PGECons
PostgreSQL Enterprise Consortium

PostgreSQL エンタープライズ・コンソーシアム 技術部会 WG#1
性能ワーキンググループ

2015 年度 WG1 活動報告

大規模 DB への適用性が向上した PostgreSQL9.5 の性能検証

製作者
担当企業名:アイウエオ順
SRA OSS, Inc.日本支社
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

© 2016 PostgreSQL Enterprise Consortium

改訂履歴

版	改訂日	変更内容
1.0	2016/04/18	初版

ライセンス

本作品は CC-BY ライセンスによって許諾されています。
ライセンスの内容を知りたい方は http://creativecommons.org/licenses/by/2.1/jp/ でご確認ください。
文書の内容、表記に関する誤り、ご要望、感想等につきましては、PGECons のサイトを通じてお寄せいただきますようお願いいたします。
サイト URL https://www.pgecons.org/contact/

Intel 、インテルおよび Xeon は、米国およびその他の国における Intel Corporation の商標です。
Linux は、Linus Torvalds 氏の日本およびその他の国における登録商標または商標です。
Red Hat および Shadowman logo は、米国およびその他の国における Red Hat,Inc. の商標または登録商標です。
PostgreSQL は、PostgreSQL Community Association of Canada のカナダにおける登録商標およびその他の国における商標です。
TPC, TPC Benchmark,TPC-C, TPC-E, tpmC, TPC-H, QphH は米国 Transaction Processing Performance Council の商標です
その他、本資料に記載されている社名及び商品名はそれぞれ各社が商標または登録商標として使用している場合があります 。

© 2016 PostgreSQL Enterprise Consortium

本報告書について

■本資料の概要と目的

本資料では、WG1 として PostgreSQL 9.5 のスケール性（参照系および更新系）、PostgreSQL 9.5 の新機能(Parallel VACUUM, BRIN index)による各種検証、および Linux OS のバージョン差異による性能傾向を検証し、その方法と結果を報告します。

■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。

© 2016 PostgreSQL Enterprise Consortium

目次

© 2016 PostgreSQL Enterprise Consortium

1. はじめに

1.1. 2015 年度 WG 1活動テーマ

1.1.1. 活動テーマ決定経緯

WG1 は 2012 年度より、「大規模基幹業務に向けた PostgreSQL の適用領域の明確化」を大きな目的に活動しております（2012/7/6 開催の PGECons セミナーより）。このテーマの実施にあたり、技術部会では課題領域を以下の大区分に分類しました。

表 1.1: PGECons における課題領域

性能	性能評価手法、性能向上手法、チューニングなど
可用性	高可用クラスタ、BCP
保守性	保守サポート、トレーサビリティ
運用性	監視運用、バックアップ運用
セキュリティ	監査
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
接続性	他ソフトウェアとの連携

性能に関しては更に以下の小区分に分解し議論を深め、2012 年度はスケールアップとスケールアウトの性能検証を実施しました。

表 1.2: 性能検証テーマ

性能評価手法	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
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
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
性能向上機能	クエリキャッシュ、パーティショニング、高速ロードなど
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

2012 年度の成果としては、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

© 2016 PostgreSQL Enterprise Consortium

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。

1.1.2. 定点観測（スケールアップ）

PostgreSQLに対する一般的な性能懸念として、CPU マルチコアを活かして性能を出せるか、というものがあります。

つまり、CPU リソースが増えてもそれによって性能が向上しないのではないかという懸念です。これに対して、2012 年度、2013 年度は、物理コア数が 80、メモリが 2TB という非常に大規模なリソースを持ったサーバ環境を用意し、各年度の最新バージョンである PostgreSQL9.2 および 9.3 において、検索性能がどこまでスケール出来るかを評価しました。

2014 年度でも引き続き PostgreSQL9.4 を対象にスケールアップ検証を実施しました。定点観測として PostgreSQL9.3 との性能比較を示すことで、現在利用されている PostgreSQL をバージョンアップする検討材料の一つになるのではないかと考えました。

2015 年度は、2014 年度に引き続いて、PostgreSQL 9.5 を対象にスケールアップ検証を実施し、PostgreSQL 9.4 に比べてスケール性が改善されていることを確認しました。

1.1.3. Parallel VACUUM

PostgreSQL 9.5 の新機能である Parallel VACUUM について検証し、複数のワーカを用いて VACUUM を実行することで、実行時間が短縮できることを確認しました。

1.1.4. BRIN Index

PostgreSQL 9.5 の新機能である BRIN Index について、従来からの機能である B-tree Index、パーティショニングとの比較を含めて検証し、適切な利用方法についての知見を得ました。

1.1.5. OS 比較

Linux OS の代表的ディストリビューション Red Hat Enterprise Linux の新旧バージョン上での PostgreSQL9.5 の性能傾向について、ファイルシステムによる違いを含めて検証しました。

1.2. 実施体制

2015 年 6 月 25 日に開催された 2015 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: 2015 年度 WG1 参加企業一覧

SRA OSS, Inc.日本支社
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、日本電信電話株式会社は、「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。

© 2016 PostgreSQL Enterprise Consortium

1.3. 実施スケジュール

2015 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	2015 年 6 月 25 日
実施計画策定	2015 年 7 月～11 月
検証実施	2015 年 12 月～2016 年 2 月
2015 年度 WG1 活動報告書作成	2016 年 2 月～3 月
総会と成果報告会	2016 年 5 月 13 日

© 2016 PostgreSQL Enterprise Consortium

2. 定点観測(スケールアップ検証・参照系)

2.1. 検証概要

2015 年度は 72 コア(18 コア*4 プロセッサ)の CPU（Xeon E7-8890 v3 2.5GHz）、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 9.5 と前バージョンの 9.4 との参照性能の比較を行いました。

なお、測定時点では PostgreSQL 9.5 の正式版がリリースされていなかったため、計測には β 版である 9.5beta2 を用いました。PostgreSQL 9.4 に関しては、計測時点の最新版である 9.4.5 を用いました。

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。

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。

© 2016 PostgreSQL Enterprise Consortium

- 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

- 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

- 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
-s スケールファクタを数字(1 以上の整数)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。

© 2016 PostgreSQL Enterprise Consortium

¥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

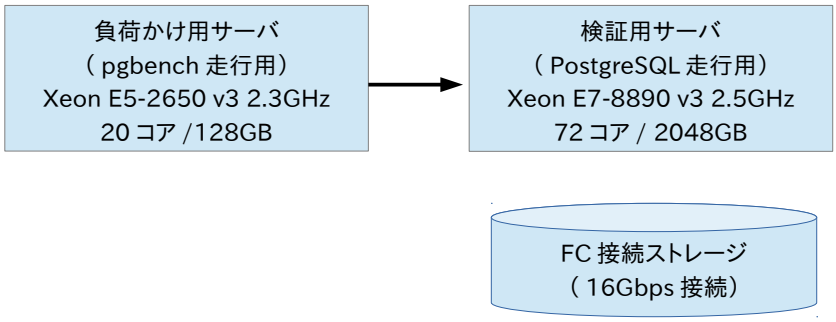

図 2.1: 検証ハードウェア構成

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。

OS	Red Hat Enterprise Linux 7.2
PostgreSQL	9.5beta2 および 9.4.5

表 2.1: 検証用サーバ

OS	Red Hat Enterprise Linux 7.2
pgbench	9.5beta2 のソースコードに含まれるものをビルドして使用

表 2.2: 負荷かけ用サーバ

© 2016 PostgreSQL Enterprise Consortium

2.4. 検証方法

2.4.1.1. 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、postgresql.conf を編集します。

```
$ initdb -D {directory} --no-locale -E UTF8
```

```
$ vi {directory}/postgresql.conf
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
wal_level = archive
checkpoint_timeout = 30min
logging_collector = on
logline_prefix = '%t [%p-%l] '
```

他に、チェックポイントを起動させる WAL 量の指定は 9.5 から checkpoint_segments（WAL ファイル数）パラメータが廃止されて max_wal_size（WAL サイズ）パラメータで指定するようになったため、9.4 では

```
checkpoint_segments = 64
```

9.5 では

```
max_wal_size = 1GB # (1GB = 16MB * 64 (WAL1 ファイルは 16MB))
```

を指定しています。

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 -D [directory] -w start
$ createdb -p [port] [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 -h [host] -p [port] -s 1000 [dbname]
```


© 2016 PostgreSQL Enterprise Consortium

2.4.1.2. 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, 100, 128} です。

© 2016 PostgreSQL Enterprise Consortium

2.5. 検証結果

結果をグラフに示します。64 クライアントを超えた辺りから 9.5 が 9.4 に TPS が上回りました。クライアント数が 100 を超えたあたりから、全 72 コアの CPU を使い切った状態になり、TPS が頭打ちになりました。

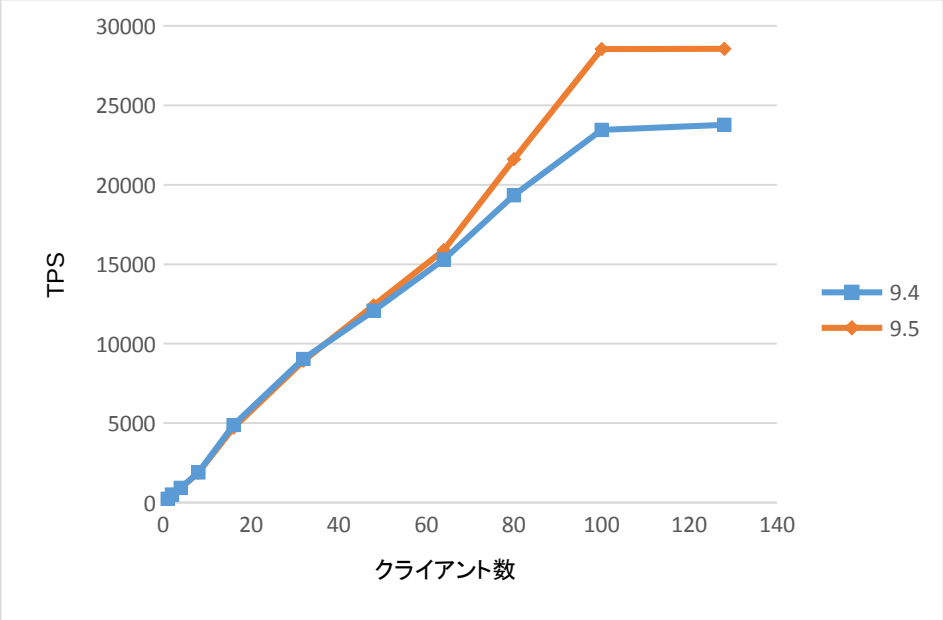

図 2.2: 各クライアント数に対する TPS

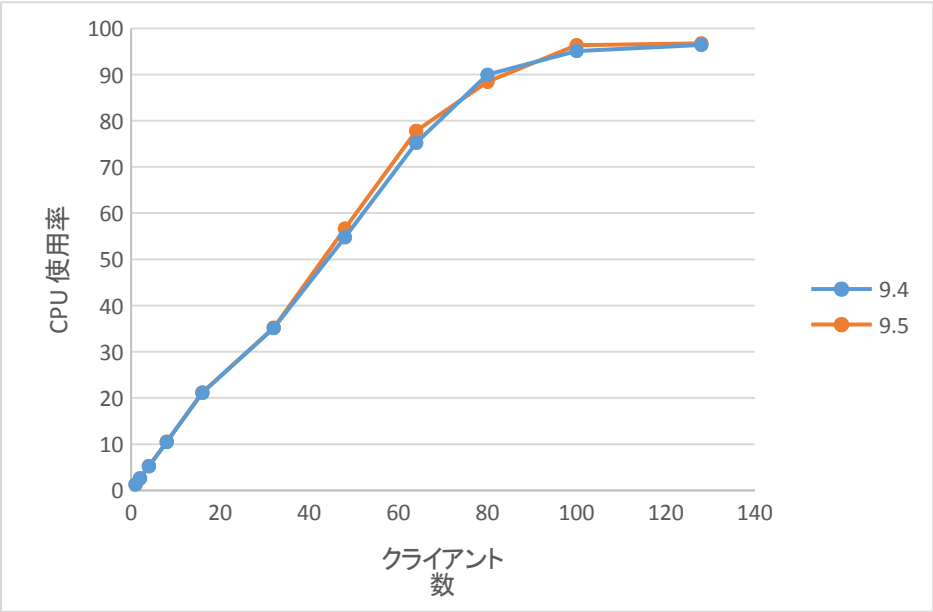

図 2.3: 各クライアント数に対する CPU 使用率（全 72 コアの平均値）

© 2016 PostgreSQL Enterprise Consortium

2.6. perf を用いて追加検証

perf は Linux で利用可能な性能解析ツールであり、実行プログラムにおけるパフォーマンスカウンタの値を取得できます。
以下では perf を用いてバージョンごとの PostgreSQL のパフォーマンスを検証します。

2.6.1. perf stat

perf stat はプロセスを監視し、各ハードウェアイベントの回数を取得できます。
性能差が大きく見られたクライアント数 100 を用いて、クライアント側から pgbench を実行します。

```
# pgbench -n -h ${host} -p ${port} -c 100 -j 50 -f ${custom_sql} -T 300 -s 1000 pgbench > {pgbench_record} 2>&1
```

サーバ側から perf stat を実行します。
ここでは 240 秒間 1 つの postgres プロセスを監視します。

```
# perf stat -p ${postgres_pid} sleep 240 2> ${perf_stat_file}
```

取得した結果を表示すると以下のように branches (分岐命令の実行回数) イベントにて差が見られました。

```
cat ${perf_stat_file}
(9.4)  143,939,452,489     branches              #  850.754 M/sec
(9.5)  173,983,928,300     branches              #  996.798 M/sec                (100.00%)
```


© 2016 PostgreSQL Enterprise Consortium

2.6.2. perf record

perf record は指定したイベント発生時のシンボルを記録することができます。このとき保存されたファイルは perf report で見ることができます。

9.4 と 9.5 で比較的差が見られた branches イベントについて perf record を実行します。

```
# perf record -e branches -p ${postgres_pid} -o ${perf_record_file} sleep 240
# perf report -i ${perf_record_file}
```

実行割合の大きい 10 個の関数を下のグラフに示しています。
9.5 では LWLockAcquire の割合が小さくなり、代わりに LWLockAttemptLock が登場しています。

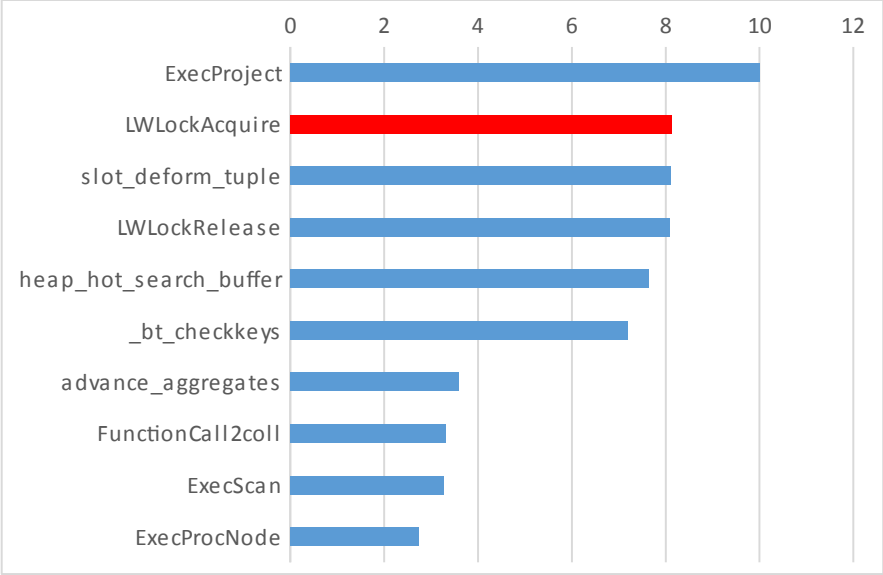

図 2.4: branches イベント時の実行関数割合[%] (9.4)

© 2016 PostgreSQL Enterprise Consortium

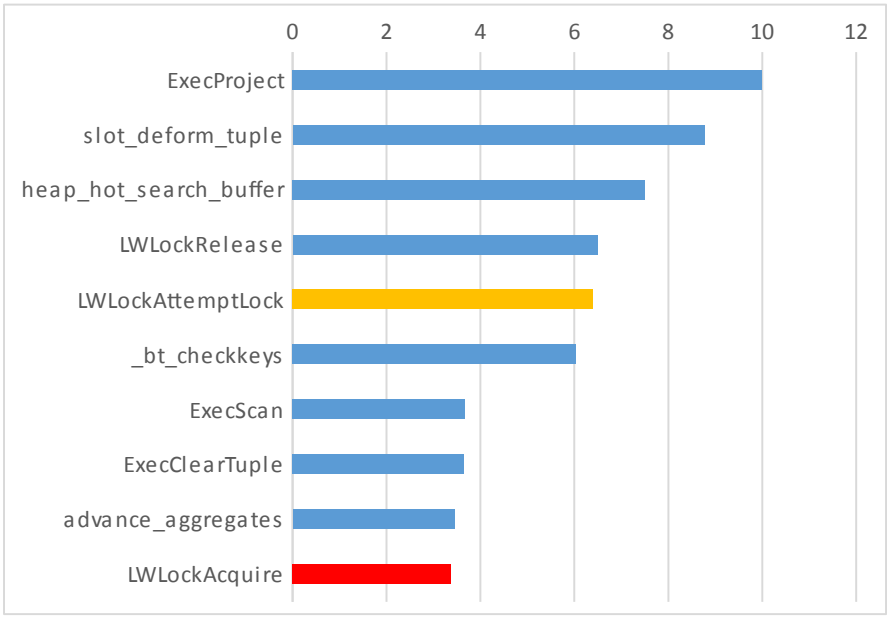

図 2.5: branches イベント時の実行関数割合[%] (9.5)

2.7. 考察

ある程度クライアント数が大きくなると 9.5 は 9.4 に上回る性能を示しました。この要因を追求するためプロファイラツールの perf による追加検証を行いました。perf stat で複数イベントの性能測定を行い、9.4 と 9.5 で大きな差が見られた branches イベントを perf record で確認したところ、実行されている関数の割合に違いが見られました。

実行されている割合が高いもののうち、9.5 では LWLockAquire が減り、9.5 から登場した関数の LWLockAttemptLock が現れていました。実際、9.5 では LWLock 実装の改訂が行われており、LWLockAquire 内のスピンロックがボトルネックとなっていた点が修正されていました。本結果からロックのスケーラビリティ性能が改善されたことが実際に確認できたといえます。

参考:

PostgreSQL 9.5.0 リリースノート（日本語）
http://www.sraoss.co.jp/technology/postgresql/9.5/

PostgreSQL 9.5.0 リリースノート（英語）
http://www.postgresql.org/docs/9.5/static/release-9-5.html

該当の LWLock 改訂コミット（英語）
https://github.com/postgres/postgres/commit/ab5194e6f617a9a9e7aadb3dd1cee948a42d0755

© 2016 PostgreSQL Enterprise Consortium

3. 定点観測(スケールアップ検証・更新系)

3.1. 検証概要

PGECons では、PostgreSQL の新バージョン・リリースに合わせて、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とした定点観測を 2012 年度から行ってきました。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ないました。また、昨年度の更新系スケールアップ検証では、その時点での新バージョンである PostgreSQL 9.4 で WAL 出力のロック競合の軽減による並列書込み性能改善が行われましたので、その効果を評価するため、ロック競合の測定も実施しました。

今年度は、定点観測として継続している 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-9.4.5 と PostgreSQL-9.5beta2)の比較による更新性能の改善状況確認に加えて、PostgreSQL 9.5 の改善点の一つとなっている共有ロック取得処理のスケーラビリティ改善についての評価も行いました。

3.2. 検証構成

検証環境のハードウエアおよびソフトウエアの主なスペックと構成は以下のとおりです。

表 3.1: 検証構成

機器	項目	仕様
クライアント(pgbench)	CPU	インテル Xeon プロセッサ E5-2650v3 @ 2.30GHz(10 コア) x 2 物理コア合計 20core, Hyperthreading ON 論理コア合計 40 コア
	搭載メモリ	128GB
	内蔵ストレージ	HDD: 1.2 TB 10K x 4
	OS	Red Hat Enterprise Linux 7.2 (for Intel 64)
	クライアント	PostgreSQL-9.5beta2 のソースコードに含まれる pgbench をビルドして使用
PostgreSQL 用サーバ	CPU	インテル Xeon プロセッサ E7-8890v3 @2.50GHz (18 コア) x 4 合計 72core, Hyperthreading OFF
	搭載メモリ	2048GB
	内蔵ストレージ	HDD: 1.2TB 10k x 10
	DB 格納用ストレージ	Fiber Channel 接続 (16Gbps) SAN データベースクラスタ($PGDATA)と WAL 領域 ($PGDATA/pg_xlog)に以下の RAID10 構成 (実効容量約 3TB)を利用 (900GB 6G SAS 10K 2.5inch 8 台)
	OS	Red Hat Enterprise Linux 7.2 (for Intel 64)
	DBMS	PostgreSQL 9.4.5 または PostgreSQL 9.5beta2

© 2016 PostgreSQL Enterprise Consortium

3.3. 検証方法

3.3.1. 環境

(a) 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、テーブルのデータ（$PGDATA ディレクトリ以下のファイル群）は、測定を行う度に 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 tar で保存しておき、それを毎回 $PGDATA に戻す方法で用意することにしました。

(b) postgresql.conf 設定値

postgresql.conf により設定するパラメータの内、幾つかをデフォルト値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出来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。また、shared_buffers は検証ハードウエア（DB サーバ）が搭載するメモリ量に合わせて設定しました。

postgresql.conf 設定（default から変更したパラメータのみ）

```
max_connections = 1000
shared_buffers = 384GB      # およそサーバ搭載容量の 20%
wal_level = archive
checkpoint_segments = 10000 # 9.4
max_wal_size = 160GB        # 9.5
checkpoint_timeout = 60min
maintenance_work_mem = 20GB
log_checkpoints = on
log_line_prefix ='%t %p %a'
log_lock_waits = on
autovacuum = off
```

なお、wal ファイル容量の上限を設定するパラメータは、9.4 の checkpoint_segments から、9.5 では max_wal_size に変更されています。両方とも、測定期間中に書き出される wal の量が設定値を超えないような値を設定し、CHECKPOINT 処理が実行されないようにしました。

3.3.2. 測定

(a) 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についても昨年度の検証を踏襲し、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。今年度は、PostgreSQL-9.5 の改善点の一つとなっている「内部のロック（lwlock）を共有モードで取得する処理のスケーラビリティ向上」の効果を検証するため、昨年度とは異なり、2種類の pgbench スクリプト（write1 と write10）を用意しました。両方とも、1 回の pgbench スクリプトで 10 回の UPDATE を PostgreSQL に要求するようになっています。

write1 スクリプトは 10 回の UPDATE をオートコミットモードで行います。このため、1 回の UPDATE が 1 トランザクションとなり、1 回の pgbench スクリプトの実行で 10 回のトランザクションが実行されます。一方、write10 では、10 回の UPDATE をまとめて 1 回のトランザクションとしています。このため、1 回の pgbench スクリプトの実行で 1 回のトランザクション、10 回の UPDATE が実行されます。このように両スクリプトで「トランザクション」と「UPDATE」の対応関係が異なりますので、これ以降、スループットは pgbench スクリプト（UPDATE10 回）を単位とする値に統一します。

© 2016 PostgreSQL Enterprise Consortium

write1 スクリプト

```
¥set naccounts 100000 * :scale
¥setrandom aid00 1 :naccounts
¥setrandom aid01 1 :naccounts
¥setrandom aid02 1 :naccounts
¥setrandom aid03 1 :naccounts
¥setrandom aid04 1 :naccounts
¥setrandom aid05 1 :naccounts
¥setrandom aid06 1 :naccounts
¥setrandom aid07 1 :naccounts
¥setrandom aid08 1 :naccounts
¥setrandom aid09 1 :naccounts
UPDATE pgbench_accounts SET filler=repeat(md5(current_timestamp::text),2) WHERE aid= :aid00;
UPDATE pgbench_accounts SET filler=repeat(md5(current_timestamp::text),2) WHERE aid= :aid01;
UPDATE pgbench_accounts SET filler=repeat(md5(current_timestamp::text),2) WHERE aid= :aid02;
UPDATE pgbench_accounts SET filler=repeat(md5(current_timestamp::text),2) WHERE aid= :aid03;
UPDATE pgbench_accounts SET filler=repeat(md5(current_timestamp::text),2) WHERE aid= :aid04;
UPDATE pgbench_accounts SET filler=repeat(md5(current_timestamp::text),2) WHERE aid= :aid05;
UPDATE pgbench_accounts SET filler=repeat(md5(current_timestamp::text),2) WHERE aid= :aid06;
UPDATE pgbench_accounts SET filler=repeat(md5(current_timestamp::text),2) WHERE aid= :aid07;
UPDATE pgbench_accounts SET filler=repeat(md5(current_timestamp::text),2) WHERE aid= :aid08;
UPDATE pgbench_accounts SET filler=repeat(md5(current_timestamp::text),2) WHERE aid= :aid09;
```

write10 スクリプト

```
¥set naccounts 100000 * :scale
¥setrandom aid00 1 :naccounts
¥setrandom aid01 1 :naccounts
¥setrandom aid02 1 :naccounts
¥setrandom aid03 1 :naccounts
¥setrandom aid04 1 :naccounts
¥setrandom aid05 1 :naccounts
¥setrandom aid06 1 :naccounts
¥setrandom aid07 1 :naccounts
¥setrandom aid08 1 :naccounts
¥setrandom aid09 1 :naccounts
BEGIN;
UPDATE pgbench_accounts SET filler=repeat(md5(current_timestamp::text),2) WHERE aid= :aid00;
UPDATE pgbench_accounts SET filler=repeat(md5(current_timestamp::text),2) WHERE aid= :aid01;
UPDATE pgbench_accounts SET filler=repeat(md5(current_timestamp::text),2) WHERE aid= :aid02;
UPDATE pgbench_accounts SET filler=repeat(md5(current_timestamp::text),2) WHERE aid= :aid03;
UPDATE pgbench_accounts SET filler=repeat(md5(current_timestamp::text),2) WHERE aid= :aid04;
UPDATE pgbench_accounts SET filler=repeat(md5(current_timestamp::text),2) WHERE aid= :aid05;
UPDATE pgbench_accounts SET filler=repeat(md5(current_timestamp::text),2) WHERE aid= :aid06;
UPDATE pgbench_accounts SET filler=repeat(md5(current_timestamp::text),2) WHERE aid= :aid07;
UPDATE pgbench_accounts SET filler=repeat(md5(current_timestamp::text),2) WHERE aid= :aid08;
UPDATE pgbench_accounts SET filler=repeat(md5(current_timestamp::text),2) WHERE aid= :aid09;
END;
```


© 2016 PostgreSQL Enterprise Consortium

(b) 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、今年度は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（warm start）。なお、PostgreSQL を動作させたままで行う測定3回の内、中央値を測定結果として用いることにしましたので、主に1回目の測定結果に影響する pg_prewarm が測定結果（中央値）に与える影響は大きくないと考えられます。

3回の pgbench 実行（5分間）は、PostgreSQL は起動したままで繰り返しました。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、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）がリセットされます。

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 -D num_clients=[clients] [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
“-D num_clients=[clients]” は，3.6.3 の検証（使用スクリプト:rr_write10）でのみ使用
```

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。

(c) 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、ここで行った測定方法では両者の差はごく僅かでした。本報告書では、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。なお、異なるコア数での CPU 使用率の比較を容易とするため、本節では、sar コマンドにより得られる CPU 使用率（最大 100%）にコア数を乗じた値を CPU 使用率として表示します。つまり、本節での CPU 使用率 100%は「1 コアを完全に使い切った状態」を意味し、18, 36, 54, 72 コアでの最大の CPU 使用率は、各々、1800, 3600, 5400, 7200%となります。

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（9.4.5 と 9.5beta2）、pgbench スクリプト（トランザクションの投入方法の違い、前節参照）です。

CPU コア数については、1 CPU チップ（ソケット）が 18 コアであることを考慮し、18, 36, 54, 72 の 4 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。

© 2016 PostgreSQL Enterprise Consortium

コア数変更スクリプト
コア番号と CPU chip の対応関係は検証サーバ機の調査に基づいて設定

```
#!/bin/bash

cd /sys/devices/system/cpu

CHIP0=' 0  1  2  3  4  5  6  7  8 36 37 38 39 40 41 42 43 44 '
CHIP1=' 9 10 11 12 13 14 15 16 17 45 46 47 48 49 50 51 52 53 '
CHIP2='18 19 20 21 22 23 24 25 26 54 55 56 57 58 59 60 61 62 '
CHIP3='27 28 29 30 31 32 33 34 35 63 64 65 66 67 68 69 70 71 '

case ${1:-72} in
  72) cpuON="${CHIP0} ${CHIP1} ${CHIP2} ${CHIP3}"
    cpuOFF="";;
  54) cpuON="${CHIP0} ${CHIP1} ${CHIP2}"
    cpuOFF="${CHIP3}";;
  36) cpuON="${CHIP0} ${CHIP1}"
    cpuOFF="${CHIP2} ${CHIP3}";;
  18) cpuON="${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
```

今年度は、上記の測定パラメータに加え、トランザクション分離レベル（read committed と repeatable read）、huge pages 使用の有無、SQL の実行計画使い回しの有無による性能差についても検証を行いました。これらの内容と結果は 3.6 節で説明します。

© 2016 PostgreSQL Enterprise Consortium

3.4. 結果

3.4.1. CPU スケーラビリティ

本節では、CPU コア数の違いによる基本性能特性の違いを示します。

(a) write1

write1 を pgbench スクリプトとしたときの基本性能特性を図 3.1 と図 3.2（スループット）、および図 3.3 と図 3.4（CPU 使用率）に示します。9.4.5 と 9.5beta2 の両方とも、スループットが最大となったのは 36 コアのケースで、54 コア、72 コアのスループットは若干下がりましたが、ほぼ同程度となりました。また、スループットがピーク値を示したクライアント数から更に増加させたときの性能低下は 9.4.5 の方が小さいことから、負荷の高い領域での動作は 9.4.5 の方が良好であるということも分かりました。CPU 使用率に関しては、18 コアでは飽和しましたので CPU 資源がボトルネックとなっていましたが、他のケースでは飽和しておらず、CPU 以外の要素がボトルネックであったことが伺えます。特に、54 と 72 コアの場合でも CPU 使用率は 3600%以下に留まっており、CPU 資源が有効に活用されていない状況となっていました。

18 コアの基本性能特性は、9.4.5 と 9.5beta2 で大きな違いがありました。9.4.5 ではクライアント数が 70 程度でピークを示し、それ以降は性能が飽和するという特性を示したのに対し、9.5beta2 ではクライアント数の増加に伴ってスループットも向上していくという、9.4.5 や 9.5beta2 の 36, 54, 72 コアのケースとは異なる特性を示しました。

(b) write10

write10 を pgbench スクリプトとしたときの基本性能特性を図 3.5 と図 3.6（スループット）、および図 3.7 と図 3.8（CPU 使用率）に示します。write1 との比較では、スループットが2倍程度になっていたのに対し、CPU 使用率では、若干の増加は見られますが、36, 54, 72 コアで 3600%を超えていない等 write1 とほぼ同じでした。10 回の update を 1 トランザクションにまとめたことで CPU 使用量が削減されたと考えられます。

スループットが最大となったのは、write1 と同様 36 コアで、54 や 72 コアに増えると低下しました。また、複数 CPU（36, 54, 72 コア）の場合、ピーク性能を示したクライアント数を超えるとスループットが低下しました。一方、1CPU（18 コア）では CPU 資源が飽和しており、36 コアよりも低いスループットとなりましたが、write1 と同様、ピークを示したクライアント数から更に増やしても性能の低下は見られませんでした。これより、1CPU と複数 CPU では、クライアント数の増加が性能に与える影響は異なっていることが分かりました。

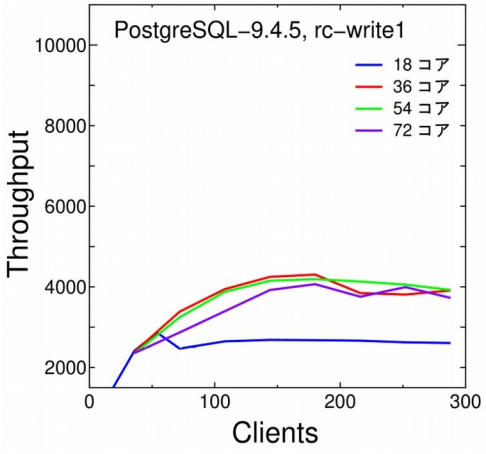

図 3.1: write1, PostgreSQL-9.4.5 でのスループット

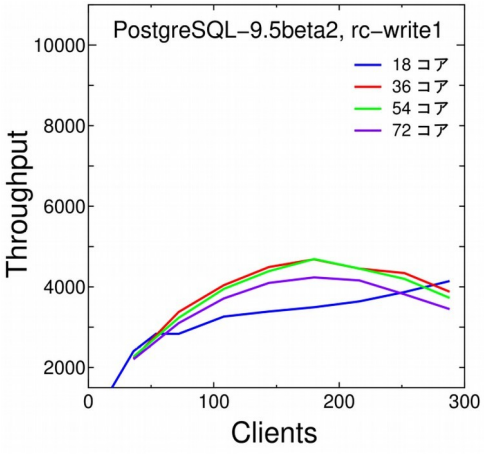

図 3.2: write1, PostgreSQL-9.5beta2 でのスループット

© 2016 PostgreSQL Enterprise Consortium

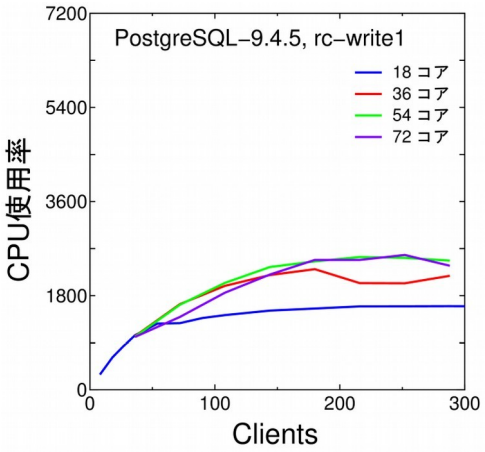

図 3.3: write1, PostgreSQL-9.4.5 での CPU 使用率

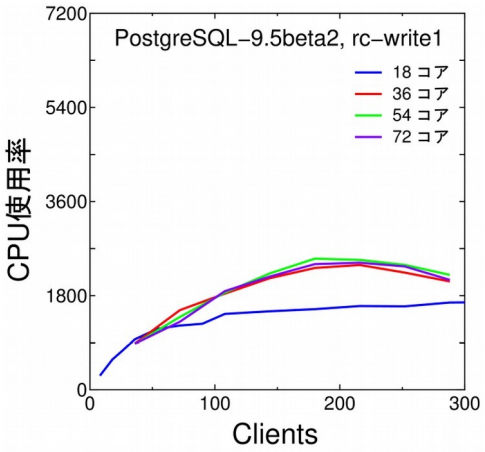

図 3.4: write1, PostgreSQL-9.5beta2 での CPU 使用率

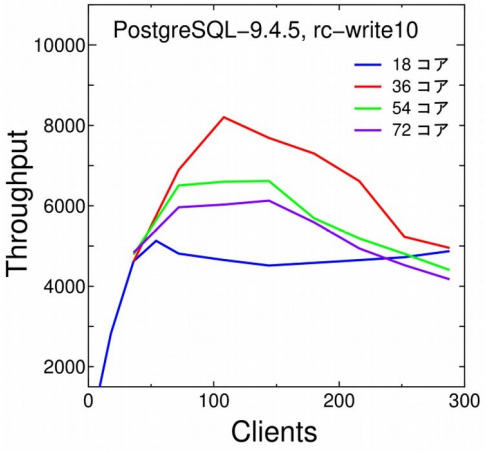

図 3.5: write10, PostgreSQL-9.4.5 でのスループット

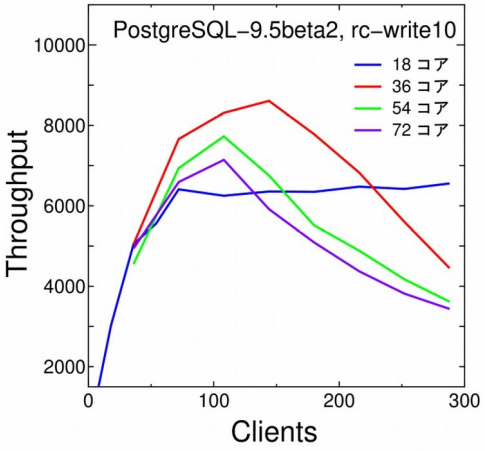

図 3.6: write10, PostgreSQL-9.5beta2 でのスループット

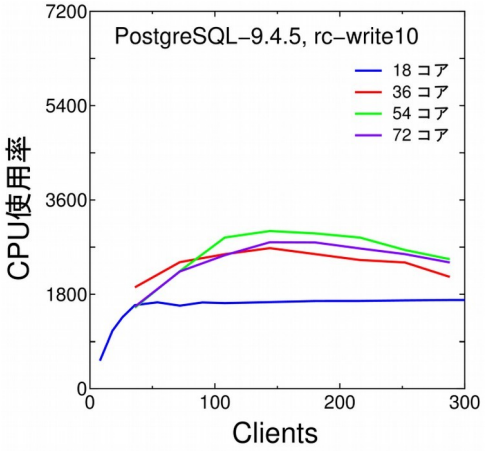

図 3.7: write10, PostgreSQL-9.4.5 での CPU 使用率

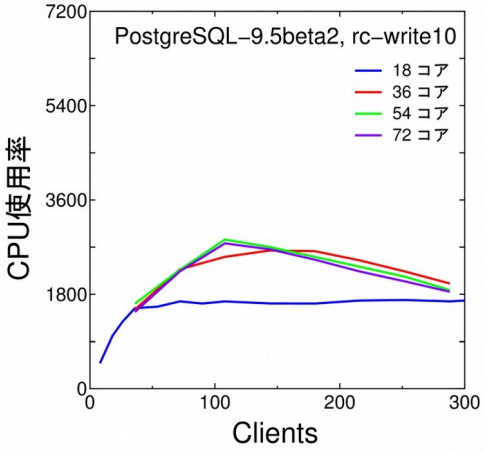

図 3.8: write10, PostgreSQL-9.5beta2 での CPU 使用率

© 2016 PostgreSQL Enterprise Consortium

3.4.2. バージョン比較

本節では、PostgreSQL-9.4.5 と 9.5beta2 の基本性能特性（スループット）を比較します。本節で示す測定結果は 3.4.1 で示したものと同じで、1枚のグラフで 9.4.5 と 9.5beta2 についての同一測定パラメータでの結果を示しました。

図 3.9 と図 3.10 に 18 コア、図 3.11 と図 3.12 に 36 コア、図 3.13 と図 3.14 に 54 コア、図 3.15 と図 3.16 に 72 コアの結果を示します。各々、pgbench スクリプトが write1 と write10 の結果です。総てのケースについてピーク性能は 9.5beta2 の方が高くなっていることから、9.5 で導入された改良が効果を発揮していると考えられます。なお、前節でも述べましたが、スループットがピーク値を示したクライアント数から更に増加させると 9.5beta2 の方が性能低下は大きく、9.4.5 に逆転されるケースも見受けられました。

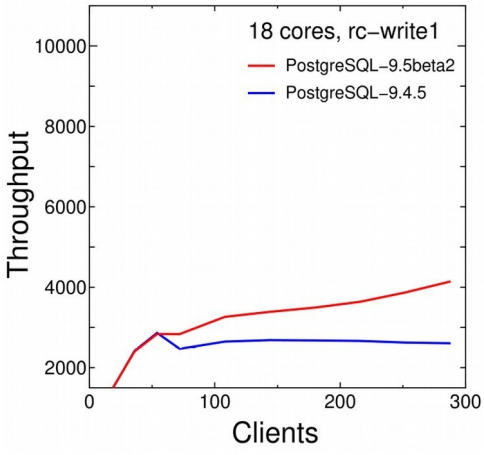

図 3.9: 18 コア, write1 でのスループット

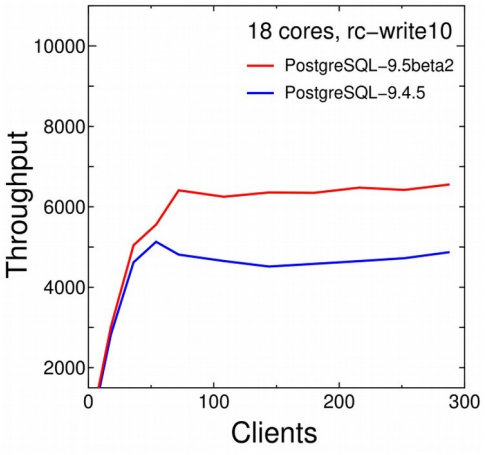

図 3.10: 18 コア, write10 でのスループット

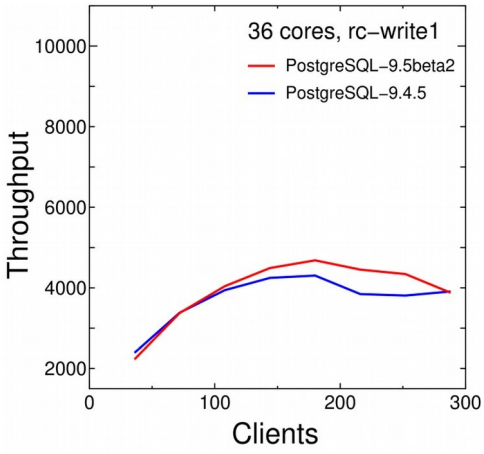

図 3.11: 36 コア, write1 でのスループット

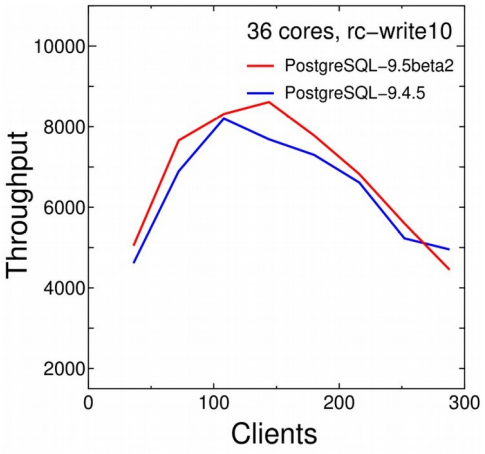

図 3.12: 36 コア, write10 でのスループット

© 2016 PostgreSQL Enterprise Consortium

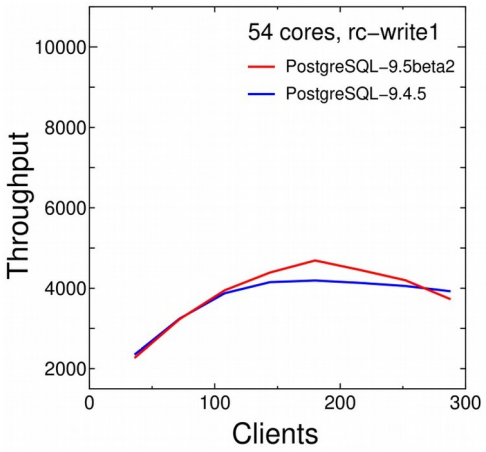

図 3.13: 54 コア, write1 でのスループット

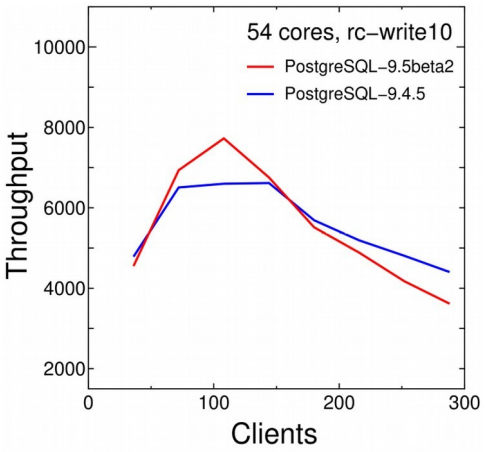

図 3.14: 54 コア, write10 でのスループット

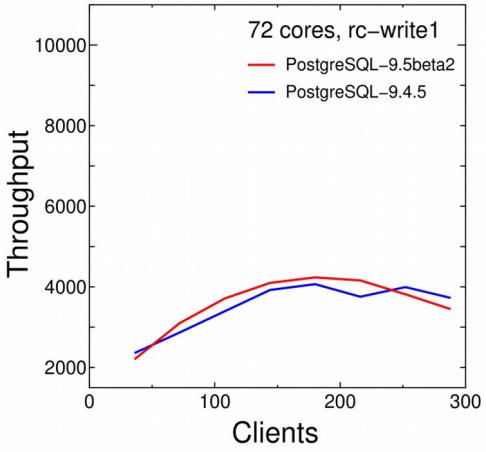

図 3.15: 72 コア, write1 でのスループット

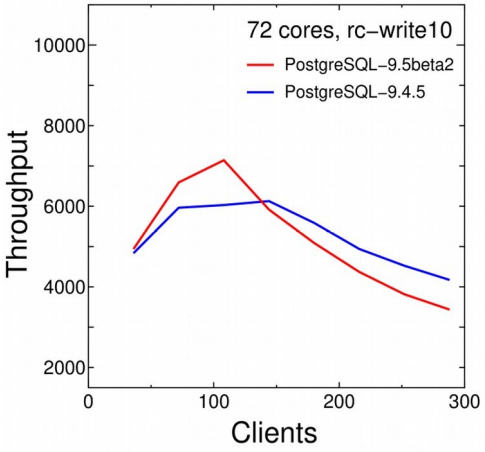

図 3.16: 72 コア, write10 でのスループット

© 2016 PostgreSQL Enterprise Consortium

3.5. 考察

ここでは、昨年度から開始した更新処理の定点観測を継承し、ほぼ同様の処理を PostgreSQL で行わせたときの性能を測定しました。昨年度と同様の結果となったのは、更新処理に関しては良好な CPU スケーラビリティは確認できなかった点です。

更新処理での CPU スケーラビリティが良好でない原因は、昨年度の検証レポートで示した PostgreSQL の内部処理として行われている排他制御（lwlock）が、9.5beta2 でもボトルネックとなっている可能性が高いと言えそうです。なお、lwlock に関しては、「共有モードでのロック獲得処理のスケーラビリティ改善」が 9.5 での強化点の一つとなっています。今年度の検証で 9.5beta2 のスループットの方が 9.4.5 より高くなっていたのは、この強化点の効果が表れたものと考えられます。一方、9.5beta2 では、スループットがピーク値を示したクライアント数から更に増加させたときの性能低下が 9.4.5 より大きいという結果も得られました。9.5 ではクライアント数を増やしすぎないよう、より注意する必要がありそうです。

昨年度と異なる第一の点は、複数 CPU（36, 54, 72 コア）構成の性能が改善されていることです。上で述べた「lwlock の改善」が行われている 9.5beta2 だけでなく、9.4.5 についても、複数 CPU 構成でのスループットが高くなっており、両方とも、1CPU（18 コア）構成よりも複数 CPU 構成の方が高いスループットとなりました。第二の点は、1CPU（今年度は 18 コア、昨年度は 15 コア）での 9.4.の結果です。昨年度は 9.4rc1 で約 40,000 update／秒のスループットとなったのに対し、今年度の検証では、9.4.5 で約 30,000 update／秒（3,000 pgbench script／秒）に留まりました。なお、9.5beta2 では、約 40,000 update／秒のスループットとなっています。

昨年度と異なる傾向となった原因については、マシン時間の制約のため詳細な追究ができませんでしたが、測定環境の昨年度と今年度の差異による可能性が高いと考えています。具体的には、以下のような点です。1) 検証ハードウエア、特に検証用サーバの差異、例えば、CPU は E7-4890v2×4 から E7-8890v3×4 に変わりました。2) OS が Red Hat Engerprise Linux の 6.5 から 7.2 になったことで、CPU スケジューラが変更されています。3) CPU コア数の変更方法を、maxcpus を指定してのリブートから、CPU hotplug 機能（/sys/devices/system/cpu/cpu*/online に 0 または 1 を echo）を利用する方法に変更しました。4) pgbench スクリプト実行1回で 10 回の update 処理を行わせるようにしました。昨年度は 1 回の pgbench スクリプトで 1 回の update を実行させていました。5） PostgreSQL-9.4 のバージョンは 9.4rc1 から 9.4.5 になりました。

今年度は、スループットを向上させる一手として、10 回の update を 1 トランザクションで行う処理形態（write10）を測定対象に含めました。期待通り、write10 は、write1 の約 2 倍のスループットが得られましたので、複数の update を 1 トランザクションにまとめる効果は大きいことが確認できました。一方、write1 と write10 の CPU 使用率には大きな差はなかったことから、write1 では、トランザクション 1 回にかかる固定的な処理の割合が、表を更新する処理（UPDATE 回数に比例する部分）と比較して大きな割合を占めているものと考えられます。write10 では、1 回の update にかかる固定的な処理が write1 の 1/10 になります。これが同じ CPU 使用率にも関わらず、性能が向上した原因と考えられます。

このように複数の update を 1 つのトランザクションにまとめると、性能面でメリットがある反面、デッドロックへの対策が必要な点に注意する必要があります。今回の測定実験では発生しませんでしたが、2 つの同じ行を 2 つのトランザクションが同時に逆方向に更新しようとすると、デッドロックが発生します。デッドロックを起こさない方法には、1) 更新する対象行を各トランザクションで別々として重ならないようにする、2) pgbench スクリプトの機能では実装できませんでしたが、更新する行の順番を、総てのトランザクションが同じルールで決める、があります。また、PostgreSQL はタイムアウトを契機としてデッドロックの発生を検出し、関与しているトランザクションを abort させるようになっていますので、デッドロック発生を前提としてアプリケーションを作成する方法もあります。具体的には、デッドロックのためにトランザクションが abort された場合、そのトランザクションを再実行させるような制御をアプリケーションに組み込んでおくわけです。今回の検証のように、行数の大きい表の行をランダムに選択して更新するような場合は、デッドロックが発生する可能性は低いですが、ゼロにはなりませんので、備えが必要です。なお、1 トランザクションで 1 回の update を行わせる場合は、複数のトランザクションが複数の行を同時に逆方向に更新するような状況は発生しません。

© 2016 PostgreSQL Enterprise Consortium

3.6. CPU 負荷削減や ProcArrayLock 競合軽減の検討と評価

今年度は、PostgreSQL サーバの CPU 負荷削減や ProcArrayLock 競合軽減に効果のある方法の幾つかについて検証を実施しました。CPU 負荷は huge pages や prepare 文を使用することで軽減できます。また、ProcArrayLock 競合は、トランザクション分離レベルを repeatable read とすることで軽減されます。本節では、これらの方法により期待できる効果についての検証結果に加え、使う上での注意点や発生する可能性のある問題点を含めて説明します。なお、本節では、検証結果の説明が冗長になるのを避けるため、PostgreSQL 9.5beta2 の結果のみを示すことにします。

3.6.1. Huge pages 利用

Huge pages を使用すると、OS（Linux)の仮想記憶がメモリを管理する単位、すなわち、TLB の 1 エントリがカバーするメモリ領域が通常より大きくなり、TLB ミスが低減されます。これにより、CPU 負荷を低減することができます。今回の検証では、huge pages の大きさは 2MB としました。通常のページ・サイズ（4K バイト）の 500 倍の大きさです。

PostgreSQL で huge pages が使えるようになったのは、9.4 からで、postgresql.conf の huge_pages パラメータ（on, off, try が指定可能）により設定します。デフォルトは try で、huge pages が使える場合はそれを使用し、使えない場合は、通常のページを使うようになっています。

(a) 設定

PostgreSQL で huge pages を使用するには、まず Linux で必要な大きさの huge pages を用意し、huge_pages を on または try とした上で PostgeSQL を起動する必要があります。PostgreSQL では huge pages を共有メモリとして使用しますので、必要な huge pages の大きさは、ほぼ PostgreSQL の共有バッファの大きさで決まります。しかしながら、PostgreSQL の共有メモリは、共有バッファ以外の用途でも使用されますので、必要となる huge pages の大きさは共有バッファの大きさよりも大きくなります。一般には、この大きさを事前に調べることは困難です。そこで、この検証では、まず、huge pages を確保しない状態で huge_pages パラメータを on にして PostgreSQL を起動しました。このようにすると、必要な共有メモリの大きさがログに出力されたメッセージから判明しますので、その値から確保する huge pages の大きさ（ページ数）を計算しました。

共有バッファを 384GB、huge_pages パラメータを on として PostgreSQL を起動すると、ログファイルに、"（省略） To reduce the request size (currently 422382649344 bytes), （省略）"というエラーメッセージが出力されました。これより、必要な huge pages 数は 201408 と計算できます（小数点以下は切り上げ）。Huge pages の用意は、'echo 201408 > /proc/sys/vm/nr_hugepages' により行いました。この操作には root 権限が必要なことと、サーバをリブートすると、再度、この操作が必要になる点に注意してください。なお、サーバのブート時、自動的に huge pages の確保を行わせるには、/usr/lib/sysctl.d/00-system.conf （RHEL6 の場合は /etc/sysctl.conf）に' vm.nr_hugepages = 201408' という行を記載しておきます。

(b) 効果

Huge pages を使用したときの基本性能特性（スループット）を、図 3.17-図 3.18（write1 と write10）に示します。総てのケースで huge pages 使用によるスループット向上を確認できました。顕著な性能改善がみられたのは、write10, 18 コアのケースで、スループットが約 1.5 倍になりました。Huge pages を使用しない場合は複数 CPU よりもスループットは低い値に留まっていましたが、図 3.18 より、huge pages を使用すると、18 コアのスループットは複数 CPU（36,54,72 コア）よりも高い値となりました。一方、18 コアの write1（図 3.17）での改善は限定的で、複数 CPU のスループットを大幅に超えるような改善は見られませんでした。Huge pages 使用により、常に write10 でみられたような改善が期待できるわけではないといえます。

(c) 考察

ここで行った検証では、総てのケースで huge pages 利用による性能向上を確認できましたので、PostgreSQL のサーバでは huge pages を利用すべき、といえます。但し、huge pages として確保したメモリ領域は、キャッシュやバッファといった通常の OS 用途で使用できないという欠点がありますので、DB（PostgreSQL）以外の処理も行わせるようなサーバに関しては、利用は避けた方が良いケースもあると思われます。逆に、PostgreSQL 専用の物理サーバであれば、huge pages を利用しない理由はないと言えます。

なお、今回利用した huge pages は、transparent huge pages とは異なりますので、注意が必要です。Transparent huge pages は、/proc/sys/vm/nr_hugepages により予め huge pages 用の領域を確保しなくても、OS が動的に huge page を用意するという仕組みです。Huge pages を確保する手間は不要ですが、

© 2016 PostgreSQL Enterprise Consortium

PostgreSQL を動作させる場合は、動的に huge pages を用意するオーバヘッドが大きくなってしまうケースがあります。実際、『主だった DB においては、transparent huge pages は使用しないという設定が推奨さている、という状況にある』というような報告も見受けられます [1]。

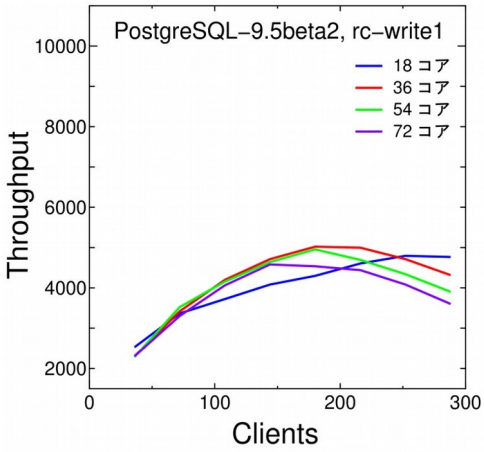

図 3.17: Huge pages 利用, write1 でのスループット

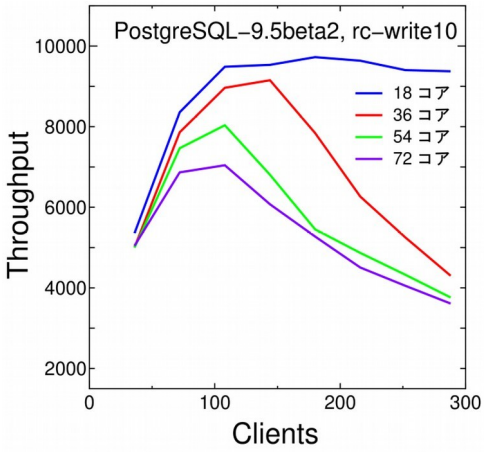

図 3.18: Huge pages 利用, write10 でのスループット

3.6.2. Prepare 文利用

PostgreSQL は、他の多くの RDBMS と同様、SQL 文によりトランザクション処理が要求されると、その SQL 文を解釈して実行計画を作成した後、その実行計画に基づいてトランザクションを実行するようになっており、通常は、トランザクション処理の度に SQL 文から実行計画を作成するようになっています。一方、作成された実行計画を保存しておき、後から処理要求された SQL 文の形が同じ場合は、保存された実行計画を使用することで、SQL 文から実行計画を作成する処理を省く方法（prepare 文）も用意されています。

(a) 設定

Pgbench にも、prepare 文を使って最初に実行計画を作成しておき、これをベンチマーク実行中に使用するという方法で PostgreSQL に対してトランザクション実行を要求する機能が用意されており、pgbench のオプションに ”-M prepared” を加えれば使えるようになっています。

(b) 効果

Prepare 文を使用したときの基本性能特性（スループット）を、図 3.19-図 3.20（write1 と write10）に示します。Huge pages を利用したとき同様、prepare 文の利用により、1) 総てのケースでのスループット向上、2) write10, 18 コアのケースで、スループットが約 1.6 倍になるという顕著な性能向上が確認できました。

(c) 考察

総てのケースで prepare 文利用による性能向上を確認できましたので、huge pages と同様、prepare は利用した方が良いように思われます。しかしながら、OS 機能である huge pages とは異なり、prepare 文を利用する際は、1) prepare 文を実行したデータベースセッションの期間中でのみ有効な点、2) 同じ形の SQL 文を繰り返し実行させる場合は有用ですが、毎回、実行させる SQL 文の形が異なるような場合には効果は期待できない、という点に注意する必要があります。マニュアルの prepare 文の項にも、『プリペアド文の利点を最大限に発揮できるのは、単一のセッションで同類の問い合わせを多数実行する場合です。』と記載されています。

また、極端なケースではありますが、最適な実行計画がデータベースの状態（典型的には行数や selectability）によって変わってしまうような場合は、ある時点で prepare 文により作成した実行計画が、時間の経過とともに最適でなくなってしまうような状況もあり得ます。このように、prepare 文については、その特性を十分に理解した上で使いこなす必要があるといえます。

1 http://blog.gachapin-sensei.com/archives/618881.html

© 2016 PostgreSQL Enterprise Consortium

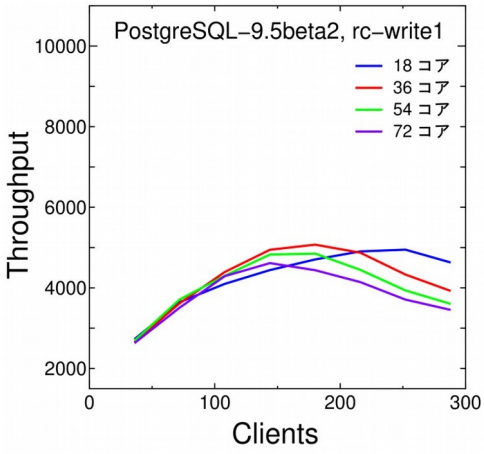

図 3.19: Prepare 文利用, write1 でのスループット

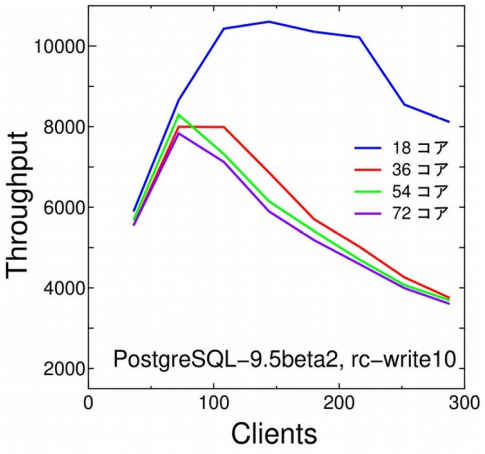

図 3.20: Prepare 文利用, write10 でのスループット

3.6.3. Isolation level を repeatable read にする

ProcArrayLock は、主に、現在実行中のトランザクションを調べる操作とトランザクションの実行状態を変更する操作の間で矛盾が生じないようにするための排他制御で使用されています。PostgreSQL がデフォルトとしている isolation level の read committed では、各々の SQL 文の実行に先立って、現在実行中のトランザクションを調べる操作が行われます。このときは、ProcArrayLock に対して、共有モードでのロック獲得要求が行われます。一方、トランザクションの実行状態を変更する操作は、トランザクションが終了したときに行われます。このときの ProcArrayLock に対するロック獲得要求は、排他モードで行われます。

Isolation level を repeatable read（または serializable）にすると、現在実行中のトランザクションを調べる操作は、各 SQL 文の実行開始時点ではなく、トランザクションの開始時点で行われるようになります。つまり、複数の SQL 文を1つのトランザクションで実行させる場合は、isolation level を repeatable read とすることで、ProcArrayLock に対する獲得要求の頻度を下げることが可能です。ProcArrayLock を獲得する操作がボトルネックとなっている場合は、この isolation level の変更によってボトルネックが緩和され、性能向上につながることが期待できます。

(a) 設定

postgresql.conf に”default_transaction_isolation = 'repeatable read'” という行を追加することで、トランザクションの default isolation level を repeatable read に変更することができます。しかしながら、この変更だけでは serialization failure が発生する問題のため、pgbench を安定して動作させることはできませんでした。

Serialization failure は、同じ行を複数のワーカプロセスが同時に UPDATE しようとした場合に発生しますので、この検証では、各ワーカプロセスが UPDATE する行が重ならないようにして測定を行いました。具体的には、以下の pgbench スクリプトを用いました。

© 2016 PostgreSQL Enterprise Consortium

rr_write10 スクリプト

```
¥set naccounts 100000 * :scale
¥set caccounts :naccounts / :num_clients
¥set mydiff :caccounts - :client_id
¥setrandom aid00 1 :caccounts
¥set aid00 :num_clients * :aid00
¥set aid00 :aid00 - :mydiff
¥setrandom aid01 1 :caccounts
¥set aid01 :num_clients * :aid01
¥set aid01 :aid01 - :mydiff
¥setrandom aid02 1 :caccounts
¥set aid02 :num_clients * :aid02
¥set aid02 :aid02 - :mydiff
¥setrandom aid03 1 :caccounts
¥set aid03 :num_clients * :aid03
¥set aid03 :aid03 - :mydiff
¥setrandom aid04 1 :caccounts
¥set aid04 :num_clients * :aid04
¥set aid04 :aid04 - :mydiff
¥setrandom aid05 1 :caccounts
¥set aid05 :num_clients * :aid05
¥set aid05 :aid05 - :mydiff
¥setrandom aid06 1 :caccounts
¥set aid06 :num_clients * :aid06
¥set aid06 :aid06 - :mydiff
¥setrandom aid07 1 :caccounts
¥set aid07 :aid07 - :mydiff
¥set aid07 :num_clients * :aid07
¥setrandom aid08 1 :caccounts
¥set aid08 :num_clients * :aid08
¥set aid08 :aid08 - :mydiff
¥setrandom aid09 1 :caccounts
¥set aid09 :num_clients * :aid09
¥set aid09 :aid09 - :mydiff
BEGIN;
UPDATE pgbench_accounts SET filler=repeat(md5(current_timestamp::text),2) WHERE aid= :aid00;
UPDATE pgbench_accounts SET filler=repeat(md5(current_timestamp::text),2) WHERE aid= :aid01;
UPDATE pgbench_accounts SET filler=repeat(md5(current_timestamp::text),2) WHERE aid= :aid02;
UPDATE pgbench_accounts SET filler=repeat(md5(current_timestamp::text),2) WHERE aid= :aid03;
UPDATE pgbench_accounts SET filler=repeat(md5(current_timestamp::text),2) WHERE aid= :aid04;
UPDATE pgbench_accounts SET filler=repeat(md5(current_timestamp::text),2) WHERE aid= :aid05;
UPDATE pgbench_accounts SET filler=repeat(md5(current_timestamp::text),2) WHERE aid= :aid06;
UPDATE pgbench_accounts SET filler=repeat(md5(current_timestamp::text),2) WHERE aid= :aid07;
UPDATE pgbench_accounts SET filler=repeat(md5(current_timestamp::text),2) WHERE aid= :aid08;
UPDATE pgbench_accounts SET filler=repeat(md5(current_timestamp::text),2) WHERE aid= :aid09;
END;
```


© 2016 PostgreSQL Enterprise Consortium

(b) 効果

Isolation level を repeatable read にしたときの基本性能特性（スループット）を、図 3.21 に示します。Isolation level 変更が効果を発揮するのは、複数の SQL 文を 1 トランザクションで実行させる場合ですので、ここでは write10 の結果のみを示します。（1SQL を 1 トランザクションで実行させる場合、現在実行中のトランザクションを調べる操作とトランザクションの実行状態を変更する操作の頻度は repeatable read と read committed で同じになります。）

図 3.6 と図 3.21 の比較より、isolation level を read committed から repeatable read に変更することでスループットが向上していることが分かります。また、この結果は、ProcArrayLock の競合がボトルネックになっていることを実証しているとも考えられます。

(c) 考察

I/O やネットワークがボトルネックになっていないにも関わらず、CPU コア数を増やしてもスループット性能が向上しない場合は、ロック競合がボトルネックになっている可能性が疑われます。現在の PostgreSQL では、その典型例が ProcArrayLock の競合で、本節で示したように、isolation level を repeatable read とすることで軽減できる可能性があります。

しかしながら、repeatable read の場合は、常には serialization failure のためにトランザクションが abort する可能性を考慮しておく必要があります。pgbench では、serialization failure が発生すると、実行を終了してしまうようになっており、安定したベンチマーク実行ができないとなりましたが、pgbench スクリプトを変更し、各ワーカプロセスが操作する行が重ならないようにすることで性能測定を実施することができました。実際のアプリケーションで同様の対策が可能か否かはアプリケーションの特性次第と考えられますので、isolation level 変更による性能向上を狙う際は、このような点を念頭に入れた上で行う必要があります。

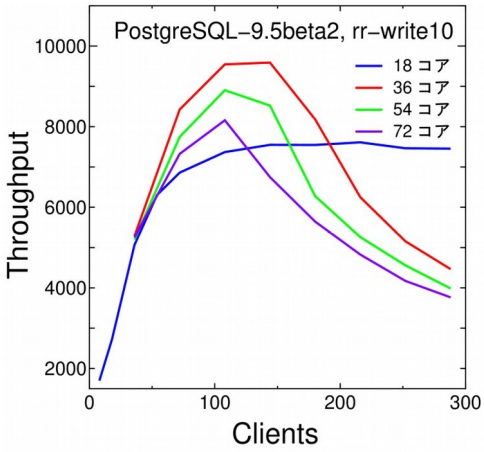

図 3.21: Repeatable Read でのスループット

© 2016 PostgreSQL Enterprise Consortium

4. Parallel Vacuum 検証

4.1. 検証目的

PostgreSQL 9.5 から、Parallel Vacuum の機能が実装されました。PostgreSQL 9.4 以前では単一のワーカプロセスが複数のテーブルに対して逐次 Vacuum や Analyze を実行していました。PostgreSQL 9.5 からは、複数のワーカプロセスで複数のテーブルに対して同時に Vacuum や Analyze を実行できるようになりました。この機能は、vacuumdb コマンドで”--jobs”オプションに並列に実行するジョブ数を指定することで利用できます。この機能が有効活用される場面として、例えば大規模なパーティショニングされたテーブルに対する Vacuum、Analyze の高速化や、XID 回収のための VACUUM FREEZE を実行した際に生じる全テーブルスキャンの時間短縮といったことが期待できます。本年度の PGECons WG1 では比較的大きなテーブルを複数用意し、VACUUM FREEZE を並列実行した際にどのような性能特性が出るかを検証しました。

4.2. 検証構成

4.2.1. ハードウェア構成

本検証に用いた構成を表 4.1 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 4.1: 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 E5-2690v3@2.60GHz(12 コア) x 2 合計 24core （ハイパースレッディングはオフ）
	搭載メモリ	256GB
	内蔵ストレージ	HDD 1.8TB 12G SAS 10k x 2
	DB 格納用ストレージ	Fiber Channel 接続（16Gbps）SAN 以下の 2 領域を利用 領域 1： PostgreSQL の DB 領域 領域 2: WAL 領域
	OS	それぞれの領域は HDD 1.8TB 6G SAS 10k x 4 (RAID10、実効容量約 3TB）で構成
	DBMS	Red Hat Enterprise Linux 7.2 PostgreSQL 9.5 beta2

4.2.2. 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 周回問題回避のための 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。PostgreSQL 9.5 から、WAL のサイズの指定方法が変わっており、今回は PostgreSQL 9.4 以前における checkpoint_segments=256 相当の値になるように、max_wal_size を指定しています。また、shared_buffer は搭載メモリの 25%の 64GB としました。

© 2016 PostgreSQL Enterprise Consortium

postgresql.conf（変更箇所のみ）

```
autovacuum = off
autovacuum_freeze_max_age = 1000000000
wal_level = archive
max_wal_size = 4GB
shared_buffers = 64GB
logging_collector = on
log_min_messages = INFO
log_line_prefix = '%t [DB=%d] [USER=%u] [PID=%p] APP=(%a) '
log_lock_waits = on
log_checkpoints = on
log_connections = on
```

4.3. 検証方法

4.3.1. 環境構築

複数の Vacuum のワーカプロセスを立てた場合、複数のワーカプロセスが同一のテーブルに対して VACUUM を実行することはなく、それぞれ異なるテーブルに対して VACUUM を実行します。したがって、なるべく各ワーカプロセスでの処理時間をそろえるために、ほぼ同一サイズのテーブルを複数用意することにしました。今回はサーバの CPU コア数が 24 個のため、2 倍の 48 個のテーブルを用意し、ジョブ数がコア数を超えた場合の結果も測定できるようにします。

ここでは、以下の前提条件を満たしているとします。

- PostgreSQL 9.5 beta2 が/usr/local/pgsql にインストールされている
- $PGDATA=/usr/local/pgsql/data が設定されている
- $PATH に/usr/local/pgsql/bin が含まれている

まず、PostgreSQL の DB を領域 1(=/data1)に、WAL を領域 2(=/data2)に作成し、PostgreSQL サーバを起動します。

```
$ initdb --local=C
$ mv $PGDATA /data1/
$ ln -s /data1/data $PGDATA
$ mv $PGDATA/pg_xlog /data2/
$ ln -s /data2/pg_xlog $PGDATA/pg_xlog
$ pg_ctl start
```

DB の初期化が完了したら、以下に示すスクリプト prepare.sh を実行し、48 個のほぼ均一なテーブルを作成します。ここでは pgbench で利用される pgbench_acconunts テーブルを利用します。スケールファクターは 200(=2000 万行、約 3GB)、フィルファクターは 80 とします。これは HOT が動くようなテーブル設計にするためです。pgbench_accounts 以外のテーブルは利用しないので消しておきます。

© 2016 PostgreSQL Enterprise Consortium

prepare.sh

```
#!/bin/bash
scale_factor=200
fill_factor=80
num_of_tables=48

for i in $( seq 1 $num_of_tables )
do
    echo "Begin : `date -R`"
    pgbench -i -s $scale_factor -F $fill_factor
    psql -c "alter table pgbench_accounts rename to pgbench_accounts_$i"
    echo "End : `date -R`"
done

psql -c "drop table pgbench_tellers;"
psql -c "drop table pgbench_branches;"
psql -c "drop table pgbench_history;"
```

生成した 48 個のテーブルに対して XID 凍結処理に十分な時間がかかるように、生成したテーブルを適度に更新します。このとき、pgbench のカスタムスクリプトを使って、XID を適度に進めます。

以下に、pgbench 用カスタムスクリプトのテンプレートを示します。このテーブルの「TBLNO」の部分をシェルスクリプト側から sed コマンドを使ってテーブル番号に書き換えて利用します。

update.sql.template

```
¥set naccounts 100000 * :scale
¥setrandom aid 1 :naccounts

UPDATE pgbench_accounts_TBLNO SET filler=repeat(current_timestamp::text,2) WHERE aid=:aid;
```

テーブルの XID を進める処理を以下に示すスクリプトで行います。ここでは、上述のテンプレートを用いて、48 個のテーブルすべてに UPDATE の処理を十分な実行し、XID を進めています。

update.sh

```
#!/bin/bash
scale_factor=200
num_of_tables=48
num_of_clients=24
exec_time=180

for i in $( seq 1 $num_of_tables )
do
    sed -e "s/TBLNO/$i/g" update.sql.template > update.sql
    pgbench -c $num_of_clients -T $exec_time -s $scale_factor -f update.sql -n
done
psql -c "checkpoint"
sleep 300
psql -c "checkpoint"
```

以上でデータの生成が終わりです。今後はこのデータを繰り返し利用するので、物理バックアップを取っておきます。

```
$ pg_ctl stop
$ mkdir ~/backup
$ cp -R $PGDATA ~/backup/
$ cp -R $PGDATA/pg_xlog ~/backup/
```


© 2016 PostgreSQL Enterprise Consortium

4.3.2. 測定

ジョブ数を変えながら、VACUUM にかかった時間を測定します。並行して sar の取得も行いきます。測定の際には、以下のようなスクリプトを用意し、実行しました。

perf.sh

```
#!/bin/bash
DIR="`date +%Y%m%d_%H%M%S`";
INTERVAL=10
CONNS='1 2 4 8 12 16 20 24 28 32 40 48'
mkdir $DIR

for i in $CONNS
do
    mkdir $DIR/$i
    #restoreDB
    pg_ctl stop -m immediate
    sleep 10
    rm -rf /data1/data
    cp -R ~/backup/data /data1/
    rm -rf /data1/data/pg_log/*
    rm -rf /data2/pg_xlog
    cp -R ~/backup/pg_xlog /data2/
    pg_ctl start
    sleep 10

    # load tables to shared buffer
    echo "---------"  >> $DIR/vacuum.time
    echo "JOBS = $i"  >> $DIR/vacuum.time
    echo "BEGIN pg_prewarm: `date`" >> $DIR/vacuum.time
    psql -c "DROP EXTENSION IF EXISTS pg_prewarm;"
    psql -c "CREATE EXTENSION pg_prewarm;"
    for j in $(seq 1 48)
    do
        psql -c "SELECT pg_prewarm('pgbench_accounts_$j');"
    done
    echo "END pg_prewarm: `date`" >> $DIR/vacuum.time

    # begin sar
    sar -A -o $DIR/$i/run.sar.data $INTERVAL > /dev/null 2>&1 &

    # get begin time
    echo "BEGIN: `date`" >> $DIR/vacuum.time

    # execure vacuumdb
    vacuumdb --verbose --freeze --analyze --jobs $i

    # get end time
    echo "END: `date`" >> $DIR/vacuum.time

    # stop sar
    kill -15 `ps | grep "sar" | cut -c 1-5` > /dev/null 2>&1

    # decode sar
    sar -A -f $DIR/$i/run.sar.data > $DIR/$i/run.sar.log

    # copy logs
   cp -R /data1/data/pg_log $DIR/$i/
done
```


© 2016 PostgreSQL Enterprise Consortium

4.4. 検証結果

4.4.1. ジョブ数と処理時間の関係

vacuumdb コマンドを開始してから応答が戻ってくるまでの時間をプロットしたグラフを図 4.1 に示します。横軸はジョブ数、縦軸は経過時間です。実線で結ばれているところは、3 回の測定うちの中央値を表しています。性能がすぐに頭打ちになってしまっていることが分かります。

グラフを表に直したものを表 4.2 に示します。表 4.2 には各ジョブ数において、ジョブ数=1 に比べて何%高速化されているかについてもあわせて載せています。この表からわかるとおり、最も良好な性能が出たのはジョブ数=40 で

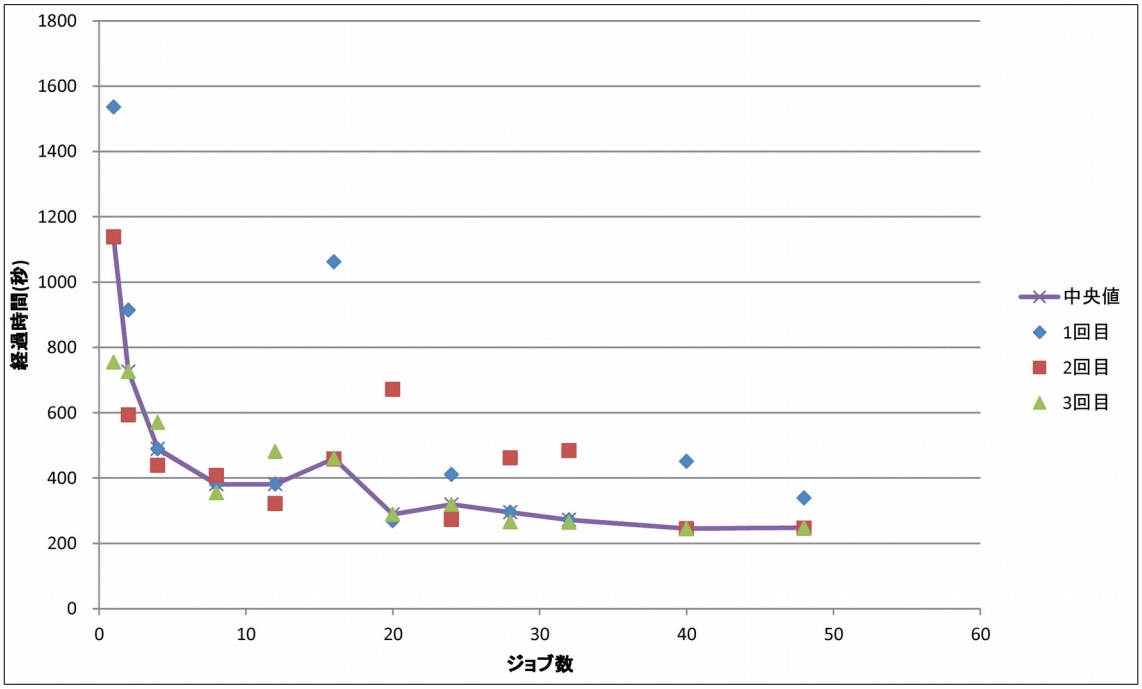

図 4.1: 処理時間とジョブ数の関係

表 4.2: 処理時間とジョブ数=1 に対する性能比

ジョブ数	処理時間（秒）	ジョブ数=1 に対する性能比
1	1138	100%
2	726	157%
4	489	233%
8	381	299%
12	381	299%
16	460	247%
20	289	394%
24	319	357%
28	295	386%
32	272	418%
40	245	464%
48	248	459%

4.4.2. CPU・ストレージの負荷状況

© 2016 PostgreSQL Enterprise Consortium

性能ピークであった 40 接続時の、CPU の負荷状況を図 4.2 と図 4.3 に、ストレージの負荷状況(util)を図 4.4 に示します。CPU は全体的に見た場合ほとんど idle 状態であり、コア別に見てもすべての CPU が 60%以上 idle 状態であることが分かります。ストレージの util についても、WAL と DB ともにある程度の使用率が確認できますが、ストレージ性能がボトルネックとは言えない状況です。

4.4.3. 各テーブルの VACUUM のタイミングと処理時間

ジョブ数=1、ジョブ数=40 のときの、テーブル単位で見たときの最短、最長、平均処理時間を表 4.3 に示します。また、横軸に時刻、縦軸に VACUUM の対象となるテーブルをとり、VACUUM のタイミングをプロットしたグラフを図 4.5、図 4.6 に示します。ジョブ数=40 のとき、同時に複数のテーブルに対して VACUUM が実行されているが、1 つのテーブルあたりの VACUUM にかかる時間がジョブ数=1 に比べて伸びていることが分かります。

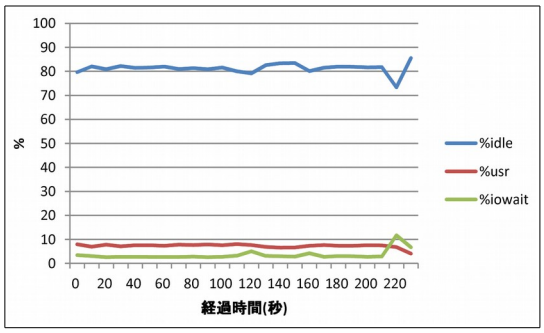

図 4.2: CPU の負荷状況(全体)

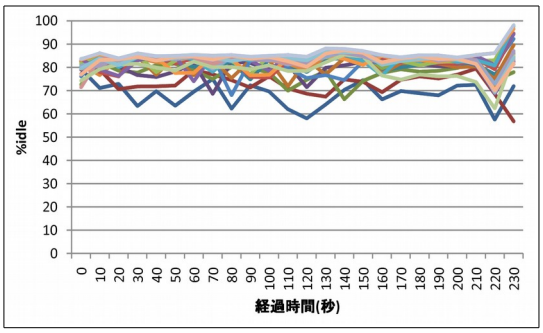

図 4.3: コア別の idle の状況

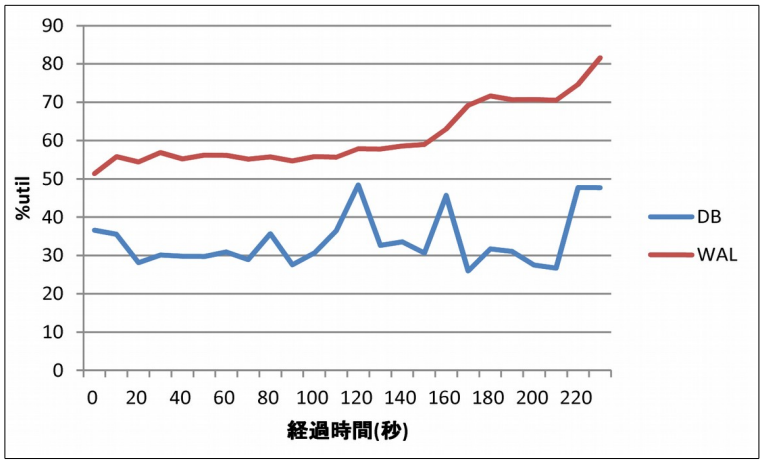

図 4.4: ストレージの負荷状況

表 4.3: テーブル単位(pgbench_account_n)でみる VACUUM の処理時間

ジョブ数	処理時間(秒)		
	最短	最長	平均
1	7.15	174.44	23.47
40	33.00	231.59	166.83

© 2016 PostgreSQL Enterprise Consortium

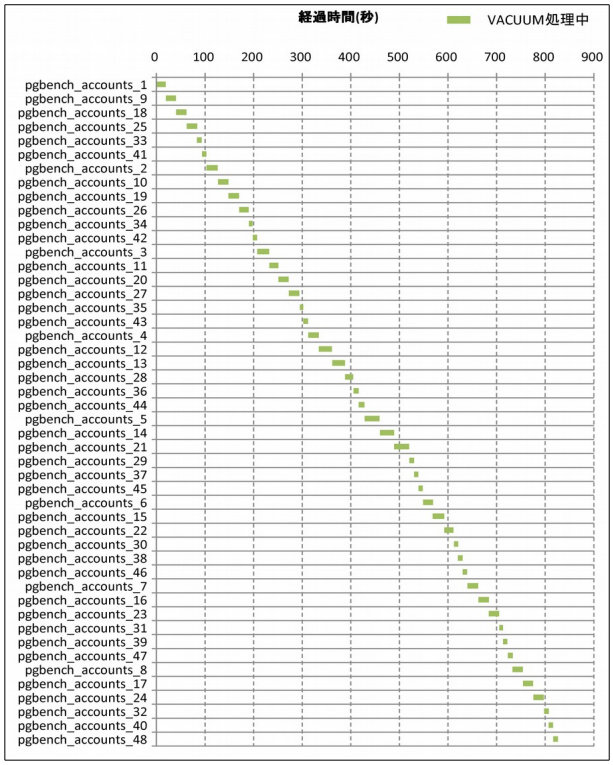

図 4.5: ジョブ数=1 のときの VACUUM の状況

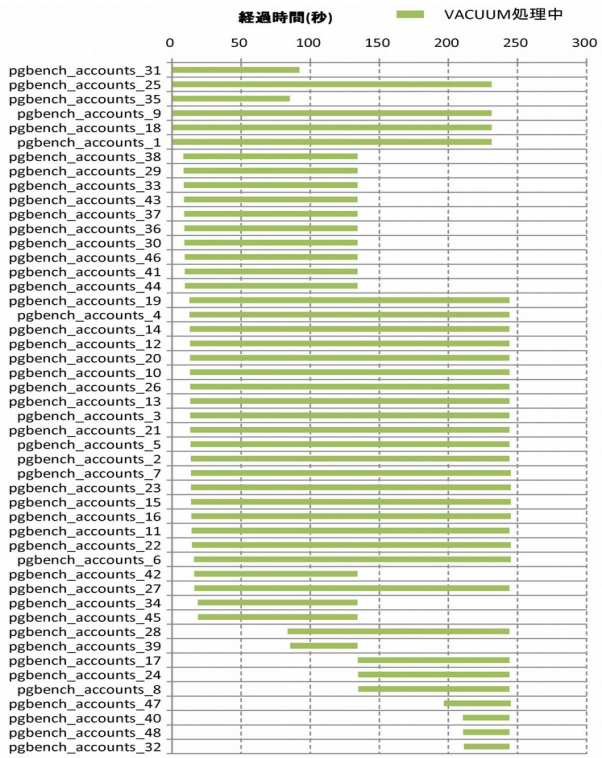

図 4.6: ジョブ数=40 のときの VACUUM の状況

© 2016 PostgreSQL Enterprise Consortium

4.4.4. completion_target を変更

postgresql.conf の checkpoint_completion_target を規定値の 0.5 から 0.9 にした場合の処理時間は以下の表のようになりました。completion_target が大きい場合に処理が遅くなっていることが確認できます。

表 4.4: completion_target を変えた場合の処理時間

ジョブ数	処理時間(秒)	
	0.5	0.9
1	958	1127
40	245	500

4.5. 考察

性能ピーク時の 40 接続の時に、1 テーブルあたりの VACUUM の時間が 1 接続の場合に比べて伸びており、接続数に比例した性能が出ておりません。表 4.2 より、接続数が 2 のときでさえ処理時間が約 1.5 倍しか向上しておらず、各ワーカプロセスの処理時間の25%が並列で動作できていないことが分かります。この 25%に該当する処理として、VACUUM の際に更新処理が走るため、WAL ライタがボトルネックになっている可能性が考えられます。ところで、4.4.4 節より、completion_target が大きい場合に処理が遅くなることが分かっています。これは WAL を DB に反映するバックグラウンドライタの処理が間に合わず、WAL の単位時間当たりの書き出しの量を減らすことで、completion_target を満たすように動いていると考えられます。さらに、4.4.2 節より、ストレージの util がビジー状態となっていないことから、バックグラウンドライタの処理性能の限界はストレージの性能によるものではないと推測できます。

したがって、Parallel Vacuum がジョブ数に比例した性能を出ていない原因として、バックグラウンドライタと WAL ライタが原因であると推測されます。VACUUM のワーカプロセスのリクエストをさばききれていないか、もしくはバックグラウンドライタと WAL ライタでロック獲得のために待ちが多く生じている可能性が考えられます。

なお、本章で実施した検証ではハイエンドのサーバとストレージで構成された環境を利用し、最大 4.6 倍の性能向上が確認できましたが、表 4.5 に示すようなシンプルな構成のサーバでは、なかなか並列性能が出ていないことも確認されています。表 4.5 に示す環境において、同様のシナリオを実施した場合、性能ピーク値がジョブ数=8 のときにわずか 1.2 倍となりました。

この結果から、I/O の負荷分散などの意識をしていない構成においてパラレル VACUUM を実行しても性能改善は見られない、といえます。適切なストレージ構成を組んだ場合に、パラレル VACUUM による性能向上が期待できる、といった点にご留意頂きたいと思います。

表 4.5: シンプルな構成

機器	項目	仕様
PostgreSQL 用サーバ	CPU	インテル Xeon プロセッサ E5645@2.40GHz(6 コア) （ハイパースレッディングはオフ）
	搭載メモリ	24GB
	DB 格納用ストレージ	HDD 256GB (RAID5、RAID1) DB と WAL の領域を分けず
	OS	Red Hat Enterprise Linux 5.6
	DBMS	PostgreSQL 9.5 beta2

© 2016 PostgreSQL Enterprise Consortium

5. BRIN Index 検証

PostgreSQL 9.5 で BRIN (Block Range INdex) Index と呼ばれるインデックス方式が登場しました。以下では btree 比較検証, パーティショニング比較検証の 2 つの検証から BRIN の性能について明らかにしていきます。

5.1. btree 比較検証

5.1.1. 検証概要

本検証ではサンプルデータの入ったテーブルに以下の各インデックスを作成し(もしくは作成せずに)、

- BRIN (pages_per_range = {1, 128, 12800} の 3 種類)
- btree
- インデックスなし

以下の項目について比較検証をします。

- インデックス作成時間
- インデックスサイズ
- データ参照時間

さらに、データ参照時間は以下の条件で比較します。

- ランダムデータを 1 件検索
- ランダムデータを範囲検索
- 単調増加データを 1 件検索
- 単調増加データを範囲検索

5.1.2. BRIN とは

BRIN (Block Range INdex) は 9.5 から新しく登場したインデックス方式で、PostgreSQL ドキュメントによると格納行の物理位置と相関関係を持ったデータを持つ大規模なテーブル向けとされています。

オプションで決定する pages_per_range (デフォルト 128) 個のヒーブブロックをブロックレンジとし、インデックスにはブロックレンジの最小値と最大値が保存されます。構造的に、デフォルトのインデックス方式である btree に比べ、インデックスサイズは小さくなります。

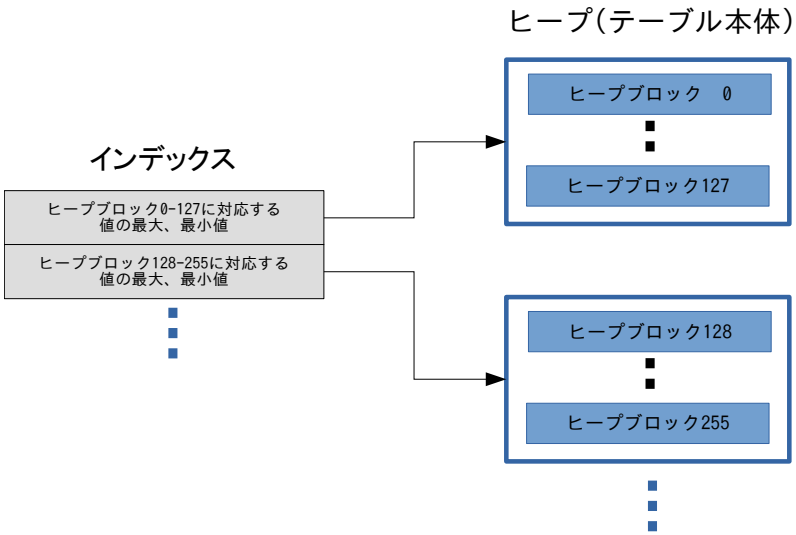

図 5.1: BRIN 構造 (pages_per_range = 128)

5.1.3. 検証構成

5.1.3.1. ハードウェア構成

© 2016 PostgreSQL Enterprise Consortium

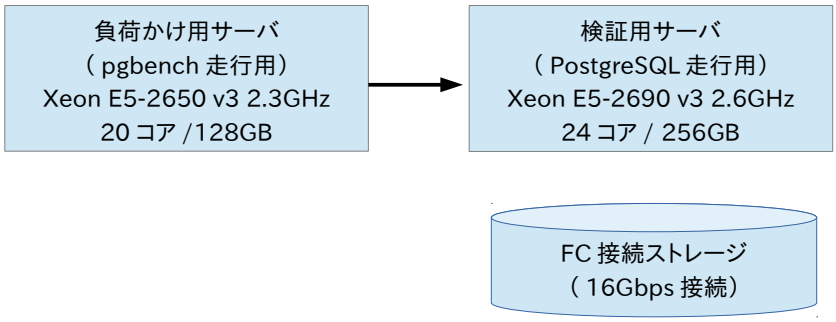

図 5.2: 検証ハードウェア構成

5.1.3.2. 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。

OS	Red Hat Enterprise Linux 7.2
PostgreSQL	9.5beta2

表 5.1: 検証用サーバ

OS	Red Hat Enterprise Linux 7.2
psql	9.5beta2

表 5.2: 負荷かけ用サーバ

5.1.4. 検証方法

5.1.4.1. 検証用サンプルデータ

BRIN はその構造上、格納行の物理位置によって性能が変化することが見込まれるので、検証用のサンプルデータは

格納行の物理位置に対して、

- 値が単調増加する（ソート済み）データ (t_mono テーブル)
- 値がランダムなデータ (t_random テーブル)

の２種類を使用しました。

また、データ件数は１億２千万件 (約 7.6GB)としました。

5.1.4.2. 検証環境

以下の手順で、検証環境を作成しました。

initdb でデータディレクトリを作成し、postgresql.conf を編集します。

```
$ initdb -D {directory} --no-locale -E UTF8

$ vi {directory}/postgresql.conf
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
wal_level = archive
checkpoint_timeout = 30min
```


© 2016 PostgreSQL Enterprise Consortium

```
max_wal_size = 1GB
logging_collector = on
logline_prefix = '%t [%p-%l] '
```

PostgreSQL を起動してベンチマーク用のデータベース brin を作成します。

```
$ pg_ctl -D [directory] -w start
$ createdb -p [port] brin
```

検証データの入ったテーブルを作成 します。

```
=# ¥set items 120000000
=# CREATE TABLE t_mono AS SELECT id, md5(id::text) AS name FROM generate_series(1, :items)
id;
=# CREATE TABLE t_random AS SELECT id, md5(id::text) AS name FROM generate_series(1,
:items) id ORDER BY random();
```

pg_prewarm を実行します。

```
=# select pg_prewarm('t_mono');
=# select pg_prewarm('t_random');
```

5.1.4.3. 検証手順

検証スクリプトの実行をします。

スクリプトは比較する各インデックスごとに以下を実行します。

- インデックス作成 (インデックスなしは除く)

```
-- BRIN (pages_per_range = 1) の場合
CREATE INDEX brinidx_t_mono_1 ON t_mono USING BRIN (id) WITH (pages_per_range=1); -- 単調増
加データ用
CREATE INDEX brinidx_t_random_1 ON t_random USING BRIN (id) WITH (pages_per_range=1); -- ラン
ダムデータ用
```

- 参照クエリ実行

```
EXPLAIN ANALYZE SELECT name FROM t_mono WHERE id = 60000000; -- 単調増加データの 1 件検索
EXPLAIN ANALYZE SELECT name FROM t_mono WHERE id BETWEEN 40000000 AND 80000000; --
単調増加データの範囲検索
EXPLAIN ANALYZE SELECT name FROM t_random WHERE id = 60000000;; -- ランダムデータの 1 件検
索
EXPLAIN ANALYZE SELECT name FROM t_random WHERE id BETWEEN 40000000 AND 80000000;
-- ランダムデータの範囲検索
```

- インデックス削除 (インデックスなしは除く)

```
-- BRIN (pages_per_range = 1) の場合
DROP INDEX brinidx_t_mono_1; -- 単調増加データ用インデックス
DROP INDEX brinidx_t_random_1; -- ランダムデータ用インデックス
```


© 2016 PostgreSQL Enterprise Consortium

5.1.5. 検証結果

各検証結果は以下の通りとなりました。

5.1.5.1. インデックス作成時間

BRIN では圧倒的に短時間でインデックスが作成できました(B-Tree に比べて BRIN では 9%から 15%の時間)。また、データ値と物理的なデータの格納位置の相関関係が単調増加になっているケースとランダムになっているケースの比較では、BRIN では単調増加ケースに対してランダムケースで 0%から 3%の増加に過ぎませんが、B-Tree では 62%の増加になっています。

また、格納行の物理位置はむしろ btree に影響しており、ランダムデータに対するインデックス作成時間はより長くなっていました。逆に BRIN には格納行の物理位置が影響していませんでした。また、BRIN は pages_per_range が大きいほどわずかに作成時間が短くなっていました。

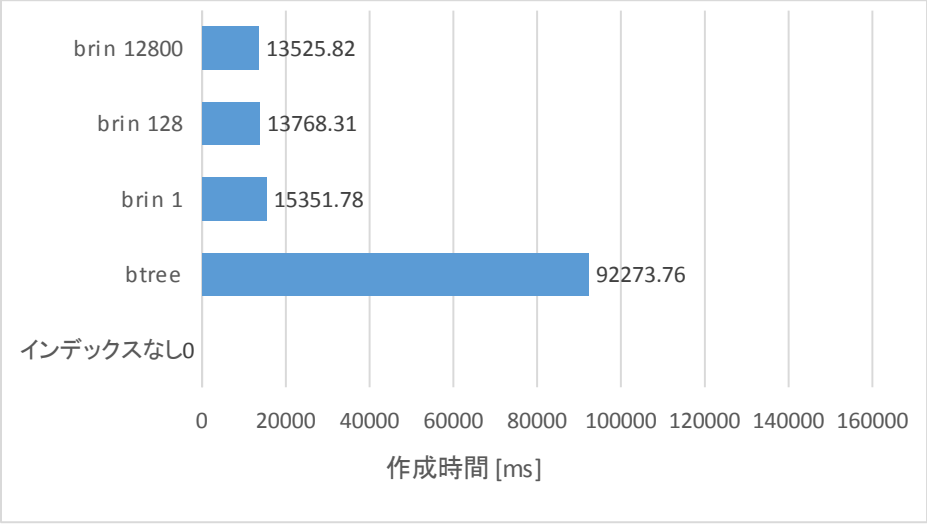

図 5.3: インデックス作成時間 (単調増加データ)

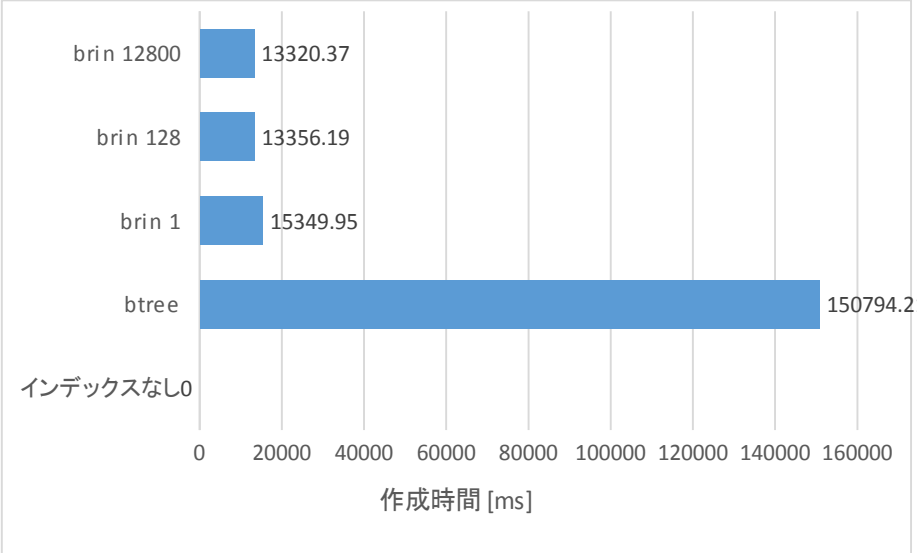

図 5.4: インデックス作成時間 (ランダムデータ)

© 2016 PostgreSQL Enterprise Consortium

5.1.5.2. インデックスサイズ

インデックスサイズはどのインデックスにおいても格納行の物理位置に依存しませんでした。また、BRIN は pages_per_range が小さいほどサイズが大きくなっていますが、一番大きい pages_per_range = 1 の BRIN でも btree のサイズに比べ圧倒的に小さいものでした。

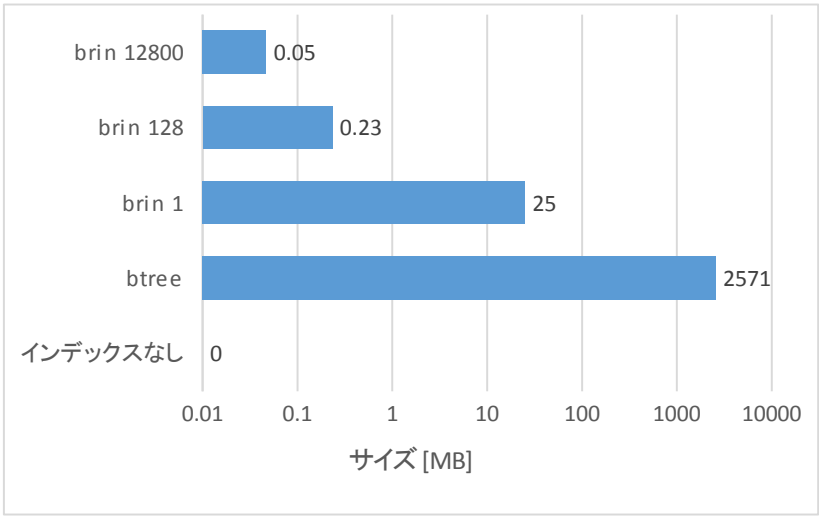

図 5.5: インデックスサイズ(単調増加データ) ※対数目盛

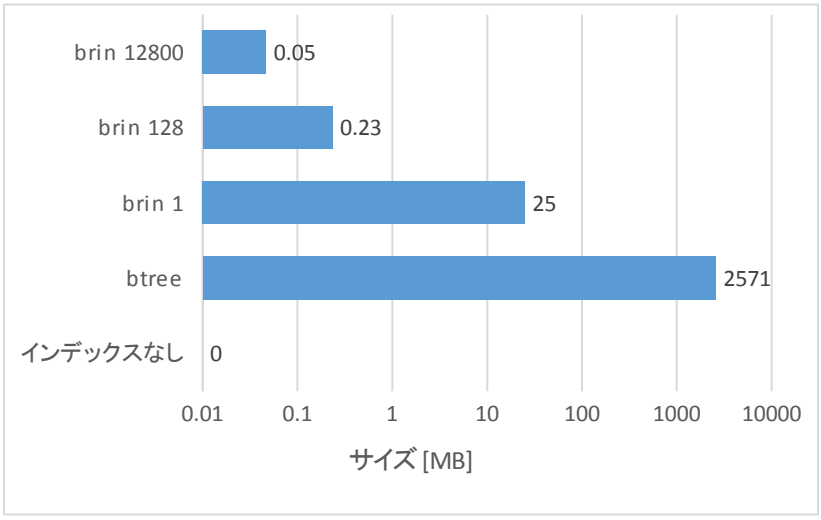

図 5.6: インデックスサイズ (ランダムデータ) ※対数目盛

© 2016 PostgreSQL Enterprise Consortium

5.1.5.3. 参照性能（1 件検索）

参照性能（1 件検索）は格納行の物理位置によって BRIN の性能が大きく変わります。ランダムデータの場合、インデックスなしの場合とどの BRIN もほとんど同じ性能になります。しかし、単調増加データの場合、btree に及ばないものの、インデックスなしの場合とは大きな差をつけています。

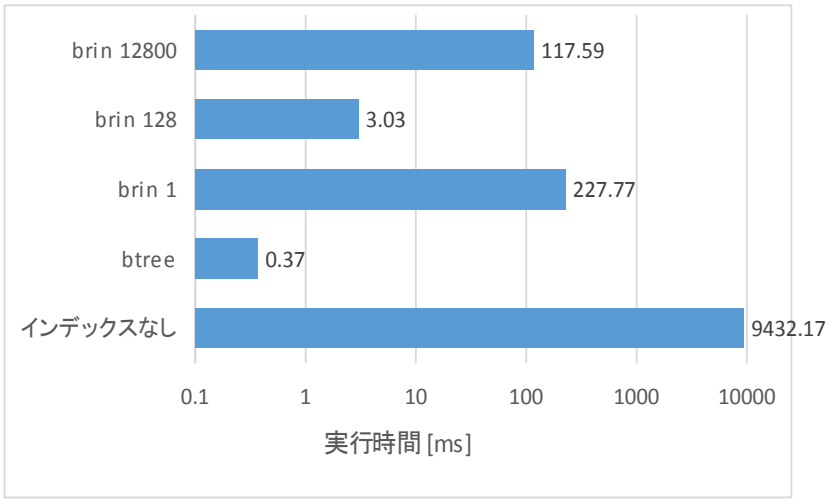

図 5.7: 参照性能 (単調増加データ、1 件検索) ※対数目盛

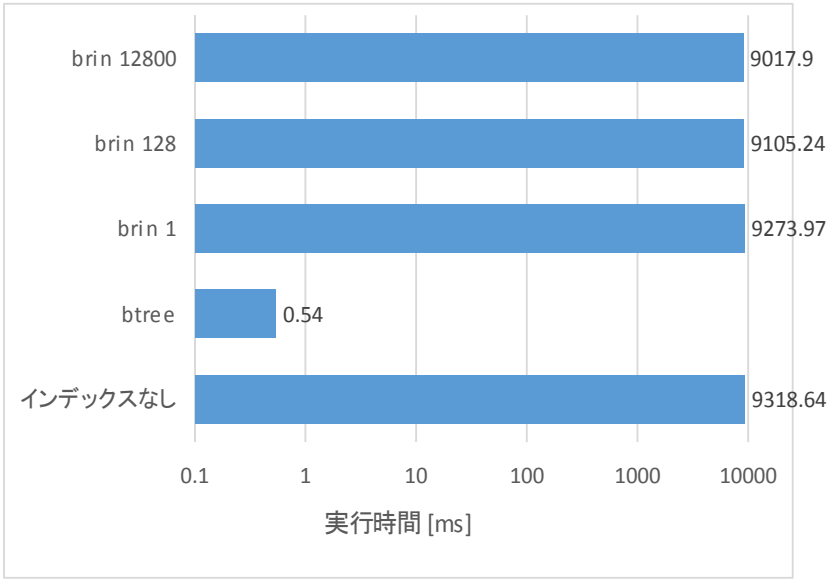

図 5.8: 参照性能 (ランダムデータ、1 件検索) ※対数目盛

© 2016 PostgreSQL Enterprise Consortium

5.1.5.4. 参照性能（範囲検索）

参照性能（範囲検索）も格納行の物理位置によって BRIN の性能が大きく変わります。ランダムデータの場合、インデックスなしの場合とどの BRIN もほとんど同じ性能になります。しかし、単調増加データの場合、btree に匹敵する性能を見せました。また、ランダムデータの場合は btree に影響が出ており、インデックスなしのシーケンシャルスキャンよりも遅くなっています。

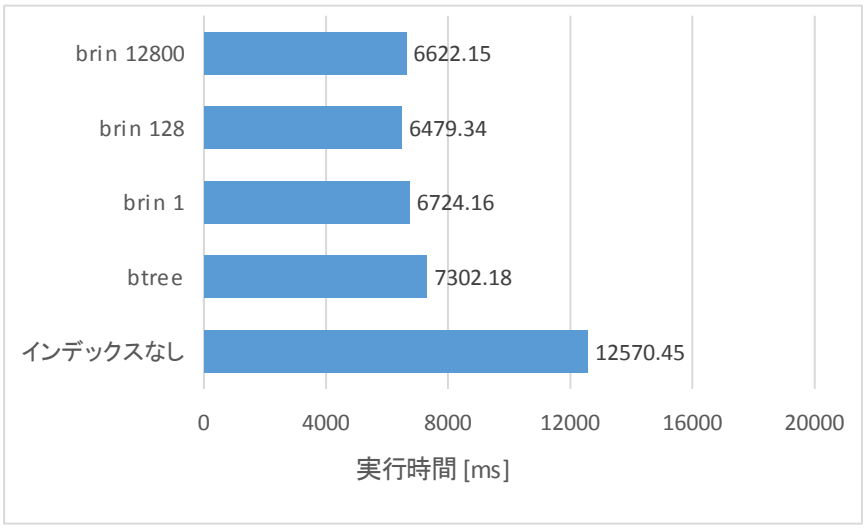

図 5.9: 参照性能 (単調増加データ、範囲検索)

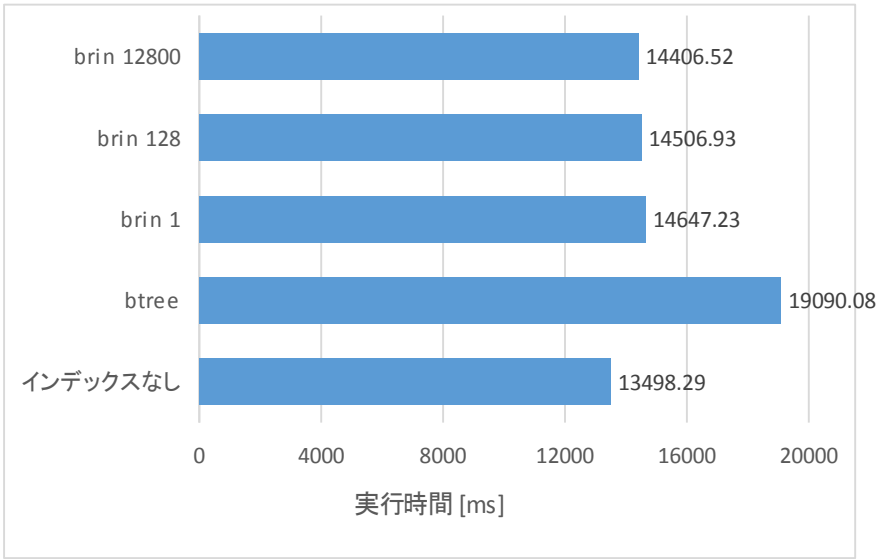

図 5.10: 参照性能 (ランダムデータ、範囲検索)

© 2016 PostgreSQL Enterprise Consortium

5.2. パーティショニング比較検証

5.2.1. 検証概要

9.5 以前より PostgreSQL にはテーブルのパーティショニング機能があります。本機能はテーブルの継承を利用し、データの制約条件を付けた子テーブルを分割することで親テーブルから透過的に、トリガを用いてデータの登録、CHECK 制約を用いてデータの参照を行う機能です。

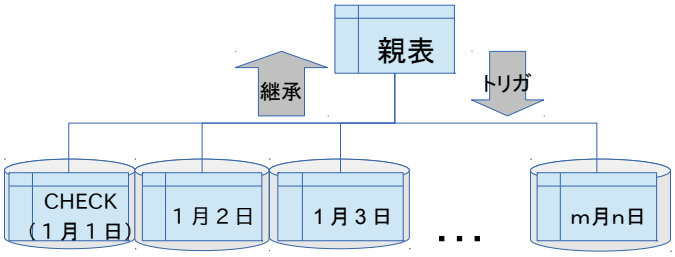

図 5.11: PostgreSQL のパーティショニング

パーティショニング機能はデータ参照時には目的の値が明らかに存在しない集合を読み飛ばすといった点で BRIN と似ています。本検証では実際に BRIN とパーティショニングを同一データに対しそれぞれ適用し、性能比較を行います。

具体的には、以下の各テーブルおよびインデックスの状態について、

- パーティションテーブル(btree あり)
- パーティションテーブル(インデックスなし)
- 単表(BRIN あり)
- 単表(btree あり)
- 単表(インデックスなし)

※「単表」は非パーティションテーブルを表します。

以下の項目を比較します。

- データ挿入性能
- データ参照性能

さらに、データ参照時間は以下の条件で比較します。

- ランダムデータを全件検索
- 単調増加データを全件検索
- ランダムデータを1パーティション分検索
- 単調増加データを1パーティション分検索

検証テーブル定義は 2013 年度のパーティショニング検証と同様のものとしており、日々のログデータを保存するようなテーブルを想定しています。また、用いるトリガは C 言語関数を用いました。

5.2.2. 検証構成

5.1.3 と同じ環境を利用しました。

5.2.3. 検証方法

5.2.3.1. 検証環境

データベース作成までは 5.1.4 と同様の手順でデータベースクラスタを作成します。

パーティション検証用データベース作成を作成します。

```
$ createdb partition
```


© 2016 PostgreSQL Enterprise Consortium

パーティショニングを行う親テーブルを作成します。同時に単表（非パーティションテーブル）として定義をコピーしておきます。(btree を含まないテーブル作成手順は省略)

```
create table access_log (
  log_id            bigserial,
  date              char(8),
  time              char(6),
  productid         int,
  place_id          int,
  machine_id        int,
  app_id            int,
  access_time_second int,
  access_count      int,
  err_code          int
);
create index access_log_key on access_log (date, time);
create table raw_access_log (like access_log INCLUDING INDEXES INCLUDING DEFAULTS
INCLUDING CONSTRAINTS);
create table brin_access_log (like access_log INCLUDING DEFAULTS INCLUDING CONSTRAINTS);
create index brin_idx on brin_access_log using brin (date, time);
```

子テーブルを作成します。1 日分のデータを保存するものなので子テーブルは date カラムで区分けしています。今回は 31 個（1 ヶ月分）の子テーブルが存在します。

```
create table access_log_20150101 (
  LIKE access_log INCLUDING INDEXES INCLUDING DEFAULTS INCLUDING CONSTRAINTS,
  CHECK (date = '20150101')
) INHERITS (access_log);
...
create table access_log_20150131 (
  LIKE access_log INCLUDING INDEXES INCLUDING DEFAULTS INCLUDING CONSTRAINTS,
  CHECK (date = '20150131')
) INHERITS (access_log);
```

親テーブルにトリガを設定します。トリガ関数の詳細は省略します。

```
CREATE OR REPLACE FUNCTION func_access_log() RETURNS trigger
   AS 'トリガオブジェクトファイル'
   LANGUAGE C;
CREATE TRIGGER trigger_access_log
  BEFORE INSERT ON access_log
  FOR EACH ROW
  EXECUTE PROCEDURE func_access_log();
```

ログデータを一旦 CSV データで作成します。先の検証と同様に二種類のデータを用意します。今回は date, time 属性において単調増加データ（ソート済みデータ） とランダムデータをそれぞれ用意しました。

5.2.3.2. 検証手順

各条件のテーブルに対してそれぞれ以下の操作を実行しました。

- CSV データ挿入

```
(パーティションテーブルの例)
¥copy access_log from .CSV ファイル (format csv); --
```

- VACUUM 実施 (BRIN のみ)

```
vacuum brin_access_log;
```

BRIN は生成後に大量のデータが挿入されると適切に動作しません。運用中は自動 VACUUM で解消されますが、本検証では明示的に VACUUM を実行してその実行時間も計測しています。

© 2016 PostgreSQL Enterprise Consortium

- SELECT 実行

```
(パーティションテーブルの例)
-- 全パーティション(1 ヶ月分)検索
explain analyze
select product_name, err_code, avg(access_time_second)
from product_master, brin_access_log
where product_master.productid = brin_access_log.productid
and date between '20140101' and '20140131'
group by product_master.product_name, brin_access_log.err_code;
-- 1 パーティション(1 日分)検索
explain analyze
select product_name, err_code, avg(access_time_second)
from product_master, brin_access_log
where product_master.productid = brin_access_log.productid
and date = '20140131'
group by product_master.product_name, brin_access_log.err_code;
```


© 2016 PostgreSQL Enterprise Consortium

5.2.4. 検証結果

5.2.4.1. データ挿入性能

データ挿入時間はパーティションテーブルは圧倒的に遅いです。btree があると余計に遅くなります。単表の場合だけで見ると、btree が一番遅いです。BRIN は大量データ挿入後に VACUUM を必要としているが、それでも btree と比較すれば早くなります。格納行の物理位置は大きく影響はしていませんでした。

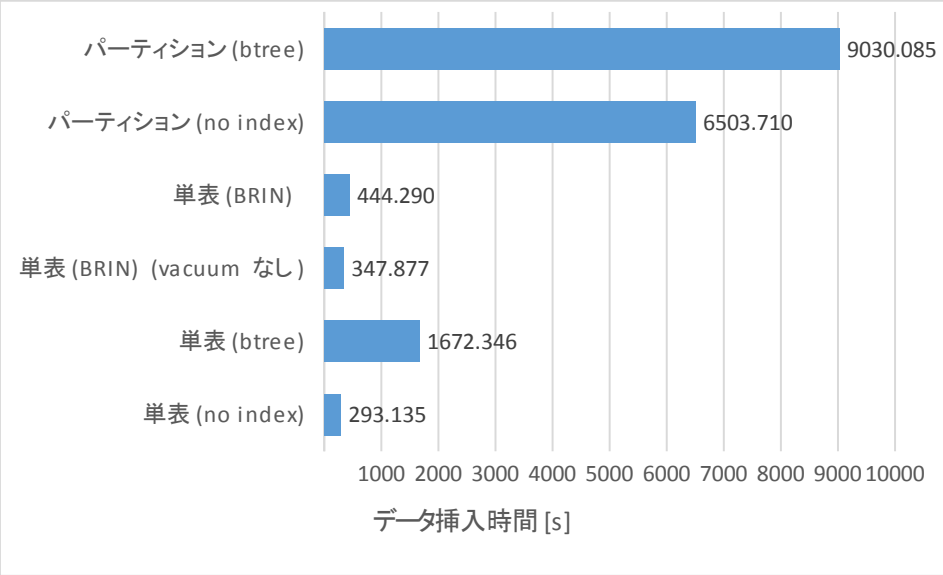

図 5.12: データ挿入性能 (ランダムデータ)

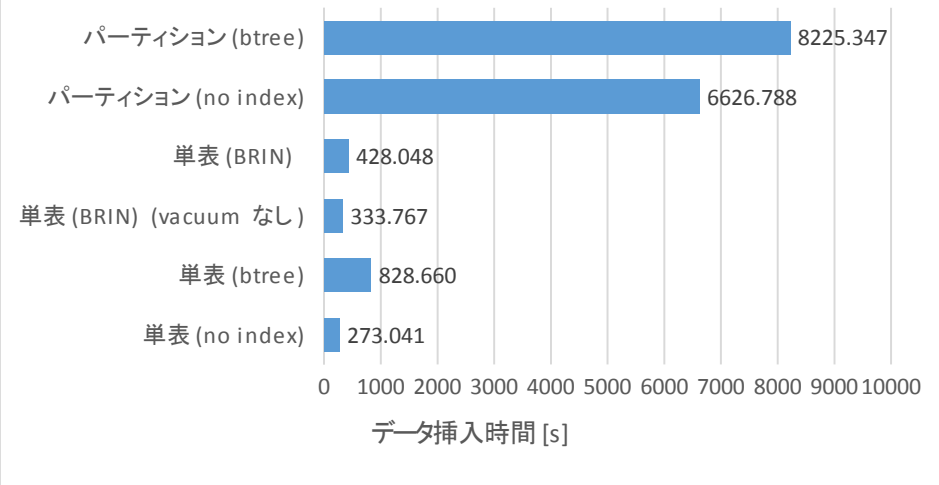

図 5.13: データ挿入性能 (単調増加データ)

© 2016 PostgreSQL Enterprise Consortium

5.2.4.2. データ参照性能（1 パーティション検索）

ランダムデータでは BRIN はシーケンシャルスキャン程度の遅さだが、単調増加データでは単表（btree）、パーティション構成に匹敵する速度を示していました。また、パーティション構成は格納行の物理位置には影響していませんでした。

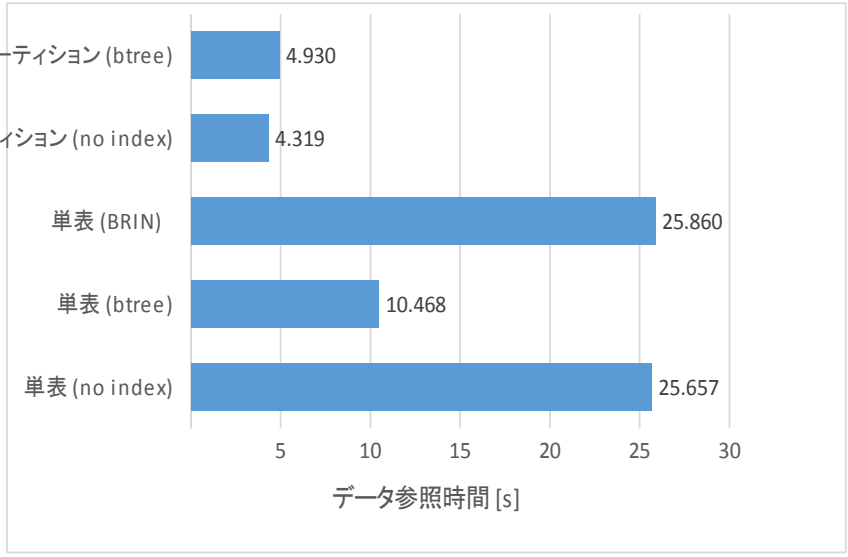

図 5.14: データ参照性能 (ランダムデータ、1 パーティション検索)

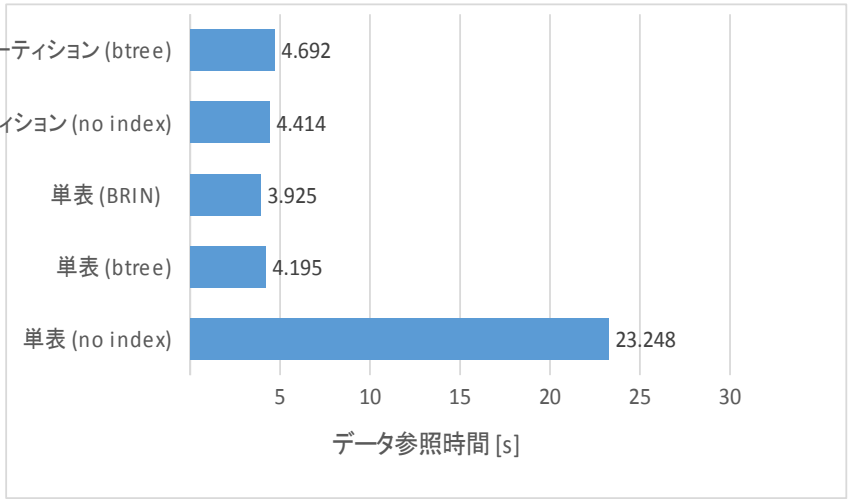

図 5.15: データ参照性能 (単調増加データ、1パーティション検索)

© 2016 PostgreSQL Enterprise Consortium

5.2.4.3. データ参照性能（全データ検索）

全データを対象に検索すると、もはや BRIN は格納行の物理位置に依らず同程度の性能を示しています。この場合、格納行の物理位置に依らないパーティション構成よりも僅かに優れた性能を示しています。

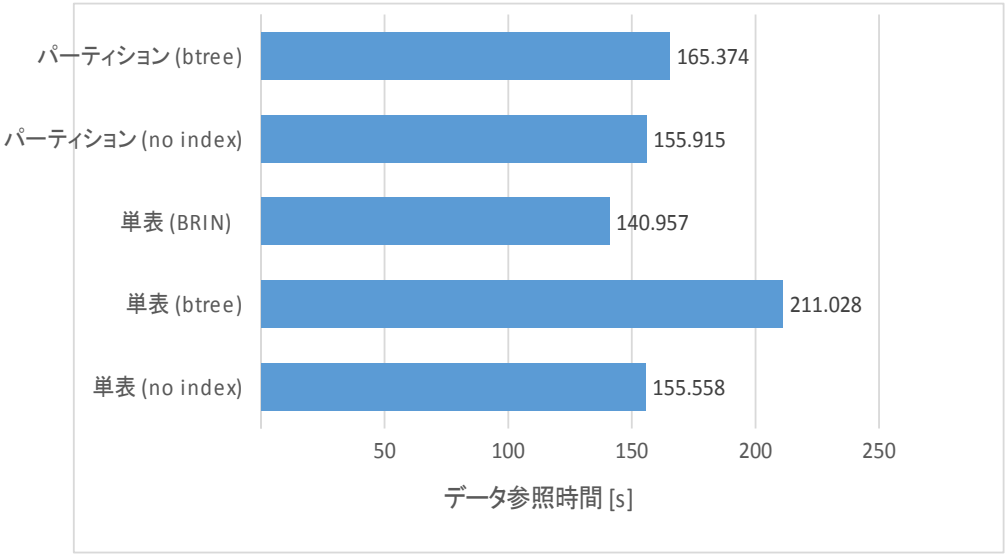

図 5.16: データ参照性能 (ランダムデータ、全件検索)

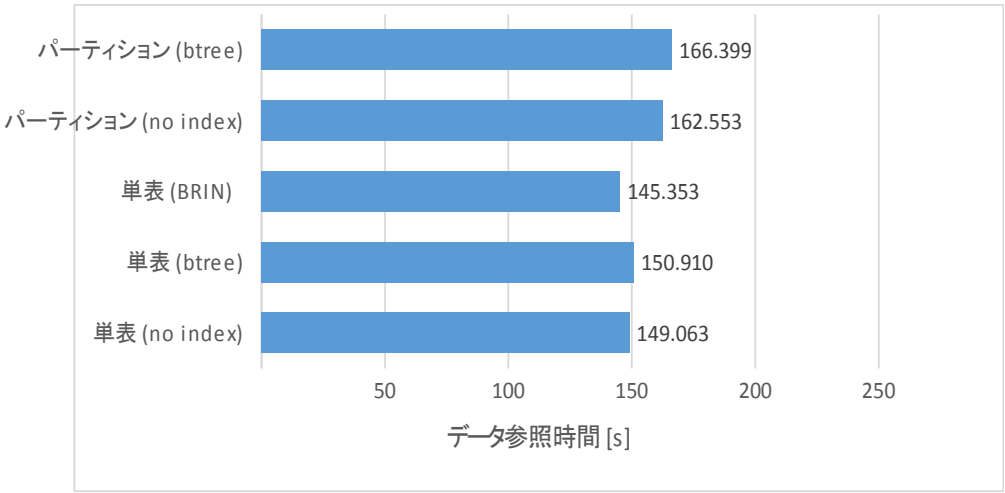

図 5.17: データ参照性能(単調増加データ、全件検索)

© 2016 PostgreSQL Enterprise Consortium

5.3. 考察

本検証では BRIN のいくつかの特性を明らかにしてきました。

まず、インデックスサイズにおいて、ある属性の値を全てのレコード分保存している btree に比べ、全レコードを分割した各範囲で最大値と最小値のみを保存する BRIN は非常に小さく抑えることが可能となっていました。さらに、BRIN はレコードを分割する範囲(pages_per_range)は任意に指定可能であることから間接的にインデックスサイズを調整することが可能でした。

インデックスの作成時間においてもツリー構造を持つ btree と比べ、各範囲で最大値と最小値のみを保存する BRIN は非常に短くなっていました。インデックス作成時間は全レコードが存在してからインデックスを作成する場合とインデックスを作成してから全レコードを挿入しインデックスが利用可能になる場合を 2 検証で確認しましたが、どちらにおいても BRIN の優位性が示されました。また、pages_per_range によってインデックス作成時間が大きく変わることはありませんでした。

参照性能においては格納行の物理位置が BRIN の性能を決定する重要な要素となります。BRIN が有効に働くのは格納行の物理位置がデータと相関関係にある場合でした。今回用いたような単調増加データに対する参照であれば、1 件検索では btree には及ばないもののインデックスが無い状態よりも十分に早く、範囲検索であれば btree を超える性能をも示しました。(本検証では検索範囲は固定としていたので BRIN が有効になる検索範囲については別途検証の必要があります。)対して、格納行の物理位置が値に対してランダムな場合、BRIN はインデックス無しの状態と殆ど変わらない性能でしかありませんでした。

また、大規模テーブルのデータに対する BRIN とパーティショニング構成について比較したことで、各々のメリットが見えてきました。

まず、テーブル構成を準備する段階でパーティショニング機能は直接利用できるインタフェースを提供していないので継承機能とトリガを用いなくてはならず、インデックスの作成のみという BRIN の簡便さも含めてメリットとなり得ます。

データ挿入性能に関して、パーティション構成はトリガ処理が挟まる分、挿入性能が大きく低下します。対して、BRIN はインデックスなしの状態から相対的に大きな性能低下は起きません。

参照性能では格納行の物理位置がデータと相関関係にある場合、パーティショニング機能を使用するよりも BRIN の方が簡便で良い性能をもたらすことも示しました。逆に、パーティショニング構成の参照性能は対象となる格納行の物理位置に依らないというメリットがあるといえます。

© 2016 PostgreSQL Enterprise Consortium

6. OS 比較検証

6.1. 検証概要

Red Hat Enterprise Linux 7（以下、RHEL7）は 2014 年 6 月にリリースされ、systemd の採用、カーネルのバージョンが 2 から 3 になるなど、大幅な変更が加えられました。今後は Red Hat Enterprise Linux 6（以下、RHEL6）に代わり、主流になっていくものと考えられます。本検証では、同一バージョンの PostgreSQL を利用する際に、RHEL6 と RHEL7 の違いによって性能に差が見られるかという点に着目して検証を行いました。

6.2. 検証構成

6.2.1. 検証構成概要

以下の図に OS 比較検証での検証構成を示します。

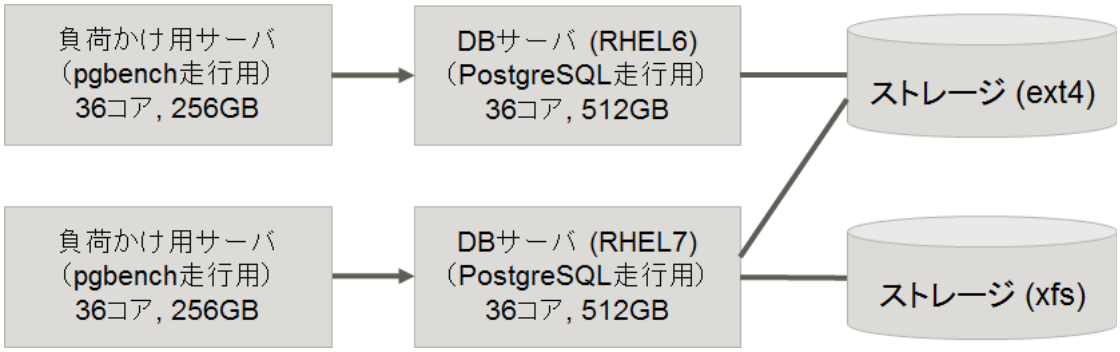

図 6.1: 検証構成概要

本検証では検証用マシン内のシステムボード 1 枚を 1 つのサーバとして使用し、OS の異なる DB サーバを 1 台ずつ、負荷かけ用のサーバを 2 台、全部で 4 台のサーバを用意しました。また、ストレージ上には 2 つのパーティションを用意し、それぞれ ext4、xfs でフォーマットを行いました。

サーバとストレージの接続は FC 接続とし、マウントは各ファイルシステムの測定時のみ行っています。

6.2.2. ハードウェア構成

検証に使用した機器の構成は以下の通りです。詳細については 2015 年度活動報告書 Appendix 検証環境 の、検証環境 2 に別途記載しています。

表 6.1: 検証ハードウェアの構成

機器	項目	仕様
DB サーバ [RHEL6, RHEL7 共通]	CPU 搭載メモリ 内蔵ストレージ	Intel® Xeon® Processor E7-8890 v3 (18Cores, 45M Cache, 2.50 GHz) x 2 512GB HDD: 300GB x 2 (RAID1+0)
負荷かけクライアント	CPU 内蔵メモリ 内蔵ストレージ	Intel® Xeon® Processor E7-8890 v3 (18Cores, 45M Cache, 2.50 GHz) x 2 256GB HDD: 300GB x 2 (RAID1+0)
DB 格納用ストレージ	容量 接続	SSD: 1.6TB x 24 (RAID1+0) 実効容量 9.6TB のパーティションを 2 個作成し、ext4 / xfs でフォーマット FC 接続 (16Gbps)

© 2016 PostgreSQL Enterprise Consortium

6.2.3. ソフトウェア構成

検証に使用したソフトウェアの構成は以下の通りです。いずれのソフトウェアも、検証開始時点(2016 年 1 月)での最新バージョンを使用しています。

表 6.2: 検証ソフトウェアの構成

機器	OS	PostgreSQL
DB サーバ[RHEL6]	Red Hat Enterprise Linux 6.7	PostgreSQL 9.5.0
DB サーバ[RHEL7]	Red Hat Enterprise Linux 7.2	PostgreSQL 9.5.0
負荷かけクライアント	Red Hat Enterprise Linux 7.2	PostgreSQL 9.5.0

6.2.4. システム設定

本検証での OS の設定は以下の通りです。ファイルシステム、I/O スケジューラ以外の設定は全てデフォルトのものを使用しています。

表 6.3: OS の設定

項目	RHEL6.7	RHEL7.2
カーネルバージョン	2.6.32	3.10.0
ファイルシステム	ext4（デフォルト）	xfs（デフォルト） ext4
I/O スケジューラ	deadline cfq（デフォルト） noop	deadline（デフォルト） cfq noop
システム起動方法	init / upstart	systemd
gcc のバージョン	4.4.x	4.8.x

6.3. 検証方法

6.3.1. 検証用サンプルデータ

本検証では pgbench を利用してベンチマーク用のテーブルを作成しました。OS の差異、ファイルシステムの差異、I/O スケジューラの差異をそれぞれ測定するため、スケールファクタ（以下、SF）を変化させ、以下の 2 つのサイズのデータベースを作成しました。

・SF = 6,000（約 75GB）

全てのデータが DB サーバのメモリに乗り切るサイズです。マシンは同一のものを使用しているため、負荷かけ開始から一定時間が経過しデータがメモリに乗り切った後は OS の性能差として測定することが可能です。

・SF = 60,000（約 750GB）

DB サーバのメモリ量より大きく、データがメモリ上に乗り切らないサイズです。一定時間が経過するとスループットは安定しますが、常に I/O が発生し続けるためファイルシステムの性能差として測定することが可能です。

```
port = 5435
listen_addresses = '*'
shared_buffers = 128GB // DB サーバのメモリの 1/4
work_mem = 1GB
wal_level = archive
checkpoint_timeout = 30min
max_wal_size = 1GB
logging_collector = on
log_line_prefix = '%t [%p-%l]'
```


© 2016 PostgreSQL Enterprise Consortium

6.3.2. データベース設定

データディレクトリを作成し、postgresql.conf を編集します。

6.3.3. 検証手順

本検証の流れは以下の通りです。

測定スクリプト実行の前に、メモリキャッシュのクリアを行います。

```
# echo 1 > /proc/sys/vm/drop_caches
# echo 2 > /proc/sys/vm/drop_caches
# echo 3 > /proc/sys/vm/drop_caches
# su - postgres
$ pg_ctl -w restart
```

次にスクリプトを使用して pgbench を用いてベンチマークの実行を行います。使用したスクリプトは以下の通りです。”custom.sql”には参照系、更新系それぞれの SQL を指定します。また、$SF には測定するデータベースのスケールファクタを指定します。

本検証では、10 秒間の平均 TPS 値を連続的に測定し、その推移を観測しました。クライアント数はコア数の 2 倍となる 72 に固定して測定を行いました。

```
PGBENCH_COMMAND=”pgbench -n -h [host_address] -p 5345 pgbench_SF_$SF -c 72 -j 36 -f
custom.sql -T 10 -s $SF”

for TRY in $(seq 1 60); do
 echo ”TRY: $TRY” >> pgbench_SF_$SF_result.txt
 $PGBENCH_COMMAND >> pgbench_SF_$SF_result.txt
done
```

参照系、更新系で使用した SQL 文はそれぞれ以下の通りです。DB サーバの負荷状態を考慮し、参照系は負荷が高くなるようにカスタムしたクエリを、更新系は pgbench のデフォルトのクエリを使用しています。

・参照系クエリ

```
¥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
```


© 2016 PostgreSQL Enterprise Consortium

・更新系クエリ

```
¥set nbranches :scale
¥set ntellers 10 * :scale
¥set naccounts 100000 * :scale
¥setrandom aid 1 :naccounts
¥setrandom bid 1 :nbranches
¥setrandom tid 1 :ntellers
¥setrandom delta -5000 5000
BEGIN;
UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;
UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;
INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (:tid, :bid, :aid, :delta,
CURRENT_TIMESTAMP);
END;
```

pgbench による測定の間、sar などの OS 情報取得コマンドを利用して CPU 使用率と%iowait の値を取得し、こちらも 10 秒間隔で推移を観測しました。

6.4. 検証結果

6.4.1. 参照系

6.4.1.1. データがメモリに乗り切る場合（SF=6,000）

測定した TPS 値、CPU 使用率、%iowait の推移のグラフを以下に示します。測定は 3 回実施し、中央値を取ったものを時系列でプロットしています。

© 2016 PostgreSQL Enterprise Consortium

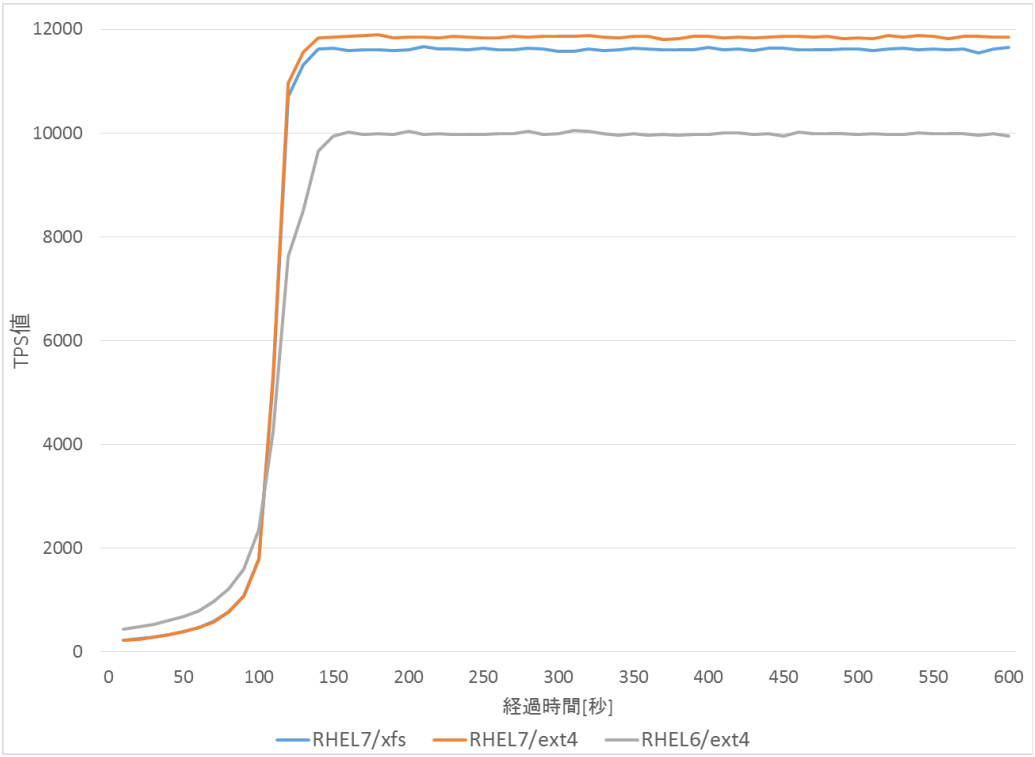

図 6.2: TPS 値推移(参照系, SF=6,000)

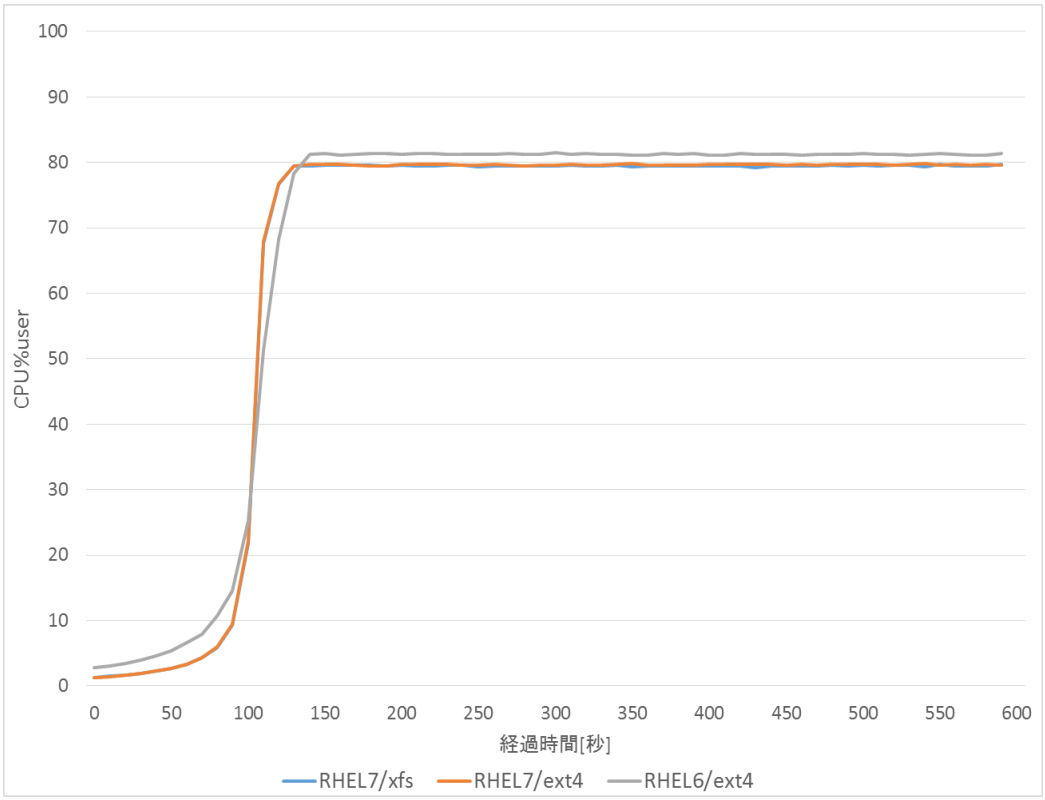

図 6.3: CPU 使用率推移(参照系、SF=6,000)

© 2016 PostgreSQL Enterprise Consortium

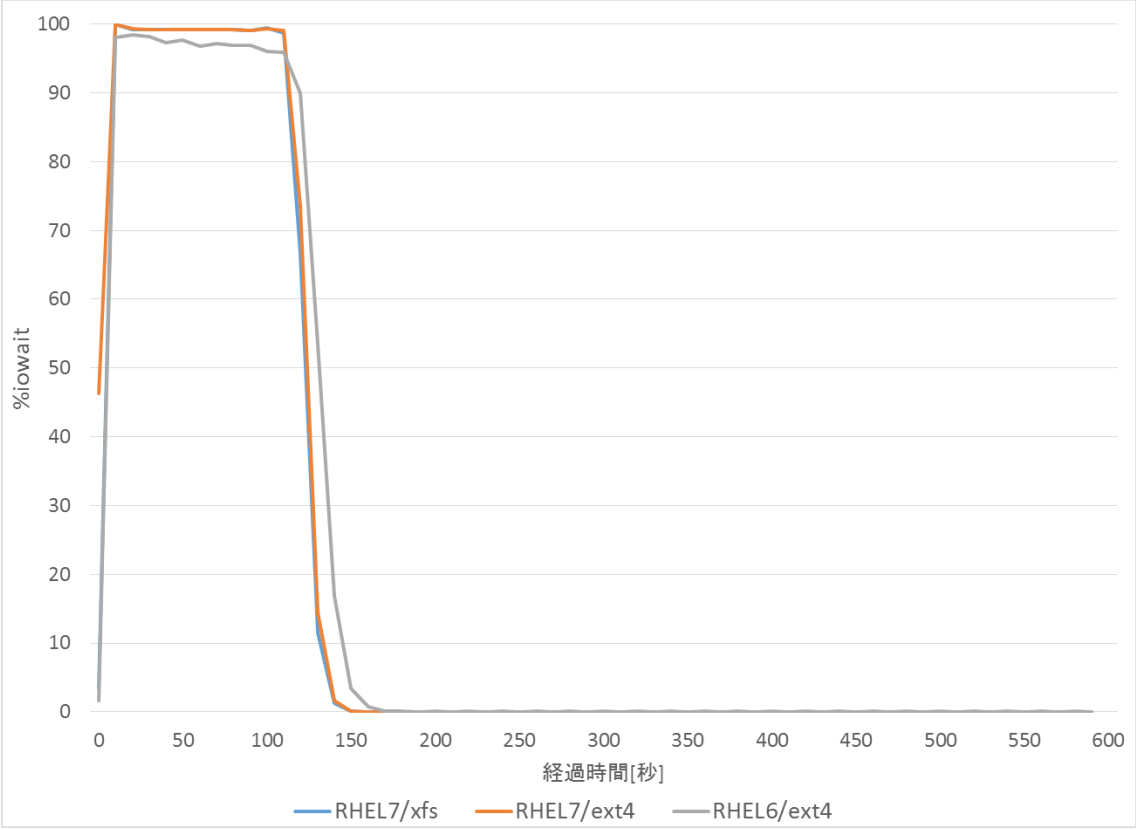

図 6.4: %iowait 推移(参照系、SF=6,000)

TPS に関しては、負荷かけ開始から一定時間が経過すると RHEL7/ext4、RHEL7/xfs、RHEL6/ext4 の順に大きな値となり、性能が良いという結果になりました。また、RHEL7 におけるファイルシステム間の差は比較的小さく、RHEL6 と RHEL7 の間では比較的大きな差となりました。CPU 使用率、I/O の使用率に関しては、どのパターンでも同じような挙動を見ることが出来ました。

© 2016 PostgreSQL Enterprise Consortium

6.4.1.2. データがメモリに乗り切らない場合（SF=60,000）

6.4.1.1.と同様に、TPS、CPU 使用率、%iowait の推移のグラフを以下に順に示します。

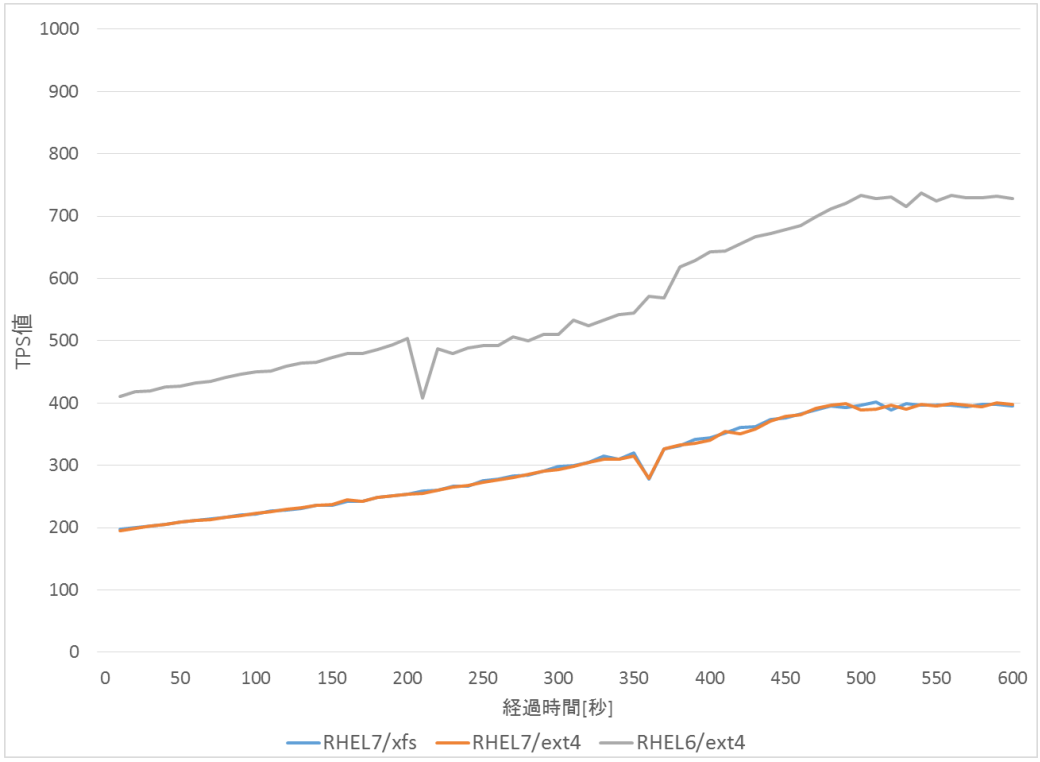

図 6.5: TPS 値推移(参照系, SF=60,000)

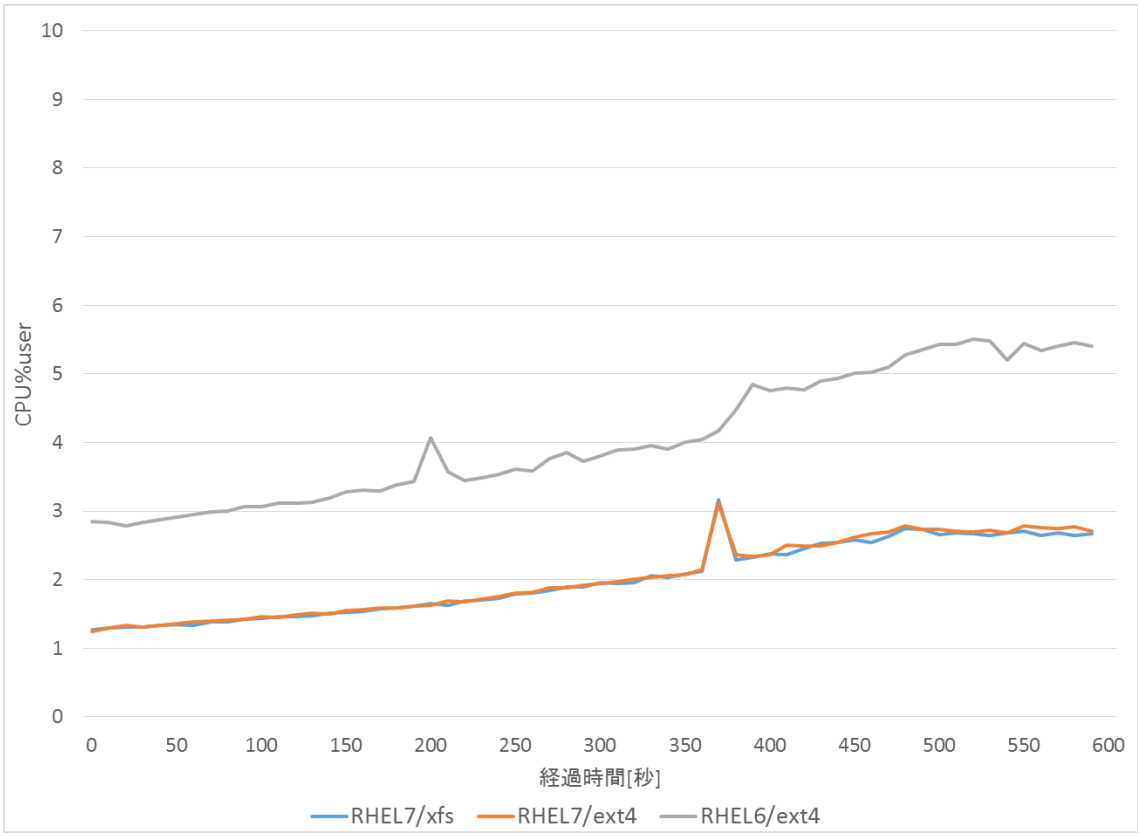

図 6.6: CPU 使用率推移(参照系、SF=60,000)

© 2016 PostgreSQL Enterprise Consortium

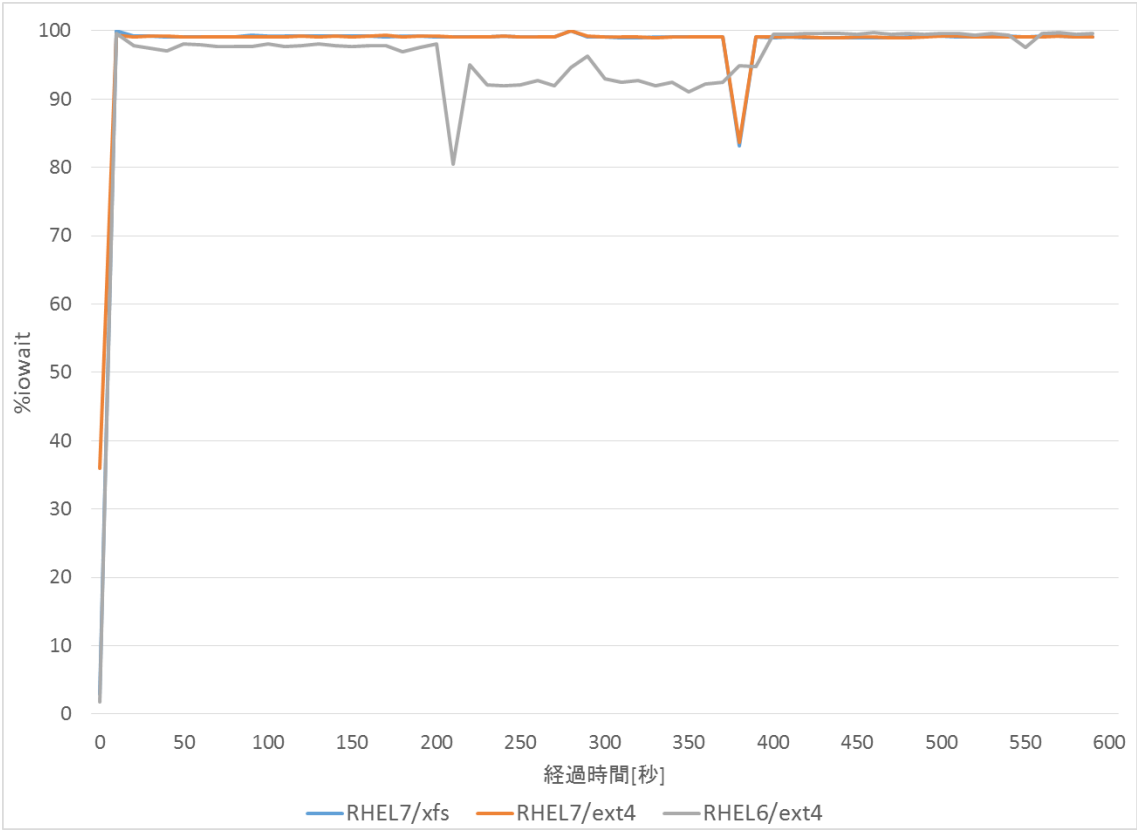

図 6.7: %iowait 推移(参照系、SF=60,000)

TPS に関しては、終始 RHEL6/ext4、RHEL7/xfs、RHEL7/ext4 の順に大きな値となり、SF=6,000 の場合とは異なる結果となりました。このことから、データベースのサイズによって高い性能が得られる OS とファイルシステムの組み合わせが異なると言うことが出来ます。

CPU 使用率に着目すると RHEL6 の方が高い値を取っており、このことが TPS の差につながったのではないかと考えられます。また、I/O の使用率に関してはどちらの OS においても同じような挙動を見ることが出来ており、SF=6,000 の場合の結果から、I/O がボトルネックとならない場合は RHEL7 の方が高い性能が得られると言うことが出来ます。

以上の結果から、参照系では OS の差異が性能に与える影響が大きい傾向が観測できました。

6.4.2. 更新系

6.4.2.1. データがメモリに乗り切る場合（SF=6,000）

参照系同様、測定した TPS 値、CPU 使用率、%iowait の推移のグラフを以下に示します。測定は 3 回実施し、中央値を取ったものを時系列でプロットしています。

© 2016 PostgreSQL Enterprise Consortium

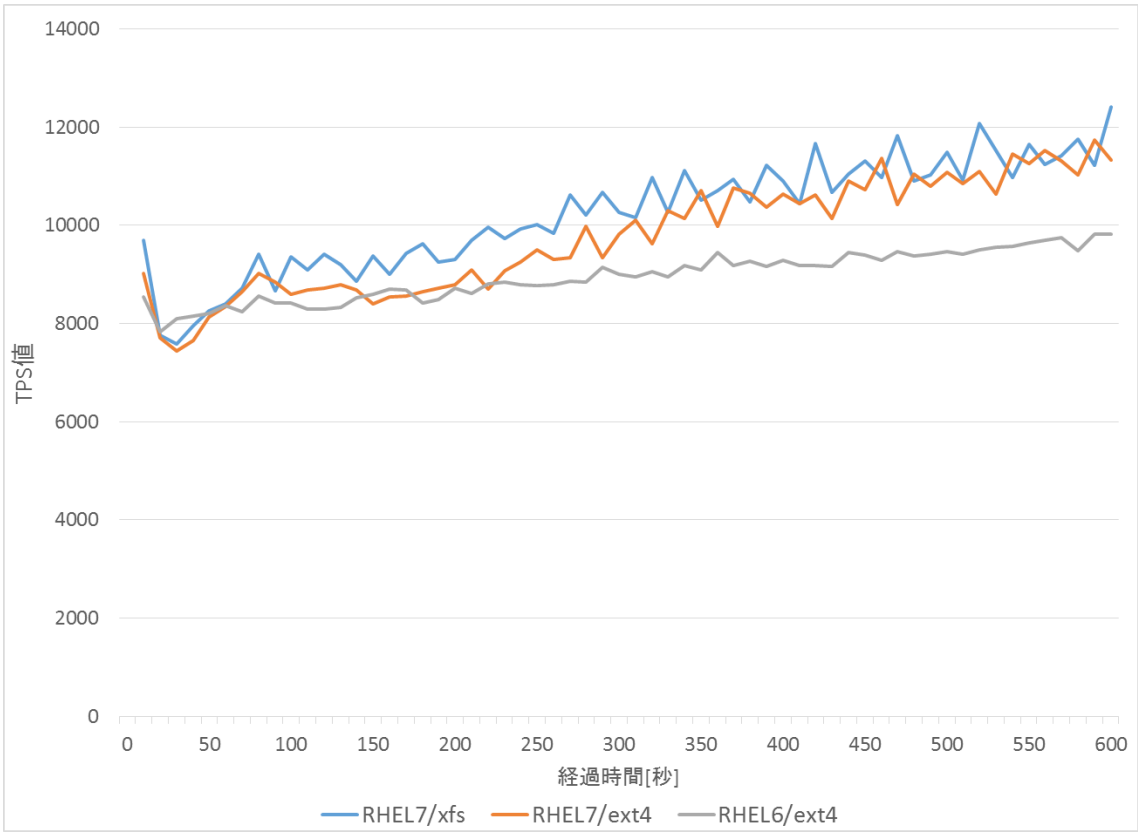

図 6.8: TPS 値推移(更新系, SF=6,000)

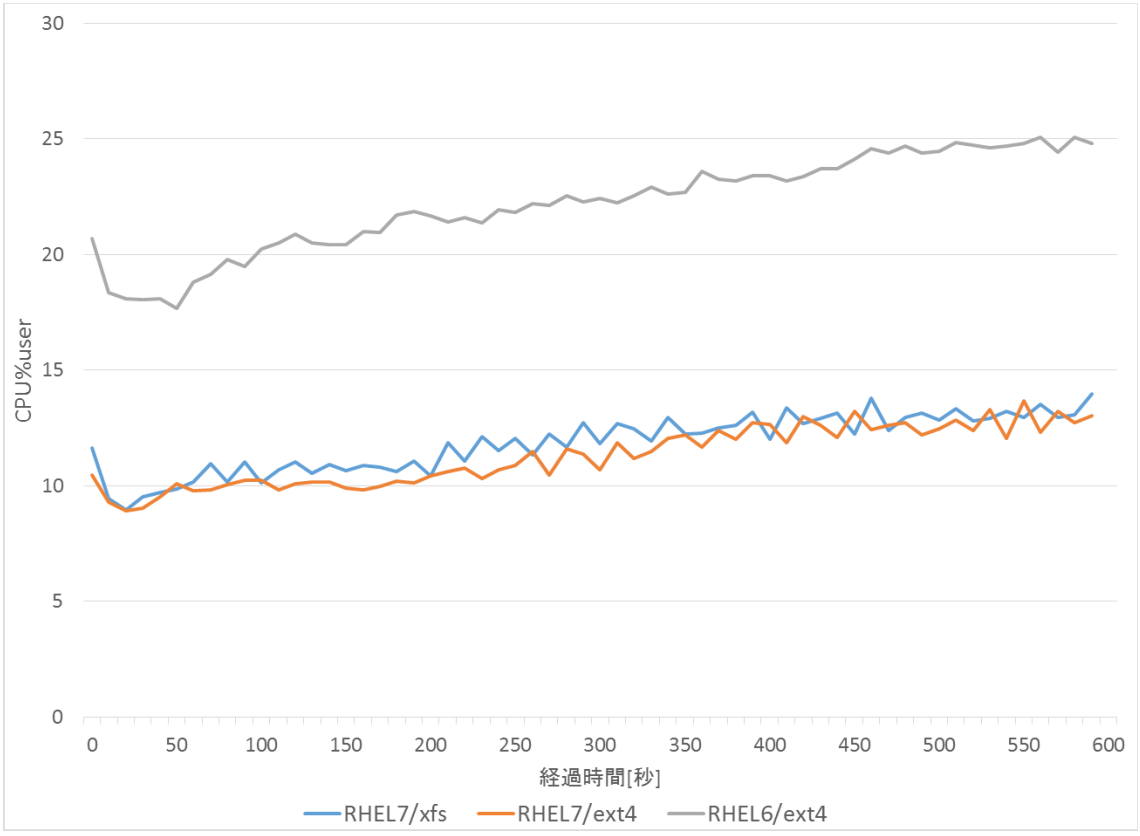

図 6.9: CPU 使用率推移(更新系, SF=6,000)

© 2016 PostgreSQL Enterprise Consortium

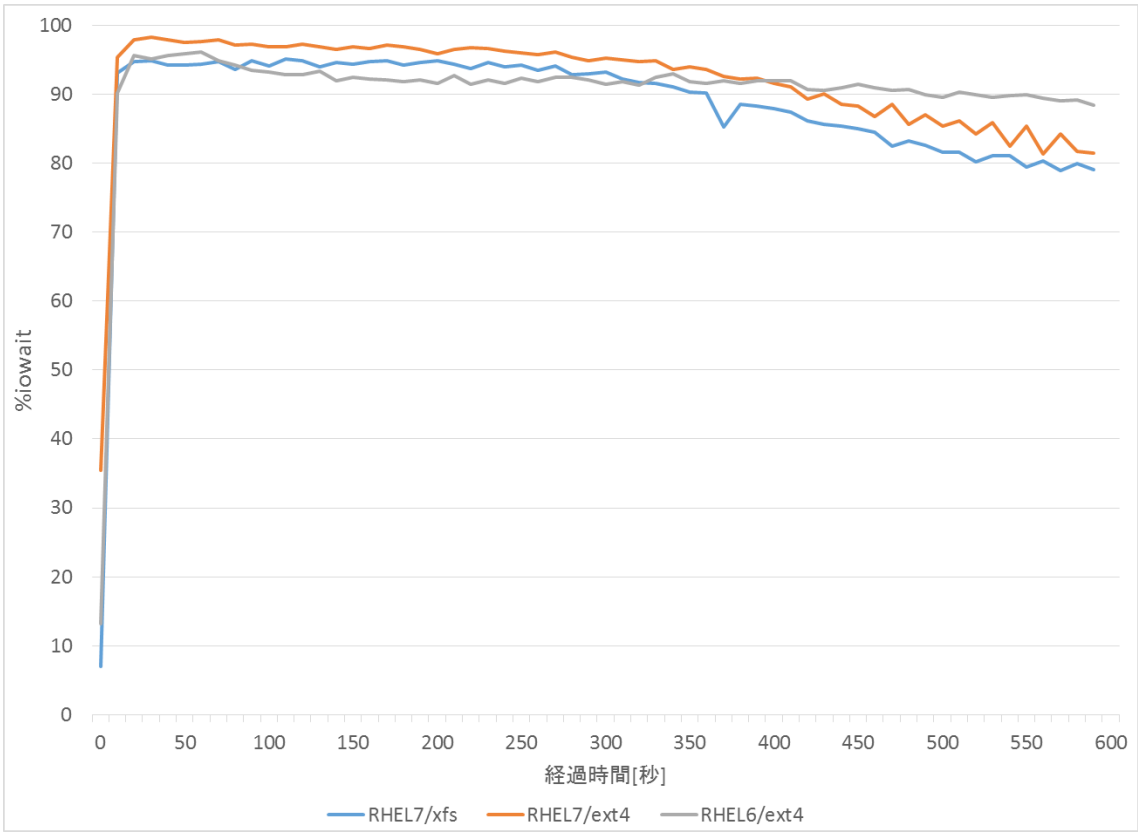

図 6.10: %iowait 推移(更新系, SF=6,000)

TPS に関しては、RHEL7/ext4、RHEL7/xfs、RHEL6/ext4 の順に性能が高いという結果になりました。また、RHEL7 のファイルシステム変更による差はそれほど大きくなく、RHEL6 と RHEL7 の OS 変更による差は比較的大きなものとなりました。また、CPU 使用率、I/O の使用率に関してはどのパターンでも同じような挙動を見ることが出来ました。

これらの結果は、参照系のデータがメモリに乗り切る場合(SF=6,000)と同様の傾向であり、データ量がメモリに対して小さい場合は RHEL7 の方が高い性能が得られると言えるのではないかと考えます。

6.4.2.2. データがメモリに乗り切らない場合（SF=60,000）

測定した TPS 値、CPU 使用率、%iowait の推移のグラフを以下に示します。測定は 3 回実施し、中央値を取ったものを時系列でプロットしています。

© 2016 PostgreSQL Enterprise Consortium

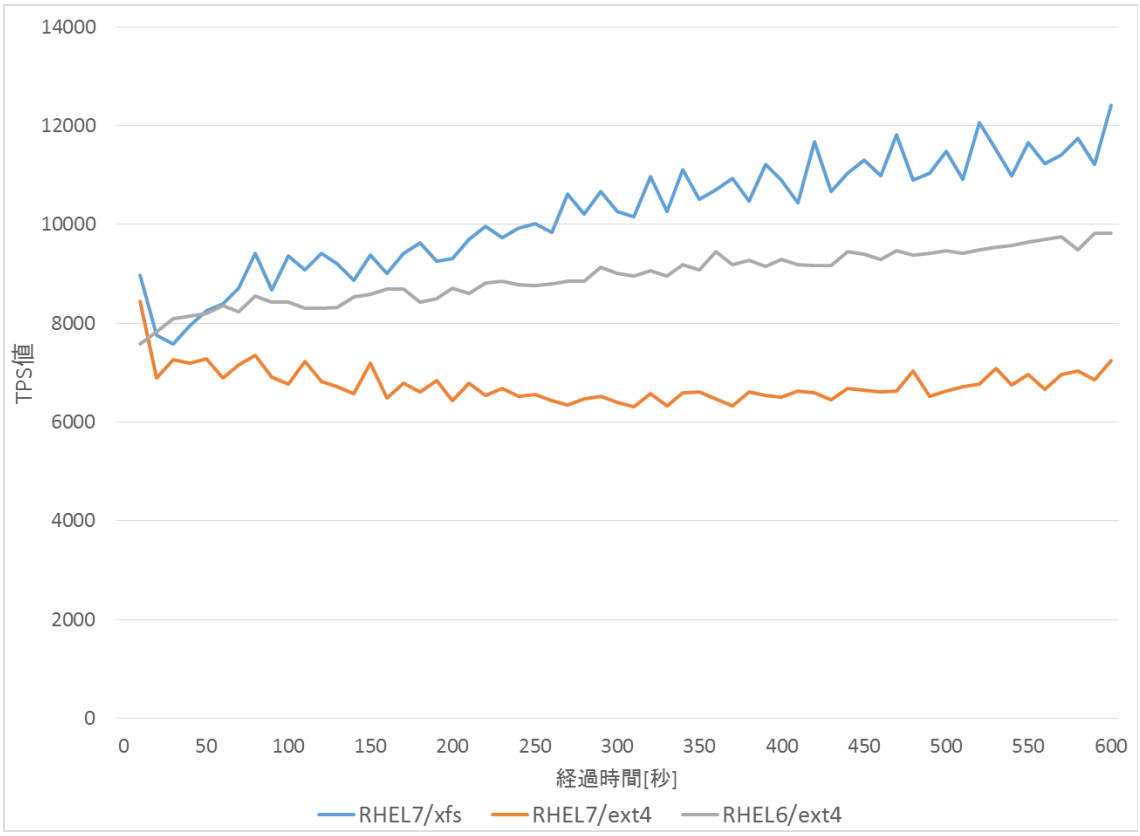

図 6.11: TPS 値推移(更新系, SF=60,000)

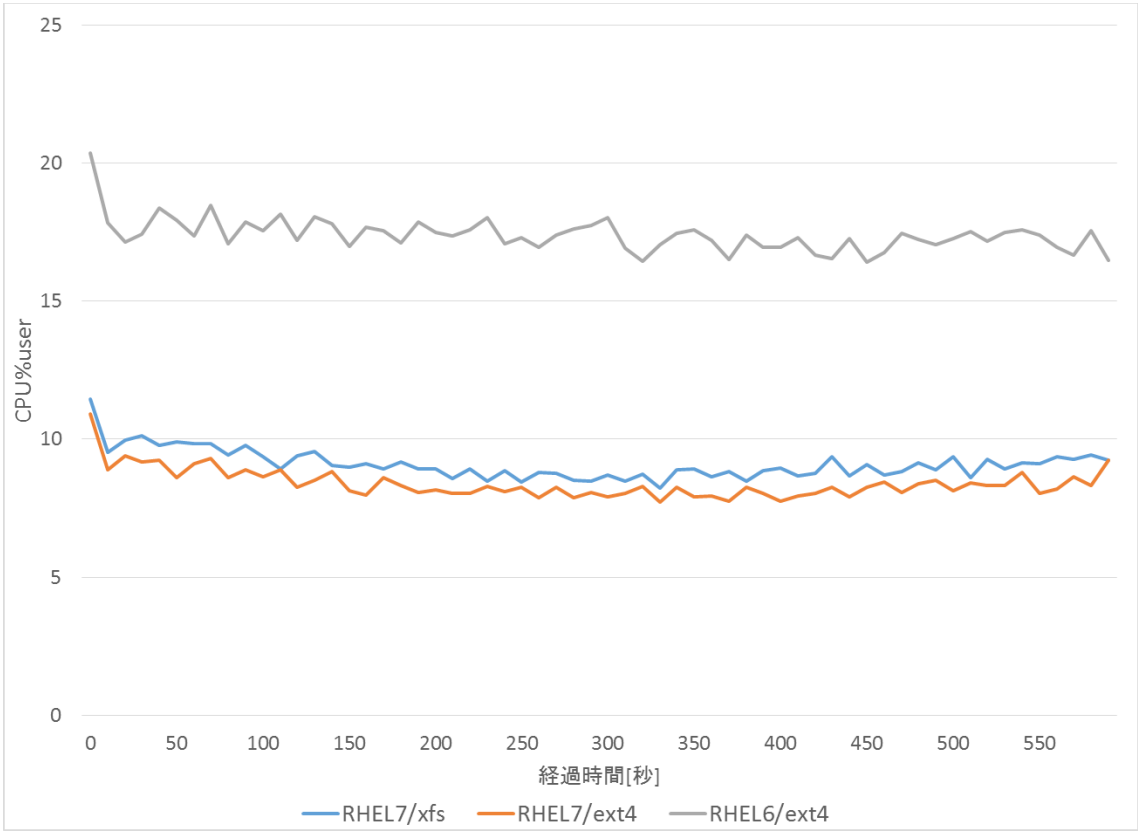

図 6.12: CPU 使用率推移(更新系, SF=60,000)

© 2016 PostgreSQL Enterprise Consortium

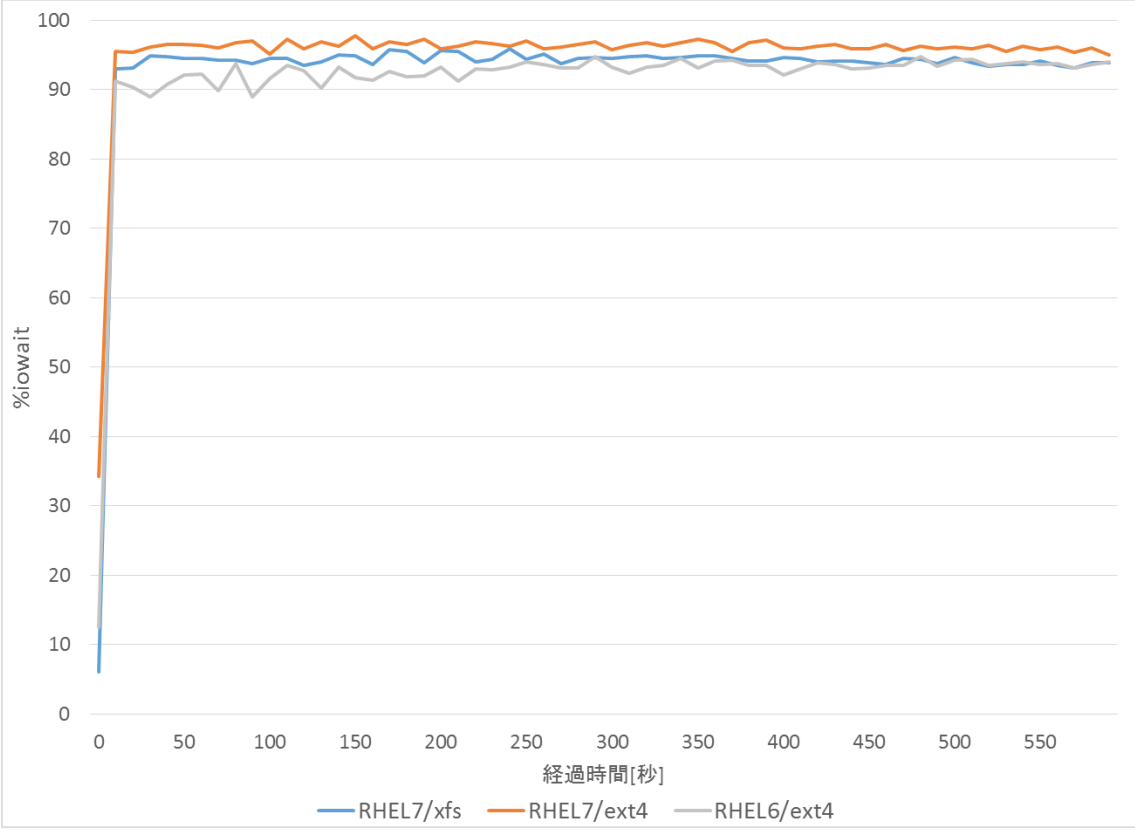

図 6.13: %iowait 推移(更新系, SF=60,000)

RHEL7/xfs の組み合わせが最も性能が高く、これはデータがメモリに乗り切る場合（SF=6,000）と同じとなりました。残りの二つの組み合わせ（RHEL7/ext4、RHEL6/ext4）では、データベースのサイズにより順位が逆転しています。

SF=6,000 の場合と SF=60,000 の場合を比較すると、RHEL7/ext4 の組み合わせの TPS 値の推移の傾向が違っていることが分かります。しかし、同じ OS あるいはファイルシステムを使用している残りの二つの組み合わせは同じような傾向を示しています。

以上から、OS とファイルシステム以外の要素がこの結果に影響していると考えられます。検証期間の制約上、残念ながらその原因の究明までには至りませんでしたが、次年度への課題としたいと考えます。

6.4.3. I/O スケジューラ比較

表 6.3 で示したように、RHEL6 と RHEL7 ではデフォルトの I/O スケジューラが異なっています。そこで、OS とファイルシステムの組み合わせ 3 パターンに対して、I/O スケジューラを変更して比較を行いました。比較した I/O スケジューラは deadline、cfq、noop の 3 種類です。参照系、更新系のクエリは前述の 6.3.3.と同じものを使用しています。参照系、更新系ともに各スケールファクタでの測定結果の TPS 値の推移グラフを以下に示します。

© 2016 PostgreSQL Enterprise Consortium

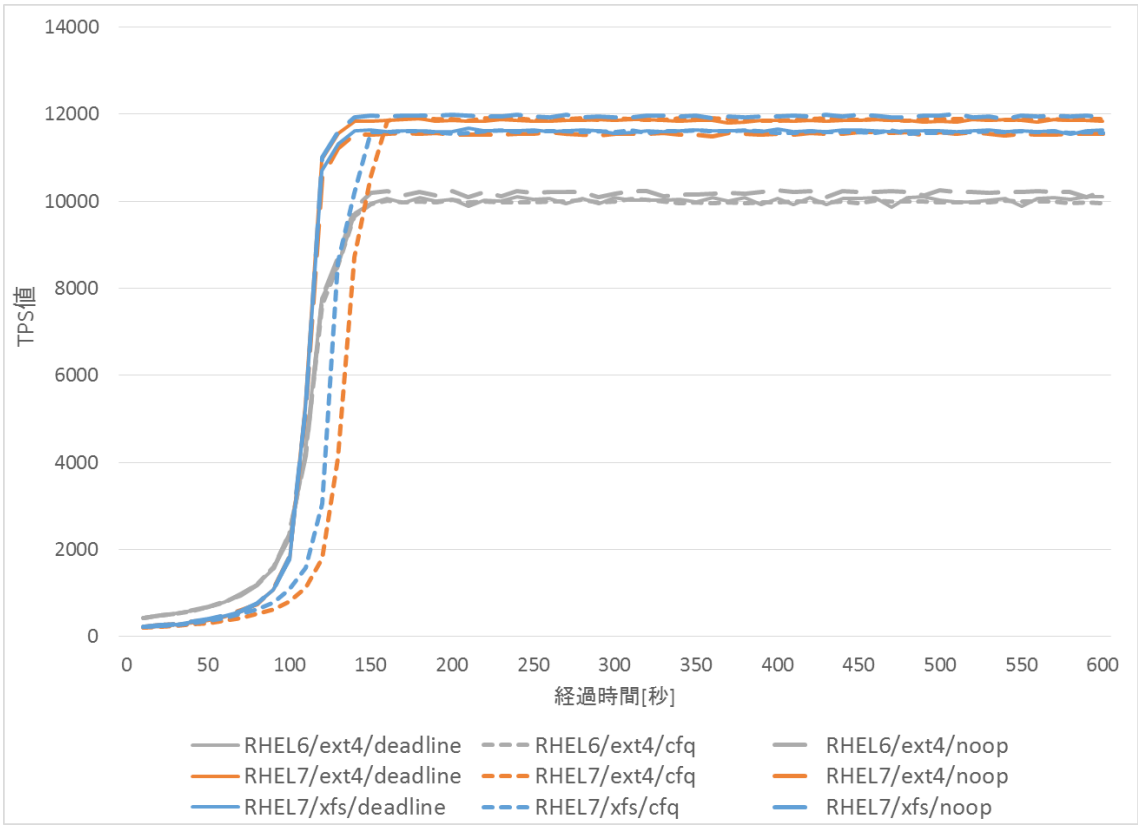

図 6.14: TPS 値推移(参照系, SF=6,000)

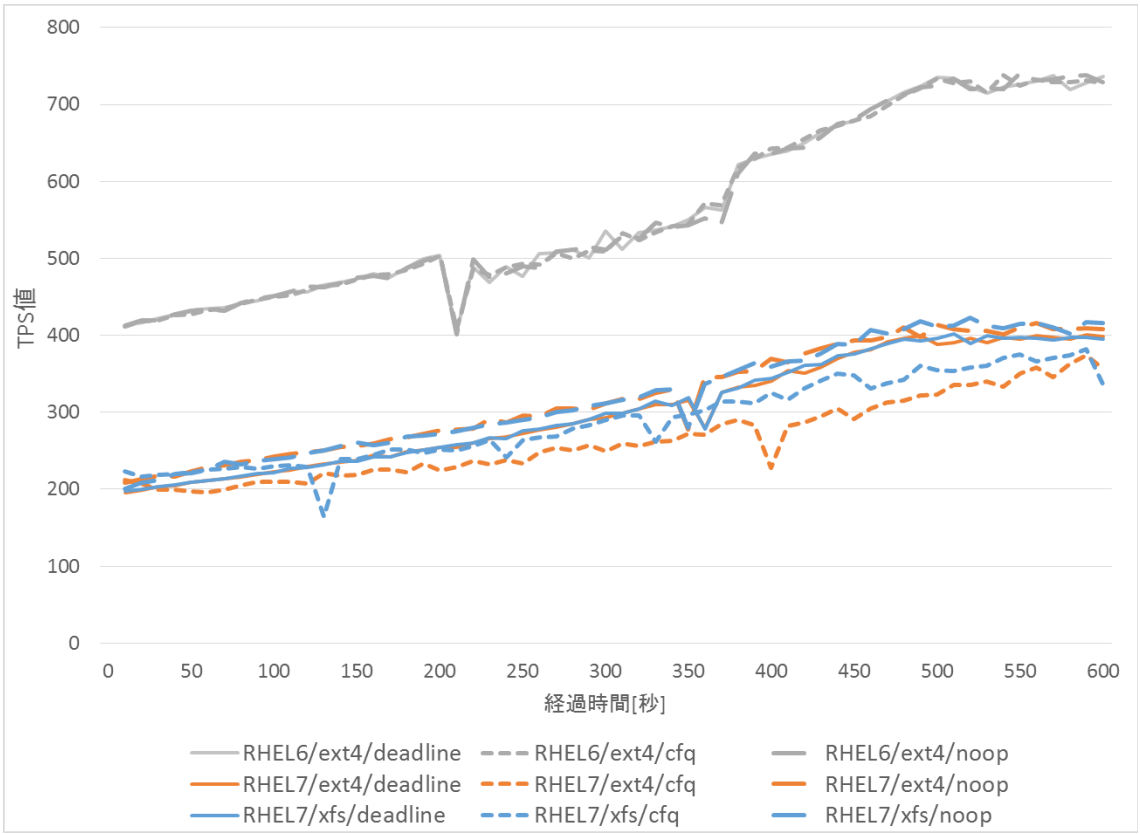

図 6.15: TPS 値推移(参照系, SF=60,000)

© 2016 PostgreSQL Enterprise Consortium

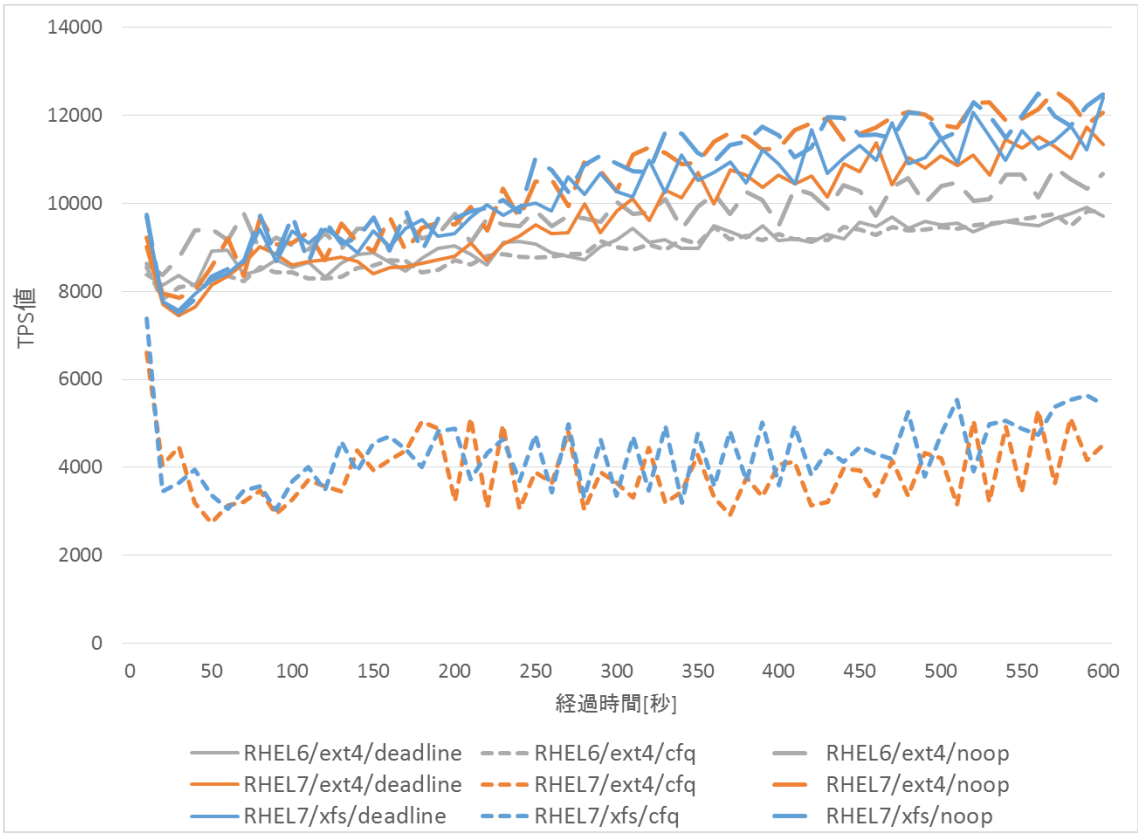

図 6.16: TPS 値推移(更新系, SF=6,000)

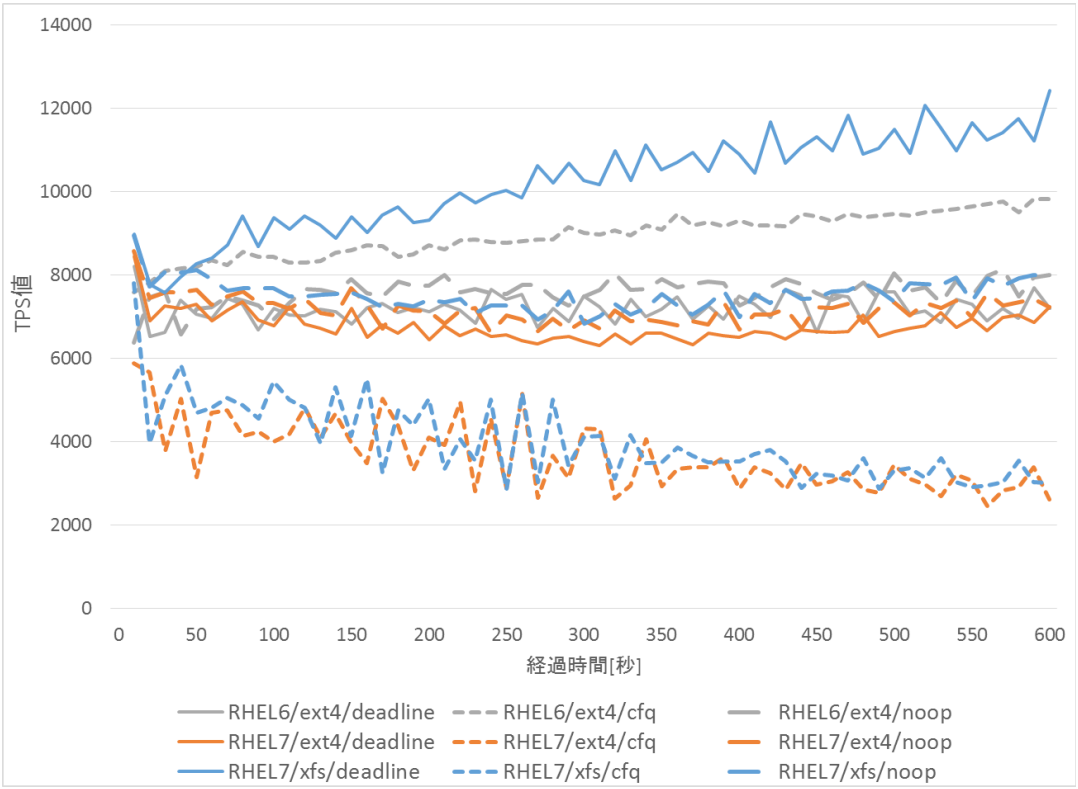

図 6.17: TPS 値推移(更新系, SF=60,000)

© 2016 PostgreSQL Enterprise Consortium

参照系に関しては、どの I/O スケジューラでも同様の結果となりました。一方で、更新系に関しては I/O スケジューラによって TPS に大きな差が生まれる結果となりました。

RHEL7/ext4 と RHEL6/ext4 に着目してみると、deadline と noop ではあまり差が見られないことが分かります。残りの cfq に関しては、RHEL6 では 3 つのスケジューラの中で最も速く、RHEL7 では 3 つのスケジューラの中で最も遅く、と大きな違いが出ています。表 6.1 に示した通り、RHEL6 では cfq がデフォルト設定となっています。また、SF=6,000 の場合のグラフから、RHEL7/cfq の組み合わせではデータ量が小さいにも関わらず性能が低く、OS 側と I/O スケジューラの相性が悪いのではないかと推測できます。

以上のことから、ファイルシステムを ext4 として比較した場合は基本的に RHEL7 の方が高い性能が得られますが、I/O スケジューラとして cfq を使用した場合のみ差が見られ、cfq をデフォルトとしている RHEL6 の方が性能が高くなっていることが分かります。このことが OS のデフォルト設定間で比較した場合、RHEL6 の方が性能が高くなるという結果をもたらしたものと考えられます。

また、更新系の場合はいずれのケースでも RHEL7/xfs/deadline の組み合わせが最も性能が高く、RHEL6 と同様に OS のデフォルト設定を使用した場合が最も性能が高くなるという結果が得られました。

6.5. 考察

本検証では OS のバージョンによる差異を比較するため、OS とファイルシステム、I/O スケジューラの組み合わせ全 9 パターンにおいて性能測定を行い、結果の比較を行いました。

参照系については、データベースのサイズによって高い性能が得られる OS/ファイルシステムが異なるという結果が得られました。また、ファイルシステムと I/O スケジューラを変更しても TPS 値があまり変化しなかったことから、OS の差による影響が最も大きいという傾向を見ることが出来ました。

更新系については、RHEL7/xfs の組み合わせが最も性能が高くなる一方で、ext4 を使用した場合はデータベースのサイズによって高い性能が得られる OS が異なるという結果となりました。また、I/O スケジューラと合わせて比較を行ったところ、RHEL6 では cfq を使用するのが良く、逆に RHEL7 では cfq を使うと性能が落ちるという結果となりました。

これらの結果から、各 OS で PostgreSQL を使用する場合は、ファイルシステムや I/O スケジューラをデフォルトのまま使用する場合が高い性能が得られることを確認できたといえます。

© 2016 PostgreSQL Enterprise Consortium

7. おわりに

今年度(2015 年度)は、PostgreSQL エンタープライズ・コンソーシアム(PGECons)の発足から数えて 4 年目となり、わたくしども技術部会ワーキンググループ 1(以下、本 WG)の活動報告書も、今回で 4 冊目となりました。第 1 章『はじめに』でも触れましたように、本 WG では PostgreSQL の性能に関する調査、検証を活動領域として、本報告書では、以下の 5 つのテーマの検証結果を報告しました。

1. 2016 年 1 月にリリースされた、PostgreSQL9.5 の性能評価、および、CPU のコア数に対するスケール性の検証
2. PostgreSQL9.5 の改善項目の一つである、排他制御の改善による性能向上の検証
3. PostgreSQL9.5 の新機能の一つである BRIN を使用したケースでの性能特性の検証
4. PostgreSQL9.5 の新機能の一つである Parallel VACUUM を使用した運用性の検証
5. Linux OS の代表的ディストリビューション Red Hat Enterprise Linux の新旧バージョン上での PostgreSQL9.5 の性能傾向についての検証

これらのテーマを継続・新規と大きく 2 つに分けると、継続てーまである上記の 1,2 番は PostgreSQL 本体の改善や変更に伴う性能傾向を調査し(「定点観測」と呼んでいます)、今年度の新テーマである 3,4 番は大規模 DB の性能と運用性の改善が期待できる PostgreSQL の新機能の検証、また 5 番は OS 環境の差異による PostgreSQL の性能傾向を検証しました。

以下では、今年度の検証活動を振り返って、具体的な進め方をご紹介することで、報告書のあとがきとしたいと思います。

まず、テーマの選定にあたっては、参加メンバからテーマを募るとともに、PGECons の成果報告会や、オープンソースカンファレンスなどに参加されたお客様に記入いただくアンケートを参考に進めました。

その結果、昨今の Big Data の流れも踏まえて、PostgreSQL をより大規模なデータベースに適用することを目指した新機能 2 種と、業務用の Linux OS として広く用いられている Red Hat Enterprise Linux のバージョンアップに伴う性能の差異を比較することとしました。

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

先にご紹介した通り、PGECons での検証テーマの選定は、みなさまのご意見を参考に進めておりますので、各種のイベントでのアンケートには、ご関心をお持ちのテーマについてお書きいただけば幸いです。また、性能について具体的なテーマをお持ちであれば、PGECons で一緒に検証を進めることが出来ないかご検討いただければ幸いです。PGECons への参加方法や、技術部会の各 WG での検証活動の年間スケジュールや活動風景の写真は、たとえば、OSC Enterprise Tokyo 2014 で発表した資料[2]にありますので、本報告書と合わせてご覧いただければと存じます。

(終わり)

2 PostgreSQL エンタープライズ・コンソーシアム技術部会, “商用 DB から PostgreSQL への移行検証結果”, https://www.pgecons.org/wp-content/uploads/2015/02/OSC_Enterprise_Tokyo_20141212.pdf ,2014.

© 2016 PostgreSQL Enterprise Consortium

著者

版	所属企業・団体名	部署名	氏名
大規模 DB への適用性が向上した PostgreSQL9.5 の性能検証 （2015 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
	日本電気株式会社	情報・ナレッジ研究所	堀川 隆
	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 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 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	NTT 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 ソフトウェア開発技術統括部 OSS 技術センター	野山 孝太郎
	富士通株式会社	共通ソフトウェア開発技術本部 ソフトウェア開発技術統括部 OSS 技術センター	高澤 亮平

© 2016 PostgreSQL Enterprise Consortium