Adjustable Transaction Isolation in XML Database Management Systems

Michael P. Haustein, Theo Härder

University of Kaiserslautern[1]

Abstract. Processing XML documents in multi-user database management environments requires a suitable storage model of XML data, support of typical XML document processing (XDP) interfaces, and concurrency control (CC) mechanisms tailored to the XML data model. In this paper, we sketch the architecture and interfaces of our prototype native XML database management system which can be connected to any existing relational DBMS and provides for declarative and navigational data access of concurrent transactions. We describe the fine-grained CC mechanisms implemented in our system and give a first impression of the so achieved benefits for concurrent transaction processing in native XML database management systems.

1 Introduction

Run an experiment on available DBMSs with collaboratively used XML documents [16] and you will experience a "performance catastrophe" meaning that all transactional operations are processed in strict serial order. Storing XML documents into relational DBMSs forces the developers to use simple CLOBs or to choose among an innumerable number of algorithms mapping the semi-structured documents to tables and columns (the so-called *shredding*). In any case, there are no specific provisions to process concurrent transactions guaranteeing the ACID properties and using typical XDP interfaces like SAX [2], DOM [16], and XQuery [16] simultaneously. Especially isolation in relational DBMS does not take the properties of the semi-structured XML data model into account and causes disastrous locking behavior by blocking entire CLOBs or tables.

Native XML database systems often use mature storage engines tailored to relational structures [13]. Because their XML document mapping is usually based on *fixed numbering schemes* used to identify XML elements, they primarily support efficient document retrieval and query evaluation. Frequently concurrent and transaction-safe modifications would lead to renumeration of large document parts which could cause unacceptable reorganization overhead and degrade XML processing in performance-critical workload situations. As a rare example of an update-oriented system, Natix [5] is designed to support concurrent transaction processing, but accomplishes alternative solutions for data storage and transaction isolation as compared to our proposal.

Our approach aims at the adequate support of all known types of XDP interfaces (event-based like SAX, navigational like DOM, and declarative like XQuery) and provides the

1. Univ. of Kaiserslautern, Dept. of Comp. Science, Database and Information Systems, D-67653 Kaiserslautern, Germany. *E-mail:* {haustein | haerder}@informatik.uni-kl.de.

well-known ACID properties [7] for their concurrent execution. We have implemented the *XML Transaction Coordinator (XTC)* [9], an (O)RDBMS-connectable DBMS for XML documents, called XDBMS for short, as a testbed for empirical transaction processing on XML documents. Here, we present its advantages for concurrent transaction processing in a native XDBMS achieved by a storage model and CC mechanisms tailored to the XML data model. This specific CC improves not only collaborative XDP but also SQL applications when "ROX: Relational Over XML" [8] becomes true.

An overview of the XTC architecture and their XDP interfaces is sketched in Section 2. Concurrent data access is supported by locks tailored to the taDOM tree [10]—a data model which extends the DOM tree—as outlined in sections 3 and 4, thereby providing tunable, fine-grained lock granularity and lock escalation as well as navigational transaction path locking inside an XML document. In Section 5, we give a first impression of concurrent transaction processing gains, before we wrap up with conclusions and some aspects of future work in Section 6.

2 System Architecture and XDP Interfaces

Our XTC database engine (*XTCserver*) adheres to the widely used five-layer DBMS architecture [11]. In Figure 1, we concentrate on the representation and mapping of XML documents. Processing of relational data is not a focus of this paper.

The *file-services* layer operates on the bit pattern stored on external, non-volatile storage devices. In collaboration with the OS file system, the *i/o managers* store the physical data into extensible *container files;* their uniform block length is configurable to the characteristics of the XML documents to be stored. A *buffer manager* per container file handles fixing and unfixing of pages in main memory and provides a replacement algorithm for them which can be optimized to the anticipated reference locality inherent in the respective XDP applications. Using pages as basic storage units, the *record*, *index*, and *catalog managers* form the *access services*. The record manager maintains in a set of pages the tree-connected nodes of XML documents as physically adjacent records. Each record is addressed by a unique life-time ID managed within a B-tree by the index manager [9]. This is essential to allow for fine-grained concurrency control which requires lock acquisition on unique identifiable nodes (see Section 4). The catalog manager provides for the database metadata. The *node manager* implementing the navigational access layer transforms the records from their internal physical into an external representation, thereby managing the lock acquisition to isolate the concurrent transactions. The *XML-services* layer contains the *XML manager* responsible for declarative document access, e. g., evaluation of XPath queries or XSLT transformations [16].

At the top of our architecture, the agents of the *interface layer* make the functionality of the XML and node services available to common internet browsers, ftp clients, and the *XTCdriver* thereby achieving declarative / set-oriented as well as navigational / node-oriented interfaces. The XTCdriver linked to client-side applications provides for methods to execute XPath-like queries and to manipulate documents via the SAX or DOM API. Each API accesses the stored documents within a transaction to be started by the XTCdriver. Transactions can be processed in the well-known isolation levels *uncommitted*, *committed*, *repeatable*, and *serializable* [1].

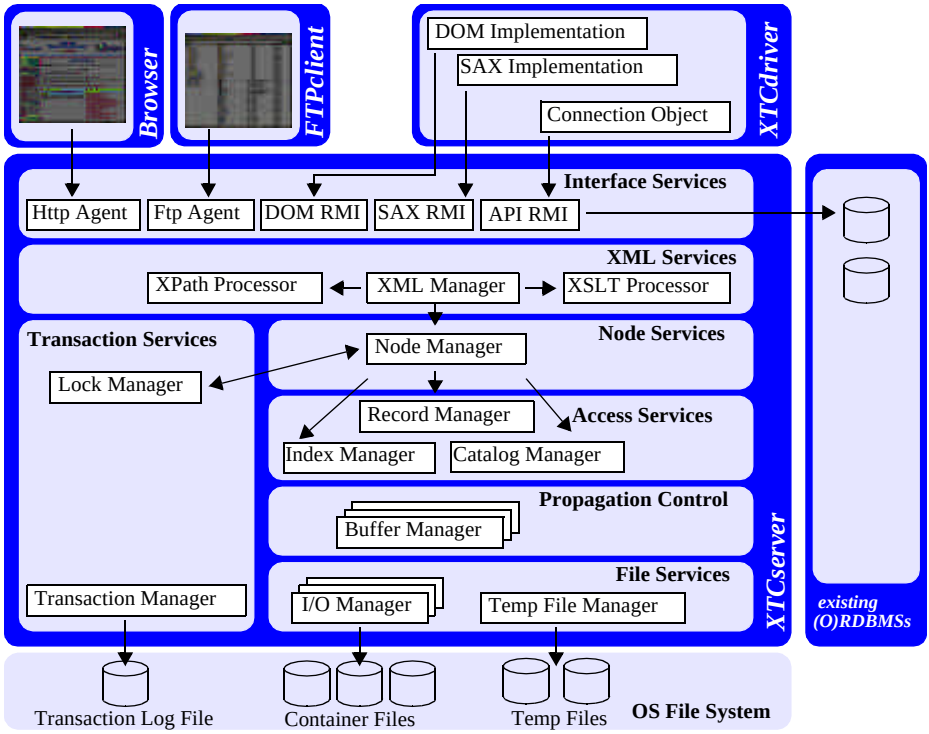


Figure 1 XTC architecture overview

3 Storage Model

Efficient and effective synchronization of concurrent XDP 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 environment. As a running example, we, therefore, refer to an XML document which is slightly enhanced for our purpose to a so-called *taDOM tree* [10], as shown in Figure 2.

AttributeRoot separates the various attribute nodes from their element node. Instead of locking all attribute nodes separately when the DOM method *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 exclusively 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 only navigating across such nodes will not be blocked, although a concurrent transaction may have modified them and may still hold exclusive locks on them.

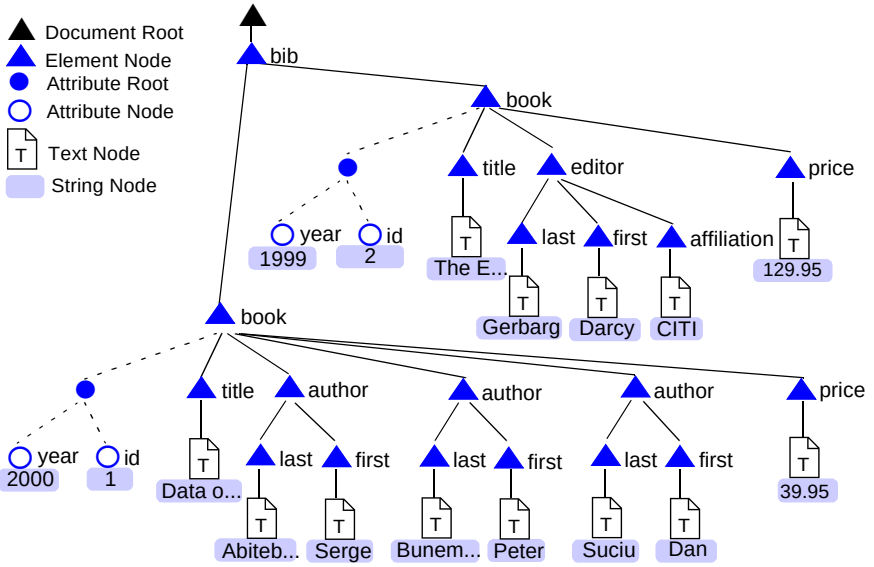


Figure 2 A sample taDOM tree

It is essential for the locking performance to provide a suitable storage structure for taDOM trees which supports a flexible storage layout that allows a distinguishable (separate) node representation of all node types to achieve fine-grained locking. Therefore, we have implemented various container types which enable effective storage of very large and very small attribute and element nodes as well as combinations thereof [9]. Furthermore, fast access to and identification of all nodes of an XML document is mandatory to enable efficient processing of direct-access methods, navigational methods, and lock management. For this reason, our record manager assigns to each node a unique node ID (rapidly accessible via a B-tree) and stores the node as a record in a data page. The tree order of the XML nodes is preserved by the physical order of the records within logically consecutive pages (chained by next/previous page pointers) together with a so-called level indicator per record.

4 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. Hence, a lock manager 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.

Because 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 *automatically*, that is, by the lock manager, acquired in either case for the path of ancestor nodes. The currently accessed node is called *context node* in the following.

This up-to-the-root locking procedure is performed as follows: If such an ancestor path is traversed the first time and if the IDs of the ancestors are not present in the so-called parent index (on-demand indexing of structural relationships [9]) for this path, the record manager is invoked to access stored records thereby searching all ancestor records. The IDs of these records are saved in the parent index. Hence, future traversals of this ancestor path can be processed via the parent index only. Navigational locking of children or siblings is optimized by such structural indexes in a similar way.

The lock modes depend on the type of access to be performed, for which we have tailored the *node lock* compatibilities and defined the rules for lock conversion as outlined in Section 4.1 and Section 4.2. To achieve optimal parallelism, we discuss means to tune lock granularities and lock escalation in Section 4.3. 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. To support this demand, we present so-called *navigation locks* in Section 4.4. Furthermore, query access methods also need strict synchronization to accomplish the well-known *repeatable read* property and, in addition, the prevention of *phantoms* in rare cases. Our specific solution is outlined in Section 4.5.

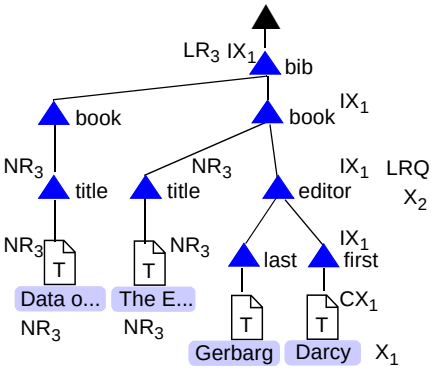
4.1 Node Locks

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. Figure 3a contains the compatibility matrix for our lock modes whose effects are described now:

- 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).

	-	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

- 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 back to a read lock if the inspection of *c* shows that no update action is needed or 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 an IX lock for all other ancestors up to the document root.

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.

Figure 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 com-

plements this action 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 Node Lock Conversion

The compatibility matrix shown in Figure 3a describes the compatibility of locks acquired on the same node by separate transactions. If a transaction T already holds a lock and requests a lock in a more restrictive or incomparable mode on the same node, we would have to keep two locks for T on this node. In general, *k* locks per transaction and node are conceivable. This proceeding would require longer lists of granted locks per node and a more complex run-time inspection algorithm checking for lock compatibility. Therefore, 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 4, 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*. An example for this procedure is given in the now following paragraph.

Assume, a user starts a transaction 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, the transaction 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} specified in Figure 4, the transaction has to convert the existing 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).

	-	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 4 Lock conversion matrix

4.3 Tunable Node Lock Granularity and Lock Escalation

Entire subtrees in the taDOM tree can be locked by both SR locks enabling shared access or X locks granting exclusive access. In either case, we want to improve flexibility, efficiency, and potential parallelism of our locking protocols by enabling tunable lock granularity and lock escalation. The combined use of them increases operational throughput because, due to lock escalation, the number of lock requests can be reduced enormously and, due to fine-tuned lock granularity, higher concurrency may be gained.

To tune the lock granularity of nodes for each transaction separately, the parameter *lock depth* ($ld \geq 0$) is introduced. Parameter *ld* describes the lock granularity by means of the number of node levels (from document root) on which locks are to be held. If a lock is requested for context node *c* whose path length to the document root element is greater than *ld*, only an SR lock for the ancestor node belonging to the lock-depth level is requested. In this way, nodes at deeper levels than indicated by *ld* are locked in shared mode using an SR lock on the node at level *ld*, that is, entire subtrees are locked starting at the specified lock-depth level of the requesting transaction. As a corollary, *ld* = 0 provides document locks, e.g., locks on the <*bib*> node in Figure 5. This allows the traversal of a large document fragment in read mode without acquiring any additional node locks. In the same way, several X locks can be replaced with a single X lock at a chosen document level $l \leq ld$.

Figure 5 shows the taDOM-tree cutout of Figure 3b illustrating the effect of the lock-depth parameter. With *ld* = 2, the NR locks of transaction T_3 on the <*title*> and <*editor*> nodes are replaced with SR locks for the <*title*> nodes. The IX, CX, and X locks of T_1 on the <*editor*> node and its descendants are replaced by a single X lock on the <*editor*> node. As a prerequisite, it requires CX and IX locks on the ancestor nodes <*book*> and <*bib*>, *respectively*. Transaction T_2 is again in a wait state, because the requested X lock is not compatible to the existing X lock of T_1.

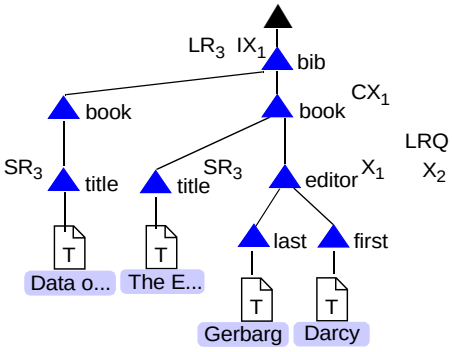


Figure 5 Coarse-grained node locks with lock depth 2

In a similar way, lock escalation can be achieved. To tune lock escalation, we introduce two parameters, the *escalation threshold (et)* and the *escalation depth (ed)*. The lock manager scans the taDOM tree at prespecified intervals. If the manager detects a subtree in which the number of locked nodes of a transaction exceeds the percentage threshold value defined by *et*, the locks held are replaced by an adequate lock at the subtree root, if possible (i. e., no conflicting locks are encountered). Read and write locks are replaced by SR and X locks. The parameter *ed* defines the maximal subtree depth starting from the leaves of a taDOM tree up to the scanned subtree root. Obviously, there is certainly a trade-off to be observed for lock escalation which decreases concurrency of read and write transactions, but, in turn, a reduction of the number of held locks and of lock acquisitions is achieved saving lock management overhead. Its empirical evaluation remains a future task.

4.4 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 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 6b) 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 *getNetxtSibling()* 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 the ER, EU, and EX lock modes corresponding to R/U/X. Their use observing the compatibilities shown in Figure 6a can be summarized as follows:

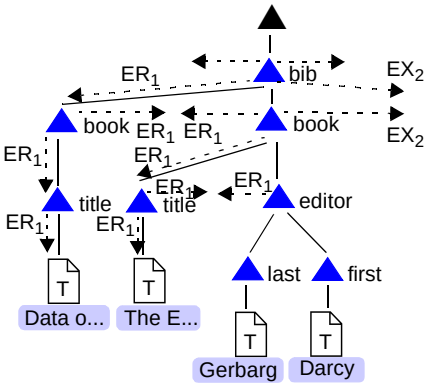
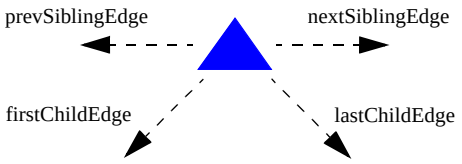


Figure 7 Use of navigation locks

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

a) Compatibility matrix



b) Virtual navigation edges on an element-node

Figure 6 Locking navigational operations in a taDOM tree

- 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 7 illustrates navigation locks on virtual navigation edges. To keep Figure 7 comprehensible, we do not show the 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.

4.5 Prevention of Phantoms

As outlined so far, our protocols enable fine-grained solutions for *repeatable read* and even *serializable* when record-oriented operations are used, i. e., direct as well as navigational access to sequences of document nodes. Note, "gaps" between nodes can be protected by edge locks which prohibit a newly inserted document node to appear as a phantom.

But how do we solve the phantom problem in XML documents for set-oriented access? If we are willing to lock larger granules and thereby potentially sacrifice some parallelism, we can use the same trick known from multi-granularity locking: we just acquire an exclusive lock one level above the working node, that is, on its direct ancestor, and prevent the transaction from being confused by phantom inserts. Obviously, this straightforward approach also increases blocking and deadlock probability. For example, if the *getElementsByTagName()* method of the DOM API is invoked on an arbitrary node *n*, all its sibling nodes and their subtrees are locked, because the parent node of *n* holds the phantom-preventing lock. Hence, this approach may turn out to be too coarse.

Because we may not guarantee "serializability" in the strict sense when fine-grained lock protocols are used for set-oriented access, we currently support the so-called consistency level 2.99 [7] in such situations. Our mechanism described in [10] is based on the concept of precision locks [14]. Because our empirical experiments outlined in Section 5 do not critically rely on effective phantom protection, we will not refine it here. While it is path oriented and can, therefore, be also exploited for (simple) declarative interfaces, phantom prevention for the full expressiveness of XQuery is subject of our future research.

5 Performance Evaluation

In our first experiment, 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 [15] 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 requesting every node by a separate RMI call. To reveal lock management overhead, each XML document is reconstructed by a consecutive traversal in depth-first order under isolation levels *committed* and *repeatable read*. Isolation level *committed* certainly provides higher degrees of concurrency with (potentially) lesser degrees of consistency of shared documents; when used, the programmer accepts a responsibility to achieve full consistency. Depending on the position of the node to be locked, it may cause much more overhead, because each individual node access requires short read locks along its ancestor path. In contrast, isolation level *repeatable read* sets long locks until transaction commit and, hence, does not need to repetitively lock ancestor nodes. In fact, they are already locked due to the depth-first traversal.

These expectations are confirmed by the results of this first experiment as depicted in Figure 8. The potential performance gain of the reduced isolation level *committed* is contrasted by the dramatically increasing lock management overhead due to repeated locking and releasing of locks along the entire ancestor path. Hence, substantial lock processing time is consumed in *committed* mode (>300% of the reconstruction time under isolation level *none*, i. e., without locking overhead), whereas the overhead for *repeatable read* is acceptable (~25%). To guarantee highly consistent documents, *repeatable read* should be used for concurrent transactions. However, its penalty of longer lock durations has to be compensated by effective and fine-granular lock modes which coincides with the objectives of our proposal.

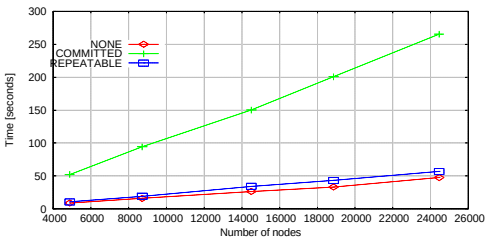


Figure 8 Document reconstruction time

The second experiment illustrates the benefits for transaction throughput depending on the chosen isolation level and lock-depth value. For this purpose, we extend the sample document of Figure 2 to a library database by grouping the books into specific topics and adding a persons' directory. The DataGuide describing the resulting XML document is depicted in Figure 9. We created the library document with 500 persons and 25,000 books grouped into 50 specific topics. The resulting document (requiring approximately 6,4 MB) consists of 483,317 XML nodes and is stored in our XDBMS [9].

We apply different transaction types simulating typical read/write access to XML documents. Transaction T_B is searching for a book with a randomly selected title. This simulates a query of a library visitor. The activities of the library employees are represented by transactions T_P, T_L, and T_R. Transaction T_P is searching for a randomly chosen person by his/her last name. Transactions T_L and T_R are simulating the lending of books. Transaction T_L randomly locates a person and a book to be lent; then it adds a new child

node containing the person's id to the <*history*> element within the located <*book*> subtree. Transaction T_R "returns" the book by setting the *return* attribute of the corresponding <*lend*> element to the current system date.

Ten clients with read transactions of type T_B and one client with a read transaction of type T_P are continuously executing for ten minutes on the library document to provide a base load on the XDBMS. Two clients are executing write transactions of type T_L and T_R making a total of 13 concurrent transactions in the system. A deadlock detector is scanning the wait-for graph of the transactions every five seconds. The XDBMS is running on an *IBM eServer xSeries 235* with two *Intel Xeon-A* 2.4GHz processors. The server machine executes *Microsoft Windows Server 2003 Enterprise Edition*, whereas the clients are running on an *IBM R32 ThinkPad*, connected with a 100Mbps network to the server.

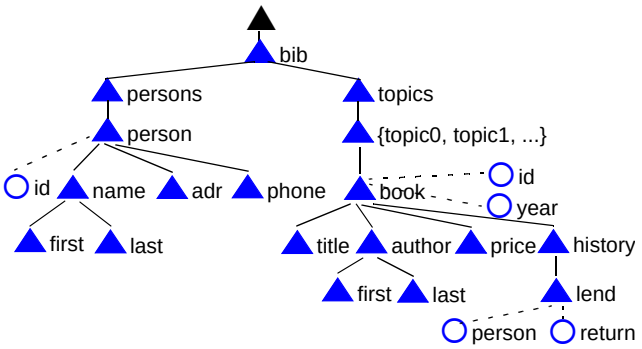


Figure 9 DataGuide of the library document

To explore transaction throughput in two different processing modes, we run this experiment in batch mode (no human interaction while a transaction is running) and with human interaction. The latter case is simulated by a delay of 5 seconds by which the duration of *long locks* is extended in each transaction, before they are released at transaction commit. At least in relational environments, everybody would expect a decrease of transaction throughput with increasing isolation levels: *none*, *uncommitted*, *committed*, *repeatable read*, *serializable*, where in our experiments both isolation levels *repeatable read* and *serializable* produce identical results. On the other dimension, with increasing lock depth—if facilitated by the element position processed in the tree—growing transaction throughput is anticipated because of shrinking lock granules.

Without surprise, maximum transaction throughput is reached for isolation level *uncommitted* in all experiments, because read locks are abandoned. Write locks, in turn, seriously interfere with concurrent transactions only at lock depth 0 and 1 (see Figure 10a and 11a), whereas they hardly affect them at lock depths 2 to 7.

5.1 Batched Transaction Processing

As the most striking observation, conducting our experiment in batch-processing mode revealed in all cases a higher throughput at isolation level *repeatable read* than at *committed*, because the long read locks avoid the subsequent traversals of ancestor paths for lock acquisitions in most cases. The curves of committed write transactions (depending on the lock depth) are similar at all isolation levels (see Figure 10a). Most of the conflicts (waiting cycles or deadlocks) are occurring at lock depth 0 resp. 1, because transactions T_B, T_L, and T_R are locking the <*bib*> resp. the <*topics*> nodes. Hence, all isolation levels nearly yield the same throughput of write transactions.

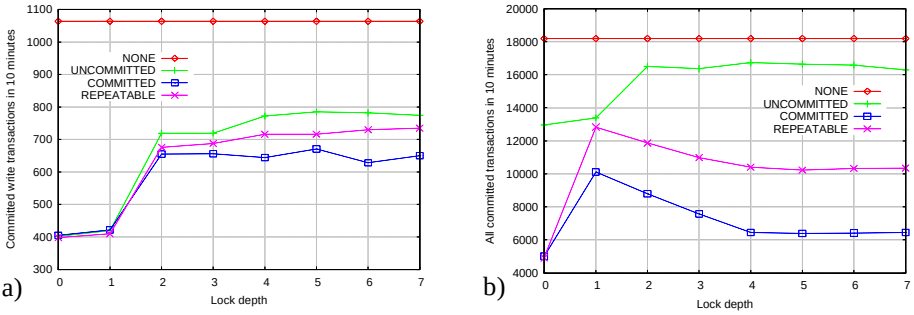


Figure 10 Successful batch-processed transactions

Enhancing the lock depth value from 1 to 2, much more write transactions commit, because most of them are executed concurrently (only those accessing the same topic have to be serialized). The number of successful write transactions is slightly increasing from lock depth 2 to 7, because the transactions are keeping long locks which avoid a repeated traversal of complete ancestor paths in most cases when additional locks are requested. But surprisingly, a higher degree of isolation also enables higher throughput of write transactions, which can be explained by the following observation: *Repeatable read* yields shorter transaction processing times than *committed* because the read operations of the write transactions T_L and T_B do not acquire and immediately release (a set of) short read locks for each node access.

The number of committed transactions (Figure 10b) is primarily depending on the commits of the readers T_B and T_P, because, compared to the writers T_L and T_R, they contain less operations and are executed by more client threads in parallel. Because of the short read locks, the throughput for *committed* behaves even worse than for *repeatable read*. The peak at lock depth 1 is caused by transaction T_P which is executed without interference while T_B, T_L, and T_R are frequently blocking each other. This peak number of commits (mainly due to T_P) decreases from lock depth 1 to 4, because more and more transactions of type T_B, T_L, and T_R successfully finish thereby increasing lock and transaction management overhead. At lock depth 4, locking conflicts of transactions T_L and T_R do not affect T_B anymore. Hence, from lock depths 4 to 7, the XDBMS seems to be in a kind of steady state and achieves stable transaction throughput.

5.2 "Interactive" Transaction Processing

Transactions interrupted by human interactions ("the human is in the loop") or performing complex operations may exhibit drastically increased lock duration times. While the average transaction response time and lock duration was far less than a second in batch-processing mode, now the average lock duration was "artificially" increased probably by more than a factor of 10. As a consequence, the finer granularity of locks and the duration of *short read locks* gained in importance on transaction throughput while the relative effect of lock management overhead was essentially scaled down. Longer lock durations and, in turn, blocking times reduced the number of successful commits (write and overall transactions) to about 50% and 10% as shown in Figure 11a and b and caused a relative performance behavior as anticipated in relational environments.

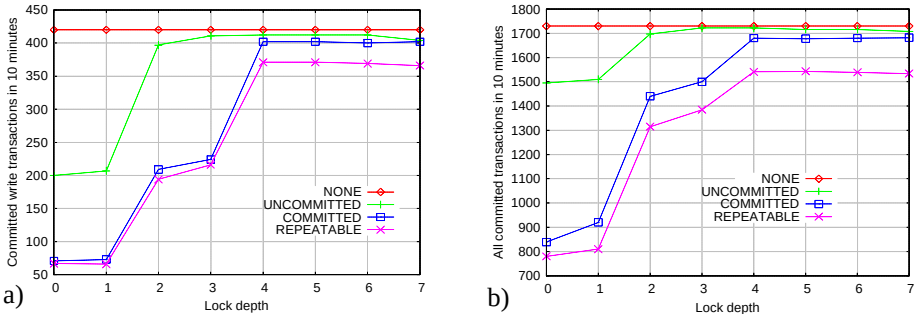


Figure 11 Successful transactions with human interaction

In general, transaction throughput can be increased by decreasing the level of isolation (from *repeatable read* down to *uncommitted*) or increasing the lock depth (if possible). As observed at lock depths 4 to 7 in Section 5.1, all transactions can be executed in parallel and our XDBMS approaches stable transaction throughput in this experiment.

For future benchmarks, we expect the gap between *uncommitted* and *committed* to grow larger for "deeper" XML documents (longer paths from the root to the leaves). Similarly, the gap between *committed* and *repeatable read* widens with an increasing percentage of write transactions (causing more waiting cycles).

6 Related Work, Conclusions and Future Work

So far, only a few papers deal with fine-grained CC in XML documents. DGLOCK [6] explores a *path-oriented protocol* for semantic locking on DataGuides. It is running in a layer on top of a commercial DBMS and can, therefore, not reach the fine granularity and flexibility of our approach. In particular, it cannot support ID-based access and position-based predicates. Another path-oriented protocol is proposed in [3, 4] which also seems to be limited as far as the full expressiveness of XPath predicates and direct jumps into subtrees are concerned. To our knowledge, the only competing approach which is also navigation oriented comes from the locking protocols designed for Natix [12]. They are also tailored to typical APIs for XDP. While the proposed lock modes are different to ours, the entire protocol behavior should be compared. Currently, we have the advantage that we do not need to simulate our protocols, but we can measure their performance on existing benchmarks and get real numbers.

In this paper, we have primarily explored transaction isolation issues for collaborative XML document processing. We first sketched the design and implementation of our native XML database management system. For concurrent transaction processing, we have introduced our concepts enabling fine-granular concurrency control on taDOM trees representing our natively stored XML documents. As the key part, we have described the locking protocols for direct and navigational access to individual nodes of a taDOM tree, thereby supporting different isolation levels. The performance evaluation has revealed the locking overhead of our complex protocols, but, on the other hand, has confirmed the viability, effectiveness, and benefits of our approach. As a striking observation, lower isolation levels on XML documents do not necessarily guarantee better

transaction throughput, because the potentially higher transaction parallelism may be (over-)compensated by higher lock management overhead. There are many other issues that wait to be resolved: For example, we did not say much about the usefulness of optimization features offered. Effective phantom control needs to be implemented and evaluated (thereby providing for isolation level *serializable*), based on the ideas we described. Then, 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.).

Acknowledgements. The anonymous referees who pestered us with many questions helped to improve the final version of this paper.

References

[1] American National Standard for Information Technology. Database Languages - SQL - Part 2: Foundation (1999)

[2] D. Brownell. SAX2. O'Reilly (2002)

[3] S. Dekeyser, J. Hidders. Path Locks for XML Document Collaboration. Proc. 3rd Conf. on Web Information Systems Engineering (WISE), Singapore, 105-114 (2002)

[4] S. Dekeyser, J. Hidders, J. Paredaens. A Transaction Model for XML Databases. World Wide Web Journal 7(2): 29-57 (2004)

[5] T. Fiebig, S. Helmer, C.-C. Kanne, G. Moerkotte, J. Neumann, R. Schiele, T. Westmann. Natix: A Technology Overview. A.B. Chaudri et al. (Eds.): Web, Web Services, and Database Systems, NODe 2002, Erfurt, Germany, LNCS 2593, Springer, 12-33 (2003)

[6] T. Grabs, K. Böhm, H.-J. Schek: XMLTM: Efficient Transaction Management for XML Documents. Proc. ACM CIKM Conf., McLean, VA, 142-152 (2002)

[7] J. Gray, A. Reuter. Transaction Processing: Concepts and Techniques. Morgan Kaufmann (1993)

[8] A. Halverson, V. Josifovski, G. Lohman, H. Pirahesh, M. Mörschel. ROX: Relational Over XML. Proc. 30th VLDB Conf., Toronto (2004)

[9] M. Haustein, T. Härder. Fine-Grained Management of Natively Stored XML Documents, submitted (2004)

[10] M. Haustein, T. Härder. taDOM: A Tailored Synchronization Concept with Tunable Lock Granularity for the DOM API. Proc. 7th ADBIS Conf., Dresden, Germany, 88-102 (2003)

[11] T. Härder, A. Reuter. Concepts for Implementing a Centralized Database Management System. Proc. Computing Symposium on Application Systems Development, Nürnberg, Germany, 28-60 (1983)

[12] S. Helmer, C.-C. Kanne, G. Moerkotte. Evaluating Lock-Based Protocols for Cooperation on XML Documents. SIGMOD Record 33(1): 58-63 (2004)

[13] H. V. Jagadish, S. Al-Khalifa, A. Chapman. TIMBER: A Native XML Database. The VLDB Journal 11(4): 274-291 (2002)

[14] J. R. Jordan, J. Banerjee, R. B. Batman: Precision Locks. Proc. ACM SIGMOD Conf., Ann Arbor, Michigan, 143-147 (1981)

[15] A. Schmidt, F. Waas, M. Kersten. XMark: A Benchmark for XML Data Management. Proc. 28th VLDB Conf., Hong Kong, China, 974-985 (2002)

[16] W3C Recommendations. http://www.w3c.org (2004)