PostgreSQL を用いた分散データベースの開発

永安 悟史*

平成 14 年 12 月 2 日

概 要

オープンソース RDBMS である PostgreSQL は、その優れた機能と性能により多くのユーザーを獲得するに至ったが、エンタープライズ利用において必要とされている機能のいくつかが欠けていた。本研究では、ネットワーク上に分散した複数のサイト間にまたがるデータの一貫性を保証する二相コミットを実装し、その応用として、特に高可用性を実現するための「レプリケーション」、および負荷分散や大容量データベースのための「分散データベース」の実装を行っている。

1 はじめに

PostgreSQL は、MVCC(Multi-Version Concurrency Control) や WAL(Write Ahead Log) などの機能を備えたオープンソースの RDBMS である。

PostgreSQL は、堅牢であること、機能が豊富であること、性能が良いこと、一般的な商用 RDBMS と違ってライセンスフィーが不要であることなどから、広く利用されるようになったが、商用 RDBMS などと比べて見劣りするのが、エンタープライズ分野におけるミッションクリティカルなシステムや大規模なデータベースを構築するための機能である。

業務でデータベースを利用する場合、ネットワーク上に分散している複数のデータベースサーバに対して透過的に問い合わせや更新処理を行いたいといったニーズや、あるいは 24 時間止められないサービスを提供（運用）する場合に、サーバを多重化して冗長性を確保することで可用性を高めたい、といったニーズが存在する。

しかし、PostgreSQL は単一のサーバ上での利用を中心に開発されているため、ネットワーク上に分散したサーバ間で一貫性を確保するための機能が提供されていない。

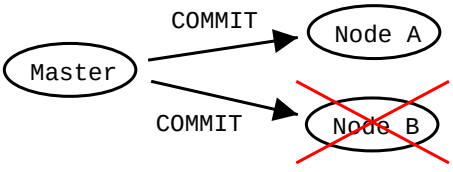
そのため本研究では、ネットワーク上に分散したノード間における一貫性を確保するための手法としての「二相コミットプロトコル（2-phase commit protocol）」を PostgreSQL に実装し、二相コミットを有効に利用する機能として「同期レプリケーション」および「分散データベース」の開発を行なっている。

2 二相コミットプロトコル

2.1 なぜ二相コミットプロトコルなのか

リアルタイムの同期レプリケーションであれ、分散データベースであれ、「複数のサイト（ノード）間の一貫性を保持しなければならない」というのはトランザクション処理の大前提である。

一般的に、複数のノードをまたいだトランザクションを処理する場合、コミット処理中に一部のノードにおいてエラーが発生し、コミット処理を確実に完了できない可能性が生じる。



*慶應義塾大学大学院 政策メディア研究科

そのような事態が起こった場合に、トランザクションログを用いて確実に復旧し、一貫性を保持することができるのが「二相コミットプロトコル(2-phase commit protocol)」である。

分散したノード間での一貫性保持のための研究はさまざま行われているが、二相コミットは

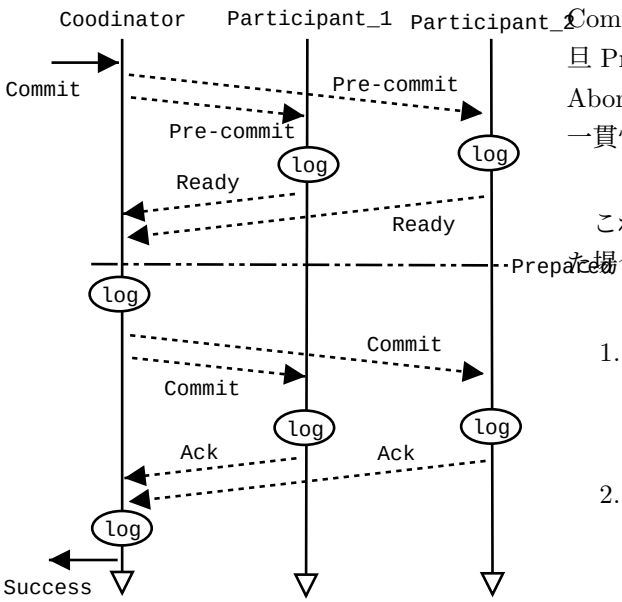
- 理論的にシンプルで理解しやすいこと
- リカバリが容易であること
- 実装が比較的容易であること

などから、商用システムなどにおいても広く実装されており、信頼性が高い。

これらのことから、本研究では二相コミットをPostgreSQL へ実装することとした。

2.2 理論

以下は、二相コミットプロトコルの処理の流れである。



二相コミットは、各ノードを取りまとめる調停者（Coordinator）とトランザクションへの参加者（Participant）の間で、それぞれメッセージを送信する際にログを記録することによって、その状態で障害が発生しても、後刻ログをもとに復旧を行うことができるようなプロトコルになっている。

1. 調停者: クライアントからの `COMMIT` が到着
2. 調停者: 参加者に対して Pre-commit を通知
3. 参加者: エラーなくコミット処理ができるなら、Pre-commit を遅滞なくログに記録し、調停者に Ready for Commit を返す
4. 調停者: すべての参加者から Ready for Commit が返ってきたら、ログに Commit 通知を記録し、すべての参加者に Commit を通知
5. 参加者: Commit をログに記録し、Ack を返す
6. 調停者: すべての参加者から Ack が返ってきたら、クライアントに `OK` を返す

という手順になる。

この際、ある参加者においてエラーあるいは障害が発生し、調停者からの Pre-commit に対して Ready for Commit を返せない場合、Abort を返す。これにより、調停者はトランザクションを完了できないことを検出し、すべての参加者に対して Commit ではなく Abort を通知する。よって、一旦 Pre-commit されたすべての参加者においても Abort を行い、一部のエラーあるいは障害によって一貫性が崩れることを防ぐのである。

これらのことから、通常の一相コミットと比較した場合に加えるべき変更は、

1. Participant が Pre-commit をログに記録でき、Pre-commit された状態（Prepared 状態）を保持できること
2. Coordinator が Participant への Commit 発行をログに記録でき、その状態を保持できること
3. Participant と Coordinator 間でこれらの通信ができること
4. 障害時にログにもとづいてログに記録した状態まで復旧できること

となる。

2.3 PostgreSQL における設計と実装

2.3.1 トランザクション状態の拡張

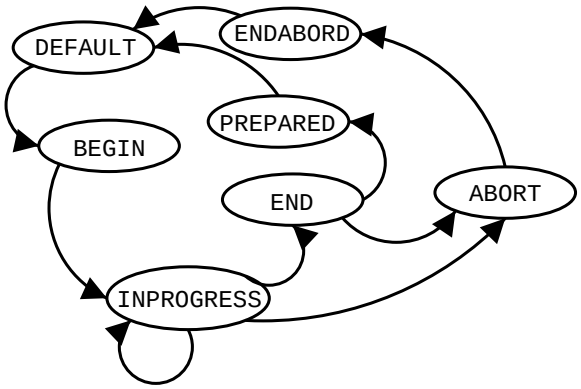
PostgreSQL では、ふたつの状態変数でトランザクションの状態を管理している。

Transaction Block State ひとつは TBLOCK_ で始まるトランザクションブロックの状態変数で、クライアント（アプリケーション）から見てトランザクションがどのような状態にあるのかを保持する変数である。

```
#define TBLOCK_DEFAULT      0
#define TBLOCK_BEGIN        1
#define TBLOCK_INPROGRESS   2
#define TBLOCK_END          3
#define TBLOCK_PREPARED     4 // 追加
#define TBLOCK_ABORT        5
#define TBLOCK_ENDABORT     6
```

通常は TBLOCK_DEFAULT であり、SQL の BEGIN コマンドによって TBLOCK_BEGIN を経て TBLOCK_INPROGRESS へ変わり、COMMIT コマンドによって TBLOCK_END を経てTBLOCK_DEFAULT へと戻る。

TBLOCK_INPROGRESS 中に SQL エラーや ABORT コマンドなどによりトランザクションがエラー状態になった場合には、TBLOCK_ABORT や TBLOCK_ENDABORT を経てトランザクションをコミットせずにTBLOCK_DEFAULT へと戻る。



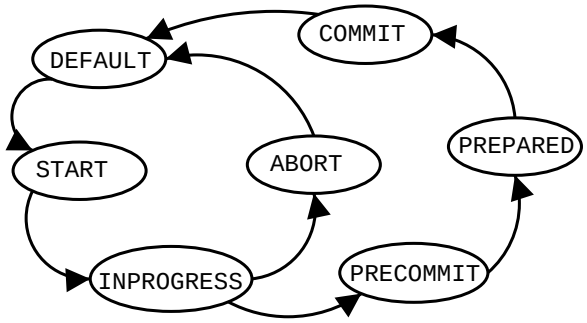
ここでは、Pre-commit を完了した Prepared 状態について、TBLOCK_PREPARED を追加した。

Transaction State 二つ目は、TRANS_ で始まるもので、より内部実装に則した状態変数である。

```
#define TRANS_DEFAULT       0
#define TRANS_START         1
#define TRANS_INPROGRESS    2
#define TRANS_PRECOMMIT     3 // 追加
#define TRANS_COMMIT_READY  4 // 追加
#define TRANS_COMMIT        5
#define TRANS_ABORT         6
```

トランザクションがコミットされ、内部でコミット処理が始まるとTRANS_INPROGRESS から TRANS_COMMIT へと移行する。コミット処理が終了すると、TRANS_COMMIT からTRANS_DEFAULT へと移行する。

ここでは、Pre-commit を処理中のTRANS_PRECOMMIT、および Pre-commit が完了して Ready for Commit 状態であるTRANS_COMMIT_READY 状態を追加した。



2.3.2 ログの拡張

トランザクションログ (WAL) のエントリの追加 トランザクションログには、処理の内容を示す定数を記録する。障害が発生してリスタートするときには、この値を用いてリカバリ処理の内容を決定する。

```
#define XLOG_XACT_COMMIT      0x00
#define XLOG_XACT_PRECOMMIT   0x10 // 追加
#define XLOG_XACT_ABORT       0x20
```

コミットログ (clog) の状態の追加 PostgreSQL は処理中のトランザクションの状態をコミットログ (clog) と呼ばれるファイルに、IN_PROGRESS、COMMITTED、ABORTED を 2bit 値として保持している。今回は、clog に対して新たにPRECOMMITTED 状態を追加した。

```
#define TRANSACTION_STATUS_IN_PROGRESS  0x00
#define TRANSACTION_STATUS_PRECOMMITTED 0x01
                                        // 追加
```

```
#define TRANSACTION_STATUS_COMMITTED    0x02
#define TRANSACTION_STATUS_ABORTED      0x03
```

2.3.3 Pre-commit と Commit への分離

従来のコミットの場合には、コミット処理を行う関数CommitTransction() において

1. Commit の前処理
2. Commit 完了をログに記録
3. Commit の後処理

のようになっていた。今回これを、

1. Commit の前処理
2. Pre-commit 完了をログに記録
3. Commit 完了をログに記録
4. Commit の後処理

のようにログの記録の部分で二重化し、1. および 2. をPreCommitTransaction() として、また 3. および 4. をCommitTransaction() として二相に分離した。

3 レプリケーション

「レプリケーション」とは、可用性を高めたり、負荷分散を行うためにデータベースの複製を作成することである。その実現方法には、「同期／非同期」、「事前／事後」など複数の手法があり、状況に応じて使い分ける必要がある。

3.1 レプリケーションの種類

3.1.1 同期／非同期

レプリケーションの作成を完全に同期させて行うか、事後的に非同期に行うかという区別である。

前者の場合、更新するデータを I/O に反映させる前にレプリカへ転送し、転送されたデータを用いて同時に更新処理を行う。同期方式のメリットは、どの時点でも完全に同じデータベースを複数作成することができることであるが、逆にデメリットは更新のときに同期を取る、つまり遅い処理を行うノードを待つため、更新処理のパフォーマンスが低下することである。

後者の場合、マスターサーバに作成されたテーブルを読み取って、レプリカサーバへのコピーを作成する方法や、マスターサーバの更新ログファイルを用いてレプリカサーバにも同じ更新処理を行う方法とがある。非同期方式のメリットは、マスターサーバは単独で動作するために、同期方式でデメリットであった更新処理のパフォーマンス低下を避けられ。逆にデメリットは、マスターとレプリカを同期させるタイムラグが発生するために、厳密に見るとマスターとレプリカで異なるデータを持っている時間帯が発生することである（このタイムラグは運用で決定することになる）。

3.1.2 クエリベースのレプリケーション

クエリベースのレプリケーションでは、SQL の問い合わせ文字列をレプリカへ転送することでレプリケーションを行う。

クエリベースのレプリケーションの実装のメリットは、実装が非常に容易であるということである。データベースのバックエンドのエンジンに対して大きな変更を行う必要がないため、実現にさほど困難はない。

デメリットは、更新クエリが、クエリあるいはノード（データベースサーバ）によって異なるレコードを生成する場合に、レコードの内容がノードによっって違ってきてしまうことである。例えば、現在の時間を表わすCURRENT_TIMESTAMP や乱数発生関数rand() などがその典型である。これらの問題を回避するには、バックエンドあるいはアプリケーション側で何らかの対応が必要となる。

3.1.3 レコードレベルのレプリケーション

レコードレベルのレプリケーションでは、レコードが生成され、I/O に反映される時点でレプリケーションを行う。

レコードレベルのレプリケーションのメリットは、クエリベースのレプリケーションで問題となっていた「ノードによって異なる値」を回避できるということである。rand() などは実際に値が生成された後でレプリケーションされるので、ノードごとに値が違うということは発生しない。

デメリットは、データベースエンジンへの変更が多岐に渡るため、実装するのがより困難であることである。

3.1.4 トランザクションログを用いたレプリケーション

トランザクションログを用いたレプリケーションは、先のふたつの方式と違い、既にレコードが生成されたという記録（ログ）を用いてレプリケーションを行う。トランザクションログに記録された内容を読み取り、どのテーブルのどのレコードに更新が行われたのかを判断して、レプリケーションを行う。

この方式のメリットは、データベースエンジン本体にほとんど手を加える必要がないということ、また、マスターとなるデータベースのパフォーマンスにほとんど影響を与えないことである。

デメリットは、マスターデータベースに更新が反映されてからレプリケーションを行うため、マスターとスレーブの間の同期にタイムラグが生じることである。

3.2 PostgreSQL における設計と実装

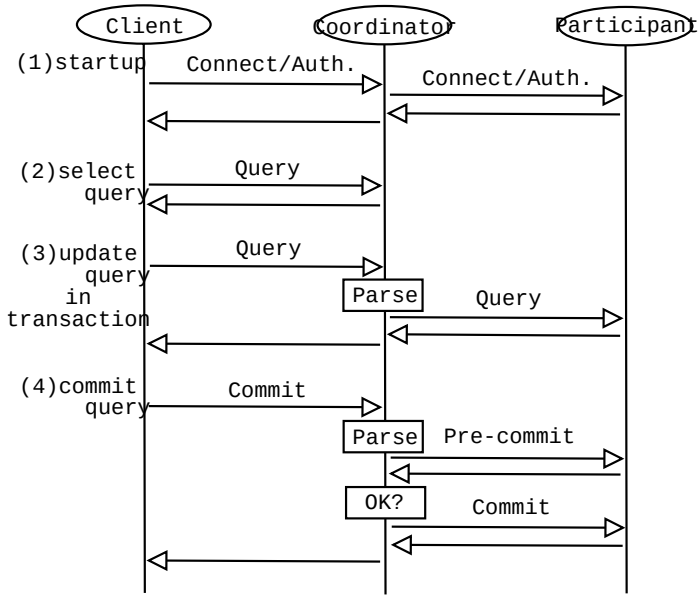
3.2.1 アーキテクチャ

本研究では、レプリケーションのアーキテクチャとして「同期方式」および「クエリベース」を採用した。

その理由として、「非同期方式」によるレプリケーションについては、複数の実装が既に行なわれていること、また先に実装した二相コミットによって同期レプリケーションにおいて「分散したサイト間の一貫性」を確保することが可能となるためである。

レプリケーションを行うレイヤーについては、バックエンドのデータベースエンジンや、バックエンド・クライアント間のプロトコルに大きな変更を必要としないことから「クエリベース」のレプリケーションを採用した。

以下は、その処理フローである。



(1) スタートアップ　クライアントアプリケーションからデータベースサーバへ接続されると、そのサーバ上でバックエンドが起動し、そのバックエンドが調停者となり、他の参加者サーバへのレプリケーションのセッションを張る。そして、すべての

(2) SELECT クエリ　SELECT を用いた問い合わせ処理の場合、すべてのノードが同等のデータを保持しているため、他のサーバへクエリを転送する必要がない。そのため、クエリを受け取った調停者は、自分自身が検索処理を行なってクライアントに返す。ただし、更新処理を伴う関数の呼び出しなどを行う場合にはこの限りではない。

(3) 更新処理　トランザクションブロックの内部（明示的なBEGIN コマンドの後）においてUPDATE や INSERT、およびCREATE など、更新を行うクエリを受け取った場合、調停者はレプリケーションクラス

タの参加ノードすべてにクエリの文字列を転送する。すべての参加者に対してこの処理が成功していれば、調停者はクライアントに対して「成功」を返す。

(4) コミット処理　クライアントアプリケーションからCOMMIT を受け取った場合、調停者は各参加者に対してコミットが可能かどうかの問い合わせを行うすべての参加者が自分自身のトランザクションログに対して Precommit を書き出し、コミット可能を返したら、調停者はすべての参加者に対してコミット確定の通知を送る。

逆に、一部の参加者において「コミット不可能」が返ってきた場合調停者はすべての参加者に対してABORT を送る。

1. クライアントからマスター（Coordinator）へ接続
2. 認証の後、マスターがレプリカへ接続・レプリカを起動
3. クライアントがマスターにクエリを送信
4. マスターがクエリを parse し、レプリカへ転送（SELECT 以外）

3.2.2　libpq プロトコルの拡張

クライアントからの接続を受けて調停者となったバックエンドが、他の参加者に対して接続を行う場合に二相コミットを行う必要があるが、このプロトコルについては、従来の PostgreSQL のクライアント・サーバ間の通信プロトコルである libpq プロトコルを拡張したものを実装した。

3.3　パフォーマンス評価

3.3.1　検証環境

サーバ環境

CPU PentiumIII 700MHz x 2 (SMP)

Memory 4GB

OS Linux kernel 2.4.18

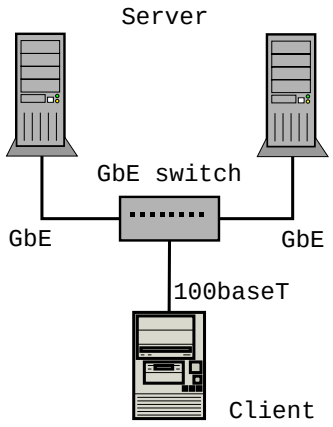
クライアント環境

CPU Celeron 1GHz

Memory 128MB

OS Linux kernel 2.4.9

ネットワーク環境



3.4　pgbench を用いた測定

3.4.1　パラメータ

pgbench は、PostgreSQL に付属している TPC-B 相当のベンチマークを行うツールである。今回は以下のパラメータを用いて測定を行なった。

スケーリングファクター 10 (100 万レコード)

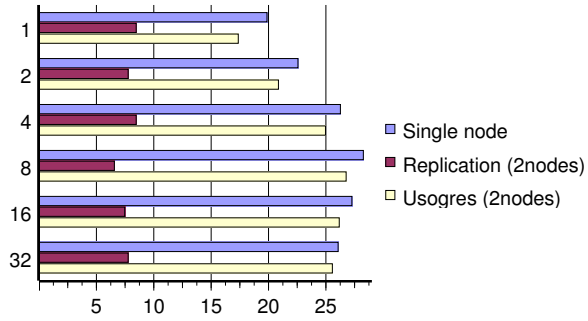
クライアント数 1,2,4,8,16,32

トランザクション数/クライアント 64

以上の条件により 3 回計測を行い、3 回目のデータを採用した。

3.4.2　通常の測定

以下は、pgbench を通常の設定で実行した結果で、出力は「tps （transaction per second)」である。

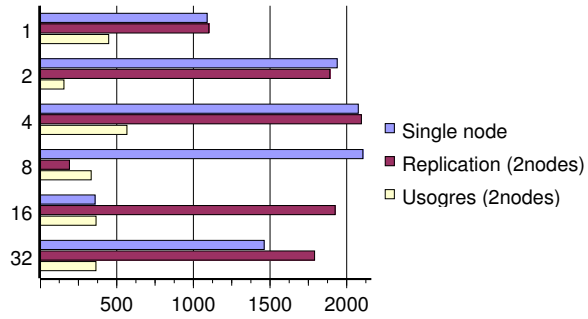


通常の PostgreSQL と比較して大幅にパフォーマンスが低下（およお 1/3）していることが分かる。ここで、比較のために Usogres を用いて同じような 2 台構成のレプリケーションのベンチマークを行なってみたところ、Usogres のパフォーマンスは単一の PostgreSQL のパフォーマンスと比較して遜色のないものであった。

このことから、二相コミットを用いたレプリケーションにおけるパフォーマンス低下の原因が、ネットワーク周りでないこと、およびリモートサイトの PostgreSQL バックエンドの起動処理などにはないことが予想される（これらの処理は二相コミットレプリケーションでも Usogres でも似たようなアーキテクチャを取っている）。

3.4.3 SELECT のみの測定

以下は、「-S」オプションを指定して、検索（SELECT）のみのベンチマークを行なった結果である。検索に比較して更新が非常に少ないシステムの場合を想定したテストとなっている。



SELECT のみのベンチマークを行なった場合、場合によって測定結果が大きくバラつくことがあったが（クライアント数 8 および 16）、単一の PostgreSQL と比較しても性能の劣化はさほど認められなかった。

3.5 更新処理のオーバーヘッドの測定

pgbench によるテストの結果を総合して予測できることは、二相コミットのレプリケーションにおいて、大きなボトルネックを生み出だしているのは、UPDATE などを始めとする更新処理であるということである。

そのため、二相コミットにおける更新処理のオーバーヘッドを測定するために以下のテストを行なった。

3.5.1 テスト内容

1. テーブル作成
   ```
   CREATE TABLE t1 (
       uid INTEGER PRIMARY KEY,
       counter INTEGER )
   ```
2. レコード作成
   ```
   INSERT INTO t1 VALUES ( 1, 0 )
   ```
3. `BEGIN` を発行（all の場合）
4. `UPDATE t1 SET counter=$i WHERE uid=1` として `$i` を 0 から 9999 まで更新
5. `COMMIT` を発行（all の場合）

テーブルとレコードの作成の後、10,000 回の `UPDATE` を発行するが、この時、10,000 回の `UPDATE` をひとつのトランザクション（`BEGIN`〜`COMMIT`）で発行するパターン（all）と、それぞれの`UPDATE` で自動的にコミット（autocommit）を行うパターン（each）の、二種類のトランザクション処理を定めた。

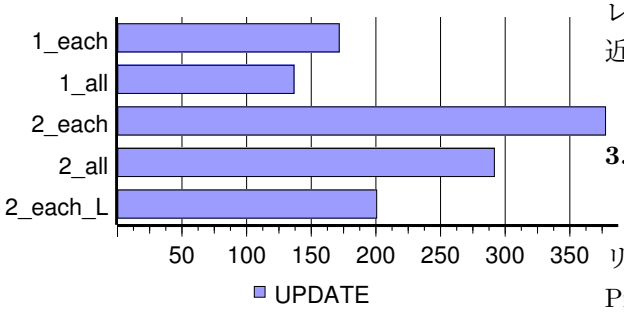
また、これら二種類のトランザクション処理（each と all）について、単独の PostgreSQL サーバ、および二台の PostgreSQL サーバ間でレプリケーションを行う二つのケースについてテストを行なった。

3.5.2 測定内容

以下はそれぞれのテストを実行した際の経過時間（秒数）である。数値は三回実行した三回目の値を

採用した。

	each	all
1 (single)	172	137
2 (replicate)	378	282
2 (rep. & lazy)	201	-



3.5.3 コストの推定

ここで、CREATE TABLE および INSERT INTO は全体の実行時間に比較して非常に短い時間で終了するので無視することにする。

1_all と2_all との比較において、BEGIN と COMMIT の発行回数は同じ（一回ずつ）であるので、経過時間の違いは「UPDATE を 10,000 回転送した際のパフォーマンス」の差と見なすことができる。経過時間の差は 155 秒なので、1 回のUPDATE を転送に 15ms ほどオーバーヘッドが生じていることになる。

次に、2_each と 2_all を比較すると、両者の違いはCOMMIT がすべての UPDATE について（暗黙のうちに）発生するか、一回だけ発生する（明示的に発生させる）かの違いである。両者の経過時間の差は 96 秒であるので、これにより、COMMIT を一回発行したときの（2nd phase の）コストは約 9ms となる。

最後に、1_each と2_each の測定内容から、両者の測定結果の差は「各UPDATE を転送して、かつ各々のUPDATE についてCOMMIT を発行し、2nd phase を実行するコスト」となる。10,000 クエリのオーバーヘッドが 206 秒であるので、1 クエリのオーバーヘッドは 2ms となり、これは先に計算した「UPDATE を転送するコスト」と「2nd phase を実行するコスト」を足した値に近いことが確認できる。

なお、表中にある 2_each_L は、Pre-commit 時にログのフラッシュ（ディスクへの強制書き出し）を行なわずに実行した場合である（lazy write）。2_each と比較すると実行時間が劇的に短くなり、レプリケーションを行なっていない 1_each により近い実行時間で処理を終えている。

3.5.4 考察

これらのことから、二相コミットを用いたレプリケーションにおけるオーバーヘッドの大部分は、Pre-commit 時のログフラッシュによって発生していると考えられる。

しかしながら、二相コミットにおいては Pre-commit 時に即時に確実にログに反映されること（ログの強制書き出し）が求められるため、実行時間において発生したオーバーヘッドは避けることのできない処理である。このオーバーヘッドをより小さくするには、ローカルへの Pre-commit とリモートへの Pre-commit をスレッドによって並列化するなどの手法を用いる必要がある（現在はローカルに Pre-commit した後、リモートサイトに Pre-commit を発行しているため、Pre-commit 処理が直列化されている）。

3.6 その他の PostgreSQL 関連レプリケーションプロジェクト（参考）

1. PGReplicate
   http://www.csra.co.jp/~mitani/jpug/pgreplicate/
2. PGReplicator
   http://pgreplicator.sourceforge.net/
3. Postgres-R
   http://gborg.postgresql.org/project/pgreplication/projdisplay.pl
4. Usogres
   http://usogres.good-day.net/
5. eRServer
   http://www.erserver.com/

4 今後の展望

二相コミットを利用した分散データベースの機能として、今後、以下のような機能を実装していく予定である。

4.1 分散トランザクション

分散トランザクションは、複数のデータベースにまたがる問い合わせ処理を透過的に行う仕組みである。

例えば、データベース A とデータベース B が、それぞれネットワークで接続された異なるノード上に存在している場合に、

```
begin;
select uid from account@A where
                    fullname='Satoshi Nagayasu';
update billing@B set bill=20000 where uid=501;
commit;
```

のようにトランザクションを発行し、複数のデータベースをまたぐトランザクションの ACID 属性を保ちつつ処理することができる。

また、複数のデータベースのカラム内によって join を行う

```
select a.fullname from account@A as a,
                        billing@B as b where
                   a.uid=b.uid and b.bill>20000;
```

のようなトランザクションを発行する機能を実装する予定である。

4.2 リカバリ処理

4.2.1 二相コミットによる一貫性のリカバリ

二相コミットを用いるメリットは、トランザクションログへの記録によるリスタート時のリカバリ処理にある。

参加者・調停者のどちらかにおいて commit 発行後に障害が発生した場合、pre-commit 時のログの記録を元に、リスタート時に参加者あるいは調停者が相手に対して再度 commit の実行を命令する。このことによって、仮に障害が発生しても、リスタート時には一貫性のリカバリ処理が行われ、一貫性を保った状態で運用を再開することができるのである。

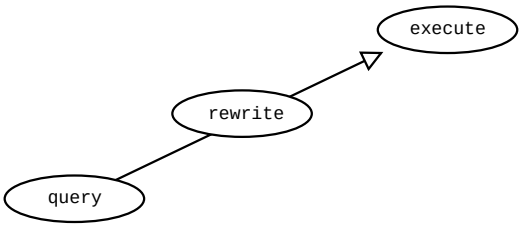
4.2.2 新規追加などによるノードの再同期

負荷分散のためのスケーラビリティの確保を目的とする場合、あるいは可用性を高めるためには、障害の起こったノードの切り離し、およびノードの追加とデータベースの再構築など、既に運用中のデータベースクラスタとの再同期が必要となる。

4.3 テーブルパーティショニング

PostgreSQL には、クライアントから受け取ったクエリをある手順に基いて書き換える「rewrite rule」というメカニズムが実装されている。

例えば「view」は、実際のテーブルの構造を異なる構造に変換することで利便性を提供するが、これも PostgreSQL ではルールシステムを用いて実装されている。「view」を実現するための書き換えのルールを定義することによって、あたかも実際のテーブルであるかのように view にアクセスすることができるようになる。



また、「view」を直接的には更新することはできないが、これも書き換えルールを UPDATE や INSERT に適用することによって、テーブルへの更新をあたかも view への更新であるかのように扱うことができるようになる。

異なるノードに対してそれぞれ異なる更新処理を、atomic かつ consistent に行うことができるようになると、「rewite rule」を応用することで、特定のテーブルを primary key などの値に応じて別々

のノードに分割保存し、読み出す時には一つのテーブルとして読み出す、といったことが可能となる。これは「テーブルパーティショニング」と呼ばれる機能である。

テーブルパーティショニングが実現すると、例えば一つのファイルシステムには入り切らないほど大きなデータを複数に分割することが可能になったり、問い合わせの負荷が非常に大きいテーブルなどを複数に分割することで、特定ののテーブルに対する負荷分散を実現することができるようになる。この機能の実装を行う。

5 最後に

今回は、「二相コミットプロトコル」という分散トランザクションの基本要素を、「同期レプリケーション」という機能とともにPostgreSQLへ実装し、その評価を行なった。このことにより、複数のノードにまたがったトランザクションの一貫性を保つことが可能となった。

今後は、今回の評価を通じて明らかになった実装上のボトルネックを解消するとともに、実際のアプリケーションや運用において有用な機能の実装を行なっていく予定である。

6 謝辞

本研究は、情報処理振興事業協会（IPA）の「未踏ソフトウェア創造事業（ユース）」のサポートを受けています。

検証環境を提供していただいたオープン・ソース・デベロップメント・ラボ（OSDL）、ならびに利用にあたって多大な協力をいただきましたスタッフの方々に感謝いたします。

最後に、日本PostgreSQLユーザー会・分散トランザクション開発分科会のメンバーの方々には、日頃から非常に有意義な議論やアドバイスをいただきましたことを感謝いたします。

参考文献

[1] M.Tamer Özsu, Patrick Valduriez; Principles of Distributed Database Systems Second Edition; 1999; Prentice Hall

[2] PostgreSQL Global Development Group; PostgreSQL オフィシャルマニュアル; 2002; インプレス

[3] ジム・グレイ, アンドレアス・ロイター; トランザクション処理 概念と技法 (上/下); 2001; 日経BP 社

[4] 石井達夫; PostgreSQL 完全攻略ガイド; 2001; 技術評論社

[5] 杉田研治; オープンソース系データベース「PostgreSQL」の機能と実装; インターフェース, 2002/10; CQ 出版社

[6] 穂鷹良介; データベースシステムとデータモデル; 1989; オーム社