Contest of XML Lock Protocols

Michael Haustein, Theo Härder, Konstantin Luttenberger
University of Kaiserslautern
P.O. Box 3049, 67653 Kaiserslautern, Germany
{haustein, haerder, luttenberger}@informatik.uni-kl.de

Abstract

We explore and compare the performance behavior of lock protocols to be used in XML DBMSs (XDBMSs, for short) supporting typical XML document processing (XDP) interfaces. In this paper, we outline 11 protocols proposed in the literature, highlight essential implementation concepts of our XDBMS and realize all of them in the same DBMS environment using so-called meta-synchronization. We design a framework for XML benchmarks including read and update transactions, run extensive empirical experiments which focus on the locking performance, and compare the results using various performance metrics. As a consequence, we can propose a group of protocols which won this practical contest under identical conditions.

1 Motivation

As XML documents permeate information systems and databases with increasing pace, they are more and more used in a collaborative way. If you run today an experiment on existing DBMSs with collaborative XML documents, you may experience a "performance catastrophe" meaning that most transactional operations are processed in strict serial order. The challenge for database system development is to provide adequate and fine-grained management for these documents enabling efficient and concurrent read and write operations. Therefore, future XML DBMSs will be judged according to their ability to achieve high transaction parallelism.[1]

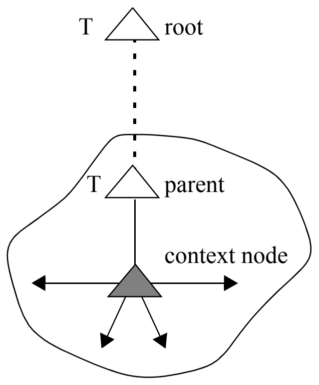
Currently, navigational and declarative language models are used to process XML documents. Because they are already available in the form of standards like SAX, DOM, XPath, and XQuery [21] and used as typical XDP interfaces, XDBMSs should be able to run concurrent transactions supporting all these interfaces simultaneously and, at the same time, guarantee ACID properties [9] for all of them.

Although predicate locking of XQuery statements [22]—and, in the near future, XUpdate-like statements—would be powerful and elegant, its implementation rapidly leads to severe drawbacks such as undecidability problems and the need to acquire large lock granules for simplified predicates—a lesson learned from the (much simpler) relational world. To provide for an acceptable solution, we necessarily have to map XQuery operations to a navigational access model to accomplish fine-granular concurrency control. Such an approach implicitly supports other XDP interfaces mentioned because their operations correspond more or less directly to a navigational access model. Our primary goal is to identify, compare, and evaluate XML concurrency control protocols being most suitable for the task outlined.

A survey of the (hardly) existing literature on locking methods tailored to fine-grained read and write operations on XML documents disclosed some unfit concepts [5, 6]. The remaining approaches led us to essentially three groups of protocols whose performance will be explored and compared in an extensive empirical study. The first group of protocols was developed in the context of the Natix system [14], the second one is a straightforward extension of the well-known protocols for multi-granularity locking (MGL) in classical (e.g., relational) DBMSs [9], whereas we have designed the remaining protocols for our native XDBMS called XTC (XML Transaction Coordinator). These protocols called the taDOM* group were steadily refined along with its progress and the experience gained from performance measurements [10].

In this paper, Sect. 2 gives a coarse characterization of these groups of protocols representing 11 individual concurrency control methods. The aspects of our testbed system XTC essential for an empirical study are outlined in Sect. 3, before we describe our experimental setting including the framework TaMix for XML benchmarks and the context of the measurements. In Sect. 5, extensive comparisons and interpretations of the performance results reveal the strengths and weaknesses of the protocol groups and allows to name a group of protocols superior to the competitors. Finally, we wrap up with conclusions.

1. A growing number of application developers believe XML and XQuery should be treated as primary data structure and access pattern [8].



structure locks

	T	M
T	+	-
M	-	-

locks for direct jumps

	IDR	IDX
IDR	+	-
IDX	-	-

content locks

	S	X
S	+	-
X	-	-

Fig. 1 Node2PL sample protocol and compatibilities for the different lock types

2 Approaches to XML Concurrency Control

Due to space limitations, it is impossible to describe all 11 rather complex XML concurrency control protocols in detail. Instead, we content ourselves with brief sketches of the main ideas behind the protocols and refer the careful reader to the original publications. All protocols are designed to achieve isolation level *repeatable read* for concurrent operations on XML trees.[2] For this reason, they not only have to lock the nodes either accessed by navigation or by direct jumps, e.g., via getElementByID(), but they also have to automatically protect their ancestor path by adequate means (typically intention locks). Furthermore, they have to isolate the edges traversed to guarantee identical navigation paths on repeated traversals. Navigational steps are issued by DOM operations getFirstChild(), getLastChild(), getPreviousSibling(), and getNextSibling(). To protect each of these operations, a corresponding logical navigation edge (which may not correspond to a physical edge in the tree representation) has to be introduced, for example, by locking the adjacent nodes.

2.1 Node2PL and its Followers

Let us begin with the first group denoted *-2PL. The primary objective of Node2PL is synchronization of transactions concurrently performing navigation and modification operations on the document tree. Starting from the document root, so-called structure locks are used in Node2PL to appropriately lock the parent (typically an element node) of the context node to which the navigation or update operation is applied—as illustrated in Fig. 1, the assumed read navigation to the context node leaves T (traverse) locks on its path from the root.

2. Isolation level *serializable* is provided by the taDOM* group, but is not used in our experiments to enable comparison with the remaining protocols which don't support this isolation level. Declarative index-based access to XML documents would need some kind of key range locking to prevent phantoms.

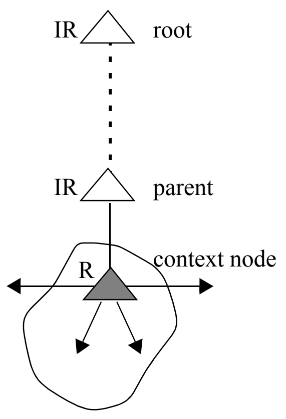
Furthermore, Node2PL strictly distinguishes structure-based and content-based accesses using different lock types. Hence, to change a node's content (e.g., of a text node), so-called content locks are used. In addition, a third lock type is introduced to protect direct jumps to nodes. A transaction directly jumping to a node addressed by an ID attribute acquires for it a special read or write lock (IDR, IDX). If the related subtree is to be deleted, IDX locks on all elements owning ID attributes must guarantee that no other transaction also jumped into this subtree reads or updates it. As we will see, this may imply a very expensive procedure. Such a penalty is especially performance-critical, because direct jumps may be rather frequent, for example, if query processing uses indexes.

Note, when the context node in the example of figure 1 is to be updated later, lock conversion (to the M (modify) mode) on the parent node is mandatory. Such conversions are a source of deadlocks in all protocols; this danger may only be alleviated by tailored intention locks. For the details of lock conversion, we refer to [14].

Node2PL is unnecessarily restrictive because, by locking the parent, it blocks the entire level of the context node, and not only its direct neighborhood. As a refinement of Node2PL's structure locks, NO2PL locks in case of updates only the nodes reachable from the context node thereby reducing its blocking granularity. Further optimizations are offered by a third variant OO2PL which locks for navigation operations only the traversed edges and for update operations only the affected navigation edges (again see [14] for details)

2.2 Multi-Granularity Locking Applied to XML Documents

As we will show in Sect. 5, the *-2PL group has some serious practical disadvantages; the most critical ones are handling of direct jumps by special lock modes IDR/IDX, missing modes for locking entire subtrees, and missing



lock compatibility matrix

	IR	IX	R	RIX	U	X
IR	+	+	+	+	-	-
IX	+	+	-	-	-	-
R	+	-	+	-	-	-
RIX	+	-	-	-	-	-
U	+	-	+	-	-	-
X	-	-	-	-	-	-

lock conversion matrix

	IR	IX	R	RIX	U	X
IR	IR	IX	R	RIX	U	X
IX	IX	IX	RIX	RIX	X	X
R	R	RIX	R	RIX	R	X
RIX	RIX	RIX	RIX	RIX	X	X
U	U	X	U	X	U	X
X	X	X	X	X	X	X

Fig. 2 URIX protocol: compatibilities and conversion rules

support for some operations, e.g., direct jumps to indexed element nodes not owning any ID attribute. For these reasons, we tried to avoid these drawbacks by adapting the well-known MGL protocols [9]—originally introduced for tables—to XML trees. As compared to classical MGL, a main difference is the double role of intention locks to indicate read/write operations deeper in the tree and to lock nodes (without locking the attached subtrees). Another difference are the much more complex conversion rules. When applied to the context node, the locks on its entire ancestor path have to be converted, too. Furthermore, we have combined the protocols with a lock depth parameter, the importance of which we have experienced in our measurements (see Sect. 5)[3].

In this way, we have derived a group of MGL protocols based on a general intention lock (IRX), on separate intention locks for read/write (IRIX), and finally an IRIX protocol enhanced by RIX and U modes [9] called URIX in Fig. 2. Special edge locks as introduced in [10] complement the node locks shown for the URIX protocol. As an example, assume no further locks are present in the protocol of Fig. 2, then a lock conversion of the context node to X can be performed by converting IR to IX on the ancestor path and R to X on the context node. In contrast to the *-2PL group, direct jumps must be protected by locking the entire ancestor path in an appropriate mode. This is very efficient when using DeweyIDs (see Sect. 3.2) for node identification

To optimize a protocol of the *-2PL group and to make it comparable to all other protocols explored, we have added the concept of intention locks borrowed from URIX with which the ancestor path to nodes accessed by direct jumps were protected. Furthermore, we have integrated a parameter for lock depth which, in turn, implied the introduction of subtree locks. Because the resulting protocol focuses on the parent of the context node, we called it Node2PLa; it served as a proxy for the *-2PL protocols in our experiments and, therefore, provided a kind of direct performance comparison (simulating the best-case *-2PL behavior) with the MGL* and taDOM* groups.

2.3 Protocols Tailored to DOM operations on XML Trees

The taDOM* group of protocols [10, 12] distinguishes node and edge locks and is a consequent optimization w.r.t. the DOM operations (covering level 2 and level 3) starting from the URIX protocol. Intention locks (IR, IX) are complemented by a read lock for individual nodes (NR)[4]. Whenever a context node is locked, its entire ancestor path is protected by appropriate locks. Definitely *new lock modes* are the so-called level locks (LR, CX) which, placed on the context node, lock all its children in the appropriate mode. Especially these two lock modes together with suitable conversion rules help to increase operation parallelism on trees and count for substantial improvement of transaction throughput. Hence, they make the *difference* of the taDOM* group as compared to *-2PL and MGL* groups.

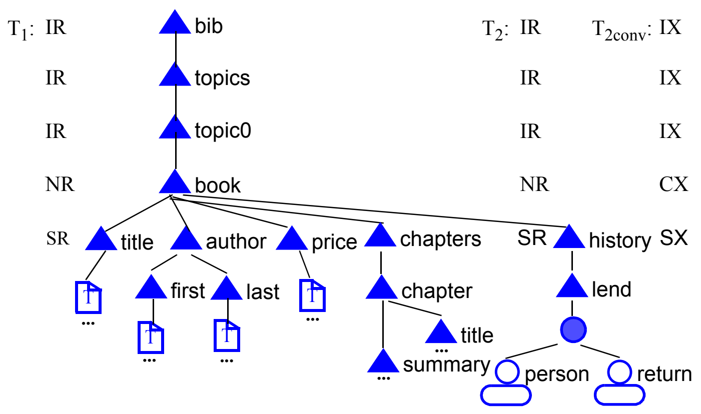
An LR lock mode (level read) locks the context node together with its direct-child nodes for shared access. For example, the method *getChildNodes()* only requires an LR lock on the context node and not individual NR locks for all child nodes. Similarly, an LR lock, requested for an attributeRoot node (see Sect. 3.1), locks all its attributes implicitly (to save lock requests for the *getAttributes()* method). A CX lock mode (child exclusive) on context node *c* indicates the existence of an X lock on some direct-child node and prohibits inconsistent locking states by preventing LR and SR lock modes. In contrast, it does not prohibit

3. 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 at level *n*.

4. The IR and NR modes show the same compatibilities in Fig. 3a; they will be differentiated when the taDOM protocol is refined.

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

a) Compatibility matrix



b) Locking example

Fig. 3 Node locking for the taDOM tree (taDOM2 protocol)

other CX locks on *c*, because separate direct-child nodes of *c* may be exclusively locked by concurrent transactions. Furthermore, we can use subtree locks on a context node which implicitly lock the entire subtree in read mode, update mode, or exclusive mode (SR, SU, SX).

Fig. 3b represents a cutout of the taDOM tree depicted in Fig. 5 and illustrates the resulting lock protocol using lock depth 4 for the following example: Transaction T_1 (TAqueryBook, see Sect. 4.2) uses an index and jumps to the *book* node in Fig. 3b to read all descendents of the book. in document order. It sets an NR lock on *book* and IR locks on all ancestors up to the root. Then, it navigates to the first child and, because lock depth 4 is reached, it places an SR lock on *title*, reads the nodes of the subtree, and proceeds to the *author* node setting again an SR lock. In this situation, T_2 (TAlendandReturn, see Sect. 4.2) enters the system, also uses index-based access to the same *book* node, and locks it by NR and its ancestors by IR. Afterwards it forwards to the last child and locks the entire subtree *history* by SR (lock depth 4). Assume it decides to lend this book; then it has to attach an additional subtree *lend'* with attributes *person* and *return* under the *history* element. For this reason, a lock conversion to SX on *history* is needed which is propagated to the root by converting NR on *book* to CX and the remaining IR locks to IX, as shown as T_{2conv} in Fig. 3b.

Although this example is very simple, it reveals a certain kind of complexity to be anticipated in XML lock protocols. Our example also nicely demonstrates the effect of lock depth. If we would have chosen lock depth 3, T_1 would have set an SR lock on *book*. This lock, because incompatible with CX, would have prohibited the lock conversion. Hence, fine-grained locking supported by the lock depth parameter enhances concurrency.

The compatibility matrix shown in Fig. 3a describes the compatibility of locks acquired on the same node by separate transactions. If a transaction T already holds a lock and requests a lock in a more restrictive or incomparable mode on the same node, we would have to keep two locks for T on this node. In general, *k* locks per transaction and node are conceivable. This proceeding would require longer lists of granted locks per node and a more complex runtime inspection algorithm checking for lock compatibility. Therefore, we replace all locks of a transaction per node with a single lock in a mode giving sufficient isolation. The corresponding rules are specified by the *lock conversion matrix* in Fig. 4, which determines the resulting lock for context node *c*, if a transaction already holds a lock (matrix header row) and requests a further lock (matrix header column) on *c*. A lock l_1 specified by an additional subscripted lock l_2 (e. g., CX_{NR}) means that l_1 has to be acquired on *c* and l_2 has to be acquired on each direct-child node of *c*.

In particular, conversion of the level locks becomes much more complex; its handling requires specialized locks to preserve the elegance and optimality of the new concept. An example for this conversion is as follows: Assume, a user starts a transaction requesting all child nodes of *c* which results in acquiring an LR lock on *c*. LR mode

	-	IR	NR	LR	SR	IX	CX	SU	SX
IR	IR	IR	NR	LR	SR	IX	CX	SU	SX
NR	NR	NR	NR	LR	SR	IX	CX	SU	SX
LR	LR	LR	LR	LR	SR	IX_{NR}	CX_{NR}	SU	SX
SR	SR	SR	SR	SR	SR	IX_{SR}	CX_{SR}	SR	SX
IX	IX	IX	IX	IX_{NR}	IX_{SR}	IX	CX	SX	SX
CX	CX	CX	CX	CX_{NR}	CX_{SR}	CX	CX	SX	SX
SU	SU	SU	SU	SU	SU	SX	SX	SU	SX
SX	SX	SX	SX	SX	SX	SX	SX	SX	SX

Fig. 4 Lock conversion matrix

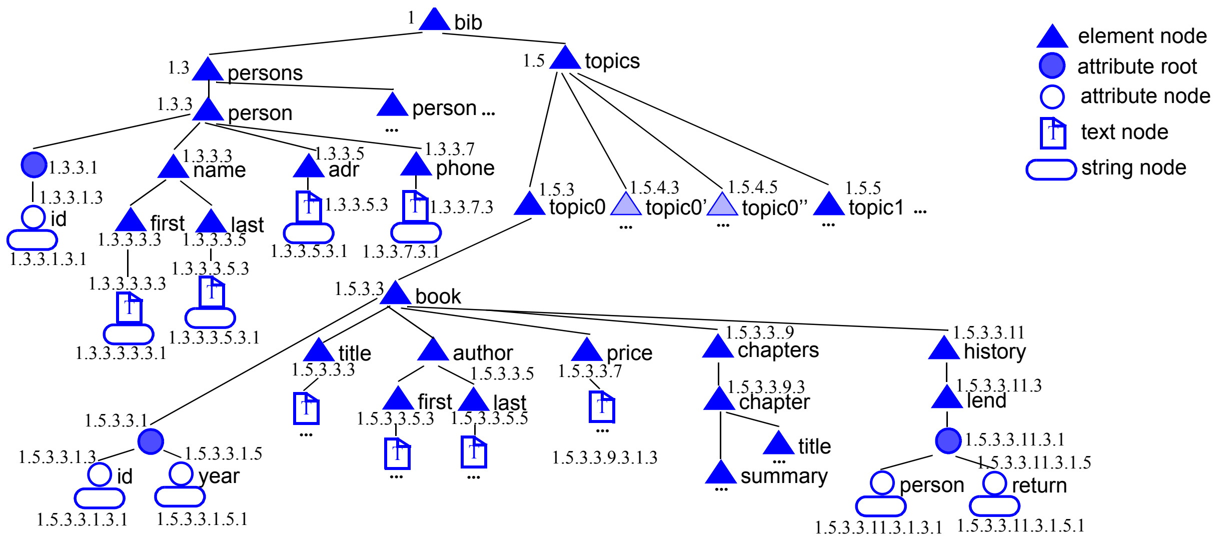


Fig. 5 A sample taDOM tree of a library document labeled with DeweyIDs

locks *c* and all direct-child nodes in shared mode. After that, the user wants to delete one of the previously determined child nodes. Therefore, the transaction acquires an SX lock on the corresponding child node and—applying the locking protocol—this requires the acquisition of a CX lock on *c* which already holds the LR lock. Using rule CX_{NR} specified in Fig. 4, the transaction has to convert the existing LR lock on *c* to a CX lock and to acquire an NR lock on each direct-child node of *c* (except the child node which is already locked for deletion by an SX lock).

So far, we have sketched the taDOM2 protocol covering all operations of the standard DOM2, whereas taDOM2+ optimizes certain situations which may occur during lock conversions. Therefore, the four lock modes LRIX/SRIX (level/subtree read intention exclusive) and LRCX/SRCX (level/subtree read child exclusive) are provided in addition. Together with a lock compatibility matrix the related conversion rules have to be derived. The DOM3 standard introduces new operations, e.g., the renaming of nodes. Hence, the taDOM3 protocol offers tailored and dedicated lock modes for the enhanced DOM standard and taDOM3+ additionally supports conversion in an optimal way. Because all taDOM* protocols are absolutely complex—taDOM3+ includes 20 lock modes and three modes for edges together with the related compatibilities and conversion rules—the reader will understand that we cannot just quickly cover them or repeat them for comprehension. We have described and compared the protocols of the taDOM* group and proven their correctness in [12]. As for all other protocols referenced, they are not visible to the programmer; in contrast, they are automatically applied by the lock manager when protecting the actual DOM operation or the corresponding operation issued when a higher level request (e.g., XQuery or XPath) is mapped to the operations of the access system.

3 System Aspects of XTC

High degrees of parallelism for read and write operations are a prime objective of each concurrency control protocol. As a prerequisite, fine-grained storage and management of XML documents has to be achieved. We have learnt that the location of node IDs together with the determination of their ancestor node IDs are of outmost importance for any locking protocol. These key tasks should be performed without accessing the nodes in the document itself, because references to external memory for locking purposes should be avoided to the extent possible. For these reasons, the design of the storage model together with addressing and indexing schemes play a performance-critical role in our context.

3.1 taDom Storage Model

Efficient and effective processing and concurrent operations on XML documents are greatly facilitated, if we use a specialized internal representation. Therefore, we have implemented in our XTC system the taDOM storage model illustrated in Fig. 5 as a slight extension of the XML tree representation defined in [21]. In contrast to the DOM tree, we do not directly attach attributes to their element node, but introduce separate attribute roots which connect the attribute nodes to the resp. elements. String nodes are used to store the actual content of an attribute or a text node. Via the DOM API, this separation enables access of nodes independently of their value. Our representational enhancement does not influence the user operations and their semantics on the XML document, but is solely exploited by the lock manager to achieve certain kinds of optimizations. In summary, our storage mechanism offers an extensible file structure as a container of single XML documents such that updates (by IUD operations) can be performed on any of its nodes. We showed that a very high degree of storage occupancy (> 96%) for taDOM trees is achieved under a variety of different update workloads [10].

3.2 Addressing and Indexing Schemes

Fast access to and identification of all nodes of an XML document is mandatory to enable effective indexing primarily supporting declarative queries and efficient processing of direct-access methods (e.g., getElementById()) as well as navigational methods (e.g., getNextSibling()). Furthermore, we have observed very early in our XTC project that, for fine-grained locking, it is indispensible to rely on an immutable labeling scheme which provides, when accessing a node, the node labels (identifiers) of all ancestors without accessing the XML document, as sketched in the example lock protocol in Sect. 2.3 .

Several candidates for such a scheme have been proposed in the literature [1, 4, 20]. While most of them are adequate to label static XML documents, the design of schemes for dynamic documents allowing arbitrary insertions within the tree—free of reorganization, i.e., no reassignment of labels to existing nodes—remains a challenging research issue. Note, we also cannot tolerate so-called dynamic schemes which modify labels on the fly, because the lock manager uses these labels as node identifiers and, on the other hand, an application may keep such a label for later direct access to a tree node. Hence, the proposed approaches are classified into range- and prefix-based schemes. While range-based schemes consisting of independent numbering elements (e.g., DocID, startPos : endPos, level, see [1]) cannot always avoid relabeling in case of node insertions, prefix-based schemes seem to be more flexible. We believe that they are at least as expressive as range-based schemes, while some suitable candidates of them guarantee stability of node IDs under arbitrary insertions, in addition. In particular, we favor a scheme supporting efficient insertion and compression while providing the so-called Dewey order (defined by the Dewey Decimal Classification System). Conceptually similar to the ORDPATH scheme [18], our scheme refines the mapping and solves practical problems of the implementation.

At the first sight, everybody claims that a scheme based on DeweyIDs cannot be efficiently implemented. Therefore, we have performed an extensive empirical evaluation, before we decided this design issue. We have shown in [13] for a large variety of XML documents that the use of DeweyIDs is feasible. Using a variable-length encoding scheme, we obtained an average DeweyID size of 5 to 10 bytes where the average and maximum tree depths were up to 9 and 38, respectively. To further reduce the DeweyID size in frequent situations, we enhance our implementation with prefix compression.

Referring to Fig. 5, our solution is sketched by examples. A DeweyID like 1.5.3.3.9 consists of several so-called divisions separated by dots (in the human readable format). The root node of the document is always labeled by DeweyID 1 and consists of only a single division. The children obtain the DeweyID of their parent and attach another division whose odd value increases from left to right. To allow for later node insertions at a given level, we introduce a parameter *distance* which determines the gap initially left free in the labeling space (at least one even division value). In Fig. 5, we have chosen the minimum distance value of 2. Furthermore, assigning at a given level a distance to the first child, we always start with *distance + 1*, thereby reserving division value 1 for attribute roots and string nodes (illustrated for the attribute root of 1.3.3 with DeweyID 1.3.3.1). Hence, the mechanism of the Dewey order is quite simple when the IDs are initially assigned, e.g., when all nodes of the document are bulk-loaded. Maintenance of Dewey order under UID operations is, however, more complex [13].

The salient features of a scheme assigning a DeweyID to each tree node include the following properties: Referring to the DeweyID of a node, we can determine the level of the node in the tree and the DeweyID of the parent node. Hence, we can derive its entire ancestor path up to the document root without accessing the document. By comparing the DeweyIDs of two nodes, we can decide which node appears first in the document's node order. If all sibling nodes are known, we can determine the exact position of the node within the document tree. Furthermore, it is possible to insert an arbitrary number of new nodes at any location without relabeling existing nodes. After insertion of *topic0'* and *topic0''* in Fig. 5, the even division value *4* (keeping the current node level) is exploited in the address of the new element nodes. In addition, we can rapidly figure out all nodes accessible via the typical XML navigation steps, if the nodes are stored in document order, i.e., in left-most depth-first order.

Fast (indexed) access to each node is provided by variants of B*-trees tailored to our requirements of node identification and direct or relative location of any node. Fig. 6a illustrates the storage structure – consisting of *document index* and *document container* as a set of chained pages – sketching a sample XML document, which is stored in document order; the key-value pairs within the document index are referencing the first DeweyID stored in each container page. In additional to the storage structure of the actual document, an *element index* is created consisting of a *name directory* with all element names occurring in the XML document (Fig. 6b).

For each specific element name, in turn, a *node-reference index* may be maintained which addresses the corresponding elements using their DeweyIDs. In all cases, support of variable-length DeweyIDs in their roles as keys and pointers is mandatory; additional functionality for prefix compression is very effective. Because of reference locality in the B*-trees while processing XML documents, most of the referenced tree pages (at least the ones belonging to the upper tree layers) are expected to reside in DB buffers – thus reducing external accesses to a minimum.

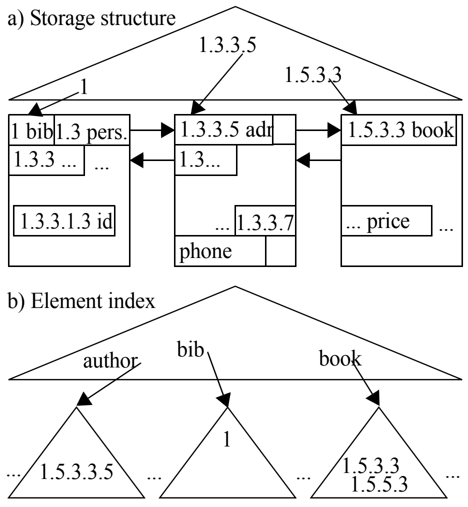


Fig. 6 Document storage using B*-trees

3.3 Meta-Synchronization

For our task, a system is needed capable of running concurrency control experiments using different locking protocols in the same physical environment. As a prerequisite, we had developed during the last two years our XTC primarily as a testbed for empirical concurrency control. The key idea to really enable cross-protocol comparison was the appropriate isolation of the XTC lock manager as a kind of abstract data type. It accepts the locking requests from the XTC node manager (and other participating components) in a more abstract form as so-called meta-lock requests including

- node locks (shared, update, exclusive)
- shared level locks
- tree locks (shared, update, exclusive)
- edge locks (shared, update, exclusive) on previous sibling edge, next sibling edge, first child edge, and last child edge
- as well as release locks at commit for isolation level *repeatable read* or at end of operation for isolation levels *uncommitted read* and *committed read.*

When using this so-called meta-synchronization, XTC has to map the meta-lock requests to the actual locking algorithm which is achieved by the lock manager's interface. Hence, exchanging the lock manager's interface implementation exchanges the system's complete XML locking mechanism. In this way, we could run XTC in our experiments with 11 different locking protocols. At the same time, all experiments were performed on the taDOM storage model which is optimized for fine-grained management of XML documents by using a refined node structure and DeweyIDs.

All mandatory concepts of a lock manager are introduced in [9]. Our implementation including lock requests, lock conversions, and lock waits is described in [11].

4 Experimental Setting

So far, we have sketched the characteristics of all approaches known to us which focus on fine-grained XML concurrency control. By looking at their features or lock compatibilities alone, it is impossible to gain sufficient insight into their relative performance. Simulation of such complex protocols reflecting practical XML document structures and workloads just evaluates the underlying processing model—not necessarily close to a real environment—and can, therefore, not reveal unknown bottlenecks or genuine “performance surprises”. Therefore, we need a series of controlled empirical experiments to determine the protocols' relative strengths in typical applications. To compare the results in the most accurate way, it is indispensable to run all experiments in the same XDBMS setting using the same database and the same workload.

4.1 Existing XML Benchmarks

The suitable selection of benchmarks is critical to the quality of the results drawn from experiments, because benchmarks should, in the first place, reflect the most important characteristics of the application domain. According to Jim Gray [7], general properties of benchmarks include design towards the application domain, relevance, portability, scalability, and simplicity. In particular, design mirroring the application domain and relevance requiring the use of typical operation mixes of it are the key properties of database benchmarks. Because concurrency control is in the focus of our explorations, the scope of the benchmark must be directed towards stretching the lock manager's behavior and must therefore include multi-user operations and contain a varying degree of update operations to be useful.

Unfortunately, existing benchmarks do not match all these requirements. XMach-1 [2], for example is designed for scalable, multi-user Web applications, but targets the XDBMS behavior, in general, without specific emphasis on concurrency control. In contrast, the scope of XMark [19] is the XML query processor and concentrates on single-user mode only. XOO7 [3] also targets on the query processor as its scope and has no particular application orientation. Hence, benchmarks for our specific needs are missing—primarily, because so far XDBMS research was strongly focused on retrieval only. Therefore, we had to design tailored benchmarks together with an automated measurement environment. For this reason, we could specifically realize the following performance metrics for each experiment:

- number of committed and aborted transactions for a prespecified lock depth and isolation level
- Average, maximal, and minimal duration of a transaction of a given type
- number and type of deadlocks for a lock protocol.

4.2 Framework TaMix for XML Benchmarks

To support expressive transaction runs and to observe at the same time the mentioned guidelines, we have designed and implemented the framework TaMix for benchmarks on XML documents. Fig. 7 sketches the TaMix infrastructure which illustrates that a number of TaMix clients can execute transactions while XTC serves their concurrent DB requests. Typical tasks of the TaMix coordinator are starting and stopping the XTC server and the TaMix clients. To configure the test runs, the coordinator owns two property files. *TaMix.Coord.props* contains management information about the clients and the server as well as directives how to load the XML documents (to start with fresh copies for each run). *TaMix.props* specifies the experiments: duration of the various test runs, number of runs for each lock protocol, lock depths to be applied, isolation levels to be used, etc. Begin and duration of individual transactions can be varied by properties *waitAfterOperation* and *waitAfterCommit* which force the related client to wait a specified period after each DB operation or transaction, respectively (simulation of think time, increase of lock wait, etc.).

TaMix records the measurement results for the specified metrics to enable later evaluation. Furthermore, in cooperation with the XTCdeadlockDetector, it collects data in case of deadlocks about the number of active transactions, the locks held, the state of the wait-for-graph, etc. Thus, we are able to analyze each deadlock event very precisely, e.g., whether it was caused by lock conversion (frequent occurrence) or by lock requests in separate subtrees (rather rare cases).

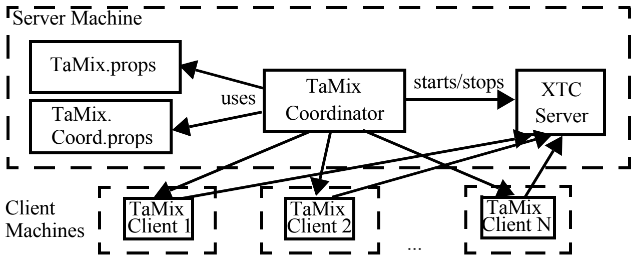


Fig. 7 TaMix infrastructure

Here we can give only a brief overview of the transaction types emulating a library application, before we describe the specific transaction mixes for our experiments:

- TAqueryBook selects a *book* element by random ID and provides details of the book. It uses a direct jump via an ID attribute into the tree (using an index) and traverses the subtree by navigational read operations.
- TAchapter: same operational read profile followed by an update of a text node.
- TAdelBook: same operational read profile, but on a random *topic* element followed by a deletion of a *book* subtree.
- TAlendAndReturn: direct location of a randomly chosen *book* element followed by complex navigational steps with updates, deletions, and insertions of elements.
- TArenameTopic locates a *topic* element by a random ID and renames it.

As any concurrency protocol, all XML lock protocols try to maximize throughput. The role of the reader transactions (TAqueryBook) is to provide a continuous system load under which the remaining IUD transactions have to compete for data sources. They provoke together with the readers wait relationships and deadlocks, which, in turn, determine the transaction throughput.

4.3 Composition of TaMix

Evaluation of concurrency control protocols is very complex because of the huge parameter space (fan-out and depth of XML trees, mix of transactional operations, specific application domains, degree of application concurrency, optimization of protocols, etc.) and the timing conditions (arrival and blocking times, transaction duration, etc.). Therefore, it is not well amenable to analytical methods. On the other hand, there are no perfect benchmarks for XDP processing on document trees either. However, by running experiments in a real XDBMS environment we gain accuracy of what is going on and we hope to derive first indicative results.

All transactions are composed to so-called clusters and operate on a *bib* document (see Fig. 5) which itself can be configured to the size desired; it is highly scalable and may range from a few Kbytes to several hundred Mbytes. For all subsequently reported results on lock performance comparisons, *bib* was composed in the following way:

- 1000 *person* elements and 100 *author* elements
- 2000 *book* elements equally distributed across 100 *topic* elements (20 per *topic*)

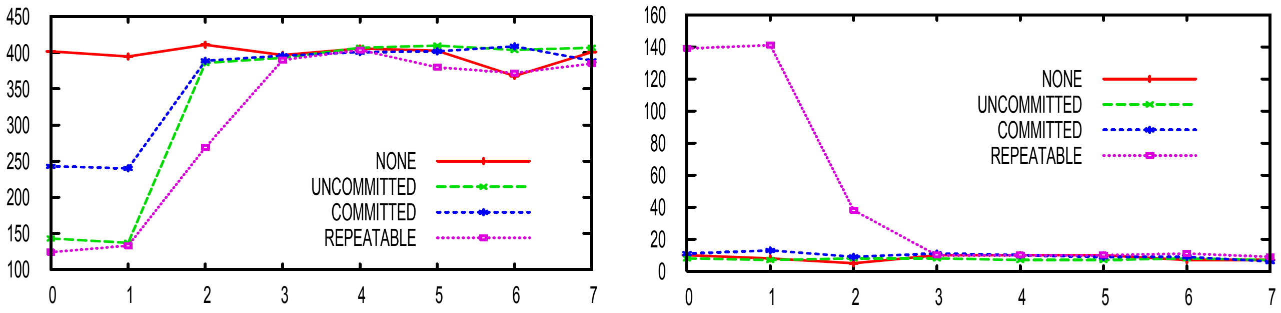


Fig. 8 CLUSTER1 under taDOM3+—influence of isolation level: transaction throughput (left) and deadlocks (right)

- each *book* owns 5 to 10 *chapter* elements
- a *history* element owns with equal probability 9 or 10 *lend* elements.

Because of space restrictions we concentrate on two cluster evaluations providing the most expressive results for the performance comparisons of the lock protocol. CLUSTER1 specifies a continuous workload for each of the 3 TaMix clients as follows (hence, TaMix Coordinator keeps 72 transactions active during a benchmark run):

- 9 transactions of type TAqueryBook and 5 of type TAchapter
- 2 transactions of type TArenameTopic and 8 of type TAlendAnd Return

The TaMix-specific parameters for CLUSTER1 characterizing the variation of the test runs were as follows:

- isolation levels: *none, uncommitted, committed, repeatable*
- lock depths where applicable: 0 to 7
- number of runs per isolation level and lock depth: 4
- run duration: 5 mins, waitAfterCommit: 2500 ms, waitAfterOperation: 100 ms
- random wait before executing the first operation of a transaction: 0 to 5000 ms

In contrast to CLUSTER1, we planned a very specific experiment with CLUSTER2 which reveals remarkable weaknesses of locking performance in some protocols. It only consists of a single execution of TAdelBook in single-user mode, however, using isolation level *repeatable*. Hence, the metrics of CLUSTER1 cannot be used in the experiment, because transaction parallelism, etc. are not indicative. Here, transaction duration is very expressive and characterizes the amount of locking overhead necessary.

Our testbed environment consists of a server machine and a number of workstations. The server is equipped with 4 Intel XEON processors (1.50 GHz each), 4 GB memory, and an IDE disk with 280 GB. The TaMix clients run on separate workstations (1.70 GHz Intel Celeron processor, 256 MB memory, and 100 Mbit ethernet connection to the server).

5 Results and Interpretation

After the consolidated description of the TaMix framework and the workloads on the *bib* document, we first discuss (out of numerous measurements) our most important findings running CLUSTER1, before we add a remark on the results of CLUSTER2.

5.1 Influence of the Isolation Level

The stronger the isolation level[5], the higher the consistency guarantees of the XDBMS, but the less transaction throughput has to be achieved, in general. To confirm this expectation, our first experiment concentrates on the influence of the isolation level on the performance behavior. We have chosen, as we will see later, the best overall protocol taDOM3+ to characterize the influence of isolation level and locking depth on transaction throughput. On the other hand, the relative behavior of all other protocols supporting lock depth is similar such that we can cover the principal behavior by considering a single protocol. Fig. 8 illustrates the behavior expected in its characteristic aspects. Note, lock depth 0 corresponds to the use of document locks, which explains the low performance. The higher the lock depth parameter, the smaller are the subtrees locked. As a consequence, throughput rapidly increases to a level where further refinement does not enhance anymore parallelism. The only surprise is the unexpected ordering of the protocols of isolation level *committed* and *uncommitted* at depth 0 and 1. Closer inspection explains this special behavior [17].

5.2 Results for Isolation Level *repeatable*

To include the *-2PL group of protocols (having no lock depth parameter) into our evaluation, we compare them as a separate group only using transaction throughput. Fig. 9 visualizes the throughput gained for CLUSTER1

5. While *none* acquires no locks at all, all others need long write locks; *uncommitted* means no read locks, *committed* and *repeatable* short and long read locks, respectively.

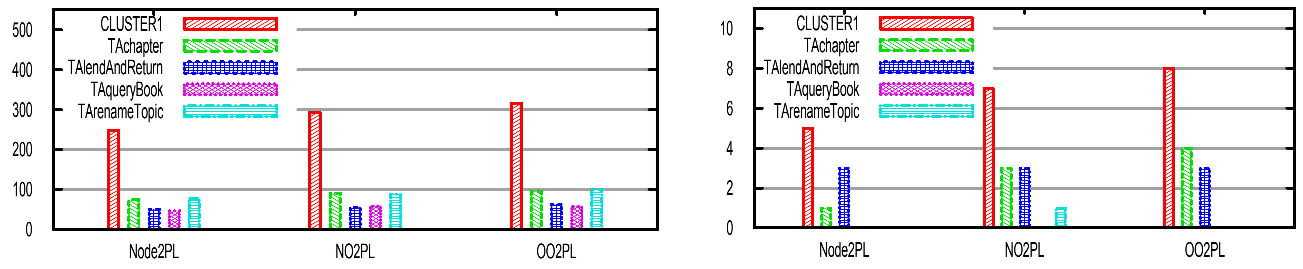


Fig. 9 Running CLUSTER1 under the *-2PL group: transaction throughput (left) and deadlocks (right)

counting all transactions and considering the individual transaction types. Although also producing a higher number of aborted transactions (deadlocks), the throughput of OO2PL is remarkably higher than that of NO2PL and that again is higher than that of Node2PL. Such a behavior can be anticipated, because Node2PL locks the entire level of the context node for any IUD operation, whereas NO2PL and OO2PL only lock its neighborhood. As compared to its group competitors, OO2PL implies the acquisition of finer and, therefore, a larger number of locks; the advantage of higher parallelism, however, clearly outweighs this processing overhead of the lock manager [11].

As already explained, we have optimized the idea underlying the *-2PL group protocols, added *lock depth*, and in this way derived Node2PLa which turned out to be generally superior to the remaining group members. Therefore, we have chosen Node2PLa as a proxy for the *-2PL group used in further comparisons.

With these explanations we can illustrate all results of CLUSTER1 obtained for isolation level *repeatable* in Fig. 10. The first impression concerns the clear gaps separating the various protocol groups (*-2PL, MGL*, taDOM*), which highlights the relative performance advantages. As compared to the *-2PL group, we obtain in the average ~50% and ~100% throughput gain for the MGL* group and taDOM* group, respectively, while less deadlocks are provoked by them (particularly in cases of lower lock depths). Furthermore, it nicely illuminates the average performance gain accomplished by fine-grained locks tailored to the effects of the operations to be isolated. However, detailed explanations of the locking behavior are impossible for these aggregated CLUSTER1 results. Of course, there is the low transaction throughput at locking depth 0 and 1, which is caused by the high number of transaction aborts at these levels which, in turn, are attributed to deadlocks. Higher lock depths mean that the nodes at upper tree levels carry intention locks, whereas locks are set on deeper tree levels dramatically reducing the blocking and deadlock potential (see Fig. 10).

Closer inspections of the separate transaction types allow us to refine the result interpretation. Fig. 11 illustrates the throughput separated by transaction type. Analyzing read transactions of type TAqueryBook, it immediately becomes clear that they (without any aborts) almost exclusively contribute to the CLUSTER1 throughput at level 0 and 1. Looking at the corresponding results of write transactions of type TAChapter, TAlendAnd Return, and TArenameTopic confirms this observation. These three writer transaction types together produced all deadlocks at level 0 and 1, whereas the reader transaction type did not contribute to transaction aborts. The course of the deadlock graphs of the writers (not shown here) again exhibits the same characteristic behavior than that of Fig. 10.

Fine-granular locking comes into play at depth levels > 1, where the three update transactions begin to compete with the readers of type TAqueryBook. The graphs nicely show that Node2PLa begins to react a level deeper (depth level > 2) to enable true reader/writer competition. This is due to the overly restrictive parent locking when processing the context node. Furthermore, Node2PLa fails to succeed almost completely with type TArenameTopic, because it is not prepared to the specific operation and has to use very large lock granules.

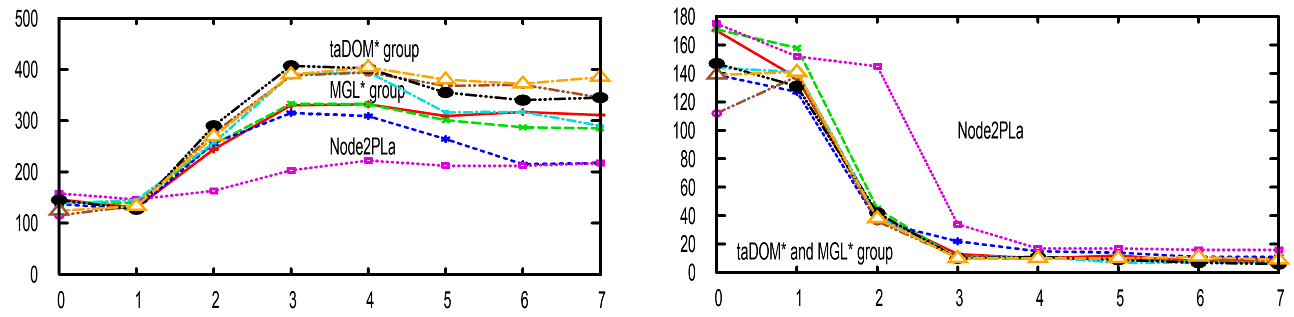


Fig. 10 Synopsis of all protocols on CLUSTER1—transaction throughput (left) and deadlocks (right)

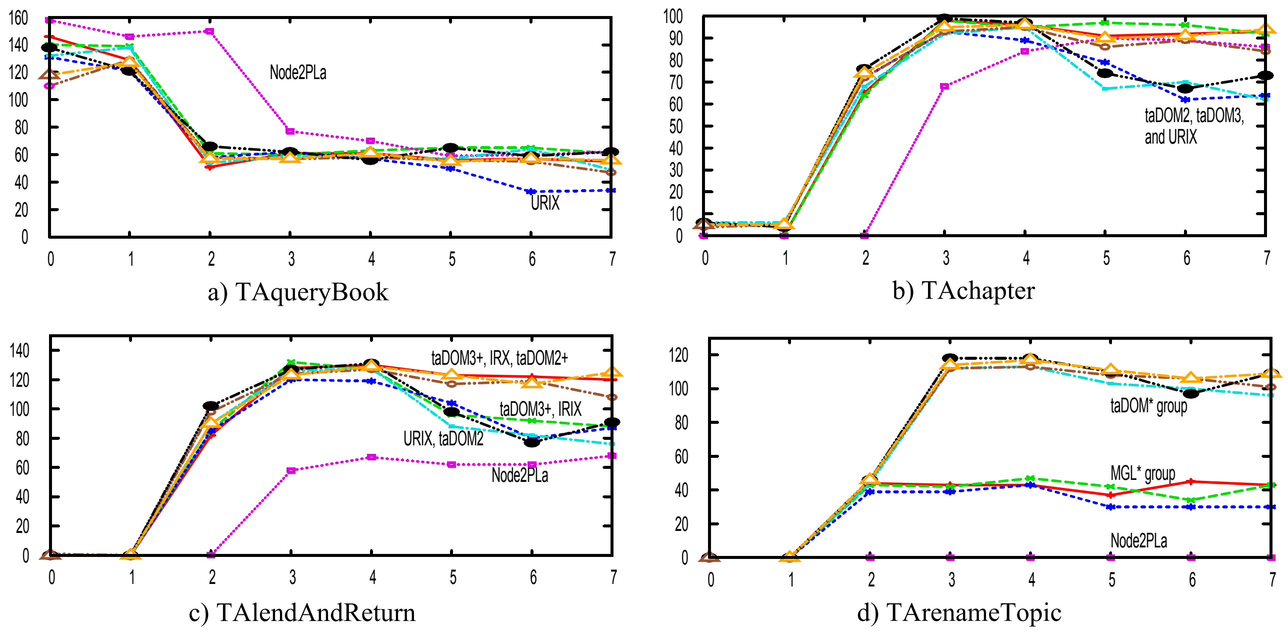


Fig. 11 Transaction throughput of CLUSTER1 separated by transaction types

Although the MGL* group can keep up very well with the best protocols in some depth ranges, it has to experience strong drawbacks in other situations. For example in processing TArenameTopic, it cannot separate the name from the content of a topic and can, therefore, not optimize locking in such situations. As a result, the MGL* group ends up in the middle position when drawing the average performance over all transaction types.

Finally, when considering the superior group, we can clearly identify the additional gain caused by specialized handling of lock conversions. However processing transaction types TAchapter and TAlendAndReturn, read locks for getChildNodes() together with a specialized operation for subtree access (in our case, getFragment nodes() [17]) and subsequent conversions needed for updates deeper in the subtree caused blocking situations which degraded the performance of taDOM2 and taDOM3 (as well as IRIX and URIX) in levels > 4. Tailored locks and their conversions prevented such a penalty.

5.3 Direct Jumps Unprotected by Intention Locks

Because CLUSTER2 only contains a single transaction of type TAdelBook, we here use as a performance metric its execution time under the various lock protocols. This experiment primarily revealed that all protocols using intention locks effectively handle all situations where large subtrees have to be deleted. In contrast, it uncovered the ponderous behavior of the *-2PL group to cope with such situations. Because they set aside the use of intentions to protect the paths to the nodes directly accessed, these protocols—before removing a subtree—need to search the entire subtree for elements owning ID attributes. Only setting IDX locks on them guarantees that other transactions do not reference anymore nodes in the tree to be deleted. The necessary location steps have to be performed via the node manager and may include accesses to external devices. Therefore, the cost of such deletions critically depends on the size of the related subtree. Hence, in our example the *-2PL group roughly consumes for the deletion twice as much time than all other protocols considered (see Fig. 12).

6 Conclusions

Specialization of concurrency control protocols matters. We have obtained convincing results that tailoring the protocol behavior to the properties of XML documents and the related navigational model delivers substantial operational benefit. Note, a blindly applied relational protocol would achieve the throughput at depth 0 in all our throughput graphs.

We have experienced that adequate edge locks and node locks—including intention, level, and subtree locks—and their conversion rules are mandatory to accomplish high transaction throughput. Lock depth itself is at least in the upper document layers a performance-critical parameter. We believe that the use of DeweyIDs is of paramount importance for the lock protocol overhead and, in turn, for the entire performance of concurrency control in XML trees. All ancestor node IDs and most other IDs needed for locking navigation steps can be derived from them (using indexes and Dewey order) without traversing the document

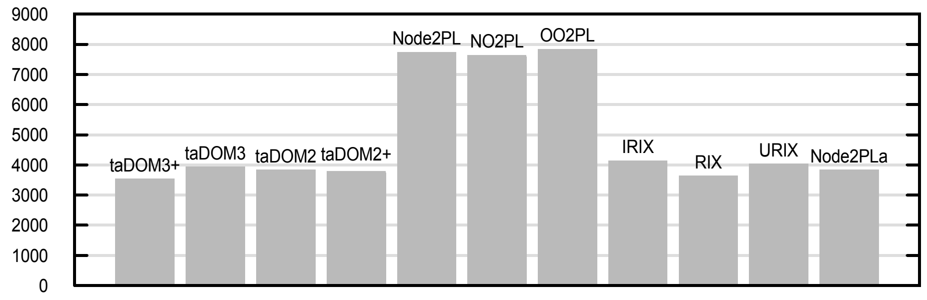


Fig. 12 Transaction execution times for all protocols on CLUSTER2

itself. Queries specified by declarative languages are assumed to be frequently processed via indexes which will require a large number of *direct jumps*. On the other hand, DeweyIDs allow structural joins and set-theoretic operations such that they become more useful than TIDs in relational DBMSs.

In our XTC project, we use taDOM trees as a storage model for XML documents thereby removing some blocking situation when navigation touches attribute or text nodes. Our taDOM* protocols, however, can be applied to DOM trees [21] as well. Indeed, we currently virtualize these extra node types such that regular DOM trees are stored, but the lock manager virtually expands the attribute and text nodes to reduce some blocking situations.

The more the locks are adjusted to the tree structure and its operations, the more complex is the resulting protocol (up to 20 lock modes in taDOM3+)—but the substantially higher is the transaction throughput. Fig. 10 confirms that the performance gain of the best protocols (taDOM*) as compared to the optimized *-2PL group may be in the order of 100%. For some transaction types favored by special properties of taDOM*, the performance gain may be ~200% (see Fig. 11d). Because the relative performance differences within the taDOM* group are rather minimal while all of them have revealed their quality, we can claim that each of them can be applied in an XDBMS guaranteeing satisfactory performance. The selection of an individual protocol of the taDOM* group may be driven by the trade-off of optimization overhead and extra benefit in specific applications.

References

[1] S. Al-Khalifa, et al.: Structural Joins: A Primitive for Efficient XML Query Pattern Matching. Proc. 18th Int. Conf. on Data Engineering, 141 (2002)

[2] T. Böhme, E. Rahm: XMach-1: A Benchmark for XML Data Management. Proc. National German Database Conference (BTW 2001), Informatik aktuell, Springer, 264-273

[3] S. Bressan, M. L. Lee, Y. G. Li, Z. Lacroix, U. Nambiar: The XOO7 Benchmark, http://www.comp.nus.edu.sg/~ebh/XOO7.html

[4] E. Cohen, H. Kaplan, T. Milo: Labeling Dynamic XML Trees. PODS 2002: 271-281

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

[6] T. Grabs, K. Böhm, H.-J. Schek: XMLTM: Efficient Transaction Management for XML Documents. Proc. ACM CIKM Conf., McLean, VA, 142-152 (2002)

[7] J. Gray: The Benchmark Handbook for Database and Transaction Systems (2nd Edition). Morgan Kaufmann 1993

[8] J. Gray: A Call to Arms. ACM Queue 3:3, 30-38, April 2005

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

[10] M. Haustein, T. Härder: Adjustable Transaction Isolation in XML Database Management Systems. Proc. 2nd Int. Database Symp., Toronto, Canada, LNCS 3186, 173-188 (2004)

[11] M. Haustein, T. Härder: A Lock Manager for Collaborative Processing of Natively Stored XML Documents, in: Proc. Braz. Symp. on Databases (SBBD), Brasilia, 230-244 (2004)

[12] M. Haustein, T. Härder: Optimizing Concurrent XML Processing, submitted (2005), http://wwwdvs.informatik.uni-kl.de/pubs/p2005.html

[13] M. Haustein, T. Härder, C. Mathis, M. Wagner: DeweyIDs - The Key to Fine-Grained Management of XML Documents, submitted (2005), http://wwwdvs.informatik.uni-kl.de/pubs/p2005.html

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

[15] H. V. Jagadish, S. Al-Khalifa, A. Chapman. TIMBER: A Native XML Database. The VLDB Journal 11(4): 274-291 (2002)

[16] J. R. Jordan, J. Banerjee, R. B. Batman: Precision Locks. Proc. ACM SIGMOD Conf., Ann Arbor, Michigan, 143-147 (1981)

[17] K. Luttenberger: Lock Protocols in XML Database Systems (in German). Diploma Thesis, Dept. of Comp. Science, Univ. of Kaiserslautern (2005)

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

[19] A. Schmidt, F. Waas, M. Kersten. XMark: A Benchmark for XML Data Management. Proc. 28th VLDB Conf., Hong Kong, China, 974-985 (2002)

[20] A. Silberstein, H. He, K. Yi, J. Yang: BOXes: Efficient Maintenance of Order-Based Labeling for Dynamic XML Data. ICDE 2005: 285-296

[21] W3C Recommendations. http://www.w3c.org (2004)

[22] XQuery 1.0: An XML Query Language. W3C Working Draft (Oct. 2004)