Evaluating Lock-based Protocols for Cooperation on XML Documents

Sven Helmer
Universität Mannheim
68131 Mannheim, Germany
helmer@informatik.uni-mannheim.de

Carl-Christian Kanne
Universität Mannheim
68131 Mannheim, Germany
kanne@informatik.uni-mannheim.de

Guido Moerkotte
Universität Mannheim
68131 Mannheim, Germany
moer@informatik.uni-mannheim.de

ABSTRACT

We discuss four 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) and are based on two phase locking. We also demonstrate the different degrees of cooperation that are possible with these protocols by various experimental results. Furthermore, we also discuss extensions of these core protocols to full-fledged protocols. Further, we show how to achieve a higher degree of concurrency by exploiting the semantics expressed in Document Type Definitions (DTDs).

1. INTRODUCTION

The rapid proliferation of the eXtentensible Markup Language (XML [4]) in many different application areas results in a rapidly growing number of XML documents. This is especially true in web-based applications where the semi-structuredness of the data makes markup languages ideal for representing data. It is our hypothesis that sooner or later users will work concurrently on 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. At the moment, most tools of this kind work on the XML documents using a standardized application programming interface (e.g. the Document Object Model (DOM) [8]). Isolating different concurrent applications (i.e. preventing them from having unwanted side effects on each other) 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, due to the lack of synchronization mechanisms. The second alternative is an existing relational, object-oriented, or object-relational database system [3, 7, 10, 16, 20, 23, 24]. In the case of relational database systems there are several different translation schemes. In one of them elements are mapped onto tuples. Elements from different documents may also share tables. In this case we need to lock the whole table when inserting nodes to avoid the phantom problem. The only translation scheme in which tables are not shared stores the XML documents in Character Large OBjects (CLOBs). In this case, however, locking is only possible at the document level by locking the whole CLOB or at random byte positions within the CLOB by range locking. Obviously, locking the whole document has too coarse a granularity, while range locking completely disregards the structure of the XML document. The third alternative is to implement a native XML base management system (XBMS) [9, 11, 15]. One of the reasons to follow the XBMS approach is that it allows incorporating synchronization protocols specifically adapted to the manipulation of XML document collections.

The development of synchronization protocols for isolating different applications has a long and successful history in the database community. One of the key concepts here is the notion of serializability, i.e. that the outcome of concurrently executed transactions is equivalent to a strictly serial execution of the transactions. Most of the protocols that guarantee serializability already found their way into textbooks more than a decade ago [2, 13, 18]. During the last decade some researchers have concentrated on defining notions weaker than serializability and developed protocols that allow a more liberal cooperation between users. For a survey on cooperating transactions and synchronization in general see [19]. Recently, the topic of synchronization was picked up again in the context of XML. Grabs et al. propose DGLOCK [12], a protocol for semantic locking on DataGuides. However, there are still some shortfalls concerning the handling of IDREFs and position-based predicates. Dekeyser and Hidders argue that predicate locks are too expensive and still too restrictive. As an alternative, they propose a path locking protocol in [5, 6]. However, a direct jump into subtrees via IDREFs is still not possible and the paths for locking may not be arbitrary (only a subset of XPath is covered).

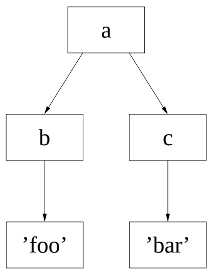
We believe that serializability should be the foundation for protocols that allow cooperation, as there is always a lowest level where actions have to be atomic and have to be isolated carefully in order to prevent the unwanted side effects mentioned before. This motivated us to start with the development of protocols that guarantee serializability [14].

The paper is organized as follows. In Section 2, we briefly describe a set of access and modification operations we will consider for our core protocols. Section 3 discusses four different core protocols. They are based on strict two phase locking and differ in their locking granularity. Two of these core protocols use mechanisms developed for synchronizing ADTs [1, 17, 22]. In Section 4, we take a closer look at the performance of the protocols. Section 5 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. Section 6 concludes the paper.

2. TRAVERSING AND MODIFYING XML DOCUMENTS

Semi-structured data, like XML documents, are often represented as ordered, labeled trees. The nodes of the tree store the names of the tags or textual data. For an example of a tree representation of an XML document, see Figure 1.

```
<a>
  <b>
    foo
  </b>
  <c>
    bar
  </c>
</a>
```



(a) An XML document (b) Tree representation

Figure 1: An XML document and its tree representation

We take a different path than than other approaches by proposing a locking protocol tailored to typical APIs for processing XML documents.

The operations of these APIs (e.g. DOM [8]) fall into four categories: mutators and observers of the contents of a node and mutators and observers of the structure of a document (for a list of some operations provided by DOM see Figure 2). 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 5, we extend these core protocols to 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:

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 2: Some DOM 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

del deletes a given node

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

3. PROTOCOLS

In this section, we introduce the core protocols for synchronizing structure traversals and modifications of XML documents. Generally speaking, our protocols are based on two phase locking [2, 18]. 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 5. Insofar it is different from regular tree locking protocols (as described in [2]). In our full protocol we do not need the assumption that all accesses are strictly from top to bottom. We also provide different locking schemes, e.g. locking pointers instead of nodes.

3.1 Lock Modes

In standard two phase locking protocols for synchronizing read and write operations, we have two kinds of locks: shared locks (S) and exclusive (X) locks. We could also allow browse locks [25] in our approach without further trouble, but to keep the following descriptions as simple as possible, we confine ourselves to shared and exclusive locks at this point. Read operations require a shared lock while write operations require an exclusive lock. We will look at the topic of content modification later, when introducing the complete protocols (Section 5).

The more difficult (and novel) subject is synchronizing structure traversal and modification via locking protocols. Therefore, we investigate this first. Similar to the shared and exclusive locks for content traversal and modification, we introduce a shared lock named T that has to be acquired for traversing the document structure and an exclusive lock named M that has to be acquired for modifying the document structure.

3.2 Compatibility Matrix

The compatibility matrix of these two locks is analogous to the one for S and X locks (see Figure 3 (a)). The standard rules of two phase locking (2PL) 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.

3.3 Doc2PL

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. Note that although this protocol is widely used in XBMS at the moment (e.g. in Tamino [21]), it does not allow cooperation on one single XML document. We include it nonetheless, as we use it as a reference for comparison.

	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

document parameters	
number of documents	100
document depth	4
minimal fan-out of node	3
maximal fan-out of a node	5

transaction parameters	
number of transactions	100
. . . concurrent transactions	5
. . . operations per transaction	50

probab. for ops	
select document	0
nthP	40
nthM	40
insA	5
insB	5
delete	10

Table 1: Default parameters for the simulation environment

3.4 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 which connect 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 [9]) 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.

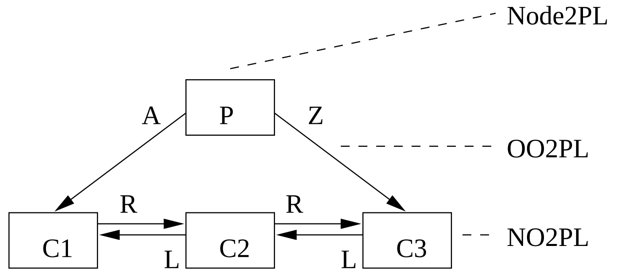


Figure 4: Conceptual list representation of XML documents

3.5 Node2PL and NO2PL

Figure 4 also shows on which items the different protocols acquire locks. The Node2PL protocol acquires locks for parent nodes. For example, if 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 $C1$ because 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, as all ways to it are blocked: $C0$ can be reached from the parent node neither by an **nthP** operation nor by an **nthM** operation, since P and $C1$ are locked exclusively.

3.6 OO2PL

Whereas in Node2PL and NO2PL we lock nodes, OO2PL locks pointers. As there are four pointers for every node (first child (A), last child (Z), left sibling (L) and right sibling(R)), we need four shared 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 [22].

4. EVALUATION

4.1 Simulation Environment

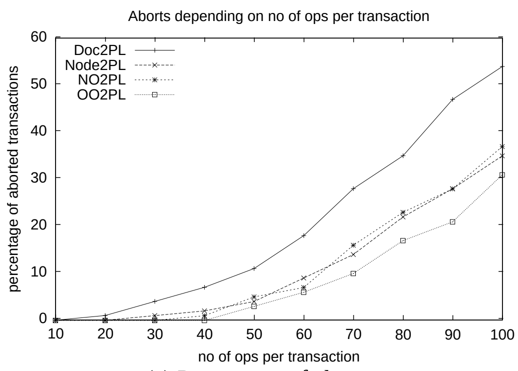
We implemented a simulation environment in which we tested the performance of the different core protocols. In the context of our simulations we generate a number of documents. Each document has a certain depth, and the fan-out of a node is determined randomly within a certain range. The default parameters chosen to generate the documents are given in Table 1. These parameters sometimes lead to abortion rates that are quite high and may be seen as unrealistic (e.g. having only five times as many documents as transactions). We did this in order to see how the protocols operate at full capacity or even beyond. In other words, we are interested in how well the protocols scale and are able to handle peak activity or even overload.

On these documents, the transactions perform operations as defined in Section 2. As its first operation, a transaction selects a document randomly. It then continues by choosing randomly any of the operations with the default probabilities indicated in Table 1. These probabilities do not apply to modifications at the root node level. Root nodes cannot be deleted and no siblings to root nodes can be inserted. When a transaction tries to access a child of a document's leaf node, this operation fails and the transaction selects a new document randomly.

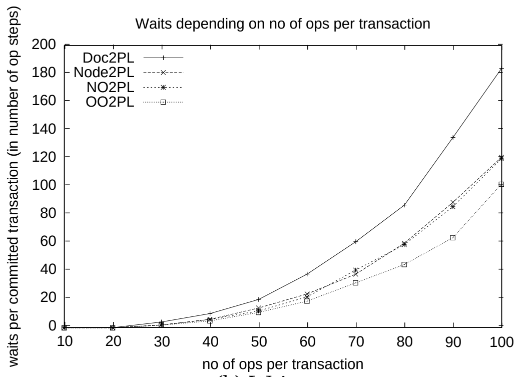
We measured the percentage of transactions that aborted as well as the average number of waits (in number of operation steps) per committed transaction. Note that these two parameters are critical for the throughput of the protocols, as one reflects the probability that a transaction will commit successfully, while the other measures the average idle time before a transaction successfully commits. We investigated these two parameters subject to variations in the number of operations per transaction, the number of concurrently running transactions, and the number of documents in our collection. Unless otherwise indicated, we use the default parameters from Table 1.

4.2 Varying the Length of Transactions

Figure 5 shows the results for varying the length of the involved transactions. The left hand part (Figure 5(a)) depicts the percentage of aborted transactions, while the right hand part (Figure 5(b)) displays the average number of wait cycles before committing. Obviously, the longer the transactions, the more conflicts occur, resulting in a higher abort rate and longer waits. We can clearly see that the higher locking granularity of the protocols Node2PL, NO2PL, and OO2PL pays off. Doc2PL runs into deadlocks much more often. OO2PL usually has an abort rate that is only half as large as



(a) Percentage of aborts



(b) Waits

Figure 5: Results for varying length of transactions

that of Doc2PL. Up to a transaction size of 40 operations OO2PL has no aborts (due to deadlocks) whatsoever. It also has the smallest idle time per committed transaction.

When comparing Node2PL to NO2PL, we can see that Node2PL dominates NO2PL, as they both show similar performance, but NO2PL needs to hold twice the number of locks to accomplish this (so its overhead is greater). OO2PL is even better than Node2PL and NO2PL, but it needs to manage four locks per node.

4.3 Varying the Number of Concurrent Transactions

Figure 6 shows the results for varying the number of concurrently running transactions. Again, the left hand part (Figure 5(a)) depicts the percentage of aborted transactions, while the right hand part (Figure 5(b)) displays the average number of wait cycles before committing. Clearly, an increase in the number of concurrently running transactions leads to more conflicts, as more transactions are simultaneously competing for the same documents. For this parameter, the performance gap between Doc2PL and OO2PL becomes even more apparent. Most of the time, the percentage of aborted transactions for Doc2PL is three to four times as high as for OO2PL. In terms of performance, Node2PL and NO2PL are very close to each other again, making Node2PL the better choice.

4.4 Varying the Number of Documents

Figure 7 shows the results for varying the number of documents in our collection. The left hand part (Figure 7(a)) depicts the percentage of aborted transactions, while the right hand part (Figure 7(b)) displays the average number of wait cycles before committing. Obviously, the more documents we have, the lower the number of conflicts, as the transactions are more spread out. In terms of performance, the picture looks quite the same. OO2PL comes in at first place, followed by Node2PL and NO2PL (which are very close to each other), and Doc2PL lags behind.

4.4.1 General result

Generally speaking, the degree of cooperation allowed on a collection of XML documents can be improved considerably by abandoning the straightforward approach of locking whole documents. We achieved this by investing resources in the lock manager increasing its lock granularity.

5. EXTENSIONS TO FULL-FLEDGED PROTOCOLS

5.1 Node contents

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

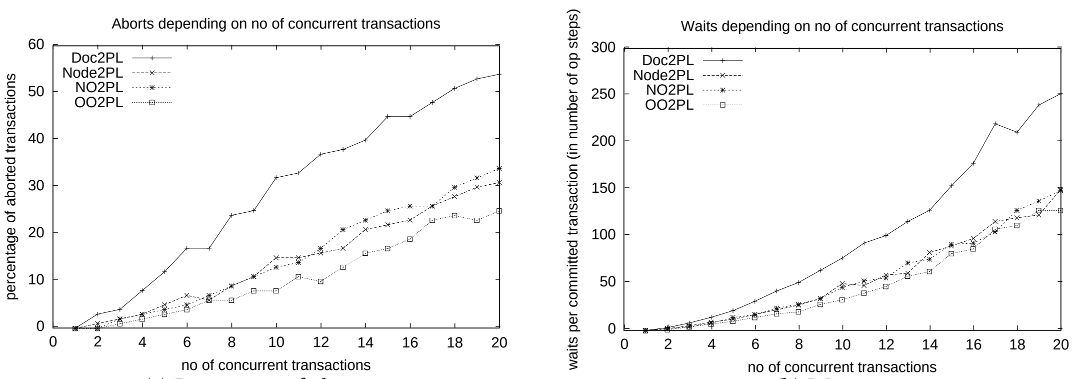
	S	X	T	M
S	+	-	+	-
X	-	-	+	-
T	+	+	+	-
M	-	-	-	-

Note that the S and X locks are compatible with the T locks, while M locks are not compatible with any other locks. All these locks can be applied at the document and node level. This way, Doc2PL, Node2PL and NO2PL can easily be extended.

In the OO2PL protocol, it does not make sense to devise a combined compatibility matrix composed of S, X, Tx, and Mx locks, because the content locks refer to nodes, while the structural locks refer to pointers. Tx locks are implicitly compatible with S and X locks, as traversing through a node using its pointers does not affect the content of a node. Mx locks are implicitly incompatible with S and X locks, because another transaction that has acquired a content lock on a node must have navigated to this node in some way, thereby setting at least one Tx lock.

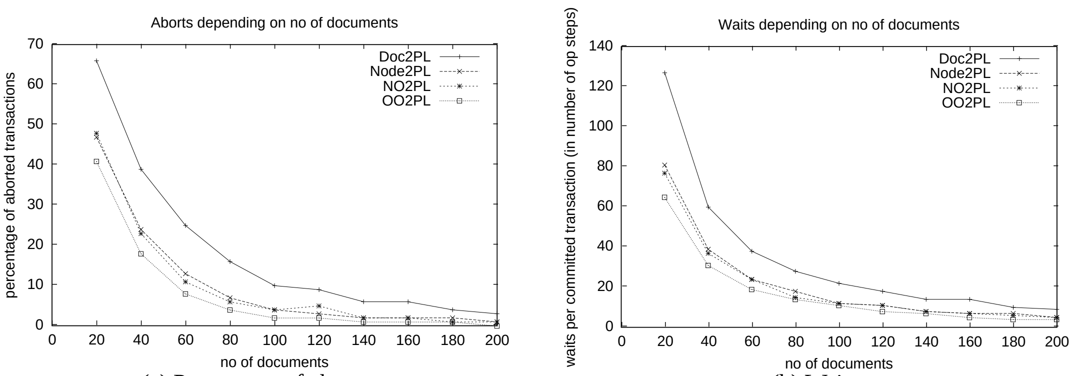
5.2 ID lookup

XML provides ID attributes which uniquely identify nodes within a document. Using IDREF and IDREFS attributes we can establish links from arbitrary nodes within the same document to nodes with an ID attribute. These links allow us to jump directly from one node to another without starting at the root and descending down a non-interrupted path. However, allowing ID jumps leads to serious problems in regular tree locking protocols, resulting in non-serializable schedules. So far, we have 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



(a) Percentage of aborts (b) Waits

Figure 6: Results for varying the concurrency



(a) Percentage of aborts (b) Waits

Figure 7: Results varying the number of documents

the child node. This is the typical requirement for tree locking protocols designed for higher concurrency on B-Tree index structures [2]. The reason why this is necessary is the following. If, for example, a node with children is deleted, we only lock the deleted node but not its descendants. If another transaction jumps to a descendant of the (about to be) deleted node via an IDREF, we are in trouble. The straightforward solution to this problem is to lock all descendants of the node we want to delete. Nevertheless, we advise against this, since it involves a lot of overhead (traversing and locking all the descendants).

Instead, we propose to keep a set of ID locks for every document. Again, we distinguish between shared ID locks (IDS) and exclusive ID locks (IDX). IDS locks are compatible with other IDS locks, while IDX locks are incompatible with all other ID locks. Every time a transaction wants to follow an IDREF link to a node with the ID i, it has to acquire an IDS lock on i. After getting this lock the transaction may navigate from this node to others (or request S or X locks) in the usual way. If we want to delete a node n_j, we have to acquire exclusive IDX locks on all descendants of n_i that possess an ID attribute. In this way, we can enforce serializability even in the presence of ID jumps. (Note that ID locks can be seen as a variant of the Tx and Mx locks of the OO2PL protocol; they can be interpreted as yet another pointer we can traverse.)

5.3 DTD-based conflict reduction

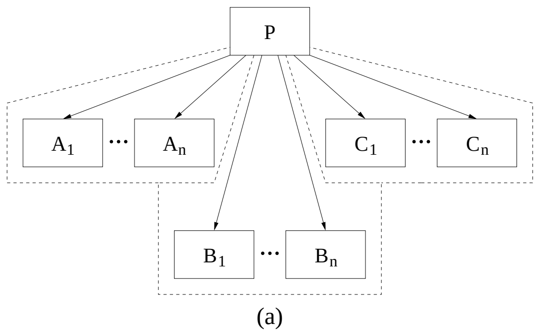
Knowledge of the DTD can reduce the number of conflicts (and hence increase the degree of cooperation) of the protocols Node2PL, NO2PL and OO2PL. We illustrate the exploitation of DTD knowledge by means of a simple example. Let a DTD specify a node's content as *A*B*C**. That is, the first couple of children are of type A, then follow the B and the C nodes. Figure 8 (a) shows an example document adhering to this 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

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

In this schedule, all protocols block T_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, any jump to one of these sets and any modi-



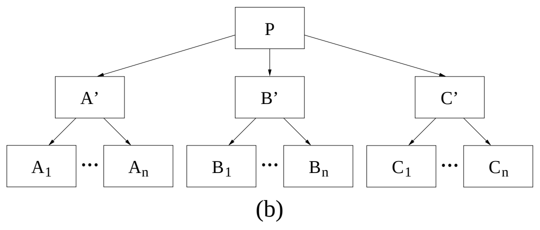


Figure 8: DTD illustration

fication of it commutes with any jump to another set and its modification. To see the reason why the operations commute consider our example document in Figure 8(a) again. 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 nodes obviously does not interfere with any change in another subtree below some other dummy node.

6. CONCLUSION AND OUTLOOK

One of the basic concepts for synchronizing accesses of many different users to the same data is the isolation of these accesses from each other. In order to isolate structure traversals and modifications on XML documents and guarantee serializability for these operations, we have introduced four different core protocols based on two phase locking. OO2PL is also based on ideas for synchronizing abstract data types. Furthermore, we discussed how these 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.

At the moment we are integrating the presented techniques into our native XML base Natix [9] to test them in real applications. We also plan to adapt timestamp-based protocols for synchronizing accesses to semi-structured data. For low-conflict environments we expect these protocols to be even better than 2PL-based ones, due to the avoidance of deadlocks.

7. 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 navigational 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] S. Dekeyser and J. Hidders. Path locks for XML document collaboration. In *Proceedings of the 3rd Int. Conf. on Web Information Systems Engineering (WISE)*, pages 105–114, Singapore, 2002.

[6] S. Dekeyser and J. Hidders. A commit scheduler for XML databases. In *5th Asia Pacific Web Conference (APWeb), LNCS 2642*, pages 83–88, Xi'an, China, 2003.

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

[8] 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.

[9] Thorsten Fiebig, Sven Helmer, Carl-Christian Kanne, Guido Moerkotte, Julia Neumann, Robert Schiele, and Till Westmann. Anatomy of a native XML base management system. *VLDB Journal*, 11(4):292–314, 2002.

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

[11] 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.

[12] Torsten Grabs, Klemens Böhm, and Hans-Jörg Schek. XMLTM: efficient transaction management for XML documents. In *CIKM '02*, pages 142–152, McLean, Virginia, USA, 2002.

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

[14] S. Helmer, C.C. Kanne, and G. Moerkotte. Lock-based protocols for cooperation on XML documents. In *Proceedings of Workshop on Web Based Collaboration (DEXA'03)*, Prague, 2003.

[15] 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. *VLDB Journal*, 11(4):274–291, 2002.

[16] 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.

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

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

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

[20] 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.

[21] H. Schöning. Tamino – a DBMS designed for XML. In *ICDE*, pages 149–154, Heidelberg, 2001.

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

[23] 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.

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

[25] R. Unland and G. Schlageter. Facility for non standard database systems. In A.K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 399–466. Morgan Kaufmann, 1992.