A Lock Manager for Collaborative Processing of Natively Stored XML Documents

Michael P. Haustein, Theo Härder
University of Kaiserslautern, 67653 Kaiserslautern, Germany
{haustein,haerder}@informatik.uni-kl.de

Abstract

Today, neither transactional provisions, in general, nor concurrency control, in particular, of DBMS-based processing are tailored to the specific needs of large and collaboratively used XML documents. Existing DBMSs more or less offer strictly serialized operations on them. To gain great progress in this area, we have implemented the XTC testbed as an (O)RDBMS-connected native XML database management system to empirically explore fine-granular concurrency control on XML documents which is, at the same time, adjusted to the specific needs of their APIs. In this paper, we give a comprehensive description of locking protocols for direct and navigational access to individual nodes of XML documents. Then, we focus on the implementation concepts for our lock manager which realizes this functionality in XTC, before we characterize its performance behavior on a rich spectrum of locking protocols using selective measurements in real applications.

1 Motivation

XML applications dramatically grow in number and complexity. At the same time, each application is expected to process increasing data volumes under tight schedules. Collaborative workflows in these application domains require concurrent read as well as write access to such XML data [15].

Currently available relational or object-relational database management systems ((O)RDBMSs) only manage structured data well. There is no effective and straightforward way for handling XML data. A "brute-force" mapping uses "long fields" or CLOBs where individual and direct access to single XML document nodes (elements or attributes) is not possible. Alternatively, an innumerable number of algorithms maps semi-structured XML data to structured relational database tables and columns (the so-called „shredding"). In any case, there are no specific provisions to process transactions on XML documents and, at the same time, to efficiently provide the ACID properties [10]. Especially isolation of concurrent transactions in RDBMSs is tailored to the relational data model and does not take the semi-structured data model and the typical XML document processing (XDP) interfaces into account. CLOBs or "shredded" mappings of XML documents to relational tables may cause disastrous locking behavior, in particular, if relational systems lock entire pages or even entire tables as their minimal lock granularity.

RDBMS-based approaches as XMLTM [7] cope with this problem by using a layer on top of an existing DBMS which executes the client-side transaction operations within self-managed transactions which have to be processed (under lower isolation levels) on an RDBMS thereby confined to the existing "relational" lock modes. The DGLOCK concept of XMLTM, for example, isolates transactions by managing path locks on a DataGuide structure and, in this way, provides for concurrent path-based transaction processing to the client applications. However, it cannot support ID-based access and position-based predicates and is not tailored to fine-grained

navigational access. Another path-oriented protocol (without existing implementation) is proposed in [4, 5] which also seems to be limited as far as the full expressiveness of XPath predicates and direct jumps into subtrees are concerned.

Native XML database systems promise tailored processing of XML documents, but most of the systems published in the DB literature are designed for efficient document retrieval and not for frequently concurrent and transaction-safe document modifications [12, 13]. This primarily results from the numbering schemes used to identify XML elements. These schemes allow for very fast computation of structural dependencies, but modifications of the document structure often lead to renumeration of large document parts.

A rare example of an update-oriented system, Natix is designed to support concurrent modifications [6] using the DOM interface. For transaction isolation, it uses special locks acquired on database records which may contain subtrees of the stored XML document, that is, quite large units of individual access. By acquiring locks on document parts in the neighborhood of the actually processed subtree or node, navigational paths of transactions inside the XML document can also be guarded [11].

To our knowledge, Natix embodies the only competing approach which is also navigation oriented. In contrast, our approach applies different solutions for data storage and transaction isolation. We aim at the adequate support of all known XDP interfaces (detailed in Section 2) and for the fine-grained isolation (by elements or attributes) of XML documents. For this reason, we have implemented XTC, an (O)RDBMS-connected XML DBMS, called XDBMS for short, as a testbed for empirical concurrency control on XML documents [9]—a research area hardly explored in universities or industry so far. Concurrent access is supported by locks tailored to the taDOM tree—a data model which extends the DOM tree [15]—as outlined in Section 3 and 4, thereby providing tunable, fine-grained lock granularity as well as navigational transaction paths inside an XML document. In Section 5, we outline the implementation concepts of our lock manager and give a series of performance measurement for the rich and complex operations of the lock manager in Section 6, before we summarize our results and conclude in Section 7.

2 Desirable APIs for XML Documents

Currently, there are a number of application programming interfaces (APIs) already standardized by international organizations. They are designed for the coverage of different application domains with varying operational demands. Most prominent are the *Simple API for XML* (SAX [2]) which gives the user event-based read-only access to an XML document and the *API for the Document Object Model* (DOM API) which offers method-based access to XML documents for query, update, and navigation tasks [15]. Its expressive power is characterized by the indicative names of its methods partially listed in Figure 1.

Other important APIs are specified by the XPath and XQuery standards [3, 1]. Their objectives are primarily related to addressing entire XML document parts by a (possibly mixed navigational and) declarative path expression and to transform query results into user-defined XML structures. Hence, this brief discussion already reveals that all these requirements at the language level of the various access models to XML data should be considered when designing adjusted concurrency control methods for collaborative use of XML documents. Most important is the provision of tailored locking support of node-based declarative, path-oriented as well as navigational access to single and typically very large XML documents. Hence, such a unified support of concurrency control is our ultimate goal.

Query operations	Update operations
– on contents:	– on contents:
Element getElementById (*String*)	*void* insertData (*Int, String)*
NodeList getElementsByTagName (*String*)	*void* appendData (*String*)
boolean hasAttribute (*String*)	*void* deleteData (*Int, Int*)
– on structure (navigation):	*void* replaceData (*Int, Int, String*)
NamedNodeMap getAttributes()	*void* setNodeValue (*String*)
NodeList getChildNodes ()	*void* removeAttribute (*String*)
Node getFirstChild ()	– on structure:
Node getLastChild()	*Node* insertBefore (*Node1*, *Node2*)
Node getNextSibling()	*Node* removeChild (*Node*)
Node getPreviousSibling()	*Node* replaceChild (*Node1*, *Node2*)
Node getParentNode()	*Node* appendChild *(Node)*

Figure 1. Methods for the DOM API

3 A Storage Model for XML Documents

Efficient and effective synchronization of concurrent access to an XML document is greatly facilitated if we use a specialized internal representation which enables fine-granular locking. For this reason, we will introduce two new node types: attributeRoot and string. This representational enhancement does not influence the user operations and their semantics on the XML document, but is solely exploited by the lock manager to achieve certain kinds of optimizations when an XML document is modified in a cooperative fashion. As a running example, we, therefore, refer in the following to an XML document which is slightly enhanced for our purpose to a so-called taDOM tree [8], as shown in Figure 2.

AttributeRoot separates the various attribute nodes from their element node. Instead of locking all attribute nodes separately when *getAttibutes()* is invoked, the lock manager obtains the same effect by a single lock on attributeRoot. Hence, such a lock does not affect parallelism, but leads to more effective lock handling and, thus, potentially to better performance. A string node, in contrast, is attached to the respective text or attribute node and only contains the value of this node. Because reference to that value requires an explicit invocation of *getValue()* with a preceding lock request, a simple existence test on a text or attribute node avoids locking such nodes. Hence, a transaction which is only navigating across such nodes will not be blocked, although a concurrent transaction have modified them and may still hold exclusive locks on them.

Essential for the locking performance is a suitable storage structure for taDOM trees supporting a flexible storage layout which allows a distinguishable (separate) node representation of all node types to achieve fine-grained locking. Furthermore, fast access to and identification of all nodes of an XML document is mandatory to enable efficient processing of direct-access methods (e. g., *getElementById()*) as well as navigational methods (e. g., *getNextSibling()*). Separate node representation together with access and identification of all XML document nodes, on the other hand, are prerequisites of fine-grained and, therefore, effective concurrency control.

For this reason, we have designed and implemented an XDBMS [9] which embodies a multi-layered architecture and, most important to our discussion, which offers a native storage structure for XML documents tailored to our objectives. In summary, our storage mechanism offers an extensible file structure as a container of single XML documents such that updates of an XML document (by IUD operations) can be performed on any of its nodes. We have shown that a very high degree of storage occupancy (> 96%) for taDOM trees is achieved under a variety

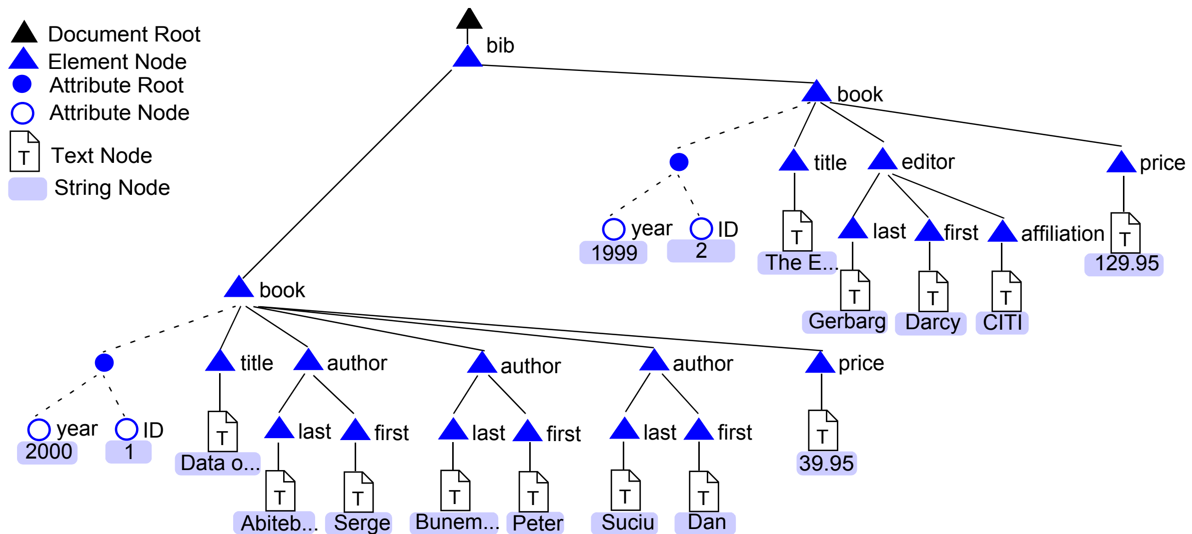


Figure 2. A sample taDOM tree

of different update workloads [9]. Fast (indexed) access to each node is provided by a variant of a B*-tree tailored to our requirements of node identification and direct or relative location of any node.

4 Supporting Flexible and Fine-Granular Concurrency Control

So far, we have explained the newly introduced node types and how fast and selective access to all nodes of an XML document can be guaranteed. In a concurrent environment, the various types of XML operations have to be synchronized using appropriate protocols entirely transparent to the different XDP interfaces supported. Because the various XDP interfaces not only allow—at the language level—declarative document access as well as navigation along nodes starting from the document root, but also enable jumps "out of the blue" to an arbitrary node within the document, locks must be *automatically* acquired in either case for the path of ancestor nodes. The currently accessed node is called *context node* in the following.

The lock modes depend on the type of access to be performed, for which we have tailored the *node lock* compatibilities [8]; we repeat them to make the paper comprehensible. When an XML document has to be traversed by navigational methods, then the actual navigation paths also need strict synchronization. This means, a sequence of method calls must always obtain the same sequence of result nodes. These mechanisms enable the isolation levels *repeatable read* and, when using coarse lock granularities, *serializable*. Our first ideas to prevent *phantoms* by fine-granularity locking are sketched in [8] and are not refined here.

4.1 Node Locks

While traversing or modifying an XML document, a transaction has to acquire a lock in an adequate mode for each node before accessing it. Because the nodes in an XML document are organized by a tree structure, the principles of multi-granularity locking schemes can be applied. The method calls of the different XDP interfaces used by an application are interpreted by the lock manager to select the appropriate lock modes for the entire ancestor path. Such tree locking is similar to multi-granularity locking in relational environments (SQL) where intention locks communicate a transaction's processing needs to concurrent transactions. In particular, they

prevent a subtree *s* from being locked in a mode incompatible to locks already granted to *s* or subtrees of *s*. However, there is a major difference, because the nodes in an ancestor path are part of the document and carry user data, whereas, in a relational DB, user data is exclusively stored in the leaves (records) of the tree (DAG) whose higher-level nodes are formed by organizational concepts (e. g., table, segment, DB). For example, it makes perfect sense to lock an intermediate XML node *n* for reads, while in the subtree of *n* another transaction may perform updates. For this and other reasons, we differentiate the read and write operations thereby replacing the well-known (IR, R) and (IX, X) lock modes with (NR, LR, SR) and (IX, CX, X) modes, respectively. As in the multi-granularity scheme, the U mode plays a special role because it permits lock conversion. Here, we only summarize the compatibilities of locks acquired on the same node by separate transactions (Figure 3a):

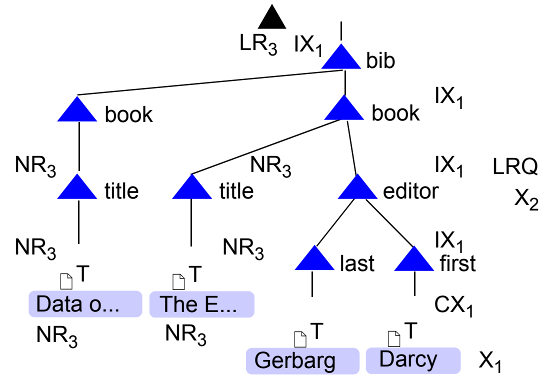
- An NR lock mode (node read) is requested for reading the context node. To isolate such a read access, an NR lock has to be acquired for each node in the ancestor path. Note, the NR mode takes over the role of IR together with a specialized R, because it only locks the specified node, but not any descendant nodes.
- An IX lock mode (intention exclusive) indicates the intent to perform write operations somewhere in the subtree (similar to the multi-granularity locking approach), but not on a direct-child node of the node being locked (see CX lock).
- An LR lock mode (level read) locks the context node together with its direct-child nodes for shared access. For example, the method *getChildNodes()* only requires an LR lock on the context node and not individual NR locks for all child nodes. Similarly, an LR lock, requested for an attributeRoot node, locks all its attributes implicitly (to save lock requests for the *getAttributes()* method).
- An SR lock mode (subtree read) is requested for the context node *c* as the root of subtree *s* to perform read operations on all nodes belonging to *s*. Hence, the entire subtree is granted for shared access. An SR lock on *c* is typically used if *s* is completely reconstructed to be printed out as an XML fragment.
- A CX lock mode (child exclusive) on context node *c* indicates the existence of an X lock on some direct-child node and prohibits inconsistent locking states by preventing LR and SR lock modes. In contrast, it does not prohibit other CX locks on *c*, because separate direct-child nodes of *c* may be exclusively locked by concurrent transactions.
- A U lock mode (update option) supports a read operation on context node *c* with the option to convert the mode for subsequent write access. It can be either converted (downgraded) to a read lock if the inspection of *c* shows that no update action is needed or (upgraded) to an X lock after all existing read locks on *c* are released. Note, the asymmetry in the compatibility definition among U and (NR, IX, LR, SR, CX) which prevents granting further read locks on *c*, thereby enhancing protocol fairness, that is, avoiding transaction starvation.
- To modify the context node *c* (updating its contents or deleting *c* and its entire subtree), an X lock mode (exclusive) is needed for *c*. It implies a CX lock for its parent node and IX locks on all other ancestors.

Note again, this differing behavior of CX and IX locks is needed to enable compatibility of IX and LR locks and to enforce incompatibility of CX and LR locks.

Fig. 3b represents a cutout of the taDOM tree depicted in Figure 2 and illustrates the result of the following example: Transaction T_1 starts modifying the value *Darcy* and, therefore, acquires an X lock for the corresponding string node. The lock manager complements this action

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

a) Compatibility matrix



b) Locking example

Figure 3. Node locking for the taDOM tree

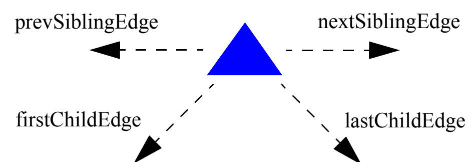
by accessing all ancestors and by acquiring a CX lock for the parent and IX locks for all further ancestors. Simultaneously, transaction T_2 wants to delete the entire <*editor*> node including the string *Gerbag* for which T_2 must acquire an X lock. This lock request, however, cannot be immediately granted because of the existing IX lock of T_1. Hence, T_2—placing its request in the lock request queue (LRQ: X_2)—must synchronously wait for the release of the IX lock of T_1 on the <*editor*> node. Meanwhile, transaction T_3 is generating a list of all book titles and has, therefore, requested an LR lock for the <*bib*> node to obtain read access to all direct-child nodes thereby using the level-read optimization. To access the title strings for each <*book*> node, the paths downwards to them are locked by NR locks. Note, LR_3 on <*bib*> implicitly locks the <*book*> nodes in shared mode and does not prohibit updates somewhere deeper in the tree. If X_2 is eventually granted for the <*editor*> node, T_2 gets its CX lock on the <*book*> node and its IX locks granted up to the root.

4.2 Navigation Locks

So far, we have discussed optimization issues for locks where the node to be accessed was specified by its unique ID. In addition, the DOM API also provides for (>20) methods which enable the traversal of XML documents where access is specified relative to the context node. 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 taDOM 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. Assume in Figure 2, a transaction T navigates through all or a range of <*book*> nodes and wants to be isolated from concurrent inserts of new <*book*> nodes. Of course, we have already introduced some lock modes which enable in this situation perfect, but (too) expensive isolation caused by (too) large lock granules. For example, if we acquire an LR lock on

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

a) Compatibility matrix



b) Virtual navigation edges on an element-node

Figure 4. Locking navigational operations in a taDOM tree

the <*bib*> node, all <*book*> nodes are implicitly granted in shared mode. An SR lock on <*bib*> would even prohibit updates on the entire document. We, however, want to support a solution only using minimal lock granules, that is, node locks of mode NR. Therefore, we introduce *virtual navigation edges* for element and text nodes within the taDOM tree (Figure 4b) which are locked in addition to their confining nodes.

While navigating through an XML document and traversing the navigation edges, a transaction has to request a lock for each edge., in addition to the node locks (NR) for the nodes visited. Note, these edges are logical objects which are not materialized but embodied by their confining nodes. Because each navigation step only performs local operations (first/last, next/previous) to a sibling or child of the context node *c*, the R/U/X locks known from relational records or tables are sufficient. Traversal operations between nodes need bidirectional isolation: For example, if *getNextSibling()* is invoked on node *c* and delivers node *n*, then, as a first step, the next-sibling edge of *c* is locked. In addition, we must lock the previous-sibling edge of *n* to prohibit path modifications between *n* and *c* through another transaction via node *n*. To support such traversals efficiently, we offer ER, EU, and EX lock modes similar to R/U/X. Their use observing the compatibilities shown in Figure 4a can be summarized as follows:

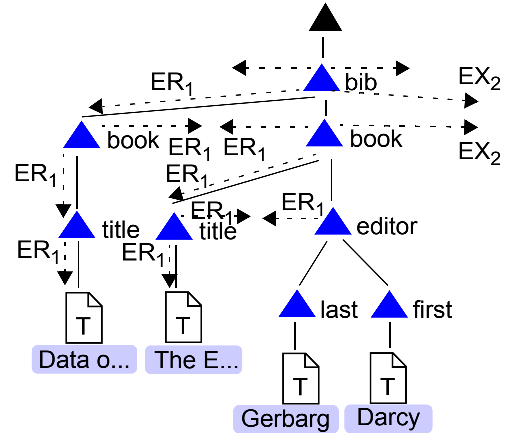


Figure 5. Use of navigation locks

- An ER lock mode (edge read) is needed for an edge traversal in read mode, e. g., by calling the *getNextSibling()* or *getFirstChild()* DOM method for the nextSiblingEdge or firstChildEdge, respectively.
- An EX lock mode (edge exclusive) enables an edge to be modified which may be needed 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 mode (edge update) eases the starvation problem of write transactions (see lock mode U in Section 4.1).

Figure 5 illustrates navigation locks on virtual navigation edges. To keep Figure 5 comprehensible, we do not show node locks, e.g., NR or CX. Transaction T_1 starts at the <*bib*> node and reads three times the first-child node (that is, the node sequence <*bib*>, <*book*>, <*title*>, <*text*>) to get the string value (Data o...) of the first book title. Then T_1 refers to the next-sibling node of the current <*book*> node and repeats twice the first-child method to get the title of the second book. At this point, the requested book is located, and T_1 finally gets the next sibling of the current <*title*> node which is the <*editor*> node. Apparently, our protocol allows concurrent transaction T_2 to append a new book by acquiring EX locks for the next-sibling edge of the last <*book*> node and for the last-child edge of the <*bib*> node. Of course, T_2 has to protect its ancestor path in a sufficient mode—its CX lock on <*bib*> is compatible with the NR lock of T_1.

5 Realization Concepts for the XML Lock Manager

As already stated, the various XML language models enable declarative access to arbitrary document parts, node navigation, and direct node access without giving any hints for operation synchronization. Therefore, synchronization of concurrent operations has to be “automatically” ac-

complished by the lock manager which is responsible for the acquisition and maintenance of locks, processing of the quite complex locking protocols and their adherence to correctness criteria, as well as optimization issues such as adequate lock granularity and lock escalation. In particular, the lock manager has to protect directly accessed nodes, edges traversed by navigational operations, and subtrees in the XML document.

5.1 Identifying Nodes

A node can be directly accessed, for example, by *getElementById()* if its ID is known. In such a case, the entire ancestor path must be sufficiently locked together with the context node. Hence, setting a lock in the tree typically results in an entire *locking path*. This up-to-the-root locking procedure is performed as follows: If an ancestor path is traversed the first time and if the IDs of the ancestors are not present in the so-called parent index, the document tree has to be accessed and searched for all ancestor records. The IDs of these records are saved in the parent index (on-demand indexing [9]). Hence, future traversals of this ancestor path can be processed via the parent index only.

Another type of node access is based on navigation—next/previous sibling, first/last child, parent—starting from the context node. Because the IDs of such relatively addressed objects are not supplied by the application, the related records have to be accessed to identify the nodes if no appropriate index is present. Depending on the size and structure of the document tree, this identification may be expensive and may require searching of large portions of the tree, that is, scanning of a large set of pages involving physical I/O. Again, on-demand indexing of structural relationships, for example, next-sibling index or last-child index, helps to optimize locking performance in a similar way as the parent index.

In any case, the lock manager protects the context node c by providing the weakest possible locking path for c. Hence, a read operation on c results in a read lock mode on c (that is, NR, SR, LR) and NR locks on all ancestors, whereas a U lock mode implies IX locks on all ancestors. The weakest possible locking path for all write operations consists of an X lock on c, a CX lock on its parent p, and IX locks on all other ancestors.

5.2 Semaphore Tables

The actual lock management is based on a semaphore concept and is realized by so-called semaphore tables. Because we need to synchronize objects of varying types occurring at diverse system layers (e.g., pages and XML-document-related objects such as nodes, edges, and indexes), Which exhibit incomparable lock compatibilities, very short to very long lock durations, as well as differing access frequencies, we decided to provide specialized semaphore tables for them (and not a common one). Size and contents of these tables can be configured. They are statically allocated at system start-up time in main memory and maintain the specific semaphores on objects that are identified by unique identifiers.

A semaphore table maintains the semaphores (locks) dynamically acquired for a specified maximum number of transactions and lockable objects (e.g., XML nodes or database buffer pages). Semaphores types (lock modes) are assigned to a semaphore table by using the corresponding compatibility matrices. The dynamically acquired semaphores are maintained by a 2-dimensional static array which makes request and release of semaphores very fast. To save memory space, an encoded and thereby compact representation of the entries for semaphores and transaction IDs can be used.

As illustrated in Figure 6, each transaction T is mapped by its TAID to a column within the semaphore table whereas the rows assigned to the objects are determined by a hash function. Operations are provided on behalf of T to add or replace semaphores on an object and to remove it after completing the processing of the object. The row for an object is determined when the first semaphore is acquired for it. “Collisions” while assigning a row to an object are resolved by choosing the next free row as a collision handling technique. The column determining together with the object row the array element keeping the semaphore is selected by TAID. Hence, a transaction can only keep a single semaphore on a given object and sometimes has to replace it by another one of stronger mode (see lock conversion). Furthermore, the ID of an object (e.g., an XML node) must be stable for the entire time period the object is locked (even if the object is modified while keeping an exclusive lock). If the last semaphore held on an object is removed, the object ID is deleted and the corresponding row is released. Hence, the row can be reused for future lock requests.

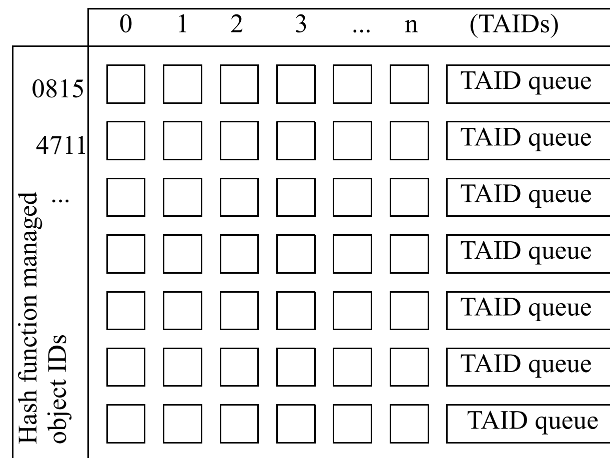


Figure 6. Semaphore table layout

The set of granted locks per object is represented by the semaphore-table row assigned to an object—the so-called object control block (OCB). All locks acquired by a transaction are represented by the respective column (indirectly) indexed by its TAID. Hence, they can be rapidly located when released at transaction commit. Static allocation of fixed-size semaphore tables enables fast lock table operations; it is, however, wasting memory space when the table is sparsely occupied. Although acceptable for our XTC testbed, we will refine our implementation using dynamically allocated linked OCBs for high transaction loads.

A lock request is performed in a FIFO manner to guarantee scheduling fairness when accessing objects (*request mode*). If the lock request queue (LRQ or TAID queue) of a requested object is not empty, the transaction is deactivated and waits in the respective LRQ until the lock can be granted. The setting of a semaphore requires a compatibility check which is performed by means of the compatibility matrix provided at semaphore table initialization. Either the requested semaphore is compatible to all semaphores already held on the object or the requested semaphore is not compatible to one or more semaphores already set. If compatible, the requested semaphore is immediately set on behalf of the invoking transaction. Otherwise, the semaphore-requesting transaction is added to the object’s LRQ and has to “sleep” until the incompatible semaphores are removed or replaced by the transactions owning them. Additionally, the pending transaction is entered into a wait-for-graph (WfG) enabling deadlock detection. Such centralized deadlock detection is necessary because cyclic waiting relationships can occur across several semaphore tables residing in different XDBMS layers (e.g. T1 is waiting for T2 because of a *node* lock request and T2 is waiting for T1 because of a *page* fix request).

5.3 Lock Management

Lock management is performed by the lock manager which initializes 6 different semaphore tables to maintain the locks for the XML nodes, the previous and next sibling edges, the first and last child edges, and the index accesses (e.g., using a B-tree to determine the physical address

of a node specified by its unique ID (see Section 3)). An additional semaphore table is created by the DB buffer manager to maintain database page read and write fixes. The principal and most frequent lock management operations are

- allocate a new object: locate a free row in the semaphore table
- remove an object: release the related row
- assign a semaphore: test compatibility (always limited to existing semaphores in the respective row)
- replace an existing semaphore: similar to semaphore assignment
- remove a semaphore: check TAID queue whether pending transactions can be resumed
- remove a transaction: remove all semaphores in the corresponding column.

Because of the static matrix implementation of semaphore tables, the elementary operation to access a table element is very fast. Hence, all these principal operations need one or more table element accesses and take advantage of the chosen implementation.

Locks on indexes or pages are "short" locks (kept only for the actual page or index access of an operation), whereas acquisition and release of node and edge locks (which correspond to the locking protocols described in Section 4) depend on the isolation levels *none, uncommitted, committed, repeatable*, or *serializable* at which a transaction is running. Isolation level *none* means that no node or edge locks at all are requested for individual operations. This mode is used, for example, for internal transactions of the lock manager to determine hierarchical relationships of nodes (e.g., a lock request on node n_1 requires a lock request on parent node n_2). At isolation level *uncommitted* only write locks (IX, CX, X, and EX) are requested and are immediately released at the end of the operation (hence, it is possible to access uncommitted data). Processing only *committed* data requires keeping "long" write locks (until transaction commit) and "short" read locks (NR, LR, SR, U, ER, and EU). Isolation level *repeatable* keeps all locks up to the end of the running transaction and guarantees that multiple read operations on the same object always obtain the same result data within a transaction. *Serializable* avoids, in addition to isolation level *repeatable,* phantom anomalies to happen.

The lock manager provides methods for the acquisition of the different lock types on DB objects (XML nodes, edges among XML nodes, index entries, and database pages) and forwards the lock requests to the adequate semaphore tables. Additionally, a single node lock request for working node *w* can result in further lock requests for nodes along the ancestor path or in lock requests for direct-child nodes of *w* (see the locking protocols described in Section 4). These additional lock requests are triggered by the lock manager; the requested nodes are identified by an internal transaction. Furthermore, with a specific lock-depth parameter [8], the lock request on a node below the given lock-depth level must be transformed by the lock manager into a lock request on the first ancestor node located at the specified lock-depth level.

5.4 Node Lock Conversion

If a transaction T does not hold a lock on object *o*, locking is performed in request mode which strictly observes scheduling fairness. In contrast, if T already holds a lock on o and wants to upgrade/downgrade the lock or requests an additional lock for *o* in another locking path, lock conversion is performed in the so-called *convert mode*. The latter case frequently occurs in trees, because operations of the same transaction on separate parts of the tree imply locking paths which are overlapping closer to the root. Downgrading a lock never causes problems, but up-

grading needs some special considerations. In convert mode, a lock is directly probed irrespective of transaction requests blocked in the corresponding LRQ. If the requested lock mode is compatible with all granted locks on *o*, conversion is performed. Otherwise, lock conversion has to be delayed. However, placing T at the end of the corresponding LRQ would provoke a deadlock situation. Therefore, we circumvent scheduling fairness and place T always at the top of the LRQ.

Lock mode U is an obvious candidate for lock conversion. Assume that transaction T requires a single U lock on node *c* somewhere in the taDOM tree. Before the lock can be granted, the lock manager has to set on behalf of T IX locks on all ancestor nodes of *c*. If later no write operation is needed, the U lock can be downgraded to an NR lock on *c*. To achieve minimal obstruction of concurrent transactions, we could reduce all related IX locks to NR, in principle. Cost effectiveness (see argument at the end of Section 5.4), however, does not allow to weaken lock protection on the ancestor nodes. More difficult is the situation where the U lock is upgraded to X. This action implies a CX lock on the parent *p* of *c* and IX locks (already present) on all other ancestors. As stated in Figure 3a, *p* may already hold (NR, IX, LR, CX) locks of other transactions. Therefore, CX of T is tested against all granted locks on *c* and p. As soon as incompatibility is found, e. g., with LR, T must wait in the related LRQ.

A second form of lock conversion occurs as follows: If T requests another lock on *o* (in an overlapping locking path), without lock conversion we would have to keep two locks for T. In general, *k* locks per transaction and object are conceivable. This proceeding would require larger OCBs and a more complex run-time inspection algorithm checking for lock compatibility. Therefore, also enforced by the chosen semaphore-table implementation, we replace all locks of a transaction per node with a single lock in a mode giving sufficient isolation.

The corresponding rules are specified by the *lock conversion matrix* in Figure 7 which determines the resulting lock for context node *c*, if a transaction already holds a lock (matrix header row) and requests a further lock (matrix header column) on *c*. A lock l_1 specified by an additional subscripted lock l_2 (e. g., CX_{NR}) means that l_1 has to be acquired on *c* and l_2 has to be acquired on each direct-child node of *c*.

	-	NR	IX	LR	SR	CX	U	X
NR	NR	-	IX	LR	SR	CX	NR	X
IX	IX	IX	-	IX_{NR}	IX_{SR}	CX	IX	X
LR	LR	LR	IX_{NR}	-	SR	CX_{NR}	LR	X
SR	SR	SR	IX_{SR}	SR	-	CX_{SR}	SR	X
CX	CX	CX	CX	CX_{NR}	CX_{SR}	-	CX	X
U	U	U	U	U	U	U	-	X
X	X	X	X	X	X	X	X	-

Figure 7. Lock conversion matrix

As an example, assume that a user starts a transaction T requesting all child nodes of *c* which results in acquiring an LR lock on *c*. LR mode locks *c* and all direct-child nodes in shared mode. After that, the user wants to delete one of the previously determined child nodes. Therefore, T acquires an X lock on the corresponding child node and—applying the locking protocol—this requires the acquisition of a CX lock on *c* which already holds the LR lock. Using rule CX_{NR}, T has to convert the LR lock on *c* to a CX lock and to acquire an NR lock on each direct-child node of *c* (except the child node which is already locked for deletion by an X lock). For large child sets, we provide some kind of lock escalation [8].

Lock conversion works fine in the isolation levels *repeatable read* or stronger. Isolation level *committed*, however, releases read locks at the end of each operation and, therefore, requires releasing of read locks represented by a single converted lock. Because the stronger mode of write locks covers the effects of read locks, such converted locks don't need to be converted back. In principle, IX-converted locks resulting from (NR, IX) requests of the same transaction could be

reduced to NR, if a U lock somewhere down the tree has to be downgraded to a read lock and afterwards released, to guarantee the weakest lock modes possible along the ancestor chain. However, this proceeding would require the logging of the transaction's lock history on a node and, therefore, contradict our lock conversion policy. Hence, we do not reduce the mode of converted locks.

5.5 Deadlock Detection

Although we manage 6 semaphore tables in our system, we can represent the existing wait relationships among concurrent transactions in a single WfG. Each transaction can at most wait for a single lock request, for which it is blocked in the LRQ of the respective object. In such a situation, it is represented by a node in the WfG. When a lock is released on an object, its LRQ is checked whether blocking situations are disappeared or not. If successful, the respective set of transactions are removed from the WfG; after having acquired their locks, they resume processing.

Whenever a transaction enters the WfG, a blocking situation and possibly a deadlock has happened. One solution is to eagerly search for a cycle in the WfG starting from and limited to the newly inserted node. Because, however, most blocking situations do not lead to deadlocks and can be resolved by waiting, this eager approach is usually not cost-effective. Therefore, we have implemented a lazy and more economic solution in which all nodes of the WfG are checked periodically (e. g., in intervals of 5 secs) for cycles. A deadlock is broken up by selecting the transaction in a detected cycle as the "victim" which has performed the lowest number of updates. After its rollback the WfG is tested for further cycles. As an additional advantage, deadlock detection can be executed by a separate thread and does not burden a thread dedicated to transaction processing.

6 Performance Measurement of Lock Management

The goal of this performance evaluation is to learn about the behavior of lock management on XML documents and to determine the cost to be attributed to lock acquisition and release. Because the weakest possible locking paths have to be established for each node access in a document tree, more overhead is anticipated as compared to locking flat structures in relational systems. Furthermore, direct node access and navigational operations imply that parent/sibling/child nodes have to be known by their IDs before a lock can be set. Such locking often requires access to the physical document representation to look up the corresponding records and figure out their IDs. Of course, these accesses are subject to index-based optimizations for which we will identify appropriate measures and adjust them in our lock manager implementation.

First of all, we consider the basic cost of lock management described so far. For this purpose, we use the *xmlgen* tool of the XMark XML benchmark project [14] to generate a variety of XML documents consisting of 5,000 up to 25,000 individual XML nodes. The documents are stored in our native XDBMS [9] and accessed by a client-side DOM application which requests all nodes by separate RMI calls. Each document is reconstructed twice within a single transaction by two consecutive traversals in depth-first order. Hence, we can directly illustrate the differences in the behavior of isolation level *committed* (a read lock has to be acquired for each node access, even if the node has already been read before by the transaction, and has to be released directly after the read operation) and isolation level *repeatable read* (a lock is only requested for the first node access and is not released until the transaction commits).

The results of this first measurement are depicted in Figure 8. Lock management in *repeatable-read* mode needs less than 25 percent of the processing time even if 50,000 nodes are requested within one transaction (reconstructing the document consisting of 25,000 nodes twice). Although isolation level *committed* favors concurrent processing (only short read locks are acquired), the reconstruction time for the document increases dramatically, because each node request requires the acquisition and immediate release of several node locks in the ancestor path of the XML tree hierarchy. Physical node access for the look-up of their IDs is a major reason for this bad locking performance behavior which urgently needs to be fixed. Index support is one option, but a better choice would be a numbering scheme where each node ID immediately allows to infer the IDs of all its ancestor nodes.

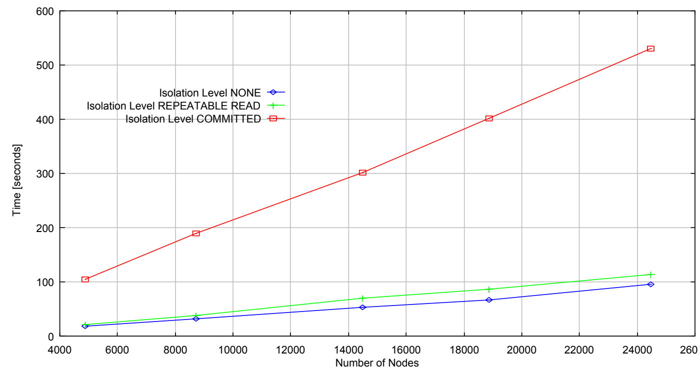


Figure 8. Document reconstruction time

For the next performance experiment, we traverse a 25,000-node document only once in *repeatable-read* mode and consider the processing time determined by different values of the lock-depth parameter. The results are shown in Figure 9. Because the document has a large fan-out at levels 4 and below, the reconstruction time and, in turn, the number of lock requests increase substantially in this range. Nevertheless, the cost for lock management does not exceed the 25-percent threshold mentioned (comparing a single document-granularity lock at lock depth 0 with a lock on each node at maximum lock depth 13).

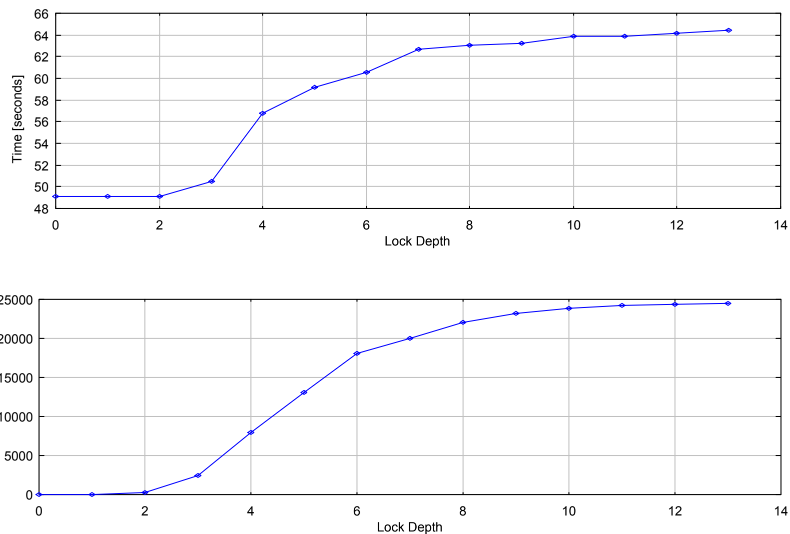


Figure 9. a) Reconstruction time b) Number of node locks

The final performance experiment gives a first impression of the advantages of node-based lock management for concurrent transaction processing on XML documents. Therefore, we extend the sample document of Figure 2 to a small library database by encapsulating 100 books into an additional <*books*> node and adding a <persons> node which includes 20 persons. The DataGuide describing the resulting XML document is shown in Figure 10.

Different transaction types simulating read/write access to XML documents are executed on the library document consisting of 1,943 nodes stored in our XDBMS. Transaction T_1 is searching for a book with a randomly selected title. This is a typical query of a library visitor. The library employees are simulated by transactions T_2, T_3, and T_4. Transaction T_2 is searching for a randomly selected person by the last name. Transactions T_3 and T_4 are simulating the lending of books. A person is randomly selected by transaction T_3 and the person's ID is added within

a new child node to a randomly selected book—lent by the person. Transaction T_4 „returns“ the book by deleting the corresponding child node from the <*book*> node.

Read transactions of type T_1 and T_2 are continuously executed for five minutes on the library document and embody a base load on the XDBMS. Concurrently, five clients are executing as many write transactions of type T_3 and T_4 as possible. A deadlock detector is scanning the waiting relationships between the running transactions every five seconds. The result of this transaction throughput benchmark analyzing the influence of the lock-depth parameter is depicted in Figure 11.

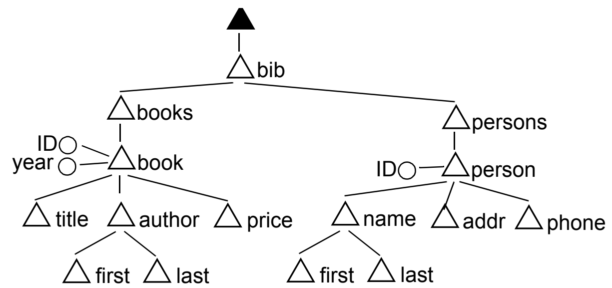


Figure 10. DataGuide of the library document

The maximum of committed write transactions is reached using a lock-depth value of 3, because the inserted and deleted nodes of transactions T_3 and T_4 are occurring at this node level and are explicitly locked. As a key advantage of our lock protocol, their sibling nodes are not affected by this lock acquisition—particularly the <*title*> node can be concurrently accessed by transaction T_1.

A higher lock-depth value decreases the number of committed write transactions a little, because a few more locks have to be acquired. A lower lock-depth value decreases the number of committed write transactions dramatically, because the coarser lock granules cause more deadlocks and suspend the exclusive lock acquisition of the write transactions until the read transactions have committed and released their read locks. Because the read transactions T_1 and T_2 are less time-consuming than the write transactions T_3 and T_4, the number of all committed transactions is decreasing with the increasing number of committed write transactions and an increasing lock-depth value (more lock acquisitions).

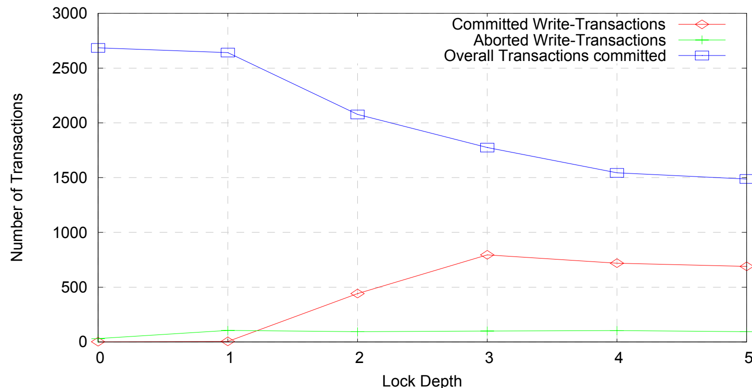


Figure 11. Transaction throughput

Again, another reason for the decreasing number of committed transactions is our simple numbering scheme which assigns sequentially ascending IDs to the XML nodes of the storage model (see Section 3). Hence, a locking path has to be determined by physical look-up. This additional burden of the lock manager reduces the system’s overall throughput.

7 Conclusions and Future Work

In this paper, we have described the design, implementation, and performance evaluation of our lock manager supporting efficient collaborative processing on XML documents. For this purpose, we have introduced our concepts enabling fine-granular concurrency control on taDOM trees representing our natively stored XML documents. We have sketched the locking protocols for direct and navigational access to individual nodes of a taDOM tree and discussed some op-

timization potential of the chosen approach. Furthermore, we have described the implementation of the lock manager, which provides its rich functionality in the XTC system, an XDBMS primarily developed as a testbed to explore XML concurrency control. The performance evaluation has revealed the locking overhead of our complex protocols, but, on the other hand, has confirmed the viability, effectiveness, and efficiency of our approach.

There are many other issues that wait to be resolved. We need to adapt our numbering scheme to allow for flexible update operations as well as locking an arbitrary XML node and its ancestor path without accessing any other node. Furthermore, we did not say much about the usefulness of optimization features offered. More effective phantom control needs to be added, before we can start to systematically evaluate the huge parameter space available for collaborative XML processing (fan-out and depth of XML trees, mix of transactional operations, benchmarks for specific application domains, degree of application concurrency, optimization of protocols, etc.)

References

1. S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, J. Siméon. XQuery 1.0: An XML Query Language. W3C Working Draft (2003)
2. D. Brownell. SAX2. O'Reilly (2002)
3. J. Clark, S. DeRose. XML Path Language (XPath) Version 1.0. W3C Recommendation (2000)
4. S. Dekeyser, J. Hidders. Path Locks for XML Document Collaboration. Proc. 3rd Conf. on Web Information Systems Engineering (WISE), Singapore, 105-114 (2002)
5. S. Dekeyser, J. Hidders, J. Paredaens. A Transaction Model for XML Databases. World Wide Web Journal 7(2): 29-57 (2004)
6. T. Fiebig, S. Helmer, C. Kanne, G. Moerkotte, J. Neumann, R. Schiele, T. Westmann. Natix: A Technology Overview. A. Chaudri et al. (eds.): Web, Web Services, and Database Systems, NODe 2002: Web and DB-Related Workshops, Erfurt, Germany, LNCS 2593, Springer, 12-33 (2003)
7. T. Grabs, K. Böhm, H.-J. Schek. XMLTM: Efficient transaction management for XML documents. Proc. Int. Conf. on Information and Knowledge Management (CIKM), McLean, Virginia, 142-152 (2002)
8. M. Haustein, T. Härder. taDOM: A Tailored Synchronization Concept with Tunable Lock Granularity for the DOM API. Proc. 7th East European Conf. on Advances in Databases and Information Systems (ADBIS), Dresden, Germany, 88-102 (2003)
9. M. Haustein, T. Härder. Fine-Grained Management of Natively Stored XML Documents. submitted (2004)
10. T. Härder, A. Reuter. Principles of Transaction-Oriented Database Recovery. ACM Computing Surveys 15(4):287-317 (1983)
11. S. Helmer, C.-C. Kanne, G. Moerkotte. Evaluating Lock-based Protocols for Cooperation on XML Documents. SIGMOD Record 33(1): 58-63 (2004)
12. H. V. Jagadish, S. Al-Khalifa, A. Chapman. TIMBER: A native XML database. The VLDB Journal 11(4): 274-291 (2002)
13. H. Schöning. Tamino—A DBMS designed for XML. Proc. 7th Int. Conf. on Data Engineering, Heidelberg, Germany, 149-154 (2001)
14. A. Schmidt, F. Waas, M. Kersten. XMark: A Benchmark for XML Data Management. Proc. 28th Int. Conf. on Very Large Data Bases, Hong Kong, China, 974-985 (2002)
15. W3C Recommendations. http://www.w3c.org (2004)