The Real Performance Drivers Behind XML Lock Protocols

Sebastian Bächle and Theo Härder

University of Kaiserslautern, Germany
{baechle,haerder}@cs.uni-kl.de

Abstract. Fine-grained lock protocols should allow for highly concurrent transaction processing on XML document trees, which is addressed by the taDOM lock protocol family enabling specific lock modes and lock granules adjusted to the various XML processing models. We have already proved its operational flexibility and performance superiority when compared to competitor protocols. Here, we outline our experiences gained during the implementation and optimization of these protocols. We figure out their performance drivers to maximize throughput while keeping the response times at an acceptable level and perfectly exploiting the advantages of our tailor-made lock protocols for XML trees. Because we have implemented all options and alternatives in our prototype system XTC, benchmark runs for all "drivers" allow for comparisons in identical environments and illustrate the benefit of all implementation decisions. Finally, they reveal that careful lock protocol optimization pays off.

1 Motivation

Native XML database systems (XDBMSs) promise tailored processing of XML documents, but most of the systems published in the DB literature are designed for efficient document retrieval only [17]. However, XML's standardization and, in particular, its flexibility (e.g., data mapping, cardinality variations, optional or non-existing structures, etc.) are driving factors to attract demanding write/read applications, to enable heterogeneous data stores and to facilitate data integration. Because business models in practically every industry use large and evolving sets of sparsely populated attributes, XML is more and more adopted by those companies which have even now launched consortia to develop XML Schemas adjusted to their particular data modeling needs.[1] For these reasons, XML databases currently get more and more momentum if data flexibility in various forms is a key requirement of the application and they are therefore frequently used in collaborative or even competitive environments [11]. As a consequence, the original "retrieval-only" focus – probably caused by the first proposals of XQuery respectively XPath where the update part was left out – is not enough anymore. Hence, update facilities are increasingly needed in

[1] World-leading financial companies defined more than a dozen XML vocabularies to standardize data processing and to leverage cooperation and data exchange [20].

XDBMSs, i.e., fine-grained, concurrent, and transaction-safe document modifications have to be efficiently supported. For example, workloads for *financial application logging* include 10M to 20M inserts in a 24-hour day, with about 500 peak inserts/sec. Because at least a hundred users need to concurrently read the data for troubleshooting and auditing tasks, concurrency control is challenged to provide short-enough response times for interactive operations [11].

Currently, all vendors of XML(-enabled) DBMSs support updates only at document granularity and, thus, cannot manage highly dynamic XML documents, let alone achieve such performance goals. Hence, new concurrency control protocols together with efficient implementations are needed to meet these emerging challenges. To guarantee broad acceptance, we strive for a *general solution* that is even applicable for a spectrum of XML language models (e.g., XPath, XQuery, SAX, or DOM) in a multi-lingual XDBMS environment. Although predicate locking for declarative XML queries would be powerful and elegant, its implementation rapidly leads to severe drawbacks such as undecidability and application of unnecessarily large lock granules for simplified predicates – a lesson learned from the (much simpler) relational world. Beyond, tree locks or key-range locks [5, 12] are not sufficient for fine-grained locking of concurrently evaluated stream-, navigation- and path-based queries. Thus, we necessarily have to map XQuery operations to a navigational access model to accomplish fine-granular locking supporting other XML languages like SAX and DOM [3], too, because their operations directly correspond to a navigational access model.

To approach our goal, we have developed a family consisting of four DOM-based lock protocols called the taDOM group by adjusting the idea of multi-granularity locking (MGL) to the specific needs of XML trees. Their empirical analysis was accomplished by implementing and evaluating them in XTC (XML Transaction Coordinator), our prototype XDBMS [9]. Its development over the last four years accumulated substantial experience concerning DBMS performance in general and efficient lock management in particular.

1.1 The taDOM Protocol Family

Here, we assume familiarity of the reader with the idea of multi-granularity locking (MGL) – also denoted as hierarchical locking [6] – which applies to hierarchies of objects like tables and tuples and is 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 [6].

Although an 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

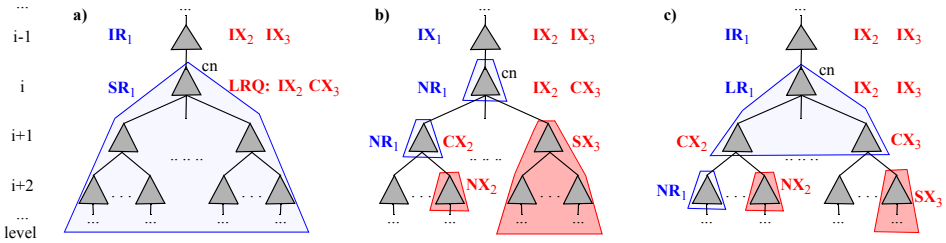


Fig. 1. Example of the taDOM3+ protocol

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.

For ease of comprehension, we will give a brief introduction into the essentials of our taDOM lock protocols (beyond the MGL modes just sketched), which refine the MGL ideas and provide tailored lock modes for high concurrency in XML trees [9]. 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 for *all siblings under a parent* 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.

To enable transactions to traverse paths in a tree having (levels of) nodes already read-locked by other transactions and to modify subtrees of such nodes, a new intention mode CX (child exclusive) had to be defined for a context (parent) node. 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). Altogether these new lock modes enable serializable transaction schedules with read operations on inner tree nodes, while concurrent updates may occur in their subtrees.[2] An important and unique feature (not applicable in MGL or other protocols) is the optional variation of the *lock depth* which can be dynamically controlled by a parameter. Lock depth n determines that, while navigating through the document, individual locks are acquired for existing nodes up to level n. If necessary, all nodes below level n are locked by a subtree lock (SR, SX) at level n.

[2] Although edge locks [9] are an integral part of taDOM, too, they do not contribute specific implementation problems and are, therefore, not considered here.

Continuous improvement of these basic concepts led to a whole family of lock protocols, the taDOM family, and finally resulted in a highly optimized protocol called taDOM3+ (tailor-made for the operations of the DOM3 standard [3]), which consists of 20 different lock modes and "squeezes transaction parallelism" on XML document trees to the extent possible. Correctness and, especially, serializability of the taDOM protocol family was shown in [9, 18].

Let us highlight by three scenarios taDOM's flexibility and tailor-made adaptations to XML documents as compared to competitor approaches. Assume transaction *T1* – after having set appropriate intention locks on the path from the root – wants to read-lock context node *cn*. Independently of whether or not *T1* needs subtree access, MGL only offers a subtree lock on *cn*, which forces concurrent writers (*T2* and *T3* in Fig. 1a) to wait for lock release in a lock request queue (LRQ). In the same situation, node locks (NR and NX) would allow greatly enhance permeability in *cn*'s subtree (Fig. 1b). As the only lock granule, however, node locks would result in excessive lock management cost and catastrophic performance behavior, especially for subtree deletion [8]. A frequent XML read scenario is scanning of *cn* and all its children, which taDOM enables by a single lock with special mode (LR). As sketched in Fig. 1c, LR supports write access to deeper levels in the tree. The combined use of node, level, and subtree locks gives taDOM its unique capability to tailor and minimize lock granules. Above these granule choices, additional flexibility comes from lock-depth variations on demand – a powerful option only provided by taDOM.

1.2 Related Work and our Own Contribution

To the best of our knowledge, we are not aware of contributions in the open literature dealing with implementation of an XML lock manager. 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 [4, 14, 15]. Four Natix lock protocols [10] focus on DOM operations, provide node locks only, and do not enable direct jumps to inner document nodes and effective escalation mechanisms for large documents. Together with four MGL implementations supporting node and subtree locks, their lock protocol performance was empirically compared against our taDOM protocols [8]. The taDOM family exhibited for a given benchmark throughput gains of 400% and 200% compared to the Natix resp. MGL protocols which clearly confirmed that availability of node, level, and subtree locks together with lock modes tailored to the DOM operations pays off.

While these publications only address ideas and concepts, no contribution is known how to effectively and efficiently implement lock protocols on XML trees. Therefore, we start in Sect. 2 with implementation and processing cost of a lock manager. In Sect. 3, we emphasize the need and advantage of prefix-based node labeling for efficient lock management. Sect. 4 outlines how we coped with runtime shortcomings of protocol execution, before effectiveness and success of tailored optimizations are demonstrated by a variety of experimental results in Sect. 5. Finally, Sect. 6 summarizes our conclusions.

2 Lock Manager Implementation

Without having a reference solution, the XTC project had to develop such a component from scratch where the generic guidelines given in [6] were used. 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 and indexes), which exhibit incomparable lock compatibilities, very short to very long lock durations, as well as differing access frequencies, we decided to provide specialized lock tables for them (and not a common one). Where appropriate, we implemented lock tables using suitable lock identification (see node labeling scheme, Sect. 3) and dynamic handling of lock request blocks and queues. Lock request scheduling is centralized by the lock manager. The actions for granting a lock are outlined below. Otherwise, the requesting transaction is suspended until the request can be granted or a time-out occurs. 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 transaction owning the fewest locks.

2.1 Lock Services

Lock management internals are encapsulated in so-called lock services, which provide a tailored interface to the various system components, e.g., for DB buffer management, node locks, index locks, etc. [9]. Each lock service has its own lock table with two pre-allocated buffers for lock header entries and lock request entries, each consisting of a configurable number (m) of blocks, as depicted in Fig. 2. 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 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 (in hash tables T (for transactions) and L (for locks)) to protect against access by concurrent threads thereby guaranteeing the maximum parallelism possible. For each locked object, a lock header is created, which contains name and current mode of the lock together with a pointer to the lock queue where all lock requests for

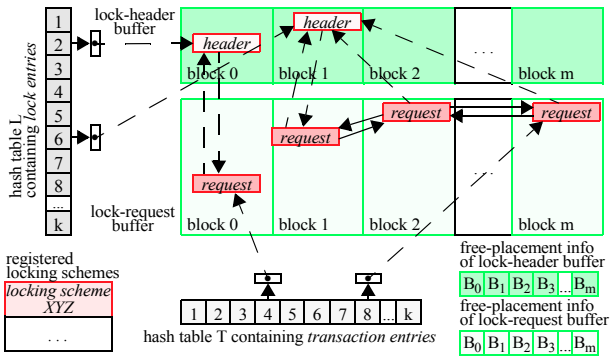


Fig. 2. Data structures of a lock table

the object are attached to. Such a lock request carries among administration information the requested/granted lock mode together with the transaction ID. To speed-up lock release, the lock request entries are doubly chained and contain a separate pointer to the lock header, as shown in Fig. 2. Further, a transaction entry contains the anchor of a chain threading all lock request entries, which minimizes lock release effort at transaction commit.

To understand the general principles, it is sufficient to focus on the management of node locks. A lock request of transaction *T1* for a node with label *ID1* proceeds as follows. A hash function delivers hT(*T1*) in hash table T. If no entry is present for *T1*, a new transaction entry is created. Then, hL(*ID1*) selects (possibly via a synonym chain) a lock entry for node *ID1* 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, various pointer chains are maintained for both entries. The lock manager enables protocol adaptation to different kinds of workloads by providing a number of registered lock schemes [9]. For checking lock compatibility or lock conversion, a pre-specified lock scheme is used.

2.2 Cost of Lock Management

Lock management for XML trees is hardly explored so far. It considerably differs from the relational multi-granularity locking, the depth of the trees may be much larger, but more important is the fact that operations may refer to tree nodes whose labels – used for lock identification – are not delivered by the lock request. 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 [3, 19] 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 document access (to disk), dramatic overhead burdens concurrency control. Therefore, node labeling may critically influence lock management cost.

In a first experiment, we addressed the question how many lock requests are needed for frequent use cases and what is the fraction of costs that can be attributed to the labeling scheme. For this purpose, we used the xmlgen tool of the XMark benchmark project [16] to

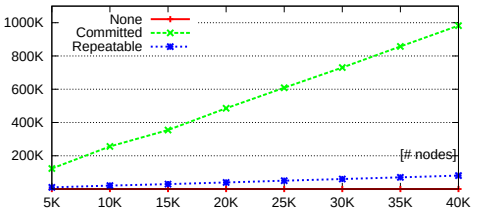
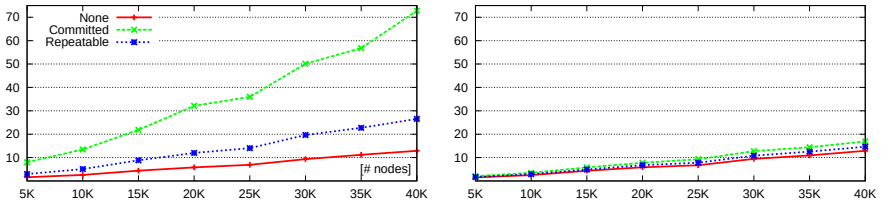


Fig. 3. Number of node locks requested

generate a variety of XML documents consisting of 5,000 up to 40,000 individual XML nodes. These nodes are stored in a B*-tree – a set of doubly chained pages as document container (the leaves) and a document index (the inner pages) – and reconstructed by consecutive traversal in depth-first order (i.e., document order corresponds to physical order) within a transaction in single-user mode.

To explore the performance impact of fine-grained lock management, we have repeated this experiment under various isolation levels [6]. Furthermore, we have reconstructed the document twice to amplify the differing behavior between isolation levels *committed* and *repeatable read* (in this setting, *repeatable* is equivalent to *serializable*). Because of the node-at-a-time locking, such a traversal is very inefficient, indeed, but it drastically reveals the lock management overhead for single node accesses. Depending on the position of the node to be locked, *committed* may cause much more locking overhead, because each individual node access acquires short read locks on all nodes along its ancestor path and their immediate release after the node is delivered to the client. In contrast, isolation level *repeatable read* sets long locks until transaction commit and, hence, does not need to repetitively lock and unlock ancestor nodes. In fact, they are already locked due to the depth-first traversal. Fig. 3 summarizes the number of individual node locks requested for the various isolation levels.

In our initial XTC version, we had implemented SEQIDs (including a level indicator) where node IDs were sequentially assigned as integer values. SEQIDs allow for stable node addressing, because newly inserted nodes obtain a unique, ascending integer ID. Node insertions and deletions preserve the document order and the relationships to already existing nodes. Hence, the relationship to a parent, sibling, or child can be determined based on their physical node position in the document container, i.e., within a data page or neighbored pages. While access to the first child is cheap, location of parent or sibling may be quite expensive depending on the size of the current node's subtree. Because intention locking requires the identification of all nodes in the ancestor path, this crude labeling scheme frequently forces the lock manager to locate a parent in the stored document. Although we optimized SEQID-based access to node relatives by so-called on-demand indexing, the required lock requests (Fig. 3) were directly translated into pure lock management overhead as plotted in Fig. 4a. Hence, the unexpectedly bad and even "catastrophic" traversal times caused a rethinking and redesign of node labeling in XTC (see Sect. 3).



(a) SEQID node labeling (b) SPLID node labeling

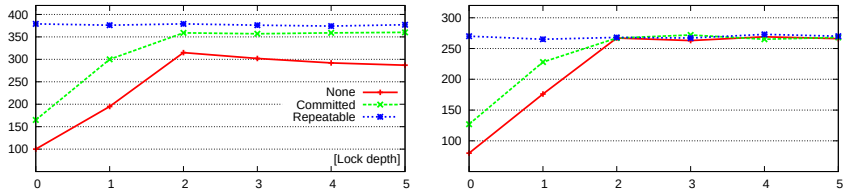
Fig. 4. Documents traversal times (sec.)

2.3 Lower Isolation Levels are Not Always Superior

As compared to *repeatable*, isolation level *committed* provides for higher degrees of concurrency with (potentially) lesser degrees of consistency concerning read/write operations on shared documents. Hence, if chosen for a transaction program, the programmer must be carefully consider potential side-effects, because he accepts responsibility (because of early releases and later reacquisitions of the same locks) to achieve full consistency. As shown in Fig. 4, *committed* may cause higher lock management overhead at the system side. Nevertheless, the programmer expects higher transaction throughput – as always obtained for isolation level *committed* in relational systems – compensating for his extra care.

In a dedicated experiment, we went into this matter whether or not the potentially high locking overhead for isolation level *committed* can be compensated by reduced blocking in multi-user mode. For this scenario, we set up a benchmark with three client applications on separate machines and an XDBMS instance on a forth machine. The clients are executing for a fixed time interval a constant load of over 60 transactions on the server. The workload – repeatedly executed for the chosen isolation levels and the different lock depths – consisted of about 16 short transaction types with an equal share of reader and writer transactions, which processed common access patterns like node-to-node navigation, child and descendant axes evaluation, node value modifications, and fragment deletions.

Fig. 5a shows the results of this benchmark run. Isolation level *none* means that node and edge locks are not acquired at all for individual operations. Of course, processing transactions without isolation is inapplicable in real systems, because the atomicity property of transactions (in particular the transaction rollback) cannot be guaranteed. Here, we use this mode only to derive the upper bound for transaction throughput in a given scenario. Isolation level *repeatable* acquires read and write locks according to the lock protocol and lock depth used, whereas *committed* requires write locks but only short read locks. Note, *committed* leads to a fewer number of successful transactions than the stronger isolation level *repeatable* – and obtains with growing lock depth (and, hence, reduced conflict probability) an increasing difference. Although less consistency guarantees are given in mode *committed* to the user, the costs of separate acquisitions and immediate releases of entire lock paths for each operation reduced the transaction throughput. Running short transactions, this overhead may not be amortized by higher concurrency.



(a) Short transactions (committed) (b) Streched run times (committed)

Fig. 5. Transaction throughput controlled by different isolation levels

After we had artificially increased the run times of the same transactions by programmed delays (in this way simulating human interaction), of course, the overall transaction throughput decreases, but meets the expectations of traditional transaction processing (see Fig. 5b): A lower degree of isolation leads to a higher transaction throughput. If few access conflicts are occurring (lock depths 2 and higher), lock management costs hardly influence the "stretched" transaction durations and transaction throughput is decoupled from the chosen isolation level. Hence, higher throughput on XML trees is not given for granted using isolation level *committed*. Surprisingly, *committed* seems to be inappropriate for large lock depths and short transactions.

3 Prefix-based Node Labeling is Indispensable

Range-based and prefix-based node labeling [2] are considered the prime competitive methods for implementation in XDBMSs. A comparison and evaluation of those schemes in [7] recommends prefix-based node labeling based on the Dewey Decimal Classification. Each label represents the path from the document's root to the related node and the local order w.r.t. the parent node. Some schemes such as OrdPaths [13], DeweyIDs, or DLNs [7] provide immutable labels by supporting an overflow technique for dynamically inserted nodes. Here, we use the generic name *stable path labeling identifiers* (SPLIDs) for them.

Whenever a node, e.g., with SPLID 1.19.7.5, has to be locked, all its ancestor node labels are needed for placing intention locks on the entire path up to the root. Hence, they can be automatically provided: 1.19.7, 1.19, and 1 for the example. Because such lock requests occur very frequently, the use of SPLIDs is *the key argument* for locking support.[3] Referring to our lock table (see Fig. 2), intention locks for the ancestors of 1.19.7.5 can be checked or newly created using hL(1.19.7), hL(1.19) and hL(1) without document access. Because of the frequency of this operation, we provide a function which acquires a lock and all necessary intention locks at a time. A second *important property* for stable lock management is the immutability of SPLIDs, i.e., they allow the assignment of new IDs without the need to reorganize the IDs of nodes present.

We have repeated document traversal using SPLID-based lock management (see Fig. 4b). Because the difference between *none* and *committed/repeatable* is caused by locking overhead, we see drastic performance gains compared to SEQIDs. While those are responsible for an up to ∼600% increase of the reconstruction times in our experiment, SPLIDs keep worst-case locking costs in the range of ∼10 – ∼20%. SEQIDs have fixed length, whereas SPLIDs require handling of variable-length entries. Coping with variable-length fields adds some complexity to SPLID and B*-tree management. Furthermore, prefix-compression of SPLIDs is a must [7]. Nevertheless, reconstruction time remained stable when SPLIDs were used – even when locking was turned off (case *none*).

[3] Range-based schemes [21] cause higher locking overhead than SPLIDs. They enable the parent label computation, but not those of further ancestors. An optimization would include a parent label index to compute that of the grandparent and so on.

Comparison of document reconstruction in Fig. 4a and b reveals for identical XML operations that the mere use of SPLIDs (instead of SEQIDs) improved the response times by a factor of up to 5 and more. This observation may convince the reader that node labeling is of utmost importance for XML processing. It is not only essential for internal navigation and set-based query processing, but, obviously, also most important for lock manager flexibility and performance.

4 Further Performance Drivers

Every improvement of the lock protocol shifts the issue of multi-user synchronization a bit more from the level of logical XML trees down to the underlying storage structures, which is a B*-tree in our case. Hence, an efficient and scalable B*-tree implementation in an *adjusted infrastructure* is mandatory.

D1: B*-tree Locking Our initial implementation revealed several concurrency weaknesses we had to remove. First, tree traversal locked all visited index pages to rely on a stable ancestor path in case of leaf page split or deletion. Thus, update operations lead to high contention. Further, the implemented page access protocol provoked deadlocks under some circumstances. Although page locking itself was done by applying normal locks of our generic lock manager, where deadlocks could be easily detected and resolved, they had a heavy effect on the overall system performance. Thus, we re-implemented our B*-tree to follow the ARIES protocol [12] for index structures, which is completely deadlock-free and can therefore use cheap latches (semaphores) instead of more expensive locks. Further, contention during tree traversals is reduced by *latch coupling*, where at most a parent page and one of its child pages are latched at the same time.

D2: Storage Manager Navigational performance is a crucial aspect of an XML engine. A B*-tree-based storage layout, however, suffers from indirect addressing of document nodes, because every navigation operation requires a full root-to-leaf traversal, which increases both computational overhead and page-level contention in the B*-tree. Fortunately, navigation operations have high locality in B*-tree leaves, i.e., a navigation step from a context node to a related node mostly succeeds in locating the record in the same leaf page. We exploit this property, by remembering the respective leaf page and its version number for nodes accessed as a hint for future operations. Each time when re-accessing the B*-tree for a navigation operation, we use this information to first locate the leaf page of the context node. Then, we quickly inspect the page to check if we can directly perform the navigation in it, i.e., if the record we are looking for is definitely bound to it. Only if this check fails, we have to perform a full root-to-leaf traversal of the index to find the correct leaf. Note, such an additional page access is also cheap in most cases, because the leaf page is likely to be found in the buffer due to locality of previous references.

D3: Buffer Manager As shown in [7], prefix-compression of SPLIDs is very effective to save storage space when representing XML documents in B*-trees. As with all compression techniques, however, the reduced disk I/O must be paid with higher costs for encoding and decoding of compressed records.

With page-wide prefix compression as in our case, only the first record in a page is guaranteed to be fully stored. The reconstruction of any subsequent entry potentially requires to decode all of its predecessors in the same page. Accordingly, many entries will have to be decoded over and over again, when a buffered page is frequently accessed. To avoid this unnecessary decoding overhead and to speed up record search in a page, we enabled buffer pages to carry a cache for already decoded entries. Using page latches, the page-local cache may be accessed by all transactions and does not need further considerations in multi-user environments. Although such a cache increases the actual memory footprint of a buffered disk page, it pays off when a page is accessed more than once – the usual case, e.g., during navigation. Further, it is a non-critical auxiliary structure that can be easily shrinked or dropped to reclaim main memory space.

A second group of optimizations was concerned with XML lock protocols for which empirical experiments identified *lock depth* as the most performance-critical parameter (see Sect. 1.1). Choosing lock depth 0 corresponds to document-only locks. In the average, growing lock depth refines lock granules, but enlarges administration overhead, because the number of locks to be managed increases. But, conflicting operations often occur at levels closer to the document root (at lower levels) such that fine-grained locks at levels deeper in the tree do not always pay off. A general reduction of the lock depth, however, would jeopardize the benefits of our tailored lock protocols.

D4: Dynamic Lock Depth Adjustment Obviously, optimal lock depth depends on document properties, workload characteristics, and other runtime parameters like multiprogramming level, etc., and has to be steadily controlled and adjusted at runtime. Therefore, we leveraged *lock escalation/deescalation* as the most effective solution: The fine-grained resolution of a lock protocol is – preferably in a step-wise manner – reduced by acquiring coarser lock granules (and could be reversed by setting finer locks, if the conflict situation changes). 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 separate node locks for each descendant visited. Hence, transactions initially use fine lock granules down to high lock depths to augment permeability in hot-spot regions, but lock depth is dynamically reduced when low-traffic regions are encountered to save system resources. Using empirically proven heuristics for conflict potential in subtrees, the simple formula $threshold = k * 2^{-level}$ delivered escalation thresholds, which takes into account that typically fanout and conflicts decrease with deeper levels. Parameter k is adjusted to current workload needs.

D5: Avoidance of Conversion Deadlocks Typically, deadlocks occurred when two transactions tried to concurrently append new fragments under a node already read-locked by both of them. Conversion to an exclusive lock involved both transactions in a deadlock. Update locks are designed to avoid such conversion deadlocks [6]. Tailored to relational systems, they allow for a direct upgrade to exclusive 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

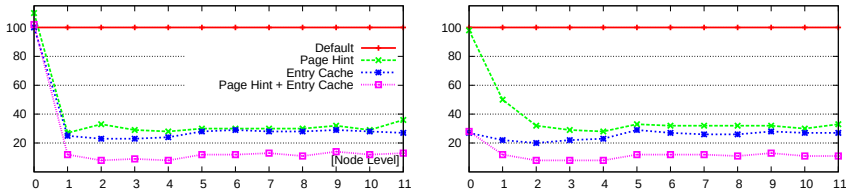
easy access patterns. Instead, they often perform arbitrary navigation steps in the document tree, e.g., to check the content child elements, before modifying a previously visited node. Hence, we carefully enriched our access plans with hints when to use update locks for node or subtree access.

5 Effects of Various Optimizations

We checked the effectivity of the infrastructure adjustments (D1, D2, and D3), before we focused on further XML protocol optimizations (D4 and D5). Based on our re-implemented B*-tree version with the ARIES protocol (D1), we verified the performance gain of navigation optimizations for D2 and D3. We stored an 8MB XMark document in a B*-tree with 8K pages and measured the average execution time of the dominating operations *FirstChild* and *NextSibling* during a depth-first document traversal. We separated the execution times for each document level, because locality is potentially higher at deeper levels.

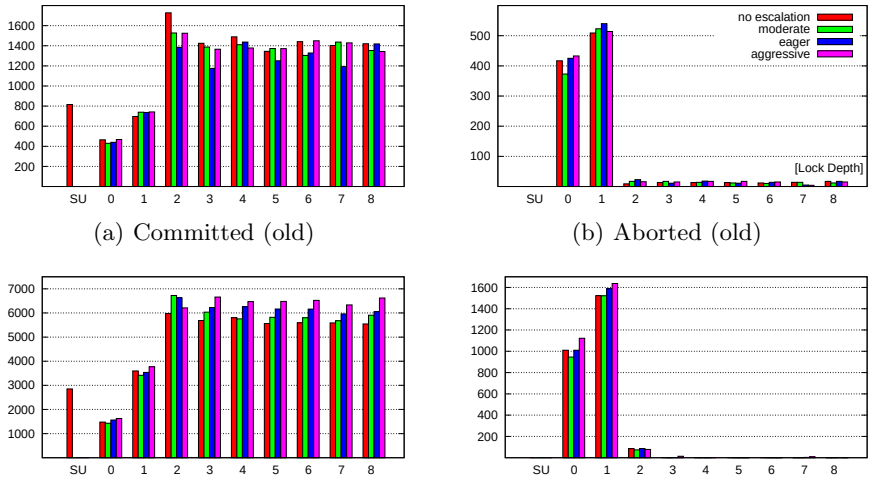
The results in Fig. 6a and b confirm that the optimizations of D2 and D3 help to accelerate navigation operations. With speed-ups of roughly 70% for all non-root nodes, the benefit of both is nearly the same for the *FirstChild* operation. The fact that the use of cached page entries results even in a slightly higher performance boost than the drastic reduction of B*-tree traversals through the use of page hints, underlines the severeness of repeated record decoding. The actual depth of the operation does not play a role here. In contrast, the page-hint optimization shows a stronger correlation to the depth of a context node. As expected, page hints are less valuable for the *NextSibling* operation at lower levels, because the probability that two siblings reside in the same leaf page is lower. For depths higher than 2, however, this effect completely disappears. For the whole traversal, the hit ratio of the page hints was 97.88%. With documents of other size and/or structure, we achieved comparable or even higher hit ratios. Even for a 100MB XMark document, e.g., we still obtained a global hit ratio of 93.34%. With the combination of both optimizations, we accomplished a performance gain in the order of a magnitude for both operation types.

To examine and stress-test the locking facilities with the lock protocol optimizations of D4 and D5, situations with a high blocking potential had to be provoked. We created mixes of read/write transactions, which access and modify a generated XMark document at varying levels and in different granules [1]. We again chose an initial document size of only 8 MB and used a buffer size large



(a) First-Child Operation (b) Next-Sibling Operation

Fig. 6. Relative execution times of navigation operations (%)



(a) Committed (old) (b) Aborted (old)

(c) Committed (new) (d) Aborted (new)

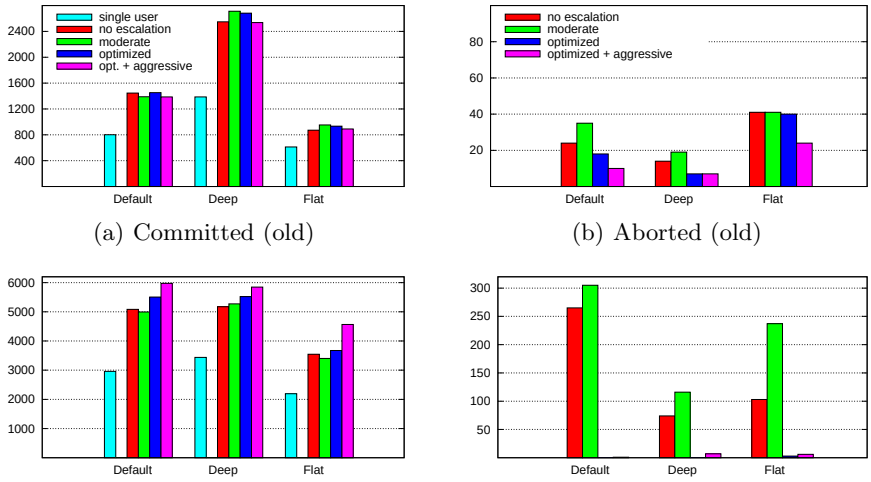
Fig. 7. Effects of lock depth and lock escalation on transaction throughput (tpm)

enough for the document and auxiliary data structures. Further details of the specific workloads are not important here, because we only aim at a differential performance diagnosis under identical runtime conditions. To get insight in the behavior of the lock-depth optimization D4, we measured the throughput of transactions per minute (tpm) and ran the experiments for three escalation thresholds (moderate, eager, aggressive) in single user mode (SU) and in multi-user mode with various initial lock depths (0–8).

To draw the complete picture and to reveal the dependencies to our other optimizations, we repeated the measurements with two XTC versions: XTC based on the old B*-tree implementation and XTC using the new B*-tree implementation together with the optimizations D2 and D3. To identify the performance gain caused by D1–D3, we focused on transaction throughput, i.e., commit and abort rates, and kept all other system parameters unchanged. Fig. 7 compares the experiment results. In single-user mode, the new version improves throughput by a factor of 3.5, which again highlights the effects of D2 and D3. The absence of deadlocks and the improved concurrency of the latch-coupling protocol in the B*-tree (D1) becomes visible in the multi-user measurements, where throughput speed-up even reaches a factor of 4 (Fig. 7a and c) and the abort rates almost disappear for lock depths > 2 (Fig. 7b and d).

Deadlocks induced by the old B*-tree protocol were also responsible for the fuzzy results of the dynamic lock depth adjustment (D4). With a deadlock-free B*-tree, throughput directly correlates with lock overhead saved and proves the benefit of escalation heuristics (Fig. 7c and d).

In a second experiment, we modified the weights of the transactions in the previously used mix to examine the robustness of the approach against shifts in



(a) Committed (old) (b) Aborted (old)

(c) Committed (new) (d) Aborted (new)

Fig. 8. Results of workload variations and adjusted escalation strategies (tpm)

the workload characteristics. In the default workload, the document was modified both in lower and deeper levels. In contrast, the focus in the two other workloads is on nodes at deeper (deep) and lower (flat) levels, respectively. Additionally, we provided optimized, update-aware variants of the transaction types to examine the effect of careful use of update locks. For simplicity, we ran the multi-user measurements only with initial lock depth 8.

The results in Fig. 8 generally confirm the observations of the previous experiment. But, throughput comparison between old and new B*-tree variants attest the new one a clearly better applicability for varying workloads. The value of update locks is observable both in throughput and in abort rates. The optimized workloads are almost completely free of node-level deadlocks, which directly pays off in higher throughput. Here, the performance penalty of page-level deadlocks in the old B*-tree becomes particularly obvious. Further, the results show that our lock protocol optimizations by the performance drivers D4 and D5 complement each other, similar to the infrastructure optimizations D1–D3.

6 Conclusions

In this paper, we outlined the implementation of XML locking, thereby showing that the taDOM family is perfectly eligible for fine-grained transaction isolation on XML document trees. We disclosed lock management overhead and emphasized the performance-critical role of node labeling, in particular, for acquiring intention locks on ancestor paths. In the course of lock protocol optimization, we have revealed the real performance drivers: adjusted measures in the system infrastructure and flexible options of the lock protocols to respond to the

workload characteristics present. All performance improvements were substantiated by numerous measurements in a real XDBMS and under identical runtime conditions which enabled performance comparisons of utmost accuracy – not reachable by comparing different systems or running simulations.

References

1. Bächle, S., Härder, T.: Implementing and Optimizing Fine-Granular Lock Management for XML Document Trees, in: Proc. DASFAA Conf., Brisbane (2009).
2. Christophides, W., Plexousakis, D., Scholl, M., Tourtounis, S.: On Labeling Schemes for the Semantic Web. In Proc. 12th Int. WWW Conf., 544-555 (2003).
3. Document Object Model (DOM) Level 2 / Level 3 Core Specification. W3C Recommendation (2004), http://www.w3.org/TR/DOM-Level-3-Core.
4. Grabs, T., Böhm, K., Schek, H.-J.: XMLTM: Efficient transaction management for XML documents. In Proc. CIKM Conf., 142-152 (2002).
5. Graefe, G.: Hierarchical locking in B-tree indexes, in: Proc. German Database Conference (BTW 2007), LNI P-65, 1842 (2007).
6. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann (1993).
7. Härder, T., Haustein, M. P., Mathis, C., Wagner, M.: Node Labeling Schemes for Dynamic XML Documents Reconsidered. Data & Knowl. Eng. 60(1): 126-149 (2007).
8. Haustein, M. P., Härder, T., Luttenberger, K.: Contest of XML Lock Protocols. In Proc. VLDB Conference, Seoul, 1069-1080 (2006).
9. Haustein, M. P., Härder, T.: Optimizing lock protocols for native XML processing. Data & Knowl. Eng. 65(1): 147-173 (2008).
10. Helmer, S., Kanne, C.-C., Moerkotte, G.: Evaluating Lock-Based Protocols for Cooperation on XML Documents. SIGMOD Record 33(1): 58-63 (2004).
11. Loeser, H., Nicola, M., Fitzgerald, J.: Index Challenges in Native XML Database systems. In Proc. German Database Conf. (BTW), LNI (2009).
12. Mohan, C.: ARIES/KVL: A key-value locking method for concurrency control of multiaction transactions operating on B-tree indexes. In Proc. VLDB Conf.: 392405 (1990).
13. O'Neil, P., O'Neil, E., Pal, S., Cseri, I., Schaller, G., Westbury, N.: OrdPaths: Insert-Friendly XML Node Labels. In Proc. SIGMOD Conf.: 903-908 (2004).
14. Pleshachkov, P., Chardin, P., Kusnetzov, S.: XDGL: XPath-based Concurrency Control Protocol for XML Data. In Proc. BNCOD, UK Bd. 3567, 145-154 (2005).
15. Sardar Z., Kemme, B.: Don't be a Pessimist: Use Snapshot based Concurrency Control for XML. In Proc. Int. Conf. on Data Engineering, 130 (2006).
16. Schmidt, A., Waas, F., Kersten, M. L., Carey, M. J., Manolescu, I., Busse, R.: XMark: A Benchmark for XML Data Management. In Proc. VLDB Conf.: 974-985 (2002).
17. Schöning, H.: Tamino – A DBMS designed for XML. In Proc. Int. Conf. on Data Engineering: 149-154 (2001).
18. Siirtola A., Valenta, M.: Verifying Parameterized taDOM+ Lock Managers. In Proc. SOFSEM Conf., LNCS 4910, 460-472 (2008).
19. XQuery Update Facility. http://www.w3.org/TR/xqupdate
20. XML on Wall Street, Financial XML Projects, http://lighthouse-partners.com/xml
21. Yu, J. X., Luo, D., Meng, X., Lu, H.: Dynamically Updating XML Data: Numbering Scheme Revisited. World Wide Web 8(1): 5-26 (2005)