Tailor-made Lock Protocols and their DBMS Integration

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

We outline the use of fine-grained lock protocols as a concurrency control mechanism for the collaboration on XML documents and show that their tailor-made optimization towards the access model used (e.g., DOM operations) pays off. We discuss how hard-wired lock services can be avoided in an XML engine and how we can, based on loosely coupled services, exchange lock protocols even at runtime without affecting other engine services. The flexible use of these lock protocols is further enhanced by enabling automatic runtime adjustments and specialized optimizations based on knowledge about the application. These techniques are implemented in our native XML database management system (XDBMS) called XTC [5] and are currently further refined.[1]

1. MOTIVATION

The hierarchical structure of XML documents is preserved in native XDBMSs. The operations applied to such tree structures are quite different from those of tabular (relational) data structures. Therefore, solutions for concurrency control optimized for relational DBMSs will not meet high performance expectations. However, efficient and effective transaction-protected collaboration on XML documents [11] becomes a pressing issue because of their number, size, and growing use. Tailor-made lock 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 [9,10], a general solution has to support fine-grained locking – besides for separating declarative XQuery requests – for concurrently evaluating stream-, navigation-, and path-based queries. 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. We have already developed a family consisting of four DOM-based lock protocols [6]. Here, our focus is on the engineering aspects how such protocols can be efficiently integrated, but sufficiently encapsulated in an XDBMS such that they can be automatically exchanged or adapted to new processing situations at runtime.

[1] This work has been partially supported by the German Research Foundation (DFG).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EDBT SETMDM Workshop 2008, March 29, 2008, Nantes, France.
Copyright 2008 ACM 978-1-59593-000-0/00/0000...$5.00.

In Section 2, we explain the need for lock protocols tailored to the specific characteristics of XML processing, before we outline the mechanism underlying the runtime exchange of lock protocols in Section 3. Using the concept of meta-locking, we sketch in Section 4 how we achieved cross-comparison of 12 lock protocols in an identical XDBMS setting without any system modification. Here, XTC (XML Transaction Coordinator, [5]) served as a testbed for all implementations and comparative experiments. Various forms of runtime adjustments on lock protocols are discussed in Section 5, before we introduce further ways of protocol specializations in Section 6. Finally, Section 7 concludes the paper.

2. FINE-GRAINED DOM-BASED LOCKING

Because our XML documents are stored in a B*-tree structure [5], the question whether or not specific tree-based lock protocols can be used immediately arises. So-called B-tree lock protocols provide for structural consistency while concurrent database operations are querying or modifying database contents and its representation in B-tree indexes [2]. Such locks also called latches isolate concurrent operations on B-trees, e.g., while traversing a B-tree, latch coupling acquires a latch for each B-tree page before the traversal operation is accessing it and immediately releases this latch when the latch for the successor page is granted or at end of operation at the latest [1]. In contrast, locks isolate concurrent transactions on user data and – to guarantee serializability [4] – have to be kept until transaction commit. Therefore, such latches only serve for consistent processing of (logically redundant) B-tree structures.

Hierarchical lock protocols [3] – 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 at each level can be isolated acquiring the usual locks with modes R (read), X (exclusive), and U (update with conversion option), which implicitly lock all objects in the entire subtree addressed. To avoid lock conflicts when objects at different levels are locked, so-called intention locks with modes IR (intention read) or IX (intention exclusive) have to be acquired along the path from the root to the object to be isolated and vice versa when the locks are released [3]. 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,

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

Figure 1. Lock compatibilities for taDOM2

because the lock protocol unnecessarily prevents writers to access nodes somewhere in the subtrees. Giving a solution for this problem, we want to sketch the idea of lock granularity adjustment to DOM-specific navigational operations. 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. Edge locks having three modes [6] mainly serve for phantom protection and, due to space restrictions, 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 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. Hence, these XML-specific lock modes 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 (the compatibility is then decided on the child nodes themselves).

Figure 1 contains the compatibility matrix for our basic lock protocol called taDOM2. To illustrate its use, let us assume that the node

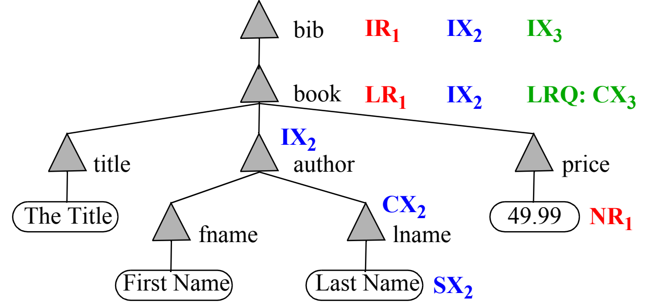


Figure 2. taDOM2 locking example

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. Our prefix-based node labeling scheme called SPLIDs (stable path labeling identifiers are similar to OrdPaths [8]) greatly supports lock placement in trees, because SPLIDs carry the node labels of all ancestors. Hence, access to the document is not needed to determine the path to a context node. 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, 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.

Hence, by tailoring the lock granularity to the LR operation, the lock protocol enhances concurrency by allowing modifications of other transactions in subtrees whose roots are read-locked.

3. USE OF A PROTOCOL FAMILY

Experimental analysis of this protocol led to some severe performance problems in specific situations which were solved by the follow-up protocol taDOM2+. Conversion of LR was particularly expensive. Assume T_1 wants modify *price* and has to acquire an SX lock on it in the scenario sketched in Figure 2. For this purpose, the taDOM2 protocol requires a CX lock on its parent *book*. Hence, the existing LR has to be converted into a CX lock and, in turn, a successful conversion requires NR locks on all children (potentially read by T_1) of *book*. As opposed to ancestor determination by the SPLID of a context node, the lock manager cannot calculate the SPLIDs of its children and, hence, has to access the document to explicitly locate all affected nodes – a very expensive operation. By introducing suitable intention modes, 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 – to optimize specific conver-

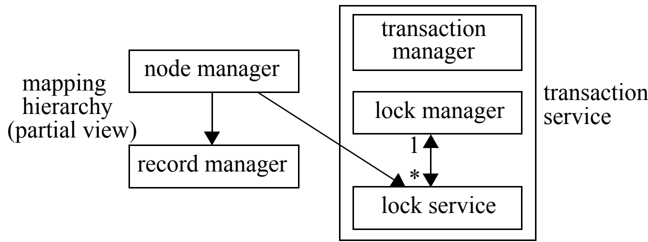


Figure 3. Interaction of node manager and locking facilities

sions – we added even more intention modes resulting in the truly complex protocol taDOM3+ specifying compatibilities and conversion rules for 20 lock modes (see [6] for details).

While in our initial implementation the taDOM2 protocol was hardwired, we looked for an elegant integration mechanism to transparently enable protocol changes (e.g., another compatibility matrix) or use of alternative protocols. Therefore, 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. 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 3 sketches the interaction of the node manager and the locking facilities involved in protocol use, lock mode selection, and application 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 able to change the lock protocol by simply exchanging the lock service used by the node manager. Furthermore, it is now even possible to use multiple protocols, e.g., taDOM2+ and taDOM3+, simultaneously for different kinds of and workloads for documents inside the same server instance.

4. 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.

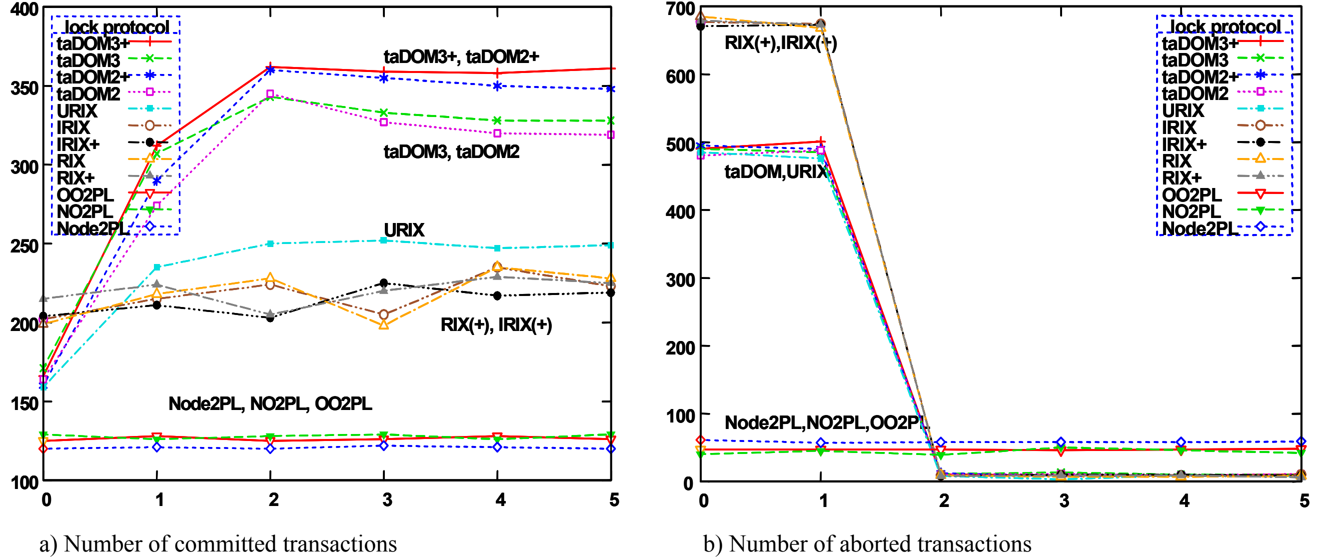
We found quite different approaches to fine-grained tree locking in the literature and identified three families with 12 protocols in total: Besides our taDOM* group with 4 members, we adjusted the relational MGL approach [3] to the XML locking requirements and included 5 variants of it (i.e., IRX, IRX+, IRIX, IRIX+, and URIX) in the so-called MGL* group. Furthermore, three protocol variants described in [7] were developed as DOM-based lock protocols in the Natix context (i.e., Node2PL, NO2PL, and OO2PL), but not implemented so far. They are denoted as the *2PL group.

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 concept of level locks and the mismatch of the *2PL group with missing subtree 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.

- *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.



a) Number of committed transactions

b) Number of aborted transactions

Figure 4. Overall results of a transaction benchmark (variation of lock depth)

Although the MGL* group is only distantly related and the *2PL group is not related at all 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 [5] 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 4a 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. On the other hand, the MGL protocols (only level locks missing) roughly doubled the transaction throughput as compared to the *2PL group. Finally, the taDOM* group almost doubled the throughput compared to the MGL* group. A reasonable application to achieve fine-grained protocols requires at least a lock depth of *2*, which also confirms the superiority of MGL and taDOM in terms of deadlock avoidance (see Figure 4b).

Hence, the impressive performance behavior of the taDOM* group reveals that a careful adaptation of lock granules to specific operations clearly pays off (see again the discussion in Section 2).

5. RUNTIME PROTOCOL ADJUSTMENT

At runtime, the main challenge for concurrency control is the preservation of a reasonable balance of concurrency achieved and locking overhead needed. The most effective and widely used solution for this objective is called *lock escalation*: The fine-grained resolution of a lock protocol is – preferably in a step-wise manner – reduced by acquiring coarser granules. For example in relational systems, the page containing a specific record is locked instead of the record itself. If too many pages are to be locked in the course of processing, the lock manager may try to acquire a single lock for the entire table. In B-tree indexes, separator keys of non-leaf pages can be exploited as intermediate lock granularities to improve scalability [2]. Although native XDBMSs often store the document nodes in page-oriented, tree-based index structures, too, an escalation from node locks to page locks is not suitable anymore. Because the nodes are stored in document order and, as a consequence, fragments of multiple subtrees can reside in the same physical page, page level locks would jeopardize the concurrency benefits of hierarchical locks on XML document trees. Hence, lock escalation in our case means the reduction of the lock depth: we lock subtrees at a higher level using a single lock instead of separately locking each descendant node visited. Consequently, the number of escalation levels is limited by the structure of a document and not by its physical DBMS representation. This is also a notable aspect of our encapsulation design.

The acquisition of a coarser lock granule than actually needed is typically triggered by the requestor of the locks itself, e.g., a subtree scan operator, or by the lock manager, when the number of maintained locks for a transaction reaches a critical level or the requested lock is at a deeper level than the pre-defined maximum lock depth. Admittedly, especially the heuristics of the lock manager is rather a mechanism enabling a system to handle large documents than an optimization for higher transaction throughput. The lock escalation is performed independently of the processing context, because it is *necessary* and not because it is *wise*. Therefore, we present in the following some concepts that allow us to go further than a simple reduction of the global lock depth and to increase the performance of the system by doing smart lock escalation.

We aim to preserve fine-grained resolution of our lock protocols in hot spot regions of a document to maintain high concurrency, while we gracefully give it up in low-traffic regions to save system resources. In doing so, we have to face the challenge to decide whether or not it is good idea to perform a lock escalation on a subtree. Again, the solution lies in the design of our locking facilities. By making the lock manager "tree-aware", we can very easily exploit its implicit knowledge about the traffic on a document. Lock requests for a specific document node trigger the lock manager to transparently acquire the required intention locks on the ancestor path. The ancestor nodes and the appropriate lock modes are, of course, provided by the lock service. Thus, the lock table 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. This information can be used as input for a heuristics to decide about local lock escalations in specific subtrees.

The example depicted in Figure 5 illustrates how this cheaply gathered information can be used effectively: Transaction T_1 requests a shared lock for the highlighted element node at level 6, but before this request can be granted, appropriate intention locks on the ancestor path have to be acquired. At level 4, the lock table recognizes that T_1 already requested 50 shared intention locks on this node. This indicates that T_1 has already done some navigation steps in the subtree rooted at the current node, and that it will probably perform some more. Therefore, the lock table *asks* the lock service if the initial lock request at level 6 should be overridden by a stronger lock at level 4 to limit the number of locks in case that T_1 continues its navigation in the subtree. Because the target level is more than one level deeper and locks of other transactions are not present on this node, the lock service decides in favor of a shared subtree lock on the ancestor node to save resources. If the node at level 4, however, would also be locked by another transaction T_2 with an intention exclusive lock, it would probably apply the escalation to the parent node at level 5 to avoid a blocking situation.

The great advantage of this approach is that collection of additional information is avoided and that all relevant information is always immediately available when needed. In our example, the levels and the distance of the current and the target level served as weights in this decision process. Nevertheless, it is also possible to incorporate high-level information like statistics about the current transaction workload or the document itself. Besides the maximum depth and the number of document nodes, especially the fan-out characteristics of the document can be helpful. Depending on the complexity of the heuristics used, however, it is reasonable to reduce the additional computational overhead to a minimum. For that reason, the lock modes and the number of requests for a specific resource might only be consulted, if the number of locks held by the current transaction reaches a certain threshold, and only if they indicate an optimization potential, further information has to be evaluated.

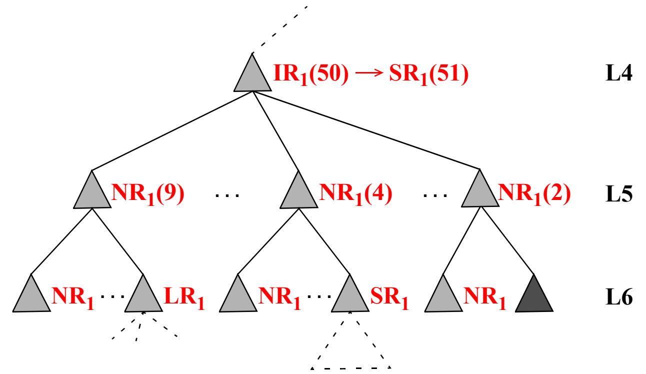


Figure 5. Lock escalation example

Although navigation on the XML document tree or evaluation of declarative queries formulated in XPath or XQuery allow many degrees of freedom and cause varying access patterns, some basic tasks like the evaluation of positional predicates are repeated very often. Hence, another promising way to optimize the application of our protocols is the adaptation to typical access patterns. To give a simple example, assume that a transaction wants to navigate to the first child node of an element that satisfies a certain search criterion. The transaction can either fetch all child nodes by a single call of *GetChildNodes()* and pick the searched one, or descend to the first child node with *GetFirstChild()* and scan for the child node with *GetNextSibling()*. In the first case, only a single LR lock is required to protect the operation, but many child nodes are unnecessarily locked possibly causing lower concurrency. In the second case, concurrency may be much higher, since only those nodes are locked that are necessary to keep the navigation path stable. However, the lock overhead is higher, too, because each call of *GetNextSibling()* requires the acquisition of additional locks.

Obviously, it depends on the position of the searched child node and the fan-out of the specific element whether the first or the second alternative is better, and, ideally, the caller respectively the query optimizer should choose the appropriate alternative. In most cases, however, it is not possible to reliably predict the best one. Then, it is reasonable to start with the second strategy to preserve high concurrency. If the search does not stop until the *k*-th child is reached, the lock service can be advised to try lock escalation and to lock all siblings with an LR lock on the parent, because further calls of *GetNextSibling()* will follow with high probability. Instead of waiting for an explicit hint, the node manager itself could also keep track of the *n* most recent navigation steps and predict the best lock mode for the next steps.

The goal of this kind of pre-claiming is to request locks that are not only sufficient for the current operation but also optimal for future operations. In contrast to the on-demand escalation capabilities we

described before, such context-sensitive heuristics can be applied much earlier and lead to better overall performance, because not only locking overhead but also danger of deadlocks are reduced.

Further heuristics are primarily designed to speed up the evaluation of declarative queries. The context nodes for a query result, for example, are usually identified before the actual output is computed. Because the output often embodies nodes in the subtree rooted at the context nodes, the respective nodes or subtrees can already be locked when a context node is identified. This avoids late blocking of a transaction during result construction. Contrarily, the lock depth can be locally increased in the identification phase to reduce the risk of blocking other transactions on examined nodes that did not qualify for the query result.

6. PROTOCOL SPECIALIZATION

So far, we described general runtime optimizations for a universal XDBMS. Further improvements are possible if we take also aspects of the applications themselves into account and adjust our lock protocols accordingly. Since the adaptations are by nature very special to a specific application area, we do only sketch a few scenarios at this point to give an idea of how such specializations may look like.

Applications that change the documents in a uniform manner appear to be a promising field. A special case are "append-only" scenarios like digital libraries where the document order does not play an important role, and new nodes or subtrees are only appended. Existing structures remain untouched and, at the most, the content of existing nodes is updated, e.g., to correct typing errors. This allows us to omit the edge locks most of the time because transactions must not protect themselves from phantom insertions between two existing nodes. In addition to that, insertions can be flagged with an additional exclusive insert lock to signal other transactions that they can skip the new last child if they do not rely on the most recent version of the document.

Many applications do also rely on a specific schema to define tree-structured compounds that reflect logical entities like customer data, addresses, or products, and are typically accessed as a whole. Hence, such entities in a document may be identified with the help of a schema and directly locked with a single subtree lock, whereas other parts are still synchronized with fine-grained node locks. In an XML-based content management system, for example, the metadata is accessed and modified in very small granules, whereas the content parts usually consist of large subtrees with some sort of XML markup that are always accessed as a logical unit. In some applications it might even be possible to change the basic lock granule from single document nodes to larger XML entities like complex XML-Schema types.

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 a fine-grained transaction isolation on XML document trees, and showed how they can be encapsulated and integrated in an XDBMS environment. By introducing the concept of meta-locking, we discussed the software-engineering principles for the exchange of the lock service responsible for driving 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.

Moreover, we presented further refinements of our encapsulation design, which allows us to easily control and optimize the runtime behavior of our lock protocols without making concessions to the encapsulation and exchangeability properties. Finally, we outlined some possibilities to customize the protocols for special application scenarios. In our future work, we 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.

8. REFERENCES

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

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

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

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

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

[6] M. Haustein and T. Härder: Optimizing lock protocols for native XML processing. To appear in Data & Knowledge Engineering 2008.

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

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

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

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

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