taDOM: A Tailored Synchronization Concept with Tunable Lock Granularity for the DOM API

Michael P. Haustein, Theo Härder

University of Kaiserslautern
P.O. Box 3049, Kaiserslautern, Germany
{haustein, haerder}@informatik.uni-kl.de

Abstract. Storing, querying, and updating XML documents in multi-user environments requires data processing guarded by a transactional context to assure the well-known ACID properties, particularly with regard to isolate concurrent transactions.

In this paper, we introduce the taDOM tree, an extended data model which considers organization of both attribute nodes and node values in a new way and allows fine-grained lock acquisition for XML documents. For this reason, we design a tailored lock concept using a combination of node locks, navigation locks, and logical locks in order to synchronize concurrent accesses to XML documents via the DOM API. Our synchronization concept supports user-driven tunable lock granularity and lock escalation to reduce the frequency of lock requests both aiming at enhanced transaction throughput. Therefore, the taDOM tree and the related lock modes are adjusted to the specific properties of the DOM API.

1 Introduction

The use of the extensible markup language XML [1] for electronic data interchange leads to an enormous growth of the number and size of files keeping semi-structured XML data. In contrast, only structured data is managed by relational or object-relational database systems ((O)RDBMSs) so far. In order to allow combined processing of XML and relational data in an efficient way, it is indispensable to maintain XML documents in these database systems, too. On the other hand, (O)RDBMSs run transactions as their unit of control thereby guaranteeing the well-known ACID properties [2] which greatly improve the overall consistency, reliability and robustness of data management. As a consequence, managing XML documents by an (O)RDBMS also provides transactional properties for semi-structured data. Nevertheless, to accomplish effective and efficient processing for such documents, the control of their traversal and manipulation operations has to be adjusted to the transaction demands.

Different approaches to store XML documents in a relational database system and their performance characteristics are discussed in detail in [10], [11], and [12]. As a major issue, the ways XML documents are stored in a relational database lead to different synchronization problems. If an XML document is contained in a single CLOB attribute, locking may only take place at the document level. In contrast, if an XML document is shredded and stored across several tables, insertion of a new XML element may also affect a large part of the document (and even other documents). Insertion of a new XML element often results in several insert operations to relational database tables (depend-

ing on the shredding algorithm). Due to inadequate synchronization mechanisms, many database systems lock each of the affected tables entirely to prevent phantoms. Hence, such a crude method causes locking of document fragments by accident, even of unrelated documents which are shredded to those tables.

While efficient mechanisms are only available for the data side, combined processing of XML and relational data inside a relational database system requires such mechanisms also for the document side, that is, a native storage format with tailored lock mechanisms for XML documents in the first place.

After having succeeded to store XML data within an (O)RDBMS in a native way equitable to relational data, we can immediately exploit the mature transactional concepts for atomicity, consistency, and durability. However, this is not true for transaction isolation. Since the structure of XML documents widely differs from the one of records and tables, we need a new concept to synchronize concurrent accesses to XML documents to provide for acceptable transaction performance.

Our primary objective is to develop a mechanism for concurrency control of semi-structured data where the properties of the document access interface are explicitly taken into account. For this reason, we refer to the DOM API [3] in order to query, traverse, and update XML documents stored in a database (see Section 2). In Section 3, we introduce the taDOM tree, a data model to represent XML documents. The structure and node types of the taDOM tree are specifically tailored to the properties of the DOM API. Based on the taDOM tree, we present in Section 4 an adjusted lock concept to synchronize concurrent accesses to XML data. Since the DOM API provides both, methods to traverse XML documents node-by-node and methods to apply simple queries, we use, in addition to node locks, a combination of navigation locks for the edges connecting the nodes, and logical locks in order to prevent phantoms. Our concept supports tunable

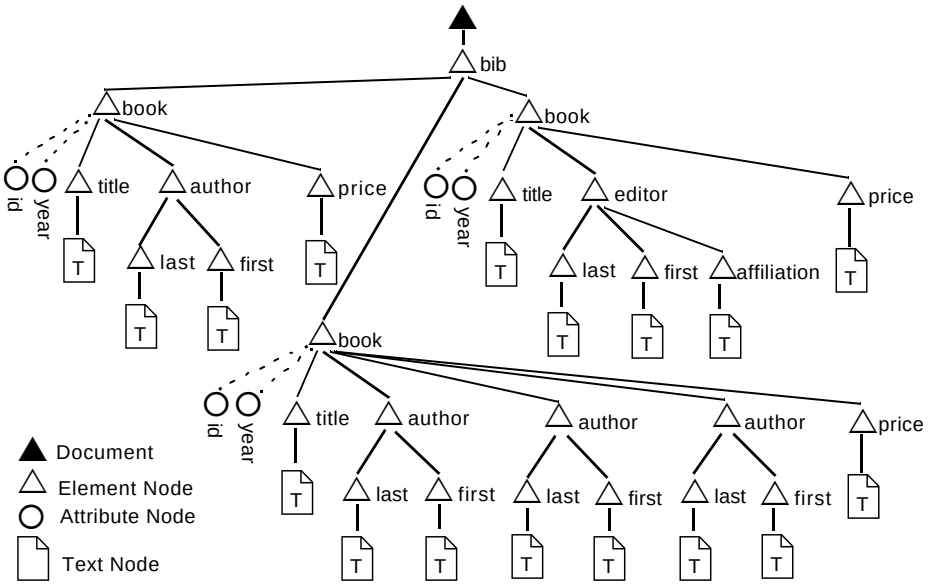


Figure 1: A sample DOM tree

lock granularity and tunable lock escalation which can be independently specified for each document. Finally, in sections 5 and 6, we give a brief overview of related work about XML synchronization and wrap up with conclusions and some aspects of future work.

2 DOM API

The DOM API [3] provides a tree-based view to traverse and update XML documents. Elements and attributes of the document are represented as nodes of a directed acyclic graph (DAG). To illustrate the DOM tree, we consider a small XML fragment depicted in Figure 2, which is a modified version of the *bib.xml* document in [4]. The document contains a list of three books where each book contains the two attributes *year* and *id* and is described by *title*, *price*, *author*, and *editor*, respectively.

Figure 1 shows the corresponding DOM tree which represents the structure of the XML fragment defined in Figure 2. The outer element (in this case the <*bib*> element) is always assigned to a *document* node. Nested elements in the XML document are connected with edges in the DOM tree. Attributes are assigned to elements; they can be queried by their names. Attribute values and text values between opening and closing element tags in the document are stored in attribute nodes or text nodes (connected to the element nodes), respectively.

```
<bib>
   <book year="1994" id="1">
      <title>TCP/IP Illustrated</title>
      <author>
         <last>Stevens</last>
         <first>W.</first>
      </author>
      <price> 65.95</price>
   </book>
   <book year="2000" id="2">
      <title>Data on the Web</title>
      <author>
         <last>Abiteboul</last>
         <first>Serge</first>
      </author>
      <author>
         <last>Buneman</last>
         <first>Peter</first>
      </author>
      <author>
         <last>Suciu</last>
         <first>Dan</first>
      </author>
      <price>39.95</price>
   </book>
   <book year="1999" bid="3">
      <title>The Economics of...</title>
      <editor>
         <last>Gerbarg</last>
         <first>Darcy</first>
         <affiliation>CITI</affiliation>
      </editor>
      <price>129.95</price>
   </book>
</bib>
```

Figure 2: Example of an XML fragment

A set of standardized methods allows the traversal of the graph node-by-node along edges, inserting or deleting nodes, as well as updating node values. Additionally, there exist some methods to apply simple queries to the XML document. For these reasons, we distinguish *navigational access*, *update access*, and *query access* when characterizing the methods of the DOM API.

Navigational access is provided by methods such as *getAttributes()*, *getFirstChild()*, *getNextSibling()*, or *getParentNode()*. The DOM API allows by invoking these and other methods to build a list of all attributes or all child nodes of a node, to navigate to the parent node, to the previous or next sibling node, and to the first or last child node. Methods such as *insertBefore(...)*, *removeChild(...)*, *setNodeValue(...)*, or *setAttribute (...)* support updating XML documents. These methods allow to insert new nodes, remove or replace child nodes, set new node values, and set or remove attributes via the DOM API.

Furthermore, simple queries for XML attributes or XML elements are provided by methods such as *getElementById(...)*, *getElementsByTagName(...)*, or *hasAttribute(...)*. Hence, an element referenced by an ID attribute or a set of elements qualified by their name can be queried.

When the DOM API is initialized for an XML document, the document is parsed and the DOM tree for the entire document is composed in main memory. While this approach provides for fast data access (after initialization), it is a very memory-consuming solution. To give a hint for the required order of magnitude, an XML document of 20 MB may consume up to 400 MB main memory [14]. In order to reduce this enormous memory consumption, some parsers ([15]) offer the option *deferred DOM*, a processing mode which loads the required nodes into memory on demand.

The current version of the DOM API is designed for single user environments. If multiple users open the same XML file, a local DOM tree is constructed for each user, leading to a potentially high degree of replication. On the other hand, a local DOM tree is not affected by modifications of its replicas, even if their update operations are in the committed state. Modifications can only be propagated to the stored XML document by simply overwriting the entire file. Of course, this proceeding leads to the loss of all updates which have been performed by concurrent users, even if their updates only affect non-overlapping sub-trees.

In contrast, *Persistent DOM* from Infonyte-DB ([16]) maps the DOM tree to a compressed binary file representation and provides concurrent access to the file with guaranteed atomicity and durability properties, but no support for the isolation and, in turn, for the consistency aspects. Any updates (committed or not) are immediately visible to all concurrent users which makes the individual user responsible for his view to the document and for the overall consistency of the XML file.

To improve this highly undesirable situation, a centralized and application-independent isolation mechanism is mandatory for the DOM API. First of all, we need a data model preserving the DOM interface properties, but allows for efficient lock acquisition and concurrent operations in addition. For this purpose, we introduce a suitable data model in the form of the taDOM tree in the next section.

3 taDOM Tree

Synchronizing concurrent accesses to XML documents calls for an appropriate logical representation of XML documents which supports fine-grained lock acquisition. To this end, we have developed the taDOM tree, an extension of the XML tree representation of the DOM (see Section 2). To illustrate its construction principles, we again consider the small XML fragment depicted in Figure 2.

Figure 3 shows the corresponding taDOM tree which represents the structure of the XML fragment. In order to construct such a taDOM tree, we introduce two new node types: *attribute root* and *string*. These new node types only appear within the taDOM tree—the logical representation of the XML document.

As far as synchronization is concerned, locks on the taDOM tree are always acquired transparently by a lock manager component introduced in Section 4. This proceeding

guarantees that the behavior of the DOM API will not change from the users point of view.

The top of the taDOM tree consists of the document node, followed by the <*bib*> element—the outer element of the document. For such a document node, a root lock can be acquired which controls access to the entire document or arbitrary parts.

In contrast to a DOM tree in Figure 1, attribute nodes are not directly connected to the element nodes. Instead an attribute root is connected to each element and organizes the attributes of the corresponding element as descendant nodes. Such attribute roots are introduced to support synchronization of the *getAttributes()* method in an efficient way.

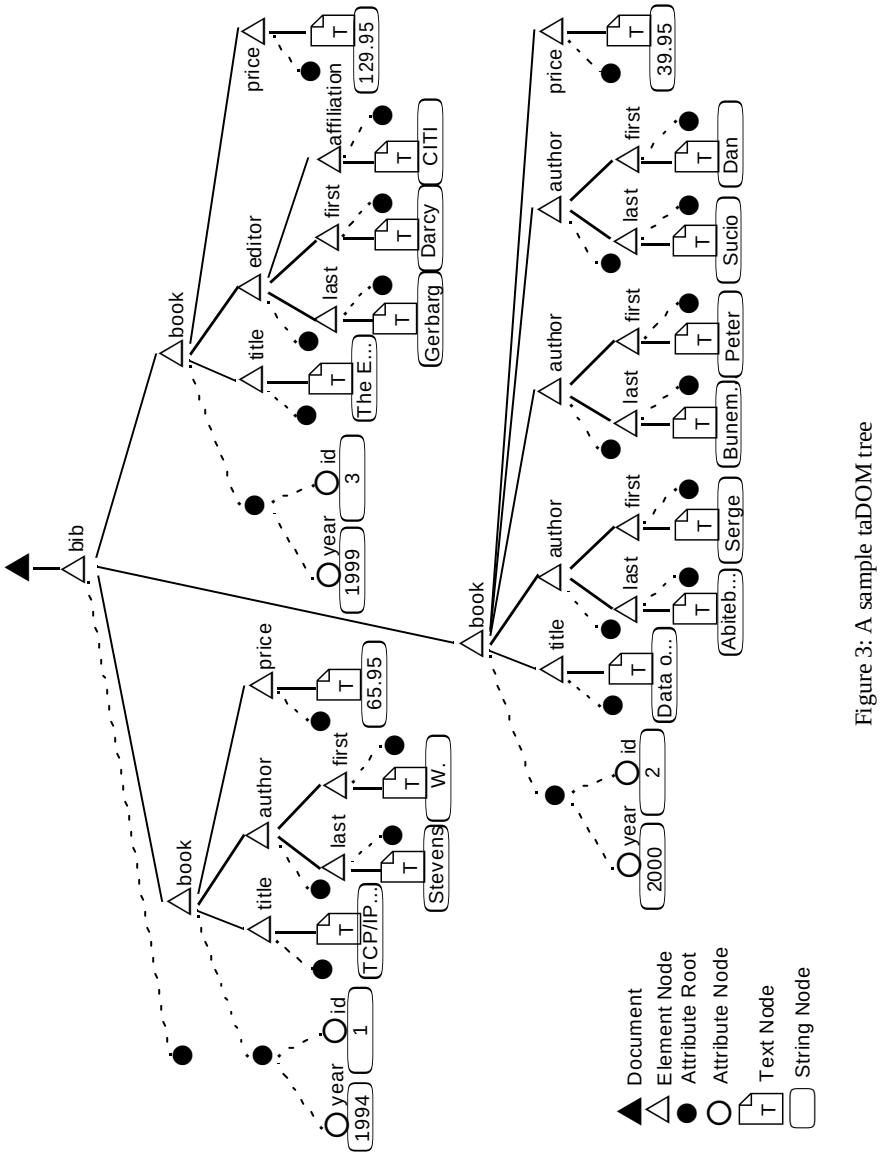


Figure 3: A sample taDOM tree

In order to build a (frequently requested) *NamedNodeMap* containing all attributes of an element, only a single lock has to be acquired for the attribute root thereby enabling shared access to all attributes. Section 4.1 gives a detailed example for that procedure.

In the taDOM tree, XML elements are represented by element nodes and XML attributes by attribute nodes. The text within elements is handled by text nodes whereas the actual values of text nodes or attribute nodes are stored in string nodes. The existence of a string node is hidden to the taDOM API users. The users read or update values as usual by method invocations on the attribute or text nodes, but reading or updating a node value causes an implicit lock acquisition on the corresponding string node. This proceeding makes sense, because the DOM API request for an element or attribute does only return an element node or an attribute node, but not the value of such an node. To get the actual value of a node, the method *getValue()* or *getData()* has to be invoked. Only when such a method is executed, the actual node values (string nodes) have to be locked.

When a taDOM tree is constructed in the way described, lock granularity can be provided even for string nodes and, therefore, our concept enables *maximal concurrency* for DOM API accesses. An example for maximal concurrency is the following. Assume transaction T_1 builds an attribute list and reads all attributes of the first <*book*> element in Figure 3. Then T_1 requests the value of the attribute *id*. Note, although transaction T_1 already operates at the attribute value level, another transaction T_2 should be able to update the value of the attribute *year* in the same element. This access should be possible, because, so far, T_1 only has information about the existence of all attributes and not about all their values.

4 Synchronization of Document Access

So far, we have only explained the newly introduced node types and how individual access to them can be improved. In a concurrent environment, all accesses to XML documents via the DOM API have to be synchronized using appropriate protocols. To facilitate our approach, we classify the related access methods in *navigational access*, *update access*, and *query access* (see Section 2).

In all cases, accessing a node within the taDOM tree requires the acquisition of a lock for the corresponding node. This procedure is described in Section 4.1. Maintaining locks, granting and declining lock requests have to be managed by a lock manager component. This component is also responsible for lock granularity and lock escalation management considered in Section 4.2.

When an XML document has to be traversed by means of navigational methods, the use of the navigation paths needs strict synchronization. This means, a sequence of navigational method calls must always obtain the same sequence of result nodes. To support this demand, we present so-called *navigation locks* in Section 4.3. Furthermore, query access methods also need strict synchronization to accomplish the well-known repeatable read property. For example, the method *getElementsByTagName()* should always yield the same node list within a transactional context. This means the insertion of a new element node with a specified tagname must be blocked if a concurrent transaction has already built a node list containing all existing nodes with this tagname. This demand

and the existence of update methods can be facilitated by logical locks which are explained in Section 4.4.

4.1 Node Locks

While traversing or modifying an XML document, a transaction has to acquire a lock for each node before accessing it. The currently accessed node will be called *working node* in the following. Its lock mode depends on the type of access to be performed. Since the DOM API not only supports navigation starting from the document root, but also allows jumps „out of the blue“ to an arbitrary node within the document, locks must be acquired in either case for the sequence of nodes from the document root downwards to the working node. According to the principles of multi-granularity (or hierarchical) locking schemes, such intention locks communicate a transaction‘s processing needs to concurrent transactions. In particular, they prevent that a sub-tree S can be locked in a mode incompatible to locks already granted to S or sub-trees of S. Figure 4 gives an overview of the compatibility matrix for the lock modes defined as an extension of the well-known DAG locking ([5]). The effects of the different lock modes can be sketched as follows:

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

Figure 4: Compatibility matrix for lock modes

- The NR mode (node read) is requested for read access to the working node. Therefore, for each node from the document root downwards to the working node, a NR lock has to be acquired. Note, the NR mode only locks the specified node, but not any descendant nodes.
- The LR mode (level read) locks an entire level of nodes in the taDOM tree for shared access. For example, the method *getChildNodes()*, called for the working node, only requires an LR lock on the working node and not a number of single NR locks for all child nodes. In the same way, an LR lock, requested for an attribute root, locks all attributes for the *getAttributes()* method.
- To access an entire sub-tree of nodes (specified by the working node as the sub-tree root) in read mode, the SR mode (sub-tree read) is requested for the working node. In that case, the entire sub-tree is granted for shared access. Since the sub-tree can be determined by the user, this lock mode enables flexible concurrency control with tunable granularity.
- To modify the working node (updating contents or deleting the entire node with its sub-tree), an X lock (exclusive) is acquired for the working node. It implies the request of a CX lock (child exclusive) for its parent node and an IX lock (intent exclu-

sive) for its ancestor nodes up to the document node. The CX lock for a node indicates the existence of an X lock for an arbitrary direct-child node whereas the IX lock indicates an X lock for a node located somewhere in the sub-tree. This varying behavior of the CX and IX locks is achieved by the compatibility of the IX and LR locks, and the incompatibility of the CX and LR locks.

- A U lock supports read with (potential) subsequent write access and prevents granting further R locks for the same object. A U lock can be converted to an X lock after the release of the existing read locks or back to an R lock if no update action is needed. It solves the problem of using a too restrictive mode when the necessity of write access is value-dependent.

Figure 5 shows a cutout of the taDOM tree depicted in Figure 3 and illustrates the result of lock request sequences for the following example: Transaction T_1 starts modifying the value *Peter* and, therefore, acquires an X lock for the corresponding string node. This action implies the acquisition of the CX and IX locks for all preceding nodes up to the root by the lock manager component. Simultaneously, transaction T_2 intends to delete the entire <*author*> node which includes the string *Peter*. Therefore, T_2 has to acquire an X lock for the corresponding <*author*> node. This request, however, is declined because of the existing IX lock of T_1 and results in the request for a U lock instead in order to prevent further locks of read transactions. Meanwhile, transaction T_3 generates a list of all book titles and requests an LR lock for the <*bib*> node to obtain read access to all child nodes. Note, LR enables access to all direct child nodes without the need to perform a lock acquisition operation for each child. Then for each <*book*> node, the paths downwards to the title strings are locked by means of NR locks.

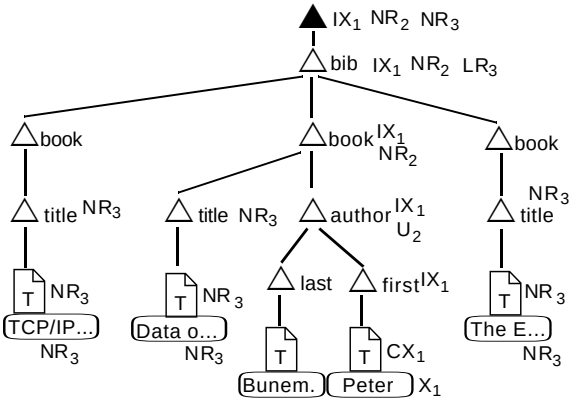


Figure 5: Node locks on the taDOM tree

4.2 Tunable node lock granularity and lock escalation

Both the SR lock, which locks an entire sub-tree in the taDOM tree in shared mode, and the X lock, which locks an entire sub-tree exclusively, enable tunable lock granularity and lock escalation for shared and exclusive locks, respectively. The combined use of them improves operational throughput because, due to lock escalation, the number of lock requests can be reduced enormously and, due to fine-tuned lock granularity, transaction concurrency may be increased.

To tune the lock granularity of nodes for each document, the parameter *lock depth* is introduced. Lock depth describes the lock granularity by means of the number of node levels (starting from the document node) on which locks are to be held. If a lock is re-

quested for a node whose path length to the document root is greater than the lock depth, only a SR lock for the ancestor node belonging to the lock-depth level is requested. In this way, nodes at deeper levels than indicated by lock depth are locked in shared mode using SR locks on nodes at the lock-depth level. This allows a transaction to traverse a large document fragment in read mode without acquiring any additional node locks. In the same way, several X locks can be merged to a single X lock at a higher level.

Figure 6 shows the locked taDOM tree cutout of Figure 5 where the effect of the lock-depth parameter is illustrated. With lock depth equal to 3, the NR locks of transaction T_3 beneath the *<title>* nodes are replaced by SR locks for the *<title>* nodes because the *<title>* nodes reside in node level 3 in this example. The IX, CX, and X locks of transaction T_1 beneath the *<author>* node are replaced by an X lock for the *<author>* node. As a prerequisite, this X lock requires a CX lock for the ancestor *<book>* node and IX locks for the *<bib>* node and the document root as described by the acquisition protocol of X locks in Section 4.1.

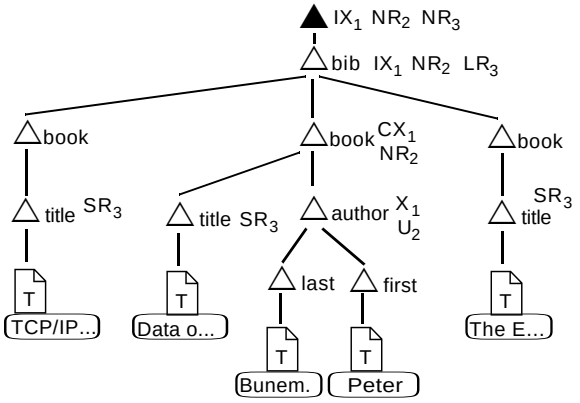


Figure 6: Coarse-grained node locks with lock depth 3

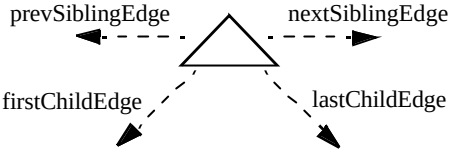
In a similar way, lock escalation can be realized. To tune the lock escalation, we introduce two parameters, the *escalation threshold* and the *escalation depth*. The lock manager component scans the taDOM tree at prespecified intervals. If the manager detects a sub-tree in which the number of locked nodes of a transaction exceeds the percentage threshold value defined by the escalation threshold, the locks held are replaced with a suitable lock at the sub-tree root, if possible. Read and write locks are replaced by SR and X locks, respectively. This replacement procedure is the same as described above for the tunable lock granularity. The escalation depth defines the maximal sub-tree depth from the leaves of a taDOM tree to the scanned sub-tree root.

Obviously, there is certainly a trade-off to be observed for lock escalation. On the one hand, lock escalation decreases concurrency of read and write transactions, but, on the other hand, a reduction of the number of held locks and of lock acquisition operations is achieved saving lock management overhead.

4.3 Navigation Locks

So far, we have discussed optimization issues for locks where the object to be accessed was specified in some declarative way (for example, with a key value or a predicate). In addition, the DOM API also provides for methods which enable the traversal of XML documents where the access is specified relative to the working node (see Section 2). In such cases, synchronizing a navigation path means that a sequence of navigational method calls or modification (IUD) operations—starting at a known node within the ta-

DOM tree—must always yield the same sequence of result nodes within a transaction. Hence, a path of nodes within the document evaluated by a transaction must be protected against modifications of concurrent transactions. For this reason, we introduce *virtual navigation edges* and corresponding *navigation locks* for element and text nodes within the taDOM tree.



	-	ER	EU	EX
ER	+	+	-	-
EU	+	+	-	-
EX	+	-	-	-

Figure 7: Virtual navigation edges and compatibility matrix for navigation locks

While navigating through an XML document and traversing the navigation edges, a transaction has to request a lock for each edge. To support such traversals efficiently, we offer the ER, EU, and EX lock modes which correspond to the well-known R/U/X locks for relational records or tables ([5], [6]). Their use observing the compatibilities shown in Figure 7 can be summarized as follows:

- An ER lock (edge read) is acquired, before an edge is traversed in read mode. For example, such an acquisition may happen by calling the *getNextSibling()* or *getFirstChildNode()* method for the nextSiblingEdge or the firstChildEdge, respectively.

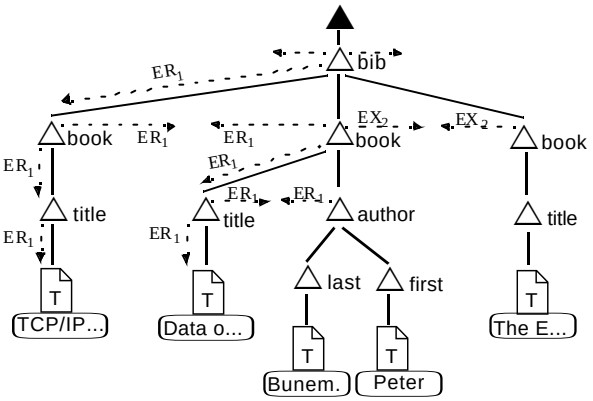


Figure 8: Navigation locks on the taDOM tree

- An EX lock (edge exclusive) is requested, before an edge is modified. It may occur when nodes are deleted or inserted. For all edges, affected by the modification operation, EX locks are acquired, before the navigation edges are redirected to their new target nodes.
- The EU lock for edge updates eases the starvation problem of write transactions as described in Section 4.1.

Figure 8 illustrates an example where navigation locks on virtual navigation edges are acquired. Transaction T_1 starts at the <*bib*> node and reads three times the first child node (this means the node sequence <*bib*>, <*book*>, <*title*>, <*text*>) in order to get the string value of the first book title. Then T_1 determines the next sibling node of the

current <*book*> node and repeats twice the first-child method to get the title of the second book. Now for this example the requested book is located, and T_1 finally gets the next sibling of the current <*title*> node which is an <*author*> node. As you can see, the synchronization concept allows another transaction T_2 to concurrently insert a new book by acquiring two EX locks for the sibling edges of the last two <*book*> nodes.

4.4 Logical Locks

The DOM API also provides the methods *getElementsByTagName(...)* and *getElementById(...)* to get elements by their tag names or by an Id attribute. The method *hasAttribute(...)* checks the existence of an attribute specified by its name. Read access to XML documents using these methods requires prevention of the phantom anomaly.

Phantoms confusing read transactions may occur by insertions of concurrent transactions. As feasible in hierarchical (multi-granularity) locking schemes, we can prevent phantoms in our hierarchical data model for XML documents by using coarser lock granules. A definite disadvantage, coarse lock granules can enhance lock conflicts or deadlock situations. Especially, the *getElementById(...)* method, which can be invoked only for the document node, would always cause a read lock for the entire document. Since this is highly undesirable, we introduce a refined concept—so-called logical locks—to prevent phantoms.

LocksTagnameQuery

TAID	lock	scopeNID	tagname
...	R/U/X	...	...

LocksAttributeQuery

TAID	lock	NID	attribute
...	R/U/X	...	...

LocksIDQuery

TAID	lock	ID
...	R/U/X	...

Figure 9: Lock tables for logical locks

To extend our synchronization concept by logical locks we introduce three lock tables which maintain the requested locks as illustrated in Figure 9. While the effects of node locks and navigation locks could be conveniently demonstrated using taDOM tree representations, the situation here is more complex and requires three separate tables for its description. Lock compatibility inside each table is handled by the conventional R/U/X compatibility matrix.

Table *LocksTagnameQuery* maintains the logical locks for element name requests. Execution of the method *getElementsByTagName(...)* requires an R lock for the corresponding transaction, tag name, and node ID of the node the method was invoked for. This ensures that the result node set of that method call does not change, because insertion of new nodes requires an X locks for the corresponding tag names in table *LocksTagnameQuery*. Deletion of nodes is still protected by node locks introduced in Section 4.1.

Table *LocksAttributeQuery* synchronizes the execution of *hasAttribute(...)* queries. The result (true or false) must not change within a transaction. This requires the acquisition of an R lock for the corresponding queried attribute name before getting the result. The other way around, inserting new attributes requires an X lock for the new attribute name

in table *LocksAttributeQuery*. Also deletion of an attribute requires an X lock for the attribute name, because the *hasAttribute(...)* method just returns the result true or false and does not call for an R lock on the corresponding attribute node.

Table *LocksIDQuery* maintains logical locks for ID attributes. Since search for ID attributes is only supported at the document level, we do not consider a scope node ID in that lock table. In the same manner as in table *LocksAttributeQuery*, searching, inserting, and deleting of attributes requires R or X locks, respectively.

Insertion of sub-trees (specified by the root node) in the taDOM tree requires the traversal of the entire sub-tree, before the actual insert operation can be performed. For each element found in the sub-tree, an X lock has to be acquired in table *LocksTagnameQuery*, and, for each ID attribute, an X lock in table *LocksIDQuery*. This is necessary, because the insertion of new elements and attributes can violate the repeatable read property for result node lists which have been filled with the *getElementById(...)* or *getElementByTagName(...)* methods.

LocksTagnameQuery

TAID	lock	scopeNID	tagname
2	R	4711	last

LocksIDQuery

TAID	lock	ID
1	R	2
1	R	4

Figure 10: Sample content of logical lock tables

Figure 10 illustrates some example contents of the logical lock tables. We consider again the XML fragment of Figure 2. At first, transaction T_1 is searching for an element with ID=2, then searching for an element with ID=4. Transaction T_2 is searching for all <*last*> nodes downwards from a <*book*> node with an assumed node ID 4711.

After having acquired the locks, another transaction T_3 is blocked when it attempts to insert a new node with ID=4 or to add a new <*author*> node with included <*last*> node to the <*book*> node with node ID 4711, because such an operation requires requesting X locks which cannot be granted. The insertion of a new <*last*> node is also forbidden in the entire sub-tree with root node ID 4711.

This example points up that lock compatibility also has to regard overlapping scopes. This means that inserting a new node requires acquisition of logical X locks for the tag name of the new node in table *LocksTagnameQuery* for each node from the document node downwards to the parent node of the new node. The deletion of existing nodes is already synchronized with read locks held by transactions accessing the nodes.

5 Related Work

So far, there are relatively few publications on synchronizing query and update access to XML documents. In [9], transactional synchronization for XML data in client/server web applications is realized by means of a check-in/check-out concept. User-defined XML fragments of stored XML documents are checked out and processed at the client side where the read and write sets of the operations are logged. Before invoking the check-in procedure, read and write sets are validated against the original document stored in the server. Upon success, the changes are propagated, whereas upon failure,

the client has to decide whether to discard its own changes or to overwrite the changes of other clients. Lam et al. [13] discuss synchronization for mobile XML data with respect to handheld clients which query XML fragments by XQL. The authors present an efficient mechanism to determine whether two XQL expressions overlap. If a conflict is detected, a resolution algorithm is applied. Hehner et al. [8] discuss, also based on the DOM API, several isolation protocols (both pessimistic and optimistic) for XML databases. Node locks are not acquired in a hierarchical context, and lock granularity is fixed for each protocol. The ID lookup problem (*getElementById()* method) as well as jumps to arbitrary nodes within an XML document are not discussed in detail. XMLTM, an efficient transaction management for XML documents, is presented in [7] where XML documents are stored in relational databases having special XML extensions and an additional transaction manager. The authors also apply a combination of hierarchical lock acquisition and logical locks. Since XMLTM does not support a specific query interface, locks to synchronize navigational operations are not coped with. They do not consider varying lock granularities, but consider reducing the degree of isolation. Performance measurements with an implementation on IBM DB2 using the XML extender are presented. Both [7] and [8] do not treat attribute nodes in a special way or discuss lock escalation to increase data throughput.

6 Conclusion and Future Work

Synchronizing concurrent access to XML documents is an important practical problem. So far, only few extensions exist in commercial database systems which poorly support concurrent document processing primarily due to coarse-grained locking. On the other hand, concurrent XML processing has not received much attention in the scientific world yet.

In our paper, we have first presented the taDOM tree, an extended data model of the DOM representation of XML documents. The taDOM tree provides lock acquisition at very fine-grained levels. Based on the taDOM tree, we have introduced a concept of combined node locks, navigation locks, and logical locks (and their compatibilities) to synchronize concurrent access tailored to the DOM API. Node locks are acquired for the actual nodes of the taDOM tree, whereas navigation locks are acquired on virtual navigation edges to synchronize operations on the navigation paths. In addition, logical locks are introduced to prevent the phantom problem. The combination of the three lock types not only provides isolation for transaction processing with the DOM API, but also offers rich options to enhance transaction concurrency, for example, tunable lock granularity and lock escalation in order to increase throughput. Whether our concept is expressive enough to synchronize non-navigational APIs (like XQuery which works on node collections)—only by using node locks and logical locks—is part of future work.

Our next step is a system implementation in order to evaluate the synchronization concept and the locking behavior for different workloads. We plan to explore the effects on concurrency and performance for varying lock granularities and lock escalation thresholds. Future steps include the conjunction of XML synchronization with native XML storage methods and joined transactional processing of XML and relational data. Such

a combined processing asks for the extension of the synchronization concept to provide the isolation property also for collection-based query languages like XQuery.

References

[1] World Wide Web Consortium. Extensible Markup Language (XML) 1.0 (2nd Ed.). W3C Recommendation (Oct. 2000)

[2] T. Härder and A. Reuter. Principles of Transaction-Oriented Database Recovery. ACM Computing Surveys 15(4), (Dec. 1983) 287-317

[3] World Wide Web Consortium. Document Object Model (DOM) Level 2 Core Specification, Version 1. W3C Recommendation (Nov. 2000)

[4] World Wide Web Consortium. XML Query Use Cases. W3C Working Draft (Nov. 2002)

[5] J. Gray and A. Reuter. Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers Inc. (1993)

[6] T. Härder and E. Rahm. Database Systems—Concepts and Techniques of Implementation. (in German) Springer (2001)

[7] T. Grabs, K. Böhm and H.-J. Schek. XMLTM: Efficient Transaction Management for XML Documents. Proc. 11th Int. Conference on Information and Knowledge Management, McLean, Virginia, USA (Nov. 2002) 142-152

[8] S. Helmer, C.-C. Kanne, and G. Moerkotte. Isolation in XML Bases. Int. Report, Faculty of Mathematics and Comp. Science, Univ. of Mannheim, Germany, Volume 15 (2001)

[9] S. Böttcher and A. Türling. Transaction Synchronization for XML Data in Client-Server Web Applications. Proc. Workshop on Web Databases, Annual Conf. of the German and Austrian Computer Societies, Vienna (2001) 388-395

[10] D. Florescu, D. Kossmann. A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in a Relational Database. Rapport de Recherche, No. 3680, INRIA, Rocquencourt, France (1999)

[11] I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and Querying Ordered XML Using a Relational Database System. Proc. ACM SIGMOD, Madison, Wisconsin (June 2002) 204-215

[12] M. Yoshikawa and T. Amagasa. XRel—A Path-Based Approach to Storage and Retrieval of XML Documents using Relational Databases. ACM Transactions on Internet Technology 1(1), (Aug. 2001) 110-141

[13] F. Lam, N. Lam, and R. Wong. Efficient Synchronization for Mobile XML Data. Proc. 11th Int. Conference on Information and Knowledge Management, McLean, Virginia, USA (Nov. 2002) 153-160

[14] T. Tesch, P. Fankhauser, and T. Weitzel. Scalable processing of XML with Infonyte-DB. Wirtschaftsinformatik-Zeitschrift 44(5) (in German), (2002) 469-475

[15] The Apache XML Project. Xerces2 Java Parser. http://xml.apache.org/xerces2-j/index.html.

[16] Infonyte-DB Version 3.0.0. User Manual and Programmers Guide. http://www.infonyte.com.