A Scalable Framework for Serializable XQuery

Sebastian Bächle and Theo Härder
Department of Computer Science
University of Kaiserslautern, Germany
{baechle, haerder}@cs.uni-kl.de

Abstract

This paper focuses on an aspect that is widely neglected in native XML database management systems: support for concurrent transactional access. We analyze the isolation requirements of the XQuery Update language and disclose typical sources of anomalies of various query processing strategies. We also present extensions to our proven XML lock protocol, which allow us to exploit dynamic schema information for query processing and protects us against XML-specific "schema phantoms". All concepts shown were implemented in our research prototype resulting in a scalable framework for serializable XQuery.

1. Introduction

After more than a decade of native XML database research, various storage, indexing, and query processing techniques have matured, and first production-level systems are available on the market. These systems provide satisfactory query response times, reliable storage, and scalability to large data volumes. In contrast to conventional database systems, however, one aspect that is typically taken for granted is widely neglected: support for ACID transactions.

Vendors bypass this scalability bottleneck by solely focusing on use cases where concurrent access is just not intended: On the one extreme, documents are stored once and are never or only sparsely updated or, on the other extreme, data is arranged in large collections of small, independent documents, too tiny to think about reasonable concurrency. The whole spectrum in between is left open, throwing away opportunities to enable new types of applications.

Our long-term objective is to close this gap and to develop concepts and architectures that finally leverage the strengths of both worlds: databases (transactional, scalable, reliable) and XML (semi-structured, schema-agnostic, portable). As a major system aspect to approach this goal, efficient transactional multi-user capabilities are needed that scale with both number and size of documents.

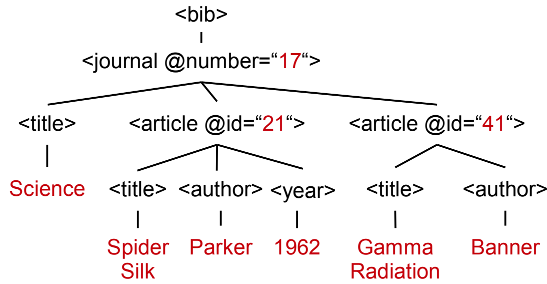
1.1. Problem statement

In the first place, transaction isolation for XML has to cope with the same challenges as traditional relational systems. It must provide stability for data read and it must protect against phantoms, i.e., preserve stability of predicates evaluated like, e.g., range scans over content. Queries and updates for semi-structured data impose here some new challenges not known so far. In the following, we point out the most important aspects:

On the one hand, there is the ordered, hierarchical nature of XML, which leads to complex dependencies between the "data atoms", i.e., the nodes of the document tree. Thus, serializability requires that structural updates are carefully isolated and do violate neither vertical nor horizontal relationships seen by concurrent transactions.

On the other hand, when using XQuery-like languages, it is particularly difficult to predict what data will be accessed by a query. As a consequence, it is generally impossible to determine in advance – just by looking at the statements – whether two given queries will conflict with each other or not. One might assume that the latter is a consequence of the absence of schema information. However, as we will show, even the presence of a schema – pre-defined or dynamically derived from the actual data – may not be sufficient. Let us assume that two transactions *T1* and *T2* access a document as depicted in Figure 1: *T1* queries the subtree of the "*Science*" journal, while *T2* inserts a small *year* fragment into the subtree of article 41. Obviously, *T1* and *T2* conflict with each other and concurrent access is prohibited. However, the statements themselves contain no hint about that there *may* be a conflict.

Using schema information, we could at least derive that there is a *conflict potential*. Unfortunately, knowledge about potential contention within a document is not sufficient to establish maximal permissive concurrency control. This is similar to the relational world, where some conflict potential is already given whenever transactions concurrently access and modify the same table. High concurrency can only be achieved when concurrency control is applied to




```
T1: for $j in //journal
    where $j/title/text()="Science"]
    return $j

T2: insert node <year>1962</year>
    into //article[@id="41"]
```

Figure 1. Conflicting transactions

the smallest meaningful granule, i.e., tuples in the relational world and nodes in the XML world.

A major difference between relational tuples and nodes in an XML tree is the amount of information that each data atom represents. A single XML node usually carries much less information – namely a name and a value – than a single tuple, which may have tens of attributes or even more. Further information is encoded in the XML document structure, typically by grouping related nodes in subtrees. Consequently, queries and updates often refer to variable-size subtrees. A fact that has to be naturally supported by XML concurrency control.

The above observation does not necessarily mean that transactions are always interested in completely expanded subtrees. Instead, queries often address subtrees by certain predicates and then read only smaller parts of them – a great chance to increase concurrency. For example, if the return clause of transaction *T1* would refer to `$j/article/author` instead, it would not conflict with the insertion of *T2* and both requests could be processed in parallel.

To summarize, we can state the following key observations:

- Read/write or write/write dependencies between two transactions can only be decided in a meaningful way at the instance level, i.e., the document.
- High concurrency requires small granules of isolation and the smallest granule available in XML is a node.
- Mostly, subtrees are the logical *target granule* for queries and updates, but they vary in size and may be nested.
- Typical usage allows concurrent reads and writes even within a single subtree.
- XML's ordered tree structure adds new dimensions for the evaluation of predicates and, thus, for the appearance of phantoms.

Taking these observations into account, it becomes obvious that design and implementation of correct and scalable concurrency control for XQuery is a challenging task. Nevertheless, we believe that it can be reduced to a small set of clear-cut measures that can be efficiently realized and improve the overall value of native XML DBMS.

1.2. Contribution

In this paper we analyze the isolation requirements of XQuery Update and disclose sources of anomalies during query processing. Our findings take general XML query processing strategies into account and, thus, are applicable to a great variety of systems.

We also present new extensions to our proven XML lock protocol taDOM providing effective protection against the XML-specific "schema phantoms". All concepts were implemented in our native XML DBMS research prototype and provide a scalable framework with guaranteed serializability for XQuery.

The remainder of this paper is organized as follows: We analyze the update primitives of XQuery Update and their isolation requirements in Section 2. Section 3 briefly introduces XML query processing and investigates general caveats and sources of phantoms which different query evaluation strategies have to observe. We present our approach implemented in our native XML DBMS prototype XTC in Section 4, and review related work in Section 6. Finally, Section 7 summarizes the paper.

2. XQuery update

The XQuery Update Facility is an XQuery extension for declarative updates of XML. Similar to SQL, it allows to query "nodes of interest" for which an update like, e.g., value change or insertion of a new subtree should be performed. Before the actual updates are applied, all of these so-called *target nodes* are collected together with the requested update operation in a *pending update list*. In a second phase, the list is checked to, e.g., eliminate duplicate deletes of a single node etc. and, finally, the updates are performed. Because of this two-phased processing, XQuery updates are called snapshot-based. Note that this defines only the *semantics* of update expressions and that XQuery neither requires or nor favors snapshot isolation in concurrent environments.

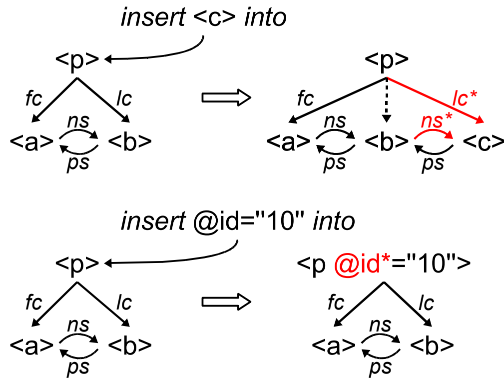


Figure 2. Affected properties of inserts

The extension defines five kinds of update expressions: *insert*, *delete*, *replace*, *rename*, and *transform*. The latter, however, applies updates only to copies of the target nodes. As these copies are per definition private to a transaction, we must not consider *transform* expressions as an issue for concurrency control.

Trivially, transactions must be guaranteed that the state of every read node is committed and stable; of course, modified nodes do not become visible to others before commit. In the following, we will focus on implicit consequences of updates for other nodes as potential sources of phantoms.

2.1. Update primitives

The *insert* expression inserts a node or subtree relative to the specified context node. The insertion position can be precisely specified for the new previous or next sibling (*before*, *after*) or the new first or last child (*first*, *last*). If sibling order does not play a role for the application, one may also specify "any new child" (*into*) to allow an actual implementation to choose the "optimal" insertion postion with regard to, e.g., storage consumption or concurrency.

From the perspective of transaction isolation, all flavors of *insert* operations modify only a fixed set of basic properties of adjacent nodes. If the root of the new subtree is an element or a text node, then the *nextSibling* or *previousSibling* properties of the right or left sibling and, depending on the insert position, the *firstChild* and *lastChild* properties of the parent are affected. For our discussion, we need not distinguish between insertion of a single node and a whole subtree, because descendants of the root must be inaccessible to other transactions, too. If the new node is an attribute, only the correspondingly named *attribute* property of the parent element is affected. Figure 2 illustrates the changed foreign node properties for these two cases.

Insertions affect many more properties influencing the outcome of concurrent queries like, e.g., position of all following siblings, containment for all ancestors, preceding axes of all following nodes, etc. But, they are transitively derived from the mentioned ones. Therefore, it is sufficient to ensure that inserts do not violate general repeatable-read and visibility rules for the five basic properties and that the evaluation of other structural predicates regards these transitive dependencies. The same observation holds for *delete*, which simply deletes the subtree rooted at the context node.

A *replace* expression distinguishes between the replacement of a node/subtree with another and the update of a node's value. The former logically translates into a sequence of *delete* and *insert*. The latter does not affect other nodes, because node identity is preserved and tree structure is not modified. Accordingly, special care is not necessary.

Finally, *rename* changes the name of element and attribute nodes. For attributes, this operation equals to a deletion followed by an insertion under the new name. Renaming elements is similar to value updates for text and attribute nodes. It is a local operation and does not modify the tree structure. Although it changes the path of all descendants, a heavily used concept in queries, we must not impose further restrictions, because the rationale of transitive properties holds.

2.2. Phantoms

Phantom anomalies arise when the result set of a previously evaluated predicate changes due to concurrent modifications [8]. We already mentioned that a node embodies only three kinds of information: name, value and its position in the document hierarchy[1]. Accordingly, we can classify three different kinds of phantoms that may appear.

Content-predicate phantoms step up when, e.g., attribute values or text nodes are modified and fall into a concurrently evaluated range scan. *Name-predicate* phantoms appear when nodes are concurrently renamed, inserted, or deleted, which fulfill a queried name predicate. Finally, *structural-predicate* phantoms arise, e.g., when a transaction navigates from a node to its first child and another transaction concurrently inserts a new first child in front of it.

In practice, these kinds of phantoms would typically appear in combination. Consider a transaction *T1* evaluates query `//a/b/@c > 5` and a second transaction *T2* adds a new attribute `c` with value `60` to any node with label `b` and a parent `a`. If `T1` now re-evaluates the query, the new attribute appears as a phantom in all kinds of categories.

The query of transaction *T1* describes a complex predicate. As there are plenty of ways to evaluate it, we cannot easily locate a single point in a system where phantoms originate. Therefore, we will investigate common query evaluation strategies for XML and distill a general principle how to prevent the emergence of phantoms.

[1]Note that we omit namespaces, types, etc. for the sake of simplicity as they do not influence the main points of this paper.

3. Query processing

XQuery engines can be categorized by their processing strategy into four groups: *streaming*, *navigational*, *relational*, and *native*, which embraces concepts of all former in conjunction with native XML techniques like indexing and metadata usage. The strategy is typically determined by the capabilities of the underlying storage system, if any, and non-functional requirements like, e.g., memory footprint. From the perspective of transactional concurrency, it is most important that an engine touches as few data as possible because the isolation aspect requires to keep accessed data stable until commit thereby limiting concurrency.

Pure streaming and navigational engines are mostly stand-alone solutions or directly embedded in applications. They work on a per-file basis, which implies that they always access whole documents making concurrent transactional read and write access impossible. Relational engines run on top of standard RDBMSs and translate XQuery to SQL, having documents shredded into relational tables. Consequently, they are per se ACID compliant and allow concurrent queries and updates of the tables with the shredded documents. The underlying concurrency control mechanisms, however, are blind for the actual semantics of XML and XQuery, and potential concurrency is jeopardized. Achievable concurrency depends on the shredding scheme and the degree to which it allows to use column-based indexes to reduce the number of tuples that must be touched and to perform XML-level updates with low side-effects on the shredded data.

Native engines can draw from plentiful techniques and data access alternatives to tackle a query. In addition with context knowledge about the inherent data model and query properties, they allow for very efficient query processing. A desired side-effect for our purposes is that efficient evaluation most often implies minimal access to a document. Accordingly, we focus on native engines and analyze pro's and con's for concurrency of native query processing algorithms.

3.1. Basic concepts

XML engines base on the traditional stages of query processing: translation, planning and optimization, and execution. First, a query is parsed, checked, and translated into an internal representation, a so-called *query plan*. In the second stage, the logical query plan is transformed into a – hopefully – efficient physical query plan, which is finally processed in the third stage. The whole process is a very complex topic, which we do not want to detail here. For our purposes, it is sufficient to look at the physical query plans, because they define how the data is accessed and, accordingly, where isolation properties might be violated.

A physical query plan is a data flow graph with a single output node and several input nodes. The output node returns the query result, typically serialized into a string, and the – possibly empty – pending update list, which has to be processed. Input nodes can be any type of physical access to documents, but also sources for "non-physical" input like node constructions, arithmetic and logical expressions, literals, function calls, etc. Inner nodes, finally, represent the actual query processing logic in form of operators or algorithms. Amongst them are traditional ones like select, join, and sort, but also – depending on the platform – XML-specific ones like structural join, twig join, or other primitives for the nested-loops-like *for* construct of XQuery.

General processing logic and various path processing algorithms are already quite complex. But even worse, interdependencies between them, introduced by their combination in a query plan, are even inscrutable. Node identifiers, for example, mostly encode structural relationships and are used by algorithms to compute identifiers of related nodes. Some systems include information to directly compute the whole path for each node and, thus, allow to evaluate many predicates for theirs ancestors without actually accessing them. In such settings, many predicates over paths and content will be evaluated in manifold constellations, making it unfeasible to enumerate all potential sources of phantoms.

The above shows that any plans to implement concurrency control by reasoning about the semantics of the query must necessarily fail. The complexity, however, only draws the attention from a clear-cut, even *trivial fact*: Independent of the query, it will be sufficient to protect both explicitly and implicitly gathered information from concurrent modification until commit, whenever a document is physically accessed. Explicit information is identity, name, and value of a node. Implicit information is all meta knowledge gained *and* exploited during query processing. If the requirement for repeatable read for both kinds of information is met, phantoms cannot occur, because even complex predicates can only be evaluated out of this basic data.

The roadmap for the realization of a waterproof isolation concept is now clear. We have to identify all alternatives for physical document access (*access path*) and determine how much information about the document a particular implementation really delivers – with a close eye on the critical properties and types of information identified in Section 2. Then, we have to install efficient, yet maximal permissive measures to protect the returned data from modifications through concurrent transactions.

3.2. Access paths

Availability of several access paths allows a system to choose the cheapest combination to answer a query. If only navigational access is available to answer a query

`//a/b/@c>5`, for example, the document must be traversed node by node to find each match. If, however, a special XML index that contains all attributes on the path //a/b is available, all candidate attributes may be read efficiently from the index and only the conditions `name=c` and `value>5` must be checked requiring document access, and so on. Generally, we observe that *the more complex the information provided by an access path is, the less nodes have to be touched, and, in turn, the higher is the concurrency achievable.*

Without – here irrelevant – consideration of physical properties, the vast number of XML storage and indexing structures can be classified into specific access path classes[2]. First and most important is the *document* store itself. It allows for direct node access, navigation in the tree structure, and bulk reconstruction of document fragments. Accordingly, in addition to its role as "node store", the document store can be seen as an index for tree structure predicates. The two other kinds of information carried by nodes, values and names, can be indexed by special content and *element/attribute* indexes, respectively. They map the property of a node to an identifier, which can be used to look up the related node in the document store. Finally, advanced *path* and *CAS* (content and structure) indexes combine all three kinds of information and allow to evaluate complex path predicates with minimal effort.

With this great variety, selection of appropriate access paths to feed operators in a query plan is challenging. Identification of path and other predicates suitable to be processed with a powerful path or CAS index, for example, is not a simple task. Nevertheless, demand for high concurrency encourages to exploit them whenever possible to reduce the number of nodes to be accessed.

4. Concurrency control in XTC

XTC is our research platform for native XML database concepts, such as storage, indexing, query processing, transaction isolation and crash recovery [13]. It uses a flexible storage layout and supports the query engine with powerful indexing capabilities. XTC also provides full ACID transactions based on the tailored XML lock protocol taDOM [12], which initially targeted only to navigational APIs. In the following, we will give a résumé of its key aspects, before we extend its scope to full XML query support.

[2] In practice, the classes are not strictly disjoint. A specific implementation might fall into several classes and may provide all or only a subset of the access operations of a class.

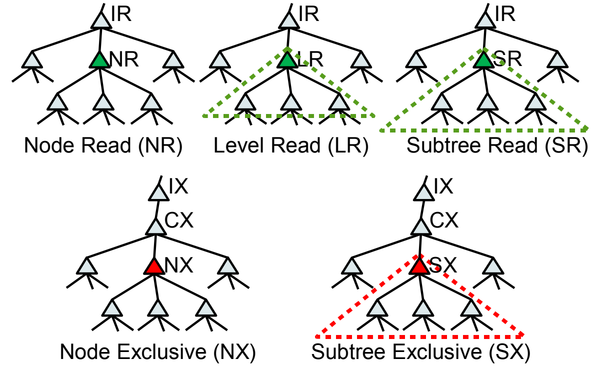


Figure 3. Special taDOM lock modes

4.1. taDOM

The key of taDOM is to transfer the idea of hierarchical locking [7] to document trees and to enrich the protocol with lock modes aligned to the permissible concurrency of XML updates. The protocol provides transactional isolation for the existing structure and content. According to the general principle, a suitable lock and intention locks for all ancestors must be acquired, before a node is accessed. To master the acquisition of intention locks efficiently, we depend on prefix-based node labels like DeweyIDs or OrdPaths [9], because they allow for the computation of all ancestor node labels without further physical access. However, such labeling schemes are already widely established in XML DBMS because of their benefits for query processing.

taDOM yields its high concurrency with lock modes that focus on minimal protection requirements of XML-specific operations. As depicted in Figure 3, it distinguishes between shared access for single nodes, tree levels, and whole subtrees, and exclusive access for single nodes and subtrees. With these modes it is, e.g., possible for a query to iterate over a node and all its children, while, at the same time, nodes and subtrees in the subtrees of its grandchildren are updated. Lock overhead can easily be controlled by switching lock coverage from the very fine node level to the coarse subtree level.

Whenever a transaction physically or logically navigates from a node to a related one, a structural predicate is evaluated that has to be kept stable. taDOM in conjunction with prefix-based node labels already delivers this protection for many predicates like, e.g., ancestor/descendant, following/preceding, following-/preceding-sibling, etc. However, the danger of phantom inserts and deletes remains in three cases: navigation from a node to its first and last child respectively, navigation between direct siblings, and navigation from an element to one of its attributes. If you recall the discussion in Section 2.1, these are the identified critical

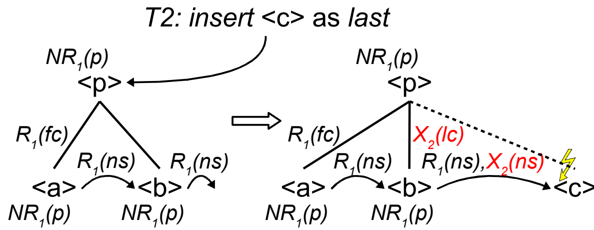


Figure 4. Edge locks

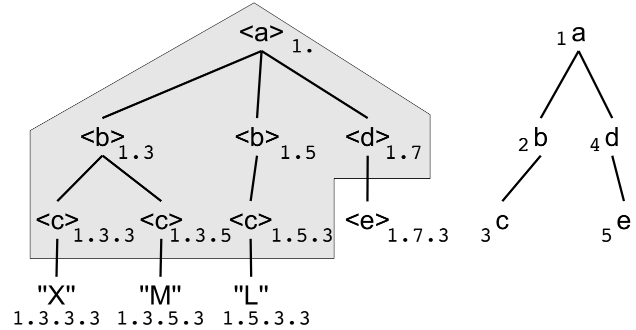


Figure 5. Document and path synopsis

foreign node properties affected by update operations.

To overcome this issue, so-called *edge locks* [11, 15] were introduced, which are simply fixed-predicate locks that must be acquired whenever one of these properties is read or modified. The example in Figure 4 illustrates the concept. Transaction *T1* starts at node *p*, navigates to its first child and to the next sibling, acquiring shared locks for each node visited and each edge passed. Transaction *T2* now attempts to append a new node *c* requiring exclusive locks for affected foreign node properties, i.e., edges. The request for the *lastChild* edge of *p* can be granted, because the lock is free. The request for the *nextSibling* property of *b*, however, is incompatible with the shared request of *T1* and *T2* must be blocked.

The initial solution proposed the processing of edge locks orthogonal to node locks with simple read, write, and update lock modes. Attribute edges were handled also separately with so-called *phantom locks*. Having learned from efficiency experiments, we merged our concepts and model all edges as *nodes with pre-defined positions* amongst the actual nodes to profit from lock escalation heuristics.

In various experiments, we proved that taDOM not only provides compelling concurrency but can also be implemented efficiently [1, 2, 14]. We also addressed advanced topics like deadlock prevention through update lock modes and dynamic lock escalation. The confinement to logical document trees makes the protocol independent of the underlying document store – except for the need of prefix-based node labels. It must only provide atomicity for concurrent read and write operations and must also take implications of sophisticated storage layouts into account for transaction rollback and crash recovery. Although these are interesting aspects, we will not discuss them further, because implementation details are beyond the scope of this paper.

4.2. Indexes

Physical storage in XTC omits redundant element nesting and stores only leaf nodes of the document tree [10]. The ancestor path, i.e., the inner structure of a document, can be computed at any time with a leaf's DeweyID and its PCR (path class reference) – an integer identifying the node's path in a structural summary of the document. The structural summary, called path synopsis, is itself a small tree reflecting all unique paths within a document. Figure 5 shows a DeweyID-labeled document and the corresponding path synopsis.

By combining DeweyIDs and PCRs, we can also create sophisticated indexes, which may be carefully adjusted to document properties and query requirements [16]. Using permutations of PCR, DeweyID, and, optionally, content of a node, we have various opportunities to build key/value pairs for different types of path and CAS indexes. They can be precisely scoped with path patterns and content type specifications to specific node sets. The path synopsis is used to identify the set of PCRs matching the path patterns, which greatly simplifies creation, use, and maintenance of an index.

Additionally, we can employ a two-staged element index, which allows for efficient access to all elements with the same name in document order. It is logically the same as a path index for the pattern `//*`, but has different physical properties. Of course, we can use this technique also for attribute names or exploit PCRs to index only a subset of all node names. Finally, XTC also supports plain conventional content indexes, which map content and attribute values to DeweyIDs.

For concurrency control, a big advantage of all our four index types is that they are realized with standard B*-trees. This enables us to employ standard index locking techniques, which do not only lock the index entries for repeatable reads, but also the gaps between them to prevent phantom inserts and deletes. Note that even renaming elements does not violate consistency of our PCR-based indexes, because a rename implies that the PCRs of all descendants also change, and, accordingly, it results in normal index updates.

Our current implementation uses ARIES key/value locking [17] for all index types, which uses separate locks on index keys. Of course, any other index locking approach preserving serializability and further sophisticated measures to

reduce the lock overhead are also applicable [4, 6]. In future work, we will work on new strategies to harmonize the interplay between index locking and taDOM to further reduce lock overhead.

4.3. Schema phantoms

The path synopsis is undoubtedly the central data structure for all document accesses. Therefore, we rely on its performance and avoid to burden it with heavy-weight concurrency control. In contrast, we relaxed some properties to increase throughput. First, a path synopsis must not necessarily be minimal, i.e., it may contain paths without counterparts in the document. In other words, a path synopsis only grows and deletes must not take care if the last instance of a path in the document is deleted; stale paths may be removed by isolated maintenance tasks. Second, a path synopsis contains no payload data and, thus, newly created paths may be shared directly with other transactions. Once a new path is created and a PCR is assigned, it is immutable.

While the above is great for minimal synchronization overhead in the data structure, we cannot completely dispense transactional concurrency control for a path synopsis. As a kind of dynamic schema, it lends itself as a powerful vehicle for query optimization and, especially, index selection. During query planning, we match path expressions against the path synopsis to get a set of matching PCRs. This set can be used by the optimizer to choose appropriate indexes to answer the query. A PCR set, however, reflects only a snapshot of the current document structure, and this may lead to so-called *schema phantoms*.

Assume a transaction *T1* matches the path expression `//c` against the path synopsis in Figure 5. The current PCR set is $\{3\}$, and the optimizer may choose to read all matching nodes from a path index `/a/b/c`. As only nodes on this path will be covered and protected by the index, a transaction *T2* may create a new path `/a/d/c` with PCR 6 in the document and commit. If *T1* re-evaluates the path `//c`, the PCR set is now $\{3, 6\}$ and the optimizer must not use the same path index again. Instead, the document may have to be scanned and nodes inserted by *T2* will appear as phantoms.

The problem of schema phantoms arises whenever the path synopsis is "misused" to make statements about the non-existence of certain paths. In the above example, the PCR set was used to justify the application of an index, which is in general too narrow to answer the desired path expression. Similar problems arise when empty PCR sets are taken as indication that whole branches of a query plan must not be executed, because they are expected to return no result. Accordingly, we can say that the phenomenon of schema phantoms already appears in the planning phase and not in the execution phase.

There are two possible solutions to the problem. The simplest way, the optimizer is never allowed to choose too narrow indexes or to cut query branches although the path synopsis indicates that it is reasonable. In a real environment, however, we can observe that a path synopsis grows only infrequently and, thus, it is desirable to leverage all indexes available to the maximum extent. We developed a straightforward solution. Whenever a transaction matches a path expression against the path synopsis, a shared lock for the expression is acquired. Transactions creating new paths must probe these locked expressions matching the new path with instant requests for exclusive locks – a type of lock that is directly released at the moment it is granted. This way, writers are delayed until all readers that might have missed the new path have ended. As schema extensions are rare situations and search for locked expressions matching the new path can be easily truncated, general impact on writers is very low. Note also that matching queries will never have to wait for a shared *expression lock*, because exclusive locks are instantly released when granted.

5. Insightful experiment

We can illustrate the effects of a specific query evaluation strategy on our locking protocol using a distinct experiment. We compared four different strategies for query *Q1* of the widely-used XMark benchmark [21]:

```
let $auction := doc("auction.xml") return
for $b in
  $auction/site/people/person[@id = "person0"]
return $b/name/text()
```

The *Scan* plan evaluates the query using a single scan over the whole document. *ElementIndex* constructs the path `/site/people/person` with a structural join over three element index scans for the respective element names. The attribute predicate and the final result construction are evaluated using navigation. Finally, *PathAndContentIndex* and *CASIndex* identify person elements using a join over a path index `/site/people/person` and content index scan for "`=person0`" and a CAS index `/site/people/person/@id` for "`=person0`", respectively. Both plans construct the final result using navigation. We executed all plans for XMark documents of size 110KB, 1.1 MB, 11MB, and 110MB in XTC on a server with 4 quad-core Intel Xeon 2,66GHz and 4GB memory. XTC was configured with 64KB pages and a 64MB buffer and lock escalation was turned off to investigate the access behavior of the different plans.

As expected, the query execution time given in Figure 6 directly correlates with the number of nodes each plan accessed. The scan scaled poorest with document size although the result size remained stable with one qualified node. The *ElementIndex* plan performed much better but

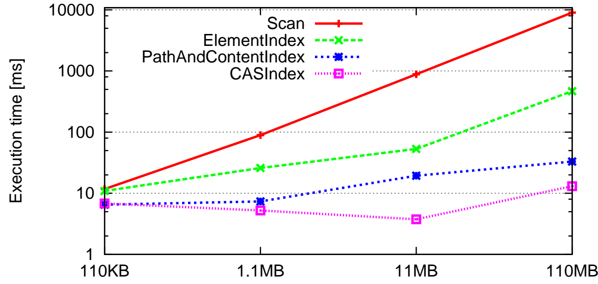


Figure 6. Query execution times

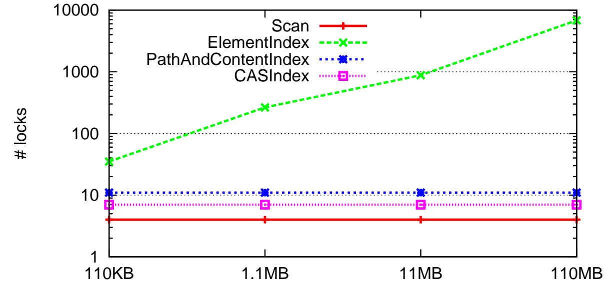


Figure 7. Number of requested locks

response time also degraded slowly, because the attribute predicate must be checked for all persons qualified by the structural join. The response times of the last two plans remained constantly low in the range from 5 to 30 ms for all sizes. Their big advantage is the ability to evaluate the highly selective content predicate with an index access first; this dramatically reduces the intermediate result size. The *CASIndex* plan only needs a single access to the CAS index to evaluate the qualifying path expression.

A look at the number of locks acquired by each execution strategy in Figure 7 might be a bit suprising at first. The *Scan*, although touching always each node, constantly acquired only 4 locks. Three were acquired to locate the document in the database[3]; and the document root node was locked with a shared subtree lock (*SR*) for the scan. Consequently, concurrency is limited by scan-based plans to shared access. The overhead of *PathAndContentIndex* and *CASIndex* is also constant. In contrast to the *Scan*, however, their locks only cover the actually accessed nodes and allow concurrent modifications everywhere else in the document. In this discipline, *ElementIndex* reveals its undesirable access behavior. Although this strategy generally performs adequately and is, therefore, widely used in native engines, it "spreads" access, i.e., locks over the whole document. Accordingly, this strategy tends to increase both runtime and memory consumption and also reduces concurrency more than necessary.

6. Related work

To the best of our knowledge, only a small fraction of the work on XML concurrency control copes with full support for XQuery and XQuery Update. Instead, most proposals focus only XPath, subsets of XPath, or home-brew path expression languages. We do not consider the latter here because they lack practical relevance and restrict ourselves to the few XQuery-capable solutions that found their way into products or prototypes.

[3]Like relational systems store their metadata catalogs in tables, XTC uses a an XML document for metadata.

MonetDB/XQuery is a powerful system implemented on top of the column-store MonetDB. It uses a pre/post encoding scheme to shred documents into relations which implies their reorganization after structural updates. Its answer to reduce maintenance effort and to increase throughput under concurrent updates is an update-friendlier table layout and a combination of shadow paging together with page-level and document-level read/write locks [3]. Commutative delta operations help to avoid concurrency bottlenecks when encoding changes have to be propagated up the document tree. The mechanism described provides snapshot isolation, but, contribution [3] neither states on concurrency achieved nor deadlock threats raised by page-level locks.

DGLOCK [5] and (S)XDGL [18, 19] are hierarchical lock protocols applied to structural summaries similar to our path synopsis instead of document nodes. SXDGL additionally uses MVCC to isolate readers with snapshot isolation. While these approaches promise minimal lock overhead, they come with some practical shortcomings. They require general predicate locks, e.g., on content, to reduce contention on the nodes of the summaries. This leads to serious performance penalties, when lock compatibility decisions require document access to check if predicates overlap. Further, identification of paths and, accordingly, choice of correct yet sufficiently permissive locks in advance is a complex issue and only manageable for simple queries.

Optimistic concurrency control for XQuery, finally, has, so far, only been proposed in [20]. However, the approach is quite complex and has never been proved to scale to serious data volumes.

7. Conclusions

Poor support for intra-document concurrency in native XML DBMSs is an unnecessary restraint. Nowadays, the way documents, i.e., semi-structured, hierarchical data, must be used and organized is determined by the DBMSs and not – as it should be – by the applications.

Analysis of general isolation requirements of XQuery and concurrency pitfalls in native XML query processing

lead to a central observation: Serializable, phantom-free, and highly concurrent query processing can be achieved when the problem is reduced to the provision of *maximal permissive concurrency control* for all data access paths within a system. One must ensure that the data delivered as well as the implicitly exhibited information is protected against concurrent modifications.

This requirement for repeatable read couples the goal of high concurrency directly with the goal to touch as few data as possible during query processing. XML's tendency to group and query related information in subtrees supports this in a natural way. Accordingly, we emphasize the value and encourage the use of efficient path indexes and CAS indexes to profit from both fast query processing and increased concurrency. With their expressiveness, relevant subtrees can be identified very quickly and further processing can be scoped to avoid scattered document access, which also reduces concurrency.

Our solution is taDOM, a hierarchical XML lock protocol, in conjunction with standard index locks. It embraces XML specifics like subtree locality, path processing, and common types of XML indexes, but does not affect the degrees of freedom which the query engine can utilize. We have implemented all concepts in our prototype to accomplish a real concurrent and guaranteed phantom-free native XML DBMS. Experiments also confirm that our concepts effectively increase concurrency and can easily trade overhead off against parallelism.

References

[1] S. Bächle, T. Härder: The Real Performance Drivers Behind XML Lock Protocols. Proc. DEXA, LNCS 5690, 38-52 (2009)

[2] S. Bächle, T. Härder, M. P. Haustein: Implementing and Optimizing Fine-Granular Lock Management for XML Document Trees. Proc. DASFAA, LNCS 5463, 631-645 (2009)

[3] P. A. Boncz, J. Flokstra, T. Grust, M. Keulen, S. Manegold, K. S. Mullender, J. Rittinger, J. Teubner: MonetDB/XQuery-Consistent and Efficient Updates on the Pre/Post Plane. Proc. EDBT, 1190-1193 (2006)

[4] D. Lomet: Key Range Locking Strategies for Improved Concurrency. Proc. VLDB, 655-664 (1993)

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

[6] G. Graefe: Hierarchical locking in B-tree indexes. Proc. BTW, LNI P-65, Springer, 18-42 (2007)

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

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

[9] T. Härder, M. P. Haustein, C. Mathis, M. Wagner: Node Labeling Schemes for Dynamic XML Documents Reconsidered. Data & Knowledge Engineering 60(1), 126-149 (2007)

[10] T. Härder, C. Mathis, K. Schmidt: Comparison of Complete and Elementless Native Storage of XML Documents. Proc. IDEAS, 102-113 (2007)

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

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

[13] M. P. Haustein, T. Härder: Optimizing Lock Protocols for Native XML Processing. Data & Knowledge Engineering, 65(1), 147-173 (2008)

[14] M. P. Haustein, T. Härder, K. Luttenberger: Contest of XML Lock Protocols, Proc. VLDB, 1069-1080 (2006)

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

[16] C. Mathis, T. Härder, K. Schmidt: Storing and Indexing XML Documents Upside Down, Computer Science – Research & Development 24(1-2): 51-68 (2009)

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

[18] P. Pleshachkov, P. Chardin, S. O. Kuznetsov: XDGL: XPath-Based Concurrency Control Protocol for XML Data. Proc. BNCOD, 145-154 (2005)

[19] P. Pleshachkov, S. O. Kuznetsov: SXDGL: Snapshot Based Concurrency Control Protocol for XML Data. Proc. XSym, 122-136 (2007)

[20] Z. Sardar, B. Kemme: Don't be a Pessimist: Use Snapshot-based Concurrency Control for XML. Proc. ICDE, 130 (2006)

[21] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, R. Busse. XMark: A Benchmark for XML Data Management. Proc. VLDB, 974-985 (2002)