Observations on Fine-grained Locking in XML DBMSs

Martin Hiller, Caetano Sauer, and Theo Härder

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

Abstract. Based on XTC, we have redesigned, reimplemented, and re-optimized BrackitDB, a native XML DBMS (XDBMS). Inspired by "optimal" concurrency gained on XTC using the taDOM protocol, we applied an XML benchmark on BrackitDB running now on a substantially different computer platform. We evaluated important concurrency control scenarios again using taDOM and compared them against an MGL-protocol. We report on experiments and discuss important observations w.r.t. fine-grained parallelism on XML documents.

1 Introduction

In the past, we have addressed—by designing, implementing, analyzing, optimizing, and adjusting an XDBMS prototype system called XTC (XML Transactional Coordinator)—all these issues indispensable for a full-fledged DBMS. To guarantee broad acceptance for our research, we provided 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 [5]. At that time, all vendors of XML(-enabled) DBMSs supported updates only at document granularity and, thus, could not manage highly dynamic XML documents, let alone achieve ambitious performance goals. For this reason, we primarily focused on locking mechanisms which could support fine-grained, concurrent, and transaction-safe document modifications in an efficient and effective way.

The outcome of this research was the *taDOM* family of complex lock protocols [7] tailor-made for fine-grained concurrency in XML structures and guaranteeing ACID-quality transaction serializability. Correctness of taDOM [13] and its superiority against about a dozen of competitor protocols were already experimentally verified using our (disk-based) XTC (XML Transaction Coordinator) [6]. Changes and developments of computer and processing architectures (e.g., multi-core processors or use of SSDs) also impact the efficacy of concurrency control mechanisms. Therefore, we want to review taDOM in a—compared to the study in [6]—substantially changed environment. Because computer architectures provided fast-growing memories in recent years, we want to emphasize this aspect—up to main-memory DBMS—in our experimental study. Furthermore, based on XTC, we have redeveloped our testbed system, called BrackitDB as a disk-based XDBMS [1, 3]. To gain some insight into the concurrency control

behavior of these reimplemented and improved system, we run an XML benchmark under various system configurations and parameter settings [8]. Because the conceivable parameter space is so huge, we can only provide observations on the behavior of important XML operations.

To build the discussion environment, we sketch the system architecture of BrackitDB, the efficacy of important taDOM concepts, and critical implementation issues in Sect. 2. In the following section, we describe the test database and the XML benchmark used for our evaluation, before we report on our measurements obtained and, in particular, the most important observations made in Sect. 4. Finally we summarize our results and conclude the study.

2 Environment of the Experimental Study

To facilitate the understanding of our study, we need at least some insight into the most important components influencing the concurrency control results.

2.1 Hierarchical DBMS Architecture

Using a hierarchical architecture model, we have implemented BrackitDB as a native XDBMS aiming at efficient and transaction-safe collaboration on shared documents or collection of documents. Hence, it provides fine-grained isolation and full crash recovery based on a native XML store and advanced indexing capabilities.

Fig. 1 gives a simplified overview of its architecture where its layers describe the major steps of dynamic abstraction from the storage up to the user interface.

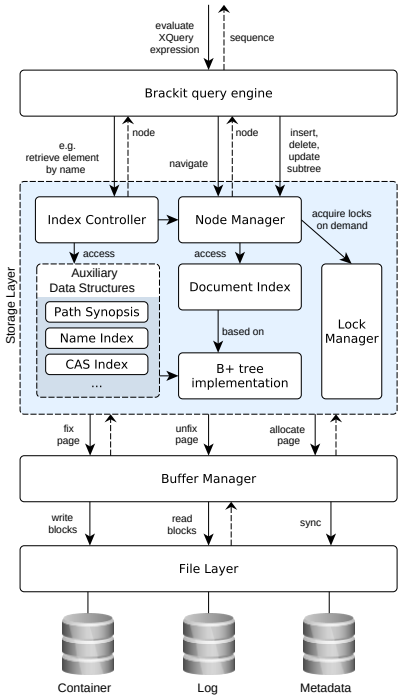


Fig. 1. Architecture of BrackitDB

File Layer, Buffer Management, and the file formats (Container, Log, Metadata)—taken from XTC— are not XML-specific and similar to their counterparts in relational DBMSs. The components of the Storage Layer are more important for our study. They manage XML documents and related index structures and provide node-oriented access which has to be isolated in close cooperation with the Lock Manager (see Sect. 2.3). The document index—a B*-tree where each individual XML node is stored as a data record in one of the tree's leaf pages—is the core structure. To identify nodes, a prefix-based labeling scheme is applied[1]. A

[1] The node labeling scheme is *the key to efficient and fine-grained XML processing* [5].

DeweyID, for example, 1.5.7 identifies a node at level 2, where its parent has DeweyID 1.5 at level 1 and its grandparent DeweyID 1 at level 0. The DeweyIDs serve as keys in the document index and easily allow to derive sibling and ancestor relations among the nodes. The latter is particularly important for locking; if, e.g., a node is accessed via an index, intention locks have to be acquired on the ancestor nodes first. Because DeweyIDs also serve as index references, they contain the ancestor path and, hence, intention locks can be set automatically.

Efficient XML processing requires the *Path Synopsis* which represents in a tiny memory-resident repository all different path classes of a document. Using a path synopsis, each existing path in the document can be indexed by the so-called *Path Class Reference* (or PCR). By storing the DeweyID together with the PCR as an index reference, the entire path of the indexed node to the document root can be reconstructed without document access. This technique also enables virtualization of the inner nodes, i.e., it is sufficient to store only the leaf nodes of an XML document [11]. Moreover, name index, content index, path index, and CAS index (content and structure)—all using the B*-tree as base structure—contribute to the efficiency of XML query processing.

The top layer in Fig. 1 is embodied by BrackitDB's XQuery engine which compiles and executes high-level XQuery expressions. Because we circumvent here the query engine to have more precise control over node operations and locking aspects, we do not need to discuss this engine.

2.2 Lock Concepts of taDOM

Due to its complexity (with 20 lock modes and sophisticated lock conversion rules [7]), taDOM cannot even be sketched here completely for comprehension. Based on some examples, we try to convince the reader of taDOM's concepts and their potential. For this reason, we visualize its very fine granularity and compare it to the well-known MGL-protocol (Multi-Granularity Locking) [4].

taDOM uses the MGL intention locks IR and IX, renames R, U, and X to SR, SU, and SX (subtree read, update, and exclusive). Furthermore, it introduces new lock modes for *single nodes* called NR (node read), NU (node update), and NX (node exclusive), and for *all siblings under a parent* called LR (level read). NR and LR 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 *cn*, because separate child nodes of *cn* may be exclusively locked by other transactions (compatibility is then decided on the child nodes). Altogether these new lock modes enable serializable schedules with read operations on inner tree nodes, while concurrent updates may occur in their subtrees. An important and unique feature is the optional variation of the *lock depth* which can be dynamically controlled by a parameter.

Lock depth n determines that, while navigating in documents, individual locks are acquired for existing nodes down to level n. If necessary, nodes below level n are locked by a subtree lock (SR, SX) at level n. Fig. 2 summarizes the lock compatibilities among taDOM's core modes.

Fig. 2. Core modes of taDOM

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

Fig. 3 highlights taDOM's flexibility and tailor-made adaptations to XML documents as compared MGL. 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 an R lock on *cn*, which forces a concurrent writer (*T2* in Fig. 3(a)) 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. 3(b, c)). As the only lock granule, however, node locks would result in excessive lock management cost and catastrophic performance behavior, especially for subtree deletion [6]. Scanning of *cn* and all its children could be done node by node in a navigational fashion (Fig. 3(b)). A special taDOM optimization—using a tailor-made LR lock for this frequent read scenario—enables stream-based processing and drastically reduces locking overhead; in huge trees, e.g., the DBLP document, a node may have millions of children. As sketched in Fig. 3(c), LR also 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.

Document entry from any secondary index [4] involves the risk of phantoms. For instance, retrieving a list of element nodes from the *name index* [11] causes every returned node to be NR-locked. While this protects the elements from being renamed or deleted, other transactions might still insert new elements with matching names. Hence, opening the name index again at a later time with the same query parameters (i.e., given element name), might fetch an extended list of element nodes, including phantom elements that have been inserted concurrently. To prevent this kind of anomalies, BrackitDB implements Key-Value Locking (or KVL [12]) for all B*-trees employed as secondary indexes. In a nutshell, KVL aims not only for locking single index entries but also protects key ranges traversed by index scans. Therefore, KVL effectively prevents phantoms.

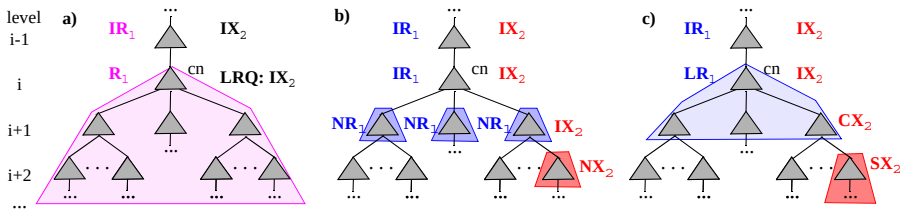


Fig. 3. Locking flexibility and effectivity: MGL (a) vs. taDOM (b, c)

2.3 Lock Management

Additional flexibility comes from dynamic lock-depth variations and *lock escalation*, which help to find an adequate balance between lock overhead (number of locks managed) and parallelism achieved (concurrency-enabling effects of chosen lock granules). An illustrative example for this locking trade-off is given above: *T1* started to navigate the child set of node *cn* in a low-traffic subtree (Fig. 3(b)). If no interfering transaction is present, lock escalation could be performed (Fig. 3(c)) to reduce lock management cost. In BrackitDB, two kinds of lock escalation are distinguished. Using *static lock depth*, locks are acquired only up to a pre-specified level n in the document tree. If a node is accessed at a deeper level, its corresponding ancestor will be locked instead. Keep in mind that escalating a lock request to an ancestor potentially widens the lock mode as well. *Dynamic lock escalation* works at runtime to overcome the inflexibility of the static approach which handles each subtree in the same way. Some parts of the document might not even be accessed at level 1, while others might accommodate hotspot regions deeper than level n. Tracking locality of nodes to collect "escalation" information is cheap, because ancestor locks have to be acquired anyway. While performing intention locking, an internal lock request counter is incremented for the parent of requested nodes, indicating that one of its children is accessed. The following heuristics determines the escalation threshold:

$$threshold = \frac{maxEscalationCount}{2^{level} * escalationGain}$$

The deeper a node is located in the document, the lower is the threshold for performing lock escalation. The parameter from the numerator, `maxEscalationCount`, constitutes an absolute basis for the number of locks that need to be held at most before escalation is applied. This value is not changed throughout different escalation strategies, whereas the denominator parameter, `escalationGain`, affects how the threshold changes from one level to the next, i.e., the higher this value is set, the more aggressive is the escalation policy for deeper nodes. Throughout the benchmark runs, the following escalation policies are applied:

	maxEscalationCount	escalationGain
moderate	1920	1.0
eager	1920	1.4
aggressive	1920	2.0

The concept of *meta-locking* realized in BrackitDB provides the flexibility to exchange lock protocols at runtime. Hence, such dynamic adaptations are a prerequisite to achieve workload-dependent optimization of concurrency control and to finally reach autonomic tuning of multi-user transaction processing [6]. This concept enables us to execute all benchmark runs under differing lock protocols in an *identical system environment*.

Lock management internals are encapsulated in so-called lock services with a lock table as their most important data structure. Lock services provide a tailored interface to various system components, e.g., for DB buffer management

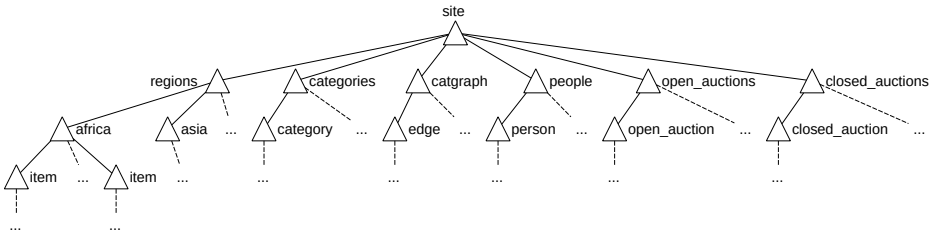


Fig. 4. Fragment of the XML document used in the benchmark

[7]. A lock table must not be a bottleneck for performance and scalability reasons. Therefore, it has to be traversed and modified by a multitude of transaction threads at a time. Hence, preservation of its physical consistency becomes a major challenge for an implementation viewpoint. Furthermore, frequent blocking situations must be avoided when lock table operations (look-up, insertion of entries) or house-keeping operations are performed. Therefore, the use of latches on individual entries of a hash table is mandatory for lock table access. As compared to a single monitor for a hash table, such a solution avoids hot spots and guarantees physical consistency under concurrent thread access [4].

Another major component of BrackitDB's lock management is the *deadlock detector (DD)*. Our DD component runs periodically in a separate thread so that deadlock resolution is still possible, even when all transaction workers are already blocked. Every time the DD thread wakes up, its task is to crawl through the lock table and to construct a *wait-for* graph on its way. While building the wait-for graph, the DD thread has to comply with the same latching protocol as every other thread accessing the lock table. If a cycle is detected, one of the participating transactions is to be aborted. The decision heuristics for a suitable abort candidate is based on the minimal number of locks obtained.

3 Benchmark Document and Workload

The benchmark data (see Fig. 4) was created by the XMark Benchmark Data Generator. Its command line tool `xmlgen` produces documents that model auction data in a typical e-commerce scenario. Words for text paragraphs are randomly picked from Shakespeare's plays so that character distribution is fairly representative for the English language. Further, the XML output includes referential constraints across the document through respective ID/IDREF pairs, thus providing a basis for secondary index scenarios.

But the most outstanding feature is arguably the *scaling factor*. Varying this factor allows for generating documents from the KB range up to several GBs. The documents still preserve their characteristics under scaling so that bottlenecks found on smaller documents would apply for larger documents proportion-

Items:	217,500
Categories:	10,000
Catgraph edges:	10,000
Persons:	250,000
Open auctions:	120,000
Closed auctions:	97,500

Fig. 5. Cardinalities

ally. It is worth noting that scaling works only horizontally, where document depth and complexity remains untouched, while scaling appends new elements of different types w.r.t. a probability distribution. The benchmark is based on an XMark document with a scaling factor of 10, resulting in a file of approximately 1.1 GB in size (stored on HDD) and specific document cardinalities (see Fig. 5).

The XMark document models an auction application where *items* coming from a certain *region* are put up for auction. Each item is thereby associated with one or more *categories*, which is realized by a sequence of child elements referring to category elements by their respective ID. Furthermore, every item contains a *mailbox* element surrounding a list of *mails*. Following on the item declarations, the *categories* element accumulates the available item categories in the system which are simply characterized by a name and a description text. The *catgraph* element defines a graph structure on top of the categories, whereas each *edge* ties two categories together. The *person* elements gathered under the *people* element act as users of the virtual auction platform. A few personal information like the name or the email address are exposed at this place, but their main usage is to be referenced by *auction* elements as bidders or sellers, respectively. In terms of auctions, XMark distinguishes between open and closed auctions and store them separately as either *open_auction* or *closed_auction* element in the corresponding subtree. But in fact, their XML structure is quite similar. Both kinds of auctions contain a description, a reference to the traded item and its seller.

The workload that is putting stress on BrackitDB is a mix of eight different transactions. Some of these are read-only, others perform both read and write accesses. Moreover, transactions are picked randomly from this pool according to predefined weights (see Table 1), which are supposed to reflect a distribution we might find in a realistic auction-based application where the focus of activity is certainly on placing bids. Here, we can sketch the workload transactions (TX) only by their names to give some flavor of the benchmark evaluated.

The transactions need a way to randomly jump into the document to begin their processing. For instance, placing a bid requires the transaction to pick a random auction element, while another operation might start on a random person element as context node. Another feature needed by some operations is the possibility to follow ID references within the document, e.g., jump from the auction to the corresponding item element. Although both requirements (random entry and navigation via references) could be met without utilizing any secondary indexes by accessing the root node and scanning for the desired node, system performance could drastically suffer from this unfavorable access plan. For that reason, we provide two secondary indexes:

- A name index maps element names to a set of DeweyIDs enabling node reconstructions or direct document access to the name occurrences.
- A CAS index [11] allows for content-based node look-ups, i.e., attribute and text nodes can be retrieved based on their string value.

When fetching random elements from the name index, the elements are selected w.r.t. some predefined *skew*. Locking-related contention must also be arti-

Table 1. Mix of transactions with their types and weights

Transaction	*Type*	*Weight*	*Transaction*	*Type*	*Weight*
Place Bid	r-w	9	Check Mails	r-o	7
Read Seller Info	r-o	4	Read Item	r-o	5
Register	r-w	1	Add Mail	r-w	4
Change User Info	r-w	4	Add Item	r-w	1

ficially enforced. Higher skew implies denser access patterns and increased contention. Skew is defined as $1-\sigma$, while σ denotes the relative standard deviation of the normal distributions. Hence, a skew of 99% (or a relative σ of 1%) implies that only 1% of the records in the page are picked with a probability of approximately 68%, effectively producing a hotspot.

Note, we report numbers (tpm, ms) for each experiment averaged over > 10 runs. Transaction order varied from run to run due to random elements fetched and, more influential, randomly picked transactions. Hence, distribution of locks and, in turn, blocking delays might have strongly differed from run to run.

4 Measurements

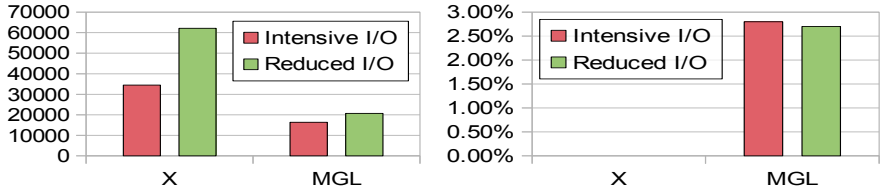
All benchmark runs performed were executed in the following runtime environment. Note that benchmark suite (BenchSuite) as well as BrackitDB server were both running concurrently on the same node during the measurements.

CPU: 2x Intel Xeon E5420 @ 2.5 GHz, totaling in 8 cores
Memory: 16 GB DDR2 @ 667 MHz
Storage: RAID-5 with 3 disks (WD1002FBYS), 1 TB each, 7200 rpm (HDD)
Log: Samsung SSD 840 PRO, 256 GB (SSD)
Operating System: Ubuntu 13.04 (GNU/Linux 3.8.0-35 x86-64)
JVM: Java version 1.7.0_51, OpenJDK Runtime Environment (IcedTea 2.4.4)

4.1 System Parameters

We need to sketch the system parameters (not discussed so far) which strongly influence the benchmark results. As buffer size, we distinguish two corner cases: A *small buffer* keeps 1% of the XMark document incl. secondary indexes, thereby provoking lots of page replacements to HDD. In turn, a *large buffer* holds the entire document in main memory; hence, I/O is only needed for logging. Every experiment starts on a warm buffer.

We run *ACID* transactions, where BrackitDB has to perform ARIES-style logging and recovery [4] based on *HDD* or *SSD*. To reveal special effects, logging (and, in turn, ACID) may be *disabled*. The skew parameter was set to 50% (*low*) or 99% (*high*). The *number of threads* was always 8. Scheduling is insofar *optimal* that a transaction is only initiated, if a worker thread is available. Hence, no queuing effects occur, apart from blocking delays in front of the Lock Manager. Note, the assignment of threads to processing cores is encapsulated by the OS, such that applications (e.g., DBMS) cannot exert influence.



(a) TX throughput (tpm) (b) TX abort rate

Fig. 6. Baseline experiments for X- and MGL-protocols

4.2 Baseline Experiments

We started running the same XMark workload under pure X- and MGL-protocols to reveal especially the influence of buffer size or I/O activity and lock management overhead. To obtain the largest spread of transaction throughput, configurations with small buffer/low skew/log on HDD (Intensive I/O) and large buffer/high skew/log on SSD (Reduced I/O) were chosen (Fig. 6). Reducing I/O to the minimum necessary more than doubled throughput in case of the X-locking. It always exclusively locked the entire document—only a single worker thread was accessing the document at a time, while all others waited for the global X-lock to be released—and thereby prevented lock conflicts (no aborts), whereas MGL handled 8 threads concurrently. Lock conflicts together with multi-threading [14] obviously led to substantial throughput loss. For *Reduced I/O*, MGL only achieved one third of the X-lock throughput, i.e., throughput of serial workload executions profits much more from a larger DB buffer than that of parallel executions. Further, up to 3% of the transactions had to be aborted.

The result of Fig. 6 is seductive and points to *degree of parallelism* = 1 as the seemingly best case. But Fig. 7 reveals the downside of this approach. The response times dramatically grow even in case of very short transactions, although optimal scheduling was provided. A mix of long and short transactions would make *parallelism* = 1 definitely unacceptable for most applications.

TX timings in Fig. 7 are avg. times over the benchmark—all runs of all transaction types. *Block time* is the aggregated wait time for a transaction in front of the Lock Manager for lock releases. *Lock request time* containing the block time summarizes all requests for document and index locks including latch and lock table processing.

The difference between both timings (right and middle bars in Fig. 7 confirms that the Lock Manager overhead is very low (∼0.1 ms)—an indication of its salient implementation. Fig. 7 further reveals the critical role of I/O on the *total runtime*—even in the case of MGL. Analyzing the deviation between the total transac-

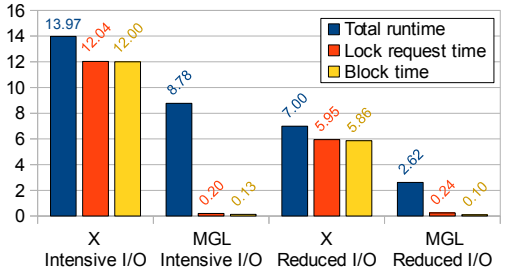


Fig. 7. TX timings (ms)

tion runtime (left bar) and the accumulated lock request time (middle bar) makes clear that the non-locking-related overhead is strongly dependent on I/O. This overhead includes TCP interaction between BenchSuite and BrackitDB, look-ups in meta-data, , accessing storage and evaluating navigation steps, logging every modification, fixing pages at buffer level and much more—these fixed costs make up a major portion of the total query runtime. In case of X-locking, only (parts of) communication and look-ups can be done in parallel leaving the major share of non-locking-related work for serial execution, whereas, in case of MGL, all work (except latch synchronization) could be concurrently done in all threads making the I/O dependency even stronger. Based on X-locking (mostly serial execution), we can approximately calculate the the total query runtime from Fig. 6(a): Intensive I/O: $\sim 33,000$ tpm $= \sim 550$ tps) $\equiv 1.9$ ms/TX and Reduced I/O: $\sim 62,000$ tpm $= \sim 1,030$ tps) $\equiv 0.97$ ms/TX. These values are confirmed by the corresponding TX timings in Fig. 7. In contrast, waits for I/O can be partially masked in MGL runtimes, however, latch waits (for buffer pages and lock table), lock conflicts, and multi-threading loss cannot be fully amortized.

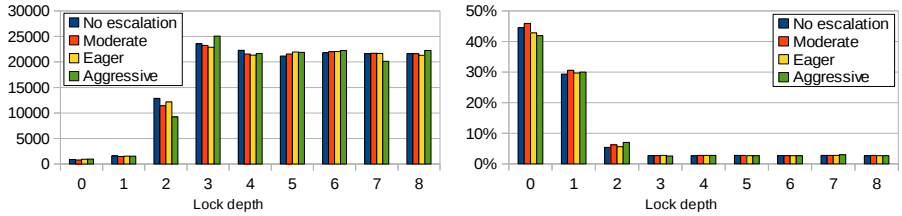
Note, due to our "optimal" BenchSuite scheduling, queuing delays before TX initiation are not considered in the given TX timings. For user-perceived *response times*, we have to add a variable time component for queuing, which is dependent on traffic density of TX requests and TX execution time variance. Similar TX timings as shown for the MGL-protocol were obtained in the subsequent experiments, such that we will not repeat this kind of figure.

4.3 Lock Depth and Lock Escalation

Because taDOM acts as an extension to MGL, providing the same set of core lock modes plus some new taDOM-specific locks, we can expect this protocol to behave similar to MGL. Because we want to focus on lock conflicts, we have chosen a configuration with large buffer and high skew.

The first observation we can make from examining Fig. 8(a) is the huge impact on the transaction throughput when varying the lock depth in the range between 0 and 3. While a lock depth of 0 basically escalates every lock request to the root node and thereby achieves a dreadful result in terms of system performance, taDOM seems to reach its climax when restricting the lock depth to 3. In fact, this pattern could be well explained by the XMark document structure and the workload. Increasing throughput values for lock depths up to 3 indicate that most transactions clash at the fourth level of the XML tree, where most of the document modifications take place. If the focus of transaction contention is not deeper in the XML tree, higher (static) lock depths only increase the lock management overhead. Fig. 8(b) is somewhat complementary to Fig. 8(a). High locking contention poses a higher risk of deadlocks which again leads to high abort rates and eventually a drastically reduced transaction throughput.

As compared to MGL (with $\sim 20,000$ tpm in Fig. 6(a)), taDOM variations are constantly superior and achieve an throughput gain of up to 20%. In this benchmark, the lock escalation policy applied seems to have only limited effect on throughput or abort rate whatsoever. A lock escalation count essentially tells



(a) TX throughput (tpm) (b) TX abort rate

Fig. 8. taDOM with variation of lock depth and lock escalation policy

us that only escalation "Aggressive" leads to actually performed lock escalations and exhibits some throughput gain of $\sim 10\%$ at lock depth 3. In this experiment, the conditions it takes for escalating were hardly satisfied for weaker escalation policies. Hence, *No escalation*, *Moderate* and *Eager* are not only coincidentally producing similar results, but they literally constitute the same scenario. On the other hand, scenarios with varying high-traffic contention in subtrees could certainly profit from a well-chosen escalation policy. Because escalation checking did hardly increase lock management overhead, the proposed policies should be offered as valuable optimization options for fine-grained XML lock protocols.

Although not comparable one-to-one, we obtained throughput values (tpm) for BrackitDB's SSD-based benchmarks that are up to 100% higher than the former HDD-based benchmarks using XTC ([6, 2]. Due to space limits, we abstain from reporting our experiment using a small DB buffer, i.e., enforcing *Intensive I/O* with a small buffer, because it did not exhibit new aspects.

4.4 Workload Variants

To figure out whether there are still optimizations for taDOM left, we prepared different implementations of the same XMark transactions. Semantically, these workload variants are equivalent, but they utilize slightly different node operations to fulfill their task, or explicitly acquire locks here and there to reduce the risk of deadlocks caused by lock conversions. The following experiment is centered around XML processing options sketched in Fig. 3(b and c). We run BrackitDB with *basic taDOM* (infinite lock depth and lock escalation disabled), where a *large buffer/high skew* configuration is used to challenge the lock protocol (as in Sect. 4.3). The study includes four different workload variants:

Navigation: Navigational child access and no preventive locking
Stream: Stream-based child access and no preventive locking
Navigation+Preventive: Navigational child access with preventive locking
Stream+Preventive: Stream-based child access with preventive locking.

The first workload variant, where the transactions always perform navigation steps like `First-Child` and `Next-Sibling` whenever child traversal is necessary, is the same as used in the experiment in Sect. 4.3. The other alternative is to call `getChildren()` to retrieve a child stream using a single LR lock on the

parent node. Although stream-based access might sound like the clear winner, there are situations where single navigation steps make more sense, e.g., when only a few of the first children are actually accessed. If indeed deadlocks were the limiting factors for transaction throughput in the previous benchmark, we should also notice a difference when applying *preventive locking*, since this strategy should drastically reduce deadlocks or even prevent them altogether. Preventive locking is a strategy where the strongest necessary lock mode is obtained before any particular node is accessed for the first time; thus, lock upgrades as the major source of deadlocks are avoided. It is clear that such a proceeding needs substantial application knowledge. Because automatic conversion is currently out of reach, we did it manually per transaction type. Hence, the results obtained are somewhat "fictitious", marking future optimization potential.

The navigational workload achieves a throughput of slightly more than 20,000 tpm. Modifying the workload implementation by substituting single navigation steps by child streams (*stream-based*) does not have a striking impact on performance either (see Fig. 9). The results for the non-preventive variants are pretty stable. In fact, the stream-based strategy consistently exposes lower lock request times throughout all benchmark repetitions. Also the total query runtime turns out to be lower if child streams are utilized instead of navigation steps, which might indicate a slight performance benefit resulting from storage-related aspects (e.g., more efficient sibling traversal).

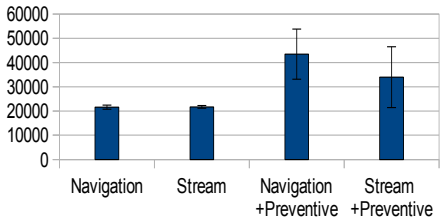


Fig. 9. Different workload variants

What improves the throughput significantly, however, is reducing the risk of deadlocks, achieved by application-specific locking optimizations in the workload variants with preventive locking. As a result, the abort rate could be brought down from 2.8% (non-preventive variants) to literally 0%. In case of the navigational approach, the *preventive* locking strategy roughly doubles the throughput, while it also improves the stream-based implementation considerably, yet not to the same extent. The higher throughput of *Navigation + Preventive* might result from frequent, incomplete child-set traversals. The reason why this diagram displays an additional error bar is to emphasize the high standard deviation for the two measurements involving preventive locking. No matter how often the experiment was repeated, these high fluctuations in the transaction throughput remain.

4.5 I/O Edge Cases

Eventually, due to the strong throughput dependency on the available I/O configuration, we want to review the four different workload variants employing taDOM under extreme I/O setups. So far, we gained the high-skew throughput results in Fig. 9 for "Reduced I/O" (I/O only for log flushes to SSD). Now, we look at the two edge cases concerning I/O activity. *Full I/O* is obtained by a tiny buffer covering only 0.1% of the document—enforcing very frequent page

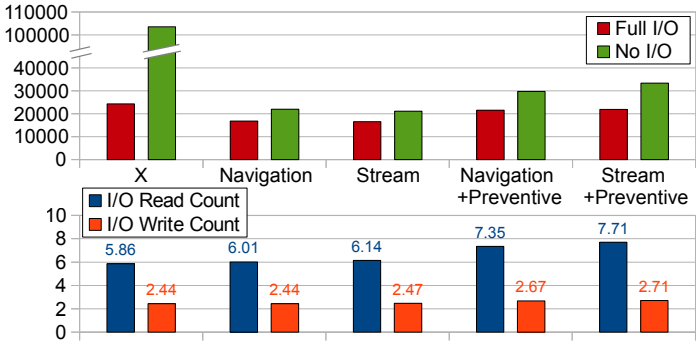


Fig. 10. TX throughput (tpm) for the I/O edge cases and I/O counts for *Full I/O*

replacements to HDD—, rather slow log flushes to HDD, and low-skew access of the benchmark transactions. *No I/O*, in contrast, uses a large buffer (no replacements needed) and disables log flushes completely. Note, this edge case is achievable in the near future, when persistent *non-volatile RAM (NVRAM)* (enabling log writes in the sub-μsec range [10]) can replace part of DRAM.

With a minimum of DB buffer size, the avg. number of I/O events per transaction for *Full I/O* is similar in all experiments. Note, *No I/O* has neither read- nor write-I/O. The X-case with its throughput spectrum from ∼23,000 tpm to >100,000 tpm is only present in Fig. 10 to highlight again the critical role of I/O, at least for serial execution. In the other experiments, based on multi-threaded processing under taDOM, the I/O impact is obvious, but far less critical. Obviously, the only moderate throughput gain by reduced I/O is a consequence of processing frictions caused by multi-threading and latch/lock/deadlock conflicts. Hence, there is future optimization potential for multi-core scheduling in the OS domain and for adjusted concurrency control in the DBMS domain.

5 Conclusions

Although seductive and dramatically simplifying DB work, we don't believe that *parallelism* = 1 is *generally acceptable* in DBMSs, because of potentially long and unpredictable response times. Stonebraker et al. [14] claim that 90% of the runtime resources are consumed by four sources of overhead, namely buffer pool, multi-threading, record-level locking, and an ARIES-style [4] write-ahead log. Even if this is correct, most applications cannot tolerate the consequences. Refinements of the underlying idea postulate *partition-wise serial execution*. Again, Larson et al. [9] showed that most transactions would frequently need expensive partition-crossing operations, making such an approach obsolete.

As a general remark, frequency of I/O events and their duration is critical for high-performance transaction processing. Large main memories and use of SSDs (in near future also NVRAM)—at least for logging tasks—help to substantially increase the level of concurrency and throughput. Yet, multi-threading and lock

conflicts diminish this gain considerably, as all result figures confirm. Because scheduling and multi-core mapping are OS tasks, improvements can not be obtained by the DBMS alone—an application program from the OS perspective.

With appropriately chosen taDOM options, we observed in our study a throughput gain of $\sim$20% compared to the widely used MGL-protocol. Optimal lock depth values strongly depend on document and workload; if wrongly chosen, they may have a suboptimal impact on concurrency. Well-chosen (or unlimited) lock depth combined with lock escalation may be a better solution with acceptable costs, because it dynamically cares for a balance of appropriate concurrency-enabling lock granules and lock management overhead. As a rule of thumb, setting the lock depth value *too high is much better than too low.*

Using application knowledge, processing XML documents could be significantly improved thereby reducing the risk of deadlocks. Scanning the child set of a node is a frequent operation, where—as a kind of context knowledge—preventive locking can be applied. As a result, it brought the abort rate down to 0% and, in turn, substantially enhanced the transaction throughput.

References

1. Bächle, S., Sauer, C.: Unleashing XQuery for Data-Independent Programming. Datenbank-Spektrum 14(2): 135-150, Springer (2014)
2. Bächle, S., Härder, T.: The Real Performance Drivers Behind XML Lock Protocols. In Proc. DEXA, LNCS 5690, 38-52, Springer (2009)
3. BrackitDB – Google Project Hosting. `http://code.google.com/p/brackit/wiki/BrackitDB` (2014)
4. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann (1993)
5. Haustein, M. P., Härder, T.: An Efficient Infrastructure for Native Transactional XML Processing. Data & Knowl. Eng. 65(1): 147-173, Elsevier (2008)
6. Haustein, M. P., Härder, T., and Luttenberger, K.: Contest of XML Lock Protocols. In Proc. VLDB, 1069-1080 (2006)
7. Haustein, M. P., Härder, T.: Optimizing Lock Protocols for Native XML Processing. Data & Knowl. Eng. 65(1): 147-173, Elsevier (2008)
8. Hiller, M.: Evaluation of Fine-grained Locking in XML Databases. Master Thesis, University of Kaiserslautern (2014)
9. Larson, P.-A. et al.: High-Performance Concurrency Control Mechanisms for Main-Memory Databases. PVLDB 5(4): 298-309 (2011)
10. Pelley, S. et al.: Storage Management in the NVRAM Era. PVLDB 7(2): 121-132 (2013)
11. Mathis, C., Härder, T., Schmidt, K.: Storing and Indexing XML Documents Upside Down. Computer Science - R&D, 24(1-2): 51-68 Springer (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, 392-405 (1990)
13. Siirtola, A., Valenta, M.: Verifying Parameterized taDOM+ Lock Managers. In Proc. SOFSEM, LNCS 4910, 460-472 (2008)
14. Stonebraker, M., Weisberg, A.: The VoltDB Main Memory DBMS. IEEE Data Eng. Bull. 36(2): 21-27 (2013)