XMLTM: High-Performance XML Extensions for Commercial Database Systems

Torsten Grabs Klemens Böhm Hans-Jörg Schek
Database Research Group, Institute of Information Systems
ETH Zentrum, 8092 Zurich, Switzerland
{grabs,boehm,schek}@inf.ethz.ch

Due to the success of XML for data interchange, relational database products now include support for processing of XML data. A common approach, subsequently referred to as *XML extensions*, stores XML documents in character-large-object (CLOB) attributes and extends the database engine with XML-specific functionality to read and update these attributes. In addition, XML extensions materialize views on XML content in so-called side tables. Triggers guarantee consistency of documents and view materializations. However, a series of preliminary experiments with XML extensions has revealed that performance of concurrent queries and updates is low. This motivates to design XML extensions more carefully, aiming at better performance. This study identifies two important shortcomings of XML extensions: (1) Database lock contention hinders parallelism unnecessarily. (2) Querying and updating of an XML document requires to load the entire document into an internal representation. We propose a solution that addresses these problems as follows: Its core is a transaction manager XMLTM. It features a new locking protocol DGLOCK. DGLOCK generalizes the protocol for locking on directed acyclic graphs for XML data. XMLTM allows to run XML processing at low ANSI isolation degrees and to release database locks early without sacrificing correctness. Regarding the second problem, we make use of the idea of decomposing XML documents into fragments. The rationale is that the internal representation is generated only for the relevant fragments. We have implemented our solution on top of the XML Extender for IBM DB2. Our experimental evaluation shows that our approach consistently yields performance improvements by an order of magnitude.

1. Introduction

XML has emerged as the universal format for data exchange. XML is attractive because it covers the full range from mainly unstructured data to highly structured data, notably relations. In addition, XML is more and more used in contexts that are mission-critical, e.g., e-commerce. Consequently, storage and retrieval from large XML document collections has become an urgent practical need. Users in such settings process *updates and queries concurrently* and have strict requirements regarding *consistency and reliability*. In the context of this article, 'high performance' means a high degree of concurrent and parallel XML updates and queries. So far, database systems have served as an efficient platform to process relational data concurrently in a consistent and reliable manner. This makes them also attractive as a platform for concurrent XML processing. Vendors of relational database engines such as IBM, Oracle, and Microsoft have extended their products to support XML.

Such off-the-shelf XML extensions of relational database systems pursue the following approach: they store XML documents as character-large-objects (CLOBs) and extend the database engine with functionality for querying and

updating the XML documents. In other words, XML documents are attributes of *document tables*. To speed up queries, separate database tables – so-called *side tables* – materialize views on the content of the XML documents. In a nutshell, this allows to use the relational query engine to efficiently process XML queries. Most of the products implement simple variants of the STORED mapping [12]: they compare the documents to a path expression or tree pattern and store each match as a tuple in a side table. Needless to say, side table content and document content have to remain consistent under updates. Triggers keep the side tables up-to-date: the mapped content of the updated document is deleted from the side tables, and the updated content is inserted. Throughout this article, we refer to the products that follow this approach as *XML extensions*, being aware of various other names for marketing purposes, e.g., XML DataBlade, XML Extender, or XML Cartridge.

Our work investigates to which degree such extensions address the requirements sketched above, and aims at more appropriate solutions if the extensions turn out to be unsatisfactory. Having conducted an experimental evaluation of XML extensions, we observe that performance of concurrent updates and queries is low, for two reasons:

1. Lock contention on document and side tables is in the way of a high degree of concurrency of update and retrieval transactions.

   Example 1: Consider two concurrent transactions that run over the document tables. The first transaction retrieves all `item` elements. The second one updates all `price` elements. Obviously, there is no flow of information between the transactions, i.e., there is no conflict. But the transaction manager of the database blocks one of them if a `price` and an `item` 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 [35], and it results in low *inter-transaction parallelism*. ◇

2. XML extensions load an XML document from the CLOB attribute into an internal representation in main memory for querying and updating. The overhead of this is excessive. On a standard PC, updating a 1 MB-sized XML document and maintaining the side tables takes about 30 seconds. This results in poor parallelism of document operations since the document remains exclusively locked during an update. In other words, off-the-shelf XML extensions currently do not allow for parallel update operations on the same document text even if there is no conflict. The effect is an example of low *inter-action parallelism*.

Our solution to these problems builds on previous work that has investigated the issue of high concurrency for different, more specific application scenarios [4, 25]: these studies follow the *two-level transaction model* and advocate an additional transaction manager that takes application semantics into account. The rationale is to implement a *transaction at the second level* as a set of independent database transactions that may commit early. According to these studies, this may lead to a higher degree of concurrency and consequently to significant performance improvements, in particular if the rate of conflicts is high. It is however unclear how the additional transaction manager should look

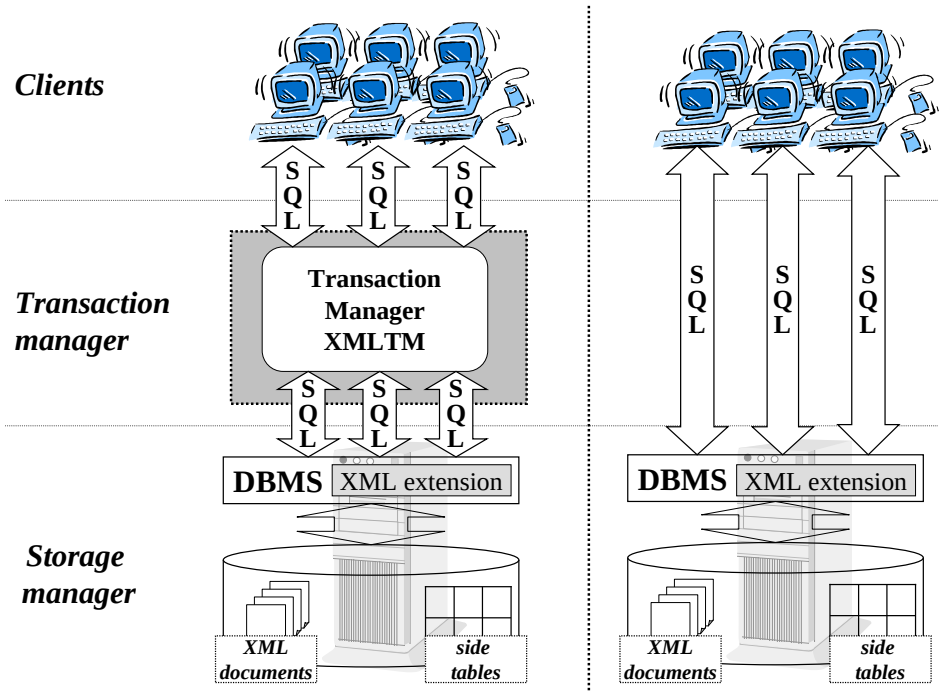

Figure 1: XML processing with an additional transaction manager (left) – with the flat transaction model (right)

like with *XML transactions*[1] as transactions at the second level. In particular, one must decide how to realize isolation and atomicity in this particular context.

Our first contribution therefore is the design and evaluation of such an additional transaction manager – called XMLTM – on top of the database system. Figure 1 (left) serves as an illustration. As part of XMLTM, we propose a locking protocol called DGLOCK to implement isolation. DGLOCK adapts the well-known locking protocol for directed acyclic graphs (DAG locking) [18] to processing of XML data. However, using the graph structure of the XML data for locking is not practical because of its huge size. Furthermore, this structure is not directly available anyhow since XML extensions map XML data to relations. Instead, we propose to use DataGuides [16] as the underlying structure for locking. While the DataGuide has been invented for query evaluation, its deployment in the context of concurrency control has not been investigated before. XMLTM also features a recovery manager that implements atomicity. The lock manager and the recovery manager in combination allow to implement an XML transaction as a set of independent database transactions. An important new optimization is that XMLTM runs these database transactions at low ANSI isolation degree since DGLOCK already guarantees serializability, as we will show.

For evaluation purposes, we have run XMLTM on top of IBM DB2 with the XML Extender for DB2. The bottom line is that XMLTM increases performance of concurrent querying and updating of XML data by an order of magnitude as compared to the common *flat transaction model* (cf. Figure 1 (right)). The experiments also show that the overhead of DGLOCK is small in settings without pseudo-conflicts, e.g., query-only workloads.

[1] An XML transaction bundles XML queries and updates. A transaction manager for such transaction must provide the usual transactional guarantees, notably isolation and atomicity.

Our second contribution is to extend XMLTM for inter-action parallelism, together with an extensive evaluation. This addresses the second problem mentioned before. Our solution transparently splits logical XML documents into physical fragments that different XML queries and updates may update in parallel. A logical document now spans several database tuples. With our fragmentation scheme, all paths from the logical document are completely contained in some fragment. The rationale is that no extra code is necessary to process typical XML queries and updates. Instead, the XML extensions as they stand can accomplish this. Our evaluation shows that the effect of a fine fragmentation granularity is high, as one might expect. But our experiments also show that the overhead for reconstructing the original document remains negligible for reasonable granularities. An investigation of different storage granularities is also important from another perspective: we expect a fine granularity to increase concurrency of queries and updates because of fewer pseudo-conflicts. Hence, we also expected that the performance gain due to XMLTM decreases with finer storage granularities. Surprisingly, this is not the case at all, as we will explain.

Finally, the reader should note two important points. First, it is not necessary to build an XML extension from scratch to implement XMLTM. Instead, database designers can benefit from our approach by adding relatively little code on top of the database system and the off-the-shelf extension. The second point is that our work 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 concurrency and parallelism with XMLTM.

The remainder of this paper is as follows: Section 2 reviews state-of-the-art XML extensions. The XML Extender for IBM DB2 is our running example. Section 3 describes on our transaction manager XMLTM. It also presents our locking protocol DGLOCK for increased inter-transaction parallelism in the XML context. Section 4 in turn proposes features such as fragmentation and storage granularity to raise the degree of inter-action parallelism. Section 5 describes the experimental evaluation of our prototype with IBM DB2 and its XML Extender, together with a detailed discussion. Section 6 covers related work. Section 7 concludes.

2. Database Extensions to Process XML Data

This section reviews the implementation of XML extensions. We use IBM DB2 with the XML Extender as a running example. Nevertheless, the only requirement on the database system to allow for XML extensions is a data type to store large texts, such as data types CLOB, short for Character-Large-OBject, or LONG VARCHAR.

Terminology. 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 graph defined by the data model of the W3C XPath Recommendation [31]. *Matches* to a *path expression* are those sub-graphs of the graph representation that qualify for the patterns of the expression as defined by XPath. XML *contents* are the parts of the document text that correspond to the match.

Example 2: Take the XML document from Figure 2 for example. The figure shows the match of the path expression `/store/auction/price[. > 1000]`. Note that path expression may have more than one match per

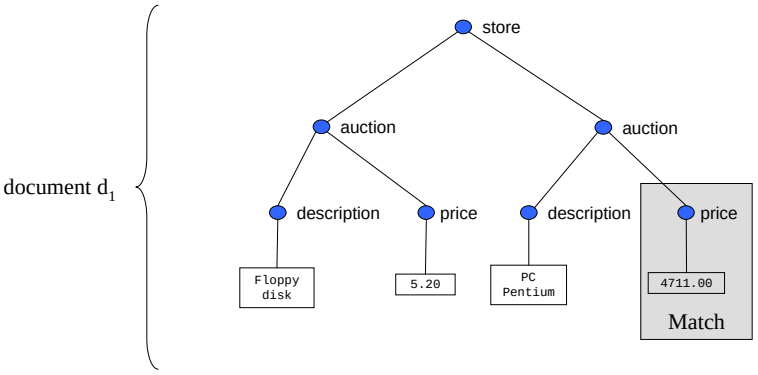

Figure 2: Graph representation of an XML document with a match of a path expression

document. This is for instance the case with path expression `/store/auction/price[. > 1]` and the example 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 to transform it to other database types, e.g., `date` or `decimal`. The extract methods take a path expression as an input parameter. It specifies the location of the content to be extracted. Other methods *update* the XML documents stored as CLOB attributes. One input parameter again is a path expression. Another parameter specifies the new content to replace the old one at the location specified by the path expression. Further methods implement helpful functionality such as loading a document from a file. An SQL statement can then incorporate these additional XML-specific methods.

Example 3: Think of a document table `xmldata` that has a `key` and an `xmltext` attribute (see the 'Document table' in Figure 3). With IBM DB2 and its XML Extender, the following SQL statement retrieves the `key` attribute and the content of all `price` elements from the documents in `xmldata`. All `price` information is converted 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. Note that XML extensions retrieve and update only the XPath 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.

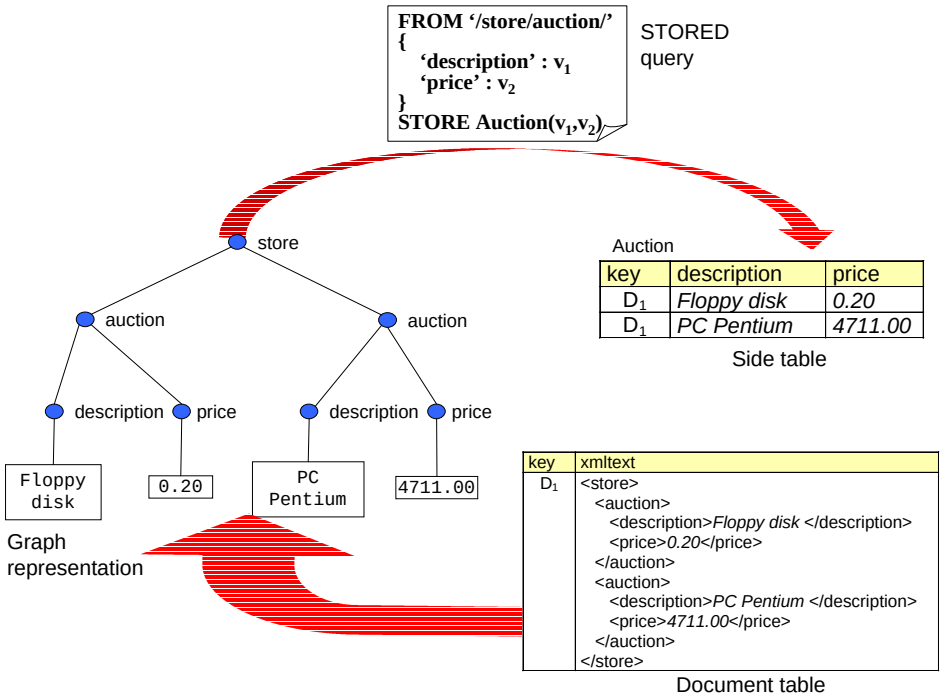

Figure 3: Mapping of an XML document using STORED and side tables

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. Subsequently, we refer to them as *XML views*. Query evaluation then benefits from the efficiency of the relational query engine. The downside is that documents and side tables must remain mutually consistent in case of updates.

To define XML views, a number of mapping strategies has been proposed in the literature. The number and layout of side tables depends on the strategies chosen. The mapping facilities that are commercially available deploy simple variants of so-called *STORED queries* [12]. 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 the graph representation of a given XML document and binds the variables. The **STORE** clause creates a tuple in the side table using the current variable bindings. Figure 3 illustrates the complete mapping process from the original XML document over the graph representation to the side tables. The `key` attribute links the rows of a side table and the original document in the document table.

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. In this case, the side tables serve as an access structure to the document tables. This means that the query inspects only those documents that potentially qualify for the query. These documents are referred to as *candidates*. A query that identifies the set of candidates of a request is a *subsuming query*. The following example illustrates the second case:

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,'//item' )) 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 'item' information is not available in side tables. The sub-query `select key from auction where price < 50` is a subsuming query in the above example. ◇

When it comes to updates, side tables and document tables have to be mutually consistent. IBM's implementation uses database triggers on the `xmltext` attributes of the document tables. When an `xmltext` is updated, the old content of the updated `xmltext` attribute is deleted from the side tables, and the updated content is inserted. The XML Extender automatically generates these triggers when a side table mapping is specified.

Performance Issues and Shortcomings of the Approach. We have carried out a detailed analysis of the aforementioned techniques of qualitative and quantitative aspects using the XML benchmark from [27]. Response times with concurrent updating and querying are very low (cf. Section 5). One explanation for this is that unnecessary lock contention hinders inter-transaction parallelism (cf. Example 1). Another problem is the rigid storage granularity for document texts. The following two sections say how we have addressed these issues.

3. Transaction Management for Concurrent XML Processing

This section describes our transaction manager XMLTM that is the core of our solution. We start by motivating second-level transaction management for XML processing. Section 3.2 explains our implementation of isolation. Section 3.3 in turn addresses atomicity.

3.1. Overview

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. The database community has developed transaction processing functionality that implements these guarantees efficiently. When using XML in contexts that are mission-critical, transactional guarantees are indispensable as well. 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 or storage manager. It allows the application programmer to group a set of requests that require isolation and atomicity to a transaction. The storage manager implements these transactional guarantees for the entire set of requests. Figure 1 (right) serves as an illustration. This option does not require any additional effort to implement transaction processing. The alternative is *transaction management at the second level* [34, 25]. With this model, a set of requests that require isolation and atomicity forms a *global transaction* or a *transaction at the second level*. An additional transaction

manager on top of the storage manager decomposes such a transaction into independent subtransactions, so-called *storage-manager transactions* or *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* and therefore releases its locks early, before the end of its global transaction. – Figure 1 (left) illustrates this architecture.

To compare the two alternatives in the context of XML, we have designed and implemented XMLTM, an additional transaction manager on top of the database system. With XMLTM, the global transactions are *XML transactions*, i.e., they comprise XML queries and updates. XMLTM intercepts the client requests to keep track of global transactions and to control the storage manager transactions. XMLTM maps a request to a set of *operations*. There is an operation for each candidate document. An operation evaluates the query or update on its candidate. Each operation runs as a storage manager transaction. We will show that these transactions can run at a lower ANSI isolation degree [18]. This optimization, together with early release of locks at the storage manager, should lead to much less lock contention.

XMLTM does not rely on conflict checking between requests, as [17], or on the semantics of a specific interface, as [26]. 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. Our solution is a locking technique that is well-suited for XML processing with an arbitrary storage manager. This means that XMLTM does not require that the underlying storage manager is indeed a database system. We only make the weaker assumption that the storage manager implements transactional guarantees for its operations.

In what follows, we refer to the requests that belong to an ongoing global transaction as *active requests.* Update requests currently being processed are *active updates*. *Active queries* in turn are read-only requests.

3.2. Implementation of Isolation at the Second 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 [5]. XMLTM is based on locking as well. XMLTM implements the locking protocol DGLOCK, a new protocol proposed in this article. DGLOCK makes 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. When XMLTM intercepts the client requests, it determines both its structural and content constraints. DGLOCK serializes conflicts resulting from structural constraints. Taking content constraints into account in addition is relatively straightforward and gives rise to further parallelism (Section 3.2.3). The remainder of this subsection now describes how DGLOCK ensures isolation.

3.2.1. Locking on the Structure of XML Documents

At a first level of analysis, one might expect 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 [24] among others. These approaches are only viable if the object graph is physically available for locking. But this typically is

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. Here, *complete* means that for each label path in the document collection there is the same path in the summary. The DataGuide is a data structure that has this characteristic [16]. Furthermore, a DataGuide is concise, i.e., it does not contain any other label paths. In what follows, we assume that a DataGuide has exactly one root. If this is not the case, one can add a virtual root above the various root nodes. The DataGuide has been developed for the OEM data model originally [2]. Transferring the notion of DataGuide to the XML data model is in principle straightforward. One must however decide whether the DataGuide describes only the primary structure, i.e., the containment relationship between elements, or the secondary structure as well, i.e., constructs such as IDREF. Subsequently, the DataGuide is a summary of the primary structure of the XML document collection, and our locking protocol is based on this kind of DataGuide. This turns out to be sufficient for all kinds of XPath expressions, including those that refer to the secondary structure. We will discuss this issue after having presented DGLOCK. Finally, for the sake of presentation, we will describe DGLOCK for XPath expressions that refer to the primary structure only and return to the issue later on.

The DGLOCK Protocol. 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). DGLOCK takes over the idea of granular locking on directed acyclic graphs (DAGs for short) [18] and adapts it to DataGuides. As usual, we differentiate between shared locks (*S* locks) for data that is only read and exclusive locks (*X* locks) for data that is written. Intention locks *IS* and *IX* denote that the request intends to place *S* respectively *X* locks at finer granularity, i.e., lower levels of the graph. Based on the structural constraints of requests, our locking protocol performs the following steps for a new request *s*. The steps are within a critical section:

1. Obtain all path expressions $\mathcal{E}$ in *s* that lead to data that is accessed, i.e., queried or updated by *s*. I.e., extract the structural constraints.

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

3. For each node $n \in \mathcal{N}$ perform the following operations:

   (a) If node *n* is updated by *s*, acquire *IX* locks on all nodes along all paths that lead from the root of the DataGuide to *n*. Request the locks in the order of increasing distance from the root, i.e., from root node to *n*. Then acquire an *X* lock on it.

   (b) If node $n \in \mathcal{N}$ is only read by *s*, acquire *IS* locks on all nodes along at least one path that leads from the root to *n*. As with *IX* locks, aquire the locks from root to *n*. Then acquire an *S* lock on it.

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

Table 1: Lock compatibility with granular locks [18]; incompatibilities marked as '-'.

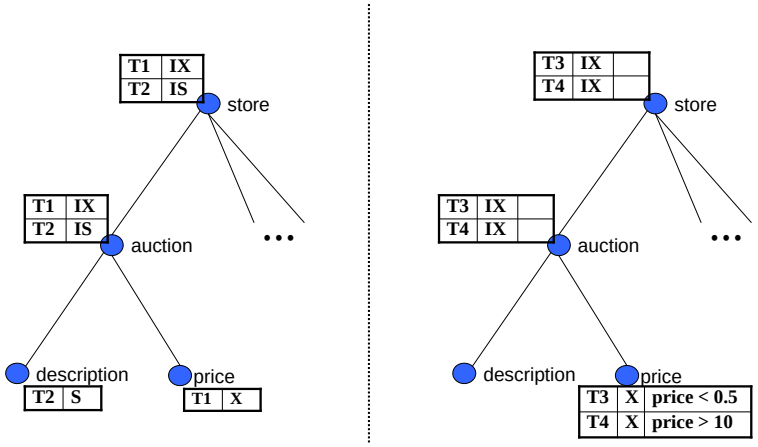

Figure 4: Granular locking on the DataGuide

To grant a lock on some node of the Data Guide, the concurrency control uses the usual compatibility matrix to check for lock compatibility (see Table 1)[2]. 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.

Appendix A contains a summary of our proof that DGLOCK is correct.

Example 5: Figure 4 (left) shows a DataGuide for the XML document from Figure 3 and locks for two transactions T_1 and T_2 (cf. Example 3). T_1 evaluates the following request:

```
update xmldata
set xmltext = db2xml.update(xmltext,'/store/auction/description','XML')
```

The request of T_2 in turn is:

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

XMLTM runs T_1 and T_2 concurrently since all locks granted with DGLOCK are compatible. ◇

Note that locking at coarser granularity is feasible as well. 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.

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

Expectations. We expect our locking protocol to allow for more parallelism of concurrent global transactions than the flat transaction model. This is because DGLOCK guarantees serializability at the second level and XMLTM can commit the database transactions immediately, releasing the locks at the storage manager. On the one hand, the blocking situation described in Example 1 does not occur. 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). We will presently address this second aspect with an optimization for tree-structured DataGuides. Nevertheless, an overhead for the additional lock manager will remain, and it is not clear whether DGLOCK actually improves performance. The experimental evaluation addresses this question.

Discussion. In what follows, we argue why we have used the DataGuide as the underlying structure for locking, as opposed to other summaries of the structure of the document. We will then compare the various kinds of DataGuides with regard to their suitability for our purposes. Finally, we comment on XPath expressions more general than the ones explicitly considered so far.

The first question is whether or not the DataGuide is appropriate for locking. This question comes up for the following reasons: it consumes main memory, and its maintenance requires additional CPU time. On the other hand, locking schema information such as DTDs is not viable, simply because it is not mandatory according to the XML specification. An alternative is to use the so-called *active DataGuide* for locking. An active DataGuide is a DataGuide that only contains paths and nodes for active requests. Thus, the active DataGuide is typically smaller than the DataGuide. Locking on an active DataGuide is the same as with a DataGuide: DGLOCK is directly applicable. Maintenance of the active DataGuide is done 'on-the-fly'. This means that DGLOCK creates the nodes when they are needed, and it deletes a node as soon as there is no lock on it and its descendants. However, recall that path expressions may contain wildcards, and dealing with them requires much effort with an active DataGuide. One has to determine all paths that qualify for the wildcard. If a path index is not available a scan of the document collection would be required to determine those nodes. However, wildcards frequently appear in path expressions. It also turns out that the maintenance argument is not really important since the number of such changes is rare in practice. For instance, the number of such changes with the XML benchmark [27] is negligible. For these reasons, we do not follow up on the alternative of active DataGuides in the remainder of this study.

Literature has identified various kinds of DataGuides, notably *minimal* ones and *strong* ones [16]. 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. Note that our argumentation why DGLOCK is correct is independent of the specializations of the notion of DataGuide introduced above. 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 on the DataGuide. With a DataGuide that is not strong, a lock on one of its nodes would lock several label paths in the general case. This effect does not occur with strong DataGuides because of the one-to-one correspondence mentioned above.

Finally, let us briefly look at XPath expressions that refer to the secondary structure as well. The notion of 'match' used in Step 2 of DGLOCK needs to be adapted to cope with secondary-structure XPath expressions and primary-structure DataGuides. This is a technical problem whose solution is straightforward. A full description is beyond the scope of this paper. The important point is that lock acquisition continues to be based on the primary structure only.

3.2.2. Reducing the Isolation Degree at the Storage Level

The objective of this subsection is to reduce the isolation degree at the storage level in order to have better performance: Our argumentation why DGLOCK is correct is based on the 'repeatable read' characteristic of ANSI isolation degree 3. The following optimization is based on the observation that storage manager transactions with XMLTM do not read data objects repeatedly. This allows to run storage manager 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. Note that we cannot go below ANSI isolation degree 2, i.e., 'read committed', 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 storage manager transaction that deletes a side-table tuple of the old document and inserts one of the new version of the document. We must prevent the other storage manager transactions from seeing the intermediate database state. This requires isolation level 2, i.e., 'read committed'. Our experimental evaluation will investigate how the reduced isolation degree affects performance.

3.2.3. Locking on the Content of XML Documents

Locking a node with DGLOCK as described so far, may unnecessarily prevent other transactions from accessing other parts of the database.

Example 6: Assume that our collection consists of documents such as this one:

```
<auction>
    <description> An example offer </description>
    <shipping> Swissair </shipping>
</auction>
```

The DataGuide for such documents has only three nodes ('auction', 'description', and 'shipping'). Updating the 'Swissair' shipping would place an exclusive lock on the 'shipping' node. This seems to be suboptimal if there is only one or a few `shipping` elements with this value. ◇

The example points to a general problem when locking is based on structural information only. DGLOCK therefore combines stuctural locking with locking based on the content of the XML documents and the content constraints of the requests. In more detail, it incorporates predicate locking as described by [10] on element content.

Example 7: Consider again the DataGuide from Example 5. T_3 and T_4 are requests that also include content constraints. Figure 4 (right) illustrates the locking for T_3 and T_4. Since their predicates on the `price` elements do not overlap, their locks are compatible and DGLOCK runs these transactions concurrently. ◇

We expect that locking based on structure and content in combination gives way to a significantly reduced degree of lock contention and a higher degree of concurrency.

3.3. Implementation of Atomicity at the Second Level

Atomicity means that all update requests of a transaction are either executed to their completion or not at all. Atomicity is typically implemented by means of recovery. When a transaction manager at the second level decomposes a global transaction into several storage manager transactions, and these transactions commit independently of each other, it must comprise recovery functionality as well. To do so, XMLTM implements a combination of undo-recovery and redo-recovery [18]. Undo-recovery is necessary to compensate the effects of early commits when a global transaction aborts. Redo-recovery avoids cascading aborts in our particular setting. The remainder of this subsection explains these points in more detail.

Undo recovery requires to log begin-of-transaction and end-of-transaction markers. Each such transaction marker also carries the identifier that was assigned to the global transaction at its beginning. This allows to determine the global transactions that have 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 complete document text to the log before a request can apply changes to the document. This yields a *before-image* of the document text, to be restored when undoing a transaction. It remains to be said what happens to the side tables: triggers keep derived information such as side tables up-to-date. Note that 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. This is the reason why second-level logging of changes of side table content is not needed.

Note that DGLOCK allows transactions to concurrently update the same document if their requests do not conflict. This provides *inter-transaction parallelism* for document operations. However, undo recovery as described so far would produce cascading aborts. This is because the logging granularity with XMLTM is larger than the locking granularity. In this situation, when several transactions have updated the same document concurrently and one of them is aborted, the other ones would have to abort as well to prevent from lost updates. But DGLOCK ensures that only non-conflicting operations change the document text concurrently. This allows the transaction manager to re-apply the changes of already committed transactions (redo-recovery). XMLTM restores the before-image and redoes the (non-conflicting) updates on the before image that are part of younger committed storage manager transactions. To allow for redo-operations, XMLTM also writes the update request and its parameters to the log. Writing the log and changing the document text occur in the same database transaction. This guarantees that there always is a log entry if the document has been changed. This implements atomicity and makes cascading aborts transparent to the clients.

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 document texts is not necessary. The downside

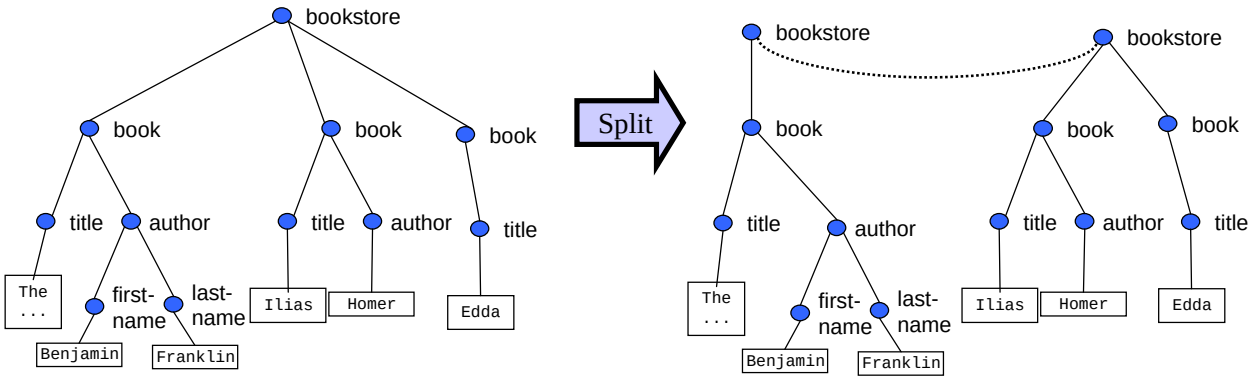

Figure 5: XML document and XML fragments of the document

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. Increasing the Performance of Document Operations

Since XML extensions store a document as a whole in a CLOB attribute, updating a large XML document with XML extensions lasts rather long. Many products parse the complete document text and load it into an internal representation similar to the graph representation. While the document text is updated, it is not available for other document operations at the same time: inter-action parallelism is low. This is however unnecessary since DGLOCK guarantees that concurrent updates do not conflict.

An update request may affect only a tiny portion of a large document text. This is unnecessarily costly because XML extensions generate an internal representation of the entire document text. An obvious measure is to split the document into several fragments on the physical level. This should avoid the problem just sketched and should lead to higher inter-action parallelism. The details however are somewhat intricate, and we introduce some terminology to describe them. A *logical document* is a complete XML document from the application perspective. An *XML fragment* is a sub-graph of the graph representation of a logical XML document such that the following holds: There is a set of nodes of the XML document, subsequently referred to as *base nodes*, such that (1) the base nodes are consecutive siblings in the XML document, (2) the base nodes are part of the XML fragment, (3) the subtrees rooted at each base node are in the XML fragment, (3) all nodes from the root to a base node, together with the edges connecting them, are part of the XML fragment, and (4) nothing else is in the fragment. A *fragmentation* is a set of fragments such that (1) a base node of a fragment is not part of another fragment, and (2) each node of the XML document is in a fragment.

Example 8: Figure 5 shows fragmentation of a logical document into two fragments. The 'book' nodes are the base nodes of the two fragments. ◇

Our implementation connects the different consecutive fragments corresponding to a document (cf. the dashed line in the figure). This allows to re-construct the original document if required.

While the idea of fragmenting XML-documents has been proposed earlier [6], previous definitions are different in that they avoid the occurrence of nodes in several fragments. The reason why our definition is different is not another performance gain in the first place, but ease of implementation: each path leading from the root to some node in the fragment is the same as in the logical document. Evaluation of typical path expressions on fragments according to our definition can use the facilities of the XML extensions without any extra code. Only in some situations, it is not sufficient to look at each fragment separately. This occurs if path expressions do not only 'go downwards', but also in the direction of other axes. Another case is that a match is not fully contained in a fragment. However, in many practical settings such as [27] such path expressions hardly occur. Otherwise we reconstruct the original documents for query processing. The idea to shift query and update processing to the XML extension as much as possible is in line with our design philosophy to keep our additions to the XML extensions as slim as possible.

Generation of the XML fragments for a logical document as follows: A system-wide parameter k specifies the average number of nodes in a fragment. We denote the average number of fragments per document as *document storage granularity*. Each fragment is stored as a row in the document table. The effect is that the granules of query and update processing are not the logical documents any more, but the fragments since each match of a typical path expression is local to a fragment. Hence, we expect that fragmenting leads to performance improvements with data centric processing[3].

Discussion. Fragmentation is applicable with both the flat and the two-level transaction model. We anticipate a higher degree of inter-action parallelism with both models. Furthermore, and less obvious, we expect that fragmentation also reduces the degree of lock contention with the flat model. This would lead to higher inter-transaction parallelism, and the relative difference between the flat and the two-level transaction model might become smaller. We are therefore interested in quantitative characteristics of fragmentation with both transaction models, and the experimental evaluation will address this.

5. Experimental Evaluation

We have carried out numerous experiments to assess our proposed extensions in quantitative terms. We want to find out if our extensions to the commercial XML extensions really improve performance of concurrent queries and updates of XML data. We focus on comparing response times and throughput of client requests with XMLTM to those with the flat transaction model. Another important question is if fragmentation is also suitable to reduce lock contention. We also want to determine a good physical data organization for pratical workloads, i.e., a good combination of side tables and fragmentation regarding the overhead for side-table maintenance and fragment integration.

We describe our experimental setup in Section 5.1. Section 5.2 reports on our findings and discusses the outcome

[3] Read and update operations with *data-centric processing* access small portions of the documents, similarly to OLTP transactions in relational databases. With *document-centric processing* in turn, access tends to go to documents as a whole, as typically is the case with document management.

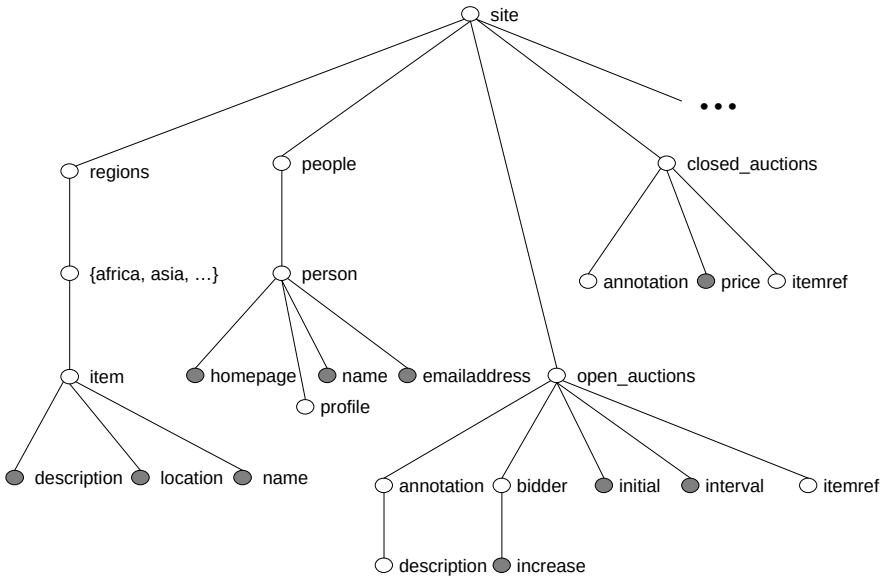

Figure 6: Excerpt of the DataGuide of the experimental documents with side table support for the grey nodes

of the experiments.

5.1. Experimental Setup

XML Documents. The XML documents in our experiments have been created with the document generator `xmlgen` of the XML benchmark project [27]. The scaling factor was 1.0 (standard), i.e., 100 MB of document data. In our experiments, the collection consists of 100 logical documents. We investigate average document storage granularities of 1, 10, and 100 fragments per logical document. With these numbers, the value of the storage granularity parameter k, i.e., the average number of graph representation nodes, is around 10.000, 1.000, and 100, respectively. The database size with the document tables and the side tables is about 300 MB. Figure 6 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. The figure also depicts the DataGuide 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 the two-level transaction model appear in a better light. Our experiments revealed that our workload already exhausts the resources of the standard PC that we used in our study. 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 system performance, as compared to a setting with think times. We distinguish between two different access patterns, namely *data centric* and *document centric processing*. We have synthetically generated the requests for both of these access patterns. Data centric query processing takes

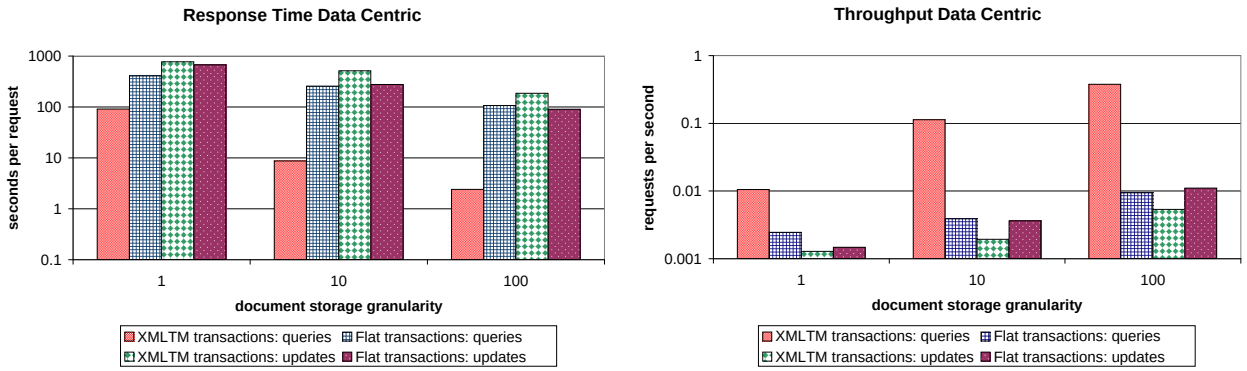

Figure 7: Data centric update and query performance: response times (left) – throughput (right)

requests similar to the queries of the XML Benchmark project [27]. Since [27] does not cover updates, we generated a set of data centric update requests. Our request invocations 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 has to access one or several fragments. The results of these requests are not complete logical documents. So it is not necessary to rebuild a document from its fragments with data centric processing. Document centric processing in turn retrieves or updates one logical document, and it rebuilds the logical document. In our setting, document centric accesses distribute evenly among the logical documents, i.e., there are no hotspots.

Hardware and Software. We have run our experiments on an off-the-shelf PC with one Pentium III Processor and 512 MB of RAM. The PC runs the Microsoft Windows 2000 Advanced Server operating system software. The second-level transaction manager 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. 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.

5.2. Outcome and Discussion of the Experiments

Data Centric XML Processing with Side Tables: Effect of Storage Granularity. Our first series of experiments investigates data centric processing on the XML documents in the presence of side tables. Figure 7 graphs average response times and throughput for the document storage granularities of 1, 10, and 100. Note that the figure uses a log-scale axis for response times and throughput. This will also be the case with subsequent figures if not stated differently. A first observation is that the performance of data centric processing increases with finer document storage granularities: updates with flat transactions for instance yield average response times of nearly 700 seconds

with storage granularity 1. With a storage granularity of 100, it is only 100 seconds per update. 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 a finer granularity depends on the choice of the transaction manager. Query performance with XMLTM transactions increases by more than an order of magnitude from storage granularity 1 to 100. 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 granularity 1 to 100. With XMLTM transactions, it is only 4. Summing up, XMLTM transactions yield higher response times of update requests for any document storage granularity 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 of the fragment to the log before updating the document. Moreover, recall that it 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 7 shows: query throughput with XMLTM transactions with storage granularity 100 is more than an order of magnitude higher than with flat transactions. Moreover, we allow queries with XMLTM transactions to run at ANSI isolation degree 2 ('read committed'). Flat transactions in turn must run at ANSI isolation degree 3 ('serializable'). Since a single update request deletes and inserts to many side tables, 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 convoi [18]: queries wait for the current update request to finish. Then the query is processed. The following query waits until the following update request has finished, and so on.

Data Centric XML Processing with Side Tables: Effect of Conflict Ratio. A follow-up question on these results is how the conflict ratio between requests affects blocking with flat transactions. We have run experiments with two different data centric 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 some concurrent queries. The second type of access pattern has a document table conflict ratio of 20%. Our hypothesis was therefore that the second access pattern yields better query performance. We were surprised to find out that it does not hold true. There was no performance gain for flat transactions when the access pattern had a conflict ratio of 20%. 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. Another reason why XMLTM outperforms the flat transaction model is that the amount of data required for locking with flat transactions is immense. For instance, a transaction that updates a typical document with 1.8 MB of XML text data requires more than 1,000 database locks,

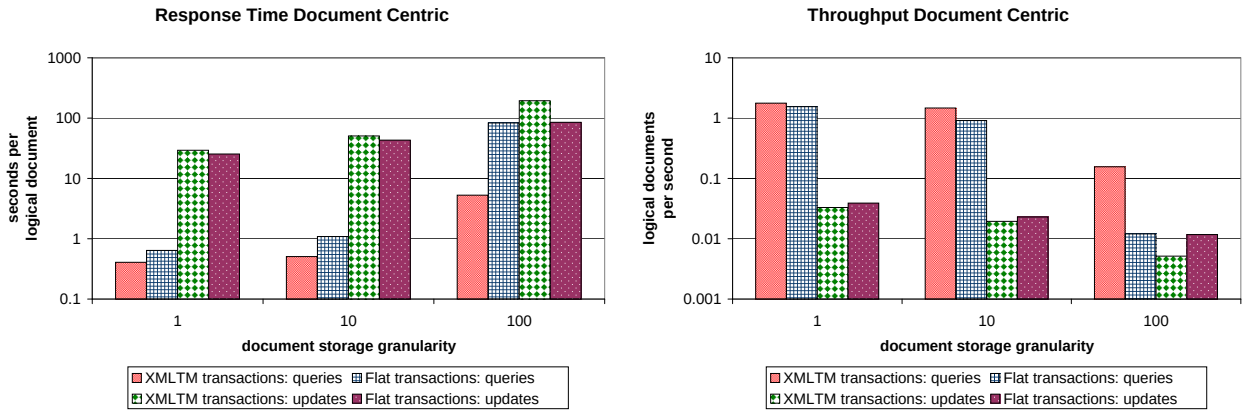

Figure 8: Document centric update and query performance: response times (left) – throughput (right)

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, the overhead for our locking protocol DGLOCK is significantly smaller.

Document Centric Processing. The other important type of request processing with an XML repository is document centric processing. Figure 8 graphs our findings for document centric processing with different document storage granularities. As with data centric processing, the additional log and commit processing with XMLTM transactions yields higher update request response times as compared to flat transactions. With coarse storage granularities, this overhead is less than 20%. But it is by a factor of two for storage granularity 100. On the other hand, queries benefit from the second level transaction management: query throughput is 2 times higher with XMLTM transactions and storage granularity 100 as compared to flat transactions. Especially noteworthy with flat transactions is that the database system changes the locking strategy from row-level locking to table locking when going from storage granularity 10 to 100. This leads again to the convoi effect between updates and queries: query and update performance with flat transactions are identical with granularity 100, as Figure 8 shows.

Another important issue is the effect of document storage granularity. As expected, document centric processing benefits from a coarse granularity. Up to 10 fragments per logical document, there is not much overhead to integrate the fragments to the logical document. Instead, integrating 100 fragments yields response times that are not competitive. Our findings for document centric updates are similar: updating a logical document, including the rebuild from its fragments, takes up to 6 times longer with storage granularity 100 as compared to storage granularity 1.

Data Centric Processing without Side Tables. Another series of experiments has compared performance of data centric processing in a setting with side tables to one without side tables. Figure 9 reports on the outcome of these experiments with different document storage granularities using flat transactions. Note that this particular figure does not use a log-scale coordinate axis. Our first observation relates to the effect of document storage granularity. The results are as expected with a fine storage granularity, i.e., with 100 fragments per logical document: query and update

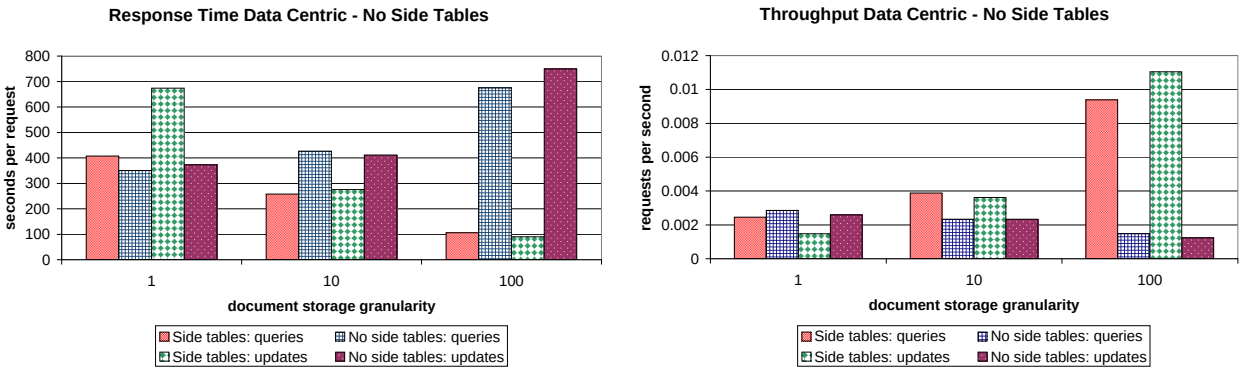

Figure 9: Data centric update and query performance without side tables: response times (left) – throughput (right)

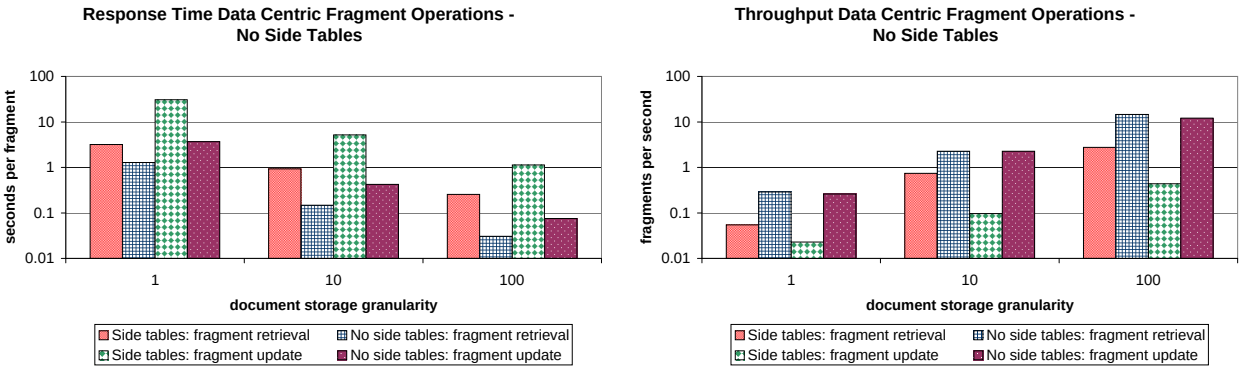

Figure 10: Performance of fragment operations with/without side tables: response times (left) – throughput (right)

performance without side tables is more than 6 times lower than with side tables. However, the results for coarser granularities such as 10 or 1 fragment(s) per logical document are rather unexpected. Having side tables improves performance only by a factor of 2 with granularity 10. The picture is even worse with granularity 1.

We conducted an additional series of experiments to shed more light on this surprising finding. These experiments investigated response times and throughput of queries and updates regarding fragments, i.e., we have measured the performance of updating a fragment or performing a query operation on it (response time) as well as the number of such operations performed per second (throughput). As Figure 10 shows, performance of update operations is significantly lower with side tables than without. The reason is that side tables require maintenance in the case of an update which is included in the numbers in the figure. Hence, Figure 10 quantifies the overhead of side table maintenance. The figure shows that the overhead is close to an order of magnitude with coarse document granularities, i.e., granularity 1, and even more with finer granularities.

However, the benefit from side tables, i.e., the decrease in the size of the candidate set, does not compensate this for coarse storage granularities, as Figure 11 shows. The figure shows the number of updates of fragments that are required for processing a request with the different storage granularities. Without side tables, the query processor has to fetch all documents stored. With side tables in turn, the situation is more differentiated: with very fine storage granularity

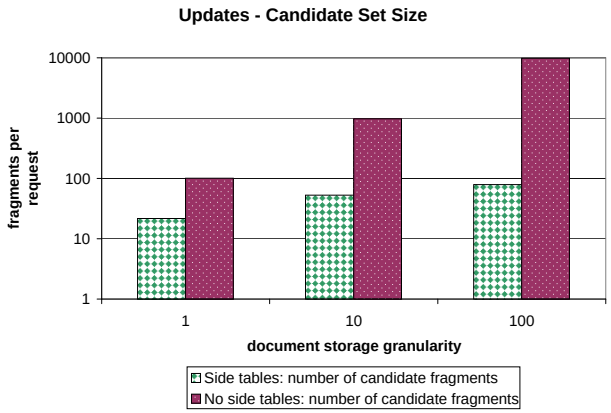

Figure 11: Number of document operations with/without side tables

the candidate set is two orders of magnitude smaller than the total number of documents. With a coarse granularity of 1 instead, the size of the candidate set is 20% of all documents. Recall our previous result that the overhead of a document update for coarse granularity is nearly an order of magnitude. In combination with the experiments on the size of the candidate set, this explains why updates for coarse storage granularities perform better without side tables as compared to a setting with side tables.

Summary of Experimental Findings. Our experiments with side tables have addressed the question if XML processing with our transaction manager XMLTM at the second level outperforms the flat transaction model. It turned out that this is the case for query performance. It improves by an order of magnitude when using XMLTM as compared to the flat transaction model. Update performance in turn is two times lower with XMLTM than with flat transactions due to the additional logging overhead with XMLTM. A surprising finding is that a fine storage granularity does not reduce the degree of pseudo-conflicts with the flat transaction model. Query performance improvements by XMLTM are even larger with fine storage granularities than with a coarse one. Regarding the question of physical data organization for XML processing, it turned out that side tables yield high performance improvements with fine storage granularities. With coarse storage granularities in turn, there is basically no improvement for concurrent XML processing by side tables.

6. Related Work

The XML community has put much effort into XML query languages. We will presently address this issue very briefly. 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. Processing of concurrent querying and update of XML data has received only little attention so far. This section finally reviews previous work on this problem.

Languages to Query and Update XML. A variety of query languages for processing XML (or semi-structured data) have been proposed, e.g., [1, 2, 7, 8, 9, 11, 15, 30, 33, 32]. So far, only [30] has explicitly considered updates. They suggest a respective extension to XQuery [32]. They also provide experimental results for an implementation on top of a relational database system. [30] only deals with updates in isolation. There is no concurrency of queries and updates. This article in turn has proposed a practical protocol to guarantee consistent and reliable processing of concurrent queries and updates, together with an extensive experimental evaluation based on a full implementation.

XML Repositories. A meaningful classification[4] of XML repositories is into (1) extensions of commercial RDBMSs, (2) native XML stores [20, 21, 29, 36], and (3) hybrid approaches, e.g., [13]. A common approach when extending database systems is to use SQL to access XML data, e.g., [19, 23, 22]. CLOB attributes of conventional database tables store the XML documents. Specialized operators extend the respective SQL dialect while their implementation provides the functionality for XML-specific processing. XPath expressions typically specify the patterns to access XML content [31]. However, all of the aforementioned approaches 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) where the storage manager implements transactional guarantees.

XML-to-DBMS Mappings. Several approaches how to map XML content to databases have been proposed in the literature, e.g., [12, 14, 28, 37]. Implementations that are commercially available deploy simple variants of the STORED mapping as introduced by [12]. A notable difference is that the commercial implementations also store the original document texts, as opposed to overflow graphs in [12]. The way these systems keep mapped XML content and original document mutually consistent leads to low performance for natural workloads, i.e., concurrent queries and updates. Our current work in turn proposes a transaction manager XMLTM that solves this problem.

Transactional Guarantees for Processing of XML Data. Relatively little previous work has dealt with transaction processing of XML data. [26] investigates isolation of simple DOM operations on single XML documents. The authors define commutativity of these operations when accessing the same node in the graph representation of the document and derive alternatives for pessimistic and optimistic concurrency control. We think that our approach in turn is more general than [26], and that it is applicable to a much broader range of practical application scenarios for several reasons: our locking protocol DGLOCK does not assume a fixed API for conflict definition in order to guarantee serializability. DGLOCK does not rely on a particular storage layout. In contrast to [26], our current work also takes recovery into account. Finally, [26] does not have an experimental evaluation. On the other hand, designing XMLTM has given rise to a variety of intricate questions, which we could address only by means of experiments based on a full implementation.

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

7. Conclusions

Efficient concurrent updates and queries of XML data in a consistent and reliable way is an important practical problem. One wonders to which degree XML extensions of commercial database systems solve this problem. A preliminary investigation has shown that relying only on the transaction processing functionality of the database system comes too short: concurrent and parallel XML queries and updates do not perform well due to lock contention and an impractical storage granularity of XML documents. A more promising approach instead uses an additional transaction manager on top of the database system. It implements isolation and atomicity for global transactions.

Our first contribution is the design, implementation, and evaluation of XMLTM, a transaction manager for concurrent processing of XML data. Building on previous work on locking in DAGs [18], we propose a granular locking technique DGLOCK that implements isolation for concurrent XML processing. DGLOCK captures the structural constraints of requests and places locks on the DataGuide. The rationale behind DataGuides is to have a compact summary of the XML document collection that serves as the underlying structure for locking. This application of DataGuides has not been investigated before. Another important innovation is that DGLOCK allows to run the database transactions that implement XML processing at a lower ANSI isolation degree and to do an early commit. This avoids lock contention on side tables and document tables. 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.

The second contribution of this article has been the extension of XMLTM to facilitate a higher degree of interaction parallelism, together with an extensive evaluation. Our implementation stores logical documents as several fragments. Our fragmentation scheme is new and differs from previous ones to allow for better re-use of the facilities of the XML extensions. Our experimental evaluation has confirmed that a fine storage granularity yields performance improvements up to an order of magnitude for data centric processing. At the same time, the overhead for integrating fragments to logical documents is pleasingly low. Document fragmentation also does not reduce the rate of pseudo-conflicts with flat transactions. As a consequence, the performance improvements with XMLTM are larger with a fine document storage granularity. This invalidates our expectation that the flat transaction model would particularly benefit from a fine storage granularity.

Given the results of this study, it is relatively simple to enhance existing XML extensions with the functionality provided by XMLTM. To increase concurrency of XML processing, locking with commercial XML extensions should take the semantics of XML into account, similarly to our locking protocol DGLOCK.

References

[1] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web – From Relations to Semistructured Data and XML*. Morgan Kaufmann Publishers, 2000.

[2] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The Lorel Query Language for Semistructured Data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.

[3] G. Alonso, A. Fessler, G. Pardon, and H.-J. Schek. Correctness in General Configurations of Transactional Components. In *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, 1999, Philadelphia, USA*, pages 285–293. ACM Press, 1999.

[4] 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.

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

[6] K. Böhm, K. Aberer, E. J. Neuhold, and X. Yang. Structured Document Storage and Refined Declarative and Navigational Access Mechanisms in HyperStorM. *VLDB Journal*, 6(4):296–311, 1997.

[7] P. Buneman, M. F. Fernandez, and D. Suciu. UnQL: A Query Language and Algebra for Semistructured Data Based on Structural Recursion. *VLDB Journal*, 9(1):76–110, 2000.

[8] S. Ceri, S. Comai, E. Damiani, P. Fraternali, S. Paraboschi, and L. Tanca. A XML-GL: A Graphical Language for Querying and Restructuring XML Documents. *WWW8 / Computer Networks*, 31(11-16):1171–1187, 1999.

[9] D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources. In *Selected Papers – The World Wide Web and Databases, Third International Workshop WebDB 2000, Dallas, Texas, USA*, volume 1997 of *Lecture Notes in Computer Science*, pages 1–25. Springer, 2001.

[10] P. Dadam, P. Pistor, and H. Schek. A Predicate Oriented Locking Approach for Integrated Information Systems. In *Proceedings of the IFIP 9th World Computer Congress, Paris, France*, pages 763–768. North-Holland/IFIP, 1983.

[11] A. Deutsch, M. F. Fernandez, D. Florescu, A. Y. Levy, and D. Suciu. A Query Language for XML. *WWW8 / Computer Networks*, 31(11-16):1155–1169, 1999.

[12] 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.

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

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

[15] N. Fuhr and K. Großjohann. XIRQL – An Extension of XQL for Information Retrieval. In *Proceedings ACM SIGIR 2000 Workshop on XML and Information Retrieval*. ACM Press, 2000. Available at: http://www.haifa.il.ibm.com/sigir00-xml/final-papers/KaiGross/sigir00.html.

[16] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 436–445. Morgan Kaufmann, 1997.

[17] 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.

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

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

[20] Z. G. Ives, A. Y. Halevy, and D. S. Weld. Integrating Network-Bound XML Data. *IEEE Data Engineering Bulletin*, 24(2):20–26, 2001.

[21] 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.

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

[23] 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.

[24] 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.

[25] 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.

[26] S. Helmer and 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.

[27] 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.

[28] 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.

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

[30] 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 on Management of data, May, Santa Barbara, CA USA*, pages 413–424, 2001.
[31] The World Wide Web Consortium. XML Path Language (XPath) Version 1.0. http://www.w3.org/TR/xpath, Nov. 1999.
[32] The World Wide Web Consortium. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery, Feb. 2001.
[33] A. Theobald and G. Weikum. Adding Relevance to XML. In *Selected Papers – The World Wide Web and Databases, Third International Workshop WebDB 2000, Dallas, Texas, USA*, volume 1997 of *Lecture Notes in Computer Science*, pages 105–124. Springer, 2001.
[34] G. Weikum. Principles and Realization Strategies of Multilevel Transaction Management. *ACM Transactions on Database Systems (TODS)*, 16(1):132–180, 1991.
[35] 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.
[36] L. Xyleme. A Dynamic Warehouse for XML Data of the Web. *IEEE Data Engineering Bulletin*, 24(2):40–47, 2001.
[37] M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura. XRel: A Path-Based Approach to Storage and Retrieval of XML Documents Using Relational Databases. *ACM Transactions on Internet Technology*, 1(1):110–141, 2001.

A. Correctness of DGLOCK

Our correctness criterion is conflict-serializability [5]. Our argumentation why DGLOCK is correct is based on the *DAG Lock Correctness Theorem* from [5, p. 73]. It states that granular locking on DAGs in combination with strict two-phase locking guarantees conflict-serializable schedules for the flat transaction model. In a multi-level setting in turn, one has to show that all levels are correct and that the order of conflicts is the same at all levels [3]. We first say why the second level is correct.

Second Level. The DAG Lock Correctness Theorem for the flat transaction model assumes that the data objects accessed and the objects locked are identical. But this is not the case with DGLOCK at the second level: DGLOCK locks nodes of the DataGuide, as opposed to the nodes in the graph representation of the documents. So we must say why locking on the DataGuide implies locking of the 'correct' nodes in the graph representation, i.e., those nodes that qualify for the structural constraints as well as their descendants. Our argument is based on the property of conciseness of DataGuides. Given the path expressions of a request, DGLOCK acquires locks on those nodes in the DataGuide that qualify for the path expression. One can see this also as implicitly placing locks on nodes of the graph representation of the documents. We call these implicit locks *graph locks.* Locking nodes along a path of the DataGuide places graph locks on all nodes along the same paths in the graph representation. Graph locks have the same lock type as the locks in the DataGuide. Since the DataGuide is complete we do indeed lock the correct nodes. This also holds for update requests: they only update the descendants of the nodes locked. Since we use granular locking, and since lock acquisition and lock release is strictly two-phase, the DAG Lock Correctness Theorem also applies to the graph locks. The schedules at the second level are therefore serializable.

First Level. To see why the schedules at the storage level are correct recall that DGLOCK blocks requests in case of conflicts at the second level. Hence, transactions at the storage level execute serially in case of conflicts at the higher level. This implies that the order of these conflicts is the same at both levels. To see why the level itself is correct, note that schedules of storage manager transactions are serializable. This is because the database system uses strict two-phase locking on database pages or rows. This concludes our argumentation why DGLOCK is correct.