Optimizing Concurrent XML Processing

Michael P. Haustein, Theo Härder
University of Kaiserslautern
P.O. Box 3049, 67653 Kaiserslautern, Germany
{haustein | haerder}@informatik.uni-kl.de

Abstract

Processing XML documents in multi-user database management environments requires a suitable storage model, support of typical XML document processing (XDP) interfaces, and concurrency control mechanisms tailored to the XML data model. In this paper, we sketch the architecture of our prototype native XML database management system and specify in detail the operations for accessing and modifying the stored documents. The key contribution is the design and optimization of fine-grained locking protocols supporting collaborative processing of XML documents. For this reason, we introduce four XML locking protocols of growing sophistication and complexity, which are based on a tree-structured storage model. Finally, we present the ideas to prove the locking protocol correctness guaranteeing the specified data processing behavior of the given XDP operations.

1. Introduction

Storing XML documents in a relational database management system (RDBMS) forces the developers either to use simple CLOBs (*character large objects*), to select some data types which enable the mapping of the documents to predefined relational schemes, or to choose among an innumerable number of algorithms shredding the documents to tables and columns. Because of their number and size, collaboration on XML documents often becomes an important issue. Typical applications are managing XML-structured operational business data or applying XML standards for collaboration in word processing applications [11] using database backends. However, concurrency control in RDBMSs does not take the properties of the semi-structured XML data model into account and causes disastrous locking behavior by blocking entire CLOBs, tables, or unnecessarily large index ranges.

Currently, native XML DBMSs (XDBMSs) are primarily designed for efficient document retrieval and query evaluation. Their document storage model is usually based on fixed numbering schemes used to identify XML elements and optimized for read-only access. Frequently concurrent and transaction-safe modifications would lead to renumerations of large document parts which could cause unacceptable reorganization overhead and degrade data processing in performance-critical workload situations. As a rare example of an update-oriented system, Natix [5] conceptually supports concurrent transaction processing, but multi-user mode is not implemented yet.

1.1 Problem Statement and Contribution

Although predicate locking of XQuery statements [14]—and, in the near future, XUpdate-like statements [15]—would be powerful and elegant, its implementation rapidly leads to severe drawbacks such as undecidability problems and the need to acquire large lock granules for simplified predicates—a lesson learned from the (much simpler) relational world. To provide for an acceptable solution, we necessarily have to map XQuery operations to a navigational access model to accomplish fine-granular concurrency control. Such an approach implicitly supports other XDP interfaces like DOM [13], and SAX [1], because their operations correspond more or less directly to a navigational access model.

We have proposed a fine-granular locking protocol called taDOM2 in [8] which enables concurrent execution of transactions using either DOM, SAX, XQuery or all of them simultaneously. We will refine and optimize it as taDOM2+. The recent standard *DOM Level 3* [13] additionally introduced new operations, for which we develop the taDOM3 protocol and its optimized version taDOM3+.

As a testbed for XML transaction processing, we have implemented the *XML Transaction Coordinator (XTC)* [8] which supports all known types of XDP interfaces (event-based like SAX, navigational like DOM, and declarative like XQuery) and provides the well-known ACID properties [7] for concurrent operations. For all four protocols, we can thus give an empirical performance comparison which clearly indicates the performance potential of our optimizations as far as enhanced parallelism and reduced locking overhead is concerned. Furthermore, we prove the correctness of the proposed lock mode compatibilities and lock conversions used by our protocols.

1.2 Related Work

As already mentioned, no (freely available) XDBMS implementation exists which would allow exploring fine-grained XML concurrency control. Hence, the few publications related to our problem either refer to conceptual work or to simulations [10]. XMLTM [6] uses a layer on top of an RDBMS which executes the client-side transaction operations within self-managed transactions which, in turn, have to be processed (under lower isolation levels) on the 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 (direct jumps to internal nodes) and position-based predicates and is not tailored to fine-grained navigational access. Another path-oriented protocol is proposed in [3] and [4] which also seems to be limited as far as the full expressiveness of XPath predicates and direct jumps into subtrees are concerned.

In this paper, Section 2 gives an overview of the XTC architecture, the storage model for XML documents, and the XDP operations. Our four locking protocols taDOM2, taDOM2+, taDOM3, and taDOM3+ are introduced and compared in Section 3, whereas Section 4 describes our ideas to prove the correctness of the locking protocols for the present XDP operations. Finally, Section 5 wraps up some aspects of future work.

2. XML Data Processing

Our XTC database engine (*XTCserver*) adheres to the widely used five-layer database architecture which is sketched in the following Section 2.1. Processing XML documents is based on the taDOM data model which is described in Section 2.2. The available node-based XDP operations for accessing and manipulating the stored documents are described in Section 2.3.

2.1 System Architecture

The five-layer architecture of our XTC system is depicted in Figure 1. 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 fixed-length block size is configurable to the characteristics of the XML documents to be stored. A *buffer manager* for each container file handles fixing and unfixing of pages in main memory and provides a page replacement algorithm for them which can be optimized to the anticipated reference locality inherent in the respective XDP application. Using pages as their 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. This is essential to allow for fine-grained concurrency control which requires lock acquisition on unique identifiable nodes (see Section 3). The catalog manager provides for the database meta-data. 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 node-based XDP operations for document accesses and modifications (considered in detail in Section 2.3) are provided at this layer's interface. In contrast, the *XML services* layer contains the XML manager responsible for declarative document access, e.g., evaluation of XQuery statements or XSL transformations.

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 establishing declarative/set-oriented as well as navigational/node-oriented interfaces. The XTCdriver linked to client-side applications provides for methods to execute XQuery statements and to browse or manipulate XML documents and materialized XQuery results via the SAX or DOM API. All client-side activities are processed within transactions running in one of the well-known isolation levels *uncommitted*, *committed*, *repeatable*, or *serializable*.

2.2 taDOM Storage Model

Efficient and effective isolation of concurrent XDP operations is greatly facilitated, if we use a specialized internal document representation (the so-called *taDOM tree*) which enables fine-granular locking. For this reason, we have introduced two new node types: *attribute root* and *string*. This representation extension does not influ-

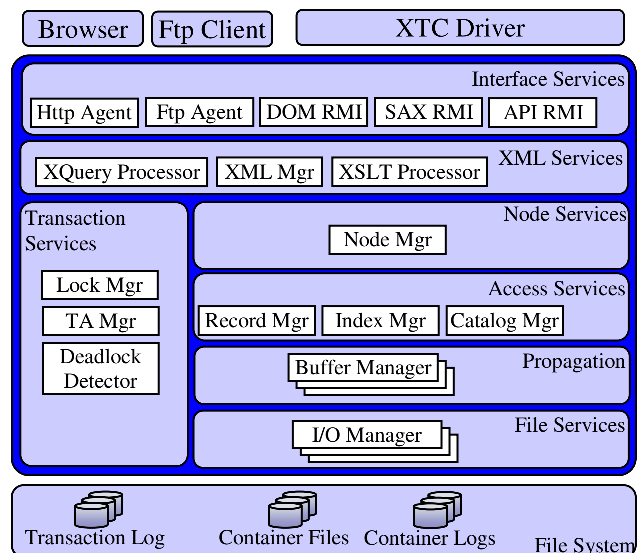


Figure 1: XTC system architecture

```
<?xml version="1.0"?>
<bib>
  <book year="2004" id="book1">
    <title>The Title</title>
    <author>
      <fname>first name</fname>
      <lname>last name</lname>
    </author>
    <price>49.99</price>
  </book>
</bib>
```

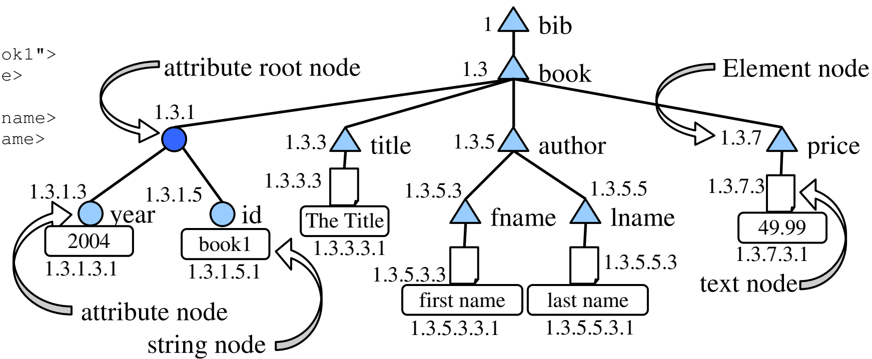


Figure 2: Transformation of *sample.xml* into the taDOM tree

ence the user operations and their semantics on the XML document, but is solely exploited by the lock manager to achieve certain kinds of optimization when an XML document is modified in a cooperative environment. As a running example, we refer the XML document *sample.xml* which is transformed for our purpose to its taDOM-tree representation as shown in Figure 2.

The attribute root separates the various attribute nodes from their element node. Instead of locking all attribute nodes of an element separately when they are listed, the lock manager achieves the same effect by a single lock on the attribute root. 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 actual value of this node. Because reference to such a value requires an explicit operation invocation with a preceding lock request, a simple existence test on a text or attribute node avoids locking their values. 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 their values[1].

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 native B*-tree implementation) and stores the node as a record in a data page. In this way, we can enforce and preserve the order of the XML nodes by the physical order of the records within logically consecutive pages (chained B*-tree leaves by next/previous page pointers).

The unique node IDs are inspired by the ORDPATH approach [12] and are adapted as so-called *DeweyIDs* to our taDOM storage model (also shown in Figure 2). The DeweyID algorithm is based on the *Dewey Decimal Classification* and assigns odd *division numbers* to the nodes within each document level consecutively in ascending order (e.g., *1.3.3*, *1.3.5*, *1.3.7,* etc.). Except for the document root element, the division value *1* is reserved for attribute root and string nodes. The different node levels are separated by dots; the DeweyID of the parent node is copied as a prefix into the DeweyID of each child node. The initial assignment of odd division values allows the insertion of an arbitrary number of new nodes at arbitrary positions into the document. For example, between the DeweyIDs *1.3.3* and *1.3.5* the new IDs *1.3.4.3*, *1.3.4.5*, *1.3.4.7,* etc. can be inserted. Again, between the IDs *1.3.4.3* and *1.3.4.5,* we are able to insert the IDs *1.3.4.4.3*, *1.3.4.4.5*, and so forth. In this way, the DeweyID allows—by considering the even and odd division values of a node ID—the calculation of its level and the IDs of all ancestor nodes upwards to the document root element without accessing the actual document, which may reside on an external device at reference time. This is a very important aspect for high-performance lock management.

After an XML document is stored, a catalog page containing some meta-data identifies the document. Its page number is attached to the DeweyID separated by a colon, to address a node uniquely within the entire XDBMS. For example, if the document given in Figure 2 is stored with catalog page number *4711,* the element *title* is identified with the ID *4711:1.3.3*.

A further advantage using the DeweyID can be exploited for the manipulation of an XML document. An inserted node at an arbitrary position is always arranged in sequential order with respect to already existing sibling nodes. In this way, a single B*-tree is sufficient for storing the entire XML document in left-most depth-first order, where an entry is formed by the byte representation of the DeweyID as the key part and the byte representation of the actual node as the value part. For element and attribute nodes, the bytes to be stored are additionally compressed using a vocabulary. This means, we do not store their names but tiny identifiers to address the names within a related tree data structure.

[1] These additional node types can be virtualized such that regular DOM trees are stored on disk but the main memory structures of an XDBS maintain our taDOM trees.

2.3 XDP Operations

In the node services layer, our node manager provides for 19 *node operations* to browse and manipulate the stored XML documents in any contrivable manner.

The *getNode()*, *getParentNode()*, *getPrevSibling()*, *getNextSibling()*, *getFirstChild()*, and *getLastChild()* operations are used to address a single context node and perform simple navigation steps to its parent, one of its siblings, or its first or last child node. The *getChildNodes()* resp. *getFragmentNodes()* operations return all direct-child nodes of a given context node resp. the context node itself and all descendant nodes for a complete fragment reconstruction addressed by the context node.

The *getValue()* operation identifies the actual value of a context node. In case of an element node, this is the element name; for attribute or text nodes, the associated attribute value or text content is returned. The other way around, the *setValue()* operation renames an element node or sets a new attribute or text value.

Executed on an element node, the *getAttribute()* operation with an attribute name as a parameter returns the corresponding attribute node (or a null value if an attribute with the name handed over does not exist) and the *getAttributes()* operations assembles a node list of all existing attributes of the element node. The *setAttribute()* operation sets a value for the attribute with the specified name or creates a new attribute with the name/value pair assigned, if such an attribute with the given name does not already exist. The *renameAttribute()* operation renames an already existing attribute node without changing its value.

Creating new element nodes is performed with the operations *appendChild()*, *prependChild()*, *insertBefore()*, and *insertAfter()* which insert a new last or first child, or a new previous or next sibling of the context element node on which they are invoked.

Finally, the *deleteNode()* operation deletes a complete XML fragment identified by the root node on which the operation is executed.

3. taDOM Locking Protocols

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 (see Section 2.2), the principles of multi-granularity locking schemes can be applied.

The method calls of different XDP interfaces used by an application are mapped by the node manager to adequate node operations (see Section 2.3). Before the actual operation is performed, appropriate locks for the affected nodes and the entire ancestor paths are automatically set by the lock manager. For this purpose, the lock manager—possibly with the help of other components such as index or record manager—has to identify the affected nodes and edges. In any case, the truly complex locking protocols are confined to the lock manager and not visible to any other component, let alone the application.

The resulting 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—in contrast to the relational world—the nodes in an ancestor path are part of the document and carry user data. In a relational database, user data is exclusively stored in the leaves (records) of the tree whose higher-level nodes are formed by organizational concepts (e.g., table, segment, database). For example, it makes perfect sense to lock an intermediate XML node *n* for an update operation, while other transactions may perform further reads or updates in the subtree of *n*.

To support concurrent transaction processing exploiting fine-grained concurrency control, we present and compare the four locking protocols *taDOM2*, *taDOM2+*, *taDOM3*, and *taDOM3+* in the following sections.

3.1 taDOM2

Except for some optimizations in the compatibility and conversion matrices, the taDOM2 protocol is based on the protocol we presented in [8]. We differentiate the read and write operations thereby replacing the well-known (IR, R) and (IX, X) lock modes with (IR, NR, LR, SR) and (IX, CX, SX) modes, respectively. As in the multi-granularity scheme, the U mode (SU in our protocol) plays a special role, because it permits lock conversion. Figure 3a contains the *compatibility matrix* for our basic lock modes. Throughout the paper, the matrix header row characterizes the current lock state of the object, whereas the matrix header column indicates the

	-	IR	NR	LR	SR	IX	CX	SU	SX
IR	+	+	+	+	+	+	+	-	-
NR	+	+	+	+	+	+	+	-	-
LR	+	+	+	+	+	+	-	-	-
SR	+	+	+	+	+	-	-	-	-
IX	+	+	+	+	-	+	+	-	-
CX	+	+	+	-	-	+	+	-	-
SU	+	+	+	+	+	-	-	-	-
SX	+	-	-	-	-	-	-	-	-

a) lock compatibility matrix

	-	IR	NR	LR	SR	IX	CX	SU	SX
IR	IR	IR	NR	LR	SR	IX	CX	SU	SX
NR	NR	NR	NR	LR	SR	IX	CX	SU	SX
LR	LR	LR	LR	LR	SR	IX_{NR}	CX_{NR}	SU	SX
SR	SR	SR	SR	SR	SR	IX_{SR}	CX_{SR}	SR	SX
IX	IX	IX	IX	IX_{NR}	IX_{SR}	IX	CX	SX	SX
CX	CX	CX	CX	CX_{NR}	CX_{SR}	CX	CX	SX	SX
SU	SU	SU	SU	SU	SU	SX	SX	SU	SX
SX	SX	SX	SX	SX	SX	SX	SX	SX	SX

b) lock conversion matrix

Figure 3: taDOM2 locking protocol

mode of the incoming lock request. Here, we repeat the effects of the lock modes to facilitate comprehension:

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

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 4 illustrates the result of the following example. Transaction T_1 starts modifying the value *last name* and, therefore, acquires an SX lock for the DeweyID of the corresponding string node. The lock manager complements this action by acquiring a CX lock for the parent DeweyID and IX locks for all further ancestor IDs. Transaction T_2 is generating a list of all child nodes of the *book* element and has, therefore, requested an IR lock on the *bib* element and an LR lock on the *book* node to obtain read access to all direct-child nodes thereby using level-read optimization. Further on, the price of the *book* node is accessed and the path downwards to the corresponding string node is locked by NR locks. Simultaneously, transaction T_3 wants to delete the entire *author* node for which T_3 must acquire an IX lock on the *bib* node, a CX lock on the *book* node, and an SX lock on the *author* node. The lock request on the *book* node cannot immediately be granted because of the existing LR lock of T_2. Hence, T_3—placing its request in the lock request queue (LRQ: CX_3)—must synchronously wait for the release of the LR lock of T_2 on the *book* node.

Note, the IR and NR modes exhibit the same behavior in the taDOM2 and taDOM2+ locking protocols. In a real implementation (like our XTCserver) they can be replaced with one proxy lock mode (e.g., NR). Here, both lock modes IR and NR are kept for completeness; later on, they will differ in the protocols taDOM3 and taDOM3+.

If a transaction T already holds a lock and requests a lock in a different mode on the same node, we would have to keep two locks for T on this node. In general, several locks per transaction and node are conceivable which would require longer lists of granted locks per node and a more complex run-time inspection algorithm checking for lock compatibility. To cope with this problem, we always replace an existing lock of a transaction with a single lock in a mode giving sufficient isolation for both the requested and the existing lock mode. The actions needed by the lock manager are described in [9]. The corresponding rules are specified by the *lock conversion matrix* in Figure 3b which determines the resulting lock for a context node *c* if a transaction already holds a lock (matrix header row) and requests a further lock (matrix header column). 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*.

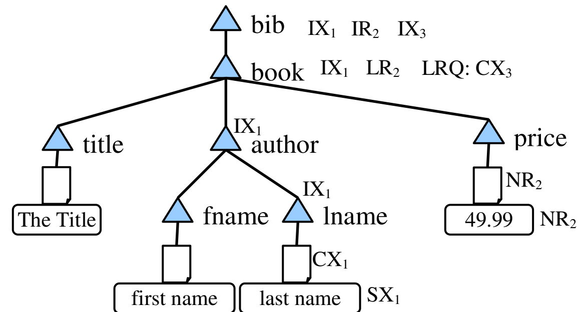


Figure 4: taDOM2 locking example

Additionally, all edges (explained below) on the child nodes' level have to be locked to prevent the insertion of new children. An example for this procedure is given in the following.

Assume, a user starts requesting all child nodes of *c* which results in acquiring an LR lock on *c*. Note again, LR locks *c* and all direct-child nodes in shared mode. Then the user wants to delete one of the previously determined child nodes. Therefore, the transaction acquires an SX lock on the resp. 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 for the conversion, the lock manager converts the existing LR lock on *c* to a CX lock and acquires an NR lock on each direct-child node of *c* (except the node which is already locked for deletion by SX).

In addition to the node lock management described above, we maintain so-called *navigation locks* to isolate navigation paths. This means, a sequence of navigational method calls or modification operations—starting at a known node within the taDOM tree—must always yield the same sequence of result nodes within a transactional context. Hence, a path of nodes evaluated by a transaction must be protected against concurrent modifications. Assume, the *sample.xml* document in Figure 2 contains several *books* and a transaction T navigates through a range of *book* nodes, then T wants to be isolated from concurrent inserts of new *books* in the examined node range.

Of course, we have already introduced some lock modes which protect such a situation, but (too) large lock granules cause (too) expensive isolation. For example, if we acquire an LR lock on the *bib* node, all *book* nodes (and not only the navigated ranges) are implicitly granted in shared mode and the LR lock prevents any insertion with its incompatibility to the required CX lock for an SX on a new *book* node. An SR lock on *bib* would even prohibit updates on the entire document. We, however, want to support a solution only acquiring minimal lock granules, that is, node locks of mode NR only for nodes visited by the navigation. Therefore, we introduce *virtual navigation edges* [8] within the taDOM tree (Figure 5) 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 for the nodes visited. Because each navigation step only performs local operations (first/last, next/previous) to a sibling or child node, the R/U/X locks known from normal record locking 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 and, in addition, the previous-sibling edge of *n* to prohibit concurrent path modifications between *n* and *c* via node *n*. If the *getNextSibling()* operation returns a null value, we also have to lock the last-child edge of the parent node of *c,* because the null value informs the transaction about the last-child position of c. To support such traversals efficiently, we offer ER, EU, and EX lock modes corresponding to R/U/X. Their use can be summarized as follows:

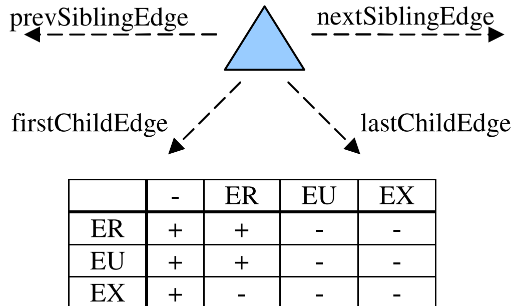
- An ER lock mode (edge read) is needed for an edge traversal in read mode, e.g., by calling the *getNextSibling()* or *getFirstChild()* operation.
- An EX lock mode (edge exclusive) enables an edge to be modified which may be needed when nodes are inserted, appended, or deleted. For all edges redirected by the modification operation, EX locks are required.
- The EU lock mode (edge update option) eases the occurrence of deadlocks for write transactions (see SU).

Note, the navigation edges are only logical objects which are not materialized within the stored document. They are only maintained by the lock manager in main memory. Additionally, as a positive side effect, the acquisition of shared navigation locks on the traversed document paths prevents the occurrence of phantoms by protecting these areas with edge locks against concurrent node insertions.

The additional concept of tunable node lock granularity and lock escalation [8] to reduce the number of maintained locks thus paying with less concurrency is not considered in the focus of this paper.

The locking protocol taDOM2 described so far consisting of the node lock compatibility and conversion matrices and the virtual navigation edge locks is able to isolate all methods specified in *DOM Level 2* [13] in an appropriate way. But considering the new methods introduced by *DOM Level 3* and all our operations provided by the node services layer, a new problematic situation appears. The *renameNode()* method of the DOM specification and the *setValue()* operation of our node manager executed on an inner element node *e* of the taDOM tree (not a leaf element node) requires the exclusive locking of the element node. With a tailored locking protocol, however, it should be possible to directly address and isolate an arbitrary node *n* in the subtree of *e* (e.g., via a secondary index) and perform arbitrary operations on *n*. In other words, the exclusive locking of a single inner node should not affect the subtree of this inner node in any way.

To support this situation in an adequate way, we introduce for the taDOM2 and taDOM2+ locking protocols



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

Figure 5: Virtual navigation edges and locks modes

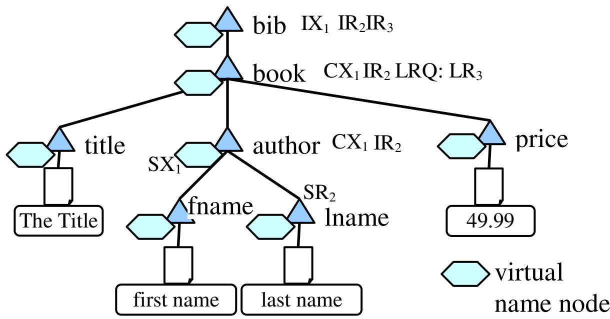


Figure 6: Locks on virtual name nodes

additional *virtual name nodes* for each element and attribute node. Similarly to the virtual navigation edges, the virtual name nodes are not materialized in the stored document and are only maintained by the lock manager in main memory. As a consequence, a lock request in shared mode on context node *c* (NR lock) is always extended by the lock manager to an NR lock request on the actual node and, in addition, on its corresponding virtual name node. The exclusive locking of a single context node *c* (and not its entire subtree) is obtained by an exclusive SX lock on its virtual name node, an implicit CX lock on the actual context node *c*, and an additional CX lock on the parent node of *c*. This idea is clarified in the following example.

An attached virtual name node is addressed with the DeweyID of its owning element extended with a *0*. For example, assuming the assigned DeweyIDs of Figure 2, the *book's* (*1.3*) virtual name node ID is *1.3.0*, the *title's* (*1.3.3*) name node ID is *1.3.3.0*, and so forth. In this way, the determined "parent node" for lock requests along all ancestor nodes up to the document root element is the actual element owning the virtual name node. Figure 6 illustrates the resulting locks after applying the virtual name-node concept. Transaction T_1 is renaming the *author* element and, therefore, locking the virtual name node of the *author* node with SX, the *author* element itself with CX, and—applying the locking protocol—all ancestor nodes with IX. An additional CX lock on the parent of the *author* node (*book*) is required to prevent another transaction from determining all direct-child nodes of the *book* element. Although transaction T_1 is now renaming the *author* element, transaction T_2 is allowed to "jump" into the document (via a secondary index) and to reconstruct the *lname* element with its complete subtree. The IR locks on the ancestor nodes required for the SX lock on *lname* comply with the existing IX and CX locks of T_1. Transaction T_3 which wants to determine all direct-child nodes of the *book* element is blocked (LRQ: LR_3), because LR is incompatible with the existing CX of transaction T_1.

3.2 taDOM2+

Considering again the lock conversion matrix of the taDOM2 locking protocol in Figure 3b, the subscripted node lock conversions IX_{NR}, IX_{SR}, CX_{NR}, and CX_{SR} represent indispensable rules to guarantee sufficient transaction isolation against concurrent modifications. But in the same way, these rules cause an additional dramatic run-time overhead on the XDBMS. It is true that, for a given DeweyID of an arbitrary node, the lock manager can calculate the IDs of all ancestor nodes (without accessing the stored XML document) and set the implicitly requested locks on them. This kind of lock acquisition is performed very rapidly. However the other way around, determining all children of a given node to set NR resp. SR locks on them (needed to conform to the conversion rules) is a very expensive operation. For a context node *c,* its direct-child nodes ch_i cannot be calculated, but have to be determined by fetching *c* and each ch_i from the stored document.

To cope with this problem, we ease the node lock conversion by introducing four new lock modes tailored to the situations triggering one of the conversions described above:

- An LRIX lock mode (level read intention exclusive) locks the context node together with all its direct-child nodes for shared access and, in addition, indicates the intention to perform write operations somewhere in the subtree, but not on a direct-child node.
- An SRIX lock mode (subtree read intention exclusive) locks the context node *c* and its entire subtree to perform read operations and indicates the intention to perform write operations somewhere in that subtree, but

	-	IR	NR	LR	SR	IX	LRIX	SRIX	CX	LRCX	SRCX	SU	SX
IR	+	+	+	+	+	+	+	+	+	+	+	-	-
NR	+	+	+	+	+	+	+	+	+	+	+	-	-
LR	+	+	+	+	+	+	+	+	-	-	-	-	-
SR	+	+	+	+	+	-	-	-	-	-	-	-	-
IX	+	+	+	+	-	+	+	-	+	+	-	-	-
LRIX	+	+	+	+	-	+	+	-	-	-	-	-	-
SRIX	+	+	+	+	-	-	-	-	-	-	-	-	-
CX	+	+	+	-	-	+	-	-	+	-	-	-	-
LRCX	+	+	+	-	-	+	-	-	-	-	-	-	-
SRCX	+	+	+	-	-	-	-	-	-	-	-	-	-
SU	+	+	+	+	+	-	-	-	-	-	-	-	-
SX	+	-	-	-	-	-	-	-	-	-	-	-	-

a) lock compatibility matrix

	-	IR	NR	LR	SR	IX	LRIX	SRIX	CX	LRCX	SRCX	SU	SX
IR	IR	IR	NR	LR	SR	IX	LRIX	SRIX	CX	LRCX	SRCX	SU	SX
NR	NR	NR	NR	LR	SR	IX	LRIX	SRIX	CX	LRCX	SRCX	SU	SX
LR	LR	LR	LR	LR	SR	LRIX	LRIX	SRIX	LRCX	LRCX	SRCX	SU	SX
SR	SR	SR	SR	SR	SR	SRIX	SRIX	SRIX	SRCX	SRCX	SRCX	SR	SX
IX	IX	IX	IX	LRIX	SRIX	IX	LRIX	SRIX	CX	LRCX	SRCX	SX	SX
LRIX	LRIX	LRIX	LRIX	LRIX	SRIX	LRIX	LRIX	SRIX	LRCX	LRCX	SRCX	SX	SX
SRIX	SRIX	SRIX	SRIX	SRIX	SRIX	SRIX	SRIX	SRIX	SRCX	SRCX	SRCX	SX	SX
CX	CX	CX	CX	LRCX	SRCX	CX	LRCX	SRCX	CX	LRCX	SRCX	SX	SX
LRCX	LRCX	LRCX	LRCX	LRCX	SRCX	LRCX	LRCX	SRCX	LRCX	LRCX	SRCX	SX	SX
SRCX	SRCX	SRCX	SRCX	SRCX	SRCX	SRCX	SRCX	SRCX	SRCX	SRCX	SRCX	SX	SX
SU	SU	SU	SU	SU	SU	SX	SX	SX	SX	SX	SX	SU	SX
SX	SX	SX	SX	SX	SX	SX	SX	SX	SX	SX	SX	SX	SX

a) lock conversion matrix

Figure 7: taDOM2+ locking protocol

	-	IR	NR	LR	SR	IX	NRIX	CX	NRCX	NU	NX	SU	SX
IR	+	+	+	+	+	+	+	+	+	+	+	-	-
NR	+	+	+	+	+	+	+	+	+	-	-	-	-
LR	+	+	+	+	+	+	+	-	-	-	-	-	-
SR	+	+	+	+	+	-	-	-	-	-	-	-	-
IX	+	+	+	+	-	+	+	+	+	+	+	-	-
NRIX	+	+	+	+	-	+	+	+	+	-	-	-	-
CX	+	+	+	-	-	+	+	+	+	+	+	-	-
NRCX	+	+	+	-	-	+	+	+	+	-	-	-	-
NU	+	+	+	+	+	+	+	+	+	-	-	-	-
NX	+	+	-	-	-	+	-	+	-	-	-	-	-
SU	+	+	+	+	+	-	-	-	-	-	-	-	-
SX	+	-	-	-	-	-	-	-	-	-	-	-	-

Figure 8: taDOM3 lock compatibility matrix

not on a direct-child node of *c*.

- An LRCX lock mode (level read child exclusive) locks the context node together with all its direct-child nodes for shared access and indicates an exclusive lock on one of these child nodes.
- An SRCX lock mode (subtree read child exclusive) locks the context node *c* as the root of subtree *s* to perform read operations on *s* and indicates exclusive access to one of the direct-child nodes of *c*.

Adding these new lock types to the lock compatibility and conversion matrices, we obtain the taDOM2+ protocol. Note, now all lock requests can be handled without accessing the stored XML document at all. For example, an existing LR lock and an IX request does not lead anymore to an NR lock on each direct-child node during conversion (like in taDOM2), but can now simply be replaced with an LRIX lock. The complete lock compatibility and conversion matrices of taDOM2+ are shown in Figure 7.

3.3 taDOM3

To support the modification of a context node by exclusively locking only the affected node and not its entire subtree, the taDOM2 and taDOM2+ protocols have introduced the so-called virtual name nodes (see Section 3.1). On the one hand, this approach enables improved concurrent transaction processing by reusing the existing locking protocols. But on the other hand, this enhanced processing carries the obligation to maintain two locks for each node (one lock for the actual node and a second one for the virtual name node). Of course, this management overhead reduces transaction throughput.

taDOM3 enriches our protocols with a special lock mode that allows locking a single node without affecting the attached subtree. In this way, the concurrent processing capabilities are preserved and only a single lock per node is maintained. The combined use of the lock modes IX and CX would only indicate the intention of write operations on some descendant nodes, but would not reveal information about read accesses to the nodes they are maintained for. For performance reasons, we cannot collect the entire locking history of nodes (otherwise for each node, several different lock modes would have to be recorded for the same transaction [9]); therefore, a currently requested IX on node *n* cannot be distinguished from an initial NR on *n* converted later to IX. For this reason, the new exclusive node lock provided in taDOM3 implies some refined lock modes:

- An NRIX lock mode (node read intention exclusive) locks a node in shared mode and, in addition, indicates the intention of an exclusive lock request somewhere in the subtree, but not on a direct-child node.
- An NRCX lock (node read child exclusive) locks the context node for read access and indicates an exclusive lock on one of its direct-child nodes.
- An NU lock mode (node update option) supports a read operation on the context node with the option to convert the mode for a subsequent write access or downgrade to a read lock (see lock mode SU or EU).
- An NX lock (node exclusive) locks the context node in exclusive mode for an update operation on the context node's content. The subtree attached to the context node is not affected by this lock.

Note again, these four new lock modes allow the same concurrent transaction processing capabilities as provided by the taDOM2 protocol with only one acquired lock per node. The concept of virtual name nodes is not required any longer. The corresponding lock compatibility and conversion matrices controlling the taDOM3 protocol are shown in Figure 8 and Figure 9. In contrast to the taDOM2 and taDOM2+ protocols, here the lock modes IR and NR embody different behaviors and have to be implemented both as individual lock modes.

	-	IR	NR	LR	SR	IX	NRIX	CX	NRCX	NU	NX	SU	SX
IR	IR	IR	NR	LR	SR	IX	NRIX	CX	NRCX	NU	NX	SU	SX
NR	NR	NR	NR	LR	SR	NRIX	NRIX	NRCX	NRCX	NR	NX	SU	SX
LR	LR	LR	LR	LR	SR	$NRIX_{NR}$	$NRIX_{NR}$	$NRCX_{NR}$	$NRCX_{NR}$	NU_{NR}	NX_{NR}	SU	SX
SR	SR	SR	SR	SR	SR	$NRIX_{SR}$	$NRIX_{SR}$	$NRCX_{SR}$	$NRCX_{SR}$	NU_{SR}	NX_{SR}	SR	SX
IX	IX	IX	NRIX	$NRIX_{NR}$	$NRIX_{SR}$	IX	NRIX	CX	NRCX	NX	NX	SX	SX
NRIX	NRIX	NRIX	NRIX	$NRIX_{NR}$	$NRIX_{SR}$	NRIX	NRIX	NRCX	NRCX	NX	NX	SX	SX
CX	CX	CX	NRCX	$NRCX_{NR}$	$NRCX_{SR}$	CX	NRCX	CX	NRCX	NX	NX	SX	SX
NRCX	NRCX	NRCX	NRCX	$NRCX_{NR}$	$NRCX_{SR}$	NRCX	NRCX	NRCX	NRCX	NX	NX	SX	SX
NU	NU	NU	NU	NU_{NR}	NU_{SR}	NX	NX	NX	NX	NU	NX	SU	SX
NX	NX	NX	NX	NX_{NR}	NX_{SR}	NX	NX	NX	NX	NX	NX	SX	SX
SU	SU	SU	SU	SU	SU	SX	SX	SX	SX	SU	SX	SU	SX
SX	SX	SX	SX	SX	SX	SX	SX	SX	SX	SX	SX	SX	SX

Figure 9: taDOM3 lock conversion matrix

	-	IR	NR	LR	SR	IX	NRIX	LRIX	SRIX	CX	NRCX	LRCX	SRCX	NU	LRNU	SRNU	NX	LRNX	SRNX	SU	SX
IR	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	-	-
NR	+	+	+	+	+	+	+	+	+	+	+	+	+	-	-	-	-	-	-	-	-
LR	+	+	+	+	+	+	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-
SR	+	+	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
IX	+	+	+	+	-	+	+	+	-	+	+	+	-	+	+	-	+	+	-	-	-
NRIX	+	+	+	+	-	+	+	+	-	+	+	+	-	-	-	-	-	-	-	-	-
LRIX	+	+	+	+	-	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-
SRIX	+	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
CX	+	+	+	-	-	+	+	-	-	+	+	-	-	+	-	-	+	-	-	-	-
NRCX	+	+	+	-	-	+	+	-	-	+	+	-	-	-	-	-	-	-	-	-	-
LRCX	+	+	+	-	-	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-
SRCX	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
NU	+	+	+	+	+	+	+	+	+	+	+	+	+	-	-	-	-	-	-	-	-
LRNU	+	+	+	+	+	+	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-
SRNU	+	+	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
NX	+	+	-	-	-	+	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-
LRNX	+	+	-	-	-	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
SRNX	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
SU	+	+	+	+	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
SX	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Figure 10: taDOM3+ lock compatibility matrix

3.4 taDOM3+

Similarly to the subscripted node lock conversions in Section 3.2, the taDOM3 protocol contains the lock conversion rules $NRIX_{NR}$, $NRCX_{NR}$, $NRIX_{SR}$, $NRCX_{SR}$, NU_{NR}, NU_{SR}, NX_{NR}, and $NX_{SR,}$ which cause explicit fetching of direct-child nodes—only to set the appropriate locks. In an analogous way to taDOM2+, the taDOM3+ protocol introduces eight tailored lock modes to prevent the lock manager from accessing nodes stored on external devices:

- An LRIX lock mode (level read intention exclusive) locks the context node *c* and all its direct-child nodes in shared mode and indicates an exclusive lock somewhere in the subtree of *c* on a non-direct-child node.
- An SRIX lock mode (subtree read intention exclusive) locks in addition to LRIX the entire subtree of the context node for shared access (and indicates an exclusive lock somewhere in the subtree).
- An LRCX lock mode (level read child exclusive) locks the context node and all its direct-child nodes in shared mode and indicates exclusive child locking on one of the child nodes.
- An SRCX lock mode (subtree read child exclusive) locks in addition to LRCX the entire subtree of the context node in shared read mode.
- An LRNU lock mode (level read node update option) locks all direct-child nodes of the context node *c* in shared mode and supports read operations on c with the option to convert the mode to write or back to read access later on.
- An SRNU lock mode (subtree read node update op-

	-	IR	NR	LR	SR	IX	NRIX	LRIX	SRIX	CX	NRCX	LRCX	SRCX	NU	LRNU	SRNU	NX	LRNX	SRNX	SU	SX
IR	IR	IR	NR	LR	SR	IX	NRIX	LRIX	SRIX	CX	NRCX	LRCX	SRCX	NU	LRNU	SRNU	NX	LRNX	SRNX	SU	SX
NR	NR	NR	NR	LR	SR	NRIX	NRIX	LRIX	SRIX	NRCX	NRCX	LRCX	SRCX	NR	LR	SR	NX	LRNX	SRNX	SU	SX
LR	LR	LR	LR	LR	SR	LRIX	LRIX	LRIX	SRIX	LRCX	LRCX	LRCX	SRCX	LRNU	LRNU	SRNU	LRNX	LRNX	SRNX	SU	SX
SR	SR	SR	SR	SR	SR	SRIX	SRIX	SRIX	SRIX	SRCX	SRCX	SRCX	SRCX	SRNU	SRNU	SRNU	SRNX	SRNX	SRNX	SR	SX
IX	IX	IX	NRIX	LRIX	SRIX	IX	NRIX	LRIX	SRIX	CX	NRCX	LRCX	SRCX	NX	LRNX	SRNX	NX	LRNX	SRNX	SX	SX
NRIX	NRIX	NRIX	NRIX	LRIX	SRIX	NRIX	NRIX	LRIX	SRIX	NRCX	NRCX	LRCX	SRCX	NX	LRNX	SRNX	NX	LRNX	SRNX	SX	SX
LRIX	LRIX	LRIX	LRIX	LRIX	SRIX	LRIX	LRIX	LRIX	SRIX	LRCX	LRCX	LRCX	SRCX	LRNX	LRNX	SRNX	LRNX	LRNX	SRNX	SX	SX
SRIX	SRIX	SRIX	SRIX	SRIX	SRIX	SRIX	SRIX	SRIX	SRIX	SRCX	SRCX	SRCX	SRCX	SRNX	SRNX	SRNX	SRNX	SRNX	SRNX	SX	SX
CX	CX	CX	NRCX	LRCX	SRCX	CX	NRCX	LRCX	SRCX	CX	NRCX	LRCX	SRCX	NX	LRNX	SRNX	NX	LRNX	SRNX	SX	SX
NRCX	NRCX	NRCX	NRCX	LRCX	SRCX	NRCX	NRCX	LRCX	SRCX	NRCX	NRCX	LRCX	SRCX	NX	LRNX	SRNX	NX	LRNX	SRNX	SX	SX
LRCX	LRCX	LRCX	LRCX	LRCX	SRCX	LRCX	LRCX	LRCX	SRCX	LRCX	LRCX	LRCX	SRCX	LRNX	LRNX	SRNX	LRNX	LRNX	SRNX	SX	SX
SRCX	SRCX	SRCX	SRCX	SRCX	SRCX	SRCX	SRCX	SRCX	SRCX	SRCX	SRCX	SRCX	SRCX	SRNX	SRNX	SRNX	SRNX	SRNX	SRNX	SX	SX
NU	NU	NU	NU	LRNU	SRNU	NX	NX	LRNX	SRNX	NX	NX	LRNX	SRNX	NU	LRNU	SRNU	NX	LRNX	SRNX	SU	SX
LRNU	LRNU	LRNU	LRNU	LRNU	SRNU	LRNX	LRNX	LRNX	SRNX	LRNX	LRNX	LRNX	SRNX	LRNU	LRNU	SRNU	LRNX	LRNX	SRNX	SU	SX
SRNU	SRNU	SRNU	SRNU	SRNU	SRNU	SRNX	SRNX	SRNX	SRNX	SRNX	SRNX	SRNX	SRNX	SRNU	SRNU	SRNU	SRNX	SRNX	SRNX	SU	SX
NX	NX	NX	NX	LRNX	SRNX	NX	NX	LRNX	SRNX	NX	NX	LRNX	SRNX	NX	LRNX	SRNX	NX	LRNX	SRNX	SX	SX
LRNX	LRNX	LRNX	LRNX	LRNX	SRNX	LRNX	LRNX	LRNX	SRNX	LRNX	LRNX	LRNX	SRNX	LRNX	LRNX	SRNX	LRNX	LRNX	SRNX	SX	SX
SRNX	SRNX	SRNX	SRNX	SRNX	SRNX	SRNX	LRNX	SRNX	SRNX	SRNX	SRNX	SRNX	SRNX	SRNX	SRNX	SRNX	SRNX	SRNX	SRNX	SX	SX
SU	SU	SU	SU	SU	SU	SX	SU	SX	SU	SX	SX	SX	SX	SU	SU	SU	SX	SX	SX	SU	SX
SX	SX	SX	SX	SX	SX	SX	SX	SX	SX	SX	SX	SX	SX	SX	SX	SX	SX	SX	SX	SX	SX

Figure 11: taDOM3+ lock conversion matrix

tion) locks additionally to LRNU the complete subtree of the context node in shared mode.

- An LRNX lock mode (level read node exclusive) locks all direct-child nodes of the context node *c* in shared mode and *c* itself in exclusive mode.
- An SRNX lock mode (subtree read node exclusive) locks the entire subtree of the context node *c* in shared mode and *c* itself in exclusive mode.

The node lock compatibility and conversion matrices of our most efficient locking protocol taDOM3+ are shown in Figure 10 and Figure 11.

3.5 Comparing the Locking Protocols

To illustrate the benefits and performance gains of our stepwise protocol evolution for XML data processing, we ran a benchmark comparing transaction throughputs and number of locks maintained. The XTCserver is installed on an *IBM xSeries 4-Xeon-Processor* machine, the client applications are running on an *IBM Thinkpad R32* connected via a 100 Mbit/s network, both running a Linux operating system.

We extended the *sample.xml* document in Figure 2 with a *chapters* element containing a random number (between 10 and 20) of *chapter* nodes, each with a *title* and a *summary* element, and created a library XML document with 25,000 books. This library document (184 MB) matching the taDOM model contains over 4.5 million XML nodes and is stored via the network connection in about 4 minutes into the server (an average bulk load performance of over one million nodes per minute).

In the benchmark, a single transaction reconstructs a random book for which it determines the nodes of the book structure by invoking the *getChildNodes()* operation at each level. This requires a lock for shared level access. After that, a randomly selected chapter is renamed (exclusive lock on the chapter name; CX and IX locks on the ancestor path) which enforces a lock conversion on the nodes holding the level read locks. The benchmark client starts 25 threads, each executing a constant workload with the sketched transaction operations for 5 minutes on the XTCserver. The number of successfully committed transactions and the maximal number of concurrently maintained locks are shown in Figure 12.

Comparing the protocols, the number of concurrently

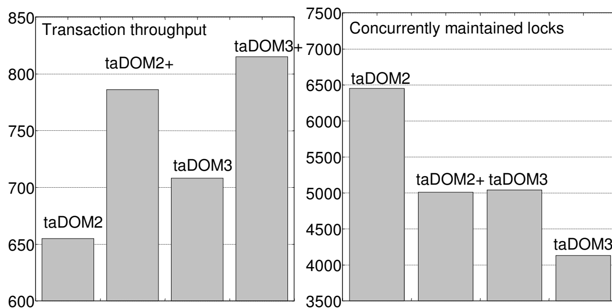


Figure 12: Comparing the locking protocols

maintained locks is dramatically reduced. First, this is caused by the especially tailored locks (from taDOM2(3) to taDOM2(3)+) which avoid lock requests on direct-child nodes when performing the subscripted conversion rules. Second, NRIX, NRCX, NU, and NX locks introduced from taDOM2(+) to taDOM3(+) do not need additional virtual name nodes for each element and attribute node and, in turn, the corresponding locks. The number of successfully committed transactions is increasing from taDOM2 to taDOM2+ and from taDOM3 to taDOM3+, because the substantial costs of child-node accesses can be avoided. This improves performance in such a manner that even taDOM2+ enables more transaction commits than taDOM3: Hence, fetching document nodes (stored records) is more performance-critical than maintaining (even a high number of) locks.

4. Correctness of the Locking Protocols

To trust the taDOM locking protocols (each has to guarantee a correct schedule), to safely exploit their performance potential, and to establish them as implementation fundamentals to be taken seriously by XDBMS vendors, we describe the basic ideas of their correctness proof in this section. This approach requires considering the full database interface providing the 19 XDP operations introduced in Section 2.3. The complete report of the proof can be accessed via our website [2]; it comprises about 280 MB of generated HTML code and contains over 38,000 individually checked test cases and over 250,000 checked lock compatibilities. Here, we can only explain the rationale of our proof technique.

4.1 The Compatibility Matrices

Observing the diversity of lock requests, the "worst case" of a request is an SX or NX lock on the context node, a CX lock on its parent, and IX locks on each ancestor node up to the document root. Hence, all different types of requested lock mode constellations for executing an arbitrary operation on a context node can be discussed on the node-relationship graph shown in Figure 13.

For our proof, we describe in a first step the behavior of each XDP operation provided by our node manager with so-called *base operations*. For example, base operations are actions like *use first child edge*, *redirect next sibling edge*, *read previous sibling node*, or *write new context node value*. Using these base operations, we can determine the read and write sets of each XDP operation executed on any node within the graph of Figure 13.

In our proof, we specify *use cases* to "execute" an XDP operation *o* on the context node CO and define for each use case four *scenarios* in which the lock requests of operation o using one of our locking protocols are specified. For some operations, we have to specify multiple use cases. As an example, the lock requests of operation *getFirstChild()* depend on the fact whether or not the context node owns child nodes; this must be distinguished by two

different use cases. With a large number of resulting combinations, each XDP operation o_i is executed for each scenario in each use case on each node of the graph. Examining the combinations, we now calculate the read and write sets of the use case operation o and the compared operation o_i. A read-write, write-read, or write-write conflict (up to this point only caused by the description with base operations) indicates a prohibition for executing these XDP operations concurrently. In such a case—now considering also the requested locks of the operations in the current scenario—at least one lock incompatibility must occur to block the concurrent execution. The other way around, if only the read sets intersect or even both the read and write sets do not intersect at all, the XDP operations can be executed in parallel and, in consequence, all requested locks of the two operations must be compatible.

As already mentioned above, 38,000 checked operation executions with over 250,000 checked lock compatibilities prove the correctness of our locking protocols corresponding to the 19 XDP operations and their correct behavior specified via base operations.

4.2 The Conversion Matrices

To prove the correctness of the conversion matrices we first define the strength-relationship of lock modes.

A lock l_1 is *stronger* than a lock l_2 ($l_1 > l_2$) if each lock l_i that is incompatible to l_2 is also incompatible to l_1. This means, the lock requests blocked by an existing lock l_2 are also blocked by the stronger lock l_1 (l_1 may even block more lock requests).

If l_1 is not stronger than l_2 then l_1 is only *weaker* than l_2 ($l_1 < l_2$) if l_2 is stronger than l_1. As a conseqence this means, that there are also locks which are neither stronger nor weaker than each other (e.g., LR and CX).

Corresponding to the theory of serializability [7], these definitions can be used to preserve the operation execution sequences of interlocked transactions: If transaction T_1 holds a lock l_1 for operation o_1 and operation o_2 of transaction T_2 is blocked on this lock until the end of T_1 (where all locks of T_1 are released), then the replacement of l_1 with a stronger lock l_1' blocks the execution of o_2 until the end of T_1 in the same manner. This means if the rules specified by the lock conversion matrices lead in each case to a resulting lock that is equal or stronger than both the previously existing lock and the requested lock, then this lock conversion preserves the operation sequences of the transactions.

A first special situation occurs for the update option locks. A downgrade request that sets the update option lock down to a weaker shared lock mode (and which would cause a violation of the correctness criteria defined above) requires the additional check of transitivity relationships. The downgrade conversion of an update lock down to a weaker shared lock mode is allowed if, for each existing lock l_e which is replaced with an update lock l_u, all locks, to which l_u may be converted to, are equal or stronger than the orginially existing lock l_e. For example, considering node lock conversion in taDOM2, an existing NR lock can be converted to SU. This is correct because SU may be converted to SR, SU, or SX, and all of them are still stronger than NR. In contrast, the conversion of an existing IX lock for a requested SU must obtain an SX. Although a resulting SU would be stronger than the existing IX and equal to the requested SU, in a following step SU may be converted down to SR which is not stronger than the previously acquired IX and would lead to an inconsistent lock state in this way.

The second special situation occurs for the subscripted lock conversion rules in taDOM2 and taDOM3 (e.g., an existing IX is converted to IX_{NR} for a requested LR). Although the resulting IX on the context node is not stronger than the requested LR, this conversion is correct. Of course, the resulting IX also blocks all requests that are blocked by the previously existing IX, because they are

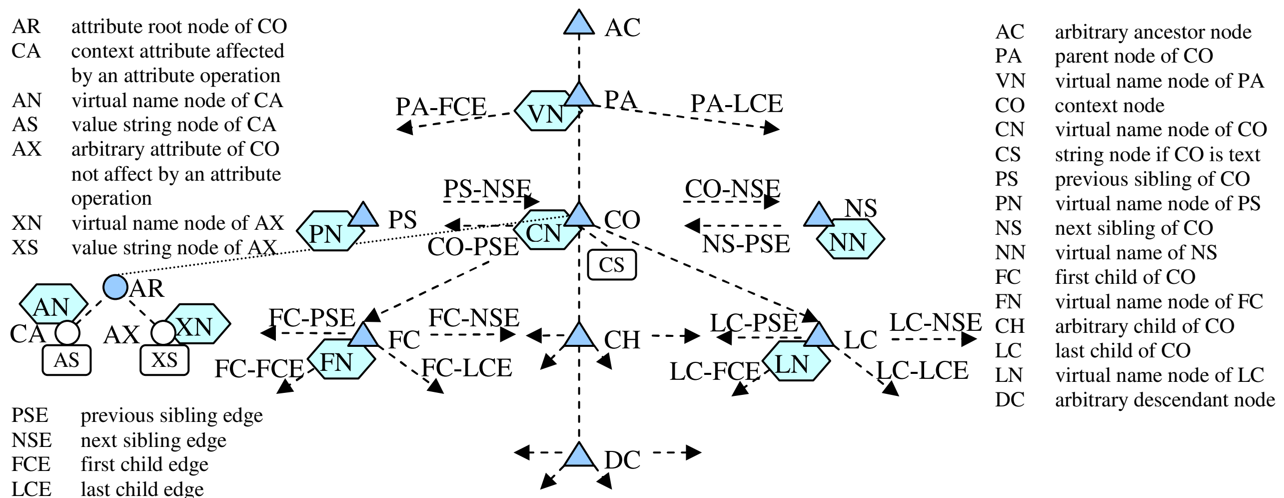


Figure 13: General operation execution on context node and surrounding elements

equal. At a first sight, looking at the compatibility matrix, CX is not blocked by the resulting IX, but this is required for the requested LR. Considering the additionally required NR locks on each child (IX_{NR}), a CX lock request cannot be granted. All locks, requested on any child node and causing a CX lock on the context node (these are SX in taDOM2 and taDOM3 and NX in taDOM3), are incompatible to the conversion-acquired NR locks on each child node. In that way, the compatible CX lock on the context node is acceptable, because the lock request will not be completed due to the incompatibility of SX and NX to the NR locks on the child nodes. Further new children cannot be added because of the acquired shared edge locks applying the IX_{NR} rule (see again Section 3.1).

Checking now the strength-relationships of the existing, requested, and converted node locks, and considering the two described special situations above (over 32,000 conditions), we can also prove the correctness of our conversion matrices.

Comprising sections 4.1 and 4.2, the complete correctness of our locking protocols is proved by the correctness of both the compatibility and conversion matrices.

5. Conclusions and Future Work

In this paper, we explored transaction isolation issues for collaborative XML document processing. We first sketched the design and implementation of our native XDBMS prototype and described the provided XDP operations. For concurrent transaction processing, we introduced our concepts enabling fine-granular concurrency control on taDOM trees representing XML documents. A tailored node identification algorithm supports native document storage and maintenance by providing for lifetime stable DeweyIDs. As the key part, we have introduced four locking protocols for direct and navigational access to individual XML nodes, thereby supporting different isolation strategies. The performance evaluation has compared their locking overhead and transaction throughput capabilities and has strongly confirmed the viability and effectiveness of our approaches. Finally, we explained our solution to prove the correctness of the protocols corresponding to a semantic description of the XDP interface. Our proof procedure systematically generates all ever possible operation execution constellations, determines their read-set and write-set intersections, and verifies the corresponding node and edge lock compatibilities with nearly 300,000 separately checked situations.

In our next steps, we concentrate on providing for an efficient phantom prevention for transactions using the SAX, DOM, and XQuery interfaces in parallel. Currently, phantoms are only prevented by our navigation locks on document areas traversed by navigation steps; but phantoms may also occur if nodes are directly addressed by their DeweyIDs via secondary index structures (e.g. accessing elements by indexed ID/IDREF values). In our design, this problem is to be solved by extended key-range locks which, in addition to a specified key range, are acquired on selected fragments of an XML document. In summary, we then can present a locking scheme which enables strict serializability and supports transactions using all common XML interfaces.

References

[1] D. Brownell. SAX2 - Processing XML Efficently with Java. O'Reilly (Jan. 2002)

[2] The XTC Project Website, Database and Information Systems Group, University of Kaiserslautern, http://wwwdvs.informatik.uni-kl.de/agdbis/projects/xtc

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

[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] T. Fiebig, S. Helmer, C.-C. Kanne, G. Moerkotte, J. Neumann, R. Schiele, T. Westmann. Anatomy of a native XML base management system. VLDB Journal 11, 292-314 (2002)

[6] T. Grabs, K. Böhm, H.-J. Schek: XMLTM: Efficient transaction management for XML documents. Proc. CIKM 2002: 142-152

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

[8] M. Haustein, T. Härder. Adjustable Transaction Isolation in XML Database Management Systems. Proc. 2nd International Database Symposium Toronto, Canada, LNCS 3186, 173-188 (2004)

[9] M. Haustein, T. Härder. A Lock Manager for Collaborative Processing of Natively Stored XML Documents, in: Proc. 19th Brazilian Symposium on Databases (SBBD), Brasilia, Brazil, 230-244 (2004)

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

[11] OASIS Open Document Format for Office Applications, http://www.oasis-open.org

[12] P. O'Neil, E. O'Neil, S. Pal, I. Cseri, G. Schaller, N. Westbury. ORDPATHs : Insert-Friendly XML Node Labels. Proc. SIGMOD 2004, Paris, France, 903-908 (2004)

[13] Document Object Model (DOM) Level 2 / Level 3 Core Specification, W3C Recommendation (Nov. 2000 / Apr. 2004)

[14] XQuery 1.0: An XML Query Language. W3C Working Draft (Oct. 2004)

[15] XUpdate – XML Update Language. http://xmldb-org.sourceforge.net/xupdate/