トランザクション分離性を考慮した データベース高速化ミドルウェアに関する研究

Database Middleware for High Performance Transaction Processing with Isolation

2011年2月

堀井 洋

トランザクション分離性を考慮した データベース高速化ミドルウェアに関する研究

Database Middleware for High Performance Transaction Processing with Isolation

2011年 2月

早稲田大学大学院　基幹理工学研究科
情報理工学専攻　並列・分散アーキテクチャ研究

堀井 洋

目次

図目次

表目次

第1章　序論

本研究では，ミドルウェアを介してアプリケーションがデータベースにアクセスすることで，アプリケーションやデータベースを修正せずに，性能を向上させる手法を，研究対象とする．

1.1　研究の背景

データベースを利用して動的なデータを扱うサービスの処理量は，近年，増加の一途を辿っている．例えば，証券取引の取引高，ネットショッピングの売上高は，10年で10倍以上になってきている．また，近年普及している，Twitter等のConsumer Generated Mediaでも，扱うデータ量に伴い，処理量も増えてきている．

上記のような，動的なデータを扱うサービスでは，一般的には，オンライン・トランザクション処理システムを用いて，データが照会・更新される．オンライン・トランザクションシステムでは，アプリケーション・サーバが，ユーザからの処理要求を受信し，要求された処理に伴うトランザクション処理をデータベースに対して要求する．例えば，ネットショッピングの場合，アプリケーション・サーバはユーザから商品の購入要求を受信し，商品の在庫数の照会，在庫数の更新，注文履歴の追加を行うトランザクションを，データベースに要求する．データベースにおけるトランザクション処理では，原子性・分離性が保障され，アプリケーション・サーバは，一貫性のあるデータ照会・更新が可能となる．

アプリケーション・サーバと比較し，データベース・サーバは，1台の処理性能を向上させることが重要である．一般的に，オンライン・トランザクション処理システムは，アプリケーション・サーバ，データベース・サーバを多く配置し，アプ

リケーション・サーバはユーザからの要求を，データベース・サーバはアプリケーション・サーバから要求されるトランザクションを，並列に処理することで，性能を向上することが可能である．このうち，アプリケーション・サーバは，永続的なデータを保持しないため，安価なサーバを多く配置させることで，容易に処理量を増加させることが可能である．しかし，永続的なデータを保持するデータベース・サーバは，可用性の高い，高価なサーバを配置させる必要がある．そのため，システムのコストを最小限にするためには，少ないデータベース・サーバで，処理量を最大化させることが重要である．

データベース・サーバの処理性能を向上させる手法としては，アプリケーションが要求するトランザクションのワークロードに特化したデータベース・サーバを構築する方法が考えられる．例えば，近年提案されている Key-Value Store では，SQL を用いた複雑な照会・更新処理ができない反面，単純な照会・更新処理を短時間で大量に処理することが可能である．しかし，このようなデータベースを利用するには，アプリケーションの修正や，既存のデータベース構成の修正が必要となる．多くの企業では，関係データベースを用いたパッケージソフトウェアや，数多くの既存アプリケーションの利用が求められるため，アプリケーションの修正が必要な新規データベース・サーバを構築することは，困難である．また，既存のデータベースは，有用な管理ツールが存在しており，運用コストも低いため，継続的な利用が望まれる．

そこで本論文では，データベース・ミドルウェアを用いて，データベース・サーバの処理性能を向上させる手法を，研究対象とする．データベース・ミドルウェアを用いたトランザクション処理システムでは，アプリケーション・サーバは，データベース・ミドルウェアをデータベースと想定し，トランザクション処理を要求する．トランザクション処理を要求されたデータベース・ミドルウェアは，トランザクションの原子性，分離性を保ちながら，汎用データベースを利用して，要求されたトランザクション処理と同等の処理を実現する．例えば，複製を用いるデータベース・ミドルウェアでは，複数のレプリカに対して各更新を処理要求することで，全

てのレプリカに同じデータを保存し，1つのレプリカ・データベースに対して各照会SQLを処理要求することで，アプリケーションに対して，高い照会処理性能を提供する．また，バッチ処理を用いたデータベース・ミドルウェアでは，複数の照会・更新処理を1つの処理にまとめてデータベースに処理要求することで，データベースとの通信回数を減少させ，かつ，データベース内でのディスクアクセスを集約させることにより，アプリケーションに対して，高い照会・更新処理性能を提供する．

従来，データベース・ミドルウェアを用いて，データベースの処理性能を向上させる処理手法は，多く提案されているが，厳密なトランザクションの分離性を提供するデータベース・ミドルウェアは存在しない．例えば，複製を用いたデータベース・ミドルウェアでは，あるトランザクションが照会中のデータに対して，他のトランザクションが更新可能な分離性（スナップショット分離性）が，多く提案されている．また，バッチ処理を用いたデータベース・ミドルウェアでは，複数のトランザクションにまたがった処理をまとめながら，各トランザクションの分離性を保障するデータベース・ミドルウェアは，存在しない．つまり，既存のデータベース・ミドルウェアでは，銀行用のアプリケーションのような，厳密なトランザクション分離性（繰り返し読み取り可能分離性やカーソル安定分離性）を要求するアプリケーションに，利用することが困難である．

1.2 研究の目的

本研究の目的は，厳密なトランザクション分離性を保障しながら，様々なアプリケーションのワークロードの性能向上を行う，データベース・ミドルウェアを提供することとする．そのため，本研究では，主に次の2つのデータベース・ミドルウェアの提案を行う．

- データベースの排他制御機構を用いて，更新SQLを複数台のレプリカ・データベースに複製し，照会SQLを1台のレプリカ・データベースで処理要求する手法では，厳密なトランザクション分離性を保障することが困難である．こ

の問題を解決するため，データベース・ミドルウェア内で，要求されるトランザクション分離性の提供に必要なロックを特定，管理することで，レプリカを用いたトランザクション処理を行うデータベース・ミドルウェアを提案する．本データベース・ミドルウェアを用いることで，アプリケーションに対し，高い照会処理性能を提供することを可能とする．

- 動的にSQLが生成されるアプリケーションは，生成されたSQLの中から，バッチ対象とするSQLを特定することが困難であった．この問題を解決するために，トランザクション内のSQL処理履歴を解析し，厳密なトランザクション分離性を保障しつつ，バッチ処理可能なSQLを，動的に決定し，バッチ処理を行うデータベース・ミドルウェアを提案する．本データベース・ミドルウェアを用いることで，アプリケーションに対し，高い更新処理性能を提供することを可能とする．

本論文では，提案するデータベース・ミドルウェアを，既存のウェブ・アプリケーション・ベンチマーク（Daytrader，TPC-W）を用いて評価する．本評価により，提案するデータベース・ミドルウェアが，アプリケーションのワークロードに特化したトランザクション処理性能を，アプリケーションを修正することなく，提供可能であることを示す．

1.3 本論文の構成

本論文では，まず，第1章にて，近年のデータベースを用いたアプリケーションの特徴的なワークロードを示し，特定のアプリケーションに特化してトランザクション性能を向上するデータベース・ミドルウェアの重要性を示す．

第2章では，各トランザクション内のSQL処理履歴を記憶し，トランザクション分離性を保障しながら，バッチ処理可能なSQLを特定する手法を提案する．本手法は，複数のトランザクション内で要求されるSQLをバッチ対象とするため，各トランザクション内で多くのSQLを処理しないオンライン・トランザクション処理（OLTP）

システムでも，効果的である．本章では，本提案手法を，JDBC インタフェースを実装するライブラリとして実装し，Ｊ２ＥＥ上で実装されたベンチマーク，Daytrader を用いて，性能向上の効果を示す．

第 3 章では，データベース・ミドルウェア内で排他制御を行い，トランザクション分離性を保障しながら，複製を用いて照会処理性能を向上するデータベース・ミドルウェアを提案する．本手法におけるデータベース・ミドルウェア内での排他制御では，要求される SQL を解析することで行うため，アプリケーションが特定のヒントを提供する必要はない．本章では，本提案手法を，第 3 章と同様，JDBC インタフェースを実装するライブラリとして実装し，J2EE 上で実装されたベンチマーク，TPC-W を用いて，性能向上の効果を示す．

第 4 章では，提案，評価した，2 つのデータベース・ミドルウェアを有用性のまとめ，第 5 章では，今後の課題，展望を示す．

第2章　自動グループ・コミット手法

2.1　まえがき

様々な用途でデータベースが利用されてる現在，データベースは，大量の更新トランザクションを，高速に処理しなくてはならない．例えば，証券取引所における株式の売買システムでは，全ての取引をデータベースに記録する必要があるため，1秒間に数万件の更新トランザクション処理が発生する．また，IBM Websphere Process Server[26] では，BPEL プロセスの状態管理にデータベースを利用するため，プロセスの状態遷移ごとに更新トランザクション処理が発生する．さらに，比較的照会処理の割合が多いショッピングサイトでも，ウェブを介して多くの利用者が注文を行うため，絶対数として更新トランザクション処理が多くなる．

大量のトランザクションを高速に処理する手法として，従来，(1）グループ・コミット [1], (2）Multiple Query Optimization（MQO）[2, 3, 4] が，広く利用されている．

グループ・コミットを利用する手法では，複数のトランザクション処理を，1つのトランザクションとして処理することで，更新トランザクション処理の高速化を実現する．例えば，銀行の夜間バッチ処理では，大量のトランザクションを処理する際，トランザクションごとにコミット処理を行わず，全トランザクションの処理が終了した時点でコミット処理を行う．本手法を利用することにより，同時トランザクション処理数が少なくなるため，オペレーションシステム上のプロセス管理に対するオーバヘッドが減少する．また，複数トランザクション分のコミット処理を一度に行うため，コミット処理に対する処理量が減少する．

MQO を利用する手法では，グループ・コミットする複数トランザクション内の

複数の更新処理を，1 つの更新処理として処理することで，更新トランザクション処理の高速化を実現する．例えば，パラメータを指定する SQL を用いて更新処理を行うデータベースでは，一度に複数のパラメータ値を受信することで，複数の更新処理を行うことが可能である．本手法を利用することにより，データベースにおける，パラメータの受信処理，処理結果の送信処理が軽減される．なお，MQO は，照会 SQL をまとめて処理する手法にも適用可能なため，本章では，上記のような更新 SQL をまとめて処理する手法を，バッチ更新と呼ぶものとする．

上記のような更新トランザクション処理の高速化を実現するには，グループ・コミットの対象となる更新トランザクションを特定し，かつ，それらの中でバッチ更新の対象となる更新処理を特定する必要がある．従来，各トランザクションの依存関係を事前に解析してグループ・コミットを行う手法 [5, 6] や，各トランザクションの原子性・分離性を保障せずに MQO を適用する手法 [2, 3, 4] は提案されている．しかし，利用者からのリクエストにより動的に各トランザクション間の依存関係が変化し，厳密な原子性・分離性が要求されるアプリケーションには，これらの手法を適用することはできない．また，各トランザクション間の依存関係を熟知したアプリケーション開発者が，グループ・コミット，バッチ更新をアプリケーションに適用することも可能だが，現在のパッケージソフトウェアや，アプリケーション・サーバ等のミドルウェアが普及した現在では，実現が困難である．

そこで，本章では，各トランザクションで処理要求される SQL を予測し，予測結果からグループ・コミットの対象となるトランザクション，バッチ更新の対象となる更新処理を特定し，自動的に適用する手法を提案する．

本手法では，アプリケーションがあるトランザクションにおける最初の更新 SQL をデータベースに処理要求した時点で，過去のコミット済みトランザクションの実行履歴を元に，今後同じトランザクション内で処理要求される可能性がある SQL を予測する．次に，同時に予測したトランザクションの中から，予測結果中に最も多くバッチ更新可能な SQL 含むトランザクションの集合を特定し，グループ・コミットの対象とする．グループ・コミット対象のトランザクション内で処理要求される

SQL は，バッチ更新の対象可能である場合，ミドルウェアによって，新たなバッチ更新用トランザクションとして，データベースに処理要求される．これらのバッチ更新，グループ・コミットは，アプリケーションが要求するトランザクション間の分離性（カーソル固定分離レベル）を保証するように，ミドルウェアによってスケジューリングされながら処理される．

我々は，本手法を，データベースに接続するためのミドルウェアとして，Java を用いて実装した．本実装を利用することで，アプリケーションや，データベースを修正することなく，グループ・コミット，バッチ更新を用いて，更新トランザクションの最適化を実現可能となる．本章では，本実装を，簡易的なベンチマークを用いて評価し，その有用性を示す．

2.2 システム構成と課題

2.2.1 システム構成とトランザクション処理

本章では，エンド・ユーザからの処理要求に応じて，アプリケーションがデータベースにトランザクション処理を要求するシステムを想定する．本システムでは，同時に複数のエンド・ユーザがアプリケーションに対して処理要求を行うものとし，同様に，アプリケーションもデータベースに同時に複数のトランザクション処理要求を行うものとする．データベースをサーバとした場合，アプリケーションはデータベースのクライアントとなるため，本章ではアプリケーションをクライアントと表記する．なお，各クライアント，データベースは，別のサーバで実行され，ネットワークを介して通信可能とする．

トランザクション処理は，クライアントがデータベースに対して，複数の SQL を処理要求した後，コミット処理，もしくは，ロールバック処理を要求することで，実現されるものとする．また，SQL の処理は，引数を指定可能な SQL（SQL テンプレート）を利用して要求されるものとする．例えば，以下のような SQL の処理を，トランザクションとして要求する場合，クライアントは，図 2.1 のように実行する．

```
1  Connection conn = DriverManager.getConnection(xxx);
2  PreparedStatement stmt = conn.prepareStatement
                    ("INSERT INTO CART VALUES(?,?)");
3  stmt.setString(1,book1);
4  stmt.setString(2,u1);
5  stmt.executeUpdate();
6  conn.commit();
```

図 2.1: Java のトランザクション処理

なお，図 2.1 では，Java を用いて記述され，JDBC ライブラリを利用してデータベースに処理要求を行うものとする．

- INSERT INTO CART VALUES(book1,u1)

図 2.1 では，まず，データベースへの接続を試み（1 行目），2 つのパラメータを指定する以下のような SQL テンプレート I1 を，データベースに登録する（2 行目）．

- I1: INSERT INTO CART VALUES(?,?)

次に，2 つのパラメータ（$book1$，$u1$）を指定し（3，4 行目），データベースに送信する（5 行目）．最後に，コミット処理を要求することで，データベースに対し，指定した SQL の処理を確定させる（6 行目）．なお，6 行目の処理が正常処理した場合，トランザクションに対するコミット処理が正常終了したことを表し，例外が発生した場合，トランザクションがロールバックしたことを表す．

図 2.1 では，パラメータとして指定する値を変更することで，以下のような更新 SQL を，同じ SQL テンプレートを利用しながら，トランザクションの処理内容を変更することが可能である．

- INSERT INTO CART VALUES(book1,u1)
- INSERT INTO CART VALUES(book2,u2)
- INSERT INTO CART VALUES(book3,u3)

```
1   Connection conn = DriverManager.getConnection(xxx);
2   PreparedStatement stmt = conn.prepareStatement
                    ("INSERT INTO CART VALUES(?,?)");
3   stmt.setString(1,book1);
4   stmt.setString(2,u1);
5   stmt.addBatch();
6   stmt.setString(1,book2);
7   stmt.setString(2,u2);
8   stmt.addBatch();
9   stmt.setString(1,book3);
10  stmt.setString(2,u3);
11  stmt.addBatch();
12  stmt.executeBatch();
13  conn.commit();
```

図 2.2: Java のバッチ更新とグループ・コミット

ここで，上記 3 つの更新 SQL を別々に処理するトランザクションを，グループ・コミット，バッチ更新を利用する場合を考える．この場合，図 2.2 のように，クライアントは，データベースにトランザクション処理を要求することになる．

図 2.2 では，まず，図 2.1 と同様，データベースへの接続を試み（1 行目），I1 を，データベースに登録する（2 行目）．次に，1 つ目のトランザクションのパラメータ値である $u1$ $book1$ を登録し（3，4 行目），バッファリングする（5 行目）．同様に，残り 2 つのトランザクションのパラメータ値である $u2$ $book2$，$u3$ $book3$ も，同様に登録し，バッファリングする（3-11 行目）．最後に，バッファリングしたパラメータ値をデータベースに送信して 3 つの更新 SQL 処理をデータベースに要求し（12 行目），トランザクションのコミット処理を要求する（13 行目）．

2.2.2 自動グループ・コミットの課題

一般的なアプリケーションは，図 2.1 のように記述されているため，アプリケーションがグループ・コミット，バッチ更新を利用する場合は，図 2.2 のようにアプリケーションを修正しななくてはならない．一方，パッケージアプリケーションや，運用中のアプリケーションは，アプリケーションの修正を行うことができない．

本章では，ミドルウェアとして，グループ・コミット，バッチ更新の適用を目的とする．具体的には，JDBC ライブラリ内部でミドルウェアが機能するシステムを，実現する．つまり，アプリケーションは図 2.1 のように処理しながら，データベースは図 2.2 で要求されるように SQL，トランザクションの処理要求を受けることを目的とする．

ライブラリ内部でグループ・コミットを行う場合，あるトランザクションが更新 SQL を要求時には，トランザクションと同時にグループ・コミットするトランザクション集を特定する必要がある．より多くの更新 SQL をバッチ更新することで，データベースの負荷を軽減することが可能になるため，適切にトランザクション集を特定する必要がある．

また，特定したトランザクション集の，各々の原子性，分離性を保ちながら，バッチ更新をスケジューリングするアルゴリズムも必要となる．トランザクション処理の一般的な分離性では，あるトランザクションが更新中のデータに関しては，他のトランザクションは照会してはならない．しかし，グループ・コミットを利用すると，データベースにとっては，グループ・コミット対象のトランザクション集は 1 つのトランザクションとして処理されるため，トランザクション集の間では，上記分離性が保障されない．つまり，ミドルウェア内では，アプリケーションが要求する分離性を保証しながら，グループ・コミット対象のトランザクション集を特定し，バッチ更新を処理要求する必要がある．

なお，本章では，トランザクションの分離性として，カーソル安定性の分離レベルをアプリケーションが要求するものとする．本分離レベルが分離性として指定されると，データベースは，(1) あるトランザクションが更新中・照会中（カーソル上）のレコードは他のトランザクションは更新できず，(2) あるトランザクションが更新中のレコードは他のトランザクションは照会できないことを保障する．

2.3 提案手法

グループ・コミット，バッチ更新をトランザクション処理に適用する手法を，ミドルウェアとして実現する手法を提案する．なお，アプリケーションは，JDBC ライブラリを利用してデータベースにトランザクション処理を要求するものとし，提案するミドルウェアは，JDBC ライブラリ内部で機能するものとする．

2.3.1 概要

提案するミドルウェアの概要を，図 3.1 に示す．

図 3.1 のように，本ミドルウェアは，JDBC ライブラリの API をアプリケーションに提供する．そのため，JDBC ライブラリを利用する全てのアプリケーションは，修正することなく本ミドルウェアを利用することが可能である．

アプリケーションから JDBC ライブラリの API を通じて要求される処理は，TCP/IP を通じて，ミドルウェア・サーバに通知される．要求された処理に応じて、ミドルウェアは，データベース・ベンダーが提供する JDBC ライブラリを利用して，データベースにトランザクション処理を要求する．データベース・ベンダーが提供する JDBC ライブラリを利用するため，データベースに依存することなく，本ミドルウェアを利用することが可能である．

ミドルウェア内では，複数のアプリケーション・サーバ内のアプリケーションが要求するトランザクション（Tx）ごとにトランザクション管理機構（TxHandler）が対応付けられる．各 TxHandler は，アプリケーションから SQL・コミット・ロールバックの処理要求を受信し，グループ・コミット判定機構（MatchMaker），スケジューラ（Scheduler）と連携して，バッチ更新・グループ・コミットを適用する．なお，ミドルウェア内で，MatchMaker は 1 つ，Scheduler は複数存在する．

各 TxHandler は，トランザクション中，最初にバッチ更新可能な SQL を受信した際，MatchMaker に対して 1 つの Scheduler を要求する（requestMatchMake）し，1 つの Scheduler を対応付けられる．本ミドルウェアでは，以下の SQL が，バッチ更

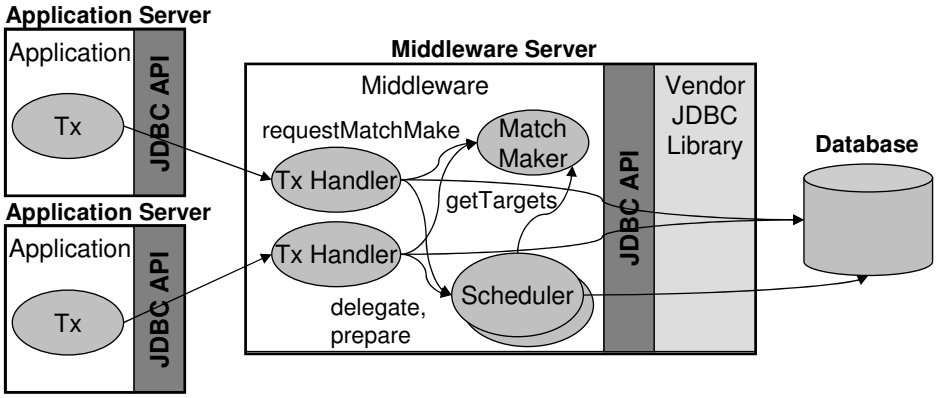

図 2.3: 全体の構成（ミドルウェア・サーバ 1 台時）

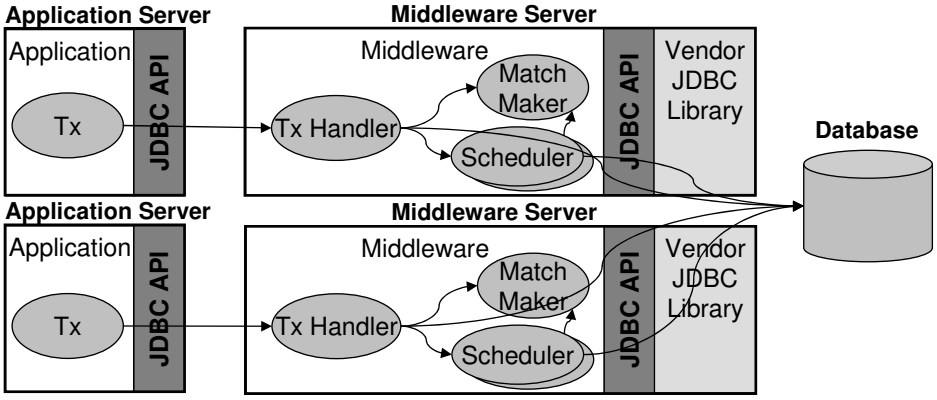

図 2.4: 全体の構成（ミドルウェア・サーバ複数台時）

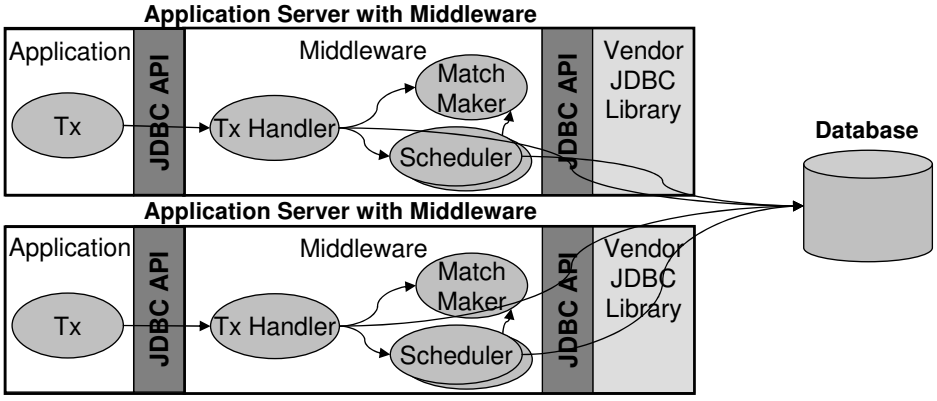

図 2.5: 全体の構成（アプリケーション・サーバ内で稼動時）

新対象の SQL となる．

- 排他ロックを伴う SQL（更新 SQL や FOR UPDATE 句を含む照会 SQL）
- 同一トランザクション内の更新を照会する SQL（図 2.8 の checkConflict を用いて特定）

Scheduler が対応付けられた TxHandler は，アプリケーションよりバッチ更新可能な SQL を受信した際，その SQL は全て Scheduler 経由で処理要求し（delegate），それ以外の SQL は直接データベースに処理要求する．また，アプリケーションよりコミット要求を受けた場合，Scheduler に対しコミット要求を行い（prepare），データベースにも直接コミット要求を行う（ロールバックも同様）．なお，TxHandler は，トランザクションに要求された SQL テンプレートと引数値を記録し，分離レベルを保障するために利用する．

各 TxHandler は，最初に上記 SQL を受信した際，グループ・コミット判定機構（MatchMaker）に対し，スケジューラ（Scheduler）を要求し（requestMatchMake），1 つの Scheduler を対応付けられる．その後，TxHandler は，特定した SQL を Scheduler 経由で（delegate），それ以外は直接データベースに送信する．また，トランザクションよりコミット要求を受けた場合，Scheduler に対しコミット要求を行い（prepare），データベースにも直接コミット要求を行う（ロールバックも同様）．なお，TxHandler は，トランザクションに要求された SQL テンプレートと引数値を記録し，2.3.5 での競合検出に利用する．

Scheduler は，複数のトランザクションに対応付けられ，delegate された SQL を，1 つのトランザクションとして，データベースに処理要求する．本章ではこのトランザクションを，グローバル・トランザクションと呼ぶものとする．Scheduler は，対応付けられた全 TxHandler が prepare を呼び出した時点で，グローバル・トランザクションのコミットをデータベースに要求し，対応付けられた全 TxHandler のトランザクションにコミット終了を通知する．TxHandler からロールバックを受信した場合は，グローバル・トランザクションをロールバックし，対応付けられた全 TxHandler

のトランザクションにロールバックを通知する．Scheduler は，グローバル・トランザクション終了後，TxHandler との対応を解除し，次のグループ・コミット対象トランザクション集を要求する（getTargets）．

本ミドルウェアでは，データベースがカーソル安定性を保障していることを想定し，同等の分離レベルを保障する．Scheduler からのグローバル・トランザクション，TxHandler からのトランザクション間の分離レベルは，データベースの排他制御機構を利用して保障される．また，同じ Scheduler 経由で更新を行うトランザクション間の分離レベルは，ミドルウェア内で競合検出を行うことで保障される．

なお，本ミドルウェアは，図 2.4 に示すように，複数のミドルウェア・サーバで稼動しても良い．また，図 2.5 に示すように，ミドルウェアをアプリケーション・サーバ内で稼動させ，TCP/IP を経由せず，直接ミドルウェアに JDBC の処理要求を通知する構成も可能である．

2.3.2 実行される SQL の予測

本ミドルウェアでは，requestMatchMake を要求した各 TxHandler が，今後トランザクション中で処理要求する可能性がある SQL の予測を行う．本予測を基に，MatchMaker がグループ・コミットの対象となるトランザクション集を，Scheduler がバッチ更新の対象となる更新 SQL を特定する．

予測の前段階として，本ミドルウェアでは，まず，各トランザクションが実行する SQL テンプレートのシーケンス（TxSeq）を監視し，その履歴を保存する．TxSeq は，トランザクションが要求した SQL テンプレートのリストで構成され，アプリケーションでは同じ TxSeq が何度も処理される．例えば，トランザクションが以下の SQL テンプレート（S は Select，I は Insert，U は Update）を利用する場合，「S1 → I1 → U1 → C」という TxSeq となる．

- S1：SELECT ID FROM USER WHERE N=?
- I1：INSERT INTO CART VALUES(?, ?)

- U1：UPDATE ITEM SET(STOCK=?)

多くのアプリケーションでは，同じ照会 SQL テンプレートを複数の TxSeq で多用するが，更新 SQL テンプレートは少数の TxSeq しか利用しないことが多い．これは，アプリケーションがデータベースの一貫性を維持する必要があるため，開発効率の観点から，同じ種類の更新を 1 つの TxSeq によって記述した方がよいことが原因と考えられる．例えば，実アプリケーションを考慮して作成されたベンチマーク，TPC-W や DayTrader では，同じ更新 SQL テンプレートを利用する TxSeq は，最大で 2 つしか存在しない．本性質を利用すると，あるトランザクションの最初の更新 SQL テンプレートと TxSeq の履歴から，そのトランザクションが次に要求する SQL テンプレートを高精度で予測することが可能である．

TxSeq の履歴が十分保存された後，ミドルウェアは，アプリケーションがトランザクション中で最初のバッチ更新対象となる SQL を要求した時点で，TxSeq の履歴を基に，今後トランザクション内で実行されうる SQL テンプレートを予測する．予測は，予測対象のトランザクション内で処理要求された SQL テンプレートと，同じ順番の SQL テンプレートを利用した TxSeq を履歴から検索し，その結果をトランザクションが取りうる TxSeq の候補（TxSeqCandidate）とすることで，予測する．例えば，以下の TxSeq が履歴に記憶されており，実行中のトランザクションの TxSeq が「S1 → I1」の場合，TS1，TS2，TS3 が TxSeqCandidate となる．また，トランザクション中の処理が進み，TxSeq が「S1 → I1 → U1 → U2」となった場合，TS2 が TxSeqCandidate となる．

- TS1: S1 → I1 → U1 → C
- TS2: S1 → I1 → U1 → U2 → C
- TS3: S1 → I1 → U1 → U3 → I2 → C

```
1  List<Tx> getTargets() {
2    Tx baseTx = [requestMatchMake を要求している Tx の 1 つを選択];
3    List<Tx> ret = new ArrayList<Tx>();
4    List<TxSeqCandidate> baseCands=[baseTx の全 TxSeqCandidate を取得];
5    for (Tx tgtTx: [requestMatchMake を要求している baseTx 以外の全 Tx を取得]){
6      List<TxSeqCandidate> tgtCands=[tgtTx の全 TxSeqCandidate を取得];
7      if ([baseCands と tgtCands 内に，必ず衝突する SQL テンプレートが存在するか])
8        continue;
9      if ([baseCands と tgtCands の TxSeqCandidate の組み合わせで,
               共通 SQL 更新テンプレートを持つ組み合わせが 1 つ以上存在するか])
10       ret.add(tgtTx);
11   }
12   if (ret.size() >= [最大グループ・コミット Tx 数])
13     [baseTx が最も頻繁に実行する TxSeqCandidate と最も類似している TxSeqCandidate を
       持つ Tx を最大グループ・コミット Tx 数残し，他 Tx を ret から削除];
14   return ret;
15 }
```

図 2.6: グループ・コミット対象トランザクションの特定するプログラム

2.3.3 グループ・コミット対象の特定

本ミドルウェアでは，MatchMaker が，グローバル・トランザクションを処理していない 1 つの Scheduler と，同時にバッチ更新対象の SQL を処理要求している複数の TxHandler を対応付けることで，グループ・コミットの対象となるトランザクション集を特定する．なお，TxHandler と Scheduler の対応は，Scheduler がデータベースに要求するグローバル・トランザクションがコミット，もしくは，ロールバックするまで，有効である．

Scheduler が対応されていない各 TxHandler は，対応する Tx より，バッチ更新対象となる SQL の処理要求を受信した際，MatchMaker に対して Scheduler の対応を要求する（requestMatchMake）．MatchMaker は，Scheduler との対応を要求中の TxHandler から，同じ SQL テンプレートを処理要求すると予測される複数の TxHandler を選択し，待機中の Scheduler を対応付ける．

TxHandler が対応付けられた Scheduler は，データベースに対してグローバル・トランザクションの処理要求を開始し，TxHandler からバッチ更新対象の SQL を受信

した場合は，開始したグローバル・トランザクション内の更新として，バッチ更新を行う．また，Scheduler は，対応付けられた全 TxHandler からコミット・ロールバック要求を受信した場合，グローバル・トランザクションのコミット・ロールバックの処理要求をデータベースに対して要求する．グローバル・トランザクションが終了した Scheduler は待機中となり，次の TxHandler の対応付けを待機する．

なお，1 つの Scheduler に対応付けられる TxHandler の数は，上限値（グループ・コミット上限値）が設定されているものとし，グループ・コミット上限値を超えた場合は，共通した SQL テンプレートが多く予測されたトランザクションを選択する．

上記 MatchMaker の処理方法の詳細を，Java を用いたアルゴリズムとして，図 2.6 に示す．なお，図 2.6 中の Tx，TxSeqCandidate は，それぞれ，トランザクション，TxSeqCandidate を表すクラスとする．また，[ ] の記述は，[ ] 中の述語を処理するものとする．

図 2.7 の 4 つのトランザクションを用いて，グループ・コミット上限値を 2 とした際の，グループ・コミット対象のトランザクション集を特定する手順を示す．なお，図 2.7 では，各ノードは各トランザクションがデータベースに要求する SQL テンプレート（S は Select，I は Insert，U は Update）とコミット（C）を表し，実線でつながる SQL テンプレートは実行要求済みの，破線でつながれた SQL テンプレートは要求が予想される SQL テンプレートを表す．コミットのノードには，そのノードにたどり着くトランザクション Candidate の名前が記されるものとする．また，図 2.6 の 2 行目の baseTx として，Tx1 が選択されたものとする．全 SQL は競合しないものとし（競合とその検出方法は，2.3.5 にて詳細を示す），TxSeqCandidate の類似性の高さは，共通の更新 SQL テンプレートの数で表されるものとする．

まず，TxSeqCandidate2，TxSeqCandidate3 が TxSeqCandidate1-1，1-2 内の U1，I1 を含むため，グループ・コミット対象となる．トランザクション数 3 は，グループ・コミット上限値 2 を越すため，1 つのトランザクションをグループ・コミット対象からはずす必要がある．そこで，TxSeqCandidate2 は TxSeqCandidate1-1 と 1 つの共通 SQL テンプレートを含んでいるのに対し，TxSeqCandidate3 は 2 つ含んでいるた

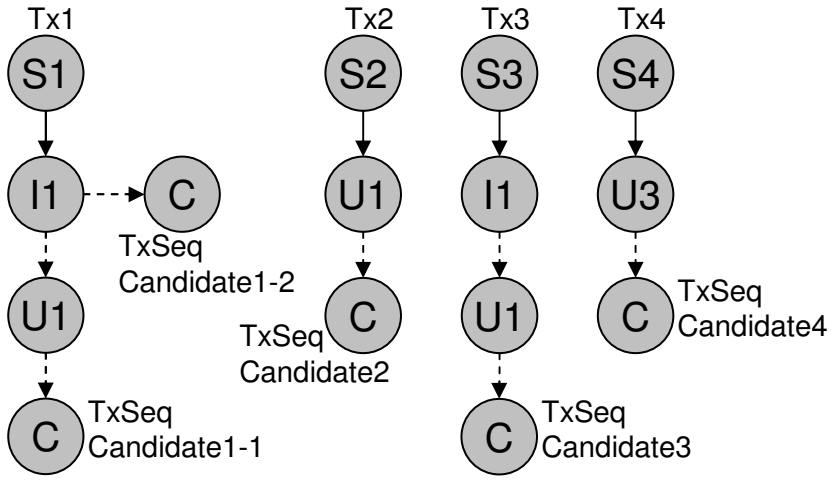

図 2.7: requestMatchMake を要求するトランザクション集

め，より類似性の高い Tx3 がグループ・コミット対象に残る．

2.3.4 バッチ更新，グループ・コミットのスケジューリング

MatchMaker により Scheduler と対応付けられたトランザクションの更新 SQL は，Scheduler によって，バッチ更新をスケジュールすることで，一度に実行する更新 SQL 数を最大化する．アルゴリズムの詳細を図 2.8 に示す．

本アルゴリズムは，Scheduler がグローバル・トランザクションを実行する関数（run）と，各 TxHandler が Scheduler に SQL を要求する関数（delegate），照会 SQL を直接実行可能か検証する関数（checkConflict），コミット可能か検証する関数（prepare）から成る．なお，checkConflict が true を返す場合，その引数の SQL は直接実行可能と判定され，それ以外は delegate 対象となる．また，Tx，SQL，SQLTemplate クラスは，トランザクション，実行可能な SQL，SQL テンプレートを示すクラスとする．これより，図 2.7 の Tx1，Tx2，Tx3 に Scheduler が対応した際の図 2.8 の挙動を，図 2.9 を用いて説明する．なお，全 SQL には競合はないものと考える．また，図 2.9 では，左からトランザクション，Scheduler，グローバル・トランザクションの実行

```
1  synchronized void run() {
2    [GTx を開始する];
3    while (![rollback を要求されている]&&![全 Tx が prepare 中]) {
4      if (![delegate 中の Tx が存在する]) { wait(); continue; }
5      for (Tx ownerTx : [delegate 中の Tx の集合を取得]) {
6        checkConflict(ownerTx, [ownerTx が delegate している SQL を取得]);
7        if ([rollback を要求されている]) {break;}
8        SQLTemplate U = [ownerTx が delegate している SQL のテンプレートを取得];
9        if (U == null||[delegate 中でない Tx が，U を delegate する可能性がある])
10         continue;
11       [U を delegate 中の Tx から引数を集め，バッチ更新を行う（照会の場合は逐次処理）];
12       [上記対象となった Tx を delegate 中から解除する];
13       notifyAll();
14     }
15     if ([rollback を要求されている]) {break;}
16     wait();
17   }
18   if ([rollback を要求されている]){[GTx を rollback する];}
19   else {[GTx を commit する];}
20   notifyAll();
21 }
22 synchronized boolean checkConflict(Tx tx, SQL sql) {
23   if (![sql は tx 以外から delegate された全 SQL と競合しない]
24     || ![sql は全 Tx が照会中のレコードを更新しない]) {
25     [GTx と全 Tx に rollback を要求する]; notifyAll(); return false;
26   } else { return [sql は tx から delegate された全 SQL と競合しない];}
27 }
28 synchronized void delegate(Tx tx) {
29   [tx を delegate 中に設定する];
30   notifyAll();
31   while (![rollback を要求されている] && ![sql が実行されている])
32     wait();
33   if ([GTx がロールバックした])[tx をロールバックする];
34 }
35 synchronized void prepare(Tx tx) {
36   [tx を prepare 中に設定する];
37   notifyAll();
38   while ([GTx が実行中である]) { wait();}
39   if ([GTx がロールバックした]) [tx をロールバックする];
40 }
```

図 2.8: グループ・コミットの実行

状況を示すものとする．グローバル・トランザクション，トランザクションの実行状況は，実行中，実行予定の SQL で表すものとし，実行済みの SQL は実線，実行予定の SQL は破線の円，実行された順番を矢印で表すものとする．なお，SQL をバッチ更新した場合は，円を重ねて表示するものとする．Scheduler の実行状況は，delegate していて，まだグローバル・トランザクションとして実行されていない SQL を，Scheduler のキューに記すことで，表すものとする．

Tx1，Tx2，Tx3 は，それぞれ，I1，U1，I1 を SQL テンプレートとする SQL の処理要求を行う際に，その SQL を引数として指定して，対応する TxHandler に対して delegate 関数を呼び出す（図 2.9-a）．TxHandler は，Tx から SQL を delegate された際，delegate を呼び出し，Tx を delegate 中にする（図 2.8-29 行目）．delegate された SQL は，Scheduler により，run 内で実行される．Scheduler による SQL の実行後，Tx は，delegate 中から解除され，アプリケーションに SQL の実行結果が返される．

run では，まず，delegate 中の Tx から 1 つの ownerTx を選択する（5 行目）．ここでは，I1 の SQL を要求している，Tx1 を選択した場合を想定する．次に，checkConflict で，ownerTx の SQL と他の Tx が実行した SQL との競合の有無を検証する．ここでは，全 SQL は競合しないと仮定しているため，Scheduler は次の処理に移行する（もし競合が検出された場合は，分離性を保てない場合があるため，グローバル・トランザクション，グローバル・トランザクションの Scheduler に対応付けられた全 TxHandler の Tx をロールバックする（25 行目））．

次に，I1 を delegate 中ではない Tx が実行する可能性があるか調べる（9 行目）．この時点で対応する全 Tx（Tx1，Tx2，Tx3）が delegate 中なので，対象となる Tx は存在しない．そこで，Scheduler は，I1 を SQL テンプレートとする SQL を delegate している Tx1 と Tx3 の SQL をバッチ更新し（11 行目），delegate 中の状態を解除する（12 行目）．

次に，ownerTx として，Tx2 を選択したとする（5 行目）．Tx2 は U1 を SQL テンプレートとする SQL を要求しているが，delegate 中でない Tx1 が U1 を SQL テン

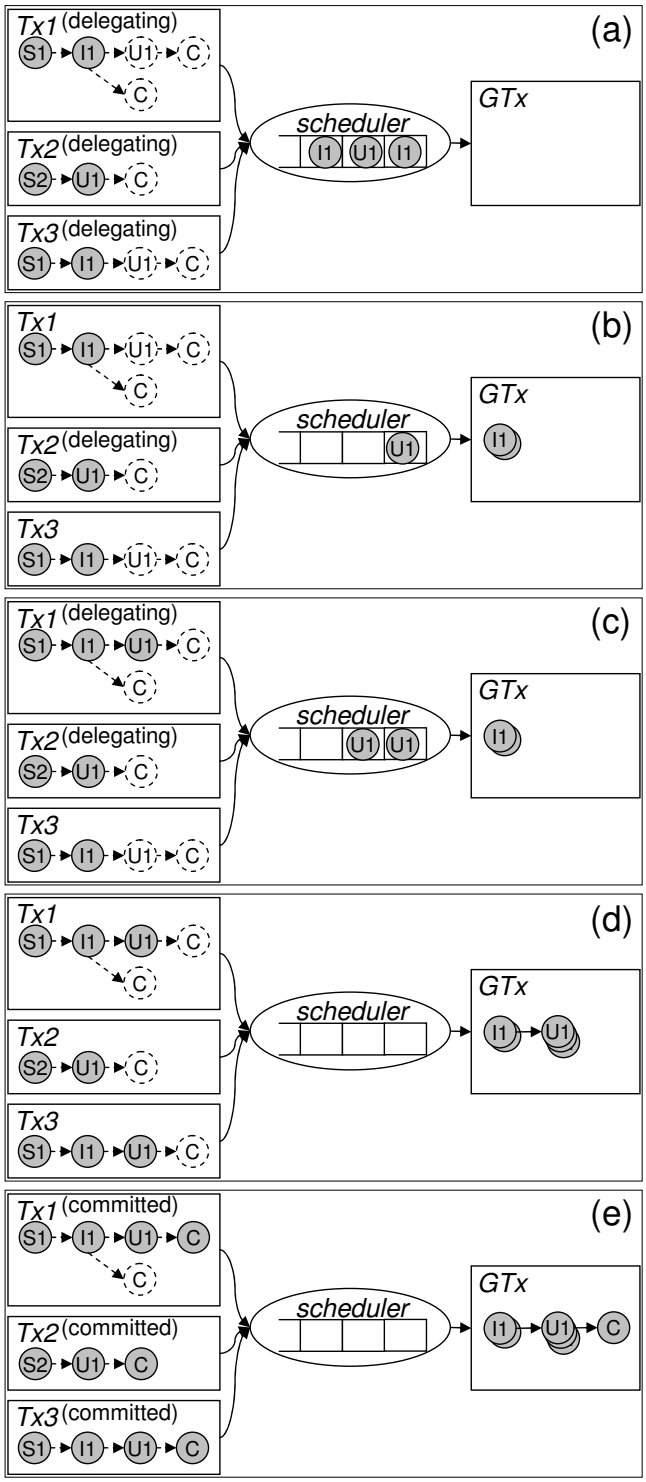

図 2.9: 図 2.7 の Tx1，Tx2，Tx3 に対するグループ・コミット

プレートとするSQLをTxSeqCandidate1-1内で要求する可能性があるため，バッチ更新しない（10行目）．最後は，Tx3をownerTxとして選択するが，すでにTx1がownerTxの際にTx3がdelegateしていたSQLは実行したので，8行目ではnullが返る．上記のようにdelegateされた全SQLのバッチ更新を試みた後，Tx1，Tx3はアプリケーション・ロジックの実行を再開し，Schedulerは，Tx1，もしくは，Tx3が次のSQLをdelegateするまで待機する（16行目）（図2.9-b）．

次に，Tx1がU1をSQLテンプレートとするSQLを要求し，delegate中になり（29行目），Schedulerに通知した場合を想定する（30行目）．通知を受けたSchedulerは，再度Tx1，Tx2が要求するSQLの実行を試みるが，Tx3がU1のSQLを要求する可能性があるため，バッチ更新せず，Tx3のdelegateを待機する（図2.9-c）．Tx3がU1のSQLの実行を要求した際は，SchedulerはTx1，Tx2，Tx3が要求するSQLをバッチ更新し，Tx1，Tx2，Tx3はアプリケーション・ロジックを再開する（図2.9-d）．

各TxHandlerはTxからコミット要求を受けた際，prepareを呼び出す．prepareでは，Txをprepare中とし，runが終了するまで待機する．Schedulerに対応付けられた全TxHandlerのTxがprepareを呼び出した際，runは終了し，グローバル・トランザクションはコミットを行い（19行目），待機中のTxHandlerに通知を行う（20行目）（図2.9-e）．

2.3.5 競合の判定方法

複数Txの更新を1つのグローバル・トランザクションとしてデータベースに処理させると，未コミットの更新を他Txで照会・更新する可能性がある．提案手法では，図2.6，2.8で示したアルゴリズムにおいて，ミドルウェア内で実行するSQLの競合（Tx間のread-write，write-read，write-writeロック競合）が発生する可能性をSQLの記述から判定し，競合するTxを異なるグローバル・トランザクションに配置して未コミット更新の照会を防ぎ，競合が防げない場合はTxをロールバックする．

図2.6では，SQLテンプレート間の競合判定を行う．以下のような，必ず競合す

る SQL テンプレートを検出し，競合する Tx は異なるグローバル・トランザクションとして処理する．

- DELETE FROM t1
- UPDATE t1 SET c1=? WHERE id=?

また，以下のように，引数値がわからないと競合を判定できない場合は，図 2.6 では競合として判定しない．

- DELETE FROM t1 WHERE id=?
- UPDATE t1 SET c1=? WHERE id=?

一方，上記の競合は，引数値が指定される図 2.8 の checkConflict では検出される．なお，図 2.8 では，以下のように，引数値がわかっても，SQL 文からは競合が判定できない場合は，競合すると判定し，同じ Scheduler に対応付けられた全 TxHandler の Tx をロールバックする．

- DELETE FROM t1 WHERE c2=?
- UPDATE t1 SET c1=? WHERE id=?

図 2.8 では，各 TxHandler が記録した SQL テンプレート とその引数を利用して上記検証を行い，競合が発生した場合は，関連する全 Tx とグローバル・トランザクションをロールバックする．

2.3.6 原子性，分離性の保障

上記で示したアルゴリズムが，必ず原子性と，カーソル安定性を保障することを簡略して示す．

まず，原子性が保障されることを示す．Scheduler を共有する Tx の全更新 SQL は，1 つのグローバル・トランザクションとして，データベースに処理要求される．グ

ローバル・トランザクションは，データベースにとっては，通常のトランザクションと同じため，グローバル・トランザクションとして処理される更新SQLの原子性は，データベースにより保障される．また，グローバル・トランザクションがコミット後，Exceptionを発生せずに各TxHandlerに経由でTxにコミット終了が通知されるため，グローバル・トランザクションと同等の原子性が保障される．

次に，カーソル安定性が保障されることを示す．同一のSchedulerに対応する全Txの更新SQL間には，checkConflictにて，依存関係がないことが保障されるため，未コミット読み取り（Dirty Read）は発生しない．また，同様に，同じSchedulerに対応付けられた全TxHandlerのTxの照会中のSQLとグローバル・トランザクションの更新SQLは競合しないため，照会中のレコードが更新されることはない．すなわち，データベースがカーソル安定性を保証する限り，全クライアントのTx間での，カーソル安定性が保障される．

2.4 性能評価

提案手法をJava（JDBC）で実装し，2つのベンチマークを用いて，その性能を評価する．

2.4.1 測定環境

全ての評価は，IBM x336（Intel(R) Xeon(TM) 3.60GHz, 1024KB L2Cache, Hyperthreading, 2CPU, PC2-3200 DDR2 ECC SDRAM DIMM 7GB）をデータベース・サーバとして，IBM HS20（Intel(R) Xeon(TM) 3.40GHz，2048KB L2Cache, Hyperthreading，2CPU，PC2-3200 DDR2 ECC SDRAM DIMM 5GB）をクライアント（アプリケーション・サーバ）として利用した．OSは，Red Hat Enterprise Linux Server 5.2（2.6.18-92.el5 #1 SMP Tue Apr 29 13:16:15:15 EDT 2008 x86_64 x86_64 x86_64 GNU/Linux）を，データベースとしてDB2 9.5（s071001, LINUX-AMD6495）を利用した．クライアントは5台から構成され，データベース・サー

バとクライアントは Gigabit Ethernet で接続する．データベース・サーバは, IBM ServerRAID6M を利用し，データベースのトランザクション・ログ，ページデータを，2 ストレージ（15krpm x 7 RAID0）に保存する．

2.4.2 簡易ベンチマークの評価

本ベンチマークでは 1 つの文字列型のカラムを持つテーブルに対し，文字列を複数回挿入するトランザクションを用いて，本手法の特性を評価する．

なお，本章では，挿入するレコード数をステップ数と呼ぶものとする．各クライアント（IBM J9 VM (build 2.3, J2RE 1.5.0 IBM J9 2.3 Linux amd64-64 j9vmxa6423-20081129 を利用した Java プログラム）は，複数のスレッドがトランザクションの実行を繰り返すものとし，DB2 の JDBC ライブラリを直接利用した場合（normarl 環境）と，提案手法を利用した場合（batch 環境）のシステム全体のスループット（TPS）を測定する．Scheduler 数は 1，1 グローバル・トランザクションあたり，最大 100 トランザクションをグループ・コミット可能とするものとし，測定中は，各クライアント，データベースの CPU 利用率を Linux のコマンドである vmstat を利用して測定するものとする．

ロールバックがない場合の評価

クライアント台数ごとのトランザクション（4 ステップ・文字列長 5 文字）のスループットの最大値と，その際のクライアント・データベースの CPU 利用率を，図 2.10 に示す．なお，スループットの最大値は，各クライアントのスレッド数を 2 スレッド数ずつ増やして評価した際の，スループットの最大値を表すものとする．また，図 2.10 は，横軸はサーバ台数を表し，縦軸はスループット（折れ線グラフ）と，データベース，クライアントの CPU 利用率（棒グラフ，左から batch 環境でのデータベース，クライアント，normal 環境でのデータベース，クライアントの CPU 利用率）を表すものとする．

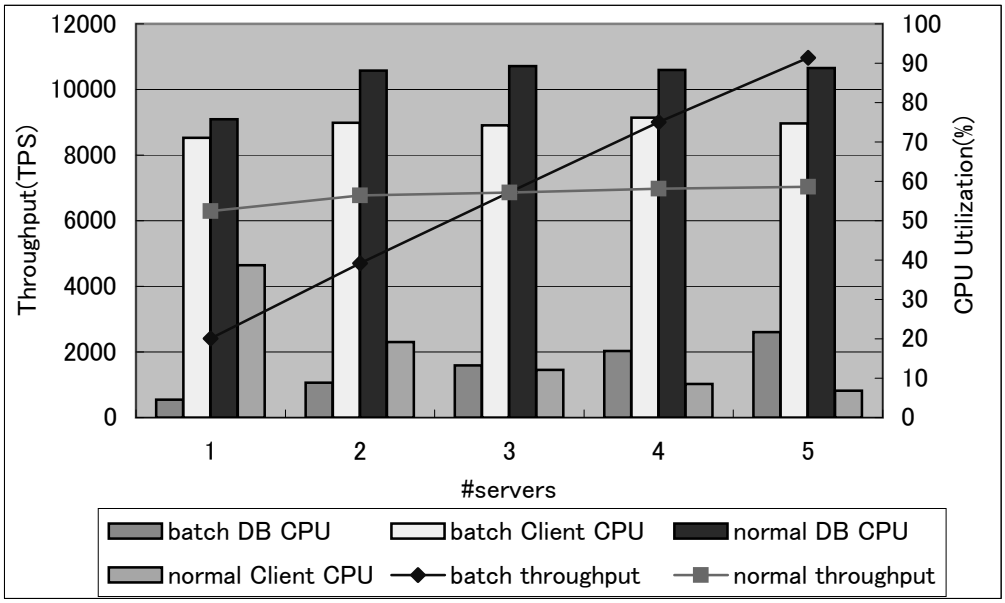

図 2.10: 4 ステップのスループットと CPU 利用率

図 2.10 に示すとおり，normal 環境ではクライアント数を変えてもスループットが変化しないのに対し，batch 環境ではクライアント数に応じてスループットが向上している．

図 2.10 における 5 サーバ時の詳細結果を，横軸をクライアントの総スレッド数に変えて，図 2.11 に示す．normal 環境ではスレッド数が増えるとデータベースの CPU 利用率が 90%を超え，スレッド数を増やしてもスループットが向上しない．一方，batch 環境では，スレッド数が増えると，クライアントの CPU 利用率が増え，スループットが向上しない．また，データベースの CPU 利用率は，常に 40%以下であり，最大スループットを示した 120 スレッドでは，21%の CPU 利用率である．つまり，normal 環境では，データベースの CPU ボトルネックになり，batch 環境ではクライアントの CPU ボトルネックになっていることがわかる．batch 環境では，データベースの CPU 利用率が十分低いので，クライアント数に応じてスループットが向上していくことが予想される．

次に，5 クライアント時の，ステップ数（文字数は 5 で固定），文字数（ステップ

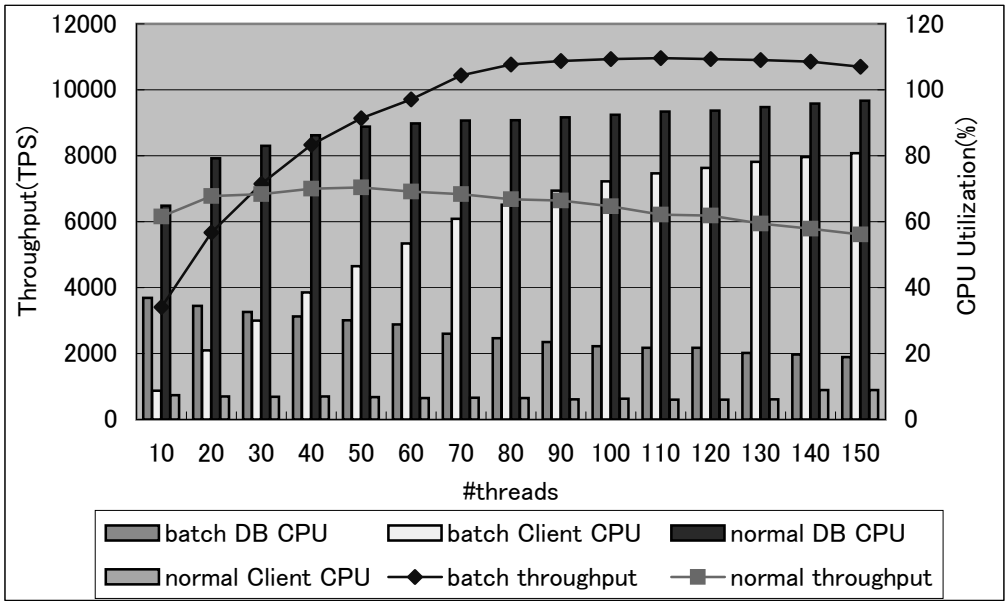

図 2.11: 4 ステップ，5 サーバのスループットと CPU 利用率の詳細

数は 4 で固定）に応じた提案手法による性能向上率を，それぞれ，図 2.12，図 2.13 に示す．なお，グラフの縦軸は，性能向上率（線グラフ）と，CPU 利用率（棒グラフ，図 2.10 と同様）を表し，横軸は，ステップ数，データサイズを表すものとする．

図 2.12 が示すように，1 ステップをバッチ更新する場合は，約 220%の性能向上しているのに対し，64 ステップをグループコミットする場合は，約 120%の性能向上しかしない．これは，ステップ数に応じてトランザクションとグローバル・トランザクション間の同期回数が増加し，クライアントの負荷が増すためである．図 2.8 では，Scheduler のスレッドの wait()（4・16 行目）と TxHandler のスレッドの notifyAll()（25・30・37），Scheduler のスレッドの notifyAll()（13・20）と TxHandler のスレッドの wait()（32・38）が，Java のモニタ機能を利用して，同期を行う．ステップ数が増えるとこの同期の数が増えるため，クライアントの CPU リソースの消費量が増える．上記より，本提案手法は，ステップ数が多いトランザクションに比べ，少ないトランザクションの方が，効果的であることがわかる．また，図 2.13 が示すように，文字数が多くなると性能向上率が小さくなる．これは，クライアントの送信

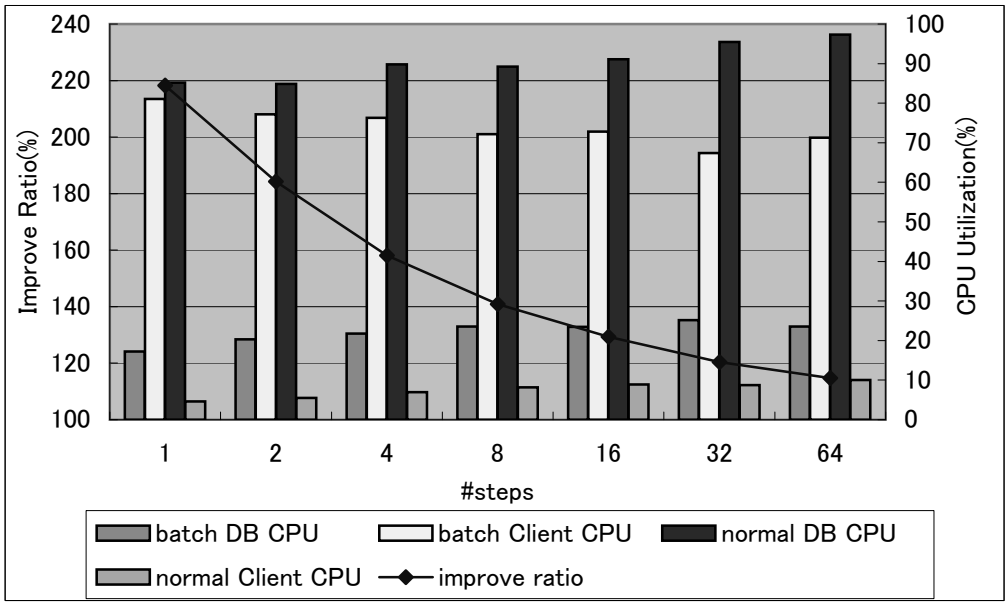

図 2.12: ステップ数に応じた性能向上率

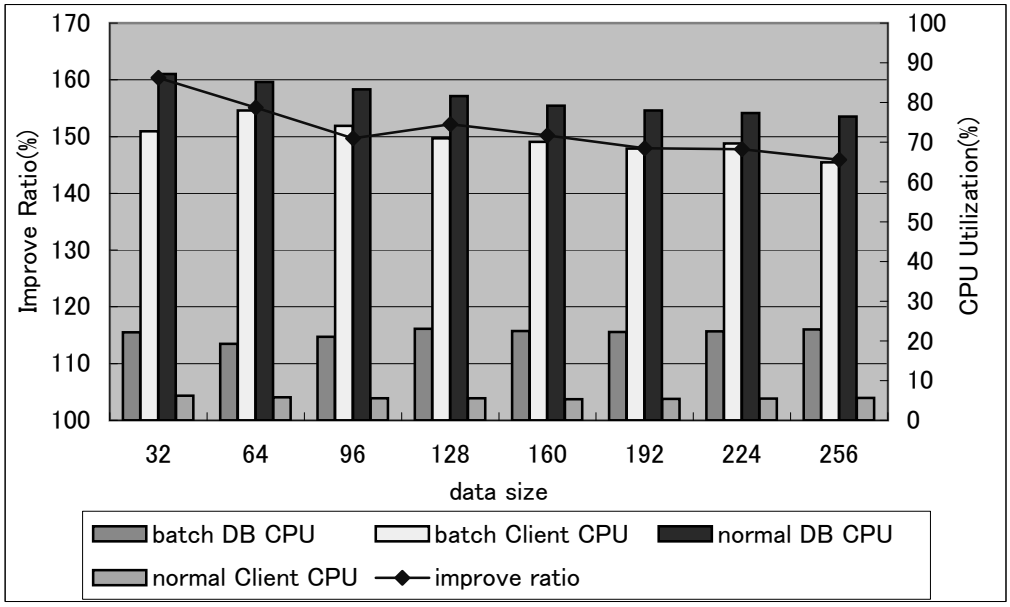

図 2.13: データサイズに応じた性能向上率

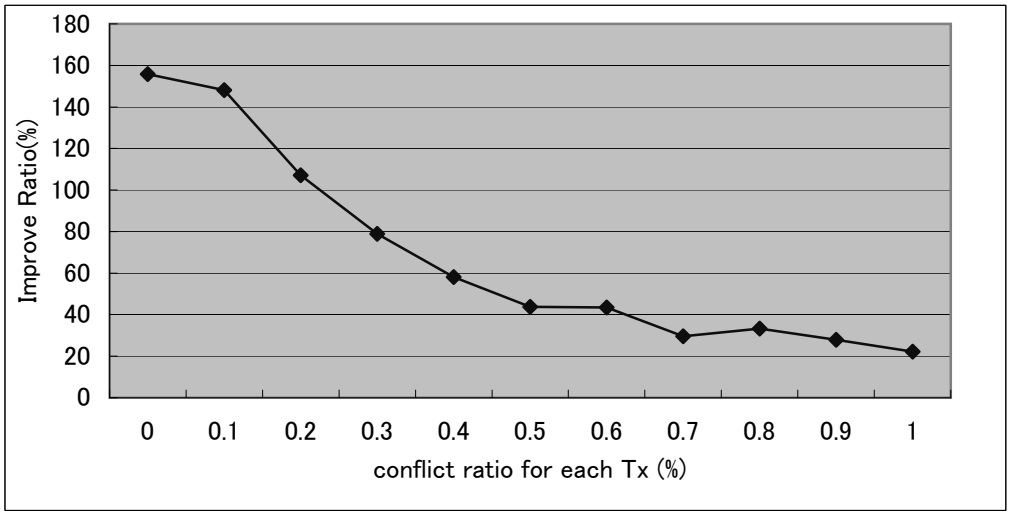

図 2.14: トランザクションの競合率に応じた性能向上率

のための負荷が大きくなることが原因と考えられる．

ロールバックがある場合の評価

2.4.2 節での全評価は，トランザクション間に依存がないことを想定しているが，実アプリケーションのトランザクション間に依存があるため，グローバル・トランザクション，トランザクションがロールバックする可能性がある．そのため，本章では，最後に，実アプリケーションを想定し，トランザクションが競合する可能性がある環境での評価も行う．文字数 5，ステップ数 4，総スレッド数 100 の際の，競合率 0 から 1%に対応した性能向上率を図 2.14 に示す．なお，トランザクション間の競合率は，2 つのトランザクションが競合する確率を表す．競合する場合は，グローバル・トランザクションとグローバル・トランザクションの Scheduler に対応付けられた全 TxHandler のトランザクションがロールバックする．

図 2.14 に示すように，競合率が高くなると，batch 環境のスループットが悪化し，0.3%以上の競合率で，normal 環境よりもスループットが低くなる．このことから，本手法は，競合率が十分に低いアプリケーションに有用であることがわかる．なお，

実アプリケーションを模したTPC-Wでは，商品購入時の競合率は，0.1%以下の競合率となるため，多くのアプリケーションでは本手法は有用であると考えられる．また，グローバル・トランザクションに対応したトランザクション間で競合が起きた場合でも，全トランザクションの依存関係が直列化可能の際には，グローバル・トランザクション，トランザクションのコミットを許容するようにすれば，より競合率を低下することが可能と考えられる [7].

2.4.3 DayTraderの評価

DayTrader[27] は，株取引をシミュレーションするWebアプリケーションである．DayTraderでは，10種類のWebページ（home，quote，login，logout，register，account，portfolio，buy，sell，profile）がスループットを測定するために用意される．ブラウザは，scenarioページを表示することで，適切なWebページに転送され，あらかじめ規定されたワークロードを実行することが可能となる．なお，ワークロードの90%以上が照会トランザクションで，15種類のTxSeqから構成される．

本実験では，5台のクライアント上にWebSphere6.1（WAS）をインストールし，WAS上でDayTraderを稼動する．稼動するDayTraderのscenarioページに対し，別サーバからは複数スレッドでHTTPリクエストを送り続け，スレッド数を変化させながら，スループットを測定した．実験結果を，図2.15に示す．なお，HTTPリクエストを送信するための負荷は十分低く，測定するスループットには影響がなかった．

図2.15に示すように，batch環境の最大スループットは，normal環境よりも12%向上した．DayTraderは90%以上が照会トランザクションであることを考慮すると，本手法は十分有用であることが考えられる．

図2.11と同様，batch環境では，normal環境に比べ，クライアント（アプリケーション・サーバ）のCPU利用率が増えている．これは，同様に，SchedulerとTxHandlerのスレッドの同期によるものと考えられる．

スレッド数10のbatch環境のスループットは，normal環境と，ほぼ同じ値であっ

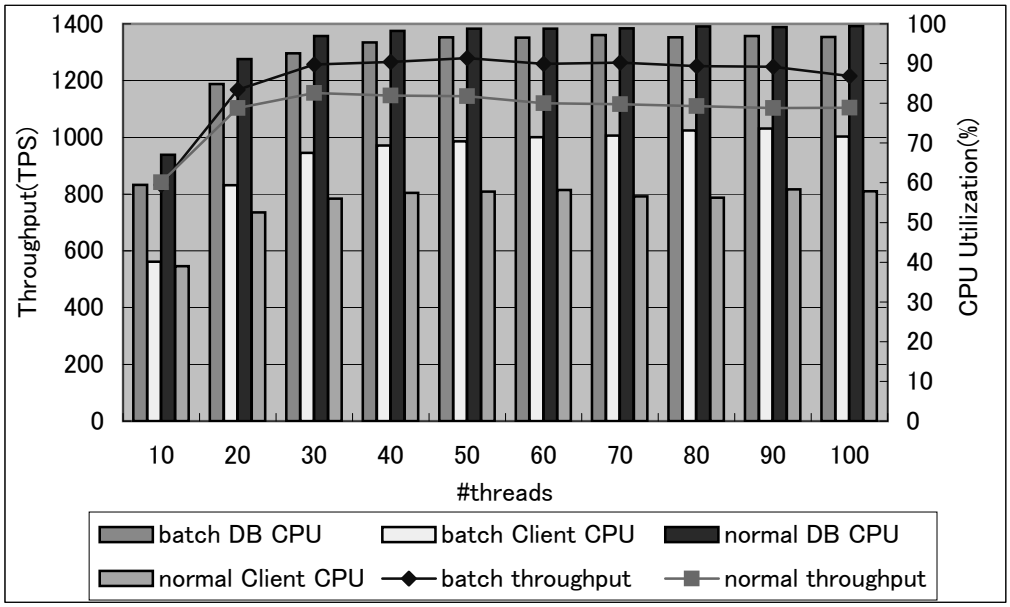

図 2.15: DayTrader のスループットと CPU 利用率の詳細

た（batch 環境のスループットは，native 環境に比べ 1%低い）．スレッド数 10 の場合，各 WAS は最大 2 リクエストを同時に処理するため，各 Scheduler は，1 つのトランザクションの更新を処理することになる．つまり，batch 環境は，normal 環境に比べ，Scheduler，TxHandler の処理分，処理量が多い．処理量が増加したにも関わらず，スループットがほとんど変わらないのは，それらの処理量は，無視できるほど小さいと言える．また，全測定において，競合の発生率は，トランザクション中 0.01%以下であった．つまり，提案手法によるオーバヘッドによって，システム全体のスループットを低下させる可能性が小さいことが考えられる．

2.5 関連研究

同時に要求されているクエリに対する共通の処理を共有することで最適化行う手法は，Multi Query Optimization（MQO）と呼ばれ，従来広く研究されている [2, 3, 4]．これらの手法では，本提案手法のように，同時実行されるクエリから共通処理を特

定し，クエリ処理の最適化を行う [2]．さらに，データベースのバッファを最適化するためのパイプライン [3] や，ディスクキャッシュを最適化するグルーピング手法 [4] 等，システムに応じた最適化の研究もされている．MQO は，実行中のクエリに対して適用可能なため，事前知識を必要としないが，各トランザクションに対する分離性，原子性を保障するしながら更新 SQL には適用する研究は，従来行われていない．

また，本章のように，要求される SQL テンプレートのパターンを学習・予測結果を利用することで，データベースの処理を軽減する手法も，提案されている [8, 9]．これらの手法では，1 つのパターン内で連続して要求される照会 SQL の共通結果を共有し [8]，将来実行させると予測する照会結果をプレフェッチ [8, 9] するが，本提案手法のように，同時要求される複数のトランザクションの更新 SQL に対する最適化には適用できない．

複数のトランザクションを，最適な順番で実行する手法は，主にロック要求の許可する順番を最適化を行うことで，広く行われている [5, 6, 10]．これらの手法は，各トランザクションがアクセスするデータをシステムが事前に認識している必要があるため，提案手法のように，どのような SQL が実行されるか認識できない環境へ応用するのは難しい．

Harizopoulos らは，Staged-Execution[11] の手法を用いて，同時に要求されるクエリを処理する手法 [12, 13, 14] を提案している．Staged-Execution では，各クエリの処理はタスクに分割され，各スレッドが同種のタスクをまとめて処理する．本手法を用いると，同時に大量のクエリを要求された場合においても，要求されたクエリより少ない数のスレッドで処理が可能となるため，データベースにおけるスレッド切り替えのオーバヘッドが小さくなることが考えられる．しかしながら，彼らの手法は照会クエリの処理にのみ適用されており，ディスクアクセスを頻繁に行う更新クエリの処理に適用されていない．

2.6 まとめ

本章では，アプリケーションを変更せずに，通常のトランザクション実行と同等の原子性，分離性を保障して，グループ・コミットを行うトランザクション処理手法を提案した．本提案手法を Java を用いて実装，評価したところ，更新トランザクションの多いアプリケーションのデータベースの CPU ボトルネックを解消し，クライアント数に応じてシステム全体のスループットを向上させることが可能であることがわかった．今後，より複雑なアプリケーションを用いて，本提案手法の有用性を評価する予定である．

第3章　ロック制御型同期複製ミドルウェア

3.1　まえがき

データベースの複製を利用して，データベースを利用するアプリケーションを書き換えることなく，データベースの性能向上を実現するためには，複製されたデータベース（レプリカ）全体が，1台のデータベースと同じ一貫性を保障する必要がある．しかし，各データベース・ベンダーは，データベース複製ツールを提供しているが，1台のデータベースと同じ一貫性を提供する機能を提供していない [15]．そのため，アプリケーションとレプリカの間にミドルウェアを配置し，ミドルウェアがレプリカの複製と一貫性を管理しながら，アプリケーションがデータベースに要求する照会・更新SQLを適切なレプリカに処理要求する，ミドルウェア型の複製手法が必須である．

従来，数多くのミドルウェア型の同期複製手法が，一台のデータベースと同等の分離性を保障する手法として提案されてきているが，多くがスナップショット分離性（*SNAPSHOT*）を保障する手法である [16, 17, 18, 19, 20]．しかし，繰り返し可能読み取り（*REPEATABLE READ*），コミット読み取り（*READ COMMITTED*），非コミット読み取り（*READ UNCOMMITTED*）の分離性を提供するデータベースは，多くのアプリケーションで利用されている [28][1]ため，これらの分離性を保障するミドルウェアも必要である．

SNAPSHOT以外の分離性を保障するミドルウェア型複製手法としては，C-JDBC[21]

[1]IBM DB2[29] や SQL Server[30] は，これらの分離性をサポートしており，そのシェアは40%以上である．

が提案されている．C-JDBC は，全てのレプリカが同じデータを保持し，アプリケーションによる照会 SQL 要求は 1 つのレプリカに，更新 SQL 要求は全レプリカに処理要求する．アプリケーションが要求する分離性は，各レプリカへの SQL 処理にも同様に要求されるため，REPEATABLE READ や READ COMMITTED を保障しながら，照会処理の多いアプリケーションの性能向上を実現可能である．しかし，C-JDBC は，これらの分離性を保障する際，複数のレプリカに跨ってデッドロックが発生する可能性がある [16]．例えば，レコード B を更新後にレコード A をレプリカ 1 で照会するトランザクション T1 と，レコード A を更新後にレコード B をレプリカ 2 で照会するトランザクションを同時に実行した場合，レプリカ 1 では T1 の照会処理が T2 のロック開放待ちに，レプリカ 2 では T2 の照会処理が T1 のロック開放待ちになる．このような複数のレプリカにまたがったデッドロックが発生すると，アプリケーションのエンドユーザへの応答ができなくなるため，実利用が困難である．

一方，Web アプリケーションのような OLTP システムは，ユーザからのリクエストを短時間で処理する必要があるため，処理に要するロックが少ない，単純な SQL が処理されることが多い．このような SQL の場合，SQL の記述や，ロック範囲を特定する SQL をレプリカに処理要求することで，低オーバヘッドで必要なロックを管理することが可能である．

そこで，本章では，ミドルウェアで各トランザクションのロックを管理し，REPEATABLE READ，READ COMMITTED，READ UNCOMMITTED の分離性を保障するミドルウェア 闇魔（Yama）を提案する．

Yama では，C-JDBC と同様，全レプリカが同じデータを保持し，更新 SQL は全レプリカで処理することで複製を行い，照会 SQL を 1 台のレプリカで処理することで照会処理の多いアプリケーションの性能向上を実現する．また，Yama は，アプリケーションに要求された SQL をレプリカに処理要求する前に，Yama 内の排他制御機構を用いて，各トランザクションに必要なロックを与える．つまり，各トランザクションが獲得中の全ロックが把握可能なため，ロックの衝突状況を確認してデッ

ドロックを迅速に検出可能である．

Yamaでは，アプリケーションが要求するSQLから，排他制御に必要となるロックを特定する．しかし，通常，アプリケーションが要求するSQLからだけでは，SQLを処理する際に必要となる全ロックを特定することはできない．そこで，Yamaでは，(1) SQLの記述からその処理時にアクセスするレコードを特定可能な場合は，そのレコードに対応するロックを特定し，(2) それ以外の場合は，要求されたSQLから別SQL（ヘルパーSQL）を生成し，1台のレプリカに問い合わせることで必要なロックの特定を試みる．また，(3) ヘルパーSQLで特定できないSQLに対しては，テーブルロックを獲得することで排他制御を行う．なお，(2) におけるヘルパーSQLの処理では，非コミット読み取りの分離性を要求することで，レプリカ内での処理の停滞を防ぐ．本処理ではレプリカが誤った値を返す可能性があるが，Yamaでは，他トランザクションでの更新中のレコードを監視することで，正確な値を認識する．

本章では，Yamaが照会中心のアプリケーションに対して有効性であることを示すため，TPC-W[31]で利用されるワークロード（browsing mix，shopping mix，ordering mix）を，5台のレプリカと，1台のYama稼動用のサーバ上で評価する．

3.2 複製ミドルウェアにおける一貫性の課題

3.2.1 トランザクションモデルと複製ミドルウェア

本章では，クライアントが，データベースに対し，トランザクションの開始を要求後，複数の照会，更新の処理を要求し，コミット，もしくは，ロールバックを要求することで，データベースに対してトランザクション処理を要求するものとする．照会，更新処理は，SQLを用いて要求されるものとし，データベースは，要求されたSQLを処理し，その結果をクライアントに返す．また，ロールバックを要求された場合は，データベースは，トランザクション内で更新されたデータを更新前の値に戻す処理を行う．

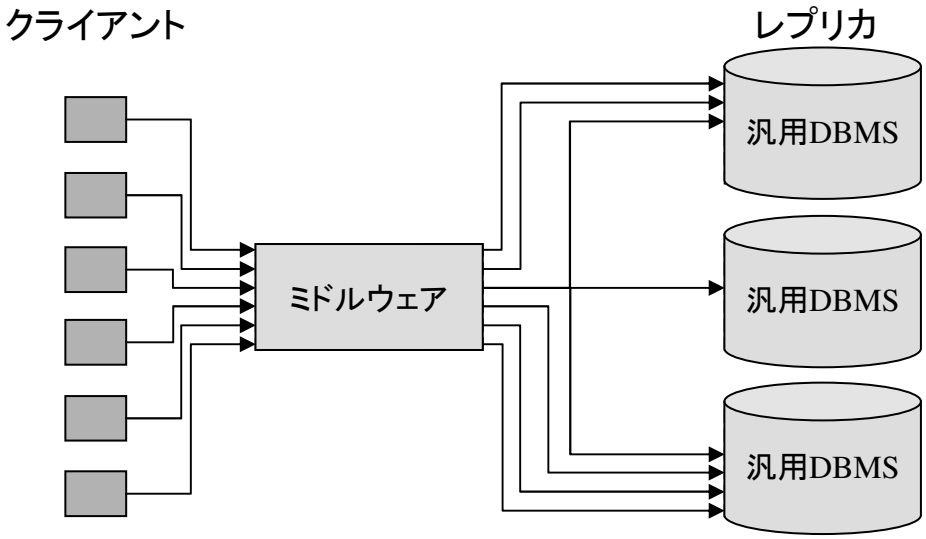

図 3.1: 同期複製のシステム構成

同期複製ミドルウェアを利用したトランザクション処理では，図 3.1 に示すように，クライアントは，ミドルウェアを 1 台のデータベースとして扱い，トランザクションの処理要求を行う．ミドルウェアは複数のレプリカを管理し，クライアントからのトランザクションの処理要求に応じて，レプリカにトランザクション処理を要求することで，要求された処理の結果をクライアントに返す．つまり，クライアントがミドルウェアに要求するトランザクションは，1 台，もしくは，複数台のレプリカのトランザクションとして処理される．本章では，クライアントがミドルウェアに要求するトランザクションをグローバル・トランザクション（GlobaleTx），ミドルウェアがレプリカに要求するトランザクションをローカル・トランザクション（LocalTx）と記述するものとする．

3.2.2 ANSI 分離性の実装

データベースでのトランザクション処理は，各トランザクションの分離性に対する制約を緩めることで，データベースの性能を向上することが可能である．ANSI[32]

では，繰り返し可能読み取り（*REPEATABLE READ*），コミット読み取り（*READ COMMITTED*），非コミット読み取り（*READ UNCOMMITTED*）が定義されている．これらは，各SQL処理に対して排他ロックと共有ロックを用いた排他制御を行うことで，以下のように実現される [22].

- REPEATABLE READ：各トランザクションが，更新するレコードの排他ロック，照会するレコードの共有ロックを獲得し，コミット，もしくは，ロールバックする際に，全ロックを開放する．
- READ COMMITTED：各トランザクションが，更新するレコードの排他ロック，照会するレコードの共有ロックを獲得し，共有ロックは照会処理後すぐに，排他ロックはコミット，もしくは，ロールバックする際に開放する．
- READ UNCOMMITTED：各トランザクションが，更新するレコードの排他ロックを獲得し，コミット，もしくは，ロールバックする際に開放する．

なお，REPEATABLE READを実現可能なシステムは，共有ロックの開放のタイミングを変更することで，READ COMMITTED，READ UNCOMMITTEDを実現できるため，本章ではREPEATABLE READを実現する手法を示す．

3.2.3 複製ミドルウェアによる分離性の実装の課題

同期複製ミドルウェアでANSIの分離性を実現するには，ミドルウェアは，要求されたSQLを処理する際にアクセスするレコードを特定し，そのレコードに対するロックを獲得した後に，レプリカにSQLの処理を要求しなくてはならない．しかし，一般的なデータベースは，SQLを処理する際にアクセスされるレコードを特定する機能を提供していない．

3.3 複製ミドルウェアによる排他制御機構の提案

SQL をレプリカで処理する前に，レコード，カラム，テーブルに対するロックを利用して GlobalTx の排他制御を行い，同期複製を行うミドルウェア，Yama（闇魔）を提案する．各 GlobaleTx によって獲得されているロックを Yama 内で解析することが可能なため，Yama は全てのデッドロックを検出可能になる．

3.3.1 SQL の分類

Web アプリケーションに代表される OLTP システムは，ユーザからのリクエストを短時間で処理する必要があるため，単純な SQL が処理されることが多い．単純な SQL の場合，SQL の記述や，レプリカに問い合わせることで，低オーバヘッドで必要なロックを認識することが可能である．

OLTP Web アプリケーションのベンチマークである TPC-W が要求する SQL を以下のように分類し，表 3.1 に各分類の SQL が要求される割合を示す．

- ロックを必要としない照会：READ UNCOMMITTED で処理可能な照会．
- 主キー値指定のテーブル照会：1 つのテーブルから，指定した値を主キー値として持つレコードの照会．
- カラム値指定のビュー照会：外部キーを利用して複数のテーブルが Join されたビューから，指定した値をカラム値として持つレコードの照会．
- その他の照会：上記以外で，ロックを必要とする照会．
- 主キー値指定のテーブル更新：指定した値を主キー値として持つレコードの更新．
- カラム値指定のテーブル更新：指定した値をカラムの値として持つレコードの更新．

表 3.1: TPC-W における SQL の要求数の割合

	mix		
	Browsing	Shopping	Ordering
ロックを必要としない照会	50.20%	29.66%	10.75%
主キー値指定のテーブル照会	19.20%	17.21%	29.02%
カラム値指定のビュー照会	15.36%	14.74%	17.89%
その他のビュー照会	-	-	-
主キー値指定のテーブル更新	14.80%	38.22%	42.30%
カラム値指定のテーブル更新	0.44%	0.16%	0.05%
その他の更新	-	-	-

- その他の更新：上記以外の更新.

表 3.1 に示すように，TPC-W では，ロックを必要とする処理では，主キー値やカラム値指定の処理が多く，単純な SQL が多く処理される．主キー値指定の処理は SQL の記述を解析することで，カラム値指定の処理はレプリカに問い合わせることで，容易に必要となるロックを特定可能である.

3.3.2 ロックの種類

Yama では，レコード，カラム，テーブルの 3 つの粒度（RecordLock，ColumnLock，TableLock）と，共有・排他の 2 種類のロック（S，X ロック）を用いて，排他制御を行う．RecordLock は各テーブルのキー値に特定の値を有するレコードに対するロックを表し，ColumnLock は各テーブルのカラムの値に特定の値を要するレコード集合に対するロックを表し，TableLock は各テーブルに対するロックを表す． ロックは GlobaleTx ごとに与えられ，共有ロックは同時に複数の，排他ロックは同時に 1 つのトランザクションが獲得し，トランザクションの終了と共に開放される．なお，ある TableLock が獲得された場合，そのテーブルに関する全ての RecordLock・ColumnLock が獲得されたものとみなす.

3.3.3 ローカル・トランザクションの実行

Yamaでは，LocalTxの実行は，全てREAD UNCOMMITTEDの分離性で実行する．また，全ての更新SQLの処理は，Yama内で更新対象となる排他ロックを獲得後に，全レプリカに処理要求する．READ UNCOMMITTEDの分離性を要求した場合，同じレコードを複数のGlobalTxが更新しない限り，レプリカ内で処理が滞ることはない．また，Yama内での排他制御により，同じレコードを複数のGlobalTxが更新することはないため，Yamaでは，レプリカ間をまたがったデッドロックが発生することはない．

3.3.4 SQL解析による排他制御

主キー値指定のテーブル照会・更新は，SQLの内容を解析し，照会・更新する主キーを認識することで，RecordLockを認識する．例えば，TPC-Wでは，以下のS_1を利用して，主キーC_IDを持つCUSTOMERテーブルからユーザ名を取得する．この場合，SQLを構文解析することにより，C_IDが100のRecordLockを獲得する必要があることを認識可能である．

- S_1:SELECT C_FNAME, C_LNAME FROM CUSTOMER WHERE C_ID=100

図3.2に，Yamaにおける，SQLから必要なRecordLockを解析して照会・更新する処理方法を示す．なお，図中のlock関数は，要求するロック配列（RecordLock, ColumnLock，TableLockの配列）と，要求するロックの種類（S，X）を指定して，ロックを獲得する関数である．

3.3.5 獲得するロックのレプリカへの問い合わせ

カラム値指定のビュー照会･テーブル更新は，SQLの記述からは必要なRecordLockを特定できない．例えば，TPC-Wでは，指定されたタイトル名を持つ本をデータベースに問い合わせる際，主キーの値が含まれない以下のようなS_2を要求する．

```
1: RecordLock[] locks =[SQL より主キーの値を特定];
2: lock(locks, S);
3: return [1 台のレプリカを選択し，SQL の処理を要求];
```

```
1: RecordLock[] locks =[SQL より主キー値を特定];
2: lock(locks, X);
3: [全レプリカに，SQL の処理を要求];
```

図 3.2: SQL 解析による照会（上）・更新（下）処理

- S_2：SELECT I_ID, A_FNAME, A_LNAME FROM ITEM LEFT JOIN AUTHOR ON I_A_ID = A_ID WHERE I_TITLE LIKE 'XXX'

Yama では，カラム値指定のビュー照会・テーブル更新に対しては，SQL を処理する際にアクセスするレコード値とカラム値を取得する SQL（ヘルパー SQL）を生成し，任意のレプリカで処理することにより，必要な RecordLock，ColumnLock を認識する．

ヘルパー SQL は，照会 SQL の場合，Join したテーブルの全ての主キーと WHERE 句で利用するカラムを，要求された SQL の SELECT 句に追加することで生成する．例えば，S_2 の場合，以下の S_3 に変換する（太字が追加した箇所を示す）．

- S_3：SELECT I_ID, A_FNAME, A_LNAME, **A_ID, I_TITLE** FROM ITEM LEFT JOIN AUTHOR ON I_A_ID = A_ID WHERE I_TITLE LIKE 'XXX'

更新 SQL の場合は，更新対象のテーブルの主キーと，WHERE 句で利用するカラムの値，更新するカラムの元の値を照会する SELECT 文を，更新 SQL 内の WHERE 句の記述を利用して生成する．例えば，以下の S_4 の場合，S_5 に変換する．

- S_4：UPDATE ITEM SET I_TITLE = 'YYY', I_A_ID = 10 WHERE I_TITLE = 'XXX'

- S_5：SELECT I_TITLE, I_A_ID FROM ITEM WHERE I_TITLE = 'XXX'

Yama は，上記のヘルパー SQL の処理結果より，必要な RecordLock と Column-Lock を認識してロックを獲得後，要求された SQL の処理を行う．図 3.3 に，その処理方法を示す．

図 3.3 の照会処理では，まず，ヘルパー SQL の生成を行い（2 行目），実行中の全 GlobalTx を認識する（4 行目）．次に，1 台のレプリカにヘルパー SQL の処理を要求し，その結果から照会するレコードと WHERE 句で利用するカラムロックを特定し，共有ロックを獲得する（5-8 行目）．ここで，Yama では，8 行目で獲得した同じロックに対して，4 行目で認識した GlobalTx と 4 行目以降に開始された GlobalTx が，排他ロックを獲得しているか調べる（9）．もし同じロックを獲得している場合，5 行目で照会した処理結果に，非コミット読み取りが発生している可能性があるため，Yama では，再度ヘルパー SQL を実行し，照会するレコードの特定を行う．同じロックが獲得されていない場合は，ヘルパー SQL の処理結果から要求された SQL の照会結果を生成する（11 行目）．

更新処理では，照会処理と同様，2-6 行目を実行した後，ヘルパー SQL の処理結果より，更新する対象のレコード値（6 行目），WHERE 句で照会するカラム値（7 行目），更新する前のカラム値（9 行目）に対応するロックを特定する．また，要求された SQL より更新後のカラム値のロックを特定し（8 行目），共有・排他ロックを獲得する（10-11 行目）．次に，10 行目で獲得した同じロックに対して，4 行目で認識した GlobalTx と 4 行目以降に開始された GlobalTx が，排他ロックを獲得しているか調べる（12 行目）．ビュー照会同様，獲得している場合は再度ロックを特定しなおし，獲得していない場合は更新を全レプリカに要求する（14 行目）．

3.3.6 解析できない SQL のロックの認識

Yama では，上記の解析対象とならない照会 SQL は，SQL から照会対象となるテーブルを認識し，テーブルロックを獲得後に照会 SQL の処理を 1 台のレプリカに

```
1:  ResultSet result;
2:  String helperSQL = [SQL からヘルパーSQL を生成];
3:  while (true) {
4:      [実行中のトランザクションを認識];
5:      result = [1 台のレプリカにヘルパーSQL の処理を要求];
6:      RecordLock[] locks = [result から照会対象の主キーを特定];
7:      ColumnLock[] cLocks= [result から WHERE 句で利用するカラム値を特定];
8:      lock(locks, S); lock(cLocks, S);
9:      if (![4 行目で認識したトランザクションと 4 行目以降に開始したトランザクション
            が獲得したロックが，8 行目で獲得した locks と衝突している])
10:         break; }
11: return [result から sql が要求する列のみ抽出];
```

```
1:  ResultSet result;
2:  String helperSQL = [SQL からヘルパーSQL を生成];
3:  while (true) {
4:      [実行中のトランザクションを認識];
5:      result = [1 台のレプリカにヘルパーSQL の処理を要求];
6:      RecordLock[] xLocks = [result から更新対象の主キーを特定];
7:    ColumnLock[] sCLocks=[result から WHERE 句で利用するカラム値を特定];
8:      ColumnLock[] preCLocks = [result から更新前のカラム値を特定];
9:      ColVal[] postCLocks = [SQL から更新後のカラム値を特定];
10:     lock(sCLocks, S);
11:     lock(xLocks, X); lock(preCLocks, X); lock(postCLocks, X);
12:     if (![4 行目で認識したトランザクションと 4 行目以降に開始したトランザクション
            が獲得した排他ロックが，10 行目で獲得した locks と衝突している])
13:         break;}
14: [全レプリカに，SQL の処理を要求];
```

図 3.3: ヘルパー SQL による照会（上）・更新（下）処理

要求する．また，解析対象とならない更新 SQL に対しては，同様に，SQL から更新対象となるテーブルを認識し，テーブルロックを獲得後に更新 SQL の処理を全レプリカに要求する．図 3.4 にその処理を示す.

```
1: Table[] locks =[SQL 内でアクセスする全テーブルに対応する共有ロックを特定];
2: lock(locks, S);
3: return [1 台のレプリカを選択し，SQL の処理を要求];
```

```
1: TableLock[] sLocks=[SQL がアクセスする全テーブルに対するロックを特定];
2: TableLock[] xLocks=[SQL が更新する全テーブルに対するロックを特定];
3: lock(slocks, sLock); lock(locks, xLock);
4: [全レプリカに，SQL の処理を要求];
```

図 3.4: テーブルロックを獲得する照会（上）・更新（下）処理

3.3.7 原子性の保障

更新グローバル・トランザクションの処理を要求された際，Yama は，各レプリカ上で，同じデータを更新する更新ローカル・トランザクションの処理要求を行う．各レプリカ上の更新ローカル・トランザクションは，Yama からコミット要求をレプリカが受けた際，2 フェーズ・コミットのように同期してコミットされず，各レプリカが独立してコミットする．このようなシステム構成の場合，(1) レプリカの障害，(2) Yama の障害時において，更新グローバル・トランザクションの原子性は，ミドルウェア側で保障しなくてはならない．

Yama では，あるレプリカに障害が発生した場合，そのレプリカを，以降，利用しないことによって，原子性を保障する．つまり，グローバル・トランザクションは，障害が発生していない全レプリカ上のローカル・トランザクションがコミットされた時点で，コミットが完了する．また，Yama は，照会 SQL，更新 SQL の処理を要求された場合，障害が発生していないレプリカに対して，処理要求を行う．上記のように処理することで，Yama は，全てのグローバル・トランザクションの原子性を，保障することが可能となる．なお，障害から普及したレプリカや，新しいレプリカを追加する場合は，追加するレプリカに他のレプリカの全データを複製後，更新・照会処理の要求を開始する．

また，Yamaが障害から普及した場合，全レプリカの全データを同期させた後，アプリケーションからのトランザクション処理要求を受け付ける．この際，各レプリカのトランザクション・ログを確認し，各レプリカがコミットしていない更新を検出し，複製することで，データの同期を行う．トランザクション・ログの確認ができないレプリカに対しては，各レプリカに優先順位を付け，更新トランザクションのコミット処理要求を，優先順位順に要求することで，各レプリカがコミットしていない更新を検出する．上記のように処理することで，Yamaを障害時においても，グローバル・トランザクションの原子性を保障することが可能となる．

3.4 評価

我々は，同期複製ミドルウェア手法 YamaをJavaを用いて実装（Yama実装）し，TPC-Wを用いて，異なるレプリカ台数でのスループットを測定，そのスケーラビリティの評価を行った．

また，我々は，Yama実装との比較対象として，C-JDBCと同様，GlobaleTxに対して要求された分離性を，LocalTxに要求するミドルウェア（Direct実装）を実装し，Yama実装と同様の評価を行った．Direct実装では，分離性を保障するための排他制御を，全レプリカが行うのに対し，Yama実装では，ミドルウェアが一括して排他制御を行う．Yama実装とDirect実装のスケーラビリティを評価することにより，排他制御方法の違いによるスケーラビリティへの影響を評価する．

3.4.1 測定環境

本評価は，ミドルウェア用のサーバ1台（Single Core Xeon 3.1GHz 2SMP，16GB DDR2 RAM），レプリカ用のサーバ5台（Single Core Xeon 3.0GHz 2SMP，8GB DDR RAM）を用いて行った．複製ミドルウェアは，JDBCをインタフェースとしてもつJavaライブラリとして実装し，JDBCを用いて各レプリカにLocalTxを要求する．なお，OSにRedhat Enterprise Linux 5.2（64bit）を，データベースにDB2

9.5 を，Java に IBM Java 1.6.0 を利用した．また，各レプリカの DB2 上で実行される LocalTx は，Direct 実装では，TPC-W が要求する分離性 (繰り返し可能読み取り，コミット読み取り，もしくは，非コミット読み取り)，Yama 実装では非コミット読み取りの分離性で処理するように設定した．

本評価では，仮想的なブラウザ（EB Emulated Browser）が，Web アプリケーションが提供する 14 種類のインタフェースを通じて，3 つのワークロード（shopping, browsing，ordering[2]）で，データベースにトランザクション処理を要求するベンチマーク，TPC-W を用いた．なお，表 3.1 に示すとおり，TPC-W は mix によっては更新 SQL を多く処理するが，更新 SQL に比べ，照会 SQL はソートやグループ処理等，複雑な処理を必要とするため，処理量的には，全ての mix が照会中心のワークロードとなる[3]．

本評価はミドルウェアのスケーラビリティの評価を目的とするため，EB が直接ミドルウェアに対してトランザクション処理を要求するプログラムを用いた．性能値は，EB 数を変化させ，最も高い単位時間当たりのトランザクション数（スループット）とする．

3.4.2 評価結果

各 mix の評価結果を，図 3.5，3.6，3.7 に示す．

両実装は，全 mix において，レプリカ台数に応じてスループットが向上している．両実装では，更新 SQL の処理は全レプリカで，照会 SQL の処理は 1 台のレプリカで処理される．つまり，レプリカ台数が増えるほど，システム全体で処理可能な照会 SQL の処理量が増加する．TPC-W のように，照会処理中心のワークロードでは，本結果が示すとおり，システム全体としてのスループットが向上することになる．

また，本結果では，browsing，shopping，ordering mix と，更新 SQL の処理量の

[2]shopping mix は更新 20%，照会 80%，browsing mix は，更新 5% 照会 95%，ordering mix は，更新 50%照会 50%の割合でトランザクション処理を要求する．

[3]最も更新処理の多い ordering mix においても，データベースは，80%以上の時間，照会 SQL の処理を行う．

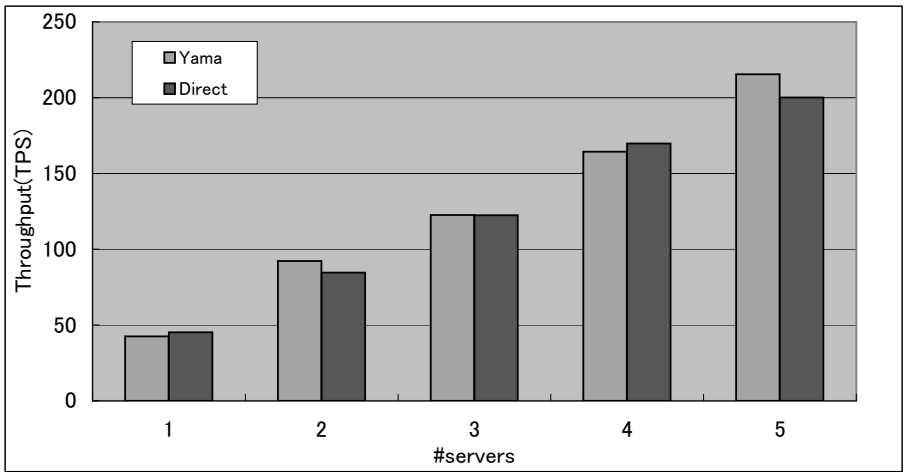

図 3.5: browsing mix の性能比較

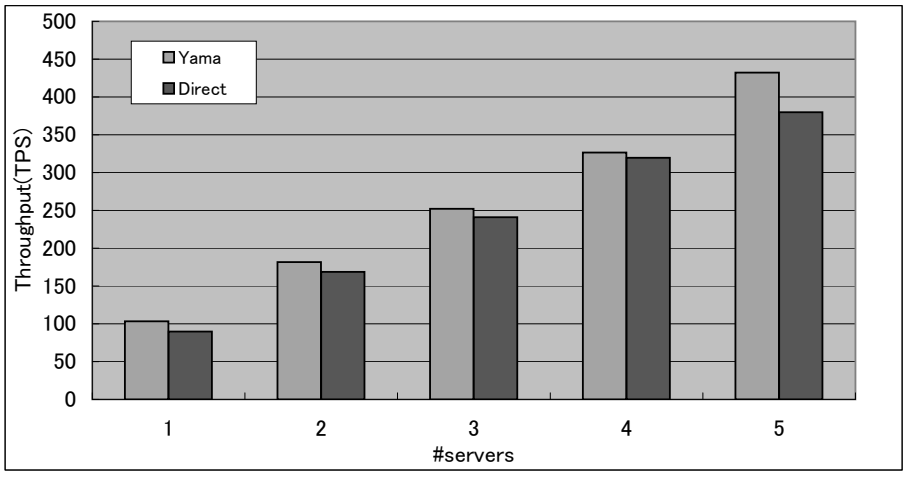

図 3.6: shopping mix の性能比較

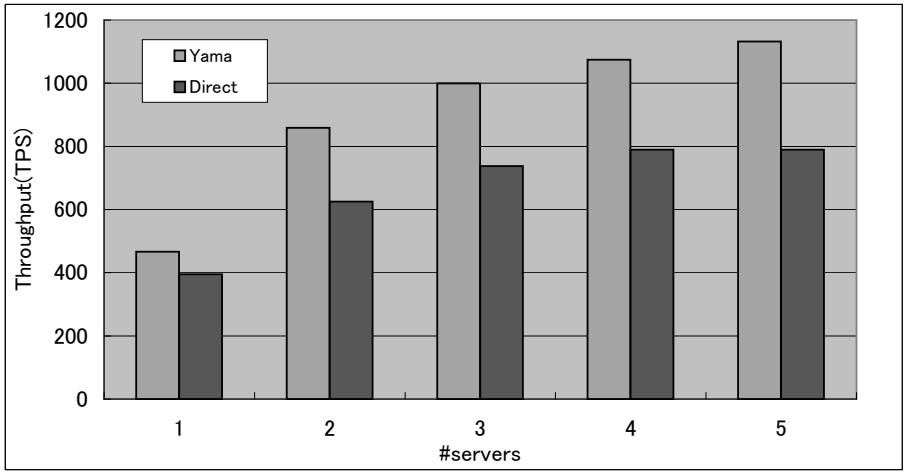

図 3.7: ordering mix の性能比較

割合が増えるにつれ，レプリカ台数を増やした際のスループット向上率が小さくなっている．これは，全レプリカが各更新 SQL を処理する必要があるため，レプリカ台数が増えるにつれ，システム全体の更新 SQL の処理量が増加するためである．

一方，mix 内の更新トランザクションの割合が多くなるにつれ，Yama 実装の方が Direct 実装よりスループットが高くなる．Yama 実装では排他ロックの処理をミドルウェア・サーバ 1 台で行っているのに対し，Direct 実装は全てのレプリカで行う分，オーバヘッドが大きい．反面，Yama 実装は，ロックが衝突した場合，レプリカへのロック対象の問い合わせ処理や，更新対象の問い合わせ処理のオーバヘッドが存在する．しかし，表 3.1 に示すとおり，更新対象の特定はほとんどが主キー値指定であるため，レプリカへのロック対象の問い合わせは少ない．また，更新中のレコードを照会する確率も小さいため，図 3.8 に示すとおり再実行の回数は小さい[4]．つまり，Yama 実装の更新 SQL 処理に対するオーバヘッドの方が，Direct 実装に比べ小さいため，更新トランザクションの割合が多くなるにつれ，Yama 実装の方が性能が高くなる．

[4]全て，1%以下の再実行率である．

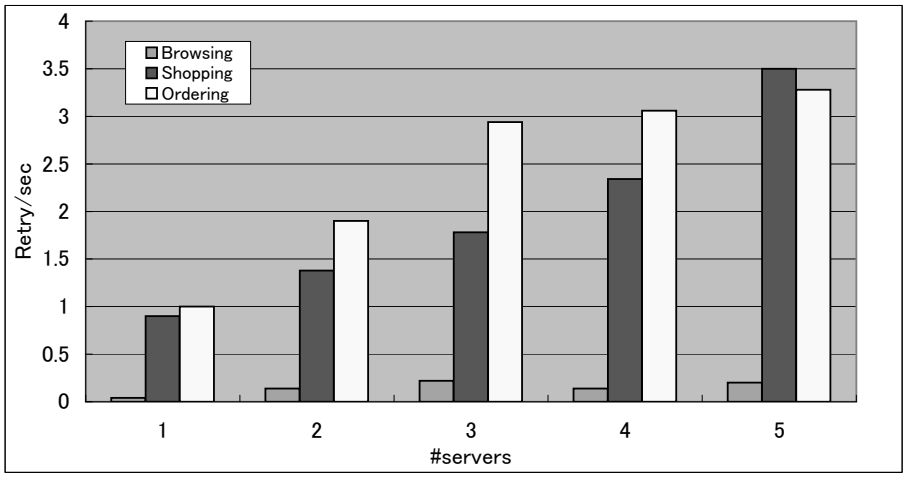

図 3.8: Yama 実装における再実行の回数

3.5 考察

3.5.1 スケーラビリティ

Yama 実装では，排他制御をミドルウェアで行うため，アプリケーションが獲得するロックの量が多くなると，ミドルウェアがボトルネックになる可能性がある．また，アプリケーションによっては，ロックが競合することにより，レプリカ台数を増やした場合でも，スループットが向上しない可能性がある．つまり，3.4 での評価は，レプリカサーバがボトルネックになっていない場合が考えられる．

図 3.9 に，5 台のレプリカ台数を利用した際のミドルウェア・サーバの CPU 利用率を示す．なお，3.4 での全ての評価におけるレプリカの CPU 使用率は，95%以上であった．これらの結果より，ミドルウェア・サーバがボトルネックになっておらず，また，ロックの競合によりトランザクション処理が停滞してないことがわかる．そのため，3.4 での評価におけるボトルネックは，レプリカサーバであることがわかる．

図 3.10 に，3.4 の評価での，Yama 実装におけるミドルウェア内のロック獲得数と，同環境における最大ロック獲得数を示す．上記結果により，同じミドルウェア・サー

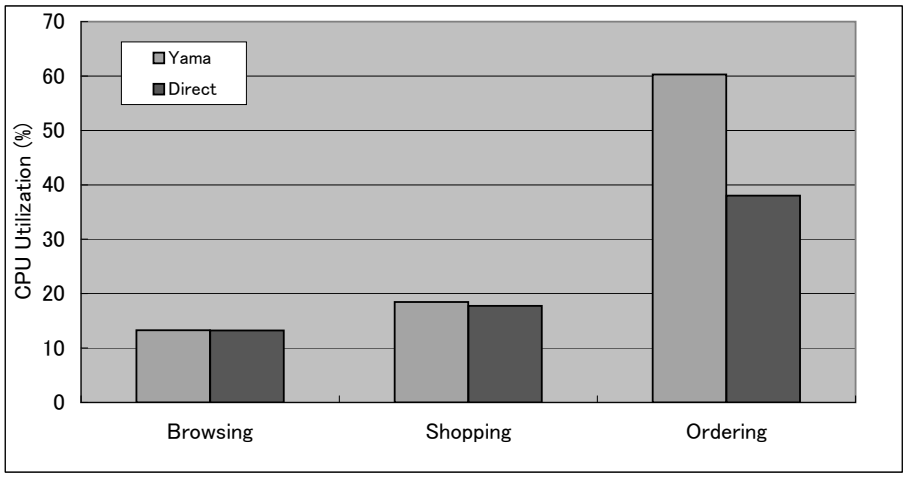

図 3.9: 両実装におけるミドルウェア・サーバの CPU 使用率（レプリカ台数 5）

バと処理性能のより高いレプリカ，もしくは，同じミドルウェア・サーバと 6 台以上のレプリカサーバを利用して，browsing mix では数十倍の，shopping mix では 5 倍程度の，ordering mix では 1.5 倍程度までのスループットの向上が見込まれる．

3.5.2 有効なアプリケーション

3.4 に示したとおり，TPC-W の browsing mix，shopping mix では，Yama 実装，Direct 実装は共に，レプリカ台数に応じてスループットが向上する．つまり，ミドルウェア型の複製手法は，データベースがほぼ照会 SQL のみを処理するアプリケーションに対して，有効であると言える．

しかし，Yama 実装，Direct 実装は，各更新 SQL を全レプリカで処理するため，データベースが更新 SQL を中心に処理するアプリケーションに対しては有効ではない．例えば，1 台のデータベース構成で，データベース・サーバの全ての CPU 時間を更新処理に費やすアプリケーションの場合，レプリカを増やしてもスループットは向上しない可能性がある．

また，Yama 実装では，3.5.1 に示すとおり，ミドルウェア・サーバが処理可能な

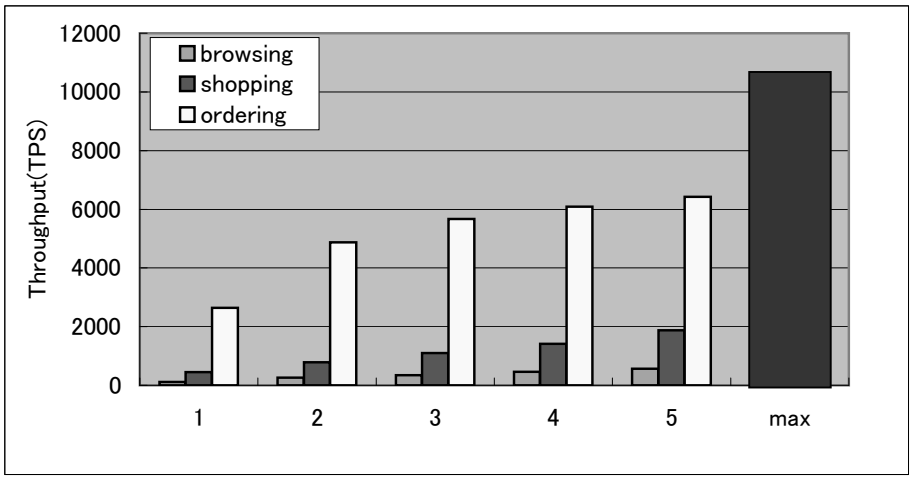

図 3.10: TPC-W での Yama でのロック獲得数と最大ロック獲得数

ロック数に，限界がある．そのため，1 つのトランザクションで，大量のレコードを照会・更新を行うアプリケーションは，ミドルウェア・サーバがボトルネックになる可能性があるため，レプリカを増やしてもスループットは向上しない可能性がある．

一方，ordering mix のように，更新 SQL の処理数は多いが，各更新 SQL が単純なアプリケーションに対しては，Yama 実装は有効である．ordering mix は，50%が更新トランザクションのワークロードだが，1 台のデータベースのシステム構成では，データベースの 80%以上の CPU 時間を，照会 SQL の処理に費やすワークロードである．さらに，Yama 実装では，各レプリカで排他制御の処理を行わない分，1 つの更新 SQL に対するレプリカでの処理量が，1 台のデータベース構成や，Direct 実装と比較して小さい．そのため，上記のようなアプリケーションに対しても，Yama 実装は，レプリカの台数に応じてスループットを向上することが可能である．

ordering mix のような，複雑な照会 SQL と単純な更新 SQL を多用するアプリケーションは，近年多用されている．例えば，オンライン・ショッピングで商品照会を行いながらユーザの商品の照会履歴の保存するアプリケーションや，更新する BPEL プロセスを照会し，BPEL プロセスの状態遷移を保存する SOA アプリケーション

[26] などが，該当する．Yama を利用することにより，これらのアプリケーションのスループットを向上させることが可能となる．

3.6 関連研究

データベースの複製は，非同期複製（Lazy Replication）と，同期複製（Eager Replication）に分類される [23]．非同期複製では，トランザクション中の更新が，トランザクションのコミット後にレプリカに反映されるのに対し，同期複製では，トランザクションのコミット前に複製される．従来，同期型複製は，デッドロックが発生する確率が高くなり，スケーラビリティが悪くなるとされていたが，近年では，高いスケーラビリティが示されることが報告されている [21, 19, 16, 24]．

Mishima と Nakamura は，First-Updater-Win のルールに基づくスナップショット分離性を保障するレプリカを利用して，スナップショット分離性を保障するミドルウェア Pangea を提案している [16]．Pangea が保障するスナップショット分離性は，Oracle[33] 等のベンダーが提供するデータベース同様，トランザクション中の照会処理では，トランザクションが開始した時点でのコミット済みの更新のみが照会できる．Pangea では，Yama 同様，照会処理は 1 台のレプリカで，更新処理は全レプリカで処理され，全レプリカでの LocalTx がコミット次第，GlobaleTx のコミット処理が完了する．また，Pangea では，トランザクション中の最初の処理（照会処理，もしくは，更新処理）はトランザクションの開始処理として定義され，更新トランザクションの場合は全レプリカで，照会トランザクションの場合は 1 台のレプリカで処理される．なお，レプリカの中の 1 台は Leader として定義され，更新トランザクションの開始処理は，Leader で処理された後，他のレプリカで処理される．スナップショット分離性を保障するレプリカ上で，上記のように LocalTx を処理することで，デッドロックが発生することが防がれる．

Pangea は，スナップショット分離性のトランザクションを要求するアプリケーションに特化しているが，他の分離性を要求するアプリケーションには利用できない．一

方，Yamaでは，ANSIが設ける分離性のうち，繰り返し可能読み取り，コミット読み取り，非コミット読み取りを保障可能である．

また，Pangeaでは，First-Updater-Winのルールを用いてデッドロックを防ぐため，複数のGlobalTxが，同時に同じレコードを更新する場合，必ずロールバックが発生することになる．そのため，本来はデッドロックが発生しない場合においても，GlobalTxがロールバックされる可能性がある．一方，Yamaでは，ロックの獲得状態を監視してデッドロックを検知するため，デッドロックの検知のオーバヘッドが発生するが，不必要なロールバックを行うことはない．アプリケーションによっては，GlobalTxのロールバックは障害として警告されることがあるため，不必要なロールバックを回避する必要のあるアプリケーションに，Yamaは有効である．

Schenkelらは，Pangeaと同様，全てのレプリカが同じスナップショットを提供する，ミドルウェア型の複製手法を提案している [24]．Schenkelらの手法では，全順序を保障するブロードキャストを利用して，全レプリカにコミット処理要求を送信することにより，全レプリカで共通のスナップショットの生成を要求する手法を提案している．本手法では，Pangeaとは異なり，ある更新トランザクションの開始処理（コミット処理）が，他の更新トランザクションのコミット処理（開始処理）と排他的に実行されない．しかし，全順序を保障するブロードキャストは，レプリカ台数が増えた際のスケーラビリティが，問題となる可能性がある．

Elniketyらや，Kemmeらは，楽観的トランザクションの要素を用いて，スナップショット分離性や，直列化可能の分離性を実現している [18, 25]．これらの手法は，コミット時に，更新ログを1箇所，もしくは，複数のサーバに送信し，受信したサーバが目的とする分離性が保障されているか否かを調べ（Validation），保障されていない場合は，ミドルウェアがトランザクションをロールバックする．また，Validation方法を変えることで，スナップショット分離性や直列化可能の分離性よりも，緩い分離性を実現可能である．しかし，彼らの手法も，全順序を保障するブロードキャストを仮定している．

今後，上記のような，繰り返し可能読み取り分離性以外の分離性を保障するミドル

ウェアでの複製手法との性能評価を行い，Yama の有用性を示していく予定である．

3.7 まとめ

本章では，ミドルウェアにおいて，ロックを用いた排他制御処理を行う同期複製手法，Yama を提案した．Yama では，SQL の記述を解析して SQL 処理時にアクセスするレコードを解析し，解析できない場合は，必要なロックを特定するためのヘルパー SQL をレプリカに処理要求する．

Yama では，レプリカ間をまたがったデッドロックを回避するため，全ての SQL は非コミット読み取りでレプリカに処理要求する．そのため，ヘルパー SQL の処理結果は，更新中の値が含まれている可能性がある．そこで，Yama では，実行中のトランザクションを監視することでヘルパー SQL の照会結果が更新中の値を含むか判定し，含む場合は更新トランザクションがコミットされた後に再度ヘルパー SQL を実行することで，正確なロック対象を特定する．

本章では，Yama を TPC-W を用いて評価し，照会処理が中心のワークロードに対し，高いスケーラビリティを示すことがわかった．

第4章　結論

本論文は，トランザクションの分離性を保障しながら，関係データベースにおけるトランザクション処理の高速化を実現する，データベース・ミドルウェアの研究をまとめたものである．ミドルウェアにてトランザクション処理の高速化を行うことで，アプリケーションを書き換えや，特別なデータベースを利用することなく，トランザクション処理の高速化を実現可能である．本研究での成果を，以下にまとめる．

- カーソル安定性の分離レベルを保障しながら，トランザクション処理の更新処理を高速化するミドルウェア手法を提案した．本手法では，ミドルウェア内で，複数のトランザクションにおける更新処理を，別トランザクションの1つの更新処理としてデータベースに要求することで，データベースにおける更新処理の高速化を実現する．本論文では，提案手法を簡易的なベンチマークやウェブアプリケーション用ベンチマーク DayTrader に適用し，それぞれ，最大220%，12%の性能向上が可能であることを実証した．

- 繰り返し可能読み取り，コミット読み取り，非コミット読み取り分離性を保障しながら，トランザクション処理の照会処理を高速化するミドルウェア手法を提案した．本手法では，複数のレプリカを利用し，ミドルウェア内で排他制御を行いながら，要求される分離性を保てる用にスケジューリングしながら，照会処理をレプリカにオフロードし，照会処理の高速化を実現する．本論文では，提案手法をウェブアプリケーション用ベンチマーク TPC-W に適用し，レプリカ台数に応じて性能向上可能であることを実証した．

上記の2つのトランザクション処理高速化手法は，組み合わせて利用することで，1つのデータベース・ミドルウェアとして機能することが可能である．1つのミドル

ウェアとして実現することで，トランザクション処理のおける照会処理，更新処理の両処理の高速化を実現することが可能となる．また，上記提案手法以外にも，既存の高速化手法を組み合わせることにより，様々なアプリケーションのワークロードの高速化を実現可能である．さらに，レスポンス時間の短縮や，可用性の向上といった，データベースの機能向上目的としたデータベース・ミドルウェアとも組み合わせて利用することが可能である．今後，既存システムの性能向上や，柔軟なシステム設計のためには，データベース・ミドルウェアは必須の技術となり，様々な研究・開発が進められていくであろう．

謝辞

本論文は，私が早稲田大学大学院基幹理工学研究科　山名早人研究室に在籍していた期間に行われた研究をまとめたものです．研究を進める過程で多くの方々のご指導，ご支援を賜りました．心から感謝致します．

山名早人教授は，私が教授の研究室に所属して以来，学部・修士課程における 4 年間，博士後期課程における 4 年間のみならず，博士後期課程入学前の社会人の時代におきましても，ご指導頂きました．人に役立つ研究，世界に通じる研究を指導していただいた先生は，私に研究者としての道を開き，歩ませてくれました．先生のご指導，ご支援なしに，本研究はあり得ませんでした．

村岡洋一教授，松山泰男教授には，本論文の審査をして頂きました．審査の場にて頂いた，貴重な助言や示唆は，本論文をまとめるためだけでなく，私の今後の研究活動においても有益な，大変貴重なものでした．

山名研究室の諸氏には，議論，助言，示唆という形で，また，研究室での生活でも非常にお世話になりました．石川隼輔氏には，常に最新の技術動向を議論し，コンテストや未踏ソフトウェア等，多くの得がたい経験を共有しました．山田真介氏には，深い情報理論に基づいた議論や，研究活動の有意義さを教えて頂きました．上田高徳氏には，研究室内外の方々との多くの交流の場を設けて頂きました．

また，本論文執筆において，日本アイ・ビー・エム（株）東京基礎研究所の皆様に，非常にお世話になりました．丸山宏氏，小坂一也氏，山本学氏は，私の社会人博士後期過程の入学を，快く受諾して頂きました．小野寺民也氏は，企業の研究者としての礎を築かせて頂き，また，博士後期過程在籍中も，温かく支援して頂きました．根山亮氏，小柳光生氏，小澤陽介氏，竹内幹夫氏，宮下尚氏，河内谷清久仁氏，榎美紀氏には，日頃の業務において，多くの助言を頂きました．

他にも，多くの方々からご支援，ご助言を頂きました．私の研究活動を可能にして下さった多くの方々に，深く感謝致します．

最後に，今日まで私の研究生活を見守って下さった両親に感謝致します．そして，あらゆる面から私を支え続けてくれる涌井道子さんに感謝致します．

参考文献

[1] D. Gawlick, and D. Kinkade, "Varieties of Concurrency Control in IMS/VS Fast Path", *In Proceedings of IEEE International Conference on Data Engineering*, vol.8, no.2, pp.3-10, 1985.

[2] T. K. Sellis, "Multiple Query Optimization", *Journal of ACM Transactions on Database Systems (TODS)*, vol.13, no.1, 1988.

[3] N. N. Dalvi, S. K. Sanghai, P. Roy, and S. Sudarshan, "Pipelining in Multi-Query Optimization", *Journal of Computer and System Sciences*, vol.66, no.4, pp.728-762, 2003.

[4] K. O'Gorman, A. E. Abbadi, and D. Agrawal, "Multiple Query Optimization in Middleware using Query Teamwork", *In Proceedings of Software-practice & Experience*, vol.35, no.4, pp.361-391, 2005.

[5] T. Ohmori, M. Kitsuregawa, and H. Tanaka, "Scheduling Batch Transactions on Shared-nothing Parallel Database Machines: Effects of Concurrency and Parallelism", *In Proceedings of IEEE 7th International Conference on Data Engineering*, pp.210-219, 1991.

[6] T. Ohmori, M. Kitsuregawa, and H. Tanaka, "Concurrency Control of Bulk Access Transactions on Shared Nothing Parallel Database Machines", *In Proceedings of IEEE the 6th International Conference of Data Engineering*, pp.476-485, Febrary 1990.

[7] D. Agrawal, A. El Abbadi, and A. E. Lang, "The Performance of Protocols Based on Locks with Ordered Sharing", *In Proceedings of IEEE Transactions on Knowledge and Data Engineering*, vol.6 , Issue 5, pp.805-818, 1994.

[8] I. T. Bowman, and K. Salem, "Optimization of Query Streams using Semantic Prefetching", *ACM Transactions on Database Systems (TODS)*, vol.25, no.4, pp.457-516, 2000.

[9] Q. Yao, and A. An, "Using User Access Patterns for Semantic Query Caching", *In Proceedings of the 14th International Conference on Database and Expert Systems Applications*, pp.737-746, 2003.

[10] D. Shasha, F. Llirbat, E. Simon, and P. Valduriez, "Transaction Chopping: Algorithms and Performance Studies", *ACM Transactions on Database Systems (TODS)*, vol.20, no.3, pp.325-363, 1995.

[11] M. Welsh, D. Culler, and E. Brewer, "SEDA: An Architecture for Well-Conditioned, Scalable Internet Services", *In Proceedings of the 18th Symposium on Operating Systems Principles (SOSP'01)*, 2001.

[12] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki, "QPipe: A Simultaneously Pipelined Relational Query Engine", *In the Proceeding of the 2005 ACM SIGMOD International Conference on Management of Data*, pp.383-394, 2005.

[13] S. Harizopoulos, and A. Ailamaki, "StagedDB: Designing Database Servers for Modern Hardware", *In Proceedings of IEEE International Conference on Data Engineering*, Vol 28, no.2, pp.11-16, 2005.

[14] S. Harizopoulos, and Ailamaki, "STEPS: Towards Cacheresident Transaction Processing", *In the Proceedings of the 30th International Conference on Very Large Data Bases*, pp.660-671, 2004.

[15] E. Cecchet, G. Candea, and A. Ailamaki, "Middleware-based Database Replication: The Gaps between Theory and Practice", *In Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pp.739-752, 2008.

[16] T. Mishima, and H. Nakamura, "Pangea: An Eager Database Replication Middleware Guaranteeing Snapshot Isolation without Modification of Database Servers", *In Proceedings of the 37th International Conference on Very Large Databases*, pp.424-435, 2009.

[17] K. Daudjee, and K. Salem, "Lazy Database Replication with Snapshot Isolation", *In Proceedings of the 32nd International Conference on Very Large Databases*, pp.424-435, 2004.

[18] S. Elnikety, F. Pedone, and W. Zwaenepoel, "Database Replication Using Generalized Snapshot Isolation", *In Proceedings of the 24th Symposium on Reliable Distributed Systems*, pp.73-84, 2005.

[19] S. Wu, and B. Kemme, "Postgres-R(SI): Combining Replica Control with Concurrency Control based on Snapshot Isolation", *In Proceedings of IEEE the 21st International Conference on Data Engineering*, pp.422-433, 2005.

[20] Y. Lin, B. Kemme, M. Patino-Martinez, and R. Jimenez-Peris, "Middleware based Data Replication Providing Snapshot Isolation", *In Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, 2005.

[21] E. Cecchet, J. Marguerite, and W. Zwaenepoel, "C-JDBC: Flexible Database Clustering Middleware", *In Proceedings of the Annual Conference on USENIX Annual Technical Conference*, 2004.

[22] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil, "Critique of ANSI SQL Isolation Levels", *In Proceeding of the 1995*

ACM SIGMOD International Conference on Management of Data, pp.1-10, 1995.

[23] J. Gray, P. Helland, P. O'Neil, and D. Shasha, "The Dangers of Replication and a Solution", *In Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pp.173-182, June 1996.

[24] R. Schenkel, N. Weikum, G. Weibenberg, and X. Wu, "Federated Transaction Management with Snapshot Isolation", *Lecture Notes in Computer Science*, vol.1773, pp.1-25, 2000.

[25] B. Kemme, F. Pedone, G. Alonso, A. Schiper, and M. Wiesmann, "Using Optimistic Atomic Broadcast in Transaction Processing Systems", *In Proceedings of IEEE Transacitons on Knowledge and Data Engineering*, vol.15, no.4, , pp.1018-1032, 2003.

[26] IBM Corp., IBM WebSphere Process Server (オンライン),
入手先 <http://www-01.ibm.com/software/integration/wps/> (参照 2011-01-02).

[27] Apache, DayTrader (オンライン),
入手先 <http://cwiki.apache.org/GMOxDOC20/daytrader.html>(参照 2011-01-02).

[28] IDC Excerpt: Worldwide Database Management Systems 2009-2013 For ecast and 2008 Vendor Shares

[29] IBM Corp., DB2,
入手先 <http://www-306.ibm.com/software/data/db2/>(参照 2011-01-02).

[30] Microsoft Corp., SQL Server 2005 Beta 2 Snapshot Isolation.
入手先 <http://msdn.microsoft.com/en-us/library/ms130975.aspx>, (参照 2011-01-02).

[31] The Transaction Processing Performance Council, TPC-W, a Transactional Web, E-Commerce Benchmark, 入手先 <http://www.tpc.org>(参照 2011-01-02).

[32] ANSI X3.135-1992, American National Standard for Information Systems - Database Language - SQL, 1992.

[33] Oracle Corporation, "Concurrency Control, Transaction, Isolation and Serializability in SQL92 and Oracle7", 1995. Whitepaper.

研究業績

種類別	題名	発表・発行掲載誌名	連名者
論文誌			
	ロック制御型同期複製ミドルウェアの提案	電子情報通信学会和文論文誌 D-I データ工学特集号, vol.J94-D, no.3, 2011 年 3 月	堀井 洋，小野寺 民也，山名 早人
	OLTP のための自動グループコミット手法の提案	電子情報通信学会和文論文誌 D-I データ工学特集号, vol.J93-D, no.3, 2010 年 3 月	堀井 洋，小野寺 民也，山名 早人
講演			
国際会議	Transactional Optimistic Active Replication	Proceedings of the 2nd international conference on Ubiquitous information management and communication, pp. 94-100, Seoul, Korea，Jan. 2008	Hiroshi Horii, Hayato Yamana
シンポジウム	Transactional Optimistic Active Replication	データベースと Web 情報システムに関するシンポジウム 2007，2007 年 11 月	堀井 洋，山名 早人
研究会	実測に基づいた MPI プログラムの実行時間予測手法，情処研報 (HPC)	vol.2001, no.88，2001 年 10 月	堀井 洋，山名 早人
	MPI プログラムの簡易実行による実行時間予測手法の評価	情処研報 (HPC)，vol.2003, no.83，2003 年 8 月	岩渕 寿寛，堀井 洋，山名 早人

種類別	題名	発表・発行掲載誌名	連名者
その他			
論文	BlueStar: A Federation-based Approach to Building Internet-Scale Data Centers	IBM Journal of Research and Development, vol.53, no.4, Nov. 2009	Anindya Neogi, Ajay Mohindra, Balaji Viswanathan, Takayuki Kushida, Hiroshi Horii
	PAXOS Consensus による OLTP 高可用化機構の提案とその実装	DBSJ Letters vol.5, no.2, pp.81-84, 2006 年 9 月	堀井 洋, 田井 秀樹, 山本 学
講演	PAXOS Consensus による OLTP 高可用化機構の提案とその実装	夏のデータベースワークショップ 2006, 2006 年 7 月	堀井 洋, 田井 秀樹, 山本 学