# Realizing Fine-Granular and Scalable Transaction Isolation in Native XML Databases*

Sebastian Bächle
University of Kaiserslautern
67663 Kaiserslautern, Germany
baechle@informatik.uni-kl.de

Theo Härder
University of Kaiserslautern
67663 Kaiserslautern, Germany
haerder@informatik.uni-kl.de

## Abstract

Based on loosely coupled services in an XML engine, we describe how to realize fine-grained lock protocols, which can guarantee transaction isolation for applications using different language models. We illustrate the superiority of the taDOM lock protocol family and its tailor-made lock modes and lock granules adjusted to the XML language model. We emphasize the importance of a prefix-based node labeling scheme for lock management. Using meta-locking, we have found the key concept for integration and evaluation of various isolation protocols that can even be exchanged at runtime without affecting other engine services. Benchmark runs convincingly illustrated the flexibility and performance benefits of our approach and revealed that careful lock protocol optimization pays off. Further, we present optimizations to enhance scalability of our lock protocols.

## 1. Motivation

Currently available relational database management systems (RDBMSs) only manage structured data well. There is no effective and straightforward way for handling semi-structured XML data. A "brute-force" mapping uses "long fields" or CLOBs where individual and direct access to single XML document nodes (elements or attributes) is not possible. Alternatively, an innumerable number of algorithms maps semi-structured XML data to structured relational database tables and columns (the so-called „shredding"). In any case, there are no specific provisions to process transactions on XML documents and, at the same time, to efficiently provide the ACID properties [14]. Especially isolation of concurrent transactions in RDBMSs is tailored to the relational data model and does not take the semi-structured data model and the typical XML document processing (XDP) interfaces into account. CLOBs or "shredded" mappings of XML documents to relational tables may cause disastrous locking behavior, in particular, if relational systems lock entire pages or even entire tables as their minimal lock granularity.

Native XML database systems promise tailored processing of XML documents, but most of the systems published in the DB literature are designed for efficient document retrieval and not for frequently concurrent and transaction-safe document modifications [20, 26]. This "retrieval-only" focus was probably caused by the early language proposals [29] where the update part was left out [30]. On the other hand, missing update requirements strongly influenced the design of their node labeling schemes used to identify XML elements. Hence, their schemes allow for very fast computation of structural dependencies, but modifications of the document structure often lead to renumeration of large document parts. A rare example of an update-oriented system, Natix claims to be designed for the support of concurrent modifications [8] using the DOM interface. Unfortunately, compelling results are not known so far, because it neither provides a suitable lock concept nor a running implementation (see Section 6).

Efficient and effective transaction-protected collaboration on XML documents becomes a pressing issue because of their number, size, and growing use. Tailor-made isolation protocols that take into account the tree characteristics of the documents and the operations of the workload are considered a viable solution. But, because of structure variations and workload changes, these protocols must exhibit a high degree of flexibility as well as automatic means of runtime adjustments.

Because a number of language models are available and standardized for XML [6, 29], a general solution for concurrency control has to support protocols for concurrently evaluating stream-, navigation-, and path-based queries. Furthermore, performance needs dictate the use of fine-grained isolation protocols as a concurrency control mechanism for transactional processing of or cooperative collaboration on XML documents. Hence, in a multi-lingual XML database management system (XDBMS), more powerful and declarative language models such as XPath and XQuery [29] are needed in addition to DOM-based models. To achieve fine-granular access to document trees, declarative requests have to be translated into sequences of DOM operations.

* This work has been partially supported by the German Research Foundation (DFG).

**Proceedings of the Spring Young Researcher's Colloquium On Database and Information Systems SYRCoDIS, St.-Petersburg, Russia, 2008**.

With these requirements, we necessarily have to map all declarative operations to a navigational access model, e.g., using the DOM operations, to provide for fine-granular concurrency control. Because of the superiority of locking (in other areas), we also focus on lock protocols for XML. We have already developed a group consisting of four DOM-based lock protocols called the taDOM family [18] and some other equivalent protocols are available from the relational world by adjusting the idea of hierarchical or multi-granularity locking [12] to the specific needs of XML trees. Here, we discuss mechanisms how such protocols can be efficiently integrated into a native XDBMS, but, on the other hand, sufficiently encapsulated from the other engine components such that they can be automatically exchanged or adapted to new processing situations at runtime.

In an XDBMS installation, we may have different XML language models (e.g., DOM. SAX, XPath, or XQuery) used by the spectrum of applications accessing the database. Hence, DB requests specified by different XML languages may be scheduled and arbitrary transaction mixes may occur. Therefore, adhering to the serializability requirement [7], we have to guarantee transaction isolation for applications using different language models and even for individual application programs using all or some of them at a time.

Our XDBMS prototype (XML Transaction Coordinator, [17]) called XTC served as a testbed for all implementations and comparative experiments. All techniques and mechanisms are provided by XTC and were empirically evaluated in our experiments. Of course, performance results continuously trigger improvements and refinements of these mechanisms.

We explain in Section 2 the properties of tree-based lock protocols and emphasize the need and advantage of lock protocols tailored to the specific characteristics of XML processing, before we sketch the novel aspects of taDOM2. In Section 3, we discuss the role of prefix-based node labeling for efficient lock management and outline the implementation of the XTC lock manager. Section 4 discusses infrastructural aspects of a lock service and sketches the concept of meta-locking. It outlines the mechanism underlying the runtime exchange of lock protocols which enabled a lock protocol contest with XTC. Our evaluation and comparison results gained by running 12 lock protocols under the same benchmark in an identical runtime environment are also summarized in Section 4. Section 5 introduces an approach to dynamically balance benefit and overhead of lock management and demonstrates its feasibility with experimental results. In Section 6, we review the related work on XML concurrency control, before we conclude the paper in Section 7.

## 2. Tree-Based Lock Protocols

In the following, we will repeat neither the properties of tree-based lock protocols used in all industrial-strength DBMSs [13] nor our own work on fine-grained XML locking [18]. Instead, we refer to these well-known protocols and only sketch and emphasize those properties for better comprehension. Our goal is to convince the reader that the use of fine-grained protocols, whose lock modes and granules can be tailored to the (internal) data structures and operations of a native XDBMS, pays off and that those protocols can be implemented in a way which enables runtime optimization, especially adaptation to changing workloads.

### 2.1 B-Tree Locking

When XML documents are managed in a native way, they are often stored in some variation of B-tree structures – actually in XTC, the tree-connected nodes of an XML document are stored in a set of doubly-chained pages indexed by a B-tree such that the entire physical document structure results in a B*-tree [17]. Hence, the question immediately arises whether or not specific tree-based lock protocols can be used. 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[1] in the database world. 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 [2]. 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. With similar arguments, index locking can not appropriately cope with the navigational DOM operations [22].

### 2.2 Multi-Granularity Locking

Hierarchical lock protocols [12] – also denoted as multi-granularity locking (MGL) – are used "everywhere" in the relational world. For performance reasons in XDBMSs, fine-granular isolation at the node level is needed when accessing individual nodes or traversing a path, whereas coarser granularity is appropriate when traversing or scanning entire trees. Therefore, lock protocols, which enable the isolation of multiple granules each with a single lock, are also beneficial in XDBMSs. Regarding the tree structure of documents, objects can be isolated acquiring the usual *subtree 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

1. Unfortunately, this mechanism is denoted by the term "lock" in the literature on operating systems and programming environments.

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

**Figure 1. Lock compatibilities of taDOM2**

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]. Hence, we could map the relational IRIX protocol to XML trees and use it as a generic solution where the properties of the DOM access model are neglected.

Using the IRIX protocol, a transaction reading nodes at any tree level had to use R locks on the nodes accessed thereby locking these nodes together with their entire subtrees. This isolation is too strict, because the lock protocol unnecessarily prevents writers to access nodes somewhere in the subtrees[1]. Giving a solution for this problem, we want to sketch the idea of lock granularity adjustment to DOM-specific navigational operations.

## 2.3 Fine-Grained DOM-Based Locking

To develop true DOM-based XML lock protocols, we introduce a far richer set of locking concepts. While MGL essentially rests on intention locks and, in our terms, subtree locks, we additionally define locking concepts for nodes, edges, and levels. So-called *edge locks* having three modes [18] mainly serve for phantom protection, but play only a minor role for the discussion in this paper. In general, edges have to be protected, too, to guarantee isolation levels *repeatable read* and *serializability*. Assume, for example, transaction T has traversed the child set under parent *p*. To enable exactly the same navigation sequence for T later on, new children may not be inserted under this parent. A subtree lock on *p* would solve this problem, is, however, not a fine-grained solution. To prevent children to be inserted into or attached to the current child set by concurrent transactions, T may use edge locks in an appropriate mode and, thereby, preserve fine granularity of isolation, i.e., impede concurrent transaction processing as little as possible. Due to space restrictions and to retain acceptable readability of the paper, we will not further discuss them.

We differentiate read and write operations thereby renaming the well-known (IR, R) and (IX, X) lock modes with (IR, SR) and (IX, SX) modes, respectively. As in the MGL scheme, the U mode (SU in our protocol) plays a special role, because it permits lock conversion. Novel concepts are introduced by *node locks* and *level locks* whose lock modes are NR (node read) and LR (level read) in a tree which, in contrast to MGL, read-lock only a node or all nodes at a level, but not the corresponding subtrees. Together with the CX mode (child exclusive), these locks enable *serializable* transaction schedules with read operations on inner tree nodes, while concurrent updates may occur in their subtrees. While the remaining lock modes in Figure 1 coincide with those of the URIX protocol, we highlighted these three lock modes to illustrate that they provide a kind of tailor-made XML-specific extension. Hence, they behave as follows:

- An NR mode 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 mode 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.
- A CX mode 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 (compatibility is then decided on the child nodes themselves).

Figure 1 contains the compatibility matrix consisting of 8 lock modes for our basic lock protocol called taDOM2[2]. To illustrate its use, let us assume that the node manager has to handle for transaction $T_1$ an incoming request *GetChildNodes()* for context node *book* in Figure 2. This requires appropriate locks to isolate $T_1$ from modifications of other transactions. Here, the lock manager can use the level-read optimization and set the perfectly fitting mode LR on book and, in turn, protect the entire path from the document root by appropriate intention locks of mode IR. After having traversed all children, $T_1$ navigates to the content of the *price* element after the lock manager has set an NR lock for it. Then, transaction $T_2$ starts modifying the value *lname* and,

1. Obviously, a situation the other way around is not ***serializable***. A modification on an inner node while concurrent reads may occur in its subtree would necessarily affect readers, e.g., when traversing this inner node. A situation where readers have entered the subtree before the writer has locked the related root, would only occur under isolation level *consistent read*.

2. Although the compatibility of NR is identical to IR in taDOM2 and only differs in some protocol optimizations, we keep these lock modes separate to emphasize that there is a difference.

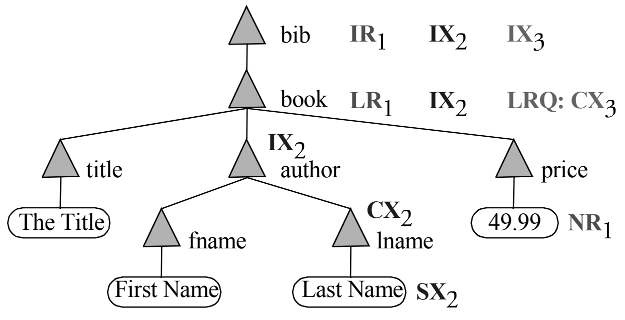


**Figure 2. Application of the taDOM2 protocol**

therefore, acquires an SX lock for the corresponding text node. The lock manager complements this action by acquiring a CX lock for the parent node and IX locks for all further ancestors. Simultaneously, transaction $T_3$ wants to delete the *author* node and its entire subtree, for which, on behalf of $T_3$, the lock manager 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 $T_1$. 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 $T_1$ on the *book* node.

A full-fledged lock protocol also needs provisions to convert a lock held by a transaction to a stronger mode, e.g., to update an object after having inspected it. As usual, these transitions are specified by a lock conversion matrix [18] on a node basis. In case, a transaction wants to upgrade a lock already granted in a specific mode (row header) to the target lock mode (column header), this matrix defines how the lock on the context node has to be converted and, as a consequence, all locks on its ancestor path have to be adjusted. Of course, if the lock upgrade is in conflict to locks of concurrent transactions, the requesting transaction has to be blocked (by waiting at the front position of the related LRQ). While most of the possible transitions in the lock conversion matrix are obvious, some complex transitions have to take place when LR locks are involved. For example, the upgrade of an implicitly read-locked child of a context node (holding an LR lock) to an exclusive lock (SX) is governed by a complex rule.

Hence, by tailoring the lock granularity to the LR operation, the lock protocol enhances transaction parallelism by allowing modifications of concurrent transactions in subtrees whose roots are read-locked. Furthermore, the lock state after successful conversion also enables the former reader $T_1$ to perform updates in all subtrees of *book* not blocked by concurrent writers (see Figure 2).

The lock modes introduced so far are the *core part* of the taDOM protocol family and are primarily responsible for its excellent performance behavior (see Figure 6). Nevertheless, experimental analysis of taDOM2 revealed some severe performance penalties in specific situations which were solved by the follow-up protocol taDOM2+. As described in [18], conversion of LR can be very cumbersome, because individual node locks have to be set on all children of the context node. As opposed to efficient ancestor determination (see Section 3.1) of a context node, identification of its children is very expensive, because access to the document is needed to explicitly locate all affected nodes. By introducing 4 additional intention modes, we can avoid access and explicit locking of child nodes in case of an LR conversion of the parent node. As a consequence, we obtained the more complex protocol taDOM2+ having *12 lock modes*. The DOM3 standard introduced a richer set of operations which led to several new tailored lock modes for taDOM3 and – again to optimize specific conversions – we added even more intention modes (again indicated by the +-suffix) resulting in the truly complex protocol taDOM3+ specifying compatibilities and conversion rules for *20 lock modes* (see [18] for details). Recently, a model-checking approach was used to show the correctness of the taDOM protocol family [27].

## 3. Lock Manager Implementation

So far, hardly anything was reported in the literature about the implementation of XML lock managers. Without having a reference solution, the XTC project had to develop such a component from scratch where the generic guidelines given in [13] were used. An initial version of the lock manager as described in [16] was based on static memory allocation of the lock buffers. Such a scheme cannot be dynamically adjusted to a widely varying number of active transactions and their lock request blocks. A reimplementation as sketched in Section 3.2 provided substantial improvements of the lock manager behavior. Some more lessons were learned the hard way, because XTC initially used a sequential node labeling scheme [17] revealing a catastrophic performance behavior.

### 3.1 The Role of Node Labeling

Why is the node labeling scheme so important for the flexibility and performance of a lock manager implementation? Many XML operations address nodes somewhere in subtrees of a document and these often require direct jumps “out of the blue” to a particular inner tree node. Efficient processing of all kinds of language models [6, 29] implies such label-guided jumps, because scan-based search should be avoided for direct node access and navigational node-oriented evaluation (e. g., *getElementById()* or *getNextSibling()*) as well as for set-oriented evaluation of declarative requests (e.g., via indexes). Because each operation on a context node (inner node) requires the appropriate isolation of its path to the root, the lock manager not only has to lock the node itself by a sufficient mode, but also has to identify all ancestor nodes to set the corresponding intention locks (or to check whether they are already granted). Therefore, the node labeling scheme used critically influences lock management overhead.

A comparison and evaluation of node labeling schemes in [15] recommends prefix-based node labeling based on the Dewey Decimal Classification [5]. As abstract properties of Dewey order encoding, each label represents the path from the document's root to the node and the local order w.r.t. the parent node; in addition, sparse numbering facilitates node insertions and deletions. Refining this idea, a number of similar labeling schemes were proposed differing in some aspects such as overflow technique for dynamically inserted nodes, attribute node labeling, or encoding mechanism. Examples of these schemes are ORDPATHs [23], DeweyIDs [15], or DLNs [3]. Because all of them are adequate and equivalent for our processing tasks, we prefer to use the substitutional name *stable path labeling identifiers* (SPLIDs) for them.[1]

Here we can only summarize the benefits of the SPLID concept; for details, see [15, 23]. It provides holistic system support which is important for lock management, too. Existing SPLIDs are immutable, that is, they allow the assignment of new IDs without the need to reorganize the IDs of nodes present. Comparison of two SPLIDs allows ordering of the related nodes in document order. As opposed to competing schemes, SPLIDs greatly support lock placement in trees, e.g., for intention locking, because they carry the node labels of all ancestors. Hence, access to the document is not needed to determine the path from a context node to the document root. Declarative queries and the required locks for them are also supported by the efficient evaluation – that is, computation without the need to access the document on disk – of the eight axes frequently occurring in XPath or XQuery path expressions: *parent/child, ancestor/descendant, following-sibling/preceding-sibling,* and *following/preceding.* Even sequential document processing and navigational operations to parent/child/sibling nodes from the context node are facilitated when suitable storage structures are available [17].

## 3.2 Lock Buffer Management

The lock manager is the key component of the locking functionality. It coordinates so-called lock services which are invoked by various system components when locks have to be acquired or released [1]. Furthermore, it serves as a factory to create lock services thereby facilitating the encapsulation of lock management internals from other system components. Besides the NodeLockService, the lock manager offers a number of different lock services (for DB buffer management, edge locks, index locks, etc.) which are equipped with interfaces tailored to the requirements of the respective component (see Figure 5).

Lock request scheduling is centralized by the lock manager. The actions for granting a lock by a lock service are considered in detail below. Otherwise, a lock service calls the *wait* method provided by the lock manager to suspend the requesting transaction until the request can be granted or a time-out occurs. Moreover, the detection and resolution of deadlocks is enabled by a global *wait-for* graph for which the transaction manager initiates the so-called transaction patrol thread in uniform intervals to search for cycles and, in case of a deadlock, to abort the involved transactions owning the fewest locks.

Each lock service has its own lock buffer containing a number of data structures as illustrated in Figure 3. To understand the general principles, it is sufficient to focus on the management of node locks. The main structures of the lock table are two pre-allocated buffers for lock header entries and lock request entries, each consisting of a configurable number (m) of blocks. This use of separate buffers serves for storage saving (differing entry sizes are used) and improved speed when searching for free buffer locations and is supported by tables containing the related free-placement information. To avoid frequent blocking situations when the lock table operations (look-up, insertion of entries) or house-keeping operations are performed, use of a single monitor is not adequate. Instead, latches are used on individual hash-table entries to protect against accesses by concurrent threads thereby guaranteeing the maximum parallelism possible.

For each locked object, a lock header encoded as a byte array is stored which contains name and current mode of the lock together with a pointer to the lock queue where all lock requests for the object are attached to. Such a lock request carries among other administration information the requested/granted lock mode together with the ID of the respective transaction. To speed-up lock release, the lock request entries are doubly chained and contain a separate pointer to the lock header, as illustrated in Figure 3. Furthermore, a transaction entry contains the anchor of a chain threading all lock request entries which minimizes lock release effort at transaction commit.

A lock request of object with SPLID = 1.19.7.5 for transaction $T_1$ proceeds as follows. A hash function delivers $h_T(T_1)$ in hash table T. If no entry is present for T1, a new transaction entry is created. Then, $h_L(1.19.7.5)$ selects (possibly via a synonym chain) a particular lock entry for object 1.19.7.5 in hash table L. If a lock entry is not found, a lock header is created for it and, in turn, a new lock request entry; furthermore, the various pointer chains are maintained for both entries. For checking the lock compatibility or the lock conversion, a registered locking scheme is used. In the same way, intention locks for the ancestors of 1.19.7.5 can be checked or newly created using $h_L(1.19.7)$, $h_L(1.19)$ and $h_L(1)$. Because of the frequency of this operation, we provide a single function which acquires a lock and all necessary intention locks.

## 3.3 Locking Efficiency and Effectiveness

What is a reasonable approach to quantify the quality of a lock protocol? It is certainly meaningless to measure the CPU cycles consumed by the lock manager when running

1. For example, such prefix-based labeling schemes are used in DB2 and MS SQL Server.

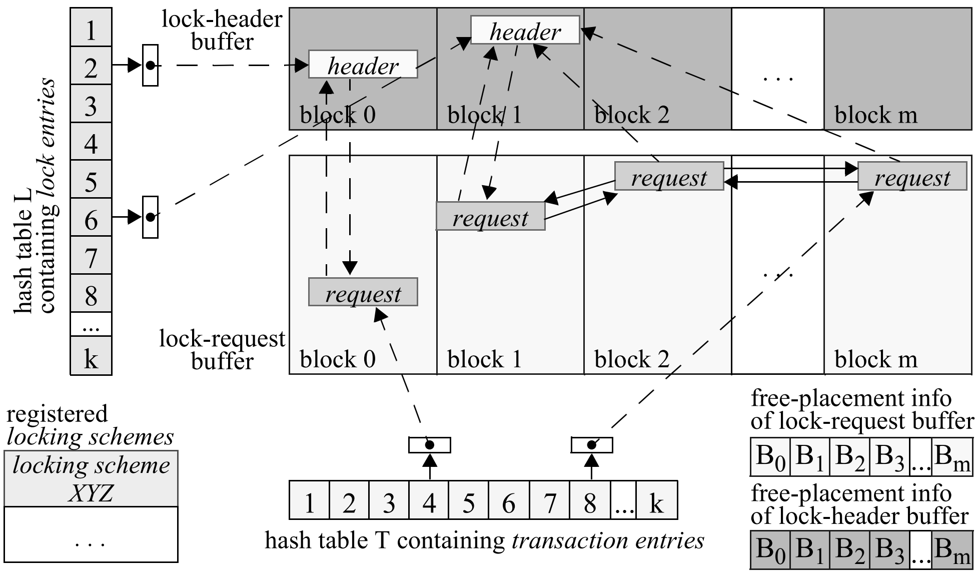


**Figure 3. Data structures of a lock buffer**

a specific protocol. Therefore, we must rather consider its *efficiency* in terms of transaction throughput obtained and compare it to its *effectiveness*, that is, how many locks have to be acquired and maintained to gain this transaction throughput. Of course, the ideal goal would be to minimize the number of (fine-grained) locks and, at the same time, to maximize throughput – obviously only achievable by tailoring the lock granules to the operation isolation needs in the best possible way.

To answer this question concerning our approach to tailor-made lock protocols, we designed a specific experiment to check efficiency and effectiveness of the taDOM protocol family. We created a library XML document (similar to the structure illustrated in Figure 2) with a size of 184 MB and over 4.5 million XML tree nodes. The transactions of the experiment were prepared to figure out the strengths of our lock protocols: A single transaction completely reconstructs a randomly selected book by invoking the *getChildNodes()* operation at each level requiring a lock for shared level access. Then, a randomly chosen 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. In our experiment, 25 threads were initialized where each was consecutively processing the transaction operations outlined above for 5 min. on the XTCserver.

The results for the successfully committed transactions and the maximum number of concurrently maintained locks are summarized in Figure 4. Hence, with the growing number of lock modes – and, therefore, with better adjusted lock granules (from taDOM2(+) to taDOM3(+)) –, the locking effectiveness is strongly improved. On the other hand, lock protocol efficiency – quantified in terms of successfully committed transactions – is increasing from taDOM2 to taDOM2+ and from taDOM3 to taDOM3+, because the substantial cost of child node accesses (needed in case of lock conversion for the determination of their node labels) can be avoided.

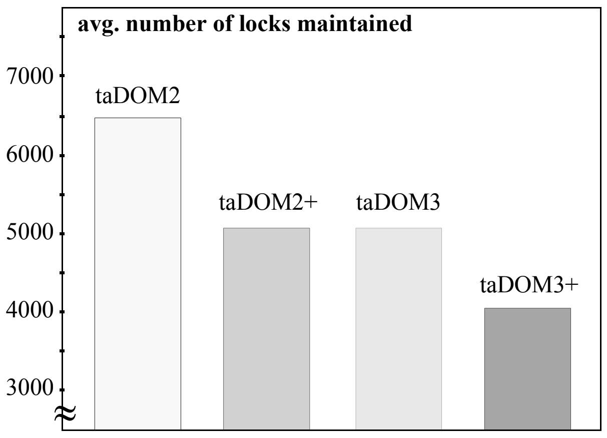
## 4. Lock Manager Adaptivity

In our initial implementation, the taDOM protocols were hard-wired and, thus, their exchange was cumbersome. Therefore, we looked for an elegant integration mechanism to transparently enable protocol changes (e.g., another compatibility matrix) or use of alternative protocols.

### 4.1 Use of a Protocol Family

As a first step, we decoupled the logic for navigating and manipulating the documents from all protocol-specific aspects by encapsulating them in so-called lock services, which are provided by the lock manager. The node manager uses them to enforce its isolation needs, instead of directly requesting specific locks from the lock manager [1]. For this purpose, the lock service interface offers a collection of methods, which sufficiently cover all relevant cases, e.g., for locking a single node or an entire subtree in either shared or exclusive mode. Figure 5 sketches the interaction of node manager and locking facilities involved in protocol use, lock mode selection, and enforcement of conversion rules.

This small restructuring reduced the responsibility of the node manager for coping with how the resources have to be locked to simply saying what resources have to be locked. Based on such "declarative" lock requests, the lock service infers the appropriate locks and lock modes according to the respective protocol and acquires them from the lock manager. The granted locks and the waiting lock requests are maintained in a conventional lock table and a wait-for graph as it is known from relational systems. For protocol-specific details like compatibility matrix and lock conversions, however, the lock manager depends on information provided by the lock service. Thus, the internal structures of the lock manager like the lock table and the deadlock detector could be completely decoupled from the used protocols, too. Finally, we are now



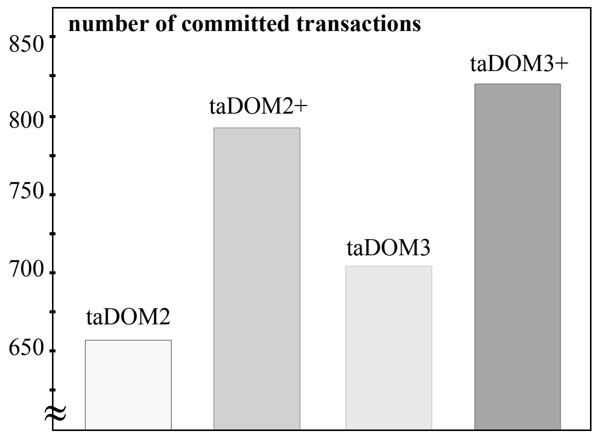


**Figure 4. Comparing effectiveness and efficiency of the taDOM lock protocols**

able to replace the lock protocol by simply exchanging the lock service used by the node manager. Moreover, it is now even possible to use several protocols, e.g., taDOM2+ and taDOM3+, simultaneously for different kinds of and workloads for documents inside the same server instance.

### 4.2 Meta-Locking

As described in the previous section, the key observation for transparent lock protocol exchange is an information exchange between lock manager and a lock service about the type of locks and compatibilities present. The lock services controlled by the lock manager can then be called by specific methods and each individual lock service can act as a kind of abstract data type. As a consequence, the node manager can plan and submit the lock requests in a more abstract form only addressing general tree properties. Using this mechanism, we could exchange all "closely related" protocols of the taDOM family and run them without additional effort in an identical setting. By observing their behavior under the same benchmark, we gained insight into their specific blocking behavior and lock administration overhead and, in turn, could react with some fine-tuning.

Even more important is a cross-comparison between different lock protocol families to identify strengths and weaknesses in a broader context. On the other hand, when unrelated lock protocols having a different focus can be smoothly exchanged, we would get a more powerful repertoire for concurrency control optimization under widely varying workloads.

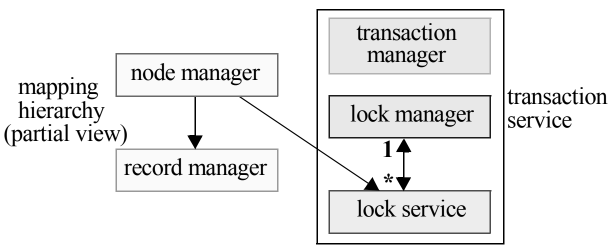
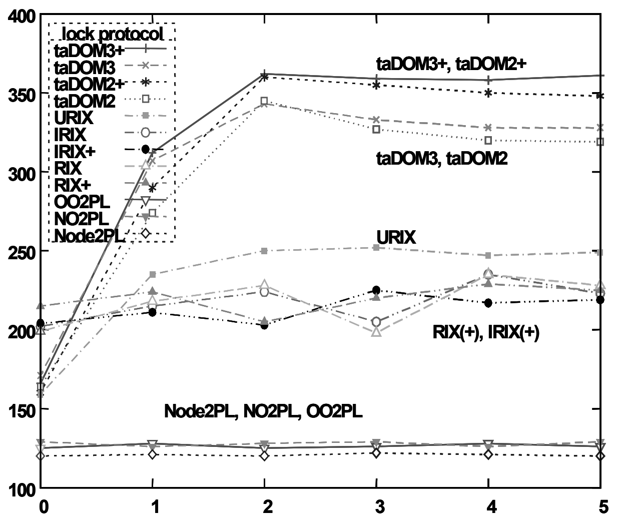


**Figure 5. Interaction of node manager and locking services**

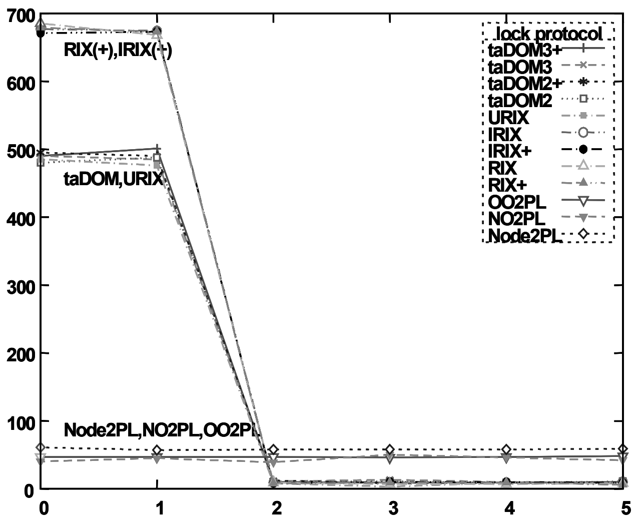
To demonstrate the usefulness of optimizing lock protocols, we implemented and explored a variety of fine-grained approaches to tree locking. We found quite different approaches to fine-grained tree locking in the literature and identified three families with 12 protocols in total: Besides our taDOM group with 4 members, we adjusted the relational MGL approach [12] to the XML locking requirements and included 5 variants of it (i.e., IRX, IRX+, IRIX, IRIX+, and URIX) in the so-called MGL group. Furthermore, we included three protocol variants described in [19], which were developed as DOM-based lock protocols in the Natix context. These protocols called Node2PL, NO2PL, and OO2PL are not implemented so far and are denoted as the *2PL group in the remainder of this paper. As will be shown, they are not competitive at all, but nicely reveal that a mismatch of lock granules required by the transactional operations and the isolation granules (here only single nodes) offered by the lock protocol may cause disastrous performance behavior in specific situations.

To run all of them in an identical system setting – by just exchanging the service responsible for driving the specific lock protocol – is more challenging than that of the taDOM family. The protocol abilities among the identified families differ to a much larger extent, because the MGL group does not know the concepts of node and level locks. The mismatch of the *2PL group with missing subtree locks and level locks is even larger.

For this reason, we developed the concept of meta-locking to bridge this gap and to automatically adjust the kinds of lock requests depending on the current service available. Important properties of a lock protocol influencing the kind of service request are the support of shared level locking, shared tree locking, and exclusive tree locking. To enable an inquiry of such properties by the node manager, the lock service provides three methods.



a) Number of committed transactions



b) Number of aborted transactions

**Figure 6. Overall results of a transaction benchmark (variation of lock depth)**

- *supportsSharedLevelLocking:* If a protocol supports the level concept, a request for all children or a scan traversing the child set can be isolated by a single level lock (i.e., LR). Otherwise, all nodes (and navigation edges) must be locked separately.
- *supportsSharedTreeLocking*: Analogously to level locks, subtrees can be read-locked by a single request, if the protocol has this option. Otherwise, all nodes (and navigation edges) of the subtree must be locked separately.
- *supportsExclusiveTreeLocking*: This protocol property enables exclusive locking of a subtree by setting a lock on its root node. If this option is not available, then subtree deletion requires traversal and separate locking of all nodes, before the deletion can take place in a second step.

For a lock request on a context node, the node manager can select a specification of the lock mode (*Read*, *Update*, or *Exclusive*) for the context node itself, the context node and the level of all its children or the context node and its related subtree. For navigational accesses, a lock mode for one of the edges *prevSibling*, *nextSibling*, *firstChild*, or *lastChild* can be specified, in addition. Then, the lock manager translates the lock request to a specific lock mode dependent on the chosen protocol.

## 4.3 Results of a Lock Contest

Although the MGL group is only distantly related to our protocol family, this meta-locking concept enabled without further “manual” interaction a true and precise cross-comparison of all 12 protocols, because they were run under the same benchmark in XTC using the same system configuration parameters. All benchmark operations and the node-manager-induced lock protocols were applied to the taDOM storage model [17] of XTC and took advantage of its refined node structure and the salient SPLID properties concerning lock management support.

As it turned out by empirical experiments, *lock depth* is an important and performance-critical parameter of an XML lock protocol. Lock depth *n* specifies that individual locks isolating a navigating transaction are only acquired for nodes down to level *n*. Operations accessing nodes at deeper levels are isolated by subtree locks at level *n*. Note, choosing lock depth *0* corresponds to the case where only document locks are available. In the average, the higher the lock depth parameter is chosen, the finer are the lock granules, but the higher is the lock administration overhead, because the number of locks to be managed increases. On the other hand, lock conflicts typically occur at levels closer to the document root (lower lock depth) such that fine-grained locks (and their increased management) at levels deeper in the tree do not pay off. Obviously, the taDOM and the MGL protocols can easily be adjusted to the lock-depth parameter, whereas the *2PL group cannot benefit from it.

In our lock protocol competition, we used a document of about 580,000 tree nodes (~8MB) and executed a constant system load of 66 transactions taken from a mix of 5 transaction types. For our discussion, neither the underlying XML documents nor the mix of benchmark operations are important. Here, we only want to show the overall results in terms of successfully executed transactions (throughput) and, as a complementary measure, the number of transactions to be aborted due to deadlocks.

Figure 6a clearly indicates the value of tailor-made lock protocols. With the missing support for subtree and level locks, protocols of the *2PL group needed a ponderous conversion delivering badly adjusted granules. Hence, even the MGL protocols – a kind of standard in the relational world – roughly doubled the transaction throughput as compared to the *2PL group. Enforced by the missing concepts of node and level locks, in various situations the MGL group could only react to lock requests in a suboptimal way, because their lock granules were badly adjusted to the needs of DOM operations. As a consequence, they only reached about half of the throughput achieved by the taDOM group.

Because of their exclusive use of node locks, *2PL protocols cannot take advantage of the lock depth parameter (as illustrated by Figure 6). A reasonable application to enable fine-grained locking, however, requires at least a lock depth of *2*; otherwise, the more sophisticated protocols cannot leverage their strengths. A certain lock depth and, dependent on it, reduced lock granules (smaller subtrees locked) are also important for deadlock avoidance. Hence, careful selection of lock granules and lock modes is particularly critical at lower lock depths, i.e., at levels close to the document root. Such a coarse-grained locking should be avoided at all, as confirmed by the taDOM and MGL protocols when lock depths < 2 were used (see Figure 6b).

In summary, the impressive performance behavior of the taDOM group reveals that a careful adaptation of lock granules and available lock modes to specific operations clearly pays off (see again the discussion in Section 2.3).

## 5. Runtime Protocol Adjustment

The value chosen for the lock depth parameter is critical to the performance of the system. Potential concurrency is limited, when lock depth is set too low, and system resources are wasted for lock management if it is chosen too high. Unfortunately, it is generally not possible to predict the optimal value for the lock depth parameter because it heavily depends on the documents' characteristics and the current transaction mix, and thus may change during a processing period. Hence, we need an effective mechanism that enables us to preserve a reasonable balance of concurrency achieved and locking overhead needed.

### 5.1 Local Reduction of Lock Depth

The most effective and widely used solution to reduce lock management overhead is *lock escalation* the XTC implementation of which we discussed in more detail in a short forerunner version [1]: The fine-grained resolution of a lock protocol is – preferably in a step-wise manner – reduced by acquiring coarser lock granules. Applied to our case, we have to reduce the lock depth and lock subtrees at higher levels using single subtree locks instead of separately locking each descendant node visited. A general reduction of the lock depth, however, would jeopardize the benefits of our tailored lock protocols. Therefore, we aim at running transactions initially at a higher lock depth to benefit from the fine-grained resolution of our lock protocols in hot-spot regions, and reserve the option to dynamically reduce the lock depth in low-traffic regions encountered to save system resources.

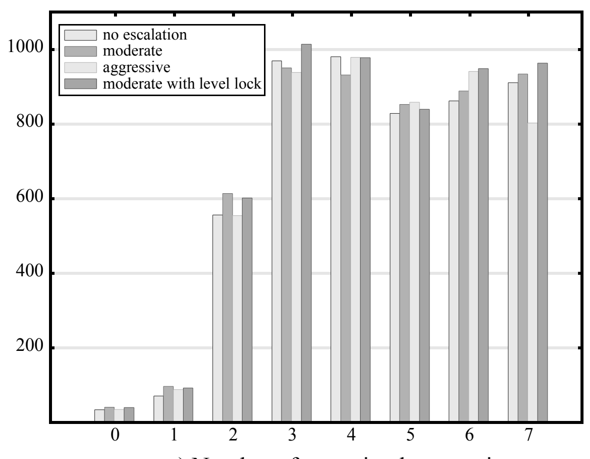
In a first step, we made our lock buffer implementation "tree-aware". Lock requests for specific document nodes trigger the lock buffer to transparently acquire the required intention locks on the ancestor path, which have to be provided by the lock service. Thus, the lock buffer knows not only the current lock mode of a node, the number of requests and the transactions that issued these requests, but also its level in the document and the level of the target node when it requests the intention locks on the path from root to leaf. We exploit this cheaply gathered information to decide about subtree-local lock escalations when the lock buffer acquires the intention locks. Depending on the fan-out characteristics of a document, we can now define a suitable escalation threshold for each level in the document. When the number of requests for a specific node reaches this threshold, the initial lock request for a deeper node is replaced by an appropriate subtree lock on the current node to save system resources. To avoid blocking situations, we check whether or not concurrent transactions already hold incompatible locks on this node.

### 5.2 Experimental Evaluation

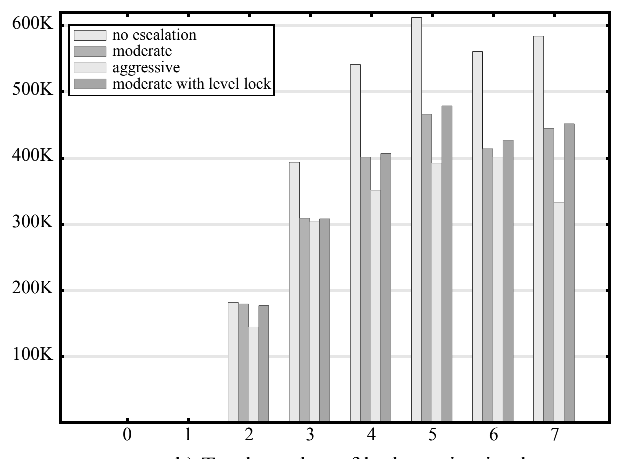
For a first experimental evaluation in our XDBMS prototype XTC, we started with three simple escalation heuristics for a mix of eight transaction types, which accessed and modified a generated XMark [28] document at varying levels and in different granules, e.g., by placing bids on items, changing user data, adding items, or reading and writing mails. To increase the conflict rate, we chose again an initial document size of only 8 MB. In our measurements, we focused on further lock-depth optimization of the taDOM3+ lock protocol, because it outperforms all other protocols (see Section 3.3), and varied the initial lock depth from 0 to 7. The first two escalation heuristics use only subtree locks to escalate a lock request, and compute the level thresholds according to $threshold = k(1920/2^{level})$ with $k$ equal to 1.0 (called moderate) respectively 0.7 (called aggressive). The third one uses the same thresholds as the first, but employs the less restrictive LR lock mode to escalate a NR lock request for a child node.

The results in Figure 7a reveal that our dynamic escalation mechanism does not have a negative effect on transaction throughput. Furthermore, in some cases the reduced lock overhead even increases throughput. The highest transaction rates were achieved at lock depth 3 and 4, which obviously fitted best to our test workload. At lock depth 5, we observed a small break, because the additional locks did not contribute to higher concurrency. At higher lock depths, however, throughput increased again.

Figure 7b demonstrates how effective simple escalation heuristics could reduce overhead of lock management. As expected, the *aggressive heuristics* saved the most locks, while the two *moderate heuristics* are head to head, but still clearly in advance to the solution without any lock escalation. Each heuristics saved the setting of at least 100,000 locks for all relevant lock depths.



a) Number of committed transactions



b) Total number of locks maintained

**Figure 7. Comparing effectiveness and efficiency of the taDOM3+ lock protocol depending on lock depth**

The number of concurrency-induced aborts and the perceived response times are important aspects for client applications. Here, Figure 8a clearly indicates that the escalation heuristics did not lead to a higher deadlock rate. Moreover, the fairly constant abort rates at lock depths higher than 2 were partially caused by page-level deadlocks (caused by fix and unfix operations on buffer pages) and not by the lock protocol. The average response times of successful transactions have their minimum at lock depth 2, but suffer from a high abort rate (see Figure 8b). The response times for higher lock depths, however, are only minimally higher because of the higher lock overhead.

Altogether, the experimental results demonstrate well that our approach is able to provide a constantly high transaction throughput even for higher lock depths, and that it can be easily adjusted to the current workload to save system resources. On the one hand, lock depth 3 marked the sweet spot in our test scenario with the highest transaction rates and the fewest lock requests. On the other hand, the good results of the heuristics for higher lock depths revealed that subtree-local lock escalations are a practical way to balance lock overhead and potential concurrency. The aggressive heuristics provided the biggest savings in combination with a good overall transaction throughput. The throughput of the *moderate heuristics with level locks* achieved in general a better throughput, but needed more locks. The comparison with the *moderate heuristics* also confirmed that exploitation of the special level lock mode of the taDOM protocols increases throughput at the cost of slightly more locks.

## 6. Related Work

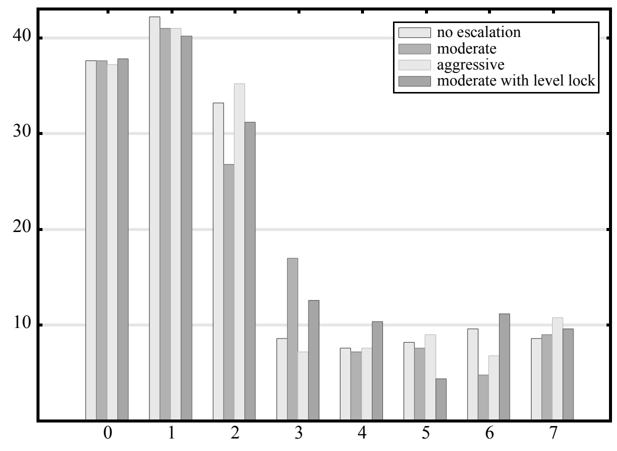
To the best of our knowledge, we are not aware of contributions in the open literature dealing with XML locking in the detail and completeness presented here. So far, most publications just sketch ideas of specific problem aspects and are less compelling and of limited expressiveness, because they are not implemented and, hence, cannot provide empirical performance results.

As our taDOM protocols, four lock protocols developed in the Natix context [19] focus on DOM operations and acquire appropriate locks for document nodes to be visited. In contrast to our approach, however, they lack support for direct jumps to inner document nodes as well as effective escalation mechanisms for large documents. Furthermore, only a few high-level simulation results are reported, which indicate that they are not competitive to the taDOM throughput performance (see Figure 6).

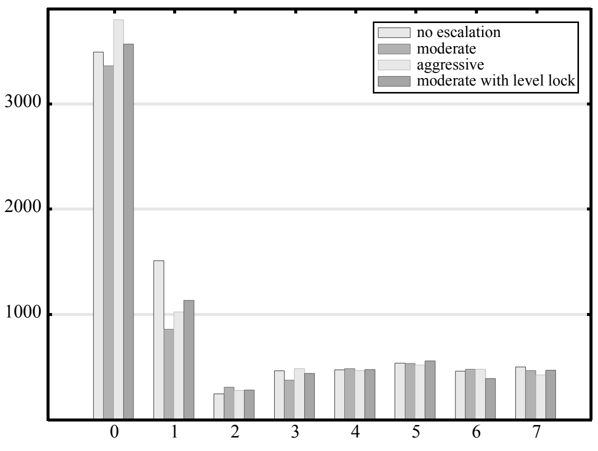
The XLP/DLP protocol [21] locks all visited document nodes, too, but was primarily designed for the evaluation of XPath expressions. It is not a strict two-phased lock protocol and allows to unlock nodes before commit, if they are not necessary to keep the set of target nodes stable. Although XLP depends on the navigation semantics of XPath, it is not able to prevent phantoms, e.g., when the child axis is evaluated. Further, the protocol does not scale, because it does not support any kind of lock escalation.

DGLOCK [10], proposed by Grabs et al., is a lock protocol for a subset of XPath that locks the nodes of a structural summary of the document instead of the document nodes themselves. Although this enables DGLOCK to cover large parts of a document with relatively few locks as compared to approaches that lock single document nodes, the protocol can cause severe performance penalties in general. Its “semantic locks” have to be deduced by analyzing the path expressions, which made it necessary to annotate them with additional content predicates to achieve satisfying concurrency. Hence, even if only simple predicates are used, a compatibility check of a lock request may require physical access to all document nodes affected by a lock respectively by its predicate. Furthermore, the protocol does not support the important descendant axis.

XDGL [24] works in a similar way, but provides higher concurrency due to a richer set of lock modes, and introduces *logical locks* to support also the descendant



a) Number of aborted transactions



b) Average response time of committed transactions

**Figure 8. Lock-strategy-induced effects on transaction performance – as a function of lock depth**

axis. The general problem of locks with annotated predicates, however, remains unsolved. SXDGL is an enhancement of XDGL that has been implemented in the Sedna system [9]. It uses additional lock modes to capture also the semantics of XQuery/XUpdate and employs a multi-version mechanism, which allows read-only transactions to get a snapshot-consistent view of the document without requesting any locks.

OptiX and SnaX [25] are two akin approaches, which make also extensive use of a multi-version architecture. OptiX is the only optimistic concurrency control approach adapted to the characteristics of XML so far. In the verification phase, it uses special algorithms that respect the hierarchical structure of the documents. SnaX is a variant of OptiX that relaxes serializability, and guarantees readers only snapshot consistency.

Finally, so-called path locks were presented in [4] as one of the first proposals for XML concurrency control at all, but are limited to a very small subset of XPath.

## 7. Conclusion

In this paper, we proposed the use of techniques adaptable to various application scenarios and configurations supporting high concurrency in native XDBMS. We started with an introduction into the basics of our tailor-made lock protocols, which are perfectly eligible for fine-grained transaction isolation on XML document trees. Prime concepts responsible for concurrency enhancements and a performance boost for transaction processing in XML trees were novel lock modes for individual nodes and child sets under a parent node (so-called level locks). We showed how they can be encapsulated and integrated into an XDBMS environment. By introducing the concept of meta-locking, we discussed the principles for the exchange of a specific lock protocol. Furthermore, we demonstrated how we could extend our approach initially designed for taDOM protocols to also support other locking approaches in our prototype XTC to cross-compare foreign protocols and to prove the superiority of our protocols with empirical tests in an identical system configuration.

Finally, we presented an effective mechanism, which allows us to easily control and optimize the runtime behavior of our lock protocols without making concessions to the encapsulation and exchangeability properties. Our future work will focus on improvements concerning the efficient evaluation of declarative queries based on the XQuery language model as well as the self-optimizing capabilities of our XDBMS prototype.

## References

[1] S. Bächle and T. Härder: Tailor-made Lock Protocols and their DBMS Integration. Proc. EDBT Workshop on Software Engineering for Tailor-made Data Management (SETMDM), Nantes, March 2008.

[2] R. Bayer and M. Schkolnick: Concurrency of Operations on B-Trees. In *Acta Informatica* 9:1–21 (1977)

[3] T. Böhme and E. Rahm: Supporting Efficient Streaming and Insertion of XML Data in RDBMS. Proc. 3rd Int. Workshop Data Integration over the Web, Riga, Latvia, (2004) 70-81.

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

[5] M. Dewey: *Dewey Decimal Classification System.* http://www.mtsu.edu/~vvesper/dewey.html

[6] Document Object Model (DOM) Level 2 / Level 3 Core Specific., W3C Recommendation

[7] K. P. Eswaran, J. Gray, R. A. Lorie, and I. L. Traiger: The Notions of Consistency and Predicate Locks in a Database System. *Commun. ACM* 19(11): 624-633 (1976).

[8] T. Fiebig, S. Helmer, C.-C. Kanne, G. Moerkotte, J. Neumann, R. Schiele, and T. Westmann: Anatomy of a native XML base management system. In *The VLDB Journal* 11(4): 292-314 (2002)

[9] A. Fomichev, M. Grinev, and S. D. Kuznetsov: Sedna: A Native XML DBMS. Proc. SOFSEM 2006: 272-281

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

[11] G. Graefe: Hierarchical locking in B-tree indexes. Proc. National German Database Conference (BTW), LNI P-65, Springer, pp. 18–42 (2007)

[12] J. Gray: Notes on Database Operating Systems. In Operating Systems: An Advanced Course, Springer-Verlag, LNCS 60: 393-481 (1978).

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

[14] T. Härder and A. Reuter: Principles of Transaction-Oriented Database Recovery. In *ACM Computing Surveys* 15(4):287-317 (1983)

[15] T. Härder, M. P. Haustein, C. Mathis, and M. Wagner: Node Labeling Schemes for Dynamic XML Documents Reconsidered. In *Data & Knowledge Engineering* 60:1, pp. 126-149 (2007)

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

[17] M. P. Haustein and T. Härder: An Efficient Infrastructure for Native Transactional XML Processing. In *Data & Knowledge Engineering* 61:3, pp. 500–523 (2007)

[18] M. P. Haustein and T. Härder: Optimizing lock protocols for native XML processing. In *Data & Knowledge Engineering* 65:1, pp. 147-173 (2008)

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

[20] H. V. Jagadish, S. Al-Khalifa, and A. Chapman: TIMBER: A native XML database. In *The VLDB Journal* 11(4): 274-291 (2002)

[21] K.-F. Jea and S.-Y. Chen: A High Concurrency XPath-based Locking Protocol for XML Databases. Information & Software Technology 48:8, 708-716 (2006).

[22] C. Mohan: ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes. Proc. VLDB: 392-405 (1990)

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

[24] P. Pleshachkov, P. Chardin, and S. Kusnetzov: XDGL: XPath-based Concurrency Control Protocol for XML Data. Proc. 22nd Britisch National Conf. on Databases (BNCOD), UK Bd. 3567, Springer, pp. 145-154 (2005)

[25] Z. Sardar and B. Kemme: Don't be a Pessimist: Use Snapshot based Concurrency Control for XML. Proc. 22nd Int. Conf. on Data Engineering (ICDE), Atlanta, GA, USA, 130 (2006)

[26] H. Schöning. Tamino – A DBMS designed for XML. Proc. 7th Int. Conf. on Data Engineering, Heidelberg, Germany, 149-154 (2001)

[27] A. Siirtola and M. Valenta: Verifying Parameterized taDOM+ Lock Managers. Proc. SOFSEM 2008, Springer-Verlag, LNCS 4910: 460-472 (2008)

[28] XMark - An XML Benchmark Project. http://monetdb.cwi.nl/xml/

[29] XQuery 1.0: An XML Query Language. http://www.w3.org/XML/XQuery

[30] XQuery Update Facility. http://www.w3.org/TR/xqupdate