PGECons
PostgreSQL Enterprise Consortium

PostgreSQL エンタープライズ・コンソーシアム 技術部会
性能ワーキンググループ(WG1)

2012 年度 WG1 活動報告書

改訂履歴

版	改訂日	変更内容
1.0	2013/04/22	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 及び関連クラスタソフトのスケールアップ、スケールアウト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電気株式会社、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。

■商標について

目次

1. はじめに

1.1. 2012 年度 WG1 活動テーマ

1.1.1. 活動テーマ決定経緯

WG1 の大きな目的は「大規模基幹業務に向けた PostgreSQL の適用領域の明確化」です（2012/7/6 開催の PGECons セミナーより）。このテーマを技術部会では課題領域を以下の大区分に分類しました。

表 1.1: PGECons における課題領域

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

性能に関しては更に以下の小区分に分解し、議論を深めました。

表 1.2: 性能検証テーマ

性能評価手法	オンラインやバッチなどの業務別性能モデル、サイジング手法
スケールアップ	マルチコア CPU でのスケールアップ性検証
スケールアウト	負荷分散クラスタでのスケールアウト性検証
性能向上機能	クエリキャッシュ、パーティショニング、高速ロードなど
性能チューニング	チューニングノウハウの整備、実行計画の制御手法

1.1.2. 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、コミュニティでは PostgreSQL 9.2 で 64 コアまでスケールするという結果も報告されていますが[1]、PGECons として自分たちで検証する必要があるのではないか、可能ならば更に大きなコア数を持つマシンで PostgreSQL の性能限界を探ってみよう、ということになりました。

そしてその結果を公開することにより、PostgreSQL の CPU マルチコア性能に対する不安を払拭していけるのではないかと考えました。

また、もしも検証の過程で技術的な問題が見つかった場合には、PostgreSQL の開発コミュニティにフィードバックすることになりました。

1.1.3. スケールアウト

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は、以下のような課題が抽出されました。

- 負荷分散の方式が確立していない
  OSS の範疇では、書き込みを含めた負荷分散の方式が確立されていない。

- 同期レプリケーション方式が確立していない
  PostgreSQL のレプリケーションは準同期であり、任意時点で分散先のデータが古い可能性がある。

同期レプリケーションに注目すると、pgpool-II[2]と Postgres-XC[3]というソリューションがあります。ただし、pgpool-II については書き込み負荷分散ができない、Postgres-XC は書き込み負荷分散が可能なものの、実績に乏しい、という問題があり、現時点での有用性を再確認するためにも、これらのソリューションの適用領域について検証を行なうことになりました。

一方、同期レプリケーションではありませんが、PostgreSQL 9.2 ではカスケードレプリケーションが実装され、その適用領域についても併せて検証することになりました。

1 Robert Haas (2012), “Did I Say 32 Cores? How about 64? “, http://rhaas.blogspot.jp/2012/04/did-i-say-32-cores-how-about-64.html
2 http://www.pgpool.net/
3 http://postgres-xc.sourceforge.net/

1.1.4. 実施体制

2012 年 7 月 26 日に開催された第 9 回技術部会（同日に第 1 回 WG1 会合開催）で以下の体制で実施することが決まりました（企業名順）。

表 1.3: 2012 年度 WG1 参加企業一覧

株式会社アシスト
SRA OSS, Inc.日本支社
NEC ソフト株式会社
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、SRA OSS, Inc.日本支社は、「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。

1.1.5. 実施スケジュール

本作業は技術部会での準備の後、2012 年 7 月 26 日（第 9 回技術部会（同日に第 1 回 WG1 会合開催））から正式作業開始しました。

表 1.4: 実施スケジュール

2012 年 7 月 26 日 （第 9 回技術部会（第 1 回 WG1 会合）にて決定）	参加企業確定、WG1 スタート
2012 年 8 月～9 月	実施計画策定
2012 年 10 月～11 月	検証実施
2012 年 12 月 7 日	中間発表セミナー
2013 年 1 月～3 月	2012 年度 WG1 活動報告書作成
2013 年 4 月 22 日	総会と成果報告会

2. スケールアップ検証

2.1. 概要

近年、コンピュータに搭載される CPU のマルチコア化やメモリの大容量化がますます進んでいます。また、PostgreSQL も、これらハードウェアの進化への対応やエンジンとして性能改善などが継続的に実施されており、メジャーバージョンが上がるにつれて性能が改善してきています。実際に、先ほども触れていますが、PostgreSQL9.2 ではロックアルゴリズムの改善が行なわれ、コミュニティからも参照系の 64core までのスケールアップ検証が報告されました。

この様に、大規模なデータベースや性能が求められるデータベースシステムに PostgreSQL が適用できる可能性がますます広がっています。そこで、本 WG では、OLTP(On Line Transactional Processing)をイメージした参照系の検証と、更新系の両方を実施し、現在の PostgreSQL の性能の特性や限界などを測定してみることにいたしました。

なお、今回の検証にあたり、80 core の CPU、メモリ 2 TB という 2012 年秋ごろとしては非常に高いスペックのマシンを利用しています。

参照系については、PostgreSQL に同梱されるツールである、pgbench を使用してコア数やクライアント接続数がどのように変化するか検証を行ないました。また、9.2 の新機能である、Index Only Scan の性能検証も行ないます。

更新系については、汎用のデータベースのベンチマークツールである JdbcRunner の Tiny TPC-C を用いて、データベースサーバのコア数やセッション数の変化により、どのような性能特性を示すのかを実際に検証いたしました。また、更新系の検証にあわせて、初期データロードや VACUUM など、運用面で気になるポイントについても検証を行なっています。

本年度のスケールアップ検証で WG1 で検討した内容をまとめると、以下の表のようになります。

表 2.1 2012 年度スケールアップ検証でやりたいこと

項番	作業内容
1	pgbench による検索性能検証(CPU コア数を変えた時の性能)
2	pgbench による検索性能検証(クライアント接続数を変えた時の性能)
3	Pgbench による Index Only Scan 検索性能検証(CPU コア数を変えた時の性能)
4	JDBCRunner による検索/更新性能検証
5	JDBCRunner を用いた、運用面で主に気になるポイント（初期ロードの時間や VACUUM など）についての検証

2.2. pgbench によるスケールアップ検証

2.2.1. 検証目的

pgbench によって検索（SELECT）負荷をかけ、コア数に応じて性能向上（スケールアップ）することを確認します。

（1） pgbench とは

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ないます。
-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
```

2.2.2. 検証構成

（2.3.2 と同じ）

2.2.3. 検証方法

postgresql.conf を編集します 。ディスク I/O の影響を極力排除し、テーブルデータをメモリにすべてのせるために shared_buffers を十分大きくしています。

```
$ vi $PGDATA/postgresql.conf

listen_addresses = '*'
max_connections = 510
shared_buffers = 20GB
work_mem = 1GB
checkpoint_segments = 16
checkpoint_timeout = 30min
logging_collector = on
logline_prefix = '%t [%p-%l]
```

test というデータベースを作成し、pgbench -i で内容を初期化します。スケールファクタ 1000、データ行数 1 億件、データベースサイズ 15 GB のデータベースです。

```
$ createdb test
$ pgbench test -i -s 1000

test=# select count(*) FROM pgbench_accounts ;
   count
------------
 100000000
(1 row)

test=# select pg_size_pretty(pg_database_size('test'));
 pg_size_pretty
----------------
 15 GB
(1 row)
```

（1） 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かったためです。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。

（2） 結果

結果を以下の図に示します。縦軸は tps（1 秒間に実行したトランザクション数）、横軸は同時接続クライアント数（pgbench の -c オプション）です。80 コア環境でコア数と同じ 80 クライアントまでスケール（tps が上昇すること）、すなわちコア数が多いとそれだけ多数の接続とトランザクションを処理できるということが確認できました。
また、クライアント数がコア数を超過したときは、tps が下がることを確認しました。

表 2.2: 結果 tps

-c	-j	tps	CPU 使用率
1	1	361.38950	2%
2	1	692.45110	2%
4	2	1383.02231	3%
8	4	2058.93810	6%
16	8	4358.79516	10%
32	16	9034.48321	19%
64	32	15845.40369	38%
80	40	17361.53462	47%
90	45	16824.25192	53%
100	50	13823.17011	59%
128	64	1358.21827	69%

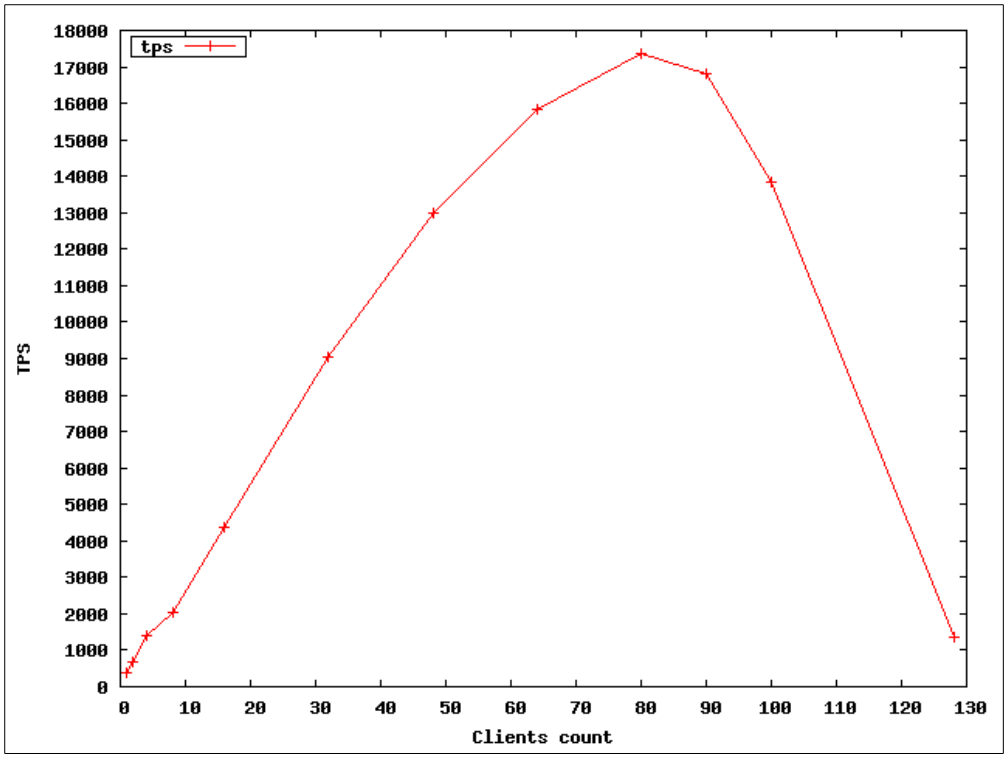

図 2.1: 負荷の高い参照系スクリプトでクライアント数を変動させたときの結果 tps

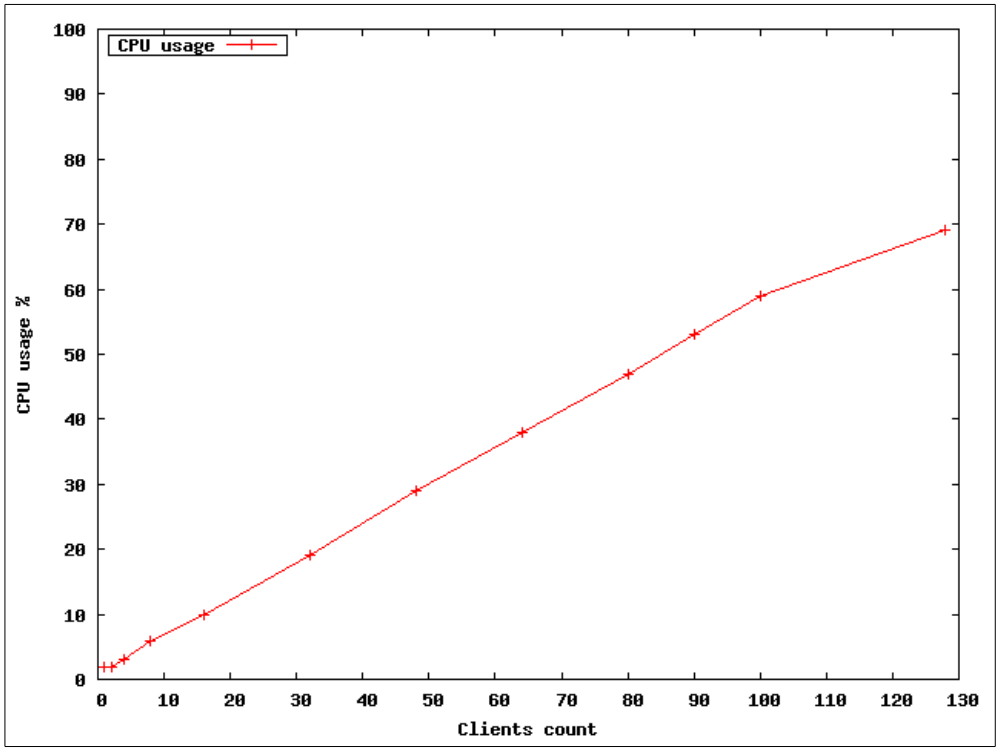

図 2.2: 負荷の高い参照系スクリプトでクライアント数を変動させたときの CPU 使用率

なおこの測定はハイパースレッド on の状態で行なっており、論理的にはこの倍の 160 コアとなっています。

参考までにハイパースレッド off で実行したときの結果も記載します。全体的に on 時より off 時の方が tps が若干高く、特に最後の 128 クライアント接続時の結果が大きく異なりました。

表 2.3: 結果 tps

-c	-j	tps
1	1	362.59
2	1	695.88
4	2	1373.25
8	4	2557.79
16	8	4450.44
32	16	8960.65
48	24	13802.14
64	32	17453.96
80	40	19225.11
90	45	18785.79
100	50	18308.97
128	64	12621.04

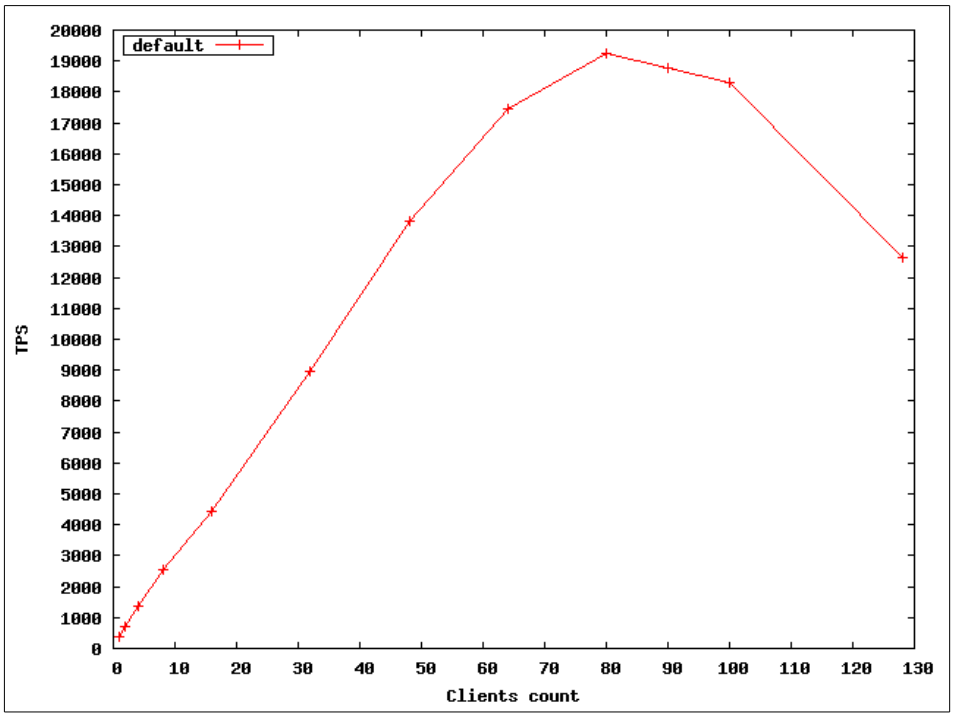

図 2.3: 負荷の高い参照系スクリプトでクライアント数を変動させたときの結果 tps（ハイパースレッド off）

2.2.4. 考察

（1） 結果

クライアントの同時接続数を増やすことにより、物理コア数と同じ 80 クライアントまでほぼリニアに性能が向上し、PostgreSQL の CPU スケーラビリティが確認できました。

予備検証では pgbench の標準シナリオ（pgbench -S）を実施しましたが、32 クライアントまでしかスケールしませんでした。このとき CPU idle がほぼ 100%であり、pgbench の標準シナリオでは充分に負荷をかけられませんでした。
そこで、前記のようにカスタムスクリプトを作成して負荷をかけたところ、80 コアまでスケールする結果が得られました。

（2） ロック競合？

また、クライアント数が多いためにロック競合がおきやすくなるのを、ソースコード内の src/include/storage/lwlock.h の設定値を変更すると改善する、という報告[4]があり、これも試しました。しかしながら本事象はロック競合が原因ではなかったようで、結果に変わりはありませんでした。

```
/* Number of partitions of the shared buffer mapping hashtable */
#define NUM_BUFFER_PARTITIONS  256 <-- 16 から変更

/* Number of partitions the shared lock tables are divided into */
#define LOG2_NUM_LOCK_PARTITIONS  6 <-- 4 から変更
#define NUM_LOCK_PARTITIONS  (1 << LOG2_NUM_LOCK_PARTITIONS)
```

4 堀川 隆（2012）, pgbench による Early Lock Release の評価（+ついでに Postgres 9.2 の強化点）, http://www.postgresql.jp/wg/shikumi/study24_materials/20120929_horikawa.pdf

2.3. 更新系処理におけるスケールアップ検証

PostgreSQL9.2 のスケールアウト検証は参照系のスケールアップ検証はコミュニティから報告されていましたが、更新系のスケールアップ特性の検証結果を目にすることがありませんでした。そこで、WG1 では、更新を中心とした処理の場合に、データベースサーバのコア数やセッション数の変化によりどのような性能特性を示すのかを実際に検証してみることにいたしました。

2.3.1. 検証目的

検証を行なうに当たり、本団体はエンタープライズ領域への PostgreSQL の適用を促進することを目標としていることから、今回は単純な素性能ではなく運用面で必要な項目についても考慮した検証を実施することとしました。

例えば、INSERT や UPDATE などの更新処理が多い OLTP を中心としたデータベースシステムにおいては、実際のエンタープライズの現場で運用上考慮すべき点いくつか出てきます。
代表的な項目を挙げると、PostgreSQL が追記型アーキテクチャを採用しているために DELETE や UPDATE に伴うガーベジを回収する VACUUM（AUTO VACUUM）や、実際に更新されたメモリ中のバッファデータををストレージに書き込む処理である CHECKPOINT 時に発生する I/O の影響などがあります。
またエンタープライズ領域に必要不可欠なバックアップからのリストアに必要なアーカイブログの出力なども実運用では必要になってきます。
当然、システムの性能要件で検討するに当たり、アプリケーションからの接続数やデータベースに要求される負荷の内容、データベースのサイズなど様々な点を検討する必要があります。

本検証では、エンタープライズとして使われることが想定できる比較的大規模のデータで、かつ必要な運用面に対する考慮がなされた環境にあったとしても、チューニングなどである程度正しい設定を行なうことで、更新系の処理でも PostgreSQL が CPU 数や接続数に応じてスケールアップすることを実機検証により明らかにすることを目的としています。

さらに、せっかくの大規模データのデータ投入を行なう機会ですので、初期データロードで使われる COPY 文や CREATE INDEX、ANALYZE の性能検証も行なってみようということになりました。当然マシンスペックやデータの特性などに左右されることはありますが、実際の環境でデータを投入される場合にどの程度時間が必要になるのかのおおよその目安として計測いたしました。また、大規模なデータを投入したからといって、データ量に比例した性能以上の劣化がないことを確認します。

以上の検討の結果、今回更新系処理におけるスケールアップ検証として実施した測定項目をまとめると以下のようになります。

1. 初期データのロード性能(COPY 文+CREATE INDEX)
2. コア数やセッション数の変更による傾向
3. shared_buffer 変更による傾向
4. データベースサイズによる性能への影響
5. CHECKPOINT や VACUUM による性能の影響

2.3.2. 検証構成

今回は、評価対象の PostgreSQL9.2.1 を搭載するデータベースサーバと負荷をかけるクライアントサーバ 2 台を用意し、またデータベース格納用に SAN のストレージを用意いたしました。主なスペックと構成は以下の図のようになっています。

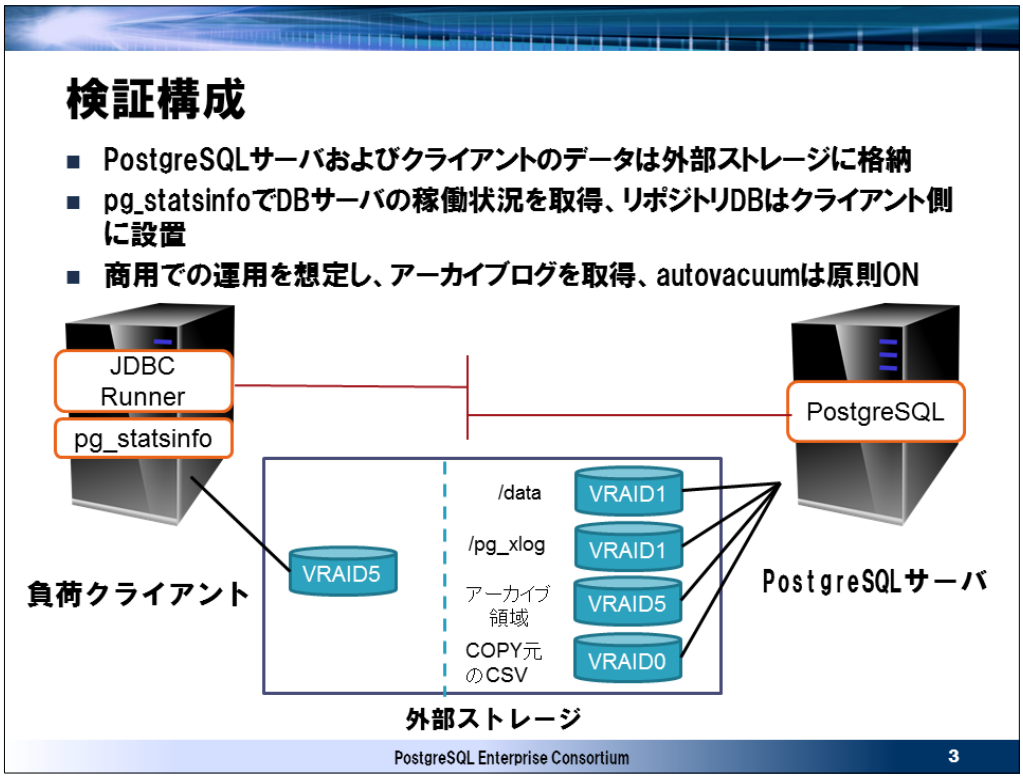

図 2.4　検証構成

また、検証環境にセットアップしたソフトウェアやハードウェアは以下のようになっています。

表 2.4　検証環境一覧

検証環境項目		内容
PostgreSQL サーバ	ハードウェア	CPU :Intel® Xeon® E7-4870 (2.4GHz 10core) 8 Processor / 合計 80Core Memory:2TB 内臓 HDD:900GB × 8 ネットワークカード:Gigabit Ethernet
	OS	Redhat Enterprise Linux 6.2
	PostgreSQL サーバ	PostgreSQL 9.2.1
	テストツール	JdbcRunner V1.2 を改造した CSV 生成ツール
クライアントサーバ	ハードウェア	CPU:Intel® Xeon® E5-2690 (2.9GHz 8core) 2 Processor / 合計 16Core メモリ:64GB 内臓 HDD:　300GB × 4 ネットワークカード:Gigabit Ethernet

	OS	Redhat Enterprise Linux 6.2
	pg_statsinfo 用 PostgreSQL	PostgreSQL 9.2.1 pg_statsinfo 2.3.0（独自修正版） [測定当時は PostgreSQL9.2 に対応したバージョンがなかったため、独自に修正を行ないましたが、現在は 2.4.0 にて対応しています]
	テストツール	JdbcRunner v1.2
ストレージ		HDD:900GB (6G SAS 10K 2.5inch) × 100 VRAID0（HDD20 本） × 1 VRAID5（HDD20 本） × 1 VRAID1（HDD20 本） × 2
ネットワーク		Gigabit EtherNet Switch で構築

負荷評価を行なうに当たり、今回は JdbcRunner という汎用データベース負荷テストツールを使いました。
JdbcRunner は各種データベースシステムを対象とした負荷テストツールです。また、Oracle Database や MySQL そして PostgreSQL を対象とした Tiny SysBench、Tiny TPC-B、Tiny TPC-C といったテストキットが付属しており、これらを利用することで簡単に負荷評価を行なうことができます。

今回は、更新処理を中心としたオンライントランザクションの一般的なベンチマークである TPC-C をベースにした Tiny TPC-C を利用しました。なお、この Tiny TPC-C は、TPC-C Standard Specification 5.10.1 の仕様を抜粋し 、JdbcRunner のスクリプトとして実装されたものになります。そのため、一般的な TPC-C のベンチマークすべてに対応しているわけではありません。

その他、JdbcRunner や Tiny TPC-C の検証内容詳細については、JdbcRunner の Web[5]をご確認ください。

なお、今回の評価に当たり、COPY 文の性能測定をするため、JdbcRunner の一部を改変しました。
JDBC Runner の Tiny TPC-C のデータ生成用スクリプト scripts/tpcc_load.js は以下の流れで作成されています。

1. テーブルの削除
2. テーブルの作成
3. INSERT 文にて各テーブルのスケールファクターに相当するデータ件数を投入
4. INDEX の生成
5. ANALYZE および VACUUM の実行

今回は 3.のデータ投入部分では、1つのスケールファクターごとにループしてデータを生成しており、全てのテーブルにまんべんなく INSERT 文が発行するように作られています。今回の検証では、COPY 文の性能測定を目的として、各テーブルごとのデータの CSV ファイルを、一度出力した後で、COPY 文を実行するように改変を行なっています。
修正したスクリプトの流れとしては以下のようになります。

5 JdbcRunner のホームページ: http://hp.vector.co.jp/authors/VA052413/jdbcrunner/

1. 各テーブルのスケールファクターに相当する CSV ファイルを生成
2. テーブルの削除
3. テーブルの作成
4. INDEX の生成
5. ANALYZE および VACUUM の実行

上記、データ生成を行うスクリプトも含めて、実行環境は jdbcrunner-1.2-pgecons2012.tar.gz にまとめて PGECons のホームページのリンクからダウンロードができます。

また、PostgreSQL の稼働状況を収集する為に、pg_statsinfo を導入しています。pg_statsinfo は PostgreSQL や OS のリソース情報、統計情報をスナップショットとして取得することを目的に開発されたオープンソースのツールです。これにより、PostgreSQL の活動状況や性能のボトルネックの調査、原因の切り分けなどが可能になります。
今回はクライアントサーバ側に、この pg_statsinfo の情報を格納するリポジトリサーバとして、PostgreSQL を構築しています。
ただし、検証を行なった 2012 年 10 月現在では、PostgreSQL9.2 には対応をしていなかったため、WG1 では pg_statsinfo2.3.0 をベースに PostgreSQL9.2 で動作するような簡易的な修正を行なっています。2013 年 1 月 29 日には pg_statsinfo2.4.0 がリリースされ、正式に PostgreSQL9.2 に対応しておりますので、今後はそちらを利用してください。

その他、実際の運用を想定し、アーカイブログの取得を実施します。ストレージの配置は、データベース、WAL 領域、アーカイブログをそれぞれ VRAID1、VRAID1、VRAID5 と別のディスクボリュームに設定いたしました。

2.3.3. 検証方法【1】初期データのロード性能(COPY 文+CREATE INDEX) [DB サーバ側]

（1） 環境作成 (postgresql.conf)

データロード用に postgresql.conf を編集します。初期ロードでは WAL 出力をスキップすることで、データロードの高速化が可能になります。そのため、archive_mode = off にし、wal_level = minimal にしています。また、shared_buffer を十分な値であろうという 50GB（サーバの規模からすると小さい値ではありますが）にしました。

また、出来るだけ I/O の過度の高負荷にならないように平準化をさせるため、checkpoint_segments を 256、checkpoint_completion_target を 0.9 と大きく設定しています。

その他、pg_statsinfo 取得用にパラメータを設定しています。

```
#
# Memory
#
effective_cache_size = 1024GB
maintenance_work_mem = 512MB
shared_buffers = 50GB
wal_buffers = 512MB
work_mem = 256MB

#
# Connection
#
listen_addresses = '*'
max_connections = 1100

#
# I/O
#
checkpoint_completion_target = 0.9
checkpoint_segments = 256
checkpoint_timeout = 10min
effective_io_concurrency = 20

#
# WAL
#
wal_level = minimal
wal_sync_method = fdatasync
wal_writer_delay = 200ms
archive_mode = off
archive_command = 'test ! -f /test4/archive/%f && cp %p /test4/archive/%f'

#
# Autovacuum
#
autovacuum = on
autovacuum_max_workers = 3
autovacuum_naptime = 1min
autovacuum_analyze_scale_factor = 0.1
```

```
autovacuum_analyze_threshold = 50
autovacuum_vacuum_scale_factor = 0.01
autovacuum_vacuum_threshold = 50
autovacuum_vacuum_cost_delay = 20ms
autovacuum_vacuum_cost_limit = -1

# Log
log_destination = 'csvlog'
logging_collector = on
log_checkpoints = on
log_directory = '/usr/local/pgsql/pg_log'
log_autovacuum_min_duration = 0

# pg_statsinfo
shared_preload_libraries = 'pg_statsinfo'
pg_statsinfo.textlog_min_messages = warning
pg_statsinfo.syslog_min_messages = disable
pg_statsinfo.textlog_filename = 'postgresql.log'
pg_statsinfo.textlog_line_prefix = '[%t, %d, %u, %p] '
pg_statsinfo.textlog_permission = 0600
pg_statsinfo.sampling_interval = 5s
pg_statsinfo.snapshot_interval = 5min
pg_statsinfo.excluded_dbnames = 'template0, template1, postgres, repository'
pg_statsinfo.excluded_schemas = 'pg_catalog, pg_toast, information_schema'
pg_statsinfo.repository_server = 'host=172.17.177.11 dbname=repository'
pg_statsinfo.adjust_log_level = off
pg_statsinfo.enable_maintenance = off
pg_statsinfo.long_lock_threashold = 30
```

AUTOVACUUM については ON にしています。また、VACUUM の回数を減らすことができる HOT(Heap Only Tuple)を有効に活用できるように各テーブルには、おおよそ効果が出そうな値として FILLFACTOR を80に設定いたしました。
これら、HOT や FILLFACTOR などの概要が知りたい方は、Let's Postgres の「HOT の効果」[6]をご参照ください。

（2） テストデータ（CSV ファイル）の作成 (tpcc_datagen.js)

データ生成を行う前に、JdbcRunner のチュートリアル[7]や、scripts/tpcc_load.js のコメントを参考にして、JdbcRunner の環境設定(Jdbcrunner.jar への CLASSPATH の設定やデータベースおよびユーザの生成）を行ってください。なお、tpcc ユーザ COPY コマンドを実行しますのでスーパーユーザに設定してください。

テストデータ（CSV ファイル）の作成用に Tiny TPC-C のデータ生成用 JAVA スクリプト tpcc_load.js をもとに tpcc_datagen.js を用意いたしました。
改変した JdbcRunner のソースについては、PGECons のホームページにて公開をしています。

tpcc_datagen.js の実行イメージは以下となっております。

```
$ java JR ./tpcc_datagen.js -param0 (scale factor)
```

これにより/test1/csv/(scale factor)/の位置に各テーブルの CSV ファイルが生成されます。

6 Let's Postgres「HOT の効果」： http://lets.postgresql.jp/documents/tutorial/hot_1/
7 JdbcRunner チュートリアル： http://hp.vector.co.jp/authors/VA052413/jdbcrunner/manual_ja/tutorial.html

今回は、大規模データの特性を見るために検証する CSV のデータ量をおおよそ 100GB、250GB、500GB、1000GB で測定することにし、JdbcRunner の Tiny TPC-C で生成データのスケールファクターを 1500、3750、7500、15000 と設定してテストデータを、VRAID0 のディスクボリュームである/test1/に出力しています。

```
$ nohup java JR ./tpcc_datagen.js -param0 1500 -param1  &
$ nohup java JR ./tpcc_datagen.js -param0 3750 -param1  &
$ nohup java JR ./tpcc_datagen.js -param0 7500 -param1  &
$ nohup java JR ./tpcc_datagen.js -param0 15000 -param1  &
```

（注意!）
この100 GB 単位のデータ生成には非常に時間がかかります（今回はぶっつけ本番で行なってしまいました）。今回の場合 1000GB(スケールファクター 15000)のデータ生成にはおおよそ 30 時間が必要でした。評価環境への接続が切れた場合にはデータ生成をやり直しということもありましたので、余裕をもってマシンの実行時間を設定したうえで nohup コマンドを使用することをお薦めいたします。

（3） テーブルの作成、COPY コマンド、インデックスの作成、統計情報の更新

作成した CSV データを COPY コマンドで投入し、インデックスの作成を行ないました。これは tpcc_load.js をベースとしたスクリプト tpcc_copy.js および、それを実行する tpcc_copy.sh を用意しました。

実際の起動では tpcc_copy.sh を実行することで、PostgreSQL サーバの再起動、テーブルの削除・生成、各テーブルの CSV のコピーおよび、インデックスの生成、統計情報の更新の順で実行されます。
tpcc_copy.sh の実行イメージは以下になります。

```
$ tpcc_copy.sh (scale factor)
```

ここでは、事前に作成したスケールファクタ 1500、3750、7500、15000 のデータをそれぞれデータベースに投入しています。

```
$ tpcc_copy.sh 1500
```

なお、ここでは、CSV ファイルの出力先を変更したい場合は、tpcc_datagen.js および tpcc_copy.js の変数 csv_dir の位置を直接変更するようにしてください。

2.3.4. 検証方法【2】コア数やセッション数の変更による Tiny TPC-C 性能傾向の測定

（1） 基本性能計測

基本性能として、スケールアップ検証を行なうための条件を決定するため、以下の条件を変更し JdbcRunner によるトランザクション処理量（TPM）を計測しました。

- **条件 1**
  共有バッファサイズの変更
  80core、64 同時接続、データサイズ 100GB で共有バッファサイズを 8GB、16GB、80GB、160GB とした際の性能傾向を確認しました。

- **条件 2**
  データサイズ
  80core、64 同時接続、共有バッファサイズを 80GB とし、データサイズを 100GB、250GB、500GB、1TB とした際の性能傾向を確認しました。

（2） スケールアップ検証

スケールアップ検証では、基本性能計測の結果を元に、共有バッファサイズを 80GB、データサイズを 100GB とし、CPU 数が 40core、80core の 2 パターンで同時接続セッション数を最小 16 同時接続から最大 256 同時接続まで増加し、平均 TPM の変化を確認しました。

2.3.5. 検証方法【3】データベース内部処理による性能影響

PostgreSQL の運用時に性能影響を与えることが懸念される処理として、checkpoint と VACUUM が想定されます。これらの内部処理を停止した場合の JdbcRunner による更新処理性能を測定し、性能への影響度合いを確認しました。

2.3.6. 考察【1】データベース内部処理による性能影響

PostgreSQL で大量データの取り込みを行なう場合、カンマ区切り(.csv)ファイルやタブ区切り(.tsv)ファイルを用意し、COPY FROM コマンドを使用する方法が一般的です。データを取り込む対象のテーブルにインデックスや制約が定義されている場合、各行についてインデックスの更新や制約のチェックが行なわれるため、所要時間が増加します。COPY コマンドに限らず、INSERT や UPDATE など DML 文によるデータ操作であっても言えることですが、特に本検証のようにテーブルの初期データ投入や、大量データの取り込みはインデックスや制約が定義されていない状態で行ないます。

データ取り込み作業は以下の順序で行ない、それぞれの処理に要した時間を計測しました。

1. COPY 文（全テーブル Index なし）
2. CREATE INDEX（主キー）
3. CREATE FOREIGNKEY（参照整合制約）
4. VACUUM、ANALYZE

４つのデータベースクラスタを用意し、それぞれ取り込みデータ量を 100GB、250GB、500GB、1000GB とします。各処理ではテーブルの全データを扱うため、ディスク I/O が発生する処理であることを考えると、取り込みデータ量が大きくなると、所要時間がリニアに増加することが予想されます。各処理での所要時間を積み上げグラフで表したものが以下の図 2.5 となります。本検証結果より、データ量が増加すると、取り込みの所要時間はほぼリニアに増加することが確認できました。

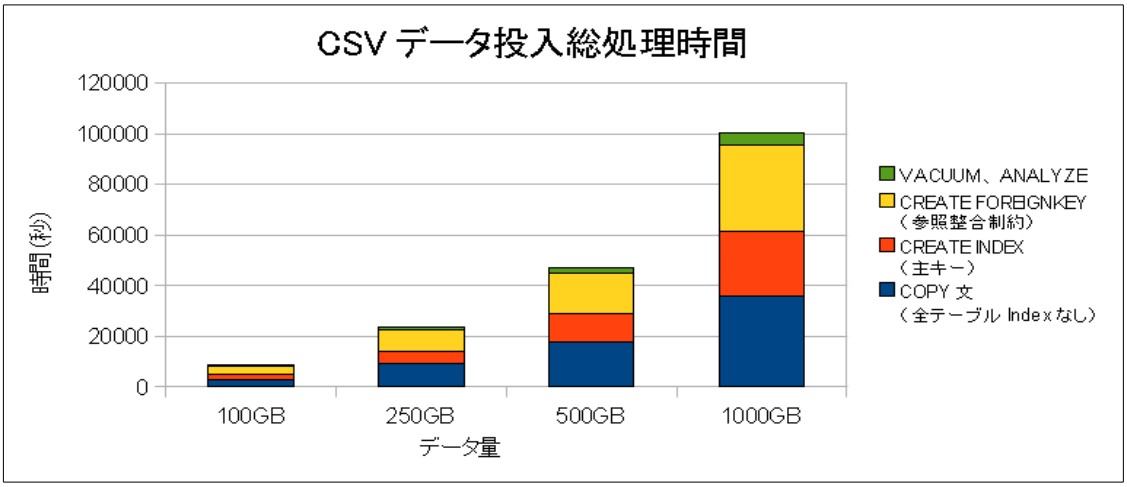

図 2.5: CSV データ投入総処理時間

また、COPY コマンド使用時は、WAL（変更履歴）データを生成しないようチューニングを行なうことで、より高速な取り込みが実現可能です[8]。）

8 （参考）大量のデータを高速に投入するには：http://lets.postgresql.jp/documents/technical/bulkload/

2.3.7. 考察【2】コア数やセッション数の変更による Tiny TPC-C 性能傾向の測定

（1） 基本性能計測

共有バッファ(shared_buffers)サイズおよび格納データサイズを変えて JdbcRunner による Tiny TPC-C トランザクション処理性能を計測しました。

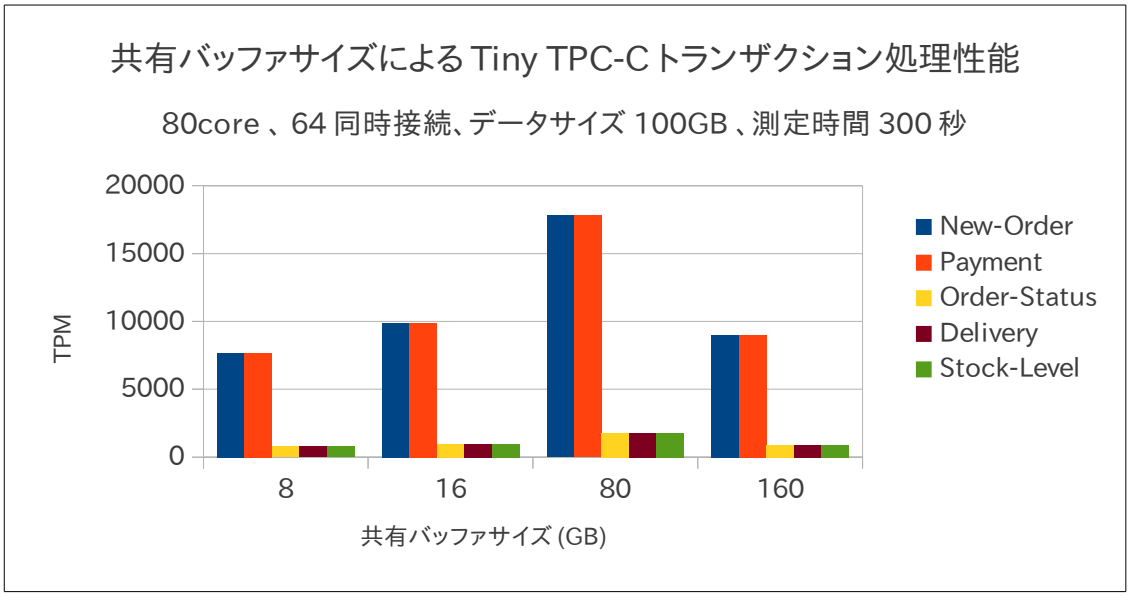

図 2.6: 共有バッファサイズによる Tiny TPC-C トランザクション処理性能

共有バッファサイズは、アクセス対象となるデータを効率的にキャッシュしておけるよう、十分大きな値としておくことが理想ですが、実際にはメモリ管理のオーバーヘッドが発生するため、一定のバッファサイズ以上では性能向上が見られないことがあります。本検証では 100GB の格納データサイズに対して 8192MB、16384MB、81920MB、163840MB の共有バッファを割り当てた際の性能を計測しました。アクセス対象データ範囲が 100GB であるため、100GB 以上の共有バッファサイズを割り当てた際に高い TPM が出ると考えていましたが、予想に反して、共有バッファサイズを 80GB とした場合に一番高い TPM を記録しました。また、共有バッファのサイズを 80GB とした場合と 8GB とした場合を比較すると約 40 パーセント程度までしか性能低下していません。他の RDBMS も含め、一般的にキャッシュヒット率が数パーセント低下すると処理性能は数十パーセント低下する、といったことが言われていますが、PostgreSQL では OS のカーネルキャッシュも活用するため、それほどの性能低下は見られなかったものと思われます。

80GB の共有バッファサイズに対して、格納データサイズを 100GB、250GB、500GB、1000GB と増加した場合の JdbcRunner による Tiny TPC-C トランザクション処理性能を計測しました。本検証ではデータサイズが n 倍に増加すると、TPM は約 1/n となることを確認しました。

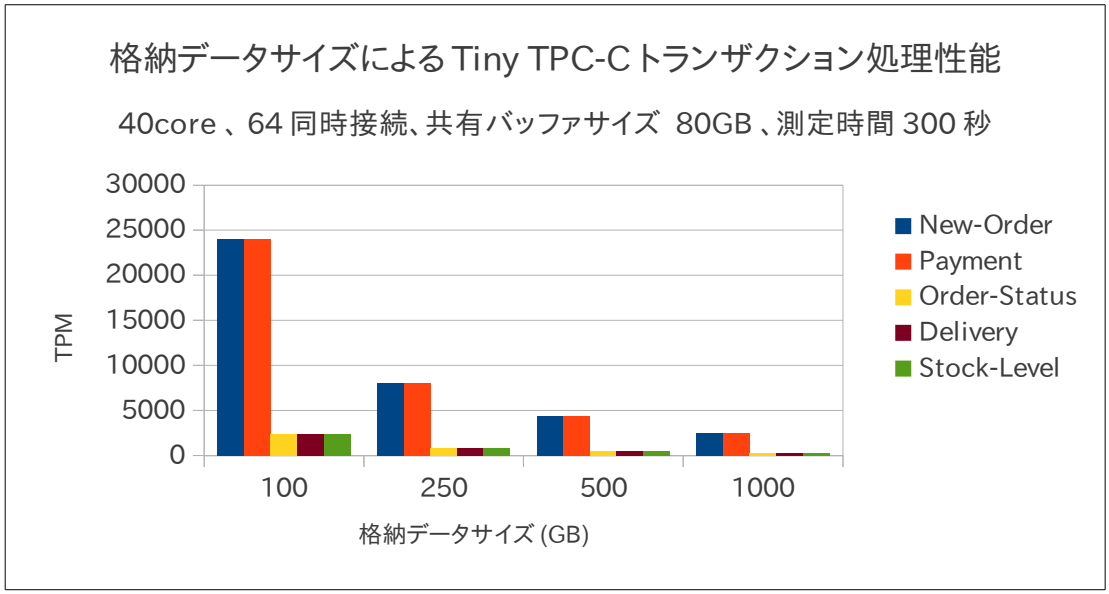

図 2.7: 格納データサイズによる Tiny TPC-C トランザクション処理性能

（2） スケールアップ検証

本検証では、80core、40core で同時接続セッション数を変動させて JdbcRunner による Tiny TPC-C トランザクション処理性能を計測しました。

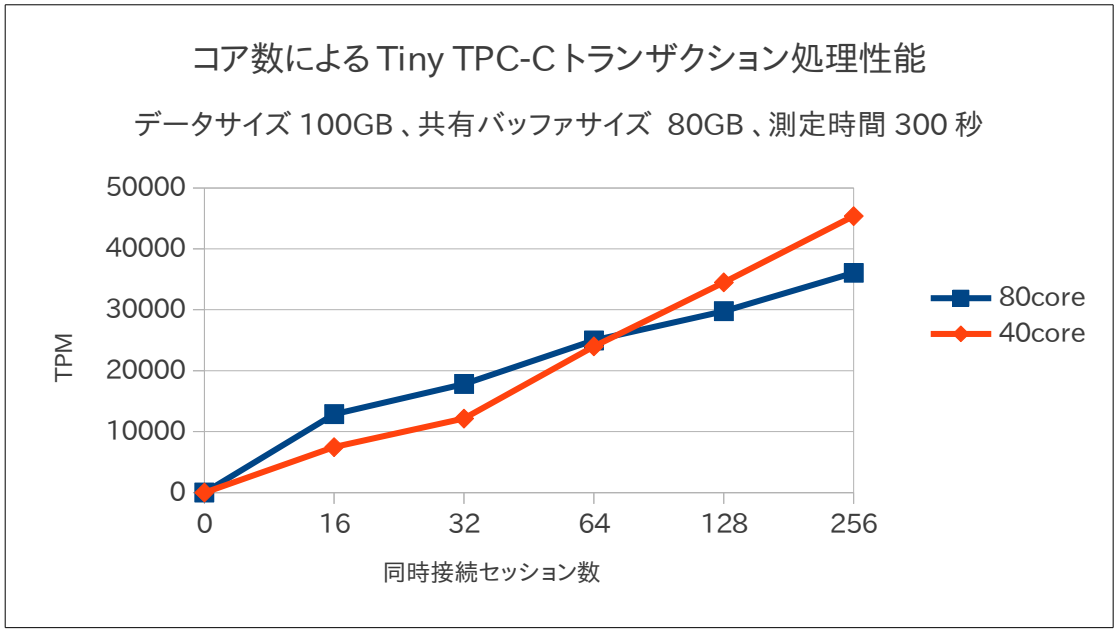

図 2.8: コア数による Tiny TPC-C トランザクション処理性能

本検証結果からは多数のコアを活用して性能向上しているという判断はつけ難く、特に期待していた同時接続数が増加し CPU 負荷が上がった場合の挙動を確認することができませんでした。原因としては、256 セッションまでの同時接続を測定しましたが、これでは負荷が足りず、40core、80core ともに CPU 使用率に余裕があったことが考えられます。

2.3.8. 考察【3】データベース内部処理による性能影響

チェックポイントでは共有バッファ上のデータとディスク上のデータを同期させるため、大量のディスク書き込みが発生します。PostgreSQL では、checkpoint_timeout や checkpoint_segments パラメータによるチェックポイント間隔の調整や、checkpoint_completion_target パラメータによるチェックポイント負荷の分散を検討することで、チェックポイントによる性能影響を抑えることを検討します。

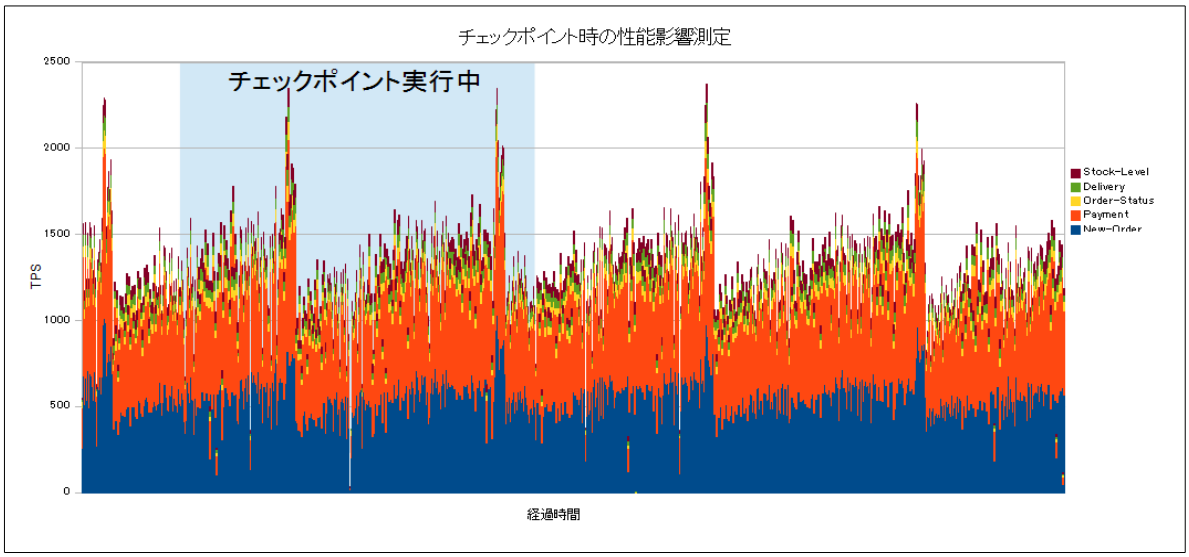

図 2.9: チェックポイント時の性能影響測定

本検証では、手動でチェックポイントを発生させた際の性能変化を確認しましたが、チェックポイント実行中にトランザクション処理量が低下することは無く、checkpoint_completion_target によるチェックポイント負荷の分散が有効に機能していることを確認できました。

また、PostgreSQL 独自の運用として、更新、削除によって不要となった領域を空き領域として登録し、再利用可能にする VACUUM 処理があります。PostgreSQL8.3 以降のバージョンではデフォルトで自動 VACUUM が有効になっています。VACUUM（自動 VACUUM を含む）処理はフロントの処理性能に影響を与えないよう、遅延 VACUUM の設定を検討します。上記検証時は自動 VACUUM を有効にし、遅延 VACUUM 設定をデフォルトとしていましたが、定期的に VACUUM が実行されることによる性能の落ち込みがあるものの、一定の周期で動作していることから、遅延 VACUUM の設定が有効に働いていることを確認できました。

2.4. スケールアップ検証サマリ

本年度のスケールアップ検証をまとめると、以下の表のようになります。

表 2.5 2012 年度 スケールアップ検証結果

項番	作業内容	実施状況	備考	次年度への課題
1	pgbench による検索性能検証(CPU コア数を変えた時の性能)	△	80 物理コアまで検証したが、32 コアまでのスケールアップしか確認できていない。CPU を使い切れていなかったた可能性がある。	CPU を使い切れるような負荷を設定する
2	pgbench による検索性能検証(クライアント接続数を変えた時の性能)	○	80 物理コアまでスケールすることが確認できた。	
3	pgbench による index-only scan 検索性能検証(CPU コア数を変えた時の性能)	△	80 物理コアまで検証したが、48 コアまでのスケールアップしか確認できていない。CPU を使い切れていたが、80 コアまでスケールしない理由は不明。	
4	JDBCRunner による検索/更新性能検証	△	80 物理コアまで検証したが、スケールが確認できなかった。	更新系のボトルネックを分析・解明する
5	JDBCRunner を用いた、運用面で主に気になるポイント（初期ロードの時間や VACUUM、checkpoint など）についての検証	○	初期ロード性能検証や AUTOVACUUM ON/OFF による性能の変化を確認できた	

参照系、更新系ともに PostgreSQL は CPU コア数が増えてもあるパターンにおいてはスケールアップすることを確認することができました。まだまだ、検証をしたいことはたくさんあり、その中の一部ではありますが、一定の性能パターンの検証を行なうことができました。また本 WG として、コミュニティとして検証活動を一とおり行なうことができ、活動のプロセスや注意点などもわかってきたことを含めると、今後につながる活動になったと考えています。

ただし、今回の検証では、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であったためか、PostgreSQL サーバ側のディスク I/O ネックである状況に落ちいってしまいました。このため、PostgreSQL を十分に使いきれていない可能性が高く、クライアントサーバ側の負荷や PostgreSQL サーバ側の設定など環境次第ではさらに性能が向上する可能性があることもわかっています。今年度実施しなかったパーティショニングなどを含めて、このあたりの原因究明については来年度以降の課題として調査・検討をしていきたいと考えています。

また、2013 年以降も PostgreSQL 9.3 がリリースされる予定になっています。WG1 では、新しいバージョンへの性能評価に備えて、来年以降も性能測定ツール選定や測定時の注意点などを整理し、今回出来なかった検証の実施やスケールアップ性能検証を今後行なっていく必要があると考えています。

3. スケールアウト検証

3.1. 概要

PostgreSQL で利用可能なスケールアウト方式としては、PostgreSQL 組み込みのレプリケーション技術であるストリーミングレプリケーションの他、pgpool-II、Postgres-XC などがあります。スケールアウト検証では、この代表的な 3 つの技術の検証を行ないました。

PostgreSQL 9.2 で実装されたカスケードレプリケーションでは、階層的なレプリケーションシステムを構成することが可能です。今回の検証では、本店と複数の町工場から構成される企業が、部材情報をバッチ転送するというシナリオ での検証を行なうことにしました。ここでの検証の狙いは、こうしたシステムでボトルネックになりやすい本店のマスタ更新処理がスムーズに行なえるかどうかです。

pgpool-II を使った検証では、pgpool-II の組み込み同期レプリケーションを使ったクラスタ構成を採用しました。この構成は、更新の遅延がないというメリットがありますが、一方でレプリケーションのオーバーヘッドが気になるところ です。ここでの検証の主な狙いは、検索性能スケールするかどうかを確認することと、同期レプリケーション構成による更新性能への影響です。

Postgres-XC は比較的新しいクラスタソフトウェアですが、更新性能がスケールすることで注目されています。ここでは、検索性能、更新性能を測定し、Postgres-XC の現時点での実用性を検証することが主目的となります。また、
pgpool-II とほぼ同じ構成で検証を行なうことにより、pgpool-II と Postgres-XC の2つクラスタソフトウェアの性能面での性格の違いと、有効な適用領域を探っていきます。

3.2. 更新系・複数台レプリケーション検証

企業のデータベースには時々刻々とデータが蓄えられ、複数の部署や企業に跨り利用されます。PostgreSQL 9.2 でサポートされたカスケードレプリケーションは、バックアップサーバとしての利用はもとより、このような複数サイトでのデータベース利用についても期待できる機能です。今回の検証では、カスケードレプリケーションを使ってデータベースをレプリケーションした場合、データの入口となるマスタサーバでどのような基本特性が性能面から検証しました。

3.2.1. PostgreSQL のカスケードレプリケーションの概要

データベースに格納しているデータは万一に備えて、複製を用意しておくことが望ましく、定期的にバックアップする手段の他に、随時データを複製する方法もあります。

PostgreSQL 9.0 では WAL レベルでの随時データを複製する「ストリーミングレプリケーション」がサポートされました。親サーバから子サーバへレプリケーションする機能です。

PostgreSQL 9.2 では、図 3.1 のように、子サーバを親サーバとして、さらに子サーバから孫サーバへレプリケーションすることが可能になりました。これはカスケードレプリケーションと呼ばれます。

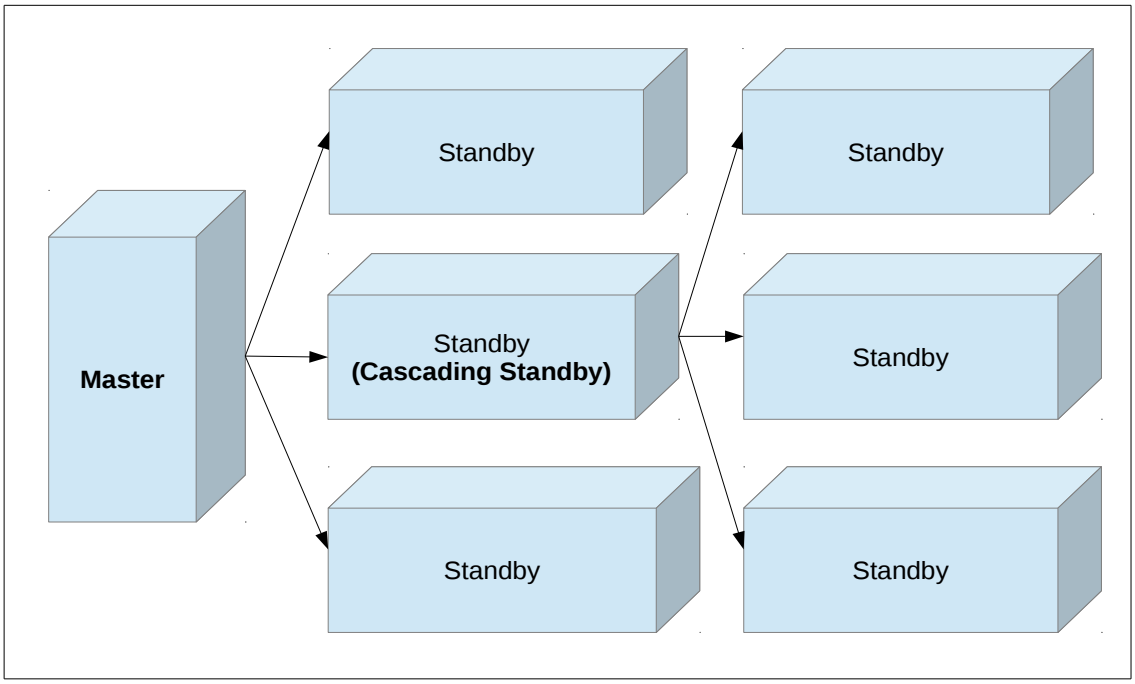

図 3.1: カスケードレプリケーション

レプリケーション元はマスタサーバ、レプリケーション先はスタンバイサーバと呼ばれ、親/子/孫の構成でレプリケーション先とレプリケーション元を兼ねる子に相当するスタンバイサーバは、カスケードスタンバイサーバと呼ばれます。

カスケードレプリケーションでは、仮にマスタサーバが止まっても、カスケードスタンバイサーバとスタンバイサーバとの間で冗長構成を保ったまま運用を継続することができ、より可用性の高いシステムとなります。

また、ホットスタンバイと呼ばれる機能によりスタンバイサーバへの参照アクセスが可能で、負荷分散を目的とした参照用のデータベースとして複数台並べることも可能です。

今回の検証では後者の性能面について特にスケールアウト特性を中心に検証しています。

3.2.2. 検証目的

企業ユーザからこのような声を聞きます。

- 月次で、本店から町工場へ部材情報をバッチ転送している。町工場でも、よりタイムリーに本店の情報を見れると嬉しい。
- 震災・停電時の BCP 対策として、ディザスタリカバリ構成をとりたい。
- 自国から様々な海外拠点へ、DB のデータを送って使用したい。

今回の検証では、カスケードレプリケーションが 2012 年にリリースされたばかりということもあり、様々な利用形態でも参考となる基本的な特性を確認するため、なるべくシンプルなモデルを選択することにしました。
カスケードレプリケーションを企業システムへ適用した場合に、基本特性として性能上の懸念があるのか/ないのか、実際に企業で利用されるような機器を使って確認することが、今回の検証目的です。

（1） 本店-町工場モデル

図 3.2 に示すような、本店/町工場モデルを想定して検証しました。月次や日時バッチ処理により、本店のデータを町工場に反映するような運用から、カスケードレプリケーションに変更した場合に性能面でどのようになるかについてです。この形態では、この他、全社サーバ/部門サーバのようなケースも想定されます。
町工場がどんどん増えていき、レプリケーションするデータベースの数が増えていった場合、本店サーバではデータ更新とレプリケーションを同時に実行する必要があるため大きな負荷がかかります。
このようなモデルでの検証観点としては、レプリケーションにかかる性能、可用性、運用性、移植性、相互運用性が挙げられ、これらの中で特に性能面について、本店の DB 更新性能にどう影響が出てくるかについて検証しました。

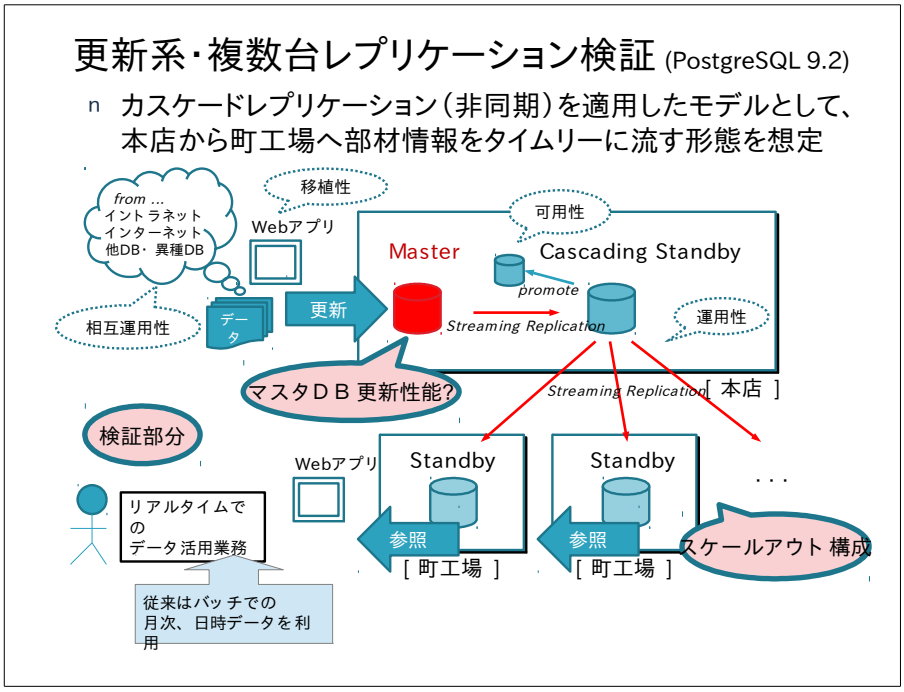

図 3.2: 本店-町工場モデル

（2） 検証ポイント

レプリケーションには同期/非同期があり、今回の検証はバッチ運用をリアルタイム運用に変更した場合の性能検証で、非同期レプリケーションとしました。

今回の検証では、カスケードレプリケーションにするとマスタサーバの更新性能の基本特性としてはどうなるか、図 3.3 に示すように、孫ノードとなるスタンバイサーバのノード数増加に応じて、マスタサーバの更新性能も低下していく傾向なのか、それとも、それほど影響がないのか、どちらになるのか、実機で明らかにすることがポイントです。

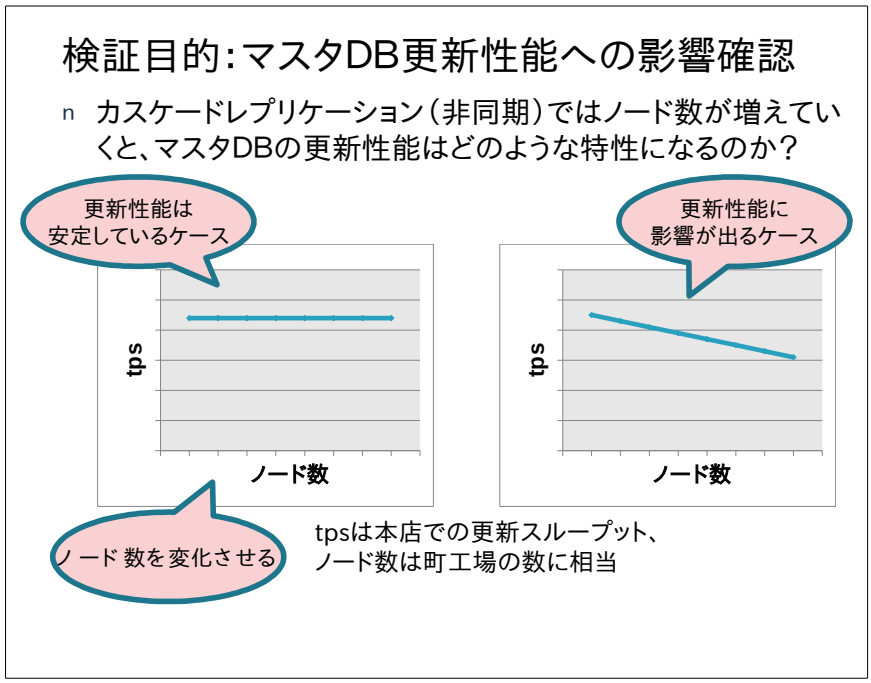

図 3.3: カスケードレプリケーションでのマスタ DB 更新性能はどうなるか?

もちろん、理論上は非同期なため、それほど影響がないことが想定されますが、実際にはハードウェア構成含め様々な要因が関係してくるため、PostgreSQL ソフトウェア自身が持っている基本特性を確認しようというのが今回の検証の主旨です。

3.2.3. 検証構成

検証構成は図 3.4 のとおりです。DBMS は PostgreSQL 9.2.1、OS は RedHat Enterprise Linux 6.2 x86_64 を使用しました。ハードウェアについては検証環境 2（提供：株式会社日立製作所）を使用しました。

本店/町工場モデルで本店部分に相当する、マスタサーバ（Master）とカスケードスタンバイサーバ（CascadingStandby）は各々外部ストレージに接続しました。外部ストレージは RAID5(4D+1P)で 450GB をデータベースクラスタとして使用しました。町工場に相当する、参照のみのスタンバイサーバ（Standby）は外部ストレージではなく、RAID1 構成の内蔵ディスクを使用し、スタンドアローンのサーバとしました。

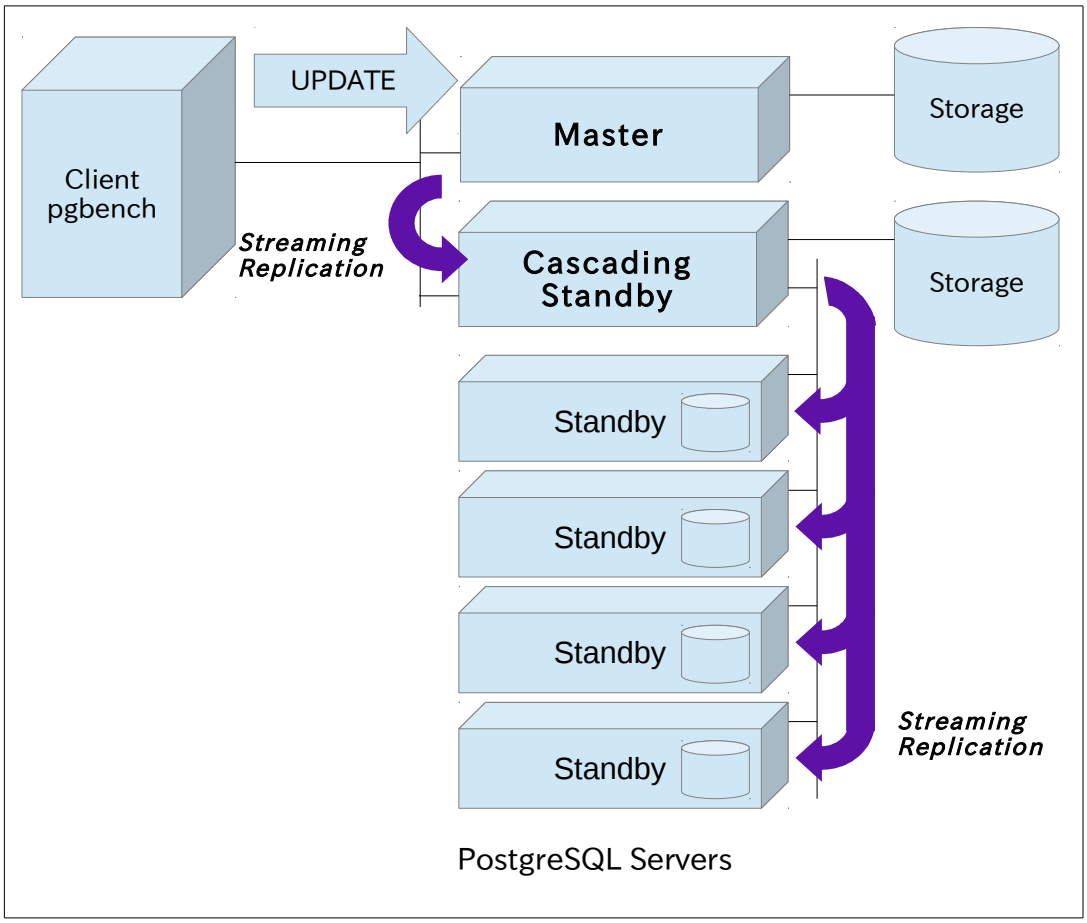

図 3.4: 検証構成

ネットワークは、本店内の LAN でクライアントからマスタへのリクエストもレプリケーションも実行、本点と町工場間に相当するカスケードスタンバイとスタンバイ間は別ネットワークの構成としました。カスケードスタンバイとスタンバイ間は、実際には、本店と町工場が遠いことを考慮して、より低速なネットワークにした方が良いですが、今回の検証では、カスケードスタンバイとスタンバイ間のネットワークも含め、すべてギガビットイーサを使用しました。

3.2.4. 検証方法

○ 環境作成

更新性能を測定するために、性能面で PostgreSQL の設定として配慮した点は次のとおりです。

(1) 参照性能の影響が極力小さくなるよう十分な共有メモリを設定。(shared_buffers)
(2) 基本特性を測定するためにはチェックポイントの影響を排除する必要があるため、測定中にチェックポイントが発生しないよう設定。(checkpoint_segments, checkpoint_timeout)
(3) ストリーミングレプリケーションのみの性能特性となるよう WAL アーカイブは使用しないでプライマリの WAL ファイルセグメントのみで実行できるよう設定。(archive_mode, wal_keep_segments)

● postgresql.conf ファイル

```
listen_addresses = '*'
max_connections = 110
shared_buffers = 8GB                <= (1)十分なバッファを設定
wal_level = hot_standby
checkpoint_segments = 1000          <= (2)マシン時間の関係で
checkpoint_timeout = 1h             <= (2)マシン時間の関係で
#archive_mode = off                 <= (3) archive_mode = off なため
max_wal_senders = 10
wal_keep_segments = 1000            <= (3)archive_mode = off なため。
hot_standby = on
logging_collector = on
logline_prefix = '%t [%p] '
```

検証環境は、PostgreSQL 9.2 のドキュメント[9]を参考に、マスタを作成後、ベースバックアップを作成してカスケードスタンバイ、さらにベースバックアップを作成してスタンバイ、という手順で構築しました。

● ベースバックアップのコマンド例

```
$ pg_basebackup -D $PGDATA -h 192.168.122.180
```

● pg_hba.conf ファイル

```
host  replication  postgres  192.168.122.0/24  trust
```

● recovery.conf ファイル

```
standby_mode = 'on'
primary_conninfo = 'host=192.168.122.180'
```

○ 測定

今回の検証では、更新系の高負荷な測定として、PostgreSQL に標準で付属しているベンチマークツールである pgbench を使用しました。実行シナリオはデフォルトの OLTP 系の TPC-B に基いたシナリオを実行しました。スケールファクタ（scaling factor）は 100（1000 万レコード/約 1.5GB）としました。スケールアウト検証として、スタンバイが 1 台～4 台のケースで、クライアントからマスタに負荷をかけた性能値を測定しました。

● データベースの作成

9 PostgreSQL 9.2 Documentation : 25.2.6. Cascading Replication
http://www.postgresql.org/docs/9.2/static/warm-standby.html#CASCADING-REPLICATION

```
$ createdb testdb
$ pgbench -is 100 testdb
```

- ベンチマークの実行

```
$ pgbench -h [master host] testdb -c 100 -j 10 -T 30
```

3.2.5. 考察

（1） スケールアウト基本特性

マスタ／カスケードスタンバイ／スタンバイ×4台の非同期カスケードレプリケーション構成で、クライアントからpgbench を使ってマスタへ負荷をかけた時の性能測定結果が表 3.1、図 3.5 です。スタンバイサーバのノード数を1、2、3、4と変化させた場合、いずれのノード数でも 23000tps 付近で性能が安定している結果となりました。

この結果から、PostgreSQL の非同期カスケードレプリケーションは、スタンバイサーバ（マスタから見ると孫ノード）を増やしていくようなスケールアウト性が必要な環境でも、マスタの更新性能に大きな影響が出ない基本特性を持っていることが確認できました。

表 3.1: カスケードレプリケーションのスケールアウト基本特性

standby nodes	tps
1	22845.161596
2	23045.513852
3	23038.639685
4	23223.991829

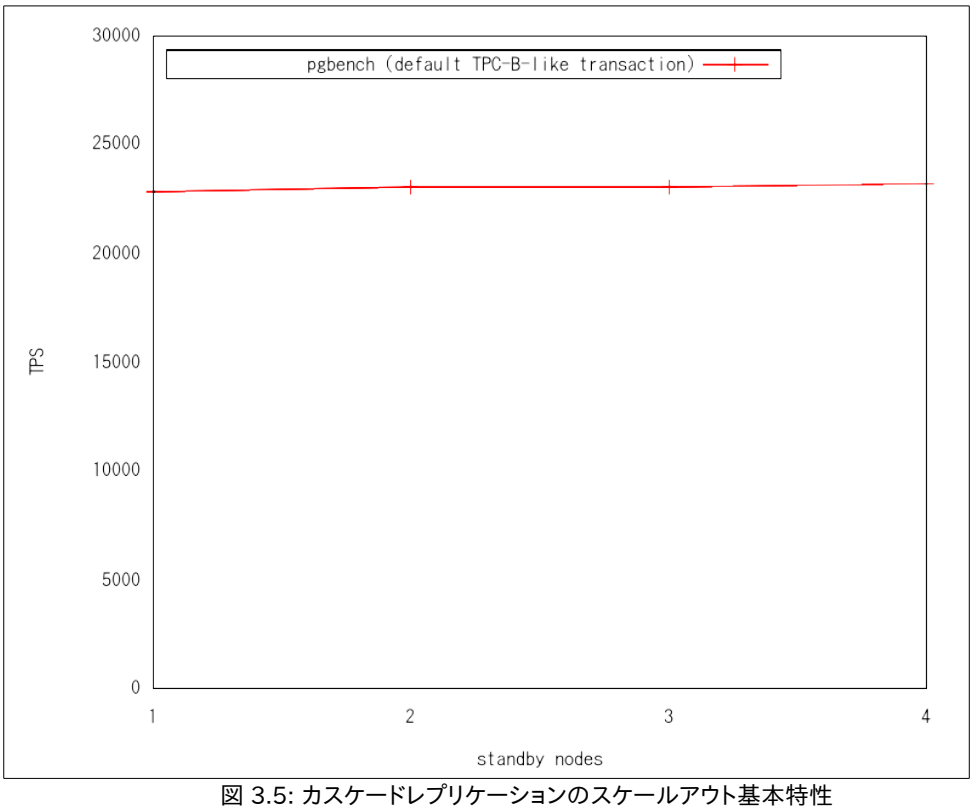

図 3.5: カスケードレプリケーションのスケールアウト基本特性

（2） 非同期レプリケーションのコスト

非同期レプリケーションの“コスト“についても実測しました。表 3.2、図 3.6 はレプリケーションなし（単純な PostgreSQL 単体の性能）とレプリケーションあり（今回の検証での最大構成＝4ノードレプリケーション）での性能比較です。レプリケーションあり/なし、どちらの場合もほぼ 23000tps 付近の性能となっており、非同期レプリケーションのコストは極めて低いことが確認できました。なお、今回の測定では、レプリケーションなしよりもレプリケーションありの性能の方が良い値となっており、非同期レプリケーションのコストは測定誤差レベルの範囲でした。

表 3.2: レプリケーション有無での性能比較

レプリケーションなし	レプリケーションあり
23041.525007	23223.991829

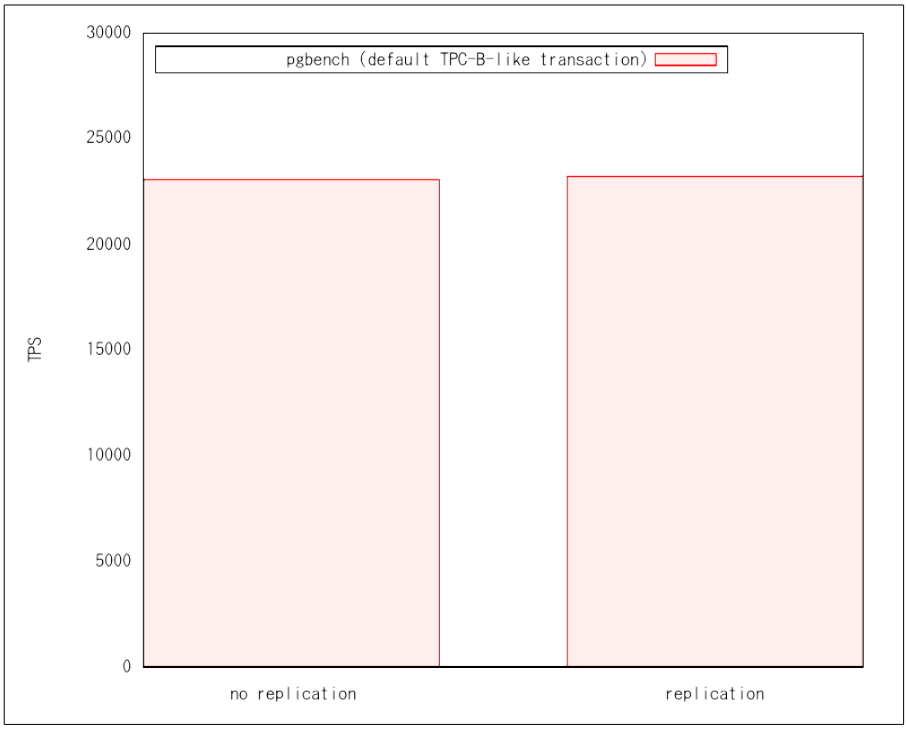

図 3.6: レプリケーション有無での性能比較

（3） 測定時の CPU 使用率

今回の測定では、スタンバイ×1 台～4 台まで変化させて性能を測定しました。この中でスケールアウトとして最大構成となるスタンバイ×4 台構成で測定した際の CPU 使用率を中心に見ていきます。

pgbench を使って負荷をかけている最中の、マスタ／クライアントおよび、カスケードスタンバイ、スタンバイ 1～4 の CPU 使用率を順に見ていきます。いずれのグラフも、計測時間は 30 秒、横軸が経過時間(秒)でメモリの間隔は 3 秒毎となっています。縦軸が CPU 使用率(%)です。

CPU リソースとして最も負荷が高いのはマスタ(図 3.7)で、CPU 使用率はおよそ us=63% sy=25% wa=1% id=11%でした。us+sy+wa を足した CPU 使用率は 89%で、マスタには十分に負荷がかかっている状態です。

なお、クライアント(図 3.8)は us=5% sy=13% wa=0% id=82%で、CPU には余裕があり、負荷不足によるクライアントネックにもなっていないことが確認できます。]

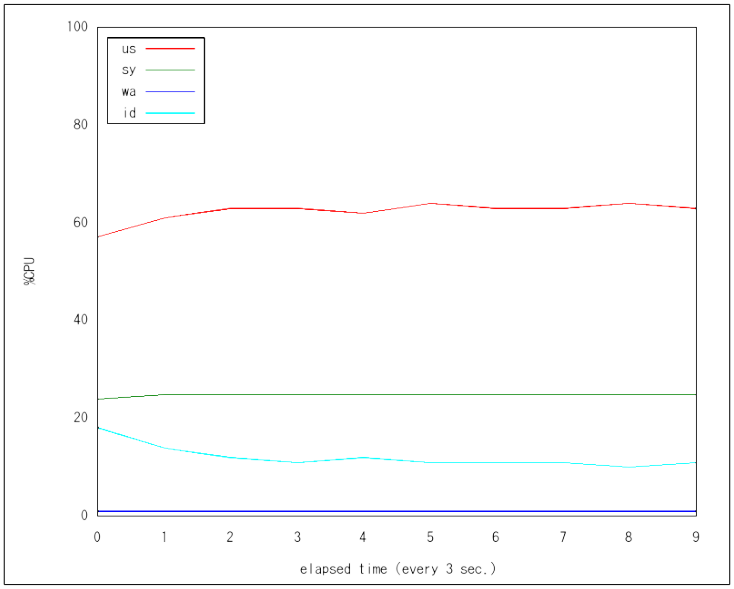

図 3.7: マスタサーバの CPU 使用率

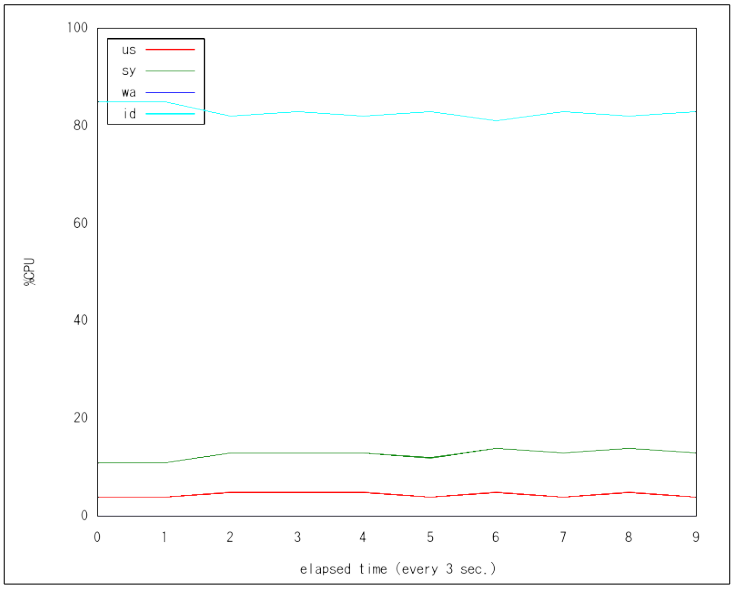

図 3.8: クライアントの CPU 使用率

カスケードスタンバイ(図 3.9)の CPU 使用率は、us=3% sy=2% wa=5% id=90%でした。マスタよりもやや I/O 負荷が高くなる傾向がありますが、全体としては 10%程度の CPU 使用率で、それほど負荷は高くありませんでした。

カスケードスタンバイとスタンバイ間のスケールアウト特性は、カスケードスタンバイをマスタとした複数のスタンバイ構成のイメージとなります。カスケード構成ではない、純粋なマスタとスタンバイ間のスケールアウト特性については、高負荷時においてスタンバイ数が増加するとクライアントから見た性能が少しずつ低下していく特性が 12 ノードを使った検証により知られています[10]。

今回の検証は、この知見をベースに、まだあまり知られていない、カスケード構成でのクライアントから見たマスタ性能に着目したものです。

カスケードスタンバイは、スタンバイのノード数増による共通リソース(ストレージ、ネットワーク、CPU)競合の影響を考慮する必要があります。今回の検証では、スタンバイ×1 台におけるカスケードスタンバイの CPU 使用率(図 3.10)と比較しても、大きな性能差はないため、この影響はあまり大きくないケース(カスケードスタンバイの性能

10 PostgreSQL Conference 2011 K-2【基調講演】PostgreSQL 9.0 ストリーミングレプリケーションの実力
http://www.postgresql.jp/events/pgcon2011/about/view

ネックにはなっていない状態）での検証となっています。

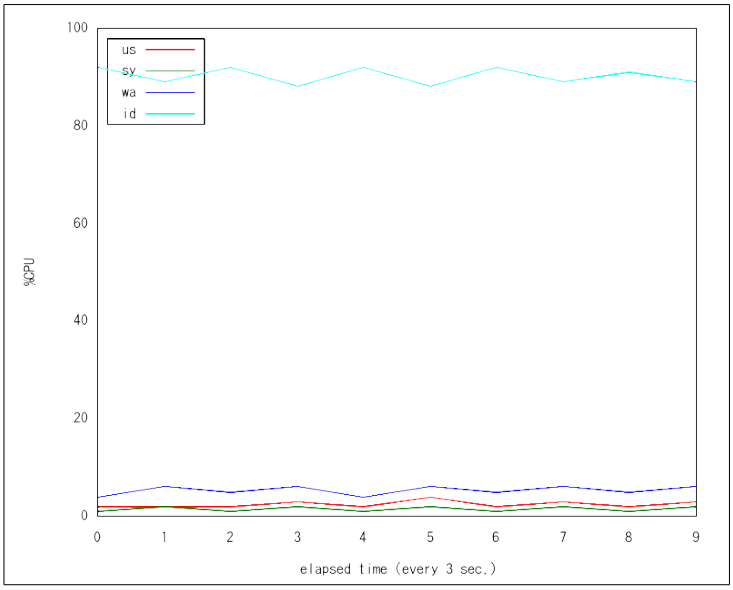

図 3.9: カスケードスタンバイサーバの CPU 使用率

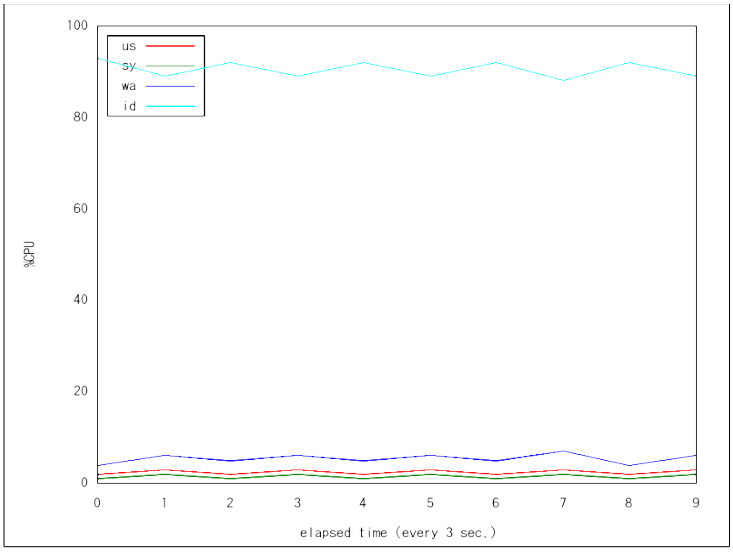

図 3.10: スタンバイ×1 台でのカスケードスタンバイサーバの CPU 使用率

スタンバイ 1～4 の CPU 使用率（図 3.11 ～ 図 3.19）は、いずれも us=2% sy=1% wa=5% id=92%で、ノード毎に大きなバラツキもなく、CPU に余裕のある状態でした。同期レプリケーションでのスケールアウト構成の場合は、レプリケーション先のマシン能力が全体性能に大きく影響しますが、非同期レプリケーションでのスケールアウト構成の場合、特に、レプリケーション先のマシン能力が高くない場合での適用も視野に入れることができます。

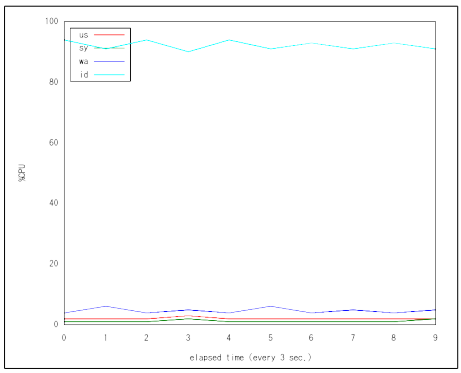

図 3.11: スタンバイサーバ 1 の CPU 使用率

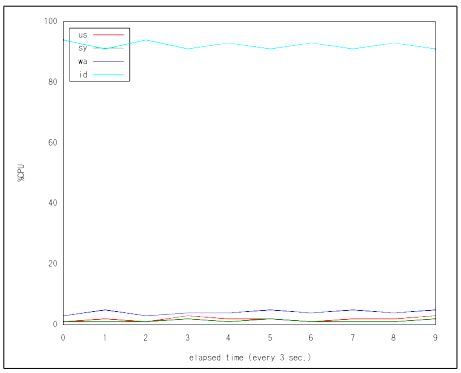

図 3.12: スタンバイサーバ 2 の CPU 使用率

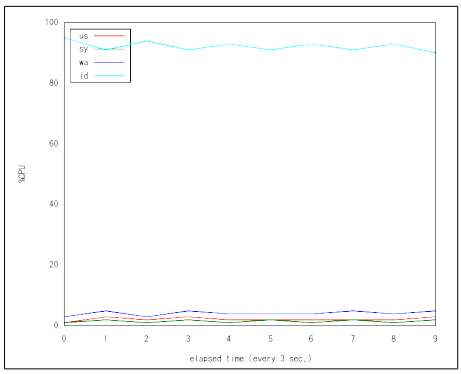

図 3.13: スタンバイサーバ 3 の CPU 使用率

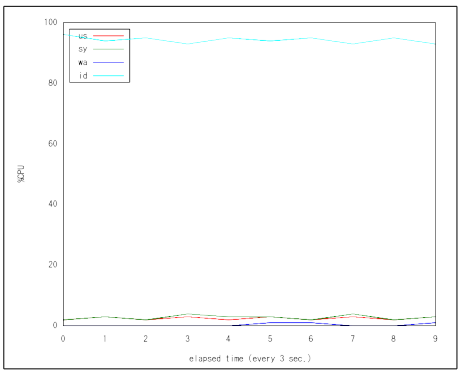

図 3.14: スタンバイサーバ 4 の CPU 使用率

システム全体として見ると、マスタの純粋な更新性能が最も大きな割合を占めており、システム全体から見た非同期カスケードレプリケーションで使用する CPU は小さいと言えます。

なお、今回の検証では、トップ性能を測定することは目的としておらず、基本特性を確認することが目的なため、WAL ファイルを別ディスクにする等の更新系チューニングは施していません。一般に、更新系の非同期レプリケーション構成ではマスタの負荷高騰がシステム全体の性能上のボトルネックとなることもあり、PostgreSQL 9.2 のカスケードレプリケーションは、このようなケースにも配慮された特性を持っていることが、CPU 使用率からもうかがうことができます。

（4） ネットワークやストレージ構成について

今回の検証では、本店/町工場での非同期レプリケーションを想定し、ネットワークもストレージもなるべく別になるよう配慮しました。

しかしながら、今回使用した検証環境は、実際に別々の場所にあるわけではなく、同一センタ内にある 3 ブレード搭載したシャーシが 2 つと、1 つの外部ストレージを 6 つの RAID グループに分割したものを使いました。

データ量が多い場合、ネットワークやストレージの能力や構成がマスタの更新性能に影響を与える可能性があり、本題とはずれますが、これについても実測しましたので付記しておきます。

図 3.15 は、マスタとカスケードスタンバイ間、カスケードスタンバイとスタンバイ間のネットワークが同一で、ストレージも同一にした場合の性能測定結果です。この測定では、スタンバイのノード数が多くなるほど、マスタの更新性能が低下する傾向となりました。非同期レプリケーションだからといって、必ずマスタの更新性能への影響が低いかというと、必ずしもそうではなく、スケールアウト上のボトルネックが出てきてしまう可能性があるケースもあるということを示しています。

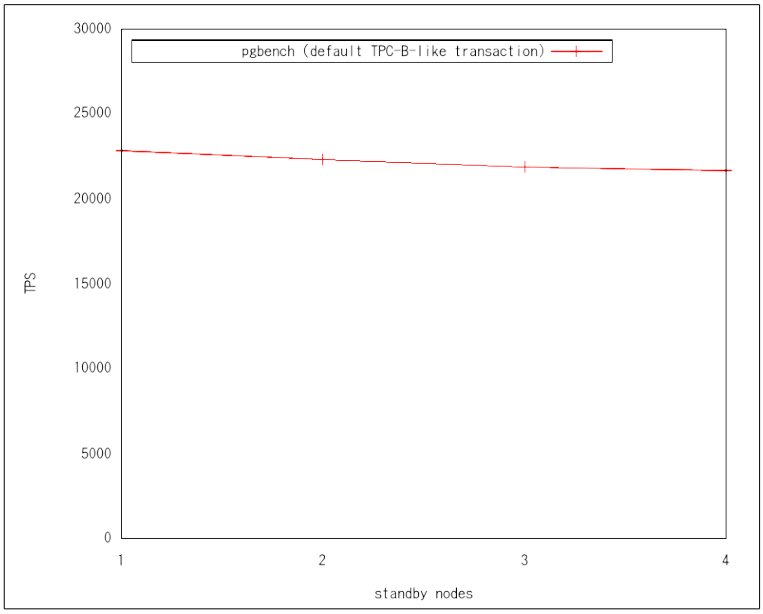

図 3.15: スケールアウトしないケース

（5） その他：データサイズが大きい場合

試行錯誤中、スケールファクタ=10000 (約 150GB), -T 1800 (30 分)で実測した際の、マスタの CPU 使用率が図 3.16、カスケードスタンバイの CPU 使用率が図 3.17 です。横軸が 1 分間隔の経過時間、縦軸が CPU 使用率です。

マスタの CPU 使用率が wa=80%程度で処理のほとんどが I/O、カスケードスタンバイは CPU 使用率が 0%に近い状態でほとんど動いていません。データロードも1時間程度必要で、レプリケーションの特性検証にはならないため、大きいデータサイズでの検証は中止しました。

データサイズが大きい場合、レプリケーションのコストよりも I/O のコストが大きな割合を占めることを示唆する性能情報です。

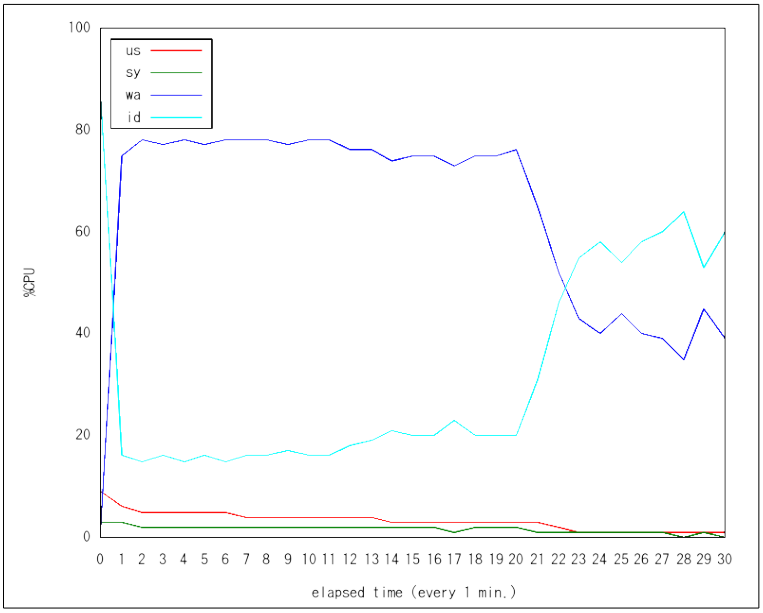

図 3.16: スケールファクタ=10000 におけるマスタサーバの CPU 使用率

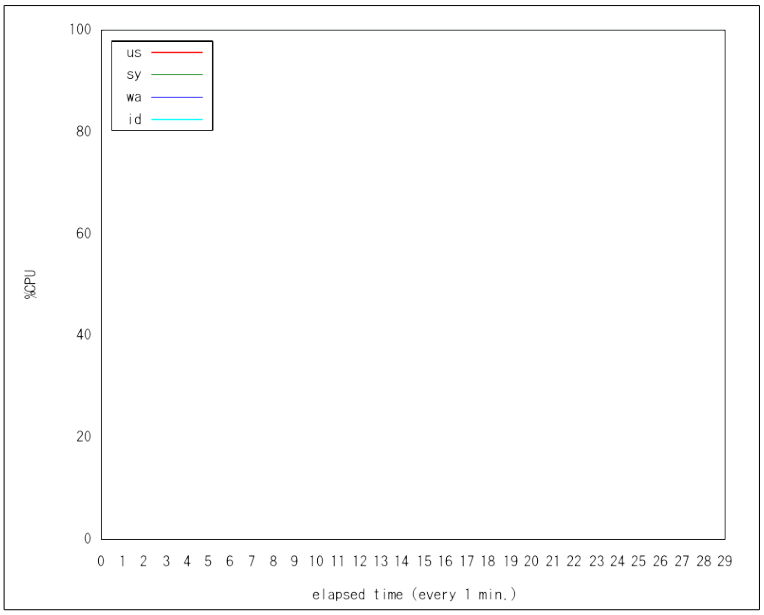

図 3.17: スケールファクタ=10000 におけるカスケードスタンバイサーバの CPU 使用率

3.3. pgpool-II によるスケールアウト検証

3.3.1. pgpool-II の概要

（1） pgpool-II とは

pgpool-II は、OSS として開発、公開されている PostgreSQL 専用のクラスタリング用ミドルウェアです。

注) pgpool-II オフィシャルサイト: http://www.pgpool.net /

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

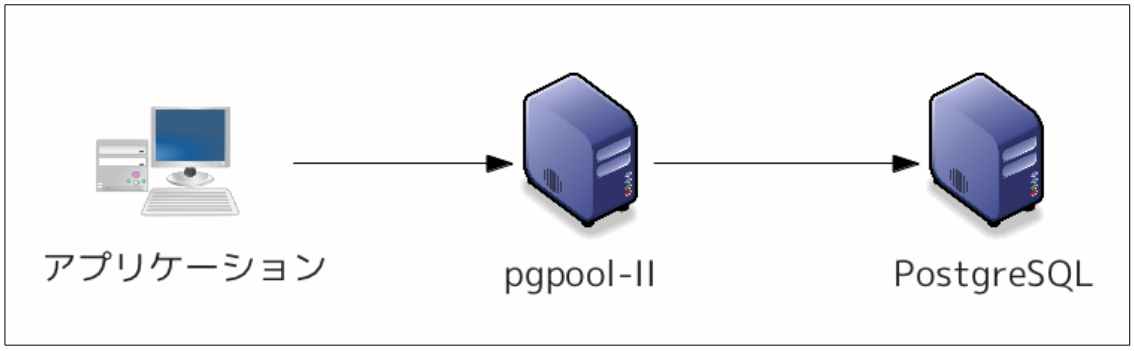

図 3.18: pgpool-II と PostgreSQL

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

（2） pgpool-II の機能

pgpool-II の機能は大きく分けて以下の4つがあります。

○ コネクションプーリング

PostgreSQL への接続を保存しておき、再利用することによって PostgreSQL への接続オーバヘッドを低減し、システム全体のスループットを向上することができます。

[注: 本検証では pgbench が持続的に PostgreSQL に接続するため、この機能の恩恵は受けません。]

○ レプリケーション

pgpool-II は複数の PostgreSQL サーバを管理することができます。レプリ ケーション機能を使用することにより、物理的に 2 台以上の DB サーバにリアルタイムでデータを保存することができ、万が一どれかの DB サーバに障害が発生しても運用を継続することができます。PostgreSQL に障害が発生した際には、自動的に PostgreSQL

サーバが切り離されます（フェイルオーバ）。

○ 負荷分散

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

○ パラレルクエリ

複数のサーバにデータを分割して受け持たせ、それぞれのサーバに同時に検索問い合わせを投げて、問い合わせの処理時間を短縮するパラレルクエリが利用できます。特に大規模なデータベースに対して検索を実行するときに威力を発揮します。

○ レプリケーション機能

本検証で主に利用する機能はレプリケーションと負荷分散です。レプリケーションについて詳しく見ていきます。

pgpool-II では、２つのレプリケーションモードがあります(pgpool-II を起動する際にどれかを選択します)。この他、Slony-I を使うレプリケーションモードもありますが、現在ではあまり使われていないので、説明は省略します。

1) ネィティブレプリケーションモード

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

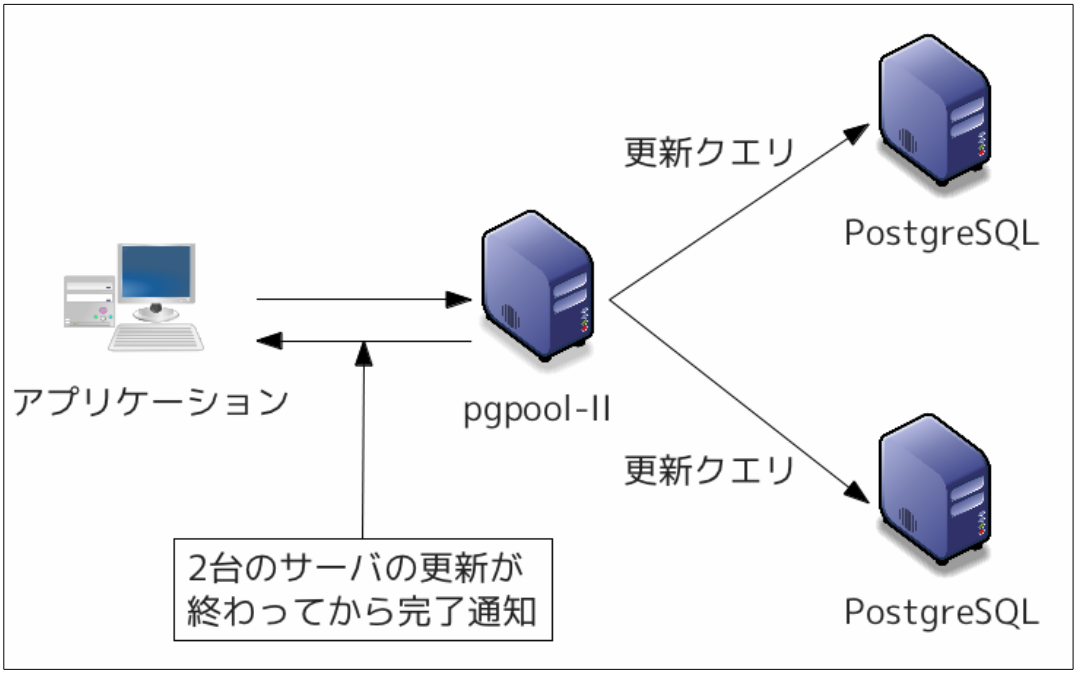

図 3.19: pgpool-II のレプリケーションモード

2) ストリーミングレプリケーションモード

pgpool-II 自体はレプリケーションを行なわず、PostgreSQL にストリーミングレプリケーションを設定することによって、レプリケーションを実現する方法です。
pgpool-II は、コネクションプーリングと負荷分散、それにフェイルオーバのみを行ないます。

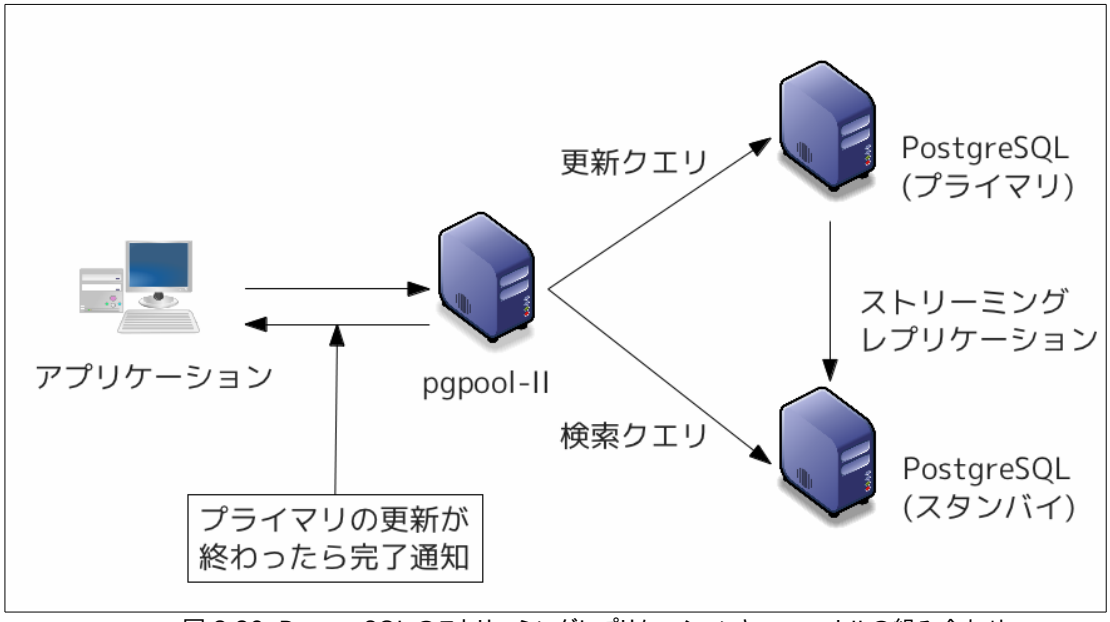

図 3.20: PostgreSQL のストリーミングレプリケーションと pgpool-II の組み合わせ

（3） ネィティブレプリケーションモード

今回の検証で利用したのはネィティブレプリケーションモードです。この方式では、以下のようにレプリケーションが行なわれます。

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

このように、クライアントに更新クエリの応答が返ってきた後はすべてのサーバで更新が完了していることが保証されるため、これは同期レプリケーションとなります。

同期レプリケーションでは、クライアントは負荷分散によって異なるサーバに振り分けられた SELECT がタイミングによって異なる結果（古い結果）を受け取る可能性がないため、PostgreSQL に直接接続していることを前提にしたプログラムの修正の必要がありません。

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

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

図に、pgpool-II の同期レプリケーションにおける更新性能の理論値を示します。

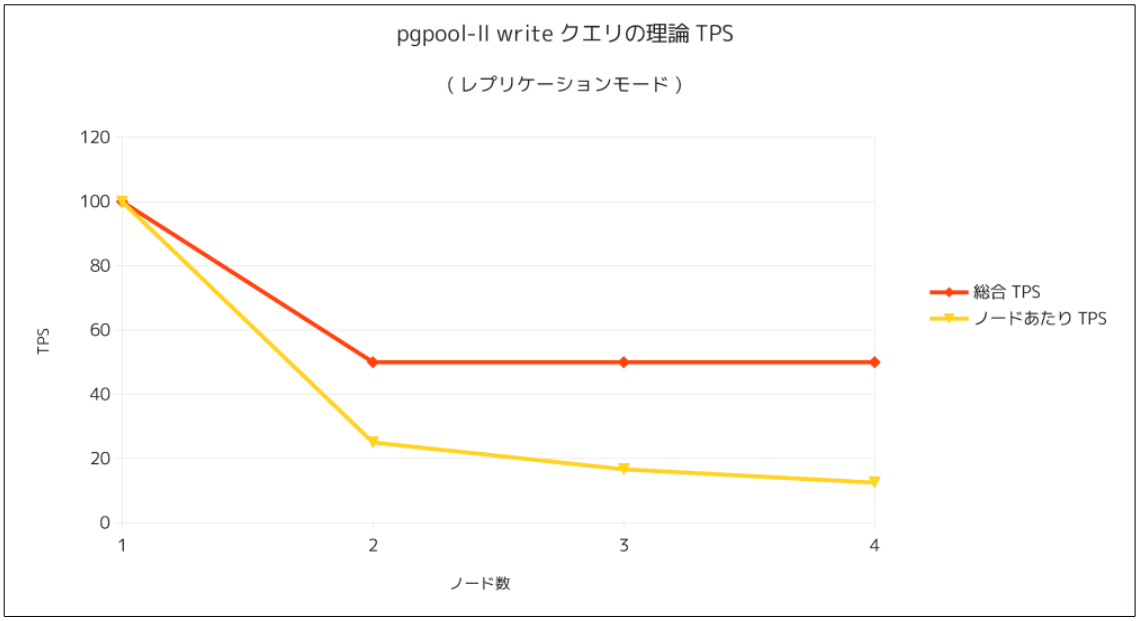

図 3.21: pgpool-II の同期レプリケーションにおける更新性能の理論値

一方、ストリーミングレプリケーションモードでは、マスタサーバのデータよりもスタンバイサーバのデータが遅れていることを常に意識するように、プログラムや運用で工夫をする必要があります(たとえば、画面から更新を受け付けた後「更新が反映されるまで少し時間がかかる場合があります」のようなメッセージを出力するなど)。

反面、更新性能の低下は、ネィティブレプリケーションモードほど激しくありません。

3.3.2. 検証目的

前項 pgpool-II の概要で説明したとおり、pgpool-II を提供したシステムでは、以下の特性が現れることが予想されます。

・更新系： 投入ノード数を増やすほど、1 ノード当たりの処理性能が低下する
・参照系： 投入ノード数に比例して、ノード全体での処理性能が向上する

この検証は、PostgreSQL および pgpool-II をエンタープライズなシステムに適用した場合も同様の傾向がみられるか否かを確認することを目的とします。

実際にはエンタープライズなシステムでよく利用されるスペックのサーバ上(検証に使用したサーバのスペック詳細は「検証構成」を参照)で、PostgreSQL および pgpool-II を適用したシステムに対し、pgbench を用いて負荷を掛け、性能の変化を測定します。

3.3.3. 検証構成

下図のように、ハードウェアおよびソフトウェアを配置ししました。

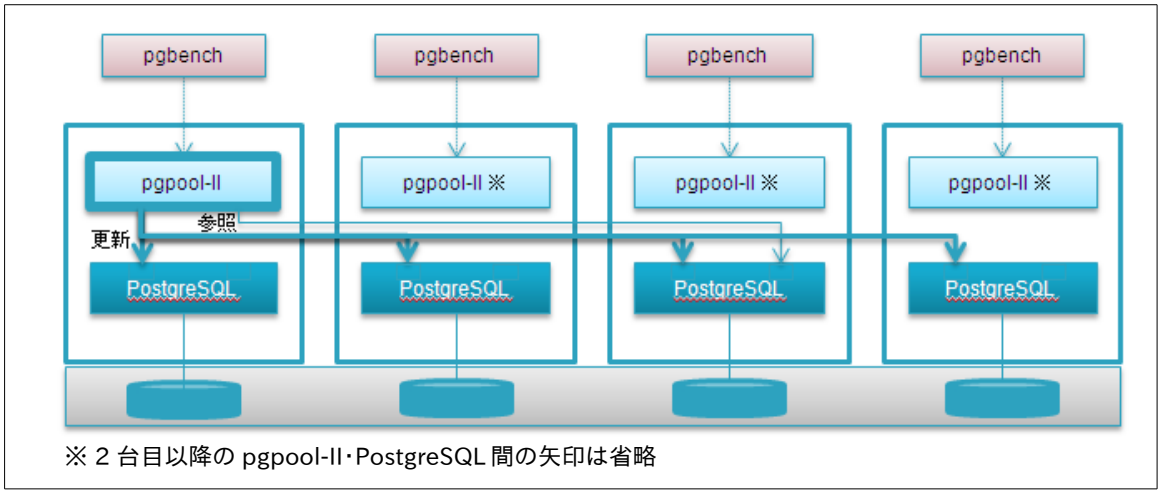

図 3.22: ハードウェアおよびソフトウェアの配置

検証作業時には、実際に動作させるノード（動作させるクライアント（pgbench）とサーバ（pgpool-II & PostgreSQL）のセット）を1から4まで変化させ、動作ノード数の増減による性能を測定し、特性を分析しました。

なお、検証試験に際し、pgpool-II にクラスタ対象とする PostgreSQL を認識させるため、設定ファイル（pgpool.conf）に作業対象とする PostgreSQL の情報を記述しています。

表 3.3: 設定ファイル記述例

設定ファイル名	記述例

pgpool.conf	# - Backend Connection Settings -
	
	# 1 台目の PostgreSQL に関する情報
	backend_hostname0 = '192.168.1.30'
	# Host name or IP address to connect to for backend 0
	backend_port0 = 5432
	# Port number for backend 0
	backend_weight0 = 1
	# Weight for backend 0 (only in load balancing mode)
	backend_data_directory0 = '/vol1/pgsql/data'
	# Data directory for backend 0
	backend_flag0 = 'ALLOW_TO_FAILOVER'
	# Controls various backend behavior
	# ALLOW_TO_FAILOVER or DISALLOW_TO_FAILOVER
	
	# 2 台目の PostgreSQL に関する情報
	# 検証対象の PostgreSQL が 2 台未満の場合はこの部分をコメント化する
	backend_hostname1 = '192.168.1.31'
	backend_port1 = 5432
	backend_weight1 = 1
	
	backend_data_directory1 = '/vol1/pgsql/data'
	backend_flag1 = 'ALLOW_TO_FAILOVER'
	
	# 3 台目の PostgreSQL に関する情報
	# 検証対象の PostgreSQL が 3 台未満の場合はこの部分をコメント化する
	backend_hostname2 = '192.168.1.32'
	backend_port2 = 5432
	backend_weight2 = 1
	
	backend_data_directory2 = '/vol1/pgsql/data'
	backend_flag2 = 'ALLOW_TO_FAILOVER'
	
	# 4 台目の PostgreSQL に関する情報
	# 検証対象の PostgreSQL が 4 台未満の場合はこの部分をコメント化する
	
	backend_hostname3 = '192.168.1.40'
	backend_port3 = 5432
	backend_weight3 = 1
	
	backend_data_directory3 = '/vol1/pgsql/data'
	backend_flag3 = 'ALLOW_TO_FAILOVER'

更新系・参照系それぞれの検証に使用したハードウェアおよびソフトウェアの詳細を以下に記載します。

表 3.4: 更新系測定時の検証環境一覧

検証環境項目		内容
PostgreSQL サーバ	ハードウェア	CPU :Intel® Xeon® E5-2470(2.30GHz 8core) 2 Processor / 合計 16Core Memory:32GB 内臓 HDD:300GB × 2 ネットワークカード:Gigabit Ethernet
	OS	Redhat Enterprise Linux 6.2
	PostgreSQL サーバ	PostgreSQL 9.2.1
	クラスタリング用ミドルウェア	pgpool-II3.2.1
クライアントサーバ	ハードウェア	CPU :Intel® Xeon® E5-2470(2.30GHz 8core) 2 Processor / 合計 16Core Memory:32GB 内臓 HDD:300GB × 2 ネットワークカード:Gigabit Ethernet
	OS	Redhat Enterprise Linux 6.2
	PostgreSQL	PostgreSQL 9.2.1
	テストツール	gbench
ストレージ		HDD:900GB (600GB SAS × 20)

表 3.5: 参照系測定時の検証環境一覧

検証環境項目		内容
PostgreSQL サーバ	ハードウェア	CPU :Intel® Xeon® E5-2670(3.06GHz 6core) 2 Processor / 合計 12 Core Memory:64GB 内臓 HDD:600GB × 2 ネットワークカード:Gigabit Ethernet
	OS	Redhat Enterprise Linux 6.2
	PostgreSQL サーバ	PostgreSQL 9.2.1
	クラスタリング用ミドルウェア	pgpool-II3.2.1
クライアントサーバ	ハードウェア	CPU:Intel® Xeon® X5576 (3.06GHz 6core) 2 Processor / 合計 12Core メモリ:16GB 内臓 HDD: 600GB × 4 ネットワークカード:Gigabit Ethernet
	OS	Redhat Enterprise Linux 6.2
	PostgreSQL	PostgreSQL 9.2.1
	テストツール	gbench
ストレージ		HDD:3TB (450GB SAS × 15) VRAID5 (500GB)× 6(うち 4 を検証で使用)

3.3.4. 検証方法

(1) 更新系

○ 環境作成

スケールファクタ 1000 で初期化しました（データベースサイズは 15GB）。

```
$ pgbench -i -s 1000
```

また、シェアードメモリを 16GB 用意しました（作成したデータをメモリに載せきるため）。

○ 測定

検証対象とするノード（動作させるクライアント（pgbench）とサーバ（pgpool-II & PostgreSQL)のセット）を1から4まで変化させ、それぞれの環境下で以下のコマンドを実行し、処理できる TPS の値を測定しました。

```
$ pgbench -c 100 -t 300 -j 20
```

なお、試験ケースとして pgbench 組み込みの試験ケースを利用しました（組み込み試験ケースは 4:1 の割合で更新系 SQL と参照系 SQL を発行。このため、本検証は厳密な意味では参照系のみの検証ではありません）。

また、検証時 IO ネック発生による性能劣化を避けるため（後述）、それぞれのノードを別 RAID グループのディスク上に配置しました。

上記コマンド（実施時間(-T): 300 秒、同時実行クライアント数(-c): 100、ワーカスレッド数(-j): 20）により得た実行結果は以下のとおりです。

○ 結果

後述の図表に記載したデータが得られ、これにより以下の各傾向が確認できました。
 2 ノード以上では、ノード数が増えるほど合計の tps が減少する
同期レプリケーションを行なうため、更新処理を行なわせた場合性能が出にくい
各ノードのデータの書き込み先を物理的に分散させることで、性能の劣化度を多少改善できる

注: 後述のグラフの縦軸は tps です。横軸はノード数です。

表 3.6: 結果 tps

ノード数	コネクション数	DB ノード数	tps中央値
PostgreSQL 9.1	100	1	759
1 ノード	100	1	759
2 ノード	200	2	722
3 ノード	300	3	624
4 ノード	400	4	560

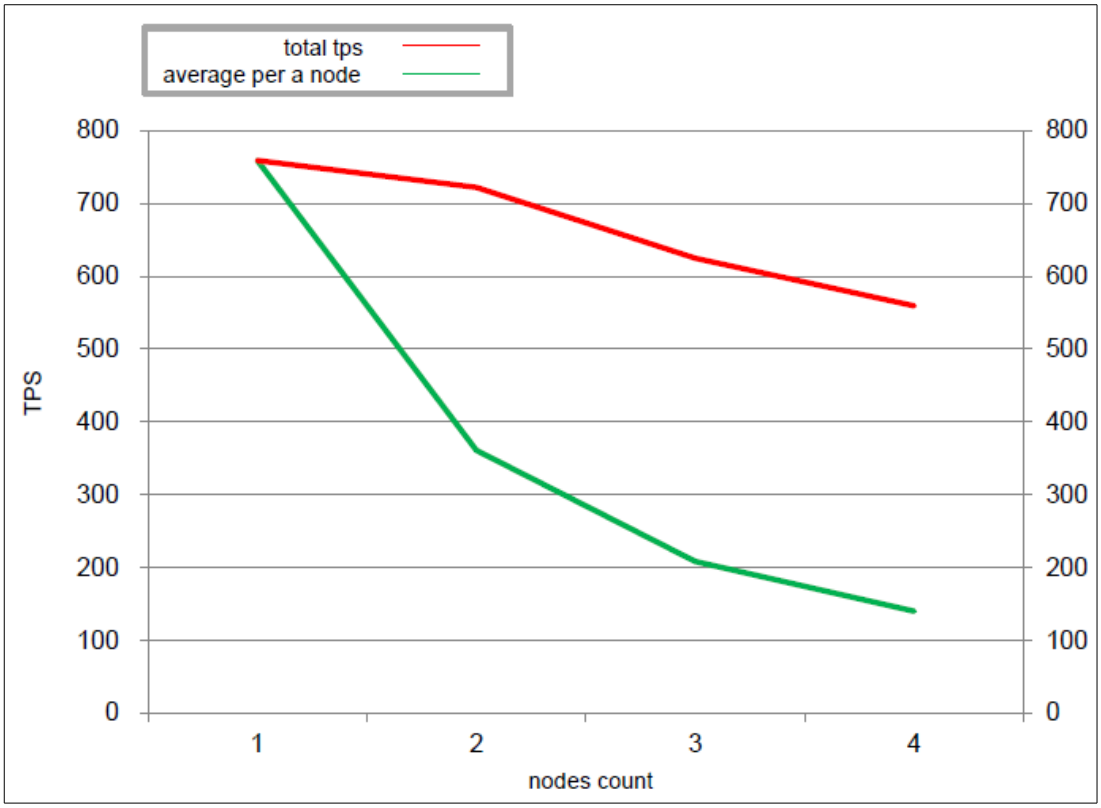

図 3.23: 更新系の結果

（1） 参照系

スケールファクタ 1000 で初期化しました（データベースサイズは 15GB）。

```
$ pgbench -i -s 1000
```

また、シェアードメモリを 16GB 用意しました（作成したデータをメモリに載せきるため）。

○ 測定

検証対象とするノード（動作させるクライアント（pgbench）とサーバ（pgpool-II & PostgreSQL)のセット）を1から4まで変化させ、それぞれの環境下で以下のコマンドを実行し、処理できる TPS の値を測定しました。

```
$ pgbench -c 100 -t 300 -j 20 -f ro_script -n
```

なお、試験ケースとして以下のカスタムスクリプトを作成し、検証を行ないました。

スクリプト名	内容
ro_script	￥set nbranches :scale ￥set ntellers 10 * :scale ￥set naccounts 100000 * :scale ￥set range 2000 ￥set aidmax :naccounts - :range ￥setrandom aid 1 :aidmax ￥setrandom bid 1 :nbranches ￥setrandom tid 1 :ntellers ￥setrandom delta -5000 5000 SELECT count(abalance) FROM pgbench_accounts WHERE aid BETWEEN :aid and :aid + :range;

上記コマンド（実施時間(-T): 300 秒、同時実行クライアント数(-c): 100、ワーカスレッド数(-j): 20、実行シナリオ名(-f): カスタムシナリオファイル名を指定、試験前バキュームの禁止(-n)）により得た実行結果は以下のとおりです。

○ 結果

後述の図表に記載したデータが得られ、これによりノード数が増えるほど、処理できる合計の tps が増える事が確認できました。

注: 後述のグラフの縦軸は tps です。横軸はノード数です。

表 3.7: 結果 tps

ノード数	コネクション数	DB ノード数	tps中央値
PostgreSQL 9.1	100	1	28,564
1 ノード	100	1	23,554
2 ノード	200	2	47,152
3 ノード	300	3	71,166
4 ノード	400	4	94,876

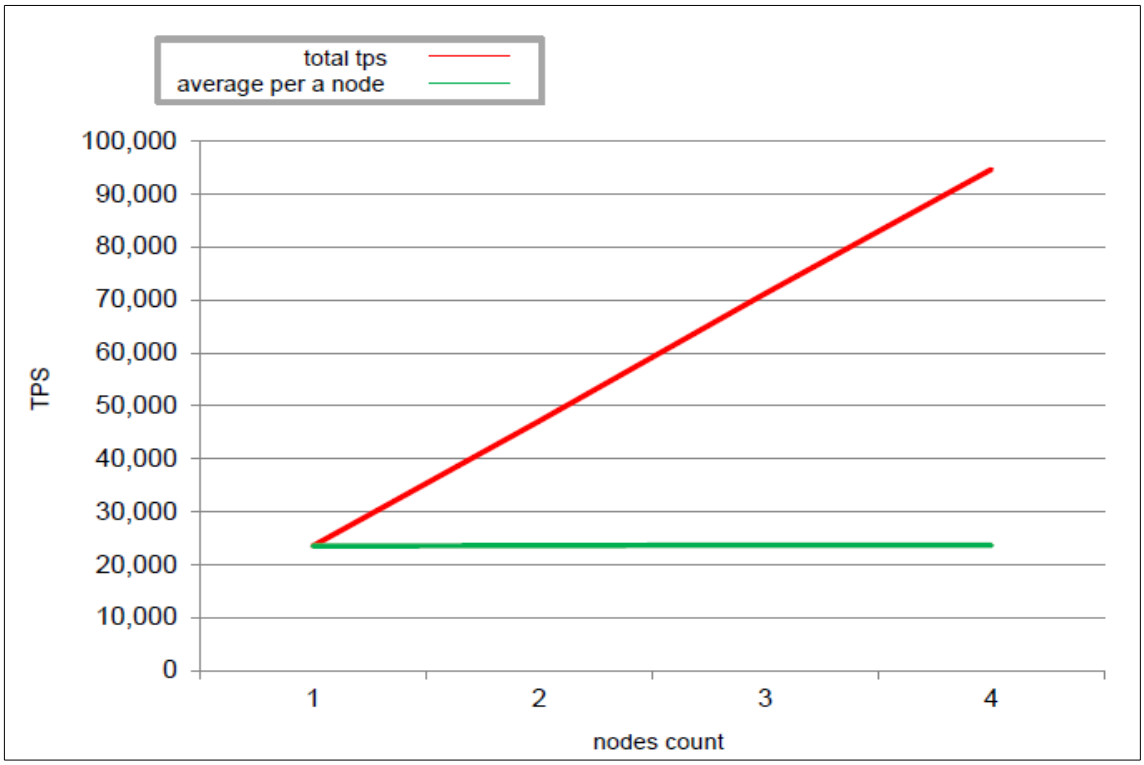

図 3.24:参照系の結果

3.3.5. 考察

検証結果（更新系: 表 3.5、参照系: 表 3.6 を参照）から、多量のメモリやストレージを搭載した所謂エンタープライズ用途のマシン上においても pgpool-II を適用したシステムでは、以下の特性が現れることが確認できました。

・更新系： 投入ノード数を増やすほど、1 ノード当たりの処理性能が低下する
・参照系： 投入ノード数に比例して、ノード全体での処理性能が向上する

このことから pgpool-II を適用したシステムは、Web マーケットシステムのような、不特定多数なユーザ（買い物客）があらかじめ DB 等に格納されたデータ（商品リスト等）を閲覧していく、参照が主目的のシステムをスケールさせる手段としては大変有用であると考えられます。

反面オンライン送金システムのような、ユーザ（送金者）のデータ（預金残高）の更新ありきのシステムをスケールさせる手段には不向きと考えられます。

ただし、不得意である更新系システムへの適用した場合も、I/O ネックが k 直力発生しないチューニングを行なうことで、性能の改善や向上を図ることが可能です。

以下のデータは、更新系システムを構成する各 PostgreSQL のデータ格納先を I/O が競合する場所に配置した場合と、I/O の競合が発生しない場所に配置した場合の検証例です

I/O の競合が発生しないチューニングを行なうことで、性能の向上を見込めることが、検証結果から読み取れます。

I/O ネックの発生を抑えるチューニング等を施し、サーバ側 CPU の稼働率を向上させることにより、更新系・参照系システム共に、性能向上が図れます。

表 3.8:結果 tps（I/O 競合が発生しないように配置した場合）

ノード数	コネクション数	DB ノード数	tps中央値
1 ノード	100	1	759
2 ノード	200	2	722
3 ノード	300	3	624
4 ノード	400	4	560

※ 検証の際、データの格納先として別々の RAID 上の領域を利用

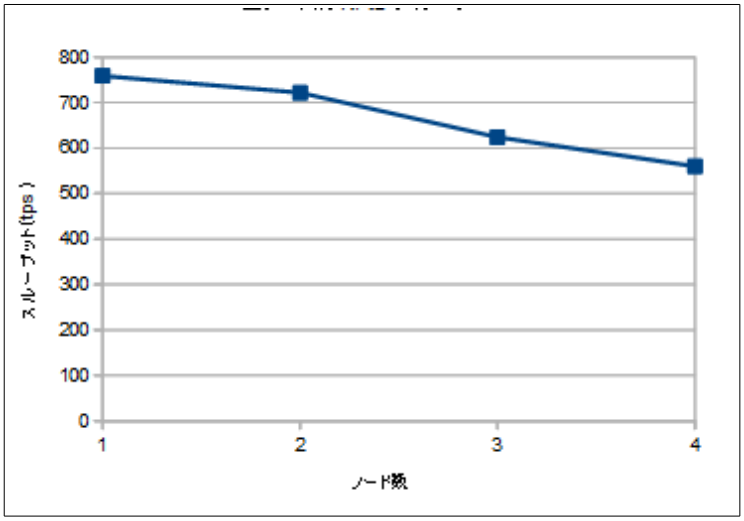

図 3.25: 結果（I/O 競合が発生しないように配置した場合）

表 3.9: 結果 tps（I/O 競合が発生しないように配置した場合）

ノード数	コネクション数	DB ノード数	tps中央値
1 ノード	100	1	759
2 ノード	200	2	288
3 ノード	300	3	189
4 ノード	400	4	169

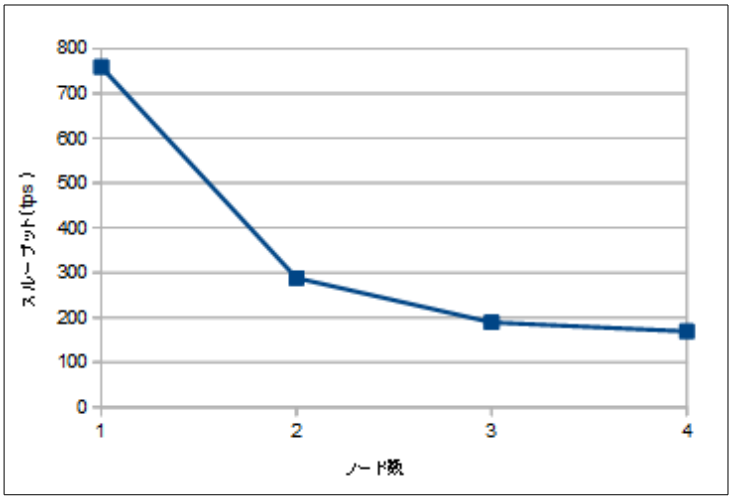

図 3.26 結果（I/O 競合が発生しないように配置した場合）

※ 検証の際、DB ノード 1～3 のデータの格納先として同一の RAID 上の領域を利用

3.4. Postgres-XC によるスケールアウト検証

3.4.1. Postgres-XC の概要

（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[11]。

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 コミュニティの Web サイト(英文)からすべての情報(ソースコード、マニュアル、評価用セット、講演のマテリアル等)が入手できます。また概要と動作原理については鈴木[12]、坂田[13]の解説があります。
この節では、上で紹介した解説をもとに Postgres-XC によるスケーラビリティの実現手法を説明します。

○ 基本的な構成

Postgres-XC は図 3.27 に示されるような、3 種類のコンポーネント（コーディネータ、データノード、GTM）から構成されます。

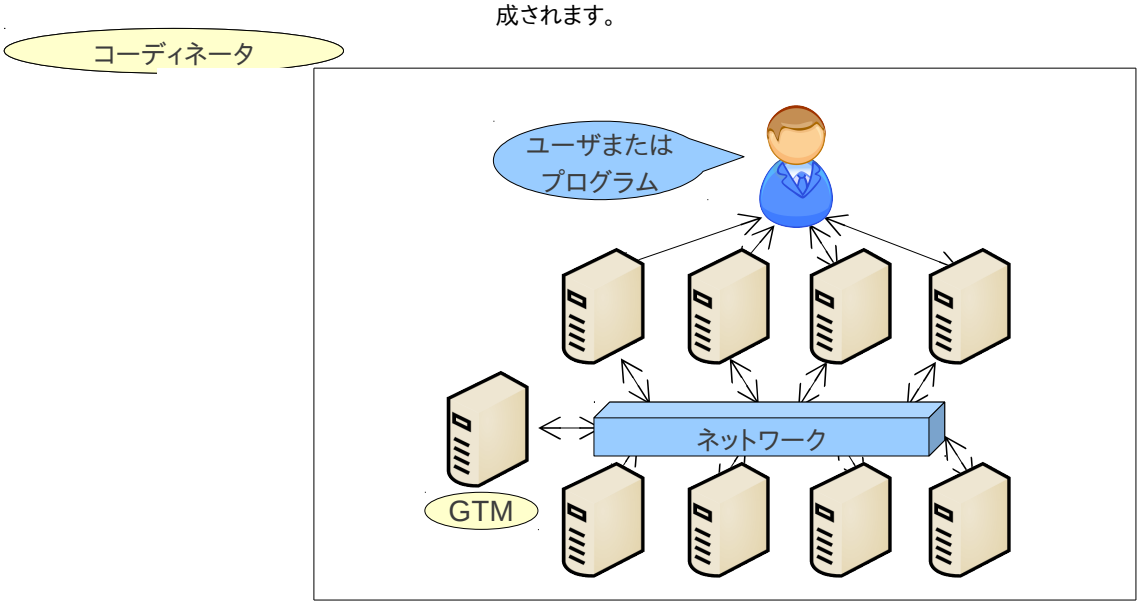

図 3.27: Postgres-XC の構成図

- コーディネータ

11http://sourceforge.net/projects/postgres-xc/
12http://thinkit.co.jp/story/2010/10/26/1828
13http://www.ntt.co.jp/journal/1205/files/jn201205038.pdf

ユーザ(アプリケーションプログラム;AP)からの参照や更新などのリクエストを受け付けて、次に示すデータノードにそれを引き渡します。また、データノードから戻ってきたリクエストの結果を取りまとめて AP に返却します。

- データノード
  データベースを構成するデータを格納すると同時に、コーディネータからのリクエストに基づいて、自ノード内のデータベースに対する処理を行ないます。

- GTM（global transation manager）
  Postgres-XC を構成する複数のノードにまたがって、クラスタ全体として一貫した(ACID 性のある)処理ができるよう、処理を制御します。

○ 推奨構成

Postgres-XC によるデータベースサーバを実現するためには、これらのコンポーネントを実際のサーバ上に配置する必要があります。コミュニティでよくみられる配置は、1 台のサーバにコーディネータとデータノードをそれぞれ 1 つずつ配置するものです。これは、コーディネータでの処理が主に CPU 資源を必要とするのに対して、データノードでの処理が I/O 資源を必要とするという特徴があり、コーディネータとデータノードを 1 つのサーバに同居させることで、サーバに備わっている CPU と I/O という資源を効率よく利用することが期待できるためです。以下、本書ではこれを「**推奨構成**」とし、この構成に沿って説明します(図 3.28 参照)。

推奨構成のもう一つの特徴は、GTM proxy の採用です。GTM は Postgres-XC クラスタ全体のトランザクションを制御するコンポーネントですが、全てのサーバから参照されるためかなりの高負荷になります。そこで、GTM の負荷を各サーバに分散させるために、各サーバ上で GTM proxy が動作するように設定します。設定例は、インストールマニュアル[14]を参照してください。

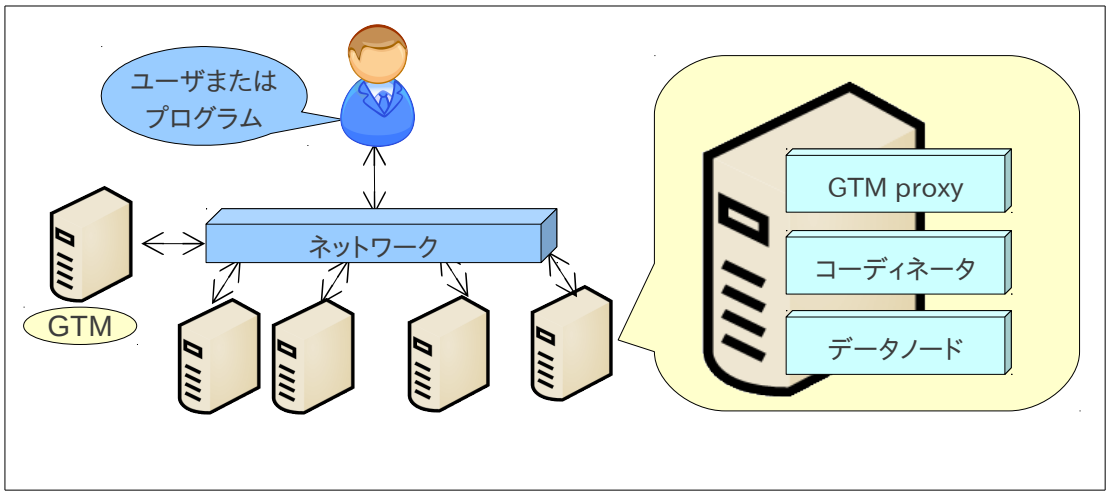

図 3.28: 推奨構成

（2） 高速化の基本的な考え方

この章では、Postgres-XC における、処理の高速化に必要となる考え方を説明します。カギとなるのはシャーディングとレプリケーションというデータ配置の方法と、絞り込みとプッシュダウンというクエリ処理の方法です。

○ シャーディングによる高速化

近年、ハードウェア価格の低廉化を受けて、データを分散して複数のサーバに格納し、検索・管理する、シャーディ

14「Postgres-XC Install Manual Version 0.9.7」http://jaist.dl.sourceforge.net/project/postgres-xc/misc/PG-XC_InstallManual_v0_9_7.pdf

ング(database shard)が注目されています[15]。その基本となる考え方は、論理的な 1 つの表に格納されている複数の行(レコード)を物理的に別個のサーバに格納することで、その表に対する検索などの操作に利用できるリソースの量を増やすことです(図 3.29)。またサーバごとに格納されている行の属性値によって、検索を絞り込むことでリソース利用の効率化を図ります。

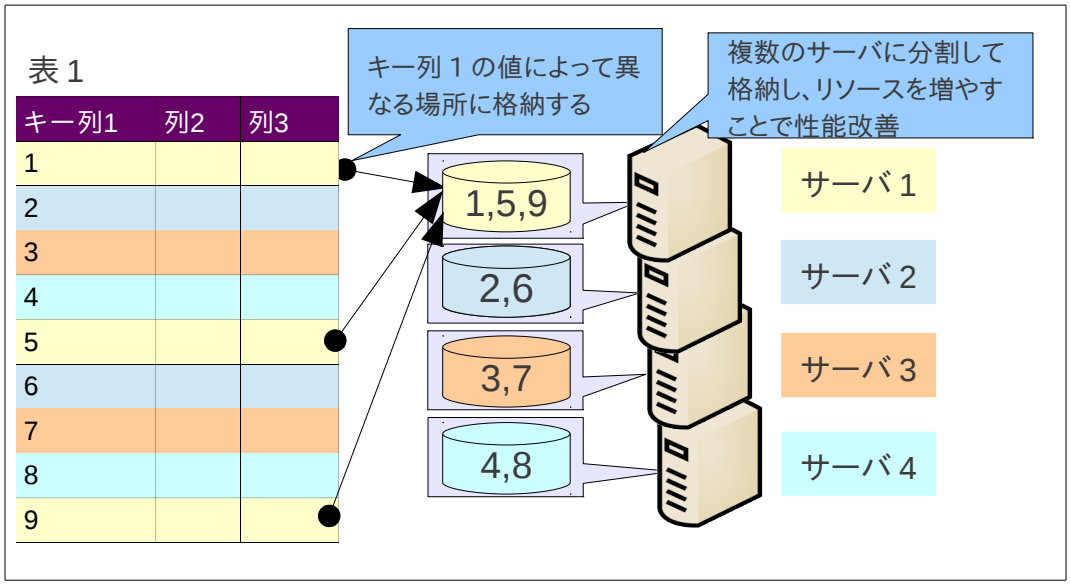

図 3.29: シャーディングの考え方

Postgres-XC も同様の考え方に基づいて、データを分散して格納し処理する機能を持ちます。即ち、事前に指定された一定のルールに従って、投入されるデータをデータノードに分散して格納することができ、問い合わせに際しては参照するデータノードを最小化します。

こうした基本的なデータ分散の仕組みによってシャーディングの利点を活かすためには、データベースに格納されるデータとそれに対する問い合わせを分析して、問い合わせ時に関連するデータノードの数が少ない構成にする必要がありますが、後で説明します。

○ レプリケーションによる高速化

先に述べたシャーディングによる高速化と並んで、Postgres-XC が提供するもう一つの高速化の手段は、指定された表の複製であるレプリカを各データノードに格納するレプリケーション[16]です。レプリカはすべて同一の内容を持つように制御されています。クエリを実行する際にはどのデータノードでもローカルに格納されているレプリカを参照すれば正しい内容が得られるため、特に参照系のクエリでは負荷を分散ことができるため、高速化が期待できます。

○ 高速化のためのクエリの最適化

このような考え方に従って、Postgres-XC には通常の PostgreSQL に加えて、2 つの最適化機能が用意されています。

- **絞り込み**

論理的な 1 つの表をデータノードに分散して格納する際には、一定のルールに従ってレコードが個々のデータノードに格納されます。このためルールのタイプとレコードの値によっては実際にデータノードを検索しなくても、格納さ

15「Shard (database architecture)」http://en.wikipedia.org/wiki/Sharding
16 ここでいうレプリケーションは、XC クラスタ上で 1 つの表の複製を複数のデータノードに格納することを意味します。通常の PostgreSQL でいうレプリケーションとは意味や手法が異なっている点にご注意ください。

れていないデータノードが特定できる場合があります。このようなデータノードが特定できる場合、そのデータノードにはクエリのリクエストを送らず、データが格納されている可能性があるデータノードにだけリクエストを送ります。その結果、クラスタ全体としてデータノードの負荷と（クラスタ内の）通信の負荷を軽減します。

もう一つの絞り込みの方法は、レプリケーションによって複製されている表への参照クエリに関するものです。この場合、全てのデータノードに表のレプリカが格納されていますから、コーディネータから見て「もっとも近い」データノードにだけリクエストを送ることで、通信の負担が軽減されます。Postgres-XC 固有の SQL コマンドである CREATE NODE 文における PREFERRED ノードの指定があります。

- **プッシュダウン**

問い合わせの種類によっては、データノードとコーディネータで処理を分担しますが、データノードでできる処理はできるだけデータノードで実行します。たとえばソートでは、各々のデータノードでソートした結果をコーディネータでマージする方が、全てのデータをコーディネータに収集してからソートするよりも応答時間が短くなります。このほか、結合（JOIN)処理でも、一定の条件を満たす場合には、可能な限りデータノードでも処理するようにプッシュダウンが行なわれます[17]。

（3） アプリケーションの例

この章では、先に説明した Postgres-XC の高速化の手法を、実際のアプリケーションに適用して高速化する例を紹介します。アプリケーションとしては、Postgres-XC コミュニティで配布している XC 用の pgbench を用います[18]。

○ pgbench の概要

pgbench は、PostgreSQL と一緒に配布されている contribute パッケージに含まれている、ベンチマークのためのワークロード（負荷）ツールです。TPC-B モデルに準拠した単純なトランザクションをデータベースに対して発行します。

pgbench は銀行業務を単純化したモデルをもっており、図 3.30 にある branch（支店）、tellers（行員）、account（口座）、history（履歴）という 4 つの表に対して、以下に示すような 5 つの操作を順次実行する 1 つのトランザクションを処理します。

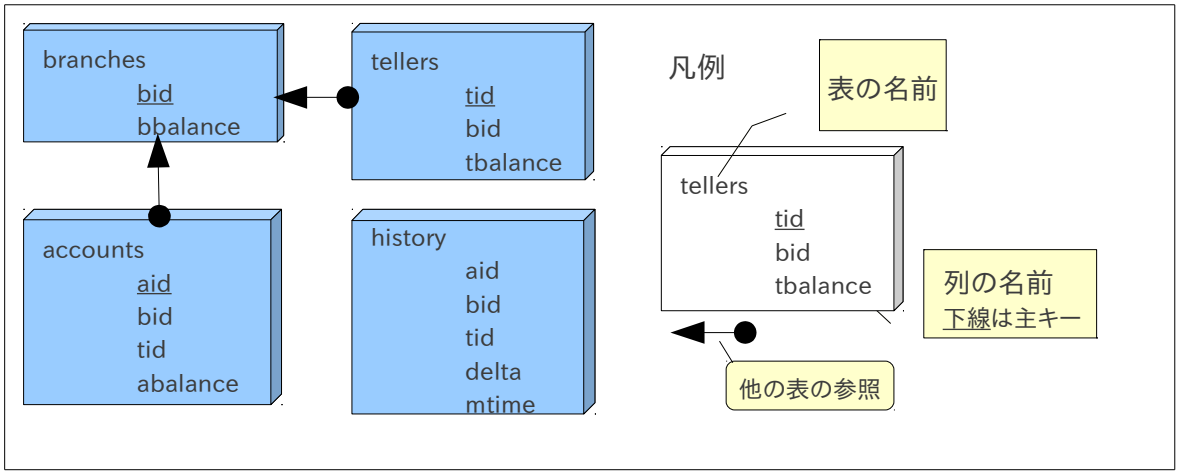

図 3.30: pgbench のテーブル構造

17「Postgres-XC Write Scalable PostgreSQL Cluster」,2012/9/18, http://www.slideshare.net/stormdb_cloud_database/postgresxc-write-scalable-postgresql-cluster

18 ここでは、Postgres-XC コミュニティのリポジトリ http://postgres-xc.git.sourceforge.net/git/gitweb-index.cgi にある pgbench.c（ver.1.0.2 2012 年 8 月 15 日版）をもとに説明します。

1. 行員(tid)が口座番号(aid)に振込額(delta)を振り込む(表 accounts を更新する)
2. その口座(表 accounts)の先ほどの口座(aid の行)の残高を読み出す
3. 行員表(tellers)の先ほどの行員(tid)の行の操作額(balance)に振込額(delta)を加え、更新する
4. 支店表(branches)にある先の支店(bid)の行の支店の残高(balance)に振込額を加え、更新する
5. 以上の操作を表すデータ(aid, bid, tid, delta, 現在時刻)を履歴表(history)に追加する

○ 高速化のための検討

一般に、あるシステムに Postgres-XC を適用して高速化するためには、(1)データの格納方法の検討、(2)リクエストを分散できるようなクエリの検討、の 2 つの手順が必要となります。

● データの格納方法の検討

Postgres-XC を有効に活用して性能を向上するには、先にご紹介したシャーディングの考え方に沿って、データを適切に分散する必要があります。Pgbench のテーブル構造を分析すると、

- accounts、tellers、history の 3 つの表は branches の主キーである bid をカラムに持つことから、口座(account)、行員(teller)、履歴(history)の各行データは支店(branch)に属しており、1 つの口座は 1 つの支店に属する、1 人の行員は 1 つの支店に属する、という性質がある。

- (pgbench の元になっている)TPC-B の仕様では、無作為に選ばれた行員に対して、その行員が属する支店を選ぶ。口座もまた無作為に選び出されるが、85%の確率で行員が所属する支店の口座を選び、15%の確率で他の支店の口座を選ぶ[19]。

- そこで、bid をキーとして、branches、tellers、accounts、history の 4 つの表を支店 ID(bid)に基づいてシャーディングします。このようにすると、85%のトランザクションはただ 1 つのデータノード上のデータのみを使用し、残りの 15%は 2 つのデータノードのデータを使用します。その結果、各データノードに負荷が分散されると同時に、1 つのトランザクションに必要となるデータノードの数を減らすことができます。

上記のような検討に基づいて4つの表のすべてを BID 列の値に基づいて各データノードに分散するようにします。どの表でも DDL での指定方法は同じなので、branches 表を例に示します(図 3.31)。図の下線の部分が、データの分散方法を指定しています。Postgres-XC が提供するデータ分散の方法は、指定されたカラムの値のハッシュ値に基づいて格納先のデータノードを決めるもの、データノードの数でカラムの値を割った時の剰余によるもの(MODULO)、到着順に順次格納先のデータノードを割り当てるものという、3 種類があります。
この例では、bid が与えられたときに格納先のデータノードが特定されるようにしたいので、bid の値のハッシュ値によって行データの格納先のデータノードを決めるように指定しています。

オリジナルの DDL

```
CREATE TABLE pgbench_branches bid int not null,bbalance int,filler char(88);
```

Postgres-XC 用に書き換えたもの

```
CREATE TABLE pgbench_branches bid int not null,bbalance int,filler char(88)
        DISTRIBUTE by HASH (bid)
```

図 3.31: DDL の修正

● リクエストの分散方法の検討

19 Transaction Processing Performance Council (TPC), “TPC BENCHMARK(TM) B”, http://www.tpc.org/tpcb/spec/tpcb_current.pdf

データを分散して格納することと並んで、リクエスト（SQL の DML 文）の実行が各データノード上に分散されるように、リクエストを工夫する必要があります。データの格納方法を検討した際に、各表の持っている BID に従って、その行をデータノードに割り当てるようにしました。スキーマがこのように定義されていると、Postgres-XC はリクエストされた SQL 文に基づいて、データノードに対して処理を支持します。そのため、アプリケーションが発行する SQL 文には”bid”が明示的に含まれている必要があります。引き続きオリジナルの PostgreSQL 用の pgbench を例に説明します。

オリジナルの DML

```
UPDATE pgbench_accounts SET abalance = abalance + :delta
WHERE aid = :aid
```

Postgres-XC 用に書き換えたもの

```
UPDATE pgbench_accounts SET abalance = abalance + :delta
WHERE aid = :aid AND bid = :bid
```

図 3.32: DML の修正

ここでは、トランザクションの中で最初につかわれる SQL 文を例に挙げています（図 3.32）。Postgres-XC 用に書き換えた個所は下線で示しています。オリジナルの SQL 文は、ある口座番号(aid)への振り込みを実行しますが、どの支店であるかは指定されていません。テーブル構造からは口座番号が決まれば、口座のある支店は暗黙のうちに決まるので、明示されていません。

このままの SQL 文を Postgres-XC が受け取っても処理範囲の絞りこみに使う情報(bid)が明示されていないので、すべてのデータノードに対して、この SQL 文を送りつけて処理させます。そこで、図の下のように SQL 文を書き換えます。テーブル構造の検討から、accounts 表においては列 aid が主キーになっていますから、その値が決まれば、列 bid の値も自動的に決まります。そこで、（意味的には冗長ですが）WHERE 句に”bid=:bid”という述語を加えることで、この SQL の結果の行においては、列 bid の値は一つに決まることを Postgres-XC のプランナに伝えるようにしています。このようにすると、この SQL の処理はただ 1 つのデータノードで実行すれば完全に得られることがわかるため、最適化処理の絞り込み機能が働くことによって、1 つのデータノードだけにこのリクエストが送られます。

3.4.2. 検証目的

Postgres-XC は比較的新しいクラスタソフトで、性能や機能、信頼性に関する知見で公開されているものは多くありません。そこで今回は性能面に焦点を絞って Postgres-XC の性能特性を検証します。

ストリーミングレプリケーションや pgpool-II とは違って、Postgres-XC は、データを分散配置することにより、更新系処理で性能がスケールすることが期待さるアーキテクチャです。本検証では、まず更新系処理でどの程度性能がスケールするかを見ていきます。また更新系処理のみならず、参照系の性能特性も検証します。

3.4.3. 検証構成

検証に使用する Postgres-XC のバージョンについては、2012 年 11 月時点の最新リリースである 1.0.1 を対象としています。Postgres-XC 1.0.1 は PostgreSQL 9.1.5 をベースとしているため、PostgreSQL 単体との比較をする際は、2012 年 11 月時点の最新メジャーバージョンである 9.2 系ではなく、9.1.5 を対象としています。

検証を実施するにあたり、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。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状況について検証します。

また、全ての Coordinator と Datanode が GTM へ接続する場合、GTM との通信処理がボトルネックになる可能性があります。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。

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、RAID5(4D+1P)で構成しています。ネットワーク環境は全てギガビットイーサを使用しており、Postgres-XC のコンポーネント同士の通信に使用するネットワークと、クライアントから各 Coordinator への接続に使用するネットワークは分割しています。

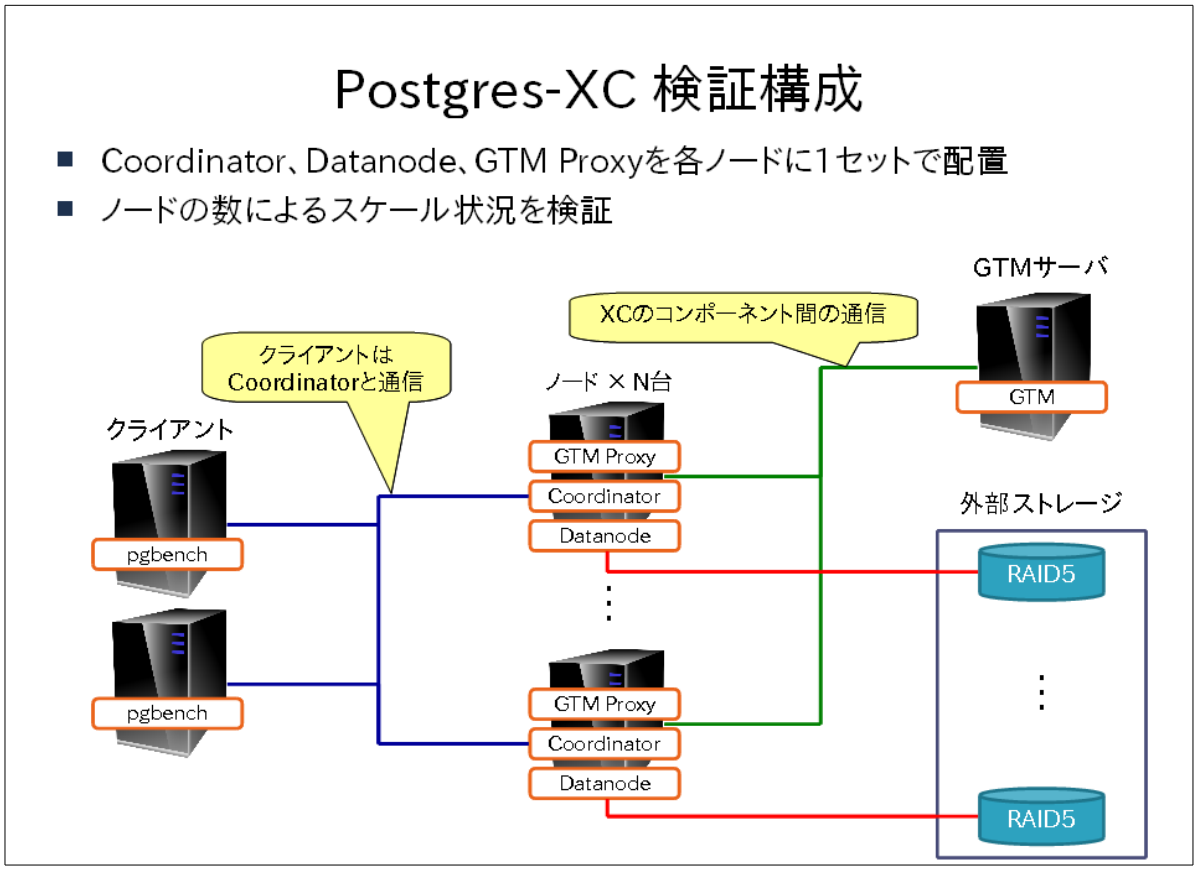

図 3.33: Postgres-XC 検証構成

なお、本検証では Postgres-XC のスケールアウト構成による性能の傾向を確認することに主眼を置いているため、各コンポーネントの冗長構成やバックアップのためのアーカイブログ出力等については設定していません。

Postgres-XC の検証では、更新系と参照系の測定に異なるハードウェアを使用しています。各検証において使用したハードウェア構成やサーバの台数について下表に示します。ハードウェア構成の詳細なスペック等については付録を参照してください。

表 3.10: Postgres-XC 検証 HW 構成

検証内容	ハードウェア構成	サーバ台数		
		GTM	ノード	クライアント
更新系	検証構成 2（付録参照）	1	1～4	1～2
参照系	検証構成 3（付録参照）	1	1～8	1～2

Postgres-XC 固有の設定については、接続先の情報や一意とする名称等、主に設定が必要なパラメータのみデフォルトから変更しています。

表 3.11: Postgres-XC パラメータ

コンポーネント（設定ファイル）	パラメータ	設定値	コメント
GTM (gtm.conf)	nodename	'gtm'	GTM に付与する名称。
	port	6668	GTM の待ち受けポート番号を指定。
GTM Proxy (gtm_proxy.conf)	gtm_host	GTM の IP アドレス	接続先の GTM のホスト、待ち受けポート番号を指定。
	gtm_port	6668	
Coordinator (postgresql.conf)	pgxc_node_name	'coordX' (X は連番)	Coordinator ごとに異なる名称を指定。Postgres-XC のコンポーネントとして登録する際などに使用。
Datanode (postgresql.conf)	pgxc_node_name	'dnodeX' (X は連番)	Datanode ごとに異なる名称を指定。Postgres-XC のコンポーネントとして登録する際などに使用。

また、Coordinator と Datanode の設定については PostgreSQL のパラメータを継承しています。評価を実施するにあたり、影響の考えられるパラメータについてのみ変更をしており、細かいチューニングは実施していません。

表 3.12: PostgreSQL パラメータ

パラメータ	設定値	コメント
max_connections	1000	Coordinator の場合、個々の Coordinator への最大接続数を設定。 Datanode の場合は、全ての Coordinator から接続されるため、 「Coordinator への最大接続数 * Coordinator 数」よりも大きな値を設定する必要がある。
shared_buffers	8GB	デフォルト値は小さすぎるため、環境に合わせて調整。
port	5432、5433	Coordinator と Datanode を同一サーバに同居させているため、クライアントからの接続を受け付ける Coordinator を 5432、Datanode を 5433 とした。
max_prepared_transactions	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。
checkpoint_segments	16	デフォルト値は小さすぎるため、環境に合わせて調整。
autovacuum	off	測定結果への影響を考慮して停止。

3.4.4. 検証方法

（1） 更新系

○ 環境作成

スケールファクタ 100 で初期化します（データベースサイズは 1.5GB）。なお、pgbench は Postgres-XC 同梱の改造版を用い、-k オプションという独自のオプションをつけています。これは、各データノードに自動的にデータを分散するものです。

```
$ pgbench -i test -k -s 100
```

また、各測定後に毎回初期化を行ないました。

なお、チェックポイント処理で負荷がかかる可能性があるため、チェックポイントの発生頻度を下げるために checkpoint_segments を増やしました。

```
checkpoint_segments = 1000
```

○ 測定

以下のコマンドの結果を取得しました。100 クライアント、10 スレッドで 600 秒（10 分）実行した結果です。

```
$ pgbench -n -k -c 100 -j 10 -T 600
```

○ 結果

データノードのノードが増えるほど各ノードでの処理数（平均の列）が減り、総合的（合計の列）には tps が高くなりました。また、ノードが 1 台のときは合計 tps が PostgreSQL 単体より低くなりますが、ノードが 2 台以上だとそれを上回ることが確認できました。
グラフの縦軸は tps です。横軸はノード数で、ノード 0 は PostgreSQL 単体での tps です。

表 3.13: SF=100 の時の結果 tps

ノード数	ノード 1	ノード 2	ノード 3	ノード 4	合計	平均
PostgreSQL 9.1	1135.00	-	-	-	1135.00	1135.00
1 ノード	1032.08	-	-	-	1032.08	**エラー表現**
2 ノード	710.71	708.38	-	-	1419.08	709.55
3 ノード	754.40	755.47	737.09	-	2246.97	748.99
4 ノード	583.26	568.35	569.22	638.55	2359.38	430.20

○ 結果

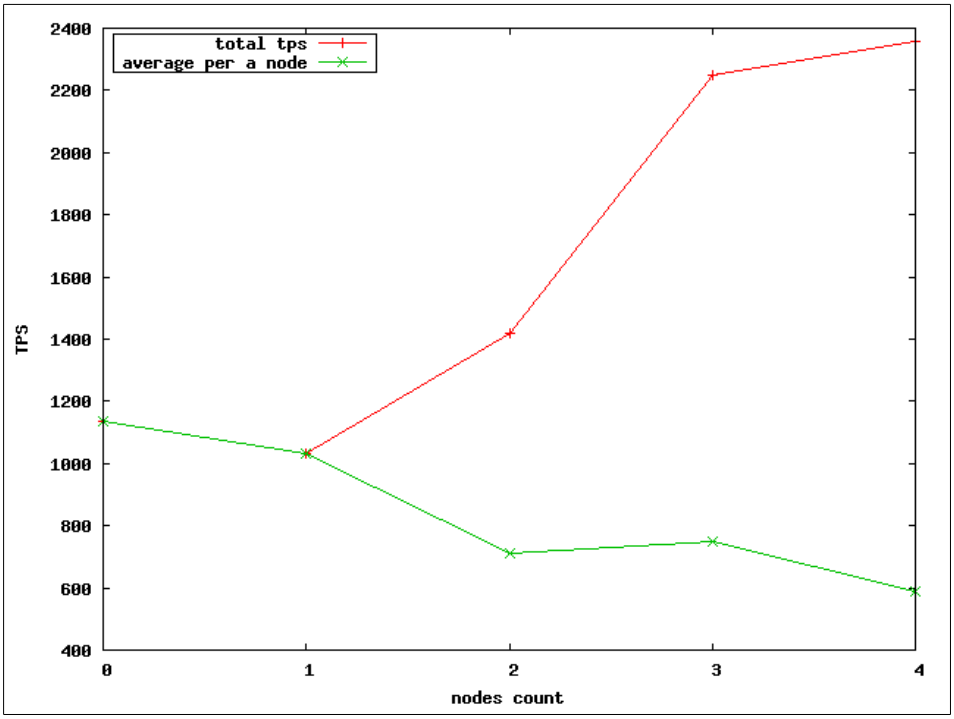

図 3.34: 更新系の結果(スケールファクタ=100)

更にデータ量を増やし、スケールファクタ=5000(データ量で 75GB)の場合でも、ノードを増やしてスケールする結果が得られましたが、性能向上の割合はスケールファクタ=100 の時ほど顕著ではありませんでした。

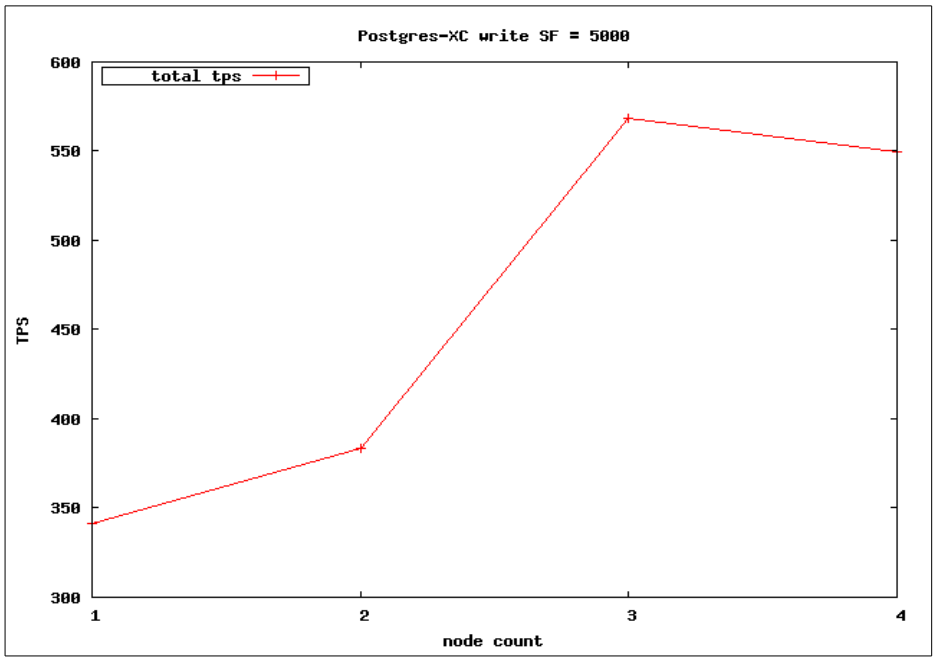

図 3.35: 更新系の結果(スケールファクタ=5000)

（2） 参照系

○ 環境作成

スケールファクタ 1000 で初期化します。こちらも、pgbench は Postgres-XC 同梱の改造版を用い、-k オプションをつけて登録データを分散しています。

```
$ pgbench -i test -k -s 1000
```

○ 測定

以下のコマンドの結果を取得しました。100 クライアント、10 スレッドで 300 秒（5 分）実行した結果です。

```
$ pgbench -c 100 -j 10 -n -k -S -T 300
```

○ 結果

ノード数が増えるほど合計 TPS は上がっていますが、8 ノードまで増やしても単体 PostgreSQL の性能を上回ることができませんでした。
データが複数の Datanode に分散しているため、参照系では性能がでにくい、という特性が確認できました。

表 3.14: 結果 tps

ノード数	ノード 1	ノード 2	ノード 3	ノード 4	ノード 5	ノード 6	ノード 7	ノード 8	合計	平均
PostgreSQL 9.1	70449	-	-	-	-	-	-	-	70449	70449.00
1 ノード	11673	-	-	-	-	-	-	-	11673	11673.00
2 ノード	14711	15491	-	-	-	-	-	-	30202	15101.00
3 ノード	12743	12825	14136	-	-	-	-	-	39704	13234.67
4 ノード	9668	10958	14686	9971	-	-	-	-	45283	11320.75
8 ノード	5719	5720	5715	5442	7122	7122	7126	5777	49743	6217.88

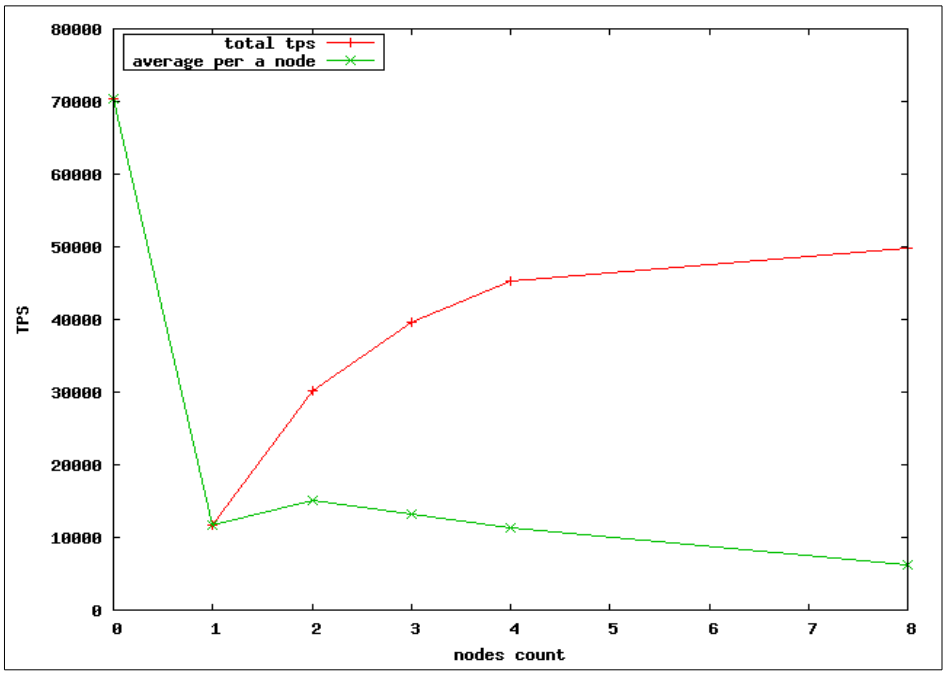

図 3.36: 参照系の結果(スケールファクタ=1000)

○ 更にデータを増やす

更にデータ量を増やすと、データがキャッシュに乗る効果が現れて性能が大きく向上します。以下のグラフはスケールファクタを 5000 にしたときの参照系の結果ですが、3 ノード以上でキャッシュの効果が顕著に現れているのが分かります。

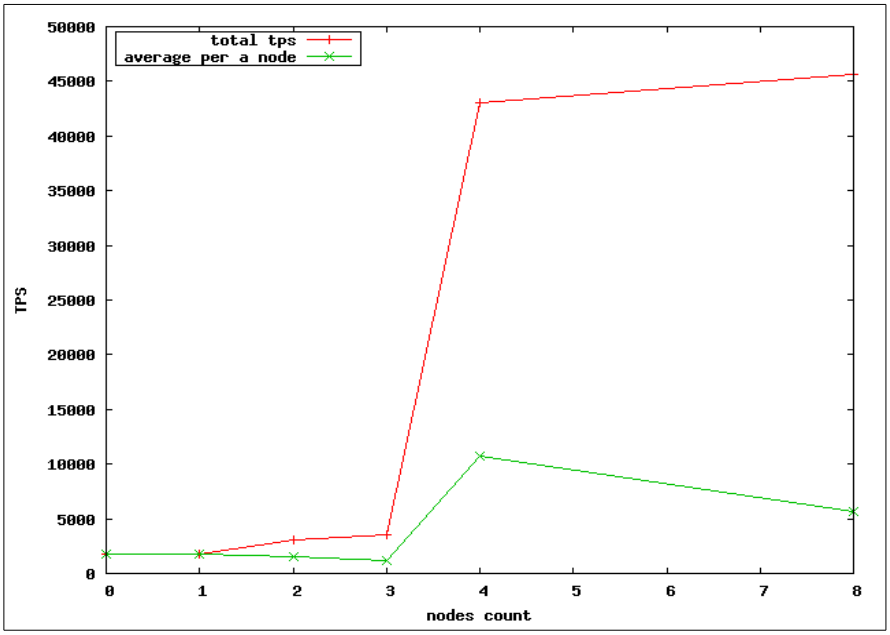

図 3.37: 参照系の結果(スケールファクタ=5000)

3.4.5. 考察

Postgres-XC は更新負荷を分散することによって、更新処理のスケーラビリティを確保するアーキテクチャになっています。更新系の検証では、そのことが裏付けられる結果となりました。

このことから、データ量が増えるにつれて Postgres-XC の更新処理におけるメリットがより明確になるのではないかと予測してスケールファクタ=5000(スケールファクタ=100 のデータ量が 1.5GB であるのに対し、スケールファクタ=5000 ではデータ量が 75GB)でも検証を行ないました。しかし、結果は予測に反してむしろスケールアウトしにくい結果になりました。この理由は今のところ不明で、大容量のデータのときの Postgres-XC の更新系の振る舞いについては今後の検証が必要と思われます。

一方参照系においては、スケールファク=1000 の時には、ノード数を 8 まで増やしても単体の PostgreSQL に及ばない結果となってしまいました。しかし、データ量を増やすことによって、キャッシュの効果が現れ、性能が向上しました。

以上から、特に大容量のデータを扱うような業務で、更新と参照がミックスするような業務においても、Postgres-XC を導入することによって良好な性能が得られる可能性があると思われますが、更なる検証が必要です。また、参照系では JOIN を伴うような比較的複雑な問い合わせで Postgres-XC がどのような振る舞いをするかについても、今後検証を行ないたいところです。

3.5. スケールアウト検証サマリ

PostgreSQL のスケールアウト検証として、PostgreSQL9.2 のカスケードレプリケーションの基本特性、PostgreSQL に pgpool-II を組み合わせた方法での参照/更新性能、PostgreSQL ベースの分散 DBMS である Postgres-XC の参照/更新性能について、企業で使用されるような機器を使用して検証しました。
共同検証の結果、次の点を確認することができました。

- 参照スケールアウト構成のバックエンド DB として威力を発揮する PostgreSQL のカスケードレプリケーション
- PostgreSQL のフロントに置くことで参照スケールアウト構成に威力を発揮する pgpool-II
- 更新スケールアウトで有望な Postgres-XC

負荷の増大に伴って横方向に機器を増やし性能キャパシティを広げるのがスケールアウトです。今回の検証では PostgreSQL を中心にしたスケールアウト構成として代表的なものをとりあげました。企業ユースでは、DBMS 単体だけでなくアプリケーションサーバや Web サーバなど様々な要素が関連します。参照/更新比率を変化させた検証など、他にも有用な検証が考えられますが、今回のスケールアウト検証結果が、PostgreSQL を適材適所でスケールアウト構成をとる際の参考になれば幸いです。

4. おわりに

本報告書では、2012 年 4 月に 10 社が集まって発足した PostgreSQL エンタープライズ・コンソーシアム(PGECons)の初年度の活動として、技術部会ワーキンググループ1（WG1）が実施したスケールアップ検証とスケールアウト検証について、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。

WG1 の参加企業が確定して活動を開始したのが 2012 年 7 月末で、8 月から 9 月にかけて実施計画を策定している期間の 2012 年 9 月にタイミング良く PostgreSQL 9.2 がリリースされたこともあり、検証の対象は PostgreSQL 9.2 での新機能が中心となりました。具体的には、スケールアップでは PostgreSQL 9.2 でのマルチコア性能、スケールアウトは PostgreSQL 9.2 カスケードレプリケーションです。またスケールアウトの検証対象として、pgpool-II と Postgres-XC の性能特性（参照系・更新系）も選定されました。

今回の活動を振り返ってみると、初年度はどのように進めるかや合意の形成など、実際に活動しながら試行錯誤を繰り返して進めていくワーキンググループの形を作る期間だったかと思います。

一番良かった点としては、個人レベルではなかなか利用が難しいハードウェアを使ったリソースを活用した検証を、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って出来たことです。日頃は競合することもある各企業のメンバーが、今回オープンソースでのコミュニティ活動として PostgreSQL のエンタープライズ活用を促進させる共通の目的で一緒に検証を実施することで、本報告書での成果発表とともに技術者同士の交流を深めることが出来て、PGECons の設立趣旨に合致したかと感じています。

今回実施できなかった「パーティション」機能の検証や今年リリース予定の PostgreSQL 9.3 の新機能の検証などを次年度に実施したいと考えています。

PostgreSQL がエンタープライズで利用されるようなきっかけとなるように今後も活動を継続していきますので、何かやってみたいと思われた方は次の活動を一緒に実施していきましょう!