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

2013 年度 WG1 活動報告

大規模ＤＢを見据えた PostgreSQL の性能検証

製作者
担当企業名:アイウエオ順
株式会社アイ・アイ・エム
株式会社アシスト
SRA OSS, Inc.日本支社
NEC ソリューションイノベータ株式会社
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
株式会社日立製作所
富士通株式会社

改訂履歴

版	改訂日	変更内容
1.0	2014/04/17	初版

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 および関連クラスタソフトによる、スケールアップ、パーティショニング、ハードウェア活用、および、スケールアウトの性能を検証した作業内容と結果を報告します。

■謝辞

検証用の機器を日本電気株式会社、日本ヒューレット・パッカード株式会社、富士通株式会社(順不同、敬称略)よりご提供いただきました。
また、一部の検証結果の収集・分析について株式会社アイ・アイ・エム様に性能評価サービスを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。

目次

1. はじめに

1.1. 2013 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 開発の 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

大分類	概要
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

小分類	概要
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。

1.1.2. 定点観測(スケールアップ検証)

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う懸念です。これに対して、2012 年度の取り組みとして、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、当時の最新バージョンであった PostgreSQL9.2 において、検索性能、更新性能がどこまでスケール出来るかを評価し、数値の公開にいたっております。

2013 年度も引き続き 2012 年度と同等のハードウェア環境を用意、PostgreSQL9.3 を対象にスケールアップ検証を実施し機能エンハンスによる性能影響を見極めることとしました。PostgreSQL9.3 を対象とした検証結果(定点観測)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。

1.1.3. パーティショニング検証

大規模基幹業務へ PostgreSQL を適用するには、データ容量の巨大化は避けては通れない課題です。これに対して、主要な商用RDBMSにはテーブルに挿入されたデータの格納先(ディスク)を分割させる、パーティショニング機能を備えています。

対して、PostgreSQL ではストアドプロシージャや、検査制約などの機能を組み合わせて、データ格納先を分割する(パーティショニング)を実現する必要があります。そこで、PostgreSQL におけるパーティショニングの有用性を、性能面(検索、更新)、運用面(領域の循環利用)を観点にした性能検証結果として示すこととしました。

1.1.4. ハードウェア(SSD)活用検証

近年 SSD の低価格化・大容量化とエンタープライズモデルの広がりを受けて、データベースの新たな高速化手法としてストレージデバイスに SSD を採用することに注目が集まっています。

そこで、PostgreSQL をデータベースとした場合、如何に SSD を活用すると費用対効果の高いシステムが構築できるのかを観点として性能検証結果を示すこととしました。

1.1.5. スケールアウト検証

データベースサーバを複数台使ってより高い性能を得るスケールアウトに関して、2012 年度の活動において PostgreSQL のカスケードレプリケーションや、pgpool-II、Postgres-XC といった PostgreSQL を中心にしたスケールアウト構成の代表的なものを取り上げて、それぞれの有用性を示すことができました。

2013 年度は、更新スケールアウトが可能な Postgres-XC に注目し、クラスタを構成するサーバの台数に応じて性能が向上することの性能検証結果を示すこととしました。

1.2. 実施体制

2013 年 6 月 6 日に開催された 2013 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:アイウエオ順)。

表 1.3: 2013 年度 WG1 参加企業一覧

株式会社アイ・アイ・エム
株式会社アシスト
SRA OSS, Inc.日本支社
NEC ソリューションイノベータ株式会社
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
株式会社日立製作所
富士通株式会社

この中で、富士通株式会社が「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。

1.3. 実施スケジュール

2013 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	2013 年 6 月 6 日
実施計画策定	2013 年 8 月～10 月
検証実施	2013 年 11 月～12 月
2013 年度 WG1 活動報告書作成	2014 年 1 月～3 月
総会と成果報告会	2014 年 4 月 25 日

2. 定点観測(スケールアップ検証)

2.1. 概要

近年の CPU マルチコア化やメモリ大容量化の傾向から、昨年 2012 年度は、80 コアの CPU、メモリ 2TB という非常に高いスペックの、マシンを利用し PostgreSQL 9.2.の参照系、更新系の性能を測定しました。
今年度はほぼ同環境で、PostgreSQL 9.3 の参照性能の測定を行ないました。

また、PostgreSQL 9.3 には page checksum の機能が追加されており、これによるオーバヘッドを把握するために、page checksum 有効/無効時の参照性能を測定 しました。page checksum は、データブロックごとにチェックサムを付与し、データブロック破損の検知を行なうものです。

そのほか、2012 年度は実施できなかった、CPU コア数を変えての性能測定も行ないました。

したがって、以下の 3 つについて検証を行ないました。

表 2.1: スケールアップ検証 検証内容

	概要	測定方法
1	PostgreSQL 9.2 と 9.3 の参照性能の比較	80 コアの CPU で、クライアント数を 1 から 128 まで変動させて性能を比較する。
2	PostgreSQL 9.3 で、page checksum を使っているときと使っていないときでの、参照性能の比較	80 コアの CPU で、クライアント数を 1 から 128 まで変動させて性能を比較する。
3	PostgreSQL 9.3 の CPU コア数の違いによる参照性能の比較	CPU コア数を 1 から 80 まで変動させ、クライアント数が 80 のときの性能を比較する。

2.2. pgbench とは

この検証では、pgbench というベンチマークツールを使用しました。

pgbench は PostgreSQL に contrib として付属する簡易なベンチマークツールです。
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が実行できる他、検索のみなどのシナリオも搭載されています。また、独自のシナリオをスクリプトとして用意しておき、実行することもできます。

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)が表示されます。なお、「including connections establishing」は、PostgreSQL に接続する時間を含んだ TPS、「excluding connections establishing」は含まない 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 1.5MB のデータベースが作成されます。

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

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表に示します。

● 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
-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ることができます。本検証で利用した機能を簡単に説明します。

```
¥set nbranches :scale
```

-s で指定したスケールファクタを変数「nbranches」に設定します。なお、
¥set 文では四則演算も利用できます。

```
¥set ntellers 10 * :scale
```

設定した変数は、スクリプトに書き込んだ 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. PostgreSQL 9.2 と 9.3 の参照性能の比較

2.3.1. 検証目的

PostgreSQL 9.2 と 9.3 の参照性能を比較しました。バージョン間で大きく挙動が変わることなく、同傾向、同程度の参照性能が出ることを期待しています。

2.3.2. 検証構成

2013 年度活動報告書 Appendix1 検証環境の検証環境 1 に別途記載しています。

2.3.3. 検証方法

(1) 環境

2012 年度と同様のデータベースクラスタを作成しました。

initdb 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
checkpoint_segments = 16
checkpoint_timeout = 30min
logging_collector = on
logline_prefix = '%t [%p-%l] '
```

pgbench を使い、スケールファクタ 1000 でデータベースクラスタを初期化します。

```
$ pgbench -i -h [pgpool host] -p [pgpool port] -s 1000
```

(2) 測定

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。内容としては、ランダムに 10000 行を取得する、というものです。

```
¥set nbranches :scale
¥set ntellers 10 * :scale
¥set naccounts 100000 * :scale
¥set row_count 10000
¥set aid_max :naccounts - :row_count
¥setrandom aid 1 :aid_max
¥setrandom bid 1 :nbranches
¥setrandom tid 1 :ntellers
¥setrandom delta -5000 5000
```

```
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 -h [pgpool host] -p [pgpool port] test -c [clients] -j [threads] -T 300 -n -f custom.sql
```

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。

それぞれ 3 回ずつ実行し、その中央値を結果とします。

(3) 結果

PostgreSQL 9.2（赤、＋ マーク）も 9.3（緑、×マーク）も、80 clients が最大の TPS となりました。また、9.3 の方が若干低い結果になりました。表の CPU 使用率は、実行時間中のすべてのコアにおける sar の「%idle」列平均値です。

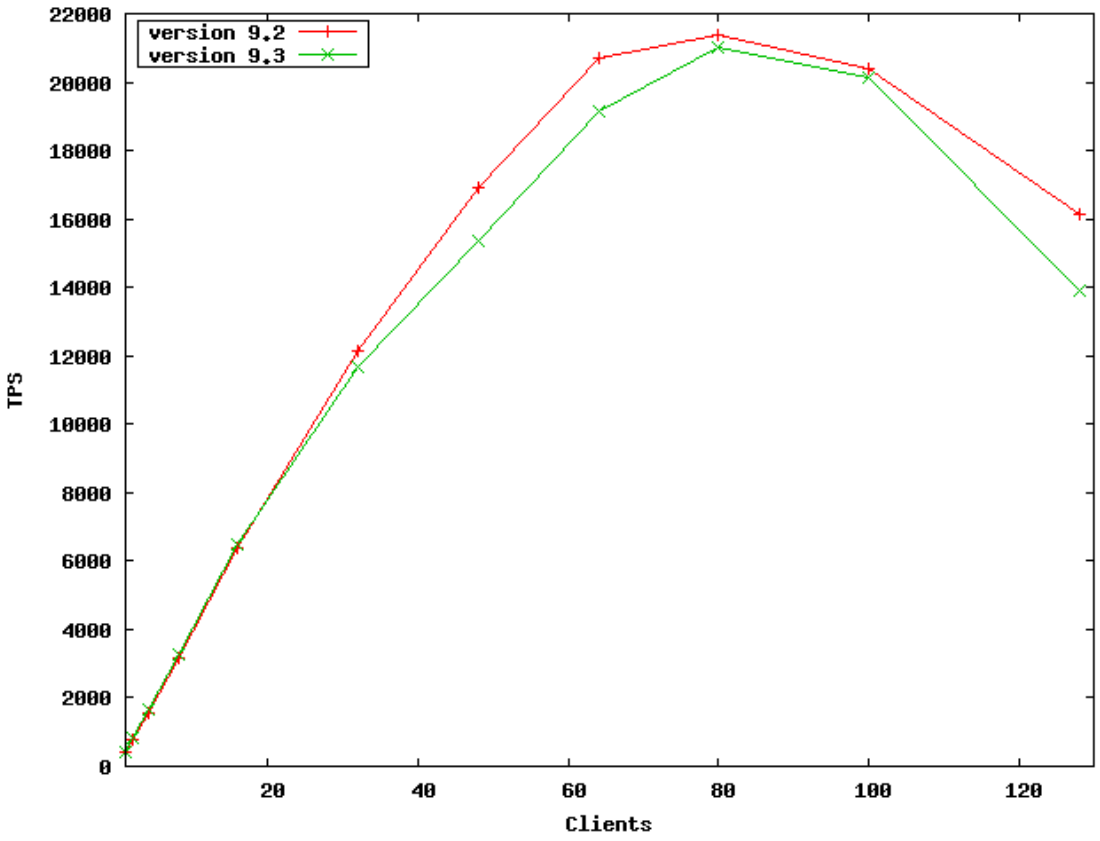

図 2.1: PostgreSQL 9.2 と 9.3 の参照性能の比較

表 2.2: PostgreSQL 9.2 と 9.3 の参照性能の比較

クライアント数	PostgreSQL 9.2		PostgreSQL 9.3		
	TPS	CPU idle	TPS	CPU idle	9.2 の TPS との比較
1	403.38	98.25 %	418.77	98.30 %	103.82 %
2	792.78	97.12 %	821.87	97.09 %	103.67 %
4	1571.25	94.78 %	1634.70	94.78 %	104.04 %
8	3150.75	90.01 %	3263.18	89.99 %	103.57 %
16	6383.30	80.39 %	6509.86	80.36 %	101.98 %
32	12123.73	60.85 %	11685.23	60.87 %	96.38 %
48	16916.39	41.28 %	15373.38	41.22 %	90.88 %
64	20697.27	21.71 %	19126.20	21.54 %	92.41 %
80	21378.30	3.71 %	21022.63	3.62 %	98.34 %
100	20405.70	1.34 %	20152.03	1.46 %	98.76 %
128	16141.99	0.66 %	13893.93	0.69 %	86.07 %

2.3.4. 考察

クライアント数が 32 以上になると PostgreSQL 9.2 より 9.3 の方が TPS 値が下がり、最大で 14%(クライアント数 128 のとき)の低下が見られました。

また、クライアント数 100 以上では 9.3 の方が CPU idle が多く、PostgreSQL に充分負荷がかかっていないことがうかがえます。

今回使用した SQL は去年の検証でも利用したもので、PostgreSQL 9.2 にあわせて調整したものでした。PostgreSQL 9.3 にあわせて調整しなおせば、また違った結果が得られたかもしれません。この点は今後の定点観測の課題とします。

2.4. PostgreSQL 9.3: page checksum 有無による参照性能の比較

2.4.1. 検証目的

page checksum 利用時、デフォルトである非利用時にくらべて、参照性能がどのくらい劣化するのかを確認しました。

2.4.2. 検証構成

PostgreSQL 9.2 と 9.3 の参照性能の比較の構成と同じ。

2.4.3. 検証方法

(1) 環境

page checksum を使うために「--data-checksums」オプションをつけて initdb します。postgresql.conf を PostgreSQL 9.2 と 9.3 の参照性能の比較と同様に編集します。

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

同じく、pgbench を使いスケールファクタ 1000 でデータベースクラスタを初期化します。

```
$ pgbench -i -h [pgpool host] -p [pgpool port] -s 1000
```

(2) 測定

PostgreSQL 9.2 と 9.3 の参照性能の比較の測定方法と同じ。

(3) 結果

page checksum のオーバヘッドはおもに、ページのデータの整合性を検証する計算処理です。そのため、クライアント数が多いときに影響が強くあらわれています。ただ、クライアント数が少なくCPU 利用率が低い場合には、その影響はほとんど無いといえます。

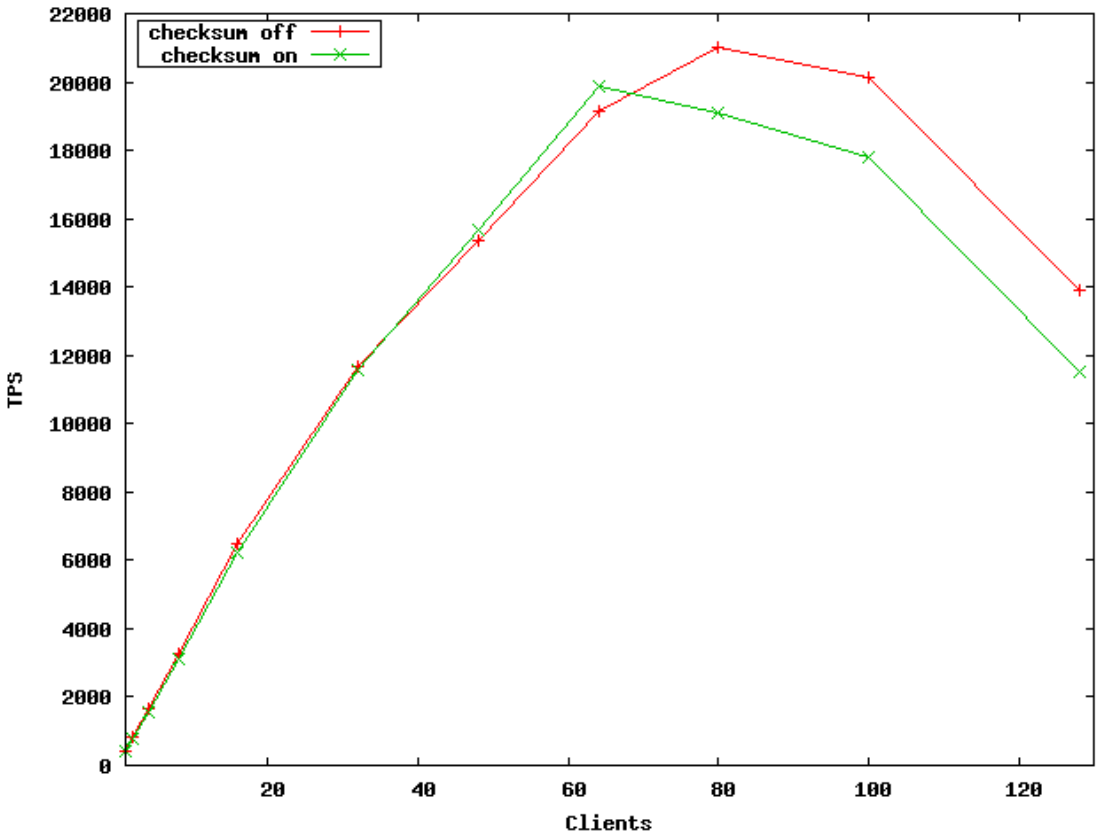

図 2.2: PostgreSQL 9.3: page checksum 有無による参照性能の比較

表 2.3: PostgreSQL 9.3: page checksum 有無による参照性能の比較

クライアント数	PostgreSQL 9.3: page checksum なし		PostgreSQL 9.3: page checksum あり		
	TPS	CPU idle	TPS	CPU idle	checksum なしとの比較
1	418.77	98.30 %	414.53	98.25 %	98.99 %
2	821.87	97.09 %	801.86	97.06 %	97.56 %
4	1634.70	94.78 %	1573.79	94.74 %	96.27 %
8	3263.18	89.99 %	3125.97	90.03 %	95.80 %
16	6509.86	80.36 %	6236.96	80.30 %	99.93 %
32	11685.23	60.87 %	11548.55	60.89 %	98.83 %
48	15373.38	41.22 %	15680.36	41.26 %	81.98 %
64	19126.20	21.54 %	19880.09	21.65 %	103.94 %
80	21022.63	3.62 %	19100.96	3.34 %	90.86 %
100	20152.03	1.46 %	17812.27	1.29 %	88.39 %
128	13893.93	0.69 %	11527.28	0.63 %	83.00 %

2.5. PostgreSQL 9.3(特殊コンパイル): page checksum 有無による参照性能の比較

2.5.1. 検証目的

checksum アルゴリズム自体は、FNV-1a hash に基づいて作成されたものですが、不具合があるため PostgreSQL ではこれをこのままは使わず、改良したものが使われています。
checksum は乗算処理が重いため、ページを 32 カラムの 2 次元配列で扱うことで並列に処理するように改善しています。この 2 次元配列をベクトル化するのに際して、以下のコンパイルオプションを使うと有益であるとされています。

```
-msse4.1 -funroll-loops -ftree-vectorize
```

このことは、PostgreSQL 9.3 のソースコードの src/include/storage/checksum_impl.h にかかれています。

2.5.2. 検証構成

PostgreSQL 9.2 と 9.3 の参照性能の比較の構成と同じ。

2.5.3. 検証方法

(1) 環境

ソースコードの Makefile.global の CFLAGS にこのコンパイルオプションをつけ、PostgreSQL をコンパイルしなおします。

```
$ make clean

$ vi {ソースコード展開ディレクトリ}/src/Makefile.global
CFLAGS = -O2 -Wall -Wmissing-prototypes -Wpointer-arith -Wdeclaration-after-statement -Wendif-labels
-Wmissing-format-attribute -Wformat-security -fno-strict-aliasing -fwrapv -msse4.1 -funroll-loops -ftree-
vectorize

$ ./configure --prefix={インストール先}
$ make
$ make install
```

page checksum を使うために「--data-checksums」オプションをつけて initdb します。

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

pgbench を使いスケールファクタ 1000 でデータベースクラスタを初期化します。

```
$ pgbench -i -h [pgpool host] -p [pgpool port] -s 1000
```

(2) 測定

PostgreSQL 9.2 と 9.3 の参照性能の比較の測定方法と同じ。

(3) 結果

コンパイルオプションをつけたうえでの checksum あり状態（青、＊ マーク）は、オプション無し時（緑、× マーク）より性能の劣化が少なく、checksum なし状態（赤、＋ マーク）に近い結果となりました。

また、クライアント数が多いときには、checksum なし状態よりもさらによい TPS が出ました。この理由は不明ですが、page checksum 以外のところにもコンパイルオプションの変更による影響が出たことによる可能性があります。

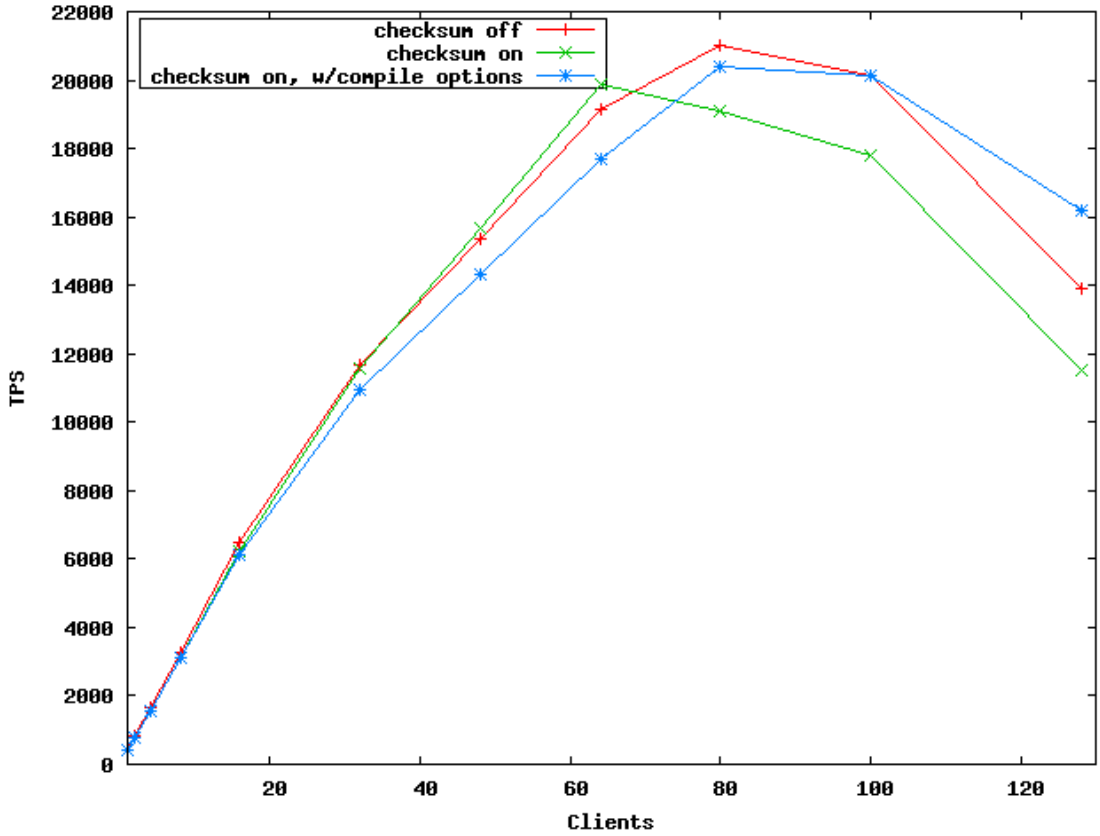

図 2.3: PostgreSQL 9.3: page checksum 有無による参照性能の比較（コンパイルオプションつき）

表 2.4: PostgreSQL 9.3: page checksum 有無による参照性能の比較(コンパイルオプションつき)

クライアント数	PostgreSQL 9.3: page checksum なし		PostgreSQL 9.3: page checksum あり			PostgreSQL 9.3: page checksum あり (コンパイルオプションつき)		
	TPS	CPU idle	TPS	CPU idle	checksum なしとの比較	TPS	CPU idle	checksum なしとの比較
1	418.77	98.30 %	414.53	98.25 %	98.99 %	402.43	97.12 %	96.10 %
2	821.87	97.09 %	801.86	97.06 %	97.56 %	791.78	97.09 %	96.34 %
4	1634.70	94.78 %	1573.79	94.74 %	96.27 %	1572.36	94.70 %	96.19 %
8	3263.18	89.99 %	3125.97	90.03 %	95.80 %	3116.98	89.98 %	95.52 %
16	6509.86	80.36 %	6236.96	80.30 %	99.93 %	6143.26	80.30 %	94.37 %
32	11685.23	60.87 %	11548.55	60.89 %	98.83 %	10965.65	60.80 %	93.84 %
48	15373.38	41.22 %	15680.36	41.26 %	81.98 %	14317.56	41.11 %	93.13 %
64	19126.20	21.54 %	19880.09	21.65 %	103.94 %	17673.27	21.48 %	92.40 %
80	21022.63	3.62 %	19100.96	3.34 %	90.86 %	20377.04	3.50 %	96.93 %
100	20152.03	1.46 %	17812.27	1.29 %	88.39 %	20142.16	1.42 %	99.95 %
128	13893.93	0.69 %	11527.28	0.63 %	83.00 %	16207.84	0.00 %	116.65 %

2.5.4. 考察

同時接続クライアント数が少ないときには、page checksum 無効時よりもむしろ、page checksum(コンパイルオプションつき)の方がよい TPS が出ています。これは、コンパイルオプションの変更によって、page checksum 処理以外のところにも良い影響が出たためと考えられます。

2.6. PostgreSQL 9.3: CPU コア数の違いによる参照性能の比較

2.6.1. 検証目的

CPU コア数が 80 の状態で、クラアント数の数にる参照性能の違いを検証しました。

2.6.2. 検証構成

PostgreSQL 9.2 と 9.3 の参照性能の比較の構成と同じ。

2.6.3. 検証方法

(1) 環境

PostgreSQL 9.2 と 9.3 の参照性能の比較の構成と同じ。

(2) 測定

PostgreSQL 9.2 と 9.3 の参照性能の比較の測定方法と同じ。
ただし、各 CPU コア数環境で 1 回ずつのみ測定しました。

(3) 結果

コア数と同じである 80clients までスケールしました。

グラフの横軸は CPU コア数、縦軸は TPS です。

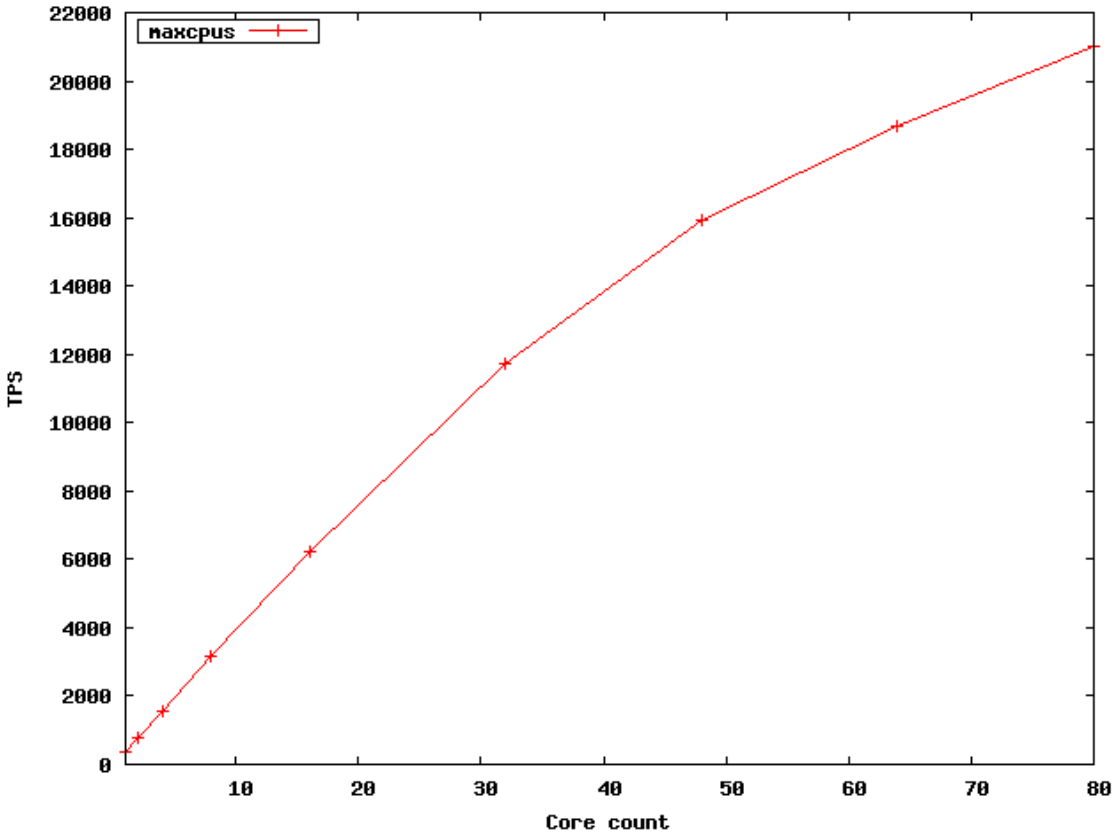

図 2.4: PostgreSQL9.3: CPU コア数の違いによる参照性能の比較

表 2.5: PostgreSQL9.3: CPU コア数の違いによる参照性能の比較

CPU コア数	TPS	CPU idle
1	385.17	0.00 %
2	785.06	0.00 %
4	1581.16	0.00 %
8	3171.91	0.00 %
16	6248.99	0.00 %
32	11739.36	0.01 %
48	15925.23	0.60 %
64	18685.82	1.46 %
80	21022.63	3.63 %

2.6.4. 考察

同時接続クライアント数を 80 に固定し、コア数を変化させたときの TPS は、ほぼコア数に比例して向上しています。このことから、PostgreSQL のスケールアウト性能が良好であることが確認できました。

3. パーティショニング検証

3.1. 検証概要

エンタープライズ領域への PostgreSQL 適用にあたり、データ容量の巨大化は避けて通れない課題です。テーブルに格納するデータ量の増大は性能劣化やデータベース運用を困難にする要因となるため、主要な商用 RDBMS はテーブルの格納先を複数の領域に分割するパーティショニング機能を備えています。

PostgreSQL はテーブルの継承、トリガを使用した子テーブルへの更新、CHECK 制約による読み飛ばしを組み合わせたパーティショニング機能を提供しており、本章では蓄積するログデータを想定したパーティションテーブルへのデータ投入、検索、メンテナンス性能検証結果を紹介します。

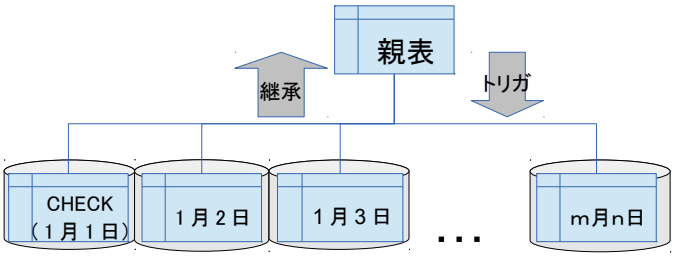

図 3.1: PostgreSQL のパーティショニング

3.2. 検証目的

本検証を通じ、大容量データへの検索、運用処理性能に対するパーティショニングの効果と、パーティションテーブルの更新処理方式による性能差を示し、性能要件や開発コストに見合った方式を検討するための情報提供を目的とします。

3.3. 検証構成

3.3.1. 機器構成

DB サーバ上で PostgreSQL を 1 インスタンスと性能測定用クライアントを実行しています。

本検証では、パーティションごとにボリュームを割り当てず、すべて同一のテーブル空間上にデータを配置しています。DB の格納先パーティションは FC 接続のストレージ上に配し、ファイルシステムは ext4 を採用しています。

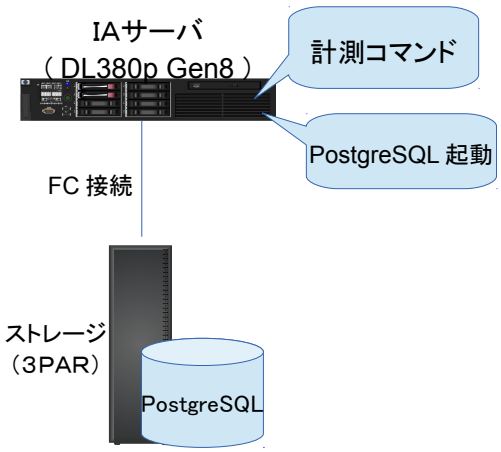

図 3.2: 検証構成

サーバスペックは以下の通りです。

表 3.1: 評価機スペック

CPU	インテル ®Xeon® プロセッサー E5-2690(2.90GHz , 8C/16T, 20MB)×2
メモリ	128GB
OS	Redhat Enterprise Linux 6

3.3.2. DB 設定

性能測定に当たり、以下のパラメータを変更しています。

表 3.2: DB パラメータ設定

パラメータ名	設定値	内容
shared_buffers	12288MB	物理メモリの 25%を目安として設定値を決定。
work_mem	256MB	ソートワークファイルを使用しないよう、十分な領域を確保。
wal_buffers	128MB	トランザクション処理中に頻繁に WAL ファイル出力が行われないよう、十分な領域を確保。
checkpoint_segments	256	大量更新に備え、CHECKPOINT 間隔を広げる。
checkpoint_timeout	30min	
checkpoint_completion_target	0.9	CHECKPOINT による性能が不安定になる状況を避けるため、ゆっくりと CHECKPOIN T処理を行うよう設定。
maintenance_work_mem	512MB	VACUUM 検証に向け、十分な領域を確保。
autovacuum	off	VACUUM 運用を行わない(パーティションごとの INSERT/TRUNCATE)ため、自動 VACUUM 機能は使用しない。

3.4. 検証方法

本評価では大量データを DB に格納する例としてログデータの格納先として PostgreSQL の採用を想定したシナリオを作成し、更新、参照性能や運用上のメリットを性能面から検証します。データの格納件数は 1 日(1 パーティション)あたり 768 万件とし、データの維持期間やトリガの実装など条件を変えて、挿入、検索、運用に関する性能測定を行います。

表 3.3: 検証項目

大項目	検証項目
データ挿入	トリガ関数として静的関数(PL/pgSQL)、動的関数(PL/pgSQL)、C 言語関数を使用し、単表およびパーティションテーブル(1 か月、3 か月、6 か月)に対してデータの挿入を行います。
検索	シーケンシャルスキャン(enable_seqscan)、インデックススキャン(enable_indexscan)、ビットマップスキャン(enable_bitmapscan)のいずれかのみを有効にして、単表およびパーティションテーブルの 1 日分のデータを対象とした集計処理を行います。また、インデックスオンリースキャン(enable_indexonlyscan)を行う場合と更新により通常のインデックススキャンになってしまったケースの性能比較も行います。
運用	1 日分のデータの削除処理性能を検証するため、単表に対しては条件指定の DELETE を、パーティション表に対しては対象パーティションの TRUNCATE を行っています。合わせて、VACUUM 処理時間の検証も行います。

3.4.1. データ挿入

データ挿入性能を計測する際、以下のトリガ関数を使用した性能比較を行います。

1. 静的関数
   挿入するデータのパーティションキーを判定し、各パーティションへの INSERT 文を選択して実行します。実行する SQL 文は関数定義の時点で決定しています。パーティション数分の IF 文を同レベルに並べているため、後半のパーティションは実行する判定処理が増え、性能が劣化する可能性があります。
2. 動的関数
   挿入するデータのパーティションキーから INSERT 文を都度生成して実行します。毎回 INSERT 文の解析を行うためオーバヘッドがかかります。パーティション数に応じた判定処理が不要なため、パーティション数が多い場合には静的関数と比べて安定した性能が見込めます。
3. C 言語関数
   C 言語関数で直前に実行した INSERT 文のプランをキャッシュし、前回と同じパーティションキーを持つデータであればキャッシュしたプランを使用します。今回は日付をパーティションキーとしているため前回実行したキャッシュが使用できる確率がきわめて高く、最も高速に INSERT を実行できると予想できます。

表 3.4: トリガ関数の実装方式別比較

比較項目	静的関数	動的関数	C 言語関数
関数の作りこみ	容易	容易	難しい
パーティション追加時の定義変更	必要	不要（設計に依存）	不要（設計に依存）
性能	低速	低速	高速
パーティション増加による性能劣化	大	中	小

3.4.2. 検索

1 パーティション分のデータを集計する条件検索を行います。この際、シーケンシャルスキャン、インデックススキャン、ビットマップスキャンのいずれかのみを有効にする指定を行います。クエリイメージは以下の通りです。

```
SELECT product_name, err_code, AVG(access_time_second)
  FROM product_master, access_log
  WHERE product_master.productid = access_log.productid
    AND date BETWEEN '20140101' AND '20140131'
  GROUP BY product_master.product_name, access_log.err_code ;
```

インデックスオンリースキャンは、データ挿入後に全く更新されない状態で以下のクエリを実行します。インデックスのみを参照するため、単表とパーティション表の性能差があまり大きく表れない可能性があります。

```
SELECT count(date)
  FROM access_log
  WHERE date BETWEEN '20140101' AND '20140131'
    AND time BETWEEN '000000' AND '120000';
```

3.4.3. 運用

本シナリオでは、定められたログの保存期間が経過したログを消去する運用とします。

データ削除処理の性能を確認するため、単表に対しては日付を指定した DELETE の、パーティション表に対しては最古のパーティションを指定した TRUNCATE 性能を計測します。

また、VACUUM および ANALYZE 性能も計測しますが、VACUUM 処理はインデックスファイル全体をチェックするため、パーティションテーブルの処理性能は単表に比べて大幅に改善することが見込まれます。

3.5. 検証結果

3.5.1. データ挿入

4.4.1 で記載したトリガ関数を使用して、データ挿入性能を比較しました。またいずれの関数も使用せず、パーティション表に直接挿入した場合および、パーティション表と同一のデータを格納した単表 (非パーティション)についても同様に測定しました。検証結果を以下に記載します。

表 3.5: データ挿入処理の応答時間(秒)

	1ヶ月 (約 30 パーティション)	3ヶ月 (約 90 パーティション)	6ヶ月 (約 180 パーティション)
静的関数	8,150	99,916	0
動的関数	2,390	7,945	16,920
C 言語関数	1,713	4,910	9,988
トリガ関数未使用	1,064	3,094	16,160
単表 (非パーティション)	1,017	3,075	0

(*1) 検証期間の都合により計測を打ち切り。

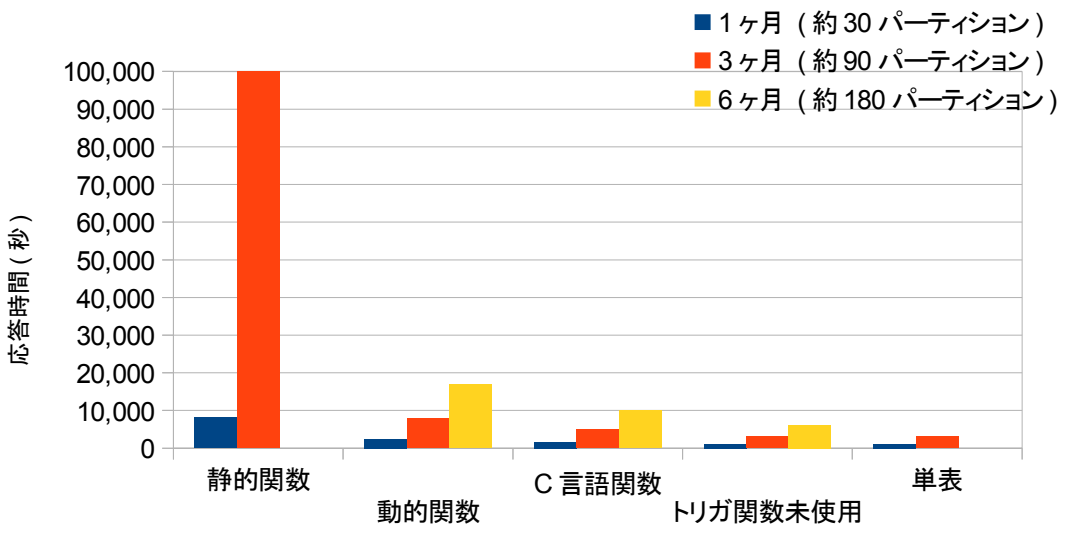

図 3.3: データ挿入応答時間の比較

トリガは挿入するデータのパーティションキーを 1 行ずつ検査するため、トリガを使用せずに直接パーティションに挿入するケースおよび、非パーティション表に挿入するケースがほぼ同等の応答時間となり最速でした。
トリガ関数を使用した場合で比較すると、その中では C 言語関数が最短であり、使用言語や実装の違いによりデータの挿入時間が大きく異なることがわかりました。

また、この挿入処理の応答時間は CPU 使用率と関係しています。次に示す CPU 使用率のグラフに表されているとおり、静的関数では IF 文の繰り返し処理に CPU 時間を消費し、挿入処理の応答時間が長くなっています。

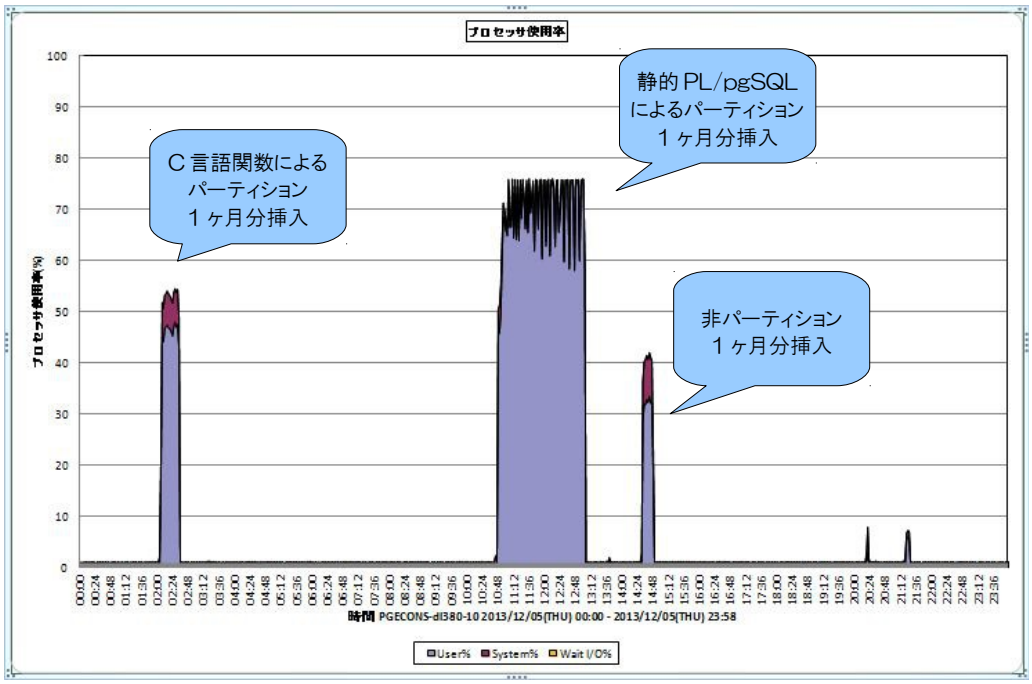

図 3.4 : CPU 使用率の推移 その 1

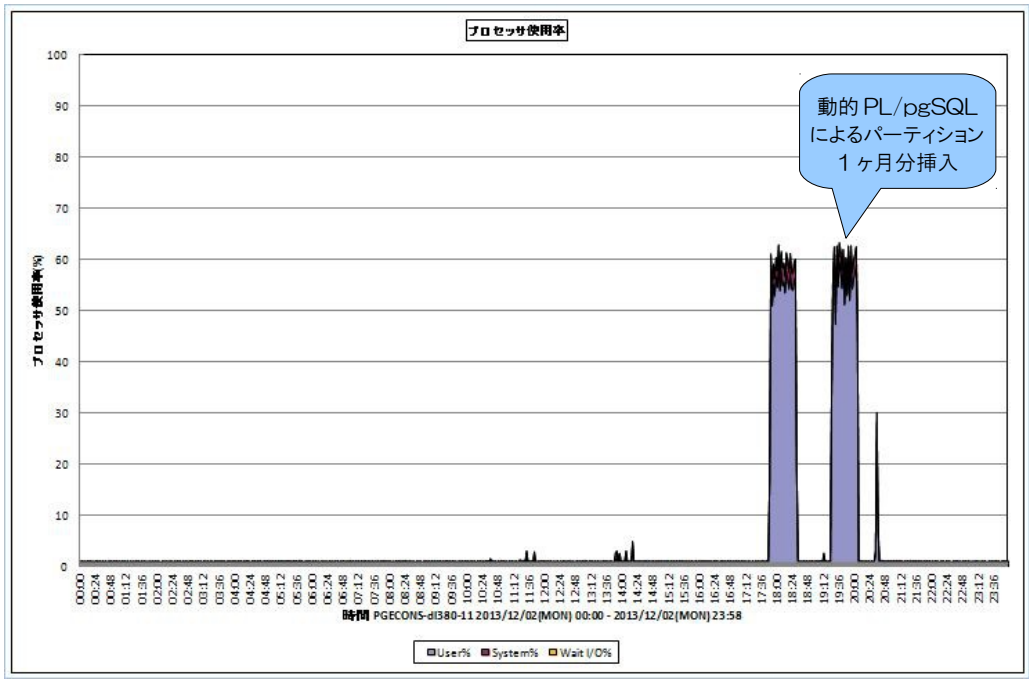

図 3.5 : CPU 使用率の推移 その 2

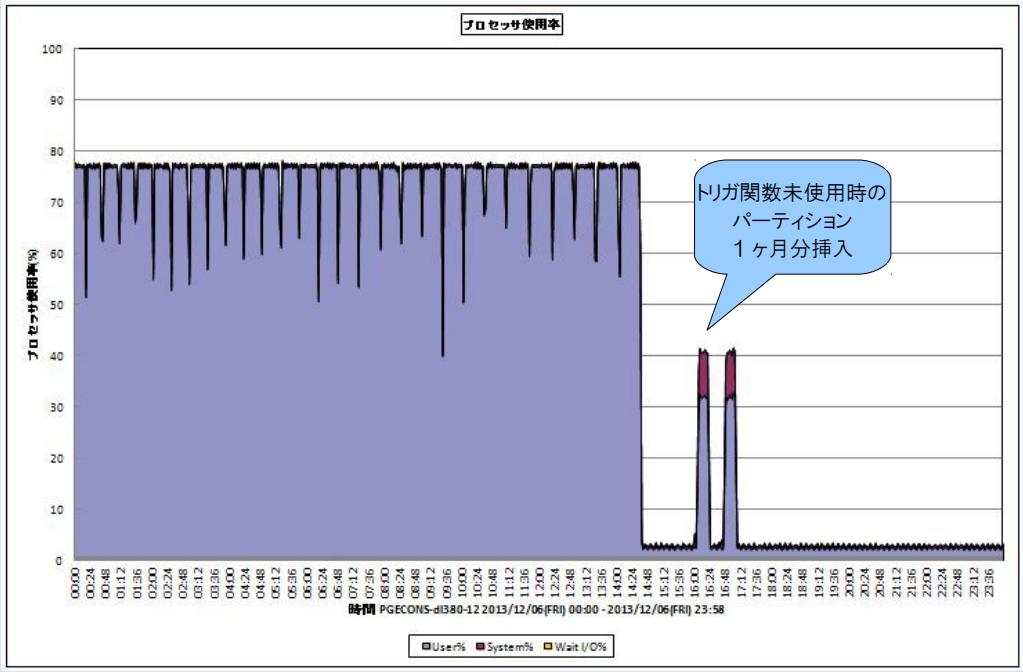

図 3.6：CPU 使用率の推移 その 3

参考までに検証に使用したトリガ関数を付録として紹介します。

- 添付ファイル１：静的関数（１ヶ月分）
- 添付ファイル２：動的関数
- 添付ファイル３：C 言語関数

3.5.2. 検索

単表、パーティションテーブルのそれぞれに対して、1 パーティション分のデータを集計する条件検索を実行し、その応答時間を比較しました。ログデータを 6 ヶ月、3 ヶ月、1 ヶ月と変動させた場合も傾向は変わらず、パーティションテーブルに対して処理を行った場合に単表と比較し大幅な応答時間の短縮がみられました。

以下に 6 ヶ月分のデータを対象に検索を行った場合の応答時間のグラフを記載します。

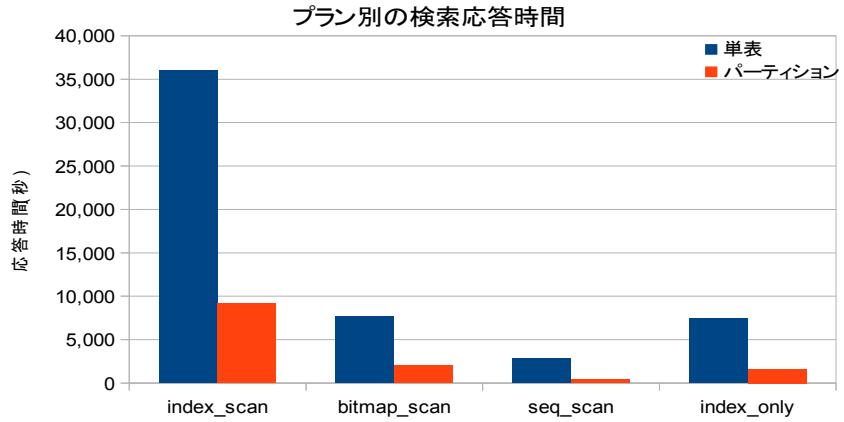

図 3.7 検索応答時間の比較(6month)

パーティションテーブルでは、6 ヶ月分の全データのうち、WHERE 句の条件に合致する範囲のみを子表から読み取るため、ディスク I/O の量が大幅に削減できたと考えられます。それに対して、単表の場合は WHERE 句の評価をするために 6 ヶ月分の全データを読み取らなければならず、大量のディスク I/O が発生しています。

ディスク I/O の推移を表したグラフでは、瞬間の I/O ブロック数は変わりありませんが、同等の I/O 量がクエリが完了するまで継続的に発生しており、読み込んだ総ブロック数では大きく異なる結果となったことがわかります。

この機能をパーティションプルーニングと呼び、パーティションが有効に作用したことが本検証から確認できました。

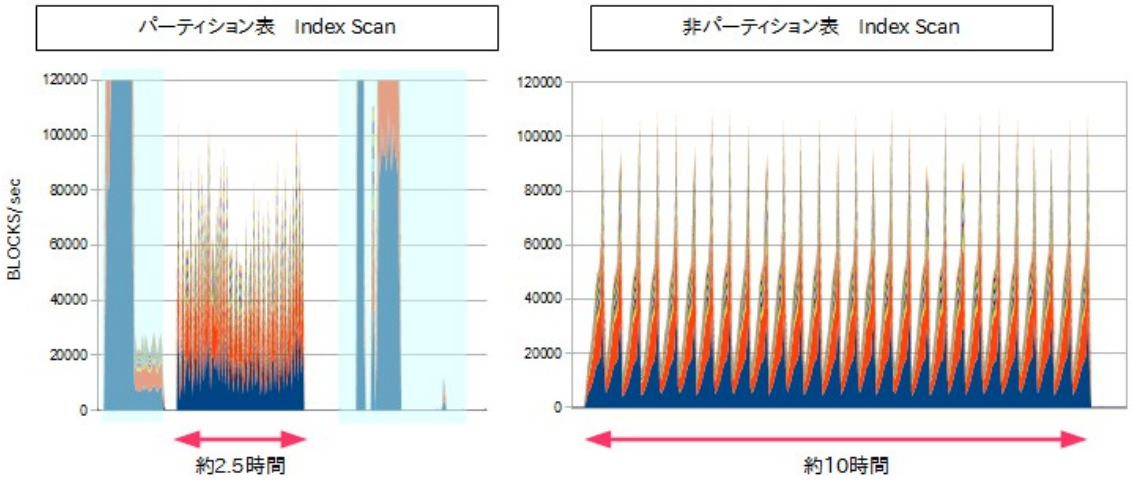

図 3.8 索引スキャン時のディスク I/O の比較

3.5.3. 運用

単表、パーティションテーブルのそれぞれに対して、1 パーティション分のデータを削除することを考え、単表では WHERE 句に該当する日付を指定した DELETE を実行し、パーティションテーブルでは最古のパーティション表を指定した TRUNCATE を実行しました。また、単表の場合、大量件数の更新となるため、VACUUM による不要領域の回収および ANALYZE による統計情報の収集は必ず実行しておくことが重要です。これらの応答時間を下記に記載します。

表 3.6: TRUNCATE/DELETE 応答時間の比較（秒）

	DELETE/TRUNCATE□ □	VACUUM□ □	ANALYZE□ □
□ □ □ □□ □□□ 6□ □	0.02	--	--
□ □ □ □□ □□□ 3□ □	0.02	--	--
□ □ □ □ □□ □□□ 6□ □	300	3,420	197

TRUNCATE では、データサイズを変え複数パターンで検証を行いましたが、いずれも数十ミリ秒で処理を完了しました。TRUNCATE はデータへのアクセスは行わないためマスタ表のデータサイズによらず一定の応答時間となったと考えられます。TRUNCATE では不要となった領域は即座に回収されるため、VACUUM や ANALYZE は効果がなく、本検証でも実施していません。

単表から該当の行を DELETE する処理ではディスク I/O が多発し、応答時間は 300 秒 (5 分) となりました。また、VACUUM は 3,420 秒 (57 分)を要し、この間もディスク I/O が多発していました。ANALYZE 処理は数分で完了していますが、ANALYZE では設定されたサンプリング数の範囲内でデータを取得し、統計情報とするため、ディスク I/O はほとんど発生しないためと考えられます。

3.6. 考察

データ挿入では、各パーティションテーブルを直接指定した挿入が最も高速である一方、トリガ関数を使用する場合にはパーティションの切り方を十分に考慮した C 言語関数により挿入の性能および処理データ件数の増加による処理時間の伸びも緩やかであることが確認できました。C 言語関数の作成は PL/pgSQL のトリガ関数の作成に比べて利用者への負担が大きいと考えられますが、同一レンジへの挿入が連続するテーブルであれば比較的汎用的なレンジパーティショニング用のトリガを作成することができます。また、ハッシュパーティショニングやリストパーティショニングでは再利用するプランのパーティション判定条件の一般化が難しいものの、個別の C 言語関数を準備することはそれほど難しくはなく、I/O 分散を目的としたパーティショニングでは有効であると推測できます。

検索処理ではレンジ数の増加に対してパーティション性能の不利が顕在化することが予想されましたが、一貫してパーティションテーブルが良好な測定結果となりました。今回は 1 日分のデータの集計を行うクエリを実行したため、非常に実 I/O 数が多く、単表に対する検索には不利な条件となりました。インデックスを使用したごく少数のレコードを抽出する場合には、パーティション数増加によるプラン生成が性能に与える影響が大きくなると推測されるため、実際に実行されるクエリとデータ量、メモリサイズ（キャッシュヒット率）などを想定して事前に検証を行うことを強く推奨します。

運用に関しては、今回のような一定期間のデータを対象とした処理を行うテーブルに対してはパーティションが非常に有効です。レコードの DELETE に対して TRUNCATE 処理は非常に軽く、VACUUM で回収した空き領域へのレコード挿入による断片化など、検索、更新性能への影響も低減させる可能性があります。

また、VACUUM はデータブロックについては更新された箇所のみを処理する一方、インデックスファイルは全体を参照する必要があるため、パーティショニングによる運用負荷は大幅に改善しています。また、今回の検証では行いませんでしたが、インデックスの再作成においても、負担を大幅に軽減する可能性があります。

トリガ関数作成の負担や更新におけるオーバヘッド、パーティション数増加による検索プラン作成のオーバヘッド増加、

パーティション運用による DBA の負担などのデメリットはあるものの、大容量データを運営する上でパーティションは非常に有効な機能であることが確認できました。
インデックスによる一本釣など軽量な（I/O 数の少ない）クエリ、パーティション数の制限、レンジ以外のパーティショニングについては今回は検証できませんでしたので、今後の機会に検証し、情報を公開したいと考えています。

4. ハードウェア活用(SSD)検証

4.1. 概要

近年のSSD[1]の低価格化・大容量化とエンタープライズモデルの広がりを背景に、データベースのストレージデバイスとしてSSDを採用することに注目が集まっています。

SSDの採用により、多くの大規模データベースシステムで性能のボトルネックになっているディスクI/Oの改善を見込むことができます。

一般的にはアプリケーションの修正やDBMSのパラメータチューニング等によって性能改善を行いますが、その場合、試行も含めた作業期間がある程度必要になります。SSD採用は、基本的にディスクの置き換えのみで作業が完了するため、短期間で性能向上を実現できます。これはSSD採用による性能改善の利点の一つです。

今回の検証では、PostgreSQLのストレージデバイスにSSDを採用した場合、実際にはどのような性能向上効果を示すのかを検証します。

4.2. 検証目的

データベースのストレージデバイスには、通常HDDが使われています。

HDDは磁気ディスクにデータを記録しており、磁気ヘッドがデータを読み書きします。

データアクセスにかかる時間は、磁気ヘッドを目的のトラックへ動かす時間(シークタイム)と磁気ディスクを目的の位置まで回転させる時間(サーチタイム)とデータの転送時間の合計になります。

特に磁気ヘッドを頻繁に動かすランダムアクセスの場合、シーケンシャルアクセスと比較してデータアクセス時間は長くなります。

一方、SSDはフラッシュメモリにデータを記録しています。データの読み書きに磁気ディスクの回転や磁気ヘッドの移動などの機械的動作は不要です。そのため、HDDよりも高速にデータアクセスすることが可能です。ただし、容量単価が高く、また、書き込み回数に上限がある等の欠点も存在します。

SSDの接続インターフェースに採用されている規格には、SAS、SATAとPCIeの大きく3つの規格が存在します。

現在の主流であるSASやSATAはHDDの性能を想定した規格であるため、将来的にSSDの性能が向上してSASやSATAの転送速度がボトルネックになると懸念されています。

そこで注目されている規格がPCIeです。PCIeバスとSSDを直接接続することによって、遅延が大幅に削減され、SSDの性能が向上します。さらにPCIeはマルチレーン化することで転送速度を上げることが可能になるため、SASやSATAのように転送速度がボトルネックになる懸念はありません。

本検証では、従来規格のSATA SSDと注目されている規格であるPCIe SSDの2種類のSSDを使用することとします。

SSDがHDDよりも高速にデータアクセスができることは一般的によく知られていますが、実際にそれが事実かどうか確かめるために事前検証を行いました。

事前検証の方法として、今回の検証に使用する2台のマシン(PCIe SSD搭載マシンとSATA SSD搭載マシン)それぞれのHDDとSSDのI/O性能を計測することとしました。

検証ツールはfioを使用し、オプションには以下を指定しました。

```
size=1G
direct=1
blocksize=8k
numjobs=4
group_reporting
runtime=60
```

3回計測した帯域幅の中央値を採用し、PCIe SSD搭載マシンとSATA SSD搭載マシンそれぞれのHDDとSSDのディスク帯域幅を比較します。

1 本章中で「SSD」と記載した場合、記憶領域にフラッシュメモリを使用するSSD(Flash SSD)を指すこととします

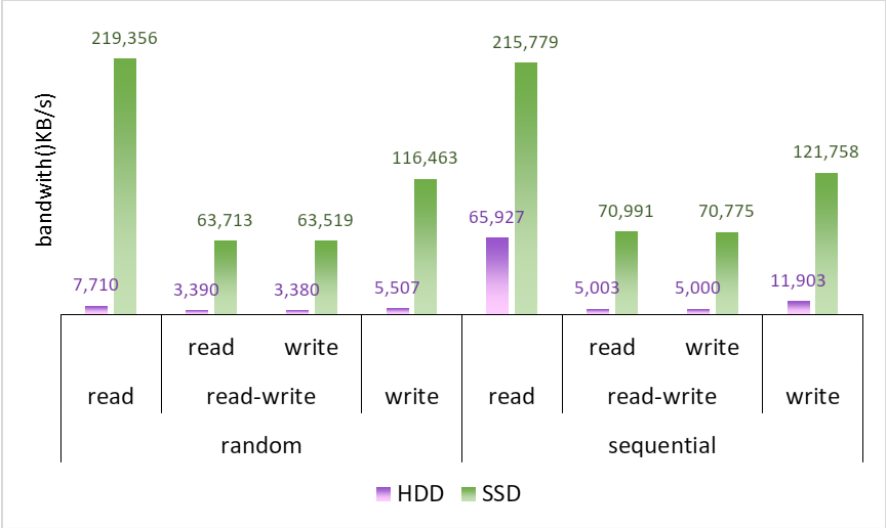

図 4.1: I/O 性能(PCIe SSD 搭載マシン)

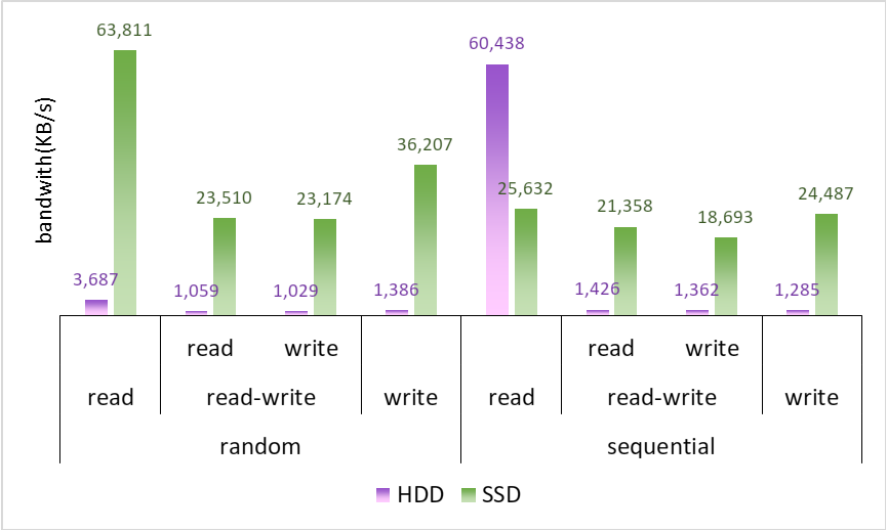

図 4.2: I/O 性能(SATA SSD 搭載マシン)

図 4.1 と図 4.2 から、SATA SSD 搭載マシンでのシーケンシャル読み込みを除くすべての項目で、SSD が HDD を大きく上回る I/O 性能があることがわかります。
ランダムアクセス時の性能差が特に大きくなっており、PCIe SSD 搭載マシンでは、SSD のランダム読み込みの帯域幅が HDD 比で約 28.5 倍、ランダム書き込みの帯域幅が約 21.1 倍でした(図 4.1)。また、SATA SSD 搭載マシンでは、ランダム読み込みの帯域幅が HDD 比で約 17.3 倍、ランダム書き込みの帯域幅が約 26.2 倍でした(図 4.2)。

PCIe SSD と SATA SSD を比較すると、全体的に PCIe SSD の性能が高いことが確認できます。
また、PCIe SSD ではランダム読み込みとシーケンシャル読み込みの性能にほとんど差はありませんでしたが、SATA SSD ではシーケンシャル読み込みでランダム読み込みの半分以下の帯域幅となっていました。

今回は、このようにデータアクセス性能の高い SSD を PostgreSQL のストレージデバイスに採用した場合、どのような性能改善効果が得られるかを検証してみることにしました。
性能検証の観点として、以下の 3 つを前提条件としました。

- 比較観点として、ストレージデバイスに HDD を採用した場合と SSD を採用した場合の性能差を検証
  - 一般的なストレージデバイスである HDD を SSD 性能検証の比較対象に設定
- 比較観点として、接続インターフェースや性能の異なる SSD を採用した場合の性能向上の度合を検証

  - SSD にも接続インタフェースの違いなど、様々な種類の SSD が存在するが、それぞれの SSD で性能改善効果に違いが出るかどうか検証
  - PCIe SSD、SATA SSD
- エンタープライズ向けのデータベースシステムを想定
  - データサイズが数百 GB 程度になるよう、データモデルを設計

また、前提条件を踏まえ、以下の2点を SSD 性能測定の測定項目としました。

- ランダムアクセス性能の高い SSD を採用した場合の効果を確認する
  - ディスクにランダムアクセスする検索方式を優先的に採用
  - インデックススキャン、インデックスオンリースキャンの検索方式の場合の性能を測定
- データベース資源をすべて SSD に配置するとストレージにかかるハード費用が増大するため、本検証で最も効果的な資源配置のパターンを発見
  - SSD に配置するデータベース資源のパターンを規定し、配置パターン別に比較
  - テーブルのみ配置、インデックスのみ配置、WAL のみ配置等
  - インデックススキャンの性能測定と併せて検証

4.3. 検証構成

データベースサーバとして、PostgreSQL 9.3.2 を搭載するサーバを 2 台用意しました。
データベースのストレージデバイスとして、異なる接続形態の SSD と HDD をそれぞれのデータベースサーバに搭載しています。
主なスペックと構成は以下のとおりです。

表 4.1: 検証構成

機器	項目	仕様
PCIe SSD 搭載マシン	CPU	インテル® Xeon® プロセッサ E5-2690 (2.90GHz/8 コア/20MB) x 2
	搭載メモリ	192GB
	DB ストレージ用 HDD	SAN ディスクアレイ装置(iSCSI 接続) 600GB(10krpm) x 5, RAID5
	DB ストレージ用 SSD	PCIe SSD(MLC) 1.2TB x 1
	OS	Red Hat Enterprise Linux 6.2 (for Intel64)
	DBMS	PostgreSQL 9.3.2
	検証ツール	PostgreSQL 9.3.2 同梱 pgbench[2]
SATA SSD 搭載マシン	CPU	インテル® Xeon® プロセッサ E5-2697v2 (2.70GHz/12 コア/30MB) x 2
	搭載メモリ	48GB
	DB ストレージ用 HDD	SAS 内蔵ディスク 600GB(10krpm) x 2, RAID1
	DB ストレージ用 SSD	SATA SSD(MLC) 200GB x 2, RAID1
	OS	Red Hat Enterprise Linux 6.2 (for Intel64)
	DBMS	PostgreSQL 9.3.2
	検証ツール	PostgreSQL 9.3.2 同梱 pgbench

2 pgbench の詳細は 2.2 を参照してください

4.4. 検証方法

本検証では、PostgreSQL に標準で付属しているベンチマークツールである pgbench を使用して、以下の 2 つの検証を行いました。

- SSD 配置パターン別インデックススキャン性能検証
- インデックスオンリースキャン性能検証

本検証の検証方法の概要は表 4.2 の通りです。

表 4.2: 検証方法

項目	SSD 配置パターン別インデックススキャン性能検証	インデックスオンリースキャン性能検証
検証環境	・PCIe SSD 搭載マシン ・SATA SSD 搭載マシン	同左
検証ツール	PostgreSQL 同梱ツールである pgbench を使用	同左
検証項目	・TPC-B ベースのデフォルトスクリプトを更新系・参照系ともに実行 ・前提条件はすべて同じ ・データディレクトリおよびデータベースオブジェクトの SSD 配置パターン別に TPS(Transaction Per Second, 1 秒間のトランザクション実行回数)を測定	・インデックスオンリースキャン用の独自スクリプトを実行 ・インデックスオンリースキャンが成功する条件下と失敗する条件下それぞれで測定(*) ・データディレクトリおよびデータベースオブジェクトの SSD 配置パターン別に TPS を測定
SSD 配置パターン	以下の5パターンそれぞれ測定 ・データディレクトリ配下およびデータベースオブジェクトをすべて HDD に配置 ・WAL のみ SSD に配置、それ以外をすべて HDD に配置 ・インデックスのみ SSD に配置、それ以外をすべて HDD に配置 ・テーブルのみ SSD に配置、それ以外をすべて HDD に配置 ・データディレクトリ配下およびデータベースオブジェクトをすべて SSD に配置	以下の2パターンそれぞれ測定 ・データディレクトリ配下およびデータベースオブジェクトをすべて HDD に配置 ・データディレクトリ配下およびデータベースオブジェクトをすべて SSD に配置

(*) テーブル更新後に VACUUM が実行されず、Visibility Map のビット情報がクリアされている場合、インデックスオンリースキャンでもテーブルからデータを取得します。これを本検証ではインデックスオンリースキャンの失敗と呼びます。

本検証の検証方法の詳細を以下に記載します。

4.4.1. 環境構築

postgresql.conf を修正し、検証用のデータベースパラメータを設定しました。

PCIe SSD のキャッシュが十分に働くためのメモリを確保するため、PCIe SSD 搭載マシンでは共有メモリを搭載メモリの 1/4(48GB)に設定し、SATA SSD 搭載マシンも同じく搭載メモリの 1/4(12GB)を共有メモリに配分しました。リソース使用に関する下記パラメータは PCIe SSD 搭載マシンと SATA SSD 搭載マシンで、共有メモリと同じ比率(PCIe SSD 搭載マシン側は SATA SSD 搭載マシンの設定値の4倍の値)になるように設定しました。

- temp_buffers
- work_mem
- maintenance_work_mem
- wal_buffers
- effective_cache_size

また、測定中にチェックポイントが動作して性能に影響を与えないように、checkpoint_segments の値を PCIe SSD 搭載マシンで 4000 に、SATA SSD 搭載マシンで 1000 に変更しました。

以下に、デフォルトの設定から変更したパラメータを PCIe SSD 搭載マシン、SATA SSD 搭載マシンそれぞれ記載します。

表 4.3: PCIe SSD 搭載マシンのデータベースパラメータ

パラメータ名	設定値	内容
shared_buffers	48GB	物理メモリの 25%を目安として設定値を決定。
temp_buffers	32MB	一時領域として十分な領域を確保。
work_mem	32MB	ソートワークファイルを使用しないよう、十分な領域を確保。
shared_preload_libraries	'pg_stat_statements'	検証中のデータベース情報を取得するため、'pg_stat_statements'を指定
wal_buffers	32MB	トランザクション処理中に頻繁に WAL ファイル出力が行われないよう、十分な領域を確保。
checkpoint_segments	4000	大量更新によって、CHECKPOINT が発生しないよう、セグメント数を増やす。
effective_cache_size	48GB	物理メモリの 25%を目安として設定値を決定。
maintenance_work_mem	20GB	事前のデータ作成で CREATE INDEX が高速動作するよう、十分な領域を確保。
autovacuum	off	自動 VACUUM による性能劣化を防ぐため、自動 VACUUM を無効化する。

表 4.4: SATA SSD 搭載マシンのデータベースパラメータ

パラメータ名	設定値	内容
shared_buffers	12GB	物理メモリの 25%を目安として設定値を決定。 （PCIe SSD 搭載マシン設定値の 1/4）
temp_buffers	8MB	一時領域として十分な領域を確保。 （PCIe SSD 搭載マシン設定値の 1/4）
work_mem	8MB	ソートワークファイルを使用しないよう、十分な領域を確保。 （PCIe SSD 搭載マシン設定値の 1/4）
shared_preload_libraries	'pg_stat_statements'	検証中のデータベース情報を取得するため、'pg_stat_statements'を指定
wal_buffers	8MB	トランザクション処理中に頻繁に WAL ファイル出力が行われないよう、十分な領域を確保。 （PCIe SSD 搭載マシン設定値の 1/4）
checkpoint_segments	1000	大量更新によって、CHECKPOINT が発生しないよう、セグメント数を増やす。 （PCIe SSD 搭載マシン設定値の 1/4）
effective_cache_size	12GB	物理メモリの 25%を目安として設定値を決定。 （PCIe SSD 搭載マシン設定値の 1/4）
maintenance_work_mem	5GB	事前のデータ作成で CREATE INDEX が高速動作するよう、十分な領域を確保。 （PCIe SSD 搭載マシン設定値の 1/4）
autovacuum	off	自動 VACUUM による性能劣化を防ぐため、自動 VACUUM を無効化する。

4.4.2. データベース作成

pgbench のコマンドを使用してデータベースを作成しました。

共有メモリに対するデータサイズの割合を同一にするため、スケールファクタの値は PCIe SSD 搭載マシンで 26,000（260 億レコード/約 390GB）、SATA SSD 搭載マシンで 6,400（64 億レコード/約 96GB）としました。

また、配置パターン別性能検証のために、インデックスとデータをそれぞれ HDD 内の個別のテーブル空間に配置しました。

以下に、実行したコマンドを記載します。

```
$ pgbench -i -q -s スケールファクタ -U guest --index-tablespace=ts_index --tablespace=ts_table postgres
```

4.4.3. SSD 配置パターン別インデックススキャン性能検証の測定

pgbench のコマンドを使用して測定を行いました。
スクリプトは pgbench デフォルトのものを使用し、pgbench のオプションには、以下を指定しました。

- -S
  - 参照系の場合は pgbench コマンドに-S オプションを指定し、更新系の場合は-S オプションを指定せずに実行
- -c コネクション数
  - pgbench のコネクション数は測定マシンのコア数と同じ値を指定
- -j スレッド数
  - スレッド数はコア数の半分を指定
- -T 実行時間
  - 検証期間を短縮するために pgbench の実行時間が 10 分を超えないよう-T 600 を指定
- -n
  - pgbench 実行時に VACUUM が動作しないよう、-n オプションを指定
  - 本来、pgbench 実行時の VACUUM は性能劣化を防ぐために必要ですが、本測定では更新系の pgbench コマンド実行ごとにすべてのデータディレクトリ配下のファイルをデータベース作成時のものに置き換えることで代替しました

以下に、更新系で実行したコマンド例を記載します。

```
$ pgbench -n -c コネクション数 -j スレッド数 -T 600 postgres
```

以下に、参照系で実行したコマンド例を記載します。

```
$ pgbench -S -n -c コネクション数 -j スレッド数 -T 600 postgres
```

SSD 配置パターンは、以下のように実現しました。

- データディレクトリ配下およびデータベースオブジェクトをすべて HDD に配置
  - データディレクトリを HDD に配置
- WAL のみ SSD に配置、それ以外をすべて HDD に配置
  - データディレクトリを HDD に配置
  - データディレクトリ配下の pg_xlog をシンボリックリンクに置き換え
  - 上記シンボリックリンクのリンク先（SSD 内のディレクトリ）に WAL を配置
- インデックスのみ SSD に配置、それ以外をすべて HDD に配置
  - データディレクトリを HDD に配置
  - インデックス用テーブル空間の LOCATION ディレクトリをシンボリックリンクに置き換え
  - 上記シンボリックリンクのリンク先（SSD 内のディレクトリ）にインデックスファイルを配置
- テーブルのみ SSD に配置、それ以外をすべて HDD に配置
  - データディレクトリを HDD に配置
  - テーブル用テーブル空間の LOCATION ディレクトリをシンボリックリンクに置き換え
  - 上記シンボリックリンクのリンク先（SSD 内のディレクトリ）にテーブルファイルを配置
- データディレクトリ配下およびデータベースオブジェクトをすべて SSD に配置
  - データディレクトリを SSD に配置
  - インデックス用テーブル空間の LOCATION ディレクトリをシンボリックリンクに置き換え
  - 上記シンボリックリンクのリンク先（SSD 内のディレクトリ）にインデックスファイルを配置
  - インデックス用テーブル空間の LOCATION ディレクトリをシンボリックリンクに置き換え
  - 上記シンボリックリンクのリンク先（SSD 内のディレクトリ）にインデックスファイルを配置

SSD 配置パターン毎に 3 回 pgbench を実行し、その中央値を TPS の値として採用しました。

4.4.4. インデックスオンリースキャン性能検証の測定

pgbench のコマンドを使用して測定を行いました。
インデックスオンリースキャンの完全一致検索を行う、以下の独自スクリプトを使用しました。

インデックスオンリースキャン完全一致検索スクリプト
（pgbench_accounts テーブルの 1 レコードをランダムに選んで完全一致検索）

```
¥setrandom aid 1 100000 * スケールファクタ
SELECT aid FROM pgbench_accounts WHERE aid = :aid;
```

pgbench のオプションとして、独自スクリプトを実行させる-f オプションを指定しました。それ以外のオプションは 4.4.3.と同様です。
以下に、実行したコマンド例を記載します。

```
$ pgbench -n -c コネクション数 -j スレッド数 -T 600 -f スクリプトファイル名 postgres
```

SSD 配置パターンの実現方法は 4.4.3.と同様です。
本測定では、すべて HDD に配置する場合とすべて SSD に配置する場合の 2 パターンで、インデックスオンリースキャン成功／失敗時の TPS を、PCIe SSD 搭載マシンおよび SATA SSD 搭載マシン上でそれぞれ測定しました。
検証期間の関係上、SSD 配置パターン毎に 1 回 pgbench を実行した値を TPS の値として採用しました。

また、データベースのキャッシュをクリアしてするため、以下を1回の測定ごとに実行しました。

- PostgreSQL プロセスを再起動
- OS キャッシュクリア
  - 以下のコマンドを実行

```
# free
# echo 3 > /proc/sys/vm/drop_caches
# free
```

インデックスオンリースキャンを失敗させる場合、測定前に pgbench_accounts を更新しました。
pgbench_accounts のすべてのページに更新が書き込まれる、以下の pl/pgsql ファンクションを実行することで pgbench_accounts の更新を実現しました。

```
CREATE OR REPLACE FUNCTION huge_update() RETURNS boolean LANGUAGE plpgsql AS $$
BEGIN
  FOR i IN 1.. pgbench_accounts のページ数
  LOOP
    UPDATE pgbench_accounts SET abalance = abalance + 100 WHERE aid = i * ブロック内のタプルインデックス数;
  END LOOP;
  RETURN;
END
$$;
```

pgbench_accounts のページ数は以下の SQL 実行で確認しました。この場合、42622951 が pgbench_accounts のページ数になります。

```
postgres=# select relpages from pg_class where relname = 'pgbench_accounts';
 relpages
----------
 42622951
(1 row)
```

ブロック内のタプルインデックス数は以下の SQL 実行で確認しました。この場合、61 がブロック内のタプルインデックス数になります。

```
postgres=# select ctid from pgbench_accounts where aid = 122;
```

```
 ctid
--------
 (1,61)
(1 row)
postgres=# select ctid from pgbench_accounts where aid = 123;
 ctid
-------
 (2,1)
(1 row)
```

インデックスオンリースキャン失敗のためにデータベースを更新するので、不要領域による性能劣化が出ないよう、インデックスオンリースキャン失敗の pgbench 実行毎にデータディレクトリ配下のファイルをデータベース作成時のものに置き換えました。インデックスオンリースキャン成功時の pgbench 実行ではデータベースの更新が発生しないため、データディレクトリ配下の置き換えは行いませんでした。

4.5. 検証結果

本検証の結果を以下に記載します。

4.5.1. SSD 配置パターン別インデックススキャン性能検証の結果

表 4.5: SSD 配置パターン別インデックススキャン性能(PCIe SSD 搭載マシン)

pgbench	SSD 配置パターン	TPS (1 回目)	TPS (2 回目)	TPS (3 回目)	TPS (中央値)
更新系	all on HDD	210.5	129.3	198.5	198.5
	WAL on SSD	281.1	296.3	253.9	281.1
	index on SSD	270.1	352.9	349.2	349.2
	table on SSD	338.2	439.1	332.9	338.2
	all on SSD	6160.4	5923.7	5996.6	5996.6
参照系	all on HDD	484.3	449	447.1	449
	WAL on SSD	628.9	661	683.5	661
	index on SSD	976.8	970.9	960.7	970.9
	table on SSD	1337.8	1493.6	2217.2	1493.6
	all on SSD	47334.3	49934.1	50511.2	49934.1

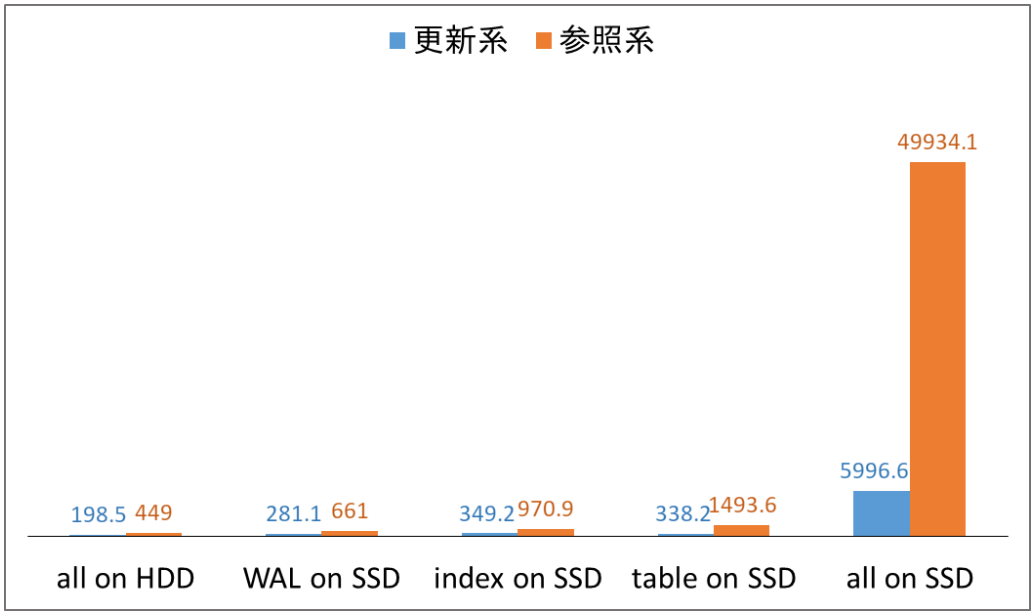

図 4.3: SSD 配置パターン別インデックススキャン性能(PCIe SSD 搭載マシン)

表 4.6: SSD 配置パターン別インデックススキャン性能（SATA SSD 搭載マシン）

pgbench	SSD 配置パターン	TPS（1 回目）	TPS（2 回目）	TPS（3 回目）	TPS（中央値）
更新系	all on HDD	101.2	108.3	118.1	108.3
	WAL on SSD	286.6	289.9	289.9	289.9
	index on SSD	224.1	204.4	194.9	204.4
	table on SSD	225	240.2	227.5	227.5
	all on SSD	1509.6	1413.9	1141.8	1413.9
参照系	all on HDD	484.3	546.7	513.1	513.1
	WAL on SSD	545.8	517.1	545.6	545.6
	index on SSD	849.3	931.7	1139.8	931.7
	table on SSD	1995.7	1459.7	1486.9	1486.9
	all on SSD	23134.2	22987.7	23129.9	23129.9

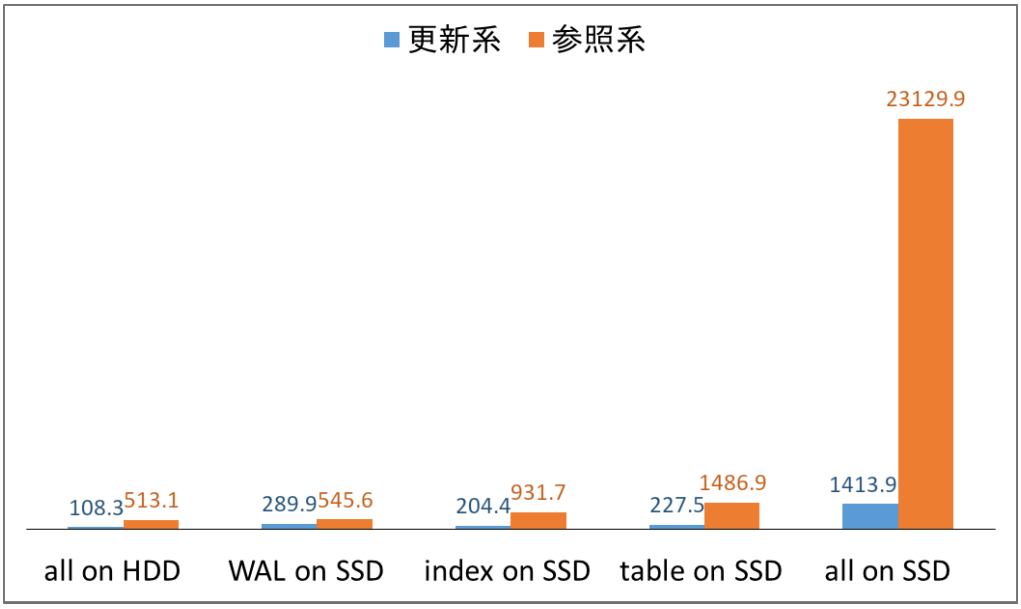

図 4.4: SSD 配置パターン別インデックススキャン性能（SATA SSD 搭載マシン）

4.5.2. インデックスオンリースキャン性能検証の結果

表 4.7: インデックスオンリースキャン性能

SSD 配置パターン	PCIe SSD 搭載マシン		SATA SSD 搭載マシン	
	TPS（index only scan 成功）	TPS（index only scan 失敗）	TPS（index only scan 成功）	TPS（index only scan 失敗）
all on HDD	2501.0	614.2(*)	2055.0	157.4
all on SSD	212393.6	25106.4	244089.4	2567.3

(*) 測定前の pgbench_accounts 更新が想定通り実行できなかったので、正確な値ではありません

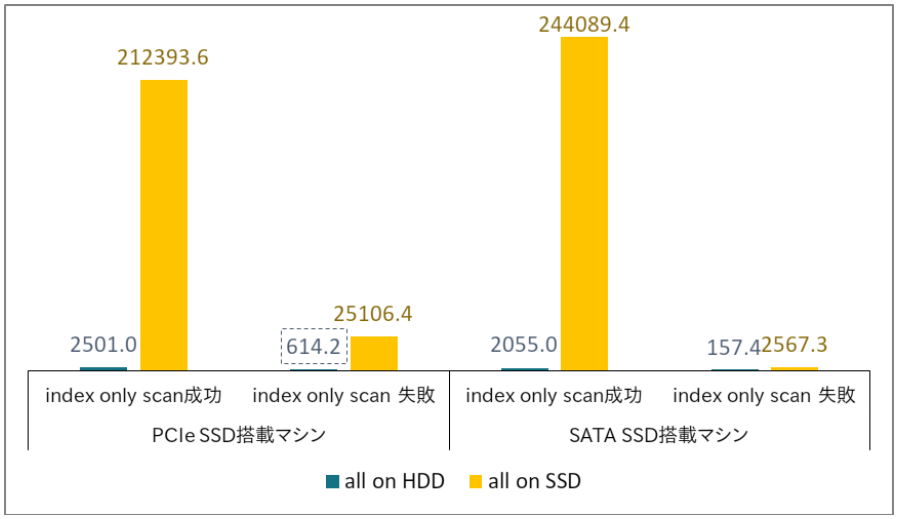

図 4.5: インデックスオンリースキャン性能

4.6. 考察

本検証の結果に関する考察を以下に記載します。

4.6.1. SSD 配置パターン別インデックススキャン性能検証の考察

検証結果（表 4.8、表 4.9 および図 4.3、図 4.4）から、データディレクトリおよびデータベースオブジェクトの配置先をすべて SSD にした場合、大きな性能向上が得られました。

配置先をすべて HDD にした場合と比較して、PCIe SSD 搭載マシンでは更新系の TPS 値が約 30 倍、参照系の TPS 値が約 111 倍になり、SATA SSD 搭載マシンでは更新系の TPS 値が約 13 倍、参照系の TPS 値が約 45 倍になりました。

これは、配置先をすべて HDD にした場合は HDD の性能限界が発生し、配置先をすべて SSD にした場合は SSD の性能限界が発生したため、TPS の値が I/O 性能に強く影響されたためと考えられます。

WAL のみ、またはインデックス、テーブルのみ SSD に配置した場合でも数倍程度の性能向上が見られますが、すべて SSD に配置した場合の TPS の値とは開きがあります。

これは、WAL、インデックス、データそれぞれがディスク I/O ネックの要因になっているため、どれか一つを SSD に配置したとしても、HDD に配置している別のファイルの I/O によって性能向上が限定的になったためと考えられます。

つまり、本測定で計測したデータベース性能は、データベースサーバの I/O 性能が強く影響した値ということになります。

この考察が正しいことを裏付けるために、データベース性能と性能測定中のデータベースサーバの I/O 性能の関係を確認してみました。

pgbench 実行中の PCIe SSD 搭載マシン上の HDD および SSD を測定対象とし、iostat コマンドによって出力された情報から、以下の2項目の平均値を求めました。

- 読み込み性能(KB/s)
  - 512 バイト（0.5KB, ファイルシステムのブロックサイズ） × iostat で出力した Blk_read/s（1 秒間のブロック読み込み量）
- 書き込み性能(KB/s)
  - 512 バイト（0.5KB, ファイルシステムのブロックサイズ） × iostat で出力した Blk_wrtn/s（1 秒間のブロック書き込み量）

HDD と SSD それぞれの読み込み性能および書き込み性能を合計することで、データベースサーバ全体の I/O 性能を算出します。

すべて HDD に配置した場合の I/O 性能および TPS 値を 1 として、I/O 性能が SSD 配置パターン別にどのような比率で変化しているかを、データベース性能の変化とあわせて確認します。

また、ここでは pgbench を 3 回実行した中で、TPS 中央値を出した際の I/O 性能および TPS 値を採用しています。

更新系検証時の結果は以下のとおりです。

表 4.8: pgbench 更新系検証時の I/O 性能およびデータベース性能（PCIe SSD 搭載マシン）

SSD 配置パターン	iostat 情報				TPS（カッコ内、全 HDD 配置比）
	項目	HDD	SSD	HDD+SSD	
all on HDD	読み込み性能 [KB/s]	3657.71	0.00	3657.71	198.5 (1)
	（全 HDD 配置比）	(1)	(0)	(1)	
	書き込み性能 [KB/s]	8119.80	0.00	8119.80	
	（全 HDD 配置比）	(1)	(0)	(1)	
WAL on SSD	読み込み性能 [KB/s]	5272.56	0.07	5272.63	253.9 (1.42)
	（全 HDD 配置比）	(1.44)	(2.04E-05)	(1.44)	
	書き込み性能 [KB/s]	1300.58	11124.14	12424.71	
	（全 HDD 配置比）	(0.16)	(1.37)	(1.53)	
index on SSD	読み込み性能 [KB/s]	2922.19	46.42	2968.61	349.2 (1.76)
	（全 HDD 配置比）	(0.80)	(0.01)	(0.81)	
	書き込み性能 [KB/s]	13236.27	3110.75	16347.02	
	（全 HDD 配置比）	(1.63)	(0.38)	(2.01)	
table on SSD	読み込み性能 [KB/s]	3850.05	3023.57	6873.61	332.9 (1.70)
	（全 HDD 配置比）	(1.05)	(0.83)	(1.88)	
	書き込み性能 [KB/s]	13308.83	3029.93	16338.76	
	（全 HDD 配置比）	(1.64)	(0.37)	(2.01)	
all on SSD	読み込み性能 [KB/s]	0.03	143504.36	143504.39	5996.6 (30.21)
	（全 HDD 配置比）	(7.41E-06)	(39.23)	(39.23)	
	書き込み性能 [KB/s]	0.00	230956.73	230956.73	
	（全 HDD 配置比）	(0)	(28.44)	(28.44)	

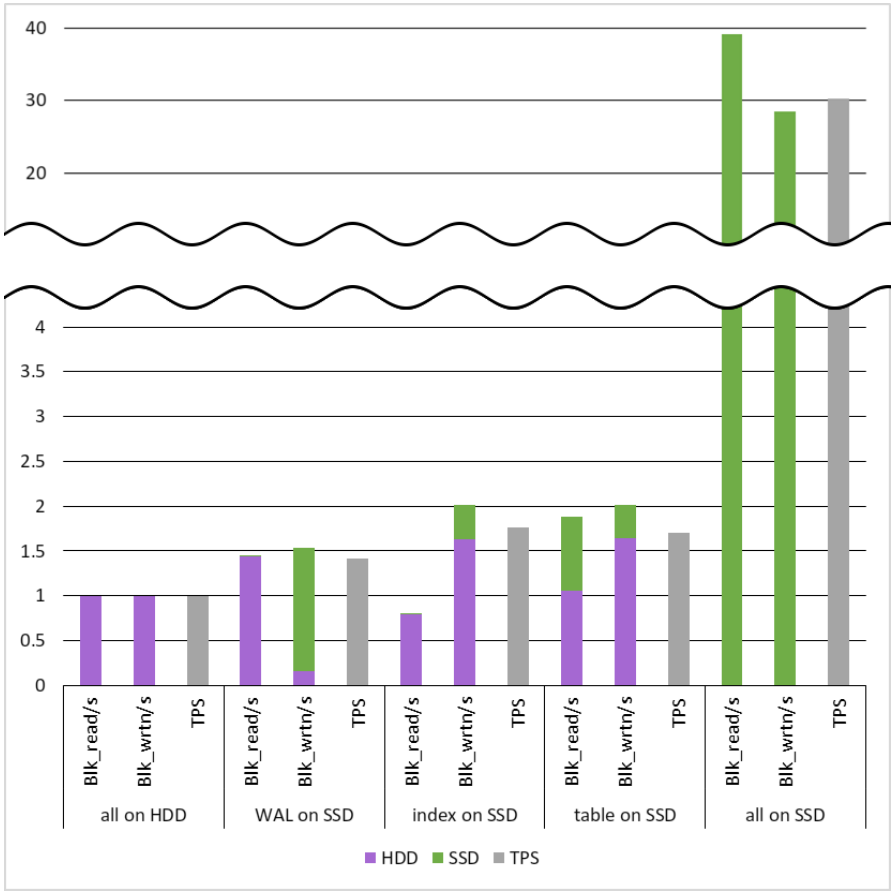

図 4.6: pgbench 更新系検証時の I/O 性能およびデータベース性能の全 HDD 配置対比（PCIe SSD 搭載マシン）

表 4.8 と図 4.6 から、更新系検証時のデータベース性能の増加比率は、HDD と SSD を合計した書き込み性能の増加比率とほぼ同じことがわかります。これは、更新系検証では書き込み性能がボトルネックになっていることを示しています。

また、WAL やデータベースオブジェクト等、一部を SSD に配置した場合でも、HDD の I/O 性能が全体の I/O 性能の大きな割合を占めていることがわかります。これは SSD の I/O 性能が HDD の I/O 性能に影響を受け、十分な値を出せなかったためと考えられます。

iostat -x コマンドで検証中の SSD、HDD それぞれの%util（ディスクビジー率）を測定した結果を実際に確認したところ、一部を SSD に配置した場合では HDD のディスクビジー率が 90%後半を示し（ディスクビジー状態となっている）、対して SSD のディスクビジー率は数%程度でした。すべてを SSD に配置した場合は SSD がディスクビジー状態となっていました。

参照系検証時の結果は以下のとおりです。

表 4.9: pgbench 参照系検証時の I/O 性能およびデータベース性能（PCIe SSD 搭載マシン）

SSD 配置パターン	iostat 情報				TPS（カッコ内、全 HDD 配置比）
	項目	HDD	SSD	HDD+SSD	
all on HDD	読み込み性能 [KB/s]	9669.19	0.00	9669.19	449 (1)
	（全 HDD 配置比）	(1)	(0)	(1)	
	書き込み性能 [KB/s]	3.86	0.00	3.86	
	（全 HDD 配置比）	(1)	(0)	(1)	
WAL on SSD	読み込み性能 [KB/s]	11416.09	0.00	11416.09	

SSD 配置パターン	iostat 情報				TPS (カッコ内、全 HDD 配置比)
	項目	HDD	SSD	HDD+SSD	
	(全 HDD 配置比)	(1.18)	(0)	(1.18)	661 (1.47)
	書き込み性能 [KB/s]	1.60	208.98	210.58	
	(全 HDD 配置比)	(0.41)	(54.08)	(54.49)	
index on SSD	読み込み性能 [KB/s]	10716.72	15050.75	25767.47	970.9 (2.16)
	(全 HDD 配置比)	(1.11)	(1.56)	(2.66)	
	書き込み性能 [KB/s]	3.69	0.00	3.69	
	(全 HDD 配置比)	(0.96)	(0)	(0.96)	
table on SSD	読み込み性能 [KB/s]	21131.29	15208.68	36339.97	1493.6 (3.33)
	(全 HDD 配置比)	(2.19)	(1.57)	(3.76)	
	書き込み性能 [KB/s]	0.74	0.00	0.74	
	(全 HDD 配置比)	(0.19)	(0)	(0.19)	
all on SSD	読み込み性能 [KB/s]	0.00	1116246.73	1116246.73	49934.1 (111.21)
	(全 HDD 配置比)	(0)	(115.44)	(115.44)	
	書き込み性能 [KB/s]	0.00	6.36	6.36	
	(全 HDD 配置比)	(0)	(1.65)	(1.65)	

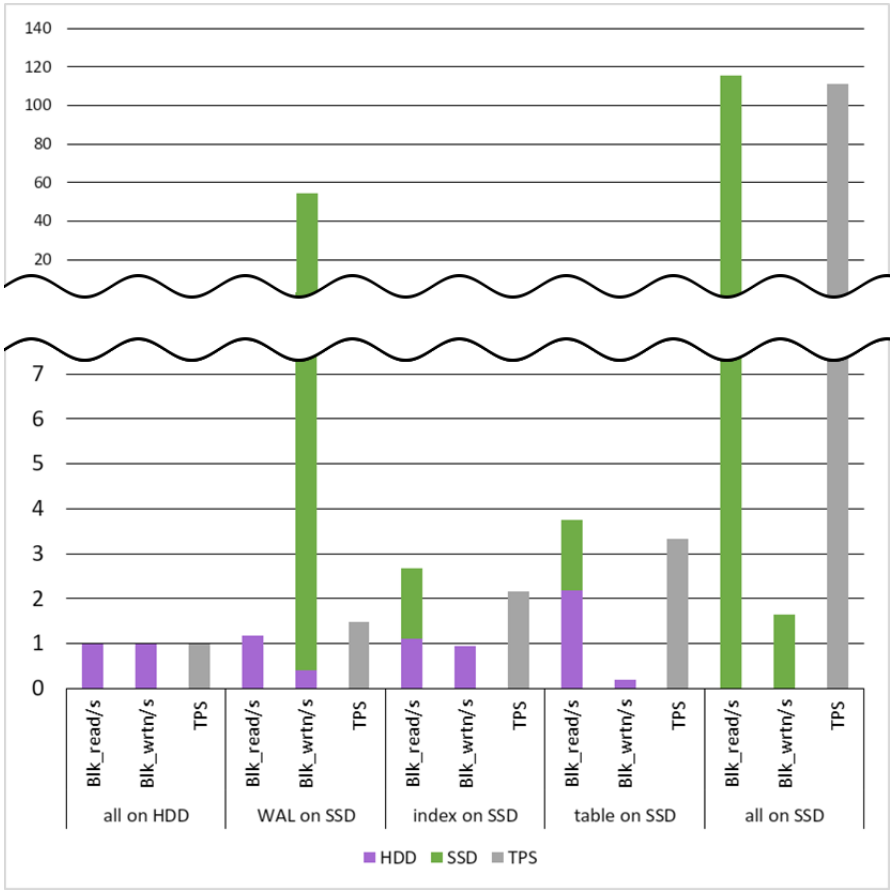

図 4.7: pgbench 参照系検証時の I/O 性能およびデータベース性能の全 HDD 配置対比（PCIe SSD 搭載マシン）

表 4.9 と図 4.7 から、参照系検証時のデータベース性能の増加比率は、HDD と SSD を合計した読み込み性能の増加比率とほぼ同じことがわかります。これは、参照系検証では読み込み性能がボトルネックになっていることを示しています。
また、更新系と同様に、一部を SSD に配置した場合はすべてを SSD に配置した場合と比較して、SSD のディスク読み込み性能を十分に発揮できていないことがわかります。

更新系検証および参照系検証の I/O 性能とデータベース性能の関係を確認した結果から、本測定では測定中の I/O 性能がデータベース性能に大きな影響を与えていることがわかりました。

4.6.2. インデックスオンリースキャン性能検証の考察

検証結果（表 4.7 と図 4.5）から、SSD 配置パターン別インデックススキャン性能検証と同様に、データディレクトリおよびデータベースオブジェクトの配置先をすべて SSD にした場合、大きな性能向上が得られました。
配置先をすべて HDD にした場合と比較して、PCIe SSD 搭載マシンではインデックスオンリースキャン成功時の TPS 値が約85倍、インデックスオンリースキャン失敗時の TPS 値が約 41 倍になり、SATA SSD 搭載マシンでは成功時の TPS 値が約 119 倍、失敗時の TPS 値が約 45 倍になりました。
この結果から、データベースのストレージデバイスを SSD にした場合、インデックススキャンと同様に、インデックスオンリースキャンでも大きな性能向上を得られることがわかります。
インデックスオンリースキャン失敗時の方は、成功時と比較してデータベース性能の対 HDD 比が半分程度になっていますが、それでも HDD に対する大きな性能優位性（数十倍レベル）を保っています。

インデックスオンリースキャン成功時にこのような大きな効果が出た理由としては、すべて HDD に配置した場合はディスク I/O がボトルネックとなり、すべて SSD に配置した場合はディスク I/O のボトルネックが解消され、CPU 性能により処理性能が左右されるようになったためと考えられます。

以下の図 4.8 と図 4.9 では、PCIe SSD 搭載マシンでの、すべて HDD および SSD に配置した場合の CPU 使用率の推移を積み上げ面で、ディスクビジー率の推移を折れ線であらわしています。iostat -x コマンドによって出力した SSD、HDD それぞれの%util の値をディスクビジー率として採用しており、これは全体の CPU 時間に対する I/O リクエストがデバイスに発行されている時間の割合を示しています。

これらの図から、すべて HDD を配置した場合はディスク I/O がネックとなり、すべて SSD に配置した場合は CPU ネックになっていることが確認できます。

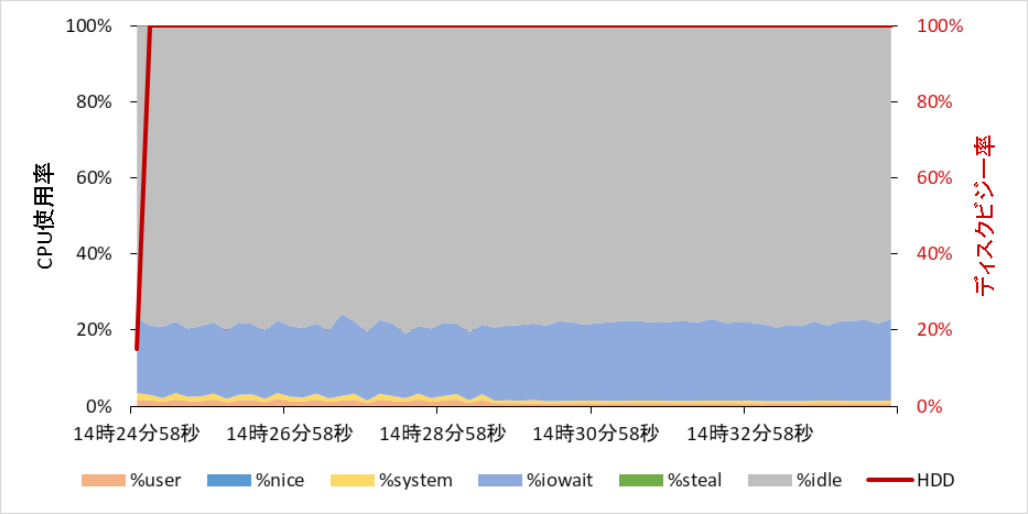

図 4.8: すべて HDD に配置した場合の CPU 使用率とディスクビジー率の推移（PCIe SSD 搭載マシン）

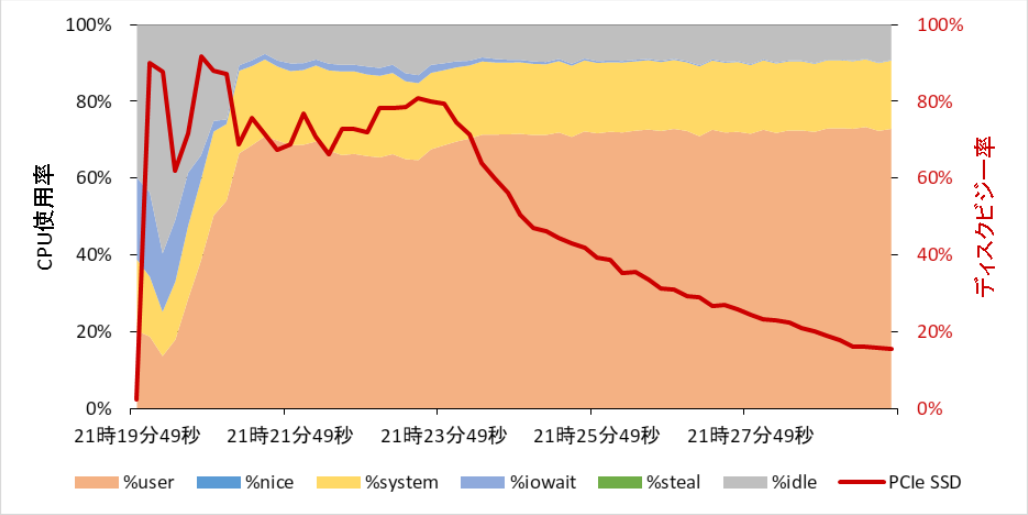

図 4.9: すべて SSD に配置した場合の CPU 使用率とディスクビジー率の推移（PCIe SSD 搭載マシン）

PCIe SSD 搭載マシンと SATA SSD 搭載マシンで比較すると、インデックスオンリースキャン成功時は PCIe SSD よりも I/O 性能が低い SATA SSD 搭載マシンの TPS 値の方がわずかに高くなっています。

これは、PCIe SSD 搭載マシンよりも SATA SSD 搭載マシンのコア数が多く（表 4.1）、かつ CPU 性能が I/O 性能よりもデータベース性能に大きく影響したためと考えられます。

逆にインデックスオンリースキャン失敗時は、成功時と比較して TPS の値が PCIe SSD 搭載マシンで約 10 分の 1 に、SATA SSD 搭載マシンで約 100 分の 1 になっており、失敗時の TPS の値が SATA SSD 搭載マシンの方が低くなっています。この理由としては、インデックスオンリースキャン失敗時は WAL 書き込みネックが発生していることが考えられます。

インデックスオンリースキャンに失敗すると、インデックスだけでなくヒープからもデータを参照します。この際、更

新後の VACUUM が実行されていないと Page Pruning 処理（ページに閉じた VACUUM 処理）が実行されます。Page Pruning 処理内では WAL に HEAP2_CLEAN レコードを挿入するので、WAL 書き込みが発生します。インデックスオンリースキャン成功時は、Page Pruning 処理 が実行されないため、WAL 書き込みは発生しません。

検証完了後、別マシン上でもインデックスオンリースキャン失敗時に WAL 書き込みが発生していることを確認しています。以下に、確認時に取得したデバッグ情報を記載します。

```
#0  XLogInsert (rmid=9 '¥t', info=16 '¥020', rdata=0x7fff60f356e0)
    at xlog.c:715
#1  0x0000000000491d1d in log_heap_clean (reln=0x7f01e7b483f0, buffer=106,
    redirected=0x7fff60f357f4, nredirected=1, nowdead=0x7fff60f35c80, ndead=0,
    nowunused=0x7fff60f35ec6, nunused=1, latestRemovedXid=53068)
    at heapam.c:6006
#2  0x0000000000495fa0 in heap_page_prune (relation=0x7f01e7b483f0,
    buffer=106, OldestXmin=53069, report_stats=1 '¥001',
    latestRemovedXid=0x7fff60f3629c) at pruneheap.c:240
#3  0x0000000000495c9f in heap_page_prune_opt (relation=0x7f01e7b483f0,
    buffer=106, OldestXmin=53069) at pruneheap.c:129
#4  0x000000000049d3be in index_fetch_heap (scan=0x15e8d90) at indexam.c:523
#5  0x0000000000060bd20 in IndexOnlyNext (node=0x15e7ca0)
    at nodeIndexonlyscan.c:109
#6  0x00000000005fb4c2 in ExecScanFetch (node=0x15e7ca0,
    accessMtd=0x60bc34 <IndexOnlyNext>, recheckMtd=0x60c1bc <IndexOnlyRecheck>)
    at execScan.c:82
#7  0x00000000005fb536 in ExecScan (node=0x15e7ca0,
    accessMtd=0x60bc34 <IndexOnlyNext>, recheckMtd=0x60c1bc <IndexOnlyRecheck>)
    at execScan.c:132
#8  0x0000000000060c242 in ExecIndexOnlyScan (node=0x15e7ca0)
    at nodeIndexonlyscan.c:232
#9  0x00000000005f0cfa in ExecProcNode (node=0x15e7ca0) at execProcnode.c:408
#10 0x00000000005eeed4 in ExecutePlan (estate=0x15e7b90, planstate=0x15e7ca0,
    operation=CMD_SELECT, sendTuples=1 '¥001', numberTuples=0,
    direction=ForwardScanDirection, dest=0x15e14b8) at execMain.c:1472
#11 0x00000000005ed2b3 in standard_ExecutorRun (queryDesc=0x14fd880,
    direction=ForwardScanDirection, count=0) at execMain.c:307
#12 0x00007f01f0b7e9fb in pgss_ExecutorRun ()
   from /usr/local/pgsql/lib/pg_stat_statements.so
#13 0x00000000005ed18d in ExecutorRun (queryDesc=0x14fd880,
    direction=ForwardScanDirection, count=0) at execMain.c:253
#14 0x0000000000721408 in PortalRunSelect (portal=0x15e5b80, forward=1 '¥001',
    count=0, dest=0x15e14b8) at pquery.c:946
#15 0x00000000007210e0 in PortalRun (portal=0x15e5b80,
    count=9223372036854775807, isTopLevel=1 '¥001', dest=0x15e14b8,
    altdest=0x15e14b8, completionTag=0x7fff60f36890 "") at pquery.c:790
#16 0x000000000071b4f1 in exec_simple_query (
    query_string=0x15a79f0 "select aid from pgbench_accounts where aid = 16;")
    at postgres.c:1048
#17 0x000000000071f4f5 in PostgresMain (argc=1, argv=0x14d8100,
    dbname=0x14d7fb0 "postgres", username=0x14d7f90 "postgres")
    at postgres.c:4005
#18 0x00000000006c73e0 in BackendRun (port=0x14f6f20) at postmaster.c:3999
#19 0x00000000006c6ba6 in BackendStartup (port=0x14f6f20) at postmaster.c:3688
#20 0x00000000006c377c in ServerLoop () at postmaster.c:1589
#21 0x00000000006c2fe8 in PostmasterMain (argc=3, argv=0x14d5f50)
    at postmaster.c:1258
#22 0x0000000000636b8 in main (argc=3, argv=0x14d5f50) at main.c:196
```

インデックスオンリースキャン失敗時の測定前処置として、検索対象であるテーブル pgbench_accounts が持つす

べてのページに更新を行っているので、大量の WAL 更新が発生し、ディスク I/O ネックが生じます。iostat -x コマンドでディスクビジー率を測定した結果を確認したところ、PCIe SSD 搭載マシン、SATA SSD 搭載マシンのいずれも SSD が実際にディスクビジー状態となっていました。

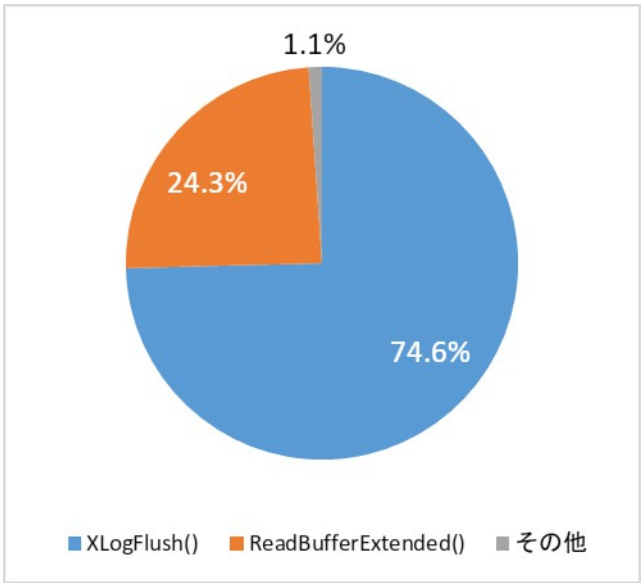

図 4.10: インデックスオンリースキャン失敗時のスタックトレース取得結果(SATA SSD 搭載マシン)

また、実行中のバックエンドプロセスがどの処理で留まっているか確認するため、測定中の SATA SSD 搭載マシンでバックエンドプロセスのスタックトレースを 10 秒間隔で取得しました(図 4.10)。その結果、インデックスオンリースキャン失敗時は WAL 書き込み実行中のプロセスが 74.6%、バッファ読み込み実行中のプロセスが 24.3%と、I/O 処理中のプロセスが全体の 97.6%を占めました。インデックスオンリースキャン失敗時は I/O 処理、中でも WAL 書き込み処理が全体の処理時間の大部分を占めると言えます。

当初はヒープからのデータ参照の性能にインデックスオンリースキャン失敗時の性能が左右されると考えていましたが、実際はこの WAL 書き込み処理性能が大きく影響することがわかりました。

WAL 書き込みはシーケンシャルアクセスであり、かつシーケンシャル書き込み性能は PCIe SSD の方が SATA SSD よりも高い(図 4.1、図 4.2)ので、その性能差がデータベース性能に反映されたものと考えられます。

4.6.3. 検証全体を通した考察

SSD 配置パターン別インデックススキャン性能検証では、すべて HDD に配置した場合に比べて、すべて SSD に配置した場合が PCIe SSD 搭載マシンの更新系のデータベース性能が約 30 倍、参照系のデータベース性能が約 111 倍と、圧倒的な性能改善効果があらわれました。これは、データモデルと実行シナリオが I/O 性能に依存した性能を出すものであったためと考えます。

インデックスオンリースキャン性能検証でも、成功時は PCIe SSD 搭載マシンのデータベース性能が対 HDD 比で約 85 倍、参照系のデータベース性能が約 119 倍と、大きな性能向上がみられました。インデックスオンリースキャン失敗時も数十倍の効果があらわれました。

SSD 採用によって最大限の効果を得る、今回のようなデータモデルと実行シナリオの場合、多大な効果が得られますが、データモデルや実行シナリオが異なる場合、同じような性能改善効果が得られるとは限りません。

仮にテーブルがすべてオンメモリになる程度の大きさのデータモデルの場合、データアクセスは最初以外すべてキャッシュアクセスとなり、ディスク I/O がほとんど発生しないため、CPU や WAL 書き込み、ロック競合などが SQL 実行のネックになります。

また、実行シナリオに範囲検索や複雑な集計、演算処理を含む場合、CPU 処理がネックとなり、I/O 性能にデータベース性能が影響しないこともあります。

今回は 1 つのデータモデルと、インデックススキャン・インデックスオンリースキャンの実行シナリオで性能を検証しました。しかしデータベースには、業務や用途毎にデータモデル、実行シナリオが存在します。

「どのような業務、どのような用途で性能改善効果が現れるか」を追求するため、引き続き様々な環境で SSD 採用した場合の性能検証を行う必要があると考えています。

5. スケールアウト検証（Postgres-XC）

5.1. 概要

5.1.1. はじめに

Postgres-XC は NTT OSS センタ、EnterpriseDB 社ほかが参加するコミュニティによって開発が進められている、以下のような特徴をもった分散 DBMS です[3]。

1. データ実体を共有しない複数のサーバで構成される "Shared Nothing" 型クラスタ DBMS
2. データの分散配置によって、参照・更新処理ともにサーバ台数に応じて性能が向上する"スケーラビリティ"を実現
3. PostgreSQL と API レベルでの互換性があり、トランザクションの ACID 性を保証するので、複数のサーバを用いて性能向上を図りつつ、通常の PostgreSQL と同様の利用が可能

Postgres-XC の概要や動作原理については、PGECons の 2012 年度の報告書[4]に説明がありますので、ご参照ください。

本項では、Postgres-XC のスケールアウト性(クラスタを構成するサーバの台数に応じて、性能が向上すること)を検証した結果を報告します。

5.2. 検証目的

昨年度と同様に、Postgres-XC のスケールアウト性を検証するにあたって、pgbench[5]を使ってクラスタを構成するノードの台数を増した時のスループット性能を測定しました。

Postgres-XC を実際に使用する際の参考となるように、以下のような 2 つの観点でのスケールアウト性を評価します。

1. DB サイズを一定として、ノードの台数を増した時のスループット性能の向上度合い
2. 1 ノードあたりの DB サイズを一定として、ノードの台数を増しつつ全体としての DB サイズを大きくした時のスループット

上記の 1 は実際の業務において DB のサイズは大きく変化しないものの、利用者の増大等で性能に対する要求が高まるシナリオに対応します。以下、「スループット向上シナリオ」と呼びます。それに対して、2 は DB サイズが大きくなる業務シナリオに対応するもので「DB サイズ拡張シナリオ」と呼びます。

2012 年度の検証パタンは(具体的なサイズ等は異なるものの)スループット向上シナリオに基づいています。一方、今回追加した DB サイズ拡張シナリオは、サーバを追加しても 1 サーバあたりの DB サイズが変わらないので、スケール性の検討の簡素化が期待できます。この点はのちに 5.6 節で詳しく見ます。

5.3. 検証構成

5.3.1. 評価対象ソフトウェア

検証に使用する Postgres-XC のバージョンについては、2013 年 12 月時点の最新版である 1.1STABLE をベースとしています。Postgres-XC 1.1 は PostgreSQL 9.2 系をベースとしているため、PostgreSQL 単体との比較をする際は、2013 年 12 月時点の最新メジャーバージョンである 9.3 系ではなく、9.2.5 を対象としています。

5.3.2. 検証用プラットフォームの構成

検証に際しては、日本電気株式会社のご厚意により、別紙に示すようなハードウェア設備を使用させていただきました。

検証用の Postgres-XC クラスタを構成するにあたり、GTM 専用のサーバを 1 台、Coordinator と Datanode をセットで配置したサーバを複数台用意しています（図 5.1 を参照）。Coordinator と Datanode を配置したサーバについて、以下ノードと呼ぶこととします。検証では PostgreSQL のベンチマークツールである pgbench を使用して測定を実施し、ノードを追加していくことによる参照・更新性能のスケール性について検証します。

ネットワークはギガビットイーサを 2 系統使用します。全ての Coordinator と Datanode が GTM へ接続する場合、GTM

3 http://sourceforge.net/projects/postgres-xc/

4 PostgreSQL Enterprise Consortium, “2012 年度 WG1 報告書”, https://www.pgecons.org/download_main/ , 2013.

5 pgbench の詳細については 2.2 節(9ページ)を参照

との通信処理がボトルネックになる可能性があるため、GTM と各ノード間は専用のネットワークとします。GTM 本体への負荷を軽減することを目的として、各ノードには GTM Proxy を併せて配置して GTM との通信を任せ、Coordinator と Datanode は同一ノード上の GTM Proxy に接続するようにしています。もう 1 系統のネットワークは、XC クラスタを構成するノード間、および coordinator とクライアント間の通信に使用します。

実際にデータが格納されるのは各ノードの Datanode になります。データベースクラスタの格納先として各 Datanode に割り当てている外部ストレージは、RAID10[6]で構成しています。ストレージ装置上には 1 ノードあたり 6 台のハードディスクドライブ(15kRPM, 300GB)が接続されます。ストレージとの各ノードとの間は 8Gbps の Fibre channel 2 本で接続しています。

負荷を掛けるクライアントは 2 台のサーバを利用します。一方には性能データを採取するために pg_statsinfo[7]を動作させます。

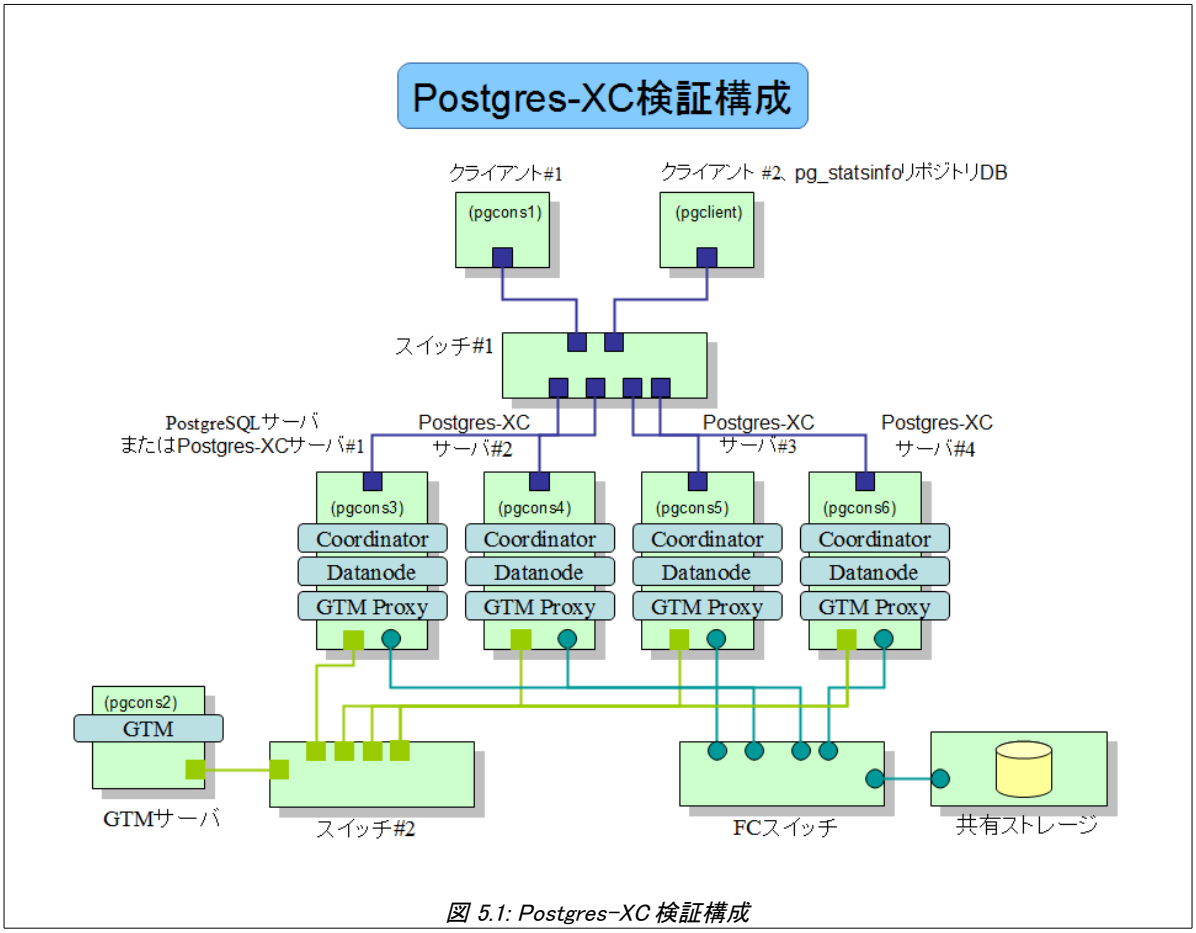

図 5.1: Postgres-XC 検証構成

6 昨年度は RAID5 構成としたが、書込み負荷が多いことに配慮して更新性能に優れる RAID10 構成とした。

7 http://pgstatsinfo.projects.pgfoundry.org/index_ja.html を参照。オリジナルの pg_statsinfo は Postgres-XC に対応しないため、適宜改修して使用。

個々のサーバ、クライアント等のスペックを表 5.1 に示します。

表 5.1: 検証構成

機器	項目	仕様
サーバ#1～#4 GTM クライアント#1	CPU	E5-2470(2.3GHz, 8Core)×2
	メモリ	32GB
	OS	Redhat enterprise Linux 6.1
クライアント#2	CPU	E5-2690(2.9GHz, 8Core)×2
	メモリ	32GB
	OS	Redhat enterprise Linux 6.2
共有ストレージ	ハードディスクドライブ	300GB, 15kRPM × 24 台 (6 台 1 組で RAID 10 を1系統構成し、全体を 4 系統として使用)
	インタフェース	Fibre channel 8Gbps × 2 系統
	内蔵キャッシュメモリ	8GB

5.3.3. Postgres-XC のパラメタ設定

今回の検証目的にそって、Postgres-XC および PostgreSQL の各種のパラメータ(postgresql.conf に書かれるもの)を表 5.2 のように設定しました。基本的な方針は次の通りです。

1. Postgres-XC が必要とする設定を行う(max_prepared_transactions)
2. 性能測定上のボトルネックになりうる箇所を回避するように、値を決める(max_connections)
3. 検証に使用するハードウェアの構成に見合った値にする(shared_buffers, checkpoint_segments)
4. 検証時の走行は短時間で完了することから、通常は DB の保守に使用されるプロセスが動作しないようにすることで、測定値のバラつきを軽減する(checkpoint_timeout, autovacuum)

表に掲げた表に掲げたパラメータ以外にも、ログ採取等の目的でデフォルトの値を変更していますが、性能には影響しないので割愛しています。

表 5.2: PostgreSQL および Postgres-XC パラメータ[8]

パラメータ	設定値			コメント
	PG	CO	DN	
max_prepared_transactions	1000	1000	1000	Postgres-XC はトランザクション実行の際に内部で PREPARE TRANSACTION を発行しているため、max_prepared_transactions には同時に実行される可能性のあるトランザクションの数を設定する必要がある。
max_connections	200	200	1000	pgbench の同時実行クライアント数を 120 としたため、それ以上の値とした(PG, CO)。Coodinator は、1 つのトランザクションから最大 4 つの Datanode に問い合わせを発行するため、Datanode の値は Coodinator の 4 倍以上とした。
shared_buffers	8GB	8GB	8GB	デフォルト値は小さすぎるため、環境に合わせて調整。
checkpoint_segments	1000	1000	1000	デフォルト値は小さすぎるため、環境に合わせて調整。
checkpoint_timeout	60min	60min	60min	測定結果への影響を考慮して停止。
autovacuum	off	off	off	測定結果への影響を考慮して停止。

5.3.4. Postgres-XC での DB 設計

Postgres-XC はデータベースを構成する表を各データノードに分散するか、あるいは全てのデータノードに重複して格納するか、選択することができます(脚注 4 の参考文献を参照)。

今回の検証では、pgbench を構成する 4 つの表に関して、表 5.3 に示すようにデータを分散して格納しています。また、分散した表はインデックス経由で検索できるようにします。

表 5.3: Postgres-XC のデータベースの物理設計

表名	データの分散方法	検索時のキー	インデックスを付与したカラム
pgbench_branches	bid をキーとして hash で分散	bid	bid
pgbench_tellers	tid をキーとして hash で分散	tid	tid
pgbench_accounts	aid をキーとして hash で分散	aid	aid
pgbench_history	aid をキーとして hash で分散	(検索対象にならない)	(なし)

8 表中の略号は次の通り。PG:PostgreSQL, CO:Postgres-XC のコーディネータ、DN:同じくデータノード。

5.4. 検証方法

ここでは、検証の目的ごとに具体的な性能測定方法について説明します。性能指標としては、スループットを用います。

5.4.1. 検証の際の条件の設定について

昨年度と同様に pgbench を用いて、更新主体、参照主体の負荷を Postgre-XC のサーバに与えます。先に 5.2 節で述べたように、2 種類のシナリオを想定して条件を設定します。

(1) スループット向上シナリオ

検証にしようするサーバのハードウェアスペックのうち、ノードのメモリサイズに着目して 4 ノード使用時に DB がメモリに乗り切る程度の DB サイズとします。ノードには 32GB のメモリが実装されていますが、shared_buffers で 8GB を指定したため、実効的には 24GB 程度がバッファとして利用されます。DB サイズはノードが 4 台の場合にデータベース内容がメモリに乗り切るようにします。pg_bench のデータベースは、スケールファクタ 1=約 15MB の関係があるので、スケールファクタは 6000（DB サイズは約 90GB）としました。

(2) DB サイズ拡張シナリオ

スループット向上シナリオにおける DB サイズの決め方と同様に、各ノードのメモリにデータベース内容が乗り切るように DB サイズを決めます。具体的には、1ノードあたりスケールファクタが 1500 ずつ増すようにします。

これらの関係を表 5.4 に示します。

表 5.4: 検証時の負荷設定

ノード数		スループット向上シナリオ		DB サイズ拡張シナリオ	
		スケールファクタ	DB サイズ	スケールファクタ	DB サイズ
PostgreSQL		6000	90GB	1500	22.5GB
Postgres-XC	ノード数 1	6000	90GB	1500	22.5GB
	ノード数 2	6000	90GB	3000	45GB
	ノード数 3	6000	90GB	4500	67.5GB
	ノード数 4	6000	90GB	6000	90GB

5.4.2. 性能の測定方法

本検証での DB サーバの性能指標は、pgbench を用いて測定開始直後から測定完了までの間に実行が完了したトランザクション数をもとに算出した平均スループット値です。Postgres-XC については同一の負荷条件に対して3回走行して得られた 3 つのスループット値を平均したものを、その条件での性能値とします。個々の走行については以下のような条件で実施しました。

1. 1 回あたり 10 分間走行する
2. 走行中の全トランザクションをもとに平均スループットを計算する

また、データベースへの更新が発生する条件(更新系と呼びます)と、参照しか発生しない条件(参照系)の 2 種類の負荷に対する性能を測定します。

測定中には以下のような方法で DB サーバの状態を観測します。

1. サーバのハードウェアの利用状況は sar で測定する
2. Postgres-XC を構成する各ノードの活動状況は pg_statsinfo で観測する

これら、サーバの状態に関しては測定結果を解釈する際に、適宜利用することとします。

5.5. 検証結果

測定結果を表 5.5 に示します。

表 5.5: スループット測定結果（単位：TPS[9]）

ノード数		スループット向上シナリオ		DB サイズ拡張シナリオ	
		参照系	更新系	参照系	更新系
PostgreSQL		910	761	7753	2777
Postgres-XC	ノード数 1	889	706	6864	2252
	ノード数 2	2903	1088	12140	2378
	ノード数 3	7173	1648	17780	2334
	ノード数 4	21289	2293	21740	2294

9 Transactions Per Second.（トランザクション毎秒）

(1) スループット向上シナリオ

スループット向上シナリオでの測定結果をグラフ化したものを図 5.2 と図 5.3 に示します。

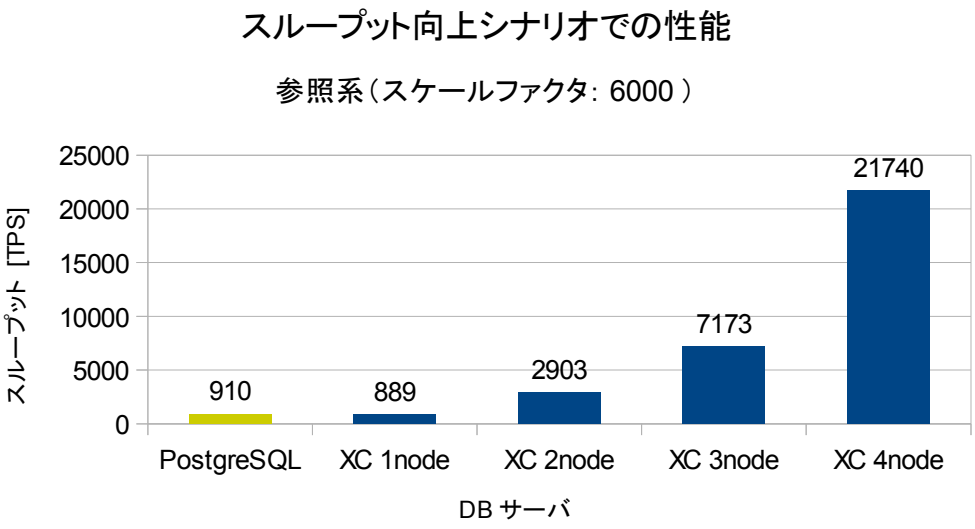

図 5.2: スループット向上シナリオでの性能（参照系）

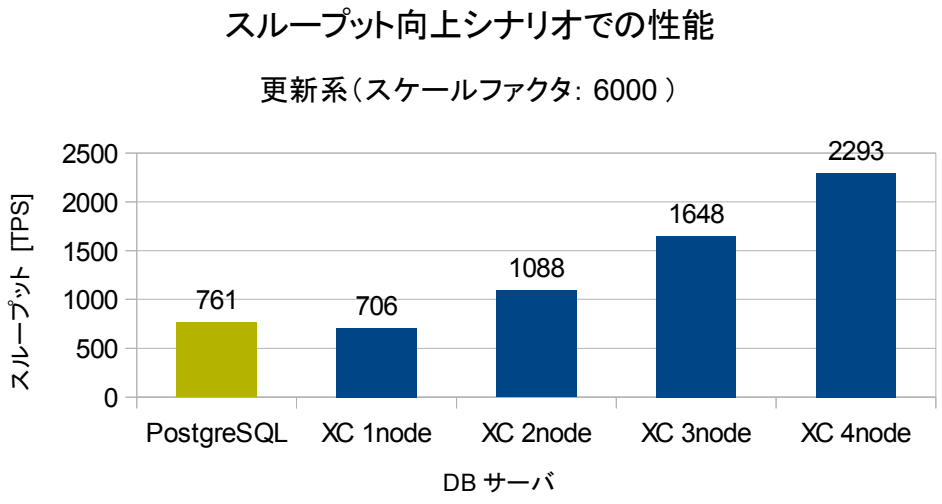

図 5.3: スループット向上シナリオでの性能(更新系)

(2) DB サイズ拡張シナリオ

測定結果をグラフ化したものを図 5.4 と図 5.5 に示します。このシナリオでノード数が 4 の時にはデータベースの規模を示すスケールファクタは 6000 となり、先に示したスループット向上シナリオでのデータベースと等しくなります。そこで、Postgres-XC と PostgreSQL との性能値を比較する意味で、先の結果の数値の一部を再掲しています。

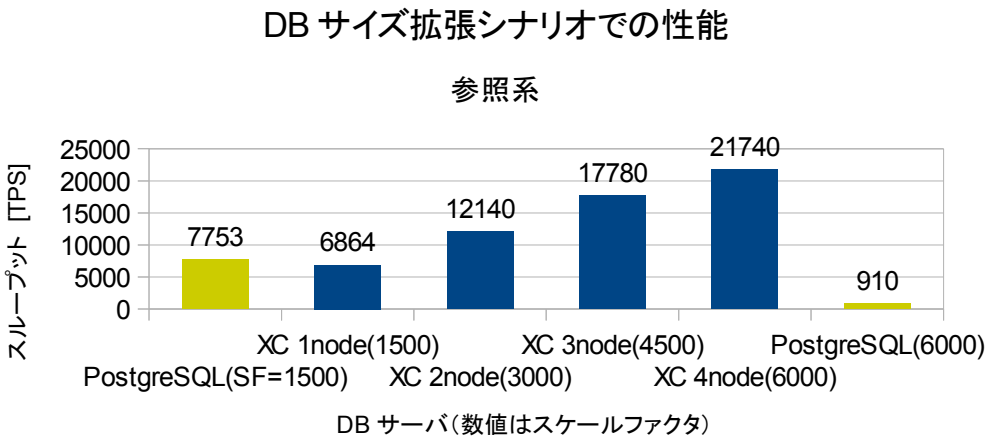

図 5.4: DB サイズ拡張シナリオでの性能（参照系）

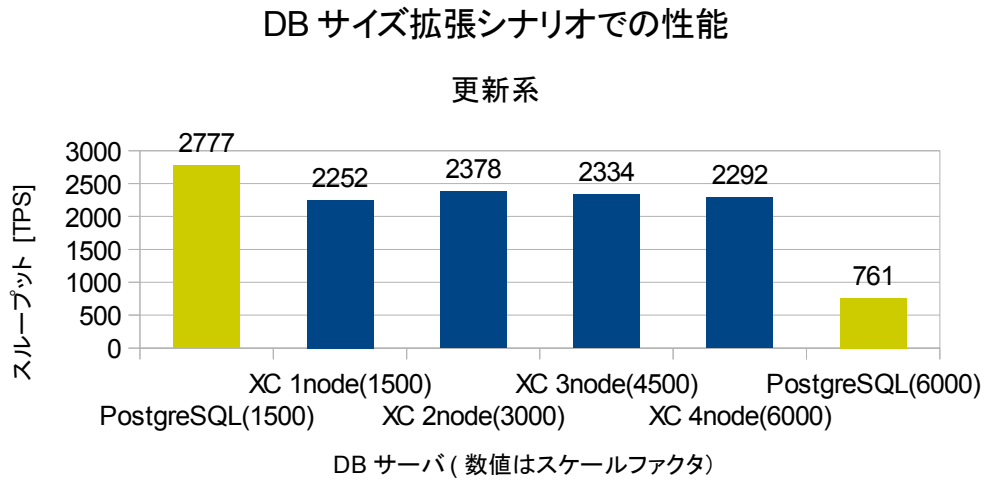

図 5.5: DB サイズ拡張シナリオでの性能（更新系）

5.6. 考察

5.6.1. スループット向上シナリオ

更新系のスケール性について、Postgres-XC のノード数を増した時の性能(スループット)の向上の度合いを検討します。先に 5.5 節で示した結果をもとに、更新系・参照系それぞれのケースでの PostgreSQL でのスループットを 1 として Postgres-XC でのスループットを表現したものを図 5.6 に示します。

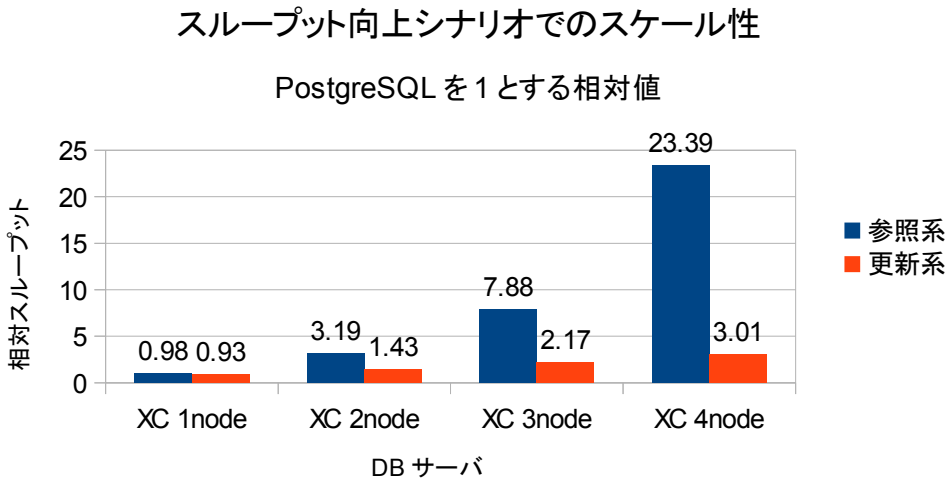

図 5.6: スループット向上シナリオでのスケール性

- 更新系のスケール性については、サーバ台数が 1 台の時に通常の PostgreSQL の約 0.93 倍となり、台数が 4 台の時に約 3.01 倍となります。
- 参照系のスケール性については、サーバ台数が 1 台の時に通常の PostgreSQL の約 0.98 倍となり、台数が 4 台の時に約 23.4 倍となります。
- 更新系に比べて、参照系のスケールアップの比率は極めて高くなります(3.01 対 23.4)。これは、参照系の pgbench の場合に、ボトルネックとなる処理が表の内容を読み出すためのランダムリードになることと、DB サイズが固定されているため、台数が増えるほど利用できるメモリサイズが大きくなって、主記憶上にキャッシュされたデータを読み出す比率が増える結果として、ノード数が増えるほどストレージにアクセスする割合が減って、性能が向上するためと考えられます。
- 更新系においても表からの読み出しがランダムリードとなる点は参照系と同じです。一方、更新系ではデータベースへの更新結果をトランザクションログ(WAL ファイル)に書きだす操作が必要となり、これが参照系に比べて性能の向上を制約しているものと見られます。

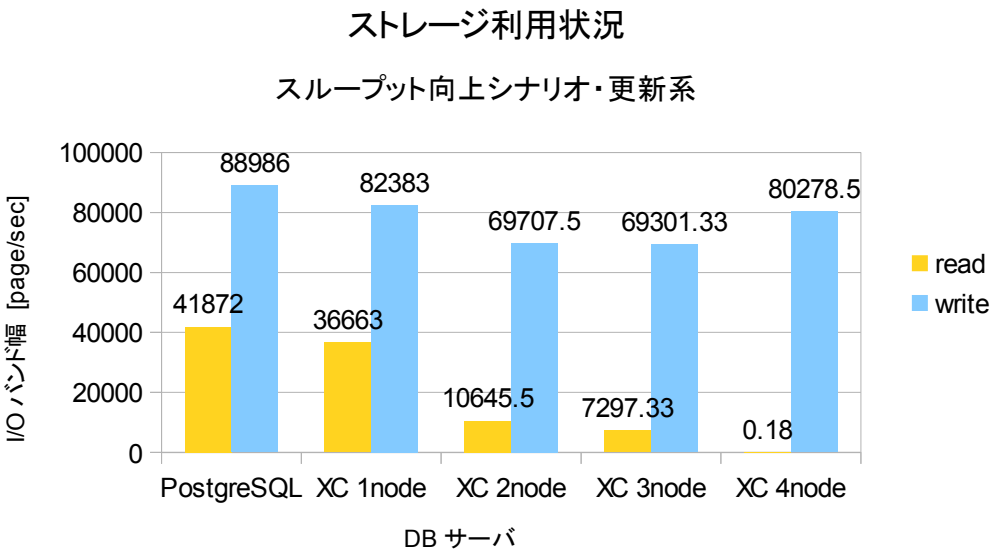

図 5.7: スループット向上シナリオ・更新系でのストレージ利用状況

この点を確認するために、ストレージの利用状況を見てみます(図 5.7)。この図では、各ノードのストレージの利用状況を I/O バンド幅(ストレージ装置からサーバへのデータ転送のスループット。単位はページ毎秒、1 ページは 512 バイト)で示しています。ノードが複数台あるケースでは、それらの平均値を示しています(ノード数 4 の時には、ストレージ装置全体で約 32 万ページ毎秒のバンド幅となります)。

グラフの read(読み出し;オレンジ色のバー)に注目すると、台数が増すとバンド幅が下がることから、ストレージからのデータ取得が減少することがわかり、キャッシュヒット率が向上していると推定されます。

他方 write(書込み;水色のバー)に注目すると、ノード 1 台あたりの書込みバンド幅がほぼ一定しているように見えます。言い換えれば、書込み処理がボトルネックとなって性能を制約しています。バンド幅をバイトに換算すると、約 40MB/sec となります。今回使用したストレージ装置の性能を別途測定したところ、コミット時に実行される fsync を含めてデータを書き込む場合には、書き込み待ちが発生するために、表 5.6 のような性能傾向を示すことが確認できました。

表 5.6: ストレージ装置の書き込みバンド幅

fsync の間隔	書込みバンド幅 [MB/sec]
8kB	3.6
128kB	40.2
fsync しない	197

PostgreSQL および Postgres-XC では更新トランザクションがコミットする際に、WAL ファイルへの書き込みを fsync を発行します。この表の測定では、sequential write を実行する際に、一定のバイト数を書き込むごとに fsync を発行するようなベンチマークテストを実行することで、WAL ファイルへの書き込みをシミュレートしたものです[10]。

更新系でのスケール性が参照系に及ばない理由として、Postgres-XC のアーキテクチャ上の観点も考える必要があります。この点は、次の DB サイズ拡張シナリオの考察で触れます。

10 ベンチマークツールは fio を使用しました(本ツールについては 4.2 節(32ページ)を参照)。

5.6.2. DB サイズ拡張シナリオ

スループット向上シナリオの場合と同様に、Postgres-XC のスケール性を図 5.8 に示します。

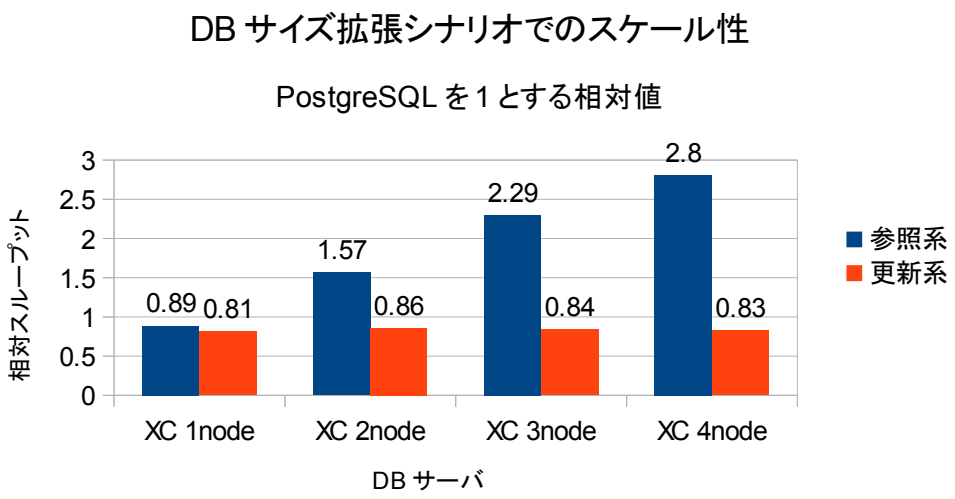

図 5.8: DB サイズ拡張シナリオでのスケール性

- 更新系のスケール性については、サーバ台数が 1 台の時に通常の PostgreSQL の約 0.81 倍となり、台数が 4 台の時に約 0.83 倍となります。
- 参照系にスケール性については、サーバ台数が 1 台の時に通常の PostgreSQL の約 0.89 倍となり、台数が 4 台の時に約 2.8 倍となります。
- 更新系については、ノード数を増した場合でも相対性能値がほぼ一定となる結果となりました。
  理想的に処理の分散が行われていれば、台数が増すにつれて性能が向上するはずですので、何らかのボトルネックが性能を制約していると考えられます。性能スケールシナリオでの更新系の結果と同様に、ストレージへの書き込みがボトルネックになっているものと考えられます。
- 参照系については、ほぼ台数に比例してスループットが向上することが確認できました。先にみたスループット向上シナリオでは、台数が増えると線形よりも速くスループットが向上しましたが、このシナリオでは、ほぼ線形です。これは、ノードの台数に比例して DB サイズを増やしているために、pgbench のランダムリードにおけるキャッシュヒット率がほぼ一定となり、ノード 1 台あたりのスループットがどのケースでも同程度となるためです。

(1) DB サイズ拡張シナリオのスケール性について

DB サイズ拡張シナリオでは、ノード数を増やした場合でも各 datanode におけるキャッシュヒット率が一定になるため、各ノードでのストレージからのデータ転送についてはノード数によらずほぼ同様になります(たとえば、キャッシュヒット率も一定となります)。このため、参照系の測定ではノード数とスループットがほぼ線形に比例する結果となりました。

一方、更新系についてはノードを追加してもクラスタ全体でのスループットはほとんど向上しませんでした。その理由は次のように考えられます。

(2) Postgres-XC 特有の振る舞い

Postgres-XC は shared nothing 型のクラスタデータベースであり、1 つの表を複数の datanode に分散して格納することができます。本検証でも 5.3.4 節で述べたように、pgbench に現れる 4 つの表のすべてを分散して格納しています。一方、pgbench の更新系トランザクションは 4 つの表のうち 3 つに関して、表の中の 1 行をランダムに選んで更新します。このため、datanode が 2 台ある構成では最大 2 つの datanode に対して、3 台および 4 台の構成では最大 3 つの datanode に対して、更新クエリ(update 文)が発行されます。pgbench の 1 つのトランザクションは、

Postgres-XC の内部では複数のサブトランザクションとして扱われます。

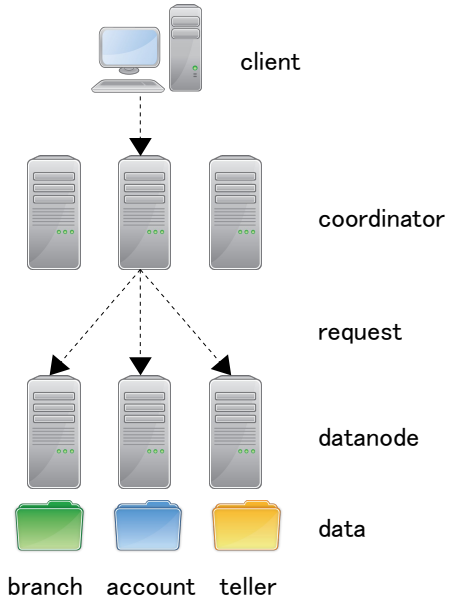

図 5.9: Postgres-XC のサブトランザクション

その様子を示したものが、図 5.9 です。この例では 3 台の datanode を持った構成の時に、1 つのトランザクションがデータベースにある 3 つの表 branch, account, teller を更新する際に、そのリクエストがすべて異なるデータノードに発行されています。pgbench では、これらの表の中のどの行を選ぶかはランダムに決められているので、場合によってはすべてのリクエストが 1 台のデータノードに集中することもあります。

Postgres-XC を用いて pgbench を実行する際に、1 つのトランザクションを処理するためにいくつの datanode が処理に参加するかを計算すると、表 5.7 の上段のようになります。

トランザクションのコミット時には、これらのサブトランザクションもコミットされます。この時 datanode 上では通常の PostgresSQL のコミット処理と同じく、WAL ファイルへの書き込みとフラッシュが行われます。

一つの datanode から見た場合、1 トランザクション毎に発行されるコミットの数は、表 5.7 の中段に示したように datanode 数で除した商となります。その値を見ると、クラスタを構成するノード数 n を増していくと、1 つの datanode が受け取るコミットの数は減るものの $1/n$ にはならず、ずっと緩やかな減り方をします。たとえばノード数が 4 の場合に、1 つの datanode が受け取るコミット数は、ノード数 1 の場合の約 44%となります。言い換えれば、コミット処理がボトルネックとなるようなアプリケーションでは、表 5.7 下段がアーキテクチャ上の最適なスケール性となります。それによれば、ノード数が 4 台の構成では、ノード数 1 台の場合の約 2.3 倍のスループットが得られると見積もることができます。今回の測定ではノード数を増してもスループットに大きな変化がないことから、これ以外にも性能を制約する要因があることを示唆しています。

表 5.7: トランザクション処理に参加する datanode 数

クラスタのノード数 n	1	2	3	4
トランザクション処理に参加する datanode 数	1	$\frac{7}{4}=1.75$	$\frac{13}{9}\simeq 1.44$	$\frac{7}{4}=1.75$
datanode から見た、1 トランザクションごとに発行されるコミット数	1	$\frac{7}{8}=0.875$	$\frac{13}{27}\simeq 0.481$	$\frac{7}{16}\simeq 0.438$
上記から予想される最適なスケール性	1	$\frac{8}{7}\simeq 1.143$	$\frac{27}{13}\simeq 2.077$	$\frac{16}{7}\simeq 2.286$

5.6.3. 2012 年度評価との比較

PGECons では、2012 年度にも Postgres-XC を用いて、性能のスケールアウト性を検証しています。2012 年度と 2013 年度では、使用したハードウェア、pgbench の設定が異なり厳密な比較はできないため、ここでは全体的な傾向を比較します。

表 5.8: 2012 年度と 2013 年度の主な測定条件

項目	2012 年度	2013 年度	備考
CPU	Intel Xeon E5-2670(2.6GHz, 8core)×2	Intel Xeon E5-2470(2.3GHz, 8core)×2	2012 年度の方が若干高性能
メモリ	64GB	32GB	2013 年度は2012年度の半分
ストレージ	HDD (15kRPM) ×15 (RAID 5)	HDD (15kRPM) ×24 (RAID 10)	RAID の構成を 5 から 10 とした
測定シナリオ	性能スケール:参照および更新	スループット向上:参照および更新	同様のシナリオで比較
DB サイズ	5000	6000	サイズは scale factor で示す

比較に使用する測定の条件を表 5.7 に挙げます。2012 年度も、今年度と同様にいくつかの測定シナリオに沿って検証を実施していますが、この比較では今年度の検証と類似のシナリオに絞っています。
それぞれの検証で得られた性能値をまとめたものを、図 5.10(参照系)、図 5.11(更新系)に示します。

(1) 参照系の性能について

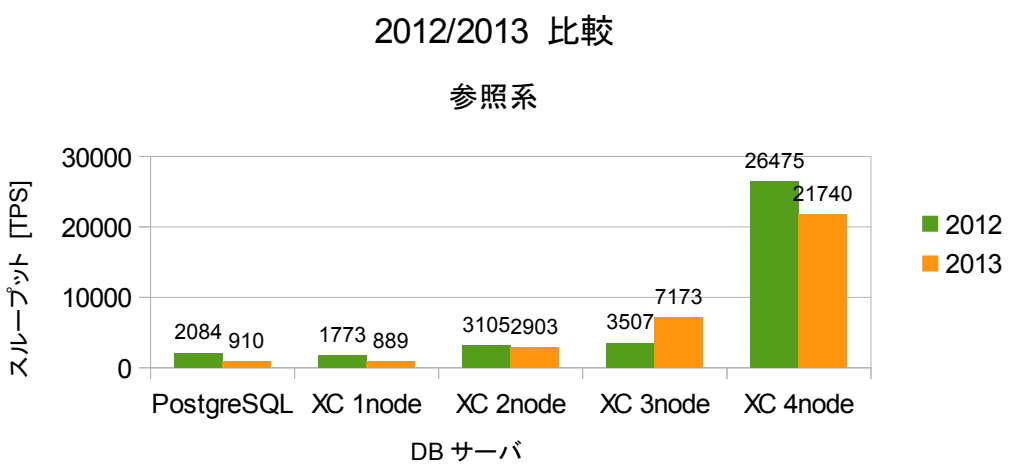

図 5.10: 2012 年と 2013 年の結果比較:参照系

参照系の性能は、2012 年、2013 年の検証結果は、ほぼ同様の傾向を示しています。異なる点としては、2012 年度の方が、1 ノードの時の性能がかなり良い(PostgreSQL, Postgres-XC ともに)。これは、2012 年度に使用したサーバ機の方が、メモリを多く搭載しておりキャッシュによる性能向上がより多く働くことと、CPU のクロックが高速である分、メモリ上のデータ処理も高速に実行されたと考えられます。

(2) 更新系の性能について

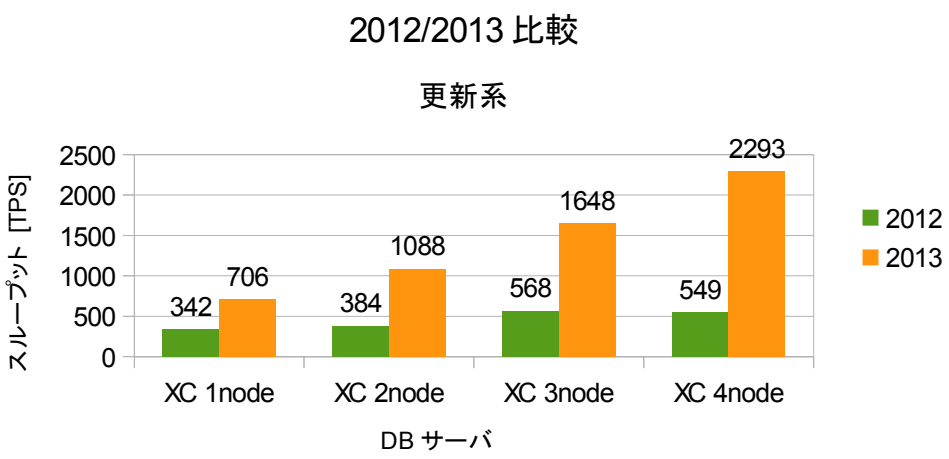

図 5.11: 2012 年と 2013 年の結果比較:更新系

更新系の性能は、2012 年、2013 年の検証結果で、かなり異なっています。異なる点としては、2013 年度の方が、2012 年度のよりも総じて高いスループットとなっている点です。先に、更新系の検証結果を考察した際に、性能に影響を及ぼす要因として、ストレージの書き込みがボトルネックとなっている点を指摘しました。2012 年度と 2013 年度のストレージの構成を比較すると、ハードディスクドライブ(HDD)の台数が 2013 年度の方が多いことに加えて、RAID の構成方法が異なっています。2013 年度の検証のためのハードウェア構成を決める際に、書き込み負荷に対して、性能上有利となる RAID10 構成を使用したことが、このようなスループットの差となって現れたものと考えられます。

5.6.4. 検証全体を通した考察

今年度の検証では、Postgres-XC を実際に使用する際の参考となるように、以下のような 2 つの検証シナリオを想定したうえでスケールアウト性を評価しました。

1. DB サイズを一定として、ノードの台数を増した時のスループット性能の向上度合い(スループット向上シナリオ)
2. 1 ノードあたりの DB サイズを一定として、ノードの台数を増した時のスループットの向上度合い(DB サイズ拡張シナリオ)

以下に、測定結果の概要をまとめます。

- どちらのシナリオにおいても、参照系では良好なスケールアウト性を示しました。とりわけスループット向上シナリオにおいては、台数が増加するにつれて利用できるメモリ量が増えることによって、キャッシュヒット率が向上し、台数に比例する以上の性能向上が見られました。
- 更新系においては、2 つのシナリオで異なる傾向が見られました。スループット向上シナリオでは、ほぼ台数に比例して性能がスケールアウトすることが確認できました。一方、DB サイズ拡張シナリオでは、ノードを追加して DB サイズを拡張してもほぼ一定のスループットとなっており、性能向上はしないものの性能劣化は見られませんでした。このことから、スループット向上シナリオのケースと比較して、何からのボトルネックによって性能が制約されている可能性が高いことが分かります。その理由としては、(1)コミット処理に起因するストレージのフラッシュ処理がボトルネックになっていること、(2)Postgres-XC 上では個々のノードへ負荷が分散されるが、アプリケーションの特性によってはサブトランザクションの処理が増えるケースがあること、が考えられます。

これらのことから、Postgre-XC の性能に関して、次のような特徴と課題があることが分かります。

- 参照系の場合、pgbench のようなランダムアクセスが主体となる比較的単純なトランザクションにおいては、台数に比例した良好なスケールアウトを期待することができます。
- 一方、更新系の測定結果からは、同じように単純なトランザクション負荷であれば、Postgres-XC はデータベースサイズの拡張に対して有効であることが示唆されました。スループットに関しては、書き込みに起因するとみられるボトルネックが顕在化するケースが見られました。こうしたケースに対しては、より適切なアプリケーションプログラムの設計や DB 設計、マシンの設定が必要であると考えられます。

5.6.5. 最後に

今回のスケールアウト検証の結果を勘案すると、Postgres-XC は pgbench に対してはおおむね良好なスケール性を示していると言えます。しかしながら、この結果だけから、同 DBMS の他のモデルでのスケール性を推測することは困難であり、実用システムへの適用性に関しては、より多くのモデルで評価する必要があります。その意味で、本報告書を PostgreSQL コミュニティや関係者に展開することで、さまざまな角度からの評価や適用性の検討へとつなげていくことが望ましいと考えられます。

6. おわりに

2012 年 4 月に発足した PostgreSQL エンタープライズ・コンソーシアム(PGECons)の活動も 2 年目にはいりました。本報告書では、技術部会ワーキンググループ 1（WG1）が今年度実施した性能検証について、目的や構成、検証方法と考察を考え方やプロセスを含めて詳細に説明しました。
今年度のテーマは、最新リリースである PostgreSQL 9.3 のスケールアップ検証、昨年できなかったパーティショニング機能の検証、近年エンタープライズでの使用が本格化してきている SSD（フラッシュストレージ）の検証、そしてスケールアウトによる更新処理へのスケーラビリティが期待される Postgres-XC の検証です。

今年度も昨年度同様、日常的に業務として PostgreSQL に携わっている技術力の高いメンバーが集まり、企業の枠を超えて活動することで、結果として本報告書にあるような大きな成果を得ることができました。今年度はさまざまなメディアで PGECons が紹介されるようになりました。そこで紹介されたとおり、競合することもある各企業のメンバーが、PostgreSQL のエンタープライズ活用を促進させるという共通の目的のもとに活動し、また交流を深めるということはとても意義のあることです。

次年度も、リリース予定の PostgreSQL 9.4 の検証をはじめとし、検証を継続していきます。ワーキンググループでの検討内容には、報告書には記載しきれない様々な情報があります。ワーキンググループはそのような貴重な情報を得る絶好の機会ですので、ぜひ一緒に活動しましょう！

著者

版	所属企業・団体名	部署名	氏名
2013 年度 WG1 活動報告 第 1.0 版 （2013 年度 WG1）	株式会社アシスト	データベース技術本部	永田 まゆみ
	株式会社アシスト	データベース技術本部	喜田 紘介
	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 OSS 技術グループ	安齋 希美
	NEC ソリューションイノベータ株式会社	PF システム事業部	岩浅 晃郎
	NEC ソリューションイノベータ株式会社	PF システム事業部	安西 直也
	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 システムソフトウェア事業部	川畠 輝聖
	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
	株式会社日立製作所	ソフトウェア本部 OSS テクノロジセンタ	吉瀬 宏
	富士通株式会社	ミドルウェア事業本部 データマネジメント・ミドルウェア事業部	山本 明範
	富士通株式会社	ミドルウェア事業本部 データマネジメント・ミドルウェア事業部	菅原 久嗣