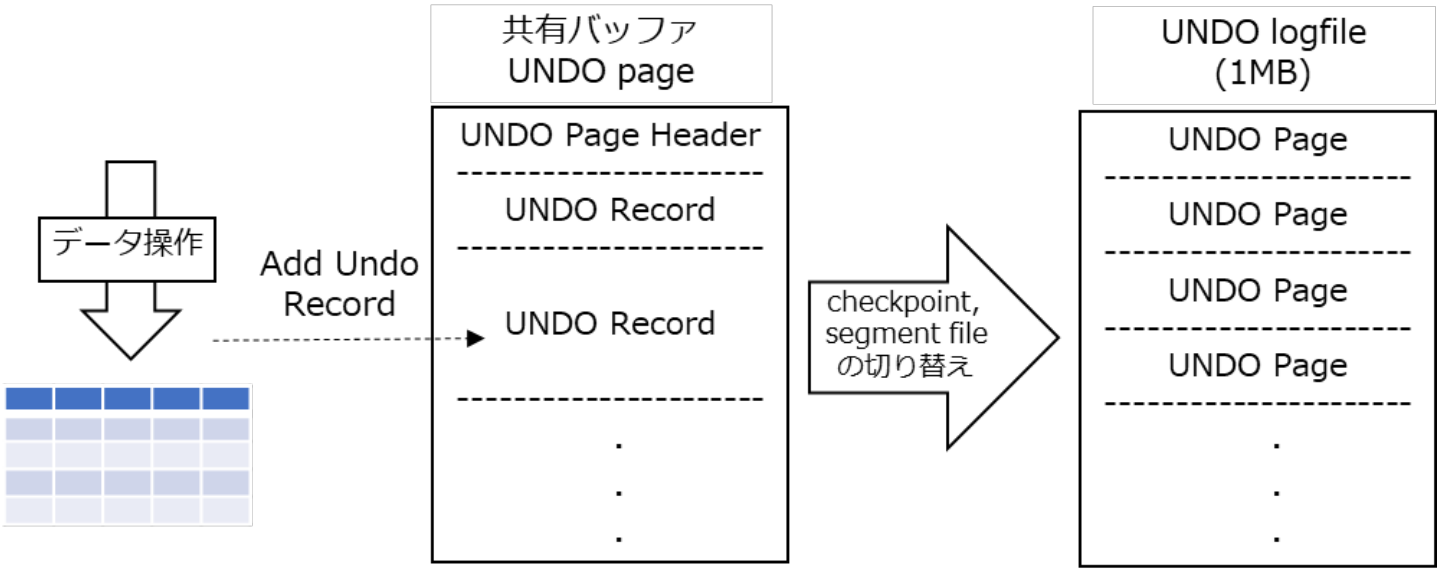
5.7.3. UNDOログの構造

※UNDOログについては2020年3月現在、アーキテクチャの変更について議論が行われているため、本書で説明する情報と正式版との構造が異なる可能性がある点にご注意ください。

UNDOログは1MBのセグメントファイルとして、$PGDATA/base/undo または、$PGDATA/pg_tblspc/VERSION/undo 配下に作成されます。
UNDOログはバックエンドプロセス単位にファイルが割り当てられ、セグメントファイルのアドレススペースが不足するまでは同じファイルが使用されます。（なお、unlogged relationへの変更はunlogged undo logに記録されます。）

UNDOログはWALと同様にリサイクルされる番号付きのセグメントファイルとしてディスクに保存されます。ただし、WALとは異なりテーブルへのアクセスのように共有バッファを介してファイルにアクセスします。バッファ上のデータはチェックポイントまたはセグメントファイルの切替時にファイルに書き出されます。

UNDOログはテーブルデータと同様にページ単位に管理されており、通常のPosgreSQLページヘッダ(チェックサムやLSNを含む)を使用しています。 1つのUNDO Pageには、複数のUNDOレコードが格納されており、要求されたバイト数に応じてスペースが割当てられています。



なお、UNDOログのメタデータの変更はWALに記録されていますが、極力WALトラフィックを抑えるため、レコードの挿入ポイントの情報については書き出しを行わない等の工夫がされています。

DB操作時で、UNDOログに追加されるUndo Recordのヘッダは以下のように定義されています。

```
typedef struct UndoRecordHeader
{
  RmgrId         urec_rmid;              /* RMGR [XXX:TODO: this creates an alignment
                                          * hole?] */
  uint8          urec_type;              /* record type code */
  uint8          urec_info;              /* flag bits */
  uint16         urec_prevlen;           /* length of previous record in bytes */
  Oid            urec_reloid;            /* relation OID */
  TransactionId  urec_prevxid;
  TransactionId  urec_xid;               /* Transaction id */
  CommandId      urec_cid;               /* command id */
} UndoRecordHeader;
```

5.7.4. 構築手順

zheapは現在、開発中のAMであるため、バイナリとして提供はされていません。

そのため、本検証ではGithubからダウンロードしたソースからビルドを行い、検証を行っています。

```
$ sudo su - postgres
$ git clone https://github.com/EnterpriseDB/zheap
$ cd ~/zheap
$ export PGHOME=~/local/zheap/pgsql
$ ./configure --prefix=${PGHOME} \
        --bindir=${PGHOME}/bin \
        --libdir=${PGHOME}/lib \
        --sysconfdir=${PGHOME}/etc \
        --includedir=${PGHOME}/include \
        --datadir=${PGHOME}/share \
        --with-pgport=5432 \
        --with-python \
        --with-openssl \
        --with-pam \
        --with-ldap \
        --with-libxml \
        --with-libxslt
$ make
$ make install
```

DBクラスタ作成と起動の手順は以下のとおりです。

```
$ export PATH=/var/lib/pgsql/local/zheap/pgsql/bin:$PATH
$ export PGDATA=~/local/zheap/data
$ initdb -D ${PGDATA} --no-locale --encoding=utf8
$ pg_ctl -D ${PGDATA} start
```

以降の検証では複数回実施した平均値で評価を行っています。 また、複数回実行する際、キャッシュヒットが起こらないように以下の操作を実行しています。

```
$ pg_ctl -w -D $PGDATA restart
$ sudo sysctl -w vm.drop_caches=1
```

5.7.5. 検証用テーブルの定義

参照性能及び、更新性能に関する検証用のテーブルとして、以下の定義のテーブルを使用しました。 各Access Methodのテーブルには初期データとして、100万レコードを挿入しました。

```
CREATE TABLE zfoo(id int, val1 text, val2 text, val3 text) USING zheap;  -- heapでは、USING heapをする
INSERT INTO zfoo (SELECT generate_series(1,1000000),md5(random()::text),md5(random()::text),md5(random()::text));
CREATE INDEX ON foo USING btree (id);
```

5.7.6. バルク挿入性能

各Access Methodを使用したStar Schema benchmark の lineorder テーブルに対してSF=10相当のデータ（1200万行）のCOPYを行った際のレイテンシを比較しました。

比較の結果、heapとzheapにおけるトランザクション全体のレイテンシの差は1%未満であり、heapとの比較で、zheapには性能差がないことを確認できました。

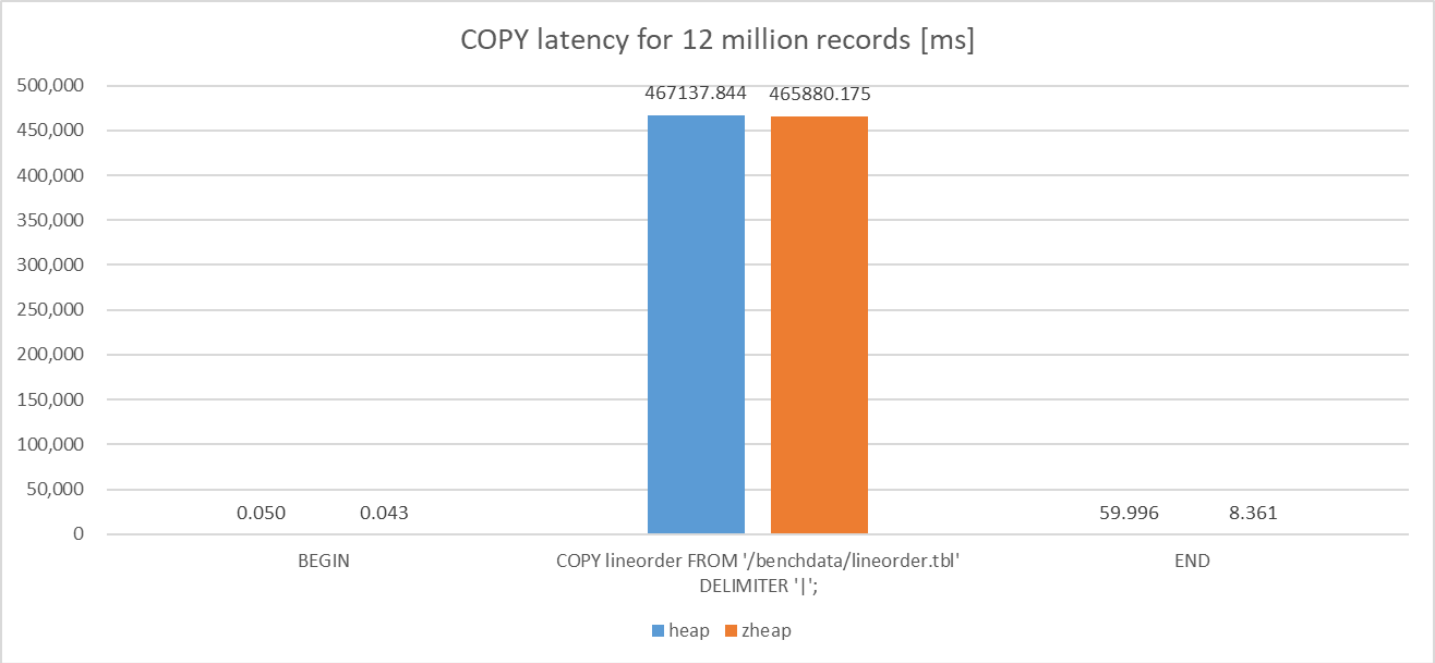


図 5.2 COPY latency for 12 million records

検証手順は以下のとおりです。環境変数TESTDIRは環境に合わせて適宜読み替えてください。

```
$ pgbench postgres -j 1 -c 1 -r -t 1 -f ${TESTDIR}/sqlSet/bulkinsert.sql
```

5.7.7. 参照性能

- 単一行のインデックススキャン

各Access Methodを使用した検証用のテーブルに対して、1行をランダムで取得する処理を同時接続数1で60秒間クエリを実行しました。

検証の結果、単一行の参照に対するlatencyについては、heapとの比較で、zheapには性能差がないことを確認できました。

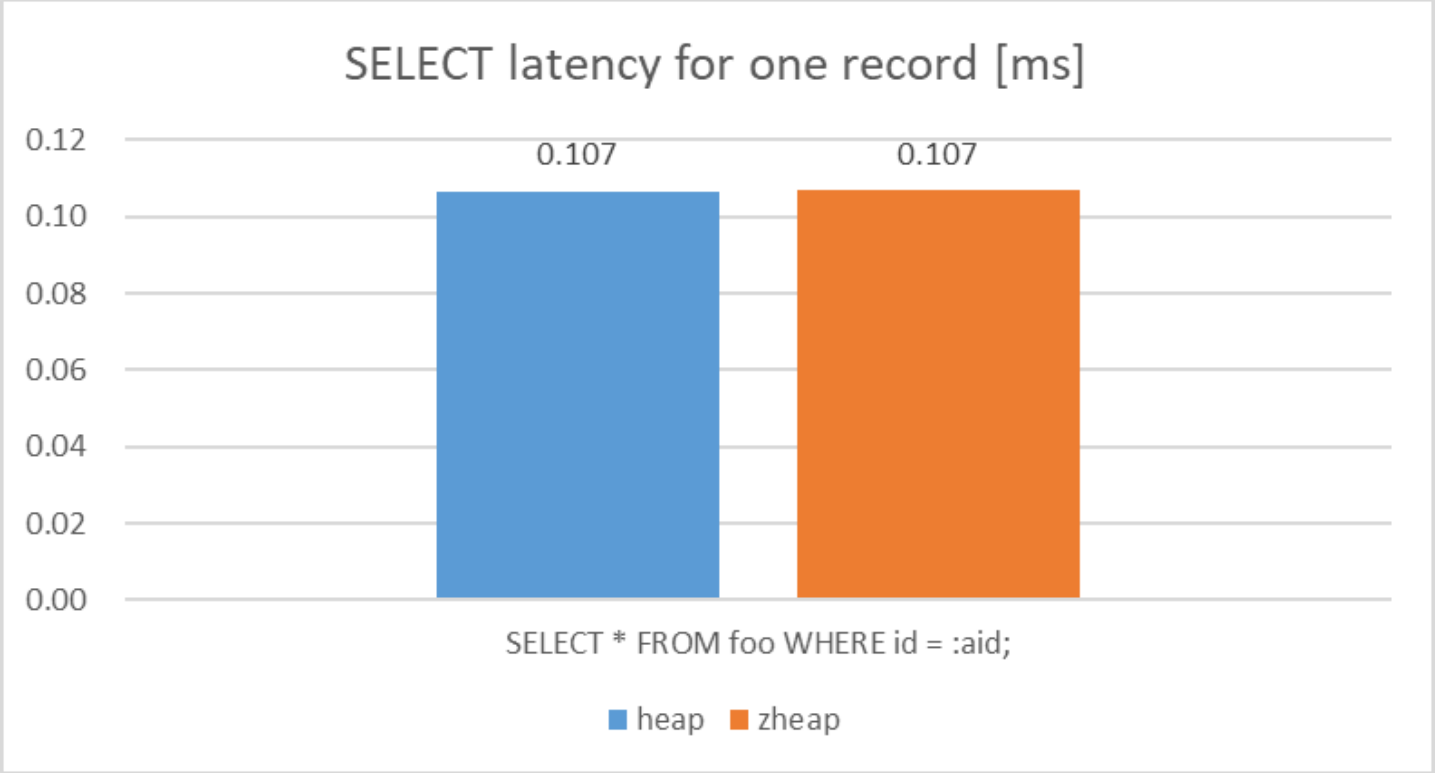


図 5.3 Latency of SELECT query (1 record)

参照用スクリプトは以下のとおりです。

```
\set aid random(1,1000000)
SELECT * FROM foo WHERE id = :aid;
```

検証手順は以下のとおり。

```
$ pgbench postgres -c 1 -j 1 -T 60 -r -f ${TESTDIR}/sqlSet/select_single.sql
```

- 連続した値域の範囲検索（インデックススキャン）

各Access Methodを使用した検証用のテーブルに対して、1000行をランダムで取得する処理を同時接続数1で60秒間クエリを実行しました。

検証の結果、範囲検索に対するlatencyについては、heapとの比較で、zheapは10%程度遅くなることが確認できました。

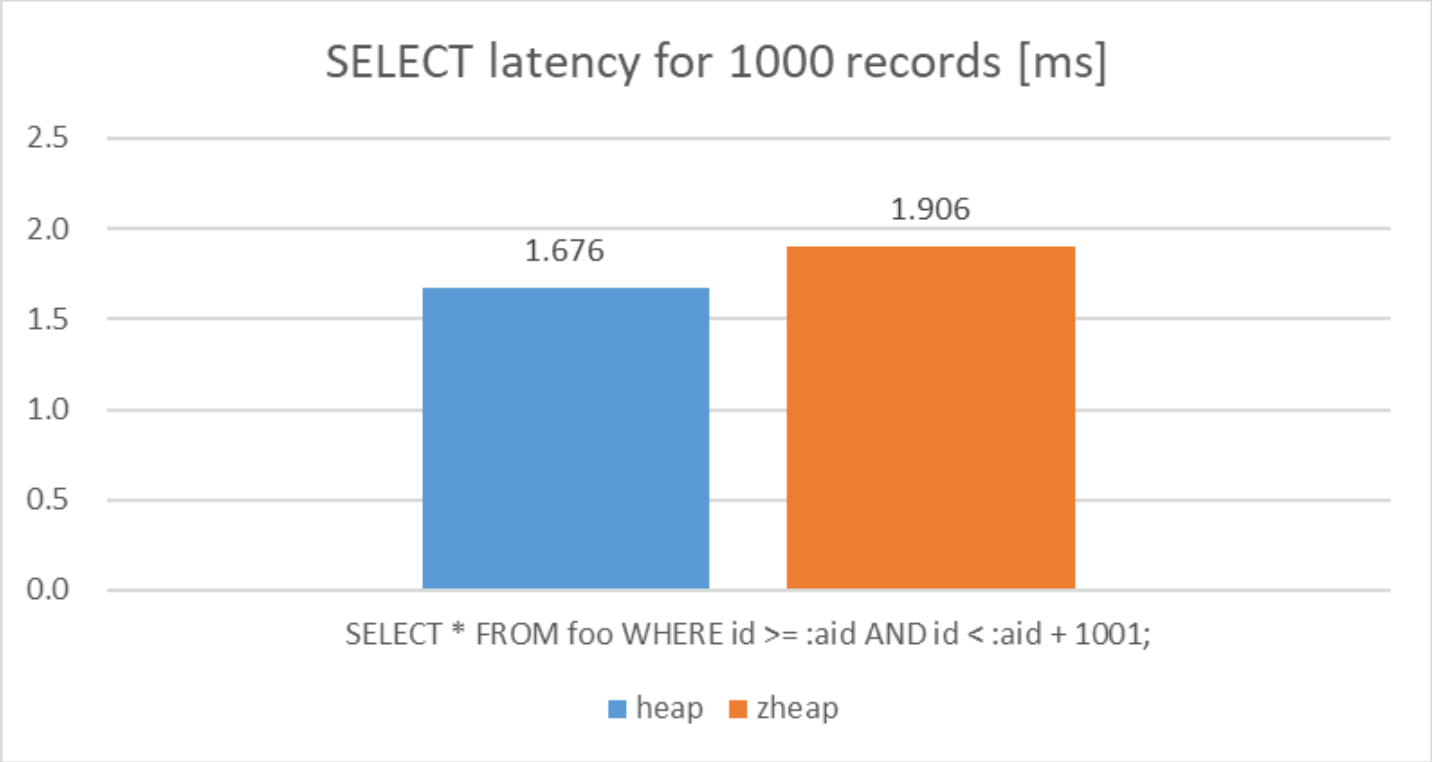


図 5.4 Latency of SELECT query (1000 records)

各テーブルへの参照用スクリプトは以下のとおり。

```
\set aid random(1,999000)
SELECT * FROM foo WHERE id >= :aid AND id < :aid + 1001;
```

検証手順は以下のとおり。

```
$ pgbench -c 1 -j 1 -T 60 -r -f select_range.sql
```

5.7.8. 更新性能

- 単一行に対する更新

各Access Methodを使用した検証用のテーブルに対して、1行をランダムで更新する処理を同時接続数1で60秒間クエリを実行しました。

検証の結果、単一行の更新に対するlatencyについては、heapとの比較で、zheapは概ねheapと同等以上の性能が出ていることを確認できまし

た。

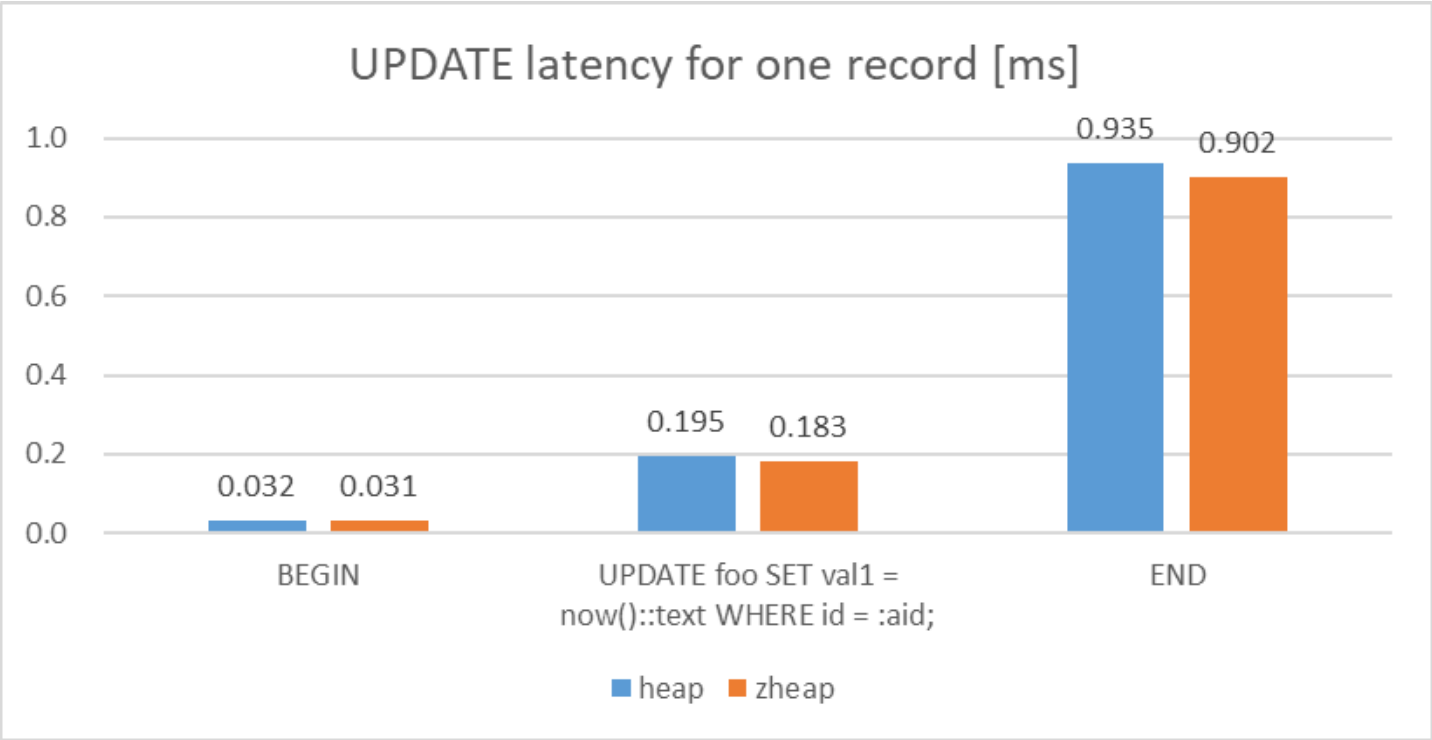


図 5.5 Latency of UPDATE query (1 record)

各テーブルへの更新用スクリプトは以下のとおり。

```
\set aid random(1,1000000)
BEGIN;
UPDATE foo SET val1 = now()::text WHERE id = :aid;
END;
```

検証手順は以下のとおり。

```
$ pgbench -c 1 -j 1 -T 60 -r -f
update_single.sql
```

- 連続した値域に対する範囲更新

各Access Methodを使用した検証用のテーブルに対して、1000行をランダムで更新する処理を同時接続数1で60秒間クエリを実行しました。

検証の結果、範囲更新に対するlatencyについては、heapとの比較で、zheapはUPDATE処理で25%、COMMIT処理では50%程度、性能が良いという結果を確認できました。

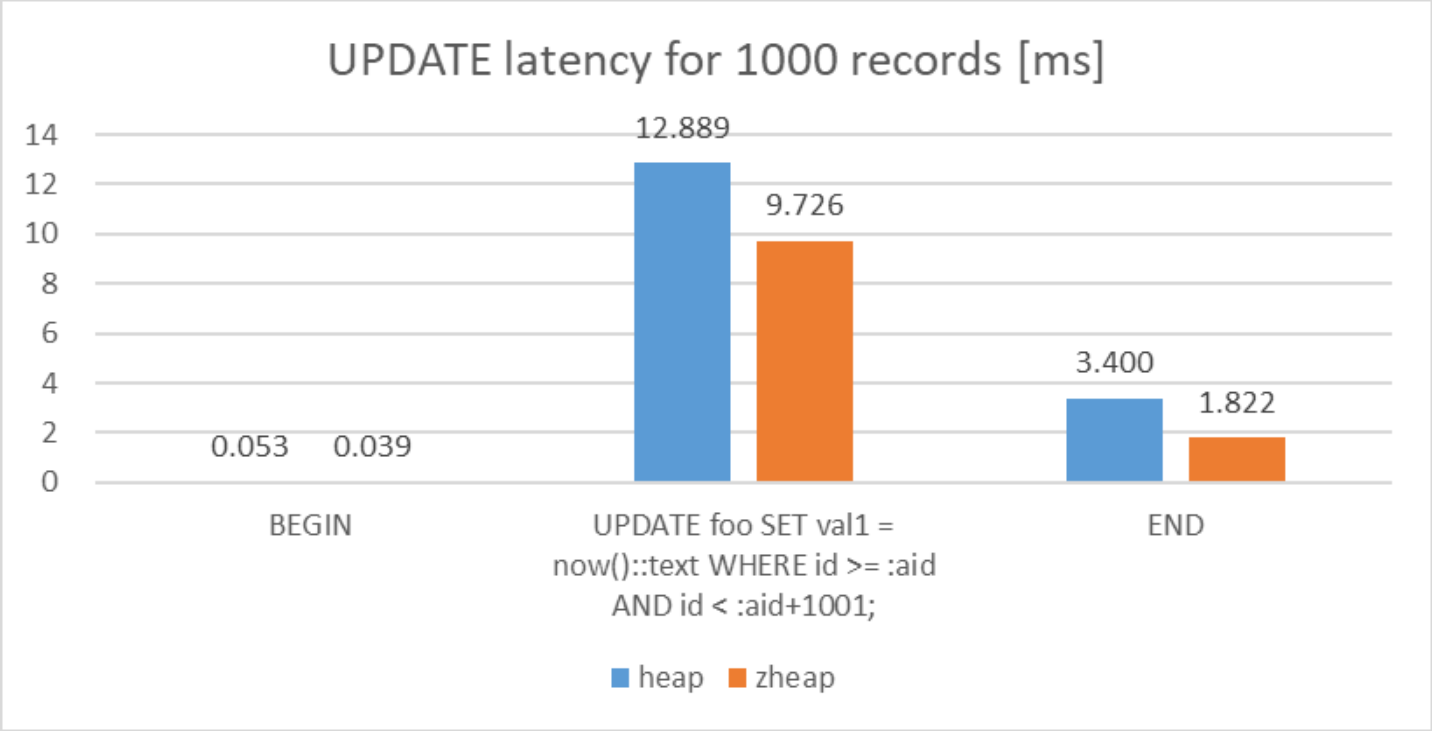


図 5.6 Latency of UPDATE query (1000 records)

```
\set aid random(1,999000)
BEGIN;
UPDATE foo SET val1 = now()::text WHERE id >= :aid AND id < :aid+1001;
END;
```

検証手順は以下のとおり。

```
$ pgbench -c 1 -j 1 -T 60 -r -f update_range.sql
```

5.7.9. ベンチマークモデル比較(pgbench)

pgbenchで使用するテーブルを各Access Methodで作成し、それらのテーブルを対象に同時接続数1でTPC-Bを実行しました。

検証の結果、TPC-Bの各クエリのlatencyについては、これまでの検証結果に沿った結果（参照性能は同等以上、更新、COMMITの性能はzheap優位）を確認することができました。

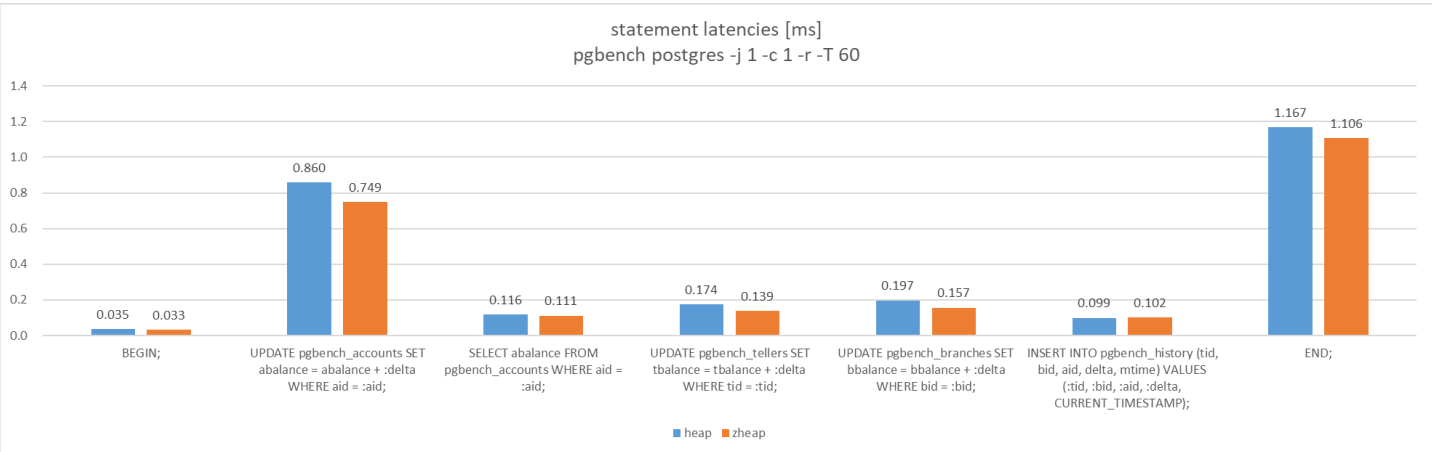


図 5.7 Latency of query like TPC-B

検証手順は以下のとおり。

```
$ pgbench -i -s 200 postgres
$ pgbench -c 1 -j 1 -T 60 -r
```

5.7.10. AM特性の確認

5.7.10.1. テーブルサイズ比較

各Access Methodでpgbench_accounts（pgbench）、lineorder（StarSchemaBenchmark）のテーブルを作成し、初期データ登録時のテーブルサイズの比較を行いました。

検証の結果、テーブルサイズについては、heapとの比較で、zheapではpgbench_accounts、lineorderともに15%程度サイズが圧縮されることが確認できました。

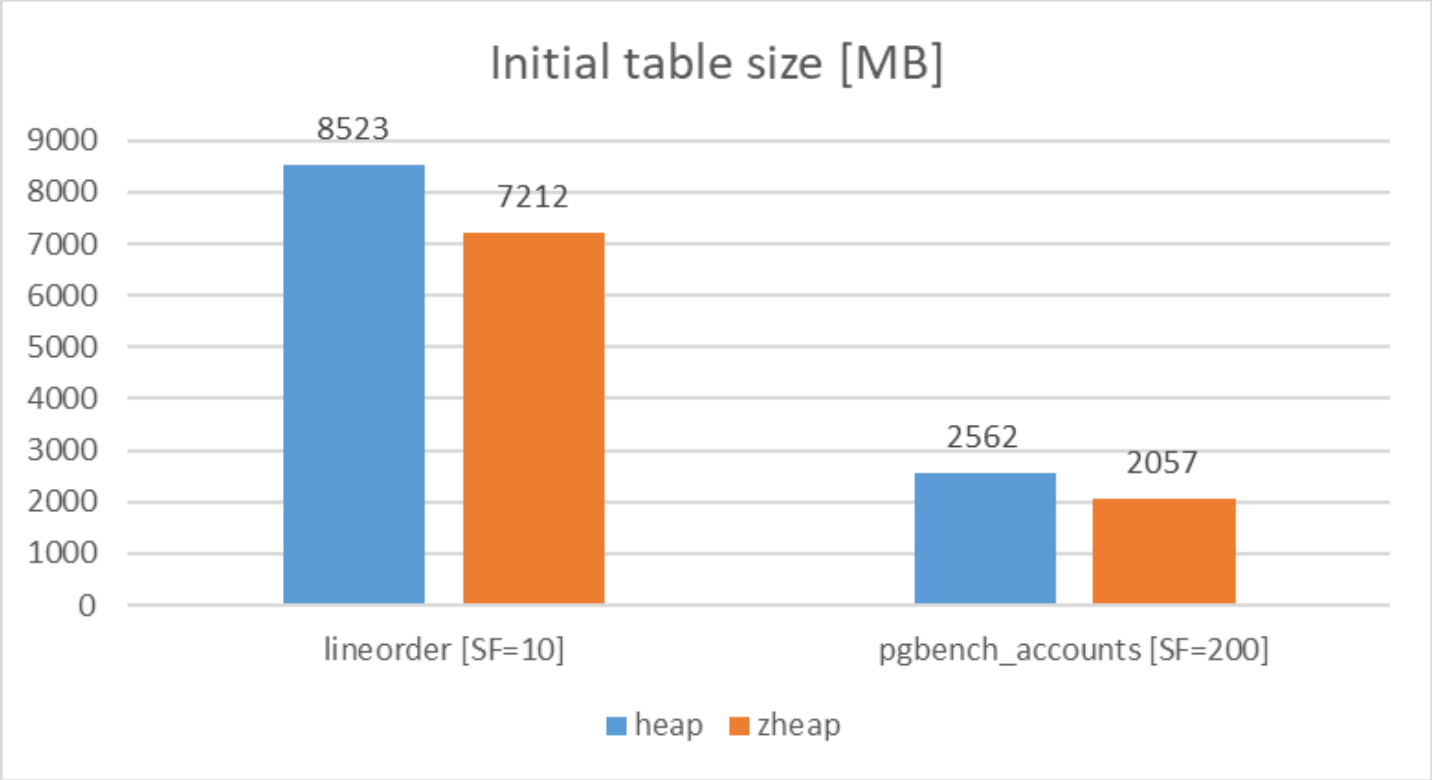


図 5.8 Table size comparison

5.7.10.2. テーブルサイズの変化

各Access Methodでpgbench_accounts（pgbench）のテーブルを作成し、TPC-B実行後(約62万トランザクション実行後)にテーブルが肥大化しているかどうか確認を行いました。 なお、本検証では肥大化することを確認するため、FILLFACTOR=100でテーブルを定義し、HOT更新が機能しない肥大化が発生しやすい環境下で実施しています。

検証の結果、テーブルサイズの肥大化については、heapが1.5%程度サイズが増加していることに対して、zheapではテーブルサイズが増加しないことを確認できました。

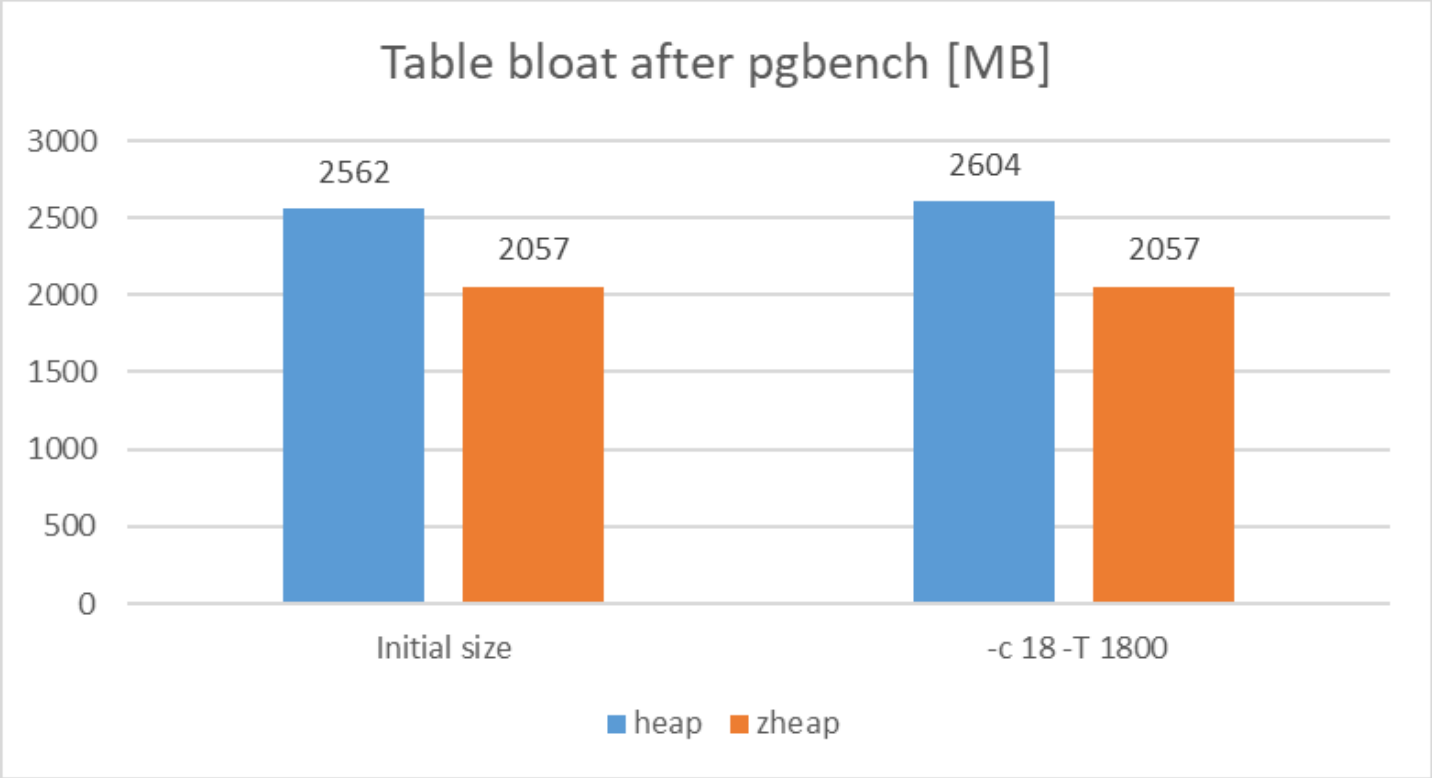


図 5.9 Table bloat size comparison

5.7.11. 設計、運用に関する確認

DB設計、運用に関する確認として、DBのメンテナンス処理やコア機能として提供されているものとzheapが併用可能か確認を行いました。 便宜上、fooテーブルはheapで作成し、zfooテーブルはzheapで作成したものとして実行結果を記載しています。

結果のサマリは以下のとおりです。

項目	zheapでの利用可否	備考
メンテナンス (VACUUM/ANALYZE/CLUSTER/REINDEX)	○（利用可能）	CLUSTERコマンドの実行ログにおいてzheapの非削除対象レコードがカウントされなかった。
バックアップ/リカバリ	○	heapとの比較において、特に違いは見られなかった。
ストリーミングレプリケーション	○	同上。
ロジカルレプリケーション	○	同上。
宣言的パーティション	○	同上。
パラレルクエリ	○	同上。
FILLFACTOR	○	同上。
追加モジュール(pgstattuple)	×（利用不可）	heap以外には対応していない。

5.7.11.1. メンテナンス（VACUUM/ANALYZE/CLUSTER/REINDEX）

- VACUUM

前述の更新性能の検証で使用したzheapのテーブルに対して、VACUUMを実行し、その結果を確認しました。

検証の結果、zheapのテーブルに大量にUPDATEを行ってもdead_tupleは発生しませんでした。 また、VACUUMの結果からはインデックスについても不要領域が発生していなかったことを確認しました。

各Access Methodのテーブルに更新を行った後の稼働統計情報とVACUUM実行時の結果は以下のとおりです。

```
postgres=# select * from pg_stat_user_tables where relname = 'foo';
-[ RECORD 1 ]-------+------------------------------
relid               | 16424
schemaname          | public
```

```
relname             | foo
seq_scan            | 2
seq_tup_read        | 1000000
idx_scan            | 43071
idx_tup_fetch       | 43071
n_tup_ins           | 1000000
n_tup_upd           | 43071
n_tup_del           | 0
n_tup_hot_upd       | 27689
n_live_tup          | 1000000
n_dead_tup          | 27412 ★ **zheapの場合はdead_tupは発生しない**
n_mod_since_analyze | 43071
last_vacuum         |
last_autovacuum     |
last_analyze        |
last_autoanalyze    | 2020-03-26 15:43:34.122166+09
vacuum_count        | 0
autovacuum_count    | 0
analyze_count       | 0
autoanalyze_count   | 1

Timing is on.
postgres=# vacuum verbose foo ;
psql: INFO:  vacuuming "public.foo"
psql: INFO:  scanned index "foo_id_idx" to remove 15382 row versions
DETAIL:  CPU: user: 0.44 s, system: 0.00 s, elapsed: 0.44 s
psql: INFO:  "foo": removed 15382 row versions in 15382 pages
DETAIL:  CPU: user: 0.11 s, system: 0.00 s, elapsed: 0.11 s
psql: INFO:  index "foo_id_idx" now contains 1000000 row versions in 2745 pages
DETAIL:  15382 index row versions were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
psql: INFO:  "foo": found 15200 removable, 1000000 nonremovable row versions in 16646 out of 16646
pages
DETAIL:  0 dead row versions cannot be removed yet, oldest xmin: 85917
There were 303 unused item identifiers.
Skipped 0 pages due to buffer pins, 0 frozen pages.
0 pages are entirely empty.
CPU: user: 0.77 s, system: 0.02 s, elapsed: 0.80 s.
psql: INFO:  vacuuming "pg_toast.pg_toast_16424"
psql: INFO:  index "pg_toast_16424_index" now contains 0 row versions in 1 pages
DETAIL:  0 index row versions were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
psql: INFO:  "pg_toast_16424": found 0 removable, 0 nonremovable row versions in 0 out of 0 pages
DETAIL:  0 dead row versions cannot be removed yet, oldest xmin: 85917
There were 0 unused item identifiers.
Skipped 0 pages due to buffer pins, 0 frozen pages.
0 pages are entirely empty.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
VACUUM
Time: 821.434 ms

postgres=# select * from pg_stat_user_tables where relname = 'zfoo';
-[ RECORD 1 ]-------+------------------------------
relid               | 16431
schemaname          | public
relname             | zfoo
seq_scan            | 2
seq_tup_read        | 1000000
idx_scan            | 42323
idx_tup_fetch       | 42323
n_tup_ins           | 1000000
n_tup_upd           | 42323
n_tup_del           | 0
n_tup_hot_upd       | 0
n_live_tup          | 1000000
```

```
n_dead_tup         | 0 ★ **heapの場合はdead_tupが発生する**
n_mod_since_analyze | 42323
last_vacuum         |
last_autovacuum     |
last_analyze        |
last_autoanalyze    | 2020-03-26 15:43:36.012058+09
vacuum_count        | 0
autovacuum_count    | 0
analyze_count       | 0
autoanalyze_count   | 1

postgres=# vacuum verbose zfoo ;
psql: INFO:  vacuuming "public.zfoo"
psql: INFO:  index "zfoo_id_idx" now contains 1000000 row versions in 2745 pages
DETAIL:  0 index row versions were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
psql: INFO:  "zfoo": found 0 removable, 1000000 nonremovable row versions in 13889 out of 13890 pages
DETAIL:  0 dead row versions cannot be removed yet, oldest xmin: 42845
There were 0 unused item pointers.
0 pages are entirely empty.
CPU: user: 0.23 s, system: 0.02 s, elapsed: 0.25 s.
psql: INFO:  vacuuming "pg_toast.pg_toast_16431"
psql: INFO:  index "pg_toast_16431_index" now contains 0 row versions in 1 pages
DETAIL:  0 index row versions were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
psql: INFO:  "pg_toast_16431": found 0 removable, 0 nonremovable row versions in 0 out of 1 pages
DETAIL:  0 dead row versions cannot be removed yet, oldest xmin: 42845
There were 0 unused item pointers.
0 pages are entirely empty.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
VACUUM
Time: 272.954 ms
```

- ANALYZE

前述の参照性能、更新性能の検証で使用したzheapのテーブルに対して、ANALYZEを実行し、その結果を確認しました。

検証の結果、heapと比較して、特に違いはないことを確認できました。

検証時の実行結果は以下のとおりです。

```
postgres=# ANALYZE VERBOSE foo;
psql: INFO:  analyzing "public.foo"
psql: INFO:  "foo": scanned 16646 of 16646 pages, containing 1000000 live rows and 0 dead rows; 30000 rows in sample, 1000000 estimated tota
ANALYZE
Time: 339.199 ms

postgres=# SELECT * FROM pg_stats WHERE tablename = 'foo' AND attname = 'id';
-[ RECORD 1 ]----------+-----------------------------------------------------------------------------------------------------------------------------------------------------
schemaname             | public
tablename              | foo
attname                | id
inherited              | f
null_frac              | 0
avg_width              | 4
n_distinct             | -1
most_common_vals       |
most_common_freqs      |
histogram_bounds       |
{64,9490,18922,28171,37975,47840,58337,67905,77951,87567,97060,108158,117758,127807,139084,148310,158854,168819,179599,189223
correlation            | 0.9691951
most_common_elems      |
most_common_elem_freqs |
elem_count_histogram   |
Time: 18.989 ms

postgres=# ANALYZE VERBOSE zfoo;
psql: INFO:  analyzing "public.zfoo"
psql: INFO:  "zfoo": scanned 13890 of 13890 pages, containing 1000000 live rows and 0 dead rows; 30000 rows in sample, 1000000 estimated to
ANALYZE

Time: 573.389 ms

postgres=# SELECT * FROM pg_stats WHERE tablename = 'zfoo' AND attname = 'id';
-[ RECORD 1 ]----------+-----------------------------------------------------------------------------------------------------------------------------------------------------
schemaname             | public
tablename              | zfoo
attname                | id
inherited              | f
null_frac              | 0
avg_width              | 4
n_distinct             | -1
most_common_vals       |
most_common_freqs      |
histogram_bounds       |
{10,10151,20422,32249,41569,50540,60285,70652,80788,91039,101666,111648,122191,131666,140720,152042,162194,171540,181507,1924
correlation            | 1
most_common_elems      |
most_common_elem_freqs |
elem_count_histogram   |

Time: 4.352 ms
```

- CLUSTER

前述の参照性能、更新性能の検証で使用したzheapのテーブルに対して、CLUSTERを実行し、その結果を確認しました。

検証の結果、heapと比較して、zheapの場合は削除対象ではない100万レコードが表示されませんでした。

検証時の実行結果は以下のとおりです。

```
postgres=# CLUSTER VERBOSE foo USING foo_id_idx ;
psql: INFO:  clustering "public.foo" using index scan on "foo_id_idx"
psql: INFO:  "foo": found 0 removable, 1000000 nonremovable row versions in 16394 pages
DETAIL:  0 dead row versions cannot be removed yet.
CPU: user: 1.13 s, system: 0.18 s, elapsed: 1.53 s.
CLUSTER
Time: 2242.479 ms (00:02.242)

postgres=# CLUSTER VERBOSE zfoo USING zfoo_id_idx ;
psql: INFO:  clustering "public.zfoo" using index scan on "zfoo_id_idx"
psql: INFO:  "zfoo": found 0 removable, 0 nonremovable row versions in 13890 pages
DETAIL:  0 dead row versions cannot be removed yet.
CPU: user: 0.98 s, system: 0.19 s, elapsed: 1.93 s.
CLUSTER
Time: 2433.532 ms (00:02.434)
```

- REINDEX

各Access Methodを使用した対象テーブルで定義されたB-treeインデックスを対象にREINDEXを実行し、確認を行いました。

検証の結果、heapと比較して、特に違いがないことが確認できました。

検証時の実行結果は以下のとおりです。

```
postgres=# REINDEX ( VERBOSE ) INDEX foo_id_idx;
psql: INFO:  index "foo_id_idx" was reindexed
DETAIL:  CPU: user: 0.59 s, system: 0.04 s, elapsed: 0.98 s
REINDEX
Time: 718.343 ms

postgres=# REINDEX ( VERBOSE ) INDEX zfoo_id_idx;
psql: INFO:  index "zfoo_id_idx" was reindexed
DETAIL:  CPU: user: 0.65 s, system: 0.04 s, elapsed: 1.15 s
REINDEX
Time: 828.794 ms
```

5.7.11.2. バックアップ/リカバリ

zheapを使用したテーブルに対して、バックアップの取得及び、PITRが実施可能か確認しました。

確認の結果、zheapを使用したテーブルであっても問題なく、バックアップ及びリカバリを実行できることを確認できました。

検証時の実行結果は以下のとおりです。

```
# 初期データ登録
psql postgres -c "CREATE TABLE bar (id int);INSERT INTO bar VALUES (1);"

# basebackup取得
pg_basebackup -h localhost -D /tmp/rep

# 初期データ登録
psql postgres -c "CREATE TABLE bar2 (id int) USING zheap;\
                  INSERT INTO bar2 VALUES (1);\
                  SELECT pg_switch_wal();\
                  INSERT INTO bar2 VALUES (2);\
                  SELECT statement_timestamp();"
psql postgres -c "SELECT pg_sleep(30);\
                  INSERT INTO bar2 VALUES (3),(4);\
                  SELECT pg_switch_wal();"

# リカバリ用の設定を追記
echo "port = 5433" >> /tmp/rep/postgresql.conf
echo "recovery_target_time = '2020-02-27 15:27:05'" >> /tmp/rep/postgresql.conf
echo "recovery_target_action = 'promote'" >> /tmp/rep/postgresql.conf
echo "archive_mode = off" >> /tmp/rep/postgresql.conf
echo "restore_command = 'cp /var/lib/pgsql/local/zheap/arch/%f %p'" >> /tmp/rep/postgresql.conf

# PITR実行及び確認
pg_ctl -D /tmp/rep start
psql postgres -p 5433 -c "SELECT * FROM bar2;"
 id
----
  1
  2
(2 rows)
```

5.7.11.3. ストリーミングレプリケーション

zheapを使用したテーブルに対して、ストリーミングレプリケーション（以降、SRとする）が実施可能か確認しました。 本検証は、1台の環境で実施しているため、セカンダリはport番号を変更し起動しました。

確認の結果、zheapを使用したテーブルであっても問題なく、SR構成を構築可能であることを確認しました。

検証時の実行結果は以下のとおりです。

```
# セカンダリの作成(standby.signalを自動作成)
pg_basebackup -h localhost -D /tmp/rep -R

# SR用の設定を追記
echo "port = 5433" >> /tmp/rep/postgresql.conf
echo "archive_mode = off" >> /tmp/rep/postgresql.conf
echo "primary_conninfo = 'user=postgres host=localhost port=5432'" >> /tmp/rep/postgresql.conf

# セカンダリ起動
pg_ctl -D /tmp/rep start

# レプリケーション状態の確認
psql postgres -c "SELECT client_addr,state,sync_state FROM pg_stat_replication"

# zheapAMを使用したテーブルを作成
psql postgres -c "CREATE TABLE bar (id int) USING zheap;\
                  INSERT INTO bar VALUES (1);"

# セカンダリのテーブルを参照する
psql postgres -p 5433 -c "SELECT * FROM bar"
 id
----
  1
(1 row)
```

5.7.11.4. ロジカルレプリケーション

zheapを使用したテーブルに対して、ロジカルレプリケーションが実施可能か確認しました。 本検証は、1台の環境で実施しているため、セカンダリはport番号を変更し起動しました。

確認の結果、zheapを使用したテーブルであっても問題なく、ロジカルレプリケーションが可能であることを確認しました。

検証時の実行結果は以下のとおりです。

```
# ベースバックアップを作成する
pg_basebackup -h localhost -D /tmp/rep
echo "port = 5433" >> /tmp/rep/postgresql.conf
echo "archive_mode = off" >> /tmp/rep/postgresql.conf

# セカンダリを起動する
pg_ctl -D /tmp/rep start

# プライマリでパブリケーションを作成する
psql postgres -c "CREATE TABLE japan (id int, pref text, city text, data text) USING zheap;\
                  CREATE PUBLICATION pub_srv1_testdb_japan FOR TABLE japan;"

# セカンダリでサブスクライバを作成する
psql postgres -p 5433 -c "CREATE TABLE japan (id int, pref text, city text, data text) USING zheap;"
psql postgres -p 5433 -c "CREATE SUBSCRIPTION sub_srv2_testdb_japan CONNECTION 'dbname=postgres port=5432 user=postgres'
PUBLICATION pub_srv1_testdb_japan;"

# プライマリに確認用のデータをINSERTする
psql postgres -c "INSERT INTO japan VALUES (1, '東京', '港', 'らーめん 象');\
                  INSERT INTO japan VALUES (2, '神奈川', '横浜', '象Love亭');"

# セカンダリにデータがINSERTされていることを確認する
psql postgres -p 5433 -c "SELECT * FROM japan"
 id |  pref  | city |    data
----+--------+------+-------------
  1 | 東京   | 港   | らーめん 象
  2 | 神奈川 | 横浜 | 象Love亭
(2 rows)
```

5.7.11.5. 宣言的パーティション

宣言的パーティションを構築する際、対象テーブルでzheapを利用可能か確認しました。 確認の結果、zheapを使用したテーブルであっても問題なく、宣言的パーティションの構成を作成できることを確認しました。 親テーブルへのAM指定はPostgreSQLとしてサポートされていないので実施できませんが、子テーブルへの指定は可能でした。また、プルーニングについても有効であることを確認しました。

検証時の実行結果は以下のとおりです。

```
postgres=# CREATE TABLE japan ( pref text, city text, data text ) PARTITION BY LIST (pref) USING zheap;
psql: ERROR:  specifying a table access method is not supported on a partitioned table

postgres=# CREATE TABLE japan ( pref text, city text, data text ) PARTITION BY LIST (pref);
CREATE TABLE
postgres=# CREATE TABLE kanagawa PARTITION OF japan FOR VALUES IN ('神奈川') USING zheap;
CREATE TABLE
postgres=# \d+ japan
                        Partitioned table "public.japan"
 Column | Type | Collation | Nullable | Default | Storage  | Stats target | Description
--------+------+-----------+----------+---------+----------+--------------+-------------
 pref   | text |           |          |         | extended |              |
 city   | text |           |          |         | extended |              |
 data   | text |           |          |         | extended |              |
Partition key: LIST (pref)
Partitions: kanagawa FOR VALUES IN ('神奈川')

postgres=# \d+ kanagawa
                           Table "public.kanagawa"
 Column | Type | Collation | Nullable | Default | Storage  | Stats target | Description
--------+------+-----------+----------+---------+----------+--------------+-------------
 pref   | text |           |          |         | extended |              |
 city   | text |           |          |         | extended |              |
 data   | text |           |          |         | extended |              |
Partition of: japan FOR VALUES IN ('神奈川')
Partition constraint: ((pref IS NOT NULL) AND (pref = '神奈川'::text))
Access method: zheap

postgres=# INSERT INTO japan VALUES ('東京','町田','リス'),('神奈川','横浜','シウマイ'),('神奈川','川崎','タンタンメン');
INSERT 0 3
postgres=# EXPLAIN SELECT * FROM japan;
                          QUERY PLAN
---------------------------------------------------------------
 Append  (cost=0.00..63.75 rows=2250 width=96)
   ->  Seq Scan on tokyo  (cost=0.00..17.50 rows=750 width=96)
   ->  Seq Scan on kanagawa  (cost=0.00..17.50 rows=750 width=96)
   ->  Seq Scan on shizuoka  (cost=0.00..17.50 rows=750 width=96)
(4 rows)

postgres=# EXPLAIN SELECT * FROM japan WHERE pref = '神奈川';
                       QUERY PLAN
---------------------------------------------------------
 Seq Scan on kanagawa  (cost=0.00..19.38 rows=4 width=96)
   Filter: (pref = '神奈川'::text)
(2 rows)
```

5.7.11.6. パラレルクエリ

zheapを使用したテーブルに対してパラレルクエリが利用可能か確認をしました。 確認の結果、zheapを使用したテーブルであってもパラレルクエリは問題なく動作することを確認できました。

検証時の実行結果は以下のとおりです。

```
-bash-4.2$ psql postgres -c "COPY zlineorder FROM '/benchdata/lineorder.tbl' DELIMITER '|'"
-bash-4.2$ psql postgres
psql (12beta1)
Type "help" for help.

postgres=# set max_parallel_workers_per_gather TO 2;
SET
postgres=# explain analyze select count(*) from zlineorder ;
                                                          QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=1236243.51..1236243.52 rows=1 width=8) (actual time=37240.607..37240.607 rows=1 loops=1)
   ->  Gather  (cost=1236243.29..1236243.50 rows=2 width=8) (actual time=37239.741..37240.709 rows=3 loops=1)
         Workers Planned: 2
         Workers Launched: 2
         ->  Partial Aggregate  (cost=1235243.29..1235243.30 rows=1 width=8) (actual time=37214.960..37214.961 rows=1 loops=3)
               ->  Parallel Seq Scan on zlineorder  (cost=0.00..1172758.03 rows=24994103 width=0) (actual time=0.941..25695.379
rows=19995351 loops=3)
 Planning Time: 0.215 ms
 Execution Time: 37240.879 ms
(8 rows)
-bash-4.2$ sudo sysctl -w vm.drop_caches=1
vm.drop_caches = 1

-bash-4.2$
-bash-4.2$ psql postgres
psql (12beta1)
Type "help" for help.

postgres=# explain analyze select count(*) from zlineorder ;
                                                    QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=1672640.10..1672640.11 rows=1 width=8) (actual time=37817.039..37817.040 rows=1 loops=1)
   ->  Seq Scan on zlineorder  (cost=0.00..1522675.48 rows=59985848 width=0) (actual time=2.136..25350.578 rows=59986052 loops=1)
 Planning Time: 0.575 ms
 Execution Time: 37822.773 ms
(4 rows)
```

5.7.11.7. FILLFACTOR

zheapを使用したテーブルに対してfillfactorが利用可能か確認をしました。

確認の結果、zheapを使用したテーブルであってもfillfactorは定義できることが確認できました。fillfactorによって、充填率が下がったことにより、テーブルサイズが拡張したことも確認できました。

しかし、zheapは追記型のアーキテクチャではないため、fillfactorを設定する意味はないため、HOT等の恩恵を得られない点に注意が必要です。

検証時の実行結果は以下のとおりです。

```
postgres=#   CREATE TABLE zfoo(id int, val1 text, val2 text, val3 text) USING zheap WITH (fillfactor='100');
CREATE TABLE
postgres=#   INSERT INTO zfoo (SELECT generate_series(1,1000000),md5(random()::text),md5(random()::text),md5(random()::text));
INSERT 0 1000000
postgres=#   CREATE INDEX ON zfoo USING btree (id)
postgres-# \d+
                    List of relations
 Schema | Name | Type  |  Owner   |  Size  | Description
--------+------+-------+----------+--------+-------------
 public | zfoo | table | postgres | 109 MB |
(1 row)

postgres=# drop table zfoo;
DROP TABLE
postgres=#   CREATE TABLE zfoo(id int, val1 text, val2 text, val3 text) USING zheap WITH (fillfactor='80');
CREATE TABLE
postgres=#   INSERT INTO zfoo (SELECT generate_series(1,1000000),md5(random()::text),md5(random()::text),md5(random()::text));
INSERT 0 1000000
postgres=#   CREATE INDEX ON zfoo USING btree (id)
postgres-# \d+
                    List of relations
 Schema | Name | Type  |  Owner   |  Size  | Description
--------+------+-------+----------+--------+-------------
 public | zfoo | table | postgres | 137 MB |
```

5.7.12. 追加モジュール

- pgstattuple

zheapを使用したテーブルに対してpgstattupleが利用可能か確認をしました。

確認の結果、pgstattupleはheap以外のテーブルには対応していないことが確認できました。

検証時の実行結果は以下のとおりです。

```
psql postgres -c "CREATE EXTENSION pgstattuple"
psql postgres -c "CREATE TABLE bar (id int) USING zheap;"
psql postgres -c "SELECT * FROM pgstattuple('bar');"
ERROR:  only heap AM is supported
```

5.8. zheapの検証結果に対する考察

本検証では、PostgreSQL 12で実装されたPlugabble Storageの検証として、現在開発中のAccess Methodであるzheapの検証を行いました。

zheapの主目的である肥大化の抑制についてですが、HOTを無効化した肥大化が発生しやすいテーブルでpgbenchによる検証を行った結果、zheapではテーブルサイズが変化しないことを確認できました。

参照系については、1レコードに対するlatencyはheapと同等の性能でしたが、範囲検索においてはzheapでわずかに性能が低いことを確認しました。 更新系については、UPDATE、COMMITにおいてzheapが有利という結果になり、処理するレコード数を増やした場合も同様の結果となりました。

テーブルサイズについては、不要なタプルヘッダの削減により、heapと比較して概ね2割程度のデータ量削減効果があることが確認できました。また、DB設計や運用に関連する要素として、DBメンテナンス系のコマンドやPostgreSQLがサポートする機能についての動作確認を行い、特にzheapであることを意識することなく使用できることを確認できました。

5.9. zedstore

5.9.1. zedstoreの概要

Greenplumが中心に開発を行っているカラムナフォーマットをサポートするAccess Methodです。

カラムナの特徴として、列指向のデータフォーマットであるため、heap等の行指向のデータフォーマットと比較してスキャンするデータ量を削減することができます。 また、zedstoreではカラムナデータフォーマットに加え、lz4によるデータ圧縮を行っており、さらなるIO量の削減を実現

しています。

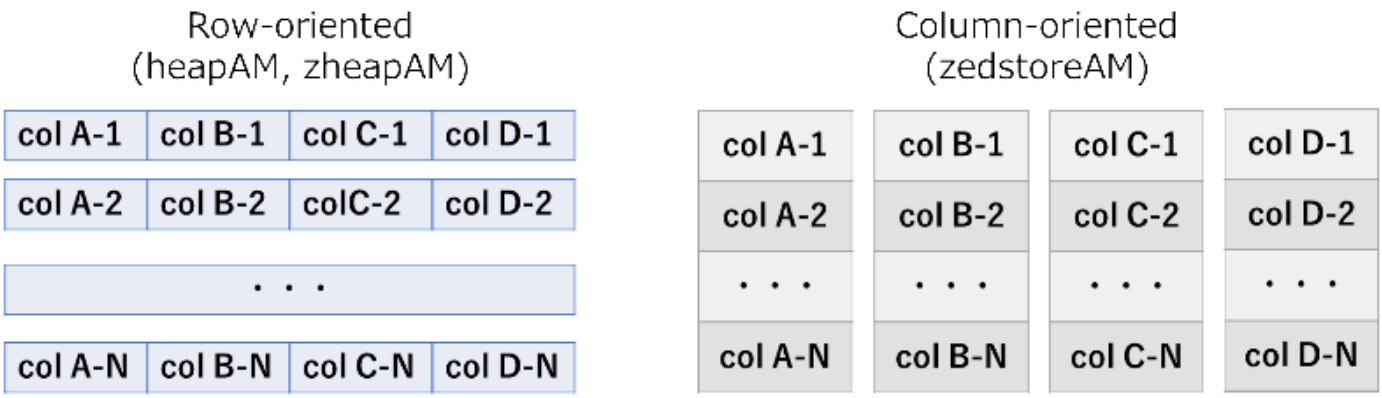


図 5.10 Image of data format

5.9.2. テーブル構造

zedstoreで定義されたテーブルは複数のBtreeとその他メタページ等で構成されます。

zedstoreで定義されたテーブルはTID(tuple ID) tree と呼ばれる各タプルの可視性情報を含んだBtree、さらにユーザデータが格納されるAttribute treeがカラム数分が存在します。ちなみにここで言うBtreeはBtreeインデックスとは関係ないため、混同しないように注意する必要があります。

zedstoreでのMVCCは、zheapと同様でUNDOログで実現しています。UNDOログとの紐付けに関してですが、zheapでは各ページにトランザクションスロットを持つことで実現していたのに対して、zedstoreではアイテム単位に直接UNDOログとの紐付け情報を持つような構造となっています。（bulk insertを行った場合は、アイテムごとではなく、アイテムの配列に対して紐付き情報が保持されます。）

zedstoreのデータは、以下のように複数の異なるフォーマットのデータが格納され、それぞれページ単位で管理されています。

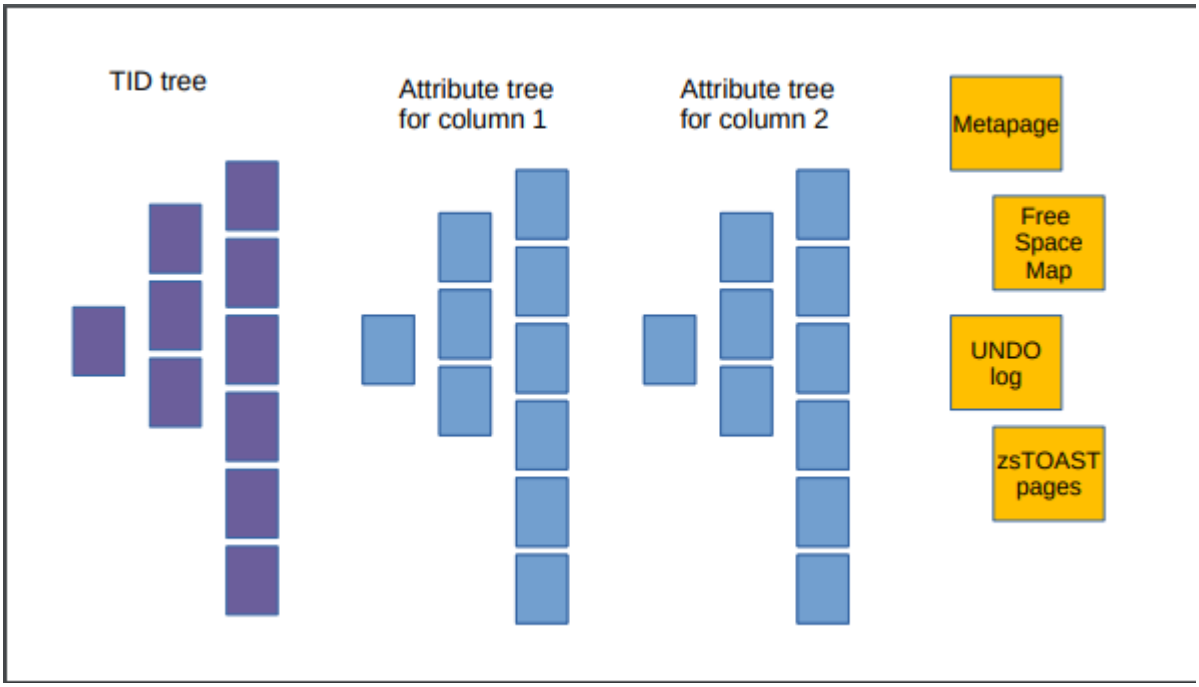


図 5.11 Image of table structure

https://www.postgresql.eu/events/pgconfeu2019/sessions/session/2738/slides/233/ZedStore-PGConfeu2019-Milan.pdf

5.9.3. 構築手順

zedstoreは現在、開発中のAMであるため、バイナリとして提供はされていません。 そのため、本検証ではGithubからダウンロードしたソースからビルドを行い、検証を行っています。

```
$ sudo su - postgres
$ git clone https://github.com/greenplum-db/postgres.git
$ cd postgres
$ git checkout zedstore
$ ./configure --prefix=/var/lib/pgsql/local/zedstore/pgsql --with-pgport=5432 --with-lz4
$ make
$ make install
```

DBクラスタ作成と起動の手順は以下のとおりです。

```
$ export PATH=/var/lib/pgsql/local/zedstore/pgsql/bin:$PATH
$ export PGDATA=~/local/zedstore/data
$ initdb -D ${PGDATA} --no-locale --encoding=utf8
$ pg_ctl -D ${PGDATA} start
```

以降の検証では複数回実施した平均値で評価を行っています。 また、複数回実行する際、キャッシュヒットが起こらないように以下の操作を実行しています。

```
$ pg_ctl -w -D $PGDATA restart
$ sudo sysctl -w vm.drop_caches=1
```

5.9.4. 検証用テーブルの定義

参照性能、更新性能の検証に使用するテーブル定義と挿入データを以下に示します。

なお、zedstore（カラムナストア）の特徴を確認するため、以下のような設計を行っています。

- 参照用テーブル
    - heapとzedstoreで読み込みブロック数の差を大きくする設計を採用します。
        - シーケンシャルスキャンを行わせるため、インデックスを定義しない。
        - カラム数を増やし、レコードあたりのデータ量を大きくする。
        - 圧縮率の異なるカラムを定義する。
        - シーケンシャルスキャンでの読み込み量を増やすため、初期データを前述の検証の5倍にする。
- 更新用テーブル
    - AMによる影響の違いを比較するため、前述のzheapと同様のテーブル定義、データ量を使用します。

```
-- 参照用
CREATE TABLE zfoo(
  id int,
  val1 text,
  val2 text,
  val3 text,
  val4 text,
  val5 text,
  val6 text,
  val7 text,
  val8 text,
  val9 text,
  val10 text,
  val11 text,
  val12 text,
  val13 text,
  val14 text,
  val15 text,
  val16 text,
  val17 text,
  val18 text,
  val19 text,
  val20 text
)USING zedstore;  -- heap用のテーブルでは、USING heapとする

INSERT INTO zfoo (
  SELECT generate_series(1,5000000),
      md5(random()::text),
      md5(random()::text),
      md5(random()::text),
      md5(random()::text),
      md5(random()::text),
      md5(random()::text),
      md5(random()::text),
      md5(random()::text),
      md5(random()::text),
      md5(random()::text),
      md5(random()::text),
      md5(random()::text),
      md5(random()::text),
      md5(random()::text),
      md5(random()::text),
      md5(random()::text),
      md5(random()::text),
      md5(random()::text),
      md5(random()::text),
      'abcdefg'
);
```

更新性能に関する検証用のテーブルには、以下の定義のテーブルを使用しました。

```
-- 更新用
CREATE TABLE zfoo(id int, val1 text, val2 text, val3 text) USING zedstore WITH (fillfactor='80');  -- heap用のテーブルでは、USING heapとする
INSERT INTO zfoo (SELECT generate_series(1,1000000),md5(random()::text),md5(random()::text),md5(random()::text));
CREATE INDEX ON zfoo USING btree (id);
```

5.9.5. バルク挿入性能

各Access Methodを使用したStar Schema benchmark の lineorder テーブルに対してSF=10相当のデータ（1200万行）のCOPYを行った際のレイテンシを比較しました。

比較の結果、heapと比較して、zedstoreではトランザクション全体でレイテンシが45%ほど遅いことを確認できました。

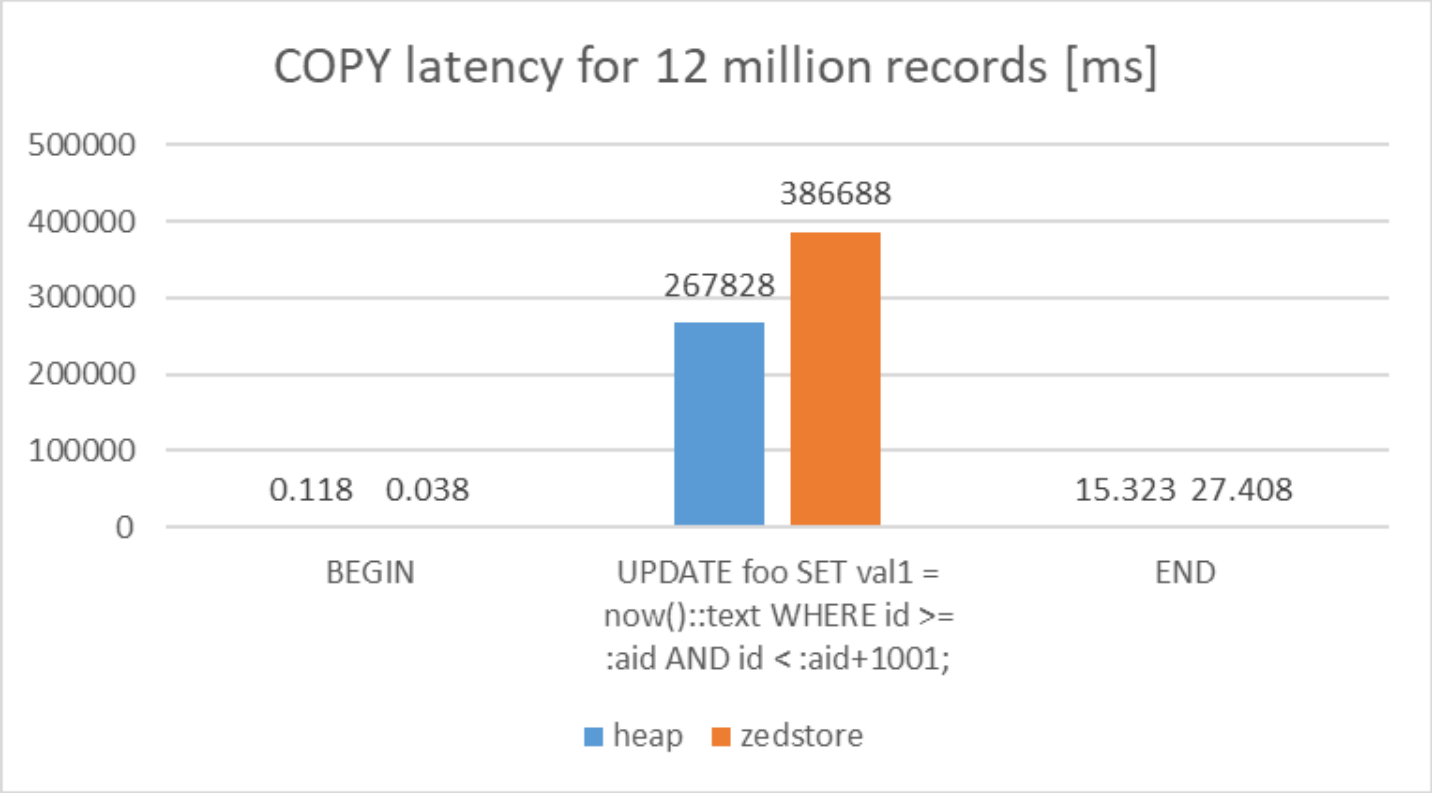


図 5.12 Latency of bulkinsert

検証手順は以下のとおりです。

```
$ pgbench postgres -j 1 -c 1 -r -t 1 -f ${TESTDIR}/sqlSet/bulkinsert.sql
```

5.9.6. 参照性能

- 単一カラムに対するシーケンシャルスキャン

各Access Methodを使用した検証用のテーブルに対して、id、val10、val20のカラムを取得する処理を実行し、読み込んだblock数、actual timeの比較を行いました。

検証の結果、heapとの比較で、zedstoreは圧縮率の高く読み込みblock数が少ないカラム(val20：固定文字列)に対する参照では高速で処理を行うことができることを確認しました。反対に圧縮の効かず読み込みblock数が多いカラム(val10：32桁のランダム文字)に対する参照では、zedstoreが2倍程度遅くなることを確認できました。ある程度圧縮率の高い連続する数値カラムに対する参照ではheapと同等の結果となりました。

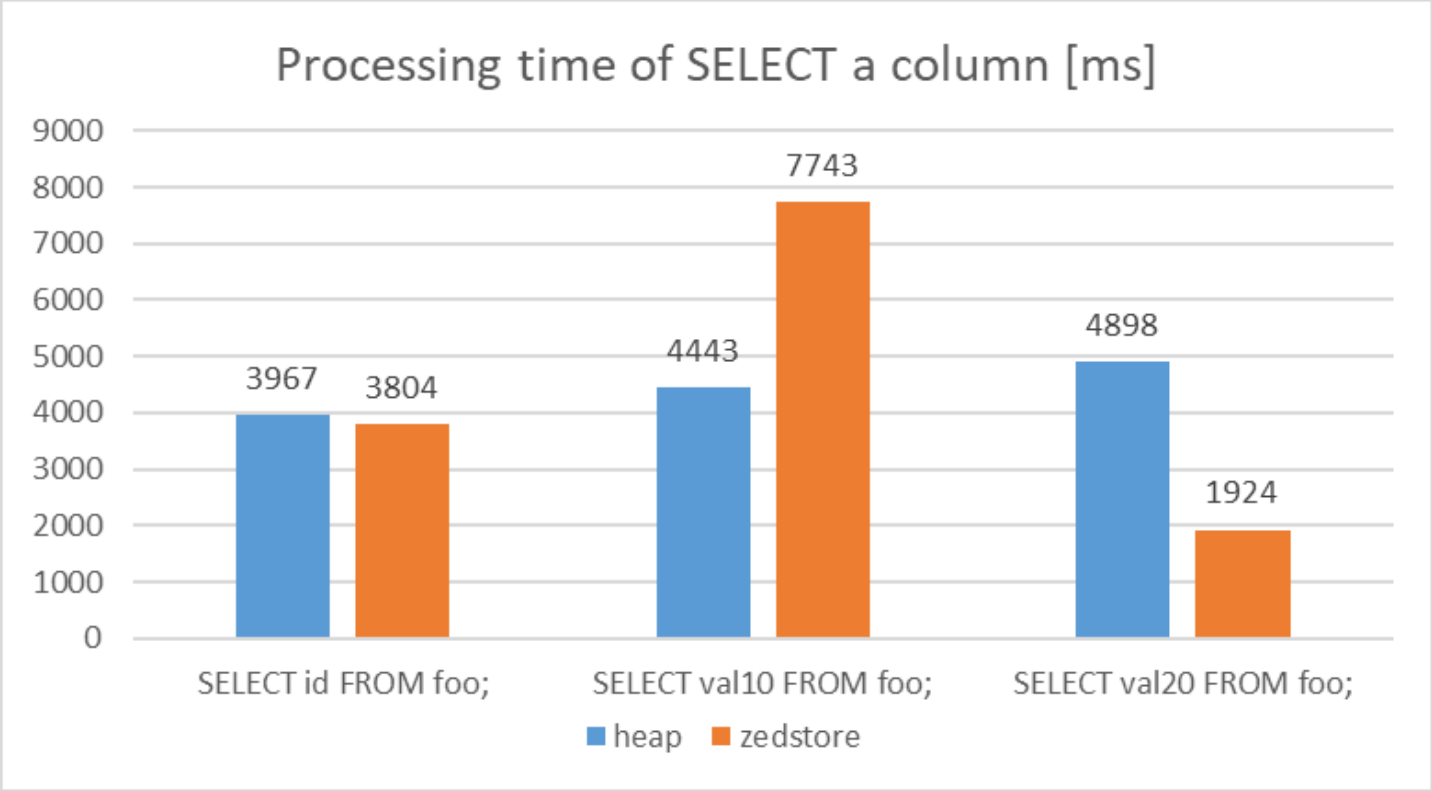


図 5.13 actual time of SELECT query

各AMにおけるカラム別の読み込みブロック数を以下に示します。なお、読み込みブロック数は EXPLAIN ANALYZEのBUFFERSオプションによる出力結果を使用しています。

heapは行指向であるため、読み込みブロック数は一定になっているのに対して、zedstoreでは特定のカラムのみへのreadしか発生していないため、大幅に読み込みブロック数を削減できていることを確認できました。

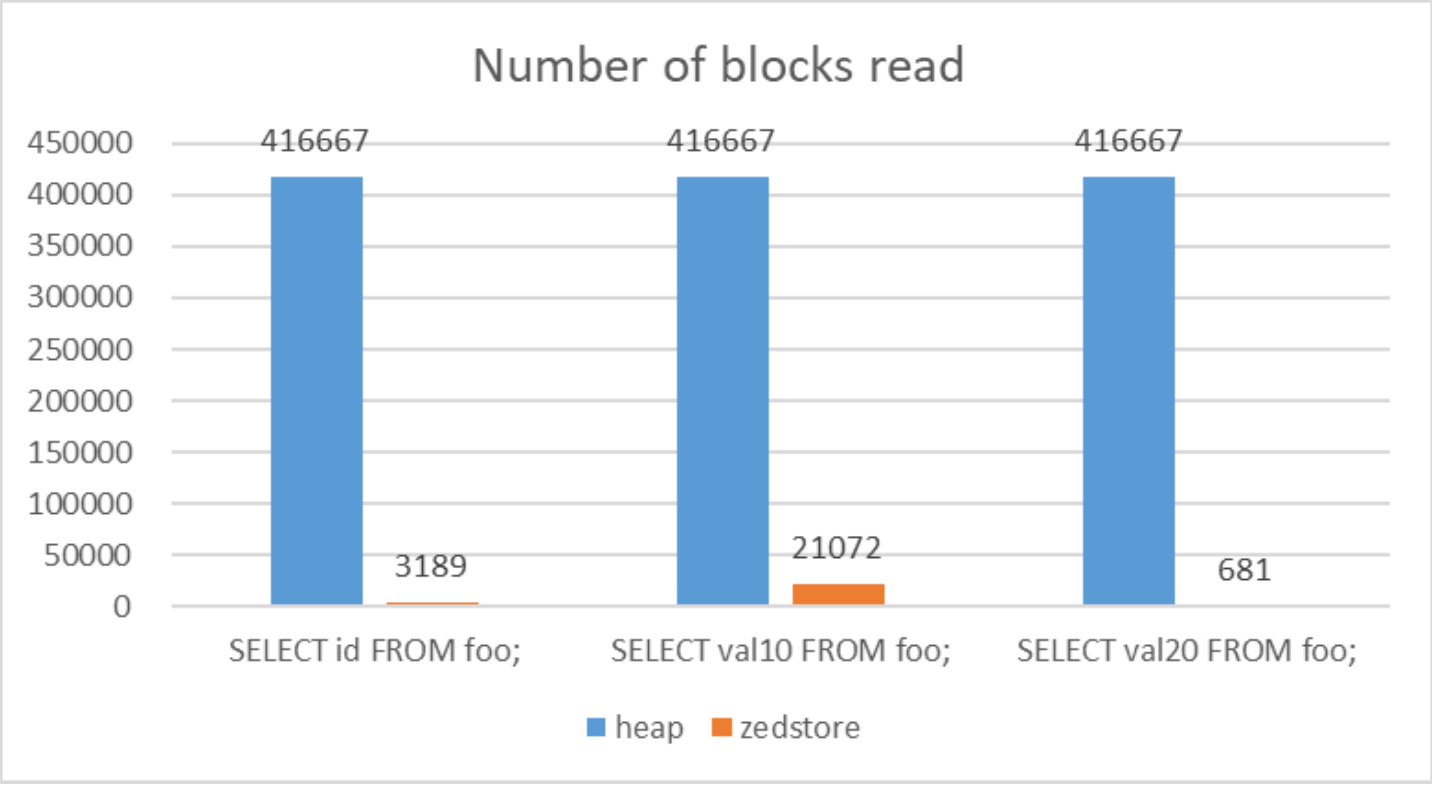


図 5.14 Number of read blocks

zedstoreの読み込みブロック数とactual timeの関係をグラフ化したものを以下に示します。

確認の結果、今回の検証においては6000ブロック以上読み込みが発生する場合はheapのほうが性能的に優位になると推定できます。

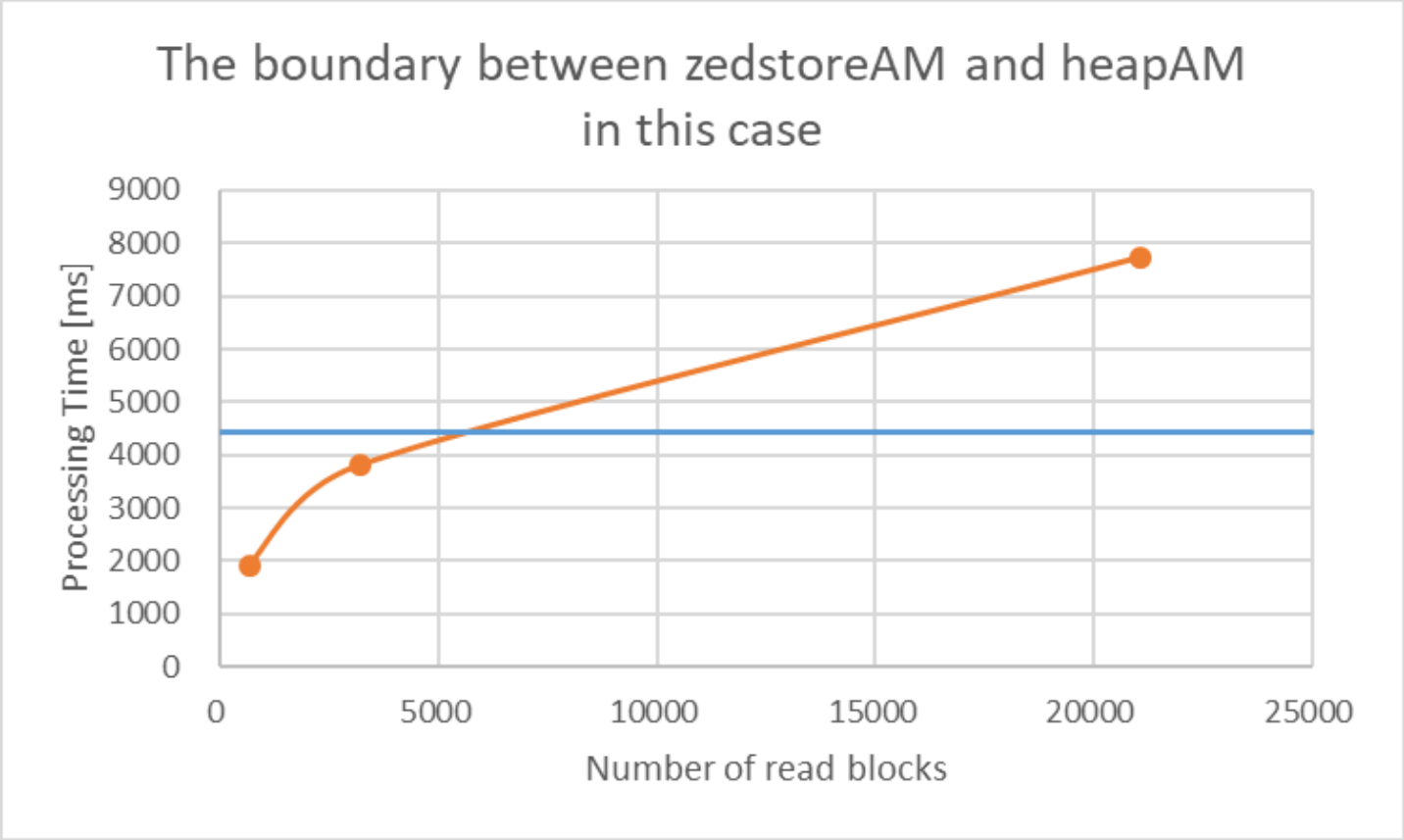


図 5.15 Performance boundary value (read block)

参照用スクリプトは以下のとおりです。

```
EXPLAIN (ANALYZE,VERBOSE,BUFFERS) SELECT id FROM foo; -- 同様にval10, val20についても作成する
```

検証手順は以下のとおり。

```
$ for t in id val10 val20
  do
    psql postgres -f ${TESTDIR}/sqlSet/select_${t}.sql
  done
```

5.9.7. 更新性能

- 単一行に対する更新

各Access Methodを使用した検証用のテーブルに対して、1行をランダムで更新する処理を同時接続数1で10分間クエリを実行しました。

検証の結果、単一行の更新に対するlatencyについては、heapとの比較で、zedstoreは2倍程度遅いことが確認できました。

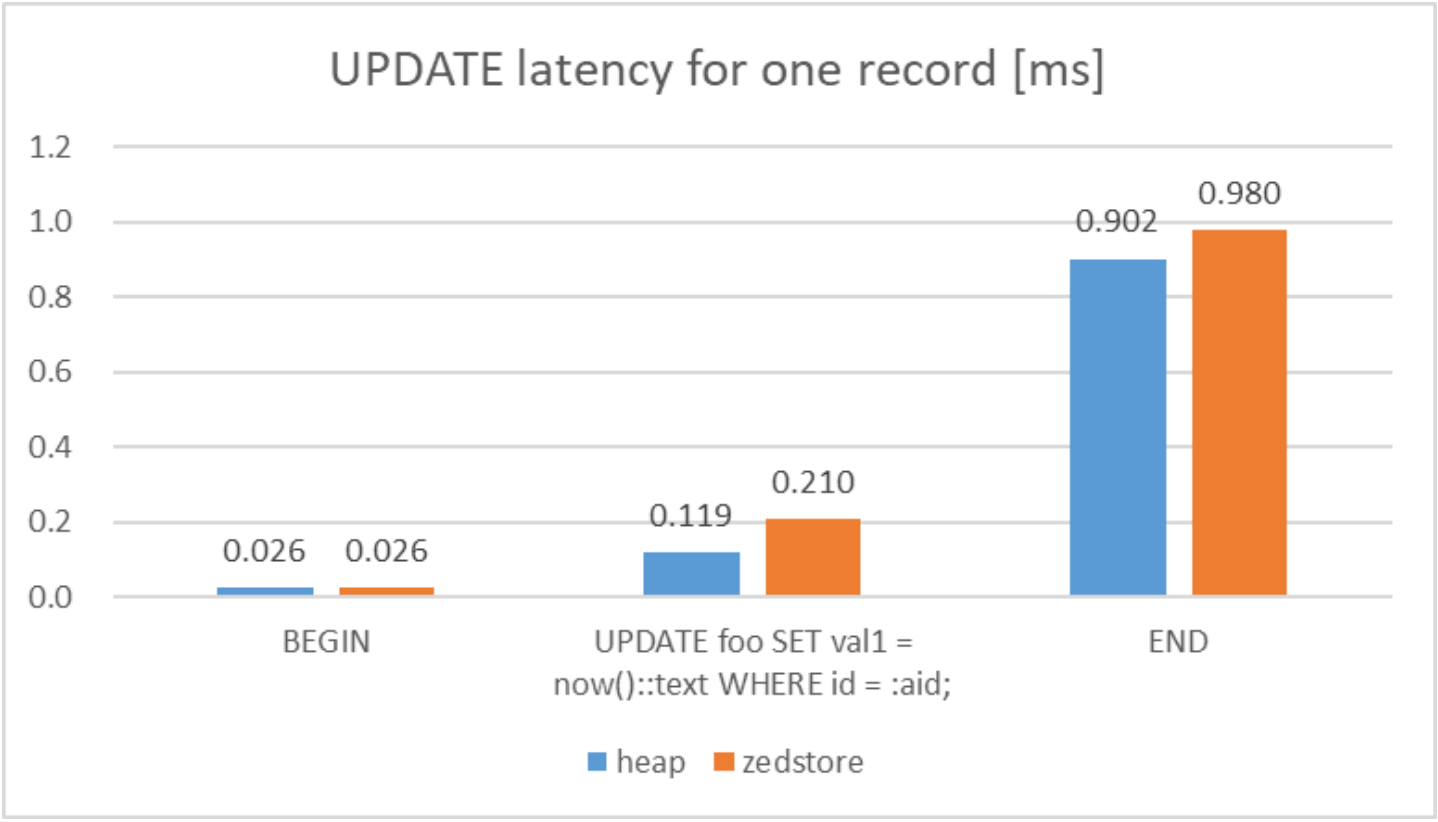


図 5.16 Latency of UPDATE query

各テーブルへの更新用スクリプトは以下のとおり。

```
\set aid random(1,1000000)
BEGIN;
UPDATE foo SET val1 = now()::text WHERE id = :aid;
END;
```

検証手順は以下のとおり。

```
$ pgbench -c 1 -j 1 -T 60 -r -f
update_single.sql
```

- 連続した値域に対する範囲更新

各Access Methodを使用した検証用のテーブルに対して、1000行をランダムで更新する処理を同時接続数1で10分間クエリを実行しました。

検証の結果、範囲更新に対するlatencyについては、heapとの比較で、zedstoreは圧倒的に遅いことが確認できました。

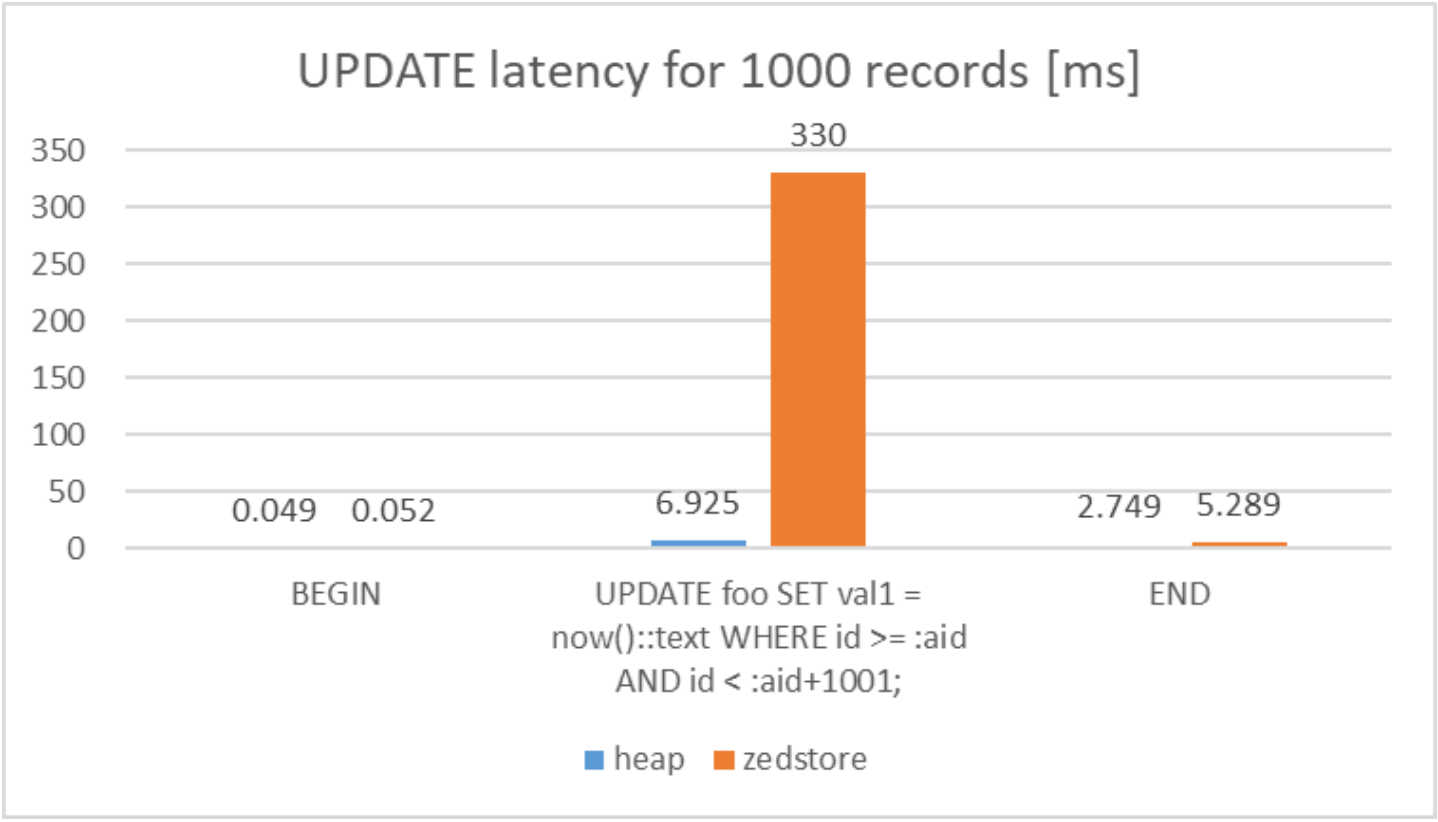


図 5.17 Latency of UPDATE query

```
\set aid random(1,999000)
BEGIN;
UPDATE foo SET val1 = now()::text WHERE id >= :aid AND id < :aid+1001;
END;
```

検証手順は以下のとおり。

```
$ pgbench -c 1 -j 1 -T 60 -r -f update_range.sql
```

5.9.8. ベンチマークモデル比較

5.9.8.1. pgbench(TPC-B)

pgbenchで使用するテーブルを各Access Methodで作成し、それらのテーブルを対象に同時接続数1でTPC-Bを実行しました。

検証の結果、TPC-Bの各クエリのlatencyについては、これまでの検証結果に沿った結果（更新、COMMITの性能はheap優位）を確認することができました。テーブルの規模が小さいため、参照性能はzedstoreが不利な条件ですが、pgbenchの場合、インデックススキャンが効くため、同等の結果になったと考えられます。

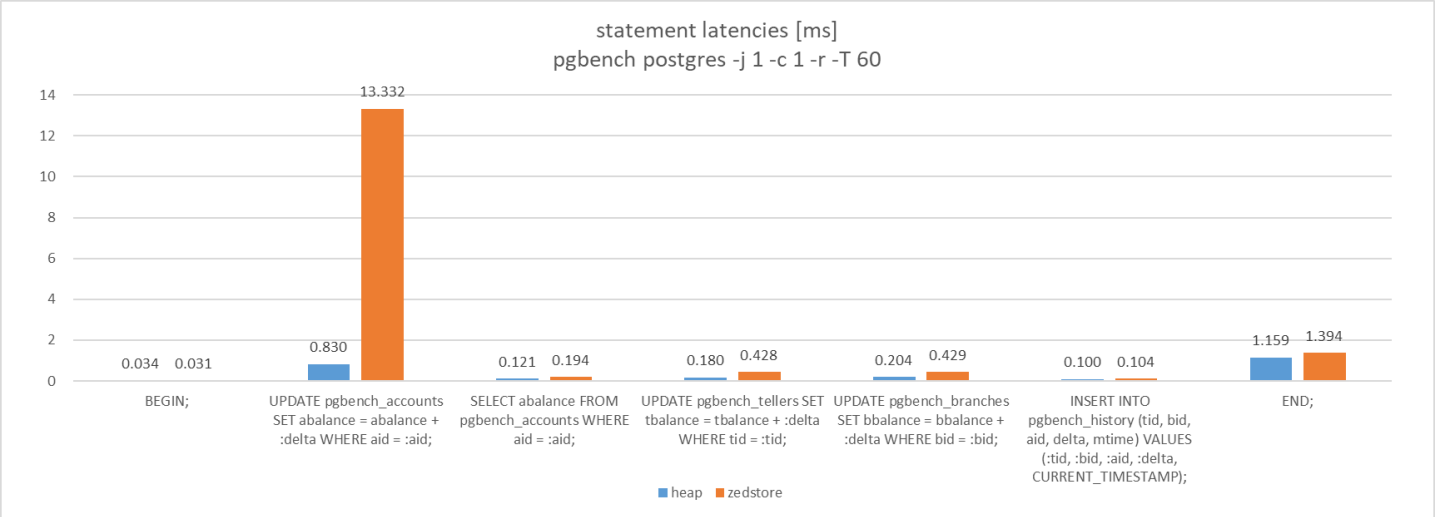


図 5.18 Latency of query like TPC-B

検証手順は以下のとおり。

```
$ pgbench -i -s 200 postgres
$ pgbench -c 1 -j 1 -T 60 -r
```

5.9.8.2. Star Schema Benchmark(OLAP)

ファクトテーブルを各Access Methodで作成し、ディメンションテーブルは全てheapで作成した環境で13種類のクエリを全て実行しました。

検証の結果、全てのクエリパターンにおいて、heapとの比較で、zedstoreでの処理時間が大幅に短縮されることを確認できました。

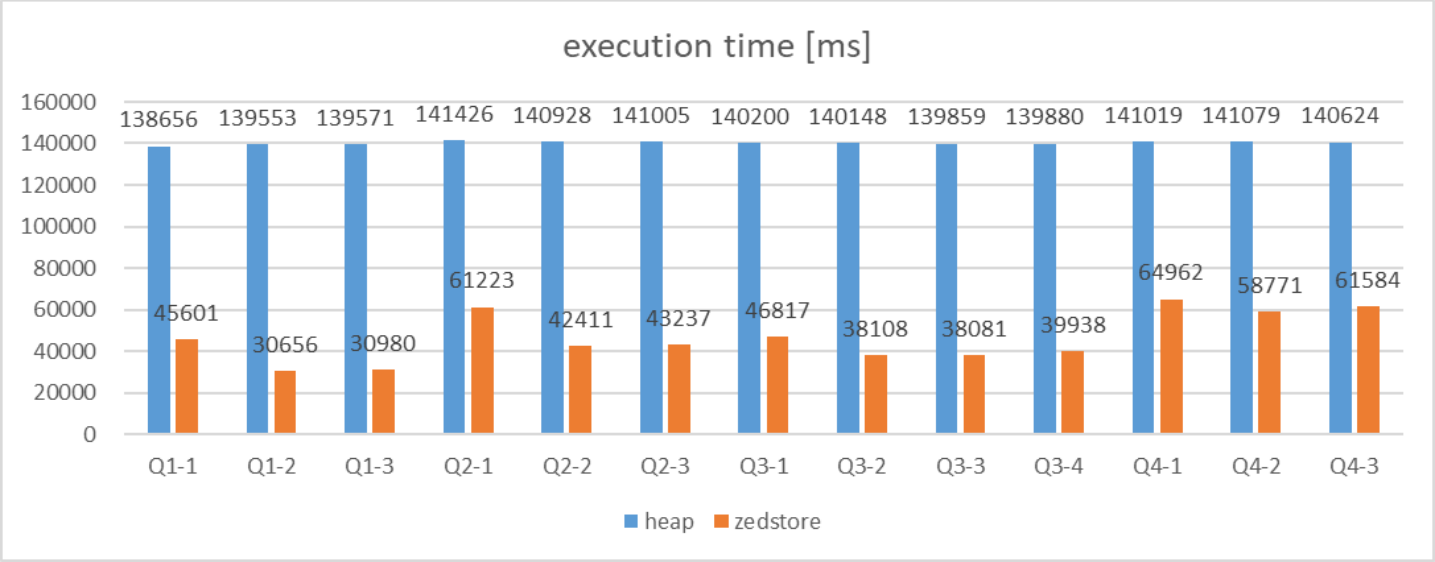


図 5.19 Execution time on Star Schema Benchmark

5.9.9. AM特性の確認

5.9.9.1. テーブルサイズ比較

各Access Methodでpgbench_accounts（pgbench）、lineorder（StarSchemaBenchmark）のテーブルを作成し、初期データ登録時のテーブルサイズの比較を行いました。

検証の結果、テーブルサイズについては、heapとの比較で、lineorderは、heapと比較してzedstoreで40%程度のサイズ、pgbench_accountsはheapと比較してzedstoreで4%程度のサイズとなった。pgbench_accountsの場合、連続した長大空白データがあるため圧縮率が非常に高かったためと推測しています。

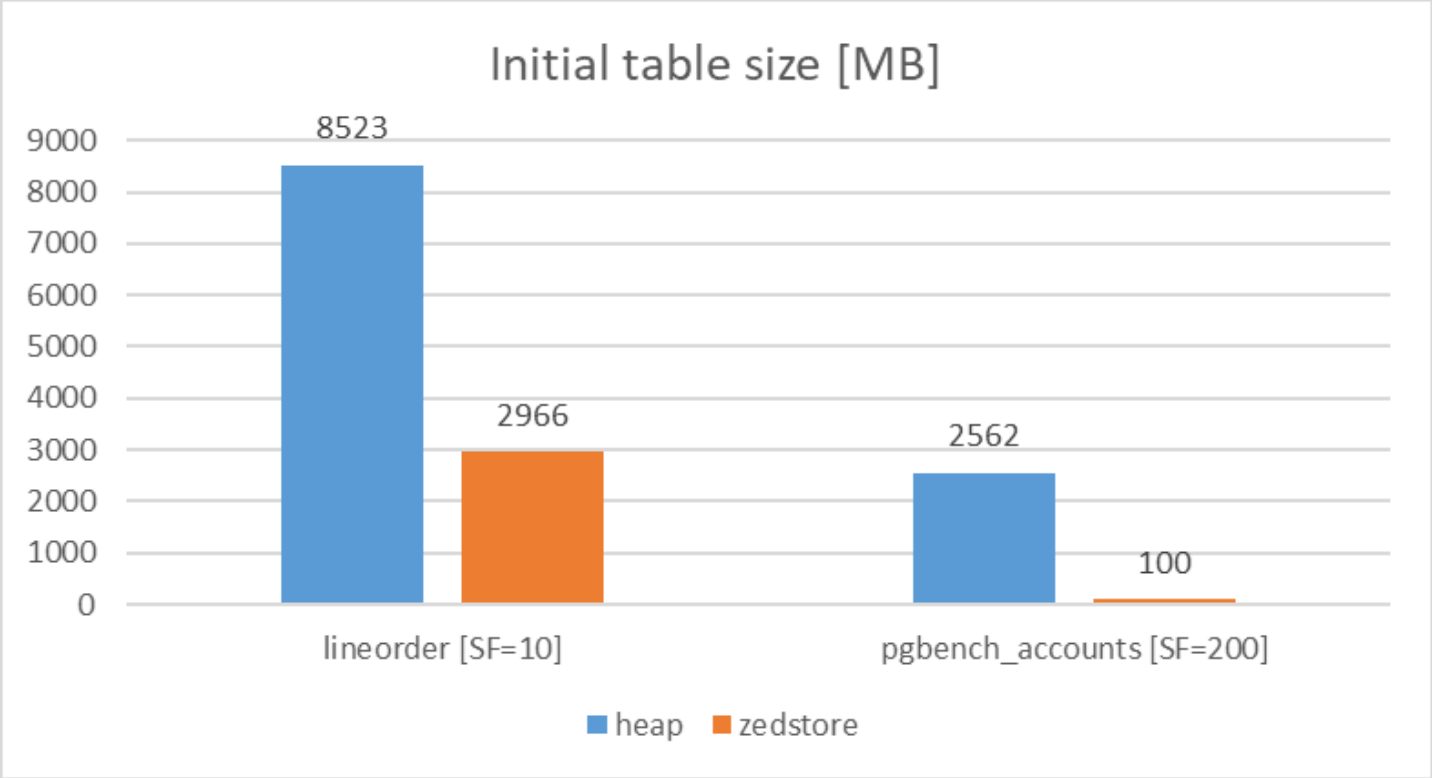


図 5.20 Table size comparison

5.9.10. 設計、運用に関する確認

DB設計、運用に関する確認として、DBのメンテナンス処理やコア機能として提供されているものとzedstoreが併用可能か確認を行いました。 便宜上、fooテーブルはheapで作成し、zfooテーブルはzedstoreで作成したものとして実行結果を記載しています。

結果のサマリは以下のとおりです。

項目	zedstoreでの利用可否	備考
メンテナンス (VACUUM/ANALYZE/CLUSTER/REINDEX)	○（利用可能）	zedstoreにおいては、ANALYZE/CLUSTERコマンドの処理時間が増加することを確認した。
バックアップ/リカバリ	○	heapとの比較において、特に違いは見られなかった。
ストリーミングレプリケーション	○	同上。
ロジカルレプリケーション	○	同上。
宣言的パーティション	○	同上。
パラレルクエリ	○	同上。
FILLFACTOR	○	同上。
追加モジュール(pgstattuple)	×（利用不可）	heap以外には対応していない。

5.9.10.1. メンテナンス（VACUUM/ANALYZE/CLUSTER/REINDEX）

- VACUUM

参照性能の検証で使用したインデックスのないzedstoreのテーブルに対して、VACUUMを実行し、その結果を確認しました。

検証の結果、問題なく不要領域の回収が行えることを確認しました。

更新後の稼働統計情報とVACUUM実行時の結果は以下のとおりです。

```
postgres=# select * from pg_stat_user_tables where relname = 'foo';
-[ RECORD 1 ]-------+------------------------------
relid               | 16837
schemaname          | public
relname             | foo
```

```
seq_scan            | 2
seq_tup_read        | 1000000
idx_scan            | 6000
idx_tup_fetch       | 6000
n_tup_ins           | 1000000
n_tup_upd           | 6000
n_tup_del           | 0
n_tup_hot_upd       | 953
n_live_tup          | 1000000
n_dead_tup          | 5895
n_mod_since_analyze | 6000
last_vacuum         |
last_autovacuum     |
last_analyze        |
last_autoanalyze    | 2020-03-27 10:10:41.755361+09
vacuum_count        | 0
autovacuum_count    | 0
analyze_count       | 0
autoanalyze_count   | 1
Time: 14.534 ms

postgres=# vacuum verbose foo ;
INFO:  vacuuming "public.foo"
INFO:  scanned index "foo_id_idx" to remove 5047 row versions
DETAIL:  CPU: user: 0.25 s, system: 0.00 s, elapsed: 0.26 s
INFO:  "foo": removed 5047 row versions in 5047 pages
DETAIL:  CPU: user: 0.03 s, system: 0.00 s, elapsed: 0.03 s
INFO:  index "foo_id_idx" now contains 1000000 row versions in 2745 pages
DETAIL:  5047 index row versions were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
INFO:  "foo": found 5032 removable, 1000000 nonremovable row versions in 16477 out of 16477 pages
DETAIL:  0 dead row versions cannot be removed yet, oldest xmin: 102451
There were 1 unused item identifiers.
Skipped 0 pages due to buffer pins, 0 frozen pages.
0 pages are entirely empty.
CPU: user: 0.46 s, system: 0.03 s, elapsed: 0.49 s.
INFO:  vacuuming "pg_toast.pg_toast_16837"
INFO:  index "pg_toast_16837_index" now contains 0 row versions in 1 pages
DETAIL:  0 index row versions were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
INFO:  "pg_toast_16837": found 0 removable, 0 nonremovable row versions in 0 out of 0 pages
DETAIL:  0 dead row versions cannot be removed yet, oldest xmin: 102451
There were 0 unused item identifiers.
Skipped 0 pages due to buffer pins, 0 frozen pages.
0 pages are entirely empty.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
VACUUM
Time: 535.067 ms

postgres=# select * from pg_stat_user_tables where relname = 'zfoo';
-[ RECORD 1 ]-------+------------------------------
relid               | 16850
schemaname          | public
relname             | zfoo
seq_scan            | 3
seq_tup_read        | 1000000
idx_scan            | 6000
idx_tup_fetch       | 1006000
n_tup_ins           | 1000000
n_tup_upd           | 6000
n_tup_del           | 0
n_tup_hot_upd       | 0
n_live_tup          | 1000000
n_dead_tup          | 0
n_mod_since_analyze | 0
```

```
last_vacuum         |
last_autovacuum     |
last_analyze        |
last_autoanalyze    | 2020-03-27 10:12:41.002016+09
vacuum_count        | 0
autovacuum_count    | 0
analyze_count       | 0
autoanalyze_count   | 1
Time: 29.988 ms

postgres=# vacuum verbose zfoo ;
INFO:  vacuuming "public.zfoo"
INFO:  scanned index "zfoo_id_idx" to remove 6000 row versions
DETAIL:  CPU: user: 0.18 s, system: 0.00 s, elapsed: 0.18 s
INFO:  "zfoo": removed 6000 row versions
INFO:  index "zfoo_id_idx" now contains 0 row versions in 2745 pages
DETAIL:  6000 index row versions were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
VACUUM
Time: 1426.891 ms (00:01.427)
```

- ANALYZE

前述の参照性能、更新性能の検証で使用したzedstoreのテーブルに対して、ANALYZEを実行し、その結果を確認しました。

検証の結果、heapと比較して、処理時間以外に特に違いがないことを確認できました。

検証時の実行結果は以下のとおりです。

```
postgres=# ANALYZE VERBOSE foo;
INFO:  analyzing "public.foo"
INFO:  "foo": scanned 30000 of 41667 pages, containing 359996 live rows and 0 dead rows; 30000 rows in sample, 499998 estimated total rows
ANALYZE
Time: 977.611 ms

postgres=# SELECT * FROM pg_stats WHERE tablename = 'foo' AND attname = 'id';
-[ RECORD 1 ]----------+-----------------------------------------------------------------------------------------------------------------------------------------------
schemaname             | public
tablename              | foo
attname                | id
inherited              | f
null_frac              | 0
avg_width              | 4
n_distinct             | -1
most_common_vals       |
most_common_freqs      |
histogram_bounds       |
{8,8811,19199,28856,39832,49215,58953,70045,79862,89647,99675,109350,118005,128200,137659,147915,157283,167913,177659,188310,
correlation            | 1
most_common_elems      |
most_common_elem_freqs |
elem_count_histogram   |

Time: 14.046 ms


postgres=# ANALYZE VERBOSE zfoo;
INFO:  analyzing "public.zfoo"
INFO:  "zfoo": scanned 3907 of 3907 pages, containing 500000 live rows and 0 dead rows; 30000 rows in sample, 500000 estimated total rows
ANALYZE
Time: 2259.623 ms (00:02.260)

postgres=# SELECT * FROM pg_stats WHERE tablename = 'zfoo' AND attname = 'id';
-[ RECORD 1 ]----------+-----------------------------------------------------------------------------------------------------------------------------------------------
schemaname             | public
tablename              | zfoo
attname                | id
inherited              | f
null_frac              | 0
avg_width              | 4
n_distinct             | -1
most_common_vals       |
most_common_freqs      |
histogram_bounds       |
{10,10151,20422,32249,41569,50540,60285,70652,80788,91039,101666,111648,122191,131666,140720,152042,162194,171540,181507,1924
correlation            | 1
most_common_elems      |
most_common_elem_freqs |
elem_count_histogram   |

Time: 4.266 ms
```

- CLUSTER

前述の参照性能、更新性能の検証で使用したzheapのテーブルに対して、CLUSTERを実行し、その結果を確認しました。

検証の結果、heapと比較して、処理時間以外に特に違いがないことを確認できました。

検証時の実行結果は以下のとおりです。

```
postgres=# CLUSTER VERBOSE foo USING foo_id_idx ;
INFO:  clustering "public.foo" using index scan on "foo_id_idx"
INFO:  "foo": found 0 removable, 500000 nonremovable row versions in 41667 pages
DETAIL:  0 dead row versions cannot be removed yet.
CPU: user: 1.49 s, system: 0.94 s, elapsed: 3.02 s.
CLUSTER
Time: 3651.239 ms (00:03.651)

postgres=# CLUSTER VERBOSE zfoo USING zfoo_id_idx ;
INFO:  clustering "public.zfoo" using index scan on "zfoo_id_idx"
INFO:  "zfoo": found 0 removable, 0 nonremovable row versions in 39263 pages
DETAIL:  0 dead row versions cannot be removed yet.
CPU: user: 6.95 s, system: 0.97 s, elapsed: 9.27 s.
CLUSTER
Time: 10542.133 ms (00:10.542)
```

- REINDEX

各Access Methodを使用した対象テーブルで定義されたB-tree インデックスを対象にREINDEXを実行し、確認を行いました。

検証の結果、heapと比較して、処理時間以外には特に違いがないことを確認できました。

検証時の実行結果は以下のとおりです。

```
postgres=# REINDEX ( VERBOSE ) INDEX foo_id_idx;
INFO:  index "foo_id_idx" was reindexed
DETAIL:  CPU: user: 0.35 s, system: 0.08 s, elapsed: 0.44 s
REINDEX
Time: 456.337 ms

postgres=# REINDEX ( VERBOSE ) INDEX zfoo_id_idx;
INFO:  index "zfoo_id_idx" was reindexed
DETAIL:  CPU: user: 1.03 s, system: 0.02 s, elapsed: 1.06 s
REINDEX
Time: 1089.927 ms (00:01.090)
```

5.9.10.2. バックアップ/リカバリ

zedstoreを使用したテーブルに対して、バックアップの取得及び、PITRが実施可能か確認しました。

確認の結果、zedstoreを使用したテーブルであっても問題なく、バックアップ及びリカバリを実行できることを確認できました。

検証時の実行結果は以下のとおりです。

```
# 初期データ登録
psql postgres -c "CREATE TABLE bar (id int);INSERT INTO bar VALUES (1);"

# basebackup取得
pg_basebackup -h localhost -D /tmp/rep

# 初期データ登録
psql postgres -c "CREATE TABLE bar2 (id int) USING zedstore;\
                  INSERT INTO bar2 VALUES (1);\
                  SELECT pg_switch_wal();\
                  INSERT INTO bar2 VALUES (2);\
                  SELECT statement_timestamp();"
psql postgres -c "SELECT pg_sleep(30);\
                  INSERT INTO bar2 VALUES (3),(4);\
                  SELECT pg_switch_wal();"

# リカバリ用の設定を追記
echo "port = 5433" >> /tmp/rep/postgresql.conf
echo "recovery_target_time = '2020-03-10 09:02:09'" >> /tmp/rep/postgresql.conf
echo "recovery_target_action = 'promote'" >> /tmp/rep/postgresql.conf
echo "archive_mode = off" >> /tmp/rep/postgresql.conf
echo "restore_command = 'cp /var/lib/pgsql/local/zedstore/arch/%f %p'" >> /tmp/rep/postgresql.conf

# PITR実行及び確認
pg_ctl -D /tmp/rep start
psql postgres -p 5433 -c "SELECT * FROM bar2;"
 id
----
  1
  2
(2 rows)
```

5.9.10.3. ストリーミングレプリケーション

zedstoreを使用したテーブルに対して、ストリーミングレプリケーション（以降、SRとする）が実施可能か確認しました。 本検証は、1台の環境で実施しているため、セカンダリはport番号を変更し起動しました。

確認の結果、zedstoreを使用したテーブルであっても問題なく、SR構成を構築可能であることを確認しました。

検証時の実行結果は以下のとおりです。

```
# セカンダリの作成(standby.signalを自動作成)
pg_basebackup -h localhost -D /tmp/rep -R

# SR用の設定を追記
echo "port = 5433" >> /tmp/rep/postgresql.conf
echo "archive_mode = off" >> /tmp/rep/postgresql.conf
echo "primary_conninfo = 'user=postgres host=localhost port=5432'" >> /tmp/rep/postgresql.conf

# セカンダリ起動
pg_ctl -D /tmp/rep start

# レプリケーション状態の確認
psql postgres -c "SELECT client_addr,state,sync_state FROM pg_stat_replication"

# zedstoreAMを使用したテーブルを作成
psql postgres -c "CREATE TABLE bar (id int) USING zedstore;\
                  INSERT INTO bar VALUES (1);"

# セカンダリのテーブルを参照する
psql postgres -p 5433 -c "SELECT * FROM bar"
 id
----
  1
(1 row)
```

5.9.10.4. ロジカルレプリケーション

zedstoreを使用したテーブルに対して、ロジカルレプリケーションが実施可能か確認しました。 本検証は、1台の環境で実施しているため、セカンダリはport番号を変更し起動しました。

確認の結果、zedstoreを使用したテーブルであっても問題なく、ロジカルレプリケーションが可能であることを確認しました。

検証時の実行結果は以下のとおりです。

```
# ベースバックアップを作成する
pg_basebackup -h localhost -D /tmp/rep
echo "port = 5433" >> /tmp/rep/postgresql.conf
echo "archive_mode = off" >> /tmp/rep/postgresql.conf

# セカンダリを起動する
pg_ctl -D /tmp/rep start

# プライマリでパブリケーションを作成する
psql postgres -c "CREATE TABLE japan (id int, pref text, city text, data text) USING zedstore;\
                  CREATE PUBLICATION pub_srv1_testdb_japan FOR TABLE japan;"

# セカンダリでサブスクライバを作成する
psql postgres -p 5433 -c "CREATE TABLE japan (id int, pref text, city text, data text) USING zedstore;"
psql postgres -p 5433 -c "CREATE SUBSCRIPTION sub_srv2_testdb_japan CONNECTION 'dbname=postgres port=5432 user=postgres'
PUBLICATION pub_srv1_testdb_japan;"

# プライマリに確認用のデータをINSERTする
psql postgres -c "INSERT INTO japan VALUES (1, '東京', '港', 'らーめん 象');\
                  INSERT INTO japan VALUES (2, '神奈川', '横浜', '象Love亭');"

# セカンダリにデータがINSERTされていることを確認する
psql postgres -p 5433 -c "SELECT * FROM japan"
 id |  pref  | city |    data
----+--------+------+-------------
  1 | 東京   | 港   | らーめん 象
  2 | 神奈川 | 横浜 | 象Love亭
(2 rows)
```

5.9.10.5. 宣言的パーティション

宣言的パーティションを構築する際、対象テーブルでzedstoreを利用可能か確認しました。 確認の結果、zedstoreを使用したテーブルであっても問題なく、宣言的パーティションの構成を作成できることを確認しました。 親テーブルへのAM指定はPostgreSQLとしてサポートされていないので実施できませんが、子テーブルへの指定は可能でした。また、プルーニングについても有効であることを確認しました。

検証時の実行結果は以下のとおりです。

```
postgres=# CREATE TABLE japan ( pref text, city text, data text ) PARTITION BY LIST (pref) USING zedstore;
psql: ERROR:  specifying a table access method is not supported on a partitioned table

postgres=# CREATE TABLE japan ( pref text, city text, data text ) PARTITION BY LIST (pref);
CREATE TABLE
postgres=# CREATE TABLE tokyo PARTITION OF japan FOR VALUES IN ('東京') USING zedstore;
CREATE TABLE
postgres=# CREATE TABLE kanagawa PARTITION OF japan FOR VALUES IN ('神奈川') USING zedstore;
CREATE TABLE
postgres=# CREATE TABLE shizuoka PARTITION OF japan FOR VALUES IN ('静岡') USING zedstore;
CREATE TABLE
postgres=# \d+ japan
                        Partitioned table "public.japan"
 Column | Type | Collation | Nullable | Default | Storage  | Stats target | Description
--------+------+-----------+----------+---------+----------+--------------+-------------
 pref   | text |           |          |         | extended |              |
 city   | text |           |          |         | extended |              |
 data   | text |           |          |         | extended |              |
Partition key: LIST (pref)
Partitions: kanagawa FOR VALUES IN ('神奈川')

postgres=# \d+ kanagawa
                           Table "public.kanagawa"
 Column | Type | Collation | Nullable | Default | Storage  | Stats target | Description
--------+------+-----------+----------+---------+----------+--------------+-------------
 pref   | text |           |          |         | extended |              |
 city   | text |           |          |         | extended |              |
 data   | text |           |          |         | extended |              |
Partition of: japan FOR VALUES IN ('神奈川')
Partition constraint: ((pref IS NOT NULL) AND (pref = '神奈川'::text))
Access method: zedstore

postgres=# INSERT INTO japan VALUES ('東京','町田','リス'),('神奈川','横浜','シウマイ'),('神奈川','川崎','タンタンメン');
INSERT 0 3
postgres=# EXPLAIN SELECT * FROM japan;
                              QUERY PLAN
-----------------------------------------------------------------------
 Append  (cost=0.00..59.25 rows=1950 width=96)
   ->  Seq Scan on tokyo japan  (cost=0.00..16.50 rows=650 width=96)
   ->  Seq Scan on kanagawa japan_1  (cost=0.00..16.50 rows=650 width=96)
   ->  Seq Scan on shizuoka japan_2  (cost=0.00..16.50 rows=650 width=96)
(4 rows)

postgres=# EXPLAIN SELECT * FROM japan WHERE pref = '神奈川';
                          QUERY PLAN
---------------------------------------------------------------
 Seq Scan on kanagawa japan  (cost=0.00..18.12 rows=3 width=96)
   Filter: (pref = '神奈川'::text)
(2 rows)
```

5.9.10.6. パラレルクエリ

zedstoreを使用したテーブルに対してパラレルクエリが利用可能か確認をしました。 確認の結果、zedstoreを使用したテーブルであってもパラレルクエリは問題なく動作することを確認できました。

検証時の実行結果は以下のとおりです。

```
-bash-4.2$ psql postgres -c "COPY zlineorder FROM '/benchdata/lineorder.tbl' DELIMITER '|'"
-bash-4.2$ psql postgres
psql (13devel)
postgres=# set max_parallel_workers_per_gather TO 2;
SET
postgres=# explain analyze select count(lo_discount) from zlineorder ;
                                                        QUERY PLAN

-----------------------------------------------------------------------------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=408740.62..408740.63 rows=1 width=8) (actual time=19629.966..19629.966 rows=1 loops=1)
   ->  Gather  (cost=408740.41..408740.62 rows=2 width=8) (actual time=19526.914..19634.155 rows=3 loops=1)
         Workers Planned: 2
         Workers Launched: 2
         ->  Partial Aggregate  (cost=407740.41..407740.42 rows=1 width=8) (actual time=19501.879..19501.880 rows=1 loops=3)
               ->  Parallel Seq Scan on zlineorder  (cost=0.00..401831.12 rows=2363712 width=32) (actual time=23.529..14755.448
rows=19995351 loops=3)
 Planning Time: 0.466 ms
 Execution Time: 19634.322 ms
(8 rows)

-bash-4.2$ sudo sysctl -w vm.drop_caches=1
vm.drop_caches = 1
-bash-4.2$ pg_ctl -D ~/local/zedstore/data restart
-bash-4.2$ psql postgres
psql (13devel)  Type "help" for help.
postgres=# explain analyze select count(lo_discount) from zlineorder ;
                                              QUERY PLAN

-----------------------------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=763399.20..763399.21 rows=1 width=8) (actual time=39624.770..39624.770 rows=1 loops=1)
   ->  Seq Scan on zlineorder  (cost=0.00..613433.76 rows=59986176 width=4) (actual time=7.432..27274.861 rows=59986052
loops=1)
 Planning Time: 3.555 ms
 Execution Time: 39624.944 ms
(4 rows)
```

5.9.10.7. FILLFACTOR

zedstoreを使用したテーブルに対してfillfactorが利用可能か確認をしました。

確認の結果、zedstoreを使用したテーブルであってもfillfactorは定義できることが確認できました。

しかし、以下の確認では充填率の変化によるテーブルサイズの拡張は見られないことから、fillfactorを定義できるものの内容は無視されていると考えられます。

検証時の実行結果は以下のとおりです。

```
postgres=#   CREATE TABLE zfoo(id int, val1 text, val2 text, val3 text) USING zedstore WITH (fillfactor='100');
CREATE TABLE
postgres=#   INSERT INTO zfoo (SELECT generate_series(1,1000000),md5(random()::text),md5(random()::text),md5(random()::text));
INSERT 0 1000000
postgres=#   CREATE INDEX ON zfoo USING btree (id)
postgres-# \d+
                        List of relations
 Schema | Name | Type  |  Owner   | Persistence |  Size  | Description
--------+------+-------+----------+-------------+--------+-------------
 public | zfoo | table | postgres | permanent   | 102 MB |
(1 row)

postgres=# drop table zfoo;
DROP TABLE

postgres=#   CREATE TABLE zfoo(id int, val1 text, val2 text, val3 text) USING zedstore WITH (fillfactor='80');
CREATE TABLE
postgres=#   INSERT INTO zfoo (SELECT generate_series(1,1000000),md5(random()::text),md5(random()::text),md5(random()::text));

INSERT 0 1000000
postgres=#   CREATE INDEX ON zfoo USING btree (id)
postgres-# \d+
                        List of relations
 Schema | Name | Type  |  Owner   | Persistence |  Size  | Description
--------+------+-------+----------+-------------+--------+-------------
 public | zfoo | table | postgres | permanent   | 102 MB |
(1 row)
```

5.9.11. 追加モジュール

- pgstattuple

zedstoreを使用したテーブルに対してpgstattupleが利用可能か確認をしました。

確認の結果、pgstattupleはheap以外のテーブルには対応していないことが確認できました。

検証時の実行結果は以下のとおりです。

```
psql postgres -c "CREATE EXTENSION pgstattuple"
psql postgres -c "CREATE TABLE bar (id int) USING zedstore;"
psql postgres -c "SELECT * FROM pgstattuple('bar');"
ERROR:  only heap AM is supported
```

5.10. zedstoreの検証結果に対する考察

本検証では、PostgreSQL 12で実装されたPlugabble Storageの検証として、現在開発中のAccess Methodであるzedstoreの検証を行いました。

zheapの主目的であるカラムナフォーマットのサポートについてですが、SSBによる検証の結果、想定する適用範囲であるOLAP系において処理時間を半分以下に短縮できることを確認できました。

参照系については、zedstoreでは読み込みブロック数が性能に大きく関わることを確認できました。圧縮が効いていて読み込みブロック数が少ないカラムに対する処理では、処理時間を半分以下に短縮でできることを確認できました。しかし、圧縮率の悪いカラムに対する処理では読み込みブロック数が少なくても性能で半分程度になってしまうことも確認しました。 更新系については、UPDATE、COMMITにおいてheapが有利という結果になり、カラムナフォーマットの傾向が出ていることを確認しました。

テーブルサイズについては、lz4による圧縮効果によりSSBのlineorderテーブルでheapと比較してzedstoreでは70%のサイズ削減となっており、扱うデータ次第では大きな効果があることを確認できました。

DB設計や運用に関連する要素として、DBメンテナンス系のコマンドやPostgreSQLがサポートする機能についての動作確認を行い、特にzedstoreであることを意識することなく使用できることを確認できました。ただし、ANALYZE、CLUSTER等のIOが発生する処理ではカラムナフォーマットの影響を受け、heapの倍程度の処理時間が掛かることが確認できました。

6. パラレルクエリ・JITコンパイル 性能検証

6.1. 検証概要

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

PostgreSQLをOLAP用途で使用したいという要望は、以前から根強くありましたが、PostgreSQL 9.5まではパラレルクエリ等のOLAP向けの機能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。 また、同じく、クエリの実行速度を向上させる目的で、PostgreSQL 11で実装された実行時(Just-In-Time; JIT)コンパイルと併用することで、さらに性能が向上するかを検証します。 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。

6.1.1. 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. パラレルクエリとは

6.2.1. パラレルクエリ

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

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

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

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

6.2.2. PostgreSQLでのパラレルクエリ

2019年現在、PostgreSQL 11、および、PostgreSQL 12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。

6.2.3. 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 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	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。

6.3. JITコンパイルとは

6.3.1. 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。

6.3.2. 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 6.3 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 6.4 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

6.3.3. 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 6.5 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

6.4. 検証構成

6.4.1. 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負荷かけサーバからDBサーバに対してSQLを発行し、実行する形式としました。

表 6.6 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)

表 6.7 負荷かけサーバ構成

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	HDD 600GB SAS 10K x 24 (RAID 5)

6.4.2. 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 6.8 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 8.1
Database	PostgreSQL 11.7 PostgreSQL 12.2

6.5. 検証方法

6.5.1. 検証項目

パラレルクエリ・JITコンパイル検証では以下の3点を検証します。

6.5.1.1. 検証A

検証Aでは、パラレルクエリの並列度による、複数クライアントからクエリを実行した際の処理性能の状況を確認します。

具体的には、max_worker_processes、max_parallel_workers_per_gather、max_parallel_workersの3項目を、 単一のクライアントからの実行を想定して、テーブルに格納されているデータサイズから算出される並列度に合わせた場合と、 パラレルワーカー数として検証マシンに搭載されている物理コア数よりも大きい値を設定した場合で、処理時間を比較します。

本検証によって、並列度のチューニングによって、OLAP用途におけるクエリが複数クライアントから発行された際の性能が、どのように異なるのかを明らかにします。

6.5.1.2. 検証B

検証Bでは、検証Aにおけるそれぞれの設定値において、JITコンパイルを有効にした場合の処理性能の状況を確認します。

本検証によって、パラレルクエリとJITコンパイルの併用が有効であるのか、複数クライアントからOLAP用途のクエリが発行された際の性能に与える影響を明らかにします。

6.5.1.3. 対照群

対照群として、バージョン11、バージョン12において、max_parallel_workers_per_gather=0、および、max_parallel_workers=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。

6.5.2. 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。 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。

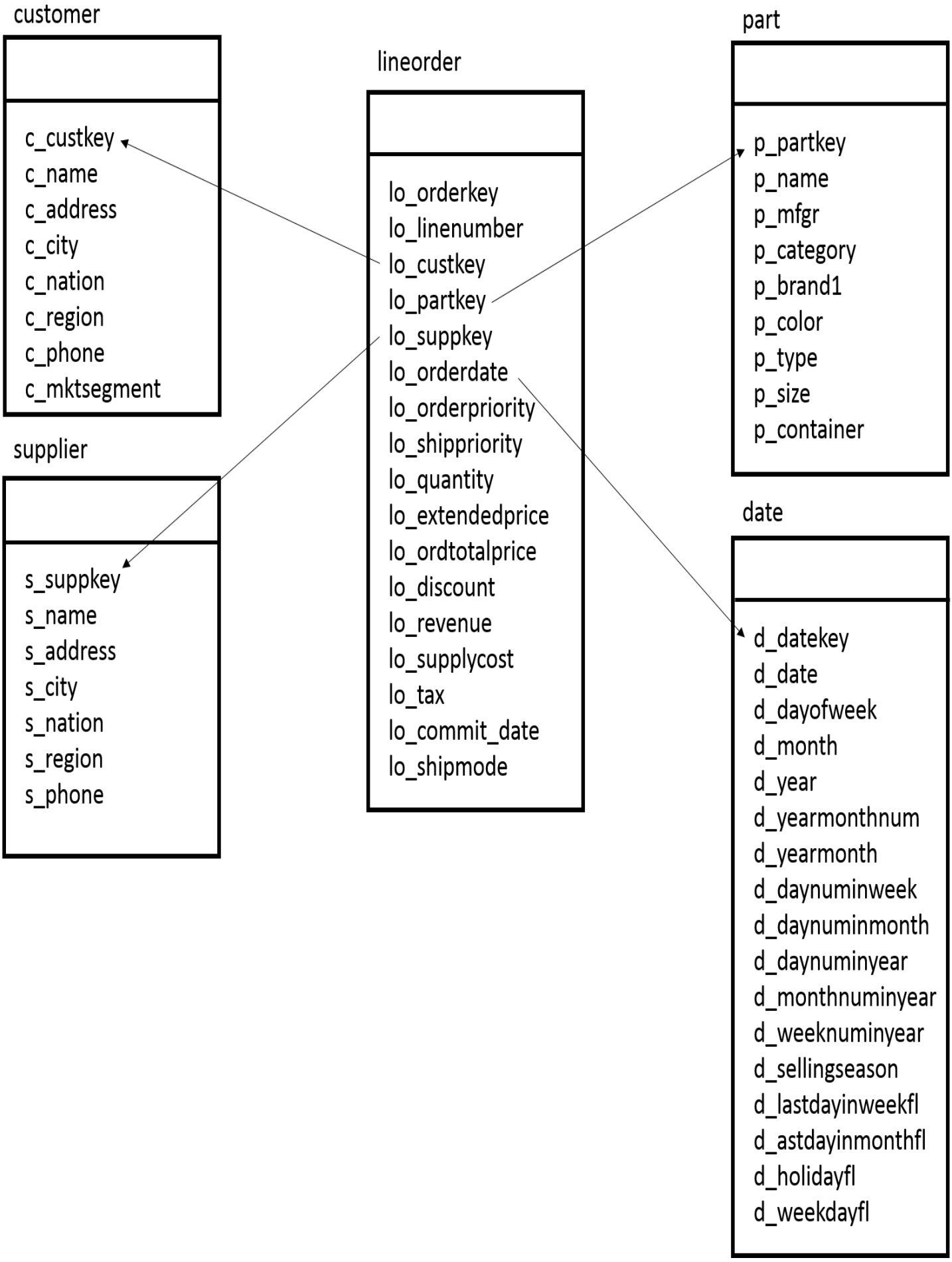


図 6.1 Star Schema Benchmarkのテーブル構成

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 6.9 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]
customer	SF × 30000	2999825	374030336
date	2556	2556	311296
lineorder	SF × 6000000	600044480	63557664768
part	200000 × (1+log2(SF)) (小数点切り捨て)	1400000	164265984
supplier	SF × 10000	1000000	114376704

6.5.3. 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;
```

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;
```

6.5.4. DB設定

性能測定にあたり、デフォルトから変更したパラメータは以下のとおりです。

表 6.10 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	100GB
work_mem	1GB
max_worker_processes [5]	13
max_parallel_workers_per_gather [5]	9
max_parallel_workers [5]	9
max_parallel_maintenance_workers [6]	0

[5] (*1*, *2*, *3*) 測定により値を変更して実施します。

[6] 影響する測定項目がないため、0としています。

6.5.5. 測定手順

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

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

6.5.5.1. 検証A測定方法

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回ずつ実施しています。

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、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 6.11 検証時設定(検証A)

パラメータ	workers = 9	workers = 100
max_parallel_workers_per_gether	9	100
max_worker_processes	13	104
max_parallel_workers	9	100
jit [7]	off	off

[7] 検証Aでは、JITコンパイル機能を使用しません。

6.5.5.2. 検証B測定方法

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回ずつ実施しています。

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、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 6.12 検証時設定(検証B)

パラメータ	workers = 9	workers = 100
max_parallel_workers_per_gether	9	100
max_worker_processes	13	104
max_parallel_workers	9	100
jit [8]	on	on

[8] 検証Bでは、すべての測定においてJITコンパイル機能を使用します。

6.6. 検証結果

6.6.1. 検証A(パラレルクエリを複数クライアントから発行した場合の応答時間)

表 6.13 検証条件

ストレージ	SQL文	保持データ量
内蔵ストレージ装置 (SSD)	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、全クエリでパラレルクエリが動作しました。

1クライアントでの実行における、v11とv12の処理時間のグラフを以下に示します。このグラフにおける数値は、5回ベンチマークを実行して採取した測定値の中央値を採用しています。

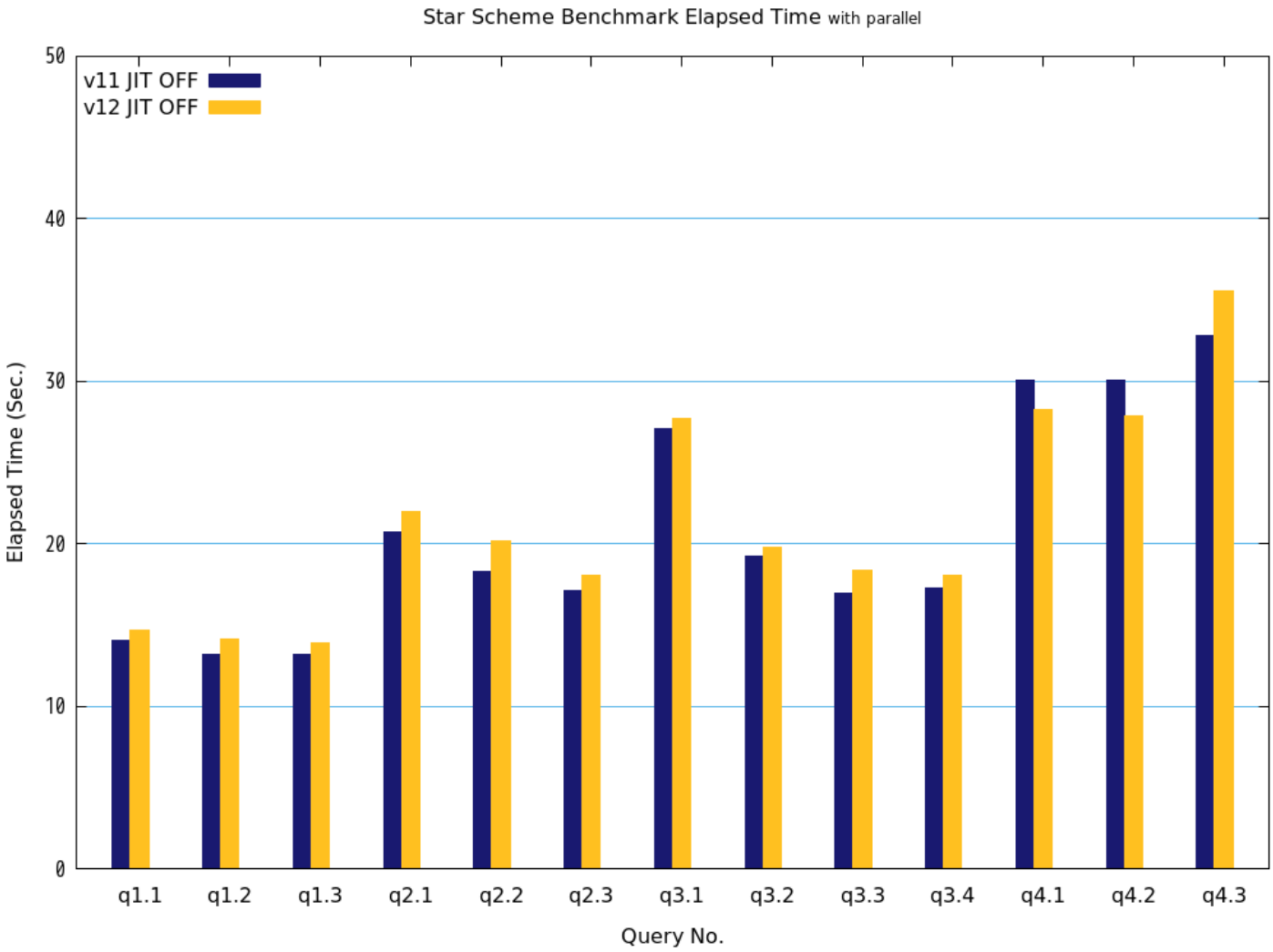


図 6.2 処理時間の比較 (1 Clients)

検証の結果、クエリの処理時間について、バージョン11との比較で、q4.1とq4.2ではそれぞれ1.06倍、1.08倍の高速化が認めらた一方で、それ以外では0.91倍(q2.1)〜0.98倍(q3.1)の結果となり、性能が劣化しているものが見られました。

次に、max_parallel_workers=9の場合の、クライアント数毎(2クライアント以上)、および、バージョン毎の処理時間のグラフを以下に示します。このグラフにおける数値は、5回ベンチマークを実行して採取した全クライアントの全測定値をプロットしています。

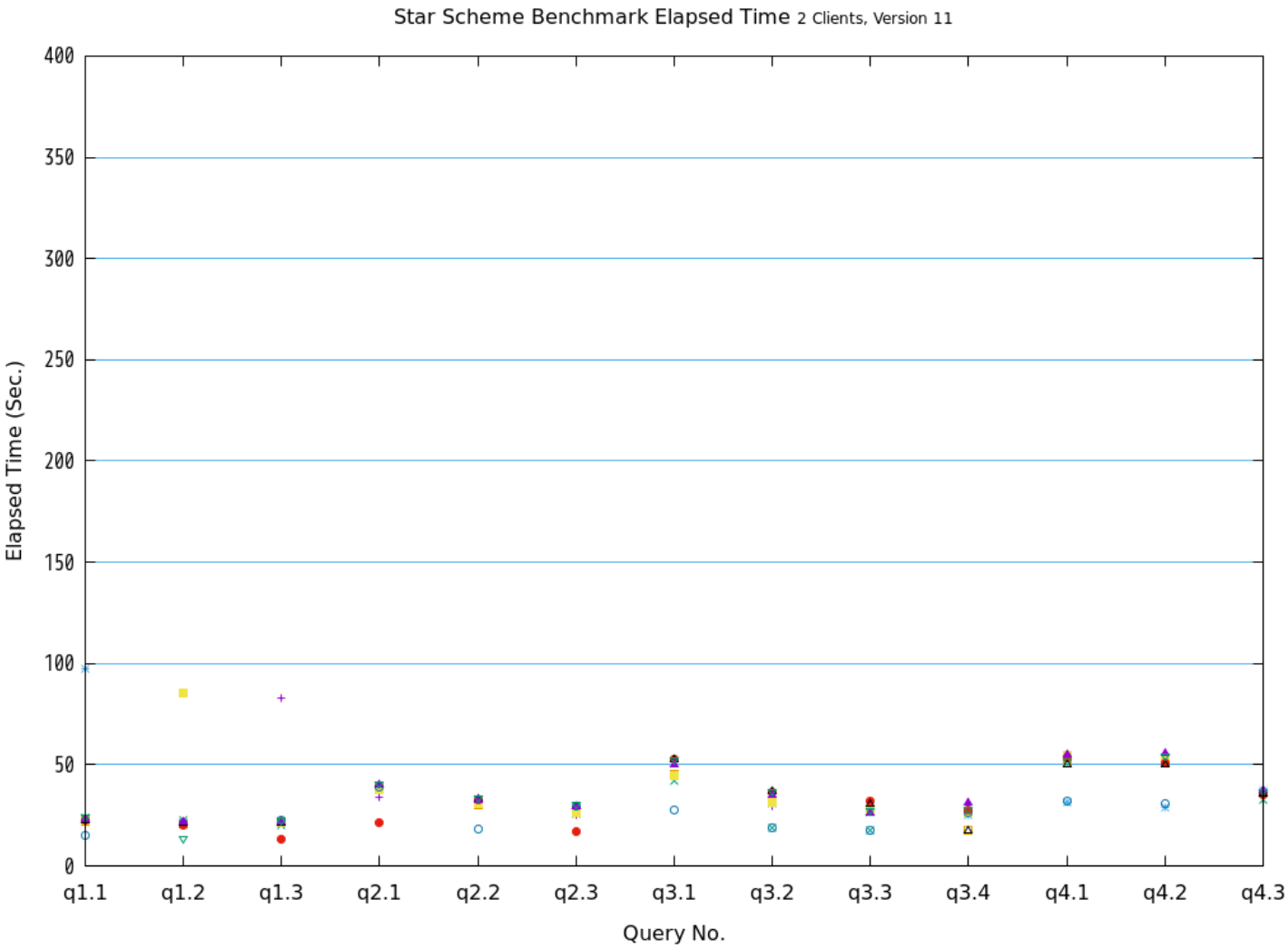


図 6.3 処理時間の比較 (v11, workers=9, 2 Clients)

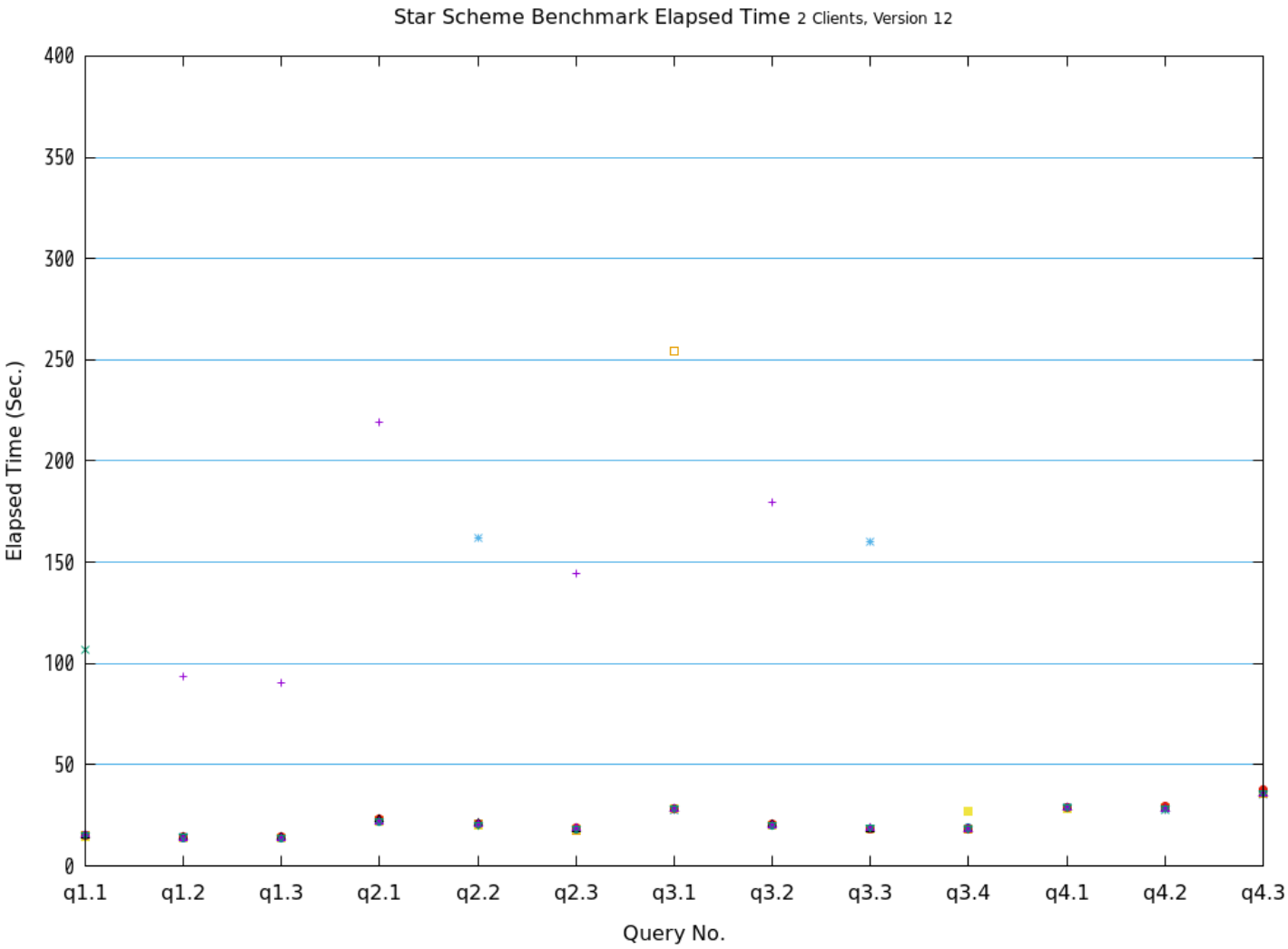


図 6.4 処理時間の比較 (v12, workers=9, 2 Clients)

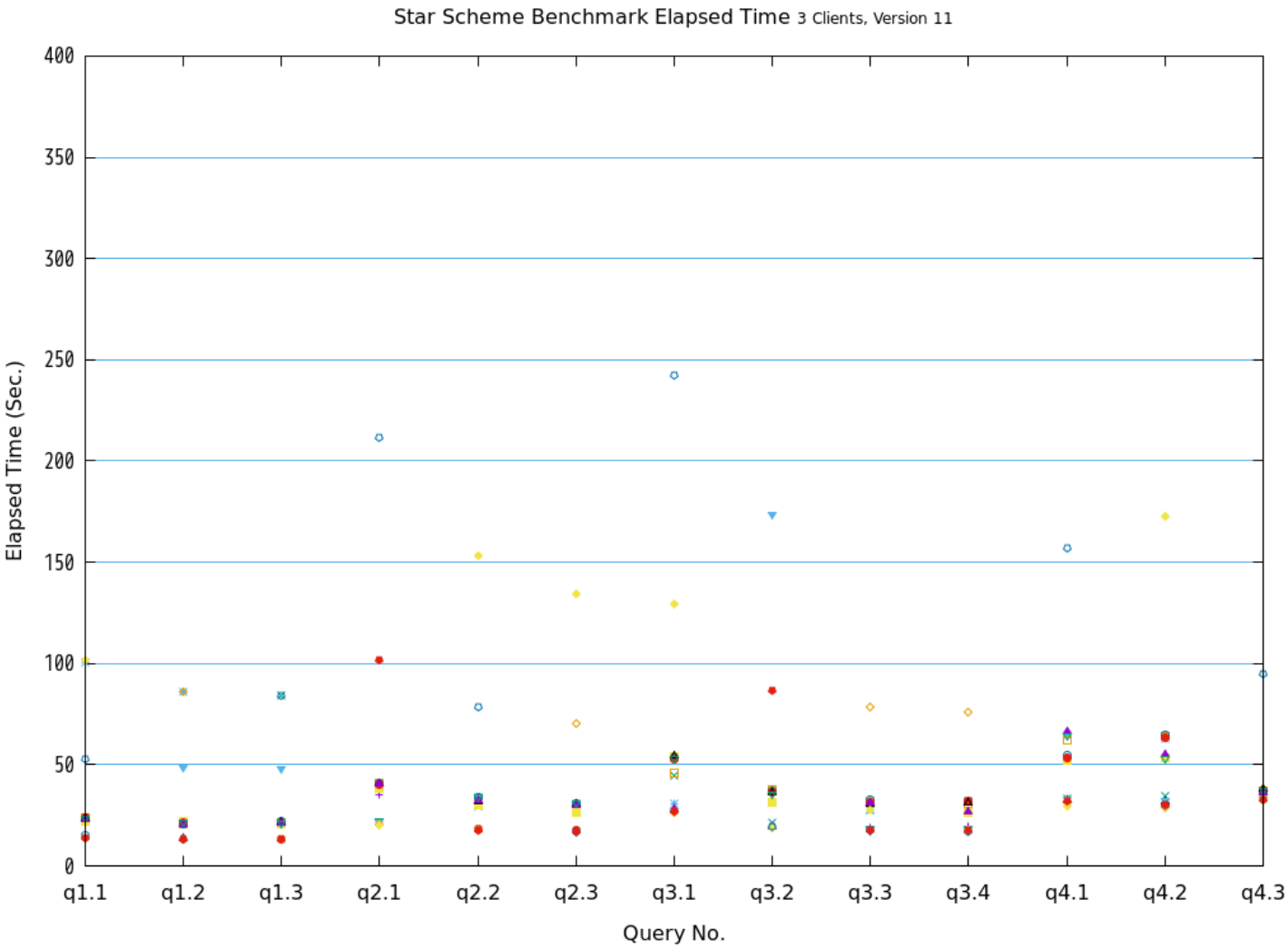


図 6.5 処理時間の比較 (v11, workers=9, 3 Clients)

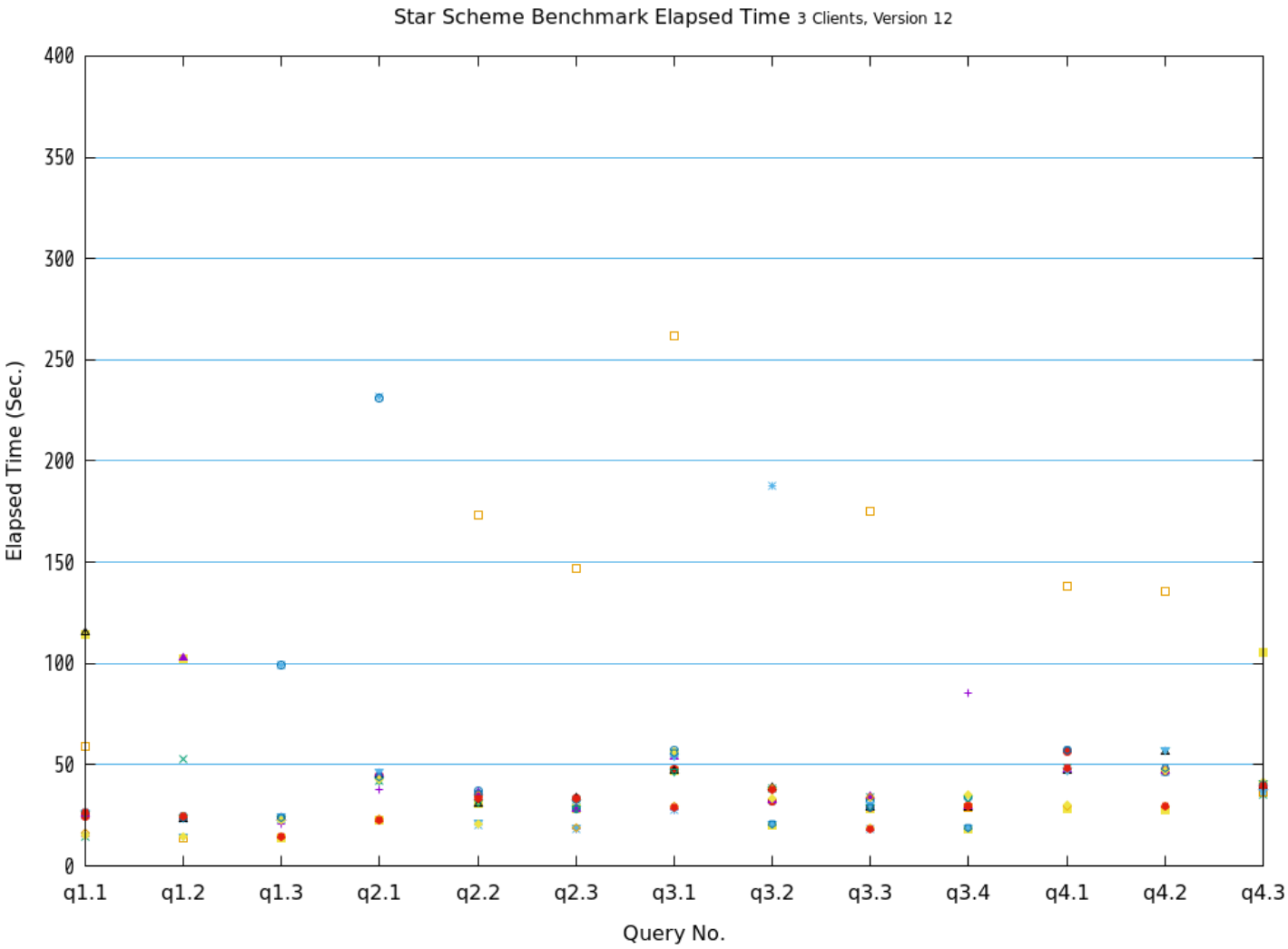


図 6.6 処理時間の比較 (v12, workers=9, 3 Clients)

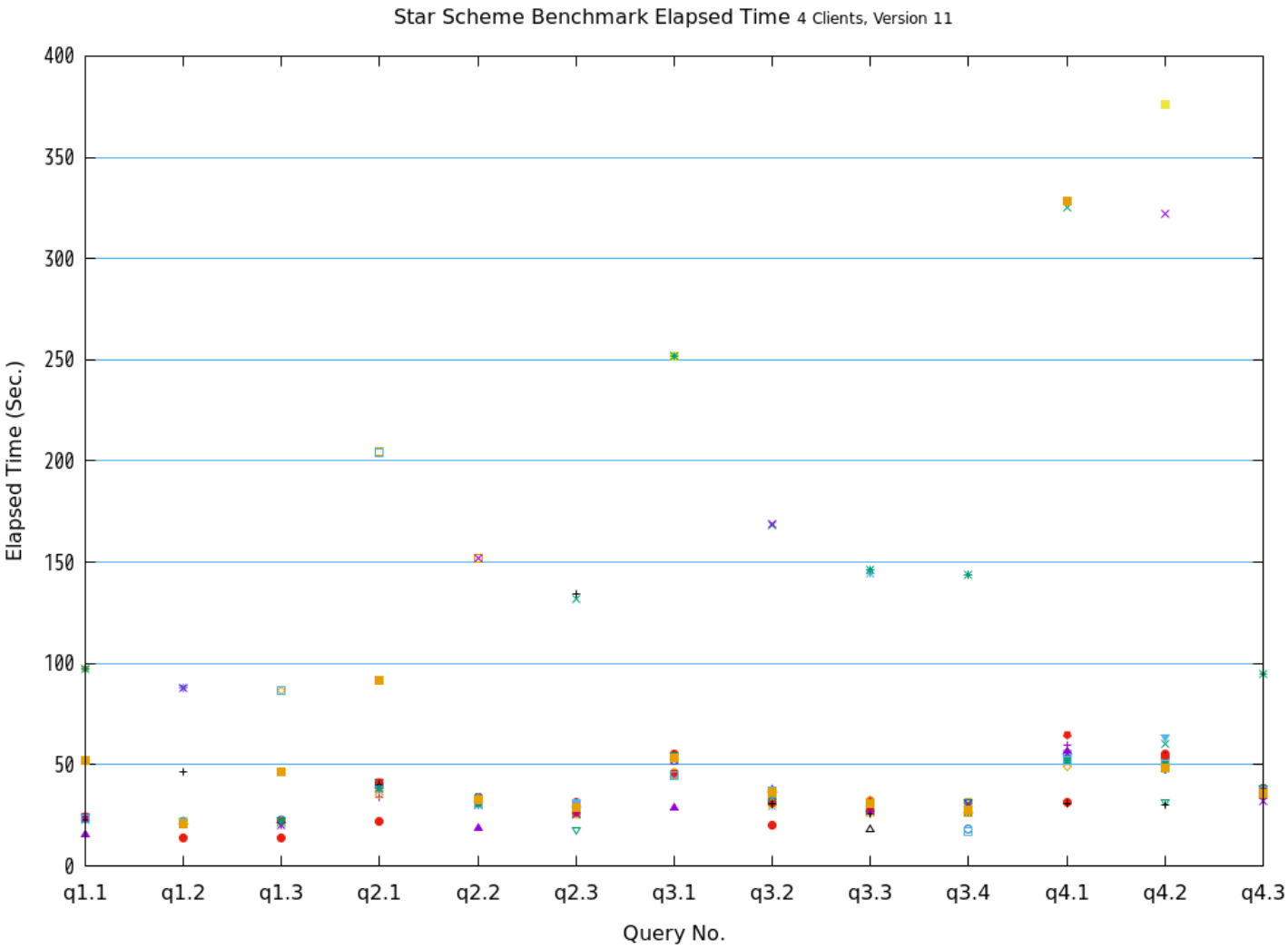


図 6.7 処理時間の比較 (v11, workers=9, 4 Clients)

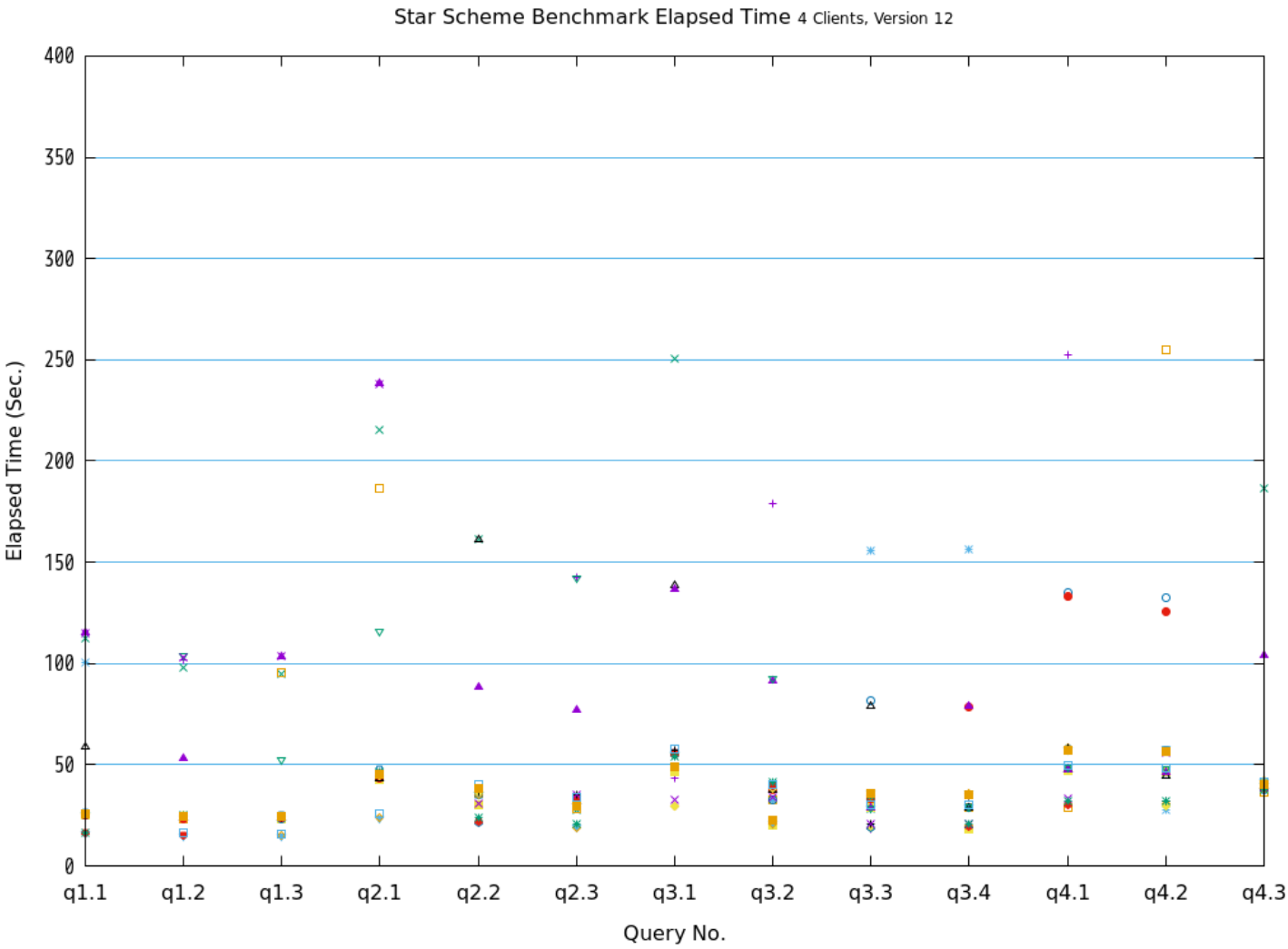


図 6.8 処理時間の比較 (v12, workers=9, 4 Clients)

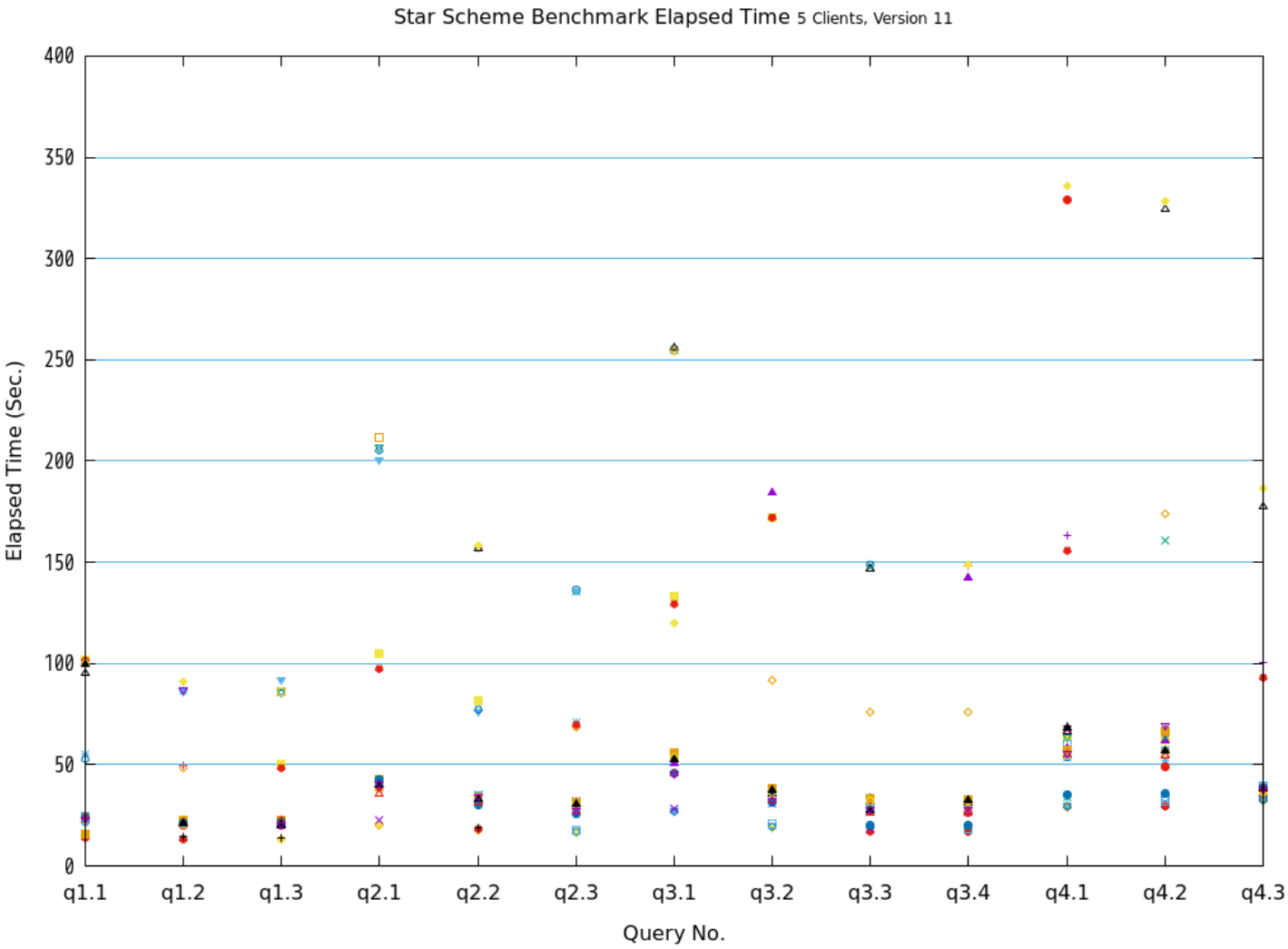


図 6.9 処理時間の比較 (v11, workers=9, 5 Clients)

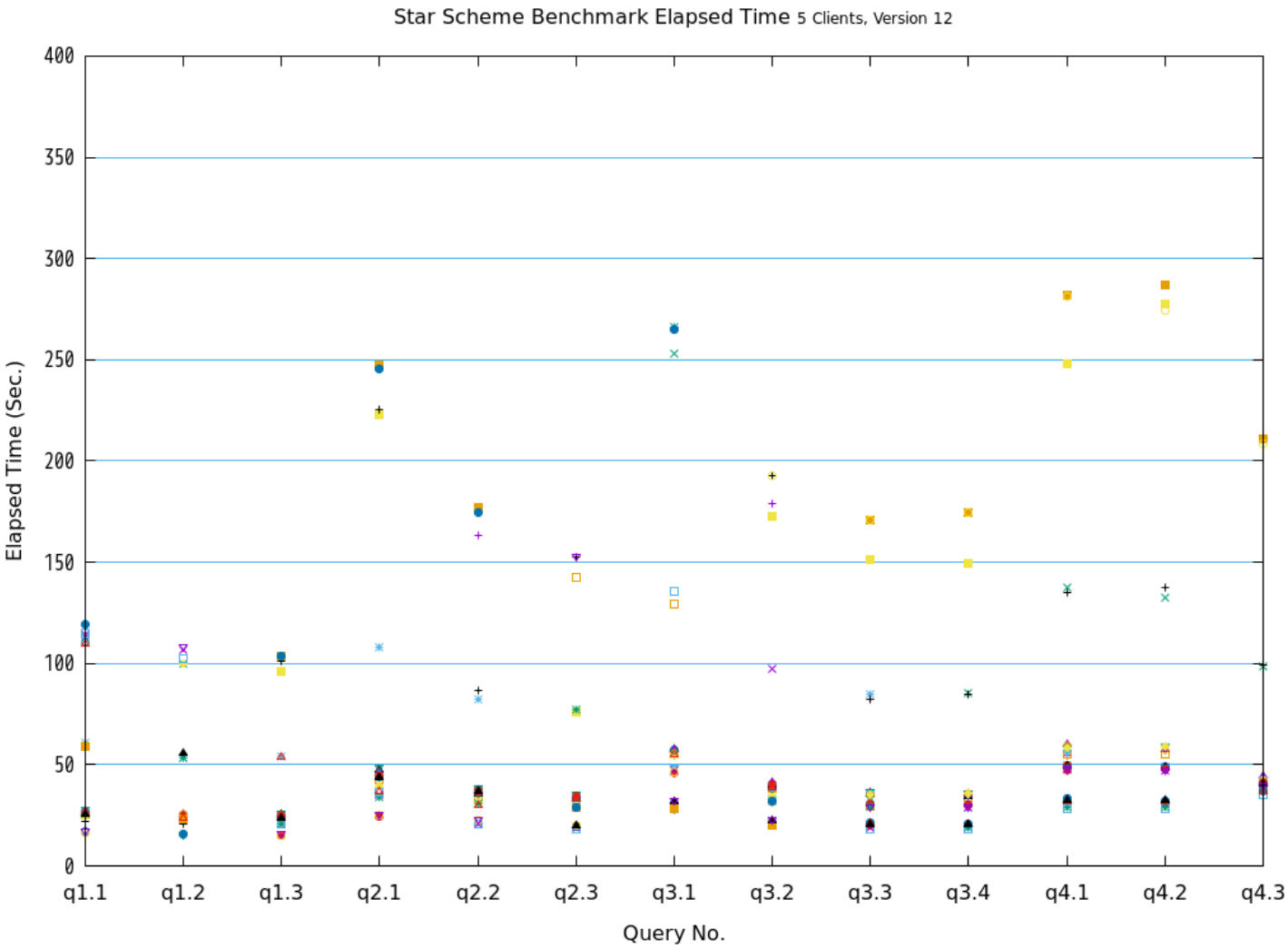


図 6.10 処理時間の比較 (v12, workers=9, 5 Clients)

検証の結果、クエリの応答時間について、多くの測定値が狭い範囲に集中しクラスターを形成している一方、一部の測定値はこのクラスターから上位の方向に大きく異なった値となっていることがわかります。 これは、パラレルワーカーの数を制限しているため、パラレルワーカーを確保できた場合とそうでない場合で、結果が分かれたことを示しています。

次に、max_parallel_workers=100の場合の、クライアント数毎(2クライアント以上)、および、バージョン毎の処理時間のグラフを以下に示します。

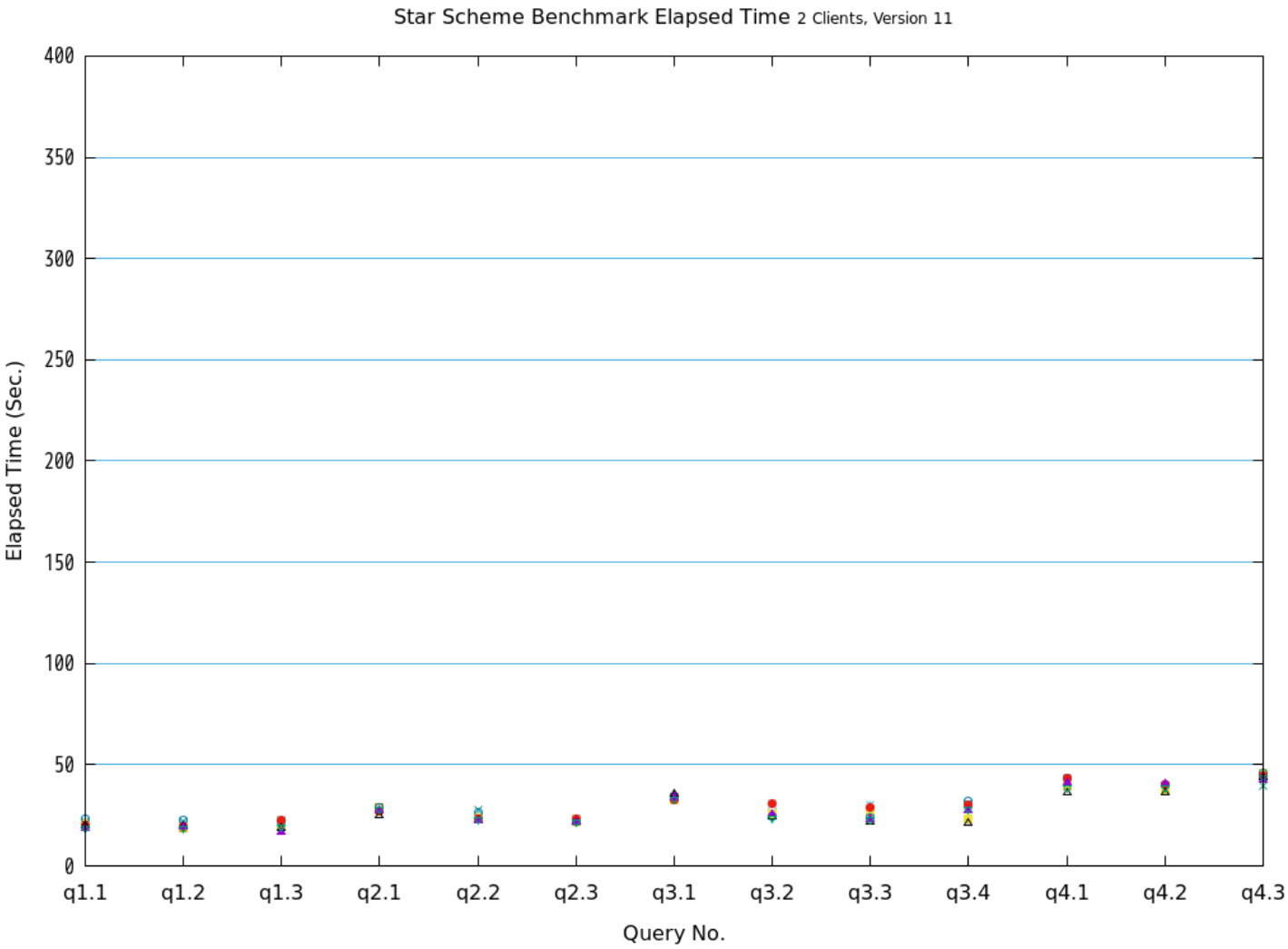


図 6.11 処理時間の比較 (v11, workers=100, 2 Clients)

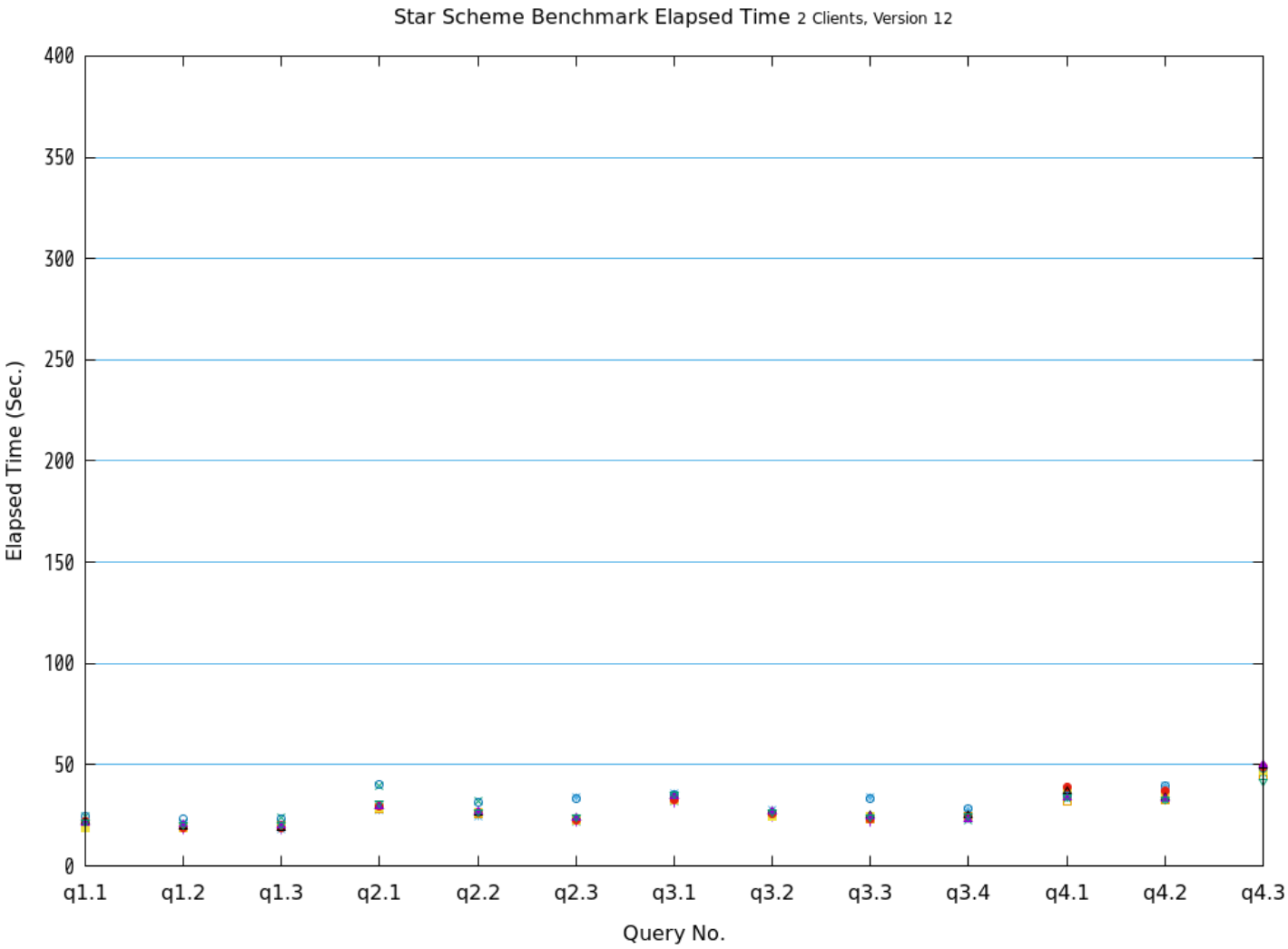


図 6.12 処理時間の比較 (v12, workers=100, 2 Clients)

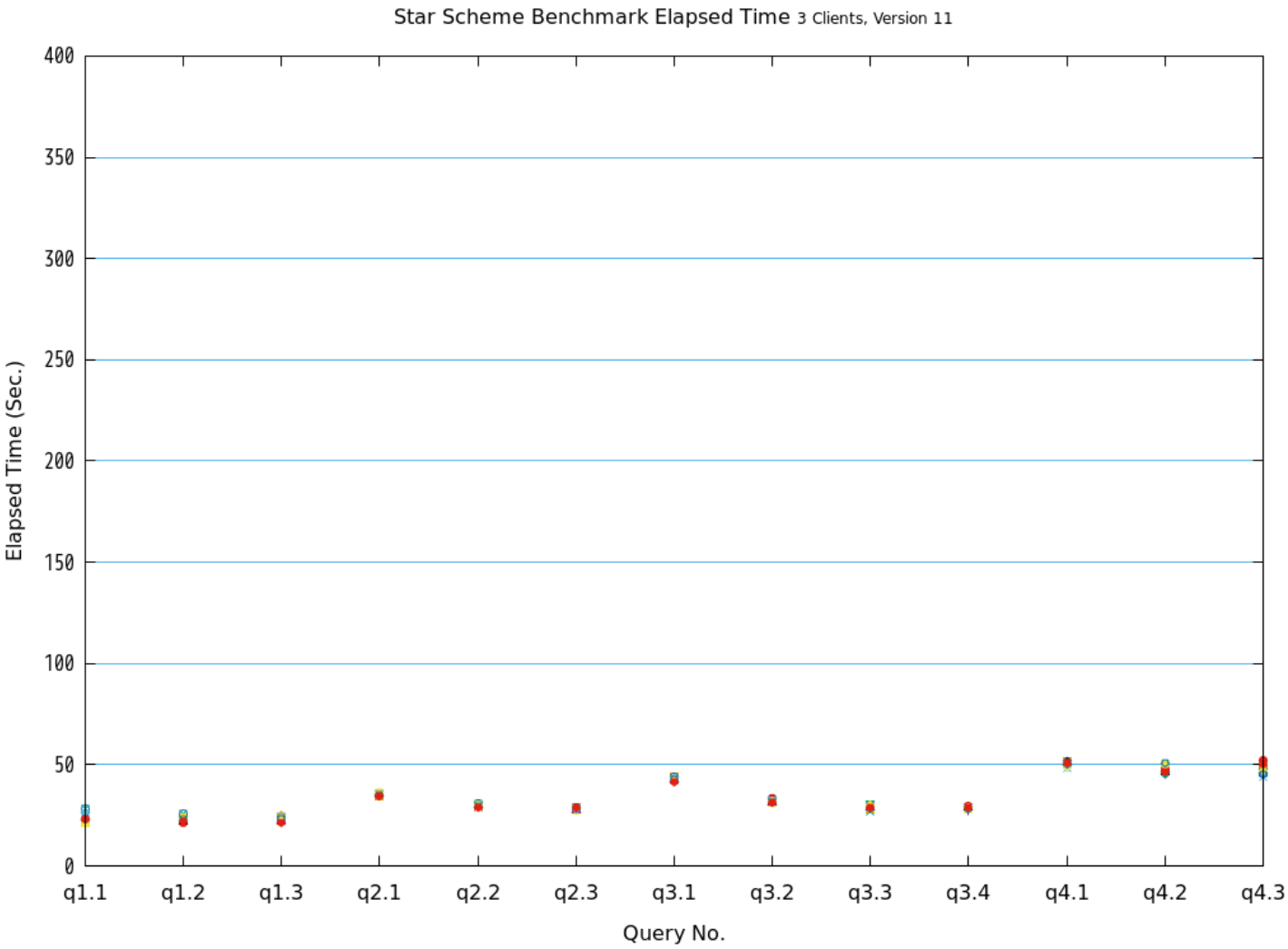


図 6.13 処理時間の比較 (v11, workers=100, 3 Clients)

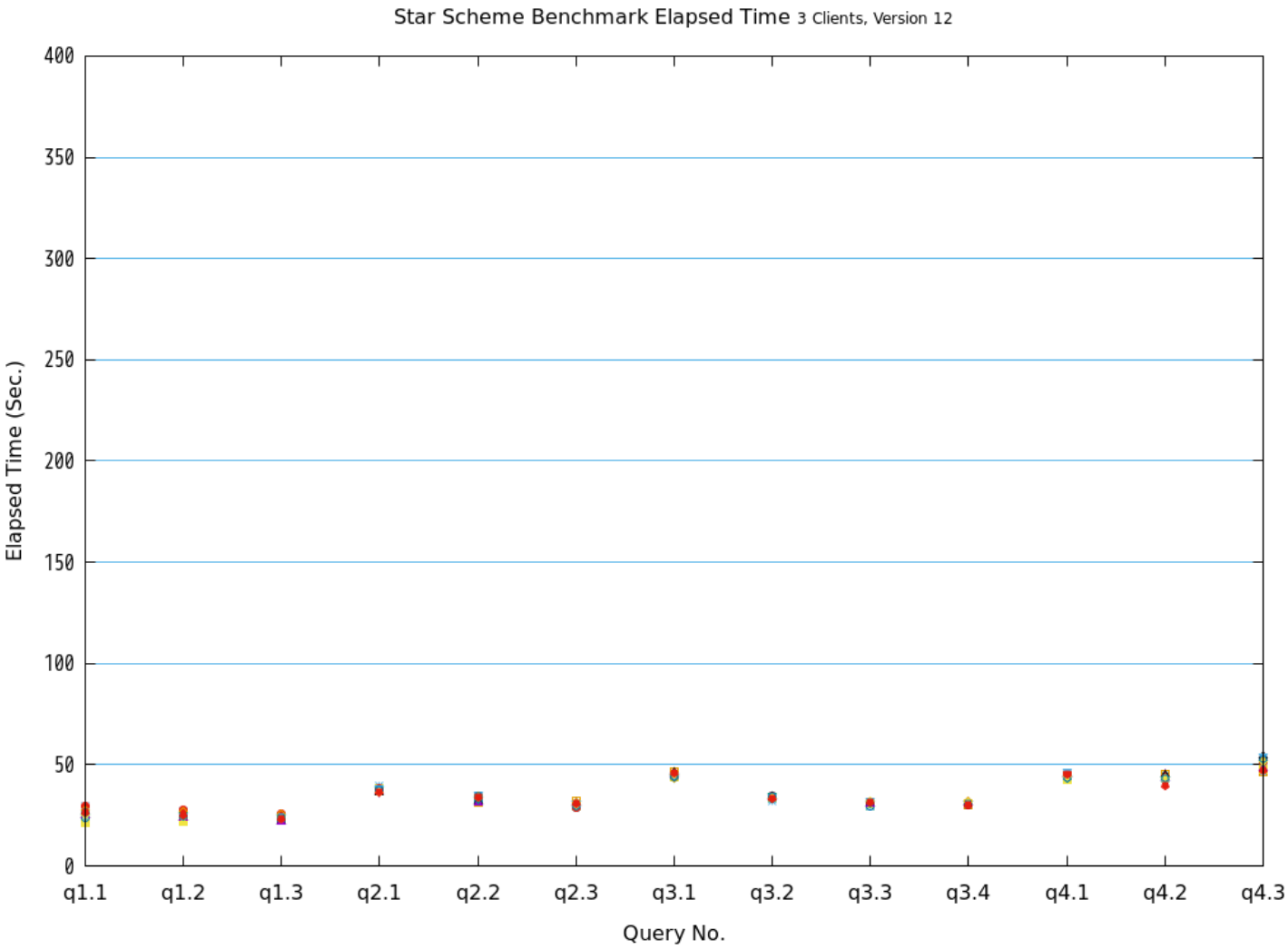


図 6.14 処理時間の比較 (v12, workers=100, 3 Clients)

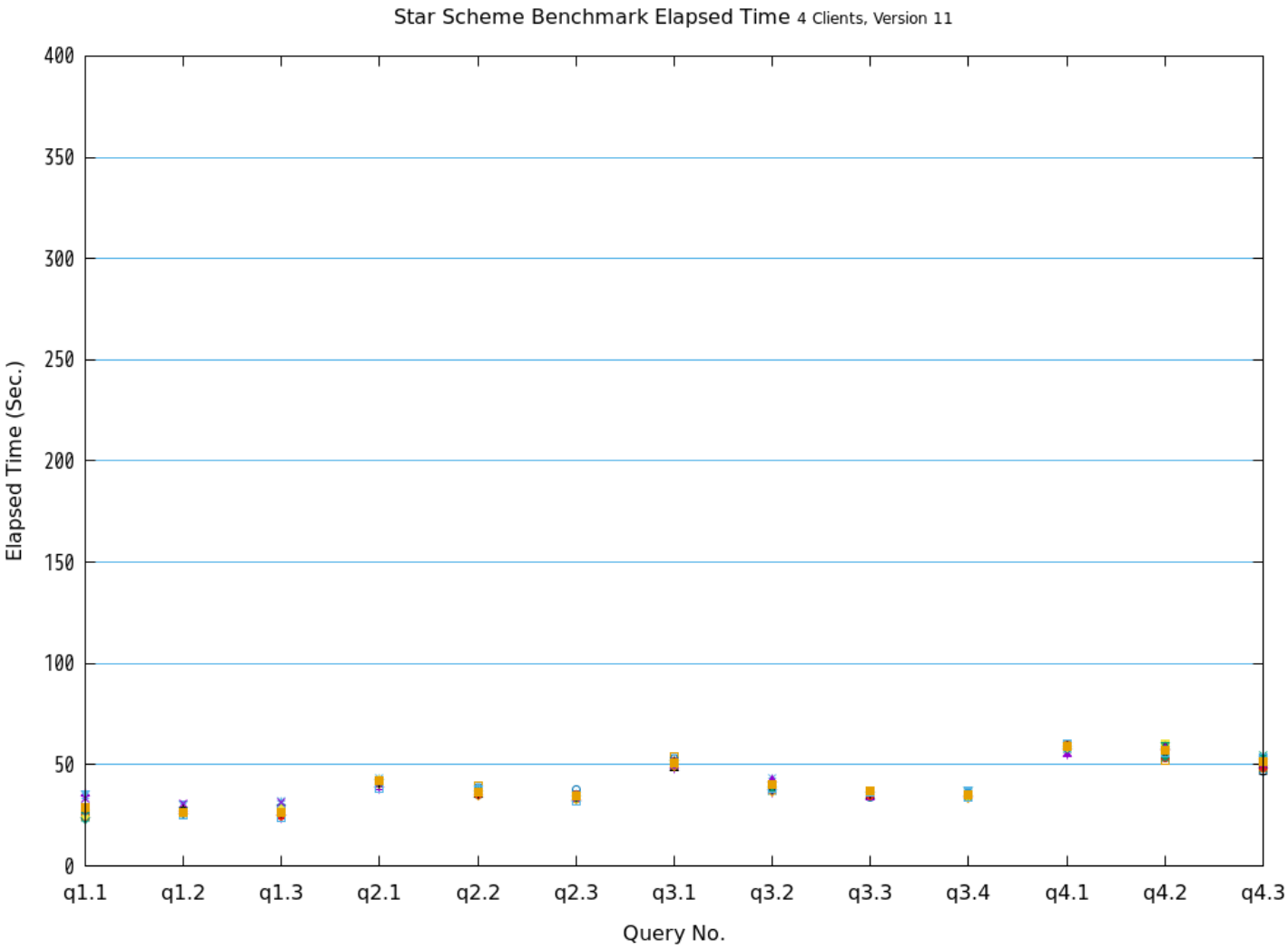


図 6.15 処理時間の比較 (v11, workers=100, 4 Clients)

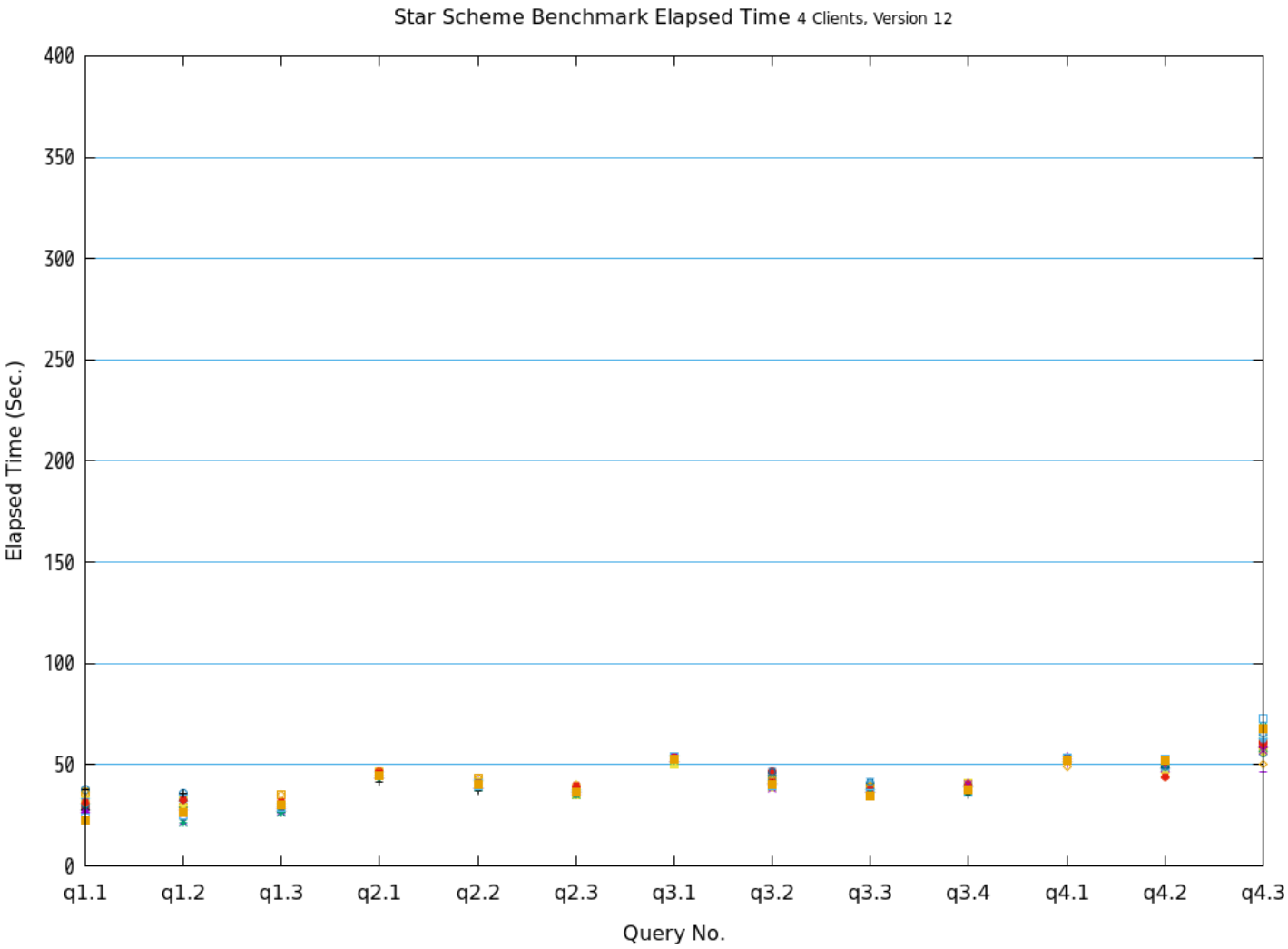


図 6.16 処理時間の比較 (v12, workers=100, 4 Clients)

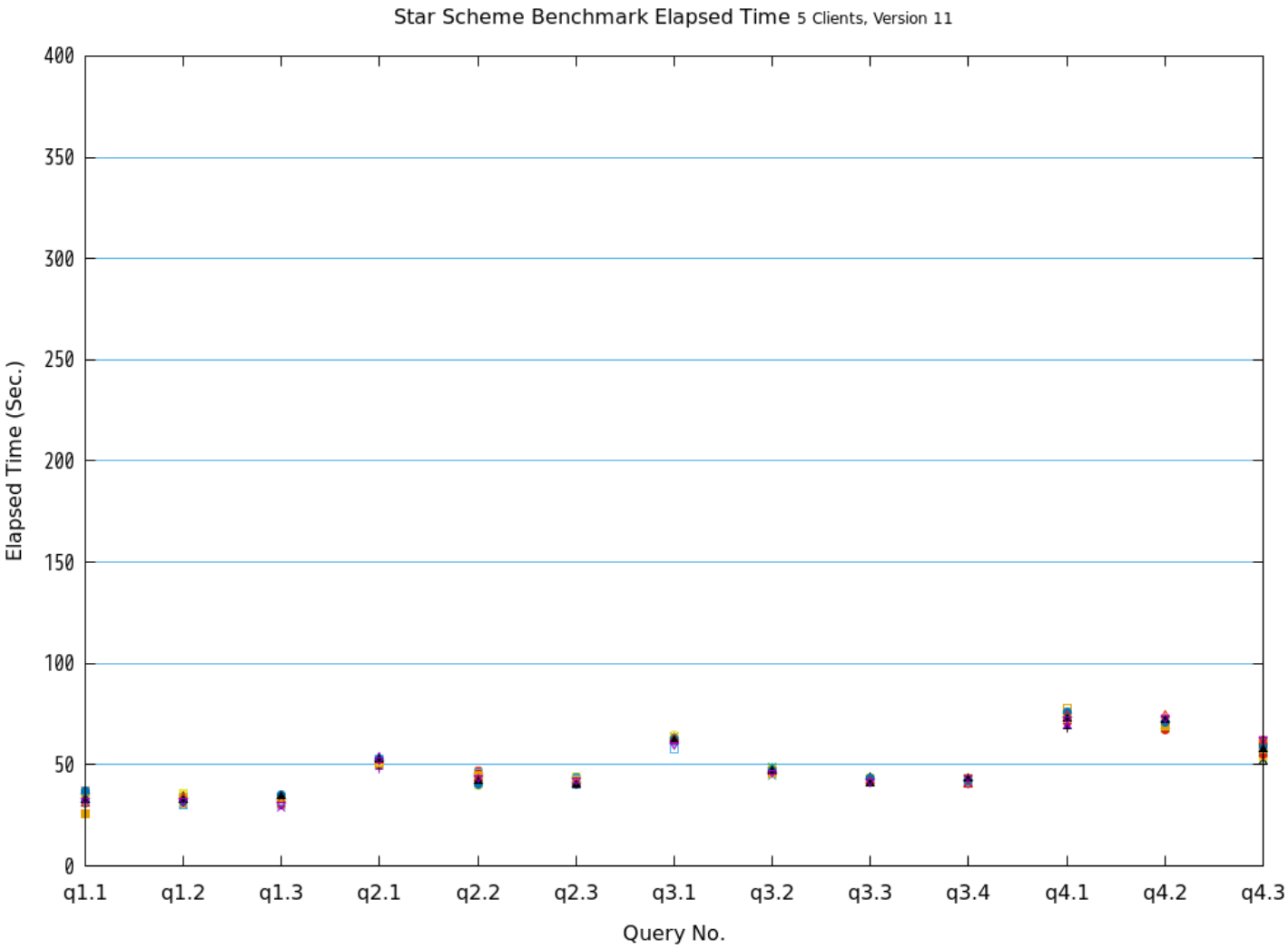


図 6.17 処理時間の比較 (v11, workers=100, 5 Clients)

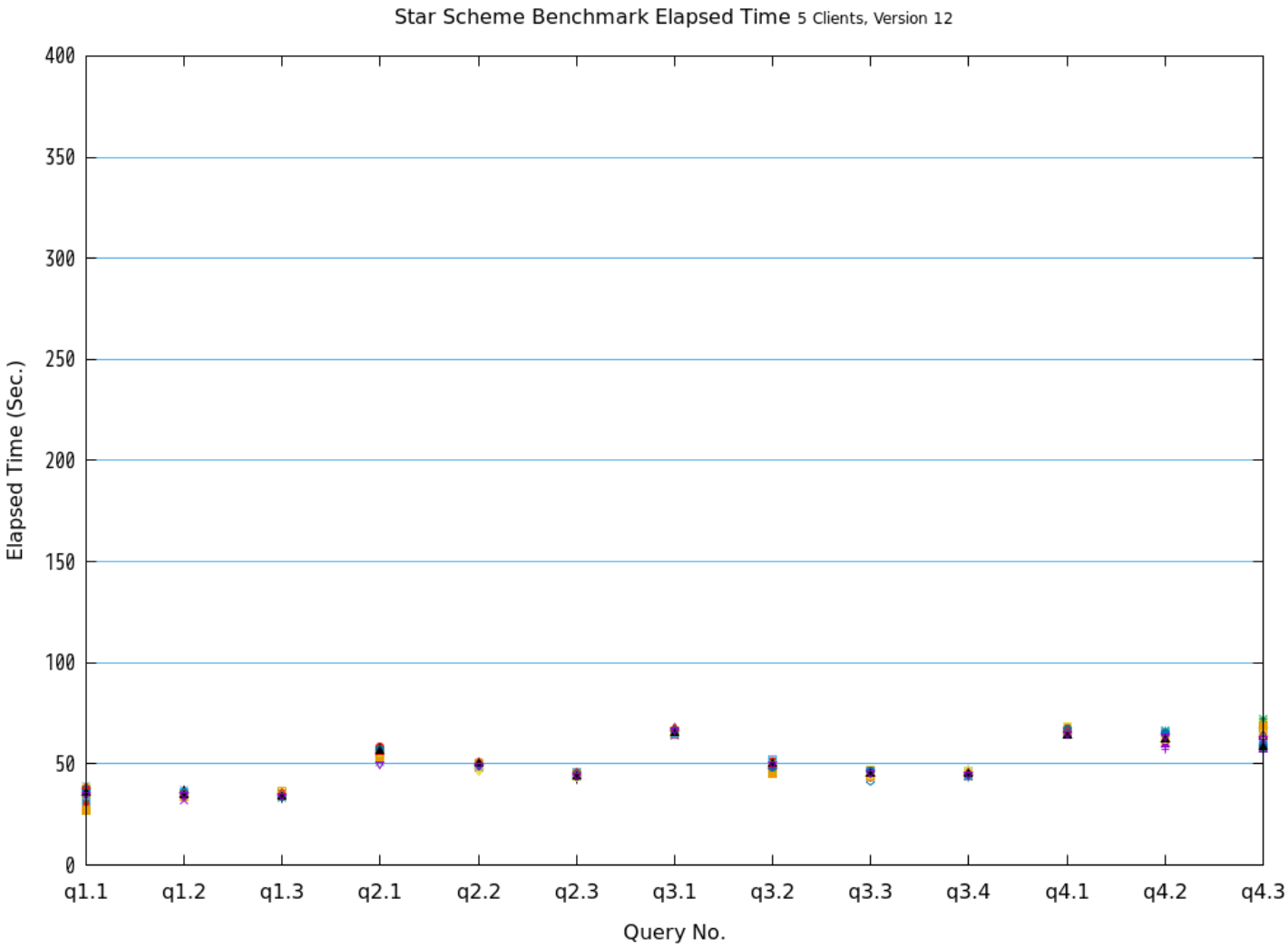


図 6.18 処理時間の比較 (v12, workers=100, 5 Clients)

検証の結果、クエリの応答時間について、すべての測定値が狭い範囲に集中しクラスターを形成したことがわかります。 これは、パラレルワーカーの数を大きくしているため、すべての場合においてパラレルワーカーを確保できたことを示しています。

6.6.2. 検証B(JITコンパイルを有効にした場合の応答時間)

1クライアントでの実行における、バージョン11とバージョン12の処理時間のグラフを以下に示します。このグラフにおける数値は、5回ベンチマークを実行して採取した測定値の中央値を採用しています。

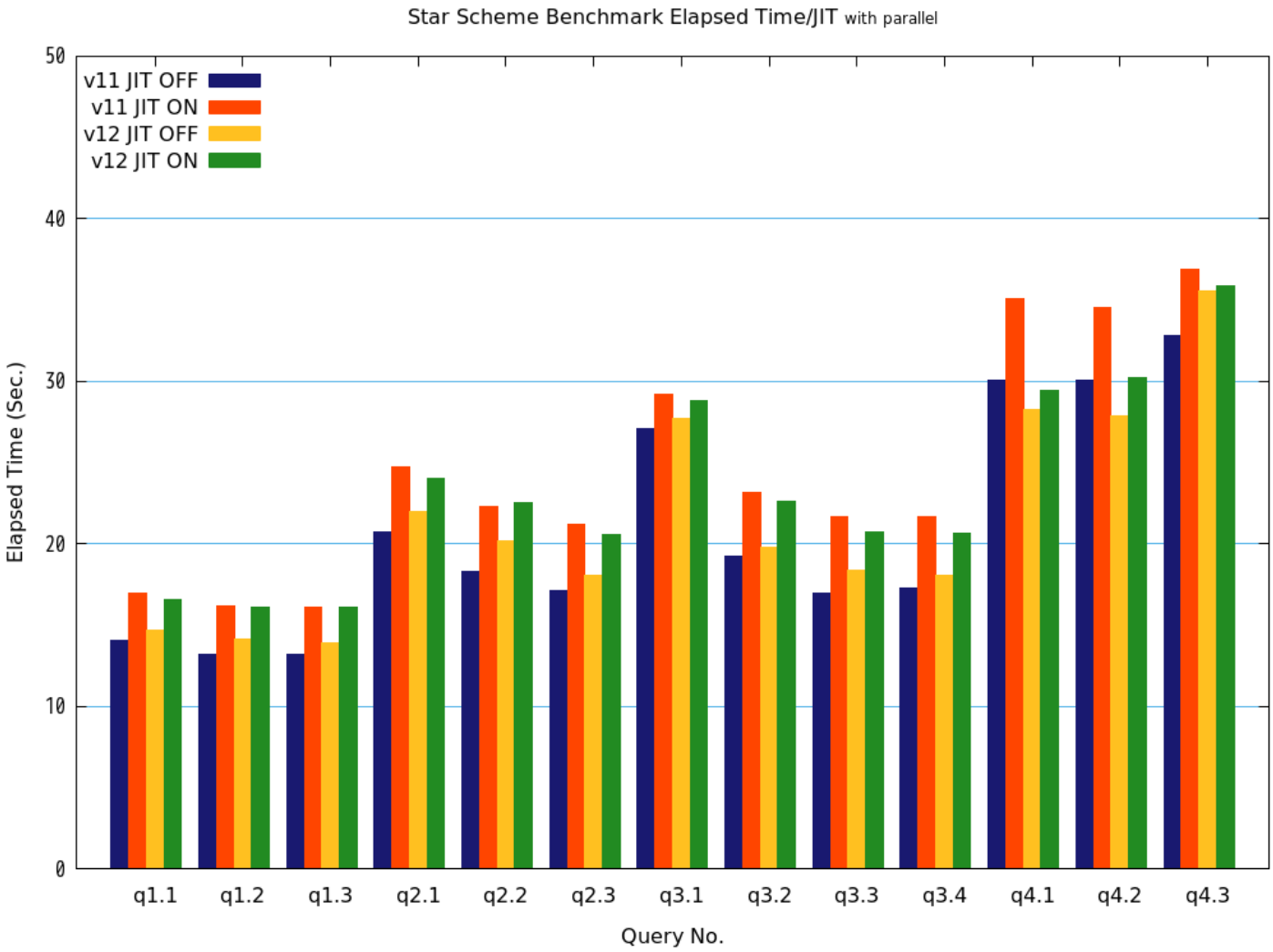


図 6.19 処理時間の比較 (1 Clients)

検証の結果、クエリの処理時間について、バージョン11では、JITコンパイル無効時の処理時間 / JITコンパイル有効時の処理時間の比率で0.78倍(q3.3)〜0.92倍(q3.1)の結果となり、すべてのクエリにおいて性能が劣化していました。 また、バージョン12では、同様にJITコンパイル無効時の処理時間 / JITコンパイル有効時の処理時間の比率で0.86倍(q1.3)〜0.99倍(q4.3)の結果となり、こちらもすべてのクエリにおいて性能が劣化していました。

JITコンパイルを有効にした状態でのバージョン11とバージョン12の処理時間を比較すると、バージョン11の処理時間 / バージョン12の処理時間の比率で、q2.2では0.99倍と若干の性能劣化が見られましたが、q2.2以外では1.00倍(q1.3)〜1.19倍(q4.1)と性能は改善していました。

次に、max_parallel_workers=9の場合の、クライアント数毎(2クライアント以上)、および、バージョン毎の処理時間のグラフを以下に示します。このグラフにおける数値は、5回ベンチマークを実行して採取した全クライアントの全測定値をプロットしています。

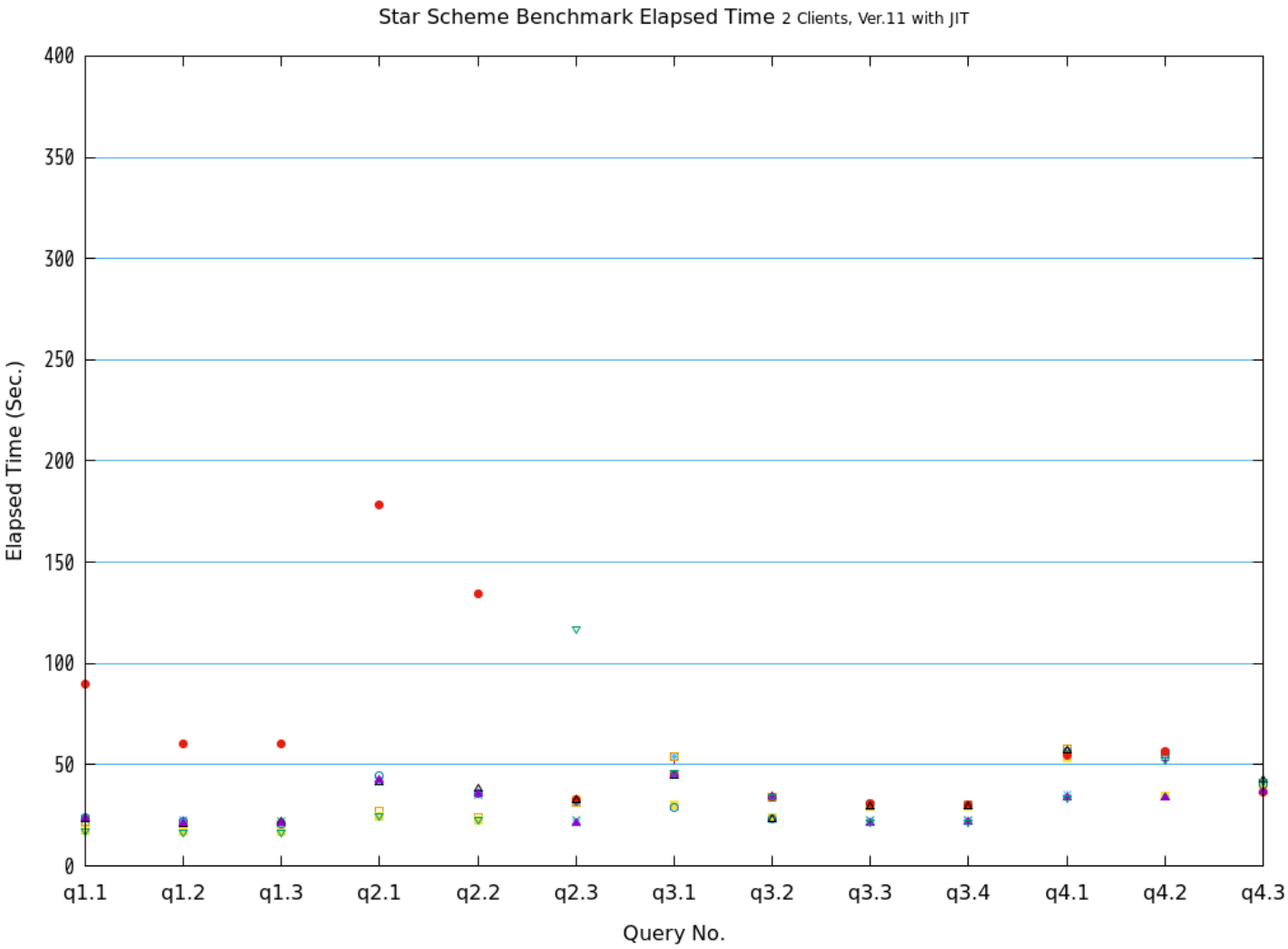


図 6.20 処理時間の比較 (v11, workers=9, 2 Clients, JITコンパイル有効)

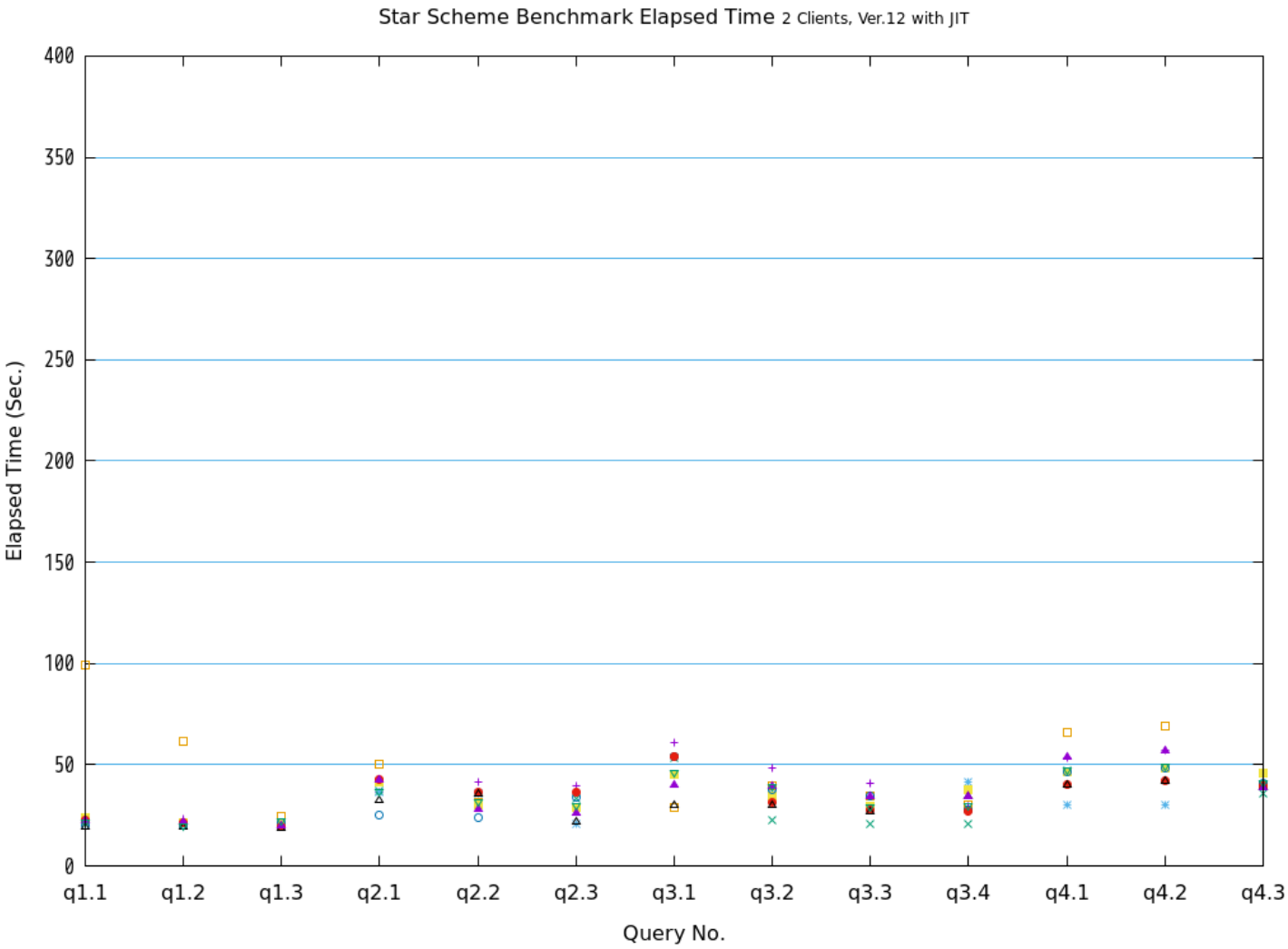


図 6.21 処理時間の比較 (v12, workers=9, 2 Clients, JITコンパイル有効)

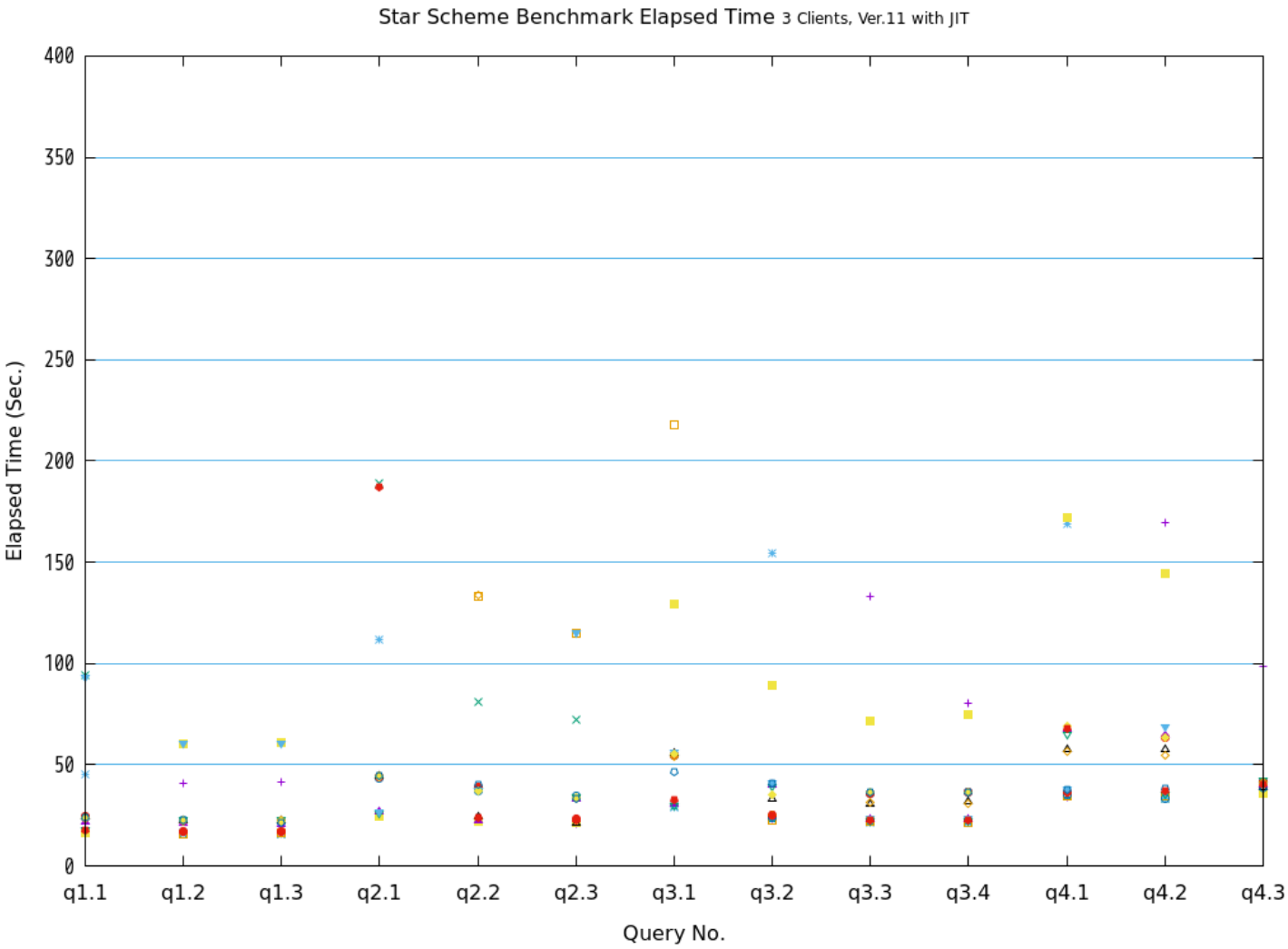


図 6.22 処理時間の比較 (v11, workers=9, 3 Clients, JITコンパイル有効)

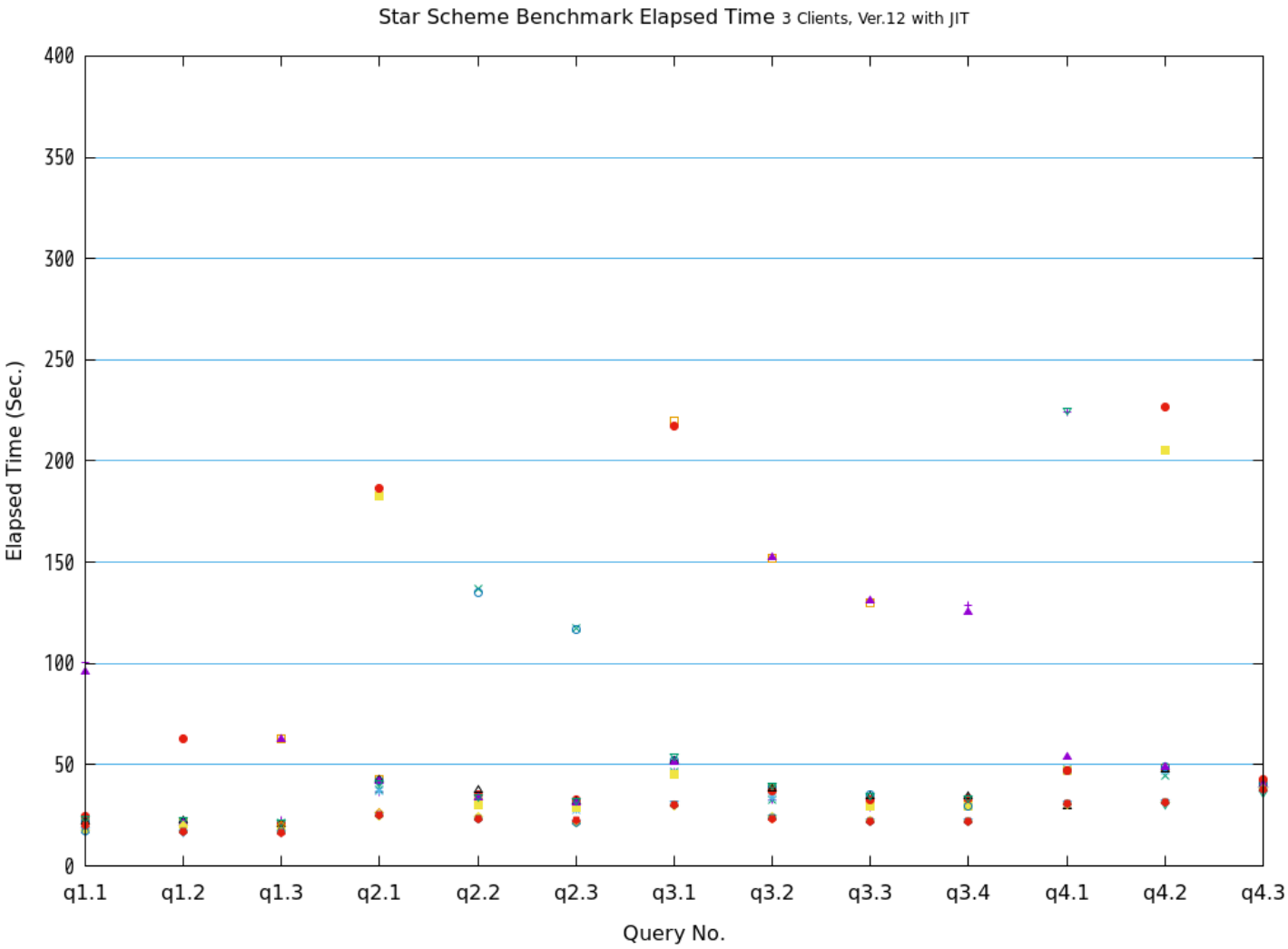


図 6.23 処理時間の比較 (v12, workers=9, 3 Clients, JITコンパイル有効)

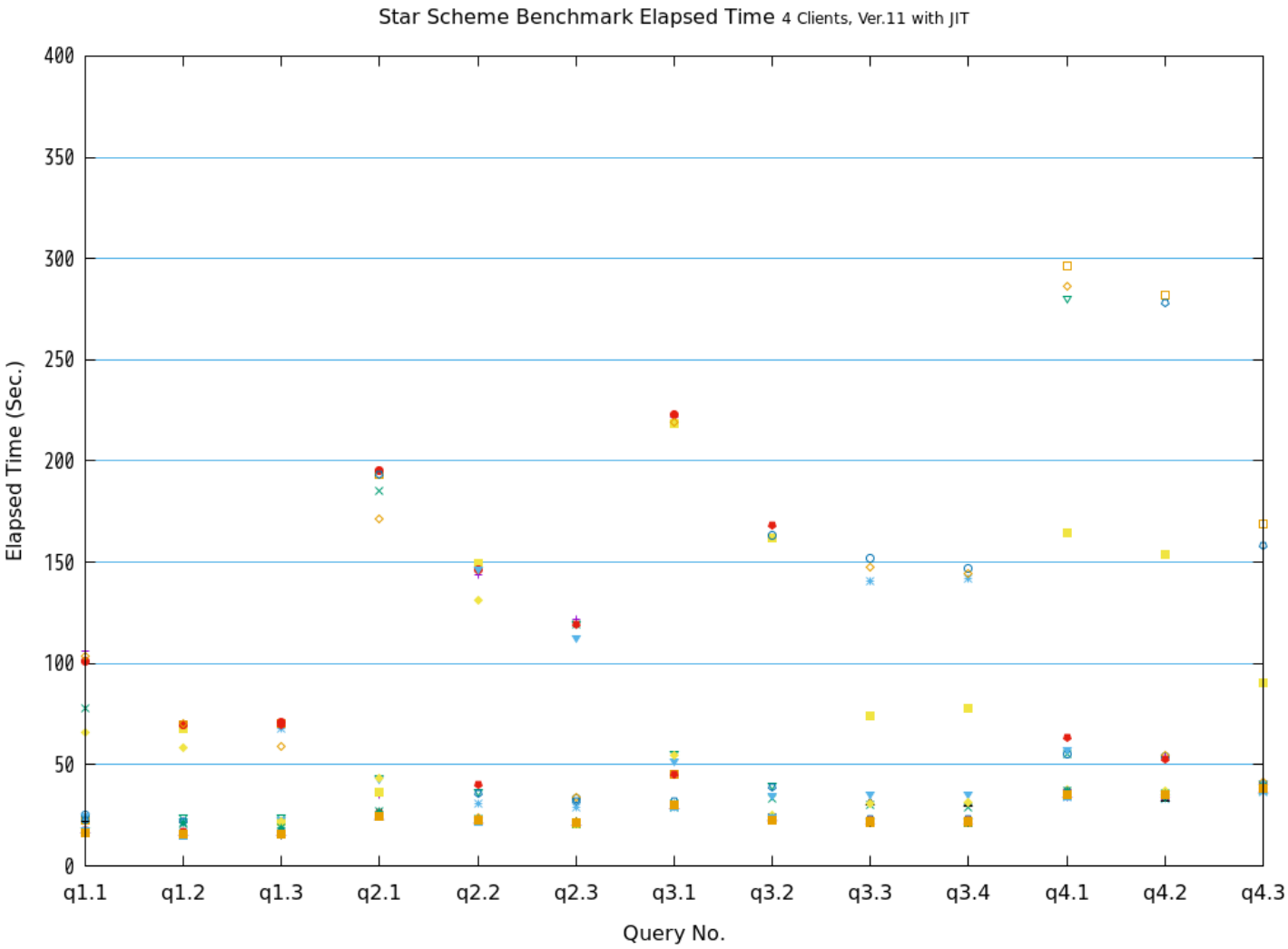


図 6.24 処理時間の比較 (v11, workers=9, 4 Clients, JITコンパイル有効)

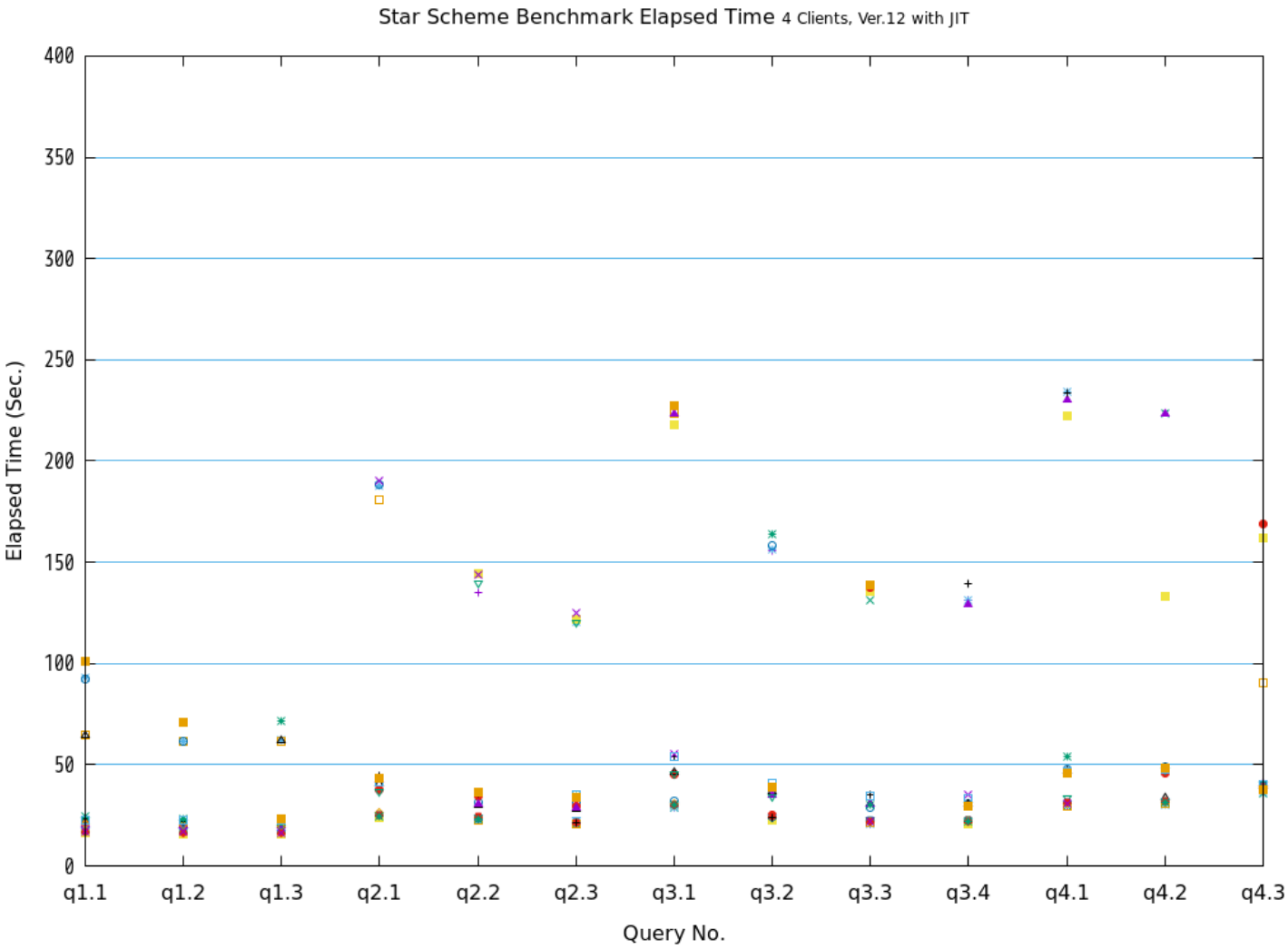


図 6.25 処理時間の比較 (v12, workers=9, 4 Clients, JITコンパイル有効)

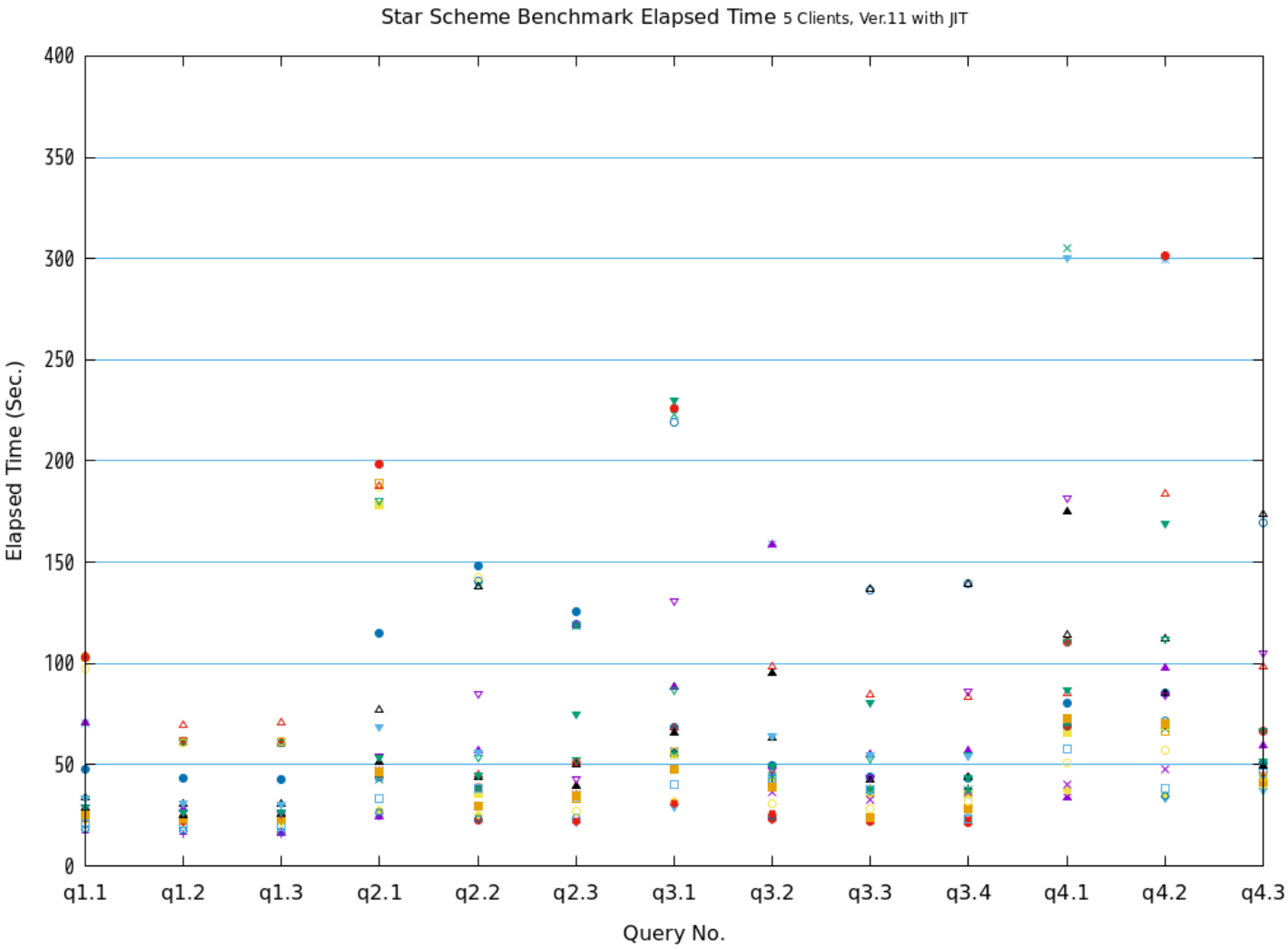


図 6.26 処理時間の比較 (v11, workers=9, 5 Clients, JITコンパイル有効)

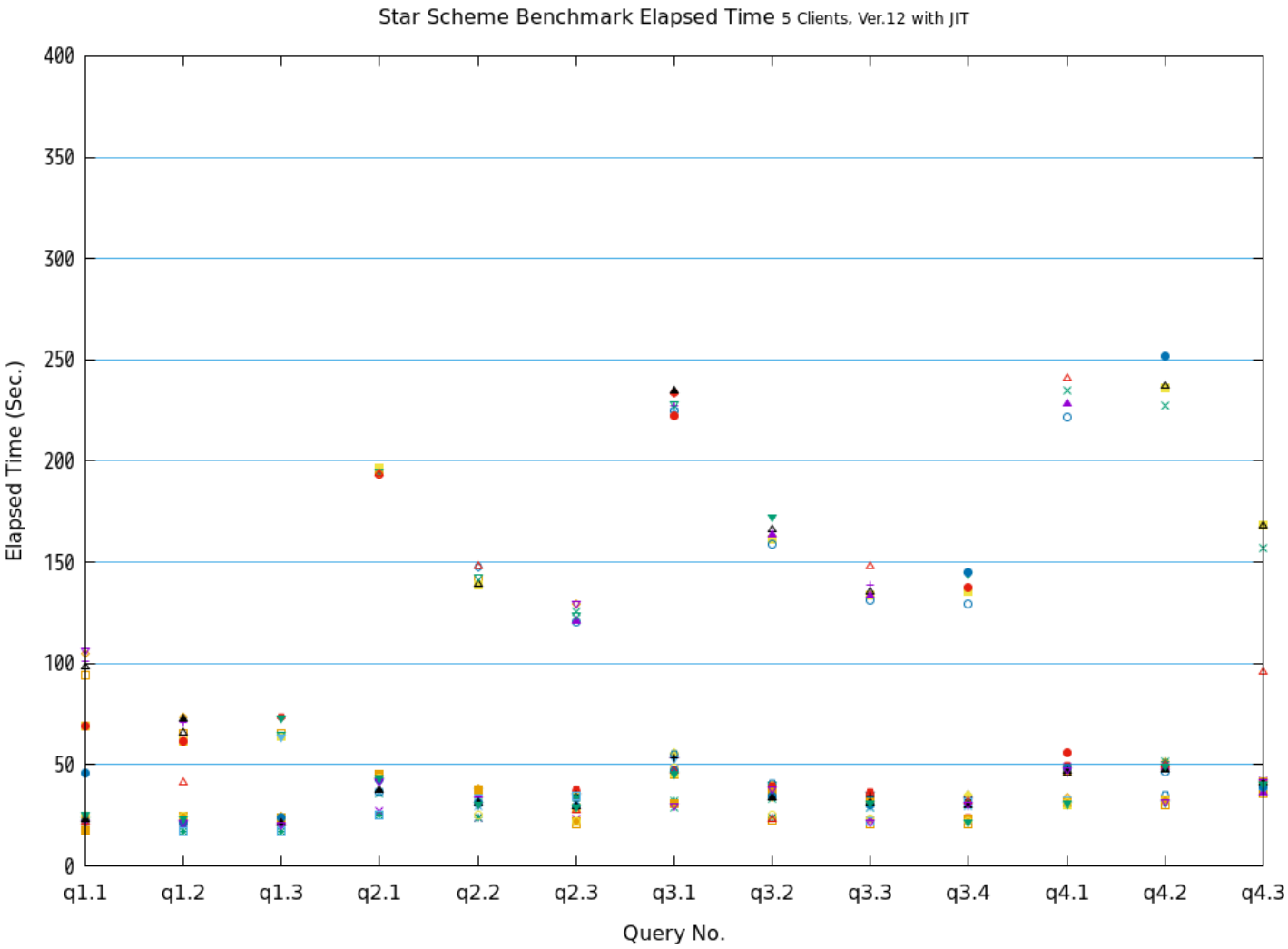


図 6.27 処理時間の比較 (v12, workers=9, 5 Clients, JITコンパイル有効)

検証の結果、クエリの応答時間について、検証Aの結果と同じく、多くの測定値が狭い範囲に集中しクラスターを形成している一方、一部の測定値はこのクラスターから上位の方向に大きく異なった値となっていることがわかります。 これは、パラレルワーカーの数を制限しているため、パラレルワーカーを確保できた場合とそうでない場合で、結果が分かれたことを示しています。

次に、max_parallel_workers=100の場合の、クライアント数毎(2クライアント以上)、および、バージョン毎の処理時間のグラフを以下に示します。

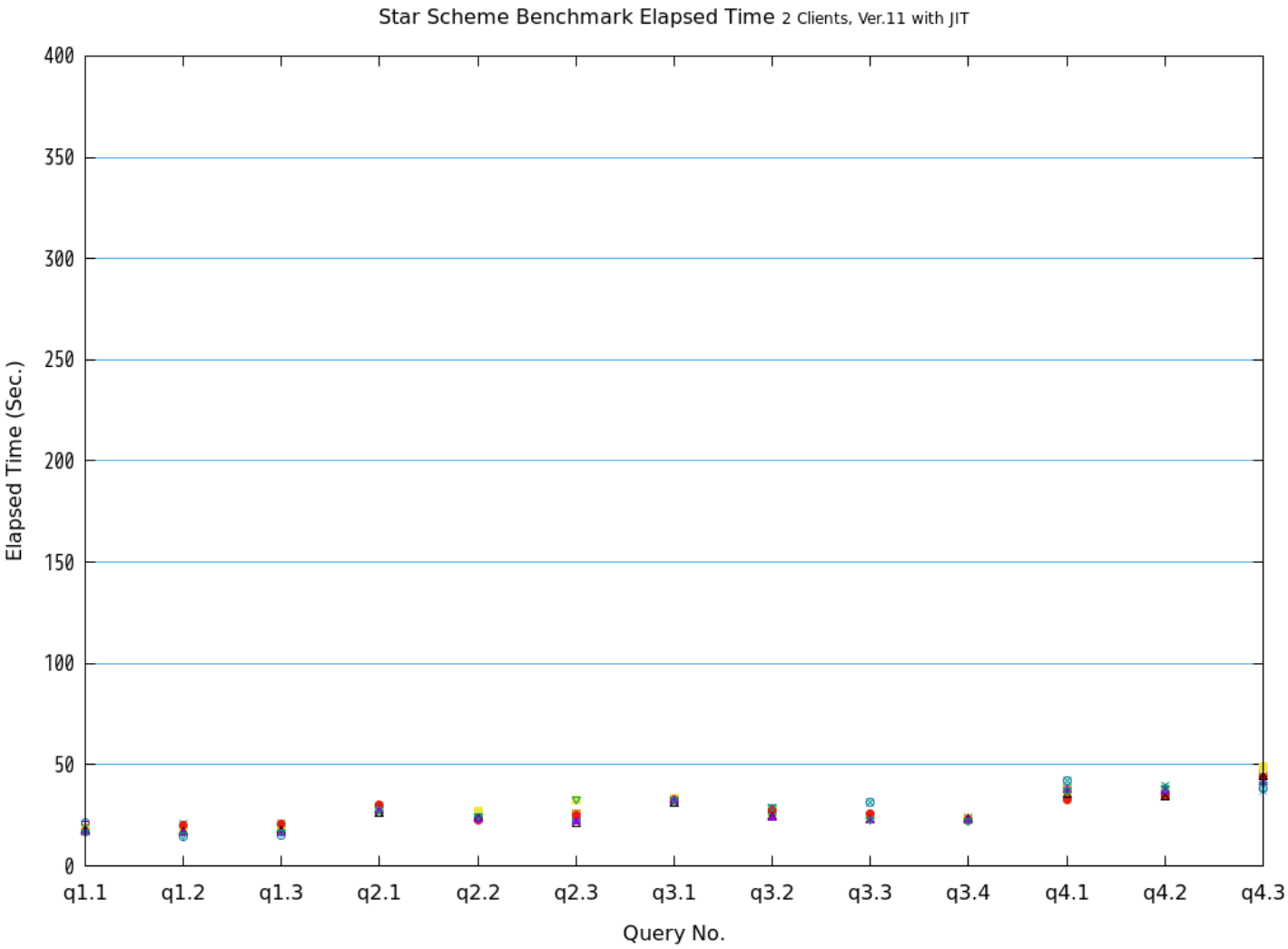


図 6.28 処理時間の比較 (v11, workers=100, 2 Clients, JITコンパイル有効)

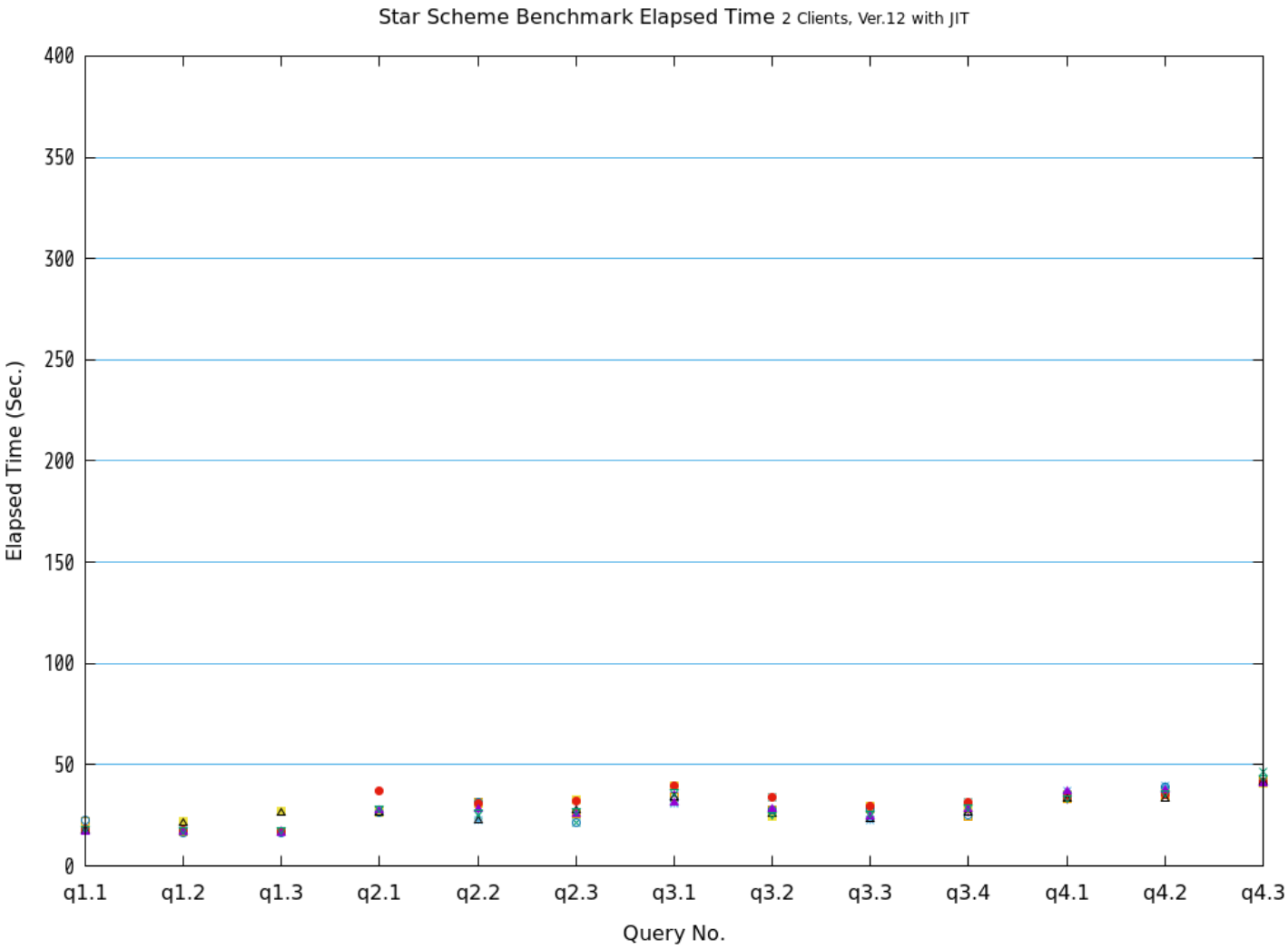


図 6.29 処理時間の比較 (v12, workers=100, 2 Clients, JITコンパイル有効)

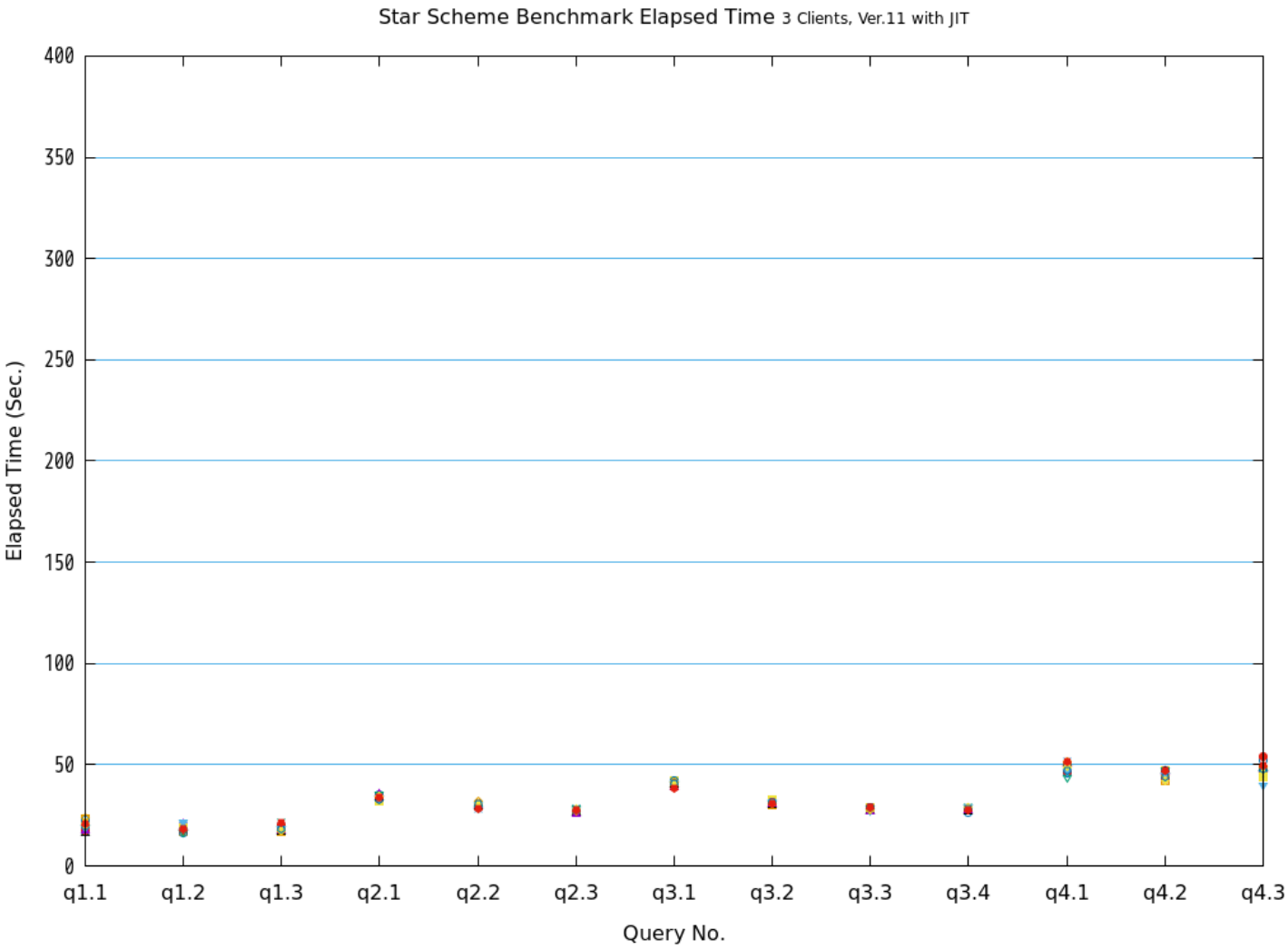


図 6.30 処理時間の比較 (v11, workers=100, 3 Clients, JITコンパイル有効)

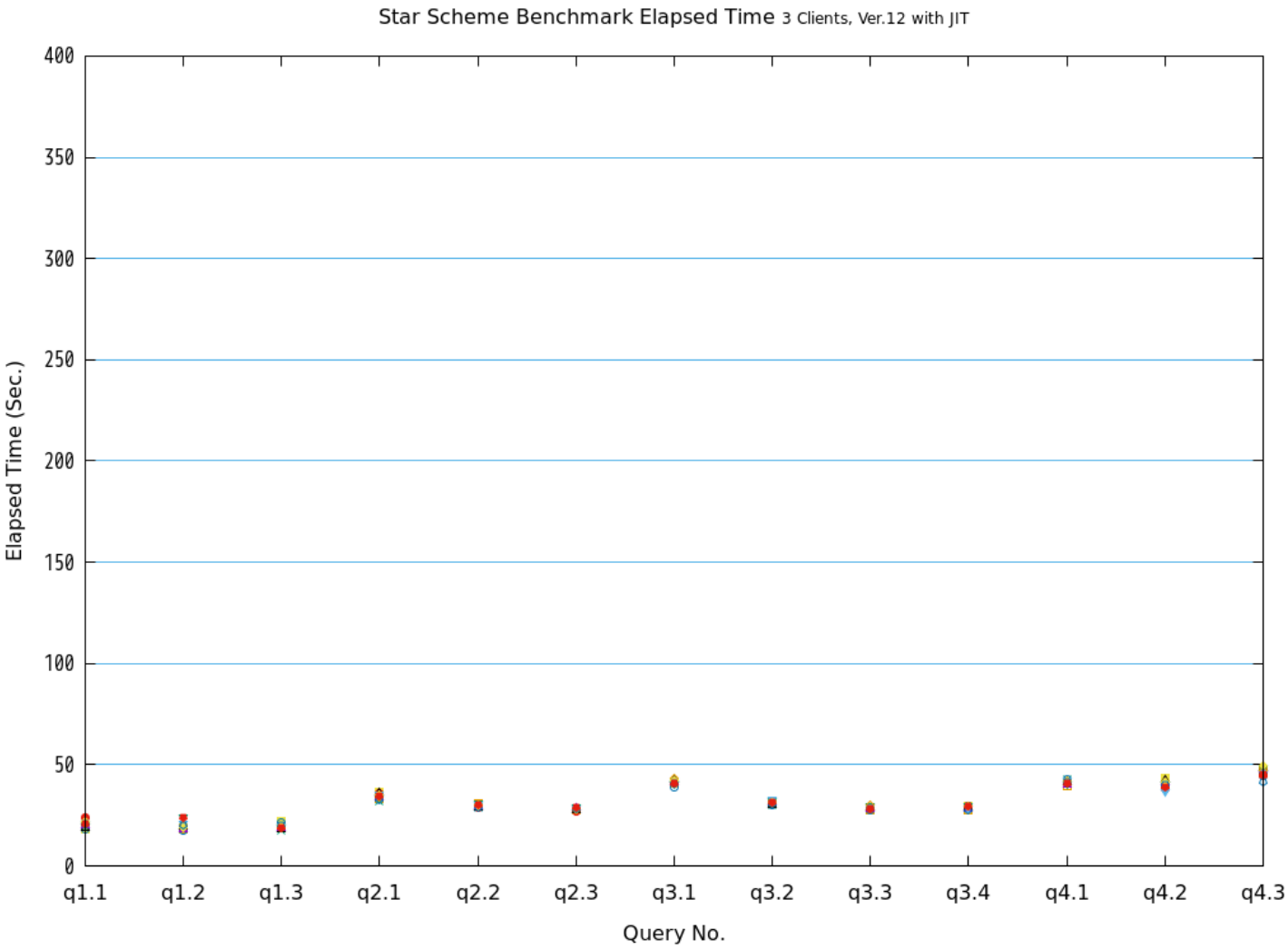


図 6.31 処理時間の比較 (v12, workers=100, 3 Clients, JITコンパイル有効)

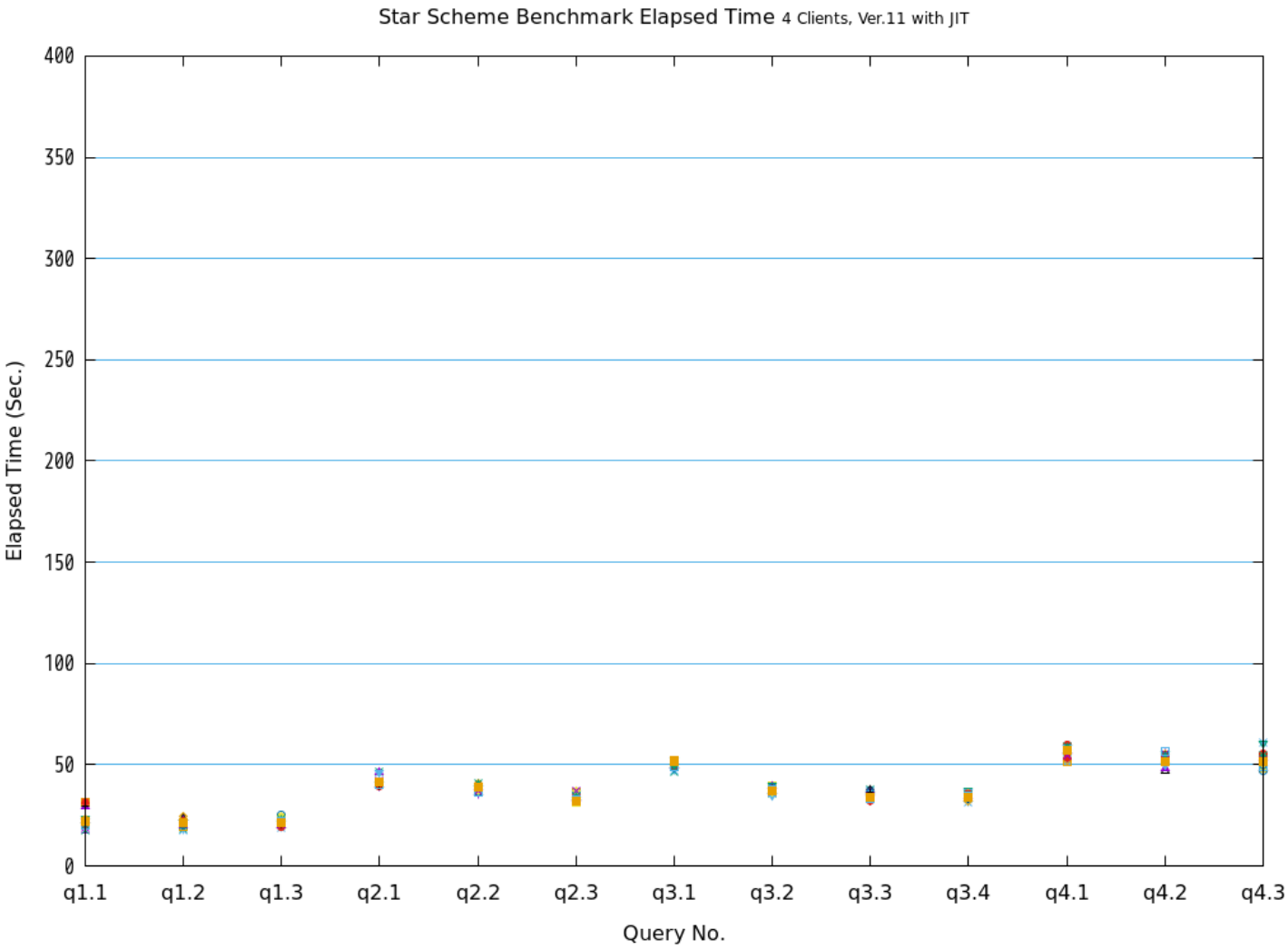


図 6.32 処理時間の比較 (v11, workers=100, 4 Clients, JITコンパイル有効)

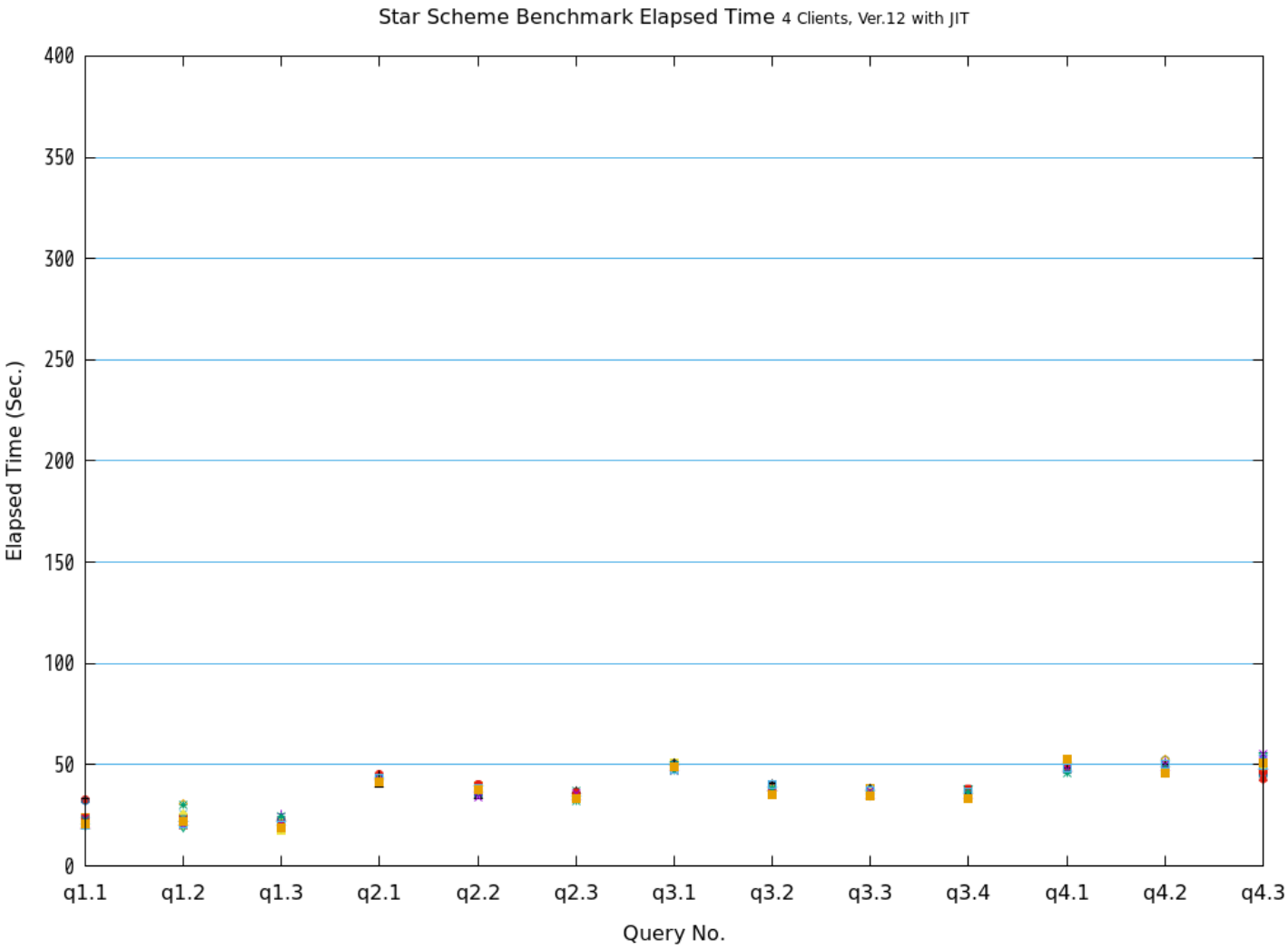


図 6.33 処理時間の比較 (v12, workers=100, 4 Clients, JITコンパイル有効)

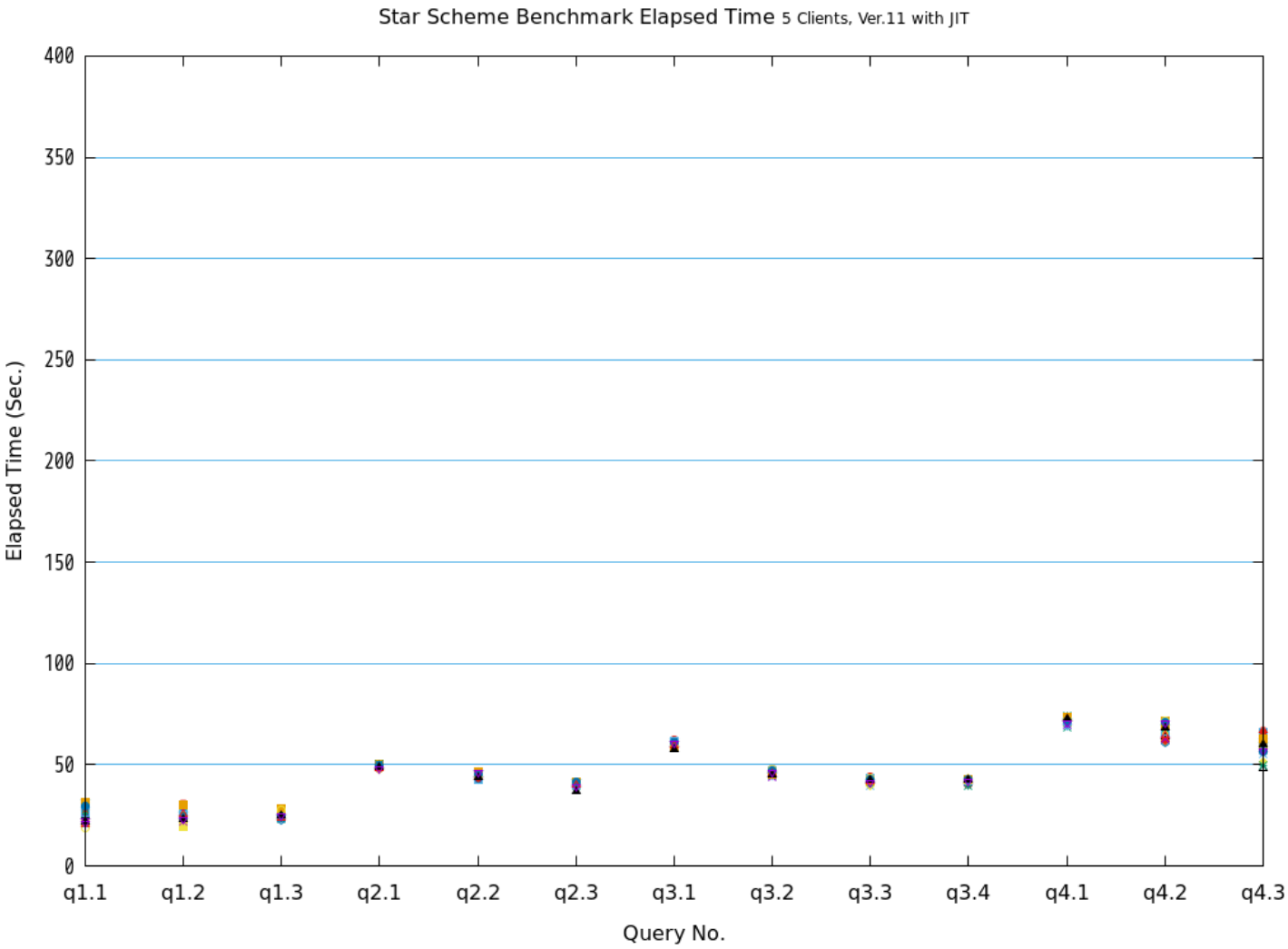


図 6.34 処理時間の比較 (v11, workers=100, 5 Clients, JITコンパイル有効)

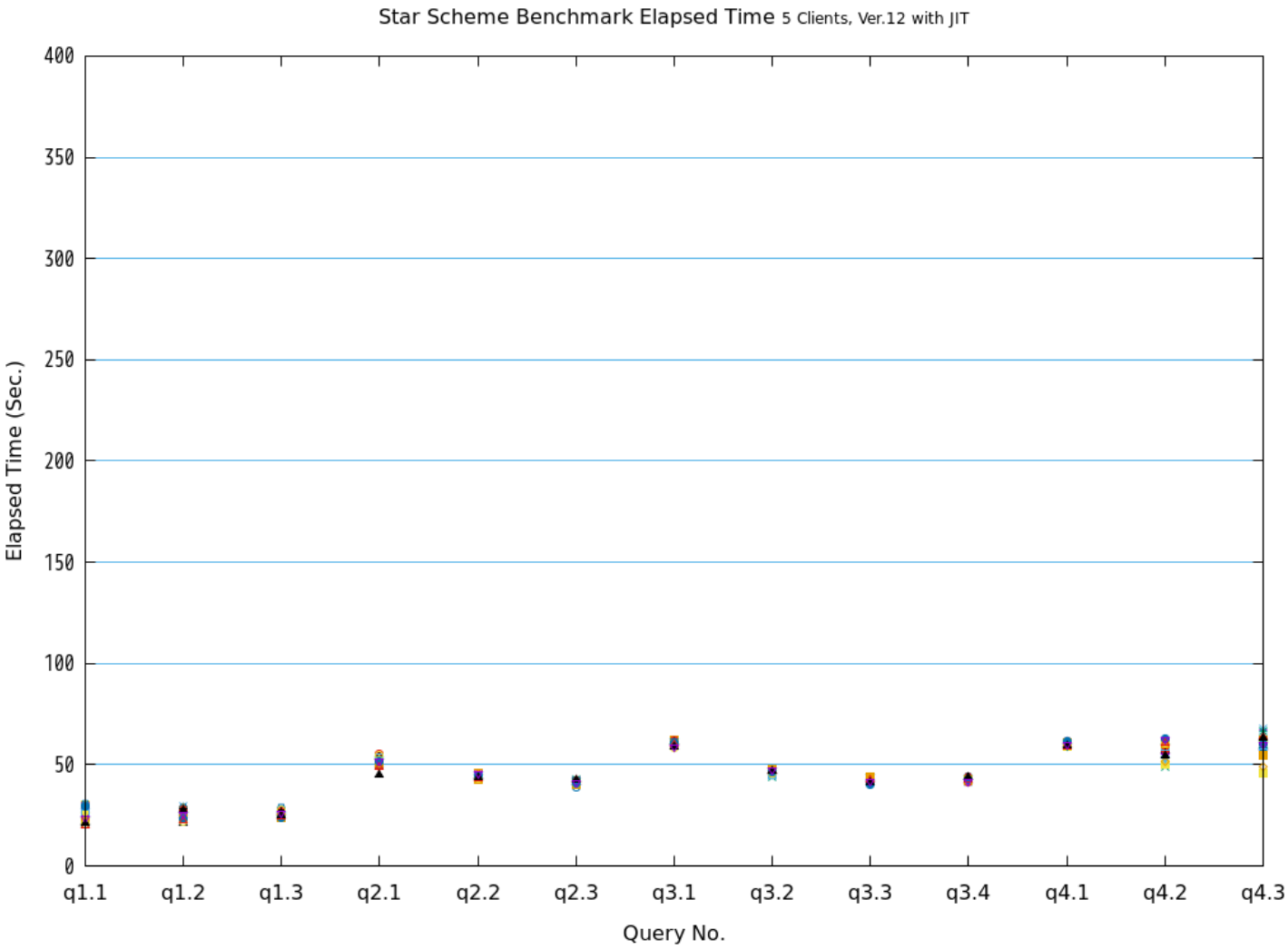


図 6.35 処理時間の比較 (v12, workers=100, 5 Clients, JITコンパイル有効)

検証の結果、クエリの応答時間について、検証Aの結果と同じく、すべての測定値が狭い範囲に集中しクラスターを形成したことがわかります。これは、パラレルワーカーの数を大きくしているため、すべての場合においてパラレルワーカーを確保できたことを示しています。

6.6.3. 対照群(パラレルクエリoff)

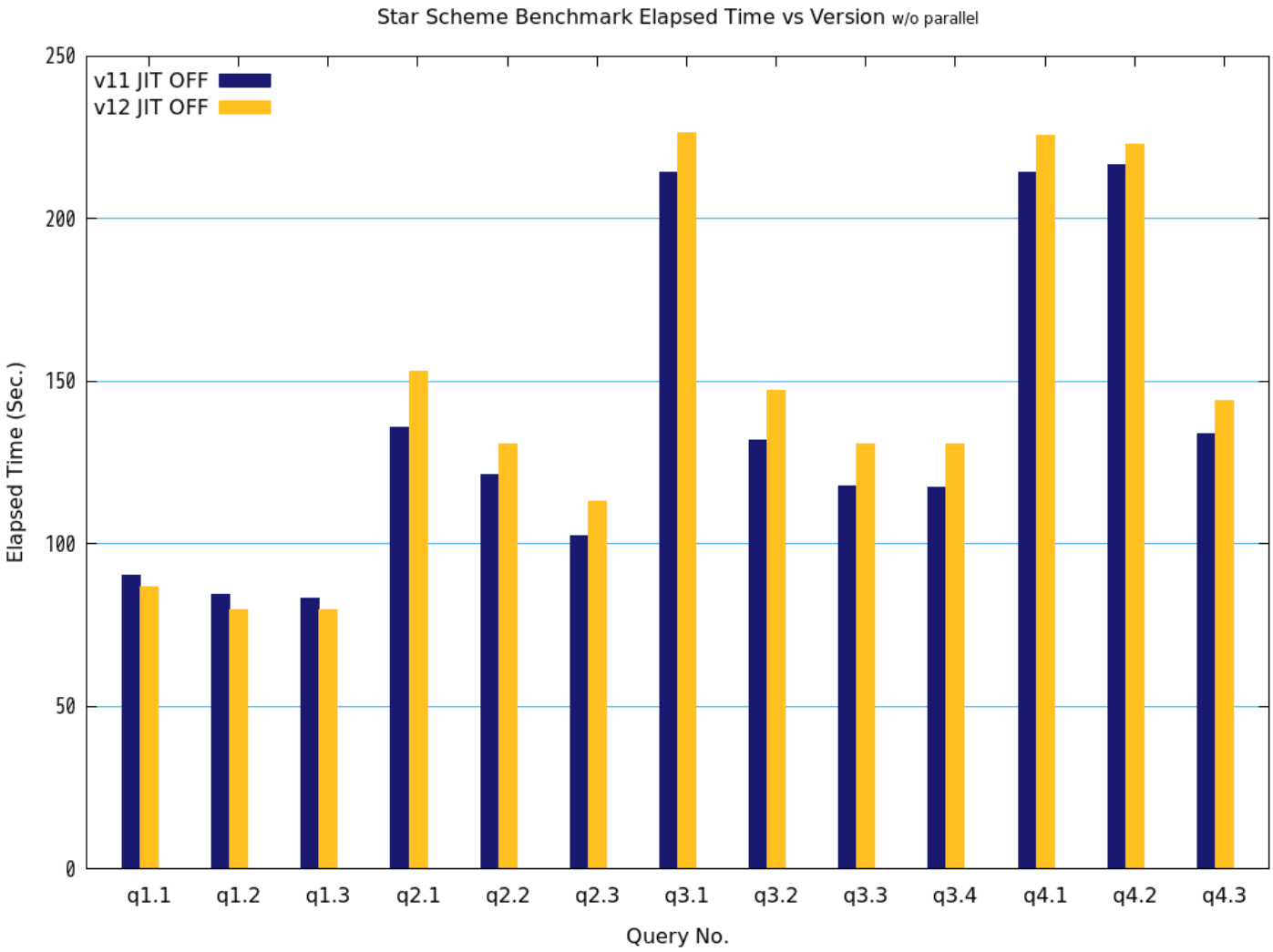


図 6.36 バージョンと処理時間の比較

表 6.14 バージョンと処理時間

クエリ	11 処理時間(秒)	12 処理時間(秒)	11 性能向上比(倍)
q1.1	90.2532	86.8419	1.0393
q1.2	84.2015	79.4986	1.0592
q1.3	83.0589	79.6268	1.0431
q2.1	135.8966	153.1511	0.8873
q2.2	121.2373	130.8429	0.9266
q2.3	102.2982	112.8655	0.9064
q3.1	214.0793	226.4705	0.9453
q3.2	131.7515	147.1222	0.8955
q3.3	117.6469	130.8489	0.8991
q3.4	117.4614	130.8053	0.8980
q4.1	214.4227	225.8097	0.9496
q4.2	216.6857	222.9774	0.9718
q4.3	133.6440	144.0128	0.9280

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11との比較で1.04倍(q1.1)〜1.06倍(q1.2)の高速化がなされている一方で、0.89倍(q2.1)〜0.97倍(q4.2)の性能劣化が見られるクエリもありました。

ここから、パラレルクエリ以外の部分で、バージョン11とバージョン12の間において、性能が向上した部分と、性能が劣化した部分があることがわかります。特に、q2.1以降で性能が劣化しています。

6.7. 考察

6.7.1. 検証A

6.7.1.1. max_parallel_workersの設定値について

検証Aでは、複数のクライアントからパラレルクエリを実行した際の動作を検証しました。

パラレルクエリが実際に動作するかは、実行された時点でパラレルワーカーを確保できるかどうかで決定します。 一方で、パラレルワーカーがもっとも性能を発揮できるのは、パラレルワーカーとCPUの物理コアが1:1で対応付けられるときです。 そのため、全体のパラレルワーカーの数を制限するパラメータである max_parallel_workers はCPUの物理コア数以下に設定するのが、性能をもっとも引き出す方法になります。

これは、パラレルクエリを実行するクライアントが1つの時には有効な方法ですが、複数のクライアントからパラレルクエリが実行された場合には、 少数のクライアントがパラレルクエリとして実行され、ほとんどのクライアントではパラレルクエリとして実行されなくなることになります。 しかし、 max_parallel_workers について、CPUの物理コア数を越える値に設定すると、パラレルワーカーとCPU物理コアがn:1の対応となり、 OSの設定にしたがってCPUを時分割で共有することになります。

システム全体として考えた場合、パラレルワーカーの性能をもっとも発揮できる設定にした方が良いのか、 それともクライアント全体にパラレルワーカーを割り当てられるようにした方が良いのかを考えます。

max_parallel_workersを9に設定した場合と100に設定した場合の、クライアント数2以上の各パターンについて、 各クライアントにおけるベンチマーク1回の所要時間の平均値を示します。

表 6.15 クエリ毎のパラレルワーカー数(クライアント数 2の場合)

バージョン	max_parallel_workers 9	max_parallel_workers 100
v11	2185.3871	1856.2326
v12	2034.8982	1862.8965

表 6.16 クエリ毎のパラレルワーカー数(クライアント数 3の場合)

バージョン	max_parallel_workers 9	max_parallel_workers 100
v11	2644.2678	2211.1354
v12	2752.8643	2267.3627

表 6.17 クエリ毎のパラレルワーカー数(クライアント数 4の場合)

バージョン	max_parallel_workers 9	max_parallel_workers 100
v11	3162.5346	2600.9449
v12	3326.2849	2725.7836

表 6.18 クエリ毎のパラレルワーカー数(クライアント数 5の場合)

バージョン	max_parallel_workers 9	max_parallel_workers 100
v11	3627.5680	3165.4339
v12	3825.6987	3260.7030

以上の結果から、いずれのクライアント数においても、max_parallel_workersを100に設定した場合の方が所要時間が短かかったことがわかります。 したがって、複数クライアントからパラレルクエリが同時に実行される場合、すべてのクライントにパラレルワーカーが割り当たるようにパラメータを設定したほうが良い、と考えます。

6.7.1.2. 実際に使用するパラレルワーカー数の決定方法

検証Aでは、複数のクライアントからパラレルクエリを実行した際の動作を検証しました。

パラレルクエリの実行には、パラレルワーカーと呼ばれる専用プロセスを、パラレルクエリを発行したクライアントを受け持つバックグラウンドプロセスが要求します。 この際、検索対象となるテーブルから算出されるワーカー数を要求することになりますが、その総数は max_parallel_workers で設定した値が上限です。 複数のクライアントから同時にパラレルクエリが実行された際、先に要求を出したバックグラウンドプロセスは要求値に近いパラレルワーカーを確保するものと予想していました。

今回測定に用いたScale Factorでは、パラレルワーカーの要求数がq4.3で4、それ以外で9となるため、 max_parallel_workers=9 とした場合の測定では、 q4.3が動作している期間を除いて、1クライアントのみがパラレルワーカーを確保して最大の性能を発揮する一方で、それ以外のクライアントはパラレルクエリとならないものと予想しました。

ところが、実際には確保されたパラレルワーカー数は、q4.3が動作していない期間でも9未満のワーカー数しか確保できないものが散見されました。 また、q4.3が1クライアントで動作している期間でも、0,4,5以外のワーカー数が見受けられました。

表 6.19 クエリ毎のパラレルワーカー数(クライアント数 5の場合)

ワーカー数	クエリ数
0	91
1	51
2	0
3	0
4	314
5	92
6	0
7	0
8	33
9	69

この原因を突き止めるため、パラレルワーカーの確保を行う部分のロジックをバージョン12のソースコードにて確認しました。 この結果、バックグラウンドプロセスがパラレルワーカーを要求する部分のコードが以下であることを確認しました。

```
/*
 * Start workers.
 *
 * The caller must be able to tolerate ending up with fewer workers than
 * expected, so there is no need to throw an error here if registration
 * fails.  It wouldn't help much anyway, because registering the worker in
 * no way guarantees that it will start up and initialize successfully.
 */
for (i = 0; i < pcxt->nworkers; ++i)
{
    memcpy(worker.bgw_extra, &i, sizeof(int));
    if (!any_registrations_failed &&
        RegisterDynamicBackgroundWorker(&worker,
                                        &pcxt->worker[i].bgwhandle))
    {
        shm_mq_set_handle(pcxt->worker[i].error_mqh,
                          pcxt->worker[i].bgwhandle);
        pcxt->nworkers_launched++;
    }
    else
    {
        /*
         * If we weren't able to register the worker, then we've bumped up
         * against the max_worker_processes limit, and future
         * registrations will probably fail too, so arrange to skip them.
         * But we still have to execute this code for the remaining slots
         * to make sure that we forget about the error queues we budgeted
         * for those workers.  Otherwise, we'll wait for them to start,
         * but they never will.
         */
        any_registrations_failed = true;
        pcxt->worker[i].bgwhandle = NULL;
        shm_mq_detach(pcxt->worker[i].error_mqh);
        pcxt->worker[i].error_mqh = NULL;
    }
}
```

上記のコードのうち、RegisterDynamicBackgroundWorker関数がパラレルワーカーを確保するための関数であり、 この関数の中ではバックグラウンドワーカーのプールについて排他ロックを確保し、確保処理が完了次第、解放する仕組みになっています。 以上のコードから、以下のことがわかります。

- パラレルワーカーの要求数に達するまで、1ワーカーずつ確保していく
- 要求時にロック待ちとなり、確保はバックグラウンドプロセスにCPU時間が割り当てられたタイミング

- 一度でも確保に失敗すると、以降はワーカー確保を試みない

このことから、複数のクライアントから同時にパラレルクエリが実行された場合は、バックグラウンドプロセス間でワーカー確保について競合状態が発生し、 パラレルワーカーの確保数について想定外の数値となったと考えます。

なお、このロジックではCPU時間が割り当てられた順にパラレルワーカーが確保されていくため、 OSカーネルでのマルチプロセス処理の実装方法や、プロセスの優先度、CPU時間割当アルゴリズム、実行時のOS全体のプロセス数によって、 結果が異なることとなり、確保されるパラレルワーカー数を予測することは困難であると考えます。 パラレルクエリを積極的に使用するシステムの場合は、この事実を十分に考慮して設計を行う必要があると考えます。

6.7.2. 検証B

6.7.2.1. JITコンパイル有効時の性能について

検証Bでは、パラレルクエリとJITコンパイルを組み合わせたときの性能の変化について検証しました。

以下に、SSB q2.1におけるmax_parallel_workers = 9 の最小値と最大値の性能向上比を示します。

表 6.20 SSB q2.1におけるクライアント数と性能向上比(v11, max_parallel_workers=9)

クライアント数	最小値(JIT OFF)	最小値(JIT ON)	性能向上比(倍)	最大値(JIT OFF)	最大値(JIT ON)	性能向上比(倍)
2	21.8798	24.6441	0.8878	40.2771	178.4997	0.2256
3	20.4043	24.2157	0.8426	211.8032	188.9400	1.1210
4	21.8994	24.1819	0.9056	204.8099	195.0259	1.0502
5	20.2655	24.2261	0.8365	211.6770	198.4486	1.0667

表 6.21 SSB q2.1におけるクライアント数と性能向上比(v12, max_parallel_workers=9)

クライアント数	最小値(JIT OFF)	最小値(JIT ON)	性能向上比(倍)	最大値(JIT OFF)	最大値(JIT ON)	性能向上比(倍)
2	21.8798	25.4238	0.8606	218.8473	50.0089	4.3762
3	22.2803	24.5522	0.9075	231.9113	186.3409	1.2446
4	23.2664	23.9570	0.9712	238.7724	190.2858	1.2548
5	24.7194	25.0932	0.9851	247.4249	196.2438	1.2608

max_parallel_workers = 9 の場合、同時にクエリの実行が開始された場合、1つのクライアントがパラレルワーカーを独占することで、その他のクライアントがパラレルクエリになりづらくなります。 この状態はクライアント数が多くなるほど発生しやすくなります。この状態に陥った場合は、パラレルワーカーを要求通り確保できたクライアントで最小値に近い応答時間を示し、まったく確保できなかったクライアントでは最大値に近い応答時間を示します。 この場合、ワーカーを確保できなかったクライアント(最大値)に対しては、JITコンパイルを有効としたほうが応答時間が短くなることがわかります。 一方で、ワーカーを確保できたクライアント(最小値)ではJITコンパイルのオーバヘッドのほうが大きくなり、応答時間が長くなっていることがわかります。

以下に、SSB q2.1におけるmax_parallel_workers = 100 の最小値と最大値の性能向上比を示します。

表 6.22 SSB q2.1におけるクライアント数と性能向上比(v11, max_parallel_workers=100)

クライアント数	最小値(JIT OFF)	最小値(JIT ON)	性能向上比(倍)	最大値(JIT OFF)	最大値(JIT ON)	性能向上比(倍)
2	25.3473	25.6464	0.9883	28.9848	30.2443	0.9584
3	34.3063	32.1211	1.0680	36.1814	35.5195	1.0186
4	37.6560	38.5972	0.9756	43.2196	46.6782	0.9259
5	47.7980	47.6245	1.0036	53.7806	50.2430	1.0704

表 6.23 SSB q2.1におけるクライアント数と性能向上比(v12, max_parallel_workers=100)

クライアント数	最小値(JIT OFF)	最小値(JIT ON)	性能向上比(倍)	最大値(JIT OFF)	最大値(JIT ON)	性能向上比(倍)
2	27.4405	26.2998	1.0434	39.9517	36.9627	1.0809
3	35.5762	32.3006	1.1014	39.3136	36.7310	1.0703
4	41.4239	40.0029	1.0355	46.6402	45.2563	1.0306
5	49.4745	45.3404	1.0912	58.4706	55.2112	1.0590

max_parallel_workers = 100 の場合、同時にクエリの実行が開始された場合でも、すべてのクライアントがパラレルワーカーを確保できます。 したがって、すべてのクライアントでパラレルクエリが動作しますが、動作しているパラレルワーカー数が、物理CPUコア数を超過した場合、OSのタスク割当に従って動作することになり、本検証の場合では、クライアント数が多くなるに従って、性能は劣化することになります。 この場合、特にバージョン12ではJITコンパイルを有効としたほうが、性能は向上しています。 これは、JITコンパイルによって、処理を行うべき命令数が減少したからではないか、と考えます。

6.8. 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。 また、PostgreSQL 11においてJITコンパイルが実装されたことにより、PostgreSQLの処理性能向上への飽くなき挑戦の1つが、形として示されたものと思います。 実際に検証を行った結果、パラレルクエリを実用的に使用する際の注意点が判明し、JITコンパイル機能との併用もOLAP用途では有効であることが確認できました。

特に、バージョン12におけるパラレルクエリとJITコンパイルの併用の効果が大きく、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より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PostgreSQL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。

今回の検証内容には影響ありませんでしたが、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. おわりに

今年度(2019 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 8 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 8 期目となりました。

本 WG では昨年度にPGEConsのワーキンググループの位置づけを見直しから継続し、「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 3 つのテーマの検証結果を報告しました。

1. PostgreSQL 12および前バージョン11と比較したスケール性の検証
2. Pluggable Storage機能として開発中のAccess Method（zheap, zedstore）の検証
3. 強化されたパラレルクエリの性能及び、JITコンパイルとの併用に関する検証

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、バージョンが12となって更に強化されたパラレルクエリに加え、JITコンパイルの併用に関する検証が選定されました。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。今年度の検証では、今後の活用が期待されるPluggable Storage機能について開発中のAccess Method（zheap, zedstore）に対する検証が選定されました。

今年度も従来どおり、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、検証計画については定例の会合で議論を重ねました。また、会合を通じて各テーマの内容をWGメンバが共有することもできました。検証計画にそって約2週間から1ヶ月程度、実機上で検証・測定作業をします。作業で得られたデータは中間報告として会合に提出し、検証結果・測定結果の疑問点を元に必要に応じてフィードバックを実施することでより良い結果を得られるようにしました。

今年度のWGでは新たな試みとして、1つのWG内に議論が閉じないように、WG1/WG3の2つのWGの定例を合同で開催することで、別のWGからの視点からの意見を出してもらい議論を活性化するということも狙ってみました。この試みにより、WGの間を超えた議論ができるようになってきましたが、まだ、一部のメンバのコメントが多いという状況もあり、もう少し改善が必要かもしれません。このあたりは、後述のオンライン会議のやり方も含めて改善してきたいと考えています。

今年度の活動のもう一つの特徴として、コロナウィルス流行により、オフラインでの会合が困難となったことをうけ、Slackによるチャット会議や、遠隔会議システムを使ったオンラインでの打ち合わせを積極的に取り入れたことがあります。初めての試みであり、まだコミュニケーションの質として、オフラインの会議には及ばない部分もありましたが、おそらくは今後もこうしたやり方がスタンダードになっていくと思われます。次年度は、チャット会議やオンライン会議のやり方を更に良くしていく必要があると感じました。

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)

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 (2019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
	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	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	上原 一樹