TRANSACTIONAL SUPPORT IN NATIVE XML DATABASES

Theo Härder, Sebastian Bächle and Christian Mathis
University of Kaiserslautern, Gottlieb-Daimler-Str., 67663 Kaiserslautern, Germany

Keywords: XML database management, concurrency control, logging and recovery, elementless XML storage.

Abstract: Apparently, everything that can be said about concurrency control and recovery is already said. None the less, the XML model poses new problems for the optimization of transaction processing. In this position paper, we report on our view concerning XML transaction optimization. We explore aspects of fine-grained transaction isolation using tailor-made lock protocols. Furthermore, we outline XML storage techniques where storage representation and logging can be minimized in specific application scenarios.

1 INTRODUCTION

When talking about transaction management, everybody implicitly refers to relational technology. It is true that the basic concepts of ACID transactions (Härder and Reuter, 1983) were primarily laid in the context of flat table processing and the related query languages and later adjusted to object orientation. As a major advance for transaction processing, Weikum and Vossen (2002) unified concurrency control and recovery for both the page and object model. Performance concerns led to a refinement of the page model to exploit records as more fine-grained units of concurrency control. Their textbook used as the "bible" in academic lectures "synthesizes the last three decades of research into a rigorous and consistent presentation" and it systematically describes and "organizes that huge research corpus into a consistent whole, with a step-by-step development of ideas" (J. Gray in the foreword of this textbook). It seemed that *everything that can be said about concurrency control and recovery is said* in this textbook already.

But new data models and processing paradigms arrived in the recent past. The available types of data, their modeling flexibility, and their contents themselves have substantially evolved and more and more surpass the realms where the relational model is appropriate. Above all, the importance of efficient XML query processing in multi-user environments grows along with the rapidly increasing sizes and volumes, the advanced applications and the pervasiveness of XML. For semi-structured data, XML together with its usages has become a (large) set of standards for information exchange and representation. It seems, the more domains are conquered by XML (by defining schemas for business cooperation), the more the relational systems approach "legacy".

Hence, efficient and effective transaction-protected collaboration on XML documents (XQuery Update Facility) becomes a pressing issue. Solutions, optimal in the relational world, may fail to be appropriate because of the documents' tree characteristics and differing processing models. Structure variations and workload changes imply that transaction-related protocols must exhibit better flexibility and runtime adjustment. "Blind" transfer of relational technology would lead to suboptimal solutions for storage and logging, because the structure part of XML often exhibits huge redundancies.

Because a number of language and processing models are available and standardized for XML (DOM, XQuery), general solutions for transaction support have to consider protocols for concurrently evaluating stream-, navigation-, and path-based queries. For this reason, a flexible XML database management system (XDBMS) has to support XPath, XQuery, and DOM/SAX. DB requests specified by different XML languages may be scheduled and arbitrary transaction mixes may occur. Therefore, serializability has to be guaranteed for those applications.

In the following, we will outline that novel approaches for XML concurrency control, document storage, as well as logging and recovery may have substantial saving and optimization potential.

Härder T., Bächle S. and Mathis C. (2008).
TRANSACTIONAL SUPPORT IN NATIVE XML DATABASES.
In *Proceedings of the Tenth International Conference on Enterprise Information Systems - DISI*, pages 368-373
DOI: 10.5220/0001725303680373

2 LOCK PROTOCOLS

So far, there hardly exist any specific concurrency control protocols for XML. Only some hierarchical lock protocols are available from the relational world by adjusting the idea of multi-granularity locking (Gray, 1978) to the specific needs of XML trees. Note, the well-known B-tree latch protocols (Graefe, 2007) cannot be used to isolate XML transactions; they only isolate concurrent read/write *operations* on B-trees and preserve their structural consistency. In contrast, locks isolate concurrent *transactions* on user data and – to guarantee serializability – have to be kept until *transaction commit*. With similar arguments, index locking can not cope with the navigational DOM operations (Mohan, 1990).

When fine-granular access to document trees has to be achieved, declarative requests have to be translated into sequences of navigating operations. Therefore, the DOM model is considered, even for declarative languages, an adequate representative as far as locking requirements are concerned.

We repeat neither hierarchical lock protocols used in all industrial-strength DBMSs (Gray and Reuter, 1993) nor our own work on XML locking (Haustein and Härder, 2008). Instead, we refer to these well-known protocols and only emphasize important properties for better comprehension.

2.1 Multi-Granularity Locking

Hierarchical lock protocols – also denoted as multi-granularity locking (MGL) – are used "everywhere" in the relational world. For performance reasons in XDBMSs, fine-granular isolation at the node level is needed when accessing individual nodes or traversing a path, whereas coarser granularity is appropriate when traversing or scanning entire trees. Therefore, lock protocols, which enable the isolation of multiple granules each with a single lock, are also beneficial in XDBMSs. Regarding the tree structure of documents, objects can be isolated acquiring the usual *subtree locks* with modes R (read), X (exclusive), and U (update with conversion option), which implicitly lock all objects in the entire subtree addressed. To avoid lock conflicts when objects at different levels are locked, so-called *intention locks* with modes IR (intention read) or IX (intention exclusive) have to be acquired along the path from the root to the object to be isolated and vice versa when the locks are released. Hence, we can map the relational IRIX protocol to XML trees and use it as a generic solution.

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

Figure 1: taDOM2 lock compatibilities.

Using the IRIX protocol, a transaction reading nodes at any tree level had to use R locks on the nodes accessed thereby locking these nodes together with their entire subtrees. This isolation is too strict, because the lock protocol unnecessarily prevents writers to access nodes somewhere in the subtrees. Giving a solution for this problem, we want to sketch the idea of lock granularity adjustment to DOM-specific navigational operations.

2.2 Fine-Grained DOM-Based Locking

To develop tailor-made XML lock protocols, Haustein and Härder (2008) have introduced a far richer set of locking concepts and developed a family consisting of four DOM-based lock protocols called the taDOM group. While MGL essentially rests on intention locks and, in our terms, *subtree* locks, these protocols additionally contain locking concepts for *nodes* and *levels*.

We differentiate read and write operations and rename the well-known (IR, R) and (IX, X) lock modes with (IR, SR) and (IX, SX) modes, respectively, to stress that subtrees (S) are locked. As in the MGL scheme, the U mode (SU in our protocol) plays a special role, because it permits lock conversion. Novel concepts are introduced by *node locks* and *level locks* whose lock modes are NR (node read) and LR (level read) in a tree which, in contrast to MGL, read-lock only a node or all nodes at a level, but not the corresponding subtrees. Together with the CX mode (child exclusive), these locks enable *serializable* transaction schedules with read operations on inner tree nodes, while concurrent updates may occur in their subtrees. While the remaining locks in Figure 1 coincide with those of the URIX protocol, we highlighted these three lock modes to illustrate that they provide a kind of tailor-made XML-specific extension.

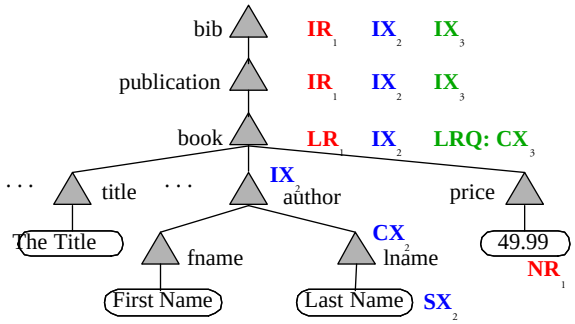

Figure 2: Application of the taDOM2 protocol.

Figure 1 contains the compatibility matrix for the basic lock protocol called taDOM2. To illustrate its use, let us assume that the node manager has to handle for transaction T_1 an incoming request *GetChildNodes()* for context node *book* in Figure 2. This requires appropriate locks to isolate T_1 from modifications of other transactions. Here, the lock manager can use the level-read optimization and set the perfectly fitting mode LR on book and, in turn, protect the entire path from the document root by appropriate intention locks of mode IR. After having traversed all children, T_1 navigates to the content of the *price* element after the lock manager has set an NR lock for it. Then, transaction T_2 starts modifying the value of *lname* and, therefore, acquires an SX lock for the related text node. The lock manager complements this action by acquiring a CX lock for the parent node and IX locks for all further ancestors. Simultaneously, transaction T_3 wants to delete the *author* node and its entire subtree, for which, on behalf of T_3, the lock manager must acquire IX locks on *bib* and *publication*, a CX lock on *book*, and an SX lock on *author*. The lock request on *book* cannot immediately be granted because of the LR lock of T_1. Thus, T_3 – placing its request in the lock request queue (LRQ: CX_3) – must synchronously wait for the LR lock release of T_1 on *book*.

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

Experimental analysis of taDOM2 led to some severe performance problems in specific situations which were solved by the follow-up protocol taDOM2+. Figure 2 reveals that conversion of LR can be very cumbersome, because individual node locks have to be set for T_1 on all children of *book*. As opposed to efficient ancestor determination of a node – delivered for free by prefix-based node labeling schemes (O'Neil et al., 2004) such as SPLIDs (stable path labeling identifiers) –, identification of its children is very expensive, because access to the document is needed to explicitly locate all affected nodes. By introducing suitable intention modes, Haustein and Härder (2008) obtained the more complex protocol taDOM2+ having 12 lock modes. The DOM3 standard introduced a richer set of operations which led to several new tailored lock modes for taDOM3 and – to optimize specific conversions – even more intention modes resulted in the truly complex protocol taDOM3+ specifying compatibilities and conversion rules for 20 lock modes.

Lock manager and lock tables are designed along the lines of (Gray and Reuter, 1993); its flexibility and efficiency are greatly influenced by the use of SPLIDs. The tree-organized locks are kept in a main-memory buffer whose layout is independent of the physical XML structures used (Section 3.1).

2.3 Results of a Lock Contest

To enable a true and precise cross-comparison of lock protocols, we implemented in our prototype system XTC (XML Transaction Coordinator (Haustein and Härder, 2007)) 5 variants of the MGL group together with 4 taDOM lock protocols. All of them were run under the same benchmark using the same system configuration parameters.

As it turned out by empirical experiments, *lock depth* is an important and performance-critical parameter of an XML lock protocol. Lock depth *n* specifies that individual locks isolating a navigating transaction are only acquired for nodes down to level *n*. Operations accessing nodes at deeper levels are isolated by subtree locks at level *n*. Note, choosing lock depth *0* corresponds to the case where only document locks are available. In the average, the higher the lock depth parameter is chosen, the finer are the lock granules, but the higher is the lock administration overhead, because the number of locks to be managed increases. On the other hand, lock conflicts typically occur at levels closer to the document root such that fine-grained locks (and their more expensive management) at levels deeper in the tree do not pay off.

In our lock protocol competition, we used a document of about 580,000 tree nodes (~8MB) and executed a constant system load of 66 transactions taken from a mix of 5 transaction types. For our discussion, neither the underlying XML documents nor

the mix of benchmark operations are important. Here, we only want to show the overall results in

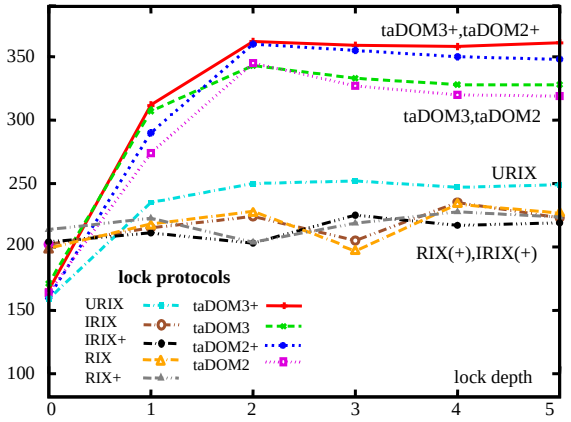

Figure 3: Number of committed transactions.

terms of successfully executed transactions (throughput) and, as a complementary measure, the number of transactions to be aborted due to deadlocks.

Figure 3 clearly indicates the value of tailor-made lock protocols. With the missing support for node and level locks, protocols of the MGL group provided only lock granules adjusted to the needs of DOM operations in a suboptimal way. As a consequence, they only reached about half of the taDOM throughput. A reasonable application to achieve fine-grained protocols requires at least lock depth *2*, which is also important for deadlock avoidance.

Hence, the impressive performance behavior of the taDOM group reveals that a careful adaptation of lock granules to specific operations clearly pays off.

3 STORAGE AND LOGGING

Besides the isolation mechanism, transaction support adheres to logging and recovery which ensures repeatability and rollback of all transaction-protected operations in regular and abnormal situations. Therefore, logging has to provide the needed data redundancy even after crashes or media failures. Further, log propagation rules (WAL and Commit rules) have to be observed (Härder and Reuter, 1983) often leading to I/O-intensive processing modes.

In general, saving I/O is the major key to performance improvements in DBMSs. Because write propagation of modified DB objects causes a large share of DB I/O and dependent log I/O, optimization of storage structures reduces log I/O at the same time. This is particularly true for storing, modifying, or querying XML documents, because they may contain substantial redundancy in the structure part, i.e., the inner nodes of the document tree. Therefore, optimization of XML storage representations may also be meaningful and show great promise for improving performance-critical transaction support.

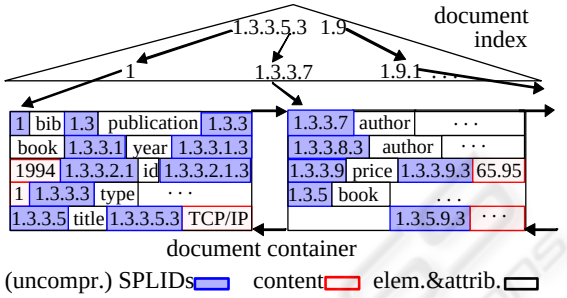

Figure 4: A completely stored XML document.

3.1 Complete vs. Elementless Storage

Efficient declarative or navigational processing of XML documents requires a fine-granular DOM-tree storage representation which is based on variable-length files as document containers whose page sizes varying from 4K to 64K bytes could be configured to the document properties. We allow the assignment of several page types to enable the allocation of pages for documents, indexes, etc. in the same container. To be flexible enough to adjust arbitrary insertions and deletions of subtrees, dynamic balancing of the document storage structure is mandatory. Fast indexed access to each document node, location of nodes by SPLIDs as well as navigation are important demands, too. As illustrated in Figure 4, we provide an implementation based on B*-trees which cares about structural balancing and which maintains the nodes stored in variable-length format (SPLID+element/attribute (dark&white boxes) or SPLID+value (dark&grey boxes)) in document order; this lends itself to effective prefix compression of the SPLIDs reducing their avg. size to ~20–30% (Haustein and Härder, 2007).

B*-trees – made up by the document index and the document container – and SPLIDs are the most valuable features of physical XML representation. B*-trees enable logarithmic access time under arbitrary scalability and their split mechanism takes care of storage management and dynamic reorganization. In turn, SPLIDs provide valuable lock management support and immutable node labeling such that all modification operations can be performed locally.

Because of the typically huge repetition of element and attribute names, getting rid of the structure part in a lossless way helps to drastically save

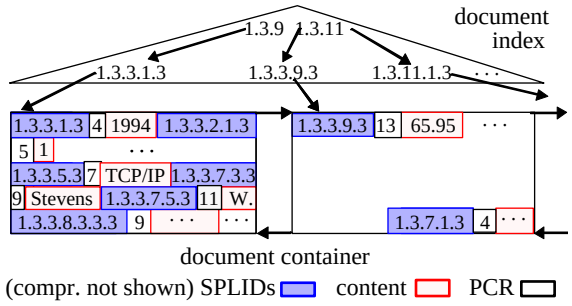

Figure 5: An XML document without elements stored.

storage space and, in turn, document I/O. As a consequence, log space and log I/O may be greatly reduced, too. The combined use of a so-called path synopsis (Goldman and Widom, 1997) storing only path classes and SPLIDs as node labels makes it possible to *virtualize* the entire structure part and to reconstruct it or selected paths completely on demand.

In an elementless layout of an XML document, only its content nodes are stored in document order – in the way as for the complete document. The stored node format is of variable length and is composed of entries of the form (SPLID, PCR, value) where PCR (path class reference) refers to a node in the path synopsis and enables the reconstruction of its entire path to the root. Compared to the sample of complete storage in Figure , the elementless XML fragment exemplified in Figure saves enormous space, but can, nevertheless, reconstructed without any loss.

3.2 Logging and Recovery

Minimizing log I/O is important for transaction optimization. Again, we could just consider the nodes of an XML document as records and "blindly" apply standard logging techniques, e.g., using physiological logging as a salient method (Gray and Reuter, 1993). Then, we had to write Undo/Redo log entries for all modifications in the structure and content part.

Instead, our XDBMS adheres to a three-level recovery providing hierarchically dependent DB consistency qualities: *block consistency*, *DML-operation consistency*, and *transaction consistency*. A very expensive method, a block-consistent state can be guaranteed by reserving a block in each container file for b*efore-image logging*. When propagating a modified block back to disk, a copy of it is first written to the before-image block. Because either the new or the old block is available, recovery can always rely on a *block-consistent* DB state. A more optimistic attitude would not apply such an overcautious method, but – if in extremely rare cases a corrupted block is detected – enforce archive recovery. Of course, such failure cases imply longer processing delays, but substantial log-rated I/O is saved in normal processing mode.

At the propagation level, the buffer manager applies *entry logging* for which each DML operation is decomposed into so-called elementary operations whose reaches are limited to a single block. Using log sequence numbers (LSNs), the log entries can be uniquely related to the blocks modified by these operations and the attached LSNs enable the decision whether or not the log entries have to be applied to the related blocks during recovery. Hence, restart can reconstruct in a kind of forward recovery (repeating history) a *DML-operation-consistent* DB state and for winner transactions even a *transaction-consistent* DB state. Finally, the transaction manager records the transaction boundaries and all inverse DML operations (logical *DML operation logging* is saving space and, thus, log I/O) to be prepared to rollback all loser transactions thereby executing DML operations on the reconstructed operation-consistent DB state.

3.3 Various Optimizations

The combined use of entry and DML operation logging already seems to require minimal log I/O in normal situations. Therefore, we focussed on operation- specific situations and improvement of related components to gain further optimization potential.

Reduced logging: For initially storing a document, stepwise rollback is not needed and complete rollback using entry logging is overly expensive. Therefore, logging of the block numbers involved is sufficient to empty the affected container pages.

Administration of Fix indicators: Blocks currently accessed by transactions have to be pinned in their buffer frames to avoid replacement. So-called *Fix marks* set by the requesting transactions indicate for the buffer manager that a block is not eligible for replacement. In the initial solution, these Fix marks were kept in the lock table where checking performed very poorly. Because search of replacement candidates is an extremely frequent task, a specialized structure recording the Fix state of all frames was added to the buffer manager.

Improved lock table management: Reimplementing the lock manager avoided static lock table allocation and large lock granules on the lock table itself. Using a pool of predefined lock request blocks

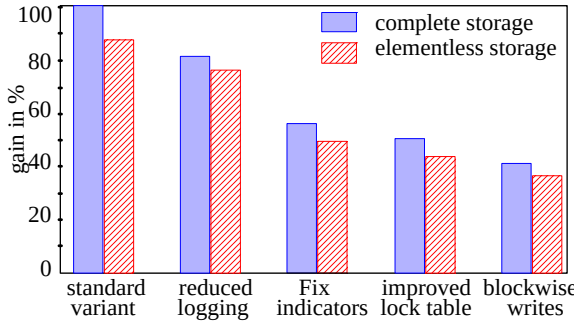

Figure 6: Effects of optimization measures.

and latches for hash table entries provided flexibility and minimal blocking. Because blocks in the DB buffer were also administrated by the lock manager, buffer management and the related entry logging immediately took advantage by this measure.

Blockwise writes: Single insertion of keys into the document index is required for document update. Document creation, in contrast, enables specialized block handling outside the DB buffer. B-tree blocks of the index can be entirely filled before transferring them to the buffer. In turn, entry logging and block propagation can be optimized by the buffer manager.

We have implemented these features in XTC and derived some indicative results for a frequent use case, i.e., the storage of a document (113 MB) including indexes, auxiliary structures, and logging provisions (Sommer, 2007). To enable comparison, we have normalized all results to the cost of the complete standard structure (100%). Hence, the difference to 100% is the saving (gain) achieved by a specific measure. In the results shown in Figure 6, we have added our optimization measures step by step such the plain effect of the indicated optimization is the difference to the previous quantities (bars). While elementless storage gains >10% for the standard variant, the aggregated saving of all measures applied reaches the level of ~40%, i.e., optimization reduces the initial costs by ~60%.

4 SUMMARY AND OUTLOOK

XTC is – to the best of our knowledge – the only (freely accessible) XDBMS offering full-fledged transaction services. It was used to explore ACID properties and served for all comparative experiments. We outlined the state of our work in XML concurrency control as well as logging and recovery. Our taDOM approach guarantees serializability and transaction isolation in multi-lingual XDBMSs and even of multi-lingual transactions. Compared to less adjusted lock protocols, the impressive performance gains of the taDOM group reveal that a careful adaptation of lock granules to specific operations clearly pays off. Furthermore, the lion's share of XML processing costs is caused by all sorts of document I/O including logging can be reduced by a variety of measures.

Currently, we are working on lock manager extensions supporting lock escalation and tailor-made locking on index structures. Another hot topic is energy efficiency in XDBMSs including the use flash memory, buffering and deferred updates, specialized logging techniques and group commit.

REFERENCES

Document Object Model (DOM) Level 2 / Level 3 Core Specifications, W3C Recommendation, http://www.w3.org/DOM/ (2005)

Goldman, R. and Widom, J.: DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. *Proc. VLDB*: 436-445 (1997)

Graefe, G.: Hierarchical locking in B-tree indexes. *Proc. National German Database Conf. (BTW 2007)*, LNI P-65, Springer, 18–42 (2007)

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

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

Härder, T. and Reuter, A.: Principles of Transaction-Oriented Database Recovery. In *Computing Surveys* 15(4): 287-317 (1983)

Haustein, M. P. and Härder, T.: An Efficient Infrastructure for Native Transactional XML Processing. In *Data & Knowl. Eng.* 61:3, 500–523 (2007)

Haustein, M. P. and Härder, T.: Optimizing lock protocols for native XML processing. In *Data & Knowl. Eng.* 65:1, 147-173 (2008)

Mohan, C.: ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes. *Proc. VLDB*: 392-405 (1990)

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

Sommer, T.: Efficient and Transaction-Protected Storage of Document Collections in XML Database Systems (in German). *Master Thesis*, TU Kaiserslautern (2007)

Weikum, G. and Vossen, G.: *Transactional Information Systems.* Morgan Kaufmann Publishers (2002)

XQuery 1.0: An XML Query Language. http://www.w3.org/ TR/xquery/ (2007)