Lock-based Concurrency Control for XML

Namiruddin Ahmed

Master of Science

Computer Science

McGill University

Montreal, Quebec, Canada

2007-01-14

A thesis submitted to the McGill University in partial fulfillment of the requirements of the degree of Master of Science

©Copyright 2006 All rights reserved

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-32651-0
Our file Notre référence
ISBN: 978-0-494-32651-0

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

ACKNOWLEDGMENTS

I am grateful to my supervisor, Bettina Kemme, for her guidance and dedication to helping me finish this project successfully. Our weekly discussions were invaluable to me in crystallizing my research ideas; in addition she helped me quickly pinpoint any deficiencies in my work based on her extensive background in database research. I also thank her for her financial support which allowed me to concentrate fully on my studies and finish on time. I would like to thank my parents and sister for their moral support for my decision to pursue undergraduate/graduate studies far away from home. Without their endless love and support, I would not be able to make such huge steps forward in my life.

TABLE OF CONTENTS

TABLE OF CONTENTS iii
Chapter 1 Introduction 8
Chapter 2 Background 13
2.1 XML Semantics 15
2.2 XML DOM Representation 17
2.3 XPath: XML Path Language 20
2.3.1 XPath Examples 20
2.4 XQuery: XML Query Language 22
2.4.1 XQuery Example 22
2.5 XQuery Extensions for Updates 24
2.5.1 Update Operation Examples 25
2.6 Transactions 28
2.6.1 Transaction Examples 29
2.6.2 Transaction Properties 30
2.6.3 Serializability 31
2.7 Concurrency Control 34
2.7.1 Lock-based Protocols 34
2.7.1.1 Deadlocks 36
2.7.2 Optimistic Concurrency Control 36
2.7.3 Concurrency on XML 38
2.8 McXML: A Native XML Database 38
2.8.1 McXML Architecture 39
2.8.2 Storage Manager 40
2.8.3 Query Execution Engine 42
Chapter 3 Related Work 44
Introduction 44
Locking-based Concurrency Control 44
3.1 Path Locking Schemes 44
3.1.1 Path Lock Propagation (PROP) 45
3.1.2 Path Lock Satisfiability (SAT) 47
3.1.3 Suitability Discussion 48
3.2 Basic Hierarchical Locking 48
3.2.1 Implementation 48
3.2.2 Suitability Discussion 51
3.3 Flexible and Fine-Granular Concurrency Control 53

3.3.1 Direct Node Access 53
3.3.2 Navigational Access 56
3.3.3 Suitability Discussion 58
3.4 2PL Protocols 59
3.4.1 Doc2PL 60
3.4.2 Node2PL 60
3.4.3 NO2PL 61
3.4.4 OO2PL 62
3.4.5 Suitability Discussion 62
3.5 DGLOCK Protocol 63
3.5.1 DataGuides 63
3.5.2 DGLOCK Protocol Description 65
3.5.3 Suitability Discussion 68
3.6 Snapshot based Concurrency Control Protocols 69
3.6.1 Snapshots 70
3.6.1.1 Reading from a Snapshot 71
3.6.2 OptiX: Optimistic Concurrency Control for XML 73
3.6.3 SnaX: Snapshot Isolation for XML 74
Chapter 4 LockX Theory 77
4.1 LockX Pitfalls 77
4.1.1 Serializability 77
4.1.2 Avoiding Phantoms 78
4.2 Lock Types 79
4.2.1 Read Locks 80
4.2.2 Write Locks 81
4.3 LockX Expected Results 84
4.3.1 Read Queries 84
4.3.2 Update Queries 87
4.4 Compatibility Matrix 88
4.4.1 Deciding conflicts 90
4.4.2 Detailed Analysis 91
4.5 Handling Aborts 94
Chapter 5 LockX Implementation 98
5.1 High-level overview 98
5.2 LockX Components 100
5.2.1 Lock Table 100
5.2.2 Compatibility Checker 102

5.2.3 Lock Manager 102
5.2.3.1 Circumventing Scheduling Fairness 103
5.2.4 Deadlock Detector 105
5.2.4.1 Detecting cycles and transaction to abort 105
5.3 Implementing Transaction Aborts 109
5.3.1 Abort Algorithm 110
5.4 Query Execution using LockX 113
5.4.1 Operation Modes 113
5.4.2 Finding nodes with matching labels 114
5.4.2.1 Preorder traversal 114
5.4.2.2 Absolute Path Search 116
5.4.2.3 Update Queries 118
5.4.3 Matching predicates 118
5.4.4 Returning Results 121
Chapter 6 Performance Evaluation 122
6.1 Experimental Setup 122
6.2 Impact of Document Structure 124
6.3 Impact of Read Operations 131
6.4 The XMark Benchmark 133
6.4.1 Auction XML document 133
6.4.2 XMark Queries 134
6.4.3 Update Operations 137
6.4.4 Evaluation 138
6.5 LockX vs. SnaX/OptiX 142
6.5.1 Flat (Worse) 143
6.5.2 XMark Benchmark (Worse) 145
6.5.3 Deep (Significantly Worse) 146
6.6 Contribution 149
Chapter 7 Conclusion 151
LIST OF REFERENCES 154

ABSTRACT

As XML gains popularity as the standard data representation model, there is a need to store, retrieve and update XML data efficiently. McXml is a native XML database system that has been developed at McGill University and represents XML data as trees. McXML supports both read-only queries and six different kinds of update operations. To support concurrent access to documents in the McXML database, we propose a concurrency control protocol called LockX which applies locking to the nodes in the XML tree. LockX maximizes concurrency by considering the semantics of McXML's read and write operations in its design. We evaluate the performance of LockX as we vary factors such as the structure of the XML document and the proportion of read operations in transactions. We also evaluate LockX's performance on the XMark benchmark [16] after extending it with suitable update operations [13]. Finally, we compare LockX's performance with two snapshot-based concurrency control protocols (SnaX, OptiX) that provide a committed snapshot of the data for client operations.

ABRÉGÉ

Comme XML gagne en popularité en tant que modèle de représentation de données, le besoin s'impose de pouvoir stocker, extraire et mettre à jour les données XML de manière efficace. McXML est un système natif de base donnée XML, développé par l'Université McGill, et qui représente les données sous forme d'arbre. McXML permet d'effectuer aussi bien des requêtes en lecture seule que six autres types de mise à jour. Pour faire face aux accès concurrents, nous proposons un protocole de contrôle appelé LockX et qui impose un verrouillage aux nœuds de l'arbre XML. LockX permet de maximiser les accès concurrents en prenant en compte dans son architecture la sémantique des accès en lecture et écriture de McXML. Nous évaluons la performance de LockX en faisant varier des facteurs tels que la structure du document XML ainsi que la proportion des opérations de lecture dans les transactions. Nous évaluons également la performance de LockX grâce au banc d'essai XMark [16], après avoir étendu son implémentation de manière appropriée [13]. Enfin, nous comparons la performance du système LockX avec deux protocoles de contrôle basé sur les clichés, c'est à dire utilisant un cliché des données dédié aux opérations client.

Chapter 1 Introduction

XML (eXtensible Markup Language) is widely believed to be the present and future of data transmission and manipulation across heterogeneous computer systems [17]. Its growth in popularity is largely attributed to its ability to provide a standardized extensible means of including semantic information within documents describing semi-structured data [1]. Almost all computer systems use XML in one form or another. Companies like Microsoft, IBM, Sun Microsystems and Oracle have all embraced XML as a data format for exchanging information with their products. Furthermore, software to parse, transform, define, query, store and transmit XML data is readily available.

The World Wide Web Consortium (W3C) [6] is in the process of standardizing query languages for XML called XPath [18] and XQuery [19]. XQuery and XPath are query languages for retrieving parts of an XML document based on path expressions. XML storage has generally been provided by relational databases which map XML data to a relational model. Queries over XML data are then converted to SQL queries and executed. More recently, native XML databases (e.g. [5, 2, 1]) have emerged. They provide features such as their own data model, storage management and query processing using XPath [18] and XQuery

[19]. They provide the benefits of preserving the native structure of XML documents and processing XML documents in schema-free environments. One such native XML database called McXML [2] has been developed at the McGill University Distributed Information Systems Lab. Since XQuery does not provide any standards for updating XML data, McXML implements its own update extensions, based on the work done by Tatarinov et al. [3], which allows it to handle many different kinds of update operations.

As such native XML databases evolve, they need to be able to manage many, potentially large documents. A variety of applications with workloads, ranging from read-only to update-intensive, need to be supported efficiently. Specifically, applications supporting concurrent access by different clients on the same document to perform reads/updates can pose challenges. One solution would be to allow only one client access to the document at one time but this would create poor performance. There will be unnecessary blocking even when clients perform their operations on different parts of the document. Hence, we need a concurrency control mechanism which allows multiple users to work on a document at the same time without affecting each other.

Controlling concurrent access to an XML document is not trivial and many complications can arise. For example, consider an online book store which stores its inventory information in an XML document. If two users attempt to buy a book at the same time, we cannot predict what will happen. To avoid such unpredictable behaviour, there is a need for transaction management. Each client's actions on the XML document are encapsulated into transactions. A transaction is an atomic unit of read/write operations on the data. Transactions provide isolation i.e. if two clients execute their transactions concurrently on a document, each client has the impression that he/she is working alone on the document. Furthermore, the interleaved execution of the transactions is equivalent to one transaction executing serially after the other.

Most concurrency control approaches for XML we have studied are based on locking [8, 10, 11, 22]. Transactions acquire locks before accessing parts of an XML document preventing other transactions from accessing these parts concurrently. None of these locking schemes were well-suited to our needs because of various reasons discussed later. In this thesis, we therefore propose a concurrency control protocol, based on locking, called LockX which was specifically designed to meet the special needs of McXML. But we believe that LockX is suitable for other native XML databases which use a tree model to

represent XML documents. Furthermore, the semantics of McXML's update and read operations are considered to maximize concurrency.

We conduct an extensive performance analysis of our implementation of LockX into McXML considering various factors such as the structure of the document and impact of differing proportions of read operations. We also analyze the performance of LockX on a benchmark auctioning application to test it in real-world conditions. Finally, we compare the performance of LockX with two snapshot-based protocols (OptiX, SnaX [13]). These protocols avoid read locks on data by providing a committed snapshot to the client. OptiX is a variation of optimistic concurrency control adjusted to use snapshots and work on XML data. SnaX provides the isolation level *snapshot isolation* [14] that does not keep track of read operations at all.

The advantage of LockX over OptiX and SnaX is that it can easily be implemented into the query execution engine of most available native XML databases. In contrast, snapshot-based concurrency control protocols require the implementation of a complete multi-version infrastructure beforehand as detailed in section 3.6. At the same time, LockX's performance for real-life XML documents, in terms of response times, is only slightly worse than those for the

snapshot-based protocols (SnaX, OptiX) even though LockX has significantly more overhead. Our abort rates are generally lower than those for SnaX/OptiX which is an important transactional concern for users.

The remainder of this thesis is structured as follows: Chapter 2 introduces XML, XQuery (including update extensions), transactions, lock-based/optimistic concurrency control protocols and McXML. Chapter 3 discusses the various locking-based concurrency control protocols we have encountered in the literature. We also discuss OptiX and SnaX in more detail. In Chapter 4, we discuss the goals of LockX, our different lock types and their inter-compatibilities. We also look at how transaction management is handled on a high level. Chapter 5 looks at LockX in detail. We discuss the data structures and algorithms we have used and how LockX has been integrated in McXML to control concurrent access to XML documents. The performance of LockX is evaluated in Chapter 6 and finally Chapter 7 concludes the thesis.

Chapter 2 Background

XML (eXtensible Markup Language) is a predefined standard way of representing and exchanging information between heterogeneous information systems. It is similar to HTML (Hyper Text Markup Language) because both are semi-structured markup languages based on markup tags. In HTML, the tags are used to describe how the data looks and how it is presented. In XML, the markup tags actually describe the data they encapsulate. HTML was designed as a platform-independent standard to enable web browsers to display data. XML was designed to provide a straightforward way of exchanging data on the web among heterogeneous sources.

Table 1 below illustrates the comparison of the same book information using both XML and HTML. The HTML document on the left uses standardized tags to generate the look and feel of the information. For example, the "Year:" text will appear bold and in its own paragraph. Its surrounding tags only control its presentation. In contrast, the "TCP/IP Illustrated" information in the XML document on the right has title tags surrounding it. This indicates that this information relates to the title of the book.

XML allows us to model information in a natural and intuitive way. The properties [1] that make XML such a powerful information modelling tool are as follows:

- *Heterogeneity*: As opposed to relational database records which are constrained to have a fixed set of fields, records can have different data fields in XML.
- *Extensibility:* In relational databases, data types have to be defined in advance and cannot be changed. In contrast, XML allows us to add new data types at will allowing us to embrace rather than avoid change.
- *Flexibility:* XML does not restrict the size of data elements. Each element can be as long or short as necessary.

<html> <head> <title>Book Information</title> </head> <body> Year: 1994 Title: TCP/IP Illustrated Author: W. Stevens Publisher: Addison-Wesley Price: 65.95 ... </body> </html>	<bib> <book year="1994"> <title>TCP/IP Illustrated</title> <author> <last>Stevens</last> <first>W.</first> </author> <publisher>Addison-Wesley </publisher> <price>65.95</price> </book> ... </bib>

Table 1: Comparison of XML and HTML documents

2.1 XML Semantics

Figure 1 below shows an instance of an XML document called books.xml. The document contains both actual data, such as the price of a book, and metadata such as the book tag. This feature of self-description makes XML documents easily understandable even for new readers. XML documents are amalgamations of different types of *nodes* such as *elements*, *attributes* and *content* nodes. The main building blocks of XML documents are elements. Elements start using the

<> tags and end using the </> tags. For example, the element named *title* starts with <title> and ends with </title>. Elements in XML can consist of other elements, attributes and content nodes. Attributes are defined in the start element tag and usually define some identifying attributes of the element. They consist of a name/value pair with the value enclosed in quotations. For example, the book elements in Figure 1 all have an attribute called year listed in their opening tag. Elements can have other elements nested inside them to an arbitrary depth. For example, the author elements have inside them two elements named *first* and *last*. Lastly, elements consist of actual content. "TCP/IP Illustrated" is actual text data enclosed inside the title element of the first book element.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<Bib>
<book year="1994">
  <title>TCP/IP Illustrated</title>
  <author><last>Stevens</last><first>W.</first></author>
  <publisher>Addison-Wesley</publisher>
  <price>65.95</price>
</book>

.....

<book year="1998">
  <title>Data on the Web</title>
  <author><last>Abiteboul</last><first>Serge</first></author>
  <author><last>Buneman</last><first>Peter</first></author>
  <publisher>Morgan Kaufmann Publishers</publisher>
  <price>39.95</price>
</book>

<book year="1999">
  <title>The Technology and Content for Digital TV</title>
  <editor>
```

```
    <last>Gerbarg</last><first>Darcy</first>
       <affiliation>CITI</affiliation>
  </editor>
  <publisher>Kluwer Academic Publishers</publisher>
  <price>129.95</price>
</book>

<book year="2000">
  <title>Content for Digital TV</title>
  <editor>
    <last>Dirtbarg</last><first>Darcym</first>
       <affiliation>CITY</affiliation>
  </editor>
  <publisher>Kluwer Publishers</publisher>
  <price>139.95</price>
</book>
</Bib>
```

Figure 1 Books.xml

XML documents need to be *well-formed* to be usefully read and parsed by various XML tools such as XML parsers. This means that the XML document needs to obey certain syntax rules such as: (1) Each element has to have matching start and end tags. (2) If an element E_2 is nested within another element E_1, E_2 must be completely enclosed within E_1. For example, Books.xml would not be considered well-formed if the ending tag of a title element was outside a book element but the starting tag was inside it.

2.2 XML DOM Representation

Since XML documents are hierarchical in nature, modern applications typically use tree representations to manipulate them in memory. Such tree structures can

be created using the W3C [6] (World Wide Web Consortium) standard DOM (Document Object Model) API. Figure 2 below shows a tree representation of the Books.xml document similar to the DOM representation. We have omitted some of the book elements in Books.xml to have a concise representation for discussion. Element nodes are depicted as light grey rounded rectangles. Text nodes are described as dark grey squares. Attributes are represented as white normal rectangles. Note that the node numbering is not part of the DOM model and has been added for the author's convenience. We urge the reader to become familiar to this tree representation as well as the Books.xml document in Figure 1 as they will be the basis for most of the examples in later sections.

We will now introduce some notation that will be used later. From a concurrency perspective, elements, text nodes and attributes are all treated the same way. Therefore, we will use the term *nodes* to describe them all in a general way. Let T_r represent a tree with root r and let p and q be two nodes in this tree. p is the parent, *parent(q)*, of q (q is a child of p) if there is a an edge from p to q. p and q are siblings if they have a common parent. We denote as $path(p_n)$ the sequence of nodes $p_1,p_2\ldots, p_n$ such that p_1 is the root of the tree and each adjacent pair of nodes p_i,p_{i+1} are such that p_i is the parent of p_{i+1}. Node p is considered to be an ancestor of node q (and q is a descendant of p) if path(p) is a prefix of path(q).

For a node *p*, *ancestors(p)* represents the set of all nodes that are ancestors to *p*. Similarly, *descendants(p)* is the set of all nodes which are descendants to *p*. *Subtree(p)* is the subtree of T_r rooted at *p*.

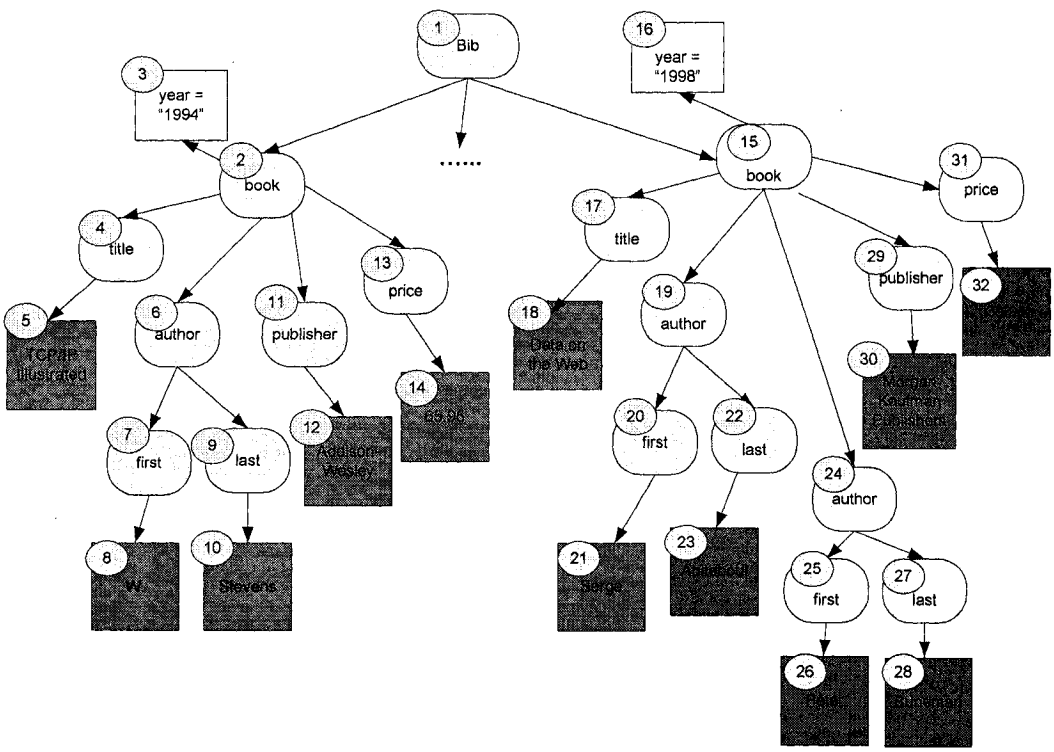

Figure 2 Books.xml tree

2.3 XPath: XML Path Language

XPath is an expression language used to select portions of an XML document. It allows information to be located by navigation using paths or arbitrarily using unique node identifiers. Predicates can also be used to narrow down the set of results. XPath can best be understood by going through a few examples which are presented below. These examples are based on Figure 2 above. For a more detailed discussion of XPath, please see work detailed by Berglund et al. [18].

2.3.1 XPath Examples

- */Bib/book*

  This expression retrieves all book elements which are direct descendants of the root element Bib. It will return as a result set *Subtree(2)* and *Subtree(15).*

- *//author*

  This expression means look for author elements which can be located anywhere in the tree hierarchy, i.e. at any level. It will return *Subtree(6), Subtree(19)* and *Subtree(24).*

- */Bib/book[author/last="Abiteboul"]*

This expression queries for all books which are direct descendants of the Bib element and have an author's last name of Abiteboul. The text enclosed in the square brackets acts as a predicate filtering out any books which do not meet this condition. The result set in this case is only *Subtree(15).*

- */*/book[@year>1994]*

  This expression illustrates the use of wildcard operators which are also allowed in XPath. It will select all book elements one level below the root level which have a year attribute greater than 1994. Note that the @ symbol represents an attribute and that this expression will accept all root elements not just Bib. In our case, the result will be the subtree rooted at the book element with year 1998 (*Subtree(15)*).

- */Bib/book[first()]*

  XPath also allows you to identify nodes based on position. The above expression selects the leftmost book under *Bib*. Therefore, *Subtree(2)* will be returned to the user.

2.4 XQuery: XML Query Language

XQuery is an XML query language which uses XPath expressions to query XML data from heterogeneous sources. XQuery is considered for XML what SQL (Structured Query Language) is for relational databases. XQuery is written as FLWOR (pronounced "flower") expressions. These FLWOR (FOR, LET, WHERE, ORDER BY, RETURN) statements are the building blocks of XQuery. XQuery is also best explained through an example.

2.4.1 XQuery Example

```
FOR $b in document("books.xml")/Bib//book
LET $t := $b/title
WHERE $b/price < 139.95
ORDER BY $b/price
RETURN $t
```

- **FOR:** The FOR clause selects all book elements which are located at any level under the *Bib* element, one by one in a loop, into a variable called $b. When this assignment has been made successfully, we say that $b has been *bound* and refer to its target nodes as its current *binding*.

- **LET**: The LET clause selects all title elements which are direct descendants of the binding of $b and binds them to $t. This assignment is made in the form of a set.
- **WHERE**: The WHERE clause selects only those book elements which have a price less than 139.95. Therefore, there is a filtering-out of all those nodes in $b which don't match this predicate condition.
- **ORDER BY**: The ORDER BY clause sorts the book elements so that their price elements are in ascending order.
- **RETURN**: The RETURN clause returns the title elements of the sorted books.

The result set is shown in Figure 3 below. Note that these title elements are from book elements which satisfy the predicate (price < 139.95) and have been sorted in ascending order according to the price.

```
<title>Data on the Web</title>
<title>TCP/IP Illustrated</title>
<title>The Technology and Content for Digital
 TV</title>
```

Figure 3 Result of XQuery Operation

2.5 XQuery Extensions for Updates

Presently, there is no formalized standard for updating XML data. In our implementation, we follow the update extensions to XQuery FLWOR expressions proposed in the influential paper written by Tatarinov et al. [3]. FLWOR expressions are replaced with FLWU expressions. The ORDER BY and RETURN clauses are both removed because update operations do not return any XML data. Instead, we introduce an UPDATE clause which accounts for the U in FLWU. The UPDATE clause allows for more specialized nested sub-operations once the target set of nodes have been identified by the FLW clauses. We will now illustrate the possible update types. Let *p* be a target node determined by the FLW expression.

- *delete(p)*: This operation deletes *Subtree(p)*.
- *replace(p, Subtree(q))*: *p* and *q* must be of the same type. If they are elements, *Subtree(p)* is deleted and replaced with the new *Subtree(q)*. Similarly if *p* and *q* are attributes, the operation simply replaces the name/value of the attribute with *q*'s name/value pair. If they are text nodes, the value of the text is changed to *q*'s value.
- *rename(p, newname)*: This operation only works on element and attribute nodes. It changes the name of the element/attribute p to *newname*.

- *Insert-into(p, Subtree(q))*: This operation inserts *Subtree(q)* as a child of *p*. Our implementation inserts *Subtree(q)* as the rightmost child of *p*.
- *Insert-after(p, Subtree(q))*: This operation inserts *Subtree(q)* as a new right sibling of *p*. *p* must be an element and *q* must be element or text. It is important to note that *q* does not have to be directly after *p*.
- *Insert-before(p, Subtree(q))*: This operation inserts *Subtree(q)* as a new left sibling of *p*. It is important to note that *q* does not have to be directly before *p*.

2.5.1 Update Operation Examples

1. FOR $b in document("books.xml")/Bib//book, $p in $b/publisher
   LET $t := $b/title
   WHERE $b/@year = 1998
   UPDATE $b {
      INSERT <award>Pulitzer Prize</award>
      RENAME $t to name
      REPLACE $p with <editor>Rapunzel Editors</editor>
   }

The statement above is an example of an update operation using the FLWU structure. It performs Insert-Into, Replace and Rename update operations within the subtree of the book element with the year attribute of 1998 as shown in Figure 4 below. The nodes with white numbering boxes have either been modified or inserted into the book subtree. *Subtree(award)* has been inserted as the child of node 15. Node 17 has been renamed from title to name. Finally, *Subtree(editor)* has replaced the publisher subtree previously at that location.

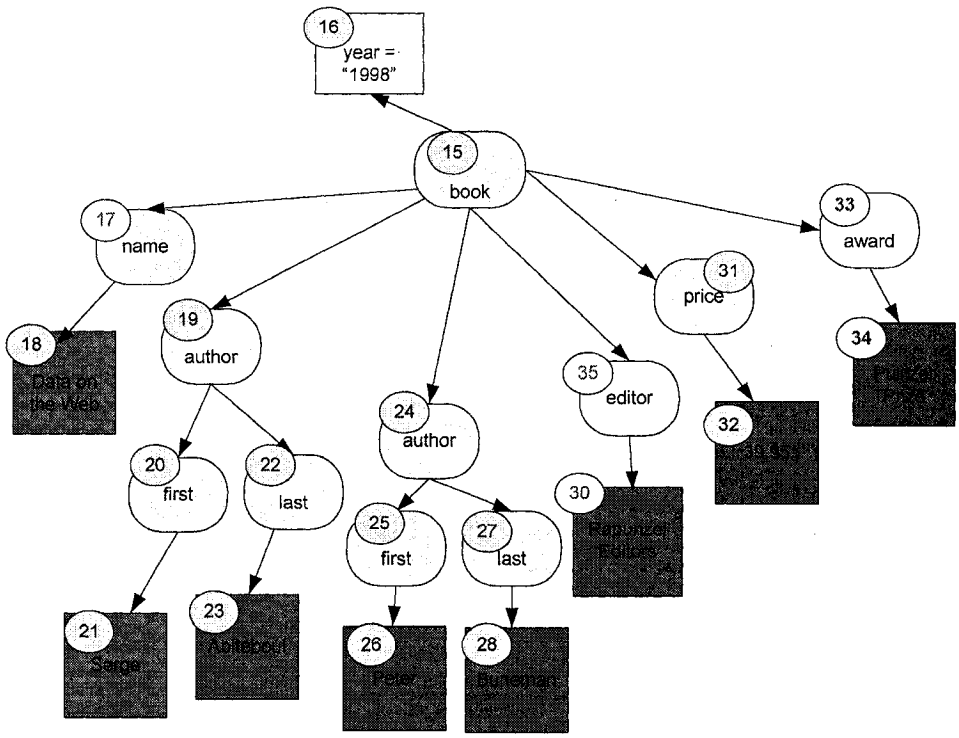

Figure 4 Update Operation 1's changes

2. FOR $b in document("books.xml")/Bib//book, $p in $b/publisher

LET $t := $b/title

WHERE $b/@year = 1994

UPDATE $b {

INSERT <publisher>O'Reilly Publishers</publisher> AFTER $p

INSERT <publisher>Morgan Publishers</publisher> BEFORE $p

DELETE $t

}

This update operation performs *Insert-after*, *Insert-before* and *delete* operations on some child elements of the book element with year 1994. Figure 5 shows the updated book subtree with new nodes having white numbering boxes. Two new publisher subtrees, *Subtree(15)* and *Subtree(17)*, have been added on either side of *Subtree(11)*. In addition, the title subtree visible in Figure 2 has been deleted.

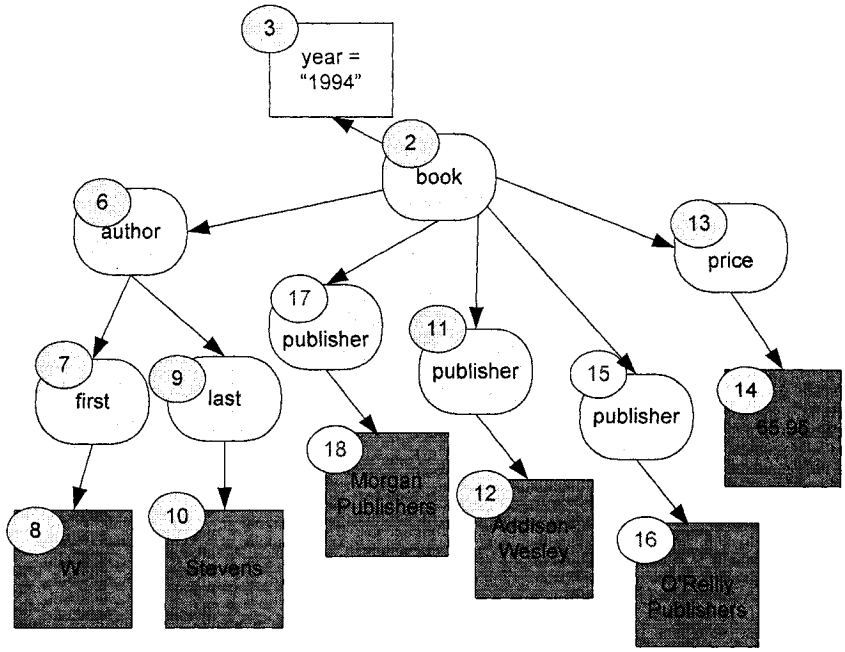

Figure 5 Update Operation 2's changes

2.6 Transactions

In the context of databases, transactions are viewed as a sequence of read/write operations which satisfy a request and ensure database integrity. They are delimited by *begin-transaction* and *end-transaction* statements. The end of a transaction has either a *commit* or *abort* operation. The *commit* operation indicates that the transaction has executed successfully and all changes are flushed to disk. On the other hand, an *abort* operation indicates something has gone wrong in the computer program and all changes are rolled back as if the transaction had never occurred.

2.6.1 Transaction Examples

Suppose that a person John with an account *A* works in a company Enron with account *B* which is used to pay company employees monthly. This is a typical use case which is ideally suited to transactions. We see below in Figure 6 what such a transaction would look like.

```
begin transaction
Read account balance of account B
Read account balance of account A
Update B to B-4000 in the database
Update A to A+4000 in the database
Commit
```

Figure 6 Account Transfer Transaction

In this situation, we want all the operations to be executed successfully or none of them. A scenario should never occur where 4000 dollars are removed from Enron's account but are not deposited in John's account.

With respect to our implementation, a transaction would be a sequence of XQuery read/write operations as we have defined earlier. For example, a typical transaction consisting of four operations could look like this:

- FOR $b in document("Bib.xml")/bib/book

  WHERE $b/title="The Techno"

RETURN $b

- FOR $b in document("Bib.xml")//book

  WHERE $b/price < 100

  RETURN $b/publisher

- FOR $b in document("Bib.xml")//book

  LET $a := $b/author

  WHERE $b/price < 100 AND $b/@year=1994

  UPDATE $b { RENAME $a TO writer }

- FOR $b in document("Bib.xml")//book[@year>2004]

  LET $t := $b/title

  UPDATE $b { INSERT <size>100 pages </size> AFTER $t }

- commit

2.6.2 Transaction Properties

In order to maintain data integrity, transactions are required to have the following ACID properties where each of these letters stands for the first letter of a property.

- *Atomicity*: Either all the operations of a transaction should succeed or the transaction has no update effect on the database.

- *Consistency*: Assuming that the database had a consistent state before the transaction started, the transaction should leave the database in a consistent state when it ends. For example, the accounts *A* and *B* from the example should have positive balances when the transaction ends. However *A* and *B* may be allowed to have an inconsistent state, such as a negative value, during the execution of the transaction.
- *Isolation*: Concurrently running transactions should get the impression that they are operating alone on the database.
- *Durability*: If a transaction manages to commit, its changes should be made persistent. Therefore, a commit is a guarantee that the transaction's changes will not be lost in any failure case. Typically, all changes of a transaction are flushed to disk for this purpose.

2.6.3 Serializability

We continue our example from section 2.6.1 regarding John's bank account *A*. Suppose that *A* is a joint account which can be accessed by John's wife Sheila. Now assume that, by some coincidence, Enron and Sheila are performing their transactions on *A* concurrently. Enron is depositing 4000 dollars in transaction T_1 while Sheila is trying to withdraw 100 dollars in transaction T_2. Also assume *A*

has an account balance of 1000 dollars before these transactions start. Table 2 shows the interleaving of the operations of these transactions in a specific order.

T_1	T_2
	Read(A)
Read(A)	
A = A + 4000	
	A = A -100
Commit	
	Commit

Table 2 Lost Updates

Both Sheila and Enron read A's value of 1000. Enron makes the commit of its new balance of 5000 to disk. Sheila then performs her update changing the balance to 900 and then commits. However, this new balance in the database does not reflect that Enron has successfully deposited 4000 dollars into the account. Sheila has effectively overwritten Enron's update. This problem is commonly known in database literature as the *Lost Update Problem* [7].

To prevent problems such as above, concurrency control protocols are built which guarantee serializability, i.e. the execution of transactions is equivalent to

some serial execution. This is done by detecting conflicts between operations from different transactions if they access the same data item and at least one is a write operation. For these conflicting operations, the execution order is important. For example, if transactions T_1 and T_2 both write the same data item x their execution order determines what the final value of x is. Now assume that T_1 reads data item x and T_2 writes it. T_1 will read a different value of x depending on whether it reads x before or after T_2 writes it. However, two read operations on x can be executed in any order or concurrently without affecting each other. We then only accept interleaving concurrent executions of transactions which have these conflicting operations in one order.

An execution E is *serializable* if there is a serial execution E_1 (one where entire transactions are executed serially one after the other without any interleaving) such that for any two conflicting operations o_1 and o_2 where o_1 executes before o_2 in E, o_1 also executes before o_2 in E_1. This approach of detecting conflicts would solve the problem above because T_2 would detect a conflict between A=A-100 and A=A+4000 (they are both write operations on the same data item). T_2 would therefore be blocked until T_1 commits. In this scenario, the execution is equivalent to a serial execution where T_1 executes before T_2 and both transactions have their changes recorded in the database.

2.7 Concurrency Control

This section introduces the basic concepts of lock-based protocols and optimistic protocols used for concurrency control. For a more detailed discussion of existing protocols, please see Chapter 3.

2.7.1 Lock-based Protocols

The primary means of achieving serializability is locking protocols. Before any operation on a data item, a suitable lock has to be obtained on it. Since there are two kinds of operations, reads and updates, we assign them *S* (shared) and *X* (exclusive) locks respectively. Based on the reasoning of section 2.6.3, we derive the compatibility matrix shown below in Table 3. The + represents compatibility and - represents incompatibility. Assume that the row is the lock already held by a transaction and the column is a lock wanted on the same data item by a different transaction. If multiple transactions want to do a read on a data item, it is permissible and they will all be granted *S* locks. If, for example, an *S* lock has been granted to a transaction T_1 on a data item, transaction T_2 wanting to acquire an *X* lock will have to wait until this *S* lock is released. There are no conflicts between locks held by the same transaction.

	S	X
S	+	-
X	-	-

Table 3 Compatibility Matrix for read/write locks

In order to guarantee serializability, transactions use a two-phase locking (2PL) mechanism. In 2PL, a transaction has a growing phase where it acquires locks and a shrinking phase where it releases locks. After releasing a lock, a transaction is not allowed to acquire any more locks. Databases normally implement strict two-phase locking where all locks are released at the end of transactions. This is necessary to avoid cascading aborts. For example, assume T_1 releases a lock on x before committing and T_2 now acquires this lock and reads x. If T_1 aborts, T_2 will also have to abort because it read a wrong value of x. Any transactions which read T_2's changes will also abort (and so on) causing a long chain of rollbacks.

2.7.1.1 Deadlocks

Since our concurrency control protocol (LockX) is based on locking, we need to introduce the concept of *deadlocks*. A deadlock refers to a specific condition when two or more transactions are each waiting for another to release a lock. For example, suppose that transaction T_1 acquires an X lock on object A and transaction T_2 acquires an X lock on object B. Now suppose that T_1 wants to acquire an X lock on object B and T_2 wants to acquire an X lock on object A. They are both deadlocked on each other with no further progress. Such deadlocks are usually detected by building a wait-for graph and detecting cycles within it. The wait-for graph for the above situation is shown in Figure 7 below. Each arrow represents a *wait-for* condition i.e. T_1 is waiting for T_2 and T_2 is waiting for T_1. One possible solution is to abort one of these transactions leading to the breaking of this cycle and hence the deadlock.

Figure 7 Wait-for Graph

2.7.2 Optimistic Concurrency Control

Optimistic Concurrency Control (OCC) is a concurrency control technique which doesn't use locking. It is based on the premise that transactions mostly don't

conflict with each other. This allows the protocol to be as permissive as possible in allowing transactions to execute.

OCC is composed of three phases:

1. WORKING: On the first operation on a data item, the transaction retrieves a committed copy from the database and caches it. This is called the *working copy* and henceforth it will be used for all operations on this data item. Read and Write sets are maintained for the data items read and written.
2. VALIDATION: This phase is used to check whether the transaction conflicts with other transactions. In backwards validation, the transaction checks for conflict with all concurrently executing transactions that have already committed. Two transactions T_i and T_j are considered concurrent if T_i started its working phase before T_j committed or vice-versa. In forward validation, the transaction checks for conflicts with active transactions that have not entered the validation phase. If there is a conflict, the transaction aborts. Otherwise, it enters the UPDATE phase. Transactions are not allowed to be in the VALIDATION phase concurrently. The validation order is used to determine the serialization order of transactions.

3. UPDATE: On a successful validation, (i.e., no conflicts) the transaction's changes are flushed to disk and committed.

2.7.3 Concurrency on XML

Concurrency control on XML data is non-trivial for a variety of reasons. Firstly, XML consists of both metadata (element and attribute names) and actual data (text, attribute values). Furthermore, this information can be nested in a complex fashion expressed by XQuery and XPath using regular expressions. Nodes in an XML DOM tree are directly dependent on ancestors. Deleting an ancestor will delete all the descendent nodes in its subtree. Therefore, whole paths of nodes have to be preserved because of dependencies between them. It is also not always clear what the target entities of a query are and therefore what should be safeguarded from other transactions. Finally, many different kinds of read/update operations exist with different conflict behaviours. We will discuss concurrency control on XML in more detail in the next chapter.

2.8 McXML: A Native XML Database

McXML is a native XML DBMS (Database Management System) developed by the DISL (Distributed Information Systems Lab) group at the School of Computer Science, McGill University. The initial system was developed by Jiafeng Wu [2]

and supported a subset of the XQuery language and all XQuery update statements, including nested updates, described by Tatarinov et al. [3].

2.8.1 McXML Architecture

Figure 8 below shows a high-level architecture of McXML's core components. We will give a step by step walkthrough of the system execution for a user update query.

1. Clients connect via RMI to a Middleware server which forwards the requests to the Query Execution Engine.
2. The Query Execution Engine requests the Storage Manager for a DOM model for the XML document in question.
3. The Storage Manager retrieves the physical model from disk and converts it to the DOM model. It returns the DOM model to the Query Execution Engine.
4. The Query Execution Engine computes the query on DOM tree making any necessary changes on the tree.
5. The Query Execution Engine returns results to the RMI Middleware.
6. The RMI Middleware forwards results to the client

7. The Storage Manager writes changes back to disk whenever a transaction is committed.

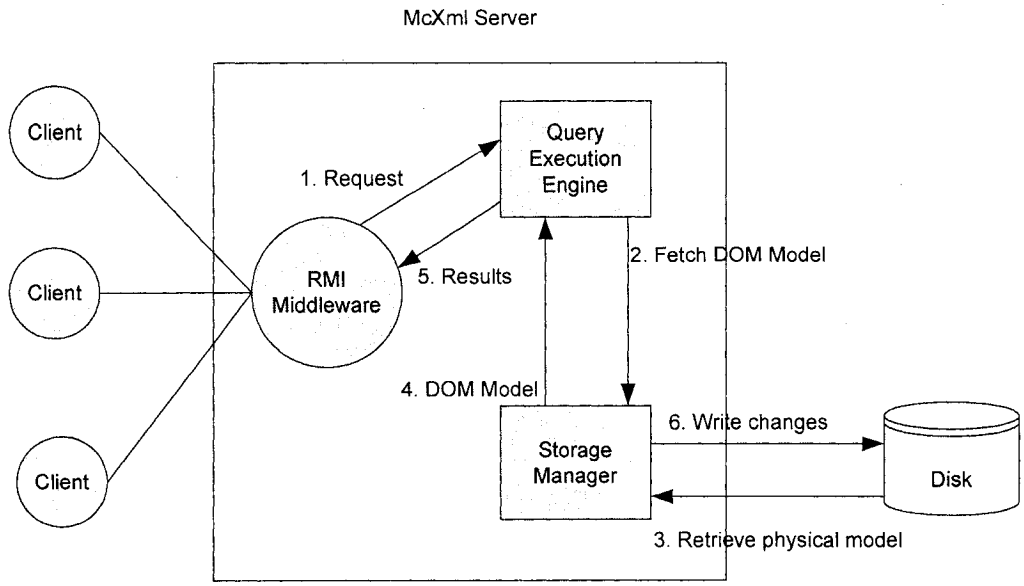

Figure 8 McXML Architecture

2.8.2 Storage Manager

The Storage Manager component was developed by Raj Suchak [4] and Jean-Sebastien Légaré. Its job is to store the XML document using a special physical model similar to Natix [5]. It is based on the concept of splitting XML documents into subtrees which are stored in records on fixed-width pages. These pages are the basic unit for transfer between disk and main-memory. The Storage Manager is responsible for converting this page-based model to a logical DOM model for

query execution purposes. However, this logical DOM model is enhanced so as to maintain information about the record/page subdivisions in the tree. This is necessary for disk write-back purposes. Figure 9 shows how the McXml logical DOM model actually looks like in memory. The boxes represent different pages on disk. As the subtrees on a page grow, the fixed-size page can no longer support the whole subtree. Therefore subtrees are split at suitable locations and placed on multiple pages. Links (shown as dotted lines) are still maintained among these pages so queries can seamlessly traverse the whole tree.

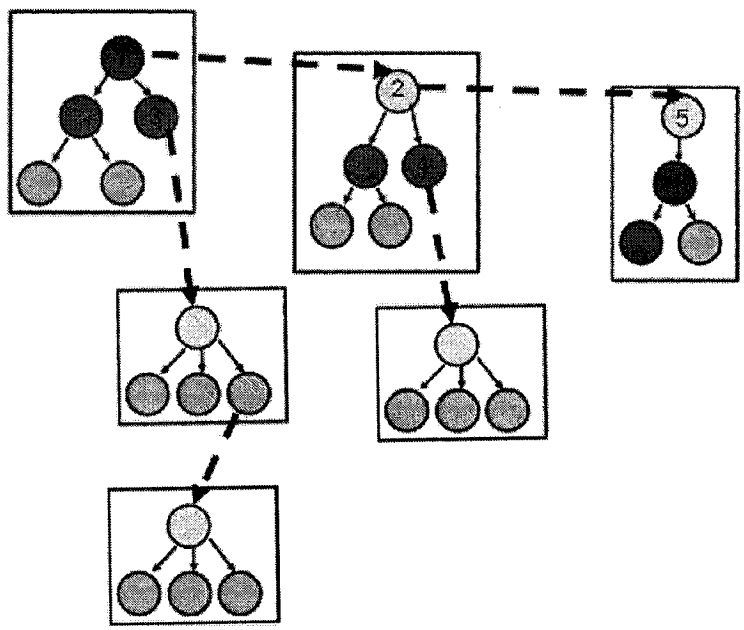

Figure 9 McXml DOM Model

If a part of an XML document changes, the records affected by the change are updated and the pages containing those records are written to disk again. Loading is done lazily, i.e., a subtree is only extracted, by loading its corresponding records, when a transaction needs to traverse that part of the tree. During transaction execution, all the changes are made to the DOM tree and written back to disk at commit time.

2.8.3 Query Execution Engine

The Query Execution Engine uses the implementation of the XQuery language (including update extensions) to execute queries on the DOM tree received from the Storage Manager. It is oblivious to the underlying physical model and just uses DOM API methods to perform its function. Consider the query below as an example for the following discussion on how queries are parsed and executed:

- FOR $bi in document("Bib.xml")/bib, $b in $bi/book
  WHERE $b/publisher="Addison-Wesley"
  RETURN $b

For each *variable* in a query, the Query Execution Engine creates a *variable* object in memory with three components: parent, value and condition. The parent component stores the variable's parent. The value component stores the XPath associated with the variable. Finally, the condition components stores any

assigned predicates. For example, for the above query, *$b* would be stored with parent *$bi*, path */book* and condition */publisher="Addison-Wesley"*. Queries are executed in multiple stages. For *$b* above, (1) the parent variable *$bi* will be evaluated and its bindings will be determined (2) the path */book* will be evaluated relative to *$bi*'s bindings. The matching nodes (*book* elements which are direct descendants of *bib*) are bound to *$b* (3) the predicate condition will be applied on *$b*'s bindings effectively filtering out those *book* nodes which don't have *Addison-Wesley* as *publisher*. (4)The bindings of *$b* are returned to the user. A similar execution logic is applied for update queries except changes are made to the DOM tree once target nodes, i.e., those assigned to relevant variables, have been identified.

Chapter 3 Related Work

Introduction

In this chapter, we look at various concurrency control protocols for XML documents presented in the literature.

Locking-based Concurrency Control

3.1 Path Locking Schemes

Dekeyser et al. [8] propose two locking schemes, Path Lock Satisfiability (SAT) and Path Lock Propagation (PROP), based on path locks. PROP acquires a multitude of read locks but has a trivial conflict checking mechanism. SAT sets very few locks but requires more work when checking for conflicts.

Dekeyser et al. [8] use a data model of XML called *XP-Tree* which is similar to the DOM tree introduced in Chapter 2. In addition they define three operations on which the locking schemes operate. Two of them are update operations and are similar to our *Insert-into* and *Delete* operations. The read operation does not support predicates and focuses only on path searches. *Q(n, p)* is defined as a query starting from context node *n* with path expression *p*.

3.1.1 Path Lock Propagation (PROP)

Shared locks are defined as tuples *(T, n, p)* which identify the owning transaction *T* of the lock, the locked node *n*, and an XPath expression *p* relative to that node. Informally, the shared lock therefore means that *T* has issued a query *p* starting from *n*. This initial lock is then used to derive other locks by a process called *read-lock propagation*. The process of read-lock propagation causes the shared locks on a node to be propagated to the nodes just below this node in the DOM tree which match *p*.

Example:

Consider the DOM tree shown in Figure 2. Suppose that a transaction *T* issues a search for all nodes that satisfy the path expression Bi*b/book/author/first*. The first shared lock will be *(T, 1, book/author/first)*. This means that *T* has acquired a shared lock on the node with identifier 1 and the path to search for is now *book/author/first*. Now, PROP derives two more shared locks on nodes 2 and node 15. The tuples are of the form *(T, 2, author/first)* and *(T, 15, author/first)* respectively. The next three shared locks derived are *(T, 6, first)*, *(T, 19, first)* and *(T, 24, first)* on nodes 6, 19 and 24 respectively. Finally we assign shared locks to nodes 7, 20 and 25 with a path expression * meaning the node itself is being read.

Exclusive locks have slightly different semantics and are expressed as tuples *(T, n, f). T* is the owner of the exclusive lock, *n* is the node being locked and *f* is either the label of the descendent element on which the modification takes place or * if the node *n* itself is being modified. The *insert-into(p, Subtree(q))* operation requires an exclusive lock *(T, p, a)* where *a* is the label of the node *q* and *p* is parent of *q*. The *delete(p)* operation requires exclusive locks *(T, p, *)* and *(T, q, a)* where *q* is the parent of *p* and *a* is the label of *p* (It is assumed that *read-lock propagation* is used to find *p* before the exclusive locks are applied) .

A shared lock such as (T_1, n, a) or $(T_1, n, *)$ conflicts with an exclusive lock (T_2, n, a) and $(T_2, n, *)$ if $T_1 \neq T_2$ and *a* is a single element. For example in Figure 2, if transaction T_1 is deleting node 6 (author), there will be an exclusive lock $(T_1, 6, *)$ on 6. Another transaction T_2 wanting to read node 7 would have to acquire a conflicting shared lock (T_2, 6, first) on 6 and will be blocked.

The complexity of PROP is as follows. Consider the shared lock (T_1, n, a_1) and the exclusive lock (T_2, n, a_2). Only the equality of a_1 and a_2 need to be checked. Thus, the time complexity of checking for conflicts is O(1).

3.1.2 Path Lock Satisfiability (SAT)

Dekeyser et al. [8] define an alternative locking scheme which requires fewer locks but is more complex with reference to testing for conflicts. Shared and exclusive locks in this scheme are defined exactly the same as in the previous scheme. In the case of read locks, it is sufficient to obtain the initial shared lock for a query operation. Thus no lock propagation is involved. The update operations require the same exclusive locks as defined earlier.

A shared lock *(T_1, n, p)* conflicts with an exclusive lock *(T_2, m, f)* if $T_1 \neq T_2$, *n* is an ancestor of *m* and the path of nodes from *n* to *m* followed by *f* can be expressed by the path expression *p*. For example in Figure 2, a shared lock *(T_1, 2, author/first)* is not compatible with an exclusive lock *(T_2, 6, first)* because the path of nodes from 2 to 6 followed by *first* matches exactly the path expression *author/first*.

SAT's space complexity is not an issue because it requires less locks than PROP. However, the conflict checking mechanism is more complex because we need to check the satisfiability of a path by a more general XPath expression.

3.1.3 Suitability Discussion

Dekeyser et al. [8] do not provide any implementation or evaluation of PROP and SAT path locking schemes. Furthermore, the terminology introduced is confusing and the protocols lack adequate detail on a theoretical level. We also note that any adequate locking protocol for McXML would have to handle predicates in the path locking mechanism. For all the above reasons, we feel that PROP and SAT are unsuitable for our implementation.

3.2 Basic Hierarchical Locking

3.2.1 Implementation

Most concurrency control protocols for XML data use a form of locking built on top of hierarchical locking schemes used in relational databases [9]. In Hierarchical Locking, there is a shared (S) lock and an exclusive (X) lock similar to 2.7.1. It also introduces two additional locks which are based on the concept of *intention locking.* Intention locks indicate the intent to perform an operation somewhere below the node being locked. There are two types of intention locks in Hierarchical Locking: *IS* and *IX.* If a query wants to return *Subtree(N)*, it would need to place IS locks on *ancestors(N)* and an S lock on N itself. Similarly, for a node N that needs to be updated, IX locks need to be placed on *ancestors(N)* as well as an X lock on N. With respect to PROP from section 3.1, all ancestor

read/write locks are replaced by IS/IX locks depending on the operation. Hierarchical Locking detects conflicts differently from both PROP and SAT. Whereas PROP and SAT use path checks, hierarchical locking compares only the lock types of transactions using a conflict matrix.

	S	X	IS	IX
S	+	-	+	-
X	-	-	-	-
IS	+	-	+	+
IX	-	-	+	+

Table 4: Compatibility Matrix for Hierarchical Locking

Table 4 above shows the compatibility matrix for Hierarchical Locking. The + sign indicates compatibility and the – sign indicates incompatibility. The row indicates the lock held by a transaction on the object and the column indicates the lock wanted by another transaction on the same object. We can see that the X lock conflicts with all lock types including itself. This makes intuitive sense from the perspective of the XML DOM tree in Figure 2. If transaction T_1 is updating the

author subtree, *Subtree(6)*, it would place IX locks on the *book* and *bib* nodes as well as an X lock on the *author* node. Another transaction should not be able to simultaneously read or update any node in *Subtree(author)* and therefore all S, X, IX and IS lock requests on the *author* node from other transactions are blocked until T_1 ends by committing or aborting. S conflicts with only X and IX because multiple readers are allowed anywhere in the concerned subtree. Intention locks, such as IS and IX, are compatible with each other because the assumption is that the actual nodes being read or written further down in the tree will be distinct. If this is not the case and two transactions want to do conflicting operations on the same node, one of them will get blocked at that node. IX conflicts with S and X because there will be read-write and write-write conflicts on the same set of nodes.

Intention locks allow transactions to protect the trees enclosing the nodes they are currently operating on. For example if Transaction T_1 is updating the *author* subtree, *Subtree(6)*, in Figure 2, its IX lock on both bib and book protects T_1 from the deletion/replacement of *Subtree(book)* and *Subtree(bib)*. They also allow us to capture conflicts earlier up the hierarchy and more efficiently. For example if a transaction T_2 was running concurrently to T_1 and wanted to read the *book* subtree, *Subtree(2)* , the conflict would be detected at the *book* node itself

because T_2's request for an S lock would conflict with T_1's IX lock on *book*. Without intention locking, T_2 would have to search the entire book subtree because a simultaneous update occurring anywhere in there would create a conflict.

3.2.2 Suitability Discussion

Although this locking protocol is simple and applies well to our problem, it also creates high blocking rates and artificial conflicts as discussed in the numerous examples below based on Figure 2. For the following discussion, we will break the generic update operation down into more specialized operations based on the update extension to XQuery described in Chapter 2. We will be using *rename(p, newname), insert-into(p, Subtree(q))* and standard XPath expressions to illustrate queries. We will assume that Transaction T_1 and T_2 are concurrently running, T_1 has acquired all the locks it needs but T_2 is in the process of acquiring them.

1. Assume that T_1 is renaming the book node with ID 2. T_2 wants to read all *title* nodes using the path search *//title*. When T_2 wants to acquire a read lock (*IS/S*) on the book node with ID 2, it will be blocked by T_1 because T_1 has an X lock on this node. However the result of the *title* search will be

the same whether the book node has been renamed before it or not. Therefore this blocking is unnecessary and an ideal locking protocol would allow these two operations concurrently.

2. Assume T_1 is inserting a new *publisher* element to the *book* with ID 2. T_2 wants to insert a *pages* element to the same *book* but it will be blocked by the X lock on the *book* node by T_1. Since the serial order of the insertions does not affect the end result (we don't care about the order of insertion), this is an artificial conflict created by this locking protocol.

3. Assume T_1 is now inserting *Subtree(editor)* into the *book* with ID 15. Therefore it has acquired an X lock on this *book* node. T_2 wants to do a search for the last names of all authors using the path search *//author//last*. These two transactions run on different parts of the tree so they should not conflict. However, while T_2 is doing a traversal of the tree to find *author* nodes, it would try to place a read lock on T_1's *book* node. T_2 will be blocked by the X lock on the *book* node held by T_1. However, this blocking is unnecessary because the end result will be the same whether T_2 is blocked at this point or not.

3.3 Flexible and Fine-Granular Concurrency Control

3.3.1 Direct Node Access

Haustein et al [10] extends the hierarchical locking protocol described above and introduces seven different locks (NR, LR, SR, IX, CX, X, U) of varying granularity described below.

- NR: An NR (Node Read) lock mode is requested for reading the context (currently accessed) node.
- LR: A LR (Level Read) lock mode locks the context node together with its direct-child nodes for shared access. This lock would save locks for XPath expressions such as *Bib/book* on the DOM tree in Figure 2. We would just need to acquire one lock for this expression rather than having to lock *Bib* and then all its direct *book* descendants.
- SR: A SR (Subtree Read) lock is requested for the context node *c* as the root of subtree *s* to perform read operations on all nodes belonging to *s*. This is the same as the S lock introduced in the previous section.
- IX: An intention exclusive lock in this case is slightly different from our previous definition. In this protocol, it indicates the intent to perform write operations somewhere in the subtree but not on a direct child node of the node being locked.

- CX: A Child Exclusive lock on context node *c* indicates the existence of an X lock on some direct child-node and prohibits inconsistent locking states by preventing LR and SR lock modes. This lock is necessary because an IX lock on the parent of the node being updated would allow a level read (LR) lock even though this is an incompatible operation.
- X: To modify the context node *c* (updating its contents or deleting c and its entire subtree) an X lock mode is needed. It implies a CX lock for its parent node and IX locks on all other ancestors.
- U: A U lock mode (update option) allows a read operation on context node *c* with the option to upgrade the mode for subsequent write access. It can be downgraded to a read lock if inspection of *c* shows that no update action is needed. If it is required, the lock mode will be upgraded to an X lock after all existing read locks on *c* are released.

	NR	IX	LR	SR	CX	U	X
NR	+	+	+	+	+	+	-
IX	+	+	+	-	+	+	-
LR	+	+	+	+	-	+	-
SR	+	-	+	+	-	+	-

CX	+	+	-	-	+	+	-
U	+	+	+	+	+	+	-
X	-	-	-	-	-	-	-

Table 5: Compatibility Matrix

Table 5 above shows the compatibility matrix of this locking protocol. The NR lock mode is compatible with all locks except X. U doesn't create a conflict because it is performing a read operation on the context node. However, if it is upgraded to X, it will then create a conflict with NR. The IX lock is incompatible with SR because the subtree under question is being modified creating the possibility of a non-serializable schedule. A LR lock on a node N protects N and its direct children from concurrent write operations. Therefore CX and X are considered incompatible to LR. The SR lock mode disallows any concurrent write operations anywhere in the subtree of the node being locked. Therefore IX, CX and X are all incompatible to SR. The CX lock conflicts with LR, SR and X. The first two will create a read-write conflict with CX which can lead to a non-serializable schedule. X can create a write-write conflict with CX if both transactions are trying to update the same nodes.

3.3.2 Navigational Access

The DOM APIs we work with when manipulating XML documents have methods which enable the traversal/modification of trees by specifying access relative to context nodes. Table 6 below shows examples of DOM operations which allow us to observe and modify the structure of XML documents. For example, *getNextSibling()* and *getFirstChild()* return the next sibling and first child of a node respectively. *appendChild()* allows us to add a child to the end of the parent node's children list. To ensure that sequences of such method calls always return the same result nodes, the concept of virtual navigation edges is introduced as shown in Figure 10 below. The edges of element nodes are locked in addition to their confining nodes. While navigating the XML document a transaction requests a lock for each edge as well as nodes visited.

Three locks ER, EU, EX are introduced to handle this edge locking by Haustein and Harder [10]:

- An ER (Edge Read) lock mode is needed for an edge traversal for reading purposes such as getting the previous sibling of the context node c.
- An EX (Edge Exclusive) lock mode enables an edge to be modified when nodes are deleted or inserted for example. For any edge affected by a

modification operation, an EX lock has to be acquired before the edge is redirected to its new target node.

- An EU (Edge Update) lock is similar to the U lock discussed earlier.

Structure Mutators	insertBefore replaceChild removeChild appendChild
Structure Observers	firstChild lastChild previousSibling nextSibling getNodeById getElementByTagName

Table 6 DOM Operations

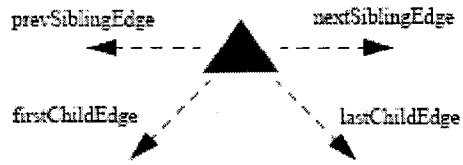

Figure 10: Virtual Navigation Edges

3.3.3 Suitability Discussion

The locking protocol introduced in this section has 10 locks as compared to the four locks maintained in Hierarchical Locking. We feel that this locking protocol is unsuitable for implementation in the McXML database system. Too many logical locks will be difficult to maintain and will add unnecessary complexity to the locking algorithms. Many of the locks are not needed for our purposes. Since we just read a specific node to match a path search or a whole subtree of nodes for returning to the user, LR is not needed. We do not need to downgrade locks. If a node does not match our path and predicate criteria, we simply remove its lock. Upgrades can be done directly on read/write locks such as S and IX, to save on the number of locks. Edge locking cannot be implemented in its entirety in our system because nodes do not hold pointers to their siblings; all traversals are done through the parent node. Furthermore, we feel that edge locking is too low a granularity level and can be done without.

3.4 2PL Protocols

Helmer et al. [11] propose four different core protocols (Doc2PL, Node2PL, NO2PL, OO2PL) for synchronizing access to and modification of XML documents. All four protocols require that documents are traversed top down from the root node. As in Section 3.3, it is assumed that each node in the DOM tree has pointers to its first child, last child, next sibling and previous sibling. These 2PL protocols are also based on the standard DOM operations for structure traversal and modification outlined in Table 6 in Section 3.3.

Helmer et al. [11] propose a shared lock T that has to be acquired for traversing the document structure and an exclusive lock M for modifying document structure. The conflict matrix, analogous to the one for S and X in Table 3, is shown in Table 7 below.

	T	M
T	+	-
M	-	-

Table 7: Conflict Matrix

3.4.1 Doc2PL

This protocol is the simplest and locks at the document level by placing a T or M lock at the root node depending on whether the operation is a structure traversal or modification respectively. This is acceptable if the document has only multiple readers. Any concurrent read-write or write-write operation is disallowed even if the operations are executing in different parts of the DOM tree. Surprisingly, the authors point out that this is a widely used locking protocol for XML base management systems such as in Tamino described by Chaudhri et al. [1].

3.4.2 Node2PL

The Node2PL protocol acquires locks for ancestor nodes. For example if in Figure 11 we want to traverse to the last child C3 of P, we need a T lock on P because it is C3's parent. Similarly, if we want to insert a new child C0 before C1, we need to acquire an M lock for P. This example shows an important deficiency in Node2PL. Both these operations should be allowed to execute concurrently but are blocked unnecessarily.

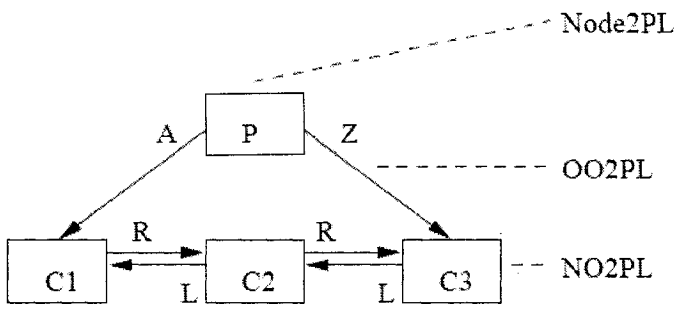

Figure 11: Different Lock Locations

3.4.3 NO2PL

This protocol acquires locks for all nodes whose pointers are conceptually traversed or modified. For example, if we want to add a new child C2.5 after C2 in Figure 11 above, it will require T locks on P and C1 because P's first child pointer and C1's right sibling pointer have to be traversed to get to C2. It will also require M locks on C2 and C3 because their right and left sibling pointers need to be modified respectively. However, we will not need a lock on C2.5 since no transaction will be able to reach this node. Both C2 and C3 are M-locked disallowing any traversals through them. Another concurrent transaction won't be allowed to set a T lock to read C3's children. This is an unnecessary conflict because the transactions are operating in the different parts of the tree and don't affect each other.

3.4.4 OO2PL

Whereas in the previous two protocols we locked nodes, OO2PL locks pointers as shown in Figure 11. As there are four pointers for every node (first child (A), last child (Z), left sibling (L), right sibling(R)) we need four traversal locks and four modification locks. The locks are TA, TZ, TL, TR, MA, MZ, ML, MR respectively.

3.4.5 Suitability Discussion

Doc2PL has the fewest number of locks: at most one per transaction per document. In Node2PL and NO2PL, we have at most one lock per transaction per node. However Node2PL never acquires any locks at the leaf level of documents whereas NO2PL does. OO2PL acquires at most four locks per transaction per node which is four times as many locks as NO2PL.

For our purposes, Doc2PL is not attractive because it does not allow concurrent read/write and write/write operations in different parts of the tree. We have already shown the deficiencies with Node2PL and NO2PL. OO2PL is similar conceptually to edge locking we saw in Section 3.3. Firstly in McXML, nodes do not hold pointers to their next sibling and previous sibling. They also only have a first child pointer so the last child can only be accessed by iterating through the whole children list of the parent node. We argue also that four times as many

locks as there are nodes in the DOM tree can lead to bad performance in large trees.

3.5 DGLOCK Protocol

3.5.1 DataGuides

DataGuides [12] are dynamically generated and maintained structural summaries of semi structured databases. Goldman and Widom [12] specify that a DataGuide must describe every unique path of the source (i.e., in our case the XML DOM tree) exactly once regardless of the number of times it appears in the source. For accuracy, the DataGuide encodes only paths that appear in the source.

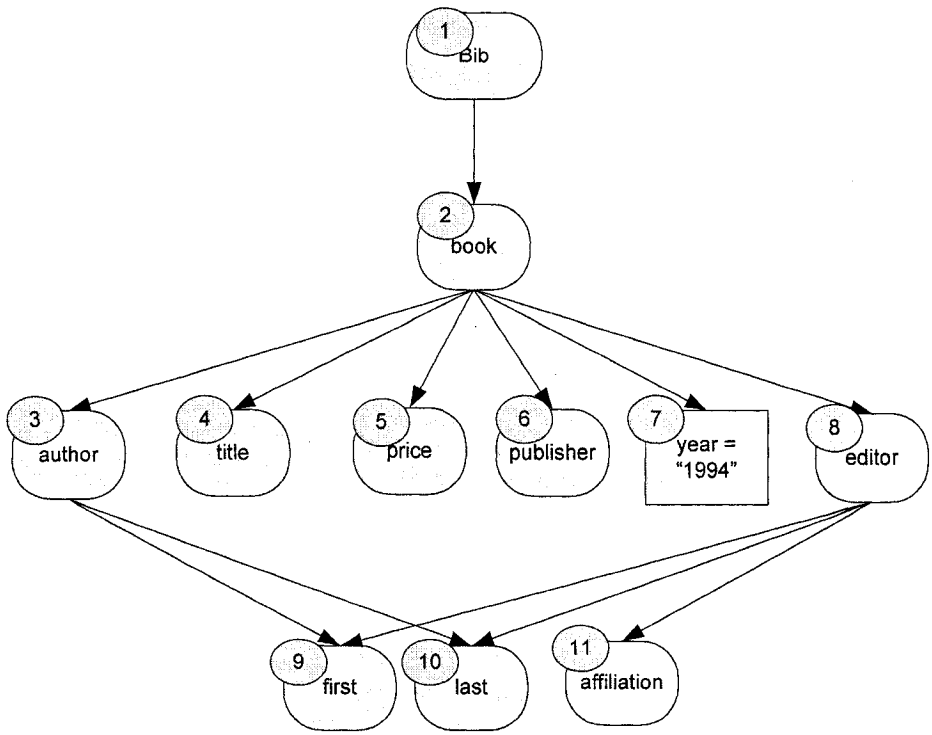

Figure 12 DataGuide for Books.xml

Figure 12 above illustrates a DataGuide for the Books.xml document in Figure 1. It matches both parts of the previous definition. For example, the path *Bib/book/author/first* appears only once in the DataGuide even though it appears multiple times in the source. In addition, every path in Figure 12 is a valid path that appears at least once in the source. DataGuides can be useful from a database user perspective. We can check whether a path of length *n* exists in the source by examining at most *n* levels of the DataGuide. For example, to check whether the path *Bib/book/publisher* exists in the source, we just need to check the first three levels of the DataGuide.

3.5.2 DGLOCK Protocol Description

Whereas in the previous protocols we have looked at implementing locking at the XML DOM tree level, DGLOCK [22] is a locking protocol which provides concurrency control by locking the DataGuide instead. Locking the DataGuide is attractive since the DataGuide is a much smaller data structure. At this point, it is important to introduce some terminology on constraints of requests. Structural constraints are constraints on the structure of documents whereas content constraints are constraints on the content of elements. For example, consider the path expression */Bib/book[price >1000]. Bib/book* is a structural constraint whereas *[price>1000]* is a content constraint. DGLOCK takes both kinds of constraints into account. With regards to structure constraints, DGLOCK uses hierarchical locking on the DataGuide of the XML document. Predicates are tagged to the locks held on the nodes of the Data Guide to deal with content constraints. It is important to handle content constraints using predicates because otherwise it might lead to many false conflicts. A simple hierarchical locking scheme would disallow any concurrent operations on two different books in Figure 2 because they are represented by the same path in the DataGuide.

	IS	IX	S	X
IS	+	+	+	P
IX	+	+	P	P
S	+	P	+	P
X	P	P	P	P

Table 8 DGLOCK lock compatibility

Table 8 above shows the DGLOCK compatibility matrix. The semantic meanings of S, X, IX and IS are the same as mentioned in Section 3.2 (Hierarchical Locking). However for some matrix entries, we see a P which represents a predicate test. DGLOCK provides for annotations of locks with simple predicates. Grabs et al. [22] describe simple predicates as conjunctions of comparison of the form x θ const where $\theta \in \{=, \in, \neq, \leq, \geq, >, \ldots\}$. The DGLOCK matrix does not contain strict incompatibilities; an incompatibility occurs only if the predicates of locks already granted and the one of the lock requested are not compliant.

The basic algorithm DGLOCK uses for a new request *s* which is as follows:

1. Extract all the constraints. We must obtain all path expressions ε that lead to data that is queried or updated by *s* (i.e. extract the structural constraints). Annotate all elements of ε with the predicates that reflect the respective content constraint.
2. Compute the set N of all nodes of the DataGuide that match any $e \in \varepsilon$ differentiating being nodes written and read.
3. For each node $n \in N$, perform the following operations using the lock compatibility matrix:
   a. If n is updated by s, acquire IX locks on nodes along the path leading from the root to n. Then acquire an X lock on n itself.

      For each path node, we must take the annotations from the DataGuide as well from ε into account.
   b. If n is only read by s, acquire IS locks on all nodes along at least one path that leads from the root to n. Then acquire an S lock on n. Once again, we must take all annotations on path nodes into account as we request locks.

The following example illustrates how DGLOCK is used to detect conflicts in concurrent XML transactions. Figure 13 shows the Books.xml DataGuide with annotations of locks and their predicates for two concurrent transactions T_1 and

T_2. T_1 wants to retrieve the author elements for all books whose price is greater than 60. T_2 wants to change the prices of all books with price less than 10 to 70. The predicate test on node 5 reveals that these two predicates are not compatible and therefore one of the transactions is blocked.

3.5.3 Suitability Discussion

Although DGLOCK can be implemented on the McXML database, we feel it is unsuitable for the following reason. DOM trees for large XML documents themselves take up large amounts of memory. A DataGuide will consume more of this limited pool of memory and will unnecessarily hinder performance. Furthermore, the maintenance of the DataGuide is non-trivial and has to be included in the transaction management overhead. Finally, predicate checking is a complex procedure. Therefore, we will implement a locking protocol which locks the XML DOM tree rather than an indirect structure.

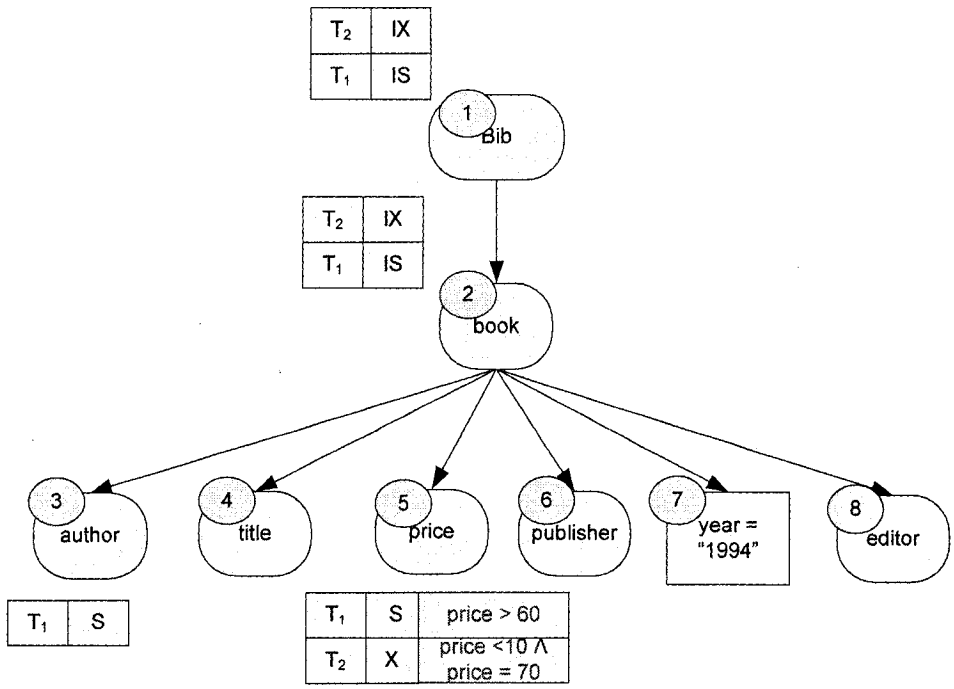

Figure 13 Locking on the DataGuide

3.6 Snapshot based Concurrency Control Protocols

Since lock acquisition can be complex and potentially leads to high blocking rates, Sardar [13] proposes two snapshot-based concurrency control protocols OptiX and SnaX. These protocols avoid any read locks by providing transactions a committed snapshot of the data. This is a practical solution because most operations in standard applications are read-intensive. OptiX enhances traditional optimistic concurrency control to work on XML while SnaX offers snapshot isolation similar to relational database systems like Oracle and Post-greSQL.

3.6.1 Snapshots

In this section, we look at the snapshot mechanism used by both protocols. Sardar [13] provides virtual snapshots using a multi-version system. The basic concept used is that every update of a data item (i.e., XML node) creates a new version. A transaction *T* then accesses the latest committed version as of the time *T* started.

Sardar [13] implements such a multi-version system using timestamps. Every transaction is assigned a unique identifier on start-up. A list $EB(T_i)$ is maintained for each transaction which holds the identifiers of all transactions T_j such that T_j committed before T_i as well as the identifier of T_i itself. Each transaction T_i then only reads versions created by transactions from $EB(T_i)$. In addition, each node *N* in an XML document is assigned two timestamps: a valid timestamp and an invalid timestamp. The valid timestamp $V(N)= ID(T_i)$ indicates the transaction T_i that created this node. Similarly, the invalid timestamp $IV(N)= ID(T_i)$ indicates the transaction T_i that deleted this node. If no transaction has deleted this node so far its invalid timestamp is set to NULL.

3.6.1.1 Reading from a Snapshot

The timestamps mentioned earlier are used to ensure that the transaction T reads a representation of the XML tree which is a snapshot at the time T starts. [13] ensures this by requiring T to only read a node N if it fulfills the following condition: $V(N) \in EB(T) \wedge IV(N) \notin EB(T)$.

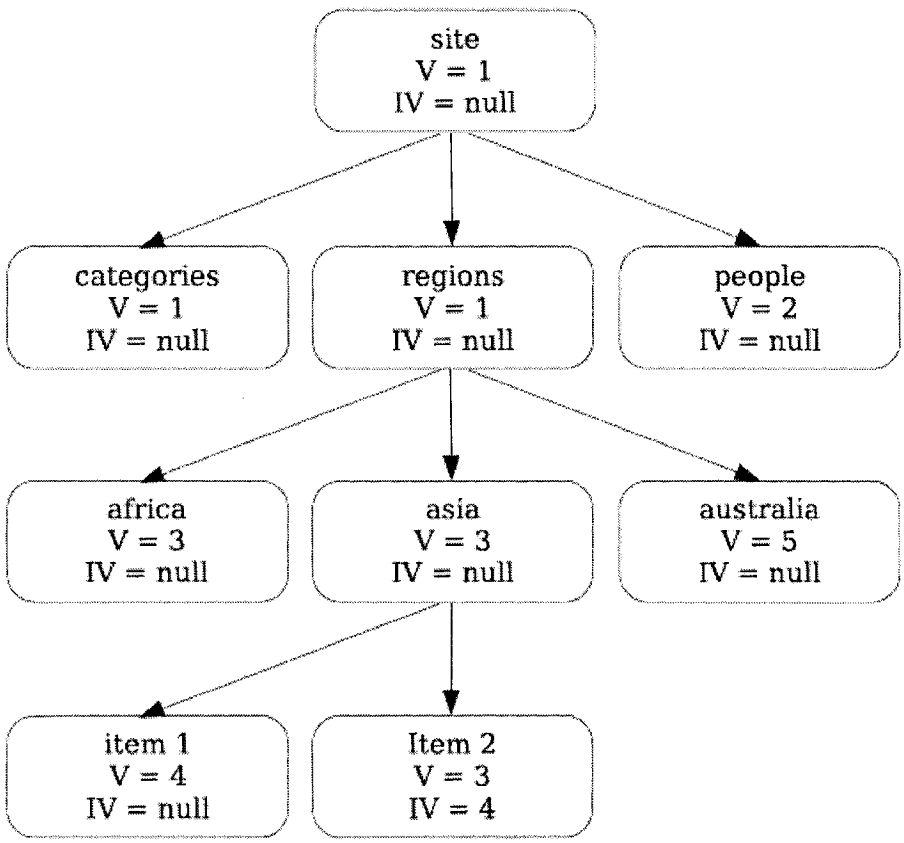

Figure 14 Complete XML tree

For example, consider Figure 14 above which represents a complete XML tree with all versions. Assume that $EB(T_5) = \{1,2,3,5\}$ and T_5 is concurrent to T_4. Based on this information, T_5 will see the XML tree shown in Figure 15 below containing the changes of $EB(T_5)$. It does not read the item 1 child of the *asia* node because it has been created by a concurrent transaction T_4. Similarly, T_5 sees the Item 2 child of the *asia* node even though it has been deleted by T_4. Therefore, it correctly uses the timestamps to see a snapshot of the tree taken when it started and does not see any changes made by concurrent transactions.

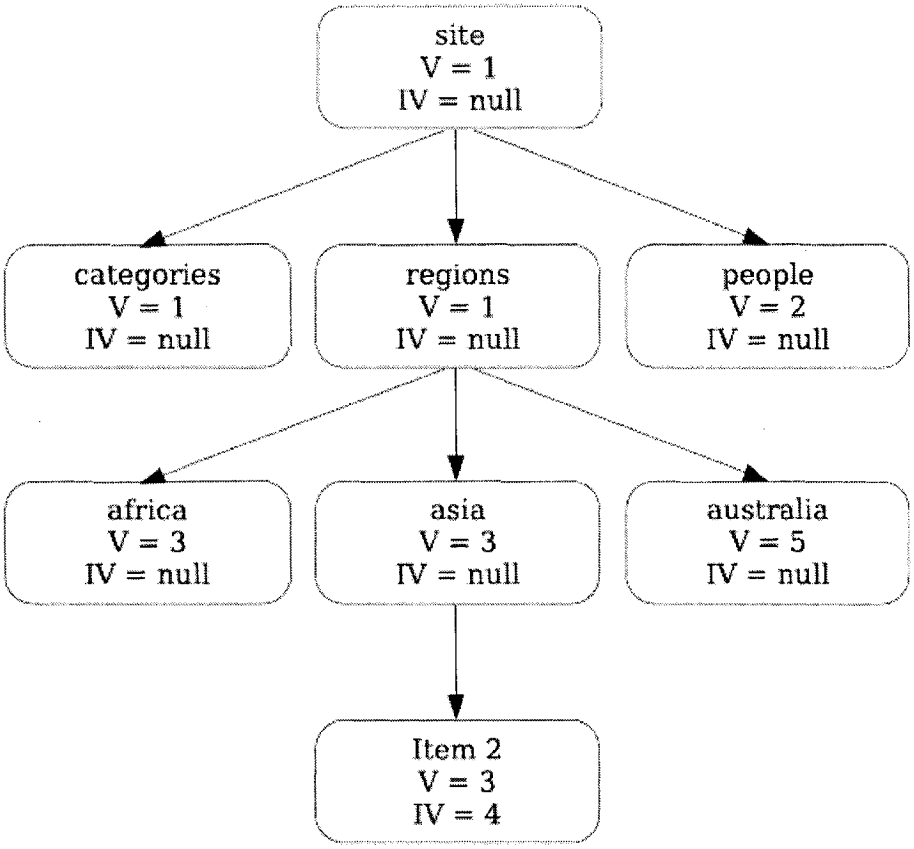

Figure 15 XML tree seen by T_5

3.6.2 OptiX: Optimistic Concurrency Control for XML

Sardar [13] adjusts traditional optimistic concurrency control so that it works with his implementation of snapshots. OptiX also takes into consideration the hierarchical structure of XML documents as well as the McXml Query Execution Engine.

Traditional optimistic concurrency control, based on backward validation, compares the validating transaction's read set with the write sets of concurrently executing transactions which have already committed. If there is an overlap in these two sets, the validating transaction aborts because it has not read the values of data items according to the appropriate serialization order. OptiX breaks down the read and write sets into more granular subsets. The read set becomes the combination of a set of nodes whose subtrees are returned to the user, a set of nodes that are explicitly read and a set of nodes that are explicitly read for an insertion after them. Similarly, the write set is composed of sets for nodes that are deleted, renamed and inserted into. These sets are compiled in the WORKING PHASE. A conflict matrix is created which is used in the VALIDATION phase to determine whether there really is a conflict between the specific read set of the validating transaction and the specific write set of the validated transactions. If so, the validating transaction aborts immediately. Otherwise, it proceeds to the UPDATE phase and writes its changes to disk.

3.6.3 SnaX: Snapshot Isolation for XML

Sardar [13] proposes a second concurrency control protocol which provides Snapshot Isolation(SI) . Snapshot Isolation (SI) is a relatively new isolation level based on multi-version concurrency control that avoids the overhead of tracking

reads [14]. In SI, a transaction *T* reads a snapshot of the database containing all the updates committed as of the time *T* started. Read transactions always succeed and do not require any concurrency control. Whereas OptiX aborted a transaction that read an object written by a previously validated transaction, SnaX aborts a transaction if it writes an object that was written by an earlier concurrent transaction. Sardar [13] indicates that not keeping track of reads at all greatly reduces the overhead of the protocol because most practical applications are read-intensive. The number of conflicts between concurrent transactions are also reduced because reads are not tracked.

Sardar [13] uses an approach similar to that of relational systems like PostgreSQL and Oracle. These systems implement SI using a combination of multi-versioning and locking. In such systems, each update of a data record of a table creates a new version of the record. A read operation on a data record reads the last committed version before the transaction started. Whenever a transaction wants to update a data record x, it has to acquire an exclusive lock on x and perform a version check. If the last committed version of x was created by a concurrent transaction, T_i aborts immediately. Otherwise it performs the operation. If a transaction T_j holds a lock on x when T_i requests it, T_i is blocked. When T_j commits, the lock will be granted to T_i. However T_i will then fail because

the version check will indicate that the last committed version is by concurrent transaction T_j. If T_j had aborted, T_i's version check might still succeed or fail depending on other concurrent transactions and whether they have committed a version of x. Version checks are performed using the timestamps introduced in 3.6.1. A version check fails if there is a data record with a valid timestamp of a concurrent committed transaction and a NULL invalid timestamp.

Chapter 4 LockX Theory

In this chapter, we introduce our concurrency control protocol LockX, based on locking, for the McXML database system. We will explain LockX on a high-level and focus on the theoretical aspects necessary to understand it well. Detailed explanation of LockX will follow in the next chapter.

4.1 LockX Pitfalls

4.1.1 Serializability

The primary goal of LockX was to design and implement a concurrency control protocol for McXML which would guarantee serializability as defined in Section 2.6.3.

As explained later in this chapter, we allow certain operations to run concurrently because semantically they don't conflict with each other. For example, two Insert-into operations don't conflict with each other. Assume transaction T_1 wants to insert into all *book* elements a *size* element. Simultaneously, transaction T_2 would like to insert into all *book* elements a *rating* element. Note that the McXML implementation would insert these new nodes at the end of the child lists for the *book* elements. Assuming concurrent execution on the tree in Figure 2, the

ordering of *size* and *rating* elements could be different in Subtree(2) and Subtree(15). However, if the execution were equivalent to a serial execution, they would have the same ordering in the two subtrees.

We argue that although the results don't indicate that such an execution is serializable, it is semantically equivalent to either possible serial schedule. We care only that the end result of such an execution has both *size* and *rating* elements added to both *book* elements; their respective ordering is not important. Therefore our definition of serializability is based on a semantic equivalence to a serial execution when a result-based equivalence is not essential. Note in a relational system this problem does not occur because the resources of a table are unordered by definition. In the tree-based XML data structure, there is an ordering between siblings.

4.1.2 Avoiding Phantoms

We wanted LockX to avoid phenomena such as *phantoms* [14]. Consider the case where transaction T_1 is retrieving all *book* elements with a price of 39.95. Another concurrent transaction T_2 meanwhile wants to set 2's *price* element to 39.95. Assume that T_2's update is executed after T_1 reads 13 and therefore deems 2 unsuitable (releasing any locks on it). T_1 will therefore return only

Subtree(15) as part of its results. However if T_1's query is run again, it will return both *Subtree(2)* and *Subtree(15)*. Therefore this execution would not be serializable because one of the results indicates that T_1 executed before T_2 whereas the other indicates the opposite.

To avoid *phantoms*, T_1 would be required to lock all *book* elements so that no unforeseen changes are made to them by other transactions. However this is too coarse a locking granularity and would create unnecessary blocking situations. For example, another transaction wanting to add a *size* element to 2 would be blocked even if 2 would never match the predicate specified by T_1 in its search. We therefore allow *phantoms* in LockX, to allow more concurrency, by using more fine-grained locking. Phantoms are also a common problem in relational databases which typically allow them in order to have fine-grained locks on the record level.

4.2 Lock Types

LockX's locks are modelled after those introduced in Section 3.2 (Basic Hierarchical Locking). However, we have changed their semantics and granularized the S and X locks to better fit the design of the McXML Query

Execution Engine. We now introduce the various lock types used by the LockX protocol.

4.2.1 Read Locks

LockX uses three different read locks (IS, S, RR) which are explained through the following XQuery example executed on the tree of Figure 2.

- ```
  FOR $b in document("Bib.xml")//book, $l in $b/author/last
  WHERE $b/price=39.95
  RETURN $l
  ```

1. S: The S (Shared) lock is used on nodes which are explicitly read by the XQuery through a structural or content constraint. Explicitly read nodes don't have their descendents read (only themselves). In the above XQuery, S locks would be placed on nodes 15 (book), 19(author) and 31(price).
2. RR: The RR(Read Return) lock is placed at the root of a subtree which is returned as the result of the query. It implicitly means that nodes below the root are also read. This lock applies only to read queries because only they contain RETURN clauses. In the above XQuery, an RR lock would be placed on the *last* element labelled 22.
  ```

3. IS: The IS (Intention Shared) lock is used on nodes which are not explicitly read by the XQuery but are part of a path to a node that is explicitly read or returned. For example in the XQuery above, the *Bib* element would have an IS lock placed on it because it is an ancestor to a *book* element which is explicitly read. This lock therefore indicates the intention to explicitly read or return a descendent node and preserves the path from conflicting operations.

4.2.2 Write Locks

From the suitability discussion in Section 3.2.2, we discovered that the X lock was too coarse a granularity for our system because we have six different kinds of update operations (delete, replace, rename, Insert-into, Insert-after, Insert-before). This can lead to artificial conflicts where concurrency is acceptable. To control exactly the conflict behaviour of these update operations, we have decided to create a separate exclusive lock for each of the update operations. The following Update XQueries from Section 2.5 will be used as examples.

- FOR $b in document("books.xml")/Bib//book, $p in $b/publisher

  LET $t := $b/title

  WHERE $b/@year = 1998

UPDATE $b {

INSERT <award>Pulitzer Prize</award>

RENAME $t to name

REPLACE $p with <editor>Rapunzel Editors</editor>

}

- FOR $b in document("books.xml")/Bib//book, $p in $b/publisher

LET $t := $b/title

WHERE $b/@year = 1994

UPDATE $b {

INSERT <publisher>O'Reilly Publishers</publisher> AFTER $p

INSERT <publisher>Morgan Publishers</publisher> BEFORE $p

DELETE $t

}

The write locks introduced by XLock are as follows:

1. RP: The RP (Replace) lock is applied on node *p* for the *replace(p, Subtree(q))* operation. It implicitly locks Subtree(p) disallowing read/updates on *descendents(p)* and *p* itself. In the first update query above, the RP lock will be applied to the *publisher* element labelled 29.

2. II: The II (Insert-into) lock is applied on node *p* for the *Insert-into(p, Subtree(q))* operation. In the first update query above, an II lock would be applied to the *book* element labelled 15.

3. RN: The RN (Rename) lock is applied on node *p* for the *rename(p, newname)* operation. In the first update query above, an RN lock will be applied to the *title* element labelled 17.

4. IA: The IA(Insert-after) lock is applied on node *p* for the Insert-after(p, Subtree(q)) operation. In the second update query above, an IA lock is applied on the *publisher* element labelled 11.

5. IB: The IB(Insert-before) lock is applied on node *p* for the Insert-before(p, Subtree(q)) operation. In the second update query above, an IB lock is applied on the same *publisher* element labelled 11.

6. D: The D(Delete) lock is applied on node *p* for the *delete(p)* operation. In the second update query above, a D lock would be applied on the *title* element labelled 4.

7. IX: The IX lock on *n* has similar semantics to IX in Basic Hierarchical Locking. For each of the above update locks on a node *p*, *ancestors(p)* have to be IX-locked and S-locked during the top-down tree traversal.

4.3 LockX Expected Results

We now give some examples of the locking patterns we are seeking at the end of query execution to see these locks in action.

4.3.1 Read Queries

Consider the McXML query operation from Section 2.4.1 shown below. Figure 16 shows what the McXML DOM tree from Figure 2 should look like once this query operation is run using LockX. The locks are shown underlined on the left hand side of the context node.

- FOR $b in document("books.xml")/Bib/book

  LET $t := $b/title

  WHERE $b/price < 139.95

  ORDER BY $b/price

  RETURN $t

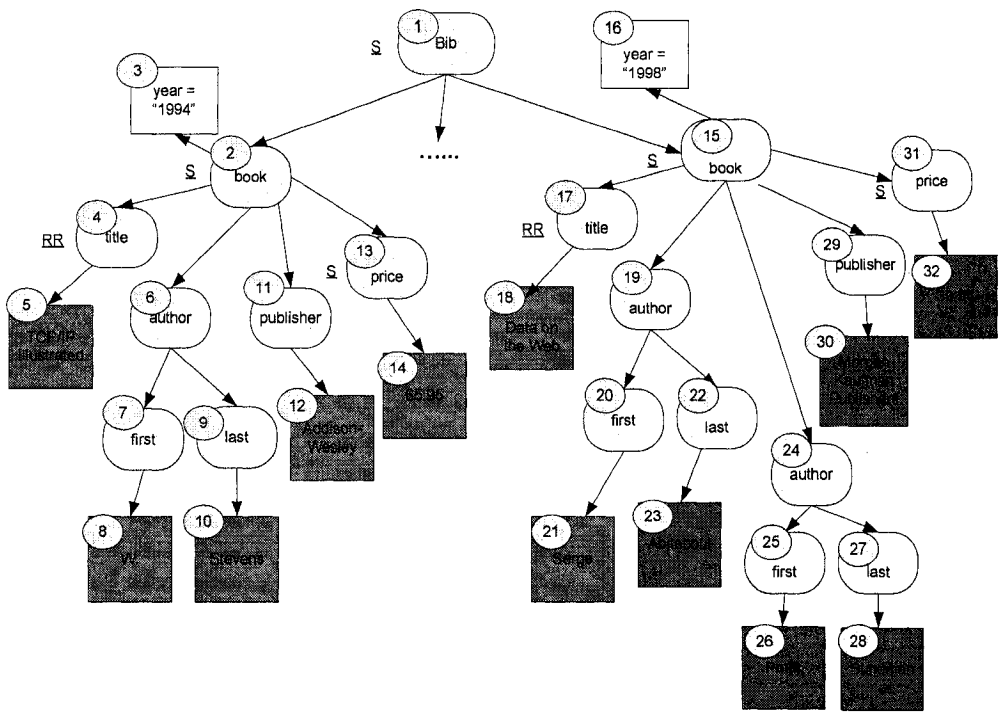

Figure 16 Locking Results (Read Query 1)

Both *book* nodes meet the search criteria. Therefore we have S locks on both as well as *Bib* and *price* elements which are also explicitly read. Since *title* elements are explicitly read, they are *S* locked initially. However they are later converted to *RR* because their subtrees are being returned by the query.

- FOR $b in document("books.xml")//book, $l in $b //last

  WHERE $l = "Stevens"

  RETURN $b

For this second query, Figure 17 shows what we expect the end result of the operation to look like. The *Bib* element is *IS* locked because it is not explicitly read but it is part of a path to the *book* element labelled 2. The *last* element is also *S* locked but notice that the path between it and its *book* element is not locked. This is not necessary because the node 2 is *RR* locked (*Subtree(2)* is returned to the user).

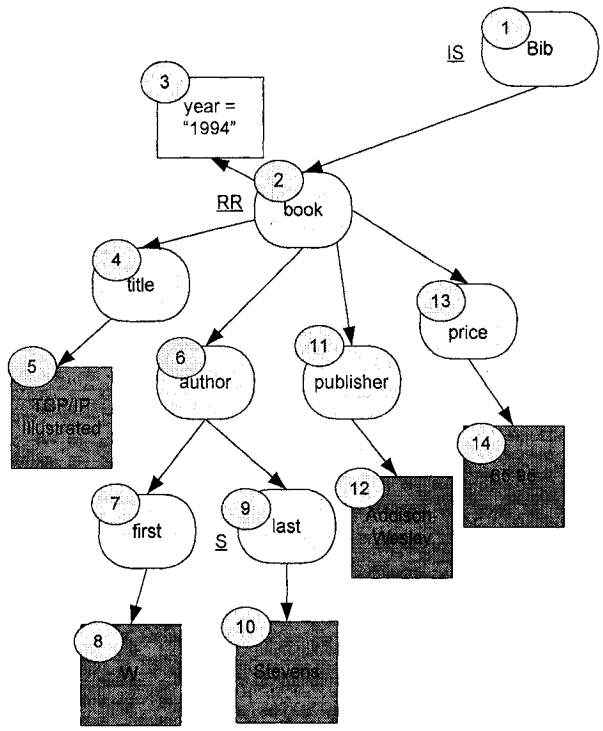

Figure 17 Locking Results (Read Query 2)

4.3.2 Update Queries

All update queries follow a similar theme of IX-locking *ancestors(N)* of node *N* being updated and applying a specific update lock (D, RN, RP, II, IA, IB) to *N.* We illustrate this with the following rename operation.

- FOR $b in document("Bib.xml")//book

  LET $a := $b/author

  WHERE $b/price < 100 AND $b/@year=1994

  UPDATE $b { RENAME $a TO writer }

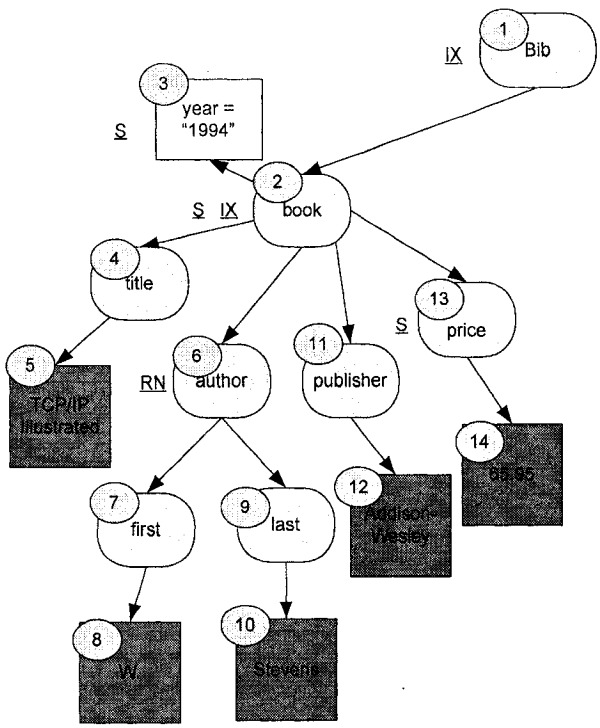

Figure 18 Locking Results(Rename Operation)

Figure 18 shows the locks acquired by the end of the operation's execution. Locks are displayed to the left of the context node as before. Node 6 is RN locked and *ancestors(6)* are IX-locked to protect its path while its being renamed. Since 13,3 and 2 are explicitly read, they are S locked as well.

4.4 Compatibility Matrix

In order to detect conflicts between the different read/write operations correctly and avoid artificial blocking situations, we have devised a compatibility matrix similar to the ones in previous sections. Table 9 below shows the compatibility matrix for the LockX concurrency control protocol on a node *n*. The horizontal row of locks represents the lock already held by transaction T_1. The vertical column of locks represents the lock another transaction T_2 wants to acquire.

Note that compared to the locking protocol described in Section 3.3, we have the same number of logical locks(10). However, by removing unnecessary locks such as LR and U, we were able to finely granularize our locking protocol, based on the varying needs of our read/write operations, to maximize overall concurrency. For example, Section 3.3's locking protocol had one update lock for all update operations whereas we have six different kinds of update locks (one

for each update operation) avoiding unnecessary blocking situations as described below.

	$n(T_{has})$	RR	S	RN	II	IA	IB	RP	D	IS	IX
$n(T_{wants})$											
RR		+	+	-	-	+	+	-	-	+	-
S		+	+	-	+	+	+	-	-	+	+
RN		-	-	-	-	-	-	-	-	+	+
II		-	+	-	+	+	+	-	-	+	+
IA		+	+	-	+	+	+	-	-	+	+
IB		+	+	-	+	+	+	-	-	+	+
RP		-	-	-	-	-	-	-	-	-	-
D		-	-	-	-	-	-	-	-	-	-
IS		+	+	+	+	+	+	-	-	+	+
IX		-	+	+	+	+	+	-	-	+	+

Table 9 LockX Compatibility Matrix

4.4.1 Deciding conflicts

We give two motivating examples that give an intuition why two locks should or should not conflict.

Example 1 (No Conflict)

In this example based on Figure 2, we assume that transaction T_1 is inserting into all *book* elements a new element of the form *<size>100 pages</size>*. Simultaneously, transaction T_2 is inserting into all *book* elements a new element of the form *<editor>McGill Editors</editor>*. Since we don't care about the ordering of nodes in the tree, these two operations don't conflict. Even if the order of execution of the two transactions is different in *Subtree(2)* and *Subtree(15)*, the end result is the same. A *size* element and *editor* element have been added to both *book* elements. Therefore two II locks are compatible with each other in Table 9.

Example 2 (Conflict)

Assume that transaction T_1 wants to rename all *book* elements in Figure 2 to *magazine* elements. A transaction T_2 wants to return all *book* elements. Consider a scenario where T_1 executes before T_2 in *Subtree(2)* but after T_2 in *Subtree(15)*. T_1 is able to rename both 2 and 15 but T_2 only returns *Subtree(15)* because 2

has been renamed. We will now try to establish a serialization order for these two transactions. If T_1 had executed before T_2, it would imply that T_2 would return no results because all book elements had been renamed. If T_2 had executed before T_1, both *book* elements would be returned because the rename operations have not taken place yet. However, we have a situation where only one of the *book* elements has been returned. We have a non-serializable schedule and hence these two operations should conflict. Therefore, the RN and RR locks are incompatible in Table 9.

With a lock incompatibility, the above situation of a non-serializable schedule can never occur. T_1 would have an RN lock on 2 and T_2 would have an RR on 15. T_2 would be waiting on T_1 to release its RN lock on 2 and T_1 would be waiting for T_2 to release its RR lock on 15. A deadlock situation would occur leading to an abort of one of the transactions.

4.4.2 Detailed Analysis

We will now look at pairs of operations, *op1* and *op2*, on the same node and discuss whether they conflict or not. The reasoning is as follows. If *op1* is a read and would read something different depending on whether it executed before or after *op2*, then they conflict and their respective locks are not compatible. If *op1*

is a write and the effect on the tree (except node ordering) is different depending on whether *op1* executes before or after *op2*, then both operations conflict. We do not discuss all combinations but only a selection of the most interesting ones.

- RR/S/IS vs. RR/S/IS: All read locks are compatible with each other because multiple readers on the same node(s) do not endanger transaction serializability.
- RR vs. II: An Insert-into operation on node *n* would change the results of a Read-return operation on *n.* The result of the read is different depending on whether it executes before or after the insert. These two locks therefore conflict.
- RR vs. IA/IB: An Insert-after/Insert-before operation on *n* does not affect *Subtree(n).* The result of the read would be the same irrespective of the order of operations and there is no conflict.
- RR vs. RP/D: These lock combinations conflict because a Replace/Delete operation on *Subtree(n)* directly affects the return of *Subtree(n).*
- RR vs. IX: An IX lock on node *n* implies an update operation is occurring somewhere in *Subtree(n).* This would affect a Read-return operation on *n* and therefore the locks are incompatible.

- S vs. RN: An S lock on a node *n* implies that its name is being explicitly read to match a path constraint. Since a Rename operation would change *n*'s name, these locks are incompatible.
- S vs. II/IA/IB: An S lock on node *n* means that the operation is only reading n. Therefore the result of the read is the same independent of whether it runs before or after the insert operations.
- S vs. RP/D: A Replace/Delete operation on *n* would affect an explicit read on *n*. Therefore these lock combinations conflict.
- S vs. IX: An IX lock on *n* affects *descendants(n)* but not *n* itself. Therefore, there is no conflict.
- RN/D/RP vs. RN/D/RN: The order of two update operations on the same node matters and the end-result on the tree can be different based on this order. Therefore, there is a conflict between these locks.
- RN/RP/D vs. II/IA/IB: A Rename/Replace/Delete operation on *n* can affect whether an Insert operation can find its context node. Therefore there is a conflict for these lock combinations.
- RN vs. IS: These locks operate at different levels of *Subtree(n)* causing no conflict. Similar logic applies to II vs. IA/IB.
- IS/IX vs. II: These locks operate on different parts of *Subtree(n).* The Insert-into operation adds a new node to *Subtree(n)* while the IS/IX lock

implies an explicit read/write on an existing node in *Subtree(n)*. Therefore there is no conflict.

- IA vs. IB: Since the Insert-after and Insert-before operations run on opposite sides of *n*, there is no conflict.
- IB vs. IB: Assuming nodes *x* and *y* are being inserted before *z*, these two operations can be run concurrently because their respective order isn't important. Similar logic applies to IA vs. IA.
- D/RP vs. IS/IX: Since a Delete/Replace operation on *n* affects *Subtree(n)*, an operation working on a descendent of *n* would be affected. Hence these lock combinations do conflict.
- IX/IS vs. IX/IS: Intention locks on *n* do not conflict with each other because they indicate the intent to read/write somewhere below *n*. If there are conflicts, they will be detected further down the tree.

4.5 Handling Aborts

As support to LockX, we have implemented transaction aborts that rollback any changes made by the transaction on the XML DOM tree. We explain the logic on a high-level; more details will follow in the next chapter.

Consider the following transaction T_1 where operations are interdependent:

1. FOR $b IN document("Books.xml")/Bib/book

   WHERE $b/@year="1994"

   UPDATE $b { INSERT <size>100 pages</size> }

2. FOR $b IN document("Books.xml")/Bib/book, $s IN $b/size

   WHERE $b/@year="1994"

   UPDATE $b { RENAME $s TO length}

3. FOR $b IN document("Books.xml")/Bib/book, $l IN $b/length

   WHERE $b/@year="1994"

   UPDATE $b { INSERT <award>Pulitzer Prize</award> AFTER $l }

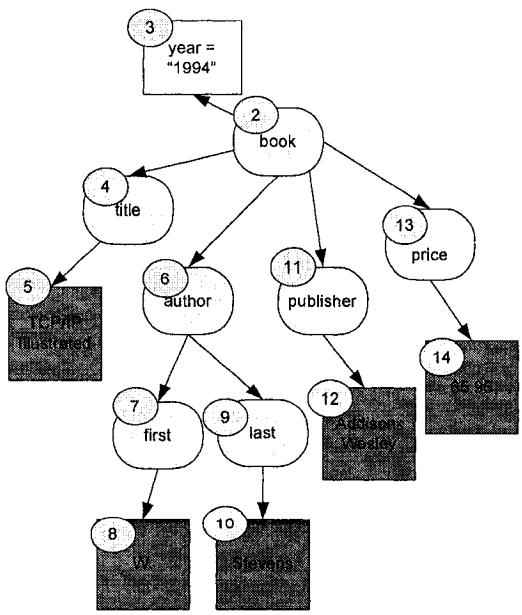

Figure 19 Before T_1

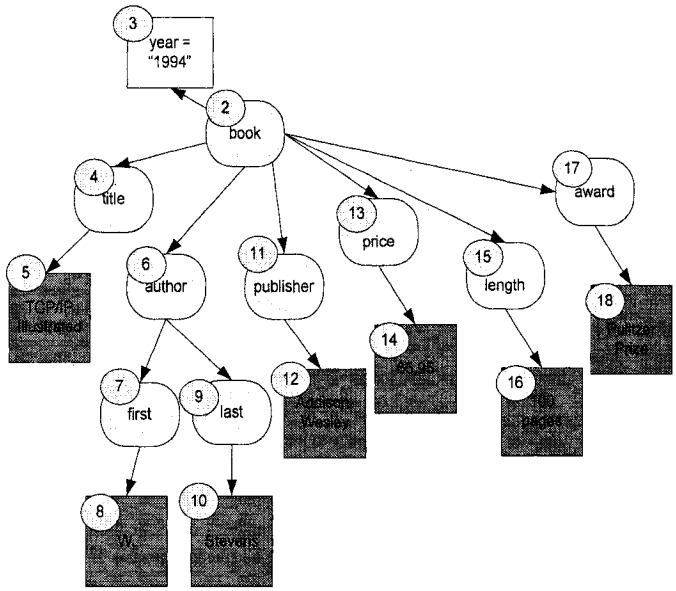

Figure 20 After T_1

Figures 19 and 20 show *Subtree(2)* from Figure 2 before and after T_1's execution. Now suppose that T_1 has to abort after executing all its operations. To revert *Subtree(2)*'s state to the one in Figure 19, it becomes necessary to apply *undo* operations for each of T_1's operations. The undo operations for T_1 are shown below in the same order as their original operations:

1. FOR $b IN document("Books.xml")/Bib/book, $s IN $b/size

   WHERE $b/@year="1994"

   UPDATE $b { DELETE $s }

2. FOR $b IN document("Books.xml")/Bib/book, $l IN $b/length

WHERE $b/@year="1994"

UPDATE $b { RENAME $l TO size}

3. FOR $b IN document("Books.xml")/Bib/book, $a IN $b/award

WHERE $b/@year="1994"

UPDATE $b { DELETE $a }

The order of execution is important because of the interdependencies between the transaction's operations. For example if they are executed in the same order as above, Subtree(15) would not be deleted because the first operation is not able to find the *size* element to delete. Executing these operations in the reverse order gives us the correct outcome.

We would like to mention that the above explanation outlines conceptually what must be done to implement aborts. Our implementation would not run the undo operations shown above. This would require us to re-find the target nodes which we already have and acquire unnecessary locks which could lead to deadlocks. Therefore, we apply undo operations directly on the relevant nodes. More details will follow in the next chapter.

Chapter 5 LockX Implementation

5.1 High-level overview

Figure 21 below shows the architecture of the LockX concurrency control protocol. There are four major components which are described below:

(1)The *Lock Manager* is the interface to the client (the Query Execution Engine) wishing to apply locking operations to nodes in the McXML DOM tree.

(2) The *Lock Table* is the component containing the data structures that hold the various locks of each transaction. It provides various operations such as adding a lock, removing a lock and converting locks.

(3) The *Compatibility Checker* is the component which defines the type of locks allowed by LockX as well as their compatibilities with each other.

(4) The *Deadlock Detector* builds a wait-for graph in memory and uses it to identify and remove deadlocks.

The two other components displayed are:

(1) Lock List: This is a list of all locks held on a node.

(2) Wait Queue: This is a queue where locks which are incompatible with the existing ones in the Lock List have to wait.

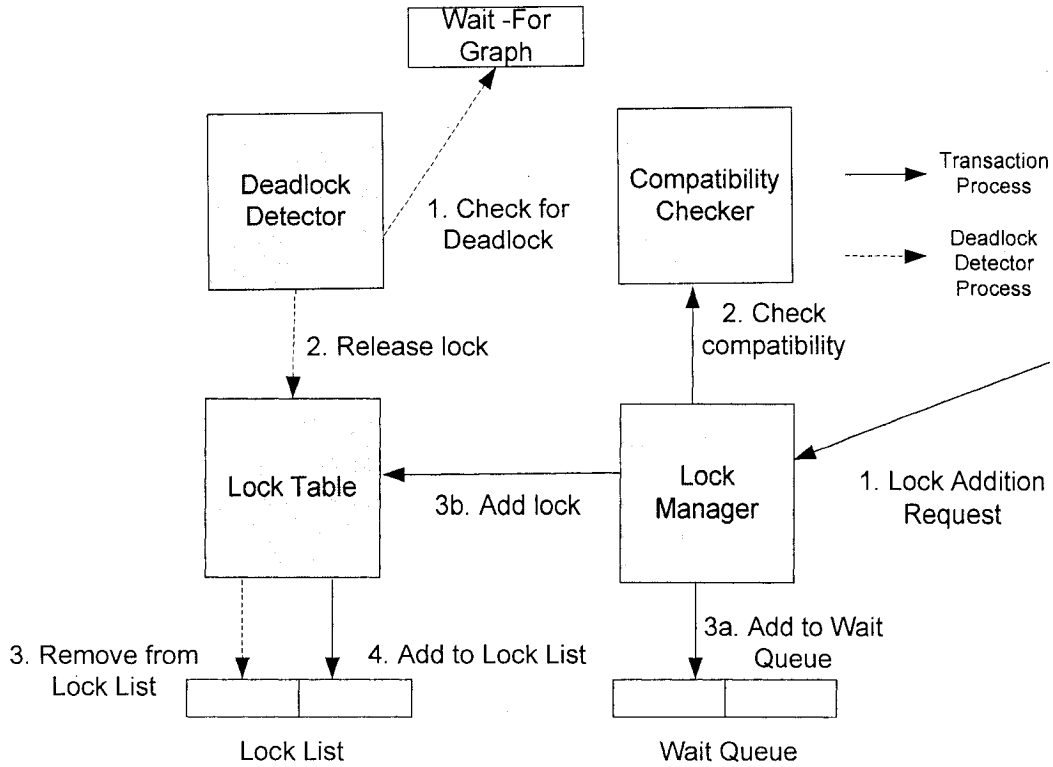

Figure 21 LockX Architecture

Figure 21 also gives a walkthrough of the LockX execution of a client request to add a lock to a specific node *N* in the XML DOM tree. The request is received by the Lock Manager which forwards it to the Compatibility Checker. The Compatibility Checker checks the lock's compatibility with the existing ones in *N*'s Lock List. If it is compatible, the Lock Manager forwards the request to the Lock Table which performs the addition to *N*'s Lock List. Otherwise, the lock must wait

in *N*'s Wait Queue for the conflicting locks to be removed from the Lock List. Periodically, the Deadlock Detector checks its wait-for graph for a cycle in a separate thread. If it finds one, it picks one transaction T in the cycle to abort and sends a request to the Lock Table to release all of T's locks from the various nodes it has accessed. When a transaction commits, it flushes from all Lock Lists any locks it holds.

5.2 LockX Components

5.2.1 Lock Table

Each lock, stored by the Lock Table, is uniquely identified by the transaction holding it and the node on which it has been applied. It is important to note, for the rest of this thesis, that nodes and transactions are represented in LockX by their unique identifiers Oid and Xid respectively. Therefore, locks are applied to the Oids representing the nodes rather than the nodes themselves.

To ease the use of LockX in McXML, it was necessary to be able to retrieve all the locks held by a transaction easily. In addition, each node in the McXML DOM tree will have a list of locks associated with it. Therefore we have implemented the Lock Table as a combination of two hash tables: McTransHash and McNodeHash. McTransHash stores for each transaction a list of all locks held by

it on various objects. Each list is uniquely identified by the transaction's identifier Xid. Each node in this list is a tuple of the form *(Oid, LockType)*. The McNodeHash stores for each node a list of all locks as well as the Xid's of their owning transactions. Each object in the list is a tuple of the form *(Xid, LockType)*. Each list is uniquely identified by an Oid. Figures 22 and 23 show what the McNodeHash and McTransHash could look like.

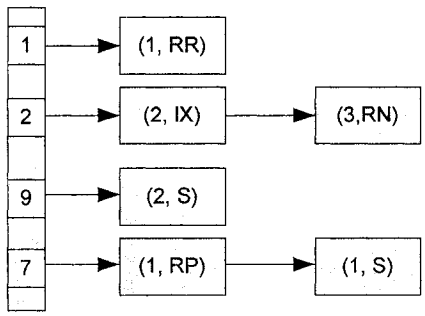

Figure 22 McNodeHash

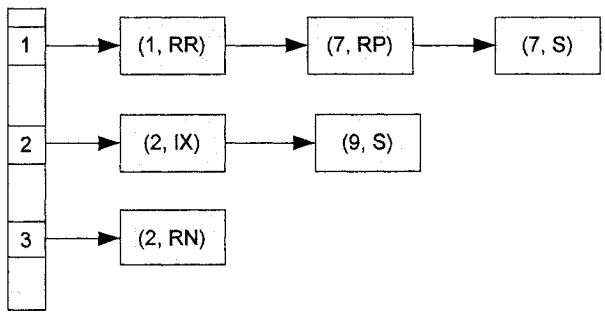

Figure 23 McTransHash

The Lock Table also provides API methods to the Lock Manager to modify these data structures. It is possible to:

1. Add a lock to a specific node *N* for a specific transaction *T*
2. Remove a lock from *N* held by *T*
3. Convert a lock *L1* to *L2* on *N* for *T*
4. Release all locks held by *T*

5.2.2 Compatibility Checker

The Compatibility Checker is responsible for storing the compatibility matrix derived in Chapter 4. This information is fed statically into LockX by the user. The Compatibility Checker then uses this matrix to compare the compatibility of two tuples of the form $(LockType_1, T_1)$ and $(LockType_2, T_2)$ received from the Lock Manager. If transactions T_1 and T_2 are the same, their locks are always considered compatible.

5.2.3 Lock Manager

The Lock Manager is the component which interacts with the client of LockX. The Lock Manager holds the logic to decide whether a transaction T_1's lock should be granted immediately on a node *N*. It compares the requested lock L_1 with all the

existing locks on *N* using the Compatibility Checker. If L_1 is compatible, it is added to the Lock Table. Otherwise, the Lock Manager stores the tuple (Xid_1, L_1) in a wait queue for *N* (where Xid_1 is the transaction identifier for T_1). This wait queue stores tuples for all transactions whose locks are incompatible with existing locks on *N*. The LockManager stores in a hash table a wait queue for each node in the McXML DOM tree. Each wait queue can be uniquely accessed using the node's *Oid* which is a key in the hashtable.

Whenever the list of locks on a node *N* is updated, the waiting transactions check whether their locks are now compatible. Only the transaction at the front of the queue is able to leave when suitable conditions arise i.e. when its requested lock on *N* is now compatible with those in *N*'s lock list. This FCFS (First Come First Served) constraint is used to ensure fairness.

5.2.3.1 Circumventing Scheduling Fairness

In some cases, the Lock Manager must insert new tuples at the front of the wait queue rather than the end. Although this circumvents fairness, it is necessary to avoid deadlocks. We explain the reasoning using the following example.

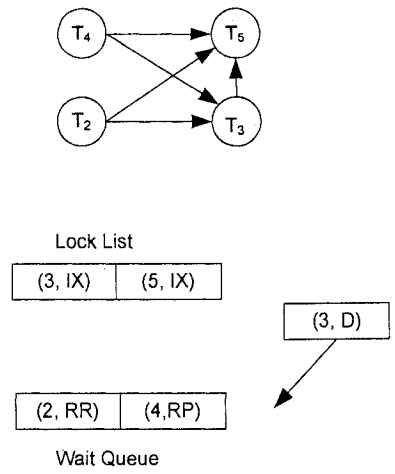

Figure 24 Undetected Deadlocks

Figure 24 shows a lock list on a node *N* as well as its corresponding wait queue. We can see that transactions 3 and 5 have an IX lock on *N.* RR and RP locks are incompatible with IX and therefore transactions 2 and 4 have to be added to the wait queue. Now suppose that transaction 3 wants to convert its IX lock to a D lock. The D lock is not compatible with the IX lock on *N* from transaction 5. We therefore add transaction 3 to the wait queue as before. The wait-for graph after this addition is also shown in Figure 24. According to this graph, there is no deadlock because no cycles are present. However, since transaction 3 is behind transactions 2 and 4 in the wait queue it is implicitly waiting for them. Adding these edges to the wait-for graph above induces a cycle and hence a deadlock. To avoid such undesirable situations, transactions with locks to convert are added to the front of the wait queue. In the above situation this will ensure that

once transaction 5's IX lock on *N* is released, transaction 3 can access *N*'s lock list and there are no unnecessary deadlocks.

5.2.4 Deadlock Detector

As shown in the previous section, the Deadlock Detector is responsible for building a wait-for graph which is tightly synchronized with the Lock Table. It then periodically detects cycles in this graph to identify deadlocks. The wait-for graph is implemented as a list of transactions (*TL*) where each transaction stores its adjacency list *AL*, i.e. all the transactions it's waiting for.

5.2.4.1 Detecting cycles and transaction to abort

- Main Algorithm

while(TL has more elements)

Pick next transaction Xid

Clear VL

If Xid ∉ NoCycleList

Result = Run "Traverse Adjacencies" Algorithm on Xid

If (Result is valid transaction id)

call "Remove Transaction" Algorithm on Result

Else

continue

- **Traverse Adjacencies Algorithm (Xid)**

Add Xid to VL

Retrieve adjacency list (AL) of Xid

If(AL is empty)

Add Xid to NoCycleList

return "No cycle"

∀ elements E in AL

If(VL does not contain E)

If (E ∈ NoCycleList)

Result = "No cycle"

Else

Result = run "Traverse Adjacencies" Algorithm on E

If(Result is valid transaction id or E is last element in AL)

If (Result is "No cycle")

Add Xid to NoCycleList

return Result

Else

Return E

- **Remove Transaction Algorithm (Xid)**

  Retrieve Adjacency List (AL) of Xid

  Clear AL

  while(TL has more elements)

  Pick next transaction T from TL

  Retrieve AL of T

  If(AL contains Xid)

  Remove Xid from AL

  Remove Xid from TL

Figure 25 Deadlock Detector Algorithms

Figure 25 above shows the three algorithms used by the Deadlock Detector to identify and remove deadlocks. The *Traverse Adjacencies* algorithm is used to identify cycles in the wait-for graph. It does this by keeping a global Visited List (VL) of all transactions that it has visited and doing a recursive depth-first search on the adjacencies of the transaction passed in. If it comes across any transaction twice, we know that there is a cycle. The *Remove Transaction* algorithm removes this transaction from the wait-for graph completely to break the deadlock. The *Main* algorithm runs periodically and calls the *Traverse*

Adjacencies algorithm on each transaction until it receives a valid transaction to remove. It then calls the *Remove Transaction* algorithm on this transaction. A global *NoCycleList* is kept for optimization purposes. If we have figured out that no cycles can be found starting from a certain node, we store this transaction identifier in the *NoCycleList.* This ensures that we don't have to run the *Traverse Adjacencies* algorithm on the same node multiple times unnecessarily.

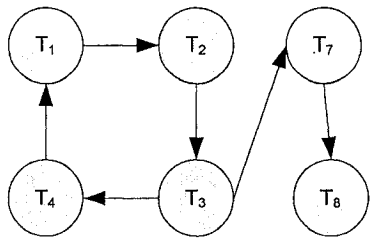

Figure 26 Deadlock Example

We will now illustrate our algorithms at work for an example. Figure 26 above shows an example of a wait-for graph with a cycle. Assume the *Main* algorithm picks transaction 1 to run *Traverse Adjacencies* on. Transaction 1 is added to the *Visited List(VL)* . Subsequently, *Traverse Adjacencies* is called recursively on transactions 2 and 3 respectively. They are also added to *VL*. Transaction 3 has two adjacencies: 4 and 7. Assume 7 is picked first to be traversed. 7 will be

added to *VL* and finally *Traverse Adjacencies* will be called on 8 causing it to be added to *VL*. Having not detected a cycle, the algorithm will backtrack calling *Traverse Adjacencies* on 4 (second adjacency of 3). This will cause 1 to be visited and therefore a cycle has been detected because 1 is already in *VL*. Transaction 1 is removed from the wait-for graph and the deadlock is broken.

5.3 Implementing Transaction Aborts

In the previous section, the transaction that is removed from the wait-for graph is aborted by LockX. We now discuss how aborts are implemented in LockX. Recall that an abort requires all operations of a transaction that have already been executed to be rolled back.

Special data abstractions have been created (one per update operation) to hold all the necessary information for an abort:

- Rename: *RNNode* holds a reference to the node *N* which has been renamed, its old name as well as the operation sequence number corresponding to this operation.
- Replace: *RPNode* holds a reference to the node that got replaced *ON*, the new node *N* and the operation sequence number.

- Delete: *DNode* contains a reference to the deleted node *N*, its old parent *p* and the operation sequence number. Since deleted nodes have a specific position in the children list of their parent nodes, we need to preserve this ordering when we re-insert *N*. For this purpose when *N* is being deleted, we take a snapshot of all the siblings after N into a special list called *AfterList*. This is necessary because any arbitrary number of the AfterList nodes might be deleted/replaced before the abort is initiated (more details follow in the algorithm).
- Insert-into/insert-before/insert-after: *ANode* stores a reference to the added node *N* and the operation sequence number.

Transactions have to hold certain state information to perform the aborts. Firstly, a list is created for each operation and the matching nodes from above are stored (*AddList* for *ANodes*, *DelList* for *DNodes* etc). In addition, we keep track of the last operation sequence number assigned (*op_seq*) for the transaction.

5.3.1 Abort Algorithm

```
For(int i=op_seq; i>1; i--){
    tempAddList = getAddList(i)
    tempDelList  = getDelList(i)
```

```
tempRnList  = getRnList(i)

tempRpList =  getRpList(i)

∀ ANodes in tempAddList:

        Retrieve N's parent

        Remove N from parent's child list

∀ DNodes in tempDelList:

        Retrieve N's AfterList, old parent p

        If(AfterList is empty)

                add N to end of p's child list

        Else{

                If(at least one node in AfterList exists in DOM tree)

                        Insert N before first node in AfterList which still exists

                Else

                        add N to end of p's child list

        }

∀ RNNodes in tempRnList:

        Retrieve N's old name

        Set N's name to old name

∀ RPNodes in tempRpList:

        Replace N with ON in N's parent's child list
```

clearTempLists()

}

Figure 27 Abort Algorithm

Figure 27 above shows the pseudocode for how transaction aborts are implemented in LockX. Based on the reasoning of section 4.4.1, we do the undo operations in the reverse order using the *op_seq* variable. From each of the operation lists (*AddList, DelList, RnList, RpList*), we extract the nodes relevant to the current operation sequence number into temporary lists. In the case of added nodes, we delete them from their parent nodes' children lists. For deleted nodes, the undo operation is slightly more complex because we need to insert them back to their previous positions. We check the *AfterList* to see which of its nodes have not been deleted/replaced from the McXML DOM tree. If *AfterList* is empty, this node *N* was at the end of *p*'s children list. We therefore insert *N* back in the same position. If at least one node in *AfterList* exists, it implies that we insert *N* before the first node in AfterList which still exists in *p*'s children list. If all the nodes in the *AfterList* have been deleted/replaced, we insert at the end of *p*'s children list. This ensures that the previous ordering of *N* is maintained. For renamed nodes, we set their names back to their old ones. Finally for replaced

nodes, we replace the current ones with the old ones. We continue this process until we have gone through all the undo operations in the reverse order.

5.4 Query Execution using LockX

Now that we have explained LockX in detail, we will describe how we have used it in the Query Execution Engine to lock nodes appropriately for read queries. Update queries work similarly.

5.4.1 Operation Modes

Since the query execution engine should acquire different kinds of locks depending on the operation, we have created three different operation modes which are explained below.

1. Read: The *Read* mode is used for read operations which acquire IS, S and RR locks.
2. IWrite: The *IWrite* mode is used on variables in an update operation relative to whose bindings the actual nodes to update are found. This mode only uses IX locks because the nodes it works on are ancestors of the nodes being updated. For example assume *$bi* is bound to *document("books.xml")/bib*, *$a* is bound to *$bi/book/*

author and *$a* is being updated. The *IWrite* mode would be applied to *$bi.*

3. D/RN/RP/II/IA/IB: An update mode is created for each of the update operations. This mode is used on variables in an update query whose binding are the nodes to be updated. Ancestors of updated nodes are IX-locked and the updated nodes themselves are locked appropriately depending on the update mode. Assume $a is being renamed from the previous example. *Parent(author),* i.e. *book*, will be IX-locked and the *author* element itself will be RN-locked.

5.4.2 Finding nodes with matching labels

The first step in query execution is to collect all nodes with matching labels from the McXML DOM tree. We will discuss the locking algorithms used for the three different styles of query execution allowed by XPath expressions.

5.4.2.1 Preorder traversal

An example of this would be *document("books.xml")//book*. The search for *book* nodes is done from the root of the books.xml DOM tree. In the McXML Query Execution Engine implementation, a preorder tree traversal is used to retrieve *book* nodes. We present in Figure 28 the locking algorithm used in the tree traversal process to lock nodes appropriately.

```
∀ nodes in tree-traversal search
    Before reading next node N, acquire IS lock
    Matching labels in query:
    If(matching label)
        Convert IS Lock to S Lock
        While(ancestor has IS lock && IS value is false)
            Set ancestor's IS value to true
    Else
        Hold IS
        Add false to N's IS value
    If(leaving Subtree(N) && N has IS lock)
        Check IS value for N
        If (false && N not path protected)
            Remove IS lock
        Else
            Keep IS lock
```

Figure 28 Preorder traversal

Figure 28 uses a boolean IS value for every node traversed by the locking algorithm while looking for a certain label. Whenever a node *N* with the correct label is found, N's IS lock is converted to S and *ancestors(N)* have their IS values set to true. A true value on a node *N* determines that the IS lock should be not be released on leaving *Subtree(N)* whereas a false value indicates the opposite. This IS value is used to ensure that *ancestors(N)* don't have their IS locks released by LockX which could expose *Subtree(N)* to conflicting concurrent operations such as *Replace* and *Delete*. A mechanism called *path protection* is used by the algorithm in Figure 28 to avoid removing locks that a transaction wants to hold. For a detailed discussion on this mechanism and its motivation, please consult section 5.4.3.

5.4.2.2 Absolute Path Search

An example of this would be *document("books.xml")/Bib/book*. Figure 29 shows the locking algorithm for absolute path searches.

∀ *node N per level*

Before reading, acquire IS lock on N

Comparing label with same level of the path expression

If(matching)

Convert IS lock to S lock

Proceed to next level

Else

If(N is not path protected)

Remove IS lock on N

If(ancestors(N) not path protected)

Release S locks on ancestors(N)

Stop searching for matches

Figure 29: Absolute Path Search

There is no need for IS locks or IS values for nodes because we are doing an explicit path search. The basic idea is to search for all nodes which match exactly this path expression. Therefore, if any stage we know that the path will not be fulfilled we release all locks and stop searching.

For example, assume we are searching for *Bib/book/size* on the tree in Figure 2. We compare the first level of the tree with *1*. Since 1 has a matching label, we convert the IS lock we had acquired to an S lock and proceed to the next level. We follow a similar procedure with 2 and 15 because they match the *book* label

and proceed to the third level of the tree. We acquire IS locks on this level and search their labels. However, we find no *size* label. We therefore release all IS locks and move up the hierarchy to release the S locks on 1,2 and 15. We immediately stop searching because the tree has no matches for this path expression.

5.4.2.3 Update Queries

Update operations use similar algorithms but different lock types. Instead of IS locks, IX locks are used before reading nodes. If a node has a label matching the XPath expression, the IX lock is either converted to a specific update lock or left as is depending on the operation mode. For the preorder traversal from the root, we use an IX value for a node similar to the IS value in Section 5.4.2.1.

5.4.3 Matching predicates

An example of a predicate could be */title="Hello World"*. Once we have retrieved the nodes with matching labels, we need to filter this list *L* to include only those nodes which match the given search criteria in the WHERE clause. Figure 30 below describes the algorithm used for this purpose.

∀ *node N found in L*

If(WHERE clause requires new path searches from N)

Find new descendant nodes SN with matching labels in WHERE clause

S lock these descendants SN

Check condition

If condition is matched

Store path from N upwards to the root

Else

Release locks from SN upto but not including N

Add N to a list of nodes to discards locks from (Discard List)

Remove N from L

∀ *node N in Discard List*

Release locks from N upwards to the root for nodes which are not path-protected.

Figure 30 Matching predicates

The algorithm in Figure 30 uses a concept of *path protection* for nodes which match both an XPath expression and a search predicate. For such nodes, we store the path leading to them from the document's root node. This means that for each node along this path, we store the node's unique identifier along with the lock type held by the transaction *T*. This is important because a transaction *T* often has multiple operations. The locking algorithms attempt to release locks immediately from nodes which don't match path or predicate criteria. However, these locks may have been needed for a previous operation and therefore need to be held till the end of the transaction according to 2PL behaviour. *Path Protection* allows transactions to remember which locks should not be removed from which nodes.

For nodes which don't match our search criteria, we remove them from our list *L* holding nodes with matching labels. We place them in a special Discard List *DL.* For each node *N* in *DL*, we then remove locks from *N* up to the root if the node's id and lock type have not been path-protected already.

For example, assume that 2 and 15 from Figure 2 matched our path expression *Bib/book* and we stored them in a list *L*. Now, we want to match the predicate *author/last="Abiteboul"* on them. First 6 and 9 are S locked but 9 does not match

the predicate so we release these locks. 2 is added to the *Discard List* and removed from *L.* Next, 19 and 22 are S locked and the predicate is checked. Since it matches, we store the path from 15 upto 1. Finally, we retrieve 2 from our *Discard List.* We remove 2's S lock and then move up to 1. However 1 has just been path protected, so we don't remove its S lock.

5.4.4 Returning Results

Based on our locking algorithms, the nodes which need to be returned by the operation have already been S locked. We therefore convert the S locks to RR locks and return the subtrees rooted at these nodes.

Chapter 6 Performance Evaluation

We now evaluate the performance of LockX by varying factors such as the structure of the tree and the proportion of read operations in transactions. We also run LockX on an auctioning application benchmark set up to simulate real-world usage conditions. Finally, we analyze how LockX performs compared to the SnaX/OptiX [13] concurrency control protocols already implemented in McXML.

6.1 Experimental Setup

For our experiments, we have used a Pentium 4 PC with a 3.4 GHz processor and 1 Gigabyte of memory running Linux. The experiments of sections 6.2 and 6.3 were carried out using synthetic XML documents generated by the XML generator located on the IBM *alpha works* website [15]. The experiments of section 6.4 were performed using documents generated with the XMark benchmark project [16].

We have four clients in the system that concurrently submit transactions so that a desired system-wide throughput is achieved. Note that this is a closed client model, i.e., a client can only start a new transaction when its previous transaction

has terminated. Each transaction has five operations which can be either queries or updates. When a transaction aborts, the client keeps resubmitting it until it is committed successfully. In the following experiments, we vary the throughput of the system and then measure the corresponding response times and abort rates of the transactions.

The abort rate is calculated as follows:

x = number of transactions aborted

y = number of transactions sent by clients

$$Abort\ Rate = x/y * 100\ (\%)$$

For example, if a transaction is aborted three times before being executed successfully by a client, it would count as four transactions with three aborted resulting in a 75% abort rate.

The response time is the average time it takes for a transaction from start to finish on the McXML server. In the above case, the response time would include the execution time for the three aborted transactions as well as the successful one.

For the following experiments, we assume that the Deadlock Detector checks for deadlocks every 250 milliseconds. We say that LockX *saturates* at a certain throughput *x* txns/sec if it cannot achieve a higher throughput based on the transactional response times of the McXML Server.

6.2 Impact of Document Structure

The first experiment evaluates how the structure of an XML document affects performance. Two synthetic XML documents were created using the XML generator of *IBM alpha works* [15]. Documents were generated by specifying three parameters; the *scaling factor* specifies the number of children of the root, the *depth* determines the maximum number of levels in the XML tree and the *fanout* specifies the average number of children of internal nodes. These three parameters collectively determine the size of the document. In our documents, except for the root (level 1), the children of the root (level 2) and the parents of the leaves (level = *depth*-1), each inner node has the same number of children. Furthermore the path length $\langle path(n) \rangle$ is the same for all leaf nodes *n*.

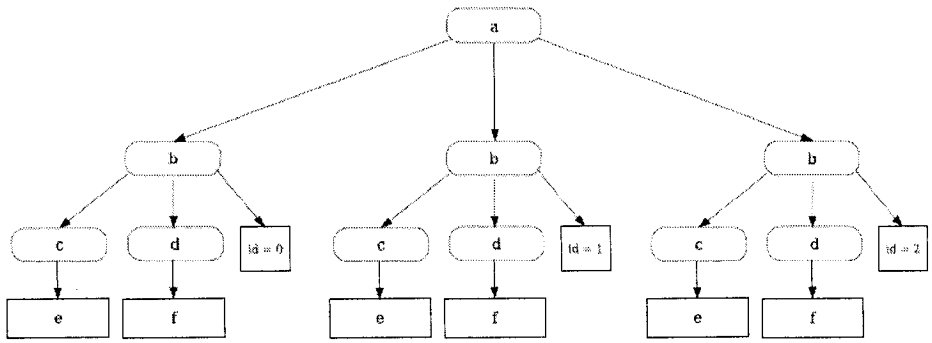

Figure 31: XML tree (scaling factor = 3, depth = 4, fanout = 2)

Figure 31 shows a sample document with scaling factor 3, depth 4 and fanout 2. The children of the root all have the same element name *b*. Each of these elements has one ID attribute. No other elements in the tree have an attribute. Each element on level *x*, where $2 \leq x \leq$ (*depth*-2) has exactly two (*fanout*) child elements. Each child element has a different name. For instance, each element with name *b* on level 2 has two children *c* and *d*. The structure of the tree makes it easy to access specific nodes since the children of the root have attributes which can be used in predicate conditions in the query. The leaves of the tree are text nodes indicated by the rectangular boxes.

	Scaling Factor	Depth	Fanout	Size
flat.xml	96	4	2	577
deep.xml	3	9	2	577

Table 11: File parameters

Table 11 shows our parameter configuration used to generate two documents; *flat.xml* has a flat wide tree while *deep.xml* has a deep narrow tree. We chose small documents to stress-test our protocols.

In this experiment, each transaction has five update operations leading to 0% reads. Each operation randomly selects one node of the document to update. Running an automatically generated workload at high submission rates is difficult when the structure of the document changes. Our benchmark therefore combines all types of update operations so that the end result of each transaction is again the same document.

Figure 32 compares LockX's response times, with increasing throughput, for *flat.xml* and *deep.xml*. LockX has consistently higher response times for *deep.xml* compared to *flat.xml*. Although similar at the start, the gap between the

response times grows larger until response times for *deep.xml* are double those for *flat.xml*. This can be attributed to the high abort rates of *deep.xml* with increasing throughput shown in Figure 33. Transaction aborts directly affect the response times because aborted transactions have to be executed from the start again.

In LockX, transactions are only aborted when a deadlock has been detected. The reason there are a high number of deadlocks for *deep.xml* in LockX is that the *scaling factor* is 3. Therefore, there are only three main paths causing a lot of conflicts between transactions acquiring locks. *fla*t.xml has a *scaling factor* of 96 resulting in many different paths for the queries to run on. *flat.xml* therefore does not have any transaction aborts causing lower response times and a higher saturation point of 32 txns/sec compared to 24 txns/sec for *deep.xml*. Note that for flat.xml with four clients experiencing a response time of around 122 ms, we cannot achieve a higher throughput than 32 txns/sec because we use a closed client model.

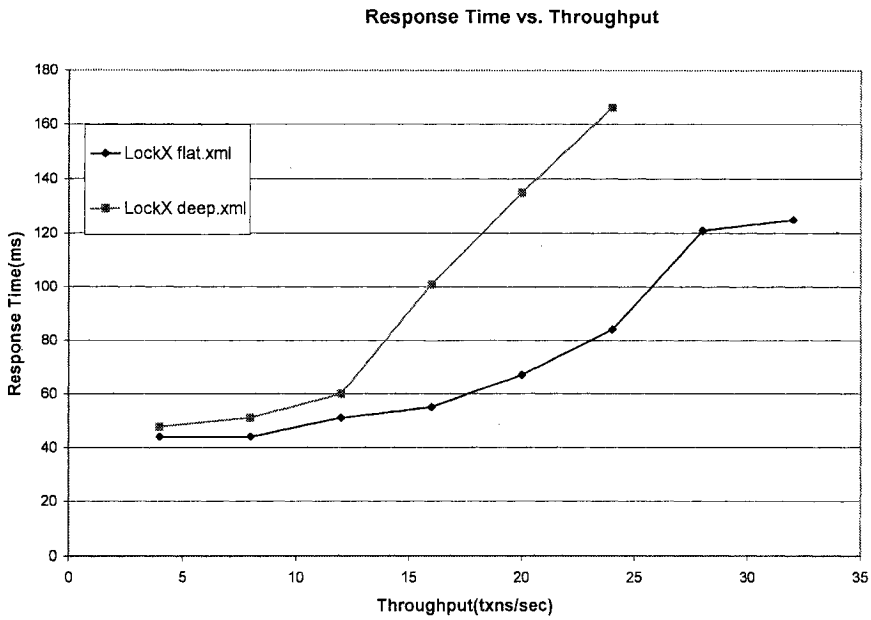

Figure 32: Response Time (Document Structure)

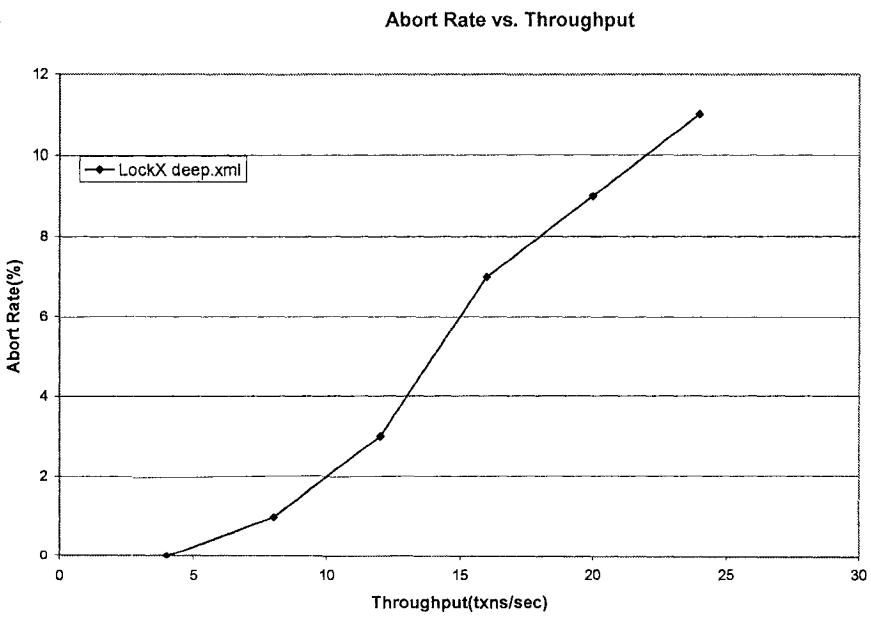

Figure 33: Abort Rate (Document Structure)

Figures 34 and 35 show again the response times for the two files with increasing throughputs. However, this time we analyze where the execution time is being spent. *Wait* is the average time spent by each transaction in wait queues waiting for locks on nodes. *Query* is the average time spent by each transaction on query processing in the McXML Query Execution Engine. *Lock* is the average time spent by each transaction dealing with locks and LockX's various algorithms.

The proportion of wait time increases from 0% to 23% and 9% to 51% for *flat.xml* and *deep.xml* respectively. The latter file has a consistently greater proportion of response time spent on waiting because of the larger number of conflicts that arise in a deep narrow tree.

deep.xml's proportion of query time drops from 31% to 19% whereas that for *flat.xml* drops from 58% to 48%. *flat.xml* has a consistently greater proportion of response time spent in *Query* state because the Query Engine has to evaluate predicate conditions on 95 child elements of the root while only 3 evaluations are needed for *deep.xml*.

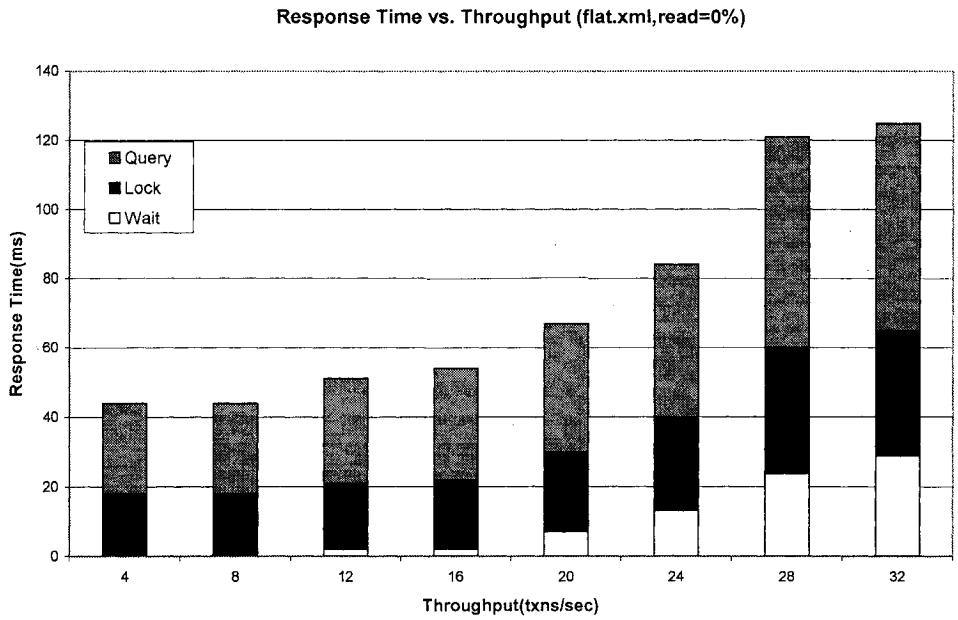

Figure 34: Response Time breakdown (flat.xml)

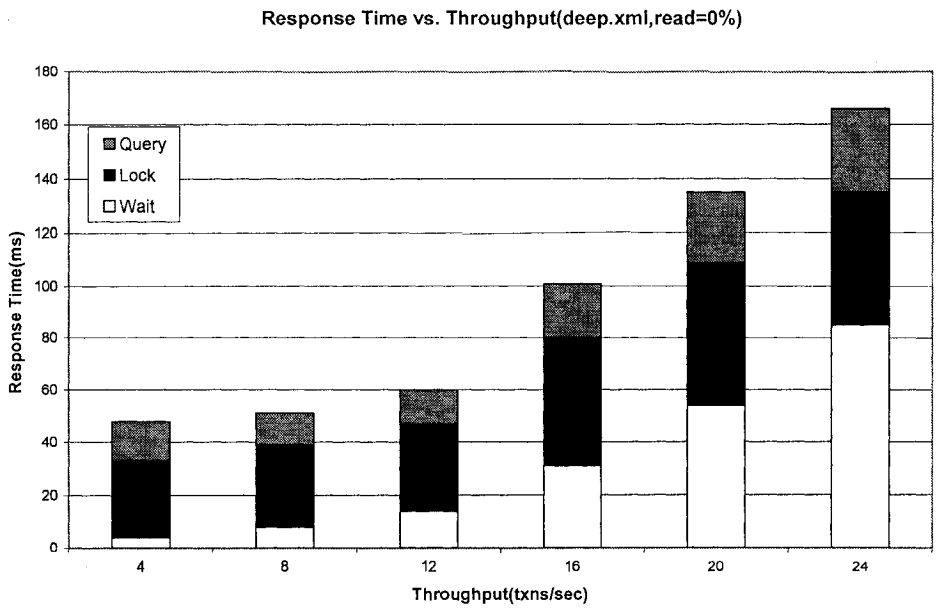

Figure 35: Response Time breakdown (deep.xml)

6.3 Impact of Read Operations

In this experiment, we analyze LockX's performance on *deep.xml* when the percentage of read and write operations are varied. Results were similar for *flat.xml* in relative terms. As in the previous section, each update operation randomly selects one node to update. Half of the read operations select a single leaf node whereas the other half choose a random level in the tree, returning one node of this level and its subtree.

Figures 36 and 37 compare LockX's performance for transactions with five operations out of which 0% or 50% are read operations. Results are similar with LockX (0% reads) having slightly higher response times and abort rates than LockX (50% reads). The gap between the two is small at the beginning but increases as we raise throughput. In the beginning, response times are similar because there are a similar number of conflicts for low throughputs.

Once again, our two performance parameters are correlated. LockX with 0% reads performs worse with respect to response times because there are more transaction aborts. Conflicts and therefore deadlocks are more likely to appear when all operations are updates because update operations are more likely to conflict with each other. We must also consider the fact that all conflicts do not

lead to deadlocks but add to the time each transaction spends in the wait queue affecting the response times. If half of our operations are reads, there will more operations which are able to execute concurrently and therefore less conflicts, less deadlocks and better performance results.

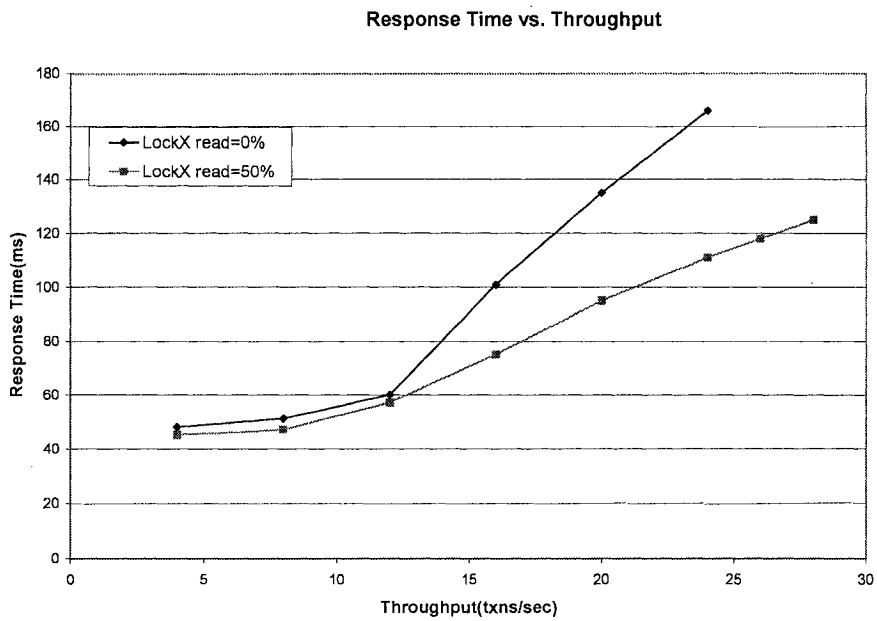

Figure 36: Response Time (Read Operations)

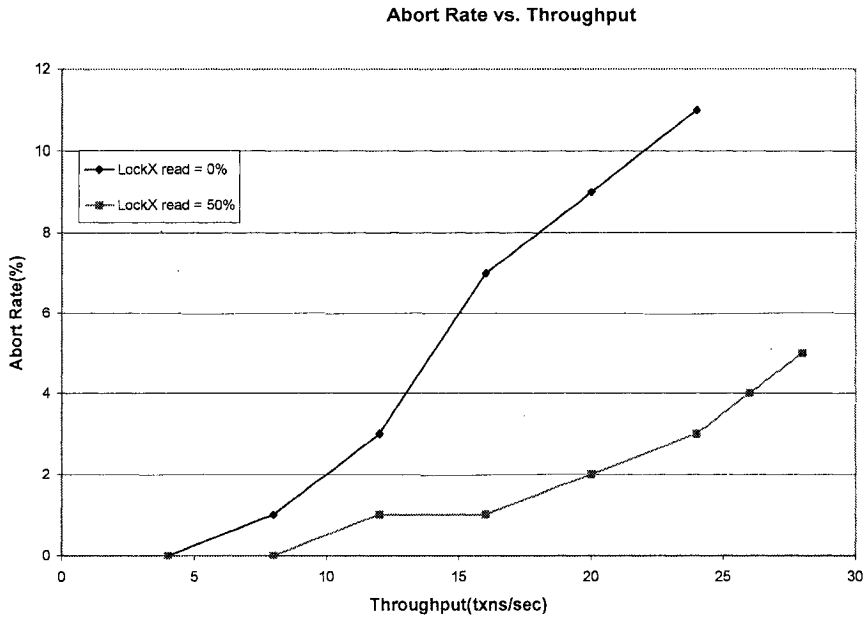

Figure 37: Abort Rate (Read Operations)

6.4 The XMark Benchmark

We now analyze the performance of LockX on an established benchmark that emulates a practical application. The XMark benchmark [16] provides XML documents and queries for an auctioning application.

6.4.1 Auction XML document

Figure 38 shows the structure of the Auction XML document we used in our experiments. Elements are shown in rounded rectangles, attributes are squares and text nodes are rectangles. The *site* root element encompasses various

information needed for an auctioning application. Firstly, information about the various users of the system is stored in *person* elements. The *open_auction* element stores a list of all open auctions in the system. Each *open_auction* element stores information about the bidder and item of interest. Finally, each auctioning item is hierarchically organized based on the continent it belongs to. For detailed schema information, please refer to work described by Schmidt et al. [16].

6.4.2 XMark Queries

In our experiments, we use the following subset of the queries proposed by XMark [16]. Sardar [13] has created corresponding query statements for each of the following queries, and we use those modified statements.

- Q1: Return the name of the person with ID 'person0' registered in North America
- Q2: Return the initial increases of all open auctions
- Q5: How many sold items cost more than 40?
- Q6: How many items are listed on all continents?
- Q7: How many pieces of prose are in our database?
- Q8: List the names of persons and the number of items they bought. Joins person, closed_auction.

- Q13: List the names of items registered in Australia along with their descriptions.
- Q15: Print the keywords in emphasis in annotations of closed auctions.
- Q18: Convert the currency of the reserve of all open auctions to another currency.
- Q19: Give an alphabetically ordered list of all items along with their location.
- Q20: Group customers by their income and output the cardinality of each group.
- Q_N: This query is not part of the original XMark [16] benchmark but we felt this would be an interesting query to test the system. It lists all items that belong to a certain category.

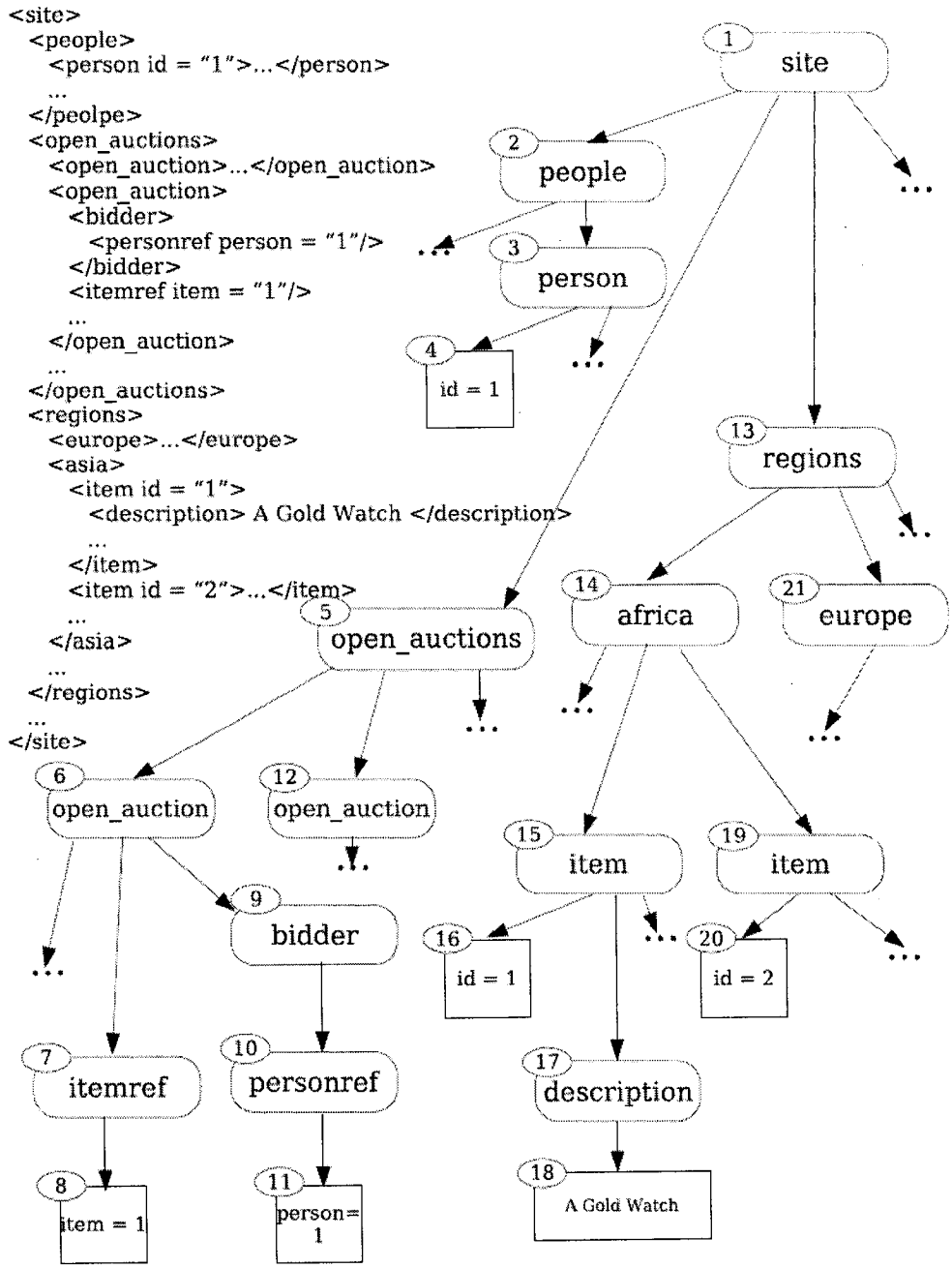

Figure 38: Auction XML Document

6.4.3 Update Operations

XMark does not contain any update operations. Therefore, Sardar [13] has proposed a set of suitable update operations as an extension to XMark. Using the XQuery update extensions [3], Sardar [13] has created corresponding update statements for each of the following:

- U1: *Create a new Person.* This update operation adds a new *person* to the XML document as a child of *site/people/.*

  Subtree(person) will contain important information about a *person* such as his/her name, e-mail, address, age and categories he/she is interested in.

- U2: *Create a new Item.* In this update, we create a new *item* and create an *open_auction* for that *item* because it is assumed that the *item* is being created to put on auction. The item is created as a child of *site/regions/(any of the six regions)/.* It contains important information about the *item* such as its name, category it belongs to, location, shipping method, description, quantity and a reference to the owner. As part of the same operation, a new *open_auction* is created for this *item* as a child of *site/open_auctions/.* The new *open_auction* has a reference to the *item* and a reference to the *seller* of the *item*, the initial price, the reserved price and other time relevant information like the closing time of bidding on this *item.*

- U3: *Bid on an item.* A *person* can bid on an *item* that is still open for auction, i.e., has an *open_auction* that refers to it. When a *person* bids on an *item*, information such as bidding time, increase in price and reference to bidder are added to the *open_auction* corresponding to that *item.* The new price of the *open_auction* is updated and a reference to this *open_auction* is also added to the *person* who bid on the *item.*
- U4: *Close an open auction.* This operation simulates the closing of an *open_auction.* When the closing time of an *open_auction* comes it is removed from the open_auctions subtree and is placed in the *closed_auctions* subtree. A *person* can no longer bid on this *item.*

6.4.4 Evaluation

We now compare the performance of LockX for two different sizes of the Auction document. XMark allows its users to create documents of different sizes by specifying a factor *f.* We generated documents of sizes 557KB (*f = 0.005*) and 207KB (*f*=0.002). Since in an auctioning site, the number of queries is usually more than the number of updates, we use transactions with 5 operations where 75% are queries. We give each update operation a different probability to be called considering that some operations are more likely to occur than others (e.g.

bidding is more likely than creating a new item). Table 12 illustrates the likelihood of each operation being chosen by a transaction.

Q1	Q2	Q5	Q6	Q7	Q8	Q13	Q15	Q18	Q19	Q20	Q_N	U1	U2	U3	U4
6.25	6.25	6.25	6.25	6.25	6.25	6.25	6.25	6.25	6.25	6.25	6.25	1.25	2.5	18.75	2.5

Table 12: Probability (%) of occurrence of each operation

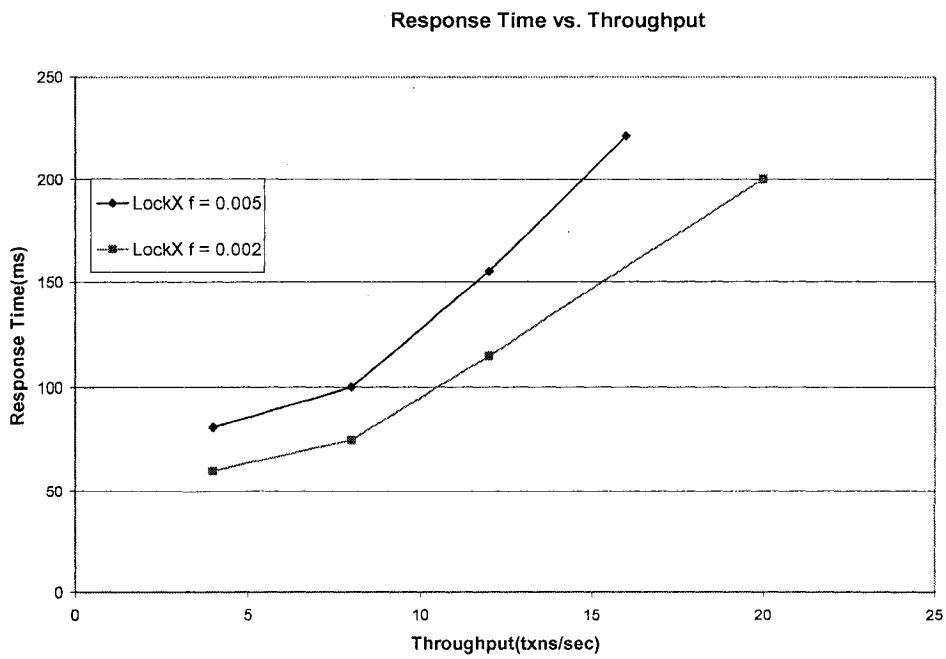

Figure 39: Response Time (XMark)

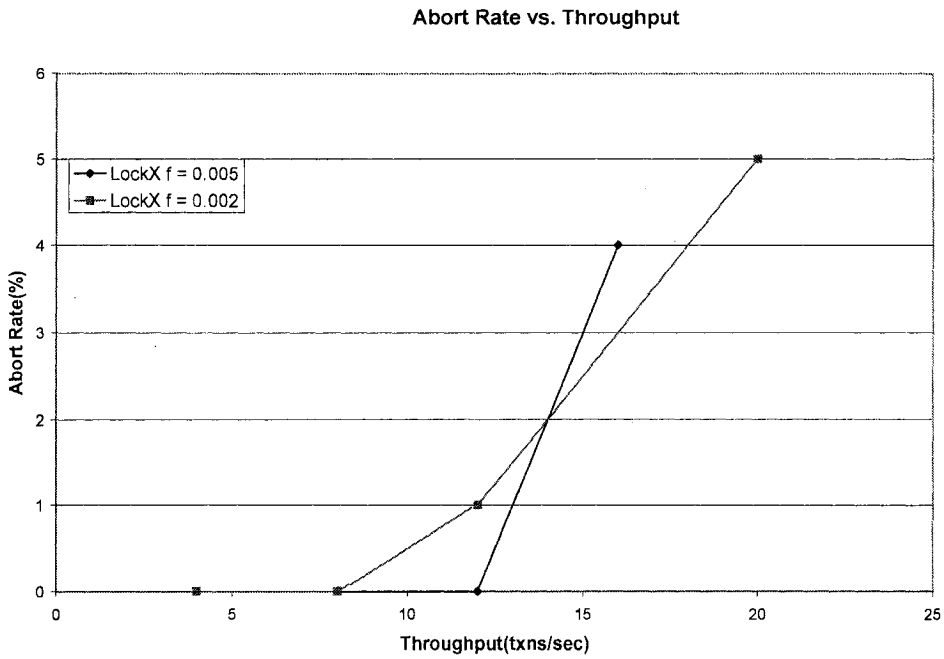

Figure 40: Abort Rate (XMark)

Figures 39 and 40 show the response times and abort rates for LockX with increasing throughput on two different document sizes. The response times for both document sizes are similar with $f = 0.005$ having slightly higher times. This higher response time for the larger document is because more time will be spent traversing and acquiring locks on the larger tree. There could also be greater waiting times because each query is potentially waiting for conflicting locks to be released on more nodes. The abort rates are low and comparable to each other. They are low because 75% of the operations are reads and therefore there is a

lower probability of conflicts between transactions compared to previous experiments.

Comparing LockX from this experiment with results for deep.xml (read=0.5), we see that LockX on the Auction document has worse response times for the same throughput range (4-20 transactions/sec) even though these experiments had extremely low abort rates. We feel that this is because the structure of Auction.xml is more complex making query execution a more expensive process. In addition, read queries (75% of our operations on the Auction document) are more complex than update queries because our update queries are generally based on simpler path and predicate searches.

Here are examples of read and update queries run on the Auction XML document:

- FOR $a in document("auction.xml")/site, $c in $a/people/person, $d in $c/watches
  WHERE $c/@id = " person35"
  UPDATE $d { INSERT <watch open_auction="open_auction20"></watch>
  }

- FOR $p in document("auction.xml")/site

  RETURN $p//description, $p//annotation, $p//email

Whereas the update query uses a simple absolute path and predicate search to identify the right *watches* element to insert into, the read query retrieves all *description*, *annotation* and *email* elements anywhere below the root *site* element. The latter query is clearly the more expensive operation not only in terms of query execution but also in terms of wait time because each node has to be locked before being read. Such read queries with expensive preorder tree traversals are used more often in this experiment than in the previous ones.

6.5 LockX vs. SnaX/OptiX

In this section, we compare the performance of our locking-based protocol LockX with the two snapshot-based protocols SnaX and OptiX [13]. Although LockX generally performs worse than SnaX and OptiX, the degree of performance difference varied based on the experiment type. We highlight three cases; one where LockX performs significantly worse and two where its performance closely approaches the other two.

6.5.1 Flat (Worse)

Figure 41 compares the response times of LockX with those for OptiX and SnaX on *flat.xml.* Response times for LockX are on average twice as high as those for the snapshot-based protocols. With a scaling factor of 96, there are many different paths in this tree causing few conflicts for LockX resulting in low response times. However there is still the overhead of locking nodes for reads and waiting for conflicting locks to be released which SnaX and OptiX don't have. The difference between the response times of LockX and SnaX/OptiX is less than a factor of two in the beginning but increases to a maximum factor of almost four. This is because as the throughput increases, the probability that different transactions are operating on the same paths increases causing more conflicts for LockX.

LockX has no aborts whereas SnaX and OptiX reach a maximum abort rate of 20% at a throughput of 40 transactions/second as shown in Figure 42. SnaX/OptiX have higher abort rates because transactions perform operations on their snapshots of the XML tree leaving the validation till later at commit time.

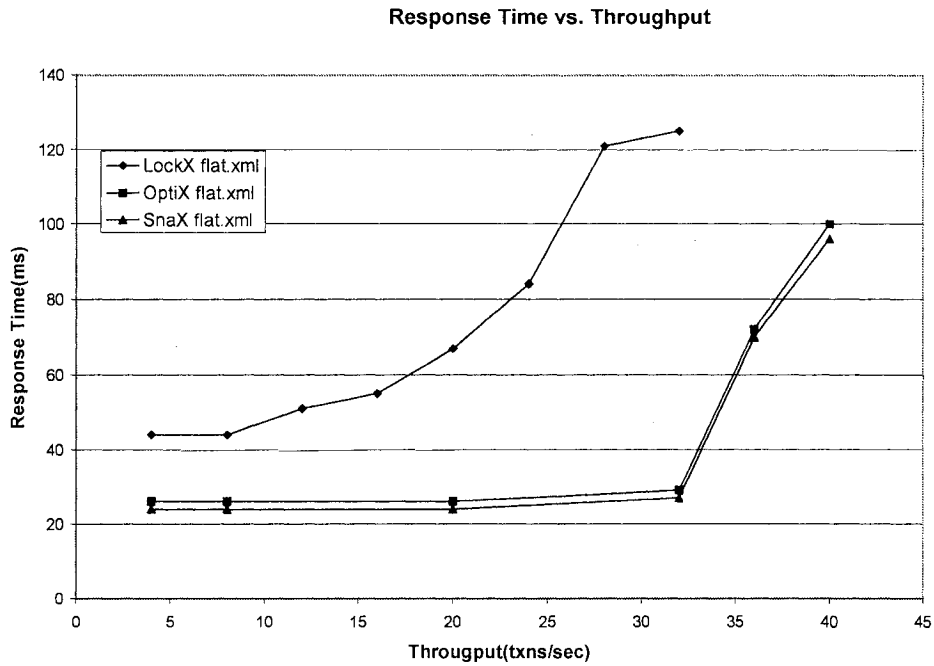

Figure 41: Response Time

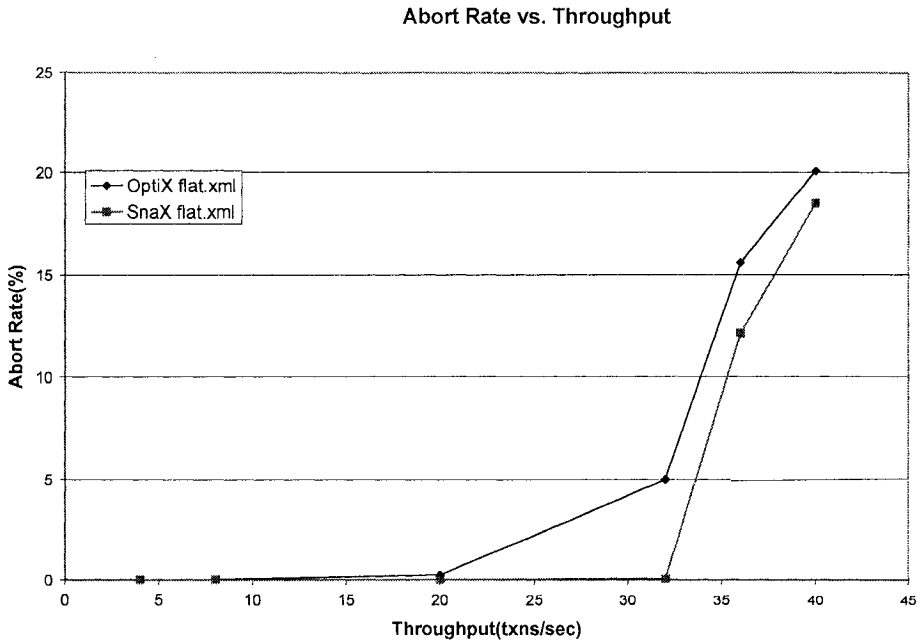

Figure 42: Abort Rate

6.5.2 XMark Benchmark (Worse)

We now compare LockX's performance with SnaX and OptiX on the Auction document with f = 0.002. Figure 43 compares the response times of the three with increasing throughput. Results are similar to the last section with LockX performing worse in response times by an average factor of 2. SnaX performs the best because it does not consider read-write conflicts and there is a low probability of write-write conflicts because of the large proportion of read operations. OptiX performs second best because it also does not have to apply read locks but it considers read-write conflicts. LockX's response times are slightly higher than those of the other two because read locks are compatible so the time spent by a transaction waiting is only because of read/write lock incompatibilities. These are less likely because the Auction tree has a moderate number of distinct paths. Because of the higher response times, LockX saturates at a throughput of 20 transactions/sec compared to 32 transactions/sec for OptiX/SnaX.

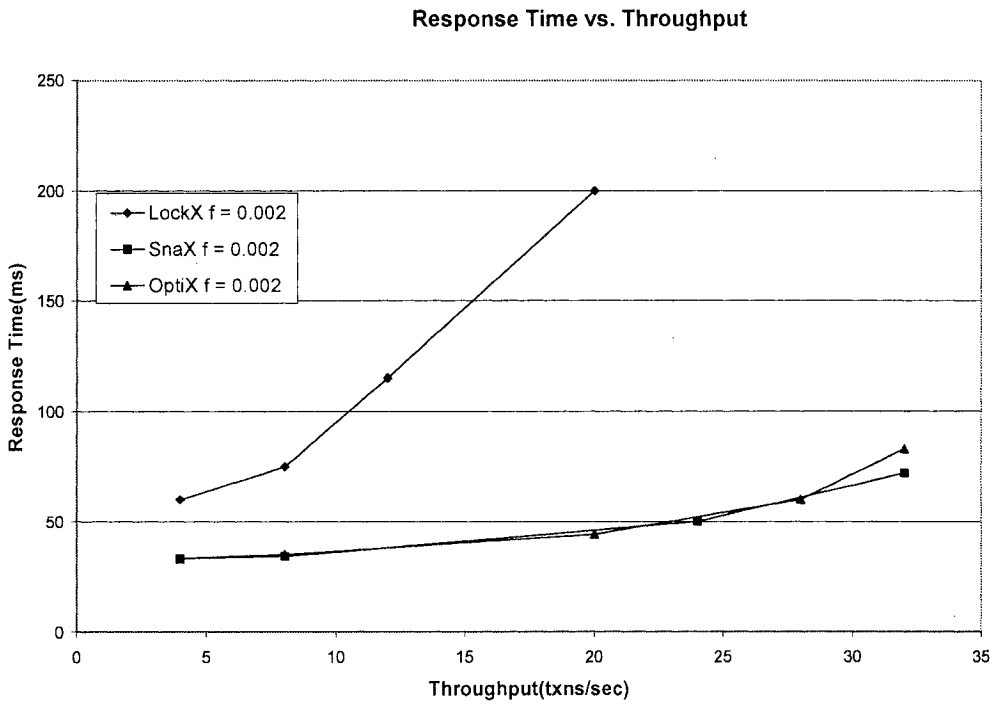

Figure 43: Response Time

6.5.3 Deep (Significantly Worse)

Figure 44 shows response times of LockX, OptiX and SnaX with increasing throughput on deep.xml with a read probability of 0%. LockX performs worse by an average factor of nine over its throughput range of 4-24 transactions/sec. The gap between LockX and SnaX/OptiX response times widens from a factor of five to a maximum of sixteen with increasing throughput.

We believe that SnaX and OptiX perform significantly better because of the snapshot-based reading mechanism and the special structure of the tree. Whereas SnaX and OptiX read the latest committed version of a node, LockX has to acquire a lock before reading any node. For read operations which do preorder tree traversals on a deep narrow tree, this can create a lot of conflicts with locks from write operations because there are only three main paths. Therefore waiting times can be a significant overhead for LockX causing the relatively high response times. OptiX/SnaX consequently saturate at a much higher throughput of 72 transactions/sec compared to 24 transactions/sec for LockX.

As described earlier in Section 6.2, there is a high likelihood for deadlocks using LockX on *deep.xml*. During the tree traversal for path/predicate searches, LockX acquires and releases many short locks. These locks are for nodes which don't match our search criteria; we don't follow a strict two-phase locking procedure for them. LockX can get into many artificial deadlocks while trying to acquire these short locks for deep trees. We believe this is the reason for LockX's higher abort rates for the same throughput range as shown in Figure 45.

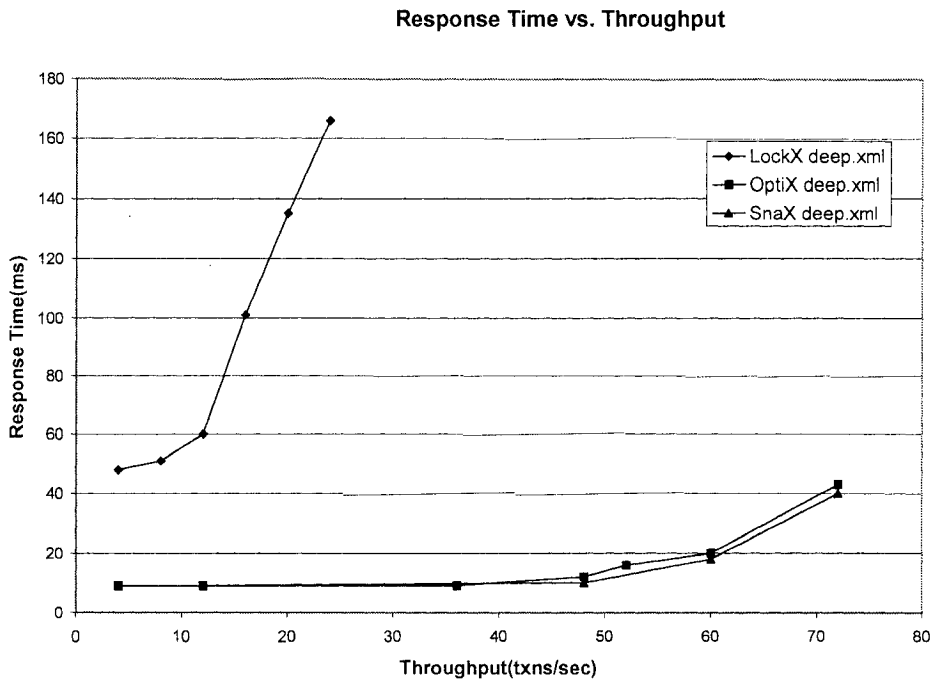

Figure 44: Response Time

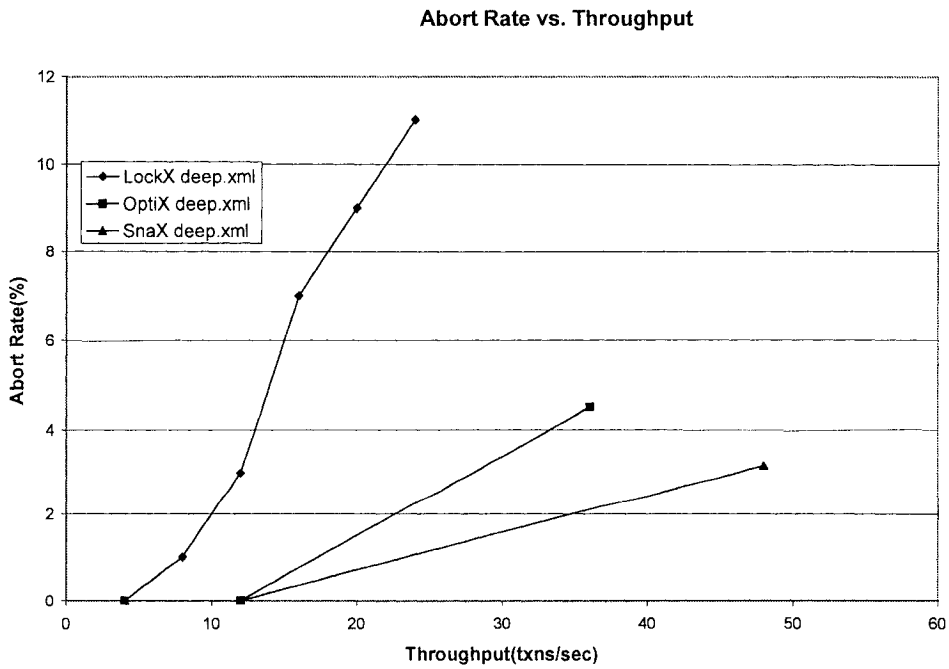

Figure 45: Abort Rate

6.6 Contribution

As expected both snapshot-based protocols (SnaX, OptiX) performed better than LockX because they do not require read locks but instead provide transactions a committed snapshot of the nodes. However, LockX only has approximately double the response times for real-life XML documents, such as the Auction document in the XMark Benchmark [16], even though it has considerable execution overhead in wait times. For flat documents, response times for LockX are also double those for SnaX/OptiX. However, LockX has no transaction aborts whereas abort rates for OptiX/SnaX go as high as 20%. LockX has nine times higher response times than SnaX/OptiX in deep narrow trees because of the large number of conflicts among transactions operating along the same paths including artificial ones on non-suitable nodes.

We feel that our contribution to the native XML database community is valid for the following reasons:

- For real-life XML documents, such as the Auction document, response times approach closely those of the snapshot-based protocols which have no locking/waiting overhead. Our abort rates are generally lower than those for SnaX/OptiX which is an important transactional concern for users.

- LockX can easily be implemented into the query execution engine of most available native XML database systems. On the other hand, OptiX and SnaX require a complete multi-version infrastructure to be implemented beforehand for the XML tree model.
- The current limitations of LockX (avoiding phantoms and enabling serializability) are open problems even in the context of relational database systems.

Chapter 7 Conclusion

In this thesis, we have presented a locking-based concurrency control protocol called LockX. LockX was designed to take the semantics of McXML's read and write operations into account to maximize concurrency. It is pluggable into any native XML database which uses a tree model for representing XML data. LockX is easily tuneable to handle different lock types and compatibility information. Deadlocks can easily be detected and removed based on cycles in the wait-for graph.

A performance analysis is done to judge the impact of document structure and read operations on LockX's response times and abort rates. LockX generally performs better on flatter trees. The higher the proportion of read operations in a workload, the lower the response times and abort rates because there are less conflicts between transactions. For flat files, the majority of the response time is spent by the Query Execution Engine to identify the required nodes. For deep narrow files at high throughputs, the majority of the response time is spent by transactions waiting to acquire locks on nodes. The locking overhead can become significant. Finally, lock contention and deadlocks are more likely on deep trees than bushy ones. LockX performs worse than snapshot-based

protocols in terms of response times but has generally lower abort rates and can be used with the existing XML database system in place.

McXML is still in its early stages and there is a lot of room for improvement. Some of the areas where future work can be done are:

- LockX: LockX has to acquire a lot of short locks on nodes which don't match the search criteria. If our locking mechanism was more selective in path/predicate searches, it would improve performance results.
- Query Execution Engine: McXML has a slow query execution engine. It can be optimized a lot to reduce the number of stages involved in executing a query successfully. An index manager can also be implemented.
- Labelling: A labelling mechanism should be implemented to ease the identification of ancestor/descendent relationships among the nodes.
- XQuery: McXML only supports a small subset of XQuery at the moment. To make it more powerful, the implementation of XQuery can be made more comprehensive.
- Page Unloading: McXML loads XML documents lazily i.e. a page is only loaded from disk when the subtree on it needs to be traversed. Similarly,

there should be unloading of unneeded pages so complete trees are not kept unnecessarily in main memory.

- Databases presumably run for long periods of time and the issue of Xids and Oids being exhausted needs to be addressed by McXML.

LIST OF REFERENCES

[1] A.B. Chaudhri, A. Rashid, and R. Zicari. *XML Data Management. Native XML and XML-Enabled Database Systems.* Addison-Wesley, first edition, 2003.

[2] J. Wu. Updating and indexing XML data. Master's thesis, McGill University, Canada, 2004.

[3] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D.S. Weld. Updating XML. In *SIGMOD International Conference on Management of data*, 413-424, 2001.

[4] R. Suchak. A page based storage manager for a native XML database. Master's thesis, McGill University, Canada, 2005.

[5] T. Fiebig, S. Helmer, C.C. Kanne, G. Moerkotte, J. Neumann, R. Schiele, and T. Westmann. Anatomy of a native XML base management system. In *International Journal on Very Large Data Bases (VLDB)*, 11(4):292-314, 2002.

[6] World Wide Web Consortium (W3C).

http://www.w3c.org

[7] DB2 Universal Database.

http://publib.boulder.ibm.com/infocenter/db2luw/v8/index.jsp?topic=/com.ibm.db2.udb.doc/admin/c0005267.htm

[8] S. Dekeyser, J. Hidders, and J. Paradaens. A transaction model for XML databases. *World Wide Web (W3C)*, 7(1):29-57, 2004

[9] J. Gray, R. A. Lorie, and G.R. Putzolu. Granularity of locks and degrees of consistency in a shared data base. In *International Conference of Very Large Data Bases (VLDB 1975)*, 428-451, 1975.

[10] M. P. Haustein, T. Härder . A Lock Manager for Collaborative Processing of Natively Stored XML Documents. In *Brazilian Symposium on Databases (SBBD)*, 230-244, 2004.

[11] S. Helmer, C.C. Kanne, and G. Moerkotte. Lock-based Protocols for Cooperation on XML Documents. In *International Conference on Database and Expert Systems Applications (DEXA)*, 230-234, 2003.

[12] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semi structured Databases. In *International Conference on Very Large Data Bases (VLDB 1997)*, 436-445, 1997.

[13] Z.M. Sardar. Snapshot based Concurrency Control Protocols for XML. Master's thesis, McGill University, Canada, 2005.

[14] H. Berenson, P.A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P.E.O'Neil A Critique of ANSI SQL isolation levels. In *SIGMOD International Conference on Management of Data*, 1-10, 1995.

[15] A. L. Diaz and D. Lovell. XML generator, 1999.
http://www.alphaworks.ibm.com/tech/xmlgenerator

[16] A. Schmidt, F. Waas, and M. Kersten. XMark: A benchmark for XML data management. In *International Conference on Very Large Data Bases (VLDB 2002)*, 974-985, 2002.

[17] Liam Quin: Extensible Markup Language (XML) 1996-2003.
http://www.w3.org/XML

[18] A. Berglund, S. Boag, D. Chamberlin, M.F. Fernández, M. Kay, J. Robie and J. Siméon. XML Path Language (XPath) version 2.0. World Wide Web Consortium (W3C) Recommendation, June 2006.

[19] S. Boag, D. Chamberlin, M. F. Férnandez, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML Query Language. World Wide Web Consortium (W3C) Recommendation, June 2006.

[20] S. Abiteboul, D. Quass, J. McHugh, J. Widom and J. L. Wiener. The Lorel query language for semistructured data. In *International Journal on Digital Libraries,* 1(1): 68-88, April 1997.

[21] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L.V.S. Lakshmanan, A. Nierman, S. Paparizos, J.M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. TIMBER: A native XML database. In *International Journal on Very Large Data Bases (VLDB)*, 11(4):274-291, 2002.

[22] T. Grabs, K. Bohm and H-J. Schek. XMLTM: Efficient Transaction Management for XML documents. In *International Conference on Information and Knowledge Management (CIKM)*, 142-152, 2002.