XMLのためのトランザクション管理

斉藤太郎[†]　森下真一[†]

XML はデータ構造の変化に対する自由度が高く記述が容易なことから, データ表現形式の事実上の標準として普及してきている. XML はもともとテキストであるため, XML のための問い合わせ言語の提案もユーザーが単体で情報を得ることのみを想定していたが, XML が広く普及した結果, オンラインで頻繁に更新される XML データをも管理していくことが要求されるようになった.

XML でのトランザクション管理において特徴的, かつ技術的に最も難しい問題は, 木構造が変化している際にも, クエリを行い, かつロックによりデータの一貫性を保つことである. また, スループットを向上させるためには, データベース全体をロックするような重い処理を回避し, ロックの範囲を局所化しなければならない. そこで本研究では, 木構造データでの一貫性と同時性をコントロールするため, 種々のロックモードを持った warning プロトコルを XML 向けに拡張することを提案する.

この提案を評価するため, XML 用トランザクション管理システム**Xerial** を設計・実装した. 本論文では, **Xerial** システムの構成について解説し, その性能を更新速度と同時性の観点から評価する.

Transaction Management for XML

Taro L. Saito[†] and Shinichi Morishita[†]

Because of its high flexibility for data restructuring and its ease of description, XML becomes a *de fact* standard of data representation. Several proposals for XML query languages seem to assume a single user environment due, in part, to XML's own nature as a plain text. However, the widespread use of XML has dictated that XML should also be used for on-line processing, which requires capability of handling frequent updates.

One of the hardest technical problems specific to XML transaction would be to maintain data consistency by locks while we are executing query on changing tree-structures. Furthermore, to increase the throughput, care has to be taken so that the whole database should not be locked, but rather, locks should be localized. To achieve this goal, we propose a *warning protocol* tailored to XML, which controls consistency and concurrency of tree-structured data with several modes of locks.

To test the performance of our proposal, we developed a system, called **Xerial**. In this paper, we show the architecture of **Xerial**, and evaluate it from viewpoints of updating speed and concurrency.

1. はじめに

XML はタグで囲まれたテキストで階層状のデータを表現するものであり, 任意のタグをユーザーが独自に使用できるその汎用性から, 近年では, さまざまなプラットフォーム間でデータをやりとりするためのプロトコルとして普及してきている.

また, XML では多種多様のデータを表現できるため, 例えば, 電子カタログなどで商品データを XML で記述し, 商品全体をデータベースとして蓄積するなど, XML データベースに対する需要が高まってきている.

これを従来の関係データベースで実現する場合, データベース設計者はあらかじめ, 商品テーブルに必用な, 商品名, 価格などの属性を全て知っている必要があり, 一度データベースを設計すると, システムを停止することなくスキーマを再構築するのは非常に難しい. 例えば, 商品に新たな属性を追加したり, 第一正規系[3),11)]を崩して商品名を 2 つにするなど, データ構造そのものを変化させたい場合である. その点, XML はデータ構造をデータそのもので表現する自己記述形式であるため, データ構造の変更が容易であり, データベースの設計段階での準備をさほど必要としないので手軽にデータベースを構築できるというメリットがある.

実用的には, 電子カタログをオンラインデータベースとして公開し, 随時, 新しい XML データを追加しながらユーザーに商品情報を提供する場合や, 銀行口座のように, 払い戻しや複数の振込み操作が同時に起

† 東京大学 情報理工学系研究科コンピュータ科学
Dept. of Computer Science, Information Science and Technology, University of Tokyo

こる状況を想定しなければならない. 銀行口座など特にデータの一貫性が重視される状況では, 適切な排他制御を行って口座の金額に矛盾が生じないような制御システムが必要となる.

XML からデータを取り出すための記述としては, 関係データベースにおける SQL のように, World Wide Web Consortium(W3C) が XQuery[1] という XML に対する問い合わせ (クエリ) 言語を提案し標準化を進めている. しかし, データベースとして XML をとらえる場合, 更新操作の定義が存在しない XQuery では不十分であるため, XQuery に更新用の記述を拡張する案もでできている[8].

しかし, 従来の XML に関する研究は一人のユーザーが XML 文書にアクセスすることを暗黙のうちに想定しており, 複数のユーザーが同時に XML データを読み書きする状況まではほとんど考慮されていなかった. それというのも, 現在ではクエリの効率をあげるための XML ストレージは多く考案され[9], 8) でも使用されているように, 関係データベースにマッピングするものが主流であるが, このようなマッピングでのトランザクションでは, 必然的に多数のテーブル間での join 操作が要求され, 結局は全てのテーブルにロックをかけて排他制御を行うために, 並列性が得られないのである[5]. このため, ロックの範囲を局所化するためのデータベース設計が重要となる.

XML におけるトランザクションで難しい点は, XML の木構造を探索しながらロックをかけなくてはならないことである. 木構造に関するロック管理方式には, 任意のノードから入ってトランザクションを開始できる Tree プロトコル[3),6),11)] があるが, まず, パス表現で開始ノードを選ぶとき, XML は非決定的な木オートマトンとみなせるため, 必ずしもノードを一つに特定できないという問題がある. また, 実際にデータベースにアクセスすることなく開始ノードを特定するには, DTD やスキーマを定義する必要があり, さらに, データ構造が更新されるたびにスキーマを再構築するコストがかかってしまう.

DTD やスキーマのない状況で, クエリを行いつつロックの範囲を局所化するために, 我々は部分木レベルでのロックを採用し, その実現方式として, 階層構造をとるデータに対してロックをかけるルールである warning プロトコル[3),4),11)] を XML 向けに拡張した. warning プロトコルは, 排他制御に通常使われるロックに加えて, warning(警告) をノードに置くことによって, 他のトランザクションに, 木構造の奥にアクセスしているトランザクションの存在を知らせるためのものである.

本研究では, Warning プロトコルに基づいて, 頻繁な更新操作に耐え得るデータベースシステム**Xerial** (エクセリアル) を設計・実装した. Xerial システムでは XML の木構造を抽出してデータベース化し, 挿入・削除の操作を伴うトランザクションが並行してデータベースを読み書きする際にも, トランザクションが一つずつ順に実行されたのと同様 (整列化可能) な実行結果を得ることを可能としている. Xerial の名前の由来は, このように XML トランザクションを serial(整列) に実行できるというところから来ている.

さらに, データベースシステムには, トランザクションの中断やシステムの異常など, 様々な障害からデータを保護する機構が欠かせない. Xerial では, 障害からの回復をも考慮し, warning プロトコルと 2 相ロック[3),11)] の考え方を合わせたトランザクションを設計している.

2. 関 連 研 究

XML におけるトランザクションを中心に扱った研究は, 我々の知る限り, Helmer らが 2 相ロックをノードやポインターレベルで応用したもののみである[5]. しかし, ノードやポインター単位での問い合わせをユーザーが手で記述するのはかなりの負担であり, 細かいロックの数が多いため, ロックマネージャーの負担が重くなることが予想される. あいにく, 5) では問い合わせ言語や実験結果が与えられてないので性能の評価はできないが, 我々の Xerial と比較して実験できれば面白いであろう. この意味で, 本研究は, XML におけるトランザクションについて真剣に取り組み, 実験的に評価した初めてのものであると言えよう.

3. 実用的な例

まず, XML データベースにおいて, 木構造という特徴を活用できる例から紹介することにしよう. **図 1** は, TPC-C ベンチマーク[10] のデータモデルを XML で記述した例である. TPC-C は関係データベースにおけるオンライン・トランザクションの性能を評価するためのベンチマークで, このモデルは企業内のデータウェアハウスを表現しており, それぞれのウェアハウスには地域ごとの顧客情報と, 顧客からの注文のデータが蓄えられている.

この階層モデルを関係データベースで表現したものの欠点は, 注文レコードの一意性を保つために, 各注文レコードが属するウェアハウス, 地域, 顧客の ID を全て一つ一つの注文レコードに冗長に保持しておかな

```
<company>
  <warehouse id="A">
    <district id="B-001">
       <customer id="M-0023">
          <name> McDonald </name>
          <order id="1">
             <item> HB pencil </item>
             <price> 15 </price>
             <num> 12 </num>
             <status> undelivered </status>
          </order>
           ...
       </customer>
        ...
    </district>
     ...
  </warehouse>
   ...
</company>
```

図 1 TPC-C データセットの XML 表現例
Fig. 1 XML Representation of TPC-C Dataset

ければならない点である[10]. しかし, XML では, 階層構造上の位置が意味を持つので, 注文のタグにそのような ID を記述せずとも, どの顧客の注文かを特定することができ, 非常に簡潔にデータを表現することができる.

この XML を木構造のデータベースにすると, 複数のユーザーが地域ごとに, 新しい顧客や注文の追加をノードの挿入操作で行い, 顧客は自分の注文状況をオンラインで確認し, 品数を増やしたり, 注文を取り消したりをノードの更新・削除の操作で行うことができる. 個々の注文の内容は XML で記述するため, 顧客のニーズや状況に応じて, スキーマの枠を越えてさまざまな情報を埋め込むことができ, より自由度の高いデータベースを設計できる.

Xerial では, XML の階層構造によってデータを分類するメリットと, 任意のデータを記述できる汎用性を生かしたデータベースを構築し, その上でデータの一貫性を保つためのトランザクション管理を行っている.

4. Xerial の仕様

4.1 データモデル

Xerial では, XML を解析して, その木構造とデータを 3 つのテーブルに分割してデータベース化する.

図 1 の XML を例に説明する. まず, それぞれのタグに一意の ID を割り当て, タグ名に#を区切り文字としてその ID を加える操作を行う. そして, XML の木構造を, 以下のように親ノードから子ノードへのポインターの形にして取り出す (index table).

```
company#1    -> warehouse#2, ...
warehouse#2 -> district#3, ...
district#3  -> customer#4, ...
customer#4  -> name#5, order#6, ...
order#6     -> item#7, price#8,
               num#9, status#10, ...
```

XML では通常, 木が複数並列に並んで森を形成しているので, それぞれの木を検索しやすくするために, 唯一の根 (root) として,

```
root#0      -> company#1
```

を加え, XML 文書全体を 1 つの木として扱う. タグに囲まれたデータに関しては,

```
7   ->  HB pencil
8   ->  15
9   ->  12
10  ->  undelivered
```

のように, それぞれのタグ ID からデータへのテーブルを作って格納する. (data table)

タグに付加されている attribute(属性) データは,

```
warehouse#2 -> id="A"
district#3  -> id="B-001"
customer#4  -> id="M-0023", index="M"
order#6     -> id="1"
```

このように, 一意なタグ名からそれぞれの属性データ含むデータへの関連付けがなされたテーブルを作る (attribute table).

以上の 3 つのテーブルがあれば, データベースから元の XML 文書を復元することも可能である.

4.2 XML の探索

XML で目的のノードを選択するための記述には, XPath[2] のように, ルートからのタグのパスで表現するものが多く使われる. 今, 図 1 の `warehouse` データを指定するためのパス表現は, 以下のようになる.

```
/company/warehouse
```

このパス表現で表されるノードを見つけるためには, index table から `root`, `company`, `waerehouse` のレコードを順に辿るのだが, 実際には `warehouse` には複数の候補が存在するので, 特定の `warehouse` を選び出すためには, すべての `warehouse` ノードを一つ一つ読み込んで, 中のデータからそれが更新したい対象かどうかを調べることになる. データを読み込んでいる最中に, 他のトランザクションによってデータが書き換えられないようにするためには, すべての `warehouse` ノードにロックをかける必要があり, 実際には更新の対象でないノードにまでロックをかけることになる. かといって, ロックの数を減らすために `company` から始まるより大きな部分木単位でロックをかけると, 並

列性がなくなってしまう.

Xerial ではこの問題を解決するために, タグで囲まれた要素と attribute データの違いを明確にする. Xerial では, attribute table 上のデータは読み込み専用と定義し, index table の補助としてパスを探索する目印にする. 例えば, attribute に ID やカテゴリなどを記述しておくと, 探索範囲とロックの対象を小さく絞ることができる.

4.3 操作の種類

この研究の目的のため, Xerial での更新操作には, 挿入, 削除, 読み込み, 書き込み, と基本的でかつ木構造へどのような操作を行っているかがわかりやすいもののみを用意した. 部分木間のデータのやりとりのように join 操作を含む表現は今回は扱わない. 命令の定義は XQuery[1] の FLWR(for, let, where, return) 構文との親和性を考慮して構成している.

命令の定義

- `SET $var = XPath`
  変数`$var` に `XPath` で指定されるノードの集合を束縛する.
- `FOR $var IN XPath`
  `XPath` の要素を一つずつ順に`$var` に束縛しながらループを実行する.
- `WHERE` *condition*`, ...`
  condition `:= $var(/XPath)` *operator const*
  operator `:= > | < | >= | <= | =`
  変数`$var` の要素の, `$var(/XPath)` で表される部分が *condition* を満たすか調べる.
- `RETURN $var | subtree($var)`
  変数`$var` で示されるノードのデータまたは, 部分木全体の XML を返す.
- `INSERT $var {` *xml-document* `}`
  xml-document を`$var` の子ノードとして挿入する.
- `DELETE $var IN $pvar/`*tag*
  `$pvar` を親ノードとする子ノード`$var` とその部分木全体を削除する.
- `WRITE $var` *expr*
  expr `:= $var` *op* (`$var` | *const*) | *const*
  op `:= + | - | * | /`
  `$var` のデータに, *expr* で計算された値を書き込む.

5. Warning プロトコル

木のように階層構造をもつデータでは, 読み書きの対象となるノード一つ一つに細かい粒度のロックをかけるよりも, より大きな単位で部分木レベルのロックを用いる方がロックマネージャの負担が軽くなり, トランザクションの効率を上げることができる.

部分木レベルでロックをかけるときには, その部分木の内部にロックを持っているトランザクションが存在しないことを確認する必要がある. これを部分木全体を探索して確認するのではなく, 通常のロックに加え, warning ロックという新しい種類のロックによって確認する手順を定めたのが warning プロトコル[3),4),11)] である.

5.1 ロックモード

ノードにかけるロックには, 読み込み用の S(Shared:共有) ロックと, 書き込み用の X(eXclusive:排他) ロックの 2 種類がある. これらのロックをかけるためには, そのノードまでのパス上にそれぞれのモードに応じた warning を置かなければならない. warning には, S 用の IS, X 用の IX の 2 つがある.

5.2 木構造を探索する手順 (プロトコル)

- すべてのトランザクションは root から入る.
- 現在の地点よりも, ロックをかけて読み書きしたいノードが奥にある場合, かけるロックの種類に応じた warning を現在のノードにおき, 子ノードに移動する.
- 他のトランザクションによって, すでにノードに warning やロックが置かれていた場合, **表 1** (この行列は対称) を見て, 共存できる warning やロックのみをかけられる. 共存できない場合は, そのノードが開放されるのを待たなくてはならない.
- 一度かけた warning とロックは, そのノードの奥に自分のかけた warning やロックがなくなるまでは放せない.

ロック共存行列 (表 1) は, 例えば, あるノードに IS が置いてあったとき, 部分木の奥に S を持つトランザクションが存在する可能性があるので, ここに X を置くのは認めない, ということ意味している. 従って, このプロトコルに従ったトランザクションでは, ノードへの S ロックと X ロックは部分木全体へのロックと同等になる.

5.3 木構造が動的に変化する場合への拡張

ノードの削除や挿入を行う際には, その対称となるノードの親に X ロックをかけてから実行する必要がある. さもなくば, 親から子への枝をたどる操作と, 子の挿入・削除の操作が同時に起こることがあり, 見つかるべきものがみつからない, などトランザクションの一貫性が損なわれてしまうからである. この一貫性を保つために, warning プロトコルに以下の 2 つのルールを追加する.

- ノードの挿入・削除の際には, 対称となるノード

Lock held \ Lock requested	IS	IX	S	X
IS	Yes	Yes	Yes	No
IX	Yes	Yes	No	No
S	Yes	No	Yes	No
X	No	No	No	No

表 1 ロック共存行列
Table 1 Lock Compatibility Matrix

の親に X ロックをかける.

- ノードを辿るには, 親ノードに warning を置いて初めてその子ノードへのポインタや, 子の attribute を調べることができる. (すでに部分木レベルのロックを持っている場合を除く)

2 つ目のルールは, ノードに warning を置くまで index table 上のレコードを参照できないことを意味する. 挿入・削除の操作の際には X ロックが置かれていて, 他のトランザクションはそこに warning を置くことができないので, 更新途中の index table レコードを保護できる.

6. 整列化可能性と障害回復

トランザクションを並列に実行する場合には, その実行結果が, トランザクションがある順番で一つずつ独立に実行されたのと同じ (整列化可能) であることを保証する必要がある. 読み書き専用のロックを伴う場合には, **コンフリクト整列化可能**[3),11)] という概念が用いられ, 次に述べる重要な定理がある.

Theorem 6.1 2 相ロックプロトコルに従うトランザクションの集合からなるスケジュールはコンフリクト整列化可能[11)]

ここで, warning プロトコルにもう一つ次のようなルールを加える.

- トランザクションの終了までは取得した全ての warning とロックを持ち続け, 終了と同時にその全てを解放する.

このルールにより, ロックの取得と解放を 2 段階に完全に分離する 2 相ロック[11)] を warning プロトコルに適用でき, 拡張された warning プロトコルでは, ロック共存行列 (表 1) に反するロックを 2 つのトランザクションが同時に獲得することはありえないことから, 定理によりコンフリクト整列化可能性が保証される.

また, 2 相ロックを適用することにより, 更新途中のデータを読む Dirty Read が生じなくなり, それに伴う Cascading Rollback も未然に防げるという利点がある[3),11)].

7. トランザクション言語

7.1 問い合わせ言語と更新操作

XML におけるトランザクションでは, まず読み書きの対象となるノードを特定するための問い合わせが必要となる. 従って, トランザクションを実行するための記述は, 問い合わせ部分と更新操作を混合したものとなる. Xerial では, XQuery をベースに, 4.3 で述べた命令の記述を合わせ, warning プロトコルのために特別なルールを設けた構文を定義した.

基本的な構文

Xerial でのトランザクションの構文は, 大きく 2 つに分かれている. Warning プロトコルで, 目的のノードまでのパス上に warning を置いていくための `SET` 部分と, 実際にノードデータの読み書きを行う `TRANSACTION` 部分からなる.

```
SET $var = XPath
TRANSACTION $var {
  XQuery and Update Operations ...
}
```

最初の `SET` で定義される `$var` までのパスには warning が置かれ, `$var` のノードを親とする部分木には `TRANSACTION` 内での操作に応じたモードのロックがかけられる. `TRANSACTION` 内で行われる読み書きは, `$var` で指定された部分木内のノードに対して実行できる.

7.2 具 体 例

ここでは図 1 の XML を用い, 基本的な操作ごとにトランザクションの例を挙げる.

7.2.1 条件を満たすノードの検索

```
SET $x = /company/warehouse[@id="C"]
TRANSACTION $x {
   FOR $y IN $x/district,
       $z IN $y/status
   WHERE $y/name = "dist A"
   RETURN $z
}
```

上の例は `root` と `company` に warning を置き, `<warehouse id="C">` に S ロックをかけてから条件を満たす `district` を探し, その `status` データを返す.

7.2.2 ノードの挿入

ノードの挿入操作では, 挿入する位置の親ノードまでのパス上に IX で warning を置き, 親には X ロックをかける. 以下の例は, `district` ノードに, 新しい `customer` を挿入する操作を表す.

```
SET $x0 = /company/warehouse[@id="A"],
    $x = $x0/district[@id="B-001"]
TRANSACTION $x {
   INSERT $x {
      <customer id="D-144">
         <name> David </name>
      </customer>
   }
}
```

7.2.3 ノードの削除

ノードの削除は, 対象のノードだけではなく, その親のノードからのポインターも同時に削除するため, DELETE 文では親ノードを明示的に記述する.

```
SET $x = /(中略)/customer[@id="C-031"]
TRANSACTION $x {
   DELETE $y IN $x/order
   WHERE $y/date = "19/02/2002"
}
```

上の例では, $x に束縛された customer ノードに X ロックをかけ, その子である order ノードのうち, 条件を満たす date が含まれているもののみを削除する.

7.2.4 データの更新

WRITE 文でデータを更新する際, 対象となるデータまでのパスがあらかじめわかっていると, 下の例のようにロックの粒度を細かくし, トランザクションの並列性を高めることができる.

```
SET $x0 = /company/warehouse[@id="A"]
    $x1 = $x0/(中略)
    $x = $x1/order[@id="5"]/num
TRANSACTION $x {
   WRITE $x $x+10
}
```

8. 実　　装

Xerial は C++で実装され, トランザクションの実行は主にクエリコンパイラとスケジューラの 2 つで行われる. データベースサーバをモデルとして設計し, クライアントからの要求に対してスレッドを生成して, マルチスレッド環境で稼動する.

8.1 BerkeleyDB

データベースには, データを効率良く格納しレコードを検索するためのディスク管理や, 複数の読み書き, ロックの取得・開放の同時処理など, 様々な機能が必要とされる. このため, 本研究では BerkeleyDB[7] という, キーとデータのペアからなる簡単な構造のレコードを格納するデータベースライブラリを使用している. BerkeleyDB にはあらかじめ, B 木やハッシュなどのテーブルをデータベースとして構築する機能と, ロックやトランザクション, 障害回復のための機能など, 通常のデータベースに必要な機能のほとんどがカスタマイズ可能な形で提供されている.

8.2 XML の変換

XML 文書をデータベースに変換するために, xml2db というプログラムを実装した. xml2db では, XML ファイルを自作の SAX パーサで一行ずつ読み込みながら, 4.1 で述べた 3 つのテーブルに変換していく.

8.3 クエリ・コンパイラ

クエリ・コンパイラは 7.1 でのトランザクション文を解析し, warning プロトコルに従ってデータベースにアクセスする手順を示すプログラムを生成する. 実際にデータベースにアクセスするまで具体的なデータはわからないので, ここで生成されるプログラムは抽象的なものとなる.

8.4 スケジューラ

クエリ・コンパイラで生成されたプログラムを, 適切な位置で warning やロックを挿入しながら, トランザクションを実行する. SET 命令の部分では warning をおきながら木を探索し, TRANSACTION で指定される変数にはロックを置く. TRANSACTION の内部では, FOR ループなどの変数に適宜ノードを束縛し, 更新操作を行う.

9. 実　　験

9.1 ハードウェア

実験には, Peintium III 1GHz × 2(デュアル・プロセッサ), メモリ 2GB, 10000 RPM の SCSI ハードディスク 2 台を NTFS でフォーマットした Windows2000 マシンを用いている. データベースシステムでは, 一般にプロセッサの処理速度よりも, ハードディスクへのアクセス効率がスピードの鍵を握っているため, シングルプロセッサとデュアルプロセッサの比較実験は行っていない. ハードディスクはそれぞれログとデータベース用に分けて使用した.

9.2 目　　的

実験の目的は, XML 用に拡張した warning プロトコルによって, どの程度のスループットが得られるかを見ることにある. トランザクションの実行は, warning プロトコルを用いた**parallel execution** と, root の X ロックを取得する**serial execution** の 2 種類の方法を比較する. parallel execution ではスレッドの数だけ並列にアクセスし, serial execution ではデータベース全体へのロックと同等で, 関係データベースへ

Transactions	Lock Node	Mode	Insert	Modify	Delete	Average Execution Time (sec.)	S1 (%)	S2 (%)	
Search District	Warehouse	S	0	0	0	0.0031	40	5	特定の District を検索
Insert Customer	District	X	30	1	0	0.0110	20	10	XML 文書を挿入
Delete Customer	District	X	0	1	30 ~	0.0953	10	2	Customer を削除
Insert Order	Customer	X	22	1	0	0.0172	15	40	XML 文書を挿入
Write Payment	Customer	X	0	2	0	0.0078	10	25	支払い金額を更新
Delete Order	Customer	X	0	1	22 ~	0.0203	3	3	Order の削除
Order Status	Order	S	0	0	0	0.0063	2	15	特定の order を参照

表 2 更新するレコード数と混合の割合
Table 2 Number of Update Records and Mixture Percentage

のマッピングを近似したモデルとなる.

9.3 方　　法

今回は同時にデータベースへアクセス可能なスレッドを 50 個生成し, それぞれにトランザクションを 1 つずつ割り当てる. スレッドが解放されるたびに新たなトランザクションをスレッドに渡し, 絶え間なくトランザクションを実行し続け, 100,000 個のトランザクションが終了するまでの時間を計測する. また, トランザクションがスレッドに割り当てられてから終了するまでの時間を計測し, 平均の反応時間 (response time) も計算する.

測定に使用する XML は図 1 の例で, `warehouse` が 5 つ, それぞれの `warehouse` の下に `district` が 10 個ずつ, その下に `customer` が 50 個ずつ, `order` が 5 個ずつ, と階層状に連なった 11.5MB のものとした. tag, attribute, data のテーブルはそれぞれ, 3433271, 17555, 293160 個のレコードを持つ.

トランザクションは, 典型的な更新操作の例として, 条件を見たすノードの検索, 挿入・削除, 値の更新・参照など 7 種類を用意する (**表 2**). ロックをかけるノードは 1 つランダムに選び, トランザクションを生成する. 100,000 個のうち各操作の割合を, 大きい単位でのロックが多い**S1** と, `order` の挿入を中心とするより一般的な更新を想定した例である**S2** の 2 種類に分けて実験を行い, 並列性による効率の違いを検証する. 個々のトランザクションのコストについては, 単独で 10 回実行したときの平均時間を表 2 に載せた.

9.4 結　　果

100,000 個のトランザクションを処理するまでの時間の経過を**図 2** に示す. 各トランザクションについては, ロックの取得, 更新, 反応時間のそれぞれについて平均をとり, **表 3** にまとめた.

表 3 では**S1** と**S2** に共通して, parallel execution が, serial execution にロックの取得時間で勝り, 更新時間については serial の方が短くなっている. しかし, 全体としては parallel execution がより短い反応時間でトランザクションを実行し, warning プロトコルによる性能の向上が伺える.

S1 では warning プロトコルによる並列度が低いため, 全体のロックを取る手法に近くなり, **S2** よりは parallel と serial の性能差が小さくなっているが, それでも 2 倍以上のスループットを実現している. さらに, **S2** においては 100,000 個のトランザクションの実行時間を 5 分の 1 以下にまで短縮することができた.

9.5 並列性とディスクアクセスのトレードオフ

ここでより並列性を上げるために, 同時にデータベースにアクセスするスレッドの数を増やしたとしても効果は望めないであろう. なぜなら, 見た目の並列度はあがるが, 同時にディスクにアクセスしているスレッド数に比例してディスクの検索効率が悪くなるからである. このことは, 表 3 の update time が, serial execution の方が速いことからも伺える. スレッド数の増加は結果として, トランザクションの実行時間を長くし, ロックを長く保持し続けることによってスループットの減少につながる.

この影響を減らすためには, XML から作成した 3 つのテーブル別々のディスク上に置き, ファイルのシークタイムを減らしてディスクのアクセス効率を上げるなどの方法が考えられる.

10. 結論と今後の課題

本研究では, XML の自己記述形式という特徴を, より自由度の高いデータベースを構築するためのものとしてとらえ, その上で並列にかつ安全にトランザクションを行う方法について述べ, その効率を実証した.

また, 記述上での便利さのみで使われ, タグデータとの明確な違いがなかった attribute が, XML トランザクションで有効に活用できることを示し, XML の記述形式にひとつの示唆を与えた.

トランザクション言語に関しては, 基本的な操作の

		Lock Time	Update Time	Response Time	Elapsed Time (100,000 Transactions)	Transactions Per Second
S1	Serial	1.35492	0.01682	1.37406	2751.72	36.341
	Parallel	0.51611	0.05503	0.57365	1151.06	86.876
S2	Serial	1.91648	0.01042	1.92879	3867.19	25.859
	Parallel	0.30188	0.03580	0.34242	693.06	144.288

表 3 トランザクションの平均反応時間 (単位:秒)
Table 3 Average Response Time(sec.)

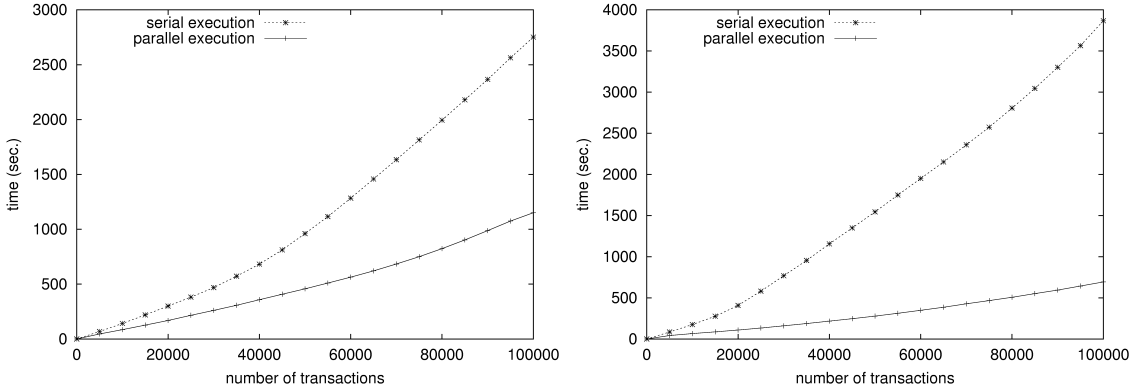

図 2 トランザクションの処理時間 (左: **S1**, 右: **S2**)
Fig. 2 Transaction Execution Time (left: **S1**, right: **S2**)

みを定義し, XQuery を元にすることで, より複雑な操作にも対応できるよう拡張性を残した. しかし, 複数の木の間でデータのやりとりを行う場合, 2 相ロックではデッドロック[11] の危険が増すことになるので, 工夫が必要である.

従来の研究では, XML を情報検索のためのデータとして扱うことが多く, 主に問合せの効率を高めることに関心が集まっていた. このことを考慮すると, attribute を目印として使わず, 大きなレベルでロックを取る必要が生じた時にも, いかにトランザクションのスループットを向上させるかが今後の課題となる.

本研究では整列化可能性という, 最も高いレベルでデータの一貫性を保護する理論を説明したが, 広い範囲での検索のみを目的とし, リアルタイムな更新によって多少間違った結果が返ってきてもかまわないのであれば, 一貫性のレベルを下げて[4] トランザクションの効率を高めることも必要であろう.

参 考 文 献

1) D. Chamberlin, J. Clark, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu. XQuery 1.0: An XML query language, W3C working draft, 07 June 2001. www.w3.org/TR/xquery.
2) J. Clark, S. DeRose. XML path language (XPath) version 1.0, W3C recommendation, 16 November 1999. www.w3.org/TR/xpath.
3) H. Garcia-Molina, J. D. Ullman, and J. Widom. Database system implementation. Prentice Hall, 2000.
4) J. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Granularity of locks and degrees of consistency in a shared data base. *IFIP Working Conference on Modeling of Data Base Management System*, pages 365-394, 1976.
5) S. Helmer, C.-C. Kannea and G. Moerkotte. Isolation in XML bases. *Technical Report of the University of Manheim*, 15/01, 2001
6) A. Silberschatz annd Z. Kedem, Consistency in hierarchical database systems, J. ACM 27:1 (1980), pp 72-80.
7) Sleepycat Software. BerkeleyDB. Available from www.sleepycat.com.
8) I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, 2001.
9) F. Tian, D. DeWitt, J. Chen and C. Zhang. The design and performance evaluation of alternative XML storage strategies, submitted for publication, 2000. 18
10) Transaction Processing Performance Council. www.tpc.org.
11) J. D. Ullman. Principles of database and knowledge-base systems. Volume I. Computer Science Press, 1988.