Transaction Management for XML Stored in Relational Database Systems*

Peter Pleshachkov

Institute for System Programming RAS, Russia
peter.pleshachkov@gmail.com

Abstract. Nowadays, modern commercial relational database management systems (RDBMS) enable functionality to store, query and update XML data. One of the key problems is efficient handling of concurrent access to XML data stored in RDBMS.
In this paper, we present an efficient concurrency control method for XML data stored in RDBMS. Our approach is based on Grabs et. al. work on XMLTM, an XML transaction manager built on top of RDBMS. We investigate limitations and drawbacks of XMLTM, and propose solutions to overcome them. The paper summarizes our results to date and identifies directions for future work.
The goal of our research is efficient and general transaction manager for concurrent XML processing in RDBMS.

1 Introduction

Today a lot of applications use XML to store and exchange data. Their amount is quite significant and continues to grow.

To answer new requirements, most commnercial RDBMS vendors extend their products with XML support. Current systems (e.g. MS SQL Server 2005) use binary large objects (BLOBS) to store XML documents in a certain internal format. Users can create special indices on XML documents to speed up queries. The systems enable specialized query languages support such as XQuery/XPath.

However, current implementations use internal RDBMS locking for concurrency control. This approach means that the granularity of locking is the whole XML BLOB, while access is made only to some XML nodes in it. Obviously this creates a bottleneck when concurrent access is an issue.

To solve this problem Grabs et. al. [3] proposed to use multilevel transaction management[10]. According to this approach, an additional transaction manager XMLTM is built on top of RDBMS. It schedules concurrent updates and queries using DGLOCK protocol. XMLTM decomposes XML transaction into a number of small independent transactions over RDBMS. They are called DB transactions. And as XMLTM can commit DB transactions early, the locks on the entire BLOB are released early thereby allowing other transactions waiting

* This work was partially supported by the grant of the Russian Basic Research Foundation (RBRF) N 05-07-90204-b.

for the locks to proceed. Moreover, DB transactions are run at a lower ANSI isolation level (Read Commited). These factors allow to improve the parallelism of XML transactions.

We consider this approach to be the most promising, since XMLTM does not rely on the underlying storage scheme used by relational databases for XML documents. This means that it will work even for native XML databases.

Unfortunately there are certain limitations and drawbacks in the Grabs method. Namely: (1) supported query language is a restricted version of XPath, whereas modern RDBMSs support more flexible and powerful XQuery language, (2) DGLOCK protocol is based on multi-granularity locking scheme [4], which is too restrictive for a wide range of XML-operations and (3) DGLOCK does not guarantee serilizability of produced schedules. Also (4) XMLTM does not support ordered XML and (5) performance of updates degrades due to significant overhead needed to log before-images of updated documents by XMLTM.

The goal of our research is to overcome these XMLTM drawbacks and limitations and design a more efficient and general transaction manager called SXTM (Semantic XML Transaction Manager) for concurrent XML processing in RDBMS. The key contributions of our research are:

- We proposed a locking protocol to schedule XML transactions, which exploits the semantics of XML queries and updates to improve the degree of concurrency in comparison with original DGLOCK protocol.
- We introduced a way to reduce a size of log which is needed to guarantee atomicity of XML transactions.

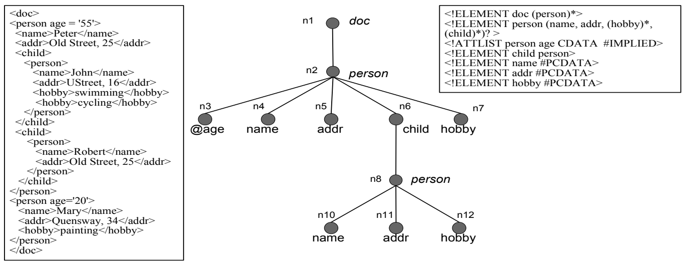
The remainder of the paper elaborates our contributions in Section 2. In Section 3 we discuss previous work on concurrency control for XML data. In Sect 4 we give directions for future work and make conclusions.

2 Results to Date

2.1 Preliminaries

In SXTM we use XQuery to query XML documents. To update XML documents we use primitive update operations like insert, delete and rename. It is clear, that any complex update operation may be expressed via these operations. We consider three kinds of insert operations. The operations `InsertInto(locpath, constr)`, `InsertAfter(locpath, constr)` and `InsertBefore(locpath, constr)` insert new node defined by `constr` as the last child, following sibling and preceding sibling respectively for each *target* node defined by `locpath`. Here `Locpath` (location path) is the basic operation used both in XQuery queries and update operations. The operation `Delete(locpath)` removes the *target* subtrees defined by `locpath`. The operation `Rename(locpath, QName)` assigns new name defined by `QName` to the *target* nodes defined by `locpath`. We denote the intermidiate nodes and destination nodes of `Locpath` as `ITM` and `DST` respectively. Below in this paper, we use `II` , `IA`, `IB`, `RN`, `D` and `LP` to denote `InsertInto`, `InsertAfter`, `InsertBefore`, `Rename`, `Delete` and `Locpath` operations respectively.

In the Fig. 1 we present a sample of XML document *Gtree*, its DataGuide and DTD. We will use *Gtree* document in all examples throughout the paper.



```
<doc>
<person age = '55'>
  <name>Peter</name>
  <addr>Old Street, 25</addr>
  <child>
    <person>
      <name>John</name>
      <addr>UStreet, 16</addr>
      <hobby>swimming</hobby>
       <hobby>cycling</hobby>
    </person>
  </child>
  <child>
     <person>
       <name>Robert</name>
       <addr>Old Street, 25</addr>
     </person>
   </child>
</person>
<person age='20'>
   <name>Mary</name>
   <addr>Quensway, 34</addr>
   <hobby>painting</hobby>
</person>
</doc>
```

```
<!ELEMENT doc (person)*>
<!ELEMENT person (name, addr, (hobby)*,
(child)*)? >
<!ATTLIST person age CDATA  #IMPLIED>
<!ELEMENT child person>
<!ELEMENT name #PCDATA>
<!ELEMENT addr #PCDATA>
<!ELEMENT hobby #PCDATA>
```

Fig. 1. An XML document GTree, its DataGuide and DTD

2.2 XDGL2 Locking Protocol

In this section we present a high level overview of new locking protocol called XDGL2. Except for some refinements, the XDGL2 protocol is based on our XDGL protocol presented in [8].

XDGL2 requires transaction to follow the strict two-phase locking protocol (S2PL)[12]. According to S2PL, a transaction acquired a lock keeps it until the end.

In SXTM we maintain the DataGuide (DG for short) structure for locking purposes rather than the document iself as the document is not physically available. A DG describes every unique label path of a document exaclty once, regardless of the number of times it appears in that document, and encodes no label path that does not appear in the document. Exactly by utilizing DG as a locking structure we abstract from underlying storage scheme used for XML documents in DBMS. Thus, this generalizes the method.

To cope with deadlocks we use deadlock detector which periodically builds wait-for graph[11] of the transactions and analyzes it for cycles. When a cycle is found, a victium is selected and aborted.

There are two kinds of locks in XDGL2: structural locks and logical locks which we describe in the following subsections.

Structural locks

– `P` (pass) lock. This *node lock* is used by `LP` operation. It is set on the DG's nodes which match the `ITM` nodes of `LP`. This lock prevents the deletion of `ITM` nodes by other transactions, but it does not prohibit any insertion of the nodes extending the sequence of `ITM` nodes (see example 1 below).

– S (share) lock. This *node lock* is also used by LP operation. It is set on the DG's nodes which match the DST nodes of LP. This lock prevents *any* modifications of DST nodes. For example, for query count(/doc/person) S lock is set on node n2 (see Fig. 1).
– SI (shared into) lock. This *node lock* is used by II operation. It is set on the DG's nodes which match the target nodes of II. This lock prevents the modification of II's target nodes and insertion of another nodes *into* the target nodes by concurrent transactions. The SA (shared after) and SB (shared before) locks are defined in a similar way.
– XN (exclusive new) lock. This *node lock* is used by insert operations. It is set on the DG's node which matches the newly created nodes. This lock allows passing by the new node, (i.e it compatible with P lock), but other operations (e.g. modidfication) on the node are prohibited.
– X (exclisive) lock. This *node lock* is used by operations which modify internal nodes of the document. This lock prevents any concurrent reads and updates of the node. In our set of update operations this lock is used by RN operation since it renames the target nodes inside the document. It is set on the DG's nodes which correspond to the RN's target and new nodes. Note, if we rename the leaf node then we only need to acquire X lock on the RN's target nodes and XN lock on the new nodes.
– ST (shared tree), XT (exclusive tree) locks. The ST (XT) lock is a *tree* lock. The ST (XT) lock is set on the DG's node and implicitly locks all its descendants. The ST (XT) lock prevents any updates (reads and updates) in the entire subtree. For example, XT lock is set on target nodes of D operation.
– IS (intention shared), IX (intention exclusive) locks. The IS lock must be obtained on each ancestor of the node, which is to be locked in one of the shared modes. It ensures that there are no locks on the coarser granules locking the node in conflicting mode. IX lock is defined in a similar way.

To deal with value-based constraints, each structural lock has an annotated value-based predicate[1]. The XDGL2 compatibility matrix for structural locks is shown in the Fig. 2. There are no strict incompatibilities in matrix. Symbol CP (check predicates) in matrix means that the requested lock is compatible with granted locks only if conjunction of annotated predicates is not satisfiable.

Next we study a couple of examples to illustrate the locking mechanisms[2].

Example 1. Let us consider transactions T1={II(/doc ,<person/>)}, T2={ II(/doc,<person/>)} and T3={/doc/person/name}. According to XDGL2 the following sets of structural locks {n1: (SI, IX), n2: XN}, {n1: (SI, IX), n2: XN} and {n1: P, n2: P, n4: ST} must be obtained by transactions T1, T2 and T3 respectively. Thus, we obtain that T1, T3 and T2, T3 can run concurrently whereas T1 and T2 can not run concurrently due to the conflict of SI locks on node n1.

[1] In XDGL2 we annotate locks only with simple predicates (conjunction of comparisons with constants)
[2] Note, that in example 2 we omit logical locks

	granted										
requested	S	P	SI	SA	SB	XN	X	ST	XT	IS	IX
S	+	+	+	+	+	cp	cp	+	cp	+	+
P	+	+	+	+	+	+	cp	+	cp	+	+
SI	+	+	cp	+	+	cp	cp	+	cp	+	+
SA	+	+	+	cp	+	cp	cp	+	cp	+	+
SB	+	+	+	+	cp	cp	cp	+	cp	+	+
XN	cp	+	cp	cp	cp	+	cp	cp	cp	+	+
X	cp	cp	cp	cp	cp	cp	cp	cp	cp	+	+
ST	+	+	+	+	+	cp	cp	+	cp	+	cp
XT	cp	cp	cp	cp	cp	cp	cp	cp	cp	cp	cp
IS	+	+	+	+	+	+	+	+	cp	+	+
IX	+	+	+	+	+	+	+	cp	cp	+	+

Fig. 2. Lock Compatibility Matrix

Example 2. For transactions `T1=D(/doc/person/@age)}` and `T2={for $v in //person return <name2>{$v/ name}</name2>}` we need to obtain the sets of structural locks `{n1: (IX, P), n2: (IX, P),n3:XT}` and `{n1: IS, n6: IS, n2: S, n8: S, n4: ST, n10: ST}` respectively. Since locks required by `T1` and `T2` are compatible, we conclude that `T1` and `T2` can be executed concurrently (thus we prevent document order conflict).

Note, that DGLOCK from XMLTM does not allow concurrent executon of transactions from examples 1 and 2.

Logical Locks Now we turn to the discussion of the logical locks, which are used to prevent phantoms. Let us show how a phantom could appear. Suppose that transaction `T1` reads all of *age* attributes in *GTree*(i.e. `T1` issued *//@age* query). In the meantime transaction `T2` inserts new *age* attribute into *person* element with *name* 'John' (see Fig. 1). The new *age* attribute is the phantom for transaction `T1`. Generally speaking, phantoms can appear when update operation extends the DG[3] (adds new path to DG) and this modification results in the changing of target nodes of previously executed operations.

Thus, we introduce two locks. The first lock is `L` (logical) lock, which must be set on DG's node to protect the node's subtrees in the document from a phantom appearance. A logical lock specifies a set of *properties*. Essentially, a property is a logical condition on nodes. This lock prohibits the insertion of new nodes, which possess these properties. The second lock is `IN` (insert new node) lock, which specifies the properties of new node. The update operation, which extends the DG, should obtain the `IN` lock on each ancestor of the new node.

Here we list all possible combinations of properties for `L` lock: (1) node-name='name1' (e.g. *//person*), (2) node-name='name1', node-value *relop* 'val1' (e.g. *//name[.≠'John']*), (3) node-name='name1', child-name='name2', child-value *relop* 'val1' (e.g. *//person[name ≠ 'John']*). Here *relop* is a comparison operation.

To check that the new node's properties do not interfere with the `L` lock properties, the `IN` lock should specify three properties of a new node: new-node-parent-name, new-node-name, new-node-value.

Thus, `L` and `IN` locks are incompatible if one of the following conditions holds:

[3] In our set of update operations II, IA, IB and RN operations can extend the DG.

– If `IN`'s new-node-name equals to a node-name of `L` lock and `L` does not contain any other properties (case (1) from the above)[4].
– If `IN`'s new-node-name and new-node-value both match appropriate values of `L` lock consisting of two properties. That is, node-name=new-node-name and new-node-value relop 'val1' $\neq$ #f (case (2) from the above).
– If all `IN`'s properties match three properties of `L` lock. That is node-name=new-node-parent-name, child-name=new-node-name and new-node-value relop 'val1' $\neq$ #f (case (3)).

If node's name is a wildcard '*' then it equals to any node-name.

DTD-based Concurrency Enhances We already discussed the benefits we got by utilizing the DG for locking. However, there are some drawbacks. The problem with DG is that it lacks the notion of document order. Therefore, the evaluation of ordered-based axes (e.g. preceding-sibling, following-sibling) on DG may result in the lock of unnecessary DG nodes. As a consequence, we can get unnecessary conflicts among transactions.

To reduce the number of such conflicts we use DTD, which specifies the document order. For instance, on the basis of the *GTree*'s DTD information `<!ELEMENT person (name, addr, (hobby)*, (child)*)?!>` we can resolve the conflict between transactions `T1={/doc/person/ addr/preceding-sibling::*}` and `T2={D(/doc/person/hobby)}`. Indeed, without DTD information `T1` must acquire `ST` lock on all DG's nodes of level 2 (`n3`, `n4`, `n5`, `n6`, `n7`), whereas `T2` must obtain `XT` lock on node `n7` of level 2. Thus, there is a conflict between `T1` and `T2` on node `n7`. But using DTD we know that preceding sibling of *addr* element can only be *name* element, and `T1` must acquire `ST` lock only on node `n4` thereby allowing `T2` to run concurrently.

Correctness

Theorem 1. *All schedules of concurrent transactions generated by XDGL2 scheduler are serializable.*

Formal proof of this theorem is available in [9].

2.3 Atomicity of XML transactions

Decomposing XML transaction into a number of independent DB transactions, we can no longer implement XML transaction's abort by RDBMS[10]. Only active DB transactions can be rolled back by RDBMS. For this reason, XMLTM logs the before-images of updated XML documents. An abort is implemented by restoring old text value of the document. Since the logging granularity in XMLTM is greater than the locking granularity the cascading aborts are possible.

[4] In case of RN operation we compare only new-node-name property of IN lock with node-name property of L lock. Another properties of IN lock are not considered.

We propose a new way to guarantee atomicity of XML transactions, which allows to reduce the size of log significantly. For clear presentation and because of space limitations we consider only insert and delete operations.

To undo the insert operations we propose to log *reverse* operations, which delete the inserted nodes. As a consequence, an extra logging overhead for insert operations is low. This way for `II(/doc/person, <name/>)` operation we write into the log only `D(/doc/ person/name[position=last()])`. Similarly, for operation `IB(/doc/person/hobby, <addr/>)` we need to log `D(/doc/person/addr::preceding-sibling[position()=1])` operation.

An undo of `D` is somewhat more difficult and expensive - we need to log both the before-images of deleted subtrees and their positions in the document. Thus, before executing `D` operation we need to execute preliminary query, which extracts needed information and then write its results into the log. For instance, for `D(/doc/person[name='John'])` the preliminary query is `for $node at $pos in /doc/person where $node/name = 'John' return ($node, $pos)`. Here `$node` variable is bound with the deleted subtree, whereas `$pos` defines the position of the subtree. To undo this `D` operation we will execute the following query for each `$node`: `IA(/doc/node()[postion()=$pos-1], $node)`.

Note that no extra logging is needed for the *last* DB transaction since its commit means the end of XML transaction. We use this fact to reduce a significant logging overhead incured by `D` operation. We propose to execute `D` operation in the following way (1) if an operation `op` in XML transaction preceded by `D` operation depends on the results of `D` than execute `D` before `op`'s execution, (2) else execute `D` operation in *last* DB transaction [5].

3 Concurrency Control for Native XML DBMSs: Related Work

The protocol developed in [2] relies on the fact that XML is usually accessed by means of XPath query language. They propose to use "path locks" to increase concurrency and provide the best results for XPath queries. Unfortunately, the authors deal with too restrictive subset of XPath.

The method presented in [7], proposes a XLP (XPath Locking Protocol) locking protocol, which is based on XPath model and has the features of richer lock modes, refined lock granualrity and lower lock conflict. Another XPath-based protocol was proposed by Choi et al.[1]. The protocol assumes that an XML document is modeled as a tree structure. Three versions of XML document are maintained. Concurrency Control is implemented by using read-write check and write-read check. Unfortunately, these checks are very expensive operations. The protocol was limited to theoretical design.

Several protocols [5], [6] based on the DOM operations were proposed. The protocols use different kinds of locks to latch nodes on different levels. These seem to work fine for DOM operations, but there is no research done to see whether they could suite well the evaluation of path expressions.

[5] Actually, we execute last DB transaction on the users COMMIT command.

Moreover, all these methods require direct access to the XML document, which is not possible in mant cases.

4 Conclusion and Future Work

This thesis is centered around the transaction management for XML stored in RDBMSs. The main goal of our research is efficient and general transaction manager for concurrent XML processing in RDBMS.

We start with a detailed analysis of existing solutions depicting those assumptions, techniques, limitations and drawbacks. In an attempt to eliminate limitations and drawbacks the thesis presents a new XDGL2 locking protocol and a new way for logging high level XML-operations to guarantee atomicity of XML transactions.

Utilizing DataGuide as a locking structure, we abstract from underlying storage scheme used for XML documents in RDBMS. Different types of locks on DataGuide structure are proposed to capture the semantics of XML operations thereby allowing to improve the degree of concurrency in comparison with existing solutions. Unlike previously proposed solutions, our protocol ensures strict serilizability and combines predicate and logical locks to provide protection from phantoms appearance.

The paper presents an approach how to reduce the overhead incured by extra logging. We use a combination of logical logging and delayed execution of delete operations, which allows to undo delete operations using RDBMS's recovery functionality.

Finally, the proposed Semantic XML Transaction Manager is primarily being implemented on top of MS SQL Server 2005 and in our next steps, we concentrate on its comprehensive experimental study.

References

1. E. H. Choi and T. Kanai. XPath-based Concurrency Control for XML Data. In Proc. DEWS 2003, 2003.
2. S. Dekeyser and J. Hidders. Path Locks for XML Document Collaboration. In Proc. WISE 2002, Singapore, 2002.
3. T. Grabs, K. Bohm, and H.-J. Schek. XMLTM: Efficient Transaction Management for XML Documents. In Proc. ACM CIKM 2002, USA, 2002.
4. J. Gray et al. Granularity of locks in a large shared databases. In Proc. VLDB 1975, USA, 1975.
5. M. Haustein and T. Haerder. Adjustable Transaction Isolation in XML Database Management Systems. In Proc. XSym 2004, LNCS 3186, 2004.
6. S. Helmer, C.-C. Kanne, and G. Moerkotte. Evaluating lock-based protocols for cooperation on XML documents. SIGMOD Record, 33(1), 2004.
7. K.-f. Jea et al. Concurrency Control in XML Document Databases: XPath Locking Protocol. In Proc. ICPADS 2002, Taiwan, 2002.
8. P. Pleshachkov, P. Chardin, S. Kuznetsov. XDGL: XPath-Based Concurrency Control Protocol for XML Data. Proc. BNCOD 2005, LNCS 3567, Sunderland, UK.

9. P. Pleshachkov, P. Chardin, and S. Kuznetsov. A DataGuide-Based Concurrency Control Protocol for Cooperation on XML Data. In Proc. ADBIS 2005, LNCS 3631, 2005.
10. G. Weikum. Principles and Realization Strategies of Multilevel Transaction Management. ACM Transactions on Database Systems (TODS), 16(1), 1991.
11. J. Gray and A. Reuter. Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers Inc., 1992.
12. K. P. Eswaran et al The notions of consistency and predicate locks in a database systems. Comm of ACM, Vol. 19, No 11, pp. 624-633, November 1976.