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

2014 年度 WG1 活動報告

大規模 DB と仮想環境を見据えた PostgreSQL の性能検証

製作者
担当企業名:アイウエオ順
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
株式会社日立製作所

改訂履歴

版	改訂日	変更内容
1.0	2015/04/23	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（定点観測、WAL 改善）、仮想化、Docker の性能を検証した作業内容と検証結果を報告します。

■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(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。

目次

1. はじめに

1.1. 2014 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 1.1.1: 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 1.1.2: 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。

さらに、2014 年度の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。

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。

1.1.3. WAL 改善

定点観測（スケールアップ）では2012年度より参照系のワークロードについてのスケールアップ性の検証を行っています。今年度は PostgreSQL9.4 が WAL の性能改善項目が含まれることもあり、更新系のスケールアップ検証も実施することといたしました。

参照系の定点観測と同様に PostgreSQL9.4 と PostgreSQL9.3 を単純な更新性能の比較を示すことで、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ればと考えました。

1.1.4. 仮想化

近年、ハードウェア技術の進化により、CPU の高速化と多コア化、そしてメモリの大容量化が進んでいます。また Intel 64 アーキテクチャでの仮想化技術が進展しているため、従来一般的には物理環境での利用が基本であったデータベースサーバの仮想化が進んでいます。そこで本検証では、仮想化によるオーバヘッドがデータベース性能に与える影響の大きさを確認しました。

1.1.5. Docker

近年 Docker が注目されています。Docker を利用することにより、アプリケーションの可搬性を高め、運用コストの削減、またデプロイの効率化が達成できると期待されています。しかし Docker で実行単位となる「コンテナ」の環境下でアプリケーションを動かした時の性能面の影響についてはまだ未知数のところが多いのが現状です。Docker 上で PostgreSQL を動作させた時の性能を中心に検証を行いました。

1.2. 実施体制

2014 年 5 月 8 日に開催された 2014 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 1.2.1: 2014 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
株式会社日立製作所

この中で、日本ヒューレット・パッカード株式会社は、「主査」として、WG1 の取りまとめ役を担当することになりました。

1.3. 実施スケジュール

2014 年度は、下記スケジュールで活動しました。

表 1.3.1: 実施スケジュール

活動概要	スケジュール
WG1 スタート	2014 年 6 月 19 日
実施計画策定	2014 年 7 月～11 月
検証実施	2014 年 12 月～2015 年 2 月
2014 年度 WG1 活動報告書作成	2015 年 2 月～3 月
総会と成果報告会	2015 年 5 月 14 日

2. 定点観測(スケールアップ検証)

2.1. 検証概要

近年のCPUマルチコア化やメモリ大容量化の傾向を受け、2012年度および2013年度は、80コアのCPU(Xeon E7-4870(10Core) × 8)、メモリ 2TBという非常に高スペックのマシンを利用し、PostgreSQL 9.2およびPostgreSQL9.3の参照系の性能を測定しました。

今年度はマシンのスペックが60コアのCPU(Xeon E7-4890v2(15Core) × 4)、メモリ2TBと変わっていますが、定点観測としてPostgreSQL 9.4を用いて前年度と同様の参照性能を測定し、PostgreSQL 9.3との比較を行いました。なお、測定時点ではPostgreSQL 9.4の正式版がリリースされていなかったため、計測にはリリース候補版である9.4 RC1を用いました。PostgreSQL 9.3に関しては、計測時点の最新版である9.3.5を用いました。

また、昨年度はPostgreSQL 9.3から追加されたpage checksumの機能の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。今回も同様にPostgreSQL 9.4において page checksum 有効/無効がパフォーマンスに与える影響について計測しました。

検証した内容は以下の通りです。

表 2.1.1: スケールアップ検証 検証内容

	概要	測定方法
1	PostgreSQL 9.4と9.3の参照性能の比較	60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 9.4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	60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。

2.2. 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。

- 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。

¥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. PostgreSQL 9.3 と 9.4 の参照性能の比較

2.3.1. 検証目的

PostgreSQL 9.3 と 9.4 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

2.3.2.1. 検証ハードウェア構成

検証環境のハードウェア構成の概略を示します。

なお、使用したハードウェアについては 2014 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。

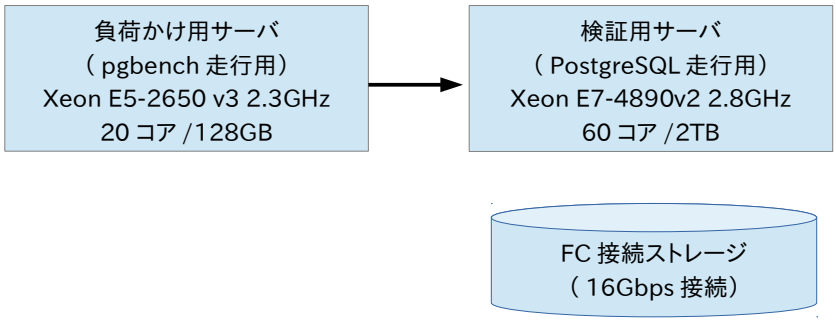

図 2.3.1: 検証ハードウェア構成

2.3.2.2. 検証ソフトウェア構成

検証環境のソフトウェア構成を示します。

OS	Red Hat Enterprise Linux 6.5
PostgreSQL	9.4RC1 および 9.3.5

表 2.3.1: 検証用サーバ

OS	Red Hat Enterprise Linux 6.5
pgbench	9.4RC1 のソースコードに含まれるものをビルドして使用

表 2.3.2: 負荷かけ用サーバ

2.3.3. 検証方法

2.3.3.1. 環境

以下の手順で、2013 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] '
```

shared_buffers に大容量のメモリを設定できるようにカーネルパラメータを変更します。

```
# vi /etc/sysctl.conf
-) kernel.shmmax = 68719476736   # 64 GB
+) kernel.shmmax = 687194767360  # 640 GB
# sysctl -p
```

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] 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]
```

2.3.3.2. 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。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。

2.3.3.3. 結果

結果をグラフと表に示します。グラフの横軸は同時接続クライアント数、縦軸は TPS です。PostgreSQL 9.4（緑、×マーク）も 9.3（赤、＋マーク）の両者で、クライアント数 80 の時が最大の TPS となりました。また、9.4 の方が若干低い結果になりました。表の CPU idle 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。

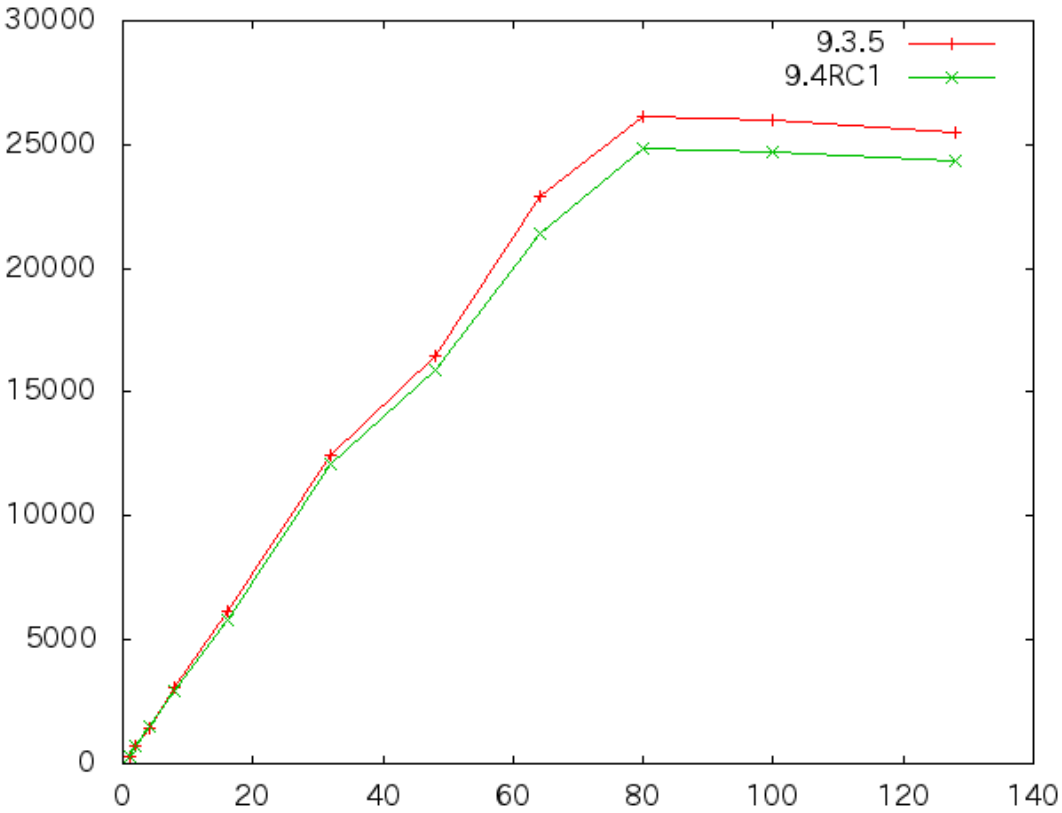

図 2.3.2: PostgreSQL 9.3 と 9.4 の参照性能の比較

表 2.3.3: PostgreSQL 9.3 と 9.4 の参照性能の比較

クライアント数	PostgreSQL 9.3		PostgreSQL 9.4		
	TPS	CPU idle	TPS	CPU idle	9.3 の TPS との比較
1	311.40	98.35 %	304.03	98.35 %	97.63 %
2	724.55	96.79 %	714.73	96.78 %	98.64 %
4	1433.14	93.62 %	1517.56	93.62 %	105.89 %
8	3078.75	87.14 %	2945.36	87.18 %	95.67 %
16	6167.51	74.31 %	5823.29	74.29 %	94.42 %
32	12479.78	51.84 %	12100.80	51.11 %	96.96 %
48	16467.25	38.95 %	15916.20	37.40 %	96.65 %
64	22900.36	19.96 %	21391.74	19.33 %	93.41 %
80	26112.23	2.31 %	24814.63	1.87 %	95.03 %
100	25998.55	0.81 %	24689.46	0.74 %	94.96 %
128	25485.38	0.14 %	24366.55	0.14 %	95.61 %

2.3.4. 考察

TPS の絶対値の比較においては、クライアント数 4 の場合を除く全体において PostgreSQL 9.3 より 9.4 の方がわずかに TPS 値が低く、2～7%程度の低下が見られました。これには、9.4 で行われた何らかの変更が影響しているものと考えられます。

スケーラビリティの観点からは、PostgreSQL 9.3、9.4 の両方で、クライアント数の増加に伴い TPS も上昇し、クライアント数 80 までほぼリニアにスケールする結果を得ました。これにより、PostgreSQL9.4 が昨年度に検証した 9.3 と同傾向のスケーラビリティ特性を持っていることが確認されました。

また、今回のマシンは CPU コア数 60 にもかかわらず、TPS はクライアント数 80 までスケールしています。これは、クライアントマシンとデータベースサーバ間のネットワークレイテンシがある影響により、クライアント数が 64 までのベンチマークではサーバの CPU が使い切れていないのが原因と考えられます。実際に 64 クライアントの時の CPU idle は 20%程度であり、CPU 利用率は 80%程度である一方、80 クライアントの場合には CPU idle は 2%程度であり、CPU 利用率は 98%程度となっています。SQL 処理に必要な CPU 負荷が大きくなるほど，飽和するクライアント数はコア数（この場合は 60）に近くなると推測されます。

2.4. PostgreSQL 9.4: 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。

checksum は乗算処理が重いため、PostgreSQL ではページを 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 以降のソースコードの src/include/storage/checksum_impl.h にかかれています。また実際に PostgreSQL 9.3 以降では checksum 処理のソースコードである src/backend/storage/page/checksum.c をコンパイルするときに「 -funroll-loops -ftree-vectorize」オプションを使用するようになっています。

PostgreSQL 9.3 を用いた前年度の計測では、同時接続クライアント数が少ないときには、page checksum の有効/無効で性能に大差はなく、接続数が多くなり CPU 負荷が高まると page checksum のオーバヘッドが現れてくるという結果が得られました。また、接続数が少ない時には page checksum 有効時にコンパイラオプションを用いない場合よりも、コンパイラオプションを用いた場合の方がよい TPS が出たという結果も得られています。

本検証では PostgreSQL 9.4 を用いてこの結果が再現できるかどうかを確認しました。

2.4.2. 検証構成

検証構成は PostgreSQL 9.3 と 9.4 の参照性能の比較と同じです。

2.4.3. 検証方法

2.4.3.1. 環境

コンパイラオプションを使用する場合と使用しない場合の 2 通りの条件でバイナリを作成します。

通常はソースコードをそのままコンパイルすることで、上述のコンパイラオプションが有効になります。

このコンパイラオプションを使用しない場合は、src/backend/storage/page/Makefile 中の CFLAGS に CFLAGS_VECTOR を追加している部分の行をコメントアウトしてからコンパイルを行います。

```
$ vi src/backend/storage/page/Makefile

    # important optimizations flags for checksum.c
  - checksum.o: CFLAGS += ${CFLAGS_VECTOR}
  + #checksum.o: CFLAGS += ${CFLAGS_VECTOR}

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

この CFLAGS_VECTOR の中身は src/Makefile.global で定義されており、以下の通りです。

```
CFLAGS_VECTOR =  -funroll-loops -ftree-vectorize
```

page checksum を有効にするために「--data-checksums」オプションをつけて initdb します。

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

その後のベンチマーク用データベースの作成までは PostgreSQL 9.3 と 9.4 の参照性能の比較と同じです。

2.4.3.2. 測定

測定方法は PostgreSQL 9.3 と 9.4 の参照性能の比較と同じです。

2.4.3.3. 結果

結果をグラフと表に示します。グラフの横軸は同時接続クライアント数、縦軸は TPS です。コンパイルオプションありの checksum あり条件（緑、× マーク）は、コンパイラオプション無し条件（青、＊ マーク）、checksum なし条件（赤、＋ マーク）の3条件の結果を以下に示します。全体的に同程度の TPS が得られました。表の CPU idle 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。

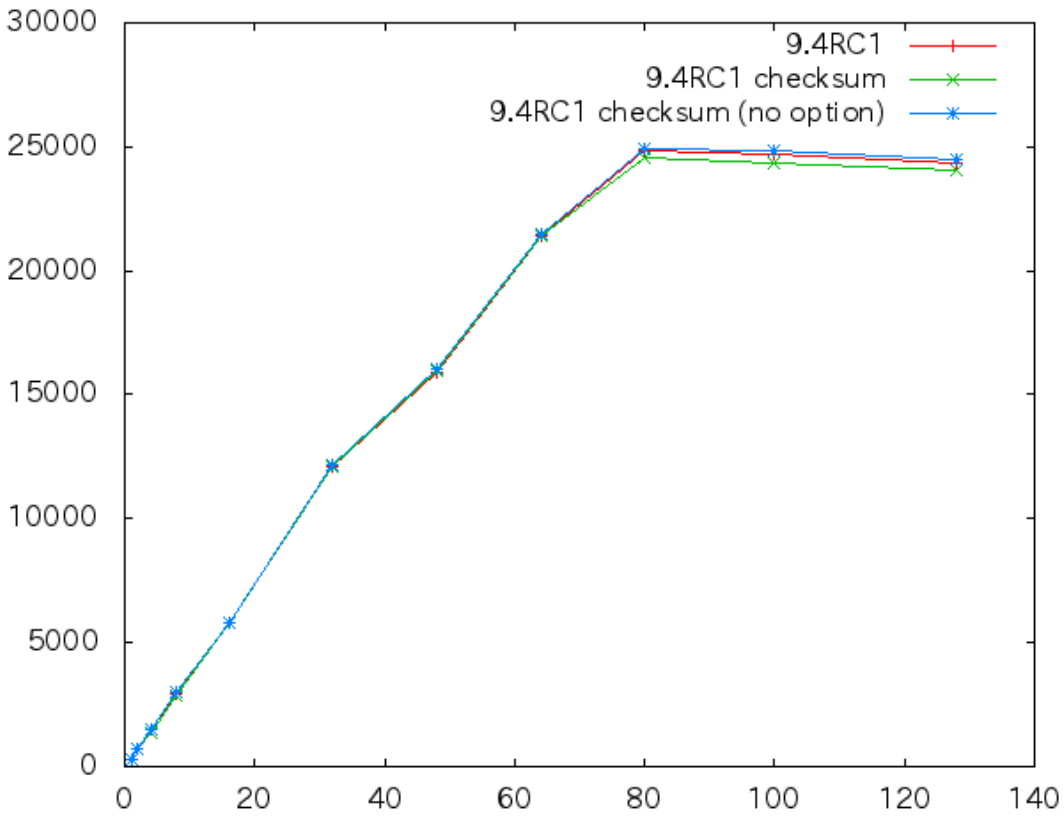

図 2.4.1: PostgreSQL 9.4: page checksum 有無による参照性能の比較

表 2.4.1: PostgreSQL 9.4: page checksum 有無による参照性能の比較

クライアント数	PostgreSQL 9.4: page checksum なし		PostgreSQL 9.4: page checksum あり（コンパイルオプションあり=デフォルト）			PostgreSQL 9.4: page checksum あり（コンパイルオプションなし）		
	TPS	CPU idle	TPS	CPU idle	checksum なしとの TPS の比較	TPS	CPU idle	checksum なしとの TPS の比較
1	304.03	98.35 %	300.44	97.40 %	98.82 %	297.44	98.35 %	97.83 %
2	714.73	96.78 %	699.74	96.79 %	97.90 %	701.83	96.77 %	98.20 %
4	1517.56	93.62 %	1384.72	93.59 %	91.25 %	1487.00	93.64 %	97.99 %
8	2945.36	87.18 %	2828.98	87.15 %	96.05 %	2977.82	87.13 %	101.10 %
16	5823.29	74.29 %	5828.26	74.30 %	100.01 %	5815.23	74.25 %	99.86 %
32	12100.80	51.11 %	12116.31	50.85 %	100.13 %	12156.27	50.83 %	100.46 %
48	15916.20	37.40 %	15965.05	37.17 %	100.31 %	16023.67	37.09 %	100.68 %
64	21391.74	19.33 %	21424.53	19.24 %	100.15 %	21482.43	19.10 %	100.42 %
80	24814.63	1.87 %	24522.98	2.03 %	98.82 %	24938.98	1.81 %	100.50 %
100	24689.46	0.74 %	24341.94	0.85 %	98.59 %	24810.29	0.76 %	100.49 %
128	24366.55	0.14 %	24085.24	0.16 %	98.85 %	24464.52	0.14 %	100.40 %

2.4.4. 考察

全条件で TPS にほとんど差がなかったことから、PostgreSQL 9.4 では page checksum の参照性能に与える影響は無視できるほど小さいと言えます。

3. WAL 改善

3.1. 検証概要

PGECons では、pgbench を用いた参照系のスケールアップ検証を定点観測として 2012 年度(PostgreSQL 9.2.x)から各バージョンの検証を行い、検証結果の公開を行ってきました。2014 年度の WG1 の検証を実施するに当たり、テーマの検討の中で参照系の定点観測だけでなく、更新系のスケールアップ検証も実施してみてはどうかという議論が行われ、参照系と同様に pgbench を用いて検証を行うこととなりました。

また、検証対象である最新のバージョン PostgreSQL9.4 では、WAL(Write Ahead Log)データ圧縮によるファイル出力量の軽減、WAL 出力のロック競合の軽減による並列書込み性能の改善という二つの更新系の性能改善が実装されています。WAL 出力量の軽減については、別の検証にて圧縮により WAL のデータ量は 84～94%になるとの検証報告[1]がありますので、そちらをご参照ください。

これらのスケールアップ状況と PostgreSQL9.4 の性能改善状況を確認するため、比較的単純な 1 つの表に対する更新を行う pgbench でのカスタムスクリプトによる更新処理を行う方法で検証を行います。

3.2. 検証目的

大規模な物理マシン環境にて、pgbench による単純な更新を実施した際の性能を測定し、以下の 2 つの内容を確認することを目的とします。

1. PostgreSQL の更新系の CPU スケーラビリティの達成状況を確認する
2. PostgreSQL9.3.5 と PostgreSQL9.4RC1 を比較し、更新性能の改善状況を確認する

なお、更新系の検証を実施する際に、一貫性を保証せず性能を改善するような、一般的に本番システムに適用しないパラメータは設定しないこととします。

3.3. 検証構成

3.3.1. 測定ハードウェア構成

主なスペックと構成は以下のとおりです。

表 3.3.1: 検証構成

機器	項目	仕様
クライアント(pgbench)用マシン	CPU	インテル® Xeon® プロセッサ E5-2650v3 (10 コア) x 2 合計 20core
	搭載メモリ	128GB
	内蔵ストレージ	HDD: 600GB 12G SAS 15k x 4 RAID10
	OS	Red Hat Enterprise Linux 6.5 (for Intel 64)
	クライアント	PostgreSQL 標準 pgbench
PostgreSQL 用サーバ	CPU	インテル® Xeon® プロセッサ E7-4890v2 (15 コア) x 4 合計60 core
	搭載メモリ	2TB
	内蔵ストレージ	1.2TB 6G SAS 10k x 10 RAID10
	DB 格納用ストレージ	Fiber Channel 接続(16Gb) SAN データベースクラスタ(/data)と WAL 領域(pg_xlog)に以下の RAID 10構成(実効容量約 3TB)を利用

1 SRA OSS, Inc. 日本支社, PostgreSQL 9.4 検証レポート 1.0.2 版(P.10～11),
http://www.sraoss.co.jp/technology/postgresql/images/20140808_PostgreSQL9.4_report_sraoss.pdf

機器	項目	仕様
		(900GB 6G SAS 10K 2.5inch 8 台)
	OS	Red Hat Enterprise Linux 6.5 (for Intel 64)
	DBMS	PostgreSQL 9.3.5 または PostgreSQL 9.4RC1

詳細な設置状況については、2014 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。

3.3.2. pgbench の利用設定

pgbench の対象テーブルサイズを約 100GB とするため、スケールファクタは 7000、また UPDATE 文実行時の HOT の効果を期待する為に FILLFACTOR は 80 として初期設定を行っています。

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

また、ブロックの排他処理が集中しないようにサイズの大きい 1 表(pgbench_accounts)への基本的な更新方法とすること、なおかつ更新の際に少し CPU を使う（演算）を行うことを含めることとしたいと考え、トランザクションを以下のように設定しました。

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

UPDATE pgbench_accounts
SET filler=repeat(md5(current_timestamp::text),2)
WHERE aid= :aid;
```

3.3.3. postgresql.conf 設定値

postgresql.conf のパラメータは以下のようにしました。
checkpoint_segments については、極端な値にしています。これは、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為に、pgbench による測定実行時には CHECKPOINT を実施しない方法とすることにしました。
また、vacuum の影響を排除するため autovacuum は off としています。
wal_level については、本番システムで設定されることが推奨される archive に変更しています。

```
max_connections = 1000
shared_buffers = 384GB      # およそサーバ搭載容量の 20%
wal_level = archive
checkpoint_segments = 100000
checkpoint_timeout = 60min
maintenance_work_mem = 20GB
log_checkpoints = on
log_line_prefix ='%t %p %a'
log_lock_waits = on
autovacuum = off
```

3.4. 検証方法

3.4.1. 測定スクリプト

pgbench による性能測定を簡便かつ確実に実施するため、測定スクリプト群を用意しました。主な操作は、DB データ作成と性能測定の実施であり、これらはベンチマーク・クライアントに配置する sh スクリプトが、DB サーバに配置した sh スクリプトやクライアント上の pgbench プログラムを用いて行います。また DB サーバのコア数を変更してのベンチマーク測定を、手作業を介することなく実施するため、コア数を指定して DB サーバを再起動する sh スクリプトも用意しました。この再起動スクリプトを実行するには、DB サーバの root 権限が必要です。

DB サーバに配置する PostgreSQL 関連のスクリプトを表 3.4.1 に示しています。これらは、環境変数 PGDATA が設定されていることを動作の前提としています。このため、.bashrc にて環境変数 PGDATA を設定した上で export しておく必要があります。なお、pgsql-start-db と pgsql-stop-db は、DBT2 や DBT5 ベンチマークのスクリプトを流用しました。

表 3.4.1: DB サーバに配置する PostgreSQL 関連のスクリプト

スクリプト名	処理内容
pgsql-install	/usr/local/src から PostgreSQL を install する
pgsql-init-db	$PGDATA の内容を initdb により初期化する
pgsql-start-db	PostgreSQL を起動する
pgsql-stop-db	PostgreSQL を停止する
pgsql-save-db	$PGDATA にある DB データを tar ファイルに保存する
pgsql-reload-db	tar ファイルに保存された DB データを$PGDATA に戻す
pgsql-change-conf	指定したファイルを$PGDATA/custom.conf にコピーする
pgsql-clear-db	$PGDATA にあるファイルを消去する（pg_xlog は別扱い）
pgsql-kill	PostgreSQL のプロセスを強制終了させる

クライアントから行う DB データ作成（initialize_pgbench）と性能測定実施（do_pgbench）が行う処理の概要を図 3.4.1 と図 3.4.2 に示しています。DB データ作成スクリプトでは、$PGDATA にベンチマーク実行に必要な DB を作成して tar ファイルとして保存しておき、性能測定実施スクリプトでは、その tar ファイルの展開と config の設定を行ってから PostgreSQL を起動し、ベンチマーク測定を実行します。この時の一回のベンチマーク測定で行う操作を図 3.4.3 に示しています。今回の測定では 5 分間の pgbench 実行を 3 回行うようにしており、pgbench 実行の 1 回目と 2 回目の間、および、2 回目と 3 回目の間には checkpoint を実行させました。また、CPU や disk の使用状況を測定するため、各測定の最初に sar を起動させました。sar の測定時間間隔は 10 秒、計測回数は 29 回（測定時間 300 秒に対応）です。

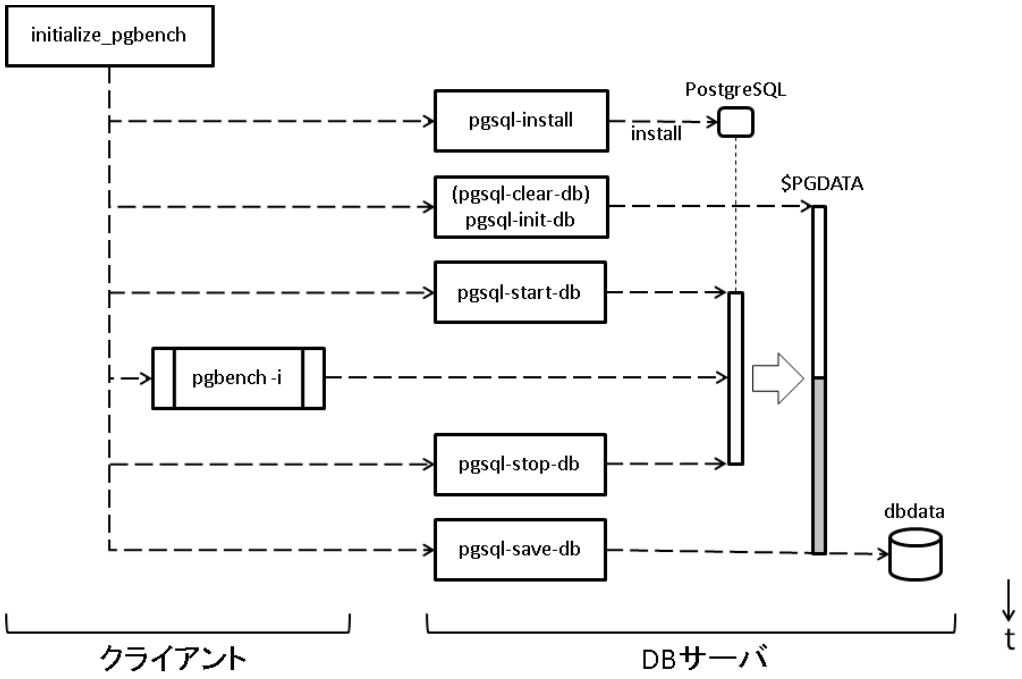

図 3.4.1: DB データ作成の流れ

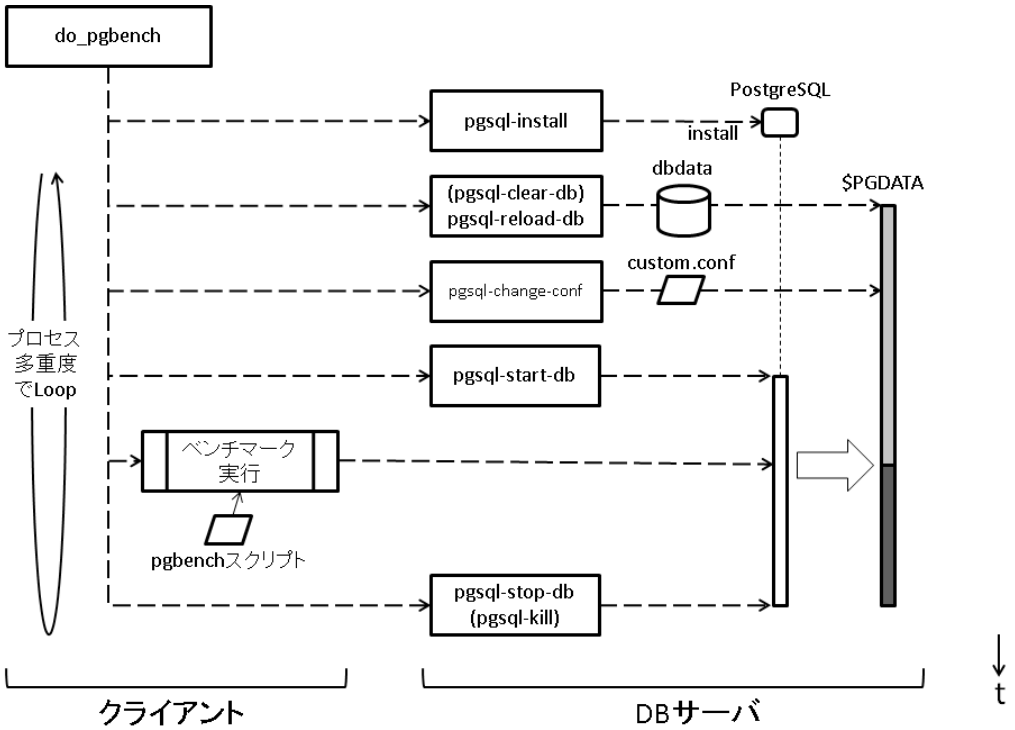

図 3.4.2:性能測定実施の流れ

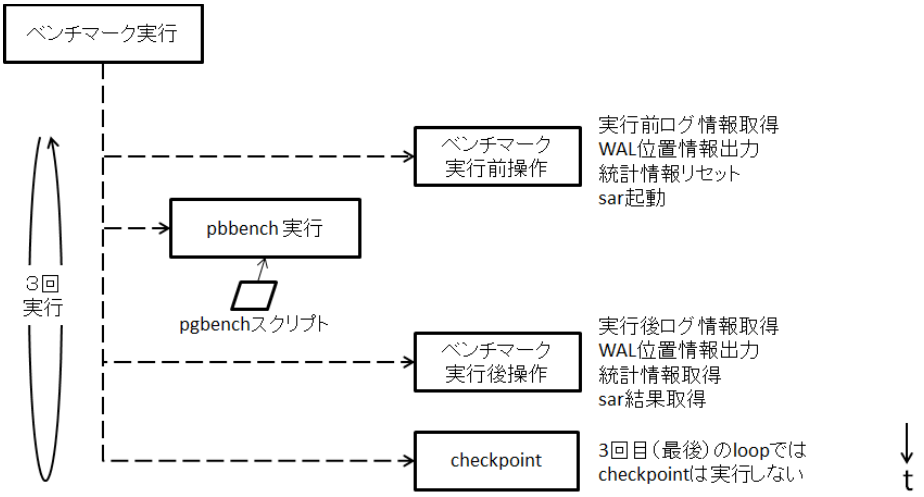

図 3.4.3:ベンチマーク実行の全体の流れ

pgbench の実行条件(実行パラメータ)は、do_pgbench の引数および環境変数で指定する方法にしました。両者の使い分けは精査した上で決めたわけではありませんが、変更されることが多い実行パラメータは引数として与え、それ以外のパラメータは環境変数で与えるようにしました。なお、ベンチマーク実行時間、スケールファクタ、pgbench のプログラム名、データベース名、sar の測定間隔は固定値としました。

表 3.4.2: do_pgbench コマンドのオプション

指定方法	実行パラメータ
第一引数	実行 ID　(実行結果を格納する directory 名の一部となる)
第二引数	プロセス多重度　複数の多重度について測定する場合は、'10 20 30' のように指定する
CONF	$PGDATA/custom.conf にコピーするファイル名
SCRIPT	pgbench のカスタムスクリプト

最後に、コア数を指定して DB サーバを再起動するスクリプトを紹介します。DB サーバ側には図 3.4.4 の sh スクリプト(REBOOT)と起動 config ファイルを用意した上で、クライアント側の sh スクリプト(関数)を実行すると、指定したコア数で DB サーバが再起動されます。この関数と性能測定実行の sh スクリプト組み合わせることで、CPU コア数を含む実行パラメータを変更しての一連の性能測定を一回のスクリプト起動で実行することができます。

サーバ側スクリプト

```
#!/bin/bash

# DIR=/boot/efi/EFI/redhat (for efi)
DIR=/boot/grub

sed -e "s/#CPUNUM#/${1:-1}/" $DIR/grub.conf_origin > $DIR/grub.conf
reboot
```

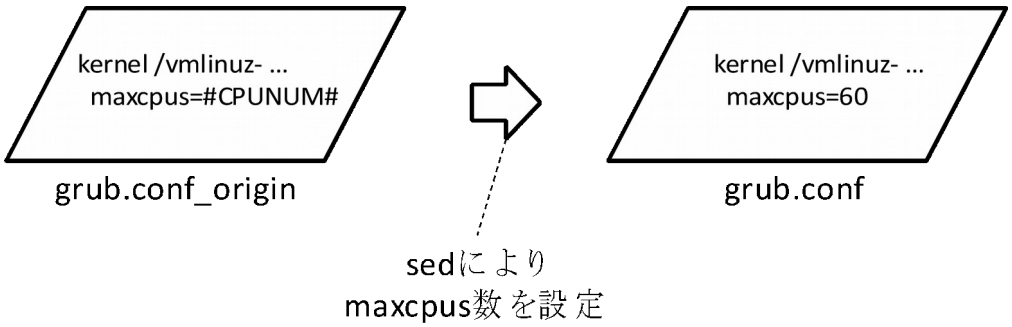

図 3.4.4:PostgreSQL サーバの CPU コア数の設定方法

クライアント側スクリプト

```
rebootDBserver()
{
    ssh $DBHOST -l root bin/REBOOT $1
    sleep 150
    while true; do
        ping -c 1 $DBHOST
        RETVAL=$?
        if [ $RETVAL -eq 0 ]; then
                break
                fi
        sleep 1
    done
    sleep 60
    if [ `ssh $DBHOST grep -c processor /proc/cpuinfo` != $1 ]
        then echo  “error:  ¥# of CPU cores is not $1. “
    fi
}
```

3.4.2. lwlock 競合の測定方法および結果集計方法

pgbench により PostgreSQL の書き込み性能を測定した所、CPU、ストレージ、ネットワークといった物理資源が飽和することなく、スループット性能が飽和しました（3.5.1 項、3.6.1 項参照）。ここでのボトルネックは、PostgreSQL 内部で行われている排他制御に起因している可能性が疑われたため、軽量ロック（lwlock）[2]の競合状況の測定を実施しました。

この測定は、競合状況測定機能を有効にして build したバイナリ・プログラム（以下、バイナリ）を用いることから、3.4.1 項の計測で用いたバイナリとは別となります。バイナリ作成方法の違いは、PostgreSQL のソース・コードから build する際の configure にて CFLAGS=-DLWLOCK_STATS を指定する点のみで、それ以外の build 方法およびベンチマーク実行方法は 3.4.1 項に示した手順と同じです。なお、結果のデータ量が膨大になるのを避けるため、競合が発生していない、もしくは、軽微であった lwlock については結果を出力しないよう、結果出力処理部分のソース・コードを改変しましたが、通常の処理を行う部分についてはオリジナルのソース・コードから変更を加えていません。

上記により作成した PostgreSQL バイナリにおいて、各バックエンド・プロセスが実行を終了すると、PostgreSQL のログ（pgstartup.log）に lwlock 競合に関する測定結果が出力されます。lwlock の管理方法は PostgreSQL 9.3.5 と 9.4RC1 で異なっているので、出力される結果フォーマットも異なっています。各々について、結果の例およびその意味を図 3.4.5 と図 3.4.6 に示します。PostgreSQL 9.3.5 では、lwlock の種類（排他制御する対象に対応）は番号で示され

2 PostgreSQL の内部処理（ワーカ・プロセスやサーバ・プロセス）間で排他制御を行うために用いられるロックのことです。SQL 文で明示的に指定する重量ロック（Heavy Weight Lock）とは異なり、ユーザ（クライアント・プログラム）から意識されることはありません。

ており、その対応は、ソース・コード（src/include/storage/lwlock.h）で定義されている enum LWLockId から判明します。一方、PostgreSQL 9.4RC1 では、lwlock の種類は、LWLockTranche と番号という 2 つの情報で示されますが、競合が発生する LWLockTranche は、PostgreSQL9.3.5 の lwlock に対応する main (tranche)と WALInsertLocks (tranche)です。main (tranche)については、lwlock 番号の意味は PostgreSQL9.3.5 と同様、ソース・コード（src/include/storage/lwlock.h）で定義されています。一方、WALInsertLocks (tranche)については、lwlock 番号は WAL バッファへの挿入を排他制御する複数のロックを識別するための番号であり、lwlock の役割は同じとなっています。

```
PID 6198 lwlock 0: shacq 0 exacq 20737 blk 4150 spindelay 6
PID 6198 lwlock 3: shacq 0 exacq 18150 blk 1025 spindelay 6
PID 6198 lwlock 4: shacq 36301 exacq 18152 blk 19130 spindelay 93
PID 6198 lwlock 7: shacq 0 exacq 39471 blk 16978 spindelay 44
PID 6198 lwlock 8: shacq 0 exacq 0 blk 32275 spindelay 0
PID 6198 lwlock 11: shacq 3171 exacq 22401 blk 4193 spindelay 64
PID 6198 lwlock 33: shacq 8238 exacq 1353 blk 208 spindelay 0
PID 6198 lwlock 34: shacq 5636 exacq 1267 blk 136 spindelay 0
PID 6198 lwlock 35: shacq 4952 exacq 1283 blk 148 spindelay 0
（中略）
PID 6198 lwlock 46: shacq 7317 exacq 1289 blk 163 spindelay 0
PID 6198 lwlock 47: shacq 5225 exacq 1363 blk 121 spindelay 1
PID 6198 lwlock 48: shacq 24845 exacq 1285 blk 475 spindelay 4
```

① ②　③　④　⑤　⑥

① :バックエンド・プロセスのプロセス番号
② :lwlock番号
③ :共有モードでのロック要求回数
④ :排他モードでのロック要求回数
⑤ :lwlockで競合が発生した回数
⑥ :lwlockの状態変数へのアクセスで発生したspin lock競合の回数

図 3.4.5:PostgreSQL 9.3 の lwlock

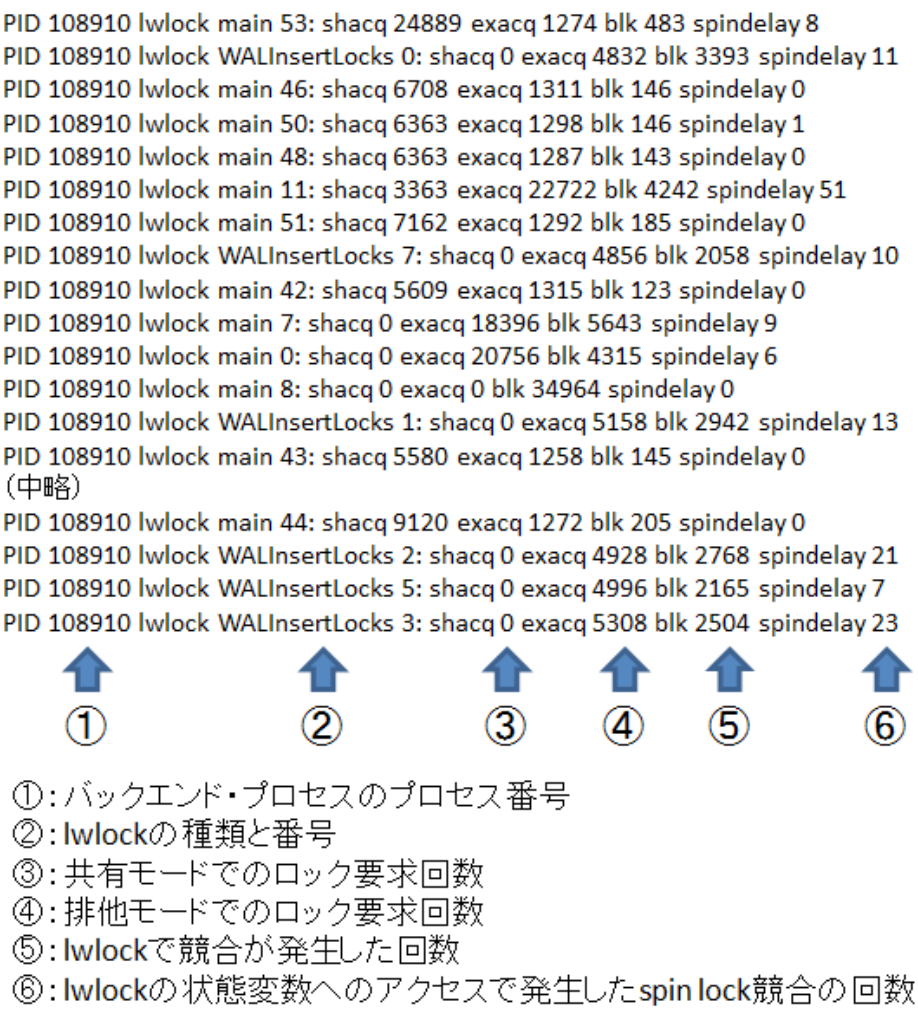

図 3.4.6:PostgreSQL 9.4 の lwlock

結果の集計では、1 回のベンチマーク実行に関わる総てのバックエンド・プロセスについて、shacq(③)、exacq(④)、blk(⑤)回数を lwlock 番号別に合計しました。PostgreSQL9.4RC1 の WALInsertLocks (tranche)については lwlock 番号で区別することなく全体を WALInsertLocks として各回数の合計を求め、main (tranche)は PostgreSQL9.3.5 と同様、lwlock 番号別に合計を求めました。図 3.6.6 と図 3.6.7 は、この結果とベンチマーク実行期間中に処理したトランザクション数の結果から求めた 1 トランザクション当たりのロック要求回数およびロック競合回数を示します。なお、今回の検証では，spindelay は参考に留め、本報告では明示をしていません。

3.5. 検証結果

3.5.1. 性能測定結果と傾向について

性能測定の結果を示します。なお、結果の値については 3 回の測定の中央値を採用しています。
PostgreSQL 9.3.5 の結果は以下の図 3.5.1 の通りです。

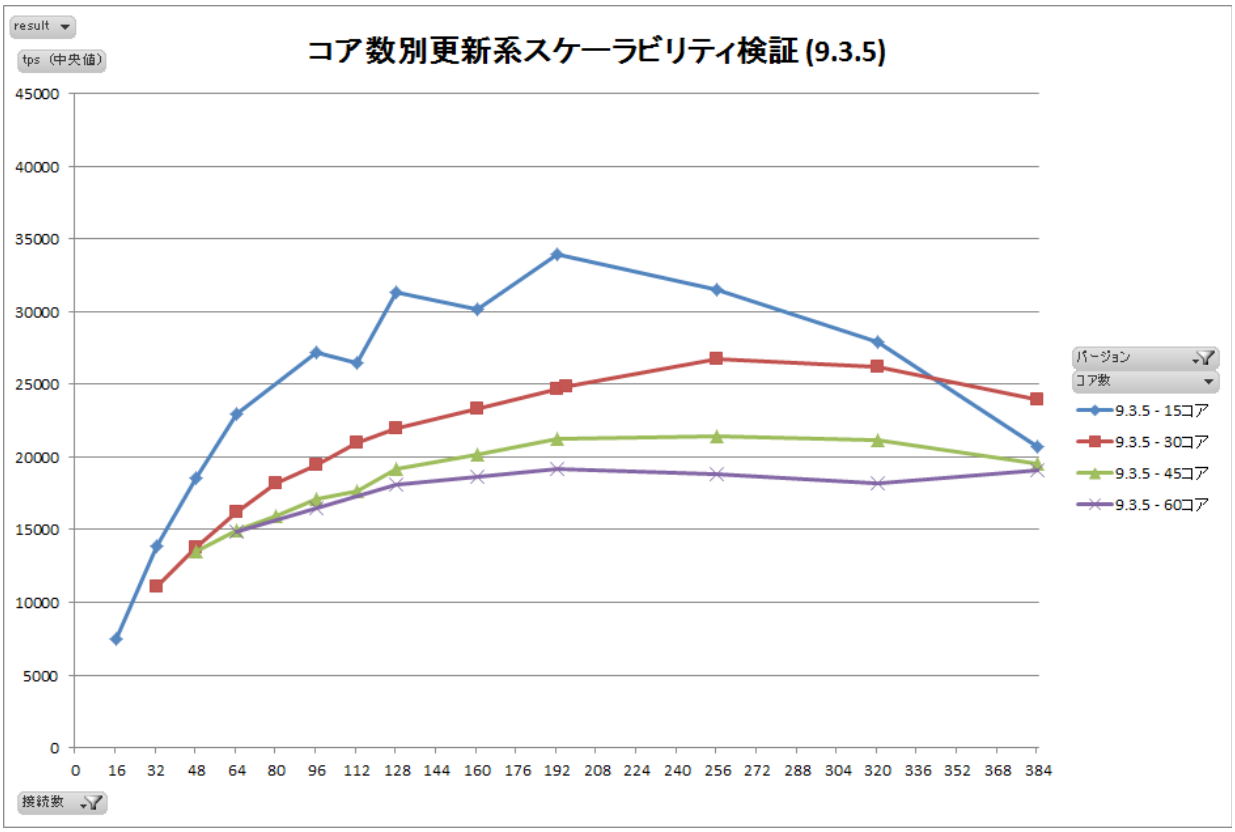

図 3.5.1:PostgreSQL9.3.5 の更新スケーラビリティ検証

PostgreSQL9.3.5 では 15 コアのケースが全般的に性能が高く、コア数が増えれば増えるほど全体的な性能は下がっていく傾向となりました。全ての接続およびコア数の中でのピーク値は 15 コア 192 接続時の約 34,000(tps)を確認しました。コアごとの接続数によるピーク値はまちまちですが、参照系のスケーラビリティ検証とは異なりディスク I/O の待ちが発生することから、コア数よりも多くの接続数でピーク値が確認できました。

また、全体で測定を行った 64 接続から 320 接続の間ではコア数が少なければ少ないほど更新性能が高いという結果になりました。このことから、PostgreSQL9.3.5 では今回行った大きい 1 表を単純に更新するケースにおいては、CPU スケーラビリティを確認することができませんでした。

次に PostgreSQL 9.4RC1 での測定結果を以下の図 3.5.2 に示します。

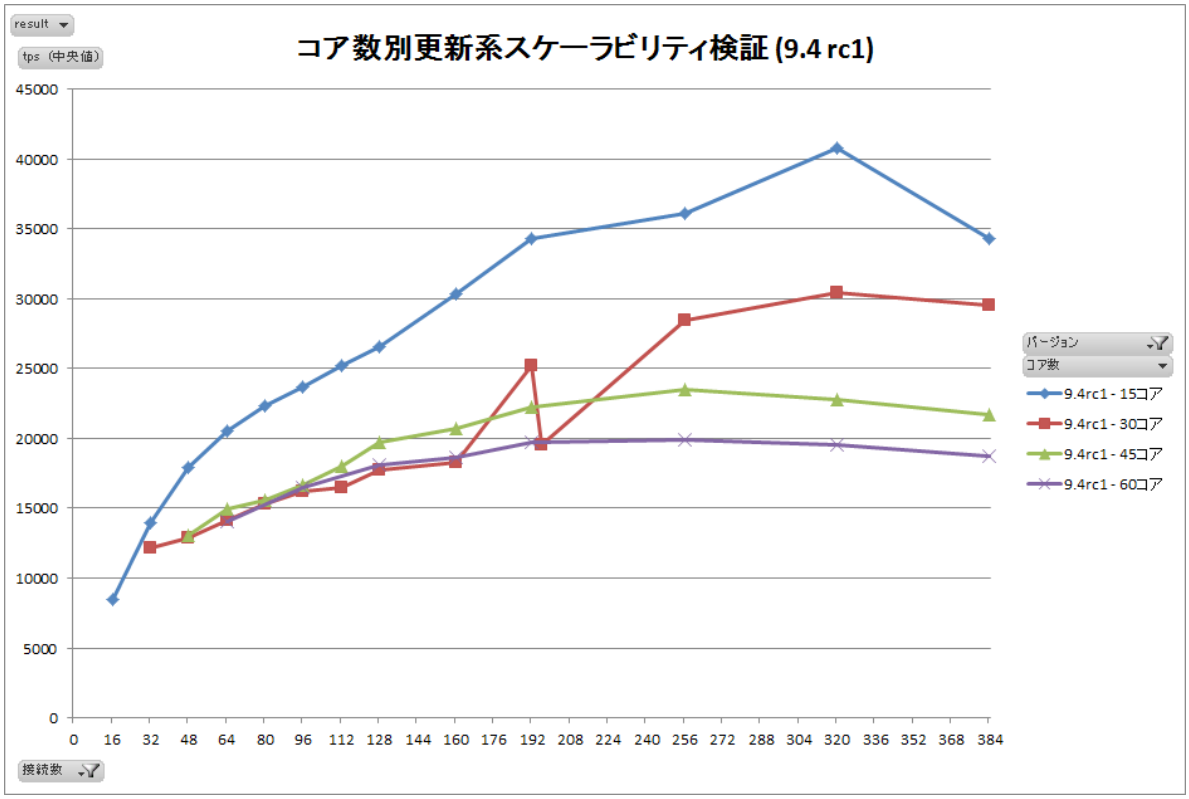

図 3.5.2:PostgreSQL9.4RC1 の更新スケーラビリティ検証

PostgreSQL9.4 RC1 でも 15 コアのケースが全般的に性能が高く、コア数が増えれば増えるほど全体的な性能は下がっていく傾向となりました。しかし、全ての接続およびコア数の中でのピーク値は 15 コア 320 接続時の約 41,000(tps)となり、PostgreSQL9.3.5 と比較して性能が向上しています。

また、PostgreSQL9.4RC1 でも、15 コアが性能が一番よいことから、今回の測定シナリオでの CPU スケーラビリティを確認することができませんでしたが、30 コア以上での接続数が少ない（48 接続から 160 接続の間）では、30 コアより45コア、60コアの性能が良いケースも確認できました。

次に PostgreSQL9.3.5 と PostgreSQL9.4RC1 の比較を行うため、15 コアと60コアでのグラフを図 3.5.3 に記載しています。

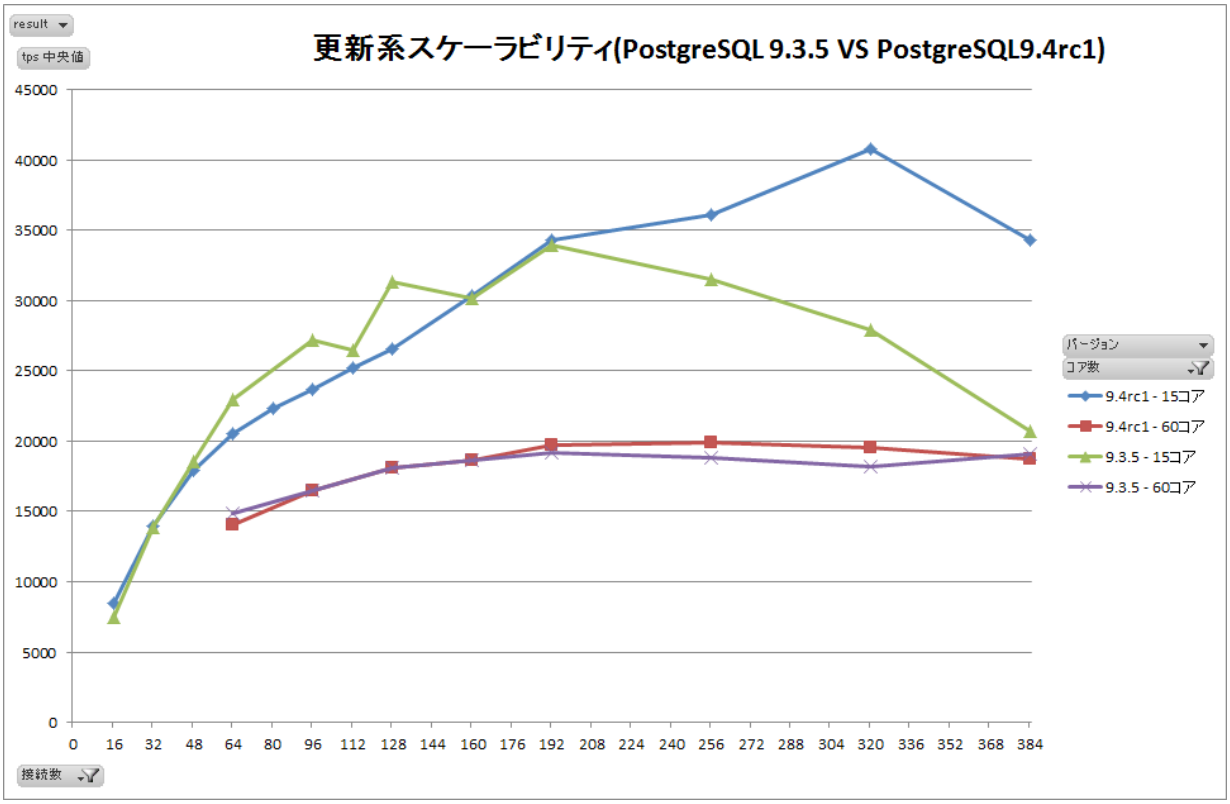

図 3.5.3:PostgreSQL 9.3.5 と PostgreSQL9.4RC1 の更新性能比較

まず 15 コアでの測定結果を比較してみると、192 接続までは PostgreSQL9.3.5 のほうが性能が少し高いところもありますが、おおよそ、PostgreSQL9.4RC1 と PostgreSQL9.3.5 が同じ傾向でスケーラビリティ性が確認できています。

192 接続を超えると PostgreSQL9.3.5 では性能が低下しているのに対して、PostgreSQL9.4RC1 では接続数 320 まで全体性能が向上していることが確認できました。同じ 384 接続で確認してみると、PostgreSQL9.4RC1 が PostgreSQL9.3.5 から約 1.7 倍の性能を発揮しています。このことから、特に高い負荷（接続数）がかかったケースでは PostgreSQL9.4RC1 のほうが PostgreSQL9.3.5 より更新性能が高いことが確認できました。

60 コアについては、全体として 15 コアのケースより性能が低く、192 接続をピークにピーク値が 20,000TPS 付近を示しているという、PostgreSQL9.3.5 と PostgreSQL9.4RC1 ではほぼ同じ傾向となりました。

3.6. 考察

PostgreSQL9.4RC1 と PostgreSQL9.3.5 のスケールアップの違いや、CPU スケールアップが確認できなかった点などについて、システムの稼働情報（sar）出力結果や lwlock の測定結果から、検討を行いました。

3.6.1. システム利用状況の確認

pgbench 稼働中の CPU とディスク I/O 稼働状況を確認し、これらによるボトルネックが発生をしていないかどうかを確認しました。

ここでは PostgreSQL9.4RC1 での 15 コアでのピーク値と、60 コアでのピーク値の状況について記載します。

まず、15 コアのピーク値である 320 接続の CPU とディスクの利用状況を図 3.6.1 に示します。

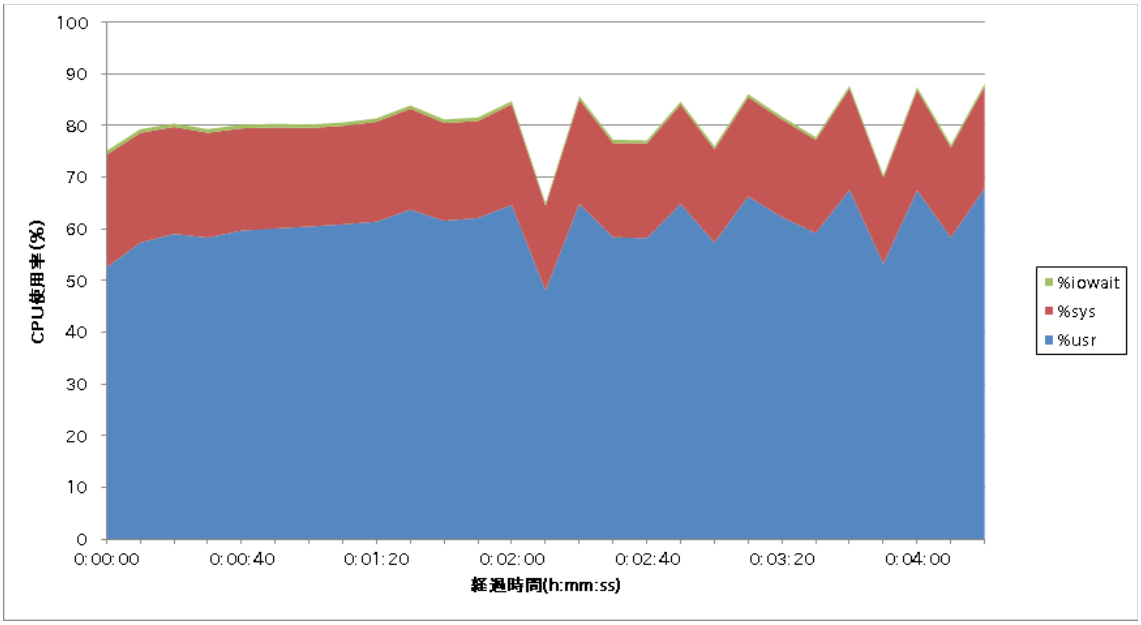

図 3.6.1:PostgreSQL 9.4RC1 15 コア 320 接続での pgbench 計測中の PostgreSQL サーバの CPU 使用状況

15 コアの CPU 稼働状況は上記の通りで、80%とかなり高い CPU 利用率とはなっていますが、100%までには到達していませんので、CPU によるボトルネックである可能性は低いと判断しています。

また、データベースクラスタ(/data)の I/O と WAL(pg_xlog)の%util を図 3.6.2 を示してあります。pgbench 中は CHECKPOINT が動作をしないように設定した意図の通り、データベースクラスタについては I/O がほとんど確認できていません。これは PostgreSQL サーバマシン中のバッファにすべての DB 領域が格納されていることを示しています。また、更新系での負荷が高くなりやすい WAL(pg_xlog)の I/O 状況については、%util のピーク値が 80%と稼働負荷自体は高くなっていますが、ストレージによるボトルネックが発生しているとは言えない状況です。

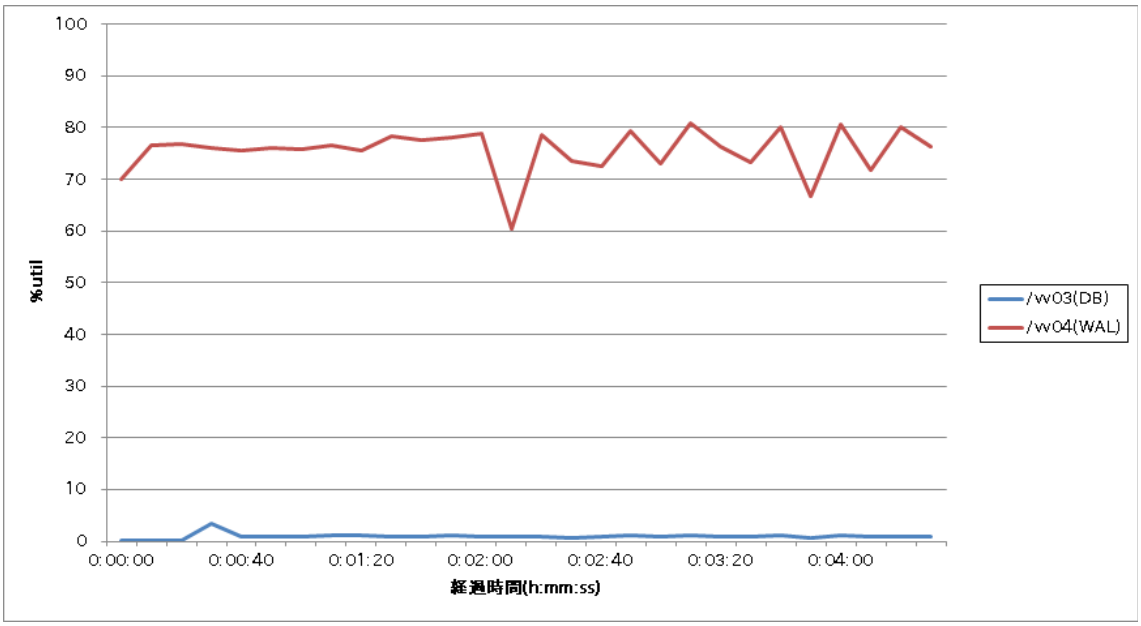

図 3.6.2:PostgreSQL 9.4RC1 15 コア 320 接続での pgbench 計測中のストレージ使用状況(DB クラスタと WAL)

つぎに60コア時のピーク時である 256 接続の状況です。

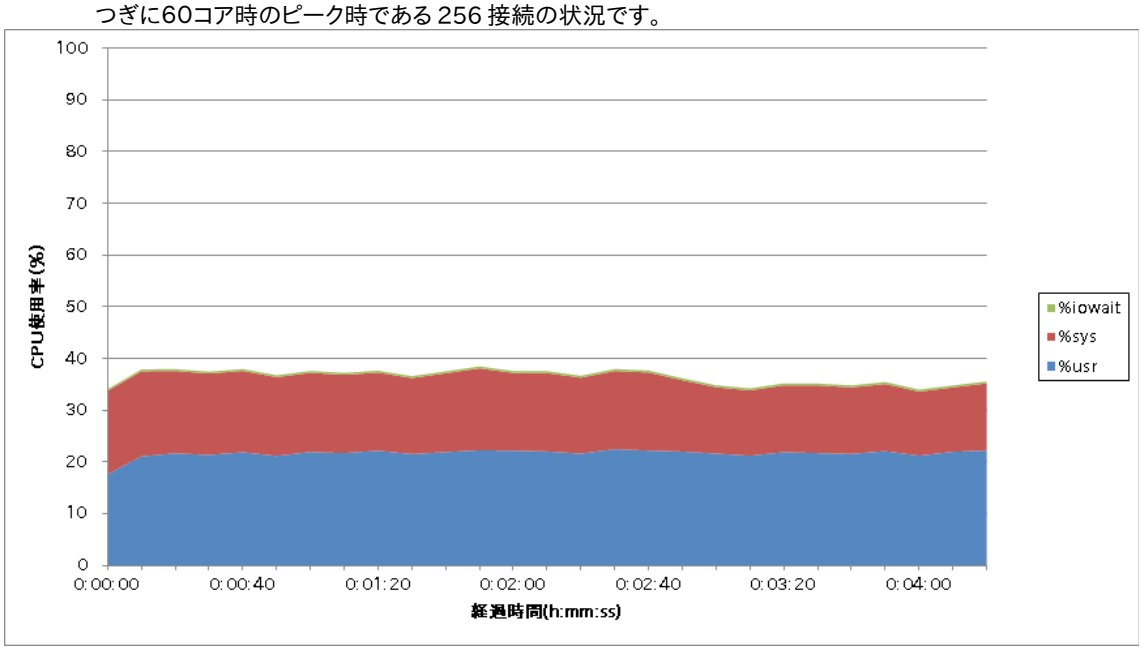

図 3.6.3:60 コア 256 接続での pgbench 計測中の PostgreSQL サーバの CPU 使用状況

60 コアの CPU 稼働状況は図 3.6.3 の通りで、全体の CPU 利用率は 35%程度であり、まだまだ CPU リソースに十分な余裕があることが確認できます。

ストレージ状況については、図 3.6.4 に記載しています。15 コアでも紹介した図 3.6.2 より 10%程度低く%util が出力されていますので、こちらについてもストレージによるボトルネックであるとは断言できる状態ではありません。

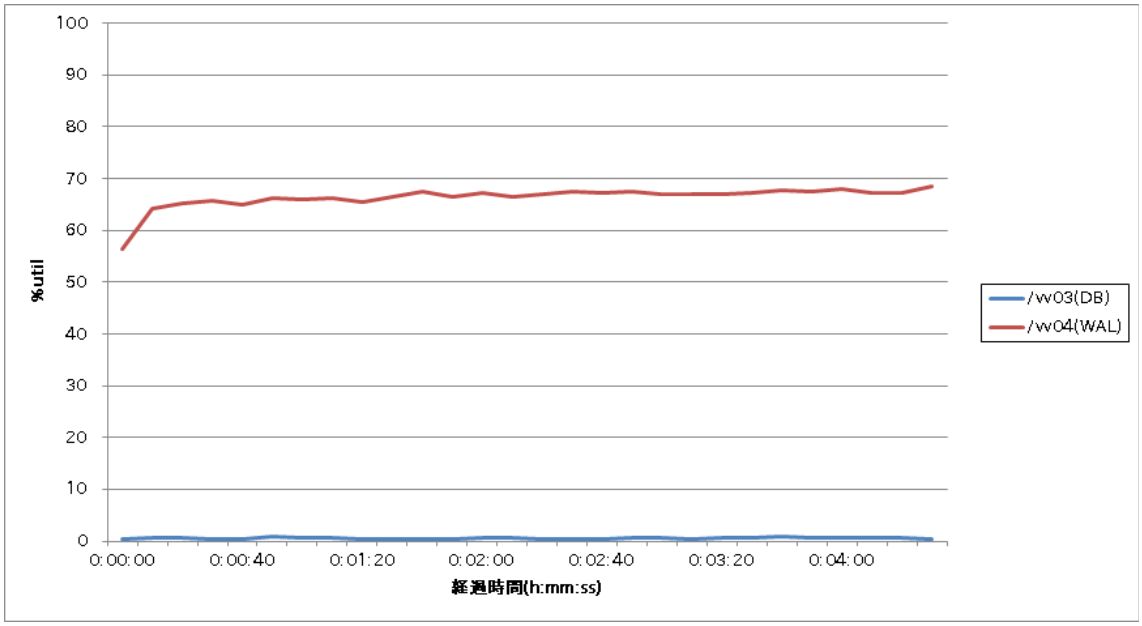

図 3.6.4:PostgreSQL 9.4RC1 60 コア 256 接続での pgbench 計測中のストレージ使用状況 (DB クラスタと WAL)

なお、PostgreSQL9.3.5 と PostgreSQL9.4RC1 についてはピーク値の接続数の条件が異なるため、一概には言えませんが、PostgreSQL9.3.5 のピーク値でもおおよそ CPU 使用率やディスクの使用率の傾向に大きな変更はありませんでした。

以上のグラフの結果から、CPU やストレージによるボトルネックが確実に発生しているとは言えないことがわかっています。そこで次にロックの状況を解析しました。

3.6.2. lwlock 要求回数および競合回数

1. 測定条件と集計方法

   lwlock 競合に関する測定は、コア数が 15、30、45、60 の 4 ケースで実施しました。本節では、結果を簡明に示すため、ロック要求回数およびロック競合回数を pgbench カスタム・スクリプトの1トランザクション（=update 1回）当りの回数に換算して示しました。また、各コア数についての結果比較を容易とするため、全コア数ともプロセス多重度が 256 の結果を示しています。測定手順では 5 分間の pgbench 実行を 3 回行いますが、最後の実行分については、ログ出力が未完了の状態で PostgreSQL が強制終了されていたケースがあったので、本節では、ベンチマーク実行の 1 回目と 2 回目を集計対象としました. 各ベンチマーク実行とも、プロセス多重度は 256 であることから、256 プロセス分の shacq、exacq、blk 回数を 3.4.2 に示した通り、lwlock 番号別に合計しました。また、ベンチマーク実行で処理したトランザクション数は、pgbench が出力する結果ファイルから取得しました。

   参考までに、ロック要求回数と競合回数の計測を行ったときのスループットを図 3.6.5 に示しています。測定オーバヘッドの存在および各コア数での最大スループットではないことから、前節に示したの結果とは若干異なってはいますが、ほぼ同様の傾向を示しています。

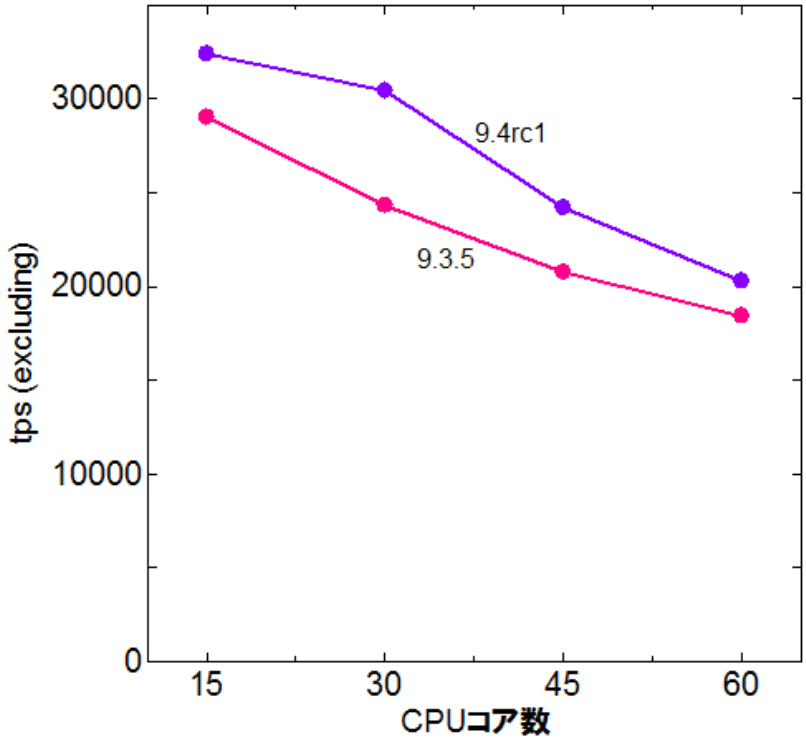

図 3.6.5:ロック要求回数と競合回数の計測時のスループット

2. ロック要求回数
   図 3.6.6 に、ロック要求回数の測定結果を示します。横軸は CPU コア数であり、左側に 9.3.5、右側に 9.4RC1 の結果を積み上げ棒グラフとして図示しました。ProcArrayLock および CLogControlLock では、排他（Exclusive）と共有（Shared）の両モードでロック要求が行われたので、これらを区別して図示しています。また、図に示していない lwlock は、無視できる程度のロック要求回数であったので、図からは省いています。図より、

   1. 9.4RC1 で制御方法が改良された WALInsertLock と WALBufMappoingLock では 9.3.5 と 9.4RC1 でロック要求回数に差がある
   2. CPU コア数の増加に伴い、9.4RC1 では WALBufMappingLock の要求回数が増えているのに対し、9.3.5 では、逆に、WALInsertLock の要求回数に減少傾向がみられる
   3. それ以外のロックについては、9.3.5 と 9.4RC の差異、および、コア数依存性は少ない

   という結果が得られました。

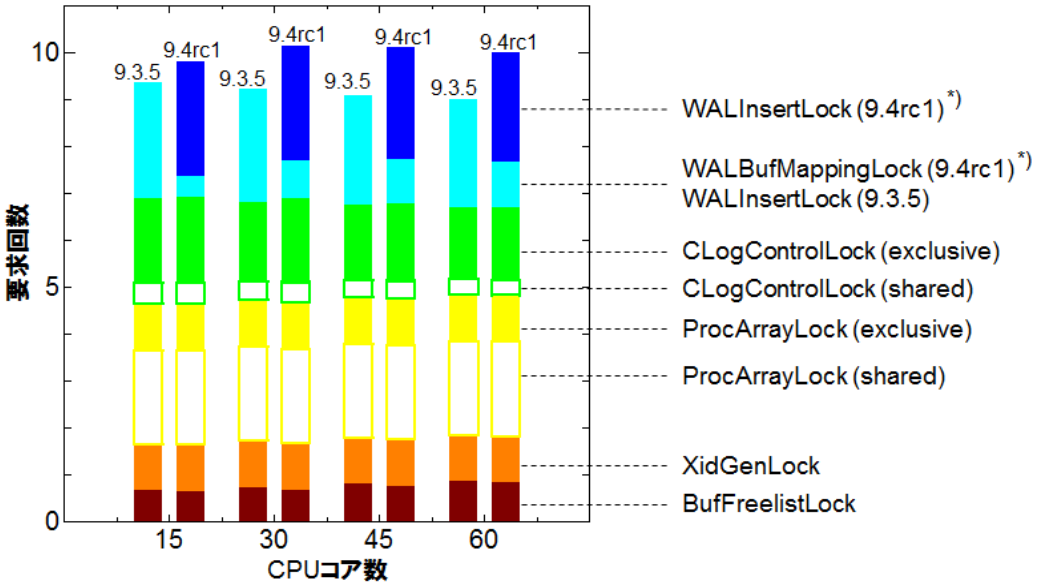

図 3.6.6:検証時の PostgreSQL9.3.5 と PostgreSQL9.4RC1 のロック要求回数

3. WAL 書き込み方法改良の効果

図 3.6.7 には、ロック競合回数の測定結果を示しています。WALInsertLock の競合回数については、9.3.5 と 9.4rc の測定結果を単純に比較すると 9.4RC1 の方が多くなっていることから、WALInsertLock の改良は逆効果になっているように思われます。しかしながら、この測定結果は競合が発生した回数を示しており、競合によって処理が滞った時間、すなわち、性能に与える影響の程度ではない点には注意する必要があります。ロックに関連する制御方法や処理の内容が同じとしますと，競合回数が大きいほど性能に与える悪影響は大きくなると考えてよいことになります。とはいえ、WALInsertLock に関する制御方法や処理の内容は、9.4RC1 で改良されていることから、競合回数の差異がそのまま性能に与える影響の差異となっているわけではありません。そこで、9.4RC1 で実施された改良の内容、すなわち、9.3.5 では WALInsertLock を獲得した 1 プロセスのみが WAL バッファへの書き込みを実行できたのに対し、9.4RC1 では、最大で 8 プロセスが WAL バッファへの書き込みを並行して実行できるようになったことを考慮し、9.4RC1 での WALInsertLock 競合の影響は 9.3.5 の 1/8 であると仮定して換算した結果を図 3.6.8 に示します。この仮定の下では、WALInsertLock が性能に与える影響は 9.3.5 よりも 9.4RC1 の方が小さくなっているという結果となりました。今回の検証では競合回数しか測定できませんでしたので競合のために処理が滞った時間については仮定をおいて影響を見積もりましたが，待機イベントを参照できる独自ツールを使って実施された別の検証でも『待機イベント「wal insert lock acquire」の発生は見られない』との結果が報告されています[3]ので，この仮定や今回の結果は妥当なものと考えております．このように，WALInsertLock の影響を正確に測定するためには、競合回数ではなく、競合のために処理が滞った時間を測定して比較する必要がある点を付記しておきます。

3 翔泳社 DBOnline 「システム用途別に見る Postgres Plus の新機能」執筆：株式会社アシスト
http://enterprisezine.jp/dbonline/detail/6639（ページ 3）

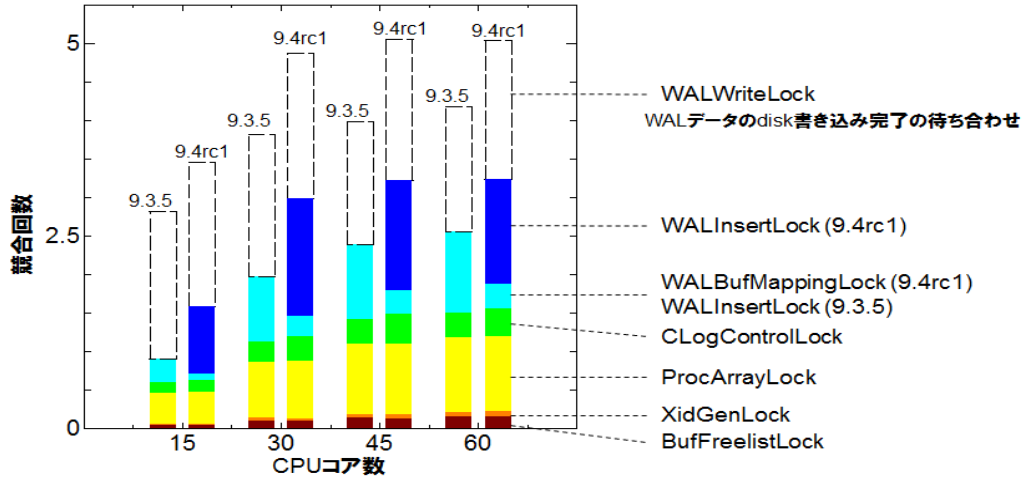

図 3.6.7:検証時の PostgreSQL9.3.5 と PostgreSQL9.4RC1 のロック競合回数

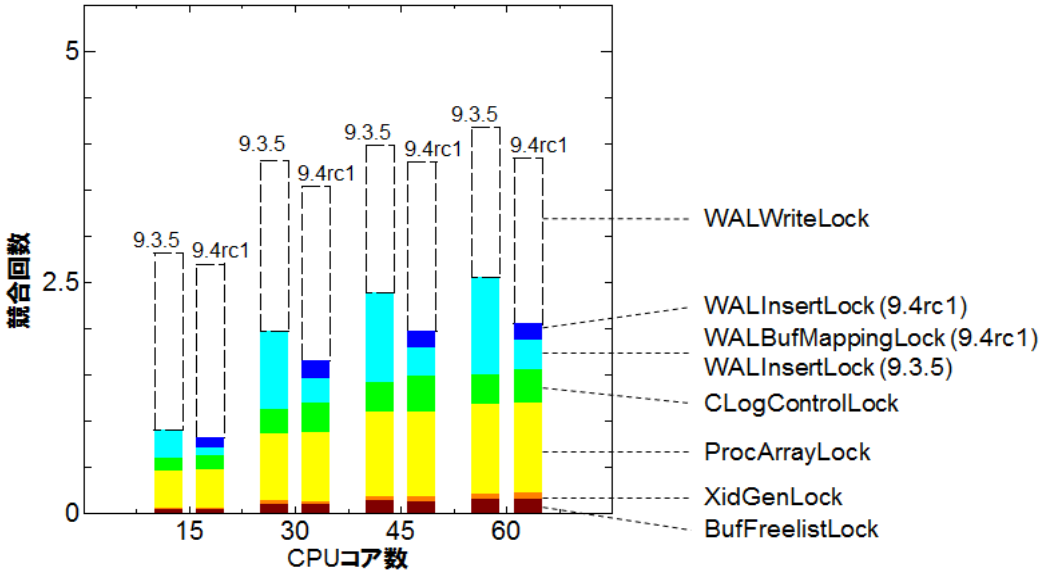

図 3.6.8:PostgreSQL 9.4RC1 の WALInsertLock を PostgreSQL9.3.5 の 1/8 と想定した場合のロック競合回数

また、ロック競合回数の測定結果（blk 欄の数字）には、WALWriteLock も大きな割合を占める要因としてリストアップされていたので図に含めましたが、この WALWriteLock は WAL を不揮発性の記録デバイス（disk 等）に書き込む際において、複数プロセスの WAL を1回のアクセスで書き込む操作を行った場合に、各プロセスに WAL 書き込み完了を通知するための手段として用いられています。すなわち、WALWriteLock の競合は、排他制御すべき資源で競合を起こしているわけではない点も付記しておきます。

4. CPU コア数増加時のボトルネック

図 3.6.8 より、CPU コア数増加による競合回数増加が目立つのは、ProcArrayLock と WALInsertLock、

（+WALBufMappingLock）の 2 種類であり、その他のロックについてもコア数の増加により競合も増加する傾向が見られました。前節で述べたとおり、この結果はロック競合回数を示しており、ロック競合によって処理が待たされる時間ではないため、各ロックについての競合回数を単純に比較することはできませんが、ボトルネックを推測する手掛かりにはなると考えられます。すなわち、9.4RC1 について図 3.6.8 を読み解くと、ロック競合の中で最も大きい割合を占めている ProcArrayLock が最大のボトルネックであり、次に同程度の割合を占めている WALInsertLock と CLogControlLock がそれに次ぐボトルネックとなっている可能性が高いと推測できます。

なお、ProcArrayLock は、MVCC においてタプルの可視性を判断する際に用いられるスナップ・ショット（どのトランザクションが処理中であるかを示す情報）の読み取りと更新を矛盾なく行うために用いられているロックであることから、このボトルネックを緩和するためには、スナップ・ショット管理方法の改良が必要であると考えられます。

3.6.3. まとめ

PostgreSQL9.4 の更新性能については、CPU コア数が一定の数以下では、特に更新量が多くなる高い負荷がかかったワークロードの場合に PostgreSQL9.3.5 と比較して性能が改善されることが確認できました。更新系の負荷が高いシステムについては PostgreSQL9.4 を採用することで効果が得られる可能性が高いと考えられます。

また、単純な更新のみを実施するワークロードでは PostgreSQL9.3.5、PostgreSQL9.4RC1 共に CPU スケーラビリティについて確認することができませんでした。
ただし、既に参照系の検証でも報告されている通り、参照系の場合の CPU スケーラビリティについては効果が高いことがわかっています。一般的には参照系のクエリの割合が高いケースのほうが多いことが想定されますので、一概に CPU を搭載することの意味がないということにはなりません。PostgreSQL 採用するシステムのワークロードを検討したうえで、PostgreSQL サーバを搭載する CPU の規模についての検討・検証を行うことが重要だと考えています。

なお、今回の測定では CPU スケーラビリティの傾向がまだつかめていなかったため、15core 単位での計測を行いました。しかし、ピーク値が 15core となってしまったことで何 core までスケールするのかもう少し粒度の小さい値を確認することができませんでした。次年度以降の活動で CPU のコア数の設定を 15core 前後の測定粒度を小さくし、どのあたりで本ワークロードのピーク値が確認できるかが確認されることを期待しています。

4. 仮想化(KVM)検証

4.1. 検証概要

近年、ハードウェアの進化による性能向上はめざましく、一つの CPU に搭載されるコアの数とメモリの容量は年々増大しています。

マルチコアについては、代表的な 2 ソケット対応の Xeon E5-2699 v3 プロセッサでは、1 プロセッサあたり 18 コア、2 プロセッサ構成では 36 コアとなります。4 ソケット対応のIvy Bridge マイクロアーキテクチャの Xeon E7-8800/4800 v2 プロセッサでは 1 プロセッサあたり最大 15 コア、4 プロセッサ構成では最大 60 コアにもなります。

またメモリの大容量化も進み TB 単位でのメモリ搭載も可能となってきています。そのためサーバを単一の用途で使うのではなく、進化した仮想化技術の活用により、アプリケーションサーバや開発環境などで仮想マシンを複数利用してサーバの集約化によるリソース利用の効率化が図られています。

特に I/O 性能が要求されるデータベース用途では、サーバを仮想化せずに物理サーバを占有して使う方法が一般的でしたが、マルチコア化とメモリの大容量化と高性能化により、クリティカルな性能が要求されないデータベース用途で仮想化が普及してきています。

今回、データベース用途に仮想化技術を適用した場合に仮想化によるオーバーヘッドでどの程度性能への影響が出るかを、仮想化技術として KVM を使用し、物理サーバと KVM 上で作成する仮想マシンでコア数とメモリ容量をあわせて比較検証しました。

4.2. 検証目的

仮想化技術を使用した場合、データベースへの性能の影響がどの程度あるか、またリソースの使われ具合がどの程度変化するかを、ハードウェア上で直接 PostgreSQL を動作させた場合と、仮想マシン上で PostgreSQL を動作させた場合の差異について比較し、検証します。

4.3. 検証構成

4.3.1. 検証構成概要

以下の図に仮想化検証での検証構成を示します。

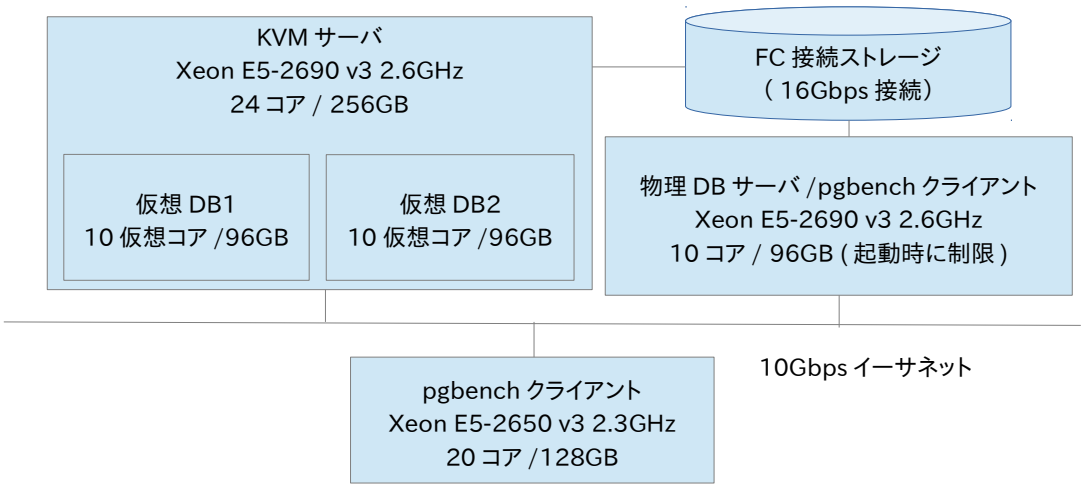

図 4.3.1:KVM 仮想化検証構成

仮想化検証用に使用したサーバは全部で 3 台です。

1)KVM サーバ:

KVM の仮想化環境を提供する KVM サーバの上に仮想 DB で使用する仮想マシンを二つ用意します。仮想マシンの構成は、仮想コア数が 10、割り当てるメモリは 96GB とし、それぞれの仮想マシン上で仮想 DB1 と仮想 DB2 を作成します。

2)物理 DB サーバ:

物理 DB サーバと仮想 DB 用仮想マシンのマシンスペック（コア数とメモリ容量）を同一にするために、物理 DB サーバの grub.conf のカーネル起動パラメータに maxcpus と mem を指定して起動することで、仮想マシンと同じ構成に制限しました。
仮想 DB 用仮想マシンを 2 台同時にベンチマークする際に、仮想 DB2 への pgbench クライアントとして使用します。
3）pgbench クライアント：
物理 DB サーバ、仮想 DB1 への pgbench によるベンチマークを実行するために使用します。

各サーバ間のネットワークは 10Gbps で接続しました。
KVM の設定は、FC 接続ストレージのボリュームにストレージプールを作り、ストレージプールから仮想 DB サーバの仮想マシンを作成します。仮想マシンには OS(RHEL6.5)と PostgreSQL9.4 と DB 本体を配置し、仮想マシンから別の FC 接続ストレージのボリュームを直接 mount して WAL 領域として使用しました。

4.3.2. ハードウェア構成

主なスペックと構成は以下のとおりです。

表 4.3.1: ハードウェア構成

機器	項目	仕様
KVM サーバ（仮想 DB 1、仮想 DB2） 物理 DB サーバ	CPU	インテル® Xeon® プロセッサ E5-2690v3 2.6GHz (12 コア) x 2　合計 24core
	搭載メモリ	256GB
	内蔵ストレージ	HDD: 600GB 12G SAS 15k x 4 RAID10
	外部ストレージ	FC 接続ストレージ上の 3TB（RAID10）領域 [KVM サーバ] 領域1)仮想マシン用 KVM ストレージプール(OS,DB 本体)/領域 2)仮想マシンに直接 mount する WAL 領域用 [物理 DB] 領域 1)DB 本体/領域 2) WAL 領域用
	OS	Red Hat Enterprise Linux 6.5 (for Intel 64)
	DBMS	PostgreSQL 9.4.0
	クライアント	PostgreSQL 標準 pgbench
pgbench クライアント	CPU	インテル® Xeon® プロセッサ E5-2650v3 (10 コア) x 2　合計 20core
	搭載メモリ	128GB
	内蔵ストレージ	1.2TB 6G SAS 10k x 10 RAID10
	OS	Red Hat Enterprise Linux 6.5 (for Intel 64)
	クライアント	PostgreSQL 標準 pgbench
FC 接続ストレージ	HDD	900GB 6G SAS 10K 2.5inch x 96 実効容量 3TB の領域を RAID10 で 8 個確保

詳細な設置状況については、2014 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。

4.3.3. 検証環境構築手順

PostgreSQL 9.4.0 はソースコードからコンパイルする方法で以下の手順でインストールしました。

```
# cd /usr/src
```

```
# wget https://ftp.postgresql.org/pub/source/v9.4.0/postgresql-9.4.0.tar.bz2
# wget https://ftp.postgresql.org/pub/source/v9.4.0/postgresql-9.4.0.tar.bz2.md5
# md5sum postgresql-9.4.0.tar.bz2
# cat postgresql-9.4.0.tar.bz2.md5          ←md5sum の結果とこの値が一致することを確認する。
# tar xjvf postgresql-9.4.0.tar.bz2
# cd postgresql-9.4.0

# yum groupinstall "Development Tools"
# yum install flex bison readline-devel zlib-devel

# ./configure -prefix=/usr/pgsql-9.4.0
# make world
# make install-world
```

KVM 上の二つの仮想マシンの CPU 割当は、物理 CPU のコアに CPU をまたがらないように固定で割り当てました。KVM サーバの物理 CPU のコア構成を以下の方法で調査しました。

```
# numactl --hardware
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 12 13 14 15 16 17
node 0 size: 130943 MB
node 0 free: 127402 MB
node 1 cpus: 6 7 8 9 10 11 18 19 20 21 22 23
node 1 size: 131072 MB
node 1 free: 128116 MB
node distances:
node   0   1
  0:  10  21
  1:  21  10
#
```

物理 CPU1(上記では node 0)はコア番号 0,1,2,3,4,5,12,13,14,15,16,17 なのでこれを仮想 DB1 に、物理 CPU2(上記では node 1)はコア番号 6,7,8,9,10,11,18,19,20,21,22,23 なのでこれを仮想 DB2 に割り当てます。

```
# virsh list --all
 Id    名前                           状態
----------------------------------------------------
 8     vmpsql01                       実行中
 9     vmpsql02                       実行中

# virsh shutdown vmpsql01
ドメイン vmpsql01 をシャットダウンしています

# virsh shutdown vmpsql02
ドメイン vmpsql02 をシャットダウンしています

# virsh list --all
 Id    名前                           状態
----------------------------------------------------
 -     vmpsql01                       シャットオフ
 -     vmpsql02                       シャットオフ

# virsh edit vmpsql01
******
```

```
【vcpu の行を下記のように変更。12 個のうち 10 個を仮想コアを割り当てて、2 個は OS 用に残しておく。】
元の状態　<vcpu placement='static'>10</vcpu>
⇒固定割り当て　<vcpu placement='static' cpuset='0-5,12-15'>10</vcpu>
*****
ドメイン vmpsql01 XML の設定は編集されました
# virsh start vmpsql01
ドメイン vmpsql01 が起動されました

# virsh edit vmpsql02
******
【vcpu の行を下記のように変更。12 個のうち 10 個を仮想コアを割り当てて、2 個は OS 用に残しておく。】
元の状態　<vcpu placement='static'>10</vcpu>
⇒固定割り当て　<vcpu placement='static' cpuset='6-11,18-21'>10</vcpu>
*****
ドメイン vmpsql02 XML の設定は編集されました
# virsh start vmpsql02
ドメイン vmpsql02 が起動されました
#
```

4.4. 検証方法

今回の検証では、参照系トランザクションは「定点観測（スケールアップ検証）」とメモリ割り当て以外は同一の検証条件で実施し、更新系トランザクションは「WAL 改善」と同一の検証条件で実施しました。

検証1. 参照系トランザクションでの仮想化による性能検証
物理サーバ（10 コア、96GB）と仮想 DB（10 仮想コア、96GB）とで参照系トランザクションのベンチマークを実施します。
検証 1-1. 物理 DB サーバ(10 コア, 96GB)への参照系ベンチマーク
検証 1-2. 仮想 DB1(10 仮想コア, 96GB)への参照系ベンチマーク

⇒検証 1-1.と検証 1-2.を比較することにより、参照系トランザクションでの仮想化によるオーバーヘッドでの性能低下の程度を知ることが出来ます。

検証2. 更新系トランザクションでの仮想化による性能検証
物理 DB サーバ（10 コア、96GB）と仮想 DB1（10 仮想コア、96GB）とで更新系トランザクションのベンチマークを実施します。
検証 2-1. 物理 DB サーバ(10 コア, 96GB)への更新系ベンチマーク
検証 2-2. 仮想 DB1(10 コア, 96GB)への更新系ベンチマーク

⇒検証 2-1.と検証 2-2.を比較することにより、更新系トランザクションでの仮想化によるオーバーヘッドでの性能低下の程度を知ることが出来ます。

検証3. 更新系トランザクションを仮想化環境で同時実行した場合の性能検証
仮想 DB サーバ1（10 仮想コア、96GB）と仮想 DB2（10 仮想コア、96GB）とで同時に更新系トランザクションを実行した場合に、仮想マシン同士が互いに与える影響を仮想マシン単体と比較調査します。

4.5. 検証結果

4.5.1. [検証 1] 参照系ベンチマークにおける仮想化による性能への影響

10 仮想コアと 96GB メモリを割り当てた仮想マシンと、10 コアと 96GB に制限した物理サーバとの間で、pgbench カスタムスクリプトを使った参照系ベンチマークをして比較しました。以下にグラフを示します。
グラフの横軸はクライアント数、縦軸は TPS です。

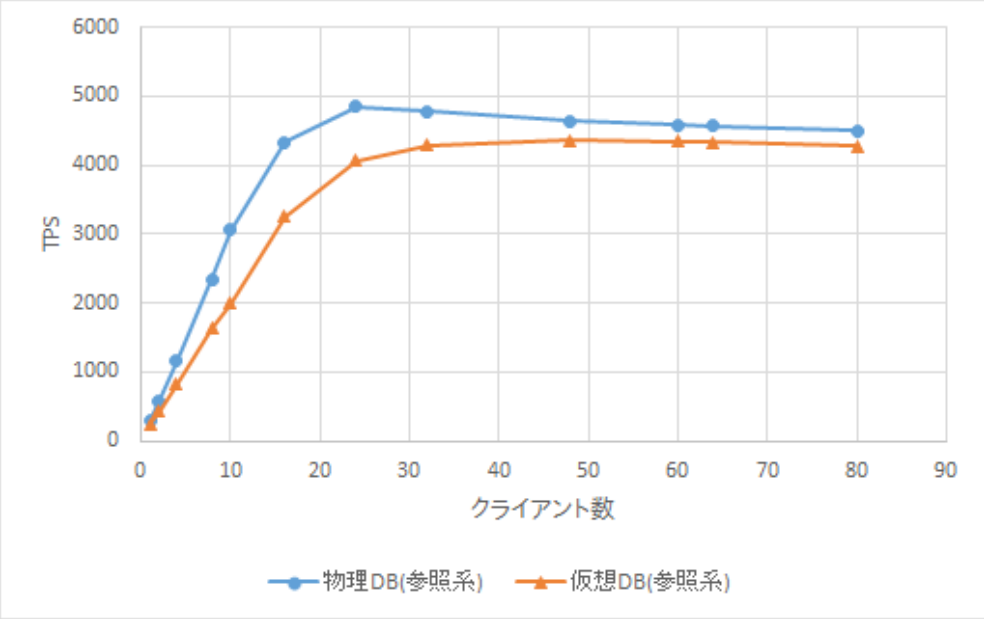

図 4.5.1:物理 DB と仮想 DB の TPS 比較（参照系）

物理 DB での性能ピークであるクライアント数 24 に至るまでの間、クライアント数の増加につれて徐々に性能差が広がっていき、クライアント数 10 と 16 では性能差は最大の 1000TPS 程度になります。物理 DB の性能ピークであるクライアント数 24 以降では性能差が縮小していきます。
性能のピークとなるクライアント数は物理 DB では 24、仮想 DB では 48 で、仮想 DB は物理 DB に比べて性能のピークとなるクライアント数は 2 倍となっています。

以下は、参照系の性能低下率です。グラフの横軸はクライアント数、縦軸は仮想化による性能低下率です。

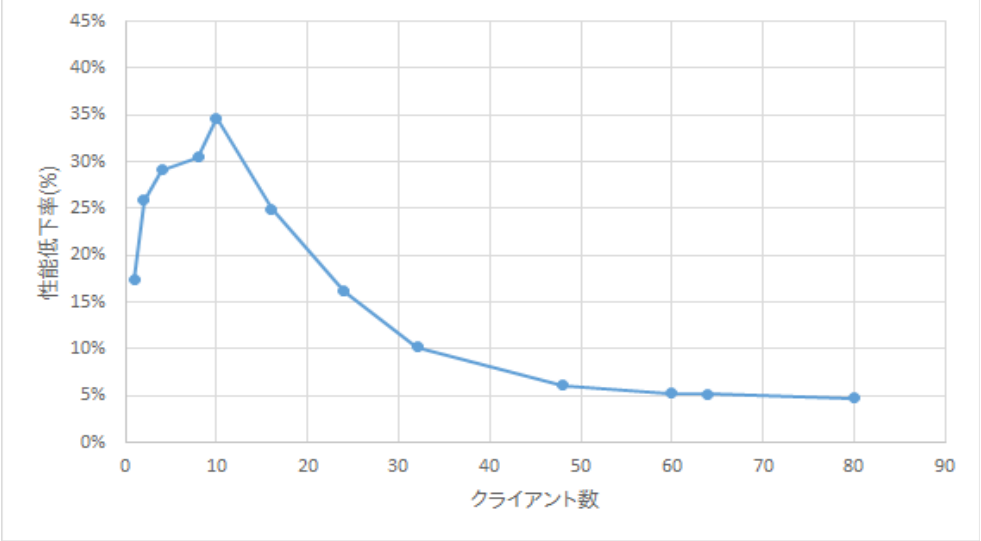

図 4.5.2:仮想化による性能低下率（参照系）

性能低下率は、クライアント数 10 で最大の 35%となり、その後は徐々に減っていきます。クライアント数 48 以降では性能差は 5%程度まで縮小します。

以下の表は、参照系トランザクションにおける物理 DB の TPS と仮想 DB の TPS の比較です。クライアント数の変化において、物理 DB と仮想 DB の TPS の変化と、性能低下率と物理と仮想の TPS の差分を示しています。

表 4.5.1: 参照系トランザクションにおける物理 DB と仮想 DB の TPS と性能低下率

クライアント数	TPS(物理)	TPS(VM)	性能低下率	TPS 差分
1	296.03	244.61	17.37%	51.42
2	584.19	433.19	25.85%	151.00
4	1153.47	817.63	29.12%	335.84
8	2343.89	1630.16	30.45%	713.73
10	3061.52	1999.02	34.70%	1062.50
16	4339.75	3256.15	24.97%	1083.60
24	4857.52	4070.93	16.19%	786.59
32	4783.32	4297.09	10.17%	486.23
48	4641.30	4358.31	6.10%	282.99
60	4582.06	4341.07	5.26%	240.99
64	4564.60	4330.17	5.14%	234.43
80	4497.80	4283.92	4.76%	213.88

物理環境での CPU 使用率の内訳を仮想環境での CPU 使用率の内訳を以下の図に示します。

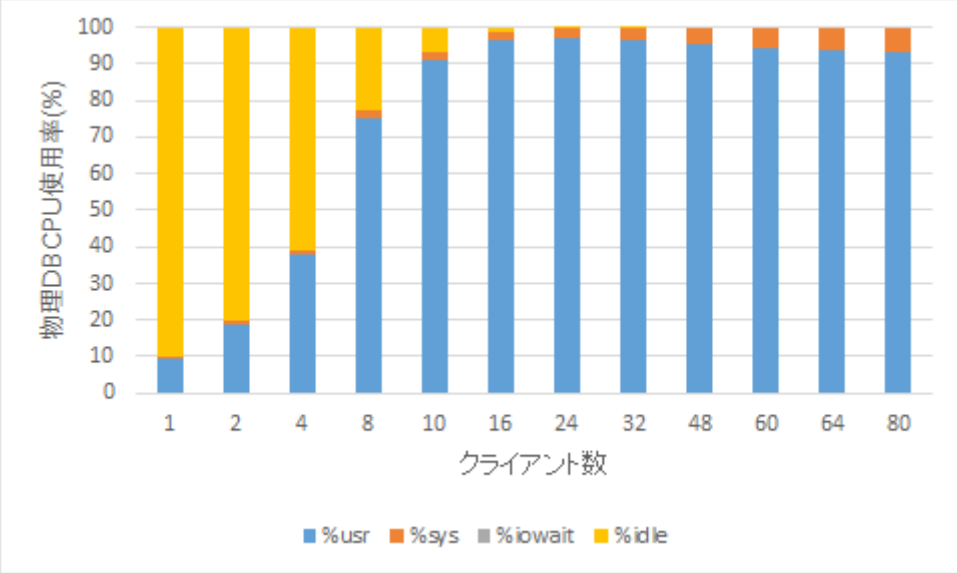

図 4.5.3:物理 DB の CPU 使用率内訳（参照系）

物理 DB の CPU 使用率は 16 クライアントで%sys と%usr で 99%、性能のピークである 24 クライアントでほぼ 100%と CPU リソースを使い切っています。24 クライアントよりクライアント数が多くなるとオーバーヘッドで TPS が低下しています。

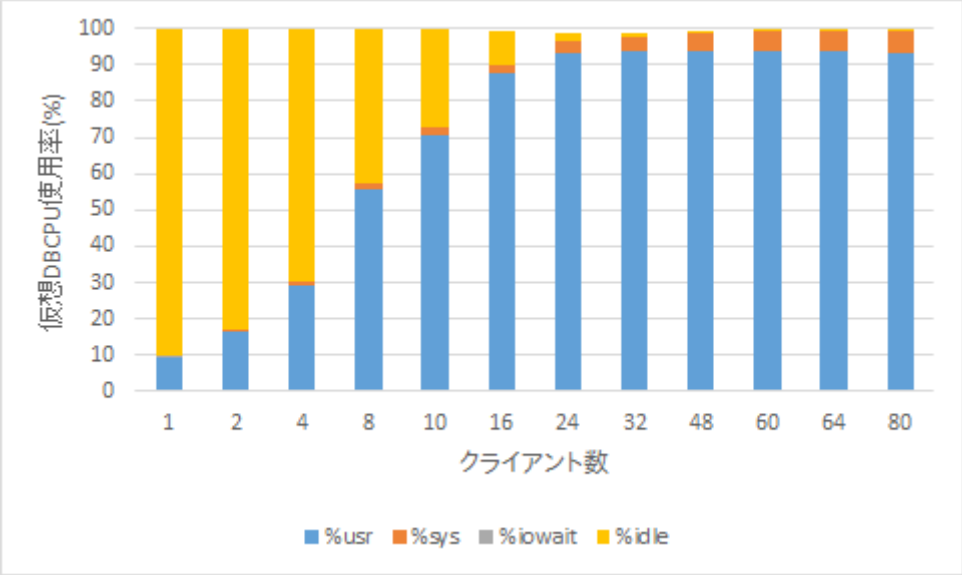

図 4.5.4:仮想 DB の CPU 使用率内訳（参照系）

仮想環境では、物理環境よりも CPU を使い切るのが遅くなっています。これは仮想化によるオーバーヘッドでクライアントからの要求が CPU に遅延して到着するためであると推定します。
また、%iowait は物理環境ではまったく発生していませんが、仮想環境では 0.1%未満とわずかながら発生しています。

4.5.2. [検証 2] 更新系ベンチマークにおける仮想化による性能への影響

10 仮想コアと 96GB メモリを割り当てた仮想マシンと、10 コアと 96GB に制限した物理サーバとの間で、pgbench カスタムスクリプトを使った更新系ベンチマークをして比較しました。以下にグラフを示します。
グラフの横軸はクライアント数、縦軸は TPS です。

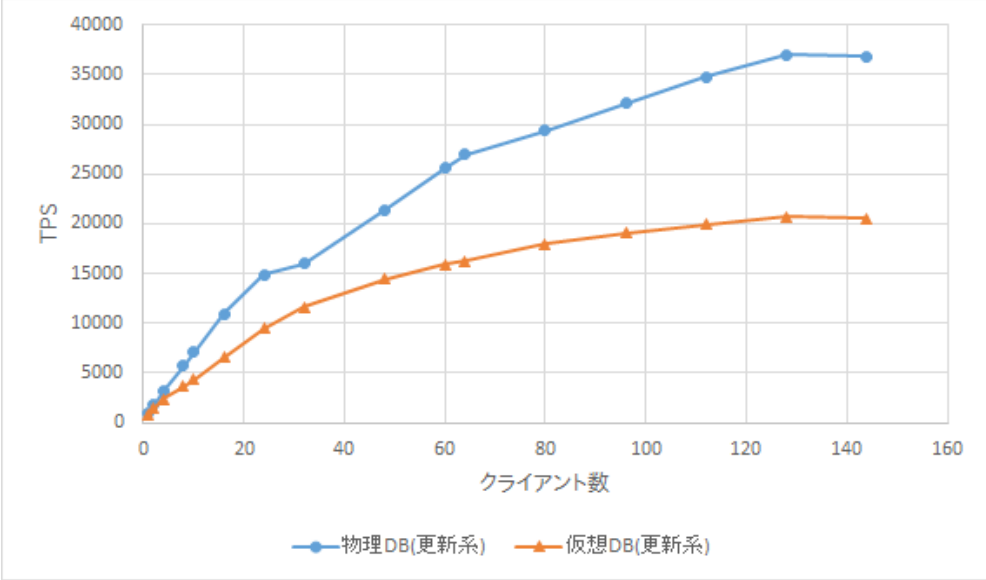

図 4.5.5:物理 DB と仮想 DB の TPS（更新系）

更新系における仮想化による性能低下はクライアント数が増大するにつれて徐々に大きくなっていきます。これは、クライアント数の増加に伴いストレージへの書込み I/O が増えて、リソース競合が増えて低下の度合いが増えていくためと考えられます。

参照系では性能のピーク後クライアント数の増大で徐々に性能差は小さくなりましたが、更新系ではクライアント数の増大により性能低下も増大していきます。

以下は、更新系の性能低下率です。グラフの横軸はクライアント数、縦軸は TPS です。

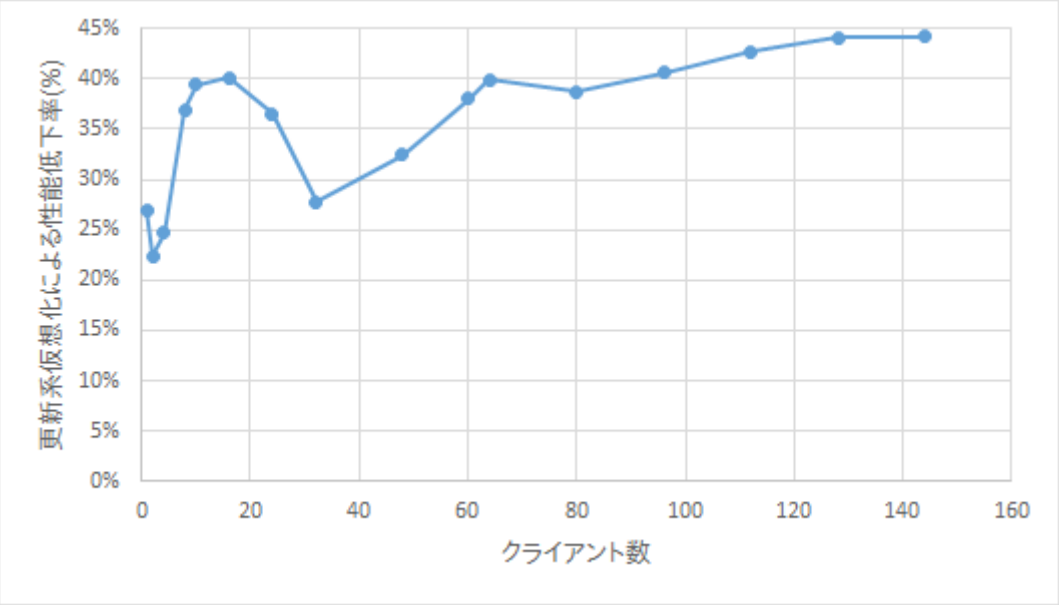

図 4.5.6:仮想化による性能低下率（更新系）

更新系における仮想化によるオーバーヘッドの性能低下率はクライアント数が 1 から 4 と 32 と 48 あたりで 25%程度でそれ以外では平均して 40%程度となっています。

以下の表は、更新系トランザクションにおける物理 DB の TPS と仮想 DB の TPS の比較です。クライアント数の変化において、物理 DB と仮想 DB の TPS がどのように変化するかと、性能低下率と物理と仮想の TPS の差分を示しています。

表 4.5.2: 更新系トランザクションにおける物理 DB と仮想 DB の TPS と性能低下率

クライアント数	TPS(物理)	TPS(VM)	性能低下率	TPS 差分
1	1015.50	741.57	26.97%	273.93
2	1788.46	1388.40	22.37%	400.06
4	3099.12	2334.86	24.66%	764.26
8	5729.70	3615.80	36.89%	2113.90
10	7110.38	4304.84	39.46%	2805.54
16	10961.07	6565.42	40.10%	4395.65
24	14904.27	9455.05	36.56%	5449.22
32	16032.00	11586.39	27.73%	4445.61
48	21389.26	14457.22	32.41%	6932.04
60	25623.00	15880.60	38.02%	9742.40
64	27001.33	16235.66	39.87%	10765.67
80	29350.88	17986.30	38.72%	11364.58
96	32070.79	19056.85	40.58%	13013.94
112	34774.43	19912.77	42.74%	14861.66

128	36996.33	20667.05	44.14%	16329.28
144	36785.53	20513.29	44.24%	16272.24

物理環境での CPU 使用率の内訳と仮想環境での CPU 使用率の内訳です。

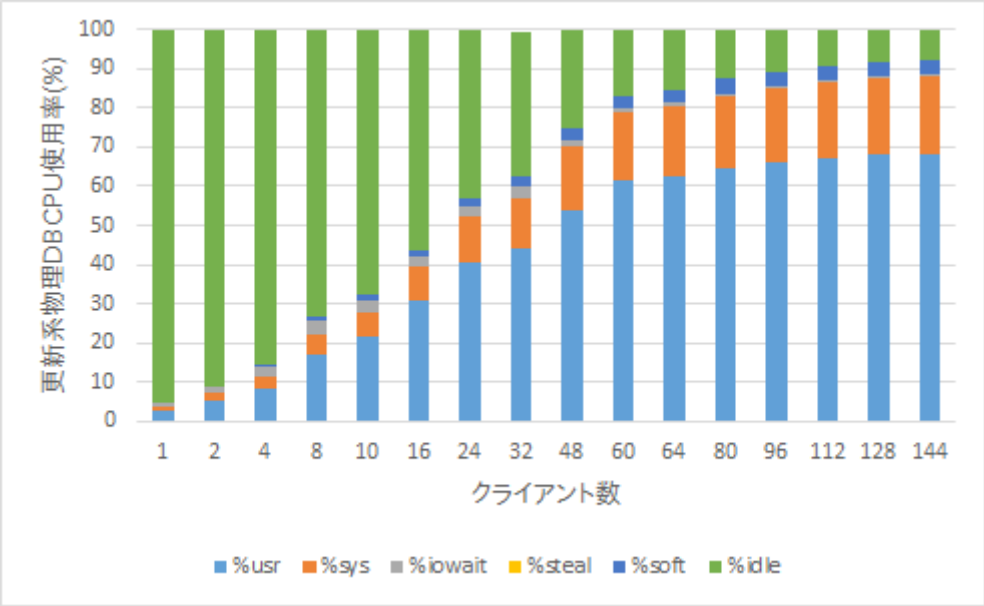

図 4.5.7:物理 DB の CPU 使用率内訳(更新系)

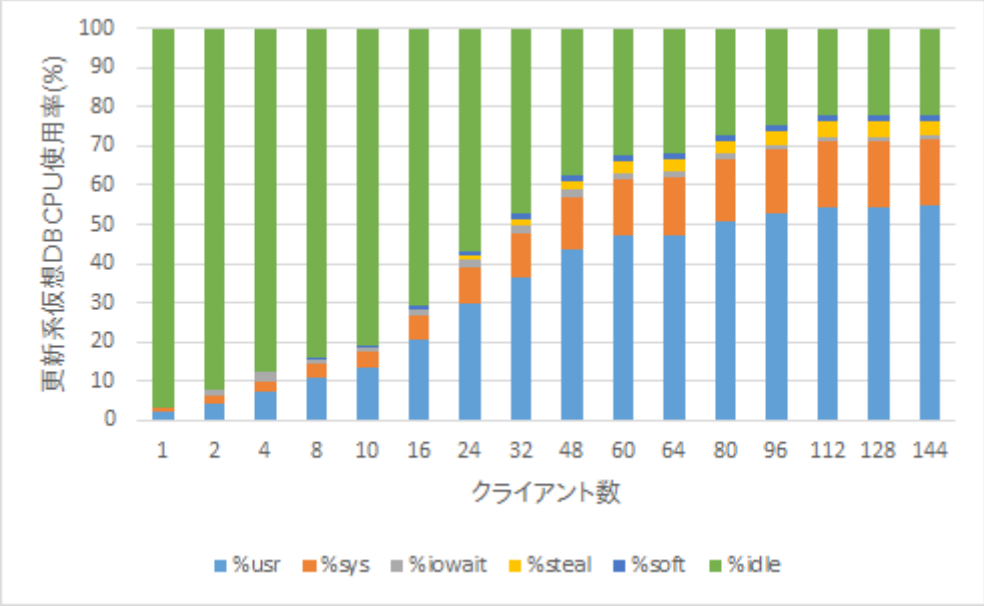

図 4.5.8:仮想 DB の CPU 使用率内訳(更新系)

%steal は物理環境ではまったく発生していませんが、仮想環境ではクライアント数の増加につれて 4%近くまで発生しています。

また%usr は物理環境では最大 70%程度ですが仮想環境では 55%程度まで、%sys は物理環境では最大 20%程度ですが仮想環境では 17%程度まで、%idle は物理環境では最小 7.9%ですが仮想環境では 22%となっており、仮想環境では物理環境に比べて CPU が有効に使いきれていないことがわかります。

4.5.3. [検証 3] 更新系ベンチマークの同時実行における仮想化による性能への影響

10 仮想コアと 96GB メモリを割り当てた仮想マシンを二つ用意して、二つの仮想 DB に対して同時に pgbench カスタムスクリプトを使った更新系ベンチマークを並列で実行して性能への影響を比較しました。以下にグラフを示します。
グラフの横軸はクライアント数、縦軸は TPS です。

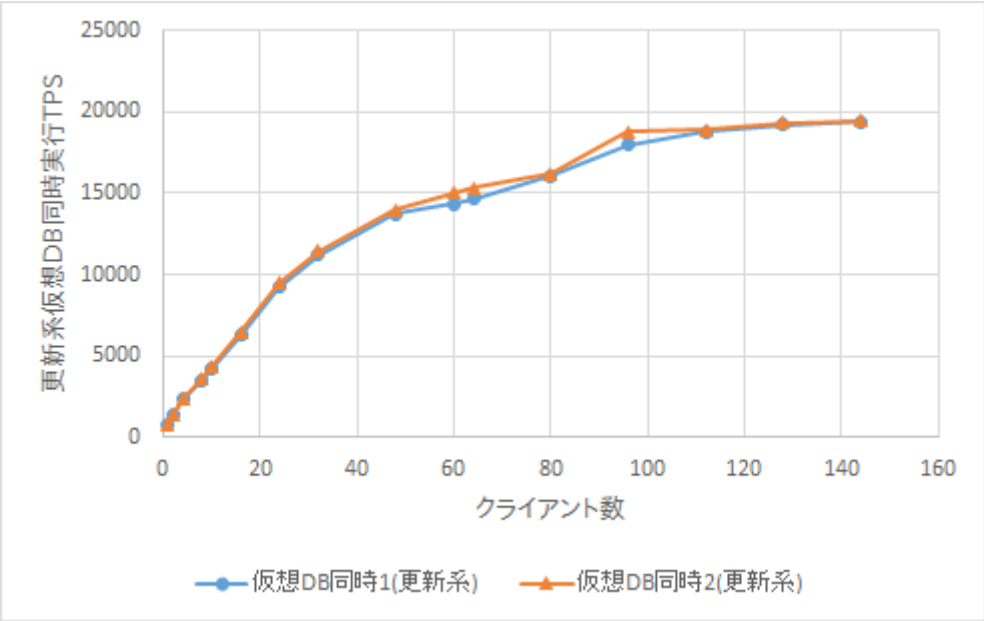

図 4.5.9:仮想化同時実行した両方の TPS（更新系）

仮想化環境での更新系トランザクション同時実行では、二つとも同様な性能特性で差はありません。

次に、更新系の仮想化単体と、更新系の二つの仮想化同時実行との比較のグラフを示します。

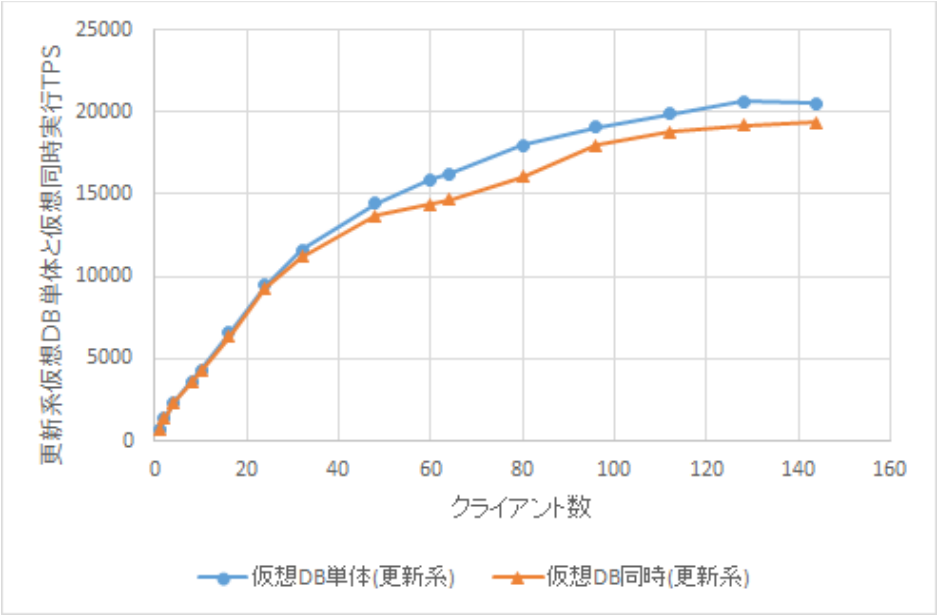

図 4.5.10:仮想化単体と複数同時実行による TPS（更新系）

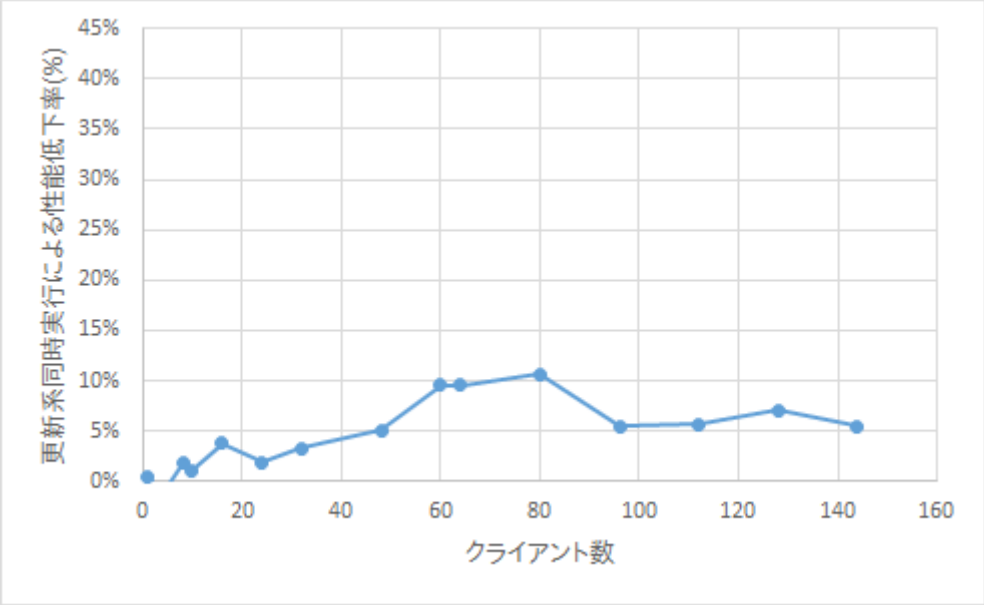

図 4.5.11:仮想化単体と複数同時実行による性能低下率（更新系）

仮想化単体での更新性能と、二つの仮想化での更新系同時実行性能との差は 36 クライアントぐらいまでほぼ差は無く、48 クライアントより大きくなっても最大 10%程度までの性能低下となっています。

以下の表は、更新系トランザクションにおける仮想環境単体の TPS と二つの仮想環境同時実行の TPS の比較です。

表 4.5.3: 更新系トランザクションにおける仮想 DB 単体と仮想 DB 混在の TPS と性能低下率

クライアント数	TPS(単体)	TPS(同時)	性能低下率	TPS 差分
1	741.57	737.54	0.54%	4.03
2	1388.40	1388.63	-0.02%	-0.23
4	2334.86	2363.03	-1.21%	-28.17
8	3615.80	3551.66	1.77%	64.14
10	4304.84	4256.87	1.11%	47.97
16	6565.42	6317.88	3.77%	247.54
24	9455.05	9275.48	1.90%	179.57
32	11586.39	11202.08	3.32%	384.31
48	14457.22	13720.48	5.10%	736.74
60	15880.60	14367.63	9.53%	1512.97
64	16235.66	14679.29	9.59%	1556.37
80	17986.30	16065.71	10.68%	1920.59
96	19056.85	18006.84	5.51%	1050.01
112	19912.77	18777.26	5.70%	1135.51
128	20667.05	19206.45	7.07%	1460.60
144	20513.29	19382.65	5.51%	1130.64

4.6. 考察

参照系トランザクションでは、物理 DB での性能ピークに至るまでの間で仮想 DB での仮想化による性能低下の度合いが大きくなっていますが、性能ピーク以降は物理 DB と仮想 DB との間の性能差異は段々減少して、クライアント数がある一定以上になると、物理 DB も仮想 DB も CPU バウンダリとなり仮想化による差異はめだたなくなります。参照系トランザクションではデータはメモリ上にキャッシュされるため I/O のオーバーヘッドではなく、仮想化によるネットワークオーバーヘッドのレスポンス遅延により性能ピークまでの間での差異が生じるものと推定します。

更新系トランザクションでは、クライアント数増加に伴うトランザクションの増大に応じて、物理 DB と仮想 DB との間の性能低下の度合いが段々増加していきます。CPU 使用率も%usr、%sys の両者とも仮想 DB での使用率は物理 DB に比較して低く、仮想 DB での%idle は 10%以上の差が生じています。これは、更新系トランザクションではストレージへの書き込みがトランザクションの増加に応じて増えていくため、仮想化によるオーバヘッドの比率が段々大きくなるためと考えられます。

現時点では CPU 使用率のみの性能分析でとどまっていますが、ストレージとネットワークの I/O の性能分析についても追加でおこなっていきます。

5. Docker 検証

5.1. 検証概要

近年 Docker が注目されています。Docker を利用することにより、アプリケーションの可搬性を高め、運用コストの削減、またデプロイの効率化が達成できると期待されています。本検証では、Docker 上で PostgreSQL を動作させた時の性能を中心に検証を行います。

5.2. 検証目的

Docker で実行単位となる「コンテナ」の環境下でアプリケーションを動かした時の性能面の影響についてはまだ未知数のところが多いのが現状です。そこで PGECons では Docker における性能検証を行うことになりました。

検証した内容は次の通りです。

表 5.2.1: Docker 検証 の検証内容

	概要	測定方法
1	非コンテナ環境での参照性能	非コンテナ環境でクライアント数を 1 から 128 まで変動させて参照性能を測定する。これが基準値となる
2	コンテナ環境で、仮想ネットワークの有無による参照性能の変化	コンテナ環境でクライアント数を 1 から 128 まで変動させて参照性能を測定する。そのとき、仮想ネットワークを利用するケースとしないケースを測定する
3	コンテナ内のファイルシステムの利用の有無による参照性能の変化	コンテナ環境でクライアント数を 1 から 128 まで変動させて参照性能を測定する。そのとき、コンテナ内のファイルシステムを利用するケースとしないケースを測定する
4	非コンテナ環境での更新性能	非コンテナ環境でクライアント数 32 で更新性能を測定する。これが基準値となる。
5	コンテナ環境で、仮想ネットワークの有無による更新性能の変化	コンテナ環境でクライアント数 32 で更新性能を測定する。そのとき、仮想ネットワークを利用するケースとしないケースを測定する
6	コンテナ内のファイルシステムの利用の有無による更新性能の変化	コンテナ環境でクライアント数 32 で更新性能を測定する。そのとき、コンテナ内のファイルシステムを利用するケースとしないケースを測定する

5.3. Docker とは

Docker は Docker 社(http://www.docker.com/)が開発するコンテナ管理ソフトです。オープンソースで公開されており、各種 Linux ディストリビューション上で利用できます。

Docker は仮想環境を提供するソフトではありませんが、ホスト OS から独立したように見えるアプリケーションの実行環境を提供する点でハイパーバイザなど、仮想化ソフトとユースケースが似ており、「コンテナ型の仮想化」と呼ばれることもあります。Docker を使うと、たとえば Red Hat Enterprise Linux 上で走るコンテナの中で、Ubuntu 用のアプリケーションをそのまま動作させることもできます。Ubuntu アプリケーションから見ると、実行環境は Ubuntu に見えるため、Ubuntu 用のパッケージを変更することなくインストールしてコンテナ内で実行できるわけです。

また、仮想化ソフトに比べると消費リソースが少なく、オーバヘッドも小さいと言われています(その点を本検証では確認します)。その理由は、仮想化ソフトのように、ハードウェアのエミュレーションを行わないからです。Docker では、Linux が持つリソースの隔離機能を使い、プロセス空間、ネットワーク、ファイルシステム、ユーザ ID などのリソースを分離させることによって仮想化ソフトのような機能を実現しています。分離の単位を「コンテナ」と呼びます。

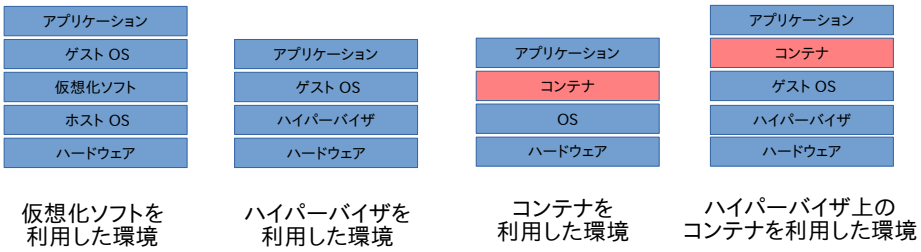

図 5.3.1: コンテナの概念

このような仕組みで Docker は動作するため、Windows 上で直接 Docker コンテナを実行することはできませんし、Linux 上で走る Docker コンテナ内で Windows アプリケーションを動かすこともできません(もちろん、Windows 上の仮想化ソフト上で Linux を動かし、その上でコンテナを実行することは可能です)。

また、Linux の「init プロセス」に相当するものがコンテナ上にはないため、プログラムの起動には工夫が必要な場合があります。本検証では、その点を考慮しながら PostgreSQL をコンテナ内で動かす方法を具体的に示します。

5.3.1. Docker コンテナのライフサイクル

コンテナを作るためには、まず Docker イメージを作成します。Docker イメージはコンテナの「ひな形」で、通常 Dockerfile という定義ファイルを記述しておいた上で、「docker build」というコマンドで作成します。作成した Docker イメージをコンテナとして実行するには、「docker run」コマンドを使います。コンテナにはコンテナ ID というユニークな ID が振られ、コンテナ ID を指定して再実行をしたり、「docker info」コマンドで詳細な情報を取得したりできます。
コンテナから Docker イメージを作成することもできます。

5.4. 検証構成

5.4.1. 検証ハードウェア構成

検証環境のハードウェア構成の概略を示します。

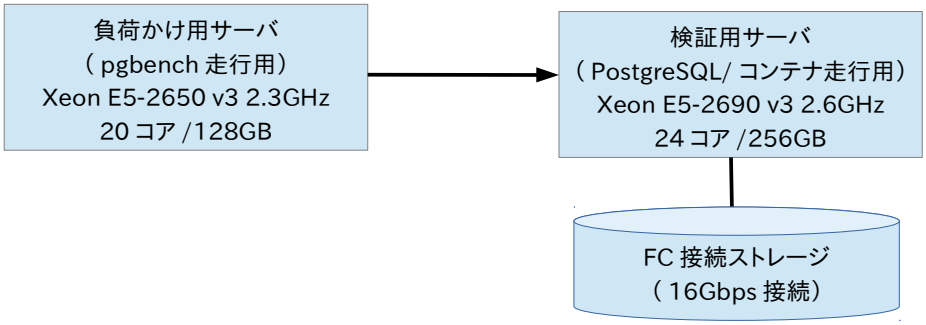

図 5.4.1: 検証用ハードウェア構成

5.4.2. 検証ソフトウェア構成

検証環境のソフトウェア構成を示します。

表 5.4.1: 検証用サーバ

OS	Red Hat Enterprise Linux 6.6
Docker	1.3.1
PostgreSQL	9.4RC1

表 5.4.2: 負荷かけ用サーバ

OS	Red Hat Enterprise Linux 6.5
PostgreSQL	9.4RC1

検証用サーバの Red Hat Enterprise Linux のバージョンは当初 6.5 にする予定でしたが、Docker がインストール出来ないため、6.6 にしました。

5.4.3. Docker のインストールと設定

Docker は以下の手順でインストールしました。

```
# rpm -ivh epel-release-6-8.noarch.rpm
# yum install docker-io
# docker info
Containers: 1
Images: 3
Storage Driver: devicemapper
 Pool Name: docker-253:0-2360778-pool
 Pool Blocksize: 65.54 kB
 Data file: /var/lib/docker/devicemapper/devicemapper/data
 Metadata file: /var/lib/docker/devicemapper/devicemapper/metadata
 Data Space Used: 602.1 MB
 Data Space Total: 107.4 GB
 Metadata Space Used: 1.004 MB
 Metadata Space Total: 2.147 GB
 Library Version: 1.02.89-RHEL6 (2014-09-01)
Execution Driver: native-0.2
Kernel Version: 2.6.32-504.1.3.el6.x86_64
Operating System: <unknown>
```

このままではコンテナ内でディスクスペースが 10GB 程度しか使えず、データベース領域が不足するため、/etc/docker.io/に以下を追加して使用可能なディスクスペースを 40GB まで増やします。

```
--storage-opt dm.basesize=50G
```

以下のように、50GB までディスクスペースがコンテナ内で使えるようになります。

```
# df -h
Filesystem                                 Size  Used Avail Use% Mounted on
/dev/mapper/vg_dl360gen919-lv_root          50G  7.1G   40G  16% /
tmpfs                                       64G  224K   64G   1% /dev/shm
/dev/sda2                                  485M   41M  420M   9% /boot
/dev/sda1                                  200M  264K  200M   1% /boot/efi
/dev/mapper/vg_dl360gen919-lv_home         497G  1.1G  471G   1% /home
```

5.4.4. Docker コンテナの作成

コンテナ用のOSとしては、Red Hat 社の Red Hat Enterprise Linux 6.5 用の公式イメージを使っています。これを“rhel6”という名前でイメージにしておき、以下の Dockerfile でコンテナを作ります。

```
FROM rhel6

MAINTAINER Tatsuo Ishii <ishii@sraoss.co.jp>

ENV PGHOME /home/postgres
ENV PG /usr/local/pgsql

RUN adduser postgres
RUN echo "postgres:postgres" | chpasswd
RUN mkdir -p $PG
RUN chown postgres.postgres $PG
RUN echo "PATH=$PG/bin:$PATH" >> $PGHOME/.bash_profile

COPY pgsql.tar.gz /tmp/
ADD postgresql.conf /tmp/
ADD pg_hba.conf /tmp/
ADD build.sh /tmp/
ADD start.sh /tmp/
CMD /tmp/start.sh
```

ここでは、PostgreSQL のバイナリはあらかじめ pgsql.tar.gz としてコンパイル済みのものを使用しています。このバイナリは、負荷かけ用サーバでソースからコンパイルしたものです。

postgresql.conf は、以下の部分だけをカスタマイズしました。

```
listen_addresses = '*'
max_connections = 128
shared_buffers = 32GB
work_mem = 256MB
checkpoint_segments = 16
checkpoint_timeout = 30min
logging_collector = on
log_line_prefix = '%t [%p-%l]'
```

pg_hbal.conf は、外部からの接続を受け付けるようにカスタマイズしました。

```
"local" is for Unix domain socket connections only
local   all             all                                     trust
# IPv4 local connections:
host    all             all             127.0.0.1/32            trust
# IPv6 local connections:
host    all             all             ::1/128                 trust
host    all             all             0.0.0.0/0               trust
```

start.sh は、yum update の実行、ssh のインストールと設定、PostgreSQL のバイナリを展開、インストールしてから postgres ユーザ権限で build.sh を実行します。

```
#! /bin/bash
cat > /etc/yum.repos.d/rhel65-local.repo <<EOF
[rhel65-local]
```

```
name=Red Hat Enterprise Linux 6.5
baseurl=file:///vol/RHEL65/
enabled=1
gpgcheck=0
EOF
yum -y update
yum -y install openssh-server openssh-client tar
yum -y install zlib-devel

# Start servers

# Allow to ssh login
sed -ri 's/UsePAM yes/#UsePAM yes/g' /etc/ssh/sshd_config
sed -ri 's/#UsePAM no/UsePAM no/g' /etc/ssh/sshd_config

service sshd start

cd /usr/local
tar xfz /tmp/pgsql.tar.gz
chown -R postgres.postgres pgsql
su -c 'sh /tmp/build.sh' postgres
host    all             all             0.0.0.0/0               trust
```

build.sh は、initdb を実行してデータベースを初期化し、PostgreSQL サーバを起動します。

```
#! /bin/sh
cd $PGHOME
source ./.bash_profile
cd /usr/local/pgsql
initdb -E UTF8 --no-locale -D data
cp /tmp/postgresql.conf data
cp /tmp/pg_hba.conf data
pg_ctl -D data start
while true
do
 sleep 60
done
```

なお、コンテナ内のファイルシステムを使用せずに、ホストのディレクトリを使用する場合には、build.sh は、以下のようになります(/vol/pgsql にホストのディレクトリがマウントされている想定です)。

```
#! /bin/sh
cd $PGHOME
source ./.bash_profile
cd /vol/pgsql
initdb -E UTF8 --no-locale -D data
cp /tmp/postgresql.conf data
cp /tmp/pg_hba.conf data
pg_ctl -D data start
while true
do
 sleep 60
done
```

5.5. 検証方法

5.5.1. 参照性能の検証

5.5.1.1. 検証目的

非コンテナ環境の PostgreSQL の参照性能を基準に、コンテナ環境での PostgreSQL の参照性能を比較します。コンテナ環境で PostgreSQL を走らせる際には、仮想ネットワークの有無、コンテナ内のファイルシステムの利用の有無の、計 4 パターンを測定し、比較します。

5.5.1.2. 検証方法

負荷かけサーバから pgbench で参照系トランザクションを 3 回実行し、その中央値を結果とします。同時接続数は 1 から 128 まで変化させます。測定トランザクションや測定方法は定点観測とまったく同じなので省略します。

Docker コンテナの起動コマンドは各ケースで以下のようになります。

表 5.5.1: Docker コンテナの起動コマンド

volume なし、仮想ネットワークなし	`docker run -i --name="pgsql94" --net=host --rm -v /RHEL65:/vol/RHEL65 -t postgresql_9_4_rc1`
volume なし、仮想ネットワークあり	`docker run -i --name="pgsql94" --rm -p 5432:5432 -v /RHEL65:/vol/RHEL65 -t postgresql_9_4_rc1`
volume あり、仮想ネットワークなし	`docker run -i --name="pgsql94" --net=host --rm -v /RHEL65:/vol/RHEL65 -v /vv31/vol:/vol -t postgresql_9_4_rc1`
volume あり、仮想ネットワークあり	`docker run -i --name="pgsql94" --rm -p 5432:5432 -v /RHEL65:/vol/RHEL65 -v /vv31/vol:/vol -t postgresql_9_4_rc1`

5.5.1.3. 検証結果

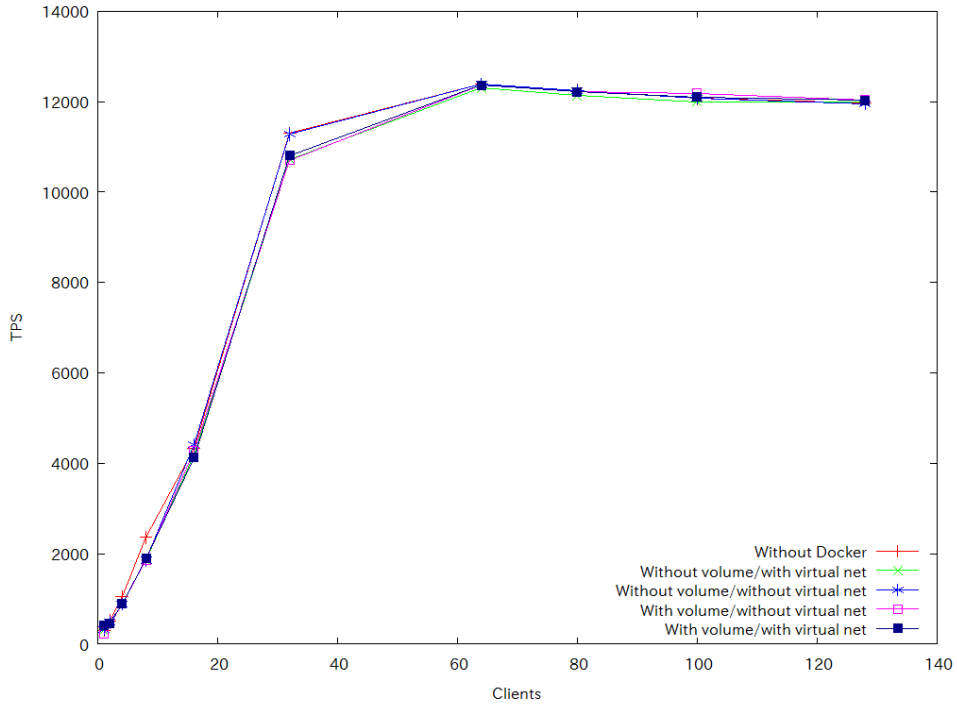

図 5.5.1: 測定結果のグラフ

表 5.5.2: グラフの凡例

グラフタイトル	コンテナ内のファイルシステムを使用	仮想ネットワークを使用
Without Docker	-	-
Without volume/with virtual net	X	○
Without volume/without virtual net	X	X
With volume/without virtual net	○	X
With volume/with virtual net	○	○

ご覧のように、すべてのケースのグラフがかなり重なっており、差異は測定誤差の範囲内と言えます。データベース全体がメモリに乗ってしまっていることもあり、検索性能に関しては Docker の影響は最小限だと言えます。

以下、詳細な TPS 値を示します。

表 5.5.3: TPS 値の詳細

同時接続数	非Docker	volumeなし、仮想ネットワークなし	volumeなし、仮想ネットワークあり	volumeあり、仮想ネットワークなし	volumeあり、仮想ネットワークあり
1	301.33	320.87	313.89	234.98	422.09
2	517.53	479.39	459.42	453.48	462.68
4	1051.00	888.33	882.45	890.99	887.31
8	2359.05	1857.23	1858.88	1857.34	1889.19
16	4339.10	4412.78	4187.84	4269.71	4125.36
32	11304.45	11277.95	10727.71	10702.16	10812.82
64	12392.89	12391.01	12304.14	12361.52	12361.25
80	12252.25	12240.29	12135.47	12222.95	12218.12
100	12073.30	12077.24	12002.27	12190.60	12104.44
128	11970.10	11959.50	11995.61	12046.36	12030.45

性能差が少ないということは、サーバリソースの利用状況に関しても同じような傾向となるはずです。sar の結果を示します。ご覧のように、殆ど差がありませんでした。

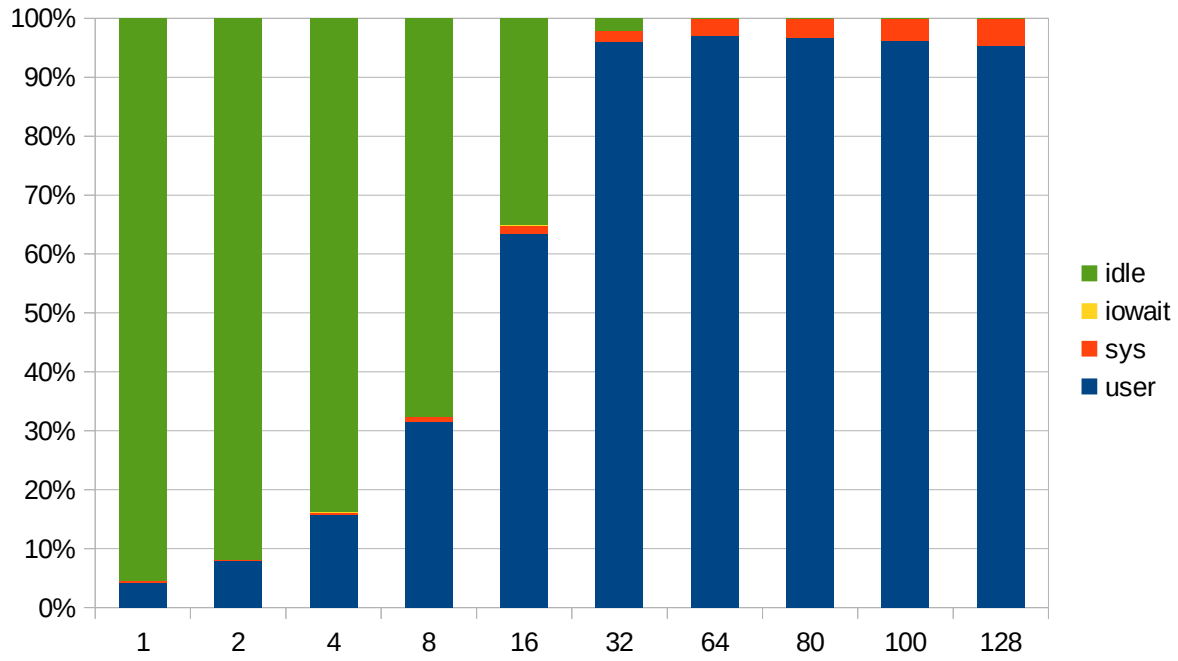

図 5.5.2: 非 Docker の場合のリソース利用状況

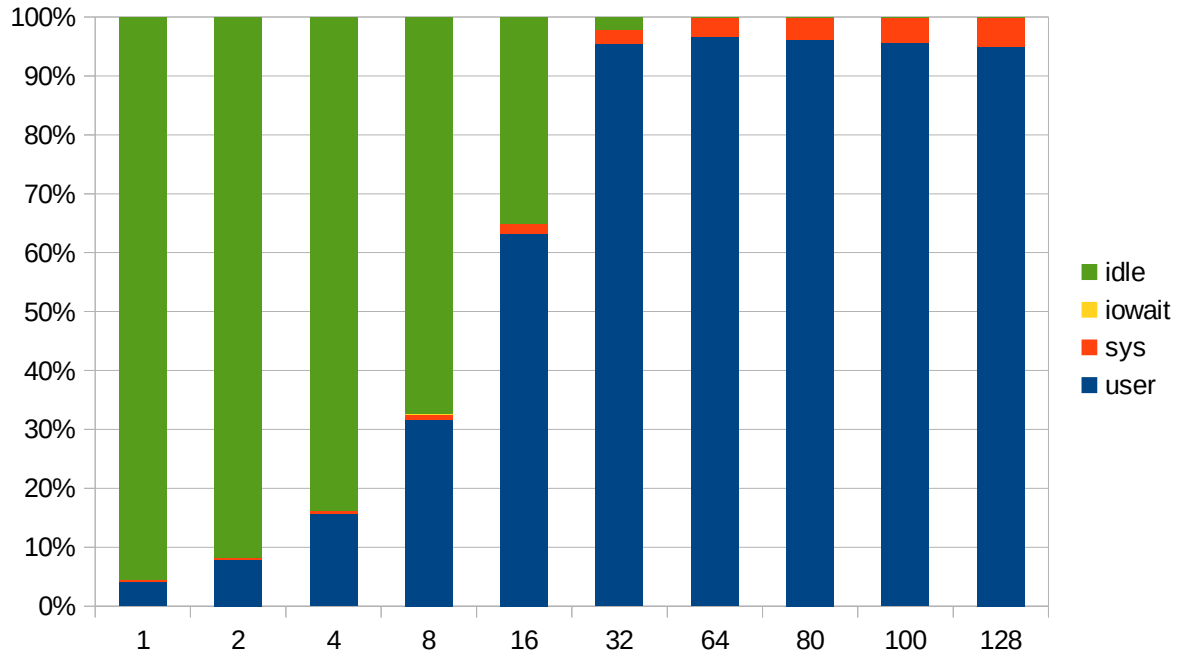

図 5.5.3: volume なし、仮想ネットワークなしのリソース利用状況

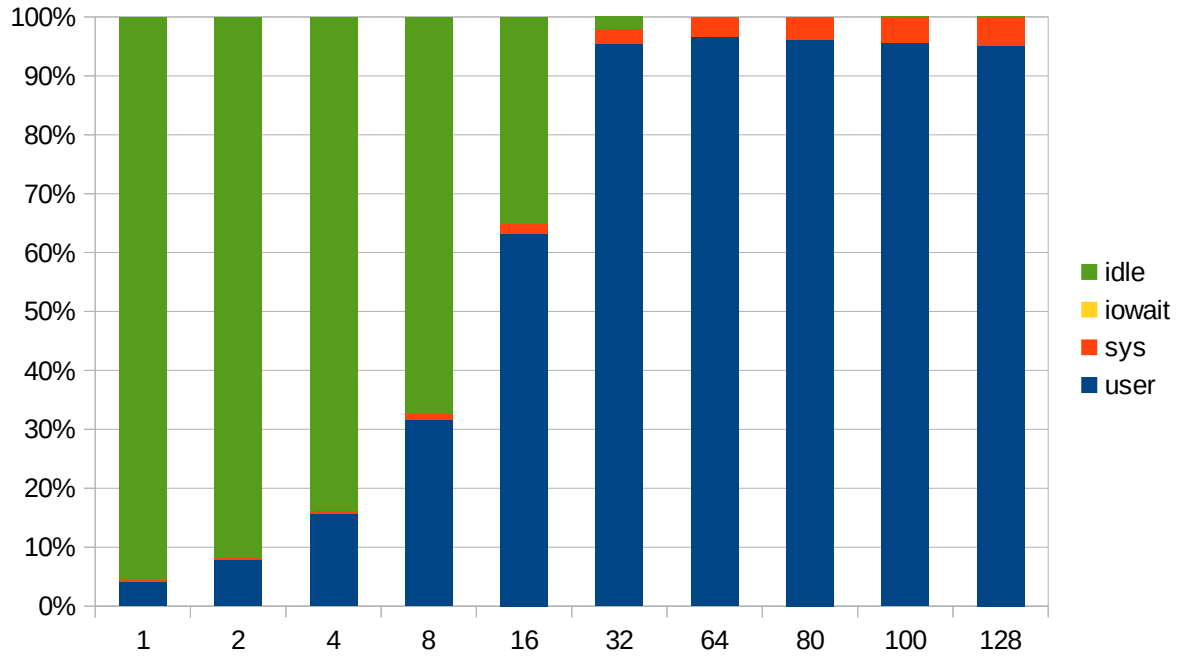

図 5.5.4: volume なし、仮想ネットワークありのリソース利用状況

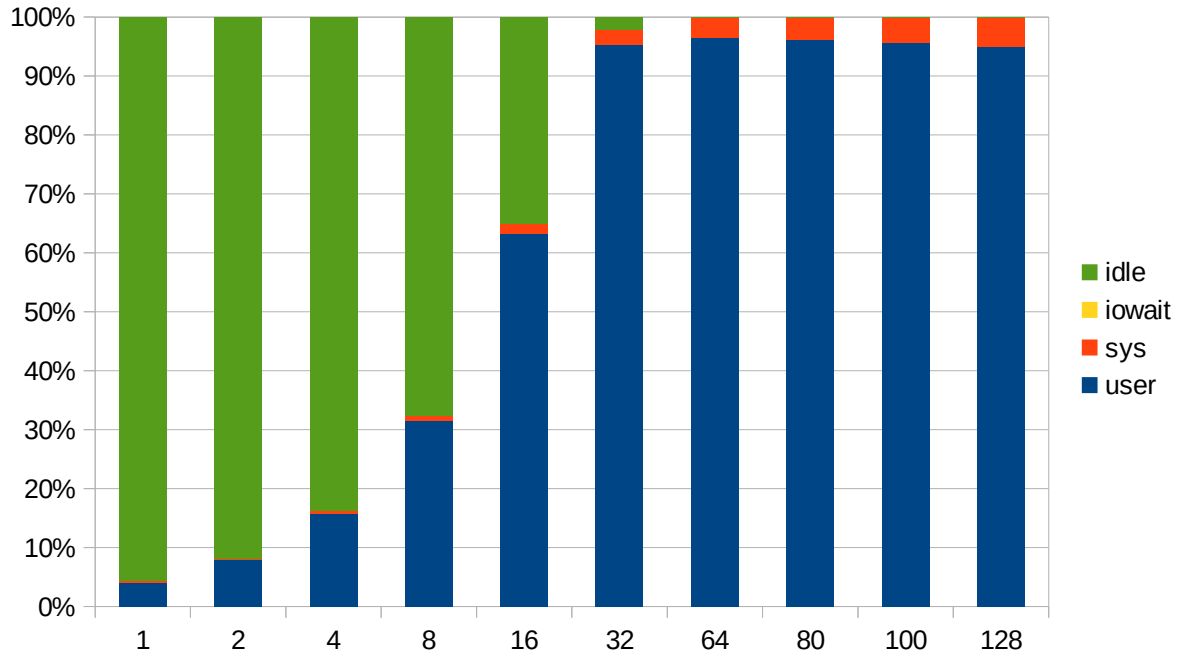

図 5.5.5: volume あり、仮想ネットワークありのリソース利用状況

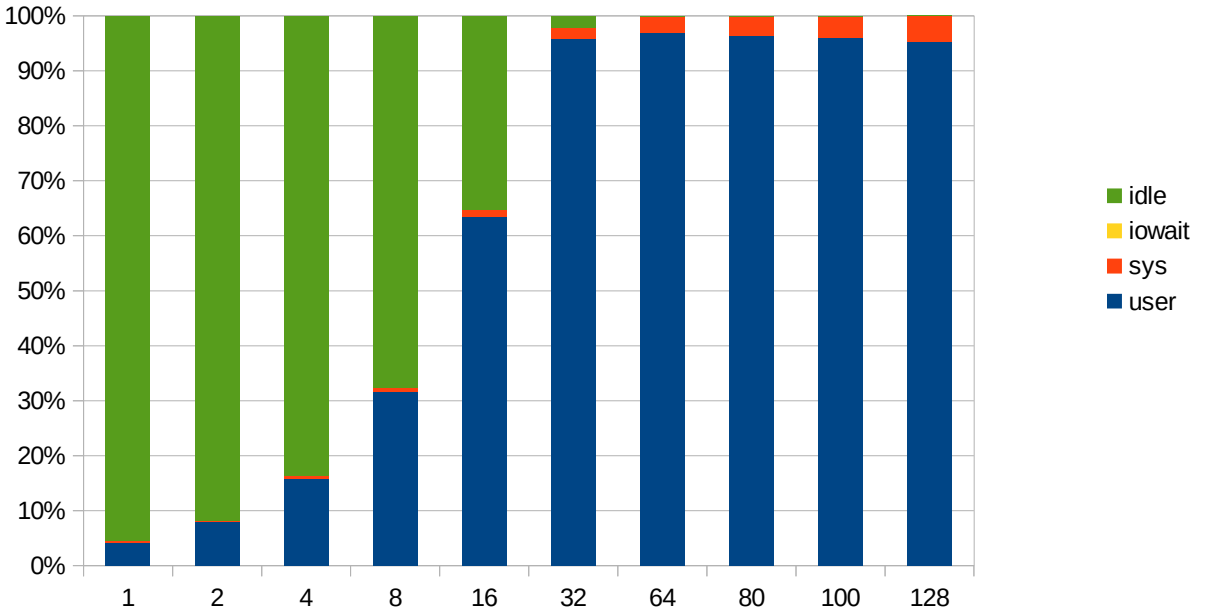

図 5.5.6: volume あり、仮想ネットワークなしのリソース利用状況

5.5.2. 更新性能の検証

5.5.2.1. 検証目的

非コンテナ環境の PostgreSQL の更新性能を基準に、コンテナ環境での PostgreSQL の更新性能を比較します。コンテナ環境で PostgreSQL を走らせる際には、仮想ネットワークの有無、コンテナ内のファイルシステムの利用の有無の、計 4 パターンを測定し、比較します。

5.5.2.2. 検証方法

負荷かけサーバから pgbench で更新系トランザクションを 3 回実行し、その中央値を結果とします。ここで-N パラメータは、pgbench で pgbench_tellers と pgbench_branches を更新しないオプションです。-N を選んだ理由は、デフォルトのトランザクションでは小さなテーブル(pgbench_branches)への更新の集中がボトルネックになり、本来の更新性能が測定できないためです。

PostgreSQL の設定パラメータ、データベースの初期化方法などは検索性能の検証と同じなので、省略します。

```
pgbench -i -s 1000
pgbench -N -c 32 -T 600
```

Docker コンテナの起動コマンドは各ケースで以下のようになります。

表 5.5.4: Docker コンテナの起動コマンド

volume なし、仮想ネットワークなし	`docker run -i --name="pgsql94" --net=host --rm -v /RHEL65:/vol/RHEL65 -t postgresql_9_4_rc1`
volume なし、仮想ネットワークあり	`docker run -i --name="pgsql94" --rm -p 5432:5432 -v /RHEL65:/vol/RHEL65 -t postgresql_9_4_rc1`
volume あり、仮想ネットワークなし	`docker run -i --name="pgsql94" --net=host --rm -v /RHEL65:/vol/RHEL65 -v /vv31/vol:/vol -t postgresql_9_4_rc1`
volume あり、仮想ネットワークあり	`docker run -i --name="pgsql94" --rm -p 5432:5432 -v /RHEL65:/vol/RHEL65 -v /vv31/vol:/vol -t postgresql_9_4_rc1`

5.5.2.3. 検証結果

測定結果を示します。

表 5.5.5: 測定ケースの概要

コンテナFS	仮想ネットワーク	名称
-	-	非コンテナ
X	X	ケース1
X	O	ケース2
O	X	ケース3
O	O	ケース4

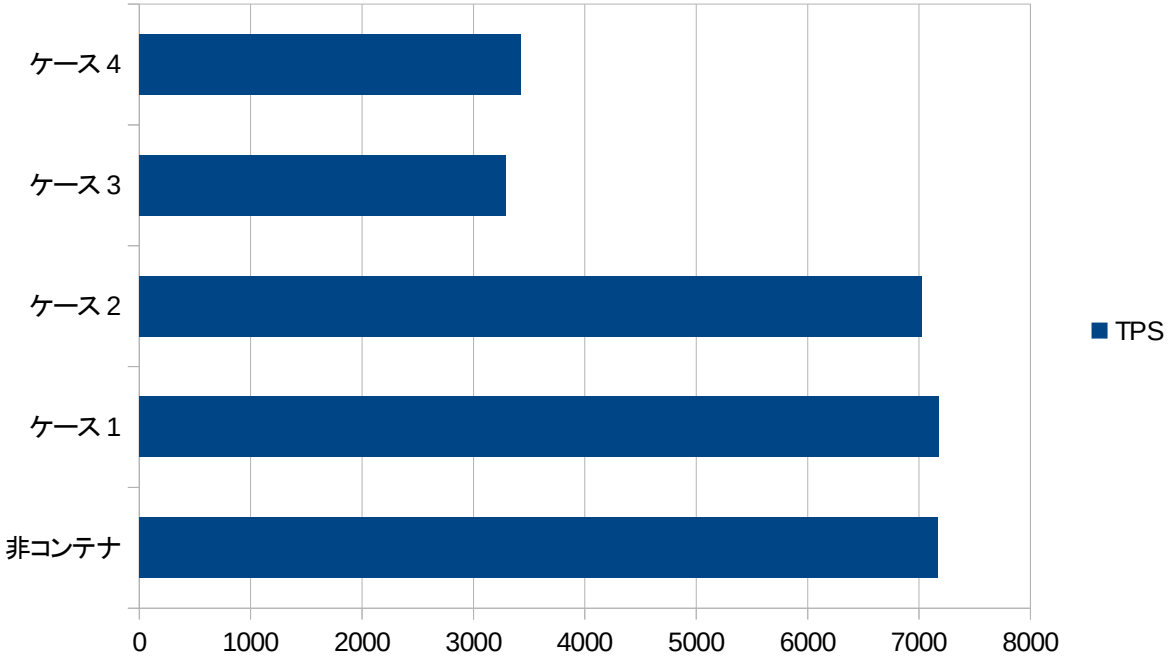

図 5.5.7: 各テストケースでの TPS 値

コンテナ内のファイルシステムを使うグループ(ケース 3, 4)と使わないグループ(ケース 1, 2)で大差が付いています。コンテナ内のファイルシステムを使わないグループでは、非コンテナの場合とほぼ同等の性能が出ています。

これはコンテナ内のファイルシステムの実装に使われている LVM thin provisioning のオーバヘッドによるものと考えられます。そこで、リソースの使用状況を sar で見てみます。

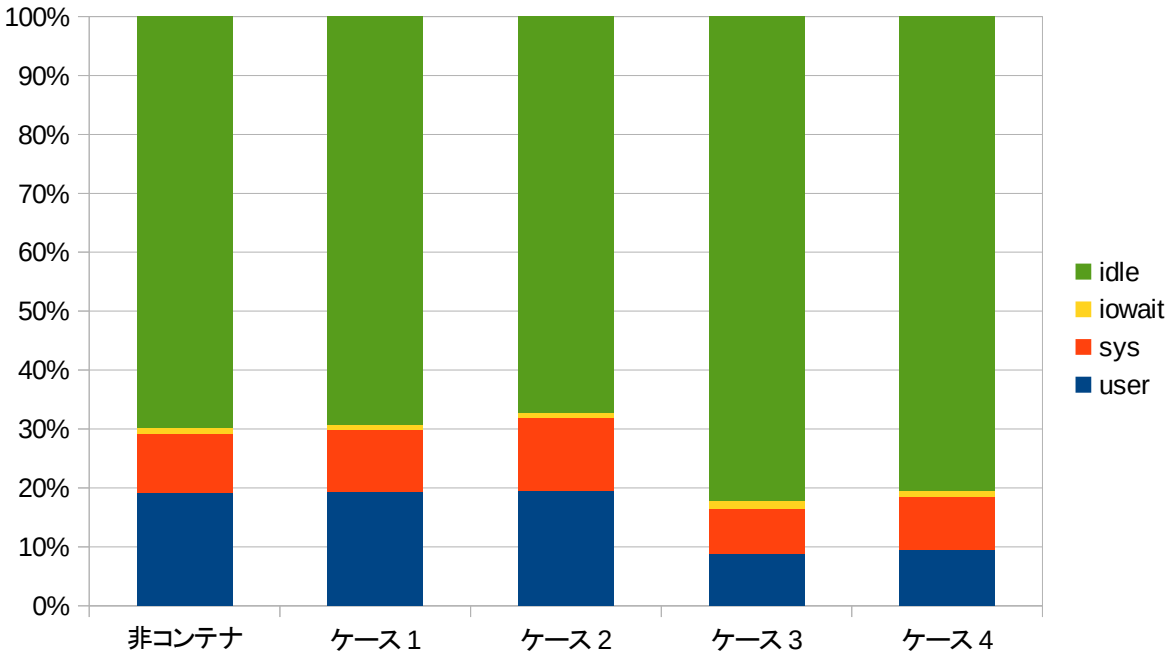

図 5.5.8: 各テストケースでのリソース使用状況

ご覧のように、各グループとも IOwait は僅かであり、主に user の部分に差異があり、続いて sys にもわずかに差異がある傾向です。これは、devicemapper 経由での書き込みか他の部分でのロック待ちが多く発生しているのではないかと推測されます。

もっとも、コンテナ内にデータベース領域を持つような運用は、運用的にデメリットこそあれ、メリットがないので通常は行わないと思われます(たとえばコンテナが破棄されるとデータが失われる)。したがってコンテナ内にデータベース領域を持つことによる性能劣化は、Docker の導入、運用に関して致命的な阻害要因にはならないでしょう。

5.6. 考察

Docker コンテナで PostgreSQL を動作させた場合の性能を、Docker コンテナを使わない PostgreSQL の性能と比較しました。検証は参照系と更新系に分けて実施しました。

本検証ではDB領域はメモリに乗った状態であり、この環境では参照性能はコンテナを使わない場合とほとんど変わらないことが分かりました。

一方更新系では、ファイルシステムをコンテナ内のものを使用する場合とそうでない場合では、コンテナ内のファイルシステムを使用する方が大きな性能劣化を見せました。本検証の環境では、そのボトルネックは I/O 待ちではなく、何らかのロック待ちによることが sar の結果から推測されました[4]。ただ、更新処理をともなうミッションクリティカルな用途でコンテナ内でDBMSを動かす場合、コンテナ内にデータを格納することはあまり実用的ではなく、ホストファイルシステムを利用することになると考えられるので、今回の検証結果でコンテナ内のファイルシステムを使った更新系の性能が悪いことが実用上問題になることはあまりないでしょう。

最後に、Docker の安定性ですが、本検証中にハングアップや不審な挙動を見せたことはなく、十分実用レベルに達していると思われます。

4ボトルネックを軽減する技術的な対策が Linux ディストリビュータから提供されている場合があります(たとえば Red Hat の Red Hat Atomic Host)。

6. おわりに

今年度(2014 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 3 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 3 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 4 つのテーマの検証結果を報告しました。

1. 2014 年 12 月にリリースされた、PostgreSQL9.4 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. PostgreSQL9.4 の改善項目の一つである、トランザクションログ(WAL)の生成方式の改善による性能向上の検証
3. Linux OS の仮想化技術である KVM 上で PostgreSQL を動作させたときの性能特性についての検証
4. Linux OS のコンテナ技術である Docker 上で PostgreSQL を動作させたときの性能特性についての検証

これらのテーマを継続・新規と大きく 2 つに分けると、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する(「定点観測」と呼んでいます)、従来からの継続性のあるテーマ、3,4 番は近年利用が増加している仮想化された OS 環境やコンテナ化された OS 環境上での PostgreSQL の性能傾向を見るという新しいテーマになります。

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。今回のテーマの 3,4 番はメンバからの提案に加えて、アンケートの結果を参考に採用されたものです。
検証テーマが決まれば、具体的な検証方法を検討します。ベンチマークプログラムを実行して数値が出ればそれで良しとする風潮が往々にしてありますが、そのプログラムによって適切に負荷が掛けられたかの確認や、性能を制約しているボトルネック、および、環境に由来するオーバヘッドの調査を通じて、より詳細に性能が確認できる方法を検討します。たとえば、テーマ 1 番では多コア CPU に対する PostgreSQL の性能を調べるために、負荷の掛け方を調整しています。
実際に測定を進めるために、検証テーマとマッチするハードウェア・OS 等を選びますが、その際には、会員企業が保有する高性能な機器を用いて、最新の環境に対する PostgreSQL の適用性を調べるようにしています。測定結果が出始めると、WG の会合で紹介されて、結果の妥当の確認やボトルネックやオーバヘッドの解明のために議論します。それらをフィードバックすることで、検証の有効性を少しでも高めるように測定作業を進めます。
こうした作業を積み重ねることで、単にベンチマークプログラムの測定値に留まらない、有用な結果が得られるように検討を進めます。たとえば、2 番のテーマでの WAL 生成の方式変更がどのようにボトルネックに影響を与えているかの考察や、4 番のテーマでのコンテナ FS のオーバヘッドの確認をご覧いただければと思います。これらは、日常の業務で PostgreSQL に深く係わり、その挙動やソースコードを熟知したメンバが集まって初めて達成できた成果です。PGECons の会合は、異なる企業に属するメンバが集まって、専門的な意見を交わす貴重な場となっており、メンバの技術力の向上にも寄与しているものと思います。

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[5]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)

5 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.

著者

版	所属企業・団体名	部署名	氏名
2014 年度 WG1 活動報告 大規模 DB を見据えた PostgreSQL の性能検証 第 1.0 版 （2014 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 OSS 技術グループ	長田 悠吾
	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
	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