XMLTM: Efficient Transaction Management for XML Documents

Torsten Grabs
Database Research Group
Inst. of Information Systems
ETH Zurich
8092 Zurich, Switzerland
grabs@inf.ethz.ch

Klemens Böhm*
Database Research Group
Inst. of Information Systems
ETH Zurich
8092 Zurich, Switzerland
boehm@inf.ethz.ch

Hans-Jörg Schek
Database Research Group
Inst. of Information Systems
ETH Zurich
8092 Zurich, Switzerland
schek@inf.ethz.ch

ABSTRACT

A common approach to storage and retrieval of XML documents is to store them in a database, together with materialized views on their content. The advantage over "native" XML storage managers seems to be that transactions and concurrency are for free, next to other benefits. But a closer look and preliminary experiments reveal that this results in poor performance of concurrent queries and updates. The reason is that database lock contention hinders parallelism unnecessarily. We therefore investigate concurrency control at the semantic, i.e., XML level and describe a respective transaction manager XMLTM. It features a new locking protocol DGLOCK. It generalizes the protocol for locking on directed acyclic graphs by adding simple predicate locking on the content of elements, e.g., on their text. Instead of using the original XML documents, we propose to take advantage of an abstraction of the XML document collection known as DataGuides. XMLTM allows to run XML processing at the underlying database at low ANSI isolation degrees and to release database locks early without sacrificing correctness in this setting. We have built a complete prototype system that is implemented on top of the XML Extender for IBM DB2. Our evaluation shows that our approach consistently yields performance improvements by an order of magnitude. We stress that our approach can also be implemented within a native XML storage manager, and we expect even better performance.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Concurrency, Relational databases, Textual databases, Transaction processing*

*Current affiliation: Department of Computer Science, Otto-von-Guericke-Universität Magdeburg, Germany

CIKM'02, November 4–9, 2002, McLean, Virginia, USA.

General Terms

Algorithms, Performance

Keywords

Transaction management for XML, XML storage managers

1. INTRODUCTION

XML has emerged as the universal format for data exchange. Storage and retrieval from large XML document collections is an important issue. XML is more and more used in contexts that are mission-critical, such as e-commerce. We foresee applications that process *XML queries and updates concurrently* and have strict requirements regarding *consistency and reliability*. One approach that seems to meet all these requirements is to use relational database technology. RDBMS vendors have extended their products accordingly, in two directions: (1) The database system allows to *publish* relational data as XML. (2) The database system *stores* XML data, i.e., it maps XML documents to database tables. A common distinction regarding (2) is between *side tables* and *document tables*. Side tables are the result of shredding XML elements to database tables and columns. Document tables in turn store complete XML documents as character-large-objects (CLOBs). Extensions of the database engine allow for querying and updating the XML documents stored as CLOBs. All XML solutions by RDBMS vendors combine shredding and materialized views on XML content with CLOB storage. This combination has several advantages: (1) the relational query engine can be used to efficiently process XML queries over the XML content stored in the side tables, and (2) access to the original document is efficient as well. To keep side tables up-to-date in the presence of updates, commercial implementations such as the one by IBM use triggers. They delete the mapped content of the updated document from the side tables and insert the updated content. Using a database system in this way has many advantages, e.g., persistence, buffering, or indexing come for free. However, relying solely on the DBMS transaction mechanism to give the usual guarantees for access to XML data is not a good idea. An initial experimental evaluation has revealed that this results in low performance. The reason is unnecessary database lock contention on document and side tables.

Example 1: Consider two concurrent transactions that run over the document tables. The first transaction retrieves

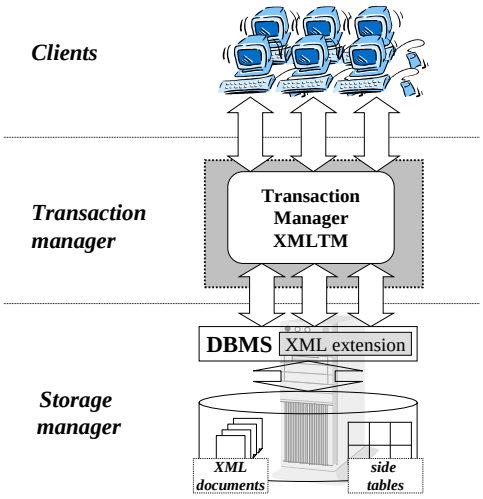


Figure 1: XML processing with XMLTM

all /description elements. The second one updates all /price elements. Obviously, there is no conflict. But the transaction manager of the database blocks one of them if a /price and a /description element appear in the same document. The same behavior occurs when the retrieval transaction is processed on the side tables: side table maintenance by the update transaction locks the side tables and hence blocks retrieval (or vice-versa). The effect is known as *pseudo-conflict* in the literature [29], and it results in low *inter-transaction parallelism*. ◇

For different, more specific application scenarios, semantic concurrency control has solved these problems. I.e., it has lead to a higher degree of parallelism and to significant performance improvements, in particular if the rate of pseudo-conflicts is high [1, 18]: The general idea is that a transaction manager takes the application semantics into account [26, 27]. This prevents from inconsistent flow of information between concurrent transactions at the application level. It also reduces the rate of pseudo-conflicts. This is because it allows releasing locks at the storage manager early, i.e., before the transaction at the application level commits.

However, it is unclear how such a transaction manager should look like when exploiting the "semantics" of XML. In particular, one must decide how to realize isolation and atomicity in this particular context. Our contribution is the design of a semantic transaction manager for XML, called XMLTM, and its evaluation as a second-layer transaction manager on top of an RDBMS.[1] We assume that the underlying storage manager[2] supports transactions. In the following, we refer to them as *database transactions*. This assumption is not a restriction, because extending XMLTM or the underlying system in this respect is easy [20].

XMLTM intercepts client *requests*, i.e., XML updates and queries to derive the semantic information to implement isolation and atomicity of concurrent *XML transactions* (cf. Figure 1). An XML transaction is a transaction at the application level that bundles one or several XML queries or updates. XMLTM uses granular locking in combination with predicate locking to guarantee correctness at the application level. Consequently, database transactions only have to guarantee consistency at the storage level. Hence, database transactions can commit early and release their locks earlier. We expect this to reduce lock contention and to increase parallelism of concurrent XML transactions, compared to a setting that uses only the transaction management of the database system.

As part of XMLTM, we propose a locking protocol called DGLOCK. It generalizes the well-known locking protocol for directed acyclic graphs (DAG locking) [10] to processing of XML data. The generalization is that it incorporates locking on content constraints, e.g., keywords and text from queries and documents, at arbitrary granularity. Another issue is that using the graph structure of the XML data for locking is not practical. This is because this structure is not directly available anyhow since XML extensions map XML data to relations. Instead, we propose to use DataGuides [7] as the underlying structure for locking. The idea with DataGuides is to create a summary of the structure of existing XML documents. We deploy this data structure to perform fine-grained locking without giving the locking algorithm access to the XML documents themselves. Since DataGuides may become big in size, the benefit in quantitative terms is not clear. Our evaluation addresses this point (with positive results). Another important new optimization is that XMLTM runs database transactions at a low ANSI isolation degree without giving up serializability, as we will show.

For evaluation purposes, we have implemented XMLTM on top of IBM DB2 with the XML Extender for DB2, and we compare it to a setting where transactional guarantees rely only on the transaction management of the database system. XMLTM increases performance of concurrent querying and updating of XML data by an order of magnitude. Furthermore, the overhead of DGLOCK is small in settings without pseudo-conflicts, e.g., query-only workloads.

The remainder of this paper is as follows: Section 2 reviews state-of-the-art XML extensions. Section 3 describes our transaction manager XMLTM and our locking protocol DGLOCK. Section 4 describes the experimental evaluation of our prototype with IBM DB2 and its XML Extender, together with a detailed discussion. Section 5 covers related work. Section 6 concludes.

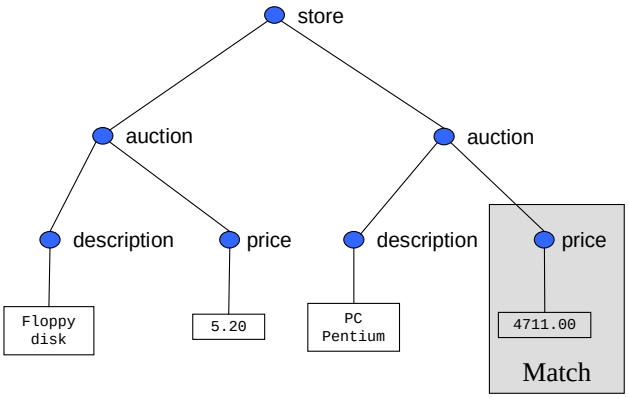
2. XML DATABASE EXTENSIONS

This study evaluates XMLTM as a second-layer transaction manager on top of a relational DBMS. Hence, this section reviews the implementation of XML extensions. We use IBM DB2 with the XML Extender as a running example. Nevertheless, XML extensions only require from the database system a data type to store large texts such as CLOB.

Terminology. In this paper, we assume that XML documents are trees. The *text of an XML document* or simply *document text* is the text together with the markup. The *graph representation* of a document is the one defined by the data model of the W3C XPath Recommendation [24]. *Matches* of a *path expression* are the sub-graphs of the graph

[1]We stress that the issue of designing such a transaction manager is not confined to transaction management at the second layer, but needs to be addressed whenever transactional guarantees over XML data are required. However, our evaluation and the presentation in this article from now on focus on XMLTM as a second-layer transaction manager.

[2]The underlying storage manager does not need to be a database, we just use the terms 'database' and 'storage manager' as synonyms throughout this article.



XPath: "/store/auction/price[. > 1000]"

Figure 2: Match of a path expression with an XML document in graph representation

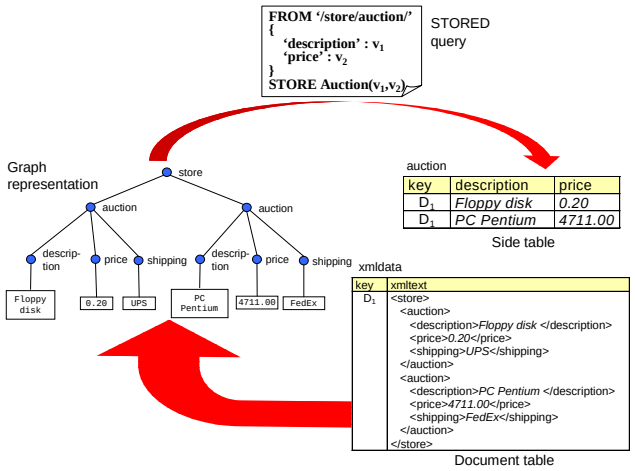


Figure 3: Illustration of STORED

representation that qualify for the path expression. XML *contents* are the parts of the document text that correspond to the match.

Example 2: Figure 2 shows the match of the path expression /store/auction/price[. > 1000] in the document depicted. A path expression may have more than one match per document. This is the case with path expression /store/auction/price[. > 1] and the document of the figure. ◇

Document Tables. XML extensions store XML documents in CLOB attributes. Additional methods, e.g., stored procedures, implement the XML-specific functionality. Some of these methods *extract* content from the XML documents. They take a path expression as an input parameter. It specifies the content to be extracted. Other methods *update* the XML documents. One input parameter again is a path expression. Another parameter specifies the new content to replace the one referred to by the path expression. An SQL statement can incorporate these XML-specific methods.

Example 3: The document table xmldata has a key and an xmltext attribute (see Figure 3). With IBM DB2 and its XML Extender, the following SQL statement retrieves the key attribute and the content of all price elements and converts price information to the data type double:

```
select key, x.returnedDouble
from xmldata,
table(db2xml.extractDoubles(
   xmltext,'//price')) as x
```

The following SQL statement updates price elements in documents with a key value of D_1:

```
update xmldata
set xmltext = db2xml.update(
   xmltext,'//price','200')
where key = D1
```
◇

We refer to SQL statements for updating and querying XML content as in Example 3 as *requests*. Clients compose the requests and submit them to the system, as Figure 1 shows. For what follows, it is important that XML extensions retrieve and update only the matches and their descendant nodes: both requests from Example 3 for instance access only data in the sub-trees rooted at the nodes that match the //price path expression.

Side Tables. Consider again the first SQL statement of Example 3. With the mechanisms described so far, the query engine inspects all documents even if only very few of them contain price information. To speed up queries, additional database tables – so-called *side-tables* – materialize views on the content of XML documents.

Literature has proposed various mapping schemes, i.e., side table definition schemes that differ in the number and layout of the side tables. The mappings that are commercially available are simple variants of so-called *STORED queries* [3]. Several STORED queries specify an *XML-to-RDBMS mapping*. Such a query consists of a **FROM** and a **STORE** clause, as Figure 3 shows. The **FROM** clause matches a pattern with a given XML document. For each match, the **STORE** clause creates a tuple in the side table based on the current variable bindings. This approach is not restricted to one document table and one side table only.

Given a query, two situations may occur: (1) the side tables alone contain all the data necessary to evaluate the query. Then an SQL statement over the side tables is sufficient. (2) The side tables alone do not suffice. They only allow to identify a superset of the documents in the query result, the so-called *candidates*. A query that identifies the candidates is a *subsuming query*.

Example 4: Consider the database schema from Figure 3 together with the following query for IBM DB2.

```
select x.returnedVarchar,
   y.returnedDouble
from xmldata as z,
table(db2xml.extractVarchar(
   xmltext,'//shipping' )) as x,
table(db2xml.extractDouble(
   xmltext,'//price')) as y
where z.key in
(select key from auction
where price < 50)
and y.returnedDouble < 50
```

This request selects information about low-priced auctions. But the 'shipping' information is not available in side tables. The query has to fetch it from the document table. The

query select key from auction where price $<$ 50 is a subsuming query. ◇

When it comes to updates, side tables and document tables have to be consistent. IBM's implementation uses database triggers on the document tables.

Performance Issues and Shortcomings. We have carried out a detailed analysis of XML extensions using the data set provided with the XML benchmark [19]. Response times with concurrent updating and querying are very low when relying solely on the transaction management functionality of the database system (see Section 4 for numbers). The explanation is that lock contention hinders inter-transaction parallelism. The following section says how we have addressed these issues.

3. TRANSACTION MANAGEMENT FOR CONCURRENT XML PROCESSING

Transactions are a key concept to guarantee reliability of information systems and data consistency in the presence of system failures and interleaved access to shared data. When using XML in contexts that are mission-critical, transactional guarantees are indispensable as well. They enable the application programmer to group a set of requests that require isolation and atomicity to a transaction. With XML stored in an off-the-shelf RDBMS, there are two ways to provide these guarantees. The first one – denoted as the *flat transaction model* – relies only on the transaction processing functionality of the database. No further implementation effort is necessary. The alternative is *transaction management at the application level* [18, 28]. With this model, a set of requests that require isolation and atomicity forms a *global transaction* or a *transaction at the application level*. An additional transaction manager on top of the storage manager decomposes such a transaction into independent subtransactions, so-called *database transactions*, and schedules them. To do so, it considers the application semantics, i.e., the conflicts at the application level. A database transaction can *commit early*, i.e., before the end of its global transaction, and therefore can release its locks early in order to increase parallelism without sacrificing correctness.

To compare the two alternatives in the context of XML, we have designed and implemented XMLTM, a transaction manager for efficient concurrent processing of XML queries and updates. The global transactions with XMLTM are *XML transactions* that comprise XML queries and updates. XMLTM intercepts the client requests to derive their semantic information, to keep track of global transactions, and to control the database transactions. XMLTM maps a request to a set of so-called *operations*, one for each candidate document. Each operation runs as a storage manager transaction. We will show that they can run at a lower ANSI isolation degree [10]. This optimization, together with early release of locks at the storage manager, should lead to much less lock contention when deploying XMLTM on top of existing XML extensions.

XMLTM does not rely on the semantics of a specific interface. The only 'restrictions' with our work in turn are that (1) XML is the underlying data format, and (2) there is a distinction between read and write operations.

3.1 Isolation at the XML Level

Isolation means that there is no inconsistent flow of information between concurrent global transactions. A flow of information is inconsistent if the schedule of the global transactions is not serializable. With conflict serializability as correctness criterion, locking is a common technique to ensure correctness [2]. XMLTM is based on locking as well.

3.1.1 The DGLOCK Protocol

XMLTM implements the locking protocol DGLOCK, a new protocol proposed in this article. In this context, we make the following distinction between constraints of requests: *structural constraints*, i.e., constraints on the structure of documents, versus *content constraints*, i.e., constraints on the content of elements. To give an example, consider path expression /store/auction/price[. $>$ 1000] (cf. Example 2): /store/auction/price is a structural constraint while [price $>$ 1000] is a content constraint. Structural constraints are constraints on the type level, content constraints are on the instance level. The main innovation of DGLOCK is that it takes both kinds of constraints into account: to cope with structure constraints, DGLOCK takes over the idea of granular locking on directed acyclic graphs (DAGs for short) [10] – but applies it to the DataGuide, rather than to the documents themselves. Predicates on the nodes of the DataGuide in turn allow to deal with content constraints. When XMLTM intercepts a request, it determines its structural constraints, its content and therefore also its content constraints.

One might have expected the transaction manager to lock the nodes in the graph representation of the documents. Previous work on object bases has already investigated the problem of locking on a graph, see [17] among others. These approaches are only viable if the graph is physically available. But this is typically not the case with any practical representation of XML data. Consequently, DGLOCK only relies on the much weaker assumption that a complete summary of the structure of the documents is available. *Complete* means that each label path in the document collection is also part of the summary. The *DataGuide* is a structure that has this characteristic [6, 7] by definition: for each label path in the data, the same path also occurs in the DataGuide exactly once. Furthermore, a DataGuide is *concise*, i.e., it does not contain any other label paths. In what follows, we also assume that a DataGuide has exactly one root. If this is not the case, one can always add a virtual root.

DGLOCK implements serializability by a two-phase locking protocol on the nodes of the Data Guide: Each new request dynamically acquires the needed locks immediately after its invocation, and the concurrency control releases them at the end of the transaction (strict two-phase locking). In addition to the usual differentiation between shared locks (S locks), exclusive locks (X locks), and intention locks (IS and IX), which reflect the intention of the request to place S respectively X locks at a finer granularity, DGLOCK provides for annotations of locks with simple predicates. Here, simple predicates are conjunctions of comparisons of the form $x\ \theta\ const$ with $\theta \in \{=, \in, \neq, \leq, \dots\}$. The difference between the compatibility matrix used by DGLOCK to decide whether a lock is granted, displayed in Table 1[3], and the one of DAG locking is as follows: The DGLOCK-matrix does not contain strict incompatibilities; an incompatibility occurs only if the predicates of locks already granted and

[3] *SIX* locks are a combination of a shared lock and an intention lock for exclusive access to finer granularities.

	Granted					
Requested	None	*IS*	*IX*	*S*	*SIX*	*X*
IS	+	+	+	+	+	*P*
IX	+	+	+	*P*	*P*	*P*
S	+	+	*P*	+	*P*	*P*
SIX	+	+	*P*	*P*	*P*	*P*
X	+	*P*	*P*	*P*	*P*	*P*

Table 1: DGLOCK lock compatibility – compatibilities marked as '+', predicate tests as 'P'

the one of the lock requested are not compliant, or if a lock does not have a predicate annotation. More formally, let $pred_1^{granted}, \ldots, pred_n^{granted}$ denote the read resp. write sets of those predicates that annotate the locks already granted, and let $pred^{req}$ denote the one of the lock requested. The lock is granted only if $P \equiv (\bigcup_i pred_i^{granted}) \cap pred^{req} = \emptyset$.

Having said this, DGLOCK performs the following steps for a new request s:

1. Extract the constraints: obtain all path expressions $\mathcal{E}$ in s that lead to data that is queried or updated by s, i.e., extract the structural constraints. Annotate each node of each element of $\mathcal{E}$ with the predicate that reflects the respective content constraint.

2. Compute the set $\mathcal{N}$ of all nodes of the Data Guide that match any $e \in \mathcal{E}$, differentiating between nodes updated and those that are only read.

3. For each $n \in \mathcal{N}$ perform the following operations using the lock compatibility matrix:

   (a) If node n is updated by s, acquire IX locks on all nodes along all paths that lead from the root to n taking the annotations of the nodes of the DataGuide and the ones of e into account.[4] Request the locks in the order of increasing distance from the root, i.e., from the root to n. Then acquire an X lock on n, again taking the annotations into account.

   (b) If node n is only read by s, acquire IS locks on all nodes along at least one path that leads from the root to n. Then acquire an S lock on n. As with IX and X locks, acquire the locks from the root to n taking the annotations into account.

If two locks are not compatible, the concurrency control delays the new lock request. The transaction is blocked until the transaction with the incompatible lock has released its lock. Deadlock detection aborts a transaction if it submits a request for a lock that leads to a cyclic lock waiting condition. Our implementation of the lock manager follows the one described in [10].

Example 5: Figure 4 shows a DataGuide for the document from Figure 3. Transaction T_1 retrieves XML content from the locations given by path expression /store/auction[price > 5]/description. Transaction T_2 in turn uses path expression /store/auction[price < 0.5]/price to set the respective price values to '4'. XMLTM runs T_1 and T_2 concurrently since all locks granted with DGLOCK are compatible, as Figure 4 (left) shows. ◇

[4]Note that even though XML documents in this paper are trees, the DataGuide is not necessarily a tree as well.

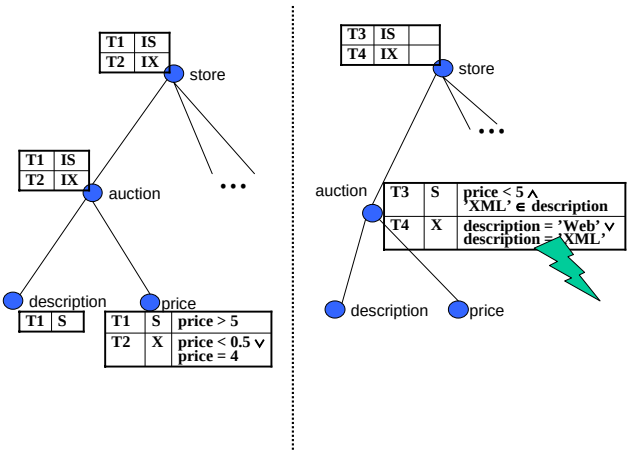


Figure 4: Locking on the DataGuide

Note that locking at coarser granularity both for structural and content constraints is feasible as well. This is particularly appropriate if the hierarchy of the DataGuide has too many levels such that lock management would become too tedious. Regarding structural constraints, this means that one uses the S or X locks instead of intention locks already at ancestor nodes. Then the locks at their child nodes are obsolete. When it comes to content constraints, the analogous idea is to annotate nodes that are higher up in the hierarchy. The following example illustrates this.

Example 6: Consider again the DataGuide for the document from Figure 3 with transactions T_3 and T_4. T_3 retrieves XML content using path expression /store/auction[price < 5 AND contains(./description, 'XML')]. T_4 in turn changes all description contents from 'Web' to 'XML' using path expression /store/auction[./description = 'Web')]. As Figure 4 (right) shows, T_4 is not granted the X lock on auction since its predicate description='XML' derived from the new content is incompatible with the shared lock held by T_3. ◇

Choice of DataGuide. In what follows, we compare different kinds of DataGuides, notably *minimal* ones and *strong* ones [7], with regard to their suitability for our purposes. Minimal DataGuides have the characteristic that the number of nodes is minimal. Given that there is exactly one root, strong DataGuides have a tree structure, and there is a one-to-one correspondence between label paths and nodes of the DataGuide. DGLOCK is correct independent of these specializations of the notion of DataGuide. Strong DataGuides however are preferred for two reasons. The first one is that Step 3(a) of DGLOCK has to lock only one path per node because of the tree structure. The other reason is a lower degree of lock contention. With a DataGuide that is not strong, a lock on one of its nodes would lock several label paths in the general case.

Expectations. We expect our locking protocol to allow for more parallelism of concurrent global transactions than the flat model. On the one hand, the blocking situation described in Example 1 does not occur. Furthermore, we expect that locking based on both structure and content gives way to a low degree of lock contention and a high

degree of concurrency. On the other hand, the lock management on the DataGuide leads to an additional overhead, namely locking overhead and additional effort to maintain the DataGuide (if not already available for other purposes), and it is not clear whether DGLOCK actually improves performance. The experimental evaluation addresses these questions.

3.1.2 Reducing the Degree of Isolation

The objective of this subsection is to reduce the isolation degree at the storage level in order to have better performance: DGLOCK is based on the 'repeatable read' characteristic of ANSI isolation degree 3. The following optimization is based on the observation that *database transactions* with XMLTM do not read data objects repeatedly. This is in contrast to XML transactions at the application level where DGLOCK implements the repeatable read property for XML transactions at the higher level. This allows to run database transactions at ANSI isolation degree 2, i.e., 'read committed'. It differs from isolation degree 3 only in that it does not give us repeatable reads while incurring less locking overhead at the storage level. However, note that we cannot go below isolation degree 2 for the following reason: side-table maintenance within a storage manager transaction updates more than one tuple in the general case. For instance, think of a database transaction that deletes a side-table tuple and inserts a new one. Other database transactions may not see the intermediate state. This requires isolation level 'read committed'.

3.2 Atomicity at the XML Level

Atomicity means that all updates of a transaction are either executed to completion or not at all. Atomicity is typically implemented by means of recovery. When a transaction manager at the application level decomposes a global transaction into several database transactions that commit independently of each other, it must comprise recovery functionality as well. To do so, XMLTM implements undo-recovery for XML transactions at the application level [10]. Undo-recovery is necessary to compensate the effects of early commits when a global transaction aborts. Atomicity of single database transactions instead is implemented by the storage manager.

Undo recovery requires to log begin-of-transaction and end-of-transaction markers. Each such marker also carries the identifier that was assigned to the global transaction at its beginning. This allows to determine the global transactions that have not completed, i.e., whose end-of-transaction marker is missing, in case of a crash. Undo-recovery aborts these transactions and compensates their effects. *Compensation* means that the effects of already committed storage level transactions are undone if their global transaction aborts. XMLTM now implements this aspect of recovery as follows: To allow for compensation, XMLTM writes the updated subtrees of the document to the log before the changes of a request come into effect. This yields a *before-image*, to be restored when undoing a transaction. In addition, XMLTM also logs the parameters of the update requests, namely the path expression and the new content. This allows to compute the compensation operation in case of recovery.

Example 7: Consider an XML transaction with a request to update the price of item '4711' to 10.00. XMLTM logs its path expression /store/auction[itemid = '4711']/price and the before-image of the price. The compensation operation of the request is thus an update request with the same path expression but with the old price as given by the before-image. ◇

It remains to be said what happens to the side tables: triggers keep derived information such as side tables up-to-date. The update of a document and the triggers run in the same storage level transaction. This guarantees that XML documents and their mapped XML content are mutually consistent. Hence, XMLTM does not need to log changes of side table content.

To sum up this subsection, recall that the basic alternative to XMLTM is the flat transaction model. It has the advantage that the additional effort for logging of potentially large before images is not necessary. The downside is that an early commit of the database transaction is not feasible and lock contention is higher. Our experimental evaluation investigates this tradeoff for workloads of concurrent XML updates and queries.

4. EXPERIMENTAL EVALUATION

We have carried out numerous experiments to assess our solution. We want to find out if it really improves performance of concurrent queries and updates of XML data. We compare response times and throughput of client requests with XMLTM to those with the flat transaction model. Another important question is the effect of the size of the document text.

The following subsections only discuss our most prominent results. For further results see [9].

4.1 Experimental Setup

XML Documents. The XML documents in our experiments have been created with the document generator xmlgen of the XML benchmark project [19]. The scaling factor was 1.0 (standard), i.e., 100 MB of XML data. For our experiments, we have generated three different collections distributing the XML data over 100, 1,000, and 10,000 documents. The average document size with these collections is 1,000 KB, 100 KB, and 10 KB, respectively. The database size with the document tables and the side tables is the same with each collection, namely about 300 MB, including side tables and indexes. We store the XML document texts in a document table xmldata as discussed in Section 2. Figure 5 shows an excerpt of the DataGuide for our experimental data. Nodes whose complete content has been mapped to side tables are marked grey. Content from further nodes not shown in the figure has also been mapped to the side tables. In total, our experimental database setting comprises 18 side tables. The DataGuide in Figure 5 is also the one that we have used for locking with XMLTM.

Workload. The experiments work with two streams of transactions. One stream queries the XML documents, the other one invokes updates. Our study focuses on a conservative setting where transactions always contain one request. Longer transactions would result in a higher degree of lock contention and let XMLTM appear in a better light. Each stream immediately submits a new request when it has received the result of the previous one, i.e., there is no think time. This models the worst case for response time and throughput, as compared to a setting with think times. Our experiments revealed that our workload already exhausts

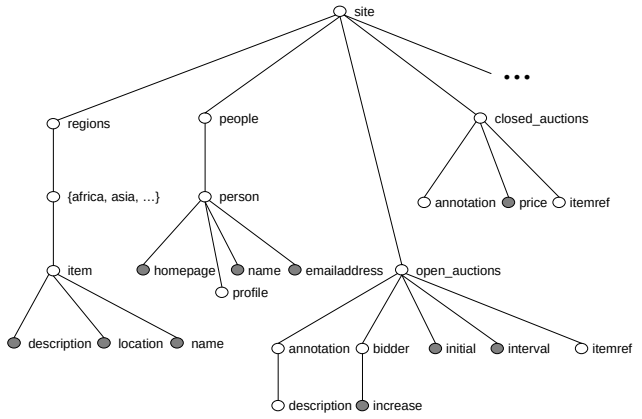


Figure 5: Excerpt of the DataGuide of the experiments

the resources of the standard PC that we used in our study. We have synthetically generated the requests similarly to the queries of the XML Benchmark [19]. Since [19] does not cover updates, we generated a set of update requests. They mimick requests of an online auction such as converting an open auction to a closed auction, adding or increasing a bid for some offer, or changing shipment types. All requests comprise a subsuming query over one or several side tables, and each request accesses one or several documents.

Hardware and Software. We have run our experiments on an off-the-shelf PC with one Intel Celeron Processor and 512 MB of RAM. The PC runs the Microsoft Windows 2000 Advanced Server operating system software. XMLTM has been implemented a set of Microsoft COM+ components. The DBMS is IBM DB2 V7.1 with the XML Extender for DB2. The database buffer size of DB2 adjusts dynamically to the current workload, which is the default option under Windows 2000. With XMLTM, our benchmarking environment routes all requests, i.e., queries and updates from the client streams, through the COM+ components that implement XMLTM (see Figure 1). With the flat transaction model in turn, the requests go directly to the storage manager. In both settings, response times and throughput are measured at the clients. When testing XMLTM, response times also include the overhead required to maintain the DataGuide.

4.2 Outcome and Discussion of the Experiments

XML Processing with Side Tables: Effect of Transaction Management and Document Size. Our first series of experiments investigates performance of XML processing in the presence of side tables. Figure 6 graphs average response times and throughput for the collections with 100, 1,000, and 10,000 documents. Note that the figure uses a log-scale axis for response times and throughput. This will also be the case with most figures that follow. A first observation is that the performance of XML processing increases with smaller document sizes: updates with flat transactions for instance yield average response times of nearly 700 seconds with 100 large documents. With 10,000 small documents, it is only 100 seconds per update request. The reason is the following: for each candidate, the IBM DB2 XML Extender loads the complete document text into an internal representation to process an update or a query. This incurs less overhead for smaller documents and explains the performance gain for smaller document sizes. Similar observations hold for all response times and throughput curves both with flat transactions and XMLTM transactions. So far, this is what one would expect. But a closer look at our results reveals that the benefit from smaller document sizes depends on the choice of the transaction manager. Query performance with XMLTM transactions increases by more than an order of magnitude from document size 1,000 KB to 10 KB. Flat transactions instead yield an improvement by a factor of 4 only. The effect on update performance in turn is somewhat different: flat transactions yield an improvement by a factor of 7 from document size 1,000 KB to 10 KB. With XMLTM transactions, it is only 4. Summing up, XMLTM transactions yield higher response times of update requests for any document size as compared to flat transactions. Update throughput with flat transactions typically is twice the one of XMLTM transactions. The reason is that updating with XMLTM transactions incurs overhead for the additional logging and commit processing. With XMLTM transactions, our transaction manager writes a before image to the log before updating the document. Our current version of the implementation simply takes the complete document text as the before-image. Clearly, more fine-grained before-images, as described in Example 7, lead to less overhead for logging. We are currently extending our implementation in this respect.

Regarding commit processing, recall that XMLTM commits the database transaction after each document update. This is not the case with flat transactions. But the downside of these long running flat transactions is lock contention on the document table and the side tables. This unnecessarily blocks queries, as Figure 6 shows: query throughput with XMLTM transactions and an average document size of 10 KB is more than an order of magnitude higher than with flat transactions. Moreover, we allow database transactions with XMLTM transactions to run at ANSI isolation degree 2 ('read committed') since DGLOCK already guarantees the repeatable read property at the application level. Flat transactions in turn must run at ANSI isolation degree 3 ('serializable'). A single update request deletes and inserts to many side tables, and the database lock manager places an X lock on the side tables. This seriously hinders concurrent query requests. We have performed a more detailed analysis of this effect using the IBM DB2 Lock Monitor. It has shown that queries and updates with flat transactions typically form a convoy [10]: queries wait for the current update request to finish. Then the queries are processed. The following query waits until the following update request has finished, and so on. The Lock Monitor also points out another reason why XMLTM outperforms the flat transaction model: the volume of data required for locking with flat transactions is huge. For instance, a transaction that updates a typical document with 1.8 MB of XML text data requires more than 1,000 database locks, and the size of the lock list is more than 80 KB. Now consider a transaction that updates several, say n, documents. Then these numbers increase by a factor of n. With less than 20 KB locking data for the update request, the overhead of DGLOCK is significantly smaller.

XML Processing with Side Tables: Effect of Conflict Ratio. A follow-up question on these results is how

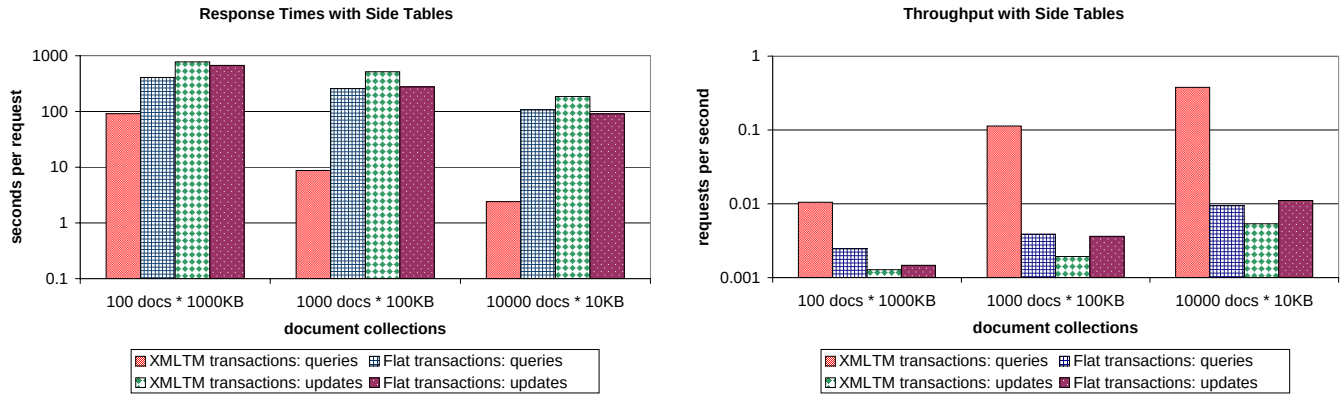


Figure 6: Update and query performance with side tables: response times (left) – throughput (right)

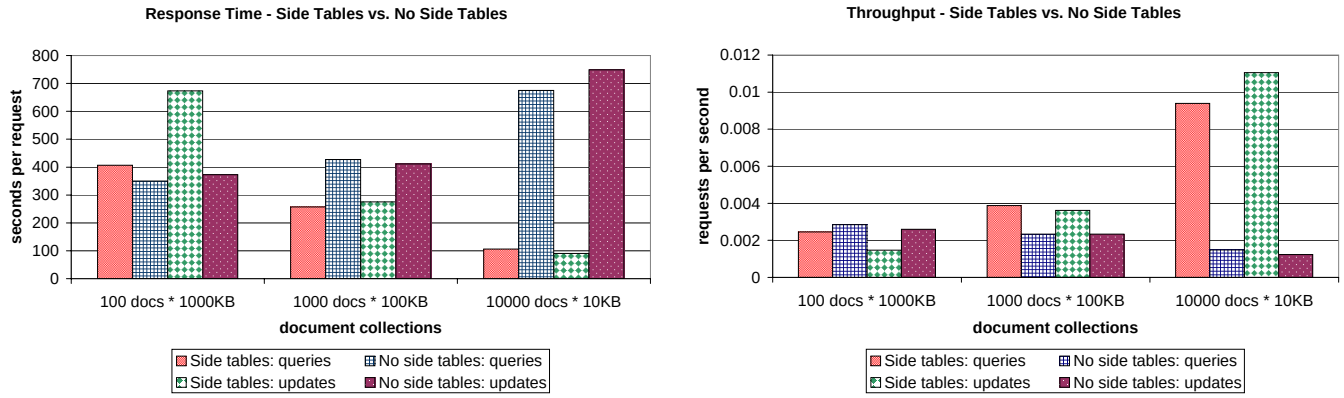


Figure 7: Update and query performance with/without side tables: response times (left) – throughput (right)

the conflict ratio between requests affects blocking with flat transactions. We have run experiments with two different access patterns, investigating again response times and throughput. The first access pattern has a document table conflict ratio of 80%. In our terminology, that means that the write sets on the document table for 4 out of 5 update requests overlap with at least one read set of the concurrent queries. The second type of access pattern has a document table conflict ratio of 20%. Our hypothesis was that the second access pattern leads to better query performance. We were surprised to find out that it does not hold true. This has to do with both side table maintenance and the locking strategy of the database: in case of an update, the triggers maintain the side tables. The current implementation of the XML Extender deletes *all* information of the document from *all* side tables and inserts the updated version of the document text. The database lock manager therefore places X locks at the table level on the side tables independently of the document table conflict ratio.

XML Processing without Side Tables. Another series of experiments has compared performance in a setting with side tables to one without. Figure 7 reports on the outcome with different document sizes using flat transactions. Note that this figure is not log-scale. Our first observation relates to the effect of document size. The results are as expected with small documents, i.e., with 10 KB per document: query and update performance without side tables is more than 6 times lower than with side tables. The results for larger documents with 100 KB per document reflect this finding. Having side tables improves performance by a factor of 2. However, side tables do not increase performance with a document size of 1,000 KB.

We conducted an additional series of experiments to shed more light on this surprising finding. Using the same workloads, these experiments measure individual response times and throughput statistics of operations. In other words, we have measured the performance of updating a document or performing a query operation on it (response time) as well as the number of such operations performed per second (throughput). Performance of update operations is significantly lower with side tables than without (cf. Figure 8). The reason is that side tables require maintenance in case of an update. Side table maintenance is included in the numbers in the figure. Hence, the figure tells us that the overhead is close to an order of magnitude with large documents and even more with smaller ones.

However, the benefit from side tables, i.e., fewer candidates (cf. Section 2), does not compensate this effect for large document sizes. Without side tables, the query processor has to fetch all documents stored. With side tables in turn, the situation is more differentiated, as a further investigation has shown: with small document sizes the candidate set is two orders of magnitude smaller than the total number of documents. With large documents instead, the size of the candidate set is about 20% of all documents. Recall our previous result that the overhead of a document update

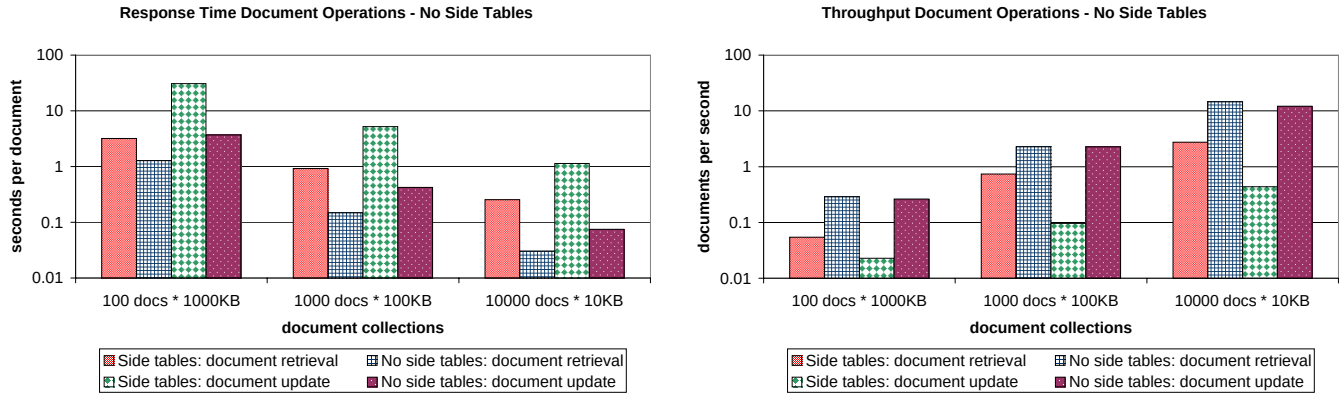


Figure 8: Performance of document operations with/without side tables: response times (left) – throughput (right)

with large document sizes is nearly an order of magnitude. In combination with the experiments on the size of the candidate set, this explains why updates for large documents perform better without side tables.

5. RELATED WORK

Processing of concurrent querying and update of XML data has received only little attention so far. This section first covers previous work on this problem. We then comment on alternatives to store XML documents. Using commercial database systems as storage managers for XML requires to map XML content to the database. We also discuss previous work on this issue.

Transactional Guarantees for Processing of XML. Relatively little previous work has dealt with updates in the XML context. So far, only [23] has explicitly considered respective declarative mechanisms, extending XQuery [25]. They also provide experimental results for an implementation on top of a relational database system. [23] only deals with updates in isolation. There is no concurrency of queries or updates. Closest to our work is [11] that investigates isolation of simple DOM operations on single XML documents. The authors define commutativity of these operations when accessing the same node of the document and derive alternatives for pessimistic and optimistic concurrency control. XMLTM in turn is more general and is applicable to a broader range of application scenarios, for several reasons: DGLOCK does not assume a fixed API for conflict definition in order to guarantee serializability and does not rely on a particular storage layout. In contrast to [11], our current work also takes recovery into account. Finally, [11] does not have an experimental evaluation. Our own previous work in the XML context [8] does not deal with XML-specific queries and updates and is less general as well.

XML Repositories. A meaningful classification[5] of XML repositories is into (1) extensions of commercial RDBMSs, (2) native XML stores [13, 22, 30], and (3) hybrid approaches, e.g., [4]. A common approach when extending database systems is to use SQL to access XML data, e.g., [12, 15, 16]. CLOB attributes of conventional database tables store the XML documents. Specialized operators extend the respective SQL dialect while their implementation provides the XML-specific functionality. XPath expressions typically specify the patterns to access XML content [24]. However, all of these proposals suffer from a low degree of concurrency of updates and queries. Our locking protocol DGLOCK solves this problem. XMLTM, which incorporates DGLOCK, has the nice characteristic that it is applicable with all of the aforementioned approaches (and systems).

Several approaches how to map XML content to databases have been proposed, e.g., [3, 5, 21]. Implementations that are commercially available deploy simple variants of the STORED mapping [3]. A notable difference is that the commercial implementations also store the original document texts, as opposed to overflow graphs in [3]. Of course, our experiments could not take all of those mappings into account, but on the logical level XMLTM is independent from the particular mapping scheme. Briefly, a general quantitative result has been that XMLTM is most advantageous when (1) the locking scheme of the storage manager does not reflect the XML semantics, or (2) the degree of redundancy, i.e., volume of XML content stored in the side tables, is high. XML extensions use triggers to keep mapped XML content and original document mutually consistent. This means that they update the materialized view in the same transaction. Hence, we do not need to deal with serializability of accesses to the views and to the original data, as discussed in [14].

6. CONCLUSIONS

Efficient concurrent processing of updates and queries of XML data in a consistent and reliable way is an important practical problem. XML extensions of commercial database systems perform poorly in this respect due to lock contention. Our contribution is the design and implementation of XMLTM, a transaction manager for concurrent processing of XML data, and its evaluation as a second-layer transaction manager on top of XML extensions. Building on previous work on locking in DAGs [10], we propose a granular locking technique DGLOCK that implements isolation for concurrent XML processing. DGLOCK captures both the structural conditions and the content predicates of requests and places locks on the DataGuide. When implemented on top of a database system, DGLOCK allows to perform an early commit of the database transactions that implement XML requests and to run them at a lower ANSI

[5]See http://www.xmldb.org/faqs.html

isolation degree. This avoids lock contention. Our experiments have shown that query performance with XMLTM is better by more than an order of magnitude than with the flat transaction model without sacrificing correctness. We stress that the range of applications of DGLOCK is broad and not limited to the setup evaluated in this study. For instance, DGLOCK can be part of a native XML storage manager implementation. It can also be part of an integrated database solution. This holds even though our work shows that it is not necessary to build an XML extension from scratch to implement XMLTM. Adding relatively little code on top of the database system and an off-the-shelf XML extension is sufficient.

The reader should note that our study does not address the issue of physical design. We simply rely on XML-to-database mapping schemes that are part of current XML extensions. The significant performance gains observed in the experiments can be fully attributed to increased parallelism with XMLTM. A further important result of our experimental evaluation is that the size of the XML document text does not affect the rate of pseudo-conflicts with flat transactions. As a consequence, the performance improvements with XMLTM even increase with small documents.

Given the results of this study, locking in the context of XML should take the semantics of XML into account to increase concurrency of XML processing, similarly to DGLOCK.

7. REFERENCES

[1] D. Barbará, S. Mehrotra, and P. Vallabhaneni. The Gold Text Indexing Engine. In *Proceedings of the Twelfth International Conference on Data Engineering, New Orleans, USA*, pages 172–179. IEEE Computer Society, 1996.

[2] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[3] A. Deutsch, M. F. Fernandez, and D. Suciu. Storing Semistructured Data with STORED. In *Proceedings ACM SIGMOD International Conference on Management of Data, June, 1999, Philadephia, Pennsylvania, USA*, pages 431–442. ACM Press, 1999.

[4] Excelon Corp. Extensible Information Server. Technical report, Excelon Corp., 2001. http://www.exceloncorp.com/platform/extinfserver.shtml.

[5] D. Florescu and D. Kossmann. Storing and Querying XML Data using an RDMBS. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.

[6] R. Goldman, J. McHugh, and J. Widom. From Semistructured Data to XML: Migrating the Lore Data Model and Query Language. In *ACM SIGMOD Workshop on The Web and Databases (WebDB'99), June 3-4, 1999, Philadelphia, Pennsylvania, USA*, pages 25–30. INRIA, Informal Proceedings, 1999.

[7] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proceedings of 23rd International Conference on Very Large Data Bases, 1997, Athens, Greece*, pages 436–445. Morgan Kaufmann, 1997.

[8] T. Grabs, K. Böhm, and H.-J. Schek. Scalable Distributed Query and Update Service Implementations for XML Document Elements. In *11th International Workshop on Research Issues on Data Engineering: Document management for data intensive business and scientific applications (RIDE-DM'2001), 2001, Heidelberg, Germany*. IEEE Computer Society, 2001.

[9] T. Grabs, K. Böhm, and H.-J. Schek. XMLTM: High-Performance XML Extensions for Commercial Database Systems. Technical report, Database Research Group, ETH Zurich, 2002. Available at: http://www.dbs.ethz.ch/∼grabs/papers/-XMLDBExtensionsTechRep.pdf.

[10] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[11] S. Helmer, C.-C. Kanne, and G. Moerkotte. Isolation in XML Bases. Technical report, University of Mannheim, Germany, 2001. Available at: http://pi3.informatik.uni-mannheim.de/staff-/mitarbeitermoer/Publications/MA-01-15.ps.

[12] International Business Machines, Corp. *XML Extender: Administration and Programming*. International Business Machines, Corp., 2000.

[13] C.-C. Kanne and G. Moerkotte. Efficient Storage of XML Data. In *Proceedings of the 16th International Conference on Data Engineering, San Diego, California, USA*, page 198, 2000.

[14] A. Kawaguchi, D. F. Lieuwen, I. S. Mumick, D. Quass, and K. A. Ross. Concurrency Control Theory for Deferred Materialized Views. In *Database Theory - ICDT '97, 6th International Conference, Delphi, Greece, January 8-10, 1997*, volume 1186 of *Lecture Notes in Computer Science*, pages 306–320. Springer, 1997.

[15] Microsoft Corp. XML and Internet Support. Technical report, Microsoft Corp., 2000. SQL Server 2000 Documentation.

[16] Oracle Corporation. Oracle9i Application Developer's Guide - XML. Technical report, Oracle Corporation, 2001. http://download-west.oracle.com/otndoc/-oracle9i/901_doc/appdev.901/a88894/toc.htm.

[17] M. T. Özsu. Transaction Models and Transaction Management in Object-Oriented Database Management Systems. In *Proceedings of the NATO Advanced Study Institute on Object-Oriented Database Systems, Izmir, Turkey, August*, volume 130 of *NATO ASI Series F: Computing and Systems Sciences*, pages 147–184. Springer Verlag, 1993.

[18] M. Rys, M. C. Norrie, and H. J. Schek. Intra-Transaction Parallelism in the Mapping of an Object Model to a Relational Multi-Processor System. In *Proceedings of 22th International Conference on Very Large Data Bases, Mumbai (Bombay), India*, pages 460–471. Morgan Kaufmann, 1996.

[19] A. Schmidt, F. Waas, M. Kersten, D. Florescu, I. Manolescu, M. Carey, and R. Busse. The XML Benchmark Project. Technical Report INS-R0103, CWI – Centruum voor Wiskunde en Informatica, April 2001.

[20] H. Schuldt, H.-J. Schek, and G. Alonso. Transactional Coordination Agents for Composite Systems. . In *International Database Engineering and Applications*

Symposium, IDEAS, August 1999, Montreal, Canada, pages 321–331. IEEE Computer Society, 1999.

[21] J. Shanmugasundaram, E. J. Shekita, R. Barr, M. J. Carey, B. G. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently Publishing Relational Data as XML Documents. In *Proceedings of 26th International Conference on Very Large Data Bases, September, Cairo, Egypt*, pages 65–76. Morgan Kaufmann, 2000.

[22] Software AG. Tamino - The XML Power Database. Technical report, Software AG, 2001. http://www.softwareag.com/tamino/.

[23] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data, May, Santa Barbara, CA USA*, pages 413–424, 2001.

[24] The World Wide Web Consortium. XML Path Language (XPath) Version 1.0. http://www.w3.org/TR/xpath, Nov. 1999.

[25] The World Wide Web Consortium. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery, Feb. 2001.

[26] R. Vingralek, H. Hasse-Ye, Y. Breitbart, and H.-J. Schek. Unifying Concurrency Control and Recovery of Transactions with Semantically Rich Operations. *Theoretical Computer Science*, pages 363–396, 1998.

[27] W. Weihl. Commutativity-Based Concurrency Control for Abstract Data Types. *IEEE Transactions on Computers*, 37(12):1488–1505, Dec. 1988.

[28] G. Weikum. Principles and Realization Strategies of Multilevel Transaction Management. *ACM Transactions on Database Systems (TODS)*, 16(1):132–180, 1991.

[29] G. Weikum and H.-J. Schek. *Database Transaction Models for Advanced Applications*, chapter Concepts and Applications of Multilevel Transactions and Open Nested Transactions (Ahmed K. Elmagarmid), pages 515–553. Morgan Kaufmann, 1992.

[30] L. Xyleme. A Dynamic Warehouse for XML Data of the Web. *IEEE Data Engineering Bulletin*, 24(2):40–47, 2001.