Available online at www.sciencedirect.com
ScienceDirect
Data & Knowledge Engineering xxx (2007) xxx–xxx
DATA & KNOWLEDGE ENGINEERING
www.elsevier.com/locate/datak

Optimizing lock protocols for native XML processing ☆

Michael Haustein, Theo Härder *

Database and Information Systems, Department of Computer Science, University of Kaiserslautern, D-67663 Kaiserslautern, Germany

Received 15 July 2007; accepted 5 November 2007

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 our prototype native XML database system called XML Transaction Coordinator (XTC) and specify the operations for accessing and modifying stored documents. The key contribution is the design and optimization of fine-grained lock protocols supporting collaborative processing of XML documents. For this reason, we introduce four XML lock protocols of growing sophistication and complexity, which are based on a tree-structured DOM storage model. The lock modes of these protocols, called taDOM* lock protocols, are tailor-made for the operations of the DOM API. Because of the protocols' complexity, their correctness is not obvious; hence, we present the ideas to prove the lock protocol correctness guaranteeing the specified data processing behavior of the given XDP operations. Finally, using XTC as our testbed system, we run extensive performance measurements to empirically evaluate our lock protocols and to compare their performance behavior against all known fine-grained competitor protocols under the same benchmark in an identical system setting. It turns out that tailor-made optimization pays off and that the taDOM* protocols are the clear winners in our lock protocol contest.

Keywords: DOM-based XML processing; XML lock protocols; Fine-grained locking; Lock compatibility; Lock conversion

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

☆ This work has been supported by the Rheinland–Pfalz cluster of excellence "Dependable adaptive systems and mathematical modeling" (see www.dasmod.de).
* Corresponding author. Tel.: +49 631 205 4030; fax: +49 631 205 3299.
E-mail address: haerder@informatik.uni-kl.de (T. Härder).
URL: http://www.lgis.informatik.uni-kl.de/cms/index.php?id=30 (T. Härder).

0169-023X/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.datak.2007.11.001

business data or applying XML standards for collaboration in word-processing applications [22] using database backends. However, concurrency control in RDBMSs ignores the properties of the semi-structured XML data model and causes disastrous locking behavior by blocking entire CLOBs, tables, or unnecessarily large index ranges.

Updating XML documents currently becomes a first-class issue for XML language models (XQuery Update Facility [28]). But today's 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. Frequent 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.

1.1. Problem statement and contribution

Although predicate locking protecting the concurrent evaluation of declarative statements of the XQuery language [27] and the XQuery Update Facility [28] 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. Beyond, tree locks or key-range locks [21] are not sufficient to support fine-grained locking for concurrently evaluating stream-, navigation- and path-based queries. 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 [26] and SAX [2], because their operations correspond more or less directly to a navigational access model. We have proposed a fine-granular lock protocol called taDOM2 in [16] which enables concurrent execution of transactions using either DOM, SAX, XQuery separately or all of them simultaneously. We will refine and optimize it as taDOM2+. The recent standard *DOM Level 3* [26] 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) [15] 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 [13] 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. Finally, to confirm the practicality and superiority of our proposal, we conduct an extensive comparative performance analysis where all known XML lock protocols (in total 12) were empirically evaluated using the same XTC environment and XML workload.

1.2. Related work

Basically, we want to develop lock protocols for tree-structured documents on which operations adhering to navigational and declarative language models are processed. XML structures and their languages have no properties in common with model properties such as encapsulation and inheritance hierarchies primarily considered by the object-oriented database community. From the research work of the relational community, we will borrow and refine ideas developed in the context of hierarchical lock protocols [12].

As already mentioned, no (freely available) XDBMS implementation exists which would allow exploring fine-grained XML concurrency control. As a rare example, Natix [8] conceptually supports concurrent transaction processing (see Section 5), but multi-user mode is not implemented yet. Hence, the few publications related to our problem either refer to coarse simulations [19] or to conceptual work only. XMLTM [10] 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 on the RDBMS thereby confined to the "relational" lock modes. The DGLOCK concept of XMLTM isolates transactions by managing path locks on a DataGuide [9] and, in this way, provides for concurrent path-based transaction processing to the clients. This is achieved by XDGL [24] in a similar way. Note, path-based processing only covers a small subset of the XQuery language. On the other hand, because a DataGuide provides a kind of summarized or schema-like structure for the underlying

Please cite this article in press as: M. Haustein, T. Härder, Optimizing lock protocols for native XML processing, Data Knowl. Eng. (2007), doi:10.1016/j.datak.2007.11.001

XML document, it only supports coarse-grained lock protocols, i.e., it locks at the level of path classes instead of single path instances. Furthermore, it cannot support ID-based access (direct jumps to internal nodes) and position-based predicates and is not tailored to fine-grained navigational access. Other path-oriented protocols are proposed in [5,6,20] which also seem 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 lock 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 lock protocols for the present XDP operations. Section 5 presents the approach to and the results of our performance measurements. Finally, Section 6 wraps up with conclusions.

2. Native 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 DOM model and taDOM storage 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. We sketch internal processing mechanisms only as far as needed to understand the concurrency control aspects.

2.1. System architecture

The five-layer architecture of our XTC system is depicted in Fig. 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. 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 is in charge of meta-data management for the database. The node manager implementing the navigational access layer transforms the records from their internal physical into an external

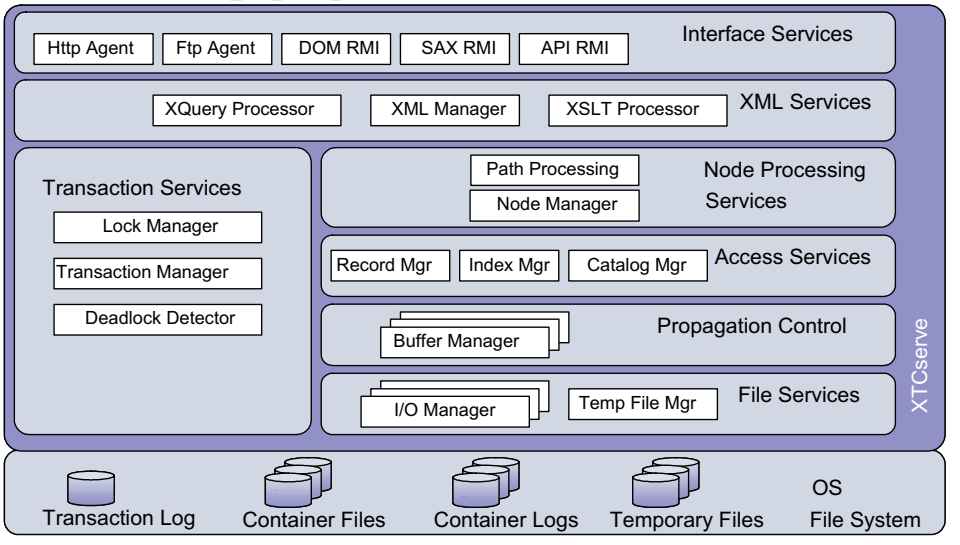


Fig. 1. XTC architecture overview.

Please cite this article in press as: M. Haustein, T. Härder, Optimizing lock protocols for native XML processing, Data Knowl. Eng. (2007), doi:10.1016/j.datak.2007.11.001

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. At the top of our architecture, the agents of the interface layer offer the XML and node services 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 offers methods to execute query statements and to browse or manipulate XML documents and materialized query results. All client-side activities are protected by transactions running in one of the well-known isolation levels uncommitted, committed, repeatable, or serializable [13].

2.2. DOM model and taDOM storage model

Our data model corresponds to that of the DOM standard [26]. A simple navigational language model for it is provided by the API's of DOM2 and DOM3. In a multi-lingual XDBMS, more powerful and declarative language models such as XPath and XQuery [27] are needed in addition. To achieve fine-granular access to document trees, often declarative requests have to be translated into sequences of node accesses which again can be described by DOM operations. While the DOM model with its element, attribute, and text nodes is also used for the physical document representation, the lock manager refers to a specialized internal document representation (the so-called taDOM tree). In this way, efficient and effective isolation of concurrent XDP operations can be facilitated. To improve fine-granular locking, we have introduced – exclusively for the lock manager's view – two new node types: attribute root and string. This representational extension 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 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 Fig. 2.

The attribute root separates attribute nodes from their element node. Instead of locking all attribute nodes 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 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.

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 record order within logically consecutive pages (B*-tree leaves chained by next/previous pointers).

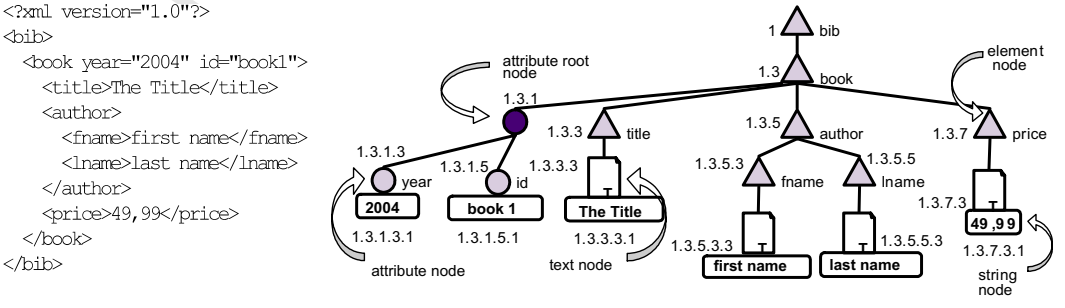


Fig. 2. Transformation of *sample.xml* into a taDOM tree.

Please cite this article in press as: M. Haustein, T. Härder, Optimizing lock protocols for native XML processing, Data Knowl. Eng. (2007), doi:10.1016/j.datak.2007.11.001

As it turned out in the course of the XTC implementation, the node labeling scheme is of utmost importance for the internal processing of XML trees and, in particular, for locking. Therefore, we compared and evaluated the spectrum of conceivable approaches.[1] While both range-based and prefix-based labeling schemes usually support comparison and evaluation of all XPath axes, only prefix-based schemes representing some kind of *Dewey Decimal Classification* directly deliver the labels of all ancestor nodes. Inspired by the ORDPATH approach [23], a number of such prefix-based labeling schemes were proposed [14] which are similar in main processing properties and only differ in some aspects such as overflow technique for dynamically inserted nodes, attribute node labeling, or encoding mechanism. Because all of them are equivalent for the internal processing tasks and, especially, for the support of the lock manager, we prefer to use the substitutional name *stable path labeling identifiers* (SPLIDs) for them.

General properties are the following: Each node label contains the label of its parent node as prefix (see Fig. 2). A node label consists of a sequence of so-called division values (separated by dots in the human-readable format). Odd division values indicate a level transition whereas even division values are used to support an overflow mechanism. Upon initial document storage, only odd division values are assigned, e.g., $d1 = 1.3.3$ and $d2 = 1.3.5$ may represent the labels of two consecutive nodes at level 3. A later insertion of a node at level 3 before $d2$ and after $d1$ would – to guarantee the labeling properties without the need to relabel (part of) the document – receive the label $d3 = 1.3.4.3$, which preserves the labeling properties, that is, it allows for correct level identification by counting simply the number of odd values, order preservation, node label comparison (e.g., $d3 < d2$), and ancestor determination (e.g., 1.3 and 1). Division value 1 at levels >1 is used to label attribute nodes (where order does not matter). An effective way to handle later insertions protracting the overflow mechanism is to provide for gaps in the labeling space, that is, to initially assign the division values $dist + 1$, $2*dist + 1$, etc. where the parameter dist governs the gap size. The minimum value $dist = 2$ should be applied to almost static XML documents whereas larger dist values avoid resorting too frequently to overflow values.

Here we can only summarize the benefits of the SPLID concept; for details, see [14]. It provides holistic system support. Comparison of two SPLIDs allows ordering of the related nodes in document order. As opposed to competing schemes, SPLIDs easily provide the IDs of all ancestors to enable intention locking of all nodes in the path up to the document root without any access to the document itself. Declarative queries are supported by the efficient evaluation – that is, computation without the need to access the document on disk – of *all query axes* occurring in XPath or XQuery path expressions. Existing SPLIDs are immutable, that is, they allow assignment of new IDs without the need to reorganize the IDs of nodes present. In theory, SPLIDs are free of maintenance under arbitrary insertions. Yet, implementation restrictions (e.g., key length <128 bytes in B-trees) may enforce subtree relabeling, either by a reorganization run or dynamically by a concurrent transaction (potentially aborting the violating transaction before). All SPLID properties are preserved and (equally important) relabeling only concerns the subtree.

A further advantage using SPLIDs 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 SPLID as the key part and the byte representation of the actual node as the value part. Efficient SPLID encoding based on Huffman trees could be further improved by applying prefix compression to them. The document order in the B*-tree greatly favored our approach such that storing a SPLID only consumed 2–3 bytes in the average. Stored element and attribute nodes are additionally compressed using a vocabulary. Instead of storing their names, tiny surrogates ($\leqslant$2 bytes) are used to identify them within a related tree data structure.

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( )*, *get-*

[1] Indeed, our results reported in [14] led us to redesign the existing mechanism in XTC, which was originally based on a straightforward sequential numbering scheme.

Please cite this article in press as: M. Haustein, T. Härder, Optimizing lock protocols for native XML processing, Data Knowl. Eng. (2007), doi:10.1016/j.datak.2007.11.001

NextSibling( ), *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 reconstruction of the XML fragment 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 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, whereas 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. Lock protocols

In existing systems, locking is the most important method for concurrency control; some variant of it is probably used in any relational DBMS. For performance reasons, fine-granular isolation at the record level is needed when accessing individual records, whereas coarse-granular locks are appropriate when traversing entire tables or the complete database, e.g., for a table scan or in case of recovery from a failure. Therefore, lock protocols, which enable the isolation of multiple granules each with a single lock, are especially beneficial, because they provide minimal overhead for lock management and, at the same time, the selection of adequate isolation units for sufficient separation of concurrent read/write transactions. For this reason, hierarchical lock protocols [12] – also denoted as multi-granularity locking – are used "everywhere" in the relational world. For example, records (or rows) of the same type are organized in a table, various tables may be grouped into a segment, and all segments together form the database. With this *conceptual hierarchy* – built up by the four levels of records, tables, segments, and database – objects at each level can be isolated acquiring the usual locks with modes R (read), X (exclusive), and U (update with conversion option), which implicitly lock all objects in the entire subtree addressed. To avoid lock conflicts when objects at different levels are locked, so-called intention locks with modes IR (intention read) or IX (intention exclusive) have to be acquired along the path from the root to the object to be isolated and vice versa when the locks are released [12].

3.1. Isolation needs in XML documents

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 logically organized by a tree structure, the principles of multi-granularity locking can be applied. Hence, hierarchical lock protocols have some ability to isolate the required cooperation/concurrency on XML documents [16]. Using the idea of "permeable" intention locks at higher tree levels, before subtrees are locked by read or write locks, the closest support for the protocols to be developed can be anticipated by them. Hence, by using some kind of intention locks on the entire ancestor path of the context node up to the document root, the lock manager can set appropriate locks on the context node or the related subtree and, in this way, automatically provide for lock granules as minimal and as weak as acceptable.

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) and these, in turn, cause the lock manager to provide adequate isolation before the actual operation is performed. Therefore, 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 lock protocols are confined to the lock manager and not visible to any other component,

Please cite this article in press as: M. Haustein, T. Härder, Optimizing lock protocols for native XML processing, Data Knowl. Eng. (2007), doi:10.1016/j.datak.2007.11.001

let alone the application. Note, only textbooks protocols are simple; even existing relational DBMSs exhibit protocols with about 15 lock modes.

Because our XML documents are stored in a B*-tree structure, the question whether or not specific tree-based lock protocols can be used immediately arises. So-called B-tree lock protocols provide for their structural consistency while concurrent database operations are querying or modifying database contents and its representation in B-tree indexes [11]. Such locks isolate concurrent operations on B-trees and are called latches in the database world.[2] For example, to minimize blocking or interference of concurrent transactions while traversing a B-tree, *latch coupling* acquires a latch for each B-tree page before the traversal operation is accessing it and immediately releases this latch when the latch for the successor page is granted or at end of operation at the latest [1]. In contrast, locks isolate concurrent transactions on user data and – to guarantee serializability [7] – have to be kept until transaction commit. Therefore, such latches only serve for consistent processing of (logically redundant) B-tree structures and do not address the isolation of concurrent read/write operations on (non-redundant) user data, as sketched in Section 2.3. With similar arguments, index locking cannot appropriately cope with the navigational DOM operations [21].

Our protocols to be derived for document tree locking are 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 the key ideas of the four lock protocols taDOM2, taDOM2+, taDOM3, and taDOM3+ and represent them by their lock modes, compatibility matrices, and conversion matrices in the following sections.[3] Finally, in Section 3.6, we compare their relative performance and cost by measuring transaction throughput and recording the number of concurrently maintained locks.

3.2. taDOM2

Providing tailor-made lock modes and granules for the operations to be isolated, we begin our optimization of XML lock protocols with the language standard DOM2 [25]. The resulting taDOM2 protocol is based on the protocol we presented in [16]. Due to observations in our empirical XTC experiments, we have recently refined and optimized its lock compatibilities and conversions.

3.2.1. Lock compatibilities

We differentiate 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. Fig. 3a contains the compatibility matrix for our basic lock modes. In the following, the matrix header row characterizes the current lock state of objects, whereas the matrix header column indicates the mode of incoming lock requests. 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 (as for multi-granularity locking).

[2] Unfortunately, this mechanism is denoted by the term "lock" in the literature on operating systems and programming environments.
[3] We use for the description of a lock protocol the same semantics and the representation techniques based on compatibility matrix and conversion matrix as known from database textbooks [13].

Please cite this article in press as: M. Haustein, T. Härder, Optimizing lock protocols for native XML processing, Data Knowl. Eng. (2007), doi:10.1016/j.datak.2007.11.001

	-	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

Fig. 3. taDOM2 lock protocol.

- An NR lock mode (node read) is requested for reading context node *c*. 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 context node *c* together with its direct-child nodes for shared access. For example, evaluation of the child axis only requires an LR lock on context node *c* 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 *getAttributes()* operations).
- An SR lock mode (subtree read) is requested for 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 multi-granularity locking), but not on a direct child of the node being locked (in contrast to CX locks).
- 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 (the compatibility is then decided on the child nodes themselves).
- 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, an asymmetry is 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 context node *c* (updating its contents or deleting *c* and its entire subtree), an SX lock mode (subtree exclusive) is needed for *c*. It requires a CX lock for its parent node and an IX lock for all other ancestors up to the document root element.

While the lock modes IR, IX, SR, SU, and SX can be more or less directly inferred from the traditional "relational" multi-granularity lock protocol, the remaining lock modes are specifically designed for the optimization of DOM-based XML concurrency. NR read-locks a single context node *c* potentially in the middle of a document, where the path from the root to the parent of *c* is protected with IR locks. LR on a context node *c* optimizes the lock acquisition for all children of *c* when they have to be scanned, for example. Finally, CX on a context node *c* is introduced to primarily prevent LR and SR on *c* thereby enhancing the potential for concurrent read/write operations on children of *c*.

Please cite this article in press as: M. Haustein, T. Härder, Optimizing lock protocols for native XML processing, Data Knowl. Eng. (2007), doi:10.1016/j.datak.2007.11.001

As indicated in Fig. 3a, the use of IR and NR modes exhibits identical lock behavior – that is, the compatibilities of IR and NR are the same – in the protocols taDOM2 and taDOM2+ (see below). In a real implementation (like our XTCserver), they can be replaced with a single proxy lock mode (e.g., NR). Later on, their compatibilities will differ in the protocols taDOM3 and taDOM3+. Therefore, we keep both lock modes separate for completeness and simplicity of protocol comparison. Note also in Fig. 3a, the differing behavior of CX and IX locks is needed to enable compatibility of IX and LR locks and to enforce incompatibility of CX and LR locks.

Fig. 4 illustrates the lock state of the following processing scenario. Transaction T_1 starts modifying the value *last name* and, therefore, acquires an SX lock for the corresponding string node. The lock manager complements this action by acquiring a CX lock for the parent node and IX locks for all further ancestors. 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.

3.2.2. Lock conversions

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 runtime 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 [17]. The corresponding rules are specified by the *lock conversion matrix* in Fig. 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*. 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 below.

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 children 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 lock 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

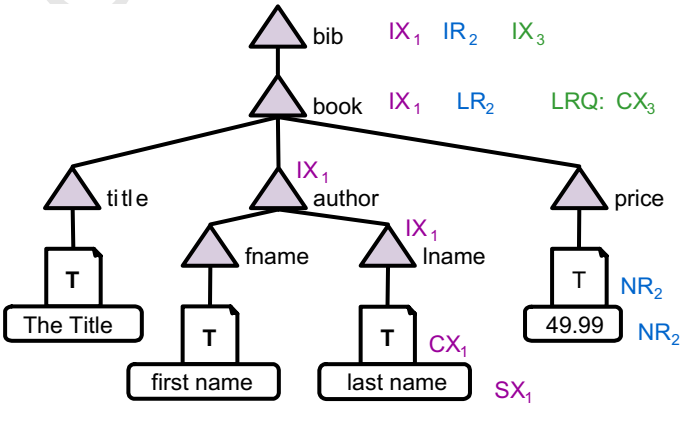


Fig. 4. taDOM2 locking example.

Please cite this article in press as: M. Haustein, T. Härder, Optimizing lock protocols for native XML processing, Data Knowl. Eng. (2007), doi:10.1016/j.datak.2007.11.001

acquires an NR lock on each direct-child node of *c* (except the node which is already locked for deletion by SX).

3.2.3. Navigation locks

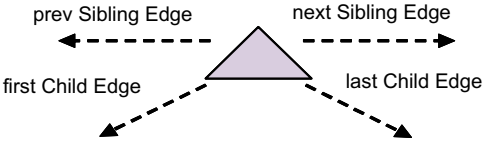
In addition to the node lock management described above, we maintain so-called *navigation locks* to isolate navigation paths. This means, a sequence of navigational method calls or modification operations – starting at a known node within the taDOM tree – must always yield the same sequence of result 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 Fig. 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, therefore, any insertion of a new *book* node is prevented by this LR lock, because it is incompatible to the CX lock required for such an insertion. 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 [15] within the taDOM tree (Fig. 5) which are locked in addition to their confining nodes.

While traversing an XML document, a transaction has to request a lock for each edge, in addition to the node locks. Traversal operations between nodes need bidirectional isolation: If, e.g., *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 must lock the last-child edge of the parent node of *c*, because the null value indicates the last-child position of *c*. To support such traversals efficiently, we offer three lock modes corresponding to R/U/X known from normal record locking:

- 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 (not materialized) objects 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.



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

Fig. 5. Virtual navigation edges and lock modes.

Please cite this article in press as: M. Haustein, T. Härder, Optimizing lock protocols for native XML processing, Data Knowl. Eng. (2007), doi:10.1016/j.datak.2007.11.001

3.2.4. Use of virtual name nodes

The lock protocol taDOM2, consisting of the node lock compatibility and conversion matrices and the virtual navigation edge locks described so far, is able to isolate all methods specified in *DOM Level 2* [26] 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 requires exclusive locking of the element node. With a tailor-made lock 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, 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+ lock protocols 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. 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 SPLID of its owning element extended with a 0. For example, assuming the assigned SPLIDs of Fig. 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. Fig. 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 lock 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 SR lock on *lname* comply with the existing IX and CX locks of T_1. Transaction T_3 which wants to access 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.

Additionally, further options for lock management are provided in XTC. The concepts of tunable node lock granularity and lock escalation [15] to reduce the number of locks maintained at the cost of potentially larger lock granules is not considered in the focus of this paper. Furthermore, implementation details of the lock manager can be found in [17].

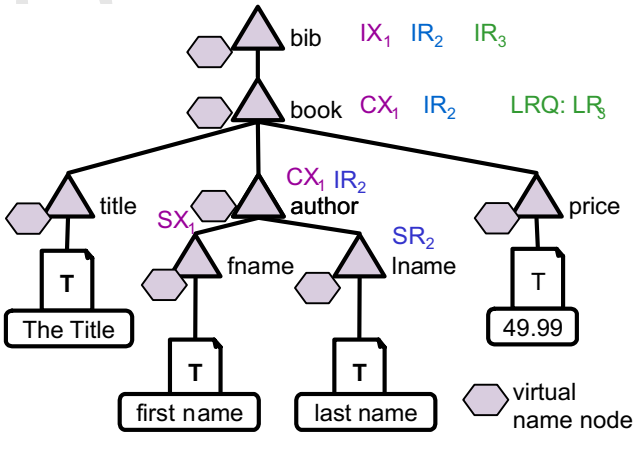


Fig. 6. Locks on virtual name nodes.

Please cite this article in press as: M. Haustein, T. Härder, Optimizing lock protocols for native XML processing, Data Knowl. Eng. (2007), doi:10.1016/j.datak.2007.11.001

3.3. taDOM2+

Considering again the lock conversion matrix of the taDOM2 lock protocol in Fig. 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 runtime overhead on the XDBMS. It is true that, for a given SPLID 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 context node c 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 context node c and its entire subtree to perform read operations and indicates the intention to perform write operations somewhere in that subtree, but not on a direct-child node of c.
- An LRCX lock mode (level read child exclusive) locks context node c 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 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 Fig. 7.

3.4. 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 3.2.4). On the one hand, this approach enables improved concurrent transaction processing by reusing the existing lock 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 [17]); 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.

Please cite this article in press as: M. Haustein, T. Härder, Optimizing lock protocols for native XML processing, Data Knowl. Eng. (2007), doi:10.1016/j.datak.2007.11.001

	-	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

(b) Lock conversion matrix

Fig. 7. taDOM2+ lock protocol.

- An NRCX lock (node read child exclusive) locks context node *c* 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 context node *c* 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 context node *c* 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. Hence, we have found a more elegant solution for the isolation of specific operations such as the *renameNode( )* method of the *DOM Level 3* specification and the *setValue( )* operation of our node manager executed on an inner element node. The related lock compatibility and conversion matrices controlling taDOM3 are shown in Figs. 8 and 9. In contrast to taDOM2 and taDOM2+, here the lock modes IR and NR exhibit different behaviors.

Please cite this article in press as: M. Haustein, T. Härder, Optimizing lock protocols for native XML processing, Data Knowl. Eng. (2007), doi:10.1016/j.datak.2007.11.001

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

Fig. 8. taDOM3 lock compatibility matrix.

	-	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

Fig. 9. taDOM3 lock conversion matrix.

3.5. taDOM3+

In Section 3.3, we have discussed that lock conversions affecting the children of a context node *c* are very expensive because the lock manager can only supply their node labels by accessing the document (usually) stored on disk. The solution in taDOM2+ was achieved by introducing tailor-made intention locks whose use prevented such I/O-intensive and, therefore, performance-critical conversion situations. Similarly to taDOM2, the taDOM3 protocol also contains a number of conversion rules. In case of a corresponding lock conversion (see Fig. 9), $NRIX_{NR}$, $NRCX_{NR}$, $NRIX_{SR}$, $NRCX_{SR}$, NU_{NR}, NU_{SR}, NX_{NR}, and NX_{SR} would cause explicit fetching of direct-child nodes or reading their node labels – only to set the appropriate locks. Therefore, in an analogous way to taDOM2+, we introduce for the taDOM3+ protocol the following eight tailor-made lock modes to prevent these conversions:

Please cite this article in press as: M. Haustein, T. Härder, Optimizing lock protocols for native XML processing, Data Knowl. Eng. (2007), doi:10.1016/j.datak.2007.11.001

- An LRIX lock mode (level read intention exclusive) locks 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 context node c 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 mode.
- An LRNU lock mode (level read node update option) locks all direct-child nodes of context node c in shared mode and supports read operations on c with the option to convert the mode to write or to read access later on.
- An SRNU lock mode (subtree read node update option) locks additionally to LRNU the complete subtree of context node c in shared mode.
- An LRNX lock mode (level read node exclusive) locks all direct-child nodes of 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 context node c in shared mode and c itself in exclusive mode.

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

3.6. Comparing the lock 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 equipped with 4 Intel XEON processors (1.50 GHz each), 4 GB memory, and an IDE disk with 280 GB. The client applications are running on an IBM Thinkpad R32 connected via a 100 Mbit/s network, both running Linux.

	-	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	+	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Fig. 10. taDOM3+ lock compatibility matrix.

Please cite this article in press as: M. Haustein, T. Härder, Optimizing lock protocols for native XML processing, Data Knowl. Eng. (2007), doi:10.1016/j.datak.2007.11.001

	-	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	SRNX	SRNX	SRNX	SRNX	SRNX	SRNX	SRNX	SRNX	SRNX	SRNX	SRNX	SRNX	SRNX	SX	SX
SU	SU	SU	SU	SU	SU	SX	SX	SX	SX	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

Fig. 11. taDOM3+ lock conversion matrix.

We extended the *sample.xml* document in Fig. 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. Finally, this library document (184 MB) matching the taDOM model contains over 4.5 million XML nodes and is stored with an average bulk load performance of over one million nodes per minute.

In the benchmark, a single transaction completely reconstructs a random book 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 min on the XTCserver. The number of successfully committed transactions and the maximal number of concurrently maintained locks are shown in Fig. 12.

Comparing the protocols, the number of concurrently maintained locks is dramatically reduced. First, this is caused by the especially tailored locks (from taDOM2(3) to taDOM2(3)+) which avoid the subscripted conversion rules. Second, NRIX, NRCX, NU, and NX locks introduced from taDOM2(+) to taDOM3(+) do not need additional virtual name nodes and 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 lock protocols

Each of the proposed lock protocols has to provide serializability for concurrent transactions [7] executing sequences of arbitrary DOM operations on XML documents, that is, the lock manager running these protocols has to guarantee correct schedules. To trust the taDOM lock protocols, to safely exploit their performance potential, and to establish them as implementation fundamentals to be taken seriously by XDBMS vendors, we will describe the basic ideas of their correctness proof.

Please cite this article in press as: M. Haustein, T. Härder, Optimizing lock protocols for native XML processing, Data Knowl. Eng. (2007), doi:10.1016/j.datak.2007.11.001

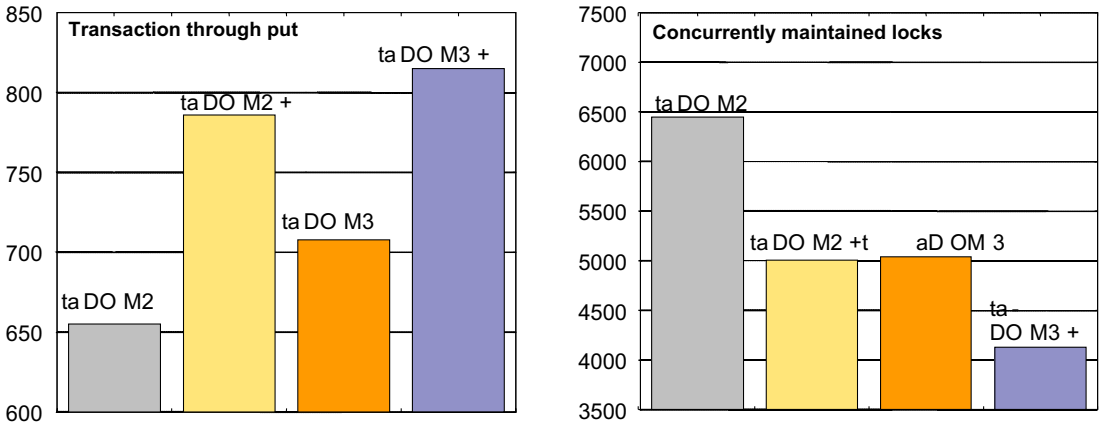


Fig. 12. Comparing the lock protocols.

Here, we can only explain the rationale of our proof technique. This kind of model checking requires considering the full DB interface providing the 19 XDP operations sketched in Section 2.3 and the derivation of 36 *use cases* covering all possible processing situations on XML documents. For each of the four lock protocols, a use case contains – besides its semantic description with base operations – a so-called scenario. A scenario describes the set of locks which have to be acquired to enable the execution of the use case under the lock protocol considered. In a concrete system, these lock requests have to coincide with the implementation of the taDOM operations. For example, given protocol taDOM2 and the use case for operation *getFirstChild( )* (with an existing first child node), NR locks have to be acquired on the context node c, its first child node f, and their virtual name nodes. Furthermore, ER locks have to be acquired on the *firstChild* edge of c and the *prevSibling* edge of f. In this way, the read-sets and write-sets of an operation execution can be determined. Hence, for all possible operation execution constellations, the intersections of the appearing read-sets and write-sets can be calculated, such that conflicting situations can be detected and the compatibilities for the related node locks and edge locks verified. More details can be found in [15].

4.1. Compatibility matrices

The schematic representation of operation executions reveals the complexity of the model checking process and the diversity of lock requests. As visualized in Fig. 13, all different types of lock mode constellations for executing arbitrary operations on a context node can be shown on the node-relationship graph. For example, the "worst case" request is an SX or NX lock on the context node, a CX lock on its parent, and IX locks on each ancestor node.

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-sets and write-sets of each XDP operation executed on any node within the graph of Fig. 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 lock 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-sets 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 that the concurrent execution of these XDP operations is prohibited. In such a case – now consid-

Please cite this article in press as: M. Haustein, T. Härder, Optimizing lock protocols for native XML processing, Data Knowl. Eng. (2007), doi:10.1016/j.datak.2007.11.001

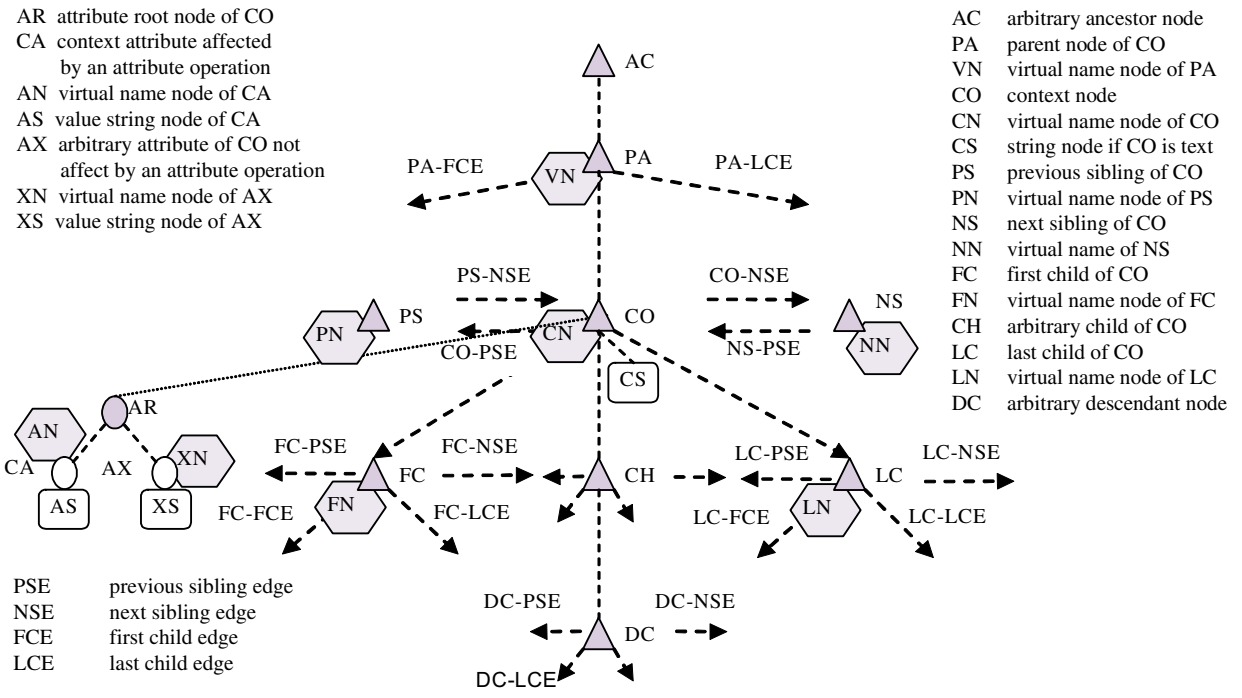


Fig. 13. General operation execution on context node and surrounding elements.

ering also the requested locks of the operations in the current scenario – at least one lock mode incompatibility must occur to block the concurrent execution. The other way around, if only the read-sets intersect or even both the read-sets and write-sets do not intersect at all, the XDP operations can be executed concurrently and, as a consequence, all requested locks of the participating operations must be compatible.

4.2. Conversion matrices

To prove the correctness of 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 consequence, there are also locks which are neither stronger nor weaker than each other (e.g., LR and CX).

Considering serializability theory [7], these definitions are 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 way. Hence, 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 originally 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 con-

trast, 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., for an existing IX, conversion rule IX_{NR} is applied 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 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 3.2).

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 Section 4, the complete correctness of our lock protocols is proved by the correctness of both the compatibility and conversion matrices.

A description of all 36 use cases is contained in [15]. The complete proof report can be accessed via our website [29]; 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.

5. Comparative evaluation of lock protocols

In order to empirically evaluate the *four lock protocols* of the taDOM group, we ran extensive performance measurements. Because isolated performance evaluation of a single protocol or a group of similar protocols cannot be related to other types of protocols and has therefore less expressiveness and power of persuasion, we additionally prepared and implemented all known competitor protocols in the framework of XTC such that all protocols are processed in an identical XDBMS environment and setting.

5.1. Competing lock protocols

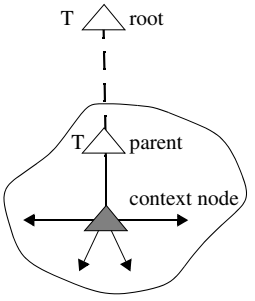
Most proposals mentioned in Section 1.2 are either schema-based and, therefore, coarse-granular, e.g., they are based on a DataGuide (DGLOCK protocol), or designed to be executed in an extra layer (XMLTM protocol) "above" a relational DBMS [10] or too restrictive for general XDBMS use, e.g. limited to XPath-oriented predicates only [5,6]. In the literature, only few ideas for fine-granular lock protocol design can be found [12,19]. These competitor protocols can be divided into two groups sketched in the following. Let us begin with Node2PL and its variations making up the first group denoted *-2PL.

5.2. Node2PL and its followers

The primary objective of Node2PL is synchronization of transactions concurrently performing navigation and modification operations on the document tree. Starting from the document root, so-called *structure locks* are used in Node2PL to appropriately lock the parent (typically an element node) of the context node to which the navigation or update operation is applied – as illustrated in Fig. 14, the assumed read navigation to the context node leaves T (traverse) locks on its path from the root.

Furthermore, Node2PL strictly distinguishes structure-based and content-based accesses using different lock types. Hence, to change a node's content (e.g., of a text node), so-called *content locks* are used. In addition, a third lock type is introduced to protect direct jumps to nodes. A transaction directly jumping to a node addressed by an ID attribute acquires for it a special read or write lock (IDR, IDX). If the related subtree is to

Please cite this article in press as: M. Haustein, T. Härder, Optimizing lock protocols for native XML processing, Data Knowl. Eng. (2007), doi:10.1016/j.datak.2007.11.001



structure locks

	T	M
T	+	-
M	-	-

locks for direct jumps

	IDR	IDX
IDR	+	-
IDX	-	-

content locks

	S	X
S	+	-
X	-	-

Fig. 14. Node2PL sample protocol and compatibilities for different lock types.

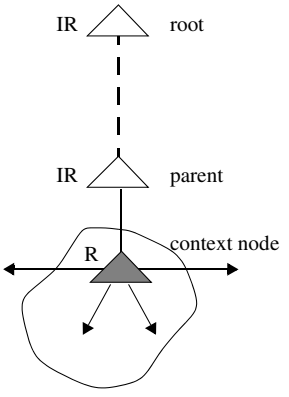
be deleted, IDX locks on all elements owning ID attributes must guarantee that no other transaction jumping into this subtree reads or updates it. If subtrees are large, this may imply a very expensive procedure. Such a penalty is especially performance-critical, because direct jumps may be rather frequent, for example, if query processing uses indexes. Note, when the context node in the example of Fig. 14 is to be updated later, lock conversion (to the M (modify) mode) on the parent node is mandatory. Such conversions are a source of deadlocks in all protocols; this danger may only be alleviated by tailored intention locks. For the details of lock conversion, we refer to [19]. Node2PL is unnecessarily restrictive because, by locking the parent, it blocks the entire level of the context node, and not only its direct neighborhood. As a refinement of Node2PL's structure locks, NO2PL locks in case of updates only the nodes reachable from the context node thereby reducing its blocking granularity. Further optimizations are offered by a third variant OO2PL locking for navigation operations only the traversed edges and for update operations only the affected navigation edges (again see [19]).

5.3. Multi-granularity locking for XML trees

Compared to taDOM*, the *-2PL group has some serious practical disadvantages; the most critical ones are handling of direct jumps by special lock modes IDR/IDX, missing modes for locking entire subtrees, and missing support for some operations, e.g., direct jumps to indexed element nodes not owning any ID attribute. For these reasons, we tried to avoid these drawbacks by adapting the well-known *MGL protocols* [12] – originally introduced for tables – to XML trees. As compared to classical MGL, a main difference is the double role of intention locks to indicate read/write operations deeper in the tree and to lock nodes (without locking the attached subtrees). Another difference are the much more complex conversion rules. When applied to the context node, the locks on its entire ancestor path have to be converted, too. Furthermore, we have combined the protocols with a lock depth parameter, the importance of which we have experienced in our measurements (see Section 5.5).[4]

In this way, we have derived a group of MGL protocols based on a *common intention lock* (IRX) and enhanced by special conversion rules (IRX+), on *separate intention locks for read/write* (IRIX) and enhanced by special conversion rules (IRIX+), and finally an IRIX protocol enhanced by RIX and U modes [13] – called URIX and shown in Fig. 15. Special edge locks as introduced in [16] complement the node locks shown for the URIX protocol. As an example, assume no further locks are present in the protocol of Fig. 15, then a lock conversion of the context node to X can be performed by converting IR to IX on the ancestor path and R to X on the context node. In contrast to the *-2PL group, direct jumps must be protected by locking the entire ancestor path in suitable mode. This is very efficient when using SPLIDs (see Section 2.2) for node identification. Hence, our comparative lock protocol evaluation comprises the taDOM group, the *-2PL group, and the MGL group with a total of 12 protocols.

[4] Lock depth *n* determines that, while navigating through the document, individual locks are acquired for existing nodes up to level *n*. If necessary, all nodes below level *n*are locked by a subtree lock at level *n*.



lock compatibility matrix

	IR	IX	R	RIX	U	X
IR	+	+	+	+	-	-
IX	+	+	-	-	-	-
R	+	-	+	-	-	-
RIX	+	-	-	-	-	-
U	+	-	+	-	-	-
X	-	-	-	-	-	-

lock conversion matrix

	IR	IX	R	RIX	U	X
IR	IR	IX	R	RIX	U	X
IX	IX	IX	RIX	RIX	X	X
R	R	RIX	R	RIX	R	X
RIX	RIX	RIX	RIX	RIX	X	X
U	U	X	U	X	U	X
X	X	X	X	X	X	X

Fig. 15. URIX protocol compatibilities and conversion rule.

5.4. Runtime environment

Our testbed environment consists of a server machine and a number of workstations (see Section 3.6) which are connected via a 100 Mbit ethernet.

5.4.1. Meta-synchronization

To compare the results in the most accurate way, it is indispensable to run all experiments in the same XDBMS setting using the same database and the same workload. Therefore, a system is needed capable of running concurrency control experiments using different lock protocols in the same physical environment. As a prerequisite, we had developed our XTC system primarily as a testbed for empirical concurrency control. The key idea to really enable cross-protocol comparison was the appropriate isolation of the XTC lock manager as a kind of abstract data type. It accepts the locking requests from the XTC node manager (and other components) in a more abstract form as so-called meta-lock requests including

- node locks (shared, update, exclusive);
- shared level locks;
- tree locks (shared, update, exclusive);
- edge locks (shared, update, exclusive) for previous sibling, next sibling, first child, and last child;
- as well as release locks at commit for isolation level *repeatable read* or at end of operation for isolation levels *uncommitted read* and *committed read*.

When using this meta-synchronization, XTC has to map the meta-lock requests to the actual locking algorithm which is achieved by the lock manager's interface. Hence, exchanging the lock manager's interface implementation exchanges the system's complete XML lock mechanism. In this way, we could run XTC in our experiments with many different lock protocols. At the same time, all experiments were performed on the taDOM storage model which is optimized for fine-grained management of XML documents by using a refined node structure and SPLIDs. All mandatory concepts of a lock manager are introduced in [13]. Our implementation including lock requests, lock conversions, and lock waits is described in [15].

5.4.2. Framework TaMix for XML benchmarks

Unfortunately, existing benchmarks do not match the requirements of transactional XML updates. XMach-1 [3], for example, is designed for scalable, multi-user Web applications, but targets the XDBMS behavior, in general, without specific emphasis on concurrency control. In contrast, the scope of XMark [25] is the XML query processor and concentrates on single-user mode only. XOO7 [4] also targets on the query processor as its scope and has no particular application orientation. Hence, benchmarks for our specific

Please cite this article in press as: M. Haustein, T. Härder, Optimizing lock protocols for native XML processing, Data Knowl. Eng. (2007), doi:10.1016/j.datak.2007.11.001

needs are missing – primarily, because so far XDBMS research was strongly focused on retrieval only. Therefore, we had to design tailored benchmarks together with an automated measurement environment.

To support expressive transaction runs, we have designed and implemented the framework TaMix for benchmarks on XML documents. TaMix provides an automated runtime environment where a specified number of TaMix clients execute transactions according to a given schedule while XTC serves their concurrent DB requests. Typical tasks of the TaMix coordinator are starting and stopping the XTC server and the TaMix clients. TaMix records the measurement results for the specified metrics to enable later evaluation. Furthermore, in cooperation with the XTCdeadlockDetector, it collects data in case of deadlocks about the number of active transactions, the locks held, the state of the wait-for-graph, etc. Thus, we are able to analyze deadlock events precisely, e.g., whether it was caused by lock conversion (frequent occurrence) or by lock requests in separate subtrees (rather rare cases).

Here we can give only a brief overview of the transaction types emulating a banking application and their activations in our experimental transaction mix. Our benchmark database is sketched in Fig. 16 and comprised about 580,000 taDOM nodes (with 10,000 *customers* and 25,000 *accounts*) resulting in a DB size of ~8 MB. The following transaction types were provided:

- *Executing bank transfers*: The program jumps to a randomly selected account element, navigates through the document and performs a few updates on *balance* and *posting*. Continuously, 5 TX/client were activated.
- *Processing standing_orders*: After accessing a random *account*, the program navigates to *standing_orders*. When reading all *orders*, the child axis is evaluated resulting in small fraction of update operations (5 TX/client).
- *Renaming customer master elements*: A *customer* element is renamed (1 TX/client). In parallel, randomly selected *customer* fragments were reconstructed (5 TX/client).
- *Creation of account statements*: The program reconstructs randomly selected *account* fragments including a small amount of update operations, e.g., insertion of an entry in *protocols* (5 TX/client).
- *Removal of customers*: A randomly selected *customer* (together with its subtree) is removed from the database which simulates the deletion of fragments (2 TX/client).

In a first experiment, these five transaction types were executed in isolation to reveal specific properties of the protocols under given operations. Then, to check their throughput behavior and susceptibility for deadlocks, all transactions were executed in a controlled mix. In all cases, three TaMix clients were continuously running the specified number of transactions. The transaction mix processed all transaction types in parallel such that a constant system load of 66 transactions was maintained. The TaMix-specific parameters characterizing the variation of all test runs for our 12 protocols were chosen as follows:

- lock depths where applicable: 0–5

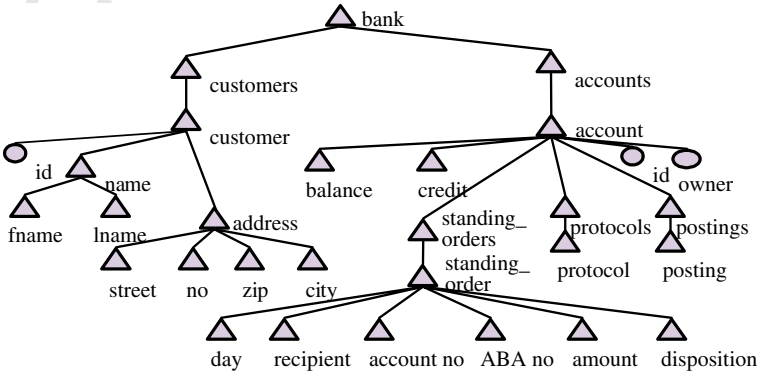
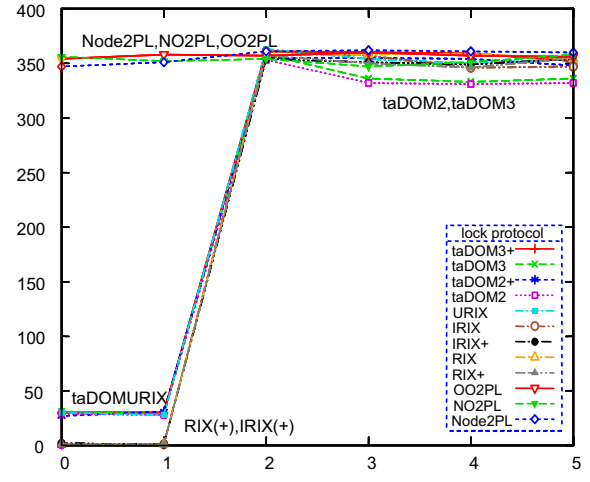
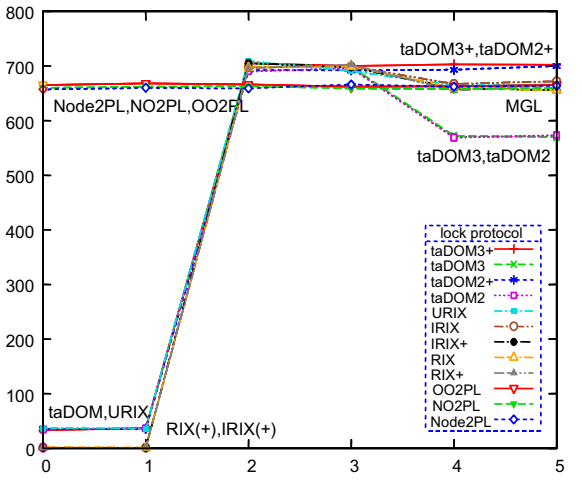


Fig. 16. Benchmark database (DataGuide).

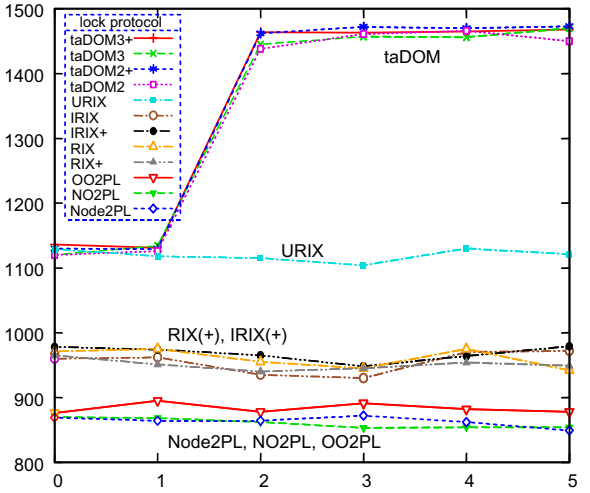
Please cite this article in press as: M. Haustein, T. Härder, Optimizing lock protocols for native XML processing, Data Knowl. Eng. (2007), doi:10.1016/j.datak.2007.11.001



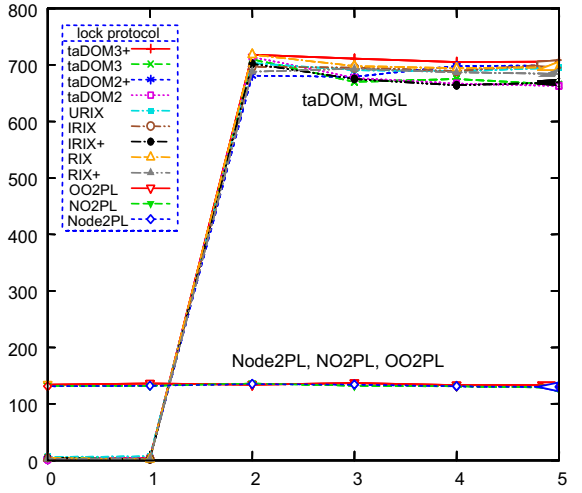
(a) Executing bank transfers



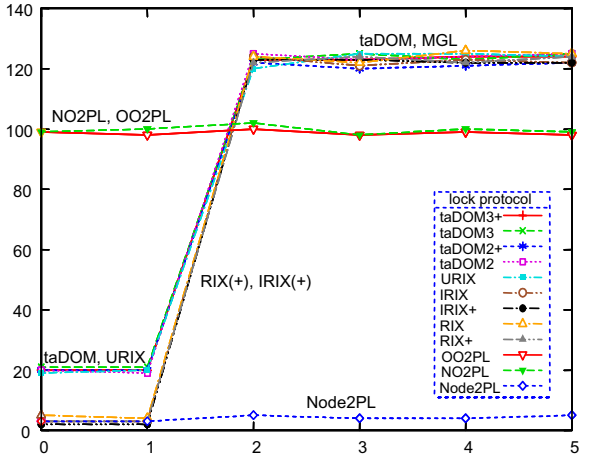
(b) Processing standing_orders



(c) Renaming customer master elements



(d) Creation of account statements



(e) Removal of customers

Fig. 17. Processing of isolated transaction types (no. of successful transactions over lock depth).

Please cite this article in press as: M. Haustein, T. Härder, Optimizing lock protocols for native XML processing, Data Knowl. Eng. (2007), doi:10.1016/j.datak.2007.11.001

- number of runs per lock depth: 3
- run duration: 5 min, waitAfterCommit: 2500 ms, waitAfterOperation: 100 ms
- random wait before executing the first operation of a transaction: 0–5000 ms

5.5. Results of the contest

The stronger the isolation level[5], the higher the consistency guarantees of the XDBMS, but the less transaction throughput has to be achieved, in general. This expectation was proven by experiments described in [18] for XML databases, too. Therefore, we assume strict isolation of transactions in our experiments. As usual for lock performance experiments, in all test cases the lock manager enforced isolation level *repeatable read*. Given the navigational operations in our transactions, this level produces results equivalent to isolation level *serializable*.

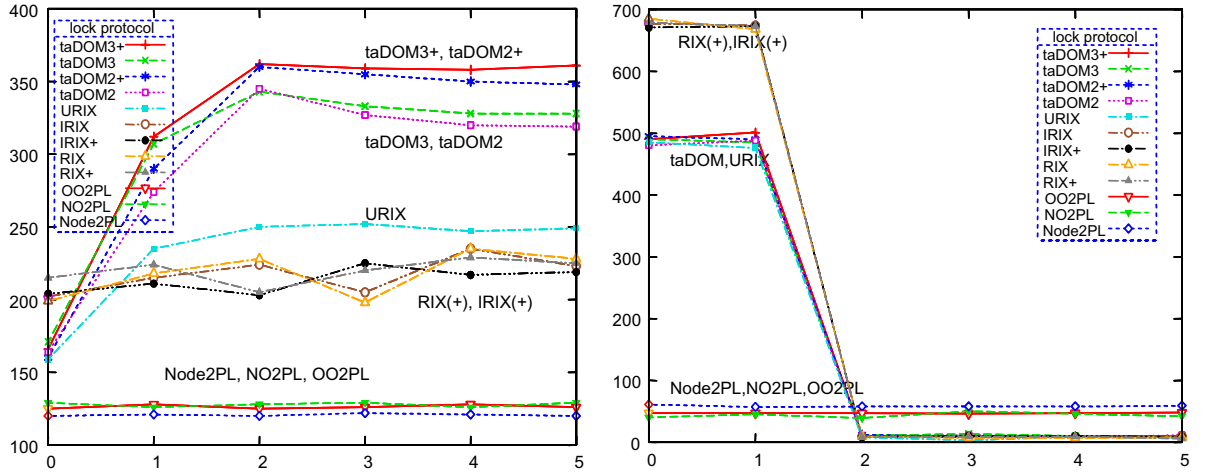
Lock granularity is a major performance factor for lock protocols. Therefore, we vary in all test runs the lock depth parameter and illustrate the number of successful or aborted transactions as a function of this parameter. Because the *-2PL group is not equipped with a lock depth parameter, their protocols show more or less constant transaction throughput behavior. On the other hand, both other protocol groups contrast that fine-granular locks are very important for satisfactory performance. Note, lock depth 0 corresponds to the use of document locks, which explains the low performance. The higher the lock depth parameter, the smaller are the subtrees locked. As a consequence, throughput rapidly increases to a level where further refinement does not enhance anymore parallelism. Of course, lock depth 0 or 1 results in unnecessarily large lock granules and should therefore not be applied.

With these explanations we can illustrate the results of isolated processing of the five transaction types in Fig. 17. Protocols of the *-2PL group can only keep up when read operations are dominating and write operations are local and dispersed such that conflicts hardly occur; this is particularly true for *bank transfers* and *standing_orders* where the entry points to the XML document are randomly selected. In contrast, the results for taDOM* and MGL* coincide in many cases where only intention locks and standard operations on subtrees are needed. However, when tailor-made locking and conversion support is required, the advantages of taDOM* are revealed. Such a situation occurs when renaming *customer* master elements. Obviously, the URIX protocol also benefits from the specialized conversion support in this situation and remarkably performs better than the other MGL* protocols.

Fig. 18a summarizes the transaction throughput of the entire mix of banking transactions. The first impression concerns the clear gaps separating of the various groups (*-2PL, MGL*, taDOM*), which highlights the relative performance advantages. Obviously, this summary reveals that the taDOM group is the clear winner. Furthermore, it confirms that fine-tuned lock modes pay off when fine-granular concurrency control is needed. For example, taDOM2+ and taDOM3+ show greater "permeance" for parallel read/write operations with growing lock depth. Although the MGL* group can keep up very well with the best protocols in some depth ranges, it has to experience strong drawbacks in other situations. For example, it does not provide lock modes equivalent to LR or CX or it cannot separate the name from the content of an element. Therefore, it cannot optimize locking in such situations. As a result, the MGL* group ends up in the middle position when drawing the average performance over all transaction types. Fig. 18b illustrates the number of transaction aborts (due to deadlocks) experienced while the entire transaction mix was processed. Neglect the results for lock depths 0 and 1 (which are only added for completeness), the superiority of taDOM* and MGL* is again clearly shown. The number of transaction aborts has to be contrasted with the number of successful transaction commits. The resulting ratio is particularly bad for the *-2PL group; less than three successful transaction commits are accompanied by a transaction abort whereas this ratio is a very low percentage for both other groups.

[5] While none acquires no locks at all, all others need long write locks; *uncommitted* means no read locks, *committed* and *repeatable* short and long read locks, respectively.

Please cite this article in press as: M. Haustein, T. Härder, Optimizing lock protocols for native XML processing, Data Knowl. Eng. (2007), doi:10.1016/j.datak.2007.11.001



(a) Number of committed transactions (b) Number of aborted transactions

Fig. 18. Transaction mix of the banking benchmark (variation of lock depth).

Hence, we can safely conclude: As compared to the *-2PL group, we obtain in our experiments in the average ~80% and ~200% throughput gain for the MGL* group and taDOM* group, respectively, while less deadlocks are provoked by them. Furthermore, our experiment nicely illuminates the average performance gain accomplished by fine-grained locks tailored to the effects of the operations to be isolated. However, detailed explanations of the locking behavior are impossible for these aggregated results.

6. Conclusions

In this paper, we systematically explored the optimization of lock protocols for transaction isolation of collaborative XDP applications. We first sketched the design and implementation of our native XDBMS prototype and described the XDP operations provided. For concurrent transaction processing, we introduced our concepts enabling fine-granular locking on taDOM trees representing XML documents. Notably, the SPLID concept greatly facilitated native document storage and maintenance. As key part, we have introduced four lock protocols providing tailormade lock compatibility/conversion modes of growing complexity. These protocols support direct and navigational access to individual XML nodes, thereby enabling different isolation strategies. The performance evaluation has compared their locking overhead and transaction throughput capabilities and has strongly confirmed viability and effectiveness of our approaches. Furthermore, 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 related node and edge lock compatibilities with nearly 300,000 separately checked situations.

To relate the performance gained by our approach to that of other approaches, we have implemented all published protocols for fine-grained locking in XTC and explored them under the same banking benchmark in an identical system environment. All these protocols took advantage of our SPLID scheme. Similar results with an earlier version of XTC are reported in [18] where we primarily focus on isolation levels. The taDOM* protocols clearly won this contest and achieved substantial throughput gain with infrequent transaction aborts. In summary, we feel that fine-grained locking together with lock modes tailored to operations to be protected is a great and practical idea.

So far, we have developed our protocols for tree-shaped XML document structures. Because our approach protects nodes and node sets specified by their SPLIDs and SPLIDs, in turn, uniquely identify the ancestor paths to the root, it should be easily possible to extend the protocols that they can cope with relationships (IDREF, IDREFS) in an XML document. Even recursive, graph-shaped structures should be amenable as long as the SPLID mechanism remains unique on the underlying physical document representation.

Please cite this article in press as: M. Haustein, T. Härder, Optimizing lock protocols for native XML processing, Data Knowl. Eng. (2007), doi:10.1016/j.datak.2007.11.001

Acknowledgements

We thank the anonymous referees for their helpful hints to improve the readability of this paper. The support of Andreas Bühmann while formating the final version is appreciated.

References

[1] R. Bayer, M. Schkolnick, Concurrency of operations on B-trees, Acta Inform. 9 (1977) 1–21.
[2] D. Brownell, SAX2 – Processing XML Efficiently with Java, O'Reilly, January 2002.
[3] T. Bühme, E. Rahm, XMach-1: A Benchmark for XML Data Management, Proc. National German Database Conference (BTW 2001), Inform. aktuell, Springer, 2001, pp. 264–273.
[4] S. Bressan, M.L. Lee, Y.G. Li, Z. Lacroix, U. Nambiar, The XOO7 Benchmark, http://www.comp.nus.edu.sg/ebh/XOO7.html.
[5] S. Dekeyser, J. Hidders, J. Paredaens, A Transaction Model for XML Databases, World Wide Web J. 7 (2) (2004) 29–57.
[6] S. Dekeyser, J. Hidders, Path Locks for XML Document Collaboration, in: Proc. 3rd Conf. on Web Information Systems Engineering (WISE), Singapore, 2002, pp. 105–114.
[7] K.P. Eswaran, J. Gray, R.A. Lorie, I.L. Traiger, The notions of consistency and predicate locks in a database system, Commun. ACM 19 (11) (1976) 624–633.
[8] T. Fiebig, S. Helmer, C.-C. Kanne, G. Moerkotte, J. Neumann, R. Schiele, T. Westmann, Anatomy of a native XML base management system, VLDB J. 11 (2002) 292–314.
[9] R. Goldman, J. Widom, DataGuides: Enabling query formulation and optimization in semistructured databases, Proc. VLDB (1997) 436–445.
[10] T. Grabs, K. Bühm, H.-J. Schek, XMLTM: efficient transaction management for XML documents, in: Proc. ACM CIKM Conf., McLean, VA, 2002, pp. 142–152.
[11] G. Graefe, Hierarchical locking in B-tree indexes, in: Proc. National German Database Conference (BTW 2007), LNI P-65, Springer, 2007, pp. 18–42.
[12] J. Gray, Notes on database operating systems, in: Operating Systems: An Advanced Course, LNCS 60, Springer-Verlag, 1978, pp. 393–481.
[13] J. Gray, A. Reuter, Transaction Processing: Concepts and Techniques, Morgan Kaufmann, 1993.
[14] T. Härder, M. Haustein, C. Mathis, M. Wagner, Node labeling schemes for dynamic XML documents reconsidered, Data & Knowledge Engineering 60(1), Elsevier, 2007, pp. 126–149.
[15] M. Haustein: Fine-Ganular Transaction Isolation in Native XML DBS (in German), Ph.D. Thesis, Univ. Kaiserslautern, 2006.
[16] M. Haustein, T. Härder, Adjustable Transaction Isolation in XML Database Management Systems, in: Proc. 2nd Int. XML Database Symposium, Toronto, LNCS 3186, 2004, pp. 173–188.
[17] 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, 2004, pp. 230–244.
[18] M. Haustein, T. Härder, K. Luttenberger, Contest of XML Lock Protocols, in: Proc. VLDB, 2006, pp. 1069–1080.
[19] S. Helmer, C.-C. Kanne, G. Moerkotte, Evaluating lock-based protocols for cooperation on XML documents, SIGMOD Record 33 (1) (2004) 58–63.
[20] K.-F. Jea, S.-Y. Chen, A high concurrency XPath-based locking protocol for XML databases, Inform. Software Technol. 48 (8) (2006) 708–716.
[21] C. Mohan, ARIES/KVL: A key-value locking method for concurrency control of multiaction transactions operating on B-tree indexes, Proc. VLDB (1990) 392–405.
[22] OASIS Open Document Format for Office Applications, http://www.oasis-open.org.
[23] P.E. O'Neil, E.J. O'Neil, S. Pal, I. Cseri, G. Schaller, N. Westbury, ORDPATHs: insert-friendly XML node labels, Proc. SIGMOD Conf. (2004) 903–908.
[24] P. Pleshachkov, P. Chardin, S. Kusnetzov, XDGL: XPath-based concurrency control protocol for XML data, in: Proc. 22nd British National Conference on Databases (BNCOD), UK Bd. 3567, Springer, 2005, pp. 145–154.
[25] A. Schmidt, F. Waas, M. Kersten, XMark: A benchmark for XML data management, Proc. VLDB (2002) 974–985.
[26] Document Object Model (DOM) Level 2/Level 3 Core Specific, W3C Recommendation.
[27] XQuery 1.0: An XML Query Language, http://www.w3.org/XML/XQuery.
[28] XQuery Update Facility, http://www.w3.org/TR/xqupdate.
[29] XTC Project Website, DBIS Group, University of Kaiserslautern, http://wwwlgis.informatik.uni-kl.de/cms/index.php?id=134.

Please cite this article in press as: M. Haustein, T. Härder, Optimizing lock protocols for native XML processing, Data Knowl. Eng. (2007), doi:10.1016/j.datak.2007.11.001

Michael Haustein studied computer science at the University of Kaiserslautern from 1995 to 2002. That followed a scientific staff membership at the chair of Prof. Härder to the end of 2005, where he designed and implemented most parts of the native XML database management system XTC and finished his doctoral examination procedure. In 2006, he changed to SAP AG, Walldorf, and in 2007 as a manager for online applications to Pro-MediSoft AG, Mannheim.

Theo Härder obtained his Ph.D. degree in Computer Science from the TU Darmstadt in 1975. In 1976, he spent a post-doctoral year at the IBM Research Lab in San Jose and joined the project System R. In 1978, he was associate professor for Computer Science at the TU Darmstadt. As a full professor, he is leading the research group DBIS at the TU Kaiserslautern since 1980. He is the recipient of the Konrad Zuse Medal (2001) and the Alwin Walther Medal (2004) and obtained the Honorary Doctoral Degree from the Computer Science Department of the University of Oldenburg in 2002. His research interests are in all areas of database and information systems – in particular, DBMS architecture, transaction systems, information integration, and Web information systems. He is author/coauthor of seven textbooks and of more than 200 scientific contributions with >120 peer-reviewed conference papers and >60 journal publications. His professional services include numerous positions as chairman of the GI-Fachbereich "Databases and Information Systems", conference/program chairs and program committee member, editor-in-chief of Informatik – Forschung und Entwicklung (Springer), associate editor of Information Systems (Elsevier), World Wide Web (Kluwer), and Transactions on Database Systems (ACM). He served as a DFG (German Research Foundation) expert and was chairman of the Center for Computed-based Engineering Systems at the University of Kaiserslautern, member of two joint collaborative DFG research projects DFG (SFB 124, SFB 501), and co-coordinator of the National DFG Research Program "Object Bases for Experts".

Please cite this article in press as: M. Haustein, T. Härder, Optimizing lock protocols for native XML processing, Data Knowl. Eng. (2007), doi:10.1016/j.datak.2007.11.001