TRANSACTION MANAGEMENT FOR XML
XMLのためのトランザクション管理

by

Taro L. Saito
斉藤 太郎

A Senior Thesis
卒業論文

Submitted to
the Department of Information Science
the Faculty of Science, the University of Tokyo
on February 13, 2002
in Partial Fulfillment of the Requirements
for the Degree of Bachelor of Science

Thesis Supervisor: Shinichi Morishita　森下 真一
Associate Professor of Information Science

ABSTRACT

XML is a format of text, which represents tree-structured data. It has higher flexibility for data restructuring and is easy to describe, so it has become widespread as a *de facto* standard of data representation. Several proposals for XML query languages seem to assume single reader environment, because of XML's own nature as a plain text. Inevitably, however, the widespread of XML demands the advanced use of XML for on-line processing, which requires capability of handling frequent updates made by multiple readers and writers. Furthermore, XML data management system should provide recoverability from transactional aborts and system failures. One of the hardest technical problems specific to XML transaction management would be to handle update operations of changing tree-structures while maintaining data consistency. To increase the throughput, we should avoid locking the whole database but ought to localize locks on smaller sub-trees. 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.

論文要旨

XML は木構造をテキストで表現するものであり, 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 の問合せ言語もいくつか提案されてきたが, そもそも XML はテキスト形式であるため, その主な目的はユーザーが単体で情報を得ることに限られていた. しかし, XML が広く普及した結果, 単なるデータ表現形式に留まらず, XML を使って頻繁に更新されるオンラインデータをも管理するニーズが出てきている. この際, 新たな機能として, 複数の読み書き操作を同時に処理する能力と, トランザクションの中断やシステムの障害からでも回復できる能力が要求される.XML でのトランザクション管理において特徴的で技術的に最も難しい問題は, 木構造を更新する操作の効率的な処理である. この操作はスキーマの変更に匹敵するため, 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, できるだけ局所的で軽いロックにより一貫性を保証し, スループットを向上させなければならない. そこで本研究では, warning を取り入れ種々のロックモードを持ったプロトコルを 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.

Acknowledgements

We would like to thank Shinichi Morishita for discussions and reviewing the paper, and members of Morishita Laboratory for their useful comments to our ideas and implementation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent years, there has been widespread interest in XML as a format of data representation. Many organizations have begun to provide several different kinds of information in XML. For example, there are XML version of DBLP bibliographies and the NCBI's protein resources. XML is a self-describing data format that uses tags to enclose data, and to construct tree-structured elements. These tags may be nested if required. In comparison with other markup languages such as HTML, XML is different, as it provides a capability for describing data structures and for defining arbitrary named tags, not for screen representation itself.

With the rapid growth of e-business, these properties have attracted considerable attentions. E-catalog is one of the more useful examples. The description of an item in catalogue should be flexible and should be able to accommodate many kinds of information. Since XML fully satisfies these requirements, it is now widely used in such applications.

Let us consider representing e-catalogs in traditional relational databases (RDB). The database designers have to know all of the catalogue attributes in advance, such as a product's name, price and so on. Once we create the database, it is very difficult to change its schema. On the other hand, XML's self-describing format is flexible, and it allows changes to the data structures. Therefore, the initial design of an XML database schema doesn't require as much initial effort, since we know that it can be easily changed later.

In spite of its flexibility, XML is verbose, since it uses tags to label the data. This disadvantage, however, has not been overly serious, since disk storages become larger and less expensive. In addition, compression techniques such as representing tags as integers [5] can be used to reduce the disk storage requirements to a more efficient

size.

When using XML for a database, we have to create indices in order to find target objects efficiently without having to scan entire XML documents. These indices should be suitable for handling the tree structure of XML. In addition, in order to increase the accessibility to the database, we need a query language for XML with similar capabilities to those supplied by SQL for a RDB. The World Wide Web Consortium (W3C) is developing a standard for an XML query language, called XQuery [1]. However, to modify the data or to insert new XML elements to the database, not only queries, but also updating operations are required. Since XQuery does not specify any update operations, initial research has appeared on incorporating update notations to XQuery [7].

Most of the current research work on XML implicitly assumes that only a single user is accessing an XML database, since XML is originally a plain text. Practically speaking, however, we have to be able to manage the scenario that multiple users are attempting to modify the database simultaneously. For instance, an e-catalog realized as an online XML database typically requires the capability to simultaneously insert new XML items while releasing the catalogue data to customers. Bank accounts should be able to manage concurrent refund and deposit actions. In applications such as bank account, maintaining the consistency of the data is extremely important. Consequently, we have to devise some method for avoiding simultaneous updates on the same record by employing locks on the database.

In the process of transforming text to a database, it is necessary to define the granularity of the database elements that must be locked. If the XML is a text, it is natural to lock the entire document. However, once it is regarded as a database, the loss of concurrent access due to the locking of the whole database would be a serious limitation.

In this paper, we propose a subtree-level locking scheme for concurrency. As a means of managing multiple update transactions in XML, we extend the *warning protocol* [4, 3] for XML. This protocol was originally designed to perform locking of hierarchical data to maintain the data consistency when multiple users were attempting to gain access to the database. The warning protocol uses a special lock called a "warning". By placing a warning on a node, we can ensure that no transaction is allowed to update the node. To realize the warning protocol, we have created a data model for XML. This data model preserves the tree-structures of XML and enables us to execute update operations including node insertions and deletions in parallel. Furthermore,

transaction schedules following the extended warning protocol become serializable [9]. As a result, the effect on the database has the same effect as that realized by executing the transactions independently in some specified sequence.

Based on our extension of the warning protocol, we have implemented a transactional XML database system called Xerial. The name 'Xerial' is derived from its ability to execute XML transactions in serial. In addition to providing update operations, Xerial also realizes a considerable improvement in transaction throughput compared to methods that lock the entire database. In addition, it is essential for database systems to protect the data from crashes, such as transaction aborts and system failures. We have combined two-phase locking and the warning protocol in order to provide recovery from such failures.

To the best of our knowledge, this is the first serious study on how to realize serializable and concurrent transactions on XML by locking schedules following the warning protocol.

This paper is organized as follows: Chapter 2 presents a motivating example of an XML transaction. Chapter 3 describes a data model of Xerial and a way to traverse an XML database. This model is quite simple. In later chapters, however, we describe how multiple update operations can be executed serially by using this model. Chapter 4 explains the warning protocol and the approach that we used in extending it for an XML transaction. The background of database theories such as serializability and recoverability is given in Chapter 5. In Chapter 6, we introduce a couple of new operators realizing the extended warning protocol, and we incorporate them into XQuery for querying and updating XML with special consideration for scalability. Chapter 7 explains the implementation of Xerial. Chapter 8 describes our experimental results. Our conclusions and recommendations for future work are presented in Chapter 9.

Chapter 2

Motivative Example

We start by illustrating a practical example that makes use of XML's tree-structure. Figure 2.1 shows an XML representation of a TPC-C [8] data model (Figure 2.2). TPC-C is a benchmark for evaluating the performance of online-transaction processing on relational databases. This model represents warehouses in a company, and it consists of information about their districts, customers, and their order details.

One of the defects in representation of the TPC-C model in relational databases is the redundant appearance of IDs (see Figure 2.3). Every order record contains the warehouse ID, the district ID, and the customer ID for the order, as a primary key. This representation looks somewhat verbose. In XML, however, such IDs are not kept in one record. The path from the root to the node representing the order in the tree tells us its identification. This XML representation avoids the redundant appearances of IDs seen in the RDB model. In addition, it is free from the limitation of the schema.

Consider creating a tree-structured database from this model. When a new customer joins the company, we will insert a new node to the database. The clients can see their order status and, if necessary, they can add new orders, or cancel orders. We can realize these operations by modifying or deleting the corresponding nodes.

If we execute such operations without any consistency control, the database will end up in an inconsistent state. For example, consider an example where an employee is trying to execute a transaction that writes 'undelivered' to the status of an order. If a forwarding agent was able to complete a transaction that wrote 'delivered' to that same order, then the completion of the other transaction would overwrite the "delivered" with 'undelivered' and the data would be inconsistent.

In order to avoid such inconsistencies, the Xerial system locks the appropriate nodes

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

Figure 2.1: XML Representation of TPC-C Data Model

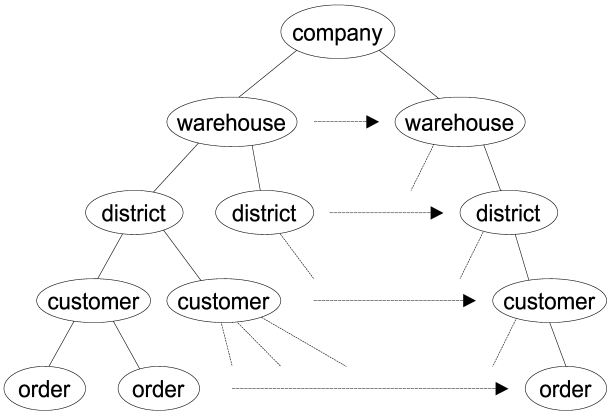

Figure 2.2: TPC-C Data Model

W-ID	D-ID	C-ID	O-ID	Item	Price	Num	Status	...
A	B-001	M-0023	1	HB pencil	15	12	undelivered	...
A	B-001	M-0023	2	Notebook	...	...	...	
A	B-001	M-0023	3	...				
A	B-001	M-0023	...					
...	...	...						

Figure 2.3: RDB Representation of Order List

so that another transaction cannot enter nodes that are currently being updated.

Chapter 3

Xerial Specifications

In this section, we show how XML is handled as a database. By analyzing an XML document, Xerial creates three database tables. These tables contain information about the tree structure, the tags, and the attribute data of an XML document. The following two sections explain how to make these tables and how to use them to traverse an XML database. Finally, several operations for querying and updating an XML database are given.

3.1 Data Model

Many research projects have been carried out on the subject of XML storage, such as mapping to relational databases. There are benefits to using existing database systems. We can save the cost of programming or we can show that the translations are ready to use. These approaches, however, usually have some limitations owing to the properties of applied systems, such as the requirements of DTD information, the constraints on the data sizes of XML elements, and so on. Even though XML mappings to RDB's have become popular today, we have to reconsider their restrictions more carefully when considering transactional environments.

Given RDB mappings, consider the scenario where we include a tag name as an attribute of table records. In order to modify one of the records, we currently hold a lock on it. At the same time, another transaction appears on its parent node and attempts to move to one of the children. However, to select the target from these children, the transaction must wait until it can obtain a read lock on a child before reading its tag name on the record. If the previously locked child is not the target, the transaction will be able to proceed and no further time will be wasted. However, time is required for the transaction to obtain a read lock on a child, and the cost of

locking becomes significant as the number of children increases.

In order to avoid this problem, we can employ another method of mapping that groups similar tags names together and creates tables for each unique tag name. However, we must now create as many tables as unique tag names. If we know all of the tag names or limit the number of them in advance, this translation could be efficient. However, if we pursue the use of arbitrarily named tags, it is not efficient to create new tables every time we insert new tags.

Now, let us consider ways to mange these problems by illustrating our data model using XML as shown in Figure 2.3 as an example. In our data model, we regard the XML document as a tree that consists of nodes, pointers to children, and node values. In order to separate the tag names and node values, we first assign a unique ID to each tag name by using the delimiter '#' followed by the ID. Then, we trace the tree structure of XML, and create pointer lists to the child nodes as follows: (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, ...
...
```

By using the index table, we can traverse the structures of the tree. Furthermore, the index table has the ability to represent arbitrarily named tags.

XML usually has several trees that form a forest. In order to increase the accessibility of each tree, we add a unique root record as shown to construct a single tree.

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

Next, we collect the data enclosed with tags: (data table)

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

This table associates tag IDs with their data. Finally, we extract attribute data from tags, and make a mapping table from the tag names to the attributes: (attribute table)

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

Since the index table preserves the order of tags, it is possible to reconstruct the original XML documents from these three tables.

3.2 Traversing XML

To select specific nodes in XML, path expressions from the root to the target, like XPath [2] are commonly used. The expression to choose the district elements from Figure 2.1 is:

```
/company/warehouse/district
```

In order to find the target, we trace the index table in order of `root`, `company`, `warehouse`, and `district`. However, we usually need a further search to specify the node in order to uniquely select one of them, because the initial search result may contain several candidates of company, warehouse, and district. For each candidate, we trace its children, and decide from their data whether the candidate is our target. In addition, to ensure that the read data will not be changed by other update transactions during the search, locks on all of the district nodes are required. However, if we can distinguish our target node in advance, most of the locks are unnecessary.

How to select the target of locks is not a simple problem. We propose one solution that uses attributes as signs to select targets. To accomplish this, we clearly show the difference between tag data and attributes. In Xerial system, we allow modification of records in data tables, but not in attribute tables. By prohibiting changes to the attribute records, we can use them without locks for selecting child tags. For example, we write the IDs of the nodes to the attributes, and add them to the previous XPath expression as shown below:

```
/company/warehouse[@id="A"]/district[@id="B-001"]
```

By using the attribute data, we can select some of them before acquiring locks on the nodes. Thus, we can minimize the search space and the number of objects to lock. In transactional databases, this method of searching the elements significantly increases the degree of concurrency that can be realized.

3.3 Operations

In this paper, the update commands in Xerial are very simple, but they do include the fundamental operations for tree structures: node insertion, deletion, and modification of node values. We construct the definition of these operations in accordance with XQuery's FLWR (for, let, where, return) syntax. However, since we have not devised a mechanism for settling deadlock yet, we have not included any operation for communicating data between subtrees, such as join operations and references using IDREFs [1] that may cause deadlock.

Definitions of Query & Update Operations

- `SET $var =` *XPath*
  Bind the variable `$var` to nodes that are designated by the *XPath* expression.

- `FOR $var IN` *XPath*
  Execute the loops while binding `$var` to each element in the set designated by the *XPath*.

- `WHERE` *condition* , ...
  condition `:= $var(/`*XPath*`)` *operator const*
  operator `:= > | < | >= | <= | =`
  Check whether the elements of `$var(/`*XPath*`)` satisfy the conditions.

- `RETURN $var | subtree($var)`
  Return the node value of `$var` or the entire subtree of `$var` as XML documents.

- `INSERT $var {` *xml-document* `}`
  Insert the *xml-document* as child nodes of `$var`.

- `DELETE $var IN $pvar/tag`
  Delete the entire subtree whose root node is `$var`, which is one of children of `$pvar`.

- `WRITE $var` *expr*
  expr `:= $var` *op* `($var |` *const*`)` `|` *const*
  op `:= + | - | * | /`
  Write the value calculated by the *expr* to the `$var` node.

Chapter 4

The Warning Protocol

For tree-structured data, it seems efficient to lock each element that we read or write. This fine-grained locking may increase the concurrency of transactions, but the most annoying obstacle to this strategy is that we cannot know the elements to lock before traversing the XML tree. Moreover, if we use depth-first searching to find a specific element, there is a danger of searching too deep. Also the load of the lock manager increases in proportion to the number of locks that we hold, which will end up making the transaction throughput worse.

The solution to the problem of using small-grained locks involves a larger granularity of locks. By using subtree level locking, we show an efficient way to execute transactions in XML that are combinations of traversing and updating. Before we place a lock on a whole subtree, we must confirm that there are no transactions in it. The warning protocol [4, 3], which involves "ordinary" locks and "warning" locks, is a rule for managing subtree-level locks on a hierarchical database. Using this protocol, we can avoid the costs of searching the entire tree for the existence of another transaction.

4.1 Lock Modes

In the warning protocol, we have two ordinary locks, S and X (shared and exclusive), for read and write nodes, respectively. To place an ordinary lock on a node, we must place a "warning" on every node in the path from the root to this node. The warning locks are denoted by prefixing I (for "intention to") to the ordinary locks; IS represents the intention to obtain a shared lock on a subelement, and IX represents the intention to obtain an exclusive lock.

4.2 Protocol

The warning protocol is a set of rules that transactions must follow in order to maintain the consistency of the database. The original rules of the warning protocol are:

- We must begin a transaction at the root of the tree.
- If the element that we wish to lock is further down the tree, then we place a warning at a current node; that is, if we want to get a shared lock on a subelement, we request an *IS* lock at this node. If we want an exclusive lock on a subelement, we request an *IX* lock on this node. We then move to the child node.
- In order to decide whether the lock that we are requesting can be granted, we use the compatibility matrix (Table 4.1). If the lock on the current node is not granted, we must wait until some of the existing locks on the node are released and our lock request can be
- We cannot remove a lock or warning if we hold any locks or warnings on its children.

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

Table 4.1: Lock Compatibility Matrix

The meaning of the compatibility matrix is as follows. When we request an X lock on a node, and an *IS* lock has been placed on the node by some other transaction, our request will be denied, because an *IS* lock implies that the conflicting transaction may hold an S lock on some descendant of this node. If an X lock is held by a transaction, no other transaction can place any warning or lock on the node until the transaction

that holds the lock finishes. Therefore, in transactions obeying the warning protocol, an S or X lock to a node becomes equivalent to a lock to the entire subtree beginning from this node.

4.3 Extentions for Insertion and Deletion

When we insert new nodes or delete nodes, we must handle locks for non-existing items. The following example illustrates the problem that may occur when inserting or deleting a node. Assume that a transaction T comes along and tries to insert new nodes as subelements of node A, but at the same time, transaction T' deletes node A. In spite of transaction T', transaction T continues to insert nodes as subelements of the deleted node A until it has finished the insertions. In this case, there is no way to find the inserted elements, because we can no longer find the pointers to them from node A. The inserted elements become phantom nodes.

To avoid such problems, transactions must obtain an X lock on a parent node before they insert or delete some nodes. For this reason, we add two rules to the warning protocol:

- Before we insert or delete nodes, we must obtain an X lock on the parent node of any node that is to be inserted or deleted, since we must make changes to the pointer within the parent node.
- Until we place a warning on a node, we cannot trace its pointers to its children. However, if we already have a lock on the subtree containing the node, we can do so without warnings.

The second rule dictates that we cannot read the record in the index table before we place a warning on the node. When an insertion or deletion occurs, the first rule ensures that an X lock is already in place, and that no other transaction can place a warning on the node because of the X lock. Thus, we can protect the record in the index table that we are currently updating.

Chapter 5

Background

A transaction is a unit of consistency that groups the reading and writing actions to databases. It is assumed that each transaction, when executed alone, transforms a consistent state into a new consistent state; that is, we can preserve consistency as long as transactions are independently executed. For efficiency, however, database systems often execute transactions simultaneously. In such cases, we must also ensure that the outcome of processing all of the concurrent transactions is the same as that produced by running the transactions independently in some order. In this section, several database theories are described that are the basis for managing consistency in a Xerial database system.

5.1 Serializability

If transactions are executed one by one, the sequence of the transactions is called a *serial schedule*. For any initial state of the database, if the effect on the database is equivalent to that of some serial schedule, we say that the schedule is *serializable*. *Serializability* is an important concept for managing the consistency of databases. In general, it is not possible to test whether two schedules have the same effect for all initial values of database items. There is an infinite number of combinations of operations to the database and of the sets of initial values for the database. In practice, however, we make some assumptions for the transactions, such as constraints on the order of actions, and make sure that the schedule satisfies serializability. If we can say that a schedule is serializable, we can execute the transactions concurrently while preserving the consistency of the database.

When a transaction involves read-only and write-only locks, to ensure that the schedule is serializable, *conflict-serializability* [9], which is a stronger condition than

serializability, is generally used. With regard to the conflict-serializability, we have the following theorem:

Theorem 5.1. *If transactions obey the two-phase locking protocol, then any legal schedule is conflict-serializable. [9]*

Two-phase locking (2PL) is widely used in commercial locking systems. The 2PL condition states that in every transaction, all lock requests precede all unlock requests.

To make transactions serializable, we add the rule below to the warning protocol:

- Once a transaction acquires warnings or locks, it must continue holding all of them. After all of the actions to the database are finished, all of the warnings and locks are released at once.

With this additional rule, we can ensure that transactions following the warning protocol become 2PL. Then, from Theorem 5.1, we can ensure that any schedule of such transactions is conflict-serializable.

5.2 Recoverability

In database systems, we must consider not only serializability but also the way to handle failed transactions and system crashes. In order to protect the database against loss of data as the result of such failures, we commonly use the *log*. This log is a history of all the changes made to the database and the status of each transaction. Each log record contains a pair of values consisting of the old value and the new value of the updated record so that we can undo the modifications to the database when some transaction is aborted.

For recoverability, we have to consider the possibility that a *cascading rollback* [9] may occur. Consider the case when a transaction writes a new value for an item and later in its processing it aborts. Some other transaction may read the "dirty" data before the aborted transaction has recovered. In this case, the transaction that has the "dirty" data must abort too and it must undo all of its modifications, since there is a possibility that it has used the "dirty" data for writing other records. This is an example of cascading rollback. In general, it is very costly to find all of the transactions that read "dirty" data, abort them, and recover all of the effects by referring to the log. Therefore, we have to avoid such troublesome cascading rollbacks in some way.

In the previous section, we combined the warning protocol and two-phase locking. The benefit of the 2PL-locking method is that it not only ensures serializability, it also

guarantees that it will not be possible for any transaction to read "dirty" data. Thus, no cascading rollback is possible. Once a transaction is finished, it will never need to be redone. Furthermore, it can be guaranteed that it will become a permanent part of the database history.

Chapter 6

Transaction Language

6.1 Extending XQuery to Incorporate with The Warning Protocol

An XML transaction using the warning protocol consists of two stages. First, a transaction places warnings on the tree while traversing the database to find the target nodes to read or write. Then, it obtains an S or X lock on a subtree and carries out several reading or updating operations. A transaction language combined with the basic operations presented in section 3.3 is described below.

To specify the target nodes that we wish to update, we extend the XQuery [1] language for a transaction in XML. XQuery is based on pattern matching of the path-expression that binds variables to objects within the XML documents. The objects in the XQuery designate the variable values. Binding operations such as `FOR`, require at least read locks if they are used in a transaction. In order to use XQuery without these read locks, we introduce a new binding operation `SET`, whose concept of binding is a little different; it binds variables not to object values but to nodes. By using the `SET` operation, we can traverse a tree using only warnings until we really need to read or write node values.

We now describe the syntax of an XML transaction:

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

The first `SET` operation indicates a node to lock designated by the *XPath* expression and binds `$var` to the node. This is a warning stage. Query and update operations are enclosed in a `TRANSACTION` clause. In this clause, we can assume that we are authorized by an S or X lock mode to access the entire subtree beginning from the node bound by `$var`. Transaction accesses are restricted to the nodes within the subtree. The description in the transaction clauses includes XQuery and update operations. Examples are shown in the next section.

6.2 Typical Transactions

In this section, we elaborate on each operation in section 3.3, and show the notations of our transaction language that use them.

6.2.1 Search

The expression below is a typical pattern using XQuery's operations, `FOR`, `WHERE`, `RETURN`. The next example illustrates how we can access the `<warehouse id="C">` and return every customer status in `"district-A"`:

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

In this example, the transaction places *IS* warnings on root and company nodes. After it acquires an S lock on a warehouse node whose attribute satisfies the condition that `id="C"`, the transaction is able to traverse in the subtree as desired. In the `TRANSATION` clause, there are loops for each `$y` and `$z`, because there may be several candidate nodes for the *XPath* expressions, `$x/district` and `$y/customer/status`. If the name tag of `$y` is equivalent to `"district-A"`, then the corresponding node values of `$z` will be returned.

6.2.2 Insertion

As described in section 4.3, a node insertion requires an X lock on the parent node. The example below inserts into the `<district id="B-001">` a new customer David

with his identifier "D-144" and the entry date 12/02/2002.

```
SET $x = /company/warehouse[@id="A"]/district[@id="B-001"]
TRANSACTION $x {
    INSERT $x {
        <customer id="D-144">
          <name> David </name>
          <entry_date> 12/02/2002 </entry_date>
        </customer>
    }
}
```

From this expression, we create a schedule that places IX warnings on the pass from root to <district id="B-001">, locks the district by X mode, and inserts new database records made from the XML documents in the INSERT clause.

6.2.3 Deletion

A deletion operation must erase not only a subtree, but also the pointer from its parent to the node being deleted. In the DELETE statement, we have to depict both the parent and the target node. This is shown in the following example that deletes the order received on 19/02/2002 from the customer "C-031" in the district "D-002" of the warehouse "B". Observe that the variable $x is specified as the parent node of $y, and the node $y is our target node for deletion:

```
SET $x0 = /company/warehouse[@id="B"],
    $x  = $x0/district[@id="D-002"]/customer[@id="C-031"]
TRANSACTION $x {
    DELETE $y IN $x/order
    WHERE $y/date = "19/02/2002"
}
```

The deletion syntax is almost the same as the syntax of the FOR loop. We can select the target nodes by the condition following the WHERE statement, and every element satisfying the condition will be deleted.

Note that the path in the SET operation is divided into two lines. This notation provides readability for the language when the lines are too long.

6.2.4 Modification

Modification to the data in a node is described with a `WRITE` statement. The example below adds 10 to the number of the `<order id="5">` from the `customer "C-031"` in the `district "D-002"` in the `warehouse "B"`:

```
SET $x0 = /company/warehouse[@id="B"],
    $x1 = $x0/district[@id="D-002"]/customer[@id="C-031"],
    $x  = $x1/order[@id="5"]/num
TRANSACTION $x {
    WRITE $x $x+10
}
```

This is an example of the fine-grained locking scheme. If the node positions that we want to modify are already known, detailed path-expressions such as seen here, can be used. This can help to provide a significant increase in the concurrency of transactions.

6.2.5 A More Complex Example

In our transaction language, we can combine several query and update operations. In the next example, we find specific orders whose `category` is `"book"` with a `price` that is higher than 10,000, and we impose a 10we insert a `comment` to the orders:

```
SET $x0 = /company/warehouse[@id="A"],
    $x  = $x0/district[@id="B-031"]/customer[@id="C-032"]
TRANSACTION $x {
    FOR $o IN $x/order,
        $p IN $o/price,
        $a IN $x/history/amount
    WHERE $o/category = "book", $p > 10000
    INSERT $o {
     <comment> tax has been imposed </comment>
    }
    WRITE $p $p * 1.10
    WRITE $a $a+$p
}
```

Once we have acquired an X lock on the customer node, we can modify any element under the customer node as we wish. This transaction obtains a lock first, and then

looks up all of the orders in the customer node and updates those that satisfy the conditions. By using the warning protocol, we can execute arbitrary queries and update operations as long as the target nodes are within the subtree.

Chapter 7

Implementation

Xerial is implemented in C++. It consists of several components, an XML parser, a lock manager, a query compiler, a transaction scheduler, and so on. The program has about 10,000 lines of source code. Xerial is designed for use by a database server. When it receives a request from a client, it creates a thread for processing the transaction, which runs in a multi-threaded environment.

7.1 BerkeleyDB

A database system is a collection of a variety of kinds of components. It requires a disk-management system for storing and searching records efficiently, a buffer-management system that supports multiple readers and writers, a locking-management system, etc...

For our research, we used the BerkeleyDB [6] in order to achieve our fundamental but quite complicated facilities. The Berkeley DB is an open source library for constructing a database whose records are very simple structures, namely *(key, value)* pairs. BerkeleyDB provides B-tree or Hash style storage for the key and value pairs, locking, transaction commitment, logging, and recovery management systems. Most of them are available in customizable forms; for example, we can set up a lock compatibility matrix as shown in Table 4.1.

Berkeley DB is an embedded library; that is, it runs in the same address space as our programs, so no inter-process communication is required. The cost of communicating between processes is much higher than the cost of making a function call. In fact, our data model, as described in section 3.1, is able to represent by using a relational database. However, because making a function call is obviously advantageous, we consider that our Xerial system can outperform any implementation that requires

communications with existing relational database systems.

7.2 XML Converter

We implemented an `xml2db` program that converts an XML document to the form described in section 3.1. For `xml2db`, we wrote our own SAX parser, which scans the XML file and translates it into a stream of events, such as start-tag, end-tag, data value, etc. By using these events, `xml2db` creates database records and stores them into three files: an index table, a data table, and an attribute table.

7.3 Query Compiler

The query compiler parses transaction statements such as those that are described in Chapter 6, and constructs a program that is a sequence of actions to the database. Actions to the database are read records, select nodes, write, delete, insert records, etc. However, during this parsing operation there is no way to know the elements that the transaction will actually access until it begins to read the database. Therefore, this is only an abstract representation of the program. The program shows the steps to bind variables to the database nodes, and the sequential order of binding, traversing, and update operations. The order is arranged for the warning protocol. For example, an instruction to bind the variable designated by the `SET` operation appears first. When the scheduler binds this variable, it must place warnings on the traversed nodes.

7.4 Transaction Scheduler

The outline of the schedule is already prepared by the query compiler. The task of the transaction scheduler is to obtain locks and to handle the possibility that the transaction may have to abort. While processing the program created by the query compiler, the scheduler places warnings and locks on appropriate nodes. If necessary, it waits until the lock requests are granted. If the process causes some exception, such as reading or writing errors, it must abort and discard all of the changes to the database. When the main process detects such a transaction abort, it halts all transactions and runs the recovery process.

7.5 Lock Manager

To realize the lock modes in the warning protocol, we use the locking subsystem of the BerkeleyDB. Its implementation manages lock requests by using lists of lock waiters and holders. When it decides whether or not to grant a lock request, it uses a FIFO ordering. It can only grant a new lock if it does not conflict with anyone on the list of holders OR anyone on the list of waiters. A lock is not granted if there is a conflict in the list of waiters in order to avoid *starvation*. For example, if the lock manager considered only the lock holders when granting a lock request, an X lock requests to a node that is read frequently or has a warning applied often may never be granted. This situation is called *live lock*. By using both the waiters and the holders lists we can ensure that the X lock request will eventually be honored. Even if the X lock is not granted at first, it is pushed on to the waiters list and no more IS, IX, and S locks can be granted immediately. Further lock requests will also be pushed on to the waiters list. After all of the transactions that hold locks that conflict with the X lock request have finished and released the locks, the X lock will be granted by using the waiters list as a FIFO queue.

Chapter 8

Experimental Evaluation

In this section, we evaluate Xerial using the XML representation of the TPC-C data model. Our goal is to achieve a higher transaction throughput using our extended version of the warning protocol than that realized by the usual locking method that locks the entire document.

8.1 Hardware

As a test vehicle, we used a Windows 2000, Pentium III 1GHz dual-processor machine, with 2GB main memory and two hard disk drives (Ultra 160 SCSI, 10000 RPM). The disks are formatted in NTFS, and used separately for the database and the log. It should be noted that the performance of database systems generally depends on the efficiency of the disk accesses, so we did not conduct any experiment comparing single- and dual-processor machines.

8.2 Data Source

The dataset for the experiment is an XML representation of the TPC-C data model (Figure 2.1). It has 5 warehouses and each of the warehouses has 10 districts. In the same way, each district has 50 customers, and 5 orders. The total size of the XML document is 11.5 MB, which breaks down into 3433271 tags, 17555 attributes, and 293160 data records. Every `warehouse`, `district`, `customer`, and `order` has an ID in its attribute. This ID is used as a guide when the scheduler traverses the XML tree. We converted the XML into a database for Xerial by using `xml2db`.

Transactions	Lock Node	Mode	Insert	Modify	Delete	Average Execution Time (sec.)	S1 (%)	S2 (%)
Search District	Warehouse	S	0	0	0	0.0031	40	5
Insert Customer	District	X	30	1	0	0.0110	20	10
Delete Customer	District	X	0	1	30 ~	0.0953	10	2
Insert Order	Customer	X	22	1	0	0.0172	15	40
Write Payment	Customer	X	0	2	0	0.0078	10	25
Delete Order	Customer	X	0	1	22 ~	0.0203	3	3
Order Status	Order	S	0	0	0	0.0063	2	15

Table 8.1: Number of Update Records and Mixture Percentage

8.3 Transaction Sets

The transactions used in the experiment are shown in Table 8.1 (their detail statements in the transaction language are in the Appendix). We consider that they are typical examples of update operations on the TPC-C model: searches for specific elements, node insertions, deletions, modifications, and references to node values. Each transaction selects a single node to lock at random. Thus, there is no possibility of *deadlock* between the transactions. In order to show the cost of each transaction, we measured the average execution time of 10 independent runs of each transaction (Table 8.1).

We performed the experiments on two kinds of transaction sets. In the first set (**S1**), transactions that require coarse-grained locking, such as the Search District operation, comprise the majority of the transactions. The second set (**S2**) assumes that order insertions and payment writing operations are the majority of the transactions. For the warning protocol, the concurrency of **S1** is lower than that of **S2**.

8.4 Experimental Methodology

To estimate how the concurrency of transactions affects the throughput, we compared the two transaction sets in the previous section. In addition, to better illustrate the difference, we also compared two locking methods. One of the methods uses the extended version of the warning protocol (*parallel execution*), and the other obtains an X lock on the root node (*serial execution*). In parallel execution, the number of transactions that concurrently access to the database is as many as the number of created threads. Serial execution is the lowest concurrency method, since it is the same as locking the entire database.

The experiment was a simulation of the transactions on a database server. Usually, there are delays caused by network communications between clients and the server.

		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

Table 8.2: Average Response Times for Transactions (sec.) and Performance of Xerial

To avoid this effect, a transaction was produced randomly on the machine on which Xerial was running. The maximum number of threads that concurrently accessed a database was set to 50. Each of them was assigned a random transaction. If some thread had finished its task, a new transaction is passed to it immediately. Therefore, the 50 threads created in Xerial continually execute transactions. We measured the time needed to finish 100,000 transactions. We also calculated the average response time, which represents the elapsed time for processing a transaction since the thread received it. The response time is the sum of the times required for query compiling, locking, and updating.

8.5 Transaction Throughput

Figure 8.1 and Figure 8.2 show the passage of time until 100,000 transactions of **S1** and **S2** finished. The average locking times, updating times, and response times are shown in Table 3. The locking time means the wait time until a transaction obtained an S or X lock. The update time is the elapsed time from the time a transaction held a lock until all updating actions were finished.

In Table 8.2, we can observe common characteristics for both the set **S1** and **S2**. The parallel execution exceeds the serial execution in locking times. On the other hand, the serial execution shows shorter update times. In total, however, the parallel execution realizes shorter response times.

Since set **S1** is designed to decrease concurrency, the improvement in the transaction throughput by the warning protocol in **S1** (Figure 8.1) is less than that in **S2** (Figure 8.2). In spite of this, the warning protocol achieved more than double throughput on the dual-processor machine for both the **S1** and **S2** transaction sets.

From these results, we have concluded that the performance of parallel execution, which is based on the warning protocol, exceeds that of serial execution. Furthermore, in the experiment using the high concurrency transaction set (**S2**), the parallel execution took less than one-fifth the time of the serial execution to process the 100,000

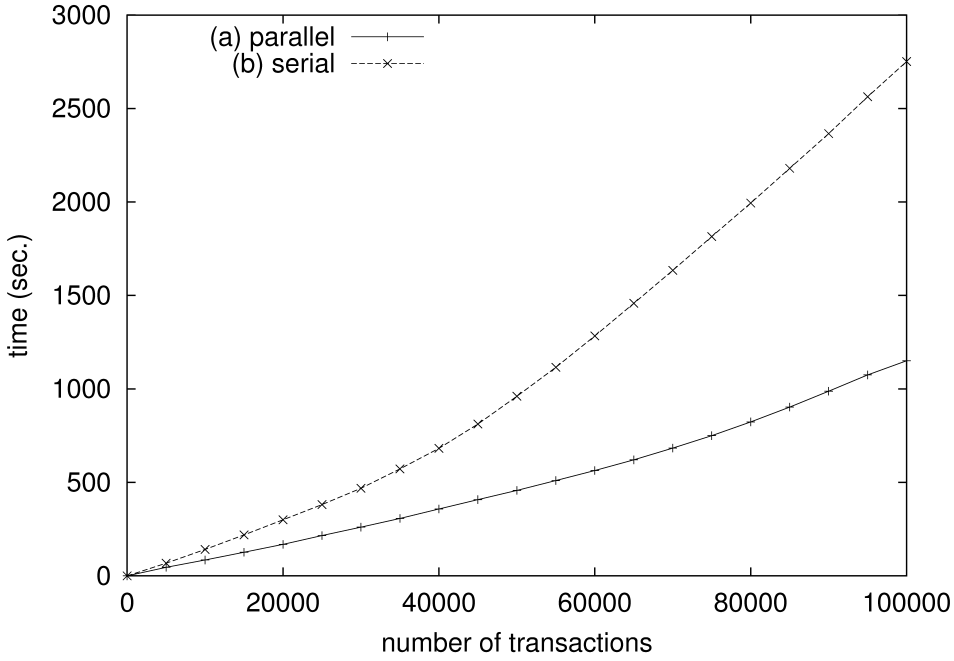

Figure 8.1: Execution Time of Transactions (**S1**)

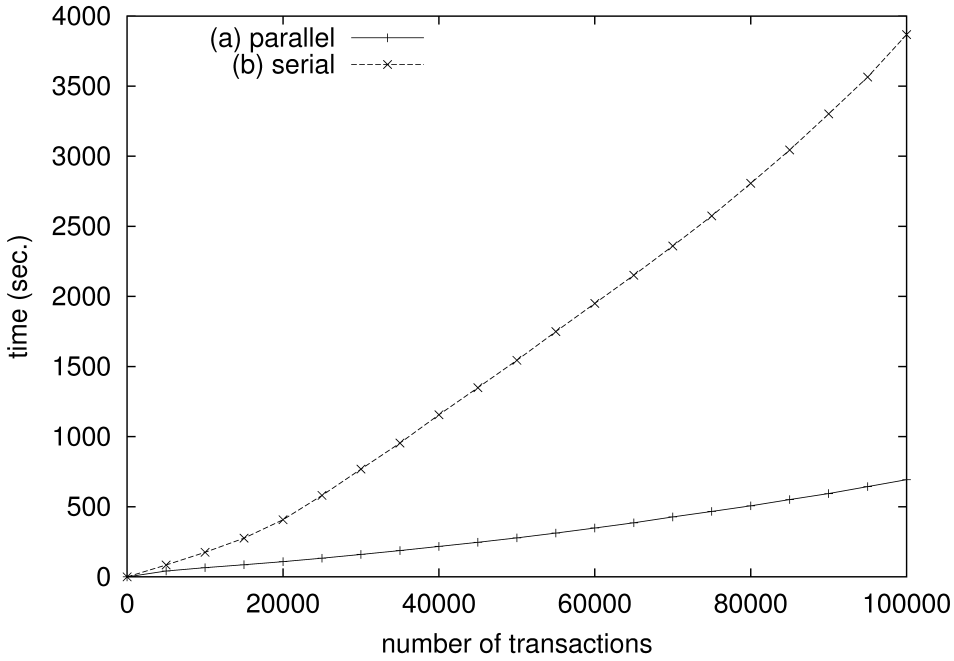

Figure 8.2: Execution Time of Transactions (**S2**)

transactions. Therefore, we have certainly achieved our goal of increasing the transaction throughput by using the warning protocol.

8.6 Tradeoff between Concurrency and Disk Access Performance

The number of active threads is the number of threads that are concurrently accessing the database. Clearly, the performance of disk accesses becomes lower in proportion to the number of active threads accessing the disk. The update times in Table 8.2 show this effect. In comparison with parallel execution, the update time of serial execution is considerably faster, since no two threads are allowed to access the database simultaneously in serial execution. There is a tradeoff between concurrency and disk access performance. Hence, we should control the maximum number of threads appropriately. If we allow threads to be created in an unlimited fashion, it may decrease the throughput, since the threads will take a longer time to update. As the number of threads increases, the lock holding time becomes longer. As a result, the rate of concurrency is decreased.

One way to decrease the effect caused by simultaneous accesses to the database is to put the tag, attribute, and data tables on different disks. On the machine used in the experiment, these three table files are stored on the same disk. By distributing them onto different disks, we can eliminate the cost of seeking the beginning of the files, which would probably lead to an improvement in disk I/O performances.

Chapter 9

Conclusion and Future Work

As part of our research, we have paid particular attention to the self-describing format of XML. We consider this characteristic useful in constructing databases that are more flexible than relational databases. We have provided a description of transaction management in XML, which is a fundamental but inherently useful facility for database systems. By extending the warning protocol, we acquired the capability to update XML databases and to handle concurrent transactions. Furthermore, by using the two-phase locking method, we provided recoverability of the database. In order to show the advantages of our proposals, we implemented a transactional database system, called Xerial. To show its efficiency and practicability, we conducted experiments using Xerial and achieved very satisfactory results.

In addition, we showed that attributes of XML, which do not have any distinct differences from tag data, can be used as guides for traversing an XML document especially in a transactional environment. This has yielded a suggested convention for XML descriptions.

In this paper, we have provided some basic operations in XML. Considering the requirement for scalability of our transaction language, we have designed it based on XQuery. Therefore, we still have enough room for more complex transactions. It should be noted, however, that when we wish to communicate data between subtrees, we have to consider the possibility of deadlock in the two-phase locking method. Consequently, we must investigate further ideas for concurrency management and deadlock prevention.

Currently, there is a great deal of research on the querying of XML documents. Most of this research has focused on the efficiency of querying wide ranges. When the targets of a query expand into the entire XML document, the warning protocol requires

a root-level lock. If we follow the conclusions of the current research, we will encounter decreased concurrency caused by the frequent occurrence of large-level locks. In such a case, it might be more effective to use another consistency management scheme such as time-stamps, or versioning of the data [9].

In our work, we have concentrated on the highest level of consistency management, serializability. However, if we assume that there are small errors in the results of transactions, or that the database may become temporarily inconsistent, it may be necessary to lower the degree of consistency in order to increase the transaction performance [4].

References

[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. Available from http://www.w3.org/TR/xquery/.

[2] J. Clark, S. DeRose. XML path language (XPath) version 1.0, W3C recommendation, 16 November 1999. http://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] H. Liefke and D. Suciu. XMill: an efficient compressor for XML data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 153-164, 2000.

[6] Sleepycat Software. BerkeleyDB. Available from http://www.sleepycat.com/.

[7] 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.

[8] Transaction Processing Performance Council. http://www.tpc.org/.

[9] J. D. Ullman. Principles of database and knowledge-base systems. Volume I. Computer Science Press, 1988.

Appendix: Sample Transactions

Note that every ID in the attributes and every string in the inserted XML documents here are created at random.

Search District:

```
SET $x = /company/warehouse[@id="4"]
TRANSACTION $x {
    FOR $y IN $x/district,
        $z IN $y/name
    WHERE $y/tax > 500
    RETURN $z
}
```

Insert Customer:

```
SET $x = /company/warehouse[@id="3"]/district[@id="5"]
TRANSACTION $x {
    INSERT $x {
       <customer id="13">
         <first>FWSqw</first>
           ...
            xml-document (28 nodes)
           ...
         <balance> 3124128 </balance>
       </customer>
    }
}
```

Delete Customer:

```
SET $x = /company/warehouse[@id="2"]/district[@id="3"]
TRANSACTION $x {
    DELETE $y IN $x/customer[@id="15"]
}
```

Insert Order:

```
SET $x0 = /company/warehouse[@id="1"]/district[@id="10"],
    $x  = $x0/customer[@id="8"]
TRANSACTION $x {
    FOR $y IN $x/history/amount
    WRITE $y $y+1
    INSERT $x {
       <order id="2">
          <entry_date>19/02/2002</entry_date>
           ...
            xml-document (20 nodes)
           ...
          <dist_info>EhqAyxWXDyXmVrThxml</dist_info>
       </order>
    }
}
```

Write Payment:

```
SET $x0 = /company/warehouse[@id="2"]/district[@id="2"],
    $x  = $x0/customer[@id="4"]
TRANSACTION $x {
    FOR $y IN $x/balance,
        $z IN $y/amount
    WRITE $y $y+$z
}
```

Delete Order:

```
SET $x0 = /company/warehouse[@id="4"]/district[@id="3"],
    $x  = $x0/customer[@id="15"]
TRANSACTION $x {
    DELETE $y IN $x/order[@id="1"]
}
```

Order Status:

```
SET $x0 = /company/warehouse[@id="5"]
    $x  = $x0/district[@id="6"]/customer[@id="6"]
TRANSACTION $x {
    FOR $y IN $x/order,
        $z IN $y/carrier_id
    WHERE $y/delivery_date/year = 2002
    RETURN $z
}
```