PostgreSQLエンタープライズ・コンソーシアム 技術部会 WG#1

# 2016年度WG1活動報告書

# 目次

# 1. はじめに

## 1.1. 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。

### 1.1.1. 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 1.1 PGEConsにおける課題領域

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

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

## 1.2. 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 1.2 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 |

## 1.3. 本資料の概要と目的

本資料は、2016年9月にリリースされた PostgreSQL 9.6 のスケール性(参照系および更新系)、新技術・機能向上(パラレルクエリ、VACUUM改善)、および近年のPostgreSQLに関連する技術(JSON/JSONB、全文検索、Pgpool-IIによるスケールアウト)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。

### 1.3.1. 2016年度の活動テーマ

2016年度は、2016年9月29日にリリースされた PostgreSQL 9.6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。

- PostgreSQL 9.6のスケール性(参照系および更新系)
- PostgreSQL 9.6の新技術・機能向上
  - パラレルクエリ
  - VACUUM改善
  - JSON/JSONB
  - 全文検索
- Pgpool-IIのスケール性

以下、各検証テーマごとの概要を紹介します。

### 1.3.2. 定点観測(スケールアップ)

今年度の定点観測の検証では、CPUのコア数が96個の高性能サーバを使用して、PostgreSQL の最新バージョン 9.6 と その1つ前のバージョン 9.5 との性能傾向を検証しました。

その結果、参照系のクエリにおいて、9.6 は 9.5 に比べて最高スループットで約20%高いことが確認できました。

更新系のクエリにおいては、9.6 は 9.5 に比べて最高スループットが約30%高いことが確認できました。また、最高スループットを達成する同時接続数についても、9.5では96クライアントであったのに対し、9.6では192クライアントとなっており、並行処理の改善が図れていることを確認しました。

### 1.3.3. パラレルクエリ

PostgreSQL 9.6では、1つのクエリを複数のプロセスで実行するパラレルクエリ機能が実装されました。今年度は、以下のようなパラレルクエリの基本的な性能特性を検証しました。

- パラレルクエリが効果的に動作できる、行数が多い表の検索に対して、 並列度を高めたときにどの程度の性能向上が図れるかを検証した。並列度1での検索性能を基準として、並列度が40程度に対して約25倍の性能向上が見られた。
- 同じく行数が多い表をパーティショニングで分割した際の性能を検証した。
  - プルーニングできない検索では、並列度が4程度に増やすと約2.2倍の性能向上が見られた。
  - プルーニングが有効な検索では、並列度が4程度に増やすと約3.23倍の性能向上が見られた。
- 行数の多い表と少ない表との結合を含む問合せで、それぞれの表をスキャンするための並列度がどのように設定されるかを検討した。その結果、行数が多い表に対する並列度(ワーカ数) でどちらの表もスキャンされていることが確認された。

### 1.3.4. JSON/JSONB

JSON (JavaScript Object Notation)は、軽量のデータ交換フォーマットであり、多くのプログラミング言語でサポートされています [JSON_ORG] 。階層構造をもったデータを柔軟に記述できることから、関係データベースでは扱いにくい半構造データ [semi_structure] の処理に向いています。

PostgreSQLでは、バージョン9.2から JSON データ型を扱えるようになりました。更に、バージョン9.4からは JSON データをバイナリ形式で格納することで処理の高速化を図った JSONB が追加されました。

ここでは、近年注目されている IoT (Internet of Things) 分野への応用で、センサ機器が半構造データを出力するユースケースを想定して、PostgreSQLの JSON/JSONB データ型の有効な利用法を検討しました。

その結果、以下のようなことが分かりました。

- JSON/JSONB 型を使用すると、半構造データをそのまま格納できるという利点がある反面、通常のテキスト型にデータを格納する場合に比べて、必要な容量が大きくなり、検索時間も長くなるというトレードオフの関係が見られた。
  - データサイズについては、通常のテキストとしてJSONデータを格納する場合に比べて、JSON型を使うと 2.83倍、JSONB 型では 3.70 倍の容量が必要となる。
  - 検索に要する時間は、通常のテキストとして JSON データを格納する場合に比べて、JSON型を使うと 9.3〜28.6 倍、JSONB 型では 2.70 〜 3.71 倍の時間を必要とする。
- 検索時間については、定型的な検索(たとえば、データを取得した時間帯による絞り込み) を行う場合、該当するデータ項目だけをカラムとして格納することで通常のデータ型(たとえば timestamp)に格納する場合とそん色のない性能となる。

### 1.3.5. VACUUM改善

PostgreSQL 9.6 では、データベースの表にあるコミット済みデータを「凍結」する VACUUM FREEZE コマンドにて、事前にマークされた FREEZE 済みのページをスキップする機能が追加されました。VACUUM FREEZEに要する時間がどの程度短縮されるのか、検証しました。

その結果、PostgreSQL 9.5 では FREEZE するタプルの多少にかかわらず、表の大きさに比例する時間を要していた VACUUM FREEZE が、9.6 では FREEZE するタプル数が少ない場合には、表の大きさとは無関係にほぼ一定の時間で完了することが確認されました。

### 1.3.6. 全文検索

PostgreSQLの適用領域の拡大、また取り扱うデータ量の増加に伴い、 PostgreSQLに格納されたデータに対する全文検索の重要性がますます高まっています。

PostgreSQLの標準機能においても、LIKE文を用いることで文字列パターンを指定した検索は可能ですが、性能の観点からは全文検索専用のインデックスを使うことが望ましいと考えられます。

本章では、特に国内においてニーズの高い"日本語検索"に対応した代表的な検索手法である pg_trgm（ピージートリグラム）、pg_bigm（ピージーバイグラム）、PGroonga（ピージールンガ）の特性を検証によって明らかにしました。

その結果、以下のようなことが分かりました。

- 今回取り上げた検索手法では、インデックスを使って検索結果の候補をまず探し、次いでその候補が検索結果として適切かを Recheck 処理で確認する。
  - 検索に要する時間は、インデックスによる候補の検索と、Recheck処理の合計となる。
  - 検索キーワードによって、候補の検索に要する時間と Recheck 処理の時間の割合は大きく異なる。
- 今回の検証では、検索キーワードが日本語 * 日本、東京都、データベース など) の場合、pg_bigm が多くのキーワードで最も高速であった。一方、検索キーワードが英語 (npm、Ansible、PostgreSQL など）の場合、pg_bigm が最も遅く、pg_trgm, PGroonga の方が多くの場合に高速であった。

### 1.3.7. Pgpool-II検証

PGECons では2012年度の検証テーマとして Pgpool-II を用いた参照負荷を分散するクラスタでの性能検証を実施しました。この時には、データのレプリケーションは Pgpool-II によって行いました。今年度はそれを受けて、最新 Pgpool-II と PostgreSQL とを用いて PostgreSQL 側でレプリケーションを実施するケースを加えて、性能の比較検証を実施して、以下のような結果を得ました。

- 参照性能に関しては、負荷分散の方式によらずほぼ同じ性能(スループット)となる
  - どの方式の場合でも同じように、台数に比例して参照処理性能が向上する
- 更新性能に関しては台数を増やしても性能は向上せず、むしろ低下する。その度合いはレプリケーション方式によって性能が異なる。
  - 低下の度合いが最も小さいのは、PostgreSQL が提供する非同期レプリケーションを使うケース。ただし、プライマリとスタンバイの間でレプリケーション遅延によるデータの不一致がある
  - 次いで小さいのは、PostgreSQL が提供する同期レプリケーションを使うケース。この場合、プライマリとスタンバイ間のデータの不一致は生じない
  - 性能低下の度合いが最も大きいのは、Pgpoo-II によるネイティブレプリケーションを使うケース。この構成ならでは利点としては、異なるバージョンの PostgreSQL サーバにデータをレプリケートできる、システム全体の構成を柔軟にできることがある

## 1.4. 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。

### 1.4.1. 2015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。

## 1.5. 実施体制

2016年7月14日に開催された2016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 1.3 2016年度WG1参加企業

| |
|---|
| SRA OSS, Inc. 日本支社 |
| NTTテクノクロス株式会社 |
| 日本電気株式会社 |
| 日本電信電話株式会社 |
| 日本ヒューレット・パッカード株式会社 |
| 富士通株式会社 |

この中で、日本電気株式会社は、「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。

## 1.6. 実施スケジュール

2016年度は、下記スケジュールで活動しました。

表 1.4 実施スケジュール

| 活動概要 | スケジュール |
|---|---|
| WG1スタート | 2016年7月14日 |
| 実施計画策定 | 2016年7月～10月 |
| 検証実施 | 2016年11月～2017年1月 |
| 2016年度WG1活動報告書作成 | 2017年2月～2017年3月 |
| 総会と成果報告会 | 2017年5月18日(東京)、2017年6月2日(大阪/報告会のみ実施) |

## 1.7. 謝辞

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

## 1.8. 参考文献

[JSON_ORG] JSONの紹介. http://www.json.org/json-ja.html 2017.4.4 閲覧

[semi_structure] Semi-structered data. https://en.wikipedia.org/wiki/Semi-structured_data 2017.4.4 閲覧

# 2. 定点観測(スケールアップ検証・参照系)

## 2.1. 検証概要

本年度も例年行っているスケールアップ参照性能を観点とした定点観測検証を行いました。(昨年の2015年度報告は こちら です。) 2016年度は96コアのCPU(インテル Xeon E7-8890 v4 2.2GHz)、メモリ2TBといった例年通りのハイエンドスペックのサーバで、最新のPostgreSQLバージョン 9.6 と前バージョンの 9.5 とのスケールアップ参照性能の比較を行いました。

## 2.2. pgbench とは

本検証では、pgbench というベンチマークツールを使用しました。

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

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

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

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

以下に、各スケールファクタに対応するデータベースサイズを示します。

表 2.1 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 2.2 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 2.3 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 2.4 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 のマニュアル をご覧ください。ここでは、本レポートで 使用している主なオプションのみを説明します。

ベンチマークテーブル初期化

```
-i ベンチマークテーブルの初期化を行います。
-s スケールファクタを数字 (1 以上の整数) で指定します。
```

ベンチマークの実行

```
-c 同時に接続するクライアントの数
-j pgbench 内のワーカスレッド数
-T ベンチマークを実行する時間を秒数で指定
```

前述のように、pgbench ではカスタムスクリプトを作成することで、独自の SQL でベンチマークを実行することができま す。ここで、本検証で利用した機能を簡単に説明します。

set 文で変数に値を設定可能です。以下の例では変数 row_count に 10000 を代入しています。

```
\set row_count 10000
```

また set 文では四則演算が利用可能です。以下の例ではスケールファクタの 100000 倍の値を「naccounts」に設 定しています。ここで「:scale」は-s オプションで指定したスケールファクタの値で置き換えられます。

```
\set naccounts 100000 * :scale
```

変数には乱数を用いることも可能です。以下の例では変数 aid に 1 から aid_max の間の乱数を代入します。

```
\setrandom aid 1 :aid_max
```

設定した変数は、以下のようにスクリプト中の SQL 文から参照できます。

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

## 2.3. 検証構成

### 2.3.1. ハードウェア構成

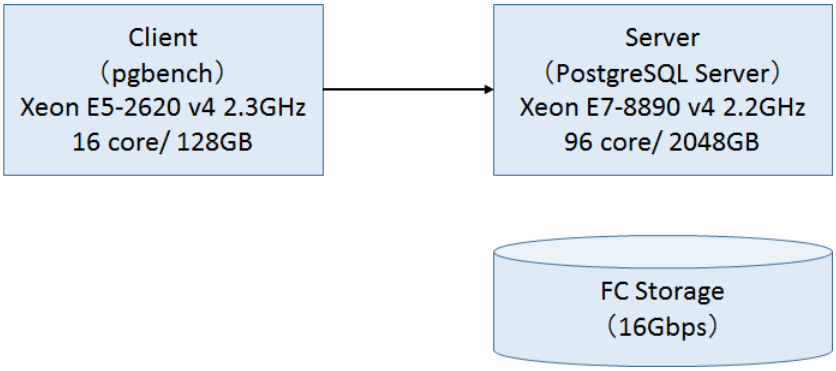



図 2.1 検証ハードウェア構成

### 2.3.2. ソフトウェア構成

検証環境のソフトウェア構成を示します。

表 2.5 検証用サーバ

| | |
|---|---|
| OS | Red Hat Enterprise Linux 7.2 |
| PostgreSQL | 9.5.5, 9.6.1 |

表 2.6 負荷かけ用サーバ

| | |
|---|---|
| OS | Red Hat Enterprise Linux 7.2 |
| pgbench | 9.6.1 |

### 2.3.3. カーネル設定

- OS デフォルト
- ハイパースレッディングは利用しない

### 2.3.4. PostgreSQL 設定

PostgreSQL設定ファイル postgresql.conf での設定は以下の通りです。

```
listen_addresses = '*' ... 負荷マシンからの接続用
max_connections = 510 ... 多めに設定
shared_buffers = 200GB ... メモリ 2TB の 1/10
work_mem = 1GB
checkpoint_timeout = 30min
max_wal_size = 1GB
logging_collector = on
logline_prefix = '%t [%p-%l] '
```

ただし、パラメータ wal_level は 9.6 から以前の archive, hot_standby 指定が統合されて replica で指定するようになったため、

9.5では

```
wal_level = archive
```

9.6では

```
wal_level = replica
```

を指定しています。

## 2.4. 検証方法

### 2.4.1. 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 -w start
$ createdb [dbname]
```

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1000で初期化します。

```
$ pgbench -i -s 1000 [dbname]
```

### 2.4.2. 測定

本検証では pg_prewarm モジュールを用います。

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

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

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

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

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

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

これを、クライアント用検証機から

```
$ pgbench -n -h [host] -p [port] -c [clients] -j [threads] -f custom.sql -T 300 -s 1000 [dbname]
```

として実行しました。SELECTのみであるためVACUUMを実行せず、pgbench クライアント数とスレッド数を変動させながら、300秒ずつ実行しています。スレッド数はクライアント数の半分としています。スケールファクタにはデータベース初期化時と同じ1000を指定します。

計測はクライアント数ごとにそれぞれ3回ずつ実行し、その中央値を結果とします。また、変動させるクライアント 数は{1, 2, 4, 8, 16, 32, 48, 64, 80, 96, 128, 160, 192}です。

## 2.5. 検証結果

検証結果のTPSおよびCPU使用率のグラフを以下に示します。9.5, 9.6 はともにクライアント数増加につれてTPSが増加し、コア数(96)近くのクライアント数128で96コアのCPUを使い切った状態でTPSは頭打ちとなっていますが、9.6のTPSは各クライアント数において常に9.5を上回る結果となりました（レイテンシにおいてもより小さい値となっています）。

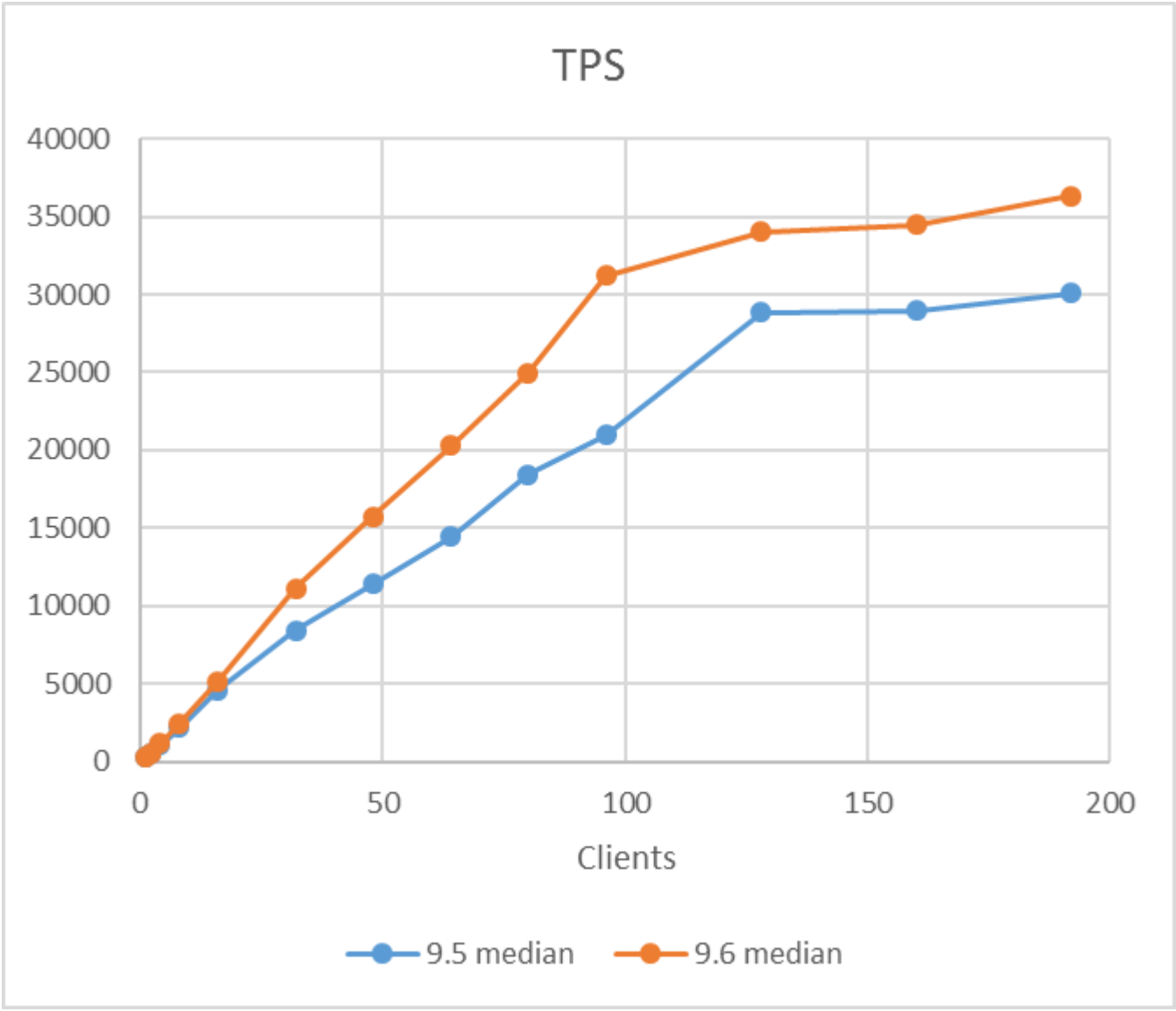



図 2.2 各クライアント数に対する TPS

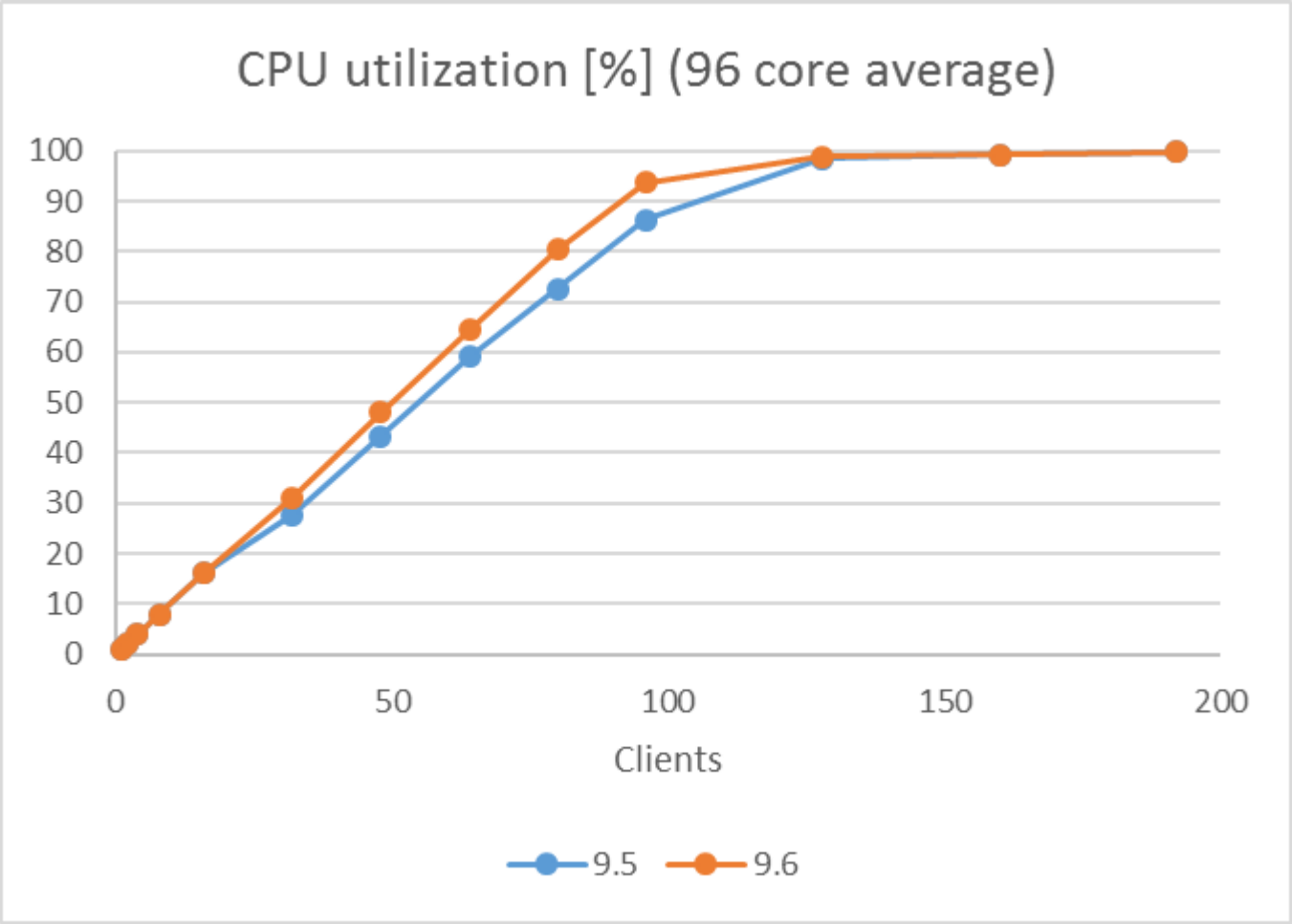

CPU utilization [%] (96 core average)
100
90
80
70
60
50
40
30
20
10
0
0
50
100
150
200
Clients
9.5
9.6

図 2.3 CPU使用率

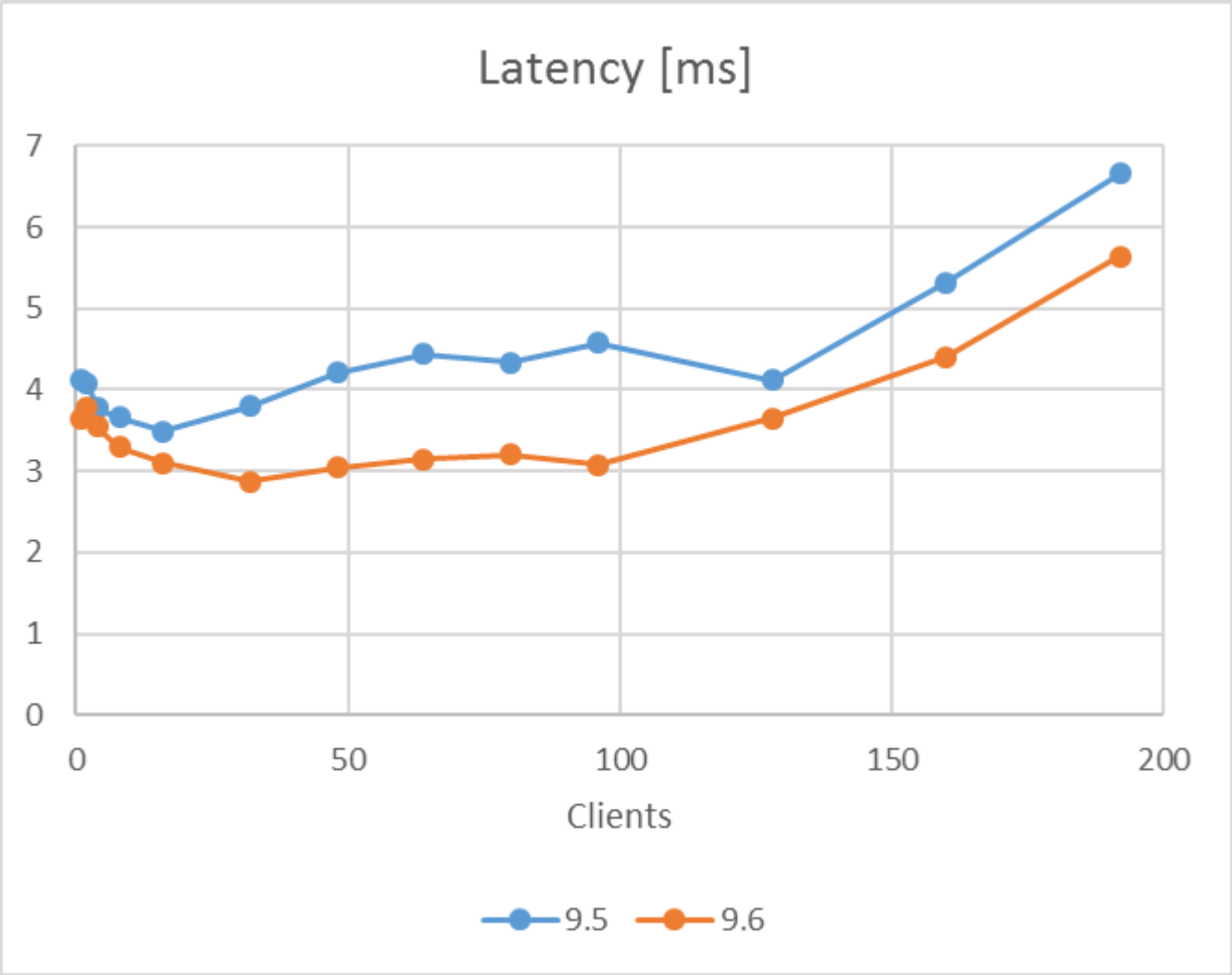



図 2.4 各クライアント数に対するレイテンシ

## 2.6. 考察

本検証では例年通り、PostgreSQL最新バージョンと前バージョンのスケールアップ参照性能比較検証を行いました。昨年度のバージョン 9.5 においてもメニーコアCPUにおけるスケールアップの性能向上は確認されておりましたが、今年度のバージョン 9.6 は更にそれが顕著でした。全てのクライアント数において9.5を上回り、かつ最大で1.5倍近くもの性能向上を示していました。

今回詳しい調査に至りませんでしたが、リリースノートではスケーラビリティの大きな改善に「共有ハッシュテーブルのフリーリスト分割による競合回避の修正」が寄与していることが触れられており、本検証結果と関係している可能性があります。本検証では9.6における性能向上は確かに確認されており、PostgreSQL をメニーコアCPUサーバで用いるメリットがここに至りさらに増してきたと言えるでしょう。

参考:

- PostgreSQL 9.6 リリースノート (日本語) http://www.sraoss.co.jp/technology/postgresql/9.6/
- PostgreSQL 9.6 リリースノート (英語) https://www.postgresql.org/docs/9.6/static/release-9-6.html
- 「共有ハッシュテーブルのフリーリストの分割」 メーリングリストでのディスカッション https://www.postgresql.org/message-id/flat/20151211170001.78ded9d7@fujitsu#20151211170001.78ded9d7@fujitsu
- 「共有ハッシュテーブルのフリーリストの分割」 コミットログ (英語) https://github.com/postgres/postgres/commit/44ca4022f3f9297bab5cbffdd97973dbba1879ed

# 3. 定点観測(スケールアップ検証・更新系)

## 3.1. 検証目的

PGEConsでは、PostgreSQLの新バージョン・新リリースにあわせて、新旧バージョンの性能比較やスケールアップ特性の検証を主な目的にした定点観測を2012年度から実施してきました。2014年度からは、それまでの参照処理に加えて更新処理についても検証を実施し、検証結果の公開を行いました。

今年度はその定点観測として継続して実施している 1)PostgreSQLの更新処理におけるCPUスケーラビリティの達成状況確認、2)新旧バージョン(今年度はPostgreSQL 9.6.1と9.5.5)の比較による更新性能の改善状況確認を行いました。

## 3.2. 検証構成

本検証のハードウェアおよびソフトウェアの主なスペックと構成を 表. 3.1 に示します。

表 3.1 検証構成

| 機器 | 項目 | 仕様 |
|---|---|---|
| DBサーバ | CPU | インテル Xeonプロセッサ E7-8890v4@2.20GHz<br>(24コア)x4 合計 96コア |
| | 搭載メモリ | 2048GB |
| | 内蔵ストレージ | HDD 1.2TB SAS 10K x 10 |
| | DB格納用ストレージ | Fiber Channel接続(16Gbps) SAN<br>以下の2領域を利用<br>領域1: PostgreSQLのDB領域<br>領域2: WAL領域<br>それぞれの領域はHDD 1.8TB 6G SAS 10K x 4<br>(RAID10、実効容量約3TB)で構成 |
| | OS | Red Hat Enterprise Linux 7.2 |
| | DBMS | PostgreSQL 9.6.1<br>PostgreSQL 9.5.5 |
| 負荷掛けクライアント | CPU | インテル Xeonプロセッサ E5-2620v4@2.30GHz<br>(8コア)x2 合計 16コア |
| | 搭載メモリ | 128GB |
| | 内蔵ストレージ | HDD 1.2TB SAS 10K x 8 |
| | DB格納用ストレージ | なし(内蔵ストレージのみ) |
| | OS | Red Hat Enterprise Linux 7.2 |
| | クライアント | PostgreSQL 9.6.1のソースコードに含まれるpgbenchをビルドして使用 |

## 3.3. 検証方法

### 3.3.1. 環境構築

#### 3.3.1.1. データベース初期設定

スケールファクタは7000、FILLFACTORは80として初期設定(テーブル作成)を行いました。これにより対象テーブルのサイズは約100GBとなります。FILLFACTORを80としたのは、UPDATE文実行時にHOT機能を働かせるためです。

```
$ pgbench -i -s 7000 -F 80
```

測定時間を短縮するために、テーブルのデータ($PGDATAディレクトリ以下のファイル群)は、測定を行うたびにpgbench -iで作成するのではなく、上記の手順で作成したテーブルのデータを保存しておき、それを毎回$PGDATAに戻す方法で用意することにしました。

#### 3.3.1.2. postgresql.confの設定値

postgresql.confにより設定するパラメータのうち、いくつかをデフォルトから変更して測定を実施しました。主な目的はCHECKPOINTによる影響を排除しできるだけ測定状況を均一化する、VACUUMの影響を排除する、本番システムを想定したwal_levelとすることです。shared_buffersは検証ハードウェア(DBサーバ)が搭載するメモリ量に合わせて設定しました。また、max_wal_sizeを大きな値にすることで、測定期間中にCHECKPOINT処理が実行されないようにしました。

リスト 3.1 postgresql.confのデフォルトからの変更内容

```
max_connections = 1000
shared_buffers = 384GB    # サーバ搭載容量の約20%
wal_level = replica # PostgreSQL 9.6
wal_level = archive # PostgreSQL 9.5
max_wal_size = 160GB
checkpoint_timeout = 60min
maintenance_work_mem = 20GB
log_checkpoints = on
logging_collector = on
log_line_prefix = '%t %p %a'
log_lock_waits = on
autovacuum = off
listen_addresses = '*'
```

### 3.3.2. 測定

#### 3.3.2.1. pgbenchスクリプト(測定対象トランザクション)

PostgreSQLで実行させるトランザクションは、例年度通りに、サイズの大きい表(pgbench_accounts)からランダムに選んだ行に対する比較的単純な更新処理(UPDATE)としました。

リスト 3.2 pgbenchのカスタムスクリプト

```
\set naccounts 1000000 * :scale
\set aid_val random(1, :naccounts)
UPDATE pgbench_accounts
  SET filler=repeat(md5(current_timestamp::text), 2)
  WHERE aid=:aid_val;
```

#### 3.3.2.2. pgbenchの実行方法

システムの定常状態に近い動作で検証を行うため、pg_prewarmを利用して、テーブルの全データをshared_bufferにロードした状態で測定を開始するようにしました。全データをshared_bufferにロードした後、PostgreSQLを動作させたままで3回pgbenchを5分間実行し、中央値を測定結果として用いることにしました。測定実施中(pgbench実行中)にCHECKPOINTが起動しないことを確実にするために、各pgbench実行の間でcheckpointコマンドを実行させました。これにより、次のCHECKPOINTが起動するまでのWAL量(のカウンタ)がリセットさせます。

pgbenchを実行させるコマンドを以下に示します。

```
pgbench -c [clients] -j [threads] -f [script] -s 7000 [dbname]

pgbench    実行コマンド

[clients] クライアント数
[threads] スレッド数 (クライアント数の1/2)
[script]  スクリプトファイル名
[dbname]  pgbench表を作成したDB名
```

昨年度までに行ってきた参照系や更新系の検証を踏襲し、pgbenchのスレッド数はクライアント数の半分としています。

#### 3.3.2.3. 測定結果の取得

スループット値はクライアント用検証機で実行させるpgbenchプログラムが実行終了時に出力します。ネットワーク接続操作を含むスループット値と含まない値の2種類が出力されますが、本報告書では、ネットワーク接続操作を含まない値(excluding connections establishing)を用いています。ここで行った性能測定では、各実行条件(測定パラメータ)について3回行ったpgbench実行から中央値を選び、それを最終的な測定結果としました。CPU使用率は、pgbenchを実行させている5分の間、sarコマンドをサーバ用検証機で動作させて計測しました。

### 3.3.3. 測定パラメータ

性能測定では、pgbenchのクライアント数(スレッド数)を変動させて性能(スループットやCPU使用率)がどのように変化するのかを調べることを基本としました。これ以降、クライアント数の変化に伴うスループットおよびCPU使用率の変化を「基本性能特性」と呼びます。この他に変動させた測定パラメータは、CPUコア数、PostgreSQLのバージョン(9.5.5と9.6.1)です。CPUコア数については、1CPUチップ(ソケット)が24コアであることを考慮し、24, 48, 96の3種類について測定を行いました。今年度使用したOSのRed Hat Enterprise Linux 7.2では、rebootすることなく動的にCPUコアの使用状態(online, offline)を切り換える機能が提供されています。そこで、CPUコア数の変更は、この機能を使用する下記スクリプトにより行いました。

リスト 3.3 コア数変更スクリプト

```
#!/bin/bash
cd /sys/devices/system/cpu
CHIP0=' 0 1 2 3 4 5 6 7 8 9 10 11 48 49 50 51 52 53 54 55 56 57 58 59 '
CHIP1=' 12 13 14 15 16 17 18 19 20 21 22 23 60 61 62 63 64 65 66 67 68 69 70 71 '
CHIP2=' 24 25 26 27 28 29 30 31 32 33 34 35 72 73 74 75 76 77 78 79 80 81 82 83 '
CHIP3=' 36 37 38 39 40 41 42 43 44 45 46 47 84 85 86 87 88 89 90 91 92 93 94 95 '
case ${1:-96} in
 96) cpuON="${CHIP0} ${CHIP1} ${CHIP2} ${CHIP3}"
 cpuOFF="";;
 72) cpuON="${CHIP0} ${CHIP1} ${CHIP2}"
 cpuOFF="${CHIP3}";;
 48) cpuON="${CHIP0} ${CHIP1}"
 cpuOFF="${CHIP2} ${CHIP3}";;
 24) cpuON="${CHIP0}"
 cpuOFF="${CHIP1} ${CHIP2} ${CHIP3}";;
 *) cpuON="${CHIP0} ${CHIP1} ${CHIP2} ${CHIP3}"
 cpuOFF="";;
esac
for n in ${cpuON}
do
 echo 1 > cpu${n}/online
done
for n in ${cpuOFF}
do
 echo 0 > cpu${n}/online
done
```

## 3.4. 検証結果

### 3.4.1. CPUスケーラビリティ

基本性能特性を 図. 3.1 と 図. 3.2 に示します。9.5.5では最大のスループットとなったのは48コアのケースで、96コアのケースではスループットが少々下がるがほぼ同じ傾向となりました。どちらもピークとなったクライアント数は96でした。一方、9.6.1では傾向が大きく異なり、今回の検証で最大のクライアント数である192が最大のスループットとなっており、48コアも96コアもクライアント数を更に増やすことでスループットの向上が望めるような結果となりました。24コアと96コアを比較すると、最大で約60%のスループット向上が確認できました。コア数が24の場合は9.5.5、9.6.1ともに、64接続数以降、横ばいとなる結果となりました。

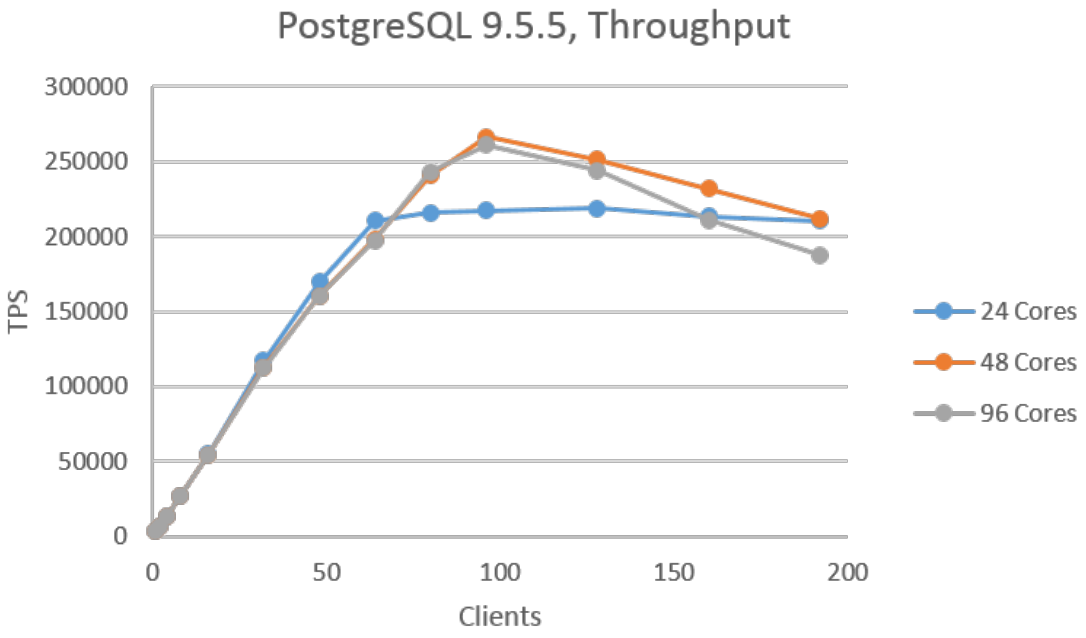



図 3.1 PostgreSQL 9.5.5のスループット

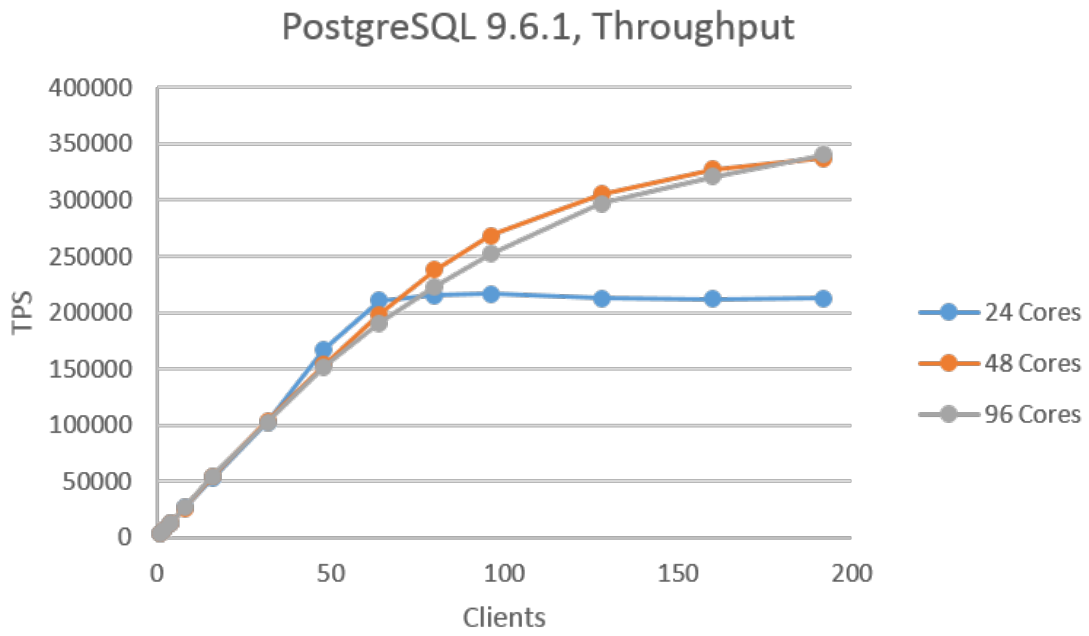



図 3.2 PostgreSQL 9.6.1のスループット

### 3.4.2. バージョン比較

図. 3.3 に24コア、図. 3.4 に48コア、図. 3.5 に96コアの結果を示します。24コアの場合は9.6.1がわずかにスループットが下がりますが、ほぼ同じ性能を示しました。48コア、96コアの場合は、クライアント数が少ない場合は9.5.5が若干スループットが高く、クライアント数が96を過ぎたあたりから9.6.1が9.5.5に比べて高いスループットを示す結果が得られました。今回の検証では、9.6.1は9.5.5よりも最大で約30%のスループット向上が確認できました。

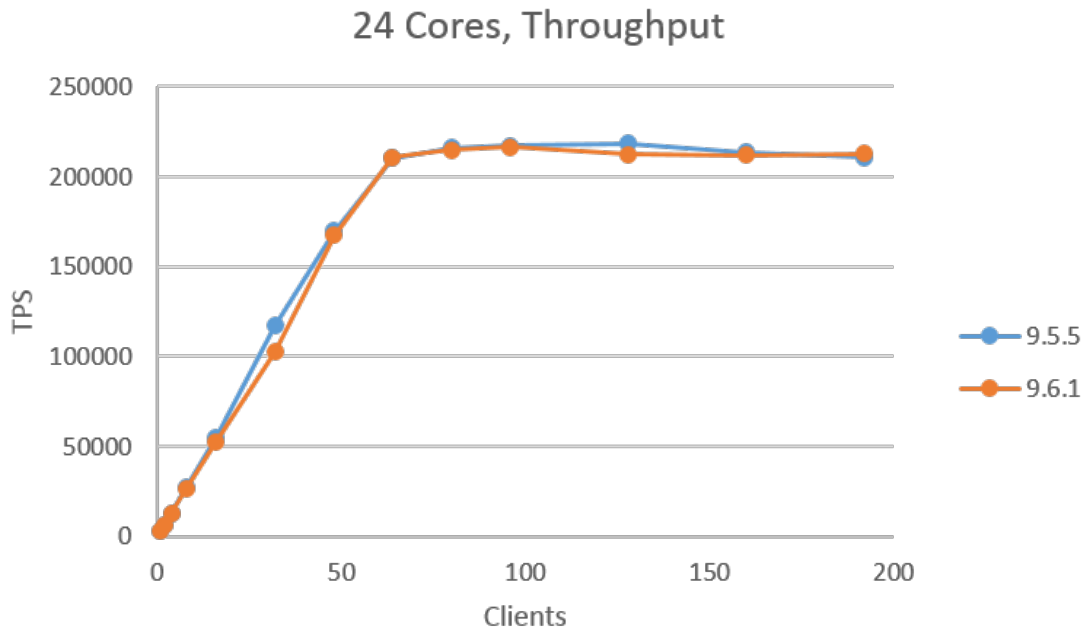



図 3.3 24コアでのスループット

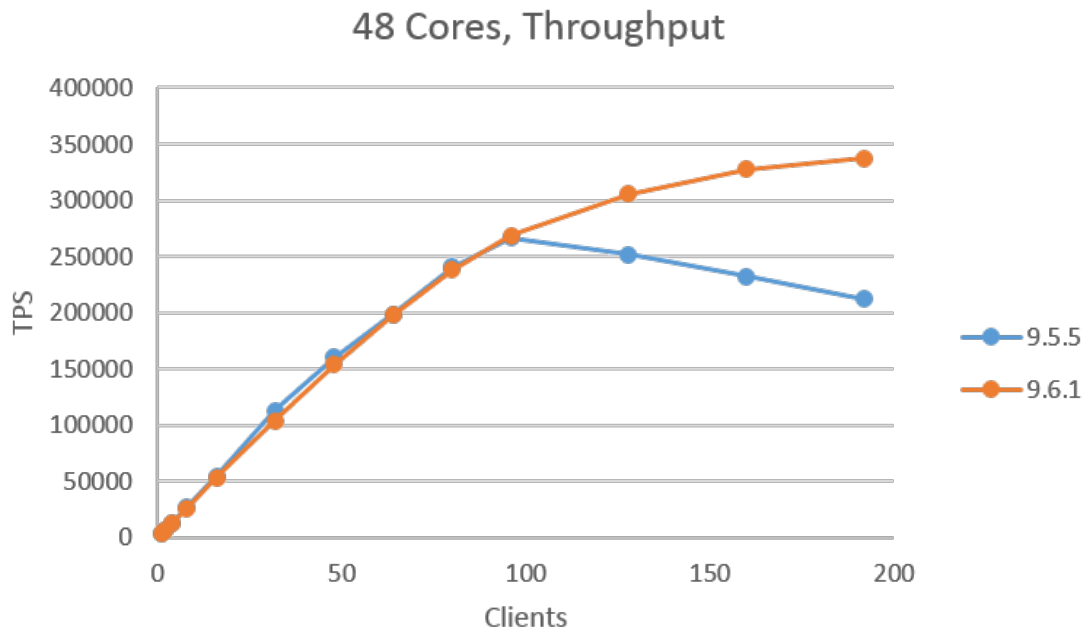



図 3.4 48コアでのスループット

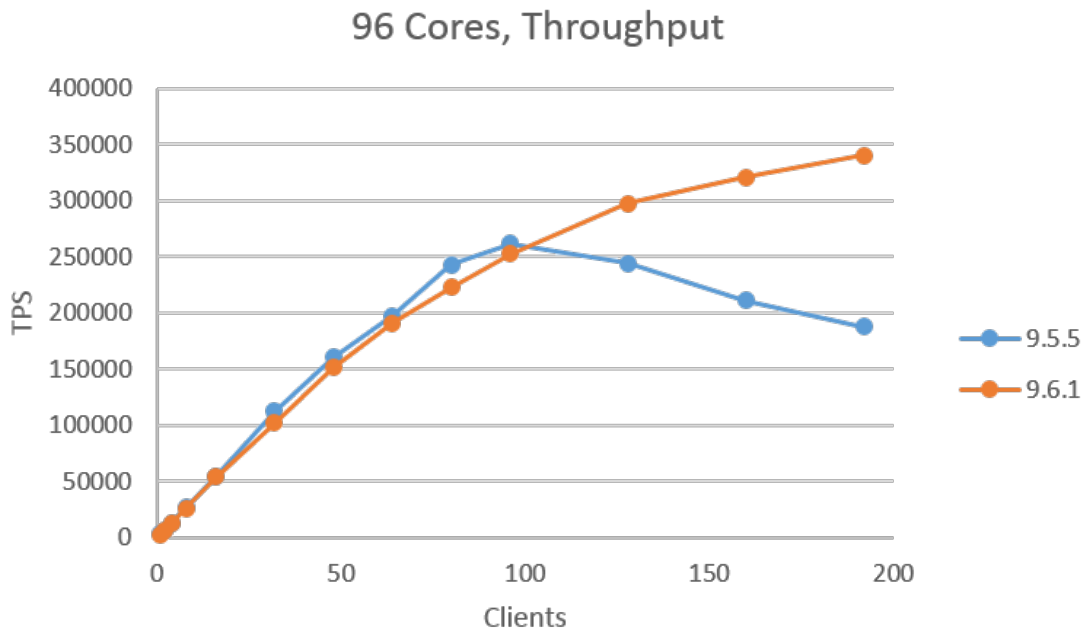



図 3.5 96コアでのスループット

## 3.5. 考察

本検証ではクライアント数が多い場合に、バージョン9.6が9.5と比較して更新性能が向上したことが確認されました。特にバージョン9.6ではCPUコア数の倍の192までクライアント数を増やしても、性能が劣化は発生しませんでした。

CPUコア数96、クライアント数192でのCPU使用率について 図. 3.6 に示します。9.6.1は9.5.5と比べてCPU使用率が高く、CPUをより効率よく利用できていることがわかります。このことが性能差に影響していると考えられます。

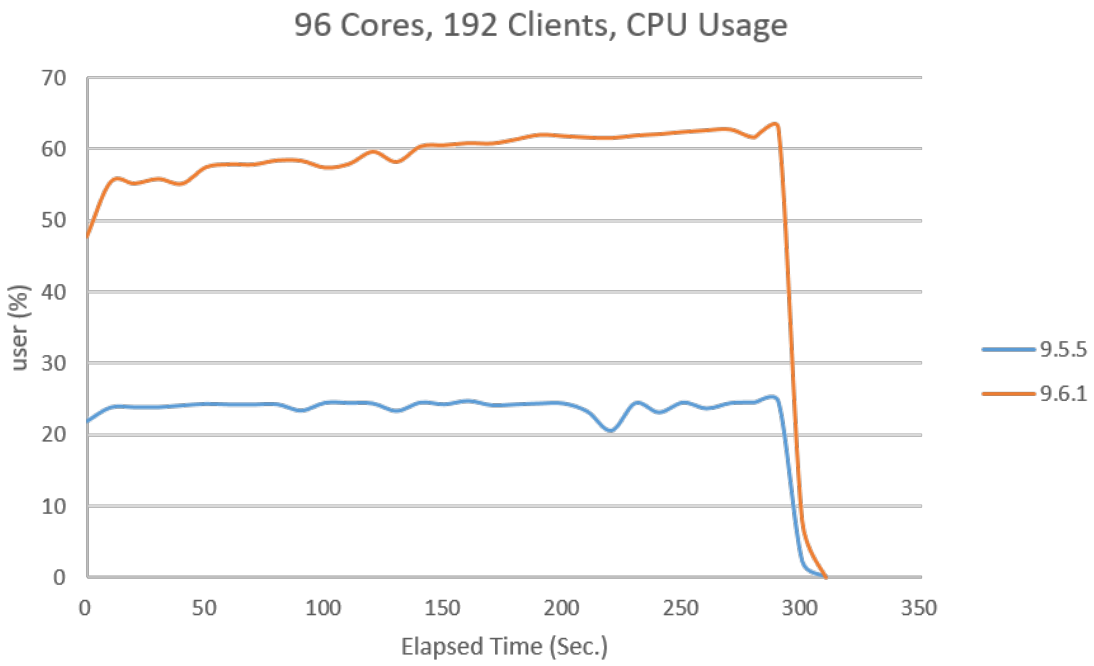



図 3.6 96コアでのスループット

この原因として、PostgreSQL 9.6でProcArrayLock、ClogControlLockの競合の改善が行われたことが影響していると考えられます。ProcArrayLockは複数のプロセスが同時にコミットするときに利用されるロックで、9.5までは複数のプロセスが同時にコミットする場合、それぞれのプロセスでロック競合の解決を行っていました。9.6からは、最初にロックを獲得したプロセスがほかのプロセス分の処理も行うことで、性能改善を図っています。ClogControlLockは、トランザクションの状態を読み書きする際に利用されるロックで、9.6でバッファサイズを拡張することにより、メニーコアに対して性能の改善を図っています。

なお、今回の検証は非常に高性能なストレージを用いて、I/Oボトルネックになりにくい環境であったため、有効なスケーラビリティが確認されましたが、ストレージの性能によっては異なる傾向になる可能性が高いことにご留意いただきたいと思います。

# 4. パラレルクエリ

## 4.1. 検証概要

近年、ハードウェアの性能向上はめざましく、一つのプロセッサに搭載されるコアの数とメモリの容量は年々増大しています。

メニーコアについては、代表的な2ソケット対応のBroadwellマイクロアーキテクチャ インテル Xeon E5-2699 v4では、1プロセッサあたり22コア、2プロセッサ構成では44コアとなります。4ソケット対応のBroadwellマイクロアーキテクチャのインテル Xeon E7-8890 v4に至っては1プロセッサあたり24コア、4プロセッサ構成では96コアにもなります。またメモリの大容量化も進みハイエンドサーバではTB(テラバイト)単位でのメモリ搭載がされています。

PostgreSQL 9.5までは、メニーコア環境であっても1つのクエリでは1つのコアしか使用されませんでした。商用ソフトウェアでは以前から実装されていて、PostgreSQLでも長年実装が待ち望まれてきたメニーコアや大容量メモリを有効活用するパラレルクエリが、ついにバージョン9.6で実装されました。

そこで、パラレルクエリにより複数のコアをどのように利用して性能向上を果たすのかを検証しました。

## 4.2. パラレルクエリとは

### 4.2.1. 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、メニーコアの利点を有効活用する処理方式です。

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。それに対して検索で使用する10プロセスあって複数のプロセス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。

### 4.2.2. PostgreSQL9.6でのパラレルクエリ

PostgreSQL 9.6で実装されているパラレルクエリに関する処理は、パラレルスキャン、パラレルジョイン、パラレル集約の三種類です。

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

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

```
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------
-----------------------------------
Gather  (cost=1000.00..112816.10 rows=38400 width=52) (actual time=0.198..224.017 rows=99740 loops=
1)
  Output: log_id, date, "time", product_id, place_id, machine_id, app_id, access_time_second, acces
s_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 tim
e=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 4.1 テーブルサイズによるワーカー数

| ブロック数 | 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 |
| ... | ... | ... | ... |

※ 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。

### 4.2.3. 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。

表 4.2 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 | 説明 |
|---|---|---|
| max_worker_processes | 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 | 0 | 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。 |
| 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。 |
| parallel_setup_cost | 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 | 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。 |

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

## 4.3. 検証構成

### 4.3.1. 検証ハードウェア構成

検証環境のハードウェア構成の概略を示します。

本パラレルクエリ検証ではDBサーバ上でSQLを直接実行する形式としました。

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

表 4.3 検証用DBサーバ構成

| 機器 | 項目 | 仕様 |
|---|---|---|
| 2TBメモリサーバ | CPU | インテル Xeon E7-8890v4@2.20GHz<br>24コアx4 合計 96コア<br>ハイパースレッディング:off |
| | 搭載メモリ | 2048GB |
| | 内蔵ストレージ | HDD 1.2TB SAS 10K x 10 |
| | DB格納用ストレージ | Fiber Channel接続(16Gbps) SAN<br>領域はHDD 1.8TB 6G SAS 10K x 4<br>RAID10、実効容量約3TBで構成 |
| 256GBメモリサーバ | CPU | インテル Xeon E5-2690v4@2.60GHz<br>14コアx2 合計 28コア<br>ハイパースレッディング:off |
| | 搭載メモリ | 256GB |
| | 内蔵ストレージ | HDD 1.2TB SAS 10K x 16 |
| | DB格納用ストレージ | Fiber Channel接続(16Gbps) SAN<br>領域はHDD 1.8TB 6G SAS 10K x 4<br>RAID10、実効容量約3TBで構成 |

### 4.3.2. 検証ソフトウェア構成

検証環境のソフトウェア構成を示します。

表 4.4 ソフトウェア構成

| 種類 | ソフトウェア名およびバージョン |
|---|---|
| OS | Red Hat Enterprise Linux 7.2 |
| Database | PostgreSQL 9.6.1 |

## 4.4. 検証方法

### 4.4.1. 検証項目

パラレルクエリ検証では以下の三点を検証します。

#### 4.4.1.1. 検証A

検証Aでは、パラレルクエリの並列度による処理性能の改善状況の確認をします。

パラレルクエリ適用が有効と考えられるある程度の大きさを持ったテーブルに対し、max_parallel_workers_per_gatherの値を変更し、並列度の違いによる処理時間を比較します。

テーブルサイズから計算されるデフォルトの並列度まで性能向上するのか、あるいはそれを超える設定にした場合に効果があるのか、大きすぎるとオーバーヘッドにより性能低下となるのかという点を明らかにします。

#### 4.4.1.2. 検証B

検証Bでは、パーティションテーブルでのパーティション絞り込み（パーティションプルーニング）とパラレルクエリとの比較を行います。

パーティションテーブルの場合、並列度は個々のパーティションテーブルのサイズにより計算されるため、単一のテーブルよりも並列度が小さくなります。巨大なサイズの単一テーブルで並列度を大きくした場合と、パーティション化して検索対象パーティションが絞り込まれる場合とでの処理時間を比較します。

パーティション化されたテーブルとパラレルクエリとではどちらが速いのか、パーティションプルーニングが有効なSQLと無効なSQLについて測定します。

#### 4.4.1.3. 検証C

検証Cでは、複数テーブルを結合する場合のパラレルクエリの動作を確認します。

起動されるワーカー数が異なるテーブルを結合させた場合、実際に起動されるワーカー数を確認します。

サイズの大きいテーブルでは起動されるワーカー数が多く、サイズが小さいテーブルでは起動されるワーカー数は少なくなります。このサイズが異なる二つのテーブル、すなわちデフォルトの並列度が異なるテーブル同士を結合させた場合、どのような並列度で処理されるのかを確認します。

### 4.4.2. データモデル

本検証で使用するデータモデルは、2013年度のパーティション性能検証および、2015年度のBRINとパーティション比較検証で使用したモデルを使用します。具体的には以下の2テーブルを使用します。

- access_logテーブル：1日当たり768万行のデータを保持するテーブル。通常テーブルとパーティションテーブルで性能を比較します。
- product_masterテーブル：8MB(パラレルクエリが行われる程度の大きさ)のマスタテーブル。

テーブル定義は以下のとおりです。

access_log テーブル

```
create table access_log (
  log_id             bigserial,
  date               char(8),
  time               char(6),
  product_id         int,
  place_id           int,
  machine_id         int,
  app_id             int,
  access_time_second int,
  access_count       int,
  err_code           int
);
```

product_master テーブル

```
create table product_master (
  product_id         int primary key,
  product_name       text
);
```

access_logテーブルは、単一テーブルとパーティションテーブルの比較を行うため、単一テーブルのaccess_logテーブルを保持するデータベースと、パーティションテーブルのaccless_logテーブルを保持するデータベースを用意し、それぞれのデータベースに access_log テーブル、product_masterテーブルを配置します。

表 4.5 各データベースとaccess_logテーブル

| No | データベース | 説明 | データサイズ(MB) | 最大並列度 |
|---|---|---|---|---|
| 1 | base366day | 1日あたり768万行×366日（28億1088万レコード）を単一テーブルとして保持するaccess_logテーブルのあるDB | 226,391 | 11 |
| 2 | part366day | 1日あたり768万行を1パーティションに保持し、366日分のパーティション（合計28億1088万レコード）で構成されるaccess_logパーティションテーブルのあるDB | 618（※） | 5（※） |

※ データサイズは1パーティションのサイズ。最大並列度はパーティションのサイズで決まります。

### 4.4.3. DB設定

性能測定にあたり、デフォルトから変更したパラメータは以下のとおりです。

表 4.6 デフォルトから変更したパラメータ

| パラメータ | 256GBメモリサーバ | 2TBメモリサーバ |
|---|---|---|
| listen_addresses | * | * |
| shared_buffers | 100GB | 100GB |
| work_mem | 500MB | 500MB or 3GB(※) |
| wal_level | replica | replica |
| max_worker_processes | 20 | 100 |
| max_parallel_workers_per_gather(※) | 16 | 16 |
| checkpoint_timeout | 1d | 1d |
| max_wal_size | 4GB | 4GB |
| logging_collector | on | on |
| log_rotation_size | 0 | 0 |
| log_checkpoints | on | on |
| log_line_prefix | '%m %h %d %u ' | '%m %h %d %u ' |

※ 測定により値を変更して実施します。

### 4.4.4. 測定手順

#### 4.4.4.1. 検証A測定方法

1年分のデータを単一テーブルに保持するデータベース(base366day)のaccess_logテーブルから一日分(768万行)のデータを集計するSQLを、psqlを使用してexplain (analyze , verbose)をつけて実行します。

```
explain (analyze, verbose) select date, count(*), max(time), avg(access_count), min(log_id)  from a
ccess_log where log_id % 366 = 0 group by date;
```

デフォルトの設定だと230GB程度の大きさのテーブル(access_logテーブル)では、計算上の最大並列度は11です。

そこで起動ワーカー数がデフォルト計算上の最大並列度以上となるように、parallel_setup_costとparallel_tuple_costとmin_parallel_relation_sizeを以下のとおり設定し、並列度と処理時間の関係を測定します。なお、各パラメータの説明については 表． %s を参照してください。

```
set parallel_setup_cost=0;
set parallel_tuple_cost=0;
set min_parallel_relation_size=0;
set max_parallel_workers_per_gather= N;
/* N:1〜69 */
```

#### 4.4.4.2. 検証B測定方法

1年分のデータを単一テーブルに保持するデータベース(base366day)とパーティションテーブルに保持するデータベース(part366day)のaccess_logテーブルから一日分(768万行)のデータを集計するSQLを、psqlを使用してexplain (analyze , verbose)をつけて実行します。

パーティションプルーニングが効かないSQLとパーティションプルーニングが効くSQLの2つのSQLの測定を行います。

パーティションプルーニングが効かないSQL

```
explain (analyze, verbose) select date, count(*), max(time), avg(access_count), min(log_id)  from a
ccess_log where log_id % 366 = 0 group by date;
```

パーティションプルーニングが効くSQL

```
explain (analyze, verbose) select date, count(*), max(time), avg(access_count), min(log_id)  from a
ccess_log where date = '20160101' group by date;
```

#### 4.4.4.3. 検証C測定方法

1年分のデータを単一テーブルに保持するデータベース(base366day)のaccess_logテーブルと、product_masterテーブルとを結合するSQLを、psqlを使用してexplain (analyze , verbose)をつけて実行します。

```
explain (analyze,verbose) select a.product_id, p.product_name from access_log a, product_master p w
here a.product_id = p.product_id and a.log_id % 77 = 0 and date = '20160101';
```

また結合方法の違いによる動作を確認するため、明示的にネステッドループ結合あるいはハッシュ結合が選択されるように、以下のとおり設定を行って測定します。

ネステッドループ結合

```
set enable_nestloop=true;
set enable_mergejoin=false;
set enable_hashjoin=false;
```

ハッシュ結合

```
set enable_nestloop=false;
set enable_mergejoin=false;
set enable_hashjoin=true;
```

## 4.5. 検証結果

### 4.5.1. 検証A(単一テーブルに対するパラレルクエリの効果)

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

メモリサイズが2TBのサーバを使い、並列度を70まで増やした時に性能が単体処理と比較してどの程度向上するかを検証しました。

表 4.7 検証条件

| DBサーバ | ストレージ | SQL文 | 保持データ量 |
|---|---|---|---|
| 2TBメモリサーバ | FCストレージ | 768万行(一日分)を集計するSQL | 1年分(768万×366) |

並列度を変化させた場合の処理時間のグラフと表を以下に示します。

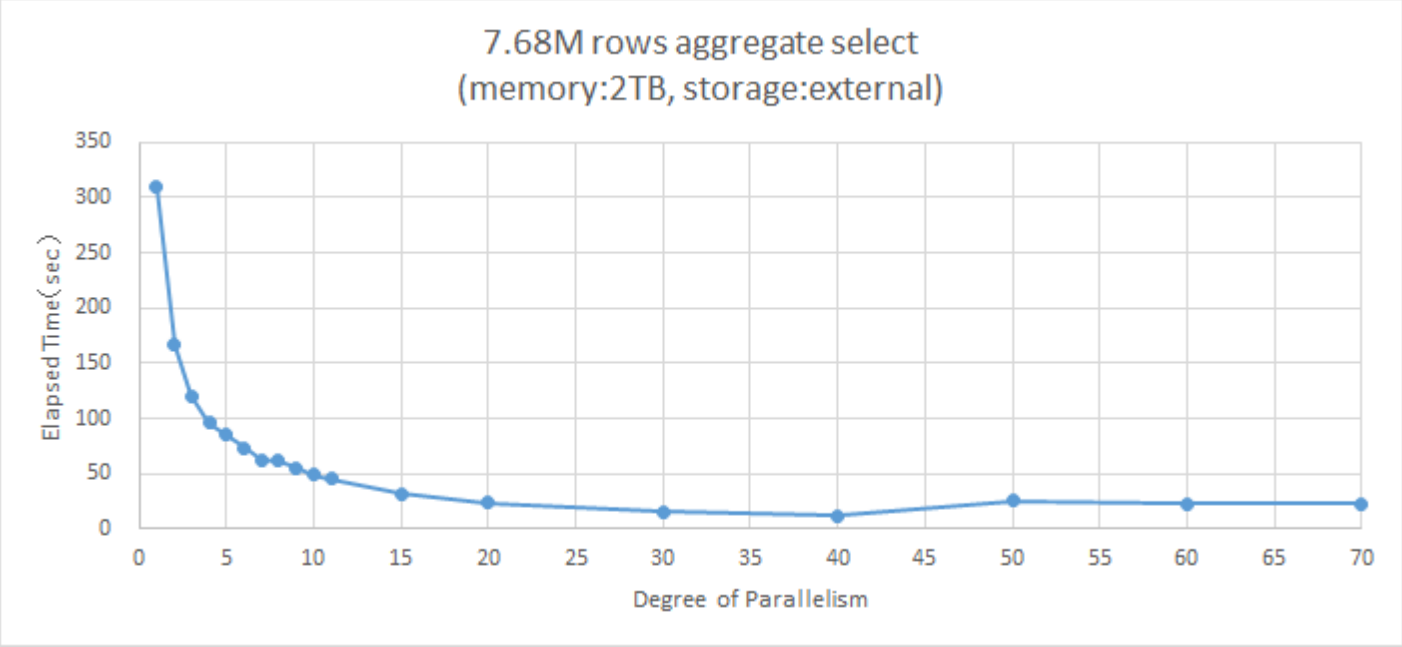



図 4.1 データがメモリに載りきる場合の処理時間の変化

表 4.8 並列度と処理時間

| 並列度 | 処理時間(ミリ秒) | 性能向上比 |
|---|---|---|
| 1 | 309,315 | 1.00 |
| 2 | 166,692 | 1.86 |
| 3 | 120,424 | 2.57 |
| 4 | 95,868 | 3.23 |
| 5 | 85,187 | 3.63 |
| 6 | 73,669 | 4.20 |
| 7 | 62,176 | 4.97 |
| 8 | 61,909 | 5.00 |
| 9 | 54,929 | 5.63 |
| 10 | 49,251 | 6.28 |
| 11 | 44,922 | 6.89 |
| 15 | 31,969 | 9.68 |
| 20 | 23,570 | 13.12 |
| 30 | 15,703 | 19.70 |
| 40 | 12,179 | 25.40 |
| 50 | 25,622 | 12.07 |
| 60 | 22,412 | 13.80 |
| 70 | 22,327 | 13.85 |

パラメータのデフォルト値での並列度11では、シーケンシャルスキャンの約7倍高速化しました。

パラメータを変更して並列度を70まで変更して計測した結果、性能向上が最大となったのは並列度40での 約25倍 でした。

並列度が40を超えるとオーバーヘッドにより性能は向上せず、並列度50～70では並列度20と同等の性能向上である約13倍にまで戻りました。

#### 4.5.1.2. 検証A 補足1:データがメモリにのりきらない場合の検証結果

2TBメモリサーバでの検証では、データサイズが230GBのテーブルはすべてのデータファイルがOSのファイルキャッシュにおさまります。そのため、検索を実施してもディスクへの物理I/Oがほとんど発生しておらず、CPUを効率的に使用できていると考えられます。この推測が正しいか検証するため、メモリサイズの小さな256GBメモリサーバ上で同じデータを使用して並列度を変えて検証を行いました。

表 4.9 検証条件

| DBサーバ | ストレージ | SQL文 | 保持データ量 |
|---|---|---|---|
| 256GBメモリサーバ | FCストレージ | 768万行（一日分）を集計するSQL | 1年分（768万×366） |

その結果の処理時間のグラフと表を以下に示します。

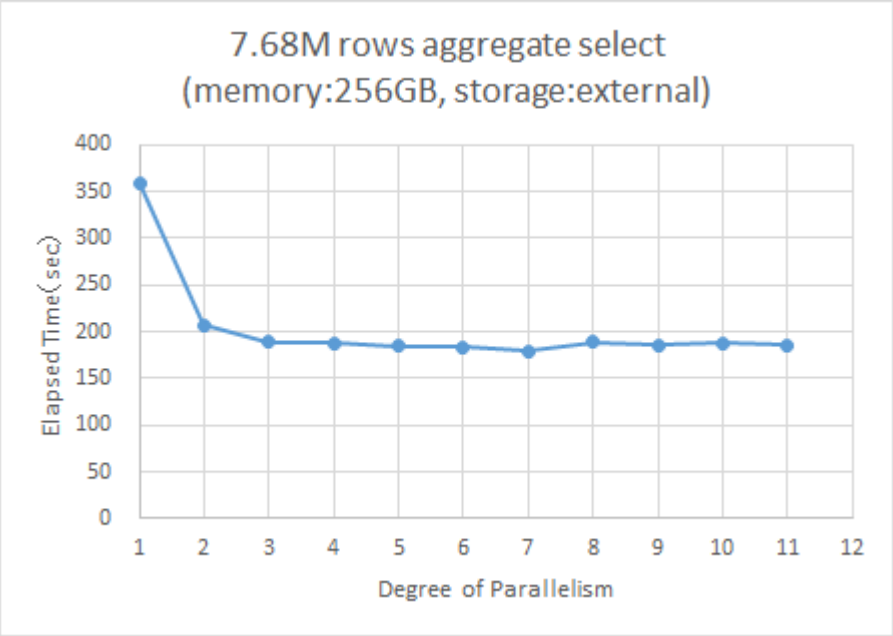



図 4.2 メモリに載りきらない場合の処理時間の変化

表 4.10 並列度と処理時間

| 並列度 | 366処理時間(ミリ秒) | 性能向上比 |
|---|---|---|
| 1 | 359,543 | 1.00 |
| 2 | 207,441 | 1.73 |
| 3 | 188,532 | 1.91 |
| 4 | 187,935 | 1.91 |
| 5 | 184,824 | 1.95 |
| 6 | 183,906 | 1.96 |
| 7 | 179,571 | 2.00 |
| 8 | 189,229 | 1.90 |
| 9 | 185,575 | 1.94 |
| 10 | 187,268 | 1.92 |
| 11 | 185,281 | 1.94 |

並列度が3以上では性能向上比がほぼ2倍で横ばいになり、データがメモリに乗りきるサイズと比較するとパラレルクエリの効果が低いことがわかります。データファイルが格納されているディスクに対するI/O 状況を sar -d コマンドで確認したところ以下のような状況でした。

表 4.11 sar -d コマンドによるディスクI/O 状況確認結果

| | 2TBサーバ | | 256GBサーバ | |
|---|---|---|---|---|
| 並列度 | 平均rd_sec/s | 平均%util | 平均rd_sec/s | 平均%util |
| 1 | 0.00 | 0.00 | 1,206,857 | 70.11 |
| 2 | 0.00 | 0.00 | 2,111,040 | 94.76 |
| 3 | 0.00 | 0.00 | 2,384,332 | 96.18 |
| 4 | 0.00 | 0.00 | 2,333,760 | 96.92 |
| 5 | 0.00 | 0.00 | 2,345,654 | 97.33 |
| 6 | 0.00 | 0.00 | 2,401,205 | 97.60 |
| 7 | 0.00 | 0.00 | 2,442,297 | 97.83 |
| 8 | 0.00 | 0.19 | 2,407,313 | 97.94 |
| 9 | 0.00 | 0.22 | 2,406,484 | 98.05 |
| 10 | 0.00 | 0.29 | 2,366,832 | 98.14 |
| 11 | 0.00 | 0.35 | 2,430,127 | 98.16 |

推測とおり、メモリ2TBのサーバではディスクI/Oがほとんど発生していないのに対し、メモリ256GBサーバでは性能向上が頭打ちになる並列度3以降のrd_sec/sが約240万/秒となっています。rd_sec/sは1回に1セクター(512バイト)のI/Oであるため、計算では約1200MB/秒のI/Oが発生していることになります。また、%utilの値もほぼ100%に近く、これがボトルネックになっていると考えられます。

#### 4.5.1.3. 検証A 補足2:ストレージ性能の違いによる検証結果

ディスクI/Oがパラレルクエリによる性能向上のボトルネックになることをさらに確かめるため、I/O性能の低い内蔵ディスクにデータファイルを配置して同じ検証を行いました。

表 4.12 検証条件

| DBサーバ | ストレージ | SQL文 | 保持データ量 |
|---|---|---|---|
| 256GBメモリサーバ | 内蔵ディスク | 768万行(一日分)を集計するSQL | 1年分(768万×366) |

その結果の処理時間のグラフと表を以下に示します。

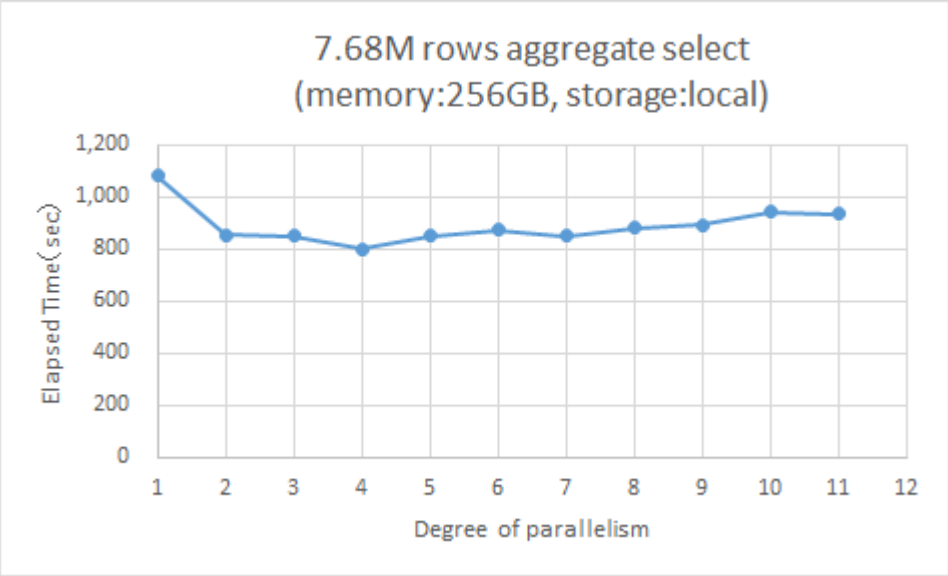



図 4.3 内蔵ディスクを使用した場合の処理時間の変化

表 4.13 並列度と処理時間

| 並列度 | 処理時間(ミリ秒) | 性能向上比 |
|---|---|---|
| 1 | 1,082,303 | 1.00 |
| 2 | 853,046 | 1.27 |
| 3 | 850,580 | 1.27 |
| 4 | 798,475 | 1.36 |
| 5 | 851,638 | 1.27 |
| 6 | 874,269 | 1.24 |
| 7 | 850,908 | 1.27 |
| 8 | 879,778 | 1.23 |
| 9 | 893,578 | 1.21 |
| 10 | 943,831 | 1.15 |
| 11 | 936,294 | 1.16 |

高性能なFCストレージと比較すると絶対的な処理時間も長く、パラレルクエリによる性能向上の幅も限定的であることがわかります。

### 4.5.2. 検証B(パーティションテーブルと単一(非パーティション)テーブルの比較)

時系列などの大量データを保持するテーブルの検索性能向上のために、テーブルをパーティション化し、検索条件で検索対象となるパーティションを絞り込む(パーティションプルーニング)という手法が使われることがあります。

パラレルクエリによる性能向上と、パーティションテーブルによる性能向上がどの程度の差があるのかを、パーティションプルーニングが効かないSQLと、パーティションプルーニングが効くSQLの2通りを検証しました。

#### 4.5.2.1. パーティションプルーニングが効かないSQL

検索条件にパーティションキーがなく、パーティションプルーニングが効かないSQLの検証結果は以下のとおりです。

表 4.14 検証条件

| DBサーバ | ストレージ | SQL文 | 保持データ量 |
|---|---|---|---|
| 2TBメモリサーバ | FCストレージ | 768万行（一日分）を集計するSQL。<br>パーティションプルーニング無効 | 1年分（768万×366） |

その結果の処理時間のグラフと表を以下に示します。

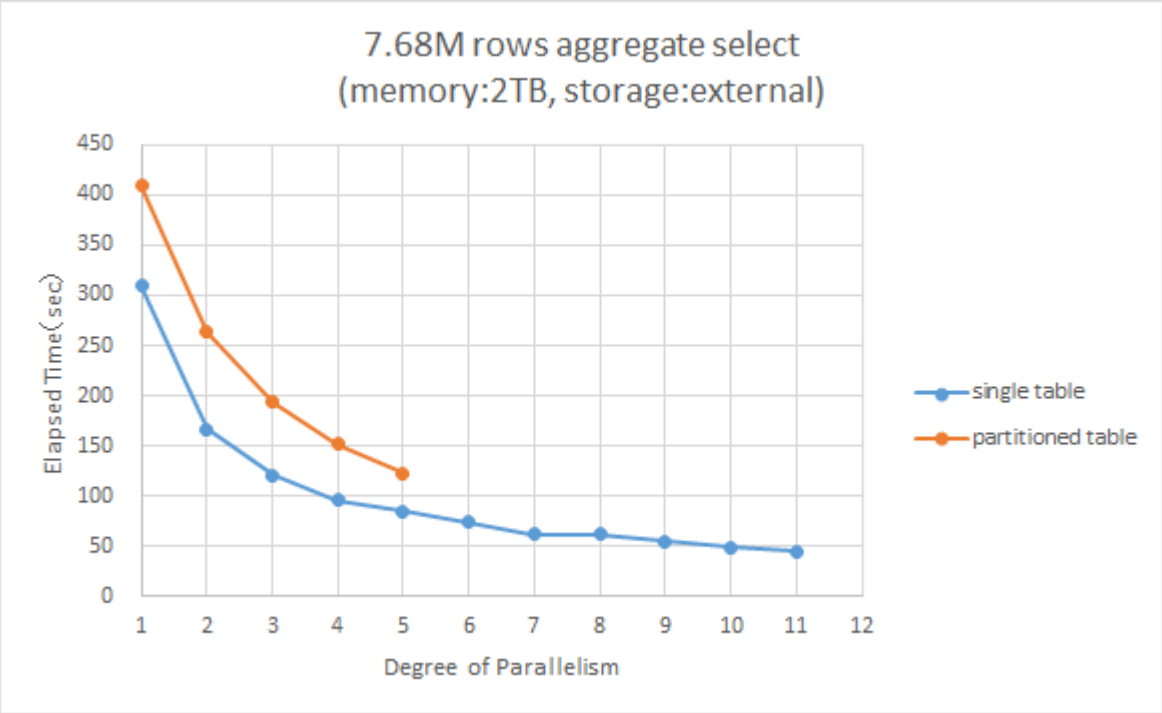



図 4.4 パーティションプルーニングが無効となるSQL処理時間比較

表 4.15 並列度と処理時間(パーティションプルーニングが無効)

| 並列度 | 単一テーブル 処理時間(ミリ秒) | パーティションテーブル 処理時間(ミリ秒) |
|---|---|---|
| 1 | 309,315 | 408,717 |
| 2 | 166,692 | 263,438 |
| 3 | 120,424 | 193,406 |
| 4 | 95,868 | 151,556 |
| 5 | 85,187 | 122,441 |
| 6 | 73,669 | |
| 7 | 62,176 | |
| 8 | 61,909 | |
| 9 | 54,929 | |
| 10 | 49,251 | |
| 11 | 44,922 | |

パーティションプルーニングが効かないSQLでは、並列度が1、つまりパラレルクエリが有効ではない状態では、単一テーブルで309秒、パーティションテーブルで409秒と、パーティションテーブルではオーバーヘッドのため約1.3倍処理時間がかかっています。ただしパーティションプルーニングが効かないSQLでもパラレルクエリにより性能向上は見込め、並列度5の時に122秒と非パラレルの場合と比べて、約3.3倍性能向上しています。

#### 4.5.2.2. パーティションプルーニングが効くSQL

検索条件にパーティションキーがあり、パーティションプルーニングが効くSQLの検証結果は以下のとおりです。

表 4.16 検証条件

| DBサーバ | ストレージ | SQL文 | 保持データ量 |
|---|---|---|---|
| 2TBメモリサーバ | FCストレージ | 768万行（一日分）を集計するSQL。<br>パーティションプルーニング有効 | 1年分（768万×366） |

その結果の処理時間のグラフと表を以下に示します。

パーティションプルーニングが効く場合は、単一テーブル293秒に比べてパーティションテーブルでは4秒と2桁ほど処理時間が短いです。

パーティションプルーニングが効く場合でもパラレル化による性能向上は見込めますが、元々速いので絶対値としての削減量は小さいです。

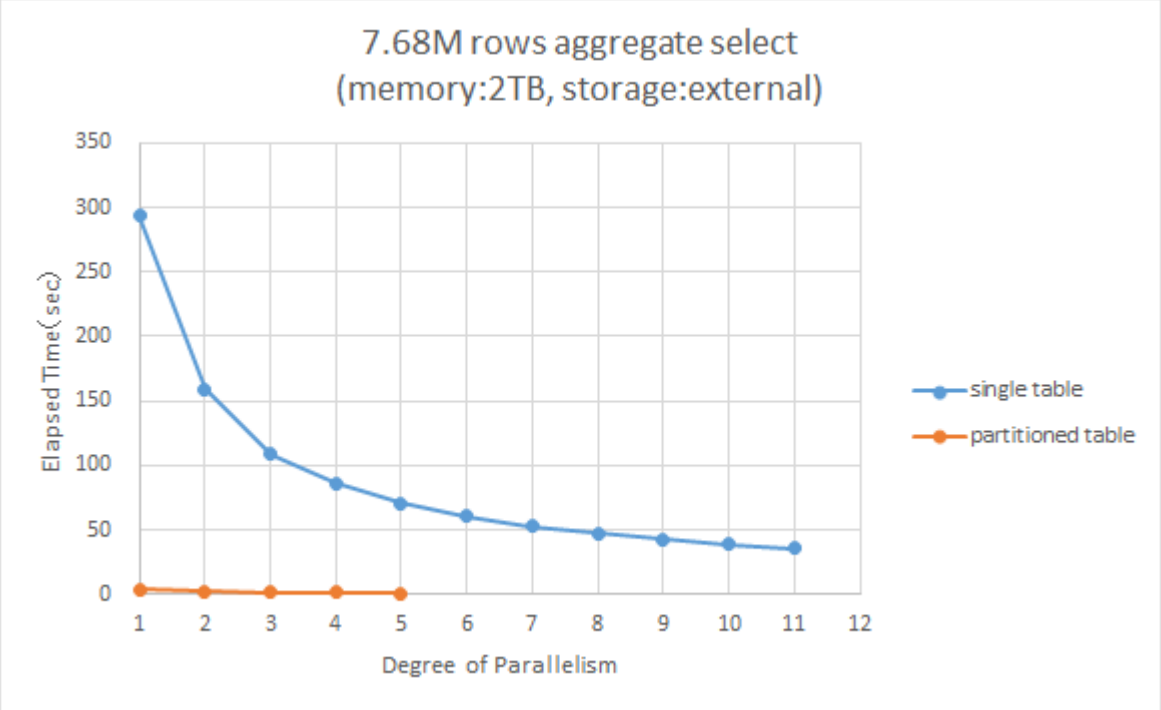



図 4.5 パーティションプルーニングが有効となるSQL処理時間比較

表 4.17 並列度と処理時間

| 並列度 | 単一テーブル 処理時間(ミリ秒) | パーティションテーブル 処理時間(ミリ秒) |
|---|---|---|
| 1 | 293,109 | 3,950 |
| 2 | 159,124 | 2,078 |
| 3 | 108,743 | 1,483 |
| 4 | 86,336 | 1,263 |
| 5 | 70,363 | 1,135 |
| 6 | 60,635 | |
| 7 | 52,904 | |
| 8 | 47,057 | |
| 9 | 42,420 | |
| 10 | 38,860 | |
| 11 | 35,863 | |

### 4.5.3. 検証C(異なるワーカー数が起動するテーブル同士結合時のワーカー数検証)

計算上起動されるワーカー数が異なるテーブルを結合した場合、実際に起動されるワーカー数がどのようになるか、また、テーブル結合の方式によってどのようになるかを検証しました。起動されるワーカーの状況の確認のためには、explain analyzeよりもexplain ( analyze, verbose )のほうが、実際に起動されたワーカー内の処理時間や処理した行の概数が表示されるため、好ましいです。

#### 4.5.3.1. ネステッドループ結合

256GBメモリサーバ上でネステッドループ結合を行うSQLを実行しました。小さいテーブル（product_masterテーブル）へのアクセスは、結合に使用するカラムにインデックス

が作成されているため、Index Scanとなります。そのためパラレルスキャンにはなりません。大きいテーブル（access_logテーブル）のスキャンは複数プロセスで並列処理され、起動されるワーカー数は10となっています。

表 4.18 検証条件

| DBサーバ | ストレージ | SQL文 | 保持データ量 |
|---|---|---|---|
| 256GBメモリサーバ | FCストレージ | 2テーブルを結合させ10万行を取得するSQL。 | 1年分（768万×366） |

explain analyzeとexplain (analyze, verbose)の出力結果は以下のとおりです。explain (analyze, verbose)のほうが詳細な情報が出力されていることがわかります。

・explain analyze

```
                                                                 QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
-------------------------------------------
Nested Loop  (cost=1000.42..34216441.40 rows=37882 width=40) (actual time=8.075..186255.334 rows=99
740 loops=1)
  ->  Gather  (cost=1000.00..33901973.20 rows=37882 width=4) (actual time=8.059..186109.137 rows=99
740 loops=1)
        Workers Planned: 10
        Workers Launched: 10
        ->  Parallel Seq Scan on access_log a  (cost=0.00..33897185.00 rows=3788 width=4) (actual t
ime=3.800..186236.448 rows=9067 loops=11)
              Filter: ((date = '20160101'::bpchar) AND ((log_id % '77'::bigint) = 0))
              Rows Removed by Filter: 255525478
  ->  Index Scan using product_master_pkey on product_master p  (cost=0.42..8.29 rows=1 width=40) (
actual time=0.001..0.001 rows=1 loops=99740)
        Index Cond: (product_id = a.product_id)
```

・explain (analyze, verbose)

```
                                                                 QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
-------------------------------------------
Nested Loop  (cost=1000.42..34216441.40 rows=37882 width=40) (actual time=6.127..180622.369 rows=99
740 loops=1)
  Output: a.product_id, p.product_name
  ->  Gather  (cost=1000.00..33901973.20 rows=37882 width=4) (actual time=6.105..180479.876 rows=99
740 loops=1)
        Output: a.product_id
        Workers Planned: 10
        Workers Launched: 10
        ->  Parallel Seq Scan on public.access_log a  (cost=0.00..33897185.00 rows=3788 width=4) (a
ctual time=6.159..180604.178 rows=9067 loops=11)
              Output: a.product_id
              Filter: ((a.date = '20160101'::bpchar) AND ((a.log_id % '77'::bigint) = 0))
              Rows Removed by Filter: 255525478
              Worker 0: actual time=7.211..180618.215 rows=9587 loops=1
              Worker 1: actual time=7.345..180618.488 rows=10195 loops=1
              Worker 2: actual time=7.529..180618.285 rows=9782 loops=1
              Worker 3: actual time=7.719..180618.567 rows=9710 loops=1
              Worker 4: actual time=7.882..180619.043 rows=9990 loops=1
              Worker 5: actual time=7.923..180618.817 rows=9523 loops=1
              Worker 6: actual time=2.440..180619.029 rows=9909 loops=1
              Worker 7: actual time=2.772..180619.381 rows=10155 loops=1
              Worker 8: actual time=7.899..180619.093 rows=9943 loops=1
              Worker 9: actual time=3.138..180619.706 rows=10282 loops=1
  ->  Index Scan using product_master_pkey on public.product_master p  (cost=0.42..8.29 rows=1 widt
h=40) (actual time=0.001..0.001 rows=1 loops=99740)
        Output: p.product_id, p.product_name
        Index Cond: (p.product_id = a.product_id)
```

#### 4.5.3.2. ハッシュ結合

256GBメモリサーバ上でハッシュ結合を行うSQLを実行しました。起動されたワーカー数は10となりました。

計算上起動されるワーカー数は大きいテーブル（access_logテーブル）が10、小さいテーブル（product_masterテーブル）が4です。explain (analyze, verbose)コマンドの結果を見ると、実際にはproduct_masterテーブルも10個のワーカーによりスキャンされている(赤字部分)ことが確認できました。

・explain analyze

```
                                                                 QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
---------------------------------------
Gather  (cost=8797.00..33911415.29 rows=37882 width=40) (actual time=79.738..238055.321 rows=99740
loops=1)
  Workers Planned: 10
  Workers Launched: 10
  ->  Hash Join  (cost=7797.00..33906627.09 rows=37882 width=40) (actual time=76.084..238048.282 ro
ws=9067 loops=11)
        Hash Cond: (a.product_id = p.product_id)
        ->  Parallel Seq Scan on access_log a  (cost=0.00..33897185.00 rows=3788 width=4) (actual t
ime=4.605..237973.441 rows=9067 loops=11)
              Filter: ((date = '20160101'::bpchar) AND ((log_id % '77'::bigint) = 0))
              Rows Removed by Filter: 255525478
        ->  Hash  (cost=3734.00..3734.00 rows=200000 width=40) (actual time=70.721..70.721 rows=200
000 loops=11)
              Buckets: 65536  Batches: 8  Memory Usage: 2316kB
              ->  Seq Scan on product_master p  (cost=0.00..3734.00 rows=200000 width=40) (actual t
ime=0.018..32.052 rows=200000 loops=11)
```

・explain (analyze, verbose)

```
                                                                 QUERY PLAN
---------------------------------------------------------------------------------------------------------------------
-------------------------------------
Gather  (cost=8797.00..33911415.29 rows=37882 width=40) (actual time=67.438..286325.054 rows=99740
loops=1)
  Output: a.product_id, p.product_name
  Workers Planned: 10
  Workers Launched: 10
  ->  Hash Join  (cost=7797.00..33906627.09 rows=37882 width=40) (actual time=69.543..286317.637 ro
ws=9067 loops=11)
        Output: a.product_id, p.product_name
        Hash Cond: (a.product_id = p.product_id)
        Worker 0: actual time=71.316..286319.780 rows=9167 loops=1
        Worker 1: actual time=71.334..286319.553 rows=9348 loops=1
        Worker 2: actual time=67.892..286320.195 rows=9203 loops=1
        Worker 3: actual time=71.601..286320.017 rows=9284 loops=1
        Worker 4: actual time=67.893..286320.432 rows=9090 loops=1
        Worker 5: actual time=71.888..286320.081 rows=9376 loops=1
        Worker 6: actual time=69.246..286320.465 rows=9134 loops=1
        Worker 7: actual time=72.278..286320.148 rows=9304 loops=1
        Worker 8: actual time=66.812..286320.426 rows=9131 loops=1
        Worker 9: actual time=67.552..286321.155 rows=9165 loops=1
        ->  Parallel Seq Scan on public.access_log a  (cost=0.00..33897185.00 rows=3788 width=4) (a
ctual time=1.925..286246.477 rows=9067 loops=11)
              Output: a.product_id
              Filter: ((a.date = '20160101'::bpchar) AND ((a.log_id % '77'::bigint) = 0))
              Rows Removed by Filter: 255525478
              Worker 0: actual time=2.199..286247.064 rows=9167 loops=1
              Worker 1: actual time=2.196..286246.933 rows=9348 loops=1
              Worker 2: actual time=2.427..286251.181 rows=9203 loops=1
              Worker 3: actual time=2.473..286247.263 rows=9284 loops=1
              Worker 4: actual time=2.684..286251.689 rows=9090 loops=1
              Worker 5: actual time=2.770..286247.360 rows=9376 loops=1
              Worker 6: actual time=2.802..286250.378 rows=9134 loops=1
              Worker 7: actual time=0.067..286244.552 rows=9304 loops=1
              Worker 8: actual time=0.053..286250.022 rows=9131 loops=1
              Worker 9: actual time=0.447..286250.427 rows=9165 loops=1
        ->  Hash  (cost=3734.00..3734.00 rows=200000 width=40) (actual time=66.977..66.977 rows=200
000 loops=11)
              Output: p.product_name, p.product_id
              Buckets: 65536  Batches: 8  Memory Usage: 2316kB
              Worker 0: actual time=68.361..68.361 rows=200000 loops=1
              Worker 1: actual time=68.444..68.444 rows=200000 loops=1
              Worker 2: actual time=64.821..64.821 rows=200000 loops=1
              Worker 3: actual time=68.435..68.435 rows=200000 loops=1
              Worker 4: actual time=64.598..64.598 rows=200000 loops=1
              Worker 5: actual time=68.450..68.450 rows=200000 loops=1
              Worker 6: actual time=65.770..65.770 rows=200000 loops=1
              Worker 7: actual time=71.617..71.617 rows=200000 loops=1
              Worker 8: actual time=66.124..66.124 rows=200000 loops=1
              Worker 9: actual time=66.344..66.344 rows=200000 loops=1
              ->  Seq Scan on public.product_master p  (cost=0.00..3734.00 rows=200000 width=40) (a
ctual time=0.017..28.920 rows=200000 loops=11)
                    Output: p.product_name, p.product_id
                    Worker 0: actual time=0.020..30.171 rows=200000 loops=1
                    Worker 1: actual time=0.018..29.697 rows=200000 loops=1
                    Worker 2: actual time=0.017..26.473 rows=200000 loops=1
                    Worker 3: actual time=0.017..30.388 rows=200000 loops=1
                    Worker 4: actual time=0.012..26.610 rows=200000 loops=1
                    Worker 5: actual time=0.018..30.037 rows=200000 loops=1
                    Worker 6: actual time=0.019..27.514 rows=200000 loops=1
                    Worker 7: actual time=0.020..33.813 rows=200000 loops=1
                    Worker 8: actual time=0.012..28.000 rows=200000 loops=1
                    Worker 9: actual time=0.021..28.487 rows=200000 loops=1
```

## 4.6. 考察

### 4.6.1. 検証A

256GBメモリサーバと2TBメモリサーバではPostgreSQLのキャッシュ領域であるshared_buffersの値は両方とも100GBに設定していました。しかし、同一のデータ量、同一のSQLをパラレルクエリで実行させた場合に、処理時間に大きな差がでました。その原因は2TBメモリサーバではディスクI/Oが発生しなかったのに対し、256GBメモリサーバでは大量のディスクI/Oが発生し、それがボトルネックになったことがあげられます。I/O性能の低い内蔵ディスクを使用した場合には、ボトルネックがさらに顕著になりました。このことから、データがキャッシュ(shared_buffersではなくOSのファイルキャッシュ)に載ると、ディスクI/Oのボトルネックが解消され、デフォルトの最大並列度以上の並列度まで処理性能が向上するといえます。本検証の2TBメモリサーバ上ではデフォルトの11よりも大きな40までスケールしました。

内蔵ディスクのようにI/O性能が高くないディスクの場合、キャッシュに載りきらないテーブルではデフォルトの最大並列度にいたるまでに性能が悪化するので注意が必要です。

### 4.6.2. 検証B

パーティションプルーニングが有効な検索では、パラレルクエリによる効果よりもパーティションプルーニングの効果が高く、パーティションテーブルへの検索の方が非常に高速でした。一方パーティションプルーニングが有効にならない検索では、パーティション化によるオーバーヘッドにより単一テーブルの検索の方が速くなりますが、パラレルクエリの効果により、どちらのテーブルでも処理性能は向上します。

以上の結果から、大量データの検索性能向上の手段としては、いままでとおりパーティションテーブルの導入が有効です。PostgreSQL 9.6のパーティションテーブルには作成方法が複雑だったり、制約があったりしましたが、次期バージョンであるPostgreSQL 10では多くの改善がされる予定ですので、期待したいと思います。

### 4.6.3. 検証C

テーブルを結合するSQLでは、起動されるワーカー数は並列度の大きいテーブル(access_logテーブル)によって決定され、ハッシュ結合の場合は小さいテーブル(product_masterテーブル)も大きいテーブルのワーカー数でスキャンされました。ただし、explain コマンドに verboseオプションをつけないと、通常の「Seq Scan」のように表示されるため、注意が必要です。

また、product_masterテーブルは10個のワーカーでスキャンされていますが、「Parallel Seq Scan」ではなく「Seq Scan」となっている点も注目すべき点です。「Parallel Seq Scan」では各ワーカー(と親プロセス)が、全体のスキャン対象を分割してスキャンを行うのに対し、「Seq Scan」では各ワーカー(と親プロセス)が、それぞれ全体をスキャンしています。

### 4.6.4. 総括

本検証を通じて、パラレルクエリによる性能向上を有効活用するためには、搭載するメモリ量を多くしたり、高性能のストレージを利用したりするなどの方法でディスクI/Oの影響を極力減らすことが必要であることがわかりました。

また、パーティションテーブルに対するパーティションプルーニングはパラレルクエリによって代替されるものではなく、依然有効であること、またパーティションプルーニングが効かない場合にも、パラレルクエリによる性能向上があることが判明したのは大きな収穫でした。

パラレルクエリが実装されたことは、ハードウェア技術の進化を生かした大規模エンタープライズ用途向けの機能の一つがPostgreSQLに備わったと感じました。次期バージョンでのパラレルクエリのさらなる進化に期待したいです。

# 5. Pgpool-II 検証

## 5.1. 検証概要

以前2012年度に Pgpool-II のレプリケーション検証を行っております (2012年度報告は こちら です)。当時は Pgpool-II が独自のレプリケーション機能を用いる方式(ネイティブレプリケーション)についての特性を検証しましたが、本年度はそれの再試に加え、PostgreSQL 9.6 で新しく登場し、ネイティブレプリケーションと特性の似た remote_apply 同期ストリーミングレプリケーション方式との参照性能および更新性能の比較検証を行います。加えて、非同期設定と同期設定の比較を行うために非同期ストリーミングレプリケーション方式も検証対象としました。

## 5.2. Pgpool-IIとは

### 5.2.1. Pgpool-II 概要

Pgpool-II は、OSS として開発、公開されている PostgreSQL 専用のクラスタリング用ミドルウェアです (Pgpool-II オフィシャルサイトは こちら です)。

サポートする PostgreSQL のバージョンは 6.4 以降です。稼働環境は Linux/UNIX で、Windows では動作しません。Pgpool-II は PostgreSQL のクライアントアプリケーションと PostgreSQL の間に割りこませる proxy のような使い方をします。つまり、PostgreSQL のクライアントアプリケーションから見ると PostgreSQL に見えて、PostgreSQL から見るとクライアントアプリケーションのように振る舞います。

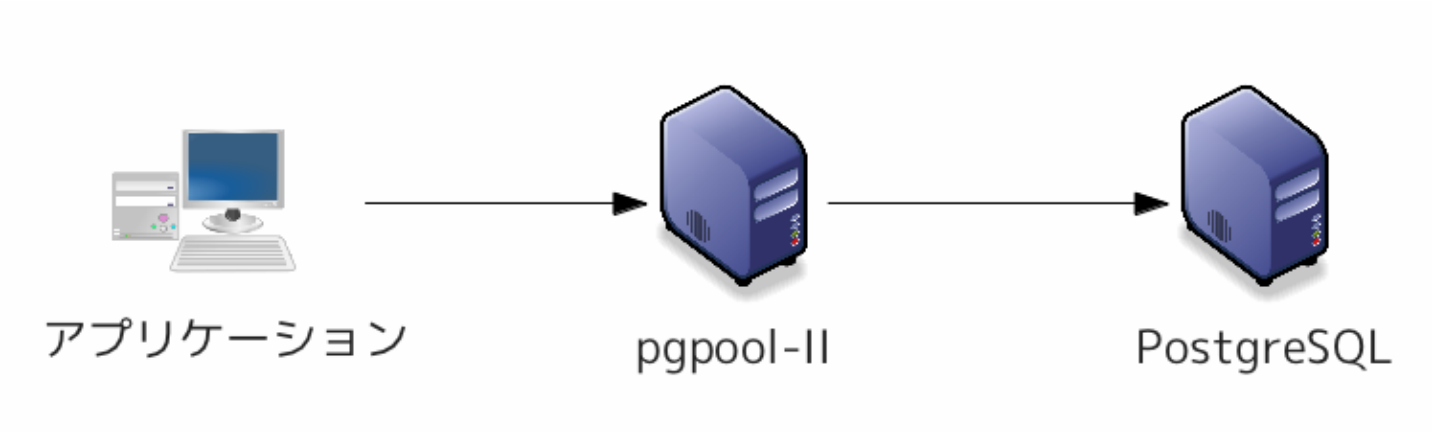



図 5.1 Pgpool-II と PostgreSQL

Pgpool-II を利用する上では、Pgpool-II は極力クライアントアプリケーションには手を入れなくて済むように設計されています（制限事項があります）。実際、本検証で使用している pgbench は、PostgreSQL で直接使う場合と同じ使い方ができます。また、PostgreSQL 本体にはまったく手を入れる必要はありません。

### 5.2.2. Pgpool-II 機能

Pgpool-II の機能は大きく分けて以下の4つがあります。

- コネクションプーリング

PostgreSQL への接続を保存しておき、再利用することによって PostgreSQL への接続オーバヘッドを低減し、システム全体の性能を向上することができます。[ 注: 本検証では pgbench が持続的に PostgreSQL に接続するため、この機能の恩恵は受けません。]

- HAクラスタ

Pgpool-II は複数の PostgreSQL サーバを管理することができます。HAクラスタ機能を使用することにより、万が一どれかの DB サーバに障害が発生しても運用を継続することができます。PostgreSQL に障害が発生した際には、自動的にPostgreSQLサーバが切り離されます（フェイルオーバ）。

- 負荷分散

レプリケーションを運用している場合、どのサーバに問い合わせても同じ結果が返ってきます。多数の検索リクエストをそれぞれのサーバで分担して負荷を軽減させ、システム全体の性能を向上させることができます。最良の場合にはサーバ台数に比例した検索性能向上が見込めます。

- レプリケーション

レプリケーション機能を用いることで複数のDBサーバにリアルタイムで同じデータを保存することが出来ます。PostgreSQL にもレプリケーション機能がありますが、Pgpool-II にはネイティブレプリケーションと呼ばれる独自のレプリケーション機能が備わっております。

本検証で主に利用する機能はレプリケーションと負荷分散です。レプリケーションについて詳しく見ていきます。Pgpool-II では、2 つのレプリケーションモードがあります (Pgpool-II を起動する前に予め設定します)。

#### 5.2.2.1. ネイティブレプリケーションモード(Pgpool-IIの独自のレプリケーション機能)

更新クエリをすべての PostgreSQL に送信することにより、データベースのレプリケーションを実現します。

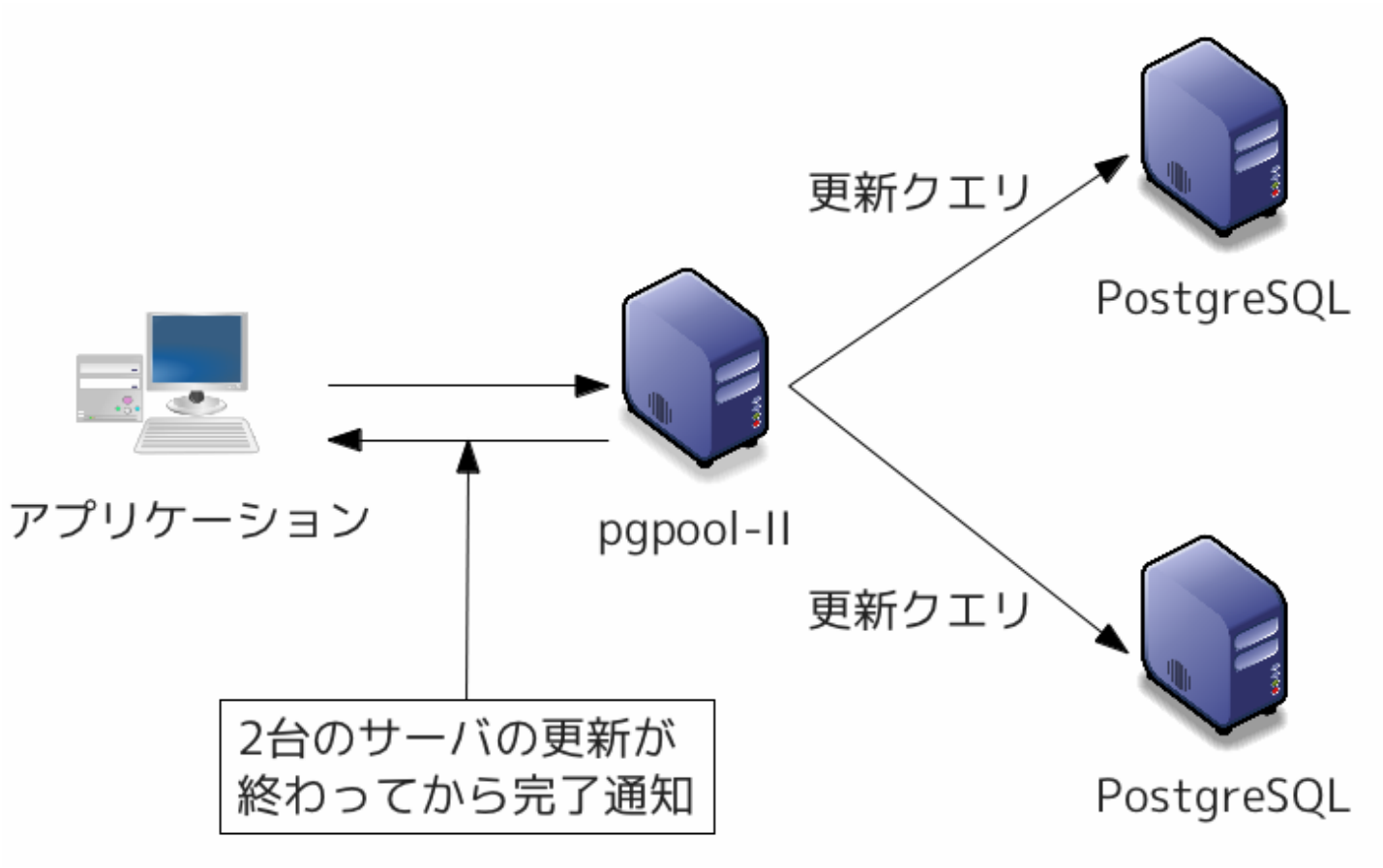



図 5.2 ネイティブレプリケーション利用(Pgpool-IIの独自レプリケーション機能を用いる方式)

この方式では、以下のように処理が行なわれます。

1. クライアントは更新クエリをPgpool-IIに送信します。
2. Pgpool-IIはまず更新クエリをマスターサーバ(健全なサーバのうち、最初に設定ファイルに記述されているサーバ)に送信、応答を待ちます。
3. マスターサーバから応答が返ってきたら、他のサーバにも同時に更新クエリを送信します。
4. すべてのサーバから更新コマンドの実行完了が返ってきたら、応答をクライアントに返します。

このように、クライアントに更新クエリの応答が返ってきた時点ですべてのサーバで更新が完了していることが保証されるため、これは同期レプリケーションとなります。同期レプリケーションでは、クライアントは負荷分散によって異なるサーバに振り分けられたSELECTがタイミングによって異なる結果(古い結果)を受け取る可能性がないため、PostgreSQLに直接接続していることを前提にしたプログラムの修正の必要がありません。

ただし、ステップ2-3で応答待ちを行なう必要があるため、直接PostgreSQLに接続して更新を行なう場合に比べて、必ず更新性能は悪くなります。

- PostgreSQL 2ノードの場合、マスター(ノード0)の更新を待ってからノード1を更新するので総合性能はPostgreSQLを直接使う場合の半分、つまり50%の性能
- PostgreSQL 3ノード以上では、2, 3...ノードは並列に更新を行なうので、やはり50%
- ノードあたりの性能は、全体性能が2ノード以上、何ノードになっても全体性能が50%なので、ノード数に応じて性能低下

図に、Pgpool-IIの同期レプリケーションにおける更新性能の理論値を示します。2012年度の検証では本特性を実際に確認しています。

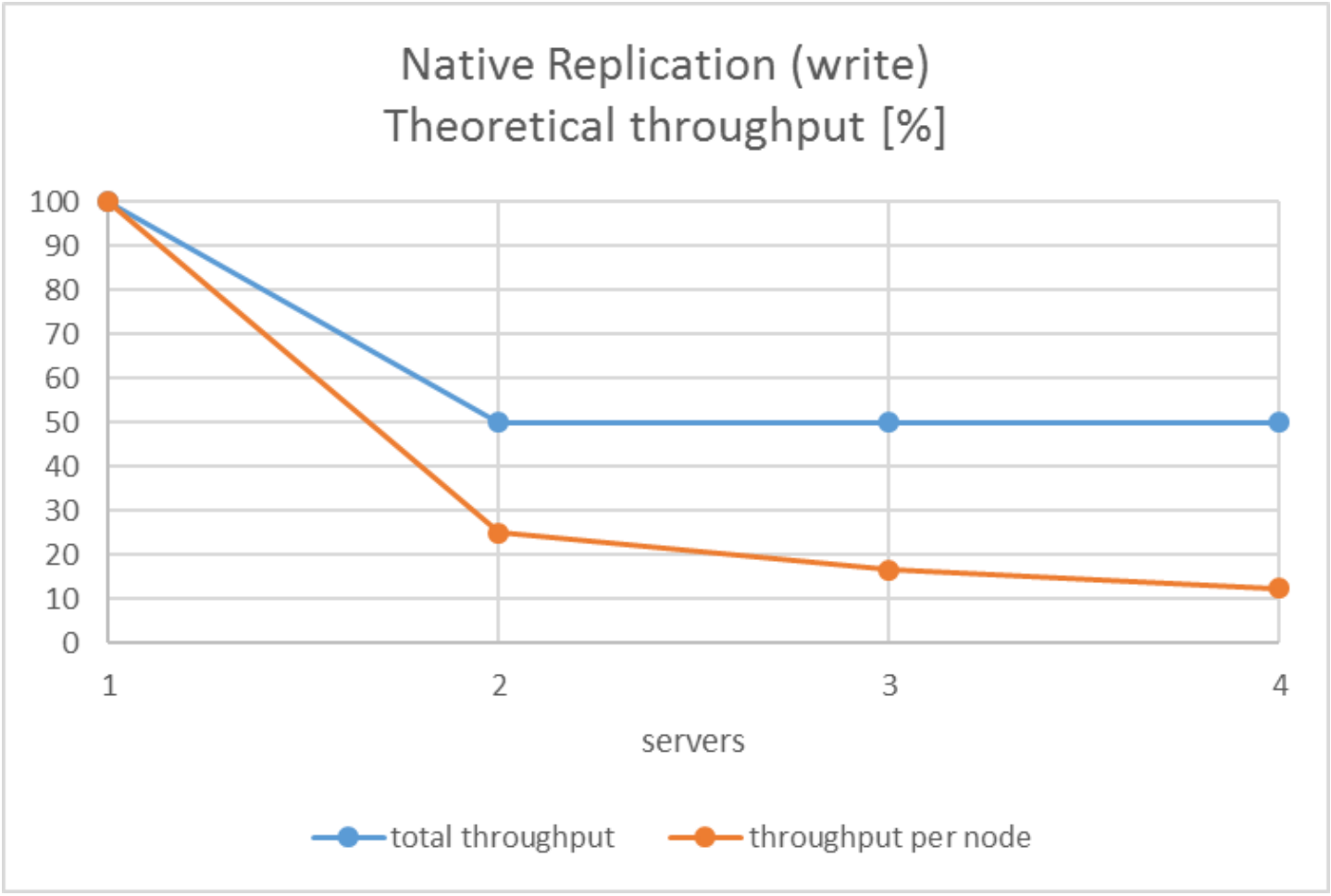



図 5.3 ネイティブレプリケーションの更新性能理論値

ただし、参照系クエリに関してはクラスタ中のただ1台のサーバで実行されます。どのサーバで実行されるかはサーバの重み付け設定(backend_weight)により決定されます。

各バックエンドは他のバックエンドの参照クエリの実行を待つ必要が無いので、参照クエリの実行性能はサーバ数によってスケールアウトします。その特性は2012年度検証で実際に確認しています。

#### 5.2.2.2. マスタースレーブモード(PostgreSQL のストリーミングレプリケーション機能を利用)

マスタースレーブモードは Pgpool-II 自体のレプリケーションの機能を利用せず、PostgreSQL のストリーミングレプリケーション機能を利用して、レプリケーションを実現する方法です(Slony-I のレプリケーション機能を利用することも可能です)。

マスタースレーブモードでは Pgpool-II は、コネクションプーリングと負荷分散、それにフェイルオーバのみを行ないます。負荷分散の際、ストリーミングレプリケーションのスタンバイは更新クエリを実行できないので、更新クエリは常にプライマリサーバに送信されます。また、ネイティブレプリケーションと同様に参照クエリはクラスタ中のただ1台のサーバで実行されます。

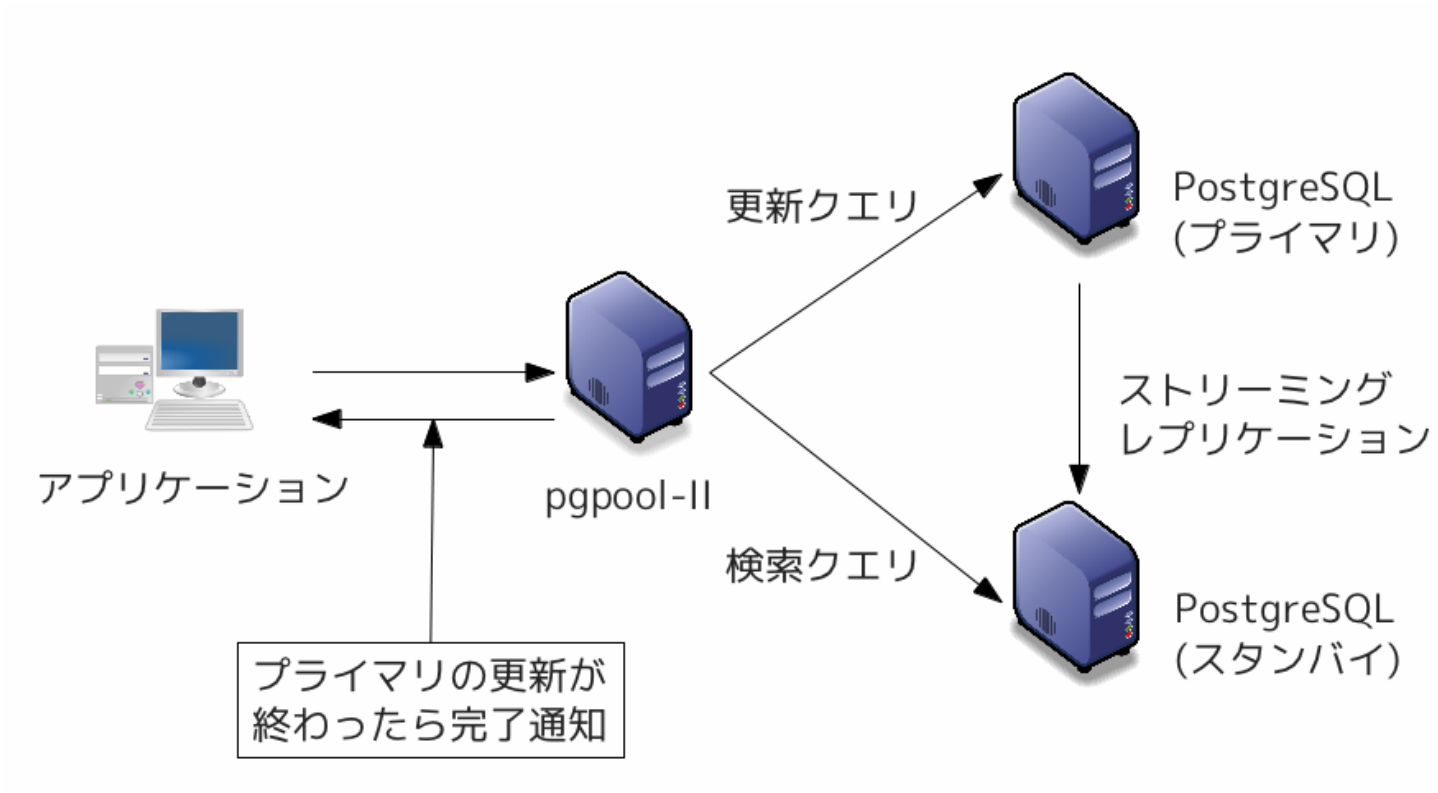



図 5.4 マスタースレーブモード(PostgreSQL のストリーミングレプリケーション機能を利用)

マスタースレーブモードでは以下のように処理が行われます。

1. クライアントは更新クエリをPgpool-IIに送信します。
2. Pgpool-IIはまず更新クエリをプライマリサーバに送信、応答を待ちます。
    1. 非同期設定の場合、スタンバイの状態は関係なく応答を返します。
    2. 同期設定の場合、スタンバイの応答を待ってから応答を返します。
3. プライマリサーバから応答が返ってきたら、応答をクライアントに返します。

PostgreSQL のストリーミングレプリケーションは2.以降でプライマリの更新WALをトランザクション単位でスタンバイに送信し、スタンバイはそのWALを適用することでデータレプリケートを行っています。

上記の 2.1., 2.2. で処理が異なる通り、PostgreSQL のレプリケーション方式は大きく分けて以下の2つがあります。

- 非同期ストリーミングレプリケーション

非同期設定(synchronous_standby_names設定なし)の場合、プライマリはスタンバイの応答を待たずにトランザクションのコミットを行います。故に、非同期ストリーミングレプリケーションでは、プライマリサーバのデータよりもスタンバイサーバのデータは遅れている事になります。反面、更新性能の低下は、ネイティブレプリケーションモードほど大きくありません。

- 同期ストリーミングレプリケーション

同期設定(synchronous_standby_names設定あり)の場合、プライマリはスタンバイの応答を待ち、トランザクションのコミットを行います。

以前はスタンバイが WAL をディスク書き込みするまで待つのが限度でしたが、PostgreSQL 9.6 からスタンバイ上でデータを見られる状態になるまで待つことが可能になりました。これに伴い、synchronous_commit 設定に remote_apply が追加されました。

```
synchronous_commit = remote_apply
```

さらに、以前は同期ストリーミングレプリケーションはただ1台のスタンバイのみに適用されるもの(※)でしたが、PostgreSQL 9.6 から複数のスタンバイに対して同期ストリーミングレプリケーションを適用することが可能になりました。それに伴い、synchronous_standby_names の書式が追加されました。

例えば、s1, s2, s3 という application_name を持ったスタンバイサーバがいるとして、以下の設定だと s1, s2 が同期先となり、s3 は予備の同期先となります。

```
synchronous_standby_names = '2 (s1, s2, s3)'
```

また、s1, s2, s3 全てを常に同期先とする場合は以下の設定となります。

```
synchronous_standby_names = '3 (s1, s2, s3)'
```

synchronous_commit = remote_apply の同期設定と複数同期ストリーミングレプリケーション設定は同時に設定可能です。つまり、複数のスタンバイに対してWAL適用まで待つ同期ストリーミングレプリケーションが可能です。本検証ではこれを利用して、全スタンバイに対してWAL適用まで待つ同期ストリーミングレプリケーションを検証対象とします。

(※)以前も複数のスタンバイ名を記述できましたが、あくまで同期先となるのは1台のみで、他のスタンバイは予備の同期先でしかありませんでした。例えば、以下の設定だと s1 が同期先となり、s2, s3 は予備の同期先となります。

```
synchronous_standby_names = 's1, s2, s3'
```

ただし、以前の書式もPostgreSQL9.6では有効です。

#### 5.2.2.3. レプリケーション方式まとめ

Pgpool-II で用いることのできるレプリケーション方式（今回の検証対象）を比較すると以下の表のようになります。表の「書き込みトランザクション完了までに必要な処理」は書き込みトランザクション完了までに単一サーバのみの動作に対して各レプリケーション動作に新たに加わる処理を表しています。このとき、[]内に処理を行うサーバを示しています。

表 5.1 レプリケーション方式の比較

| | ネイティブレプリケーション | ストリーミングレプリケーション(非同期) | ストリーミングレプリケーション(apply_remote同期) |
|---|---|---|---|
| PostgreSQLの機能か | No (Pgpool-IIの機能) | Yes | Yes |
| スタンバイで常にプライマリと同じデータを参照できるか | Yes | No | Yes |
| 書き込みトランザクション完了までに必要な処理 | [プライマリ]スタンバイへクエリ発行、[スタンバイ]クエリ実行、[プライマリ]スタンバイからの実行応答待ち | [プライマリ]スタンバイへWAL送信 | [プライマリ]スタンバイへWAL送信、[スタンバイ]WAL書き込み、[スタンバイ]WAL適用、[プライマリ]スタンバイからの適用応答待ち |

## 5.3. 検証構成

### 5.3.1. 検証構成概要

検証構成は2012年度検証に倣います。クライアントマシン(pgbench), サーバマシン(Pgpool-II と PostgreSQL が同居) をそれぞれ 1~4台で下記の構成を取り、それぞれの構成で各シナリオのパフォーマンスについて検証しました。クライアントがアクセスする Pgpool-II サーバは常に同じです。

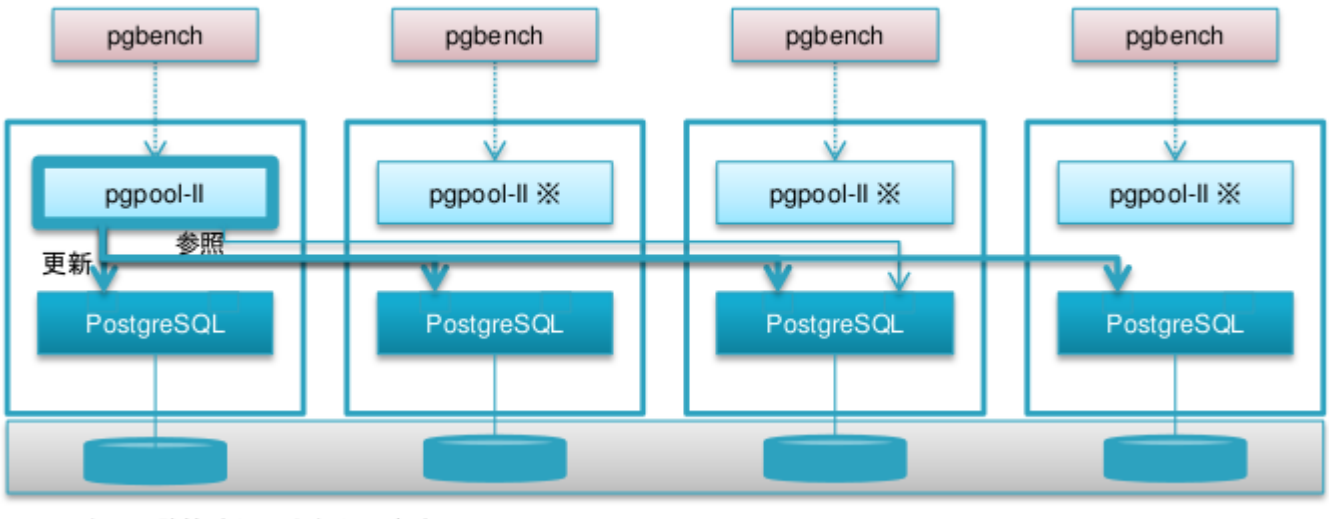



図 5.5 検証マシン構成 ※ 2 台目以降の Pgpool-II・PostgreSQL間の矢印は省略

### 5.3.2. ハードウェア構成

表 5.2 ハードウェア構成

| | |
|---|---|
| サーバ | AWS EC2 m4.xlarge |
| メモリ | 16GB |
| CPU | 4プロセッサ |

### 5.3.3. ソフトウェア構成

表 5.3 ソフトウェア構成

| | |
|---|---|
| PostgreSQL | 9.6.1 |
| Pgpool-II | 3.6.0 |

### 5.3.4. カーネル設定

- OS デフォルト
- ハイパースレッディングは利用しない

## 5.4. 検証方法

下記の比較対象をそれぞれ組み合わせて、pgbench にて TPS を計測しました。

- レプリケーション構成
    - ネイティブレプリケーション
    - ストリーミングレプリケーション（非同期）
        - ただし、稼働スタンバイノードは全て非同期
    - ストリーミングレプリケーション（remote_apply同期）
        - ただし、稼働スタンバイノードは全てremote_apply同期
- pgbench シナリオ構成
    - 更新系（pgbench デフォルトシナリオ）
    - 参照系（カスタムシナリオ）(※)
- バックエンドサーバ台数 (= クライアント台数)
    - 1 ~ 4

(※)参照系カスタムシナリオ

```
\set nbranches :scale
\set ntellers 10 * :scale
\set naccounts 100000 * :scale
\set row_count 10000
\set aid_max :naccounts - :row_count
\set aid random(1, :aid_max)
\set bid random(1, :nbranches)
\set tid random(1, :ntellers)
\set delta random(-5000, 5000)

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

### 5.4.1. 設定

#### 5.4.1.1. 各レプリケーション方式における共通設定

- PostgreSQL 設定 (postgresql.conf)

```
shared_buffers = 8GB
wal_level = replica
```

- Pgpool-II 設定 (pgpool.conf)

```
backend_hostname0 = '192.168.1.30'
backend_port0 = 5432
backend_weight0 = 1
backend_data_directory0 = '/home/postgres/data'
backend_flag0 = 'ALLOW_TO_FAILOVER'
# 同様にバックエンド4台分登録
```

- スタンバイサーバ設定 (recovery.conf)

```
standby_mode = 'on'
primary_conninfo = 'application_name=s1 user=postgres host=<プライマリサーバホスト> port=5432
sslmode=disable sslcompression=1'
primary_slot_name = 's1'
# primary_slot_name, application_name は各スタンバイで s1 - s4 として設定
```

#### 5.4.1.2. ネイティブレプリケーション

- Pgpool-II 設定 (pgpool.conf)

```
replication_mode = on
master_slave_mode = off
```

#### 5.4.1.3. ストリーミングレプリケーション(非同期)

- Pgpool-II 設定 (pgpool.conf)

```
replication_mode = off
master_slave_mode = on
master_slave_sub_mode = stream
```

#### 5.4.1.4. ストリーミングレプリケーション(remote_apply同期)

- PostgreSQL 設定 (postgresql.conf)

```
# 稼働スタンバイノードは全て同期させる
(1ノード検証の場合)
synchronous_standby_names = ''
(2ノード検証の場合)
synchronous_standby_names = '1 (s1,s2,s3)' (※ ただし、s2, s3 は停止しておく)
(3ノード検証の場合)
synchronous_standby_names = '2 (s1,s2,s3)' (※ ただし、s3 は停止しておく)
(4ノード検証の場合)
synchronous_standby_names = '3 (s1,s2,s3)'
```

- Pgpool-II 設定 (pgpool.conf)

```
replication_mode = off
master_slave_mode = on
master_slave_sub_mode = stream
```

### 5.4.2. 環境作成

pgbench のデータベースサイズは 2012 年度検証に合わせて shared_buffers (8GB) に入るサイズの 7.5GB としました。

```
$ initdb --no-locale --encoding=utf8
$ pgbench -i -s 500 test # 約7.5GB
```

### 5.4.3. 検証実施

pgbench 実行前に pg_prewarm を実行します。これによりテーブルデータはすべてバッファキャッシュに格納されます。

```
$ psql -c "select pg_prewarm('pgbench_accounts')" test
```

Pgpool-II サーバ台数と同数のクライアントマシンから同時に pgbench を走らせました。(更新系シナリオの場合)

```
$ pgbench -c 100 -T 300 -j 4 -d test
```

(参照系シナリオの場合)

```
$ pgbench -c 100 -T 300 -j 4 -d test -f ${custom_script}
```

pgbench の設定は実行スクリプトを除き、以下の値で固定しています。

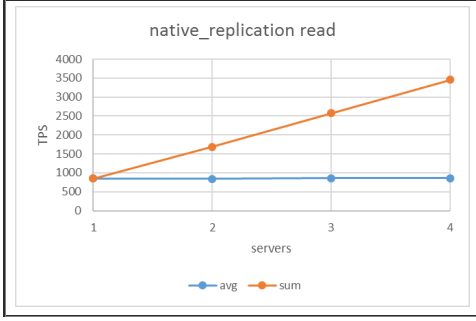
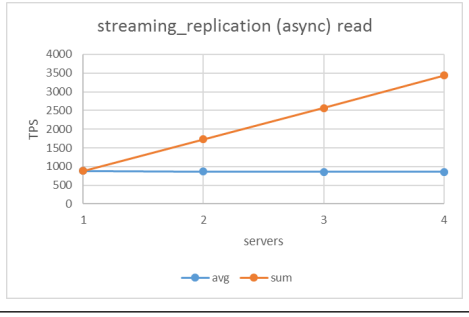
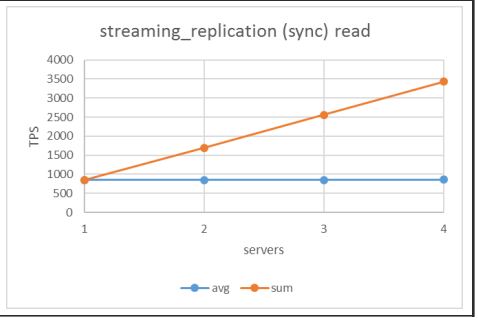
- クライアント接続数(-c): 100
- 実行時間(-T): 300 秒
- スレッド数(-j): 4

## 5.5. 検証結果

### 5.5.1. 参照系

参照系シナリオの結果は下図のようになりました。いずれのレプリケーション方式でも同様にサーバ台数が増えるにつれてTPSが比例して上昇しており、異なるレプリケーション方式においてほぼ同じ値を示していました。

表 5.4 参照系

| ネイティブレプリケーション | 非同期ストリーミングレプリケーション | remote_apply同期ストリーミングレプリケーション |
|---|---|---|
|  |  |  |



また、参考として Pgpool-II を経由しない参照性能も計測しています。このとき、サーバとクライアントはともに1台の条件で比較しています。

下図の通り、Pgpool-II を経由することに依る影響により参照性能が最大で7%低下しています。

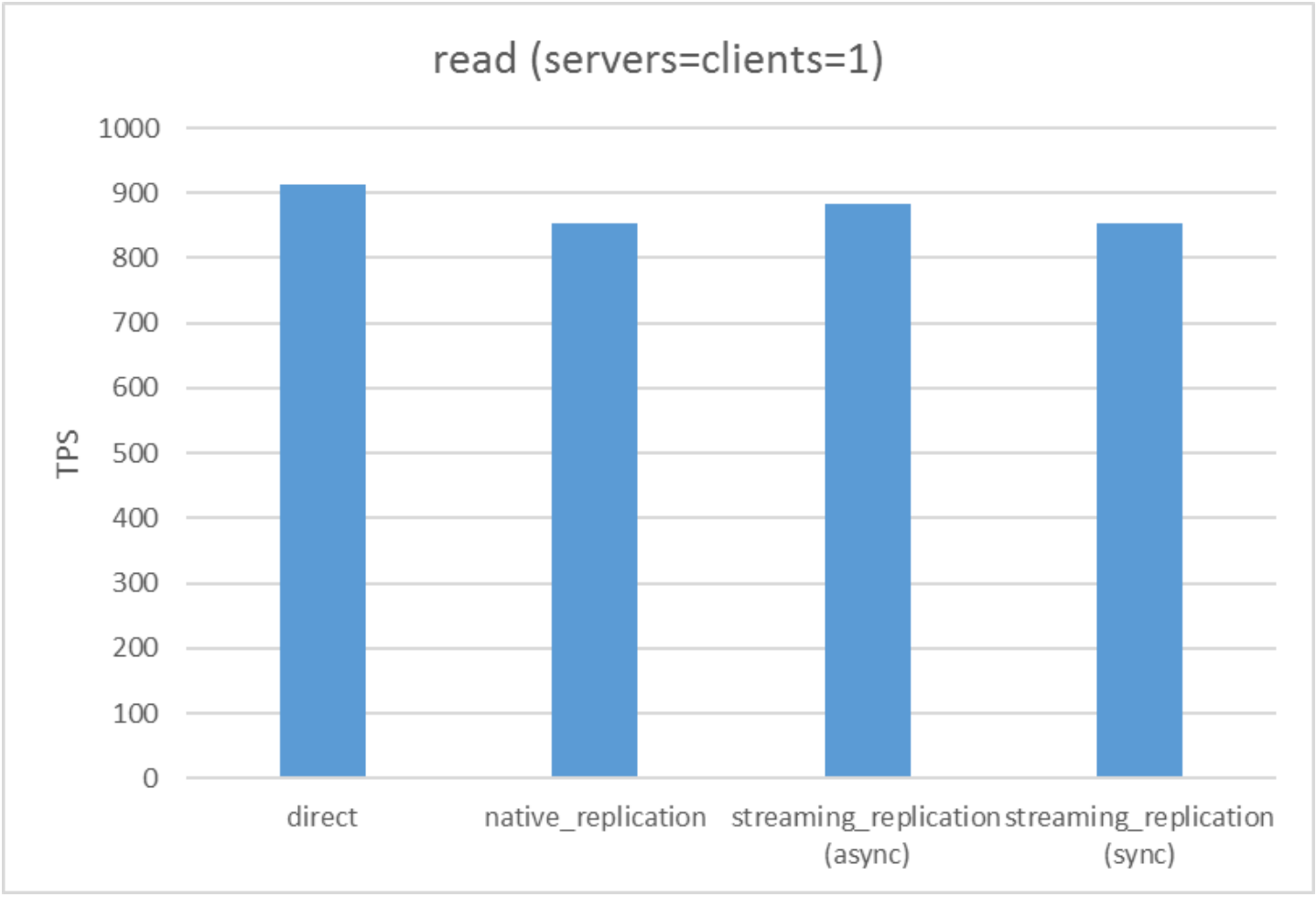


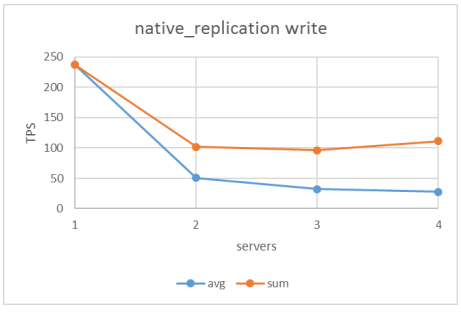
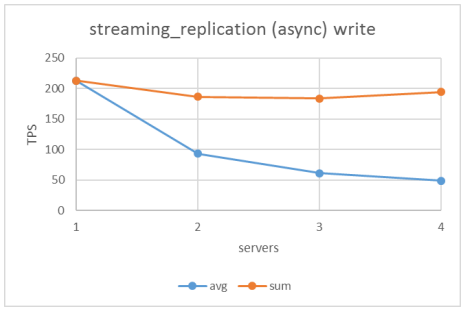
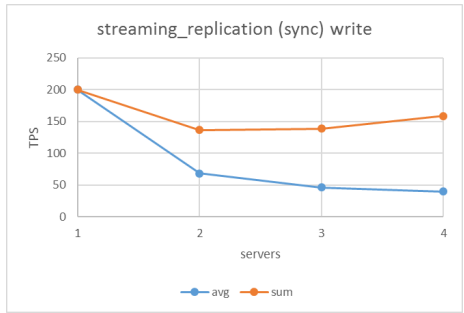

図 5.6 Pgpoolを経由しない参照性能(サーバとクライアントは1台ずつ)

### 5.5.2. 更新系

更新系シナリオの結果は下図のようになりました。非同期ストリーミングレプリケーションはサーバ数が増えても全体の更新性能は劣化しない反面、ネイティブレプリケーションとremote_apply同期ストリーミングレプリケーションは台数が増えるにつれて全体の更新性能は下がっていました。

しかし、ネイティブレプリケーションとremote_apply同期ストリーミングレプリケーションを比較するとスタンバイが存在する(ノード数が2以上の)場合、remote_apply同期ストリーミングレプリケーションの方が高い値を出していました。

表 5.5 更新系

| ネイティブレプリケーション | 非同期ストリーミングレプリケーション | remote_apply同期ストリーミングレプリケーション |
|---|---|---|
|  |  |  |



また、参考として Pgpool-II を経由しない更新性能も計測しています。このとき、サーバとクライアントはともに1台の条件で比較しています。

下図の通り、Pgpool-II を経由することに依る影響により更新性能が最大で16%低下しています。

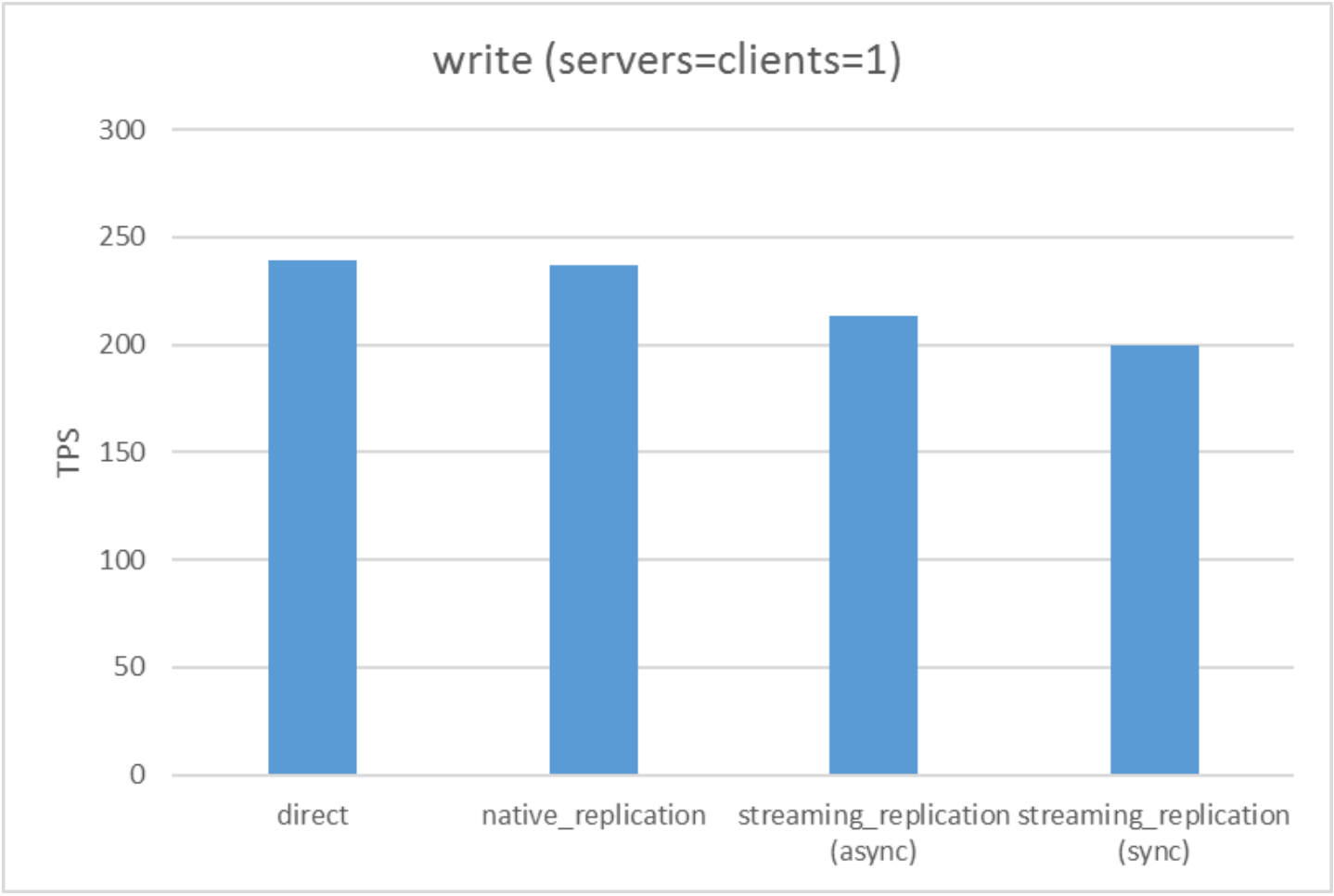



図 5.7 (参考) Pgpoolを経由しない更新性能(サーバとクライアントは1台ずつ)

## 5.6. 考察

本検証では2012年度検証の再試及び、Pgpool-IIで扱える複数のレプリケーション方式について性能比較をしました。その結果、いずれのレプリケーション方式においても参照系と更新系で大きく性能特性が異なりました。

### 5.6.1. 参照系

各レプリケーション方式に性能差はほとんど無く、同様にサーバ台数に比例するスケールアウト性能を示していました。これは、いずれのレプリケーション方式においても負荷分散の実装方式が同じで、参照系クエリは各サーバにおいて平等に並列実行できることから予測される結果でありました。ネイティブレプリケーションにおいて、この結果は2012年度検証と同様の結果でありました。

### 5.6.2. 更新系

サーバ台数が増加するにつれ、各レプリケーション方式の性能差が明らかになりました。まず、非同期ストリーミングレプリケーションでは、サーバ台数が増加しても1台(レプリケーションなし)のときから更新性能はほとんど変化はありませんでした。これは、非同期設定によりスタンバイの応答待ちがない、かつ更新クエリはプライマリサーバで実行される必要があることから予測される結果(同期による遅延もないがスケールアウトもしない)でした。また、サーバ台数が増えるごとにWAL送受信に関する負荷も増える懸念はありましたが、本結果からはその影響は軽微であるといえます。

ネイティブレプリケーションとremote_apply同期ストリーミングレプリケーションでは、サーバ台数が2台(スタンバイが1台)になると全体の更新性能が下がりました。この結果はスタンバイからの応答をプライマリサーバが待たなければならないことから予測される結果でありました。ネイティブレプリケーションにおいて、この結果も2012年度検証と同様の結果でありました。

また、スタンバイ台数が2台3台と増えたときはスタンバイ1台のときとほぼ同じ性能を示していましたが、これはスタンバイが2台以上の場合レプリケーション処理が並列に実行されるので、1台のときに比べて性能低下が少ないためです。

しかし、同じ同期レプリケーションでもremote_apply同期ストリーミングレプリケーションはネイティブレプリケーションに比べて、性能低下は小さいという結果でした。これは両者のレプリケーション実装方式において、ネイティブレプリケーションはスタンバイにクエリを送信および実行させているのに対し、remote_apply同期ストリーミングレプリケーションはWAL送信および適用していることといった違いによるものと考えられます。

### 5.6.3. まとめ

上記の結果から、性能特性ではネイティブレプリケーションよりもremote_apply同期ストリーミングレプリケーションが優れていることが示されました。また、機能特性としてネイティブレプリケーションよりもストリーミングレプリケーションが優れている事柄として以下の点が挙げられます。

- 使用できるSQLに制限がない (ネイティブレプリケーションには一部制約あり)
- 同一クラスタに非同期スタンバイと同期スタンバイを混在させることができる
- レプリケーションスロットなど、PostgreSQLのレプリケーション機能を補助する機能が利用できる

以前はスタンバイがプライマリと同じデータを参照できる状態まで待つ同期レプリケーションを実現するレプリケーション方式として Pgpool-IIのネイティブレプリケーションは一つの選択肢でした。しかし、現在では9.6で登場したremote_apply同期ストリーミングレプリケーションがPgpool-II利用においても優位性を持っているといえます。

今後Pgpool-IIで利用するレプリケーション方式は以下のような使い分けが考えられます。

- レプリケーション遅延を問題とせず更新性能を優先する要件であれば、非同期ストリーミングレプリケーション
- レプリケーション遅延が問題となりバックエンドがPostgreSQL9.6以降であれば、remote_apply同期ストリーミングレプリケーション
- レプリケーション遅延が問題となりバックエンドがPostgreSQL9.5以前であれば、ネイティブレプリケーション

ただし、いずれのレプリケーション方式も参照性能に関しては大きな差はありません。

# 6. JSON/JSONB

## 6.1. 検証概要

IoT(Internet of Things)の一種として、多くののセンサデータを格納して分析するケースがあります。さまざまなセンサデータを集約するケースでは、データ構造(スキーマ)を事前に決定することができず、RDBMSでデータを管理することが難しい場合があります。

PostgreSQLでは、以前からスキーマレスなデータを格納する手段を提供しています。特に、バージョン 9.2以降、JSONデータを管理する仕組みが追加されました。

- 9.2: JSON型が追加
- 9.3: JSON型に対する各種演算機能が追加
- 9.4: JSONをバイナリ格納するJSONB型が追加

本章では、PostgreSQLのJSON型/JSONB型を利用して、センサデータのような多様なデータをPostgreSQLで管理可能か検証した結果を報告します。そして、検証の結果、IoT用途にPostgreSQLを適用する場合、RDBのテーブル定義としてどのようなスキーマが適切かを考察します。

今回の検証では、以下の3つの観点で評価しました。

- データ挿入性能
- データサイズ
- 検索性能

## 6.2. 検証構成

### 6.2.1. 検証サーバ

今回の検証環境の構成を以下に示します。

表 6.1 検証構成

| 機器 | 項目 | 仕様 |
|---|---|---|
| PostgreSQL用サーバ | CPU | インテル Xeonプロセッサ E5-2690v4@2.60GHz<br>(14コア)x2 合計 28コア |
| | 搭載メモリ | 256GB |
| | 内蔵ストレージ | HDD 1.2TB SAS 10K x 16 |
| | DB格納用ストレージ | Fiber Channel接続(16Gbps) SAN<br>以下の2領域を利用<br>領域1: PostgreSQLのDB領域<br>領域2: WAL領域<br>それぞれの領域はHDD 1.8TB 6G SAS 10K x 4<br>(RAID10、実効容量約3TB)で構成 |
| | OS | Red Hat Enterprise Linux 7.2 |
| | DBMS | PostgreSQL 9.6.1 |

### 6.2.2. PostgreSQL

今回の検証では検証時期時点での最新バージョンPostgreSQL 9.6.1を使用しました。PostgreSQL設定は以下を変更しています。

表 6.2 修正したPostgreSQLパラメータ

| | |
|---|---|
| **shared_buffers** | 26GB |
| **effective_cache_size** | 128GB |

## 6.3. 検証方法

### 6.3.1. テーブル定義

今回の検証では7種類のモデルを使用しました。

表 6.3 検証モデル

| | |
|---|---|
| **RDBモデル** | センサデータ内の各要素を通常のPostgreSQLの通常のデータ型として表現したモデル |
| **JSONモデル** | センサデータ全体をPostgreSQLのJSONデータとして表現したモデル |
| **JSONBモデル** | センサデータ全体をPostgreSQLのJSONBデータとして表現したモデル |
| **ハイブリッドJSONモデル** | 必ず検索キーとして使用する要素をPostgreSQLの通常のデータ型、残りの要素はPostgreSQLのJSONデータとして表現したモデル |
| **ハイブリッドJSONBモデル** | 必ず検索キーとして使用する要素をPostgreSQLの通常のデータ型、残りの要素はPostgreSQLのJSONデータとして表現したモデル。 |

以下に各テーブル定義(CREATE TABLE文)を示します。

リスト 6.1 RDBモデルのテーブル定義

```
CREATE TABLE IF NOT EXISTS iot (
  deviceID char(10),
  ts timestamp,
  Latitude real,
  Longitude real,
  Temparature real,
  Humidite real,
  CO2 real,
  AtmospheiricPressure real
);
```

リスト 6.2 JSONモデルのテーブル定義

```
CREATE TABLE IF NOT EXISTS iot_json (
  info json
);
```

リスト 6.3 JSONBモデルのテーブル定義

```
CREATE TABLE IF NOT EXISTS iot_jsonb (
  info jsonb
);
```

リスト 6.4 ハイブリッドJSONモデルのテーブル定義

```
CREATE TABLE IF NOT EXISTS iot_part_json (
  deviceID char(10),
  ts timestamp,
  info json
);
```

リスト 6.5 ハイブリッドJSONBモデルのテーブル定義

```
CREATE TABLE IF NOT EXISTS iot_part_jsonb (
  deviceID char(10),
  ts timestamp,
  info jsonb
);
```

### 6.3.2. インデックス定義

今回の検証では以下の2つのパターンのインデックスを設定しました。

- B-treeインデックス
- BRIN(Block Range INdex)

B-TreeインデックスはPostgreSQLのデフォルトインデックス種別です。BRINはPostgreSQL 9.5から導入された特殊なインデックスです。B-treeインデックスのように汎用的なインデックスではありませんが、値の論理的な順序とデータベースへの物理的な格納順序に強い相関関係があるデータに対して効果的なインデックスとなります。例えば、

タイムスタンプ情報をもつログデータなどを蓄積するようなケースに於いて、タイムスタンプ情報に対して設定するケースが該当します。

どちらのインデックスもタイムスタンプ情報を対象とします。JSONモデルおよびJSONBモデルでは、JSON/JSONB演算子を用いて取り出したタイムスタンプ情報に対して式インデックスを設定します。

なお、JSONB型に対してはGIN(汎用転置インデックス)を設定することも可能ですが、今回の検証モデルでは検索条件として使うキーが固定であり、JSONBデータ内の全ての名前と値の組をインデックス化対象とするGINインデックスを使用することは適切ではないと考え、検証対象からは除外しました。

### 6.3.3. データ挿入性能

今回の検証では、以下の想定でデータ挿入のモデルを検討しました。

- 一つのセンサが10秒に1回、気温等のデータを送信すると仮定。
- そのデータを100機器分、3年間蓄積したと仮定。
- データ件数は約9.5億件。

このデータを以下の方式で挿入し、そのときの時間を測定しました。

- ロード時間(COPY FROM)。
- ロード後にインデックス作成にかかる時間。

今回の検証では1回のみ測定を行っています。

### 6.3.4. データ検索性能

今回の検証では、以下の想定でデータ検索のモデルを検討しました。

- 最近の(データ上は2019年10月)一ヶ月間のある機器(iot-00050)が収集したデータを検索対象として、以下の2つのパターンの検索性能を測定しました。
- 単純検索：全データを上記条件で検索しし、その件数を取得。
- 平均気温取得：全データを上記条件で検索し、気温の平均(AVG)をとる。
- 検証結果は3回測定したときの中間値(Median)を使用しました。

## 6.4. 検証結果

### 6.4.1. 格納性能とテーブルサイズ

表 6.4 データロード時間(秒)

| | |
|---|---|
| **RDBモデル** | 2954.592 |
| **JSONモデル** | 4965.902 |
| **JSONBモデル** | 5742.353 |
| **ハイブリッドJSONモデル** | 3433.817 |
| **ハイブリッドJSONBモデル** | 4920.315 |

表 6.5 テーブルサイズ(MB)

| | |
|---|---|
| **RDBモデル** | 69077.1 |
| **JSONモデル** | 194506.6 |
| **JSONBモデル** | 254870.7 |
| **ハイブリッドJSONモデル** | 164250.0 |
| **ハイブリッドJSONBモデル** | 217389.7 |

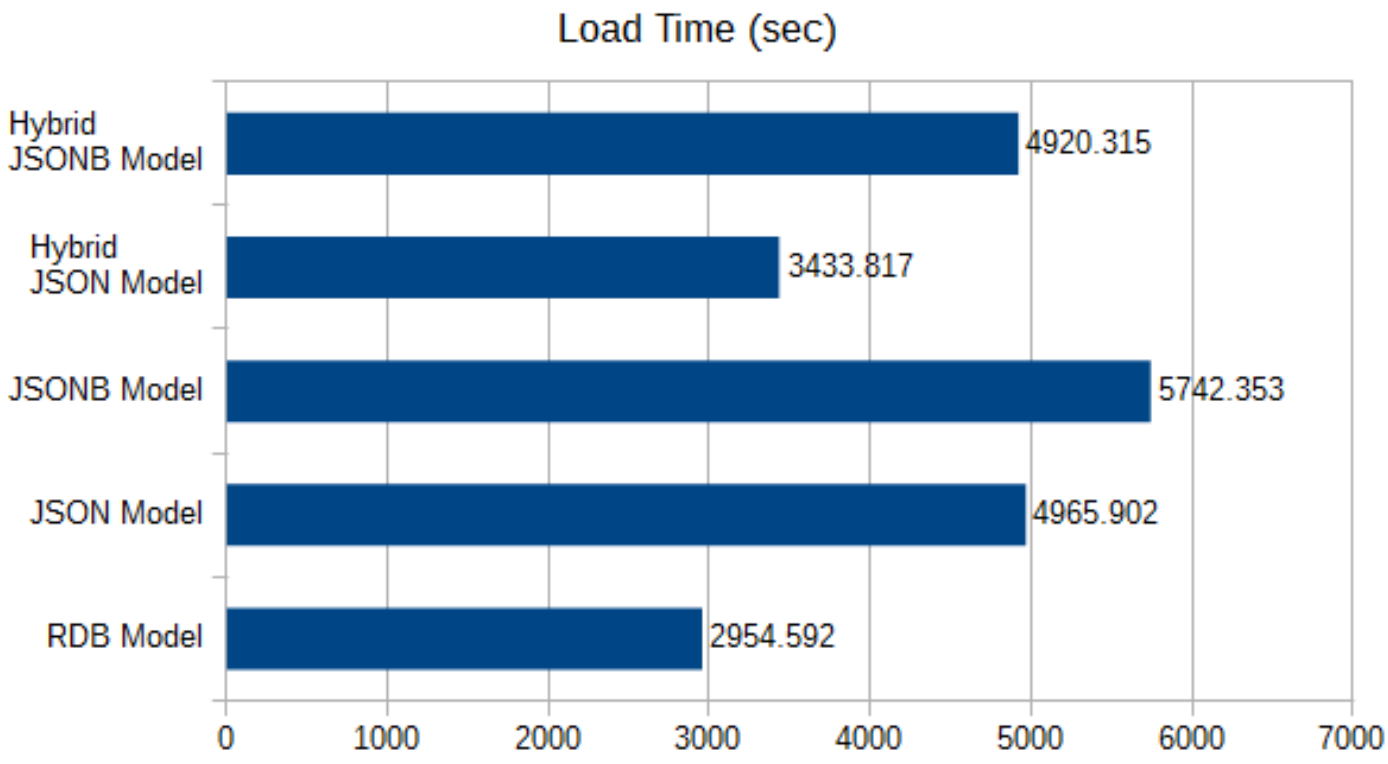



図 6.1 データロード時間

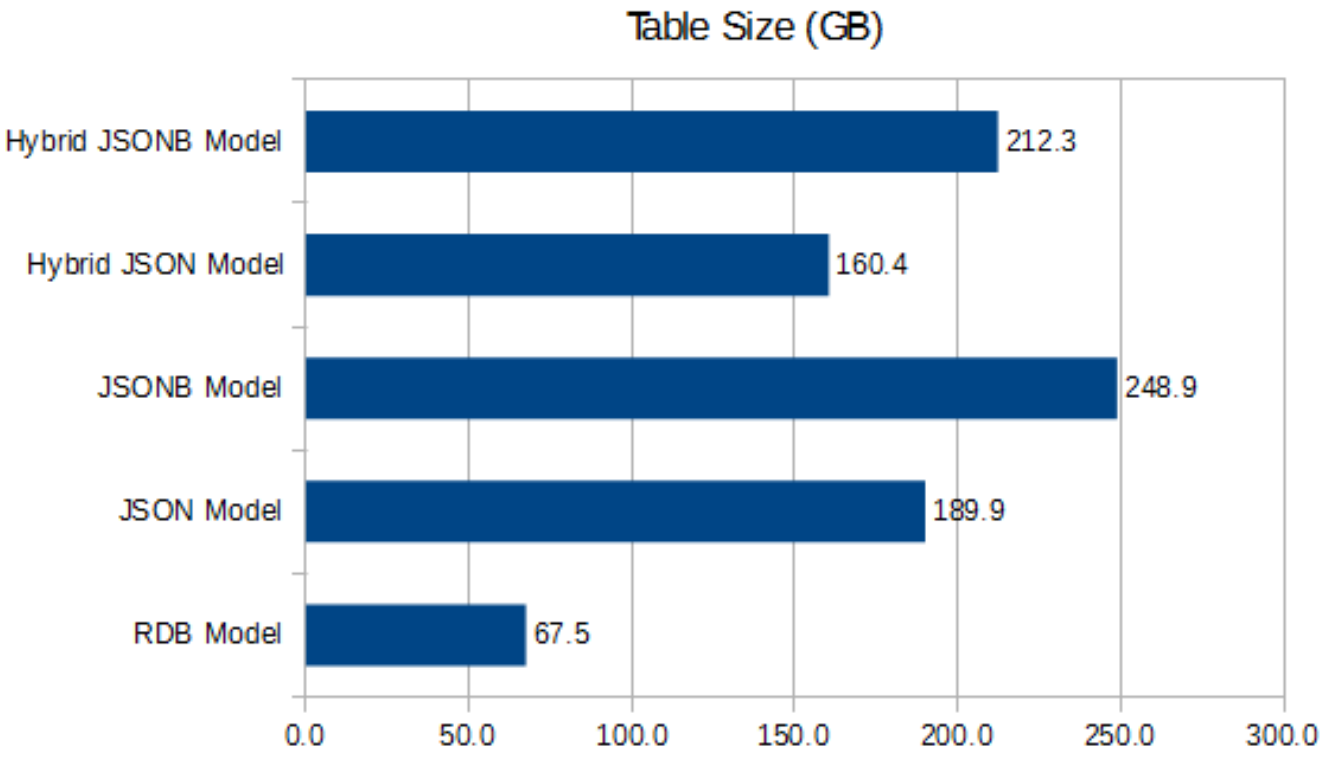



図 6.2 テーブルサイズ

### 6.4.2. インデックス作成性能とインデックスサイズ

表 6.6 インデックス作成時間(秒)

| 方式 | B-tree | BRIN |
|---|---|---|
| RDBモデル | 641.751 | 153.347 |
| JSONモデル | 2472.123 | 1330.105 |
| JSONBモデル | 1672.458 | 435.233 |
| ハイブリッドJSONモデル | 822.484 | 227.385 |
| ハイブリッドJSONBモデル | 924.307 | 312.148 |

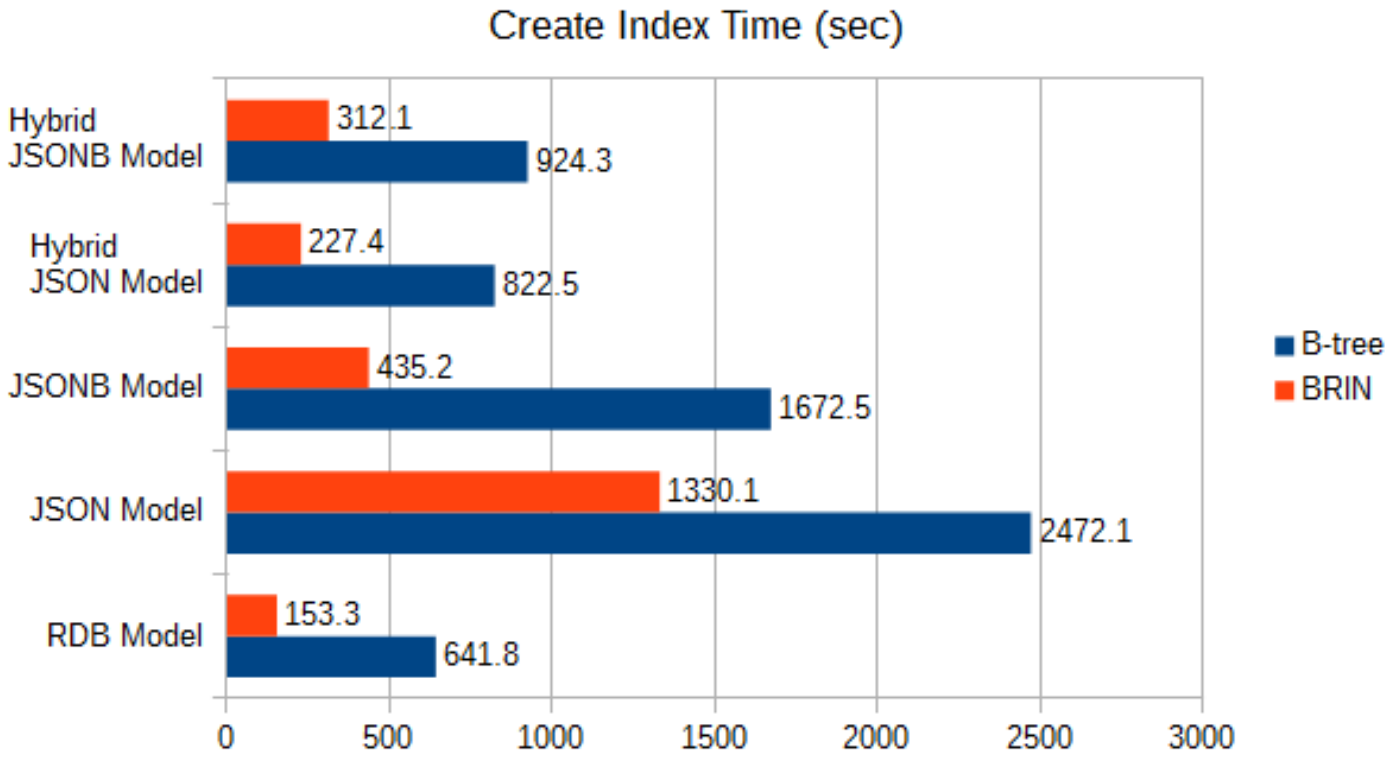



図 6.3 インデックス作成時間(秒)

表 6.7 インデックスサイズ(MB)

| 方式 | B-tree | BRIN |
|---|---|---|
| RDBモデル | 20266.1 | 2.3 |
| JSONモデル | 44873.5 | 13.8 |
| JSONBモデル | 44873.5 | 18.1 |
| ハイブリッドJSONモデル | 20266.1 | 5.4 |
| ハイブリッドJSONBモデル | 20266.1 | 7.1 |

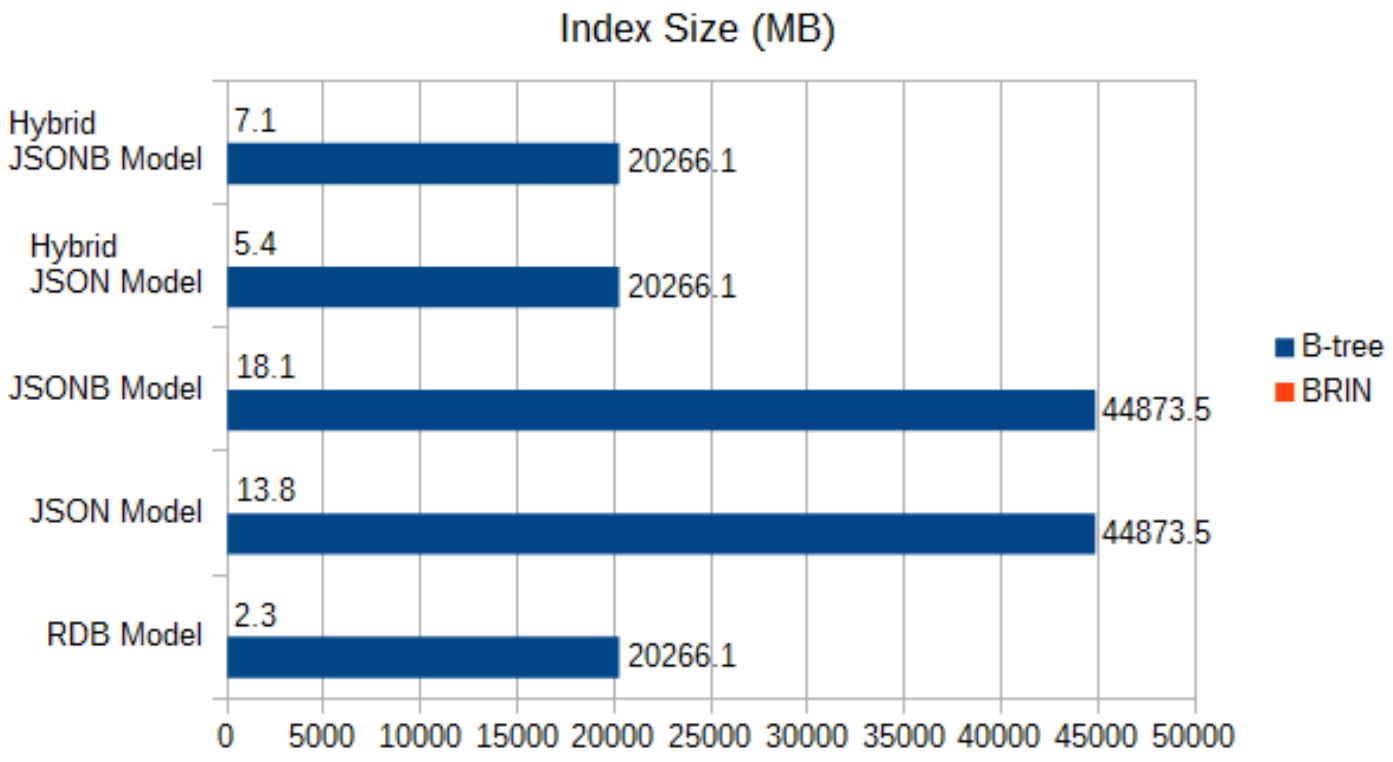



図 6.4 インデックスサイズ(MB)

### 6.4.3. データロードとインデックス作成まとめ

テーブル作成時間とインデックス作成時間を合せた結果を以下に示します。

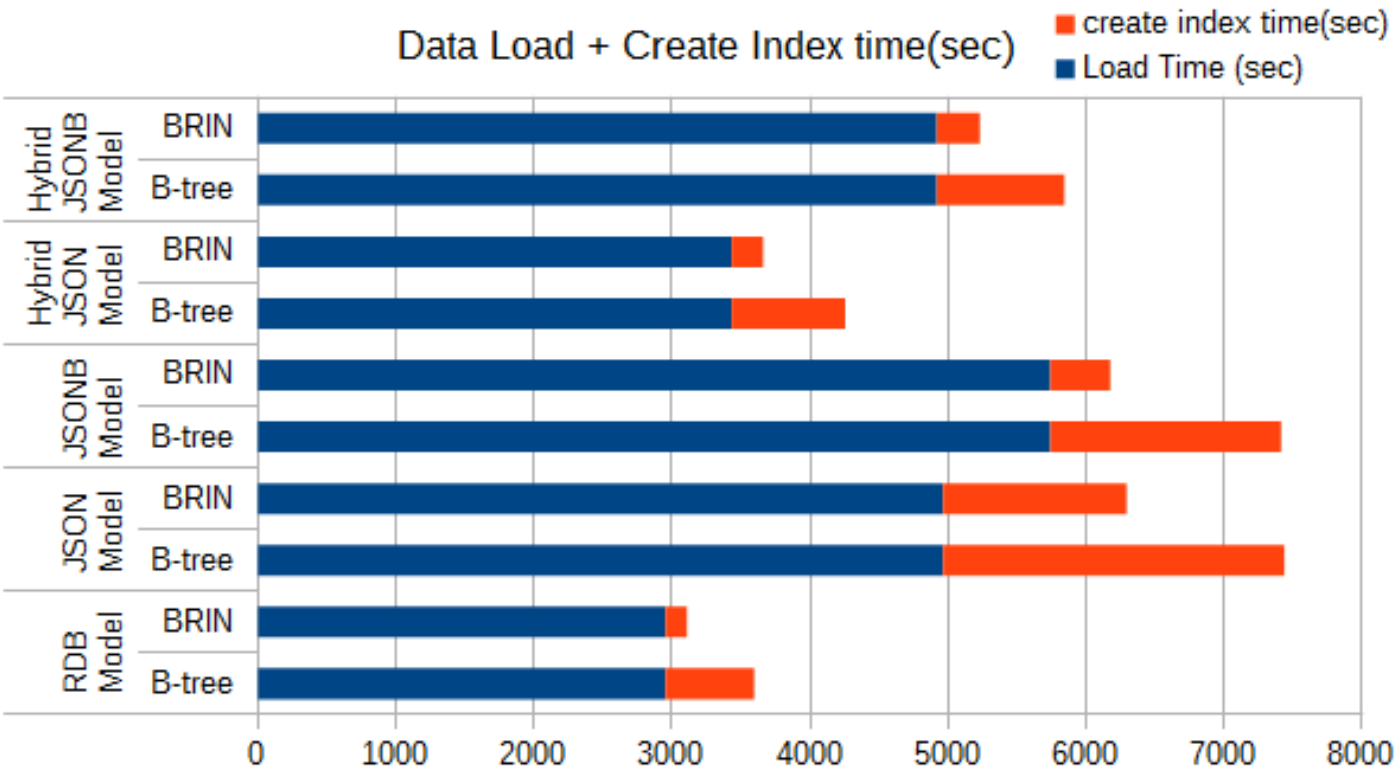



図 6.5 テーブル作成時間＋インデックス作成時間

テーブルサイズとインデックスサイズを合せた結果を以下に示します。

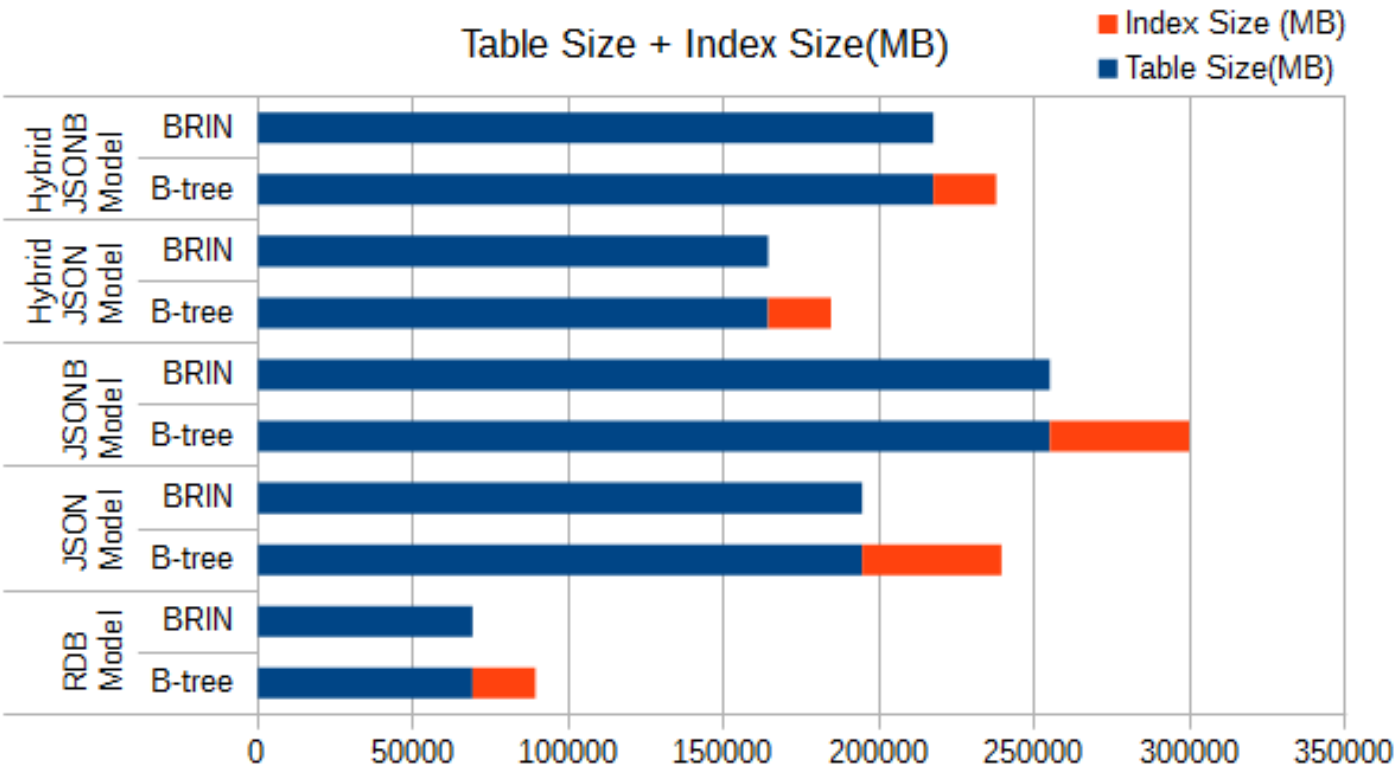



図 6.6 テーブルサイズ＋インデックスサイズ

以下に、データロードとインデックス作成の検証結果から、以下の内容が確認できました。

- データロード時間について
  - ロード時間の長さは想定どおり、通常表＜JSON＜JSONB となります。
  - JSONモデルとJSONBモデルののロード時間差と、インデックス作成時間の総和はほぼ等しくなります。これはJSONBが格納時にシリアライズすることで、ロード時間が遅くなる代わりに、インデックス作成背景で実行される取得処理が高速化されているためです。
- データサイズについて
  - 通常テーブルと比較するとJSONで約3倍、JSONBで約4倍のサイズとなります。
  - 件数が非常に大きい場合、JSON/JSONBのモデルの場合、データサイズが大きくなることに注意する必要があります。

### 6.4.4. 検索性能

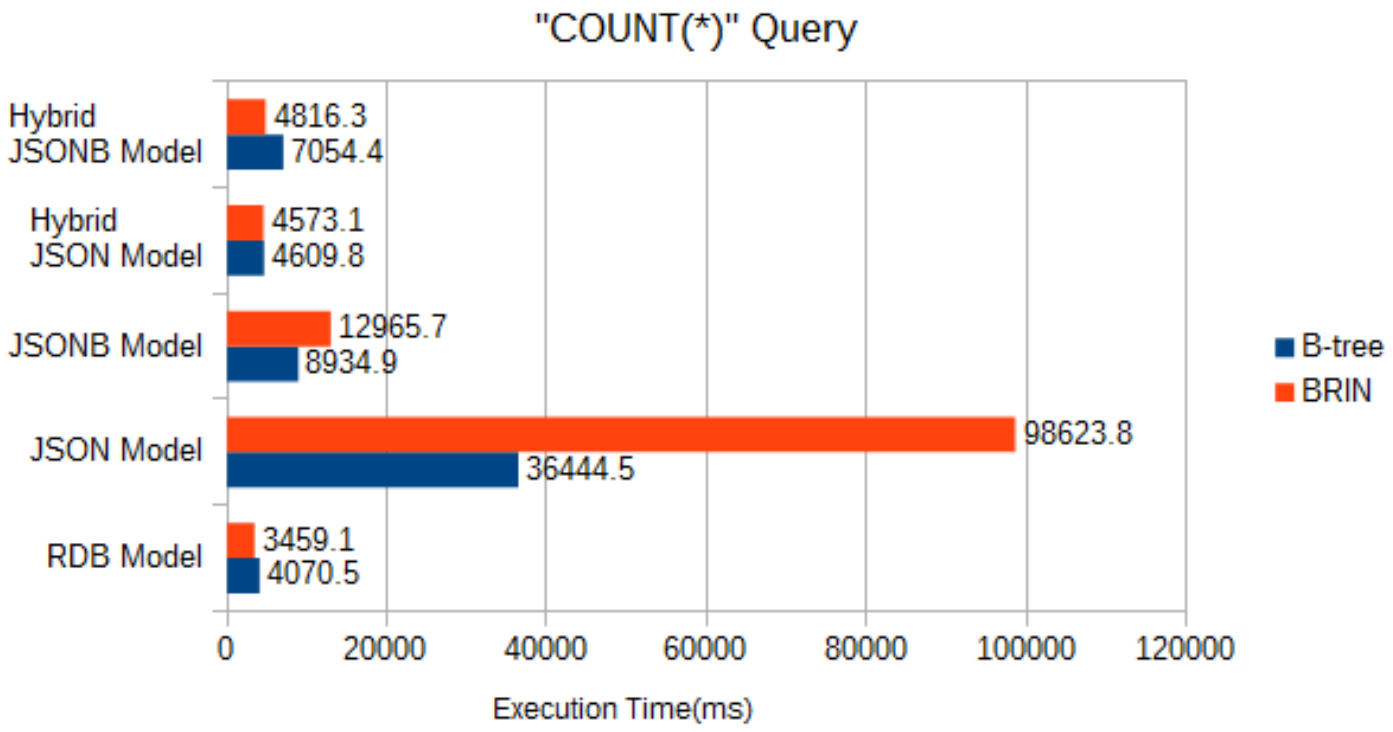



図 6.7 件数カウント検索

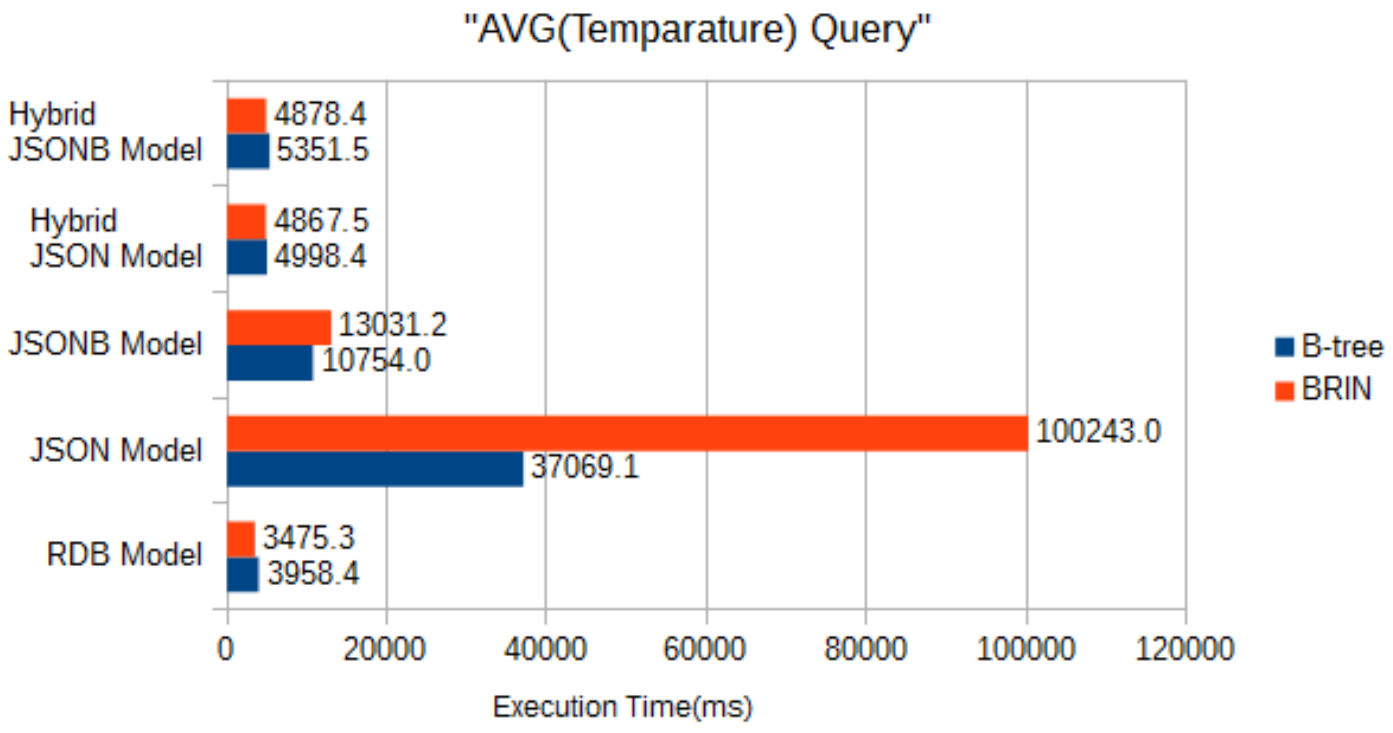



図 6.8 平均気温検索

### 6.4.5. 検索時間のまとめ

検索時間のサマリを示します。

- JSONと比較するとJSONBの検索時の性能はかなり良くなります。
- キーだけ通常列化した場合、通常表の検索性能並になる。また、この場合、JSONBとJSONはほぼ同等の性能となります。
- 今回のモデルのような非常に大きなサイズのテーブルに対して、連続した範囲を検索する場合には、B-treeインデックスとBRINは同程度の検索性能になります。

## 6.5. 考察

今回の検証から、センサデータのようなIoT機器データの収集にPostgreSQL JSON/JSONB型を使用する場合の注意点を列挙します。

- 必ず検索キーとして使う要素が確定している場合、その要素をJSON/JSONB列とは分離して通常の列に持たせることで、性能の低下を抑えつつ、JSONBの特色であるスキーマレスな情報の管理も同時に可能になる。
- 検索キーとして使用しない項目はJSONBではなく、JSONを選択する案もある。
- 数十億件単位の非常に大量なレコード数を蓄積し、かつタイムスタンプ情報のような論理的な値と物理的な格納順序に強い相関があるケースでは、BRINインデックスが有効である。

# 7. 全文検索

## 7.1. 検証概要

PostgreSQLの適用領域の拡大、また取り扱うデータ量の増加に伴い、PostgreSQLに格納されたデータに対する全文検索の重要性がますます高まっています。

標準のSQLにおいても、LIKE文を用いることで文字列パターンを指定した検索は可能ですが、この場合テーブルのデータを全てシーケンシャルに検索することになるため、検索完了までに多くの時間が必要です。そのため実用面では、PostgreSQLに全文検索用の モジュールを導入し、インデックスを利用した高速な検索を実現する手法が一般的です。

本章では、特に国内においてニーズの高い"日本語検索"に対応した代表的な検索手法である pg_trgm（ピージートリグラム）、pg_bigm（ピージーバイグラム）、PGroonga（ピージールンガ）の特性を様々な視点の検証によって明らかにします。

## 7.2. 検証構成

### 7.2.1. マシン構成

本検証に用いたハードウェア構成を 表． %s 、ソフトウェア構成を 表． %s に示します。今回はデータベースサーバと負荷クライアントは同じ筐体となっています（psqlコマンドをPostgreSQLサーバ上で実行）。

表 7.1 ハードウェア構成

| 項目 | 仕様 |
|---|---|
| 機種 | 富士通PCワークステーション「CELSIUS R670-2」 |
| CPU | インテル Xeonプロセッサ X5650@2.66GHz (6コア)x2 合計 12コア |
| メモリ | 48GB |
| 内蔵ストレージ | HDD 500GB SATA x 2 |

表 7.2 ソフトウェア構成

| ソフトウェア | バージョン | 用途 | 備考 |
|---|---|---|---|
| CentOS | 7.2（64bit） | OS | |
| PostgreSQL | 9.6.1 | データベース | |
| pg_bigm | 1.2 | インデックス | |
| PGroonga | 1.1.9 | インデックス | |
| Groonga | 6.1.5 | インデックス | PGroongaの動作に必要 |

### 7.2.2. 全文検索用モジュール

本検証で利用した全文検索用モジュールを 表． %s に示します。

表 7.3 検証対象の全文検索用モジュール

| 名称 | 説明 | ライセンス | 解析方法 |
|---|---|---|---|
| pg_trgm | PostgreSQLのcontribモジュールに同梱されており、標準で利用可能 | PostgreSQL License | 3-gram |
| pg_bigm | 日本語に対応した全文検索機能を提供するモジュール | PostgreSQL License | 2-gram |
| PGroonga | 日本語に対応した全文検索機能として実績の豊富なGroongaをPostgreSQLから利用可能にするモジュール | PostgreSQL License | N-gram |

### 7.2.3. postgresql.confの設定値

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

表 7.4 postgresql.confの設定値

| パラメータ | 設定値 | 考え方 |
|---|---|---|
| max_connections | 100 | |
| shared_bufferes | 12GB | 搭載メモリの1/4を割り当て |
| max_wal_size | 1GB | |
| maintenance_work_mem | 4GB | インデックス作成の際に必要となる領域のため、多めに確保 |

## 7.3. 検証方法

### 7.3.1. 全文検索対象のデータ

本検証の検索対象は、日本語版Wikipediaの文章データを抽出して、PostgreSQLへtext型として格納したものです。表． %s にテーブル構造とデータの例を示します。

表 7.5 テーブル名「Wikipedia」

| id（integer型） | text（text型） |
|---|---|
| 1 | PostgreSQL（ぽすとぐれすきゅーえる：発音例）は、オープンソースのオブジェクト関係データベース管理システム(RDBMS)である。... |
| 2 | MYSQL（まい・えすきゅーえる）は、オープンソースで公開されている、関係データベースサーバの一つである。... |
| 3 | ... |

Wikipediaは、利用している文章データを全てダウンロード可能な形式で公開しています (https://ja.wikipedia.org/wiki/Wikipedia:データベースダウンロード)。

本検証では、日本語版Wikipdeiaの変更履歴を含む全データを対象とします。ただし、全データは約1TBと非常に容量が大きく、また14ファイルに分割して公開されているため、本検証ではそのうち1ファイル（約79GB）に含まれる文章データ（約41GB）を 一つのテーブルに格納して検索を行っています。

### 7.3.2. インデックスの作成方法

作成したテーブルの「text」カラムに対し、CREATE INDEXコマンドでインデックスを作成します。

- pg_trgmの場合

```
CREATE EXTENSION pg_trgm;
CREATE INDEX idx_text_trgm on wikipedia USING GIN (text gin_trgm_ops);
```

- pg_bigmの場合

```
CREATE EXTENSION pg_bigm;
CREATE INDEX idx_text_bigm on wikipedia USING GIN (text gin_bigm_ops);
```

- PGroongaの場合

```
CREATE EXTENSION pgroonga;
CREATE INDEX idx_text_pgroonga on wikipedia USING GIN (text pgroonga.text_regexp_ops);
```

### 7.3.3. 検索方法

検索キーワードとしては、以下の英語／日本語の単語を利用します。

- 英語のみ: 3文字～12文字の28単語（npm、Ansible、PostgreSQL など）
- 日本語のみ: 2文字～7文字の12単語（日本、東京都、データベース など）

複数の検索キーワードをLIKEを用いて検索します。

```
SELECT COUNT(id) FROM wikipedia WHERE text LIKE '%keyword%';
```

検索に当たっては、以下の条件で測定しています。

- 同一キーワードで3回測定を行い、中央値を結果として採用。なお、キャッシュは毎回クリアする。

```
sync
sysctl vm.drop_caches = 3
```

- 確実にインデックスを利用した検索をするため、シーケンシャルスキャンは「off」にする。

```
SET ENABLE_SEQSCAN=OFF;
```

### 7.3.4. インデックスによる検索ノイズの測定

本検証で対象としている全文検索用モジュールでは、N-gramと呼ばれる方式でインデックスを作成します。インデックス作成の流れを 図. 7.1 に示します。まず、データベース中の文章を数文字で区切り、それらをキーと呼びます。そして、キーが出現する文章のIDとの組をインデックスとして管理します。図. 7.1 では2文字で文章を区切ってキーを作成するため、2-gram（バイグラム）と呼ばれます。同様に、3文字で文章を区切るものは3-gram（トリグラム）と呼ばれます。pg_bigm、pg_trgmはその名の通り、それぞれ2-gram、3-gram方式でインデックスを作成しています。

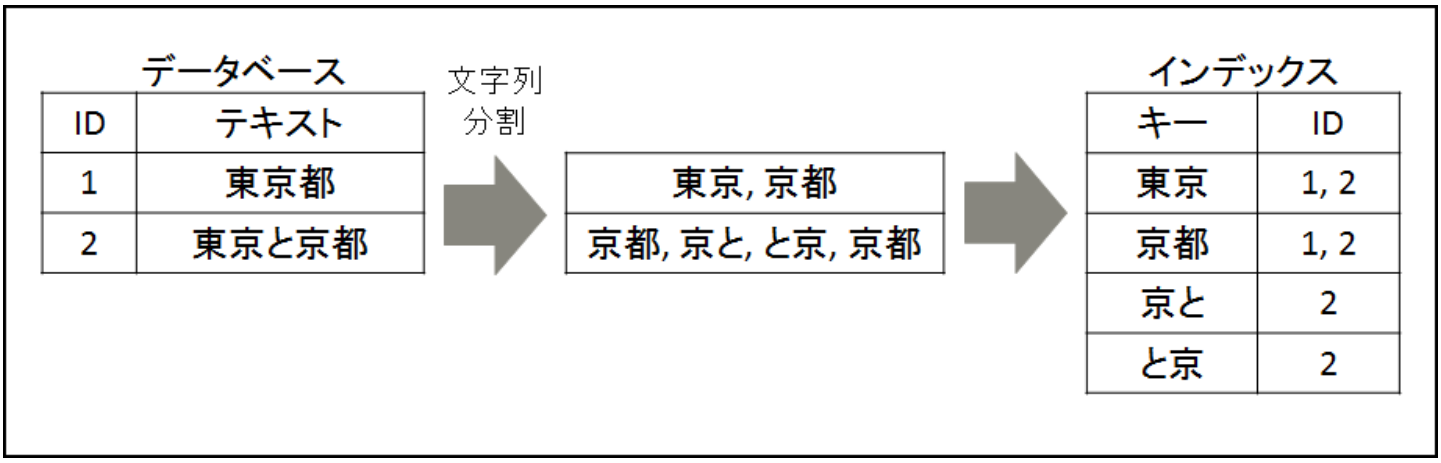



図 7.1 N-gramによるインデックス作成の流れ

検索の際は、キーワードを同じように数文字で区切り、それらが全て出現する文章のIDを 検索結果として返すことで、テーブル全体を走査することなく検索が可能となります。図. 7.2 に、検索時の流れを示します。しかし、N-gramによる検索では、検索ノイズと呼ばれる誤りが発生する場合があります。例えば、キーワード「東京都」で検索を行った場合、図. 7.2 のようにレコード「東京と京都」も 検索結果として扱われることになり、正しい検索結果が得られません。

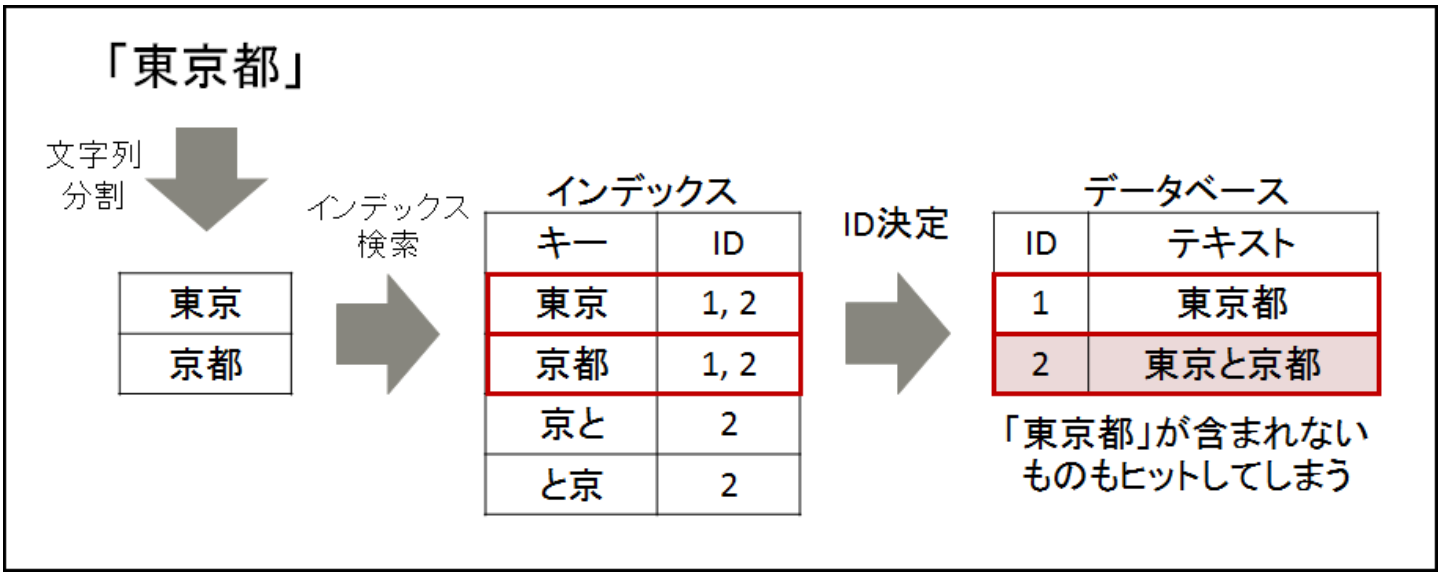



図 7.2 インデックスを利用した検索

この誤りを防ぐために、PostgreSQLではインデックスによる検索結果を候補として扱い、本当にキーワードを含むかを再検査するRecheckと呼ばれる処理を行います。これにより、誤った検索結果を取り除くことが出来ます。

PostgreSQLでは、EXPLAIN ANALYZEを実行することにより、Recheck前の検索件数を 知ることができ、インデックスのみの検索結果の精度を確認することが出来ます。本検証では、Recheck前の検索件数も測定し、Recheck処理が検索時間に与える影響を調べます。

```
EXPLAIN ANALYZE SELECT COUNT(id) FROM wikipedia WHERE text LIKE '%keyword%';
```

# 7.4. 検証結果

## 7.4.1. インデックス作成時間、データ増分の比較

表. 7.6 に各種インデックスにおける インデックスの作成時間、またインデックス作成によるデータの増分を示します。インデックスの作成時間はPGroongaが最も早く、次いでpg_bigm、pg_trgmの順となりました。一方、データ増分はpg_bigmが最も小さく、次いでpg_trgm、PGroongaの順となり、インデックスの作成時間とデータの増分にはあまり関係がなく、インデックスの実装によって 傾向が異なるという結果が得られました。

表 7.6 インデックス作成時間、データ増分比較

| インデックス | インデックス作成時間 | データ増分 |
|---|---|---|
| pg_trgm | 26時間46分 | 20GB |
| pg_bigm | 8時間16分 | 10GB |
| PGroonga | 6時間46分 | 131GB |

## 7.4.2. 英語キーワードによる比較

### 7.4.2.1. 検索時間の比較

図. 7.3 に、英語キーワードによる検索時間を示します。キーワードによりばらつきはありますが、pg_trgm、PGroongaに比べてpg_bigmが検索に時間が かかっている傾向が見られます。

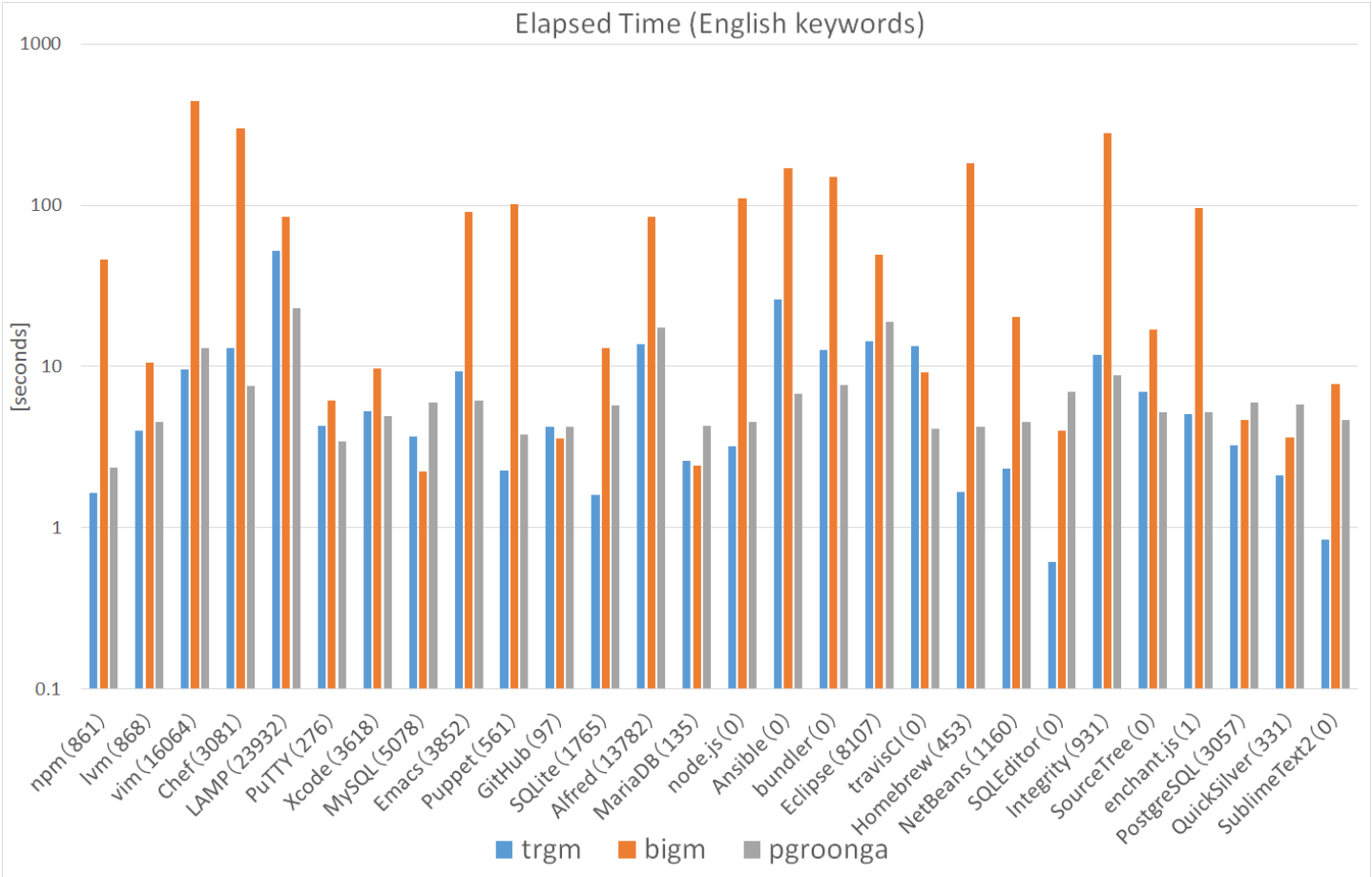



図 7.3 英語キーワードによる検索時間

### 7.4.2.2. 検索時間と件数の相関

今度は、検索時間と件数の相関図を 図. 7.4 に示します。pg_trgm、PGroongaは、おおよそ件数に比例して検索時間が増加していきますが、pg_bigmでは相関が見られない結果となりました。

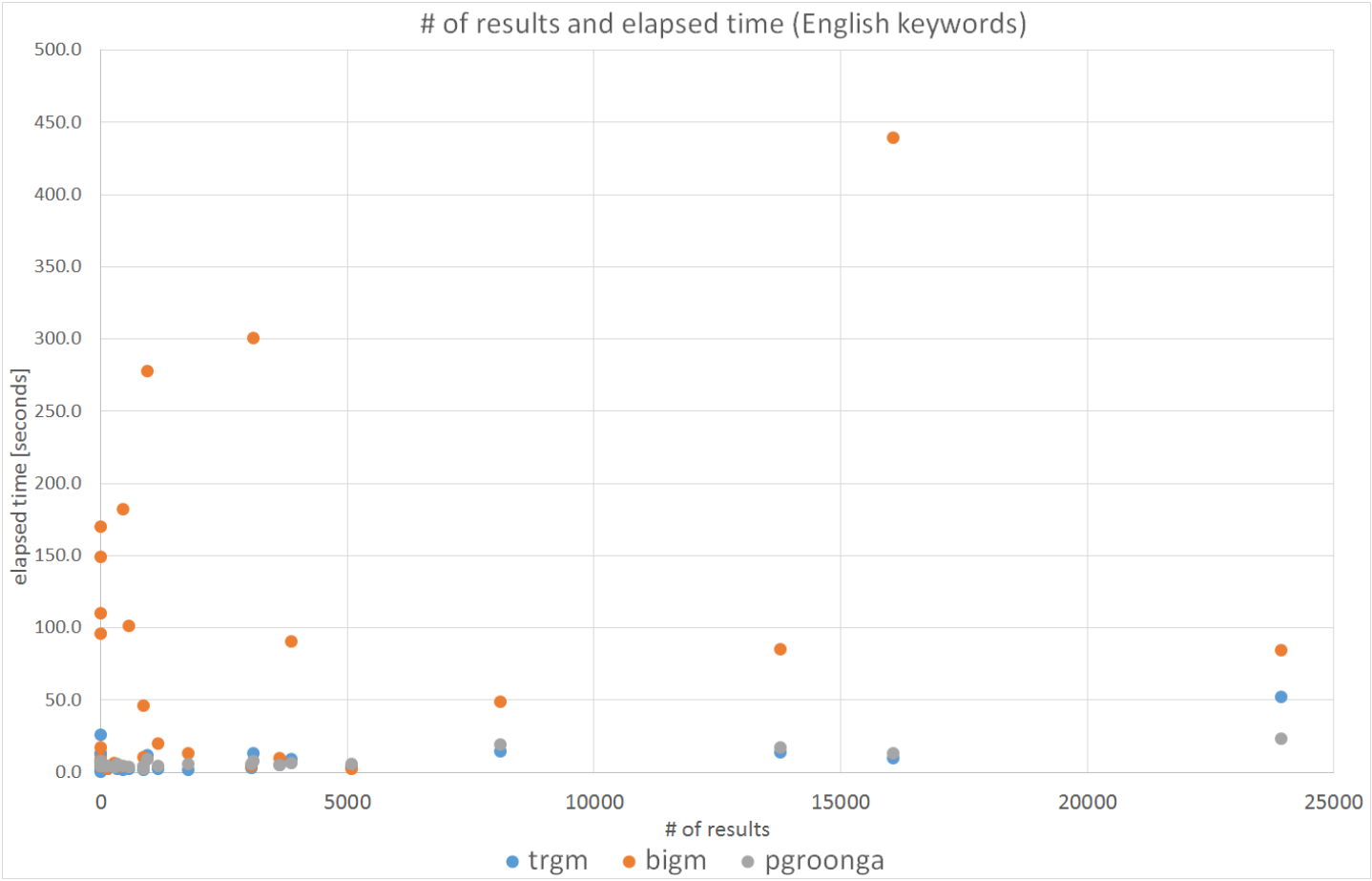



図 7.4 英語キーワードによる検索時間と件数の相関

#### 7.4.2.3. Recheck処理による影響

図. 7.5 は、Recheck処理により 除外された件数をインデックスごと比較したものです。いずれのキーワードにおいてもpg_bigmが最も件数が多く、Recheck処理に時間がかかるため、処理時間の長さに影響を与えているものと考えられます。

| keyword | pg_trgm | pg_bigm | PGroonga |
|---|---|---|---|
| npm | 260 | 33995 | 9 |
| lvm | 2675 | 7165 | 2056 |
| vim | 797 | 938956 | 797 |
| Chef | 9293 | 427832 | 1595 |
| LAMP | 41214 | 89978 | 7852 |
| PuTTY | 4344 | 825 | 69 |
| Xcode | 1780 | 5354 | 0 |
| MySQL | 171 | 136 | 171 |
| Emacs | 4770 | 85489 | 424 |
| Puppet | 499 | 84856 | 384 |
| GitHub | 2424 | 1601 | 448 |
| SQLite | 11 | 8841 | 0 |
| Alfred | 1177 | 60621 | 29 |
| MariaDB | 960 | 629 | 0 |
| node.js | 1504 | 102312 | 46 |
| Ansible | 25777 | 175957 | 87 |
| bundler | 9124 | 127969 | 0 |
| Eclipse | 11084 | 28755 | 10756 |
| travisCI | 8546 | 5324 | 0 |
| Homebrew | 995 | 189115 | 6 |
| NetBeans | 186 | 13740 | 186 |
| SQLEditor | 0 | 1662 | 0 |
| Integrity | 6998 | 345772 | 348 |
| SourceTree | 2999 | 10751 | 0 |
| enchant.js | 1539 | 79903 | 0 |
| PostgreSQL | 3 | 313 | 3 |
| QuickSilver | 419 | 1526 | 418 |
| SublimeText2 | 0 | 3367 | 0 |

図 7.5 Recheck処理により除外された件数（英語）

次に、Recheck前にヒットしていた件数と検索時間の相関図を 図. 7.6 に示します。図. 7.4 とは異なり、pg_bigmにも 検索時間と件数の相関が見えるようになりました。これらから、pg_bigmではRecheck処理により除外される件数が大きい分、検索時間が他インデックスよりも長くなるケースがあることが確認出来ました。

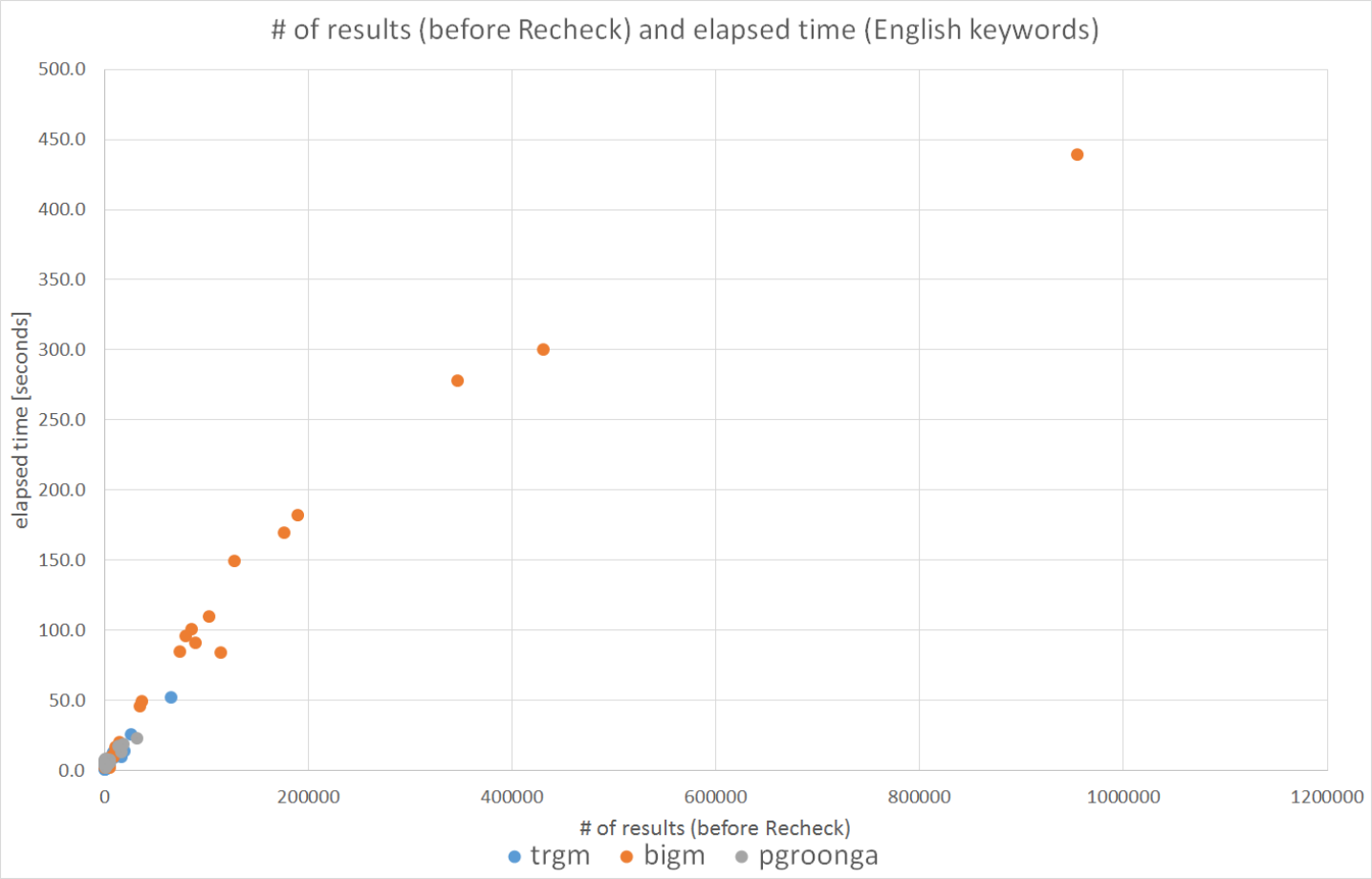



図 7.6 英語キーワードによる検索時間と件数の相関(Recheck前)

### 7.4.3. 日本語キーワードによる比較

図. 7.7 に、日本語キーワードによる検索時間を示します。キーワードが2文字の場合にpg_bigmが特に速くなっています。これは、pg_bigmが文章を2文字で区切ってインデックスを作成しているため、インデックスの効果が十分に発揮されているためと考えられます。

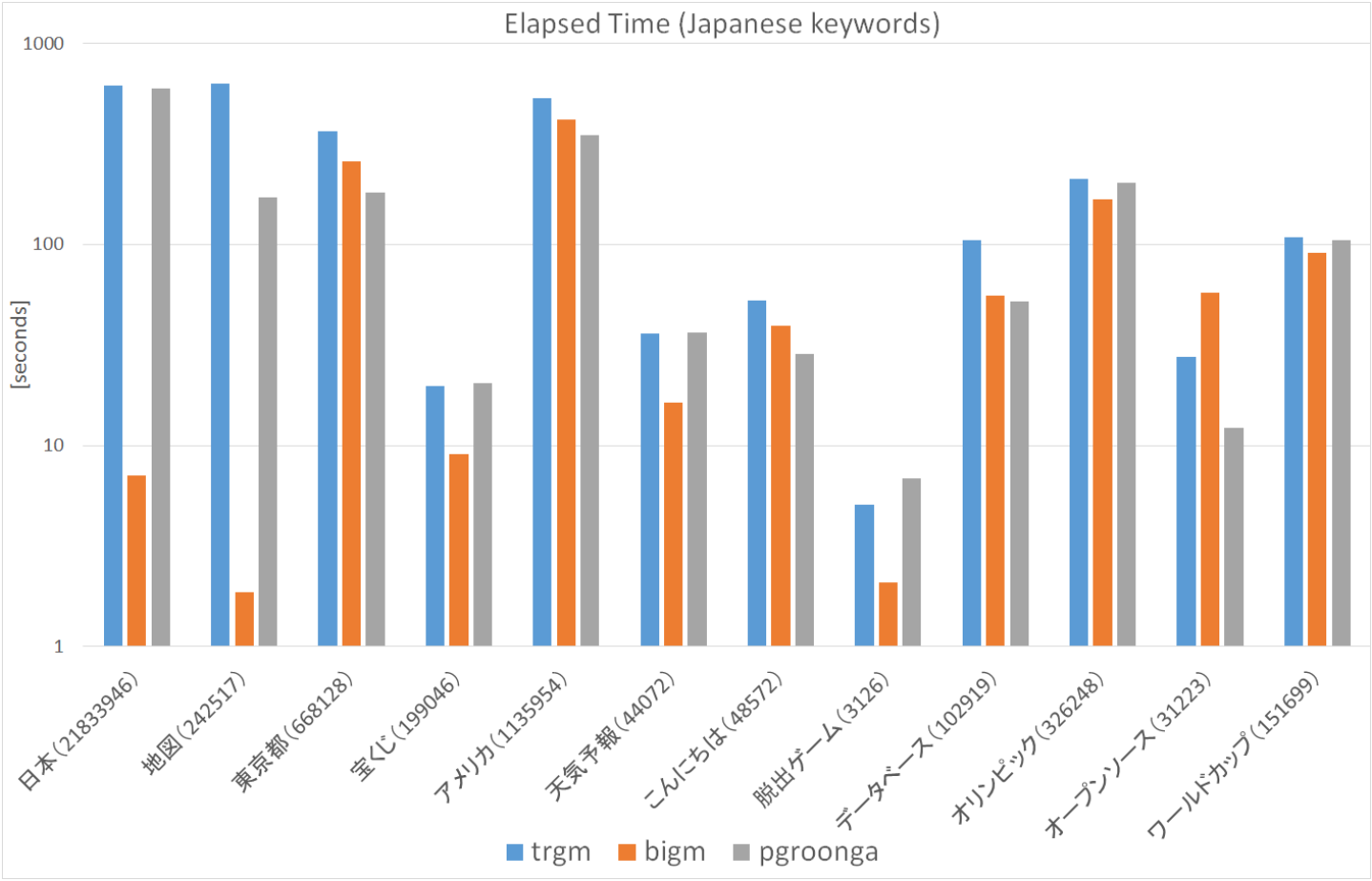



図 7.7 日本語キーワードによる検索時間

#### 7.4.3.1. 検索時間と件数の相関

また、検索時間と検索件数の相関図を 図. 7.8 に示します。おおよそ検索時間が件数に比例していますが、キーワードが2文字の場合に 外れ値が見られます。pg_bigmについては前述のとおり、インデックスが有効に働いているため 検索時間が短くなったものと考えられます。また、pg_trgmについては、インデックスが作成されているにもかかわらずシーケンシャルスキャンが採用されており、このことが所要時間の増加につながっていると考えられます。

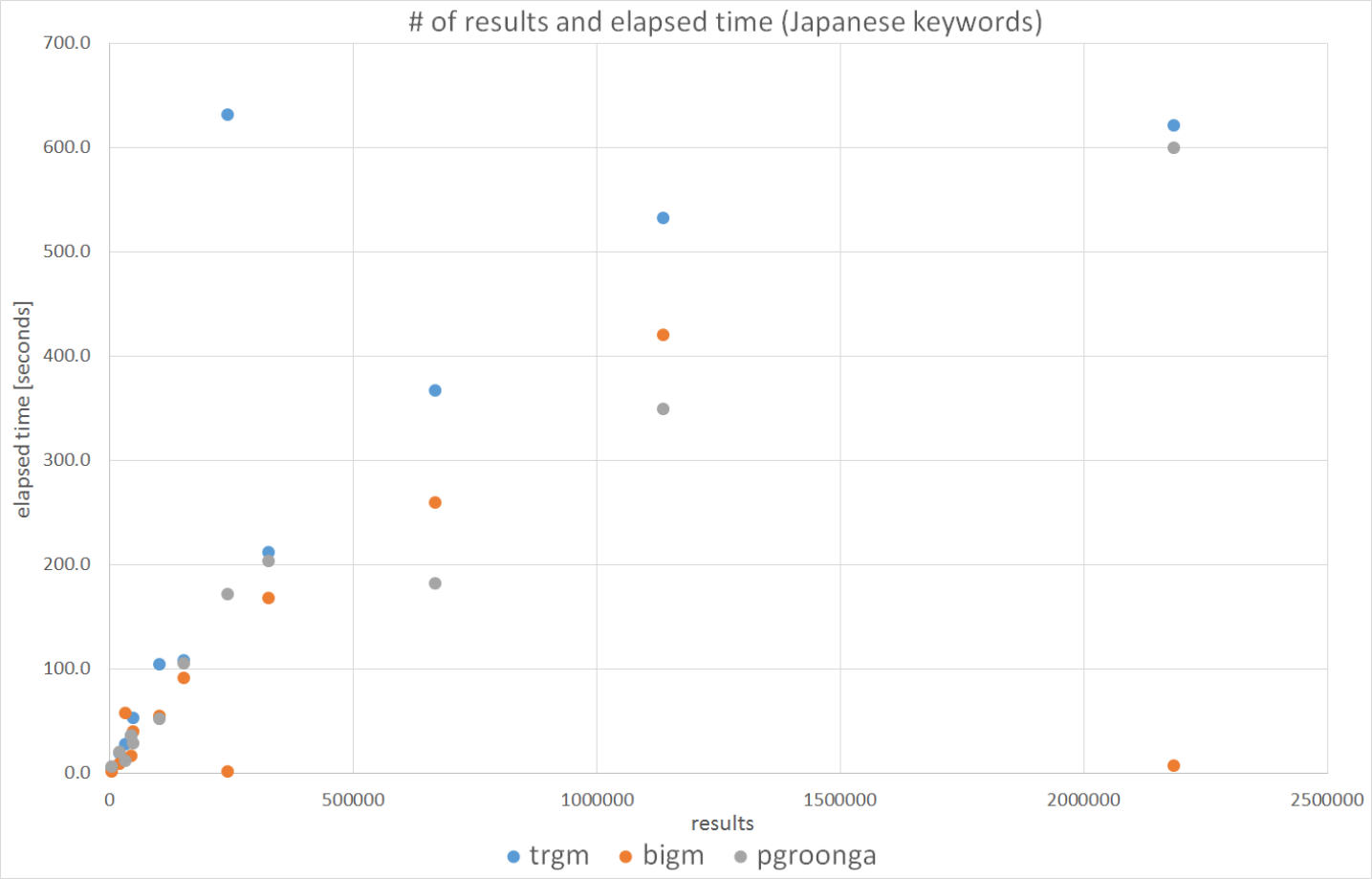



図 7.8 英語キーワードによる検索時間と件数の相関

#### 7.4.3.2. Recheck処理による影響

Recheck処理により除外された件数を 図. 7.9 に示します。pg_trgm、PGroongaにおいては、今回検証した12種類のキーワードのうち10種類で Recheck処理による除外がなされていませんでした。全てのインデックスにおいて、英語よりも除外された件数が少なくなっているため、検索件数と検索時間の関係が表れやすかったものと考えます。

| keyword | pg_trgm | pg_bigm | PGroonga |
|---|---|---|---|
| 日本 | 0 | 0 | 0 |
| 地図 | 0 | 0 | 0 |
| 東京都 | 0 | 195433 | 0 |
| 宝くじ | 0 | 0 | 0 |
| アメリカ | 0 | 6242 | 3 |
| 天気予報 | 0 | 113 | 0 |
| こんにちは | 0 | 11548 | 0 |
| 脱出ゲーム | 0 | 0 | 0 |
| データベース | 0 | 6948 | 614 |
| オリンピック | 347 | 87132 | 0 |
| オープンソース | 0 | 62169 | 0 |
| ワールドカップ | 801 | 47988 | 0 |

図 7.9 Recheck処理により除外された件数（日本語）

## 7.5. 考察

本検証では、PostgreSQLの全文検索モジュール3種を対象に、キーワード検索による 所要時間の比較を行いました。また、検索結果の件数やRecheck処理との関係を調べることで、各インデックスの特性を見ることが出来ました。

検索時間は、インデックスの読み取り時間とテーブルデータの読み取り時間の和ということが出来ます。検索によるヒット件数が多いものでは、テーブルデータの読み取り量が多くなり、インデックスの検索効率が大きく効いてくるものと考えられます。一方で、検索によるヒット件数が少ないものでは、テーブルデータの読み取りにほとんど時間がか

からず、インデックスの読み取り時間が検索時間の多くを占めることとなります。

今回の検証環境では、PGroongaにおいてインデックス自体の容量が物理メモリの容量を超えており、このことによりI/Oが多く発生してしまい、結果として検索に時間がかかった可能性があります。一方で、本検証で比較したもののうち最もインデックス容量が小さくなったpg_bigmにおいては、インデックスの読み取り自体の時間は短くなりますが、Recheck処理の対象となるものが多くなるため、検索時間が伸びてしまうケースが確認できました。

以上から、全文検索用モジュールを選定して利用する際は、各モジュールの特性に加え、データベースを配置するサーバのリソース、例えばメモリやディスクの容量、種類（HDD／SSD、SAS接続／SATA接続 など）も考慮することが必要になってくると言えます。

# 8. VACUUM改善

## 8.1. 検証目的

### 8.1.1. VACUUM FREEZEについて

PostgreSQLでは、トランザクションID(以下、XID)を比較することで、MVCCを実現しています。例えば、現在のトランザクションのXIDより新しいXIDを持った行は、トランザクション開始時点よりも未来のトランザクションで挿入されたものであり、現在のトランザクションからは見ることができないようになっています。

ところで、XIDのサイズは32bit intで表現されているので、20億トランザクションを超えるような 長時間稼働しているクラスタではトランザクションの周回が起こってしまいます。このとき、突然、過去にコミットされたはずのトランザクションが未来のものとして見えるようになり、結果として、データが損失するという事態が起こり得ます。これを防ぐ手段として、XID凍結のVACUUM処理(以下、VACUUM FREEZE)があります。

VACUUM FREEZEでは、行にXID凍結状態という印をつけ、挿入トランザクションが確実に可視になるように、十分古いコミットされたトランザクションに特別なXID(FrozenTransactionId、デフォルト値は2)を指定します。このXIDは通常のXIDの比較規則に従わず、常に通常のXIDよりも古いとみなすことで、XIDが周回してもデータの損失が起こらなくなります。VACUUM FREEZEは、XIDがしきい値となるautovacuum_freeze_max_ageを超えると自動で実行されます。また、自動でのVACUUM FREEZEでは計画的なタイミングで行えないため、手動で実施するという運用も広く行われています。

### 8.1.2. PostgreSQL 9.6における改善

PostgreSQL 9.6ではVACUUM FREEZEの処理が効率的に実行されるように改善されました。

PostgreSQL 9.5以前では、VACUUM FREEZEは対象テーブルのすべてのページをスキャンしており、大きなテーブルに対するVACUUM FREEZEは非常に処理時間がかかるものでした。また、自動VACUUMにおけるVACUUM FREEZEは、XIDがしきい値となるautovacuum_freeze_max_ageを超えると実行されるため、計画的にVACUUM FREEZEを実行しないシステムでは、不意なタイミングで重い処理が走るという挙動が起こっていました。

PostgreSQL 9.6ではVisibility Mapに対して、ページ内容が全て可視かどうかを表すフラグに加えて、ページ内容がすべてXID凍結済みかという情報をもつよう改良が入りました。XID凍結済みのフラグがついているページに対してはスキャンを行わなくなるようになりました。

これによって、データのほとんどが古いデータのまま更新されないような、すでにXID凍結済みのページを多く含むような大きなテーブルに対して、VACUUM FREEZEの処理が短時間で終わるようになりました。

### 8.1.3. 検証目的

PostgreSQL 9.6でVACUUM FREEZEの際のXID凍結済みのページをスキップするようになりましたが、その基礎性能特性を得るのが今回の検証の目的です。PostgreSQL 9.5と9.6でXID凍結済みのページ数の割合を何パターンか用意し、それぞれに対してVACUUM FREEZEを実行したときの処理時間を測定して、性能改善の程度を調べます。また、得られた性能特性に対する考察と、どのような設計・運用で恩恵が得られるかを提示することを目的とします。

## 8.2. 検証構成

### 8.2.1. ハードウェア構成

本検証に用いた構成を 表. 8.1 に示します。今回の検証では、vacuumdbコマンドをPostgreSQLサーバ上で実行しています。

表 8.1 検証構成

| 機器 | 項目 | 仕様 |
|---|---|---|
| PostgreSQL用サーバ | CPU | インテル Xeonプロセッサ E5-2690v4@2.60GHz<br>(14コア)x2 合計 28コア |
| | 搭載メモリ | 256GB |
| | 内蔵ストレージ | HDD 1.2TB SAS 10K x 16 |
| | DB格納用ストレージ | Fiber Channel接続(16Gbps) SAN<br>以下の4領域を利用<br>領域1: PostgreSQLのDB領域<br>領域2: WAL領域<br>領域3: バックアップ用領域<br>領域4: ログ用領域<br>それぞれの領域はHDD 1.8TB 6G SAS 10K x 4<br>(RAID10、実効容量約3TB)で構成 |
| | OS | Red Hat Enterprise Linux 7.2 |
| | DBMS | PostgreSQL 9.6.1<br>PostgreSQL 9.5.5 |

### 8.2.2. postgresql.confの設定値

postgresql.confのパラメータを以下のようにします。VACUUMの性能を見るため、autovacuumはオフにしておきます。加えて、XID周回問題回避のautovacuumが起こりにくくするように、autovacuum_freeze_max_ageを大きめの値にしておきます。

リスト 8.1 postgresql.confのデフォルトからの変更内容

```
autovacuum = off
max_connections = 100
shared_buffers = 192GB
wal_level = replica # PostgreSQL 9.6
wal_level = archive # PostgreSQL 9.5
max_wal_size = 1GB
listen_addresses = '*'
logging_collector = on
log_line_prefix = '%t %p %a '
log_checkpoints = on
log_lock_waits = on
log_autovacuum_min_duration = 0
autovacuum_freeze_max_age = 1000000000
```

## 8.3. 検証方法

### 8.3.1. 環境構築

#### 8.3.1.1. PostgreSQLのインストール

PostgreSQL 9.5.5と9.6.1のソースコードを入手し、それぞれ/usr/local/pgsql95、/usr/local/pgsql96にインストールします。異なるバージョンのPostgreSQLで検証するため、以下のようなスクリプトを使って切り替えます。

リスト 8.2 pgsql95-env.sh

```
#!/bin/bash
export PATH="$PATH":/usr/local/pgsql95/bin
export POSTGRES_HOME=/usr/local/pgsql95
export PGLIB=$POSTGRES_HOME/lib
export PGDATA=/data1/pgsql95/data
export MANPATH="$MANPATH":$POSTGRES_HOME/man
export LD_LIBRARY_PATH="$LD_LIBRARY_PATH":"$PGLIB"
```

リスト 8.3 pgsql96-env.sh

```
#!/bin/bash
export PATH="$PATH":/usr/local/pgsql96/bin
export POSTGRES_HOME=/usr/local/pgsql96
export PGLIB=$POSTGRES_HOME/lib
export PGDATA=/data1/pgsql96/data
export MANPATH="$MANPATH":$POSTGRES_HOME/man
export LD_LIBRARY_PATH="$LD_LIBRARY_PATH":"$PGLIB"
```

#### 8.3.1.2. DBの作成

以下のようなSQLのテンプレートを用いて検証に用いるテーブルを作成します。

リスト 8.4 gen_table.sql.template

```
\set whole 10000000*"SCALE"
\set skip :whole*"PERCENT"/100
CREATE TABLE t1 ( id BIGINT, data TEXT );
INSERT INTO t1
    SELECT id, repeat( md5( id::TEXT ), 2 )
    FROM generate_series( 1, :skip ) id;
VACUUM FREEZE;
INSERT INTO t1
    SELECT id, repeat( md5( id::TEXT ), 2 )
    FROM generate_series( :skip+1, :whole ) id;
CHECKPOINT;
```

ここで、_SCALE_は128固定、_PERCENT_は0, 20, 40, 50, 60, 80, 100と変化させます。これは、テーブルサイズがおおよそ128GBで、XID凍結済みのページ数の割合を おおよそ_PERCENT_で指定する割合に合うようにします。

実際にDBを作成するスクリプトを以下に示します。ここで、/data1はデータ領域1、/data2はデータ領域2、/data3はデータ領域3をマウントしたディレクトリです。/data1、/data2、/data3、/data4の配下にはpgsql95、pgsql96というディレクトリが作成されているとします。また、事前に設定したpostgresql.conf は/data1/pgsql95と/data1/pgsql96に格納されているとします。

リスト 8.5 gen_db.sh

```
#!/bin/bash

SCALE_LIST="128"
PERCENT_LIST="0 20 40 50 60 80 100"

if [ "$1" == "" ]; then
    echo "./gen_db.sh { 95 | 96 }"
    exit
fi

BACKUP_DIR=/data3/pgsql${1}
CONF_DIR=/data1/pgsql${1}

for SCALE in $SCALE_LIST; do
        for PERCENT in $PERCENT_LIST; do
                DB_NAME=db${SCALE}_${PERCENT}
                DB_DIR=${BACKUP_DIR}/${DB_NAME}
                echo "generate $DB_NAME"
                initdb --locale=C -D $DB_DIR
                cp ${CONF_DIR}/postgresql.conf ${DB_DIR}/
                cp gen_table.sql.template gen_table.sql
                sed -e "s/\"SCALE\"/${SCALE}/" -i gen_table.sql
                sed -e "s/\"PERCENT\"/${PERCENT}/" -i gen_table.sql
                pg_ctl start -D $DB_DIR

                sleep 10

                psql -f "gen_table.sql"
                pg_ctl stop -D $DB_DIR

                sleep 10
        done
done
```

#### 8.3.1.3. 性能測定

測定の際は以下のスクリプトを利用します。

リスト 8.6 perf.sh

```
#!/bin/bash

SCALE_LIST="128"
PERCENT_LIST="0 20 40 50 60 80 100"

if [ "$1" == "" ]; then
    echo "./perf.sh { 95 | 96 }"
    exit
fi

BACKUP_DIR=/data3/pgsql${1}
PGDATA_DIR=/data1/pgsql${1}/data
WAL_DIR=/data2/pgsql${1}

for SCALE in $SCALE_LIST; do
        for PERCENT in $PERCENT_LIST; do

                DB_NAME=db${SCALE}_${PERCENT}
                BACKUP_DB_DIR=${BACKUP_DIR}/${DB_NAME}

                for TRIAL in `seq 1 3`; do

                        echo "BEGIN PERF ${DB_NAME} ${TRIAL}/3"

                        # DB構築
                        mkdir $PGDATA_DIR
                        chmod 700 $PGDATA_DIR/
                        cp -R $BACKUP_DB_DIR/* $PGDATA_DIR/
                        mv $PGDATA_DIR/pg_xlog $WAL_DIR/
                        ln -s $WAL_DIR/pg_xlog $PGDATA_DIR/pg_xlog

                        pg_ctl start -D $PGDATA_DIR
                        sleep 10

                        # 測定
                        LOG_DIR=/data4/pgsql${1}/$DB_NAME/${TRIAL}
                        mkdir -p $LOG_DIR
                        LANG=C sar -A -o ${LOG_DIR}/sar.out 1 > /dev/null &
                        LANG=C date > ${LOG_DIR}/begin
                        vacuumdb --verbose --freeze > ${LOG_DIR}/stdout 2>&1
                        LANG=C date > ${LOG_DIR}/end

                        KILLPROC=`ps aux | grep "sar" | cut -d' ' -f 2`
                        for PROC in $KILLPROC; do
                            kill -15 $PROC
                        done

                        # DB停止
                        pg_ctl stop -D $PGDATA_DIR
                        sleep 10

                        # データ退避
                        cp -R ${PGDATA_DIR}/pg_log ${LOG_DIR}

                        # DB破棄
                        rm -rf $WAL_DIR/pg_xlog
                        rm -rf $PGDATA_DIR
                done
        done
done
```

このスクリプトでは、gen_db.shで/data3に作成したDBを/data1に戻し、WAL領域を/data2に設定しています。そして、VACUUM FREEZEを実行し、sarやログを/data4に書きだしています。これを、/data3に作成したDBに対して、3回実施しています。

## 8.4. 検証結果

図. 8.1 にXID凍結済みのページ数の割合に対する、スキャンしたページ数の結果を示します。これは仕様通り、9.5ではすべてのページをスキャンしていますが、PosgtreSQL 9.6ではXID未凍結の部分だけスキャンしていることが分かります。

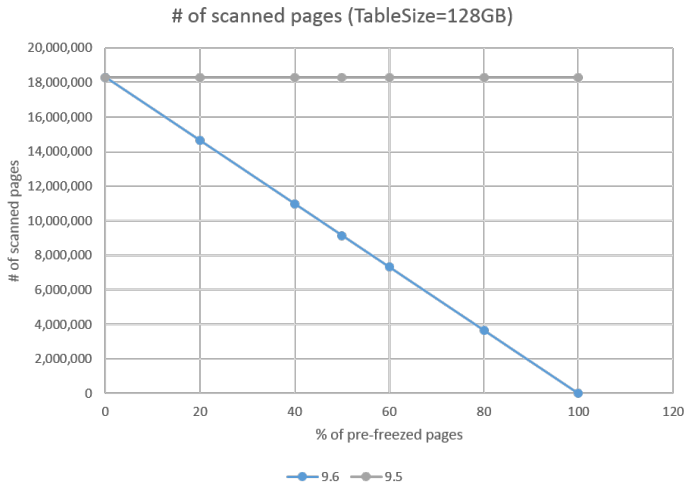



図 8.1 スキャンしたページ数

図. 8.2 にXID凍結済みのページ数の割合に対する実行時間をプロットしたグラフを示します。未凍結の場合は9.5、9.6で13秒程度の差ですが、次第に処理時間の差が大きくなっていき、100%凍結の場合、9.5で100秒かかる処理が9.6では1秒以下となり、その効果が大きいことが分かります。

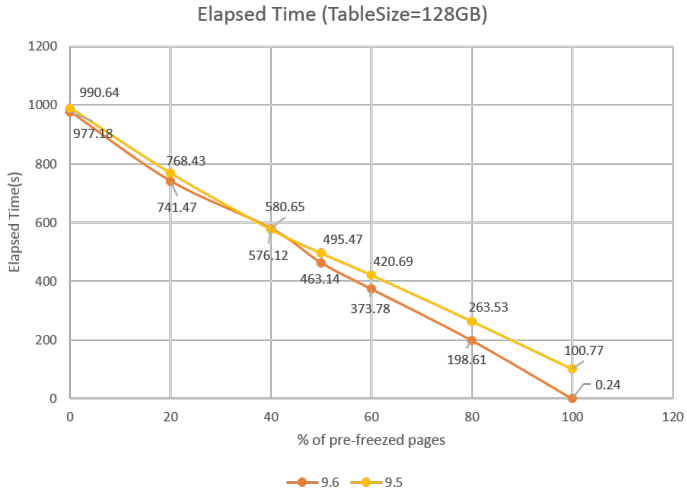



図 8.2 pg_prewarmなしの場合の実行時間

図. 8.3 にpg_prewarmを実行し、shared_buffersにオンバッファになった状態での性能特性を示します。XID未凍結の場合でも30秒ほど9.6が速く、スキャン性能自体も改善されていることが分かります。そして、全体的に50から60秒ほど9.6が高速であることが分かります。

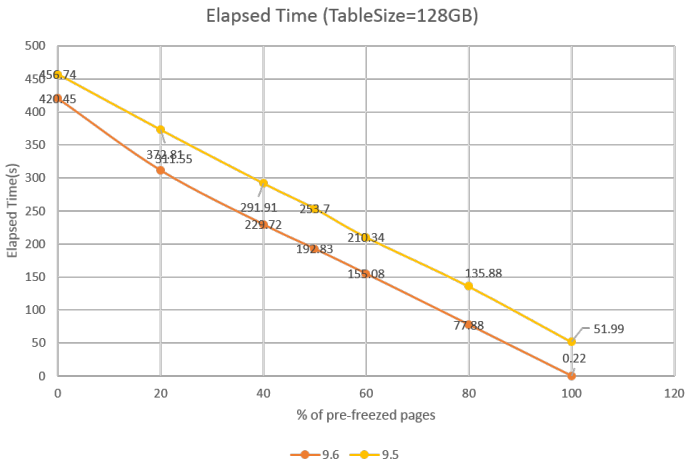



図 8.3 pg_prewarmありの場合の実行時間

## 8.5. 考察

検証結果より、確かにVACUUM FREEZE時にPostgreSQL 9.6ではXID凍結済みページをスキップしていることが確認できました。また、ページスキップ以外の要因と思われるものによって、PostgreSQL 9.6が全体的に性能向上していることも確認できました。これについて、ストレージへの負荷の状況からPostgreSQL9.5と9.6とでの性能特性の違いを考察してみます。図. 8.4 、図. 8.5 にpg_prewarm実行後のXID凍結済みのページ数が0%のテーブルに対して VACUUM FREEZEを実行したときの、sarの%utilを示します。

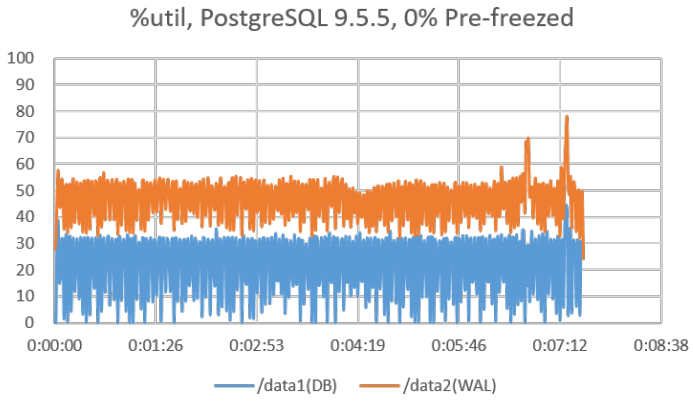



図 8.4 PostgreSQL 9.5でVACUUM FREEZEを行った場合のI/O負荷

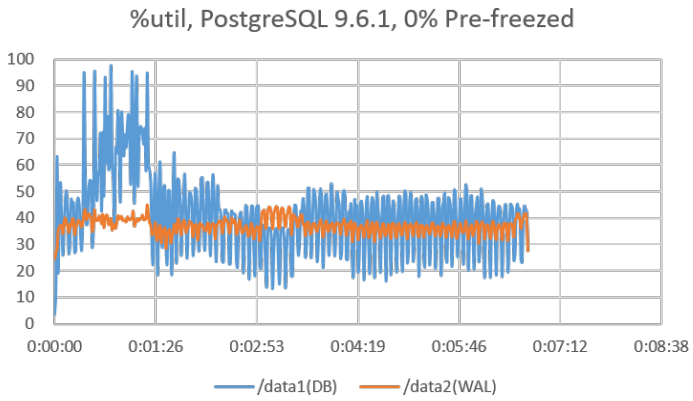



図 8.5 PostgreSQL 9.6でVACUUM FREEZEを行った場合のI/O負荷

DB領域のutilを比較すると、9.6と9.5どちらもページスキップをせずに全ページに対してXID凍結処理を行うにもかかわらず、負荷の状況が大きく異なっております。また、WAL領域のutilを比較すると、9.6は9.5と比較して負荷が小さくなっていることが分かります。PostgreSQL 9.6でWALに関わる改善として、チェックポイントの改善などがあり、これらの影響によって、ページスキップなしのVACUUM FREEZEでもPostgreSQL 9.6の方が良好な性能が得られるのではないかと推測されます。

以上より、PostgreSQL 9.6はPostgreSQL 9.5と比較して、全体的にVACUUM FREEZEの性能が改善されており、また、XID凍結済みページが多い場合に非常に高速になることがわかりました。したがって、PostgreSQL 9.6では、9.5以前で問題となっていた、不意なタイミングで発生する動作の重くなるVACUUM FREEZEの動作時間、計画的にVACUUM FREEZEを行うシステムにおけるメンテナンス時間を短くすることができ、非常に有益であることが分かりました。

# 9. おわりに

2016年度は、PGEConsのワーキンググループの位置づけを見直して、ワーキンググループ 1 (WG1) は従来の「性能ワーキンググループ」から、性能に加えて新技術の検証も行う「新技術検証ワーキンググループ」として活動しました。

この章では各検証の結果を振り返って、PostgreSQL の機能・性能の現状をまとめます。

## 9.1. 検証結果から見る PostgreSQL の今

ワーキンググループの新しい位置づけを踏まえて、今年度は以下の検証テーマを実施しました。

- PostgreSQL 9.6のスケール性(参照系および更新系)
- PostgreSQL 9.6の新技術・機能向上
  - パラレルクエリ
  - VACUUM改善
  - JSON/JSONB
  - 全文検索
- Pgpool-IIのスケール性

これらの検証の結果を、「メニーコアCPUへの対応」「大量データの取り扱い」「多様化するデータ型への対応」という観点から、PostgreSQL の機能や性能の現状をまとめると、次のように言えるでしょう。

メニーコア CPU への対応

> PostgreSQL のスケール性の検証では、PostgreSQL 9.6 は前バージョン 9.5 に対して参照系で約20% 、更新系で約30% 最大スループットが向上したことを確かめました。また、バージョン 9.6 はコア数が 24 を超える測定ケースで バージョン 9.5 よりもスループットが向上しており、メニーコア CPU でより性能を発揮することを確かめました。
>
> また、1つのSQL文(クエリ)を複数のプロセスで実行するパラレルクエリ機能の検証では、CPUコアが十分多数ある場合に、1プロセスでのクエリ処理に比べて数十分の一で応答時間となるケースもあり、大幅な高速化が確かめられました。
>
> こうした結果は、近年のCPUの高性能化がコア数を増やす方向に力点を置いていることとマッチしており、PostgreSQL は時代に即した高性能化を遂げていることを示しています。

大量データの取り扱い

> 大量データの取り扱いのための PostgreSQL の対応状況という観点では、上に挙げたパラレルクエリ機能が大量データの検索に有効であることを示しています。
>
> 従来、テーブルサイズが大きい場合には、テーブル内の古いデータを「凍結」するための VACUUM FREEZE にも時間を要していました。PostgreSQL 9.6 では凍結済みデータの処理をスキップすることで処理時間を短縮しています。今回の検証では、定期的に VACUUM FREEZE を実行して未凍結データを少なくすれば、表のサイズとは関係なく十分短い時間で VACUUM FREEZE が完了することを示しました。
>
> データの大量化に伴ってそれに対するアクセスが増えるケースでは、複数の PostgreSQL サーバによるデータベースクラスタによって負荷を分散することで、対応することが考えられます。今回の検証では、Pgpool-II によるクラスタにおいて PostgreSQL サーバの台数によってどのような性能傾向となるかを調査しました。その結果、更新負荷に対してはサーバの台数増加によるスループット向上はないものの、参照負荷に対しては台数に比例してスループットが向上することを示しました。
>
> こうした結果は、PostgreSQLではコア機能・周辺ツールを含めて、大量データを扱うための機能が向上していることを示しています。次バージョン PostgreSQL 10 ではパーティショニング機能の改善も予定されており、更なる向上が期待できます。

多様化するデータ型への対応

> JSON 形式のデータを扱う JSON/JSONB データ型での性能検証では、IoTに代表される半構造型データを PostgreSQL で適切に扱う手法を検討しました。JSON/JSONB データ型を使うと、半構造データをそのまま格納できる反面、通常のデータ型による格納・検索とはデータサイズ・性能にトレードオフがあることを示しました。
>
> 全文検索機能の検証では、日本語文字列を扱える全文検索インデックス3種を調査しました。検索時間の比較からは、それらのインデックスの種類ごとに検索するデータの性質によって得意・不得意があることを示しました。
>
> こうした結果は、PostgreSQLにおいても、半構造データの格納・検索や日本語の全文検索のための機能が整いつつあることを示す反面、その利用においてはユースケースに沿った検証が不可欠であることを示唆しています。

## 9.2. 今後の活動に向けて

先にも書いたように、今年度はワーキンググループの取り組み対象として、従来の性能に「新技術検証」を加えて検証を実施しました。取り上げたテーマには、以前から PGECons が検証してきた新バージョンの性能検証・VACUUM の改善・Pgpool-II に加えて、パラレルクエリ・全文検索・ JSON/JSONB を新たに取り上げました。

これらの検証結果をご利用いただく際には、ご自身のシステムの特性を加味した上で検討していただくことをお勧めします。とりわけ、パラレルクエリと全文検索については、クエリ・データによって性能が大きく変わることが予想されます。

今回の検証を通じて、PostgreSQL の性能・機能の向上について確認することが出来ました。次の PostgreSQL はバージョン 10.0 となり、パーティショニングやパラレルクエリの改善、ロジカルレプリケーションの実現など、さまざまな機能・性能の向上が図られる予定です。PGEConsはこれらの改善点についての調査・検証を通じて、引き続き PostgreSQL のエンタープライズ領域への普及を推進してまいります。

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

(終わり)

# 10. 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<br>(2016年度WG1) | SRA OSS, Inc. 日本支社 | | 石井 達夫（取締役支社長） |
| | SRA OSS, Inc. 日本支社 | マーケティング部 | 近藤 雄太 |
| | NTTテクノクロス株式会社 | クラウド＆セキュリティ事業部 | 原田 登志 |
| | NTTテクノクロス株式会社 | クラウド＆セキュリティ事業部 | 勝俣 智成 |
| | 日本電気株式会社 | システムプラットフォームビジネスユニット<br>クラウドプラットフォーム事業部 | 川畠 輝聖 |
| | 日本電気株式会社 | システムプラットフォームビジネスユニット<br>クラウドプラットフォーム事業部 | 慶松 明嗣 |
| | 日本電信電話株式会社 | オープンソースソフトウェアセンタ | 坂田 哲夫 |
| | 日本ヒューレット・パッカード株式会社 | テクノロジーコンサルティング事業統括 | 北山 貴広 |
| | 日本ヒューレット・パッカード株式会社 | テクノロジーコンサルティング事業統括 | 高橋 智雄 |
| | 富士通株式会社 | 共通ソフトウェア開発技術本部<br>ソフトウェア開発技術統括部<br>OSS技術センター | 野山 孝太郎 |
| | 富士通株式会社 | 共通ソフトウェア開発技術本部<br>ソフトウェア開発技術統括部<br>OSS技術センター | 高澤 亮平 |

# 11. ライセンス

# 12. Appendix 検証環境

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