Isolation in XML Bases

S. Helmer, C.-C. Kanne, G. Moerkotte[*]

Fakultät für Mathematik und Informatik
D7, 27
Universität Mannheim
68131 Mannheim
Germany
phone: +49 621 181 2582
[helmer|cc|moerkotte]@informatik.uni-mannheim.de

Abstract

The eXtensible Markup Language (XML) is well accepted in many different application areas. As a consequence, there is an increasing need for persistently storing XML documents. As soon as many users and applications work concurrently on the same collection of XML documents — i.e. an XML base — isolating accesses and modifications of different transactions becomes an important issue.

We discuss six different core protocols for synchronizing access to and modifications of XML document collections. These core protocols synchronize structure traversals and modifications. They are meant to be integrated into a native XML base management System (XBMS). Four of the six core protocols are based on two phase locking, one uses time stamps, and the last one uses a novel *dynamic commit-ordering* approach. The latter two protocols achieve a higher degree of concurrency by a novel implicit representation of multiple versions. We also discuss extensions of these core protocols to full-fledged protocols. Further, we show how the two phase locking based protocols can achieve a higher degree of concurrency by exploiting the semantics expressed in Document Type Definitions (DTDs).

1 Introduction

The eXtentensible Markup Language (XML [4]) is increasingly used in areas as diverse as commercial, engineering, financial, medical, and scientific applications. Just to cite a single milestone of this process consider the XML documents containing the annotated Drosophila Melanogaster genome[1].

The rapid proliferation of XML in many different application areas results in a rapidly growing number of XML documents. It is our hypothesis that sooner or later users will work concurrently on collections of XML documents with general purpose applications like XML editors and stylesheet processors as well as with specialized tools tailored to the needs of specific application areas. Most tools of this kind work

[1]www.fruitfly.org

on the XML documents using a standardized application programming interface (e.g. the Document Object Model (DOM) [6]). If our hypothesis holds, isolating different concurrent applications becomes an important issue.

There are essentially three possibilities of storing XML documents. The first alternative is to use a file system which — from an isolation point of view — is a bad choice. The second alternative is to use an existing relational, object-oriented, or object-relational database system [3, 5, 7, 12, 16, 18, 19]. Section 2.2 shows that this is not a good idea either. The third alternative is to implement a new full-fledged XML base management system (XBMS) [8, 10, 11]. There are many reasons for us to follow the XBMS approach. One reason is that the XBMS approach allows to incorporate synchronization protocols specifically adapted to the manipulation of XML document collections.

The development of synchronization protocols to isolate different applications has a long standing and successful history in the database community. Most of the protocols that guarantee serializability already found their way into textbooks more than a decade ago [2, 9, 14]. During the last decade some researchers concentrated on defining notions weaker than serializability and developed protocols that allow cooperation between users. For a recent survey on cooperating transactions and synchronization in general see [15].

Although cooperation will play a major role in the XML context, we start with the development of protocols that guarantee serializability. The reason is that in any case serializability is still the foundation for protocols that allow cooperation. There is always a lowest level where actions have to be atomic and have to be isolated carefully.

The paper is organized as follows. In Section 2 we briefly describe the main operations of the DOM interface specification as well as the subset of operations we will consider for our core protocols. Section 2 also illuminates why storing XML documents in a relational system is a bad idea from an isolation point of view. Section 3 discusses six different core protocols. The first four core protocols are based on strict two phase locking and differ only in their locking granularity. Two of these core protocols use mechanisms developed for synchronizing ADTs [13, 17, 1]. The last two protocols are based on timestamp ordering and a novel *dynamic commit ordering* approach. In order to increase the level of concurrency, both of these protocols exploit a novel implicit representation of different versions of documents. Section 4 discusses extensions to the core protocols necessary to support the full DOM interface. This section also shows how knowledge about the DTD of a document can be exploited to achieve a higher level of concurrency for two phase locking based protocols. Section 5 concludes the paper.

2 Problem and Motivation

2.1 Document Traversal and Modification

The underlying hypothesis of this paper is that many applications will work with XML documents using a standard API like DOM [6]. Such an API — DOM or otherwise — has to provide several operations for traversing a document, modifying the text in the document, modifying the document's structure etc. Figure 1 contains some operations

observer	structure	firstChild
		lastChild
		previousSibling
		nextSibling
		getNodeById
		getElementByTagName
	contents	getTextContents
		nodeName
		getAttribute
mutator	structure	insertBefore
		replaceChild
		removeChild
		appendChild
	contents	appendData
		deleteData
		insertData
		replaceData
		setAttribute

Figure 1: Some DOM Operations

specified in DOM. The problem investigated is how to isolate different applications working on the same set of documents using DOM or a similar API.

The operations of a typical API for XML documents fall into four categories: mutators and observers of the contents of a node and mutators and observers of the structure of a document. The latter are usually called traversal operations. Since we believe that modifying the string contents of a node can be handled by standard synchronization protocols, we concentrate first on isolating document structure traversals and modifications. This will yield core protocols. In Section 4 we extend these core protocols to also isolate content reads and modifications as well as retrieval of nodes by ID/IDREF attributes. (With an attribute of type ID, an identifier can be given to a node that is unique among all identifiers contained in the document the node belongs to. An attribute of type IDREF allows to point to a single node with a given ID. The IDREFS attribute allows to point to several nodes by giving a list of IDs. For further details see [4]).

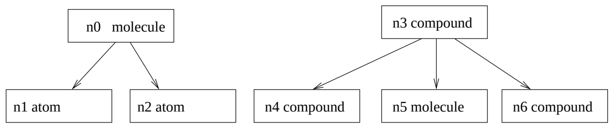
In order not to overburden the discussion, we work with a small representative set of operations a transaction can execute. We assume that a transaction first selects a document to work on. This is done via a *select document* (**sd**) operation. The result is a reference to the root node of the selected document. From there on it traverses and modifies the document structure using a sequence of the following operations:

nthP retrieves the n-th child in the child list

nthM retrieves the n-th child counting from the end of the child list backwards

insA inserts a new node after a given node

insB inserts a new node before a given node



EDGES			
id	ordinal	name	parent
n0	1	molecule	NULL
n1	1	atom	n0
n2	2	atom	n0
n3	1	compound	NULL
n4	1	compound	n3
n5	2	molecule	n3
n6	3	compound	n3

Figure 2: Two document trees and their relational representation

del deletes a given node

The distinction between attribute and element nodes is not important for synchronization purposes. We therefore talk about nodes only.

2.2 Why not relational databases

Let us briefly illustrate why storing XML documents in a relational database is not a convincing idea if isolation is required. Figure 2 shows two simple XML documents and a simplified relational representation. Assume a node is added to one of the documents. If serializability is required, the multiple granularity locking protocol which is employed by most relational DBMSs requires the inserting transaction to hold an exclusive lock on the whole table. The lock on the whole table is required to guarantee serializability which could be endangered by phantoms. As a consequence, no other document can be accessed by any other transaction. The same holds for deleting a node. In any case, *no* concurrency is possible.

Of course different translation schemes for XML documents to relations [7, 12, 16, 18, 19] lead to different consequences for synchronization. But as long as elements from two different documents can share a relation — as is the case in all translation schemes except for those using CLOBs only — the above problem remains. If each document is stored in a Character Large Object (CLOB), then locking may take place at the document level. From an isolation point of view this might not always be the best solution.

3 Protocols

This section introduces the core protocols to synchronize structure traversals and modifications of XML documents. We start by introducing several protocols based on two phase locking. Next, we introduce a core protocol based on timestamp ordering. Both protocols are well-known and already found their way into several textbooks, e.g. [2, 14]. A novel feature of our variant of the timestamp ordering protocol is that

	T	M
T	+	-
M	-	-

(a)

	TL	TR	TA	TZ	ML	MR	MA	MZ
TL	+	+	+	+	-	+	+	+
TR	+	+	+	+	+	-	+	+
TA	+	+	+	+	+	+	-	+
TZ	+	+	+	+	+	+	+	-
ML	-	+	+	+	-	+	+	+
MR	+	-	+	+	+	-	+	+
MA	+	+	-	+	+	+	-	+
MZ	+	+	+	-	+	+	+	-

(b)

Figure 3: Compatibility matrices

it is based on an implicit representation of multiple versions. In timestamp ordering every transaction receives a timestamp and transactions must commit in time stamp order. The latter requirement fits perfectly into our versioning scheme since we must eliminate older versions first. However, restricting commits to timestamp order imposes unnecessary constraints. Therefore we introduce a new protocol called *dynamic commit ordering*. Here, the commit order of transactions is determined dynamically based on the observed conflicts. As for our timestamp ordering variant, the basis for the protocol is an implicit representation of multiple versions.

It is important to note that all core protocols require that document access starts at the root node and traverses documents top down. This requirement is relaxed in Section 4.

3.1 2PL-based protocols

Lock Modes In standard two phase locking protocols for synchronizing read and write operations, there are two kinds of locks: shared locks (S) and exclusive (X) locks. Read operations require a shared lock while write operations require an exclusive lock. Since we are not yet concerned with modifying the contents of the strings contained in the document but only with structure traversal and modification, we introduce a shared lock named T that has to be acquired for traversing document structure and an exclusive lock named M that has to be acquired for modifying document structure. We introduce these locks since we will later still need S and X locks (see Section 4).

Compatibility Matrix The compatibility matrix of these two locks is analogous to the one for S and X locks (see Figure 3 (a)). For all two phase locking protocols, the standard rules have to be obeyed. Before performing an operation, the corresponding lock has to be acquired, during lock acquisition a check for conflicting locks is performed, if a conflict exists the lock requiring transaction is blocked, and locks are held till the end of the transaction. If a transaction is blocked, the wait graph is updated and if it contains a cycle, the transaction that completes the cycle is aborted.

Doc2PL Our first four protocols only differ in their granularity of locking. The first and simplest protocol *Doc2PL* locks at the document level. For applications where transactions work on different documents, e.g. one author edits one document, this easy to implement low-overhead protocol suffices.

Conceptual Document Model The next protocols lock at the node level. In order to understand these protocols and their differences, one can think of an XML document consisting of nodes with pointers connecting them. Figure 4 shows a parent node and its child nodes together with the pointers. Of course the XBMS does not have to represent documents with these pointers. For example, one could have embedded child nodes (as in Natix [11]) or an array of pointers to all children. We use the pointer model only to explain the protocols and to derive lock names. The protocols themselves are independent of the actual representation of the XML document structure.

Node2PL and NO2PL Figure 4 also shows on which items the different protocols acquire locks. The Node2PL protocol acquires locks for parent nodes. If, for example. we traverse to the nth-child of a given node P, then node P is locked in T mode. If we insert a child under node P, then node P is locked in M mode.

The protocol NO2PL acquires locks for all nodes whose pointers are — at least conceptually — traversed or modified. Refer again to Figure 4. If we introduce for example a new child $C0$ before child $C1$, then we have to acquire two exclusive locks: one for the parent node P, since its first child pointer is modified and one for the child node $C1$ since its left sibling pointer is modified. However, we do not have to acquire a lock for child $C0$, since no other transaction will be able to traverse to this node, since all ways to it are blocked: $C0$ cannot be reached from the parent node by neither an **nthP** operation nor an **nthM** operation since P and $C1$ are locked exclusively.

OO2PL Whereas in Node2PL and NO2PL we lock nodes, OO2PL locks pointers. Since there are four pointers for every node (first child (A), last child (Z), left sibling (L) and right sibling(R)), we need four shared locks and four exclusive locks. The locks are *TA*, *TZ*, *TL*, *TR*, *MA*, *MZ*, *ML*, *MR* corresponding to the above order. The compatibility matrix is shown in Figure 3(b). Again, before executing an operation, locks have to be acquired according to the pointers (conceptually) traversed or modified. OO2PL can be seen as an application of the framework for synchronizing abstract data types [17].

Overhead Let us briefly consider the number of locks to be maintained by the different protocols. Doc2PL has the fewest number of locks: at most one lock per transaction per document. In Node2PL and NO2PL we have at most one lock per transaction per node. The difference is that at the leaf level of the documents (where the most nodes are), Node2PL never acquires any locks. However, NO2PL does acquire locks for leaf nodes. OO2PL acquires at most four locks per transaction per node and hence at most four times as many locks as NO2PL.

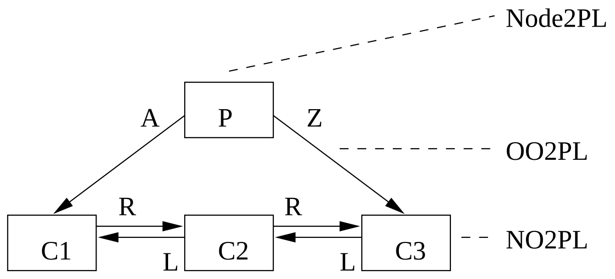


Figure 4: Conceptual list representation of XML documents

Example: Deletion The following two examples illustrate the higher degree of concurrency allowed by OO2PL compared to the other two phase locking based protocols. Consider the following schedule and the document in Figure 5, which illustrates the M locks held by T_1.

T_1		T_2	
sd	$\Longrightarrow n_1$		
nthP(2)	$\Longrightarrow n_2, n_3$		
del	delete n_3		
		sd	$\Longrightarrow n_1$
		nthM(1)	$\Longrightarrow n_4$
		nthP(1)	$\Longrightarrow n_7$

Clearly, using Doc2PL the M lock held by TA_1 on the whole document blocks already the first operation of T_2. With Node2PL, T_1 locks n_1 in exclusive M mode and again the first operation of T_2 is blocked. NO2PL requires T_1 to lock n_2 and n_4 in M mode. T_2 can acquire a T lock on n_1 and execute its first operation. Then it has to wait. Under OO2PL T_1 acquires MR and ML locks for n_2 and n_4 respectively. T_2 can still acquire a TA and a TZ lock on n_1 and n_4 respectively, and does not have to wait at all.

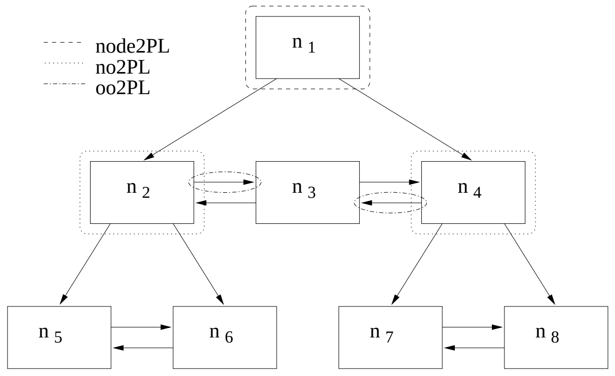


Figure 5: M-Locks held by T_1 for different 2PL versions (deletion)

Example: Insertion Consider the following schedule and the document in Figure 6, which illustrates the M locks held by T_1.

T_1		T_2	
sd	$\Longrightarrow n_1$		
nthP(1)	$\Longrightarrow n_2$		
insA	insert n_x		
		sd	$\Longrightarrow n_1$
		nthP(1)	$\Longrightarrow n_2$
		nthP(1)	$\Longrightarrow n_5$

Again, under Doc2PL and Node2PL T_2 cannot execute a single operation until T_1 releases its locks, since both require an exclusive M lock on n_1. Under NO2PL T_1 acquires an M lock for n_2 and n_3. This still allows T_2 to traverse n_1, but then it has to wait. With OO2PL T_2 does not have to wait at all, since the M locks acquired by T_1 (MR for n_2 and ML for n_3) still allow T_2 to traverse from n_1 via n_2 to n_5.

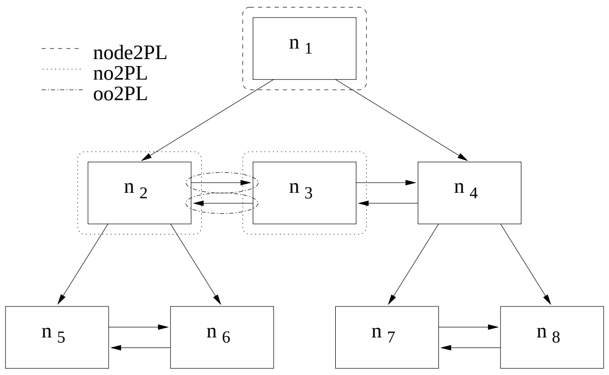


Figure 6: M-Locks held by T_1 for different 2PL versions (insertion)

3.2 Timestamp-based protocol

In the textbook version of timestamp ordering (TO) the transaction manager assigns every transaction a unique timestamp [2, 14]. These timestamps are inherited by the operations of a transaction. The scheduler then guarantees that conflicting operations are executed only in the order of their timestamps. Transactions have to commit in timestamp order.

Versioning The basic idea of our extension XTO of the timestamp based protocol for synchronizing traversals and structure modifications on XML documents is to keep implicitly *multiple versions* of the documents. In case of a deletion, the node is marked *deleted* but left in place and the timestamp of the "deleting" transaction is recorded. An inserted node is inserted right away and marked *inserted*. Again the timestamp of the inserting transaction is remembered. At the end of the inserting/deleting transaction, the marks are eliminated and the deleted nodes are removed from the document. The mechanism of marking nodes as inserted or deleted together with recording the timestamps allows us to reconstruct the version of a document for any timestamp between the first and the last not yet committed modification.

Action Table The next step is to fill the *action table* for XTO. Based on the implicit versions we can specify the complete list of actions taken by the scheduler and the transactions. Consider two operations $op_1 \in TA_1$ and $op_2 \in TA_2$ such that op_1 has already been executed and op_2 is to be scheduled. Figure 7 indicates what will happen if there is a node n which op_1 either traversed (T), deleted (D) or inserted (I) and where op_2 wants to either traverse, delete or insert node n. Please keep this scenario in mind since it is used throughout the rest of this section. In the following discussion $<_{TS}$ denotes the timestamp order of transactions.

When both operations are traversal operations, there is no conflict and no special action is to be taken: op_2 simply traverses node n. Another simple case is when n is inserted by op_2. Then n did not exist when op_1 was executed. Hence the last three cases of the action table are not possible.

For the discussion of the DT case, assume that a transaction TA_1 with timestamp $ts(TA_1)$ deletes a certain node — say $C2$ of Figure 4. Instead of removing this node, we mark the node as deleted but keep it in place. Additionally, we remember the timestamp of TA_1. If after the "deletion" of the node another transaction TA_2 wants to access the second child of node P, we consider two cases. If $ts(TA_2) < ts(TA_1)$, then TA_2 is still allowed to traverse the "deleted" node. If $ts(TA_2) > ts(TA_1)$, then the "deleted" node is ignored and the result of an **nthP(2)** child operation is node $C3$. In order to allow the first case, we have to make sure that transactions commit in timestamp order. Further, if in the second case where $C2$ is ignored by TA_2, TA_1 aborts, then TA_2 should not have ignored $C2$. Hence we are forced to abort TA_2. As a consequence, XTO does not avoid cascading aborts.

The IT case is similar to the DT case. A node inserted by TA_1 is marked as inserted. If transaction TA_2 comes along to access the inserted node, it ignores the node if it is younger than TA_1. Otherwise it traverses the node.

Consider now the TD case. When transaction TA_2 wants to delete a node that has been traversed by an older transaction TA_1, this is no problem. If it has been traversed by a younger transaction TA_1, then TA_2 must abort. Another possibility would be to abort TA_1. This requires to keep read sets. Since we only keep the maximum timestamp of all transactions that traversed a node, we have to abort TA_2 since we cannot track down all transactions younger than TA_2 that accessed node n.

For the DD case let us first consider the case where $TA_2 <_{TS} TA_1$. That is, TA_1 has deleted a node and afterward comes transaction TA_2 that is younger than TA_1 and also wants to delete the node. TA_2 is too late and hence must be aborted. If TA_2 is older than TA_1 ($TA_1 <_{TS} TA_2$), then TA_2 should never be able to see the node n that TA_1 deleted. This follows from the action table's entry for the DT case.

In the ID case, since we do not mark a node as both deleted and inserted, we must block TA_2 (in the $TA_1 <_{TS} TA_2$ case) until after TA_1 committed. Consider now the case where $TA_2 <_{TS} TA_1$. From case IT, it follows that TA_2 has to ignore the existence of the node n inserted by TA_1. Since a node can only be deleted if it has been accessed, TA_2 will never be able to delete node n. Refer to the IT case, which enforces TA_2 to ignore node n.

A case deserving special attention when using XTO is the deletion of a node n by the same transaction TA_i that inserted n before. In this case, any reader that accessed n between the insert and the delete must be aborted. Since our version of XTO does not require to keep the set of all readers but instead only monitors the maximum timestamp of all readers, we are forced to treat this case differently. If some transaction traversed node n, then the maximum timestamp of all readers of n must be larger than the stimestamp of TA_i (refer to the IT case of the action table for XTO). If this is the case, we abort the transaction TA_i. Using the established abort dependencies, we can now abort all potential readers (and possibly some more transactions). This variant of XTO, where we do not keep read sets, leads to at least one more abort than the variant keeping read sets. However, we feel that this is an unlikely situation that does not justify the additional overhead of keeping read sets.

Overhead The overhead of the XTO protocol is rather low. We need two timestamps and a status field per node. Monitoring read sets increases this overhead but might lead to fewer aborts. XTO does not avoid cascading abort. Blocking can occur in the ID case. Further, since transactions must commit in timestamp order, they may be blocked at commit time to wait for other transactions with smaller timestamps to complete. If there are no delete operations and no transaction performs a self-abort, there are no aborts under XTO. XTO must maintain a wait graph and the abort depen-

op_1	op_2	$TA_1 <_{TS} TA_2$	$TA_2 <_{TS} TA_1$
T	T	no conflict	no conflict
D	T	ignore (casc. abort)	traverse
I	T	traverse (casc. abort)	ignore
T	D	mark deleted	abort TA_2
D	D	not possible 1	abort TA_2
I	D	block TA_2 then mark del.	not possible 2
T	I	not possible 3	not possible 3
D	I	not possible 3	not possible 3
I	I	not possible 3	not possible 3

The cases marked by *casc. abort* may lead to cascading aborts. In the ID case where $TA_1 <_{TS} TA_2$, we must block TA_2 until TA_1 commits. When TA_1 commits, TA_2 may continue by marking the node as deleted. This is necessary, since we cannot mark a node as inserted and as deleted at the same time. Some cases are not possible:

1 TA_2 will never see the deleted node.

2 TA_2 will never see the inserted node.

3 A node cannot be accessed before it exists.

Figure 7: Action Table for XTO

dencies.

Example The following example illustrates a case where XTO is superior to OO2PL. Consider Figure 8 and the following schedule under the assumption that T_1 has a smaller timestamp than T_2.

T_1		T_2	
sd	$\Longrightarrow n_1$		
		sd	$\Longrightarrow n_1$
nthP(1)	$\Longrightarrow n_2$		
		nthM(1)	$\Longrightarrow n_4$
del	delete n_2		
		del	delete n_4
sd	$\Longrightarrow n_1$		
		sd	$\Longrightarrow n_1$
nthM(1)	$\Longrightarrow n_4$		
		nthP(1)	$\Longrightarrow n_3$
nthP(1)	$\Longrightarrow n_7$		
		nthP(1)	$\Longrightarrow n_6$

This schedule leads to a deadlock using OO2PL, while under XTO it runs smoothly.

3.3 Dynamic commit ordering

The main disadvantage of XTO is that timestamps prescribe the commit order of transactions. Allowing more flexibility by determining the commit order of transactions dynamically is the basic idea of the *dynamic commit ordering protocol* XCO. The idea of dynamically ordering transactions is also discussed in [1] where *recoverable* operations are allowed to execute. The notion of recoverability as used in [1] is more

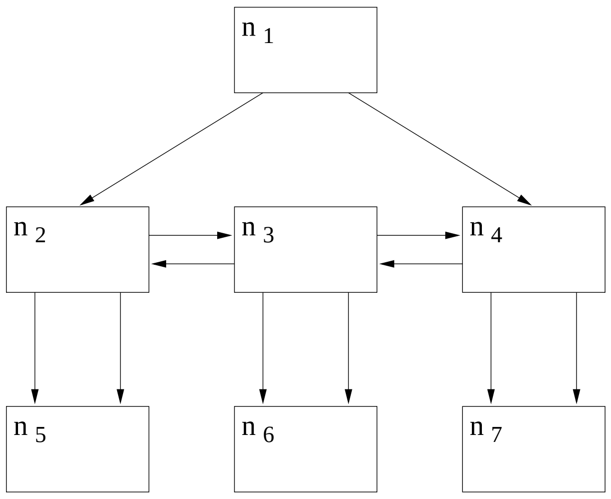


Figure 8: Example document for deletion

general than the usually used notion of commutativity. However, our traverse and modification operations are not recoverable and the protocol of [1] blocks the execution of non-recoverable operations. In XCO instead of blocking, we use the same idea of keeping implicitly multiple versions of documents by marking nodes as deleted or inserted. Different from XTO, XCO does not keep timestamps but instead keeps references to the transactions that inserted, deleted or traversed a node. The resulting overhead of this information is comparable to that of NO2PL.

Dynamic Commit Order Additionally, XCO maintains dynamically a graph representing the dynamic commit order $<_{DCO}$. During the execution of the transactions, edges are added to and removed from this graph. Adding edges to the graph takes place whenever two conflicting operations are detected and the order of their transactions is so far undetermined by $<_{DCO}$. Removal of edges takes place at the end of a transaction, i.e. when it commits or aborts. Then all edges to and from the ending transactions are removed from $<_{DCO}$. Additionally to $<_{DCO}$, XCO maintains a wait graph and abort dependencies as it does not avoid cascading aborts.

Action Table The complete list of actions taken by the scheduler and the transactions is given in Figure 9. Apart from the last column, it is almost identical to the action table of XTO. For the subsequent discussion, we assume the same scenario as for the discussion of XTO. That is, given are two operations $op_1 \in TA_1$ and $op_2 \in TA_2$ working on a node n such that op_1 has already been executed and op_2 is about to be scheduled.

When the transactions are not yet ordered, we must choose the action the transaction takes (i.e. traversing or ignoring the node) and as a consequence order TA_1 and TA_2 accordingly (see Fig. 10). In case of DT and IT, we have a choice. Hence, there exist four variants of the XCO protocol. We implemented these variants and their evaluation convinced us that the variants indicated by a* in the above table are the best ones. Under XCO transactions are allowed to commit only in $<_{DCO}$ order. Thus waits may occur at commit time.

op_1	op_2	$TA_1 <_{DCO} TA_2$	$TA_2 <_{DCO} TA_1$	$TA_1 <>_{DCO} TA_2$
T	T	no conflict	no conflict	no conflict
D	T	ignore (casc. abort)	traverse	see Fig. 10
I	T	traverse (casc. abort)	ignore	see Fig. 10
T	D	mark deleted	abort TA_2	see Fig. 10
D	D	not possible 1/abort	abort TA_2	not possible 2
I	D	block TA_2 then mark del.	not possible 3	not possible 4
T	I	not possible 5	not possible 5	not possible 5
D	I	not possible 5	not possible 5	not possible 5
I	I	not possible 5	not possible 5	not possible 5

The last column covers the case that TA_1 and TA_2 are not yet ordered. The possibilities to fill the gap in the DT, IT, and TD cases are discussed below. Some cases are not possible:

1 Only possible if DT<> case chooses traverse action (see below). In this case TA_2 is aborted.

2 Not possible since a node has to be traversed before it can be deleted. Hence, the case DT occurred before and orders TA_1 and TA_2.

3 TA_2 will not see a node inserted by any later transaction.

4 Not possible since a node has to be traversed before it can be deleted. Hence, the case IT applies first and orders TA_1 and TA_2.

5 A node cannot be accessed before it has been inserted.

Figure 9: Action table for XCO

As with XTO, the case where a transaction deletes a node it inserted before needs special attention. Since with XTO we record all transactions that read a node, we can abort specifically those that read the inserted and then deleted node. However, for both protocols recovery in this case becomes tricky. Discussing recovery for XTO and XCO is beyond the scope of this paper.

Overhead XCO must maintain a wait graph, the $<_{DCO}$ ordering, and abort dependencies. Further timestamps are recorded for each traversal, deletion and insertion.

Example The following example illustrates the superiority of XCO over XTO and OO2PL. Consider Figure 8 and the following schedule:

TA_1	TA_2	action	add to $<_{DCO}$	comment
D	T	ignore*	$TA_1 <_{DCO} TA_2$	cascading abort
		traverse	$TA_2 <_{DCO} TA_1$	may lead to D D
I	T	ignore*	$TA_2 <_{DCO} TA_1$	
		traverse	$TA_1 <_{DCO} TA_2$	cascading abort, may lead to I D
T	D	mark deleted	$TA_1 <_{DCO} TA_2$	no other choice

Figure 10: Action table for XCO ($TA_1 <>_{DCO} TA_2$)

T_1		T_2	
sd	$\Longrightarrow n_1$		
nthP(1)	$\Longrightarrow n_2$		
		sd	$\Longrightarrow n_1$
		nthM(1)	$\Longrightarrow n_4$
		del	delete n_4
		sd	$\Longrightarrow n_1$
		nthP(1)	$\Longrightarrow n_2$
		del	delete n_2
sd	$\Longrightarrow n_1$		
nthM(1)	$\Longrightarrow n_4$		

This schedule leads to a deadlock using OO2PL, under XTO either T_1 or T_2 has to abort when T_2 reaches the second delete operation (T_1 should not have been able to traverse node n_2 as T_2 (with a lower timestamp) has already deleted this node). Under XCO the schedule executes smoothly with no problems (when reaching the second delete operation, we have to order T_1 and T_2: $T_1 <_{DCO} T_2$.

4 Extensions

4.1 Full-fledged protocols

4.1.1 Node contents

In order to extend the core protocols to full protocols, we need to isolate content accesses and modifications. For the 2PL based core protocols, this can easily be done by adding the traditional S and X locks for contents with their according compatibility matrix. The compatibility matrix comprising all four locks is:

	S	X	T	M
S	+	-	+	+
X	-	-	+	+
T	+	+	+	-
M	+	+	-	-

Note that the S and X locks are compatible with the T and M locks. These locks can be applied at the document and node level. This way, Doc2PL, Node2PL and NO2PL can easily be extended. To extend OO2PL, we introduce S and X locks. These locks are compatible with any of the Tx and Mx locks.

XTO and XCO can be extended by adding two additional timestamps for node contents. The first registers the latest read while the other registers the current not yet committed update of contents. We then have to extend the action tables of XTO and XCO such that in case of a conflict, one of the transactions is either blocked or aborted.

Note that we do not need any special treatment for node attributes since they are treated as special nodes in DOM. Further, if text is only hold in text nodes — which is the case in Natix [11] — two lock modes suffice. Hence, we can — for example — replace the T and M lock modes by the S and X lock modes.

4.1.2 ID lookup

So far all protocols require access to a document by starting the traversal at the document's root. XML provides ID attributes with uniquely identify nodes within a document. Links to nodes with ID attributes are realized using IDREF and IDREFS attributes. DOM allows to jump to a node with a given ID. This way, there is no guarantee that traversal starts at a root and goes down a non-interrupted path. These ID jumps may lead to non-serializable schedules for our core protocols.

To remedy this situation, we keep for every document a set of ID locks. We remember which IDs have been the target of a jump. Assume that some transaction searched for a node with ID *id*. We then record the lock *IDJ(id)* for the according document. If a node with an ID attribute whose value is *id* is deleted/inserted, we required the transaction to hold a lock *IDD/IDI(id)*. We record changes of IDs as delete followed by an insert. These locks are compatible if their nodes differ or if they both are IDJ locks. Otherwise they are incompatible.

So far, we required that a transaction moves down a document. More specifically, a transaction must hold a lock on the parent node in order to acquire a lock for the child node. This is the typical requirement for tree locking protocols designed for higher concurrency on B-Tree index structures [2]. The reason we need this requirement is the following. If a node with children is deleted, we only lock the deleted node without locking its descendants. If now a jump to a descendant takes place, we are in trouble. There exist two solutions to the problem. The first solution is that all descendant nodes of a deleted node have to be locked. This results possibly in many locks. Further, since we did not lock deleted nodes so far, this approach does not really fit our protocols nicely — although it is possible. Therefore, we favor another approach. We traverse the whole deleted subtree for nodes having an ID attribute. For these nodes, IDD locks are acquired.

To treat ID jumps, XTO and XCO can be extended by adding the same mechanism of IDx locks. An alternative for the subtree deletion part of the extension is to check the path from a node a transaction jumped to up to the root of the document containing this node. This is possible, since we do not delete nodes until after the deleting transaction committed.

4.2 DTD-based conflict reduction

Knowledge of the DTD can reduce the number of conflicts of the 2PL based protocols Node2PL, NO2PL and OO2PL. We illustrate the exploitation of DTD knowledge by means of a simple example. Let a DTD specify a node's content as *A*B*C**. That is, the first couple of children are of type A, then follow the B and the C nodes. Figure 11 (a) shows an example document adhering to the DTD. Note that the DTD groups the children of the root node into different blocks.

Assume that there are operations **first(t)** and **last(t)** that retrieve the first/last child of type t of a given node. Consider the schedule

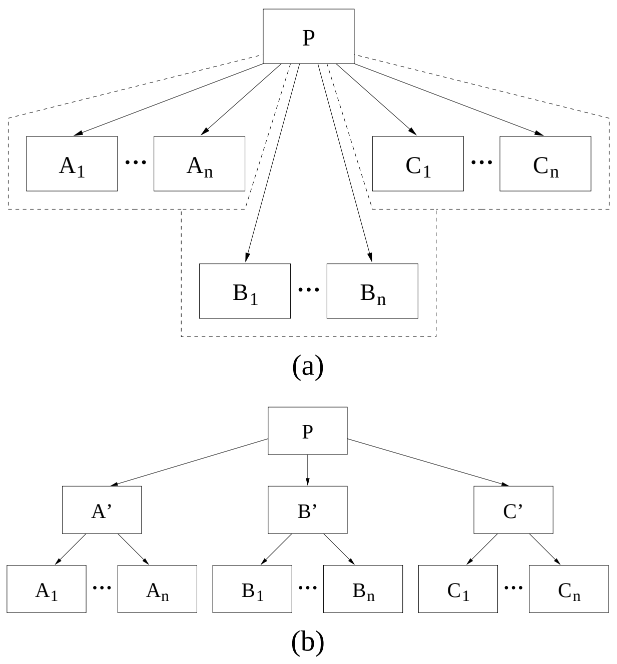


Figure 11: DTD illustration

TA_1	TA_2
first(B)	
	last(A)
insB(x)	
	insA(y)

In this schedule, all 2PL based protocols block TA_1 when it is trying to execute **insB(x)**. Assume that x is of type B and y is of type A. Then — under the given DTD — there is no conflict since **last(A)** and **insB(x)** as well as **first(B)** and **insA(y)** commute.

In general, whenever the DTD groups the children of a node into sets of disjoint type, then any jump to one of these sets and any modification of it commutes with any jump to another set and its modification. To see the reason why the operations commute it is illuminating to consider again our example document in Figure 11 (a). Since the nodes in each group are of different type, we can introduce artificial dummy nodes A', B' and C'. Executing for example a **first(B)** operation is then equivalent to jumping to B' — the artificial top node of all B nodes — and then selecting its first child. Any change taking place under any of the dummy node obviously does not interfer with any change in another subtree below some other dummy node.

5 Conclusion

We have introduced six different core protocols. Each of them isolates structure traversals and modifications and guarantees serializability for these operations. The core protocols Doc2PL, Node2PL, NO2PL, and OO2PL are based on two phase locking.

Their main difference is the locking granularity. OO2PL is based on ideas for synchronizing abstract data types. The core protocol XCO is based on timestamp ordering whereas XCO is based on dynamic commit ordering. Both of the latter two protocols use an implicit representation of multiple versions to enhance their degree of concurrency.

We also discussed how the core protocols can be extended to provide support for all concepts to cover the full DOM standard. We further illustrated that DTD knowledge can improve the degree of concurrency achieved by the two phase locking based protocols.

References

[1] B. Badrinath and K. Ramamrithan. Semantics-based concurrency control: Beyond commutativity. *ACM Trans. on Database Systems*, 17(1):163–199, 1992.

[2] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[3] K. Böhm, K. Aberer, E. Neuhold, and X. Yang. Structured document storage and refined declarative and naviational access mechanisms in HyperStorM. *VLDB Journal*, 6(4):296–311, 1997.

[4] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible markup language (xml) 1.0. Technical report, World Wide Web Consortium, 1998. W3C Recommendation 10-Feb-98.

[5] A. Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data with STORED. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, 1999.

[6] L. Wood et al. Document object model (dom) level 1 specification (second edition). Technical report, World Wide Web Consortium, 2000. W3C Working Draft 29-Sept-2000.

[7] D. Florescu and D. Kossmann. Storing and querying XML data using an RDBMS. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.

[8] R. Goldman, J. McHugh, and J. Widom. From semistructured data to XML: Migrating the Lore data model and query language. In *ACM SIGMOD Workshop on the Web and Databases (WebDB)*, 1999.

[9] P. Gray and A. Reuter. *Transaction Processing: Concepts and Technology*. Morgan Kaufmann Publishers, San Mateo, Ca, 1993.

[10] C.-C. Kanne and G. Moerkotte. Efficient storage of XML data. Technical Report 08/99, University of Mannheim, Mannheim, Germany, 1999.

[11] C.-C. Kanne and G. Moerkotte. Efficient storage of XML data. In *Proc. IEEE Conference on Data Engineering*, page 198, 2000.

[12] M. Klettke and H. Meyer. XML and object-relational database systems – enhancing structural mappings based on statistics. In *ACM SIGMOD Workshop on the Web and Databases (WebDB)*, 2000.

[13] H. Korth. Locking primitives in a database system. *Journal of the ACM*, 30(1):55–79, 1983.

[14] C. H. Papadimitriou. *The Theory of Database Concurrency Control*. Computer Science Press, 1986.

[15] K. Ramamrithan and P. Chrysanthis. *Advances in Concurrency Control and Transaction Processing*. IEEE Computer Society Press, 1997.

[16] A. Schmidt, M. Kersten, M. Windhouwer, and F. Waas. Efficient relational storage and retrieval of XML documents. In *ACM SIGMOD Workshop on the Web and Databases (WebDB)*, 2000.

[17] P. Schwarz and A. Spector. Synchronizing shared abstract data types. *ACM Trans. Computer Systems*, 2(3):223–250, 1984.

[18] J. Shanmugasundaram, H. Gang, K. Tufte, C. Yhang, D. J. DeWitt, and J. Naughton. Relational databases for querying xml documents: Limitations and opportunities. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 302–314, 1999.

[19] B. Surjanto, N. Ritter, and H. Loeser. XML content management based on object-relational database technology. In *Proc. 1st Int. Conf. on Web Information Systems Engineering (WISE 2000)*, pages 64–73, 2000.