並列トランザクション処理における排他制御方法の比較

ｓ０３３０８５
皆木　健太
計算機科学講座
田中研究室

2007 年 2 月 6 日

目次

第1章　序論

DBに対しデータの更新処理が並列に行われるDBシステムでは、DBの整合性を保つために、直列可能性を実現しなければならない。そして直列可能性を実現するための代表的な方法の一つに排他制御がある。例えばJava言語では、主にSynchronized[1]ブロックやJDK5.0（JavaTM 2 Platform Standard Edition 5.0 Development Kit）から新たに加わったJava.util.concurrentパッケージ[2]を使い、多数存在するプロセスからDBMSへアクセスに対する排他制御を行う事により直列可能性を実現できる。また、従来使われてきたDBMSの機能の一つであるLock機能[3][4]を使えば、DBMSとDBとの間で排他制御を実現できる。

今回は、テスト用PCとDBMSとの間で、Java Synchronizedを使う方法とJava.util.concurrent.atomicパッケージ[5]を使い排他制御を行う方法、DBMSの排他制御機能とストアドプロシージャ[3][4]を使い、DBMSとDBとの間で排他制御を行う方法を取り、3つの方法の違いによってどれほど性能に差が現れるか、また直列可能性を実現できているか調査を行った。

第2章　直列可能性

2.1 直列可能性の概要[6]

DBMS は複数の応用目的で共有されるデータを管理するため、多くのトランザクション要求（一連の関連する複数の処理を一つの処理単位としてまとめたものをトランザクションという）を処理しなければならない事が多い、トランザクションの処理中には、ディスクとの入出力待ちや、ユーザからの入力待ちなど待ち時間が発生するため、このような待ち時間に、より多くのトランザクションを実行させるため、通常 DBMS は、複数のトランザクションからの基本操作を、各トランザクションの作業領域を分割して実行する。これをトランザクションの並列処理と言う。トランザクションが並列処理を行う場合、複数のトランザクションから、一つの共有データに対してアクセスを行い、その共有データに対して、各々のトランザクションが共有データの更新処理を行う事がありうる。このような場合、共有データの不整合が発生しないための一つの基準として、トランザクション T_1, T_2・・・T_n を並行処理した結果が、それらを何らかの順序で逐次処理した時の結果と一致することを保証する性質の事を直列可能性と言う。

2.2 直列可能性の実現例

トランザクションの並列処理において直列可能性を実現する例として、一つの口座に対してほぼ同時に、お金の振り込み処理と引き出し処理が行われた時を考える。この並列処理に対して、直列可能性が実現されていない例を図 2.1、排他制御を行うことで直列可能性が実現されている例を図 2.2 とする。

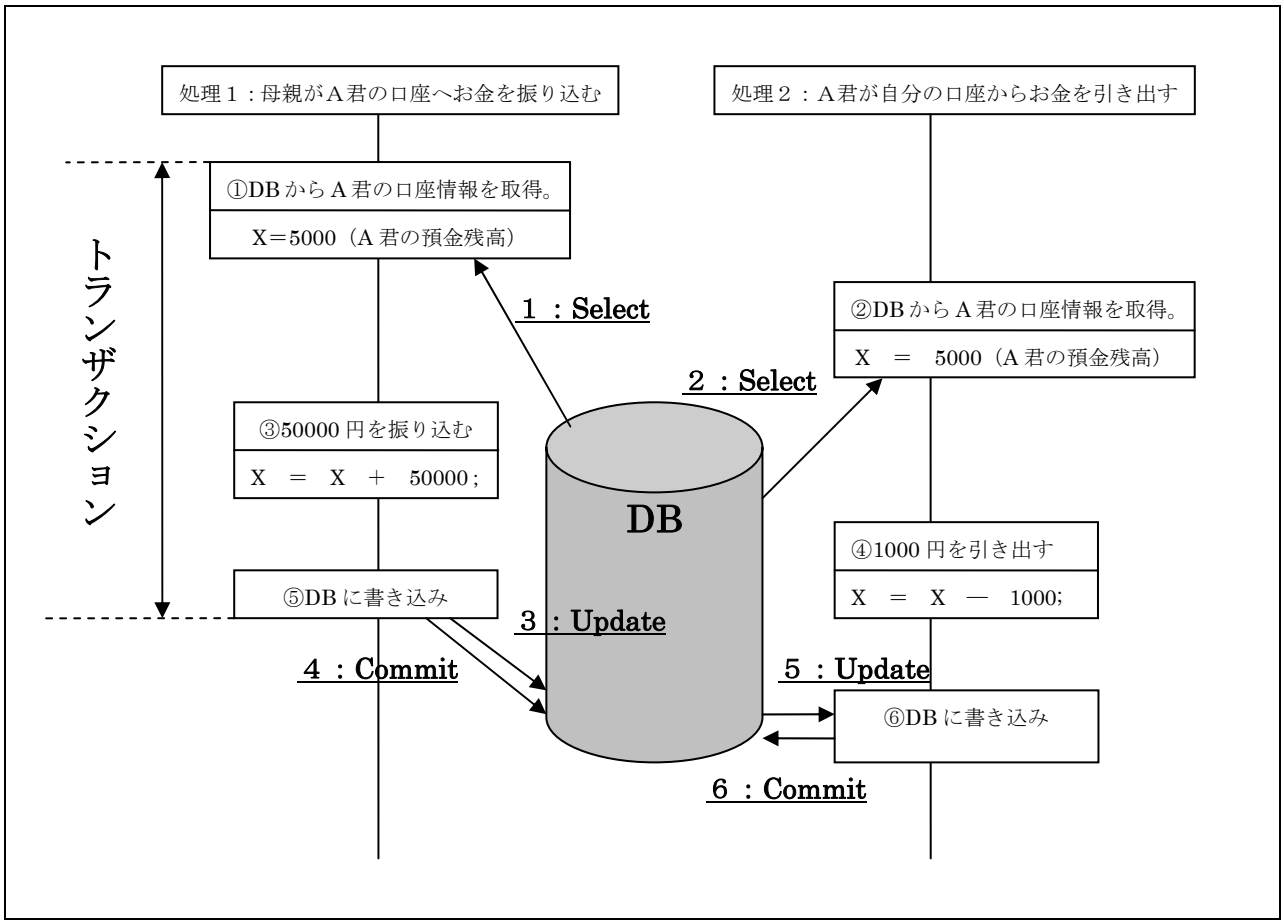


図 2.1 直列可能性の実現　失敗例

図 2.1 の例では、処理１で 50000 円振込み、ＤＢのＡ君の口座の預金残高を 55000 円に更新した上から、処理２において、元々の預金残高 5000 円から 1000 円引き出した、預金残高 4000 円の情報をＤＢに更新してしまうため、処理１において、振り込んだ 50000 円が消失してしまう。よってデータの整合性が保たれておらず、直列可能性を実現できていない。

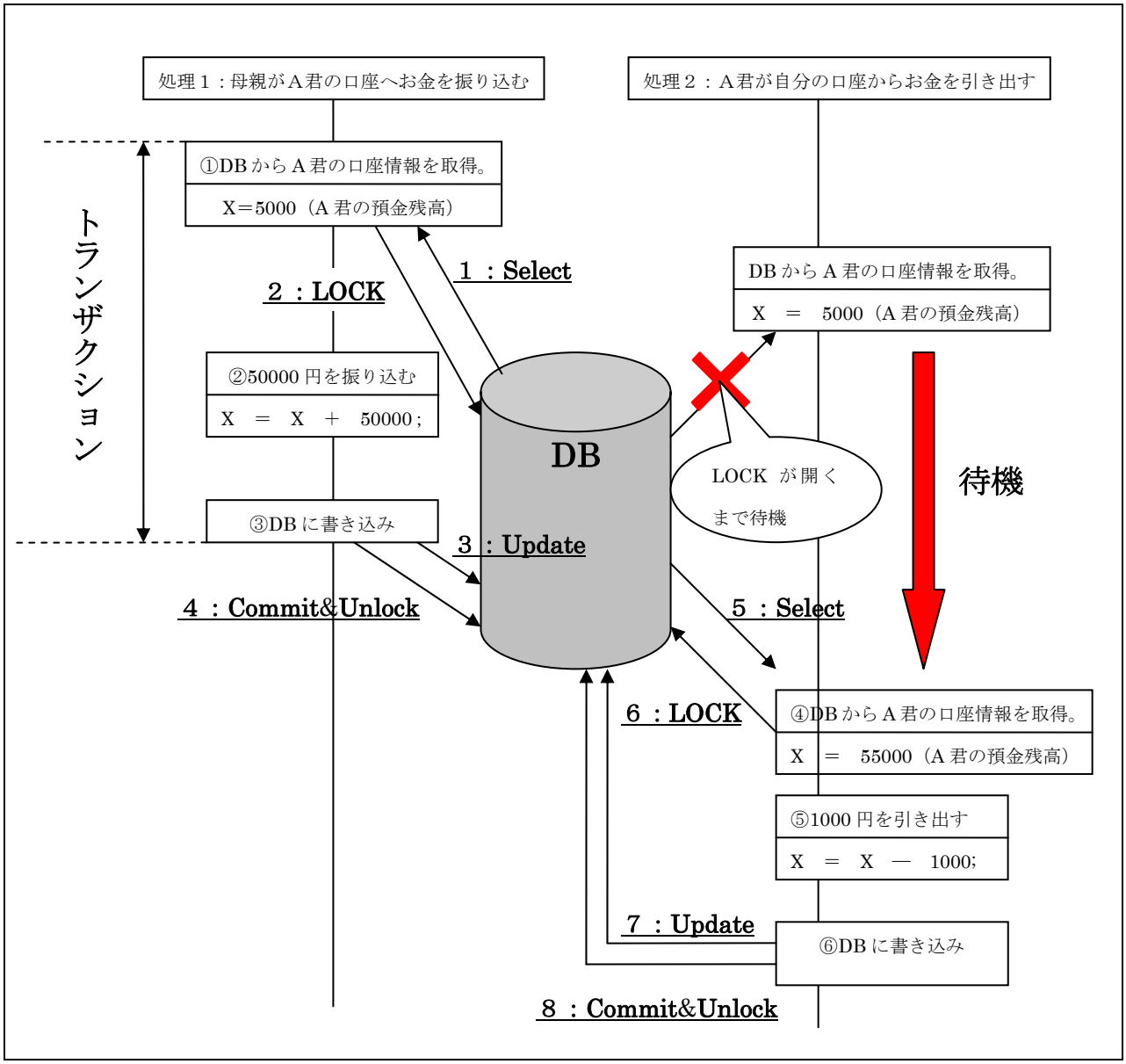


図 2.2 直列可能性の実現　成功例

図 2.2 の例では、処理１においてＤＢから預金残高の情報を取得する際に、他のトランザクション処理がＡ君の口座情報に対してアクセスできないようにＬｏｃｋをかけている。これによって処理２は、処理１のトランザクション処理が終了するまでＤＢのＡ君の口座情報にアクセスすることができないため、処理１において預金残高を 55000 円に更新した直後、預金残高 55000 円から 1000 引き出した、預金残高 54000 円をＤＢに更新することで、2 つのトランザクションがほぼ同時に一つの口座に対して、更新処理を行ってもＤＢのデータの整合性が保たれる、直列可能性を実現する事ができる。

第 3 章　排他制御

3.1 本研究で用いた排他制御の方法

前章で述べた直列可能性を実現するための代表的な方法として、排他制御が挙げられる。排他制御の方法としては、①クライアントとＤＢサーバ間で、クライアントからのＤＢサーバへのアクセスを排他的に行う方法（図 3.1）と、②ＤＢサーバからＤＢへのアクセスを排他的に行う方法（図 3.1）が考えられる。本研究では、①の方法として Java 言語の Synchronized ブロックを用いる方法、Java.util.concurrent.atomic パッケージを使い、Wait-Free かつ Lock-Free なキューを作成し、直列可能性を実現する方法、②の方法として、DBMS の排他制御機能とストアドプロシージャを用いる方法の３つの方法で排他制御を実現した。

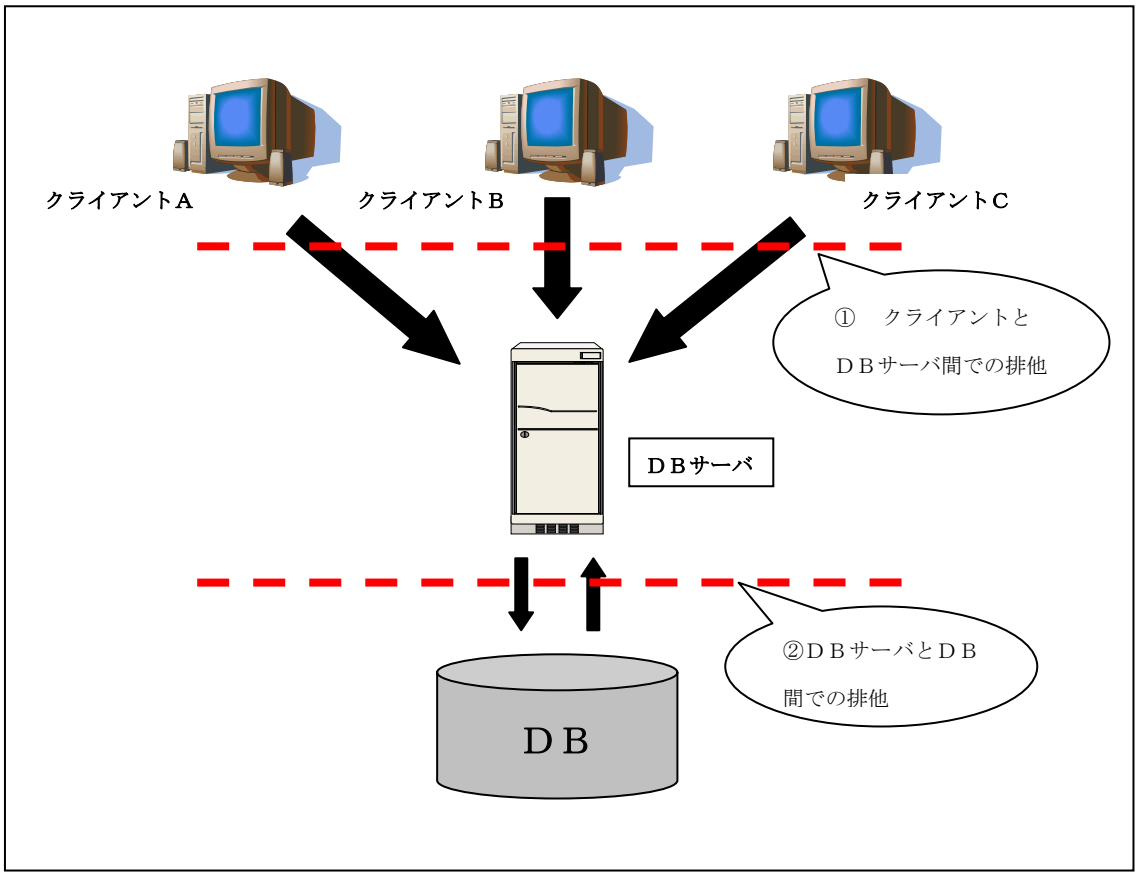


図 3.1 実験で行った排他制御の大まかな区分

3.2 Java Synchronized の概要[1]

Java Synchronized とは、Java 言語におけるスレッド間で同期を取るためのメカニズムで、マルチスレッドシステムにおいて、同時に複数のスレッドが同じオブジェクトを更新する際に、オブジェクトの状態が不正にならないようにするため、同時に複数のスレッドが実行してはならない一部のコード（クリティカルセクション）の排他制御を行う事により保護する機能である。

3.3 Java.util.concurrent.atomic パッケージ[5],[8],[9]

3.3.1 Java.util.concurrent.atomic パッケージの概要

JDK5.0（JavaTM 2 Platform Standard Edition 5.0 Development Kit）から新たに加わったパッケージで、このパッケージに属するクラスを使用することで、複数のスレッドからの共有オブジェクトに対する値の変更処理を Lock-Free かつスレッドセーフに行う事を可能にする。

このパッケージを使用することによって、CAS（Compare And Swap）命令のような、CPU の専用のアトミックな命令を Java から使用することができるようになり、これにより Wait-Free,Lock-Free のアルゴリズムを Java で書けるようになった。

JDK5.0 以前までは、スレッド間の同期処理は、主に Synchronized を使い行われてきたが、この Java.util.concurrent.atomic パッケージを使用すれば、Synchronized のようにスレッド間の共有資源に対してロックをかける必要がないため、Synchronized を使用するより性能が良く、Synchronized のような同期処理プログラムで起こりえる、デットロックなどの問題が起こりえないプログラムを書くことができる。

3.3.2 CAS(Compare And Swap)命令[7]

CAS 命令には、「メモリー位置 (V)」、以前取得したメモリアドレス内に格納されていた値である「古い値 (A)」そしてこれから取得する今現在、メモリアドレス内に格納されている値である「新しい値 (B)」という 3 つのオペランドがあり、プロセッサーは、実際にメモリー内にある値と、以前取得した古い値が一致する場合には、メモリー位置を新しい値にアトミックに更新し、一致しない場合には何もしない。そして、いずれの場合でもプロセッサーは、CAS 命令の前にその位置にあった値を返す。このようなプロセッサー専用のアトミックな命令を CAS 命令という。

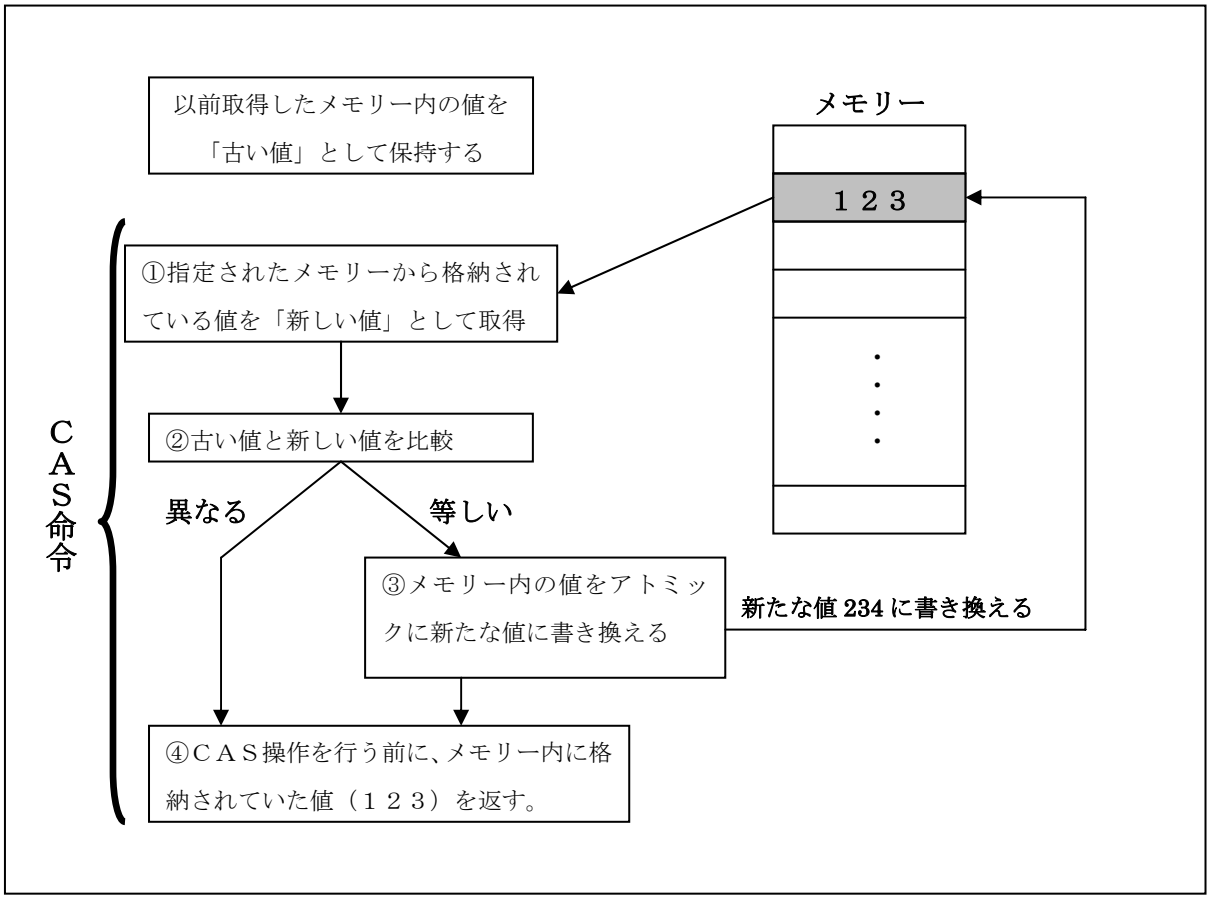


図 3.3.2 CAS 命令の概要

3.3.3 Wait-Free アルゴリズムの概要[7]

マルチスレッドシステムにおいて、何らかの原因により、あるスレッドに遅れが生じたり、そのスレッドが停止してしまった場合においても、その他のスレッドは、継続して処理を進めるような性質を持つアルゴリズムで、Wait-Freeアルゴリズムは他のスレッドのアクションやタイミング、インターリーブ、またはスピードによらず、各スレッドはそのスレッドのステップとして指定された数の演算操作を正しく行うことが保証されている。

3.3.4 Lock-Free アルゴリズムの概要[7]

Lock-free アルゴリズム とは、その時その時の瞬間に、動いているスレッドは一つで、他のスレッドは停止しているというロックによる排他制御とは違い、全ての場合においてスレッドが固まらない、実行終了していない全てのスレッドは、常に動き続けるというアルゴリズムである。

3.3.5 Wait-Free、Lock-Free アルゴリズムの具体例

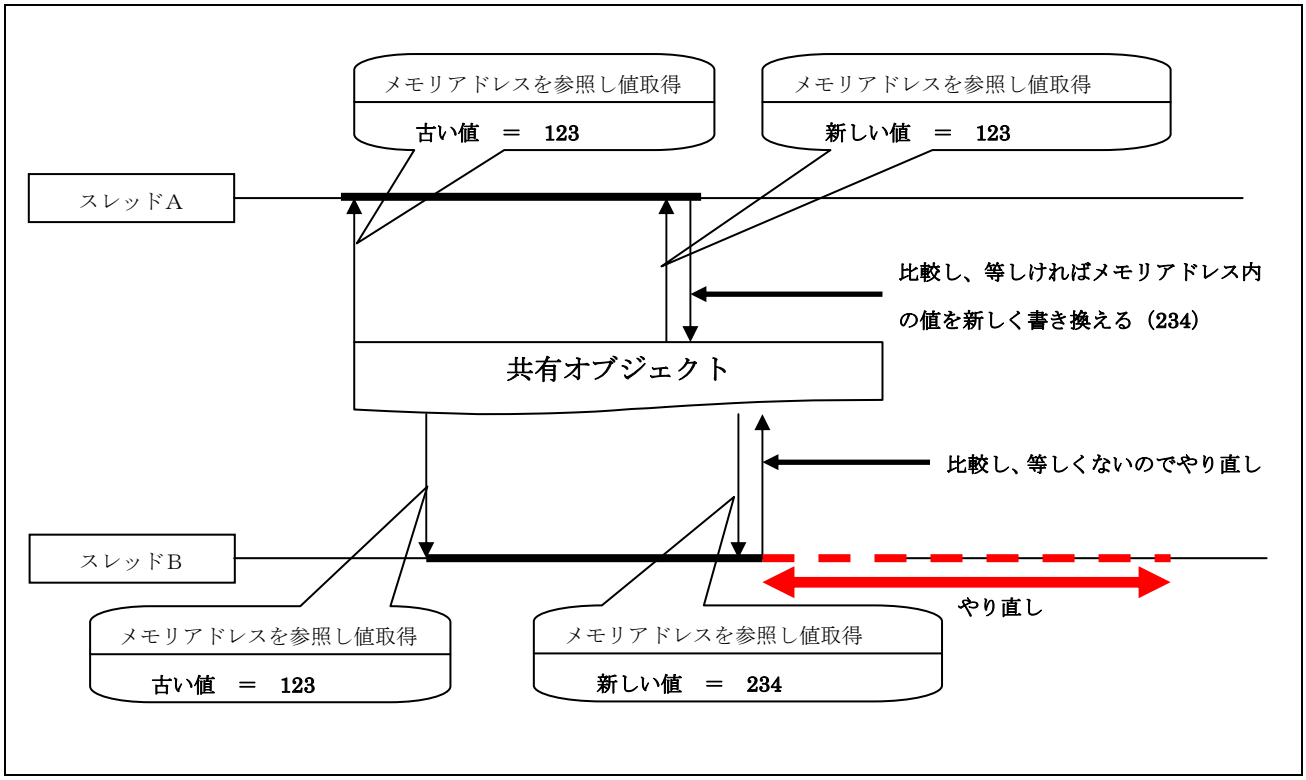


図 3.3.5 Wait-Free,Lock-Free なプログラムの例

上図の例では、まずスレッドAが、共有オブジェクトへアクセスする。するとスレッドAは、ある指定されたメモリアドレス参照し、値(123)を取得して、その値(123)を古い値としてスレッドAで保持する。その後、スレッドBが、スレッドAの処理が終了しない内に、共有オブジェクトにアクセスしたとする。スレッドBにおいても、スレッドAと同じように値(123)を取得し、古い値として保持する。スレッドAの共有オブジェクトへのアクセスが終了する直前に、スレッドAは再び指定されたメモリアドレスを参照し、値(123)を取得して、その値(123)を今度は新しい値として保持し、その後、古い値と新しい値を比較する。上図の例の場合、スレッドAの古い値と新しい値は等しいため、メモリアドレス内の値を新しい値に書き換え(234 に書き換える)、スレッドAは次の処理に移る。次にスレッドBが共有オブジェクトへのアクセスを終了する直前にもう一度メモリアドレスを参照し値(234)を取得し、古い値(123)と新しい値(234)を比較する。この場合、古い値と新しい値は等しくないため、スレッドBでは、共有オブジェクトを使って行った処理をもう一度最初からやり直す。

3.3.6 Wait-Free,Lock-Free アルゴリズムの利点[7]

Wait-Free、Lock-Free アルゴリズムを用いる事の利点を以下に挙げる。

① ロックで排他制御を行う際に起こりえる、デットロック※1、ライブロック※2、優先順位の逆転※3といった様々な弊害を防ぐ事ができる。
② ある決められた有限時間内に処理が終了する事が保証されている。
③ 多くの場合においてロックのアルゴリズムを用いる時よりも、性能が良い。

Wait-Free,Lock-Freeなマルチスレッドシステムでは、排他制御を行う際、ロックのように、スレッドが停止させる事が無いため、①の利点が成り立つ。また、Wait-Freeの他のスレッドの状態に関わらず、スレッドのステップとして指定された数の演算操作を正しく行うことを保証する性質により、②の利点が成り立つ。③の利点だが、極端に多くのスレッドが競合を起こす場合を除いて、ロックのアルゴリズムよりも性能が良い事が認められている。

(※1)デットロック：2つ以上のスレッドあるいはプロセスなどの処理単位が互いの処理終了を待ち、結果としてプログラムが停止してしまうこと。

(※2)ライブロック：資源獲得の処理が進行しているにも関わらず、どのユーザも資源が獲得出来ない状況。

(※3)優先順位の逆転：スケジューリングにおいて優先順位の低いタスクが優先順位の高いタスクが必要としているリソースを共有しているときに発生する状態である。低い優先順位のタスクがそのリソースを解放するまで高い優先順位のタスクが実行をブロックされるため、実質的に二つのタスクの優先順位が逆転する

3.4 DBMS による排他制御

3.4.1 DBMS の概要[3][4]

DBMS (データベース管理システム)とは、共有データとしてのデータベースを管理し、データに対するアクセス要求に応えるソフトウェアの事である。データの形式や利用手順を標準化し、特定のアプリケーションソフトから独立させることができる。DBMS は様々な機能を有しており、本研究で用いた、排他制御機能、ストアドプロシージャもその機能の中の一つである。

DBMS の中でも RDBMS (Relational DataBase Management Syaytem)と言われる管理方式がある。RDBMS とは、1 件のデータを複数の項目(フィールド)の集合として表現し、データの集合をテーブルと呼ばれる表で表す方式で、ID 番号や名前などのキーとなるデータを利用して、データの結合や抽出を容易に行なうことができる管理システムである。本研究では、純国産 RDBMS である HITACHI HiRDB Version 7 を用いている。

3.4.2 HiRDB の排他制御機能[3][4]

本研究で用いた DBMS、 HITACHI HiRDB Version 7 の主な排他制御機能はロックにより実現されていて、5 つの排他制御モードから、排他制御のレベルを選択することができる。

1. **共用モード (PR : Protected Retrieve)**
   1 トランザクションだけが排他資源を占有し，ほかのトランザクションには参照だけを許すモード。
2. **排他モード (EX : EXclusive)**
   1 トランザクションだけが排他資源を占有し，ほかのトランザクションには参照，追加，更新，及び削除を許さないモード。
3. **意図共用モード (SR : Shared Retrieve)**
   ある資源に対して共用モードで排他を掛けた場合，その資源の上位の資源に対して掛かるモード。ほかのトランザクションにも排他資源の参照，追加，更新，及び削除を許すモード。
4. **意図排他モード (SU : Shared Update)**
   ある資源に対して排他モードで排他を掛けた場合，その資源の上位の資源に対して掛かるモード。ほかのトランザクションにも排他資源の参照，追加，更新，及び削除を許すモード。

5. **共用意図排他モード（PU：Protected Update）**
   参照，追加，更新，及び削除を許されているモード。ほかのトランザクションには参照だけを許すモード。

また、HiRDB は排他資源という単位で排他を掛けて，不正に参照されたり，更新されたりすることを防止する。そして排他制御を行う単位として以下の４つから選択することができる。

1. **インナレプリカ構成管理単位**※1
2. **RD エリア**※2**単位**
3. **表単位**
4. **ページ**※3**単位**
5. **行単位**

本研究では、排他制御のモードを排他モードとし、排他制御の単位を行単位として、実験を行った。

(※1) インナレプリカ構成管理単位：HiRDB の全ての RD エリアを対象とした構成単位

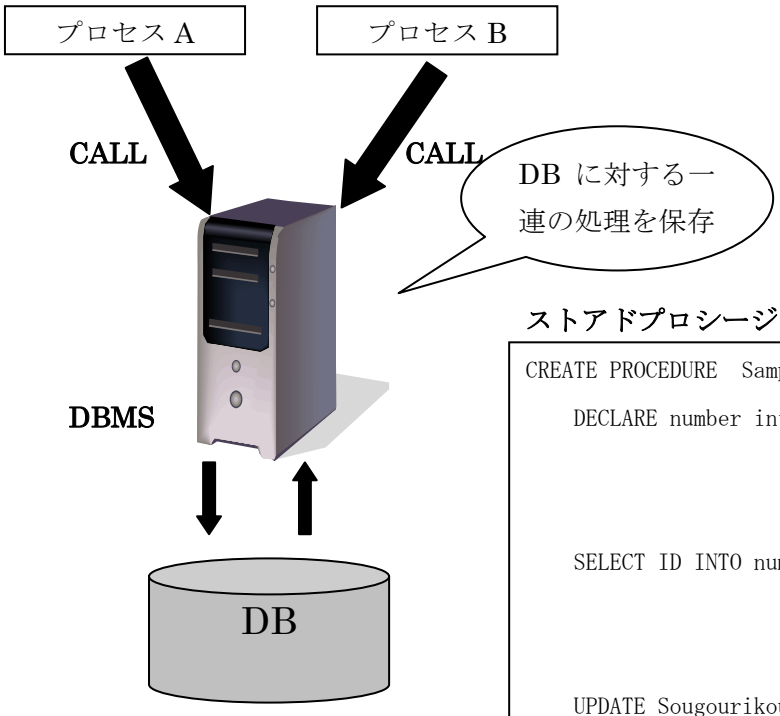
(※2) RD エリア:データの格納単位の一つで,1～16 個の HiRDB ファイルから構成される。RD エリアの中に、表や、インデックス、ストアドプロシージャーなどのデータが格納される。

(※3) ページ：データの格納単位の一つで，データベースの入出力動作の最小単位でページの種類を以下に示す。

- **データページ**：表の中の行を格納するページ。
- **インデクスページ**：インデクスのキー値を格納するページ。
- **ディレクトリページ**：RD エリアの状態の管理情報を格納するページ。

3.4.3 ストアドプロシージャ

ストアドプロシージャとは、SQL 又は Java で記述した一連のデータベースのアクセス手順を手続きとしてデータベースに登録しておく機能を示す。0 個以上の入力，出力又は入出力パラメタを持ち，SQL の CALL 文で呼び出せる。ストアドプロシジャは，CREATE PROCEDURE 又は CREATE TYPE 中の手続き本体で定義できる。SQL 又は Java で記述されたデータベース操作は，定義時にコンパイルされ，アクセス手順を記述した SQL オブジェクトが作成されて，定義情報とともにデータベースに格納される。



ストアドプロシージャの定義

```
CREATE PROCEDURE  Sample(IN  Number  ・・・) {
    DECLARE number integer;  ←変数の定義
                    ・
                    ・
    SELECT ID INTO number FROM Sougourikou;
                    ・
                    ・
    UPDATE Sougourikou set ID = Number
}
```

トランザクション

上図では、右図のように定義されたストアドプロシージャを DBMS に保存し、各プロセスは、DBMS に蓄積されたストアドプロシージャを CALL 文で呼び出すだけで、ストアドプロシージャに記述された一連の処理を実行することができる。

第4章　実行時間比較実験の概要

4.1 実験の環境

テスト用 PC

CPU：pentium4　3.20GHz

RAM：504MB

OS　：Windows XP Professional SP2

DB サーバ

CPU：Celeron　2.0GHz

RAM：515MB

OS　：Windows 2000 Server

DBMS

HITACHI HiRDB Version 7

4.2 実験の概要

本研究では、JDBC を用いて DBMS とアクセスを行った。そして

① Synchronized で、同期を取る
② Java.util.concurrent.atomic パッケージを用いて作成した、Wait-Free かつ Lock-Free なキューを使う
③ DBMS の行単位の排他制御機能とストアドプロシージャを使う

の３つの方法を使い、直列可能性を実現したトランザクションの並列処理プログラムを作成し、初期テーブルとして用意した 10000 行のテーブルに対してランダムに作成した大量の SQL 文を実行させ、並列処理の実行開始から実行終了までの時間を計測し、実行結果のテーブルが、全て同じテーブルになっているか確かめた。

なおこの実験では、実行時に SQL 文エラーを発生させないよう考慮し、ランダムに作成した SQL 文を、次のように並べ直し、各 SQL 文を実行するスレッドへ渡している。

・並び替えの例

１：ＳＱＬ文から、口座ＩＤを取り出す。
２：「口座ＩＤ　％　スレッド数」を計算して、結果得られた値によりどの配列に格納するか決定する。

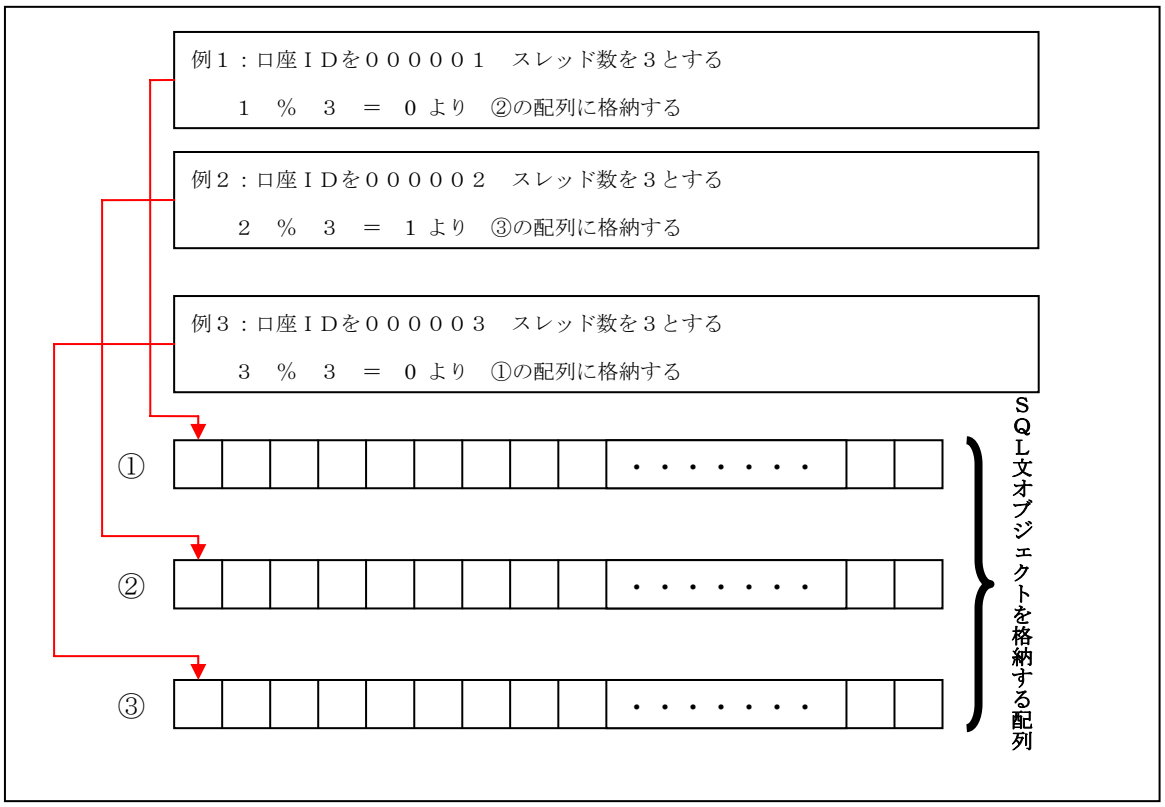


図 4.2：SQL 文の並び替え例（スレッド数が３の時を考える）

このように、一度、並び替えてやる事で、Synchronized、Java.util.concurrent.atomic、ＤＢＭＳのＬＯＣＫ機能＋ストアドプロシージャの３つの方法で直列可能性を実現した並列処理の結果が等しくなるため、正確な実験を行う事ができる。

4.3 実験で使用するテーブルの概要

今回、テーブルはテーブル名を「銀行」とし、属性は「口座 ID」、「苗字」、「名前」、「預金残高」、「暗証番号」の５つとした。主キーは「口座 ID」とし、主キーに対して INDEX 付けを行った。そして初期テーブルには、10000 行データを格納している。

図：初期テーブルの例

口座 ID	苗字	名前	預金残高	暗証番号
000001	山下	信二	123456	0912
000002	田村	新之助	987654	1892
000003	山田	太郎	600500	2689
		・		
		・		
		・		
010000	園崎	魅音	865400	8141

初期表の作成では、10000 行のテストデータを作成し、そのテストデータを基に HirDB の pdload コマンド[3][4]を使用して、Create Table 文で定義した表に対して、10000 行のデータを一括挿入している。なお DBMS には ID を s033085 として登録し、この ID を使い実験を行った。

・表の定義

```
Create Table ginkou(口座 ID char(6),苗字 char(4),名前 char(4),預金残高 int,暗証番号 int)
```

・pdload コマンドの使用例

```
% pdload –u s033085 –n 10000 –i c ginkou C:¥hirdb¥load.txt
```

-u：ユーザ名
-n：一度に挿入する SQL 文の数
-i：INDEX の作成モードを指定
　c　行を一括挿入した後 INDEX を一括して付けるように指定
　n　インデクス情報のみインデクスファイルに出力するように指定
　s　行を挿入するごとに INDEX を付けるように指定
　x　インデクスの更新を行わず、インデクス情報も出力しないように指定
ginkou：行を挿入する表名
C:¥hirdb¥load.txt：テストデータファイルのある場所を示す定義ファイルがある場所（パス）を指定

4.4 SQL 文を作成するために使用したテストデータの概要

実験で用いる大量の SQL 文を作るためには、その元となるテストデータが必要である。以下にそのテストデータの例を示す。

テストデータの例

000001, 先木, 謙遼, 441740, 1898
000002, 木木, 祐虎, 237845, 2100
000003, 尼十, 穂慶, 910000, 1438
000004, 嘉斎, 秀史, 535180, 7018
000005, 石亜, 正巧, 492843, 4623
000006, 水仁, 玄一, 646658, 6795
・
・
・

上のテストデータは、１行ごとに１つの SQL 文を作るパラメータとなる。例の１行目では、「000001」が口座 ID、「先木」が苗字、「 謙遼」が名前、「 441740」が預金残高「1898」が暗証番号のパラメータとなる。

それぞれのパラメータの設定の仕方は、口座 ID は１行目の「000001」から、1 行進むごとに ID が１ずつ増えるように付けている。苗字と名前はそれぞれ苗字や名前に使われそうな漢字を 100 個集めたテンプレートファイルを２つ用意し、その 100 個の漢字の中から２つランダムに漢字を選択して漢字 2 文字で苗字と名前を作成している。預金残高は乱数を発生させ 6 桁の数字を暗証番号は 4 桁の数字をランダムに作成している。

なおこのテストデータは、pdload コマンドで使用できるように csv 形式※1 で作成されている。

(※1)csv 形式：一つ一つのパラメータをコンマで区切ったファイル形式

4.5 実験で使用した SQL 文の概要

今回の実験で使用する SQL 文は SELECT、INSERT、UPDATE、DELETE の４つの操作系 SQL 文で、それぞれの SQL 文の様式は以下のようになっている。

SELECT 文：SELECT ＊ FROM 表名 WHERE 口座 ID＝ ・・・

※主キー（口座 ID）は、ランダムに決定する。

UPDATE 文：UPDATE 表名 SET 預金残高 ＝ ・・・ WHERE 口座 ID＝ ・・・

※新しく更新する預金残高は乱数を使用しランダムに 6 桁の整数を作成する・
主キー（口座 ID）は、ランダムに決定する。

INSERT 文：INSERT INTO 表名 VALUES（口座 ID,苗字,名前,預金残高,暗証番号）

※挿入するパラメータはテストデータファイルから取得する。

DELETE 文：DELETE FROM 表名 WHERE 口座 ID＝ ・・・

※主キー（口座 ID）は、ランダムに決定する。

・作成した SQL 文の一部

```
UPDATE GINKOU SET 預金残高 = 482654 WHERE 口座 ID = '004130'
INSERT INTO GINKOU VALUES('027772','海本','貴修',889571,5315)
INSERT INTO GINKOU VALUES('017313','口岩','祐是',629669,2588)
SELECT * FROM GINKOU WHERE 口座 ID = '007877'
DELETE FROM GINKOU WHERE 口座 ID = '007151'
DELETE FROM GINKOU WHERE 口座 ID = '009040'
SELECT * FROM GINKOU WHERE 口座 ID = '006317'
UPDATE GINKOU SET 預金残高 = 731658 WHERE 口座 ID = '008735'
INSERT INTO GINKOU VALUES('015495','原渡','郷照',960885,4658)
SELECT * FROM GINKOU WHERE 口座 ID = '005405'
INSERT INTO GINKOU VALUES('010182','島久','克健',168071,6784)
                                        ・
                                        ・
                                        ・
```

4.6 Java Synchronized を用いた実験プログラム

4.6.1 実験プログラムの概要

① スレッド数と SQL 文の記述されたファイルを入力
② 指定された数のスレッドを作成
③ 各スレッドから JDBC を使って DBMS へアクセスし、SQL 文を実行する。
（この際、Synchronized を使用し、スレッド間で同期を取らせる。）

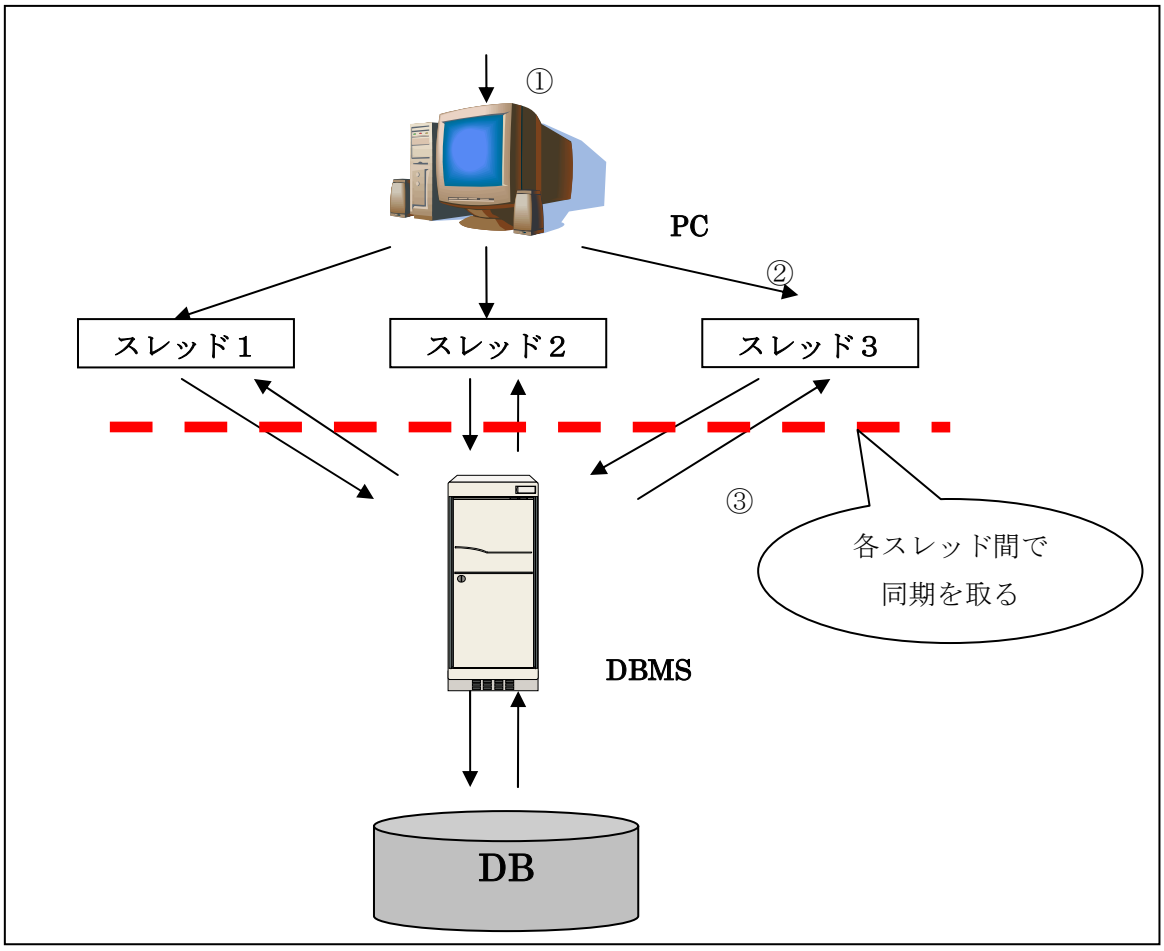
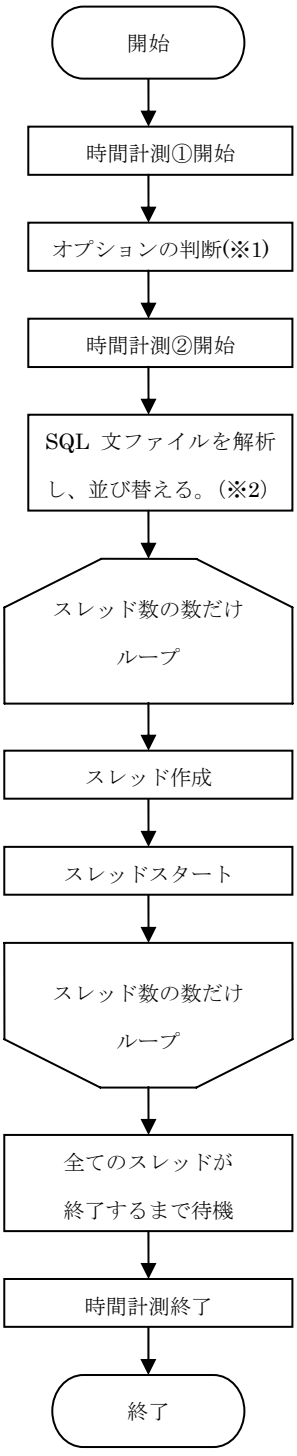


図 4.6.1 Synchronized 実験用プログラムの概要

4.6.2 実験プログラムのフローチャート



※1 入力された、スレッド数と、SQL 文のファイルが正しいかどうか調べ、それぞれ変数に格納する

※2 入力された SQL 文ファイルから SQL 文を 1 行ずつ取り出し、その SQL 文が SELECT 文なら Query オブジェクト※3 を、SELECT 文でなければ Update オブジェクト※4 を作成する。

作成された SQL 文オブジェクトは、4.2 章で説明したように、可変長のオブジェクト配列（ArrayList）に格納される。
(図：4.6.2-2 参照)

※3 指定された SELECT 文を実行するオブジェクト

※4 指定された INSERT 文を実行するオブジェクト

図 4.6.2-1：Main プログラムのフローチャート

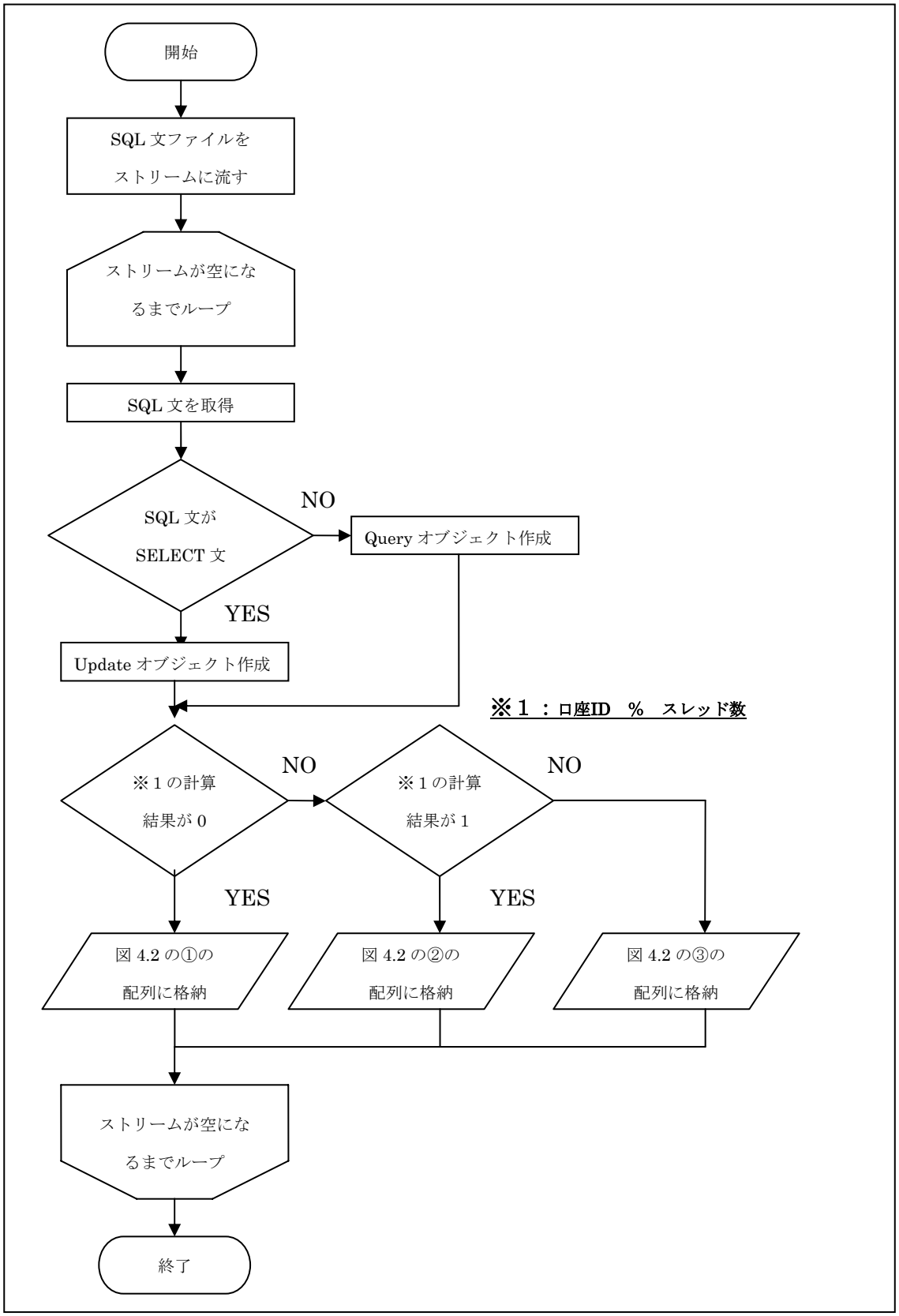


図 4.6.2-2：ＳＱＬ文を解析し、並び替える部分のフローチャート

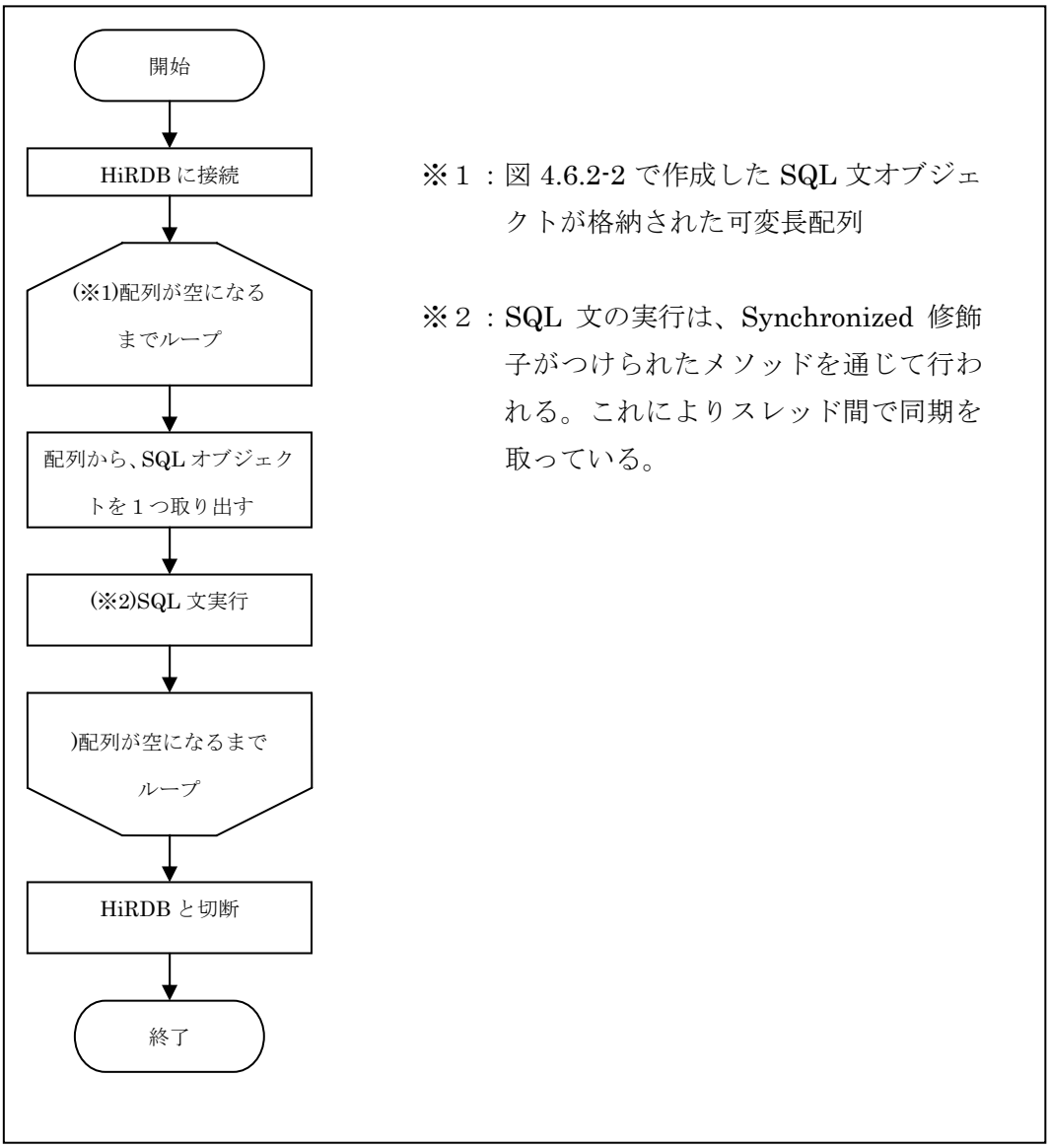


図 4.6.2-3：スレッドの動きのフローチャート

4.7 Java.util.concurrent.atomic パッケージを用いた実験プログラム

4.7.1 実験プログラムの概要

① スレッド数、SQL 文の記述されたファイルを入力する。
② 指定された数のスレッドを作成する。
③ Java.util.concurrent.atomic パッケージを使用して作成した Wait-Free なキューを作成する。
④ 各スレッドは、ロックをかけることなくキューに、SQL オブジェクトを入れていく。
⑤ キューに SQL オブジェクトを入れていくスレッドとは別に、SQL 文を実行するスレッドを作成し、スレッドはキューから SQL オブジェクトを順次取り出し実行していく。

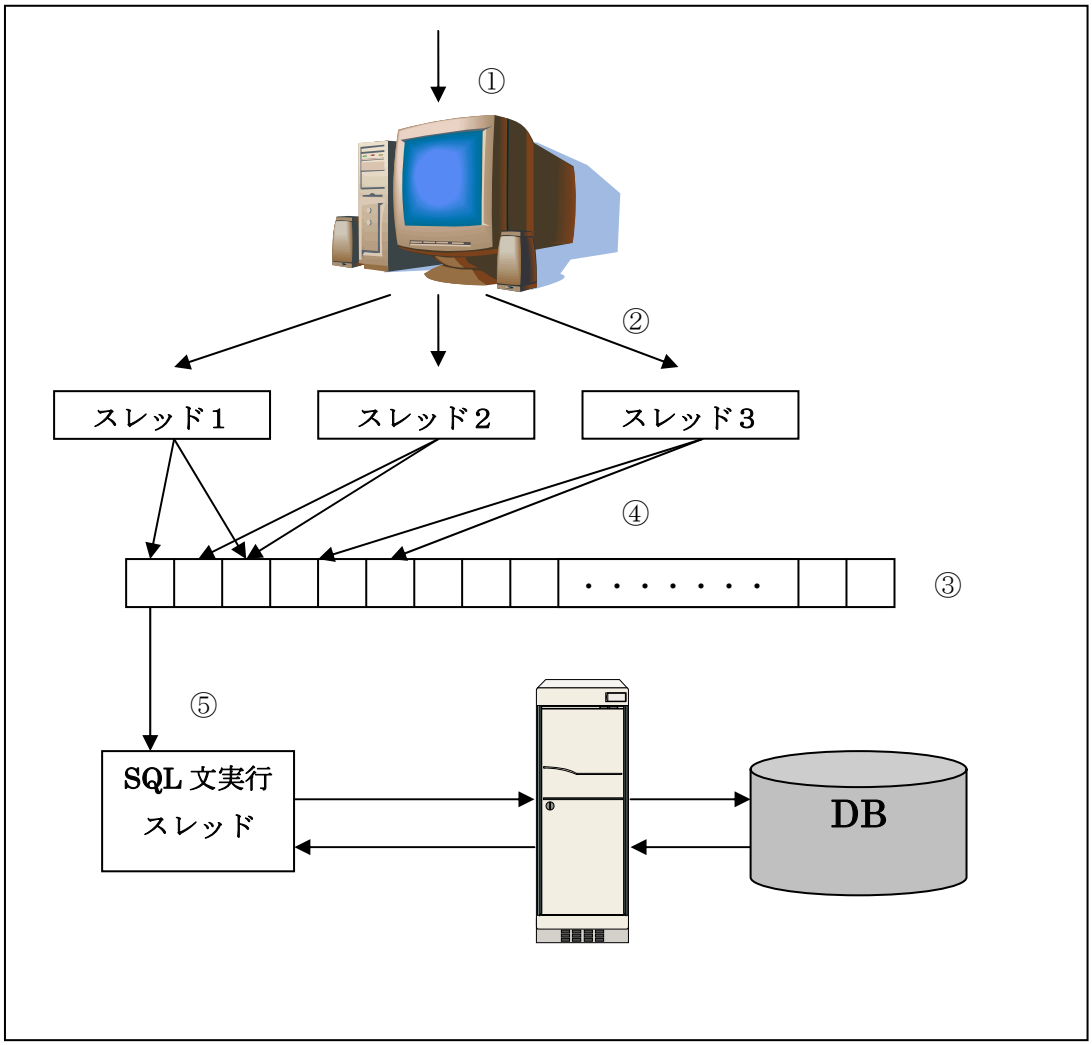


図 4.7.1 実験プログラム②の概要

4.7.2 実験プログラムのフローチャート

Main プログラムは、4.6.2 のフローチャートと同じ構造となる。

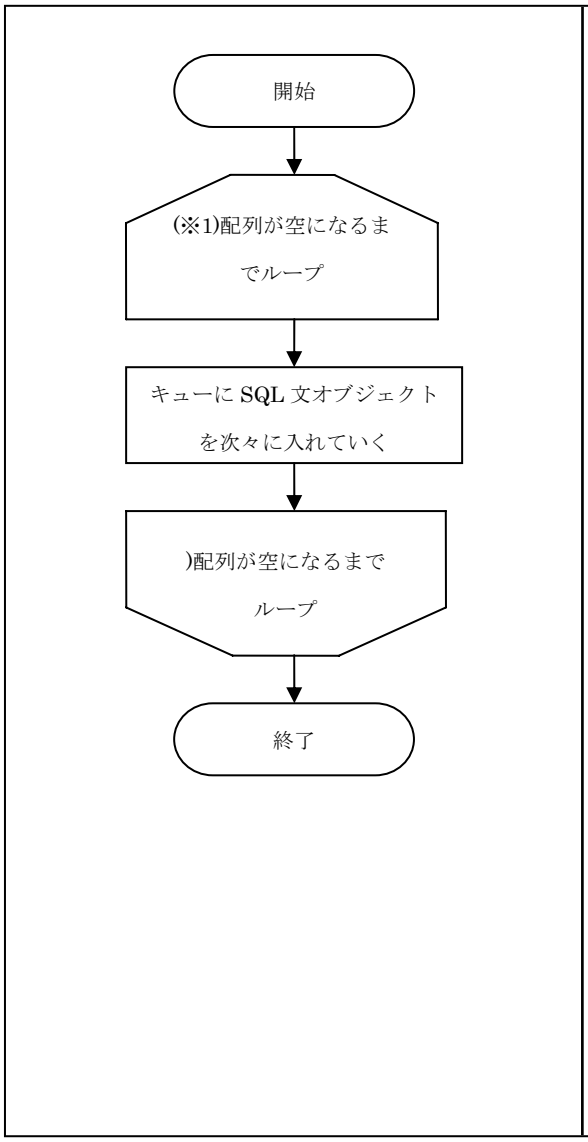


図 4.7.2-1 スレッドのフローチャート（キュー挿入）

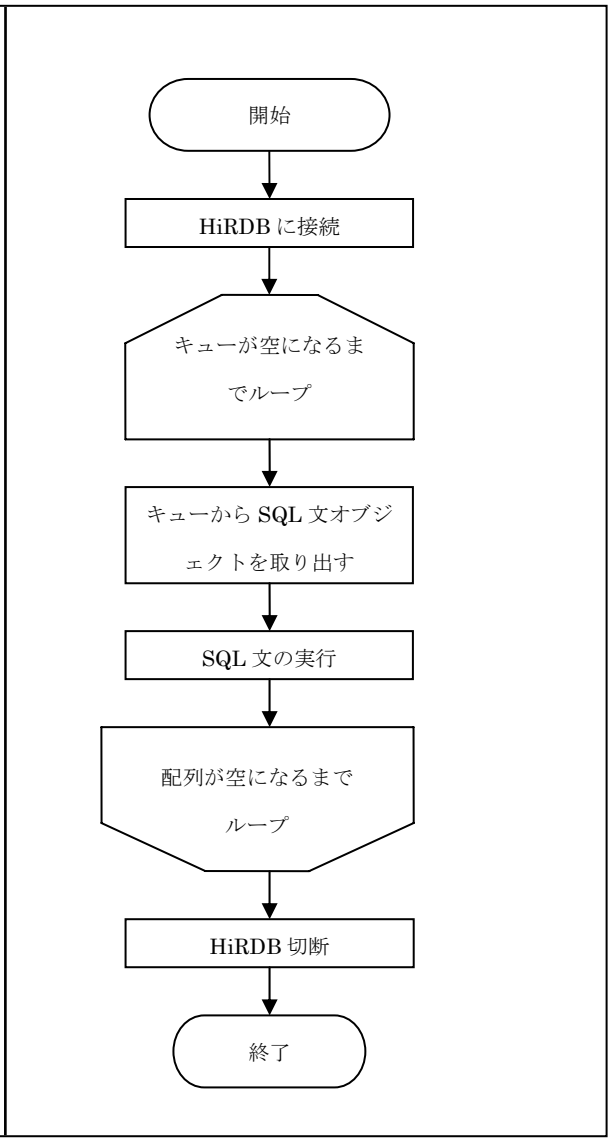


図 4.7.2-2 スレッドのフローチャート（SQL 文実行）

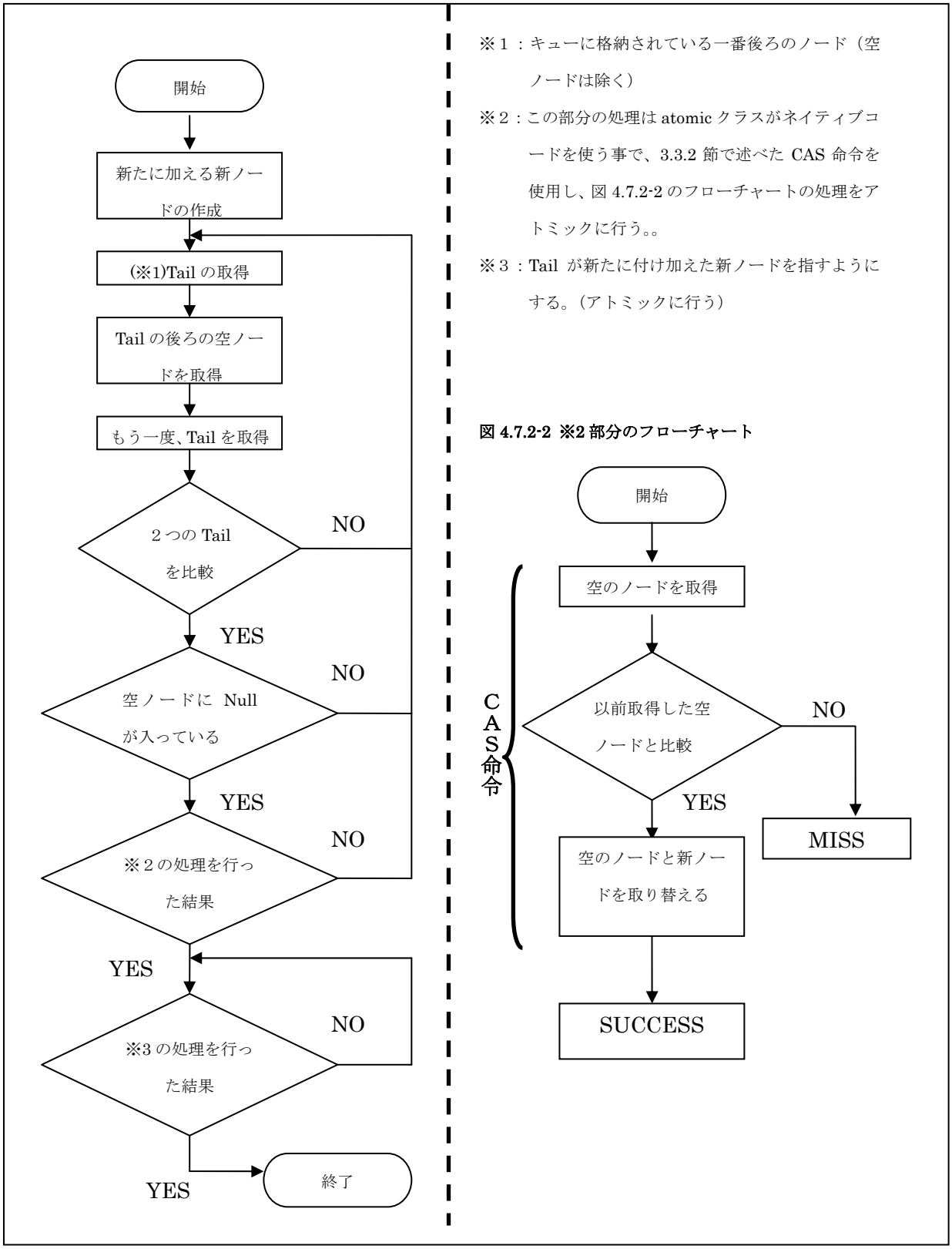


図 4.7.2-1：Wait-Free キューのフローチャート

4.8 DBMS の LOCK 機能を用いた実験プログラム

4.8.1 実験用プログラムの概要

① スレッド数、SQL 文の記述されたファイルを入力。
② 指定された数のスレッドを作成。
③ 各スレッドから DBMS へ非同期にアクセスする。
④ DBMS から DB へのアクセスを排他的に行う。

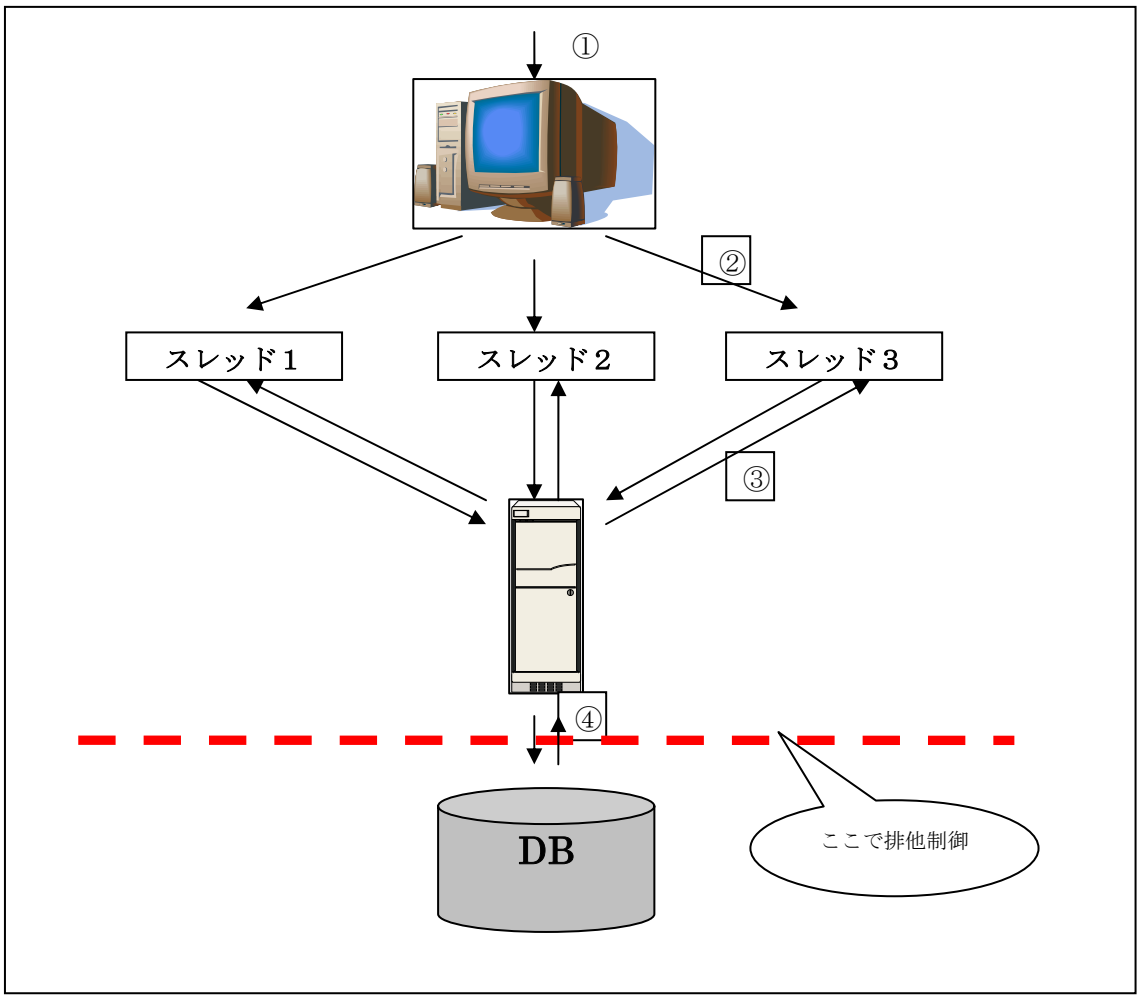


図 4.8.1　実験プログラム③の概要

実験プログラム自体は、Synchronized で各スレッドから DBMS へのアクセスを排他的に行う部分以外はすべて 4.6.2 節のフローチャートと同じとなる。

4.8.2 DBMS の設定[3][4]

この節では、DBMS が DB へのアクセスを排他的に行うようにするための、DBMS の設定に関して説明を行う。今回の実験では、GINKOU という表に対して、行レベルでの排他制御を行っている。また排他制御モードは、1 トランザクションだけが排他資源を占有し，ほかのトランザクションには参照，追加，更新，及び削除を許さない、排他モードとして実験を行った。

・テーブルの行単位で排他制御を行うようにするための設定

定義系 SQL の「Create Table」で表を作成する際に「Lock Row」を指定する。(以下に例を示す)

```
Crete Table GINKOU(口座 ID char(6),苗字 char(4),名前 char(4),
預金残高 int, 暗証番号 int)Lock Row
```

・排他制御モードの設定

制御系 SQL の Lock 文を使用して表の排他制御モードを設定する。(以下に例を示す)

```
Lock table GINKOU IN exclusive mode
```

この２つの設定を行う事により、1 行の SQL 文を一つのトランザクションとした処理においては、直列可能性を実現できる。1 行以上の SQL 文を一つのトランザクションとする場合には、ストアドプロシージャを使用するが、その詳しい設定は後の 5.4 節で述べる。

第 5 章　直列可能性の確認実験

5.1 実験の概要

前章の実験ではトランザクションを一つの SQL 文としていたが、本実験では、

①指定された口座から預金残高を参照する。

②預金残高に振り込み金額を足しこむ。

③新しい預金残高情報を DB に更新する。

の３つの処理を一つのトランザクションとして、実行するトランザクションの数を 10000、対象とする口座 ID を「000001」に限定した上で、前章で挙げた３つの直列可能性の実現方法による実行結果と、一つのスレッドでトランザクションの逐次処理を行った実行結果を比較して、全て同じテーブルが作られているか調査を行った。

5.2 Java Synchronized を用いた実験の概要

実験プログラムは、4 章で説明した内容と同じだが、実行するトランザクションが 5.1 節で述べたような内容となっている。なおこの実験では、処理を行うテーブルの行の口座 ID を’000001’に、振込む金額を 1000 円と限定し、実験を行っている。

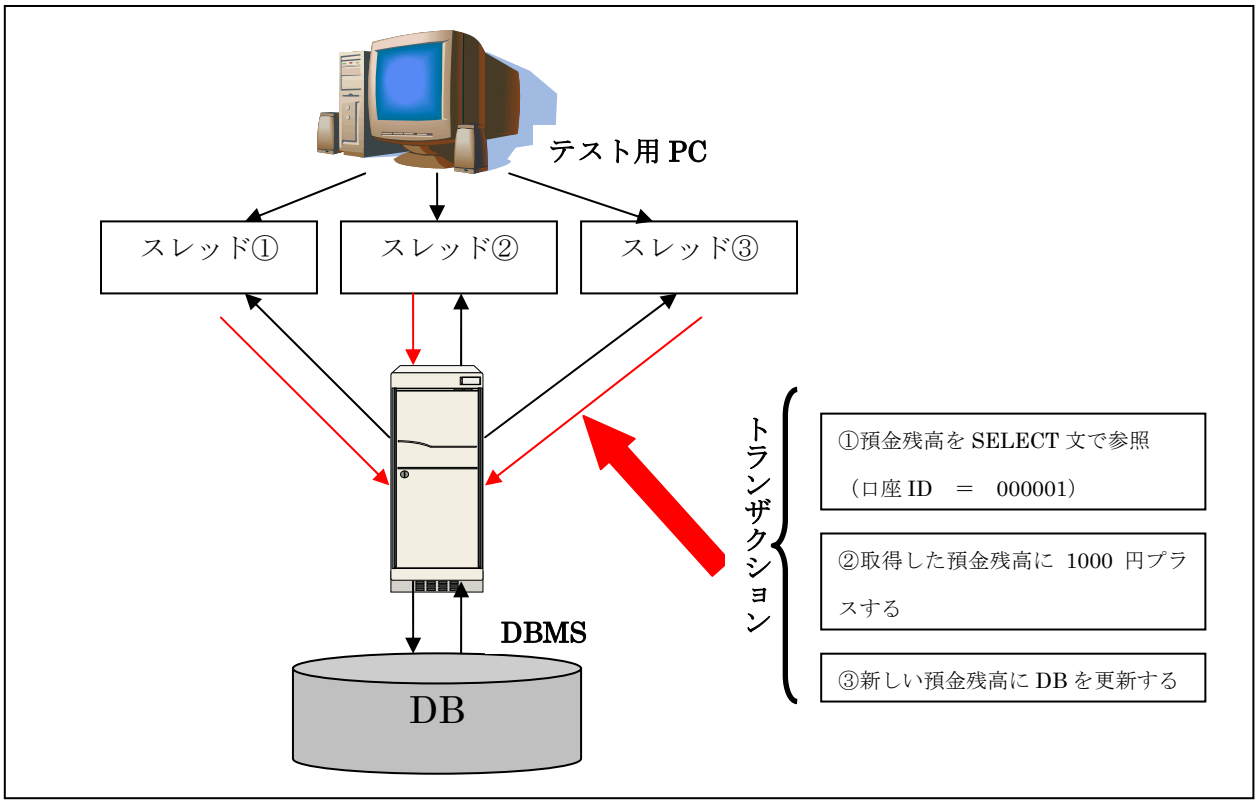


図 5.2 Synchronized を使用した方法の直列可能性の実験図

5.3 Java.util.concurrent.atomic パッケージを用いた実験の概要

実験プログラム自体は 4 章のものとまったく同じとし、実行するトランザクションを図 5.3 のようにして、実験を行った。

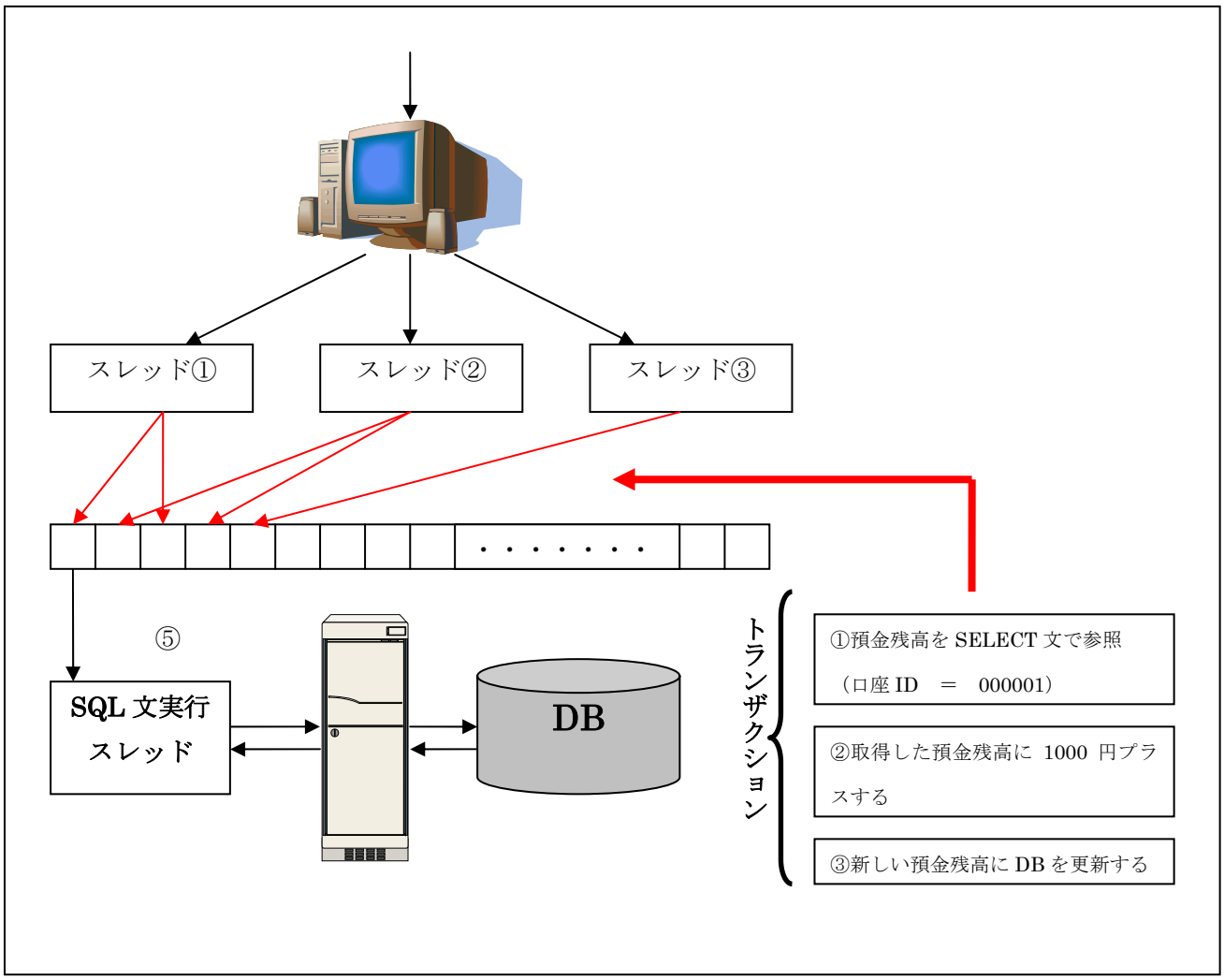


図 5.3 Java.util.concurrent.atomic を使用した方法の直列可能性の実験図

5.4 DBMS の Lock 機能＋ストアドプロシージャを用いた実験の概要[3][4]

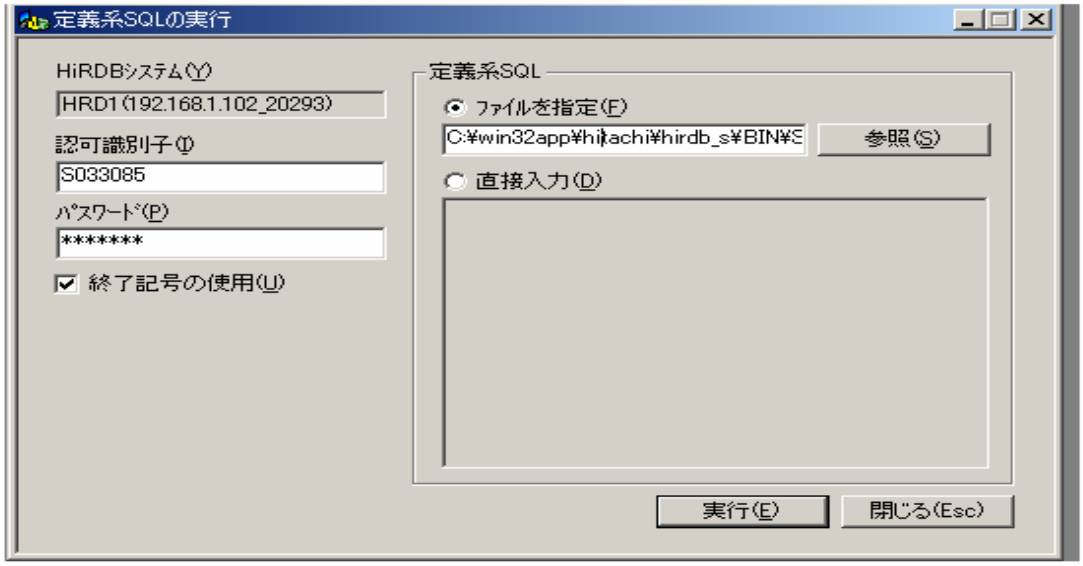
第 4 章の実験では、一つの SQL 文をトランザクションとして実行させていたが、今回の実験では複数の処理をトランザクションとして実行させるため、複数の処理を DBMS 内に蓄積させ、それを一つのトランザクションとして実行するストアドプロシージャを使い実験を行った。

・ストアドプロシージャの定義

ストアドプロシージャを定義するには、定義系 SQL である「Create　Procedure」を使う。今回作成したストアドプロシージャは、5.1 節で述べたトランザクションを実装している。

・DBMS にストアドプロシージャを登録する

① HiRDB コントロールマネージャを開く。
② マップ→表→定義 SQL の実行を選択。
③ ID とパスワードを入力する。
④ 制御ﾌｧｲﾙを指定するか、直接 SQL 文を入力して、実行する。(制御ファイルとは、ストアドプロシージャの記述されたファイルのパスが記述されたファイルの事である)



DBMS に登録したストアドプロシージャを、CALL 文を使い格スレッドから呼び出す事で３つの処理を一つのトランザクションとして扱うことができるようになり、DBMS でテーブルへのアクセスを行レベルで排他制御するようにすれば、ストアドプロシージャの処理に対するテーブルの行レベルの排他制御を行う事ができるため、直列可能性を実現できる。

第６章　実験の結果と考察

6.1 実行時間比較実験の結果

第４章で述べた３つの実験用を以下の条件において実行させ比較を行った。

①スレッド数を２、４、８、１６として実験を行った。

②実行する SQL 文の数は、1000 行, 2000 行, ･･･, 9000 行と 10000 行, 20000 行, ･･･90000 行として実験を行った。

③スレッド数と、実行したＳＱＬ文数の全ての組み合わせに対し、5 回テストを行い、得られた結果の平均を取り、実験の結果とした。

以下に実験の結果を示す。(詳細な実験結果は付録を参照)

※各グラフは、EXCEL の機能を使い近似曲線を引いている。Synchronized、atomic の結果は線形近似曲線、DBMS の Lock の結果は多項式近似曲線を使い引いていて、どちらとも最小二乗法による近似が行われている。[10]

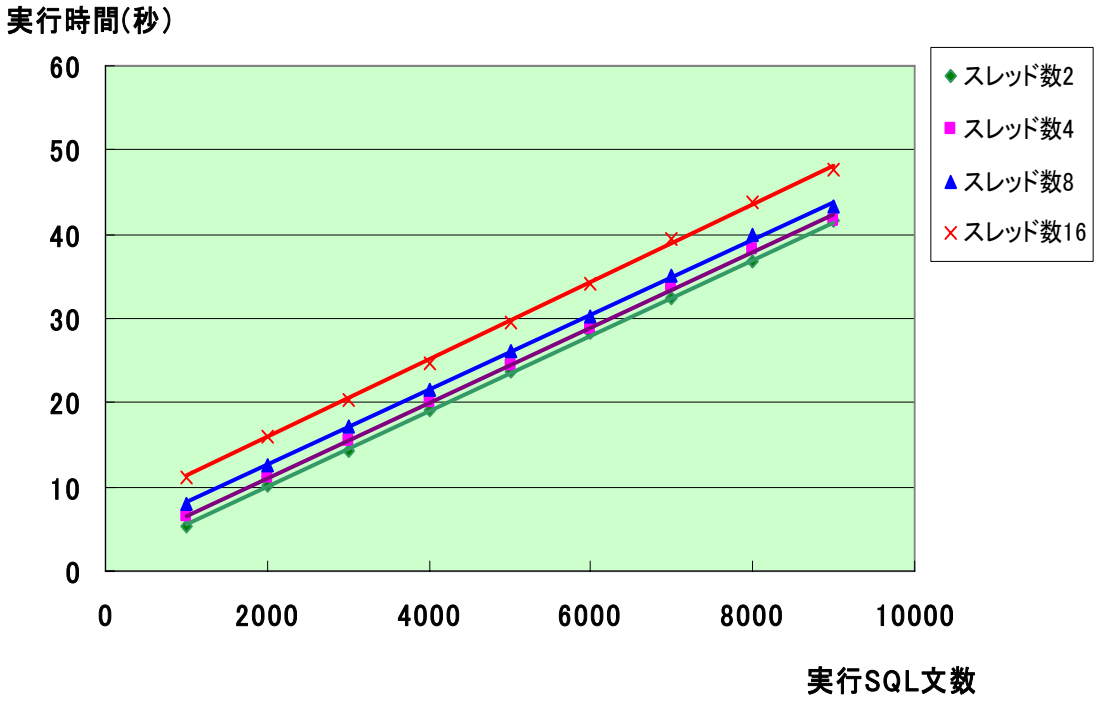


図 6.1.1 Synchronized の結果 (SQL 文 1000 行～9000 行)

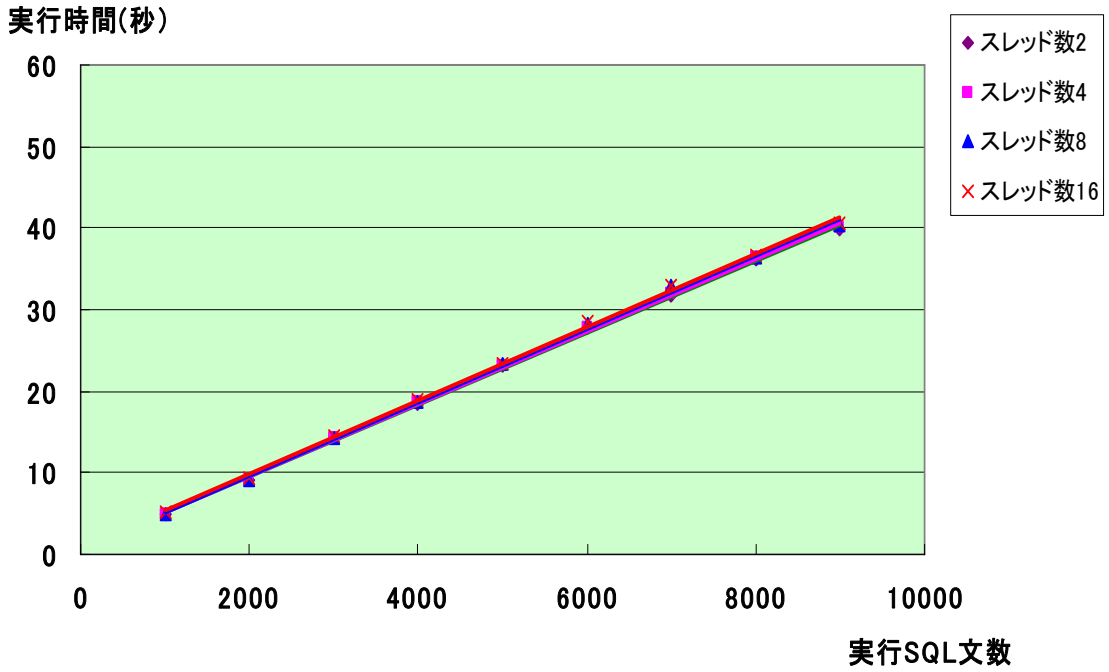


図 6.1.2 Java.util.concurrent.atomic を用いた場合の結果 (SQL 文 1000 行～9000 行)

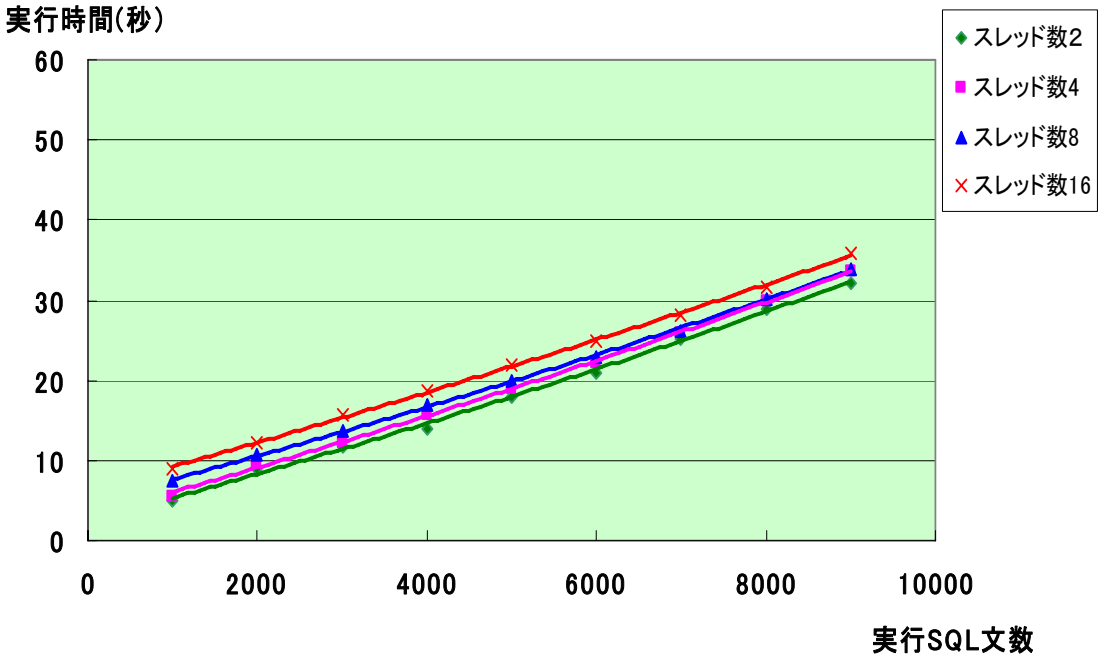


図 6.1.3 DBMS の Lock 機能を用いた場合の結果（SQL 文 1000 行～9000 行）

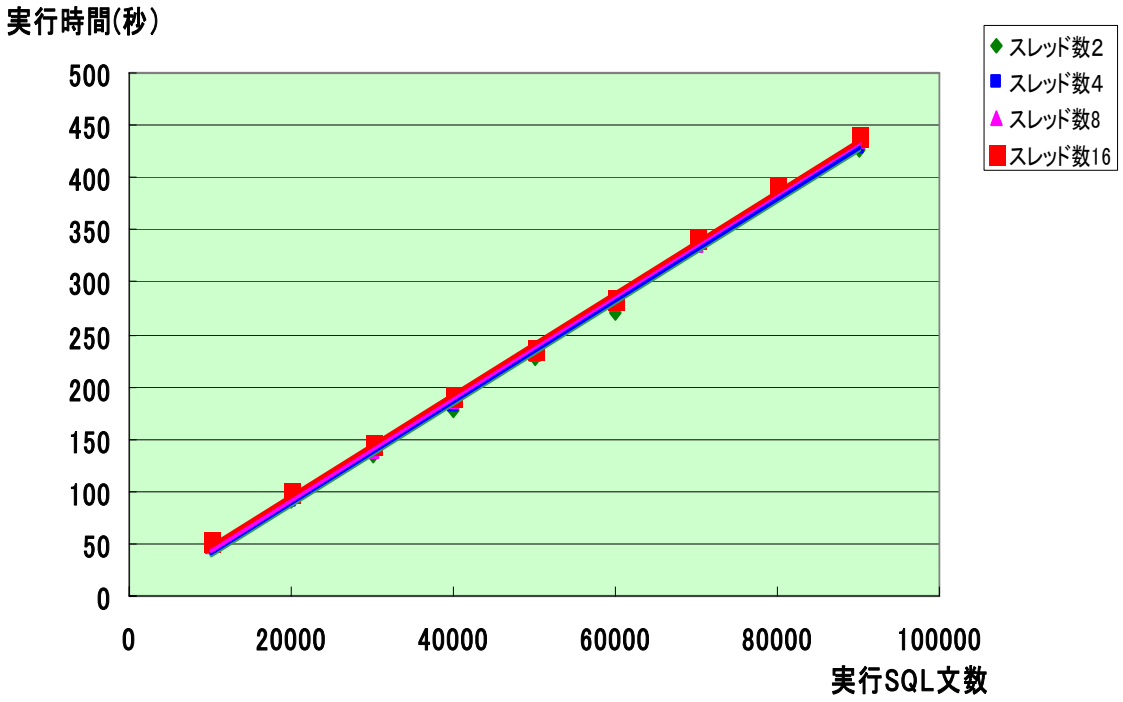


図 6.1.4 Synchronized の結果（SQL 文 1000 行～9000 行）

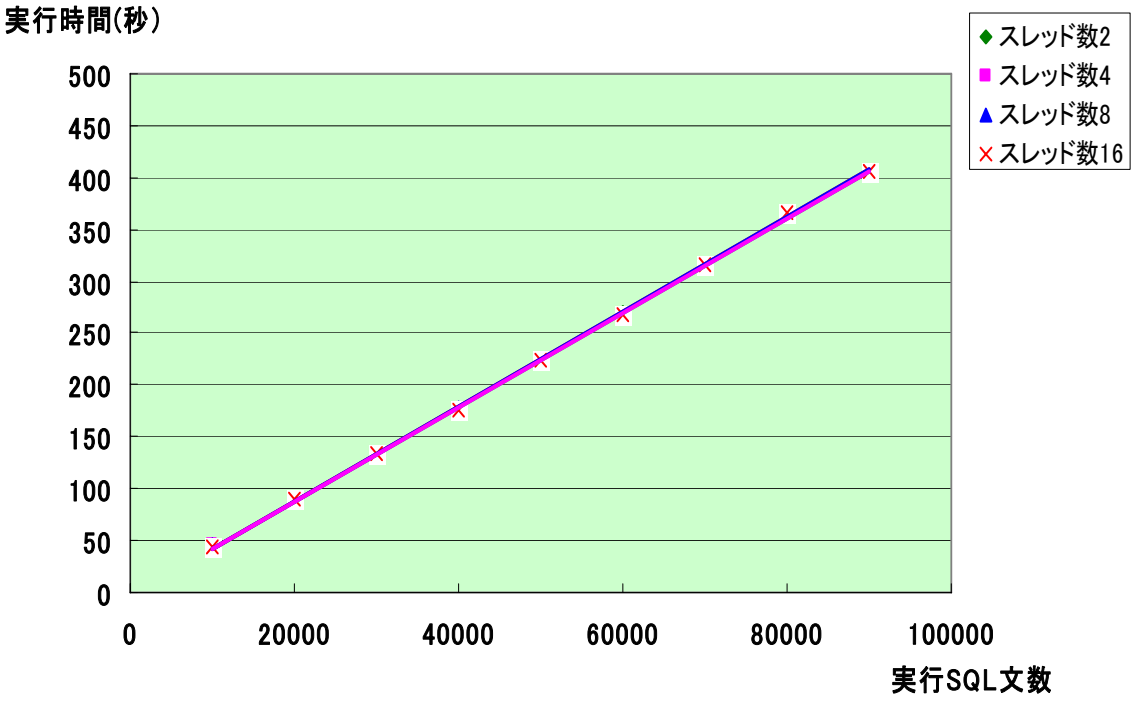


図 6.1.5 Java.util.concurrent.atomic を用いた場合の結果(SQL 文 10000 行～90000 行)

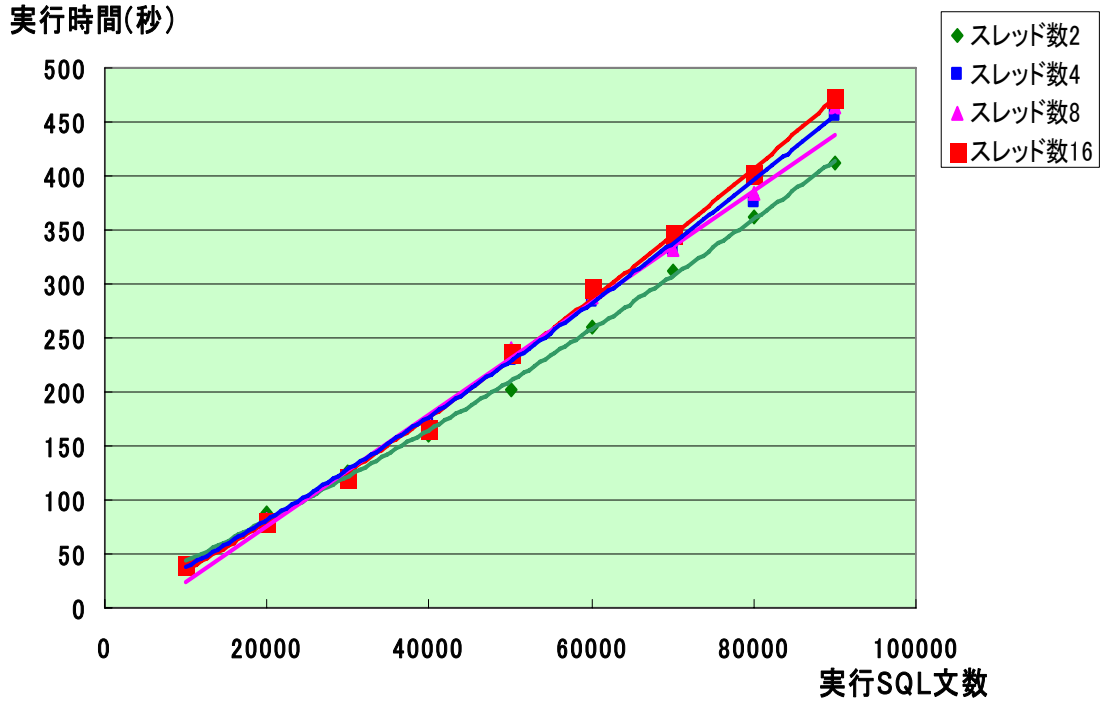


図 6.1.6 DBMS の Lock 機能を用いた場合の結果 (SQL 文 10000 行～90000 行)

6.2 実行時間比較実験の考察

実行したＳＱＬ文数が 1000 行～9000 行の場合では、Synchronized を用いた方法(以下実験①),Java.util.concurrent.atomic を用いた方法(以下実験②)の二つの方法の結果と、DBMS の Lock 機能を用いた方法（以下実験③）の結果では、実行したＳＱＬ文数が増えるのに比例して、実行時間の差が大きくなっている事がわかる(付録の図 8.1～図 8.5 参照)。10000 行程度の行数ならば実験③の方法が一番優れている事が実験結果より見て取れる。これは他の２つの実験と異なり、実験③では各スレッドから DBMS へのアクセスを非同期に行っている点が、この結果に反映されていると考えられる。また、実験②では、他の２つの実験と比べ、スレッド数の増加に対する実行時間の増加があまり見られない。実験②では、各スレッドからキューの中に SQL 文オブジェクトを非同期に入れているが、このキューにオブジェクトを格納する時間がほとんど掛からないため、キューの中にオブジェクトを格納するスレッドの数を増やしても実行時間はあまり変化しなかったと考えられる。

実行した SQL 文数が 10000 行～90000 行の場合では、SQL 文数が 30000 行の時の実験③の結果(付録の図 8.7)が他の 2 つの実験の結果に比べ、実行時間の差が一番大きくなっている事がわかる。しかし SQL 文数が 50000 行の時の、実験③の結果(付録の図 8.8)を見てみるとスレッド数が大きくなる毎に大きく実行時間が増大している事がわかる。これは、多くのスレッドから非同期に DBMS に対して、大量の SQL 文の実行要求を出したため DB サーバの

負荷が大きくなり、その不可が SQL 文の実行に影響を与え、実行時間の増加を導いたと考えられる。SQL 文数が 70000 行(付録の図 8.9)、90000 行(付録の図 8.10)と増えていくごとに、実験③の実行結果は。他の二つの実験と比べ、実行速度の遅い結果となっている。また実験②の結果が、SQL 文数が 70000 行を越えた辺りから一番良い結果を出している。この結果により、実験②の方法が実行する SQL 文数が大量に増える事で発生する負荷に対して一番強いと考えられる。

6.3 直列可能性確認実験の結果

第５章で述べた、Synchronized、atomic、DBMS の Lock 機能＋ストアドプロシージャの３つの実験の結果、作成されたテーブルと一つのスレッドにおいて逐次的にトランザクションを実行させた結果を以下のフローチャートのプログラムによって確認した所、全て正しく直列可能性が実現されている事が確認できた。

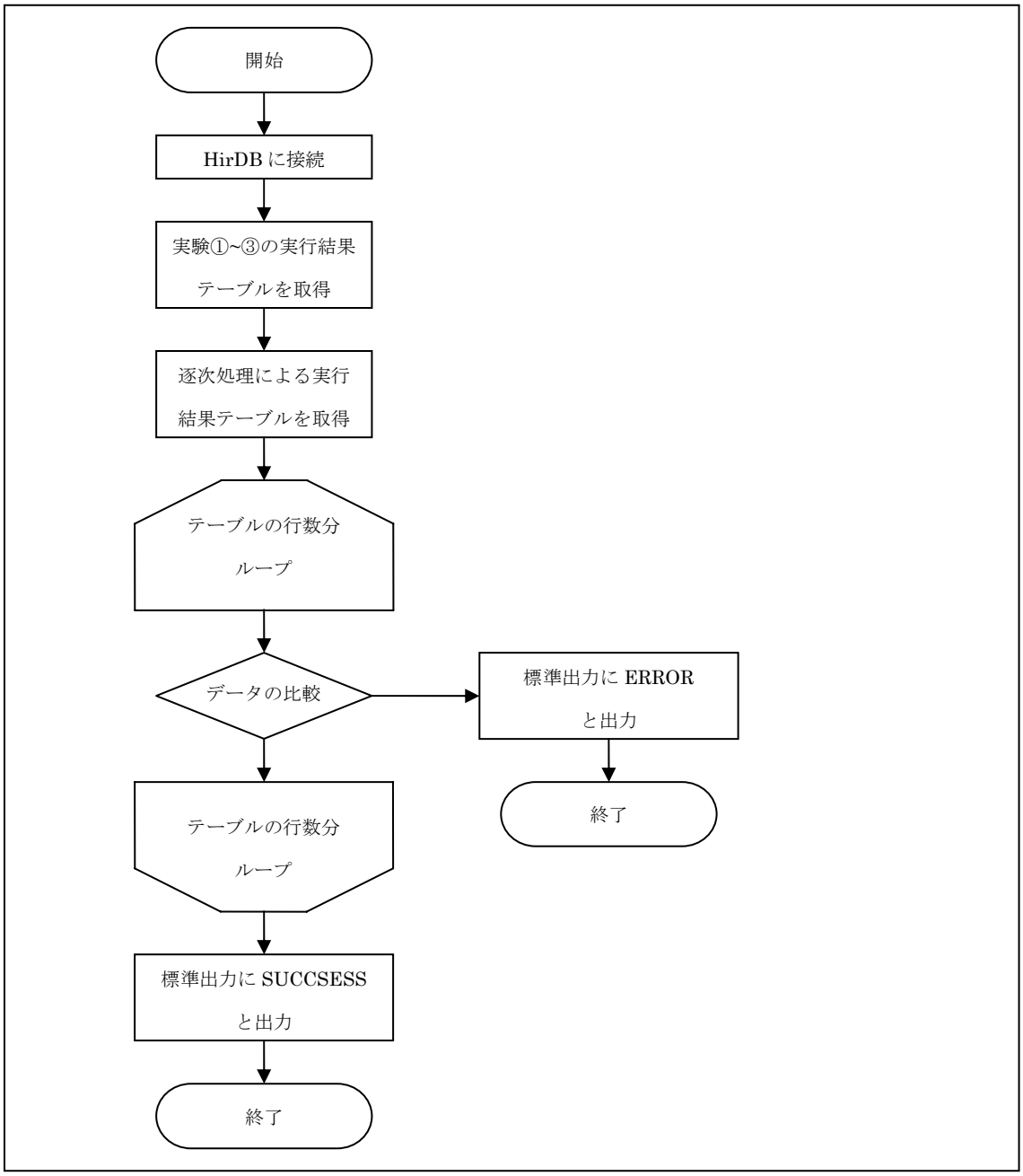


図 6.3 直列可能性確認プログラムのフローチャート

第 7 章　終論

これまで、直列可能性を実現するための方法として①Java Synchronized②Java.util.concurrent.atomic、③DBMS の Lock 機能の 3 つの方法の性能について検証を行ってきた。3 つの方法の違いによって、当初予測していたほどの大きな差は見られなかったが、実験で扱うデータを比較的容量の軽いテキストデータではなく、空間データのよう

な容量の重たいデータで実験を行えば、結果の差は、得に実験②と他の２つの実験との間で大きく見られる可能性がある。

・謝辞

本研究において最後まで熱心な御指導をしていただきました田中章司郎教授には、心より御礼申し上げます。同研究室の学部生の 4 名にはさまざまな場面での助言をしていただきました。深く御礼申し上げます。なお、本論文、本研究で作成したプログラム及びデータ、ならびに関連する発表資料等の知的財産権を、本研究の指導教官である田中章司郎教授に譲渡致します。

・参考引用文献

[1] 林晴　比古
　改定　新 Java 言語入門　シニア編　SOFTBANK

[2] サイト名　Java　2　Platform　SE5.0　Java.util.concurrent
　http://java.sun.com/j2se/1.5.0/ja/docs/ja/api/java/util/concurrent/package-summary.html

[3] HITACHI
　マニュアル 『HiRDB システム導入・設計ガイド』

[4] HITACHI
　マニュアル 『HiRDB UAP 開発ガイド』

[5] サイト名　Java　2　Platform　SE5.0　Java.util.concurrent.atomic
　http://java.sun.com/j2se/1.5.0/ja/docs/ja/api/java/util/concurrent/atomic/package-summary.html

[6] 北川　博之　著
　データベースシステム　昭晃堂

[7]サイト名　Wikipedia 「Lock-Free と Wait-Free アルゴリズム」
　http://ja.wikipedia.org/wiki/Lock-free

[8]サイト名　IBM　Java の理論と実跡：ノンブロッキング・アルゴリズムの紹介
　http://www-06.ibm.com/jp/developerworks/java/060519/j_j-jtp04186.shtml

[9]サイト名　IBM　Tiger を使いこなす：並列コレクション
http://www-06.ibm.com/jp/developerworks/java/040625/j_j-tiger06164.html

[10]サイト名　エクセルを使った最小二乗法
　http://szksrv.isc.chubu.ac.jp/lms/lms7.html

[11]増永　良文　著

リレーショナルデータベース入門　サイエンス社

・付録

表 8.1 Synchronized の結果 (SQL 文 1000 行～9000 行)

	スレッド数 2	スレッド数 4	スレッド数 8	スレッド数 16
SQL 文 1000 行	5.302	6.432	8.012	11.189
SQL 文 2000 行	10.281	11.032	12.609	16.062
SQL 文 3000 行	14.39	15.578	17.104	20.437

SQL 文 4000 行	19.016	20.078	21.609	24.703
SQL 文 5000 行	23.61	24.344	26.121	29.617
SQL 文 6000 行	28.203	28.703	30.36	34.078
SQL 文 7000 行	32.391	33.906	34.978	39.323
SQL 文 8000 行	36.719	38.344	39.813	43.796
SQL 文 9000 行	41.5	41.718	43.2	47.547

※単位は秒(sec)

表 8.2 Java.util.concurrent.atomic を用いた場合の結果(SQL 文 1000 行～9000 行)

	スレッド数 2	スレッド数 4	スレッド数 8	スレッド数 16
SQL 文 1000 行	4.987	4.902	5.001	5.19
SQL 文 2000 行	9.153	9.163	9.221	9.241
SQL 文 3000 行	14.234	14.243	14.311	14.414
SQL 文 4000 行	18.562	18.642	18.678	18.891
SQL 文 5000 行	23.001	23.112	23.241	23.312
SQL 文 6000 行	27.678	27.821	28.193	28.54
SQL 文 7000 行	31.678	31.871	32.901	32.985
SQL 文 8000 行	36.171	36.321	36.441	36.552
SQL 文 9000 行	39.849	40.111	40.251	40.52

※　単位は秒(sec)

表 8.3 DBMS の Lock 機能を用いた場合の結果 (SQL 文 1000 行～9000 行)

	スレッド数 2	スレッド数 4	スレッド数 8	スレッド数 16
SQL 文 1000 行	4.871	5.578	7.457	9.001
SQL 文 2000 行	9.018	9.781	10.667	12.271

SQL 文 3000 行	11.791	12.281	13.611	15.591
SQL 文 4000 行	14.001	15.656	16.991	18.629
SQL 文 5000 行	18.011	18.875	19.932	21.953
SQL 文 6000 行	21.031	22.14	22.91	24.978
SQL 文 7000 行	25.09	26	26.224	28.011
SQL 文 8000 行	28.991	29.843	30.213	31.69
SQL 文 9000 行	32.12	33.609	33.86	35.906

※　単位は秒(sec)

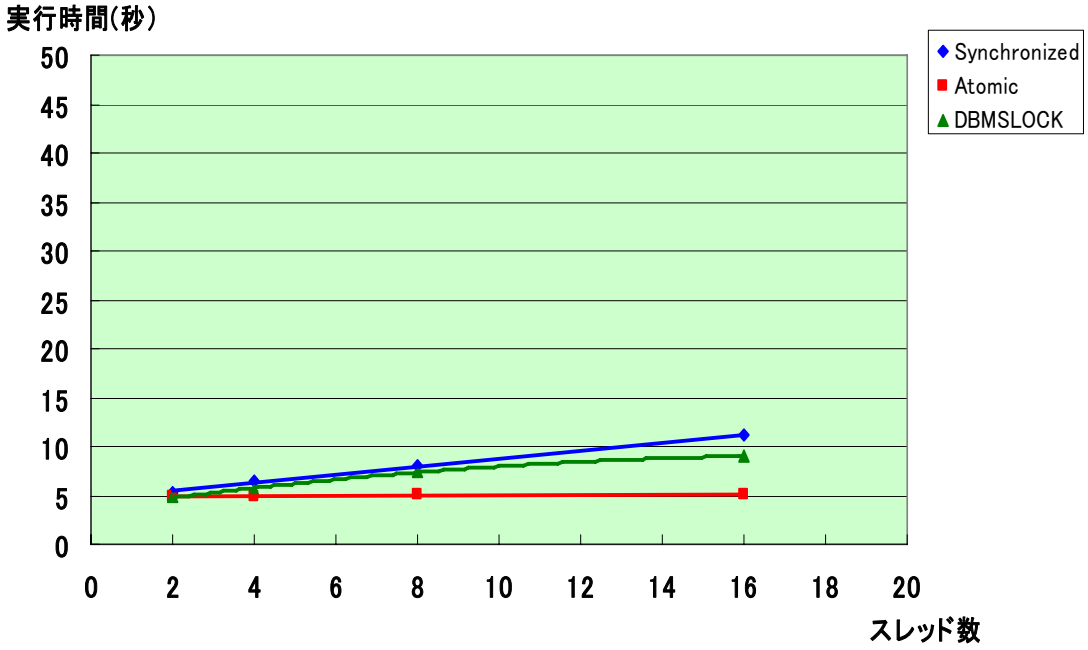


図 8.1 SQL 文が 1000 行の時の結果（縦軸：実行時間、横軸：スレッド数）

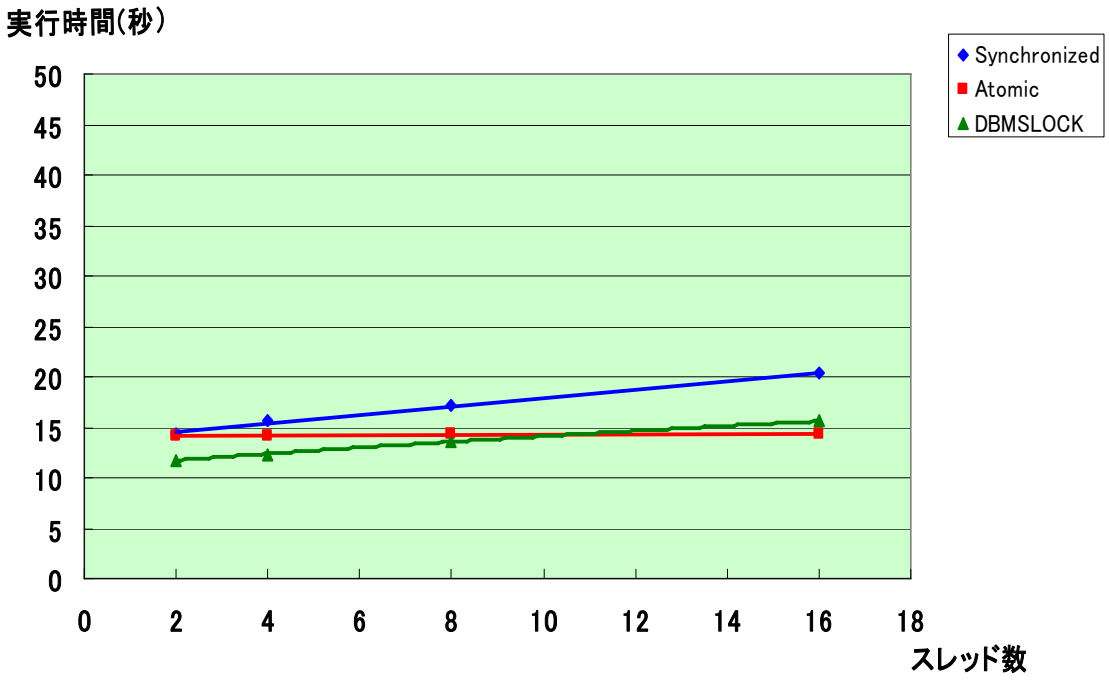


図 8.2 SQL 文が 3000 行の時の結果（縦軸：実行時間、横軸：スレッド数）

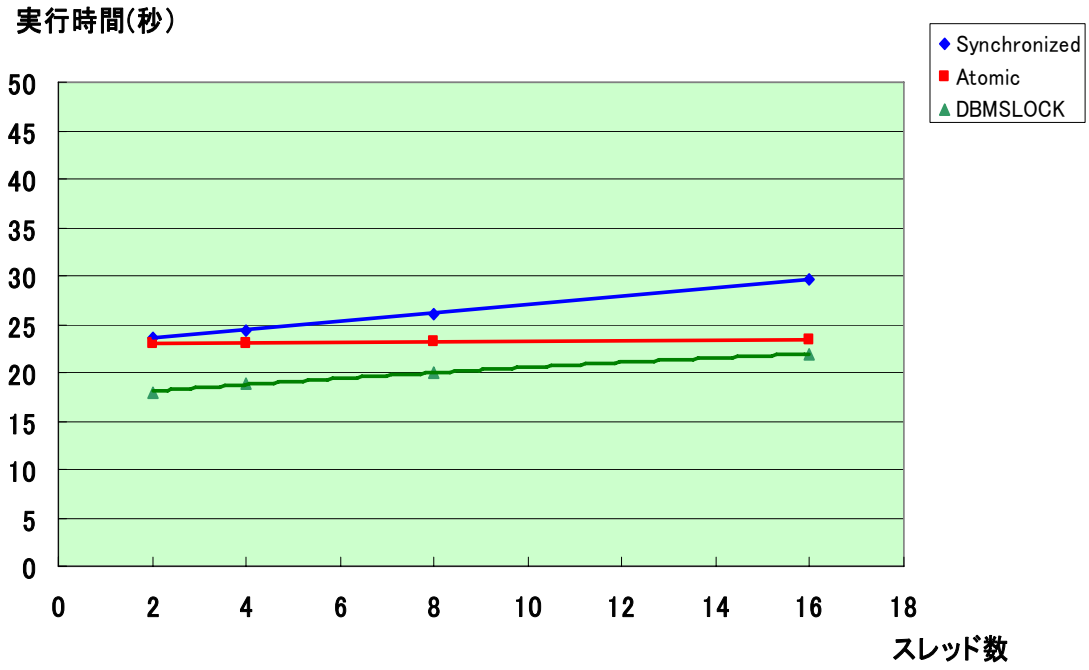


図 8.3 SQL 文が 5000 行の時の結果（縦軸：実行時間、横軸：スレッド数）

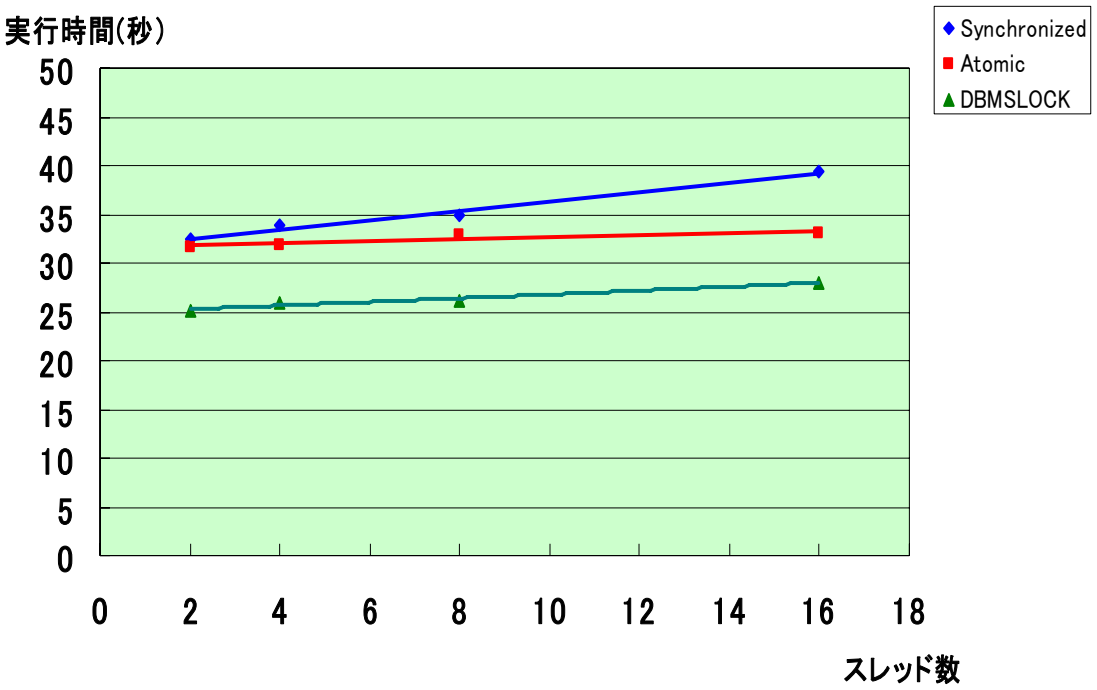


図 8.4 SQL 文が 7000 行の時の結果（縦軸：実行時間、横軸：スレッド数）

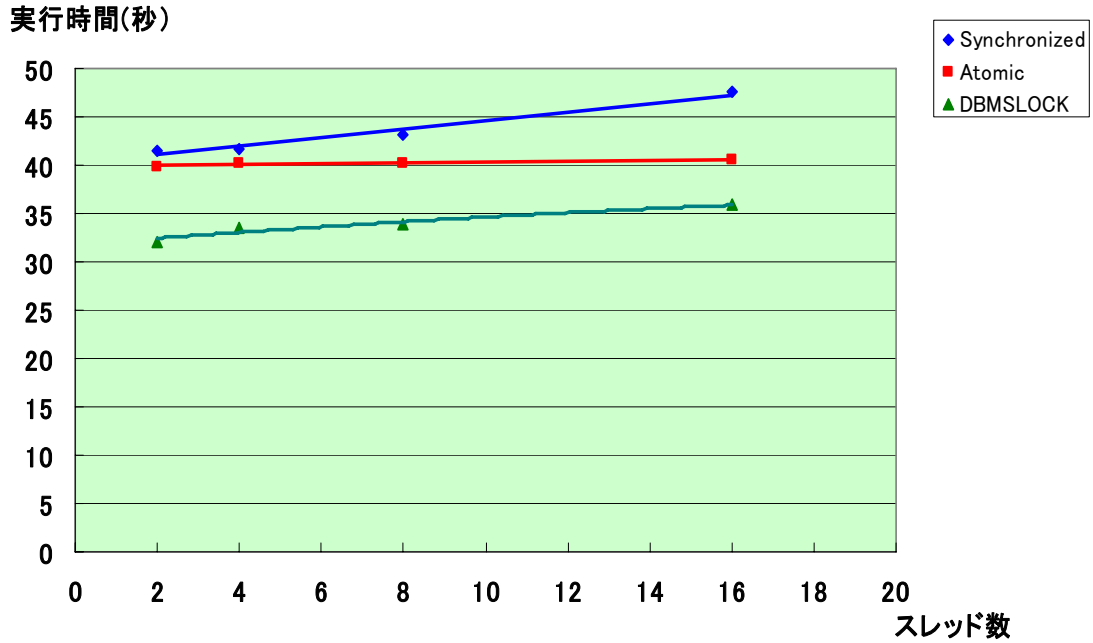


図 8.5 文が 9000 行の時の結果（縦軸：実行時間、横軸：スレッド数）

表 8.4 Synchronized の結果 (SQL 文 10000 行～90000 行)

	スレッド数2	スレッド数4	スレッド数 8	スレッド数 16
SQL 文 10000,行	45	46.432	48.563	52.109
SQL 文 20000 行	90.134	91.109	94.204	100.5
SQL 文 30000 行	134.235	136.469	138.421	146.016
SQL 文 40000 行	178.123	180.849	185.844	190.14
SQL 文 50000 行	227.154	229.578	231.283	235.578
SQL 文 60000 行	270.345	276.684	279.063	284.219
SQL 文 70000 行	336.189	338.616	336.125	341.512
SQL 文 80000 行	383.498	385.862	390.213	391.625
SQL 文 90000 行	425.61	429.17	434.167	438.42

※単位は秒(sec)

表 8.5 Java.util.concurrent.atomic を用いた場合の結果(SQL 文 1000 行～9000 行)

	スレッド数 2	スレッド数 4	スレッド数 8	スレッド数 16
SQL 文 10000 行	44.378	45.002	45.321	44.345
SQL 文 20000 行	89.312	89.875	88.012	89.216
SQL 文 30000 行	132.625	133.192	132.543	134.793
SQL 文 40000 行	175.128	174.892	177.008	176.231
SQL 文 50000 行	221.157	220.341	222.671	223.781
SQL 文 60000 行	268.891	266.912	268.333	267.971
SQL 文 70000 行	313.325	314.01	314.949	315.87
SQL 文 80000 行	365.134	366.102	366.961	367.012
SQL 文 90000 行	404.491	404.006	405.698	406.622

※単位は秒(sec)

表 8.6 DBMS の Lock 機能を用いた場合の結果（SQL 文 10000 行～90000 行）

	スレッド数 2	スレッド数 4	スレッド数 8	スレッド数 16
SQL 文 10000 行	39.112	36.544	39.007	39.529
SQL 文 20000 行	87.703	76.892	81.101	79.526
SQL 文 30000 行	126	120.578	124.531	119.873
SQL 文 40000 行	160.451	170.342	172.415	166.816
SQL 文 50000 行	202.75	229.917	239.081	236.55
SQL 文 60000 行	260.422	284.497	288.394	295.5
SQL 文 70000 行	311.4	332.001	332.9	345.701
SQL 文 80000 行	361.899	376.903	383.517	401.612
SQL 文 90000 行	412.002	455.5	464.922	471.698

※単位は秒(sec)

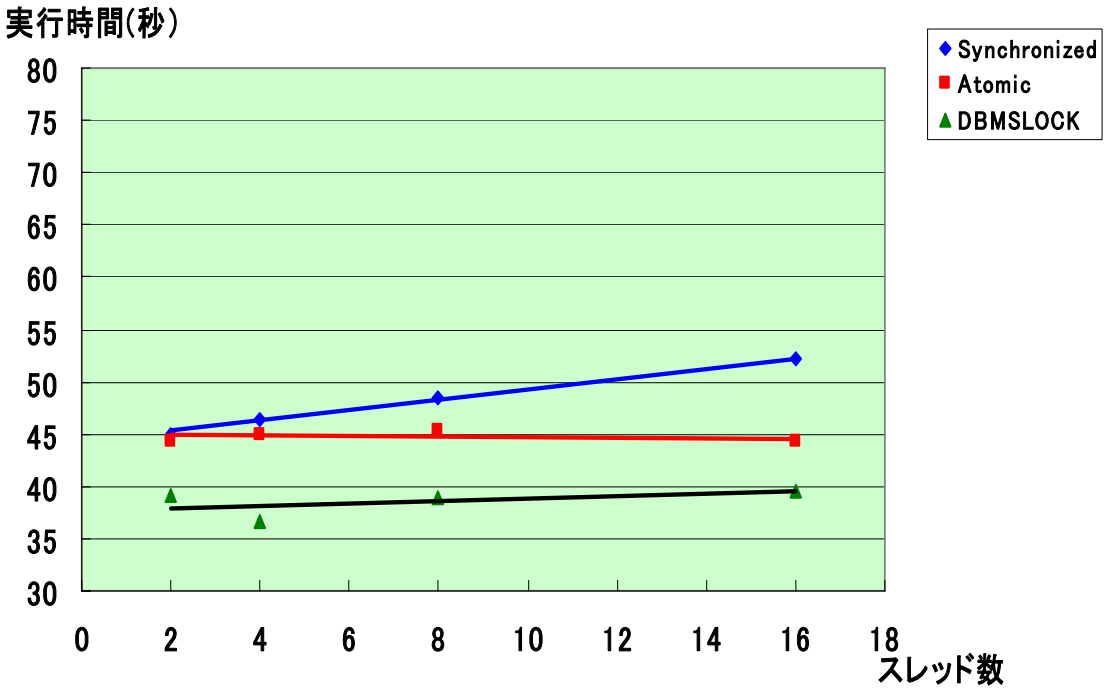


図 8.6 SQL 文が 10000 行の時の結果（縦軸：実行時間、横軸：スレッド数）

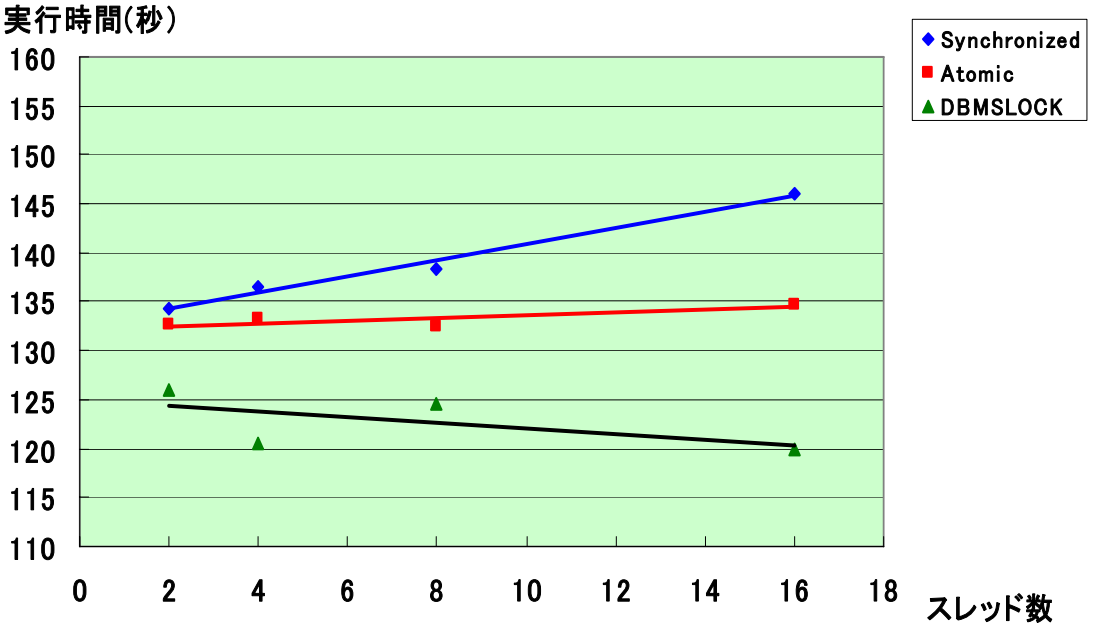


図 8.7 SQL 文が 30000 行の時の結果（縦軸：実行時間、横軸：スレッド数）

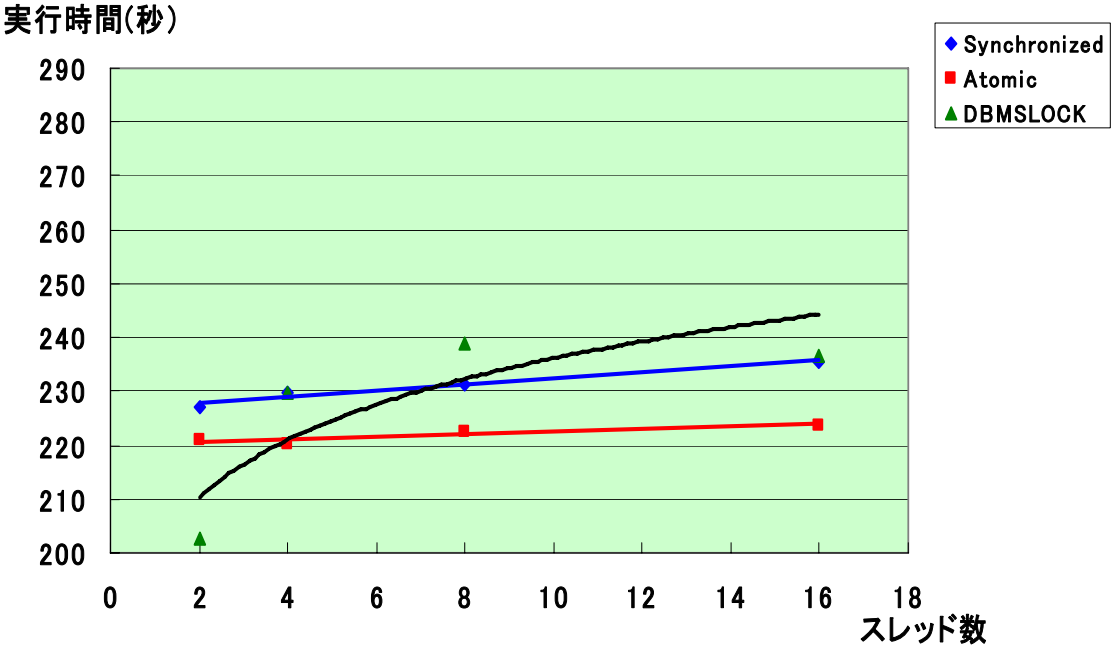


図 8.8 SQL 文が 50000 行の時の結果（縦軸：実行時間、横軸：スレッド数）

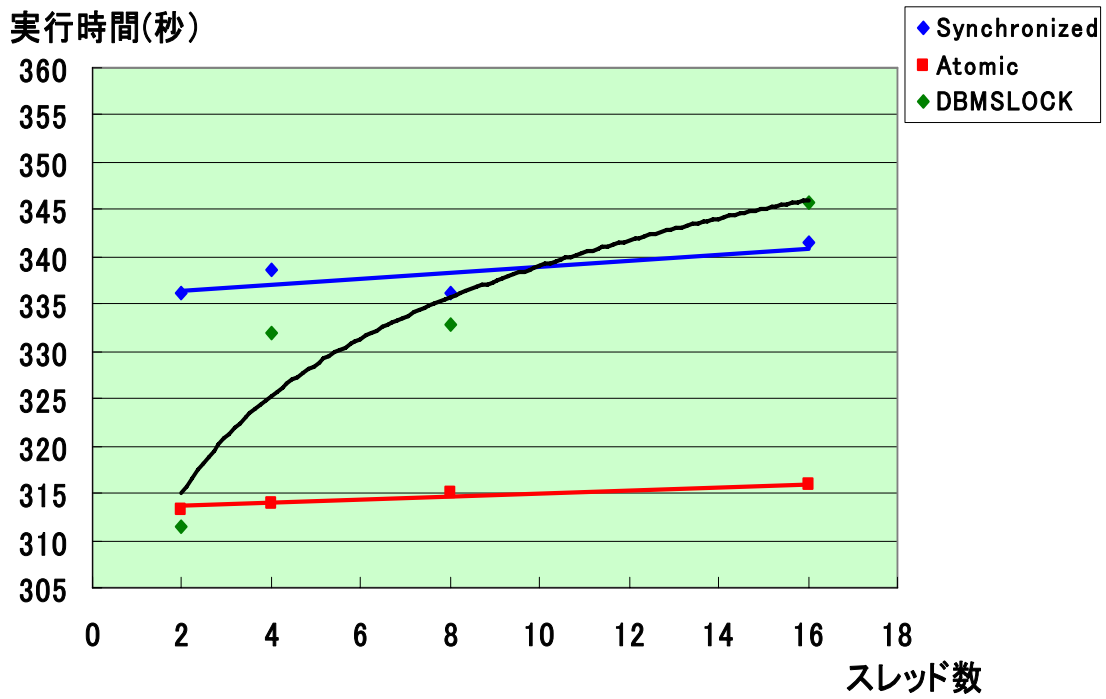


図 8.9 SQL 文が 70000 行の時の結果（縦軸：実行時間、横軸：スレッド数）

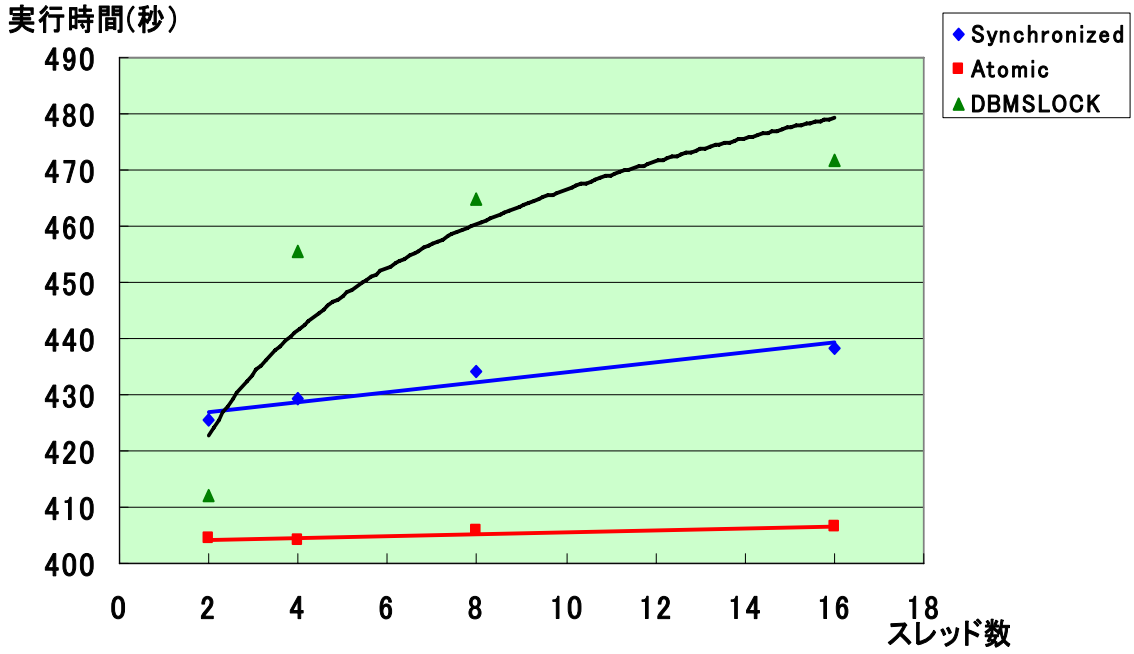


図 8.10 SQL 文が 90000 行の時の結果（縦軸：実行時間、横軸：スレッド数）