Implementing and Optimizing Fine-Granular Lock Management for XML Document Trees

Sebastian Bächle[1], Theo Härder[1], and Michael P. Haustein[2]

[1] Department of Computer Science,
University of Kaiserslautern, Germany
{baechle, haerder}@informatik.uni-kl.de
[2] SAP AG
Hopp-Allee 16, 69190 Walldorf, Germany
haustein@h-ms.de

Abstract. Fine-grained lock protocols with lock modes and lock granules adjusted to the various XML processing models, allow for highly concurrent transaction processing on XML trees, but require locking facilities that efficiently support large and deep hierarchies with varying fan-out characteristics. We discuss these and also further requirements like prefix-based node labels, and present a lock management design that fulfills all these requirements and allows us to perfectly exploit the advantages of our tailor-made lock protocols for XML trees. Our design also supports the flexible use of heuristics for dynamic lock escalation to enhance workload adaptivity. Benchmark runs convincingly illustrate flexibility and performance benefits of this approach and reveal that careful lock protocol optimization pays off.

1 Motivation

Native XML database systems (XDBMS) promise tailored processing of XML documents, but most of the systems published in the DB literature are designed for efficient document retrieval [13, 19]. This "retrieval-only" focus was probably caused by the first proposals of XQuery respectively XPath where the update part was left out [21]. With the advent of the update extension [22] and its support in commercial systems, however, the requirement for concurrent and transaction-safe document modifications became widely accepted.

Currently, all vendors of XML(-enabled) database management systems support updates only at document granularity. Although this approach reduces some of the complexity of updating XML, it is only feasible if the database consists of collections of small documents. Efficient and effective transaction-protected collaboration on XML documents, however, becomes a pressing issue for medium- to large-sized documents, especially, in the face of a growing number of use cases for XML as the central data representation format in business environments. Therefore, our goal is to provide a suitable solution for highly concurrent transaction processing on XML documents that fulfills all requirements of an XDBMS in terms of applicability and flexibility.

In recent years, a great variety of XML storage models, indexing approaches and sophisticated query processing algorithms have been proposed. Unfortunately, they all make different assumptions about the availability of specific data access operators, or even do not touch the problem of an embedding in a real system environment at all. This diversity in conjunction with the various language models and programming interfaces for XML is the most crucial problem in XML concurrency control. All those different ways of accessing the same pieces of data makes it impossible to guarantee that only serializable schedules of operations occur. Hence, we strive for a general solution that is even applicable for a whole spectrum of XML language models (e.g., XPath, XQuery, SAX, or DOM) in a multi-lingual XDBMS environment.

Because of the superiority of locking in other areas, we also focus on lock protocols for XML. We have already developed a family consisting of four DOM-based lock protocols called the taDOM group by adjusting the idea of multi-granularity locking [8] to the specific needs of XML trees. Here, we discuss mechanisms how such protocols can be efficiently implemented in a native XDBMS or any other system that requires concurrent access to XML documents.

In Sect. 2, we emphasize the need and advantage of lock protocols tailored to the specific characteristics of XML processing, sketching the properties of our taDOM protocols, and discuss the role of prefix-based node labeling for efficient lock management. Sect. 3 gives an overview of related work. In Sect. 4, we demonstrate how to embed XML lock protocols in a system environment and describe some implementation details of a lock manager that simplifies the use of a hierarchical lock protocol for large and deep document trees. In Sect. 5, we introduce an approach to dynamically balance benefit and lock overhead, outline the problem of conversion deadlocks, and demonstrate the feasibility of our approach with experimental results in Sect. 6. Finally, Sect. 7 concludes the paper and gives an outlook on future work.

2 Preliminaries

XML lock protocols aim to enable XDBMS concurrent read and write accesses of different transactions to the same documents, and thus increasing the overall performance of the system. Hence, regarding the tree structure of XML, the protocols have to synchronize read and write operations inside of a tree structure at different levels and in different granularities.

Hierarchical lock protocols [9]—also denoted as multi-granularity locking—were designed for hierarchies of objects like tables and tuples and are used "everywhere" in the relational world. The allow for fine-grained access by setting R (read) or X (exclusive) locks on objects at the lower levels in the hierarchy and coarse grained access by setting the locks at higher levels in the hierarchy, implicitly locking the whole subtree of objects at smaller granules. 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 [9].

Although the MGL protocol can also be applied to XML document trees, it is in most cases too strict, because both R and X mode on a node, would always lock the whole subtree below, too. While this is the desired semantics for part-of object hierarchies as in relational databases, these restrictions do not apply to XML where transactions must not necessarily be guaranteed to have no writers in the subtree of their current node. Hence, MGL does not provide the degrees of concurrency, that could be achieved on XML documents.

In the following, we will give a brief introduction into our TaDOM lock protocols, which refine the ideas of the MGL approach and provide tailored lock modes for high concurrency in XML trees.

2.1 TaDOM Protocol Family

To develop true DOM-based XML lock protocols, we introduced a far richer set of locking concepts, beyond simple intention locks and, in our terms, subtree locks. 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. We introduced new lock modes for single nodes called NR (node read) and NX (node exclusive), and levels of siblings called LR (level read). As in the MGL scheme, the U mode (SU in our protocol) plays a special role, because it permits lock conversion. The novelty of the NR and LR modes is that they allow, in contrast to MGL, to read-lock only a node or all nodes at a level (under the same parent), but not the corresponding subtrees.

The LR mode required further a new intention mode CX (child exclusive). It indicates the existence of an SX or NX 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 a context node *c*, because separate child nodes of *c* may be exclusively locked by other transactions (compatibility is then decided on the child nodes themselves). Alltogether these new lock modes enable serializable transaction schedules with read operations on inner tree nodes, while concurrent updates may occur in their subtrees.

For phantom protection, edge locks are used as a secondary type of locks. They have only three different modes (read, update, and exclusive) and are requested for the so-called virtual navigation edges of elements (previous/next sibling, first/last child) and text nodes (previous/next sibling). Transactions have to request shared edge locks when they navigate "over" such an edge to another node, and exclusive edge locks when they want to insert new child/sibling nodes in the tree. This mechanism signals readers node deletions of uncommitted transactions and hinders writers from inserting nodes at positions where they would appear as phantoms for others.[3]

[3] Although edge locks are an integral part of taDOM, they are not in the focus of this work and will not be further regarded due to space restrictions.

Continuous improvement of this basic concepts lead to a whole family of lock protocols, the taDOM family, and finally ended in a protocol called taDOM3+, which consists of 20 different lock modes and allows highest degrees of parallelism on XML document trees. Correctness and, especially, serializability of the taDOM protocol family was shown in [11, 20].

To illustrate the general principles of these protocols, let us assume that transaction *T1* navigates from the context node *journal* in Fig. 1 to the first child *title* and proceeds to the *editoral* sibling. This requires *T1* to request NR locks for all visited nodes and IR locks for all nodes on the ancestor path from root to leaf. Then, *T1* navigates to the first *article* to read all child nodes and locks the *article* node and all children at once with the perfectly fitting mode LR. Then, transaction *T2* deletes a *section* from the *editorial*, and acquires an SX lock for the corresponding node, CX for the *editorial* parent and IX locks for all further ancestors. Simultaneously, transaction *T3* is able to update the *firstname* node, because the LR lock of *T1* is compatible with the required IX intention modes.

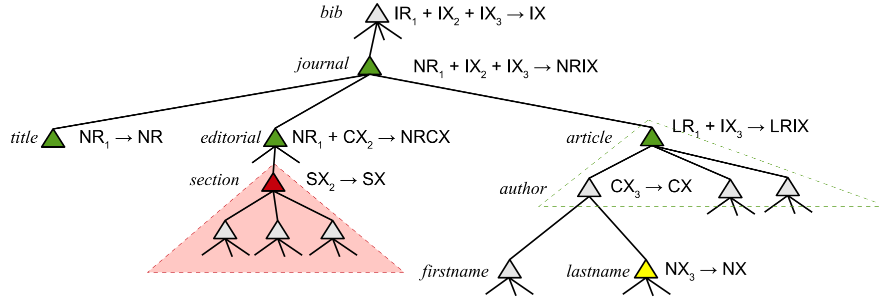


Fig. 1. Example of the taDOM3+ protocol

2.2 Node Labeling

XML operations often 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, 21] 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 requires the appropriate isolation of its path to the root, not only the node itself has to be locked in a sufficient mode, but also the corresponding intention locks on all ancestor nodes have to be acquired. Therefore, the lock manager often has to procure the labels for nodes

and their contexts (e.g., ancestor paths) requested. No matter what labeling scheme is used, document access cannot always be avoided (e.g., getNextSibling()). If label detection or identification, however, mostly need access to the document (usually stored on disk), a dramatic cost factor may burden concurrency control. Therefore, the node labeling scheme used may critically influence lock management overhead [10].

In the literature, range-based and prefix-based node labeling [4] are considered the prime competitive methods for implementation in XDBMSs. A comparison and evaluation of those schemes in [10] recommends prefix-based node labeling based on the Dewey Decimal Classification [5]. As a property of Dewey order encoding, each label represents the path from the document's root to the related 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 [14], DeweyIDs [10], or DLNs [2]. 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.

Here, we can only summarize the benefits of the SPLID concept; for details, see [10, 14]. Existing SPLIDs are immutable, that is, they allow the assignment of new IDs without the need to reorganize the IDs of nodes present – an important property for stable lock management. 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. Furthermore, comparison of two SPLIDs allows ordering of the related nodes in document order and computation of all XPath axes without accessing the document, i.e., this concept provides holistic query evaluation support which is important for lock management, too.

3 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 [12] 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.

DGLOCK [7], proposed by Grabs et al., is a coarse-grained lock protocol for a subset of XPath that locks the nodes of a structural summary of the document instead of the document nodes themselves. These "semantic locks" allow to cover large parts of a document with relatively few locks, but require annotated 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 [15] 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 axis. The general problem of locks with annotated predicates, however, remains unsolved. SXDGL [16] is snapshot-based enhancement of XDGL that uses additional lock modes to capture also the semantics of XQuery/XUpdate. It employs a multi-version mechanism for read-only transactions to deliver a snapshot-consistent view of the document without requesting any locks.

OptiX and SnaX [17] 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. SnaX is a variant of OptiX that relaxes serializability and only guarantees snapshot consistency for readers.

4 Infrastructure

As shown in Sect. 2 the taDOM protocols are applied only on the level of logical operations on prefix-labeled trees. In a real system environment, however, different data access methods might be used for performance reasons or because the underlying physical representation of a document can only support a subset of these operations.

The rationale of our approach is to map all types of data access requests—at least logically—to one or a series of operations of a DOM-like access model. Hence, our approach can be used in many different system environments, as long as they natively support or at least additionally provide a prefix-based node addressing. Note again, that this is a crucial requirement for XML lock protocols, but anyway also worthwhile for many processing algorithms and indexing methods. For the mapped operation primitives, we can apply the lock protocol independently of the actual access model or physical data representation. So, we can profit from both, the performance of fast, native data access operations as well as the concurrency benefits gained from the use of the fine-grained taDOM protocols.

Database management systems typically follow the principle of a layered architecture to encapsulate the logic for disk and buffer management, physical and logical data representation, query evaluation and so on. Except from cross-cutting system services like the transaction management, monitoring or error handling, it is relatively easy to exchange such a layer as long as it provides the same functionality to the next higher system layer. In our approach, we use this

property to introduce a new layer in this hierarchy, which realizes the mapping of data access operations to logical taDOM operations and back to physical data access operations. Fig. 2 shows the two core parts of our design—a stack of three thin layers, placed between the physical representation and the higher engine services, and the lock manager implementation itself.

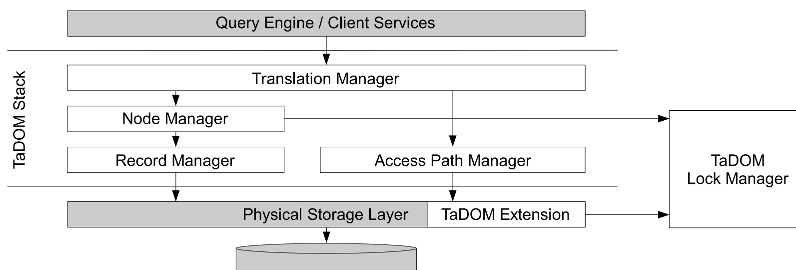


Fig. 2. Embedding of the lock protocols in a system infrastructure.

4.1 Data Access Layer

The taDOM stack provides the query engine or the client transparently with a transaction consistent, "single-user view" on the data. It is located on top of the physical access layer, which provides access primitives for the underlying storage structures. Hence, the stack replaces the classic, monolithic logical access layer, responsible for the translation of high-level data access requests to the corresponding data access primitives.

At the top is the so-called *translation manager*. It maps all different kinds of access requests to the one or a series of calls of the node manager, which provides a set of operations which can be easily covered by the lock modes of the taDOM protocols. These are the navigation and modification operations as they are known from DOM, efficient bulk operations for subtrees like scans, insertions, and deletions, and, of course, direct jumps to nodes via their SPLID. The latter is certainly the biggest strength of the whole protocol as we will see later. For our explanations, we consider first the most simple case, where the translation manager only has to provide node-wise access and manipulation operations, because they directly correspond to DOM operations.

The node manager is responsible for the acquisition of the appropriate node and edge locks according to the taDOM protocols. When the required locks for an operation are granted, it simply passes the request directly to the *record manager*, which provides the same set of operations as the node manager.

The record manager translates the operations finally into the actual physical data access operations of the physical storage layer. Depending on the chosen storage model, this can be, for example, a simple access to a main memory

representation of the document tree, or a B*-tree access. Independent of the chosen storage model, however, the physical storage layer only has to preserve *structural consistency*, e.g., if multiple threads representing different transactions access and modify the shared data structures concurrently. The *transactional consistency* will always be preserved, because the taDOM protocol applied at the node manager level already ensures that the operations will create a serializable schedule.

Depending on the needs of the higher system layers and the capabilities of the underlying storage structures, it may be desirable or necessary to use efficient indexes, e.g., to fetch attribute nodes by their value, or to fetch all element nodes of a certain name. Of course, these are complex operations, which can not be easily mapped to the navigational base model. From the point of view of the taDOM model, however, this can be also considered as "random" jumps to specific document nodes, which, in turn, are supported very well through the SPLID-labeling. Hence, with a small extension of the physical storage layer, our approach can also support efficient index accesses. The diverse index structures only have to lock every node with a node lock, before they return it as a result. In contrast to the previous case, however, index structures have to take additional care of phantoms themselves.[4]

4.2 Lock Manager

Although taDOM is based on the idea of widely used concepts of multi-granularity locking, hierarchical locking on XML trees is fairly different from the common way. While multi-granularity locking in relational databases usually requires only four or less granules of locks, e.g., for single tuples and whole tables, XML trees have a varying and dynamic depth and also varying and much smaller fanouts than a table with millions of tuples.

So far, hardly anything was reported in the literature about the implementation of XML lock managers. Without having a reference solution, we had to develop such a component from scratch where the generic guidelines given in [9] were used.

The central part of the lock manager is the lock table. It coordinates all required data structures and is also responsible for granting lock requests. For each locked object, a lock *header* is created, which contains name and current mode of the locked object together with a pointer to the lock *queue* where all lock requests for the object are attached to. Each lock request carries the ID of the respective transaction, the requested/granted mode. All lock requests of a transaction are doubly chained to speed-up lock release at transaction end.

Further necessary data structures are transaction entries to store housekeeping information for each transaction, as well as two hash tables h_{ta} and h_{lock} for fast lookups of lock headers and transaction entries. The hash tables, lock

[4] Mechanisms for the prevention of phantoms are specific to the index structures, but, because this is also a problem in relational indexes, similar solutions are usually applicable.

headers and the transaction entries use a fast latching mechanism to minimize synchronization points between concurrent threads.

A lock request is generally processed as follows. When a transaction T requests lock mode m for object o, h_{ta} is used to find the transaction entry te of T. If it does not exist, a new one is created. Then h_{lock} is used to find the lock header h of o. If o is not locked, a new lock header for o is registered in h_{lock}. If T already holds a lock for o, it tries to replace the currently granted mode with a mode that covers both the old granted and the requested mode. If it is T's first lock request for o it creates a new request r and appends it to the request queue of h. If the requested mode is compatible with all requests of other transactions, the lock is granted. Otherwise, T must wait until either all incompatible locks of other transactions are released, the request timed out or the transaction is aborted by the deadlock detector due to a circular wait-for-relationship.

Because we need to synchronize objects of varying types occurring at diverse system layers (e.g., pinning pages by the buffer manager and locking XML-related objects such as nodes, edges, and indexes), which exhibit incomparable lock compatibilities, very short to very long lock durations, we encapsulated everything in so-called lock services, which provide a convenient interface to the various system components [1].

To simplify and speed up lock management for our hierarchical lock protocols, we made our lock implementation "tree-aware". First, the node lock service automatically infers from a request, e.g., mode NR for SPLID 1.25.3.7 directly the ancestor path and the required intention locks on this path. These locks are acquired from the lock table in a single request. The lock table itself gets the transaction entry for the requesting transaction and starts granting the requests along the ancestor path from root to the leaf generally in the similar manner as sketched above.

When an intention lock on an ancestor node is granted that has been locked by the requestor before, it checks if the granted mode on the ancestor is strong enough to cover also the actual leaf request, e.g., when the node with SPLID 1.25 was already locked with an SR lock. If the ancestor lock is strong enough to satisfy the actual leaf request, we can directly stop the acquisition of further locks along the path.

5 Protocol Optimization

As it turned out by empirical experiments, lock depth is the most important and performance-critical parameter of an XML lock protocol. Lock depth n specifies that individual locks isolating a 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, conflicting operations often occur at levels closer to the document root

(at lower levels) such that fine-grained locks (and their increased management) at levels deeper in the tree do not always pay off. Obviously, taDOM can easily be adjusted to the lock-depth parameter. A general reduction of the lock depth, however, would jeopardize the benefits of our tailored lock protocols.

5.1 Dynamic Lock Depth Adjustment

Obviously, the optimal choice of lock depth depends on document properties, workload characteristics and other runtime parameters like the number of concurrent users etc., and cannot be decided statically. The most effective solution to reduce lock management overhead at runtime is lock escalation: 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 dynamically reduce lock depth and lock subtrees closer to the document root using single subtree locks instead of separately locking each descendant node visited. We aim at running transactions initially at a high lock depth to benefit from the fine-grained resolution of our lock protocols in hot-spot regions, but reserve the option to dynamically reduce the lock depth in low-traffic regions encountered to save system resources.

We use a counter for each request object, which is incremented everytime a node is locked intentionally as the direct parent for a lock request. If this counter reaches a certain threshold, indicating that—depending on the level, which we know from the tree-aware lock table—relatively much children of this node are already locked[5], it seems very likely that the transaction will access further child nodes, and it would be beneficial to escalate the intention lock request either with an LR lock if a simple NR lock was requested for the child, or even with a shared or exclusive lock for the whole subtree depending on the requested mode for the child. To avoid blocking situations, we exploit the context knowledge from lock table about the current lock mode and the requests of all transactions for that node again, and check whether concurrent transactions already hold incompatible locks, before we finally decide about the subtree-local lock escalation.

The escalation thresholds are computed from the simple formula $threshold = k * 2^{-level}$, which takes into account that typically the fanout as well as the conflict potential decreases on deep levels. The parameter k can be adjusted according to current runtime properties.

5.2 Use of Update Locks

Update locks are special lock modes used to avoid so-called conversion deadlocks. These deadlocks arise if two transactions read the same object and then both attempt to upgrade the lock for modification. In relational systems, update locks are mainly used for update cursors. They allow for a direct upgrade to exclusive

[5] The actual number of locked child nodes may be less if the same child node is locked several times.

lock mode when the transaction decides to modify the current record, or for a downgrade to a shared lock when the cursor is moved to the next record without any changes. Transactions in XDBMS do not follow such easy access patterns. Instead, they often perform arbitrary navigation steps in the document tree, e.g., to check the content child elements, before modifying a previously visited node.

Tests with our XDBMS prototype XTC revealed that many deadlocks result from the conversion of edge locks and not from the conversion of node locks. In a typical situation of such deadlocks, for example, two transactions try to append a new fragment under a node when they have already acquired a shared lock for its last-child edge while checking the value of the ID attribute of the current last child. As the insertion of a new last child requires conversion to an exclusive lock for the current last-child edge, both transactions form a deadlock. Hence, we have to carefully enrich our access plans with hints when to use update locks for accessing nodes, subtrees, or edges.

6 Experimental Results

We evaluated the described optimizations in our XDBMS prototype XTC with a mix of eight transaction types, which access and modify a generated XMark document at varying levels and in different granules. Three transaction types are read-only and access the document in various ways. The first one simply reconstructs the subtree of an item, the second iterates over the person siblings and item siblings to find the seller of an item, and the third one fetches the mails in the mailbox of an item. The update transactions examine smaller parts of the document through navigation, before they change the structure and the content of the document, e.g., by placing bids on items, by changing user data, or by inserting new users, items, or mails.

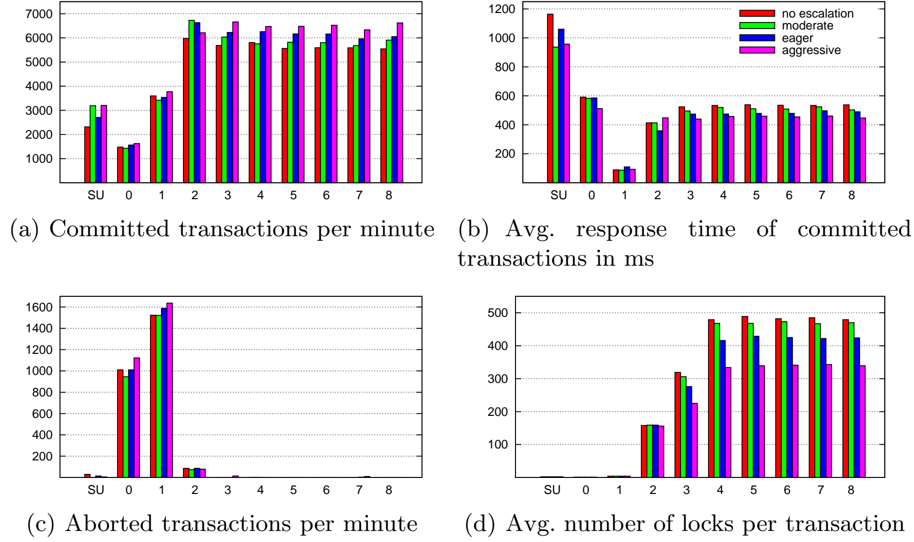
In all cases, we used an additional element index to randomly select a jump-in point for the transaction. To place a bid, for example, we first perform some navigation in a randomly selected open_auction subtree to check the current highest bid and to determine which content nodes have to be modified, before the new bid is actually inserted.

As we wanted to examine only the behaviour of the locking facilities in situations with a high blocking potential and not the influence of other system parameters, we chose an initial document size of only 8 MB and used a buffer size large enough for the document and the additional element index. In our experiments, we only used the taDOM3+ protocol, because it outperforms all other protocols of the taDOM family, and focused on lock-depth optimization.

In the first experiment, we evaluated the influence of the escalation heuristics and the parameter k on the effectiveness and efficiency of the lock protocol. The benchmark load was produced by 50 client instances, which continuously started transactions of a randomly selected type. We weighted the transaction types to achieve a balanced workload that evenly accesses and modifies the document at lower and deeper levels. For each escalation heuristics, we varied the initial lock depth from 0 to 8 and measured throughput, response time, abort rate, and the

number of locks required for each configuration in several benchmark runs of one minute. For the escalation heuristics, we chose the parameter k equal to 2048 (called *moderate*) and 1536 (*eager*) respectively 512 (*aggressive*).

To document the general benefit of an XML lock protocol, we run the benchmark also in a “single-user mode”, which allowed scheduled transactions only exclusive access to the documents.



(a) Committed transactions per minute

(b) Avg. response time of committed transactions in ms

(c) Aborted transactions per minute

(d) Avg. number of locks per transaction

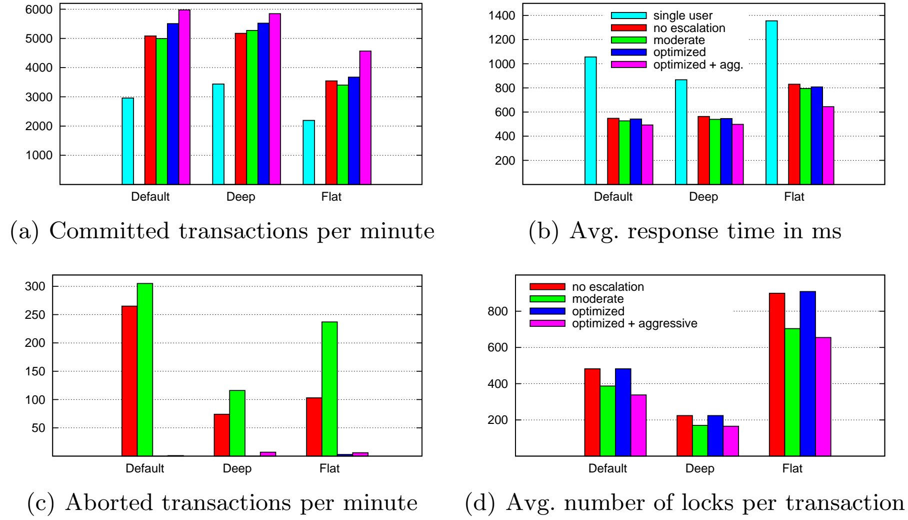
Fig. 3. Effect of lock escalation on transaction throughput and response times.

The results in Fig. 3(a) reveal that the reduced lock overhead of our dynamic escalation mechanism has a positive influence on transaction throughput. The highest transaction rates were already achieved at lock depth 2, which seems fitted best to our test workload. For all higher lock depths, throughput remains relativley constant for all heuristics and also remarkably higher than in single user mode. The comparison of the escalation heuristics proves that less lock overhead directly leads to higher throughput.

The average response times of successful transactions (Fig. 3(b)) mirror the results shown in Fig. 3(a) for the lock depth 2 and higher. Shortest response times correspond to highest transaction throughput.

Fig. 3(c) shows that the lock depths 0 and 1 suffered from extremely high abort rates, caused by a high amount of conversion deadlocks, which immediately arise when a write transaction first uses shared subtree locks at root level or level 1, and then tries to convert this lock into an exclusive subtree lock the perform an update operation. Fig. 3(c) also clearly indicates that the escalation heuristics do not lead to higher deadlock rates when the maximum lock depth is chosen appropriately.

Fig. 3(d) demonstrates how effective simple escalation heuristics could reduce overhead of lock management. The number of required locks grows with higher lock depths and then saturates at a certain level. As expected, the aggressive heuristics achieved the best results in this category and saved in the average 30%[6] compared to the plain protocol variant without any escalation. Also the eager heuristics could save a mentionable amount of locks, whereas the benefit of the moderate heuristics was only marginal.



(a) Committed transactions per minute

(b) Avg. response time in ms

(c) Aborted transactions per minute

(d) Avg. number of locks per transaction

Fig. 4. Results of workload variations and adjusted escalation strategies.

For our second experiment series, we took the balanced workload of the previous experiment (denoted *default*) and changed the weights of the transaction types to create a workload mix that mainly consists of transaction types that access and modify the document at deeper levels (*deep*) and another one that mainly operates on higher document levels (*flat*). We ran the benchmark with maximum lock depth 8, which allows fine-grained locks even deep in the document, and again in single-user mode. For these configurations, we also evaluated variants with moderate escalation heuristics from the previous experiment, a variant where we modified the transaction types to make careful use of update locks (*optimized*) to avoid conversion-induced deadlocks, and a fourth variant (*optimized + aggressive*) that combined the use of update locks with the aggressive heuristics, which produced the best results in the previous experiment.

The results in Fig. 4(a) proof again that all variants of taDOM3+ achieve higher transaction throughput than the solution with exclusive document access.

[6] The actual savings potential is in fact even considerably higher, because acquired locks can be removed as soon as they become obsolete after an escalation.

The optimized version with aggressive escalation nearly achieves a gain of 20% in throughput as compared to the plain taDOM version. Of course, we observe similar results for the response times in Fig. 4(b), too.

The abort rates in Fig. 4(c) show the benefit of carfully set update lock modes during processing. The deadlock rate decreases to nearly zero, which in turn explains the throughput gain in Fig. 4(a).

Finally, Fig. 4(d) illustrates that our optimizations complement each other. On the one hand, correct application of update lock modes is helpful if lock escalations are used, because this increases danger of deadlocks otherwise. On the other hand, lock escalations help to reduce the overhead of lock management.

Altogether, the experimental results demonstrate well that a fine-grained locking approach pays off and provides higher throughput and shorter response times than exclusive document locks. The experiments also confirmed that taDOM3+ in combination with our adaptations is able to provide constantly high transaction throughput for higher lock depths, and that it can effectively and efficiently be adjusted to varying workloads to achieve high concurrency without a waste of system resources.

A closer analysis of the shown results revealed that our protocols would allow even much higher concurrency in the XML tree for lock depths higher than 2. Here, however, the data structures of the physical storage layer—which is a B*-tree in our prototype—became the bottleneck.

7 Conclusions and Outlook

In this paper, we explained the realization of fine-grained concurrency control for XML. We started with an introduction into the basics of our tailor-made lock protocols, which are perfectly eligible for a fine-grained transaction isolation on XML document trees, and emphasized the advantages of prefix-based node labeling schemes for lock management. Thereafter, we turned on general implementation aspects, where we showed how the XML protocols can be integrated in a layered architecture and how even different storage models and indexes can be incorporated into our concept. We also explained how we adapted the a widely used lock manager architecture for our needs and presented ways to optimize the runtime behaviours of the lock protocols.

In our future work, we will focus on the integration of advanced XML indexes, which make use of structural document summaries, in our isolation concept, and the interplay between XML concurrency control on the one hand and efficient query evaluation algorithms for declarative queries based on XQuery on the other hand.

References

1. Bächle, S., and Härder, T.: Tailor-made Lock Protocols and their DBMS Integration. In Proc. EDBT'08 Workshop on Software Engineering for Tailor-made Data Management: 18-23 (2008).

2. Böhme, T., and Rahm, E.: Supporting Efficient Streaming and Insertion of XML Data in RDBMS. In Proc. 3rd Int. Workshop Data Integration over the Web, Riga, Latvia, (2004) 70-81.
3. Chen, Q., Lim, A., Ong, K. W., and Tang, J.: Indexing XML documents for XPath query processing in external memory. Data & Knowledge Engineering 59(3): 681-699 (2006).
4. Christophides, W., Plexousakis, D., Scholl, M., and Tourtounis, S.: On Labeling Schemes for the Semantic Web. In Proc. 12th Int. WWW Conf., Budapest, 544-555 (2003).
5. Dewey, M.: Dewey Decimal Classification System. http://frank.mtsu.edu/ vvesper/dewey2.htm.
6. Document Object Model (DOM) Level 2 / Level 3 Core Specific., W3C Recommendation.
7. Grabs, T., Böhm, K., and Schek, H.-J.: XMLTM: Efficient transaction management for XML documents. In Proc. CIKM Conf., 142-152 (2002).
8. Gray, J: Notes on Database Operating Systems. In Operating Systems: An Advanced Course, Springer, LNCS 60: 393-481 (1978).
9. Gray, J., and Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann (1993).
10. Härder, T., Haustein, M. P., Mathis, C., and Wagner, M.: Node Labeling Schemes for Dynamic XML Documents Reconsidered. Data & Knowledge Engineering 60(1): 126-149 (2007).
11. Haustein, M. P., and Härder, T.: Optimizing lock protocols for native XML processing. Data & Knowledge Engineering 65(1): 147-173 (2008).
12. Helmer, S., Kanne, C.-C., and Moerkotte, G.: Evaluating Lock-Based Protocols for Cooperation on XML Documents. SIGMOD Record 33(1): 58–63 (2004).
13. Jagadish, H. V., Al-Khalifa, S., and Chapman, A.: TIMBER: A native XML database. The VLDB Journal 11(4): 274-291 (2002).
14. O'Neil, P., O'Neil, E., Pal, S., Cseri, I., Schaller, G., and Westbury, N.: ORDPATHs: Insert-Friendly XML Node Labels. In Proc. SIGMOD Conf.: 903–908 (2004).
15. P. Pleshachkov, P. Chardin, and S. Kusnetzov: XDGL: XPath-based Concurrency Control Protocol for XML Data. In Proc. 22nd Britisch National Conf. on Databases (BNCOD), UK Bd. 3567, Springer, pp. 145-154 (2005)
16. P. Pleshachkov, P. Chardin, and S. Kusnetzov: SXDGL: Snapshot Based Concurrency Control Protocol for XML Data. In Proc. 5th International XML Database Symposium, XSym 2007, LNCS 4704, Springer, 122-136 (2007).
17. Sardar Z., and Kemme, B.: Don't be a Pessimist: Use Snapshot based Concurrency Control for XML. In Proc. 22nd Int. Conf. on Data Engineering, 130 (2006).
18. Schmidt, A., Waas, F., Kersten, M. L., Carey, M. J., Manolescu, I., and Busse, R.: et al.: XMark: A Benchmark for XML Data Management. In Proc. VLDB Conf.: 974-985 (2002).
19. Schöning, H.: Tamino – A DBMS designed for XML. In Proc. 17th Int. Conf. on Data Engineering: 149-154 (2001).
20. Siirtola A., and Valenta, M.: Verifying Parameterized taDOM+ Lock Managers. In Proc. SOFSEM Conf., Springer, LNCS 4910: 460-472 (2008).
21. XQuery 1.0: An XML Query Language. http://www.w3.org/XML/XQuery
22. XQuery Update Facility. http://www.w3.org/TR/xqupdate
23. Yu, J. X., Luo, D., Meng, X., and Lu, H.: Dynamically Updating XML Data: Numbering Scheme Revisited. World Wide Web 8(1): 5-26 (2005)