A Locking Protocol for Scheduling Transactions on XML Data*

© Peter Pleshachkov
Institute for System Programming RAS
peter@ispras.ru

Petr Chardin
Moscow State University
pchardin@acm.org

Abstract

In this paper we propose a new DataGuide-based locking protocol for isolation of concurrent XML transactions. The protocol adopts *DataGuide* structure for locking purposes. We extend the multigranularity locking protocol by adding node and logical locks on DataGuide. This allows to enhance concurrency of XML-specific transactions, thereby increasing the overall system performance.

1 Introduction

The widespread use of eXtensible Markup Language (XML) [2] in scientific data repositories, digital libraries and across the Web prompted the development of a method for efficient synchronizing of concurrent updates and queries for XML-data. As XML continues to gain popularity and the amount of XML-data constantly grows, the proper scheduling of concurrent queries and updates becomes an important issue.

When several transactions have access to the same document at the same time, we need to protect each of them against the others. To simplify this task, we usually want to serialize transactions. Serializability [10] requires the result of transactions processing to be equal to the one produced by some serial execution of the same transaction set.

Most systems ensure serializability by controlling access to each data item according to some particular concurrency control scheme. Locking-based protocols are used most widely. In these methods a transaction must lock a data item before first access to the item, and unlock it when transaction is done with it. Locking protocols usually use two types of locks: exclusive locks and shared locks. Exclusive locks prevent any other lock to be held on a data item, while shared locks permit to acquire more shared locks on the same item. A locking protocol describes rules, according to which a transaction should lock and unlock the data pieces.

Eswaran et al.[6], introduced the locking protocol, which is most popular by now, – the two phase locking protocol (2PL). This protocol specifies that no data item can be unlocked until all data items to be accessed have been locked. Eswaran et al. have also demonstrated that the two phase criterion is necessary and sufficient to ensure serializability.

Another kind of protocols is tree-based protocols [14]. They are used in databases organized (logically or physically) as trees. However, the tree protocol is not adequate for XPath-like queries as the one only supports a top-down access to the document. And XPath [3] language supports queries with more complex navigational behaviour (via child, descendant, attribute, parent, preceding-sibling and following-sibling axes). Thus, the tree protocol does not support significant subset of XPath.

In a short paper [13] we presented a novel locking method for XPath operations - XDGL. The paper presented very first results and incomplete in many extents.

In this paper we investigate a more wide subset of XPath language including *preceding-sibling* and *following-sibling* axes, *wildcards* and *predicates* in location steps. Moreover, we examine a more complex update operations such as *move*, *replace* and *rename.*

In XML-enabled DBMSs, XML data usually is not represented as a tree structure physically. This is why XDGL employs the DataGuide structure for locking. Because of this, our approach can be easily implemented on top of any system, which stores XML. It could be a Native XML DBMS, a relational or object-oriented DBMS.

Another advantage of our approach is that DataGuide is usually much smaller than the document itself. Hence it may be held in main memory even for large XML documents. This way, the locking overhead is small.

In our locking method we employ two kinds of locks: *tree locks* and *node locks*. The tree locks are useful for protecting the whole subtrees addressed by XPath location paths. The node locks are useful for protecting single DataGuide's nodes (note, that it may match many

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

Proceedings of the Spring Young Researcher's Colloquium on Database and Information Systems SYRCoDIS, St.-Petersburg, Russia, 2005

nodes in document). For example, node locks are used by insert operations.
We introduce special *logical locks* and *insert new node locks* to avoid phantoms appearance. The logical lock could be set on the DataGuide's node and specifies a node's name which should be protected. I.e. the node with such name cannot be inserted under this node. A transaction which inserts a new node must obtain *insert new node lock* on each ancestor of the new node. This is needed as it may be a phantom for another transaction
The remainder of the paper is organized as follows. Section 2 presents the XML query and update languages, which are of interest in this paper. In Section 3 we define DataGuide structure and present the example of DataGuide for XML document. Section 4 is devoted to proposed locking protocol. It contains a number of small examples, which help to understand the protocol and its benefits. In Section 5 we give a brief overview of related work. Section 6 contains a summary of our work.

```
<file_system>
<catalog name = 'home'>
  <date>10 March, 2003</date>
  <access>777</access>
  <file>ls.cpp</file>
  <file>ls.h</file>
  <catalog name ='socol'>
     <date>7 April, 2004</date>
     <access>754</access>
  </catalog>
  <directory name='barracuda'>
       <date>9 April, 2005</date>
       <access>750</access>
  </directory>
</catalog>
<catalog name='system'>
  <date>1 January 2003</date>
  <access>743</access>
  <file>passwords</file>
</catalog>
</file_system>
```

Figure 1: an XML document *FS*

2 Data Manipulation Language

This section gives an overview of query and update languages. We also consider several examples of queries and updates of the sample XML document *FS* depicted in Figure 1. The document conforms to the DTD shown in Figure 3.

2.1 Query language

We use XPath language to retrieve nodes from the XML documents. XPath defines the syntax and semantics for location paths. Each location path consists of a sequence of location steps separated by '/'. In turn, location step consists of *axis*, *node-test* and optional *predicate*. Syntactically it looks like *axis::node-test[predicate].* XPath defines the following semantics for evaluation of location step. A location step starts with a set of context nodes (defined as result of the previous location step). An axis specifies the direction of movement from the context nodes, node-test specifies the type of the nodes to be selected and predicate filters the selected nodes. The result of location path is result of last location step
In this paper we investigate only a subset of XPath. The following axes are of interest: *child*, *descendant*, *attribute*, *following-sibling* and *preceding-sibling*. Besides we take into consideration only simple predicates (comparison of node's value with constant).
Finally, let us consider a couple of queries for *FS* document. The query */file_system//catalog* retrieves all *catalog* elements in *FS*. The location path */file_system/catalog[@name='system']* consists of two location steps. It addresses *system catalog* element located under *file_system* element. Note, that *@name='system'* is a simple predicate.

2.2 Update language

To change the document one should use update operators. We examine five types of update operators: insert, delete, move, replace and rename operators.

- Insert operator:
  INSERT constructor (INTO | BEFORE | AFTER) path-expr
- Delete operator:
  DELETE path-expr
- Move operator:
  MOVE path-expr1 (INTO | BEFORE | AFTER) path-expr2
- Replace operator:
  REPLACE path-expr WITH constructor
- Rename operator:
  RENAME path-expr AS NCNAME

Here *constructor* is an element or attribute constructor. We specify an element constructor as *element {elem-name} {content};* meaning of the *elem-name* and *content* is straightforward. There are complex element constructors. In such constructor *content* itself is the nested element constructor. In a simple case *content* could be just a text.
One can specify the attribute constructor as *attribute {name} {text}*; Here *name* and *text* specifies the name and the value of the attribute.
We introduce three types of insert operators: insert-into, insert-before and insert-after. These operators insert new node defined by *constructor* as the last child, previous sibling and next sibling for each node selected by *path-expr* respectively. If *constructor* specifies an attribute constructor, then we could only use insert-into operator that adds new attribute to each node selected by *path*-expr. It also means that each of the selected nodes should be of element type.
Delete operator removes subtrees of all nodes specified by *path-expr* from the document. That is to say, our delete operator uses the deep deletion semantics.

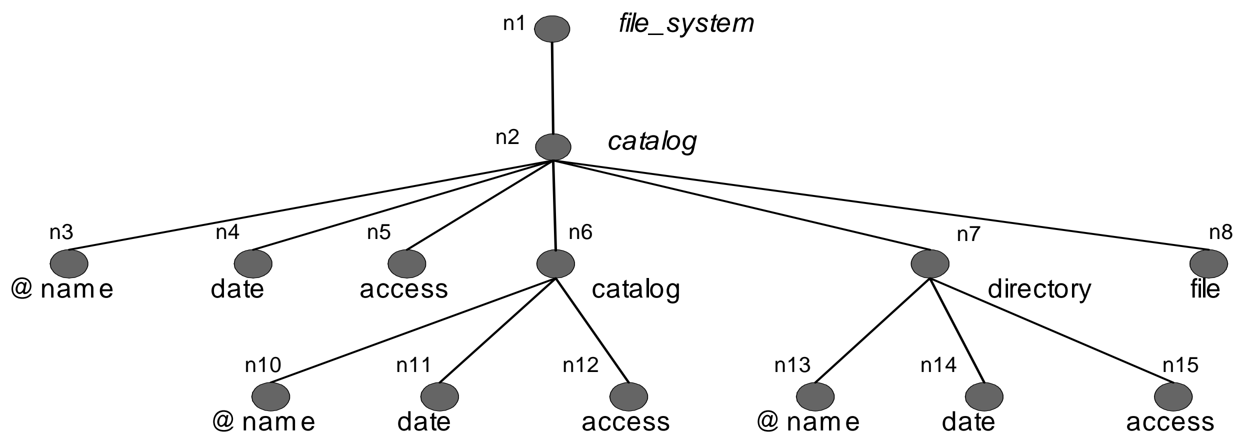

Figure 2: DataGuide of the document *FS*

Move operator transfers subtrees from the place specified by *path-expr1* to the place specified *by path-expr2*. The semantics of *INTO*, *AFTER* and *BEFORE* keywords is the same as in insert operator.

Replace operator substitutes node defined by *constructor* for the subtrees specified by *path-expr.*

Rename operator gives a new name (*NCNAME*) for the nodes specified by *path-expr*.

Let us consider a couple of examples. The following update operator adds new *file* element to system catalog: *INSERT element{file}{'hosts'} INTO /file_system/catalog[@name='system'].* The operator *RENAME /file_system//catalog AS directory* gives a *directory* name for all *catalog* elements in *FS* document.

3 DataGuide

DataGuide [7] is a data structure that summarizes an XML document. It is concise and accurate because DataGuide describes every unique label path of a document exactly once, regardless of the number of times it appears in that document, and encodes no label path that does not appear in that document. Figure 2 shows the DataGuide of *FS* document.

```
<!ELEMENT file_system (catalog | directory )*>
<!ELEMENT catalog (((date, access)| (file)*),
(catalog)*)>
<!ATTLIST catalog name CDATA #REQUIRED>
<!ELEMENT directory catalog>
<!ELEMETN date #PCDATA>
<!ELEMENT access #PCDATA>
<!ELEMENT file #PCDATA>
```

Figure 3: DTD of the document *FS*

4 Locking Protocol on DataGuide

A transaction must lock a data item before the first access to that data item, and unlock it when all accesses to the item are complete. Our protocol requires transaction to follow strict two-phase locking protocol (S2PL) [1]. According to S2PL a transaction, acquired a lock, keeps it until the end.

While traversing or modifying an XML document, a transaction has to acquire a lock in an acceptable mode for each node before accessing it. Since the nodes in an XML document are organized as a tree-like structure, the principles of multigranularity locking scheme (MLS) [9] may be exploited. The MLS introduces intention locks which prevent a subtree *t* from beeing locked in a mode incompatible to locks already granted to *t* or subtrees of *t.* However, the straightforward adoption of MLS for synchronization concurrent XML transactions may result in a low concurrency. For instance, one transaction might need to lock some intermediate node *n* of an XML document in a read mode, while another transaction may wish to perform an update of some node in the subtree of *n*. However, MLS's share and exclusive locks implicitly lock the entire subtree which is too restrictive. Example1 studies some drawbacks of multigranularity locking scheme adopted for XML.

Example 1: Let us suppose that transaction *T1* has issued the XPath query */file_system/catalog/access*. It should be possible for transaction *T2* to insert an empty element *<catalog/>* as a child of *file_system* element. According to MLS, the entire DataGuide's *access* subtree is to be locked in the shared mode by *T1*. At the same time, catalog subtree has to be locked in the exclusive mode by T2. And since *catalog* subtree includes *access* subtree (see Figure 2) and shared lock held by T1 is not compatible with exclusive lock held by T2. Therefore, *T1* and *T2* cannot be executed concurrently.

In fact, transactions *T1* and *T2* do not conflict. They would conflict if *T2* inserted *<catalog><access>777</access></catalog>* element inside *file_system* element.

In this paper we introduce granular locking protocol on DataGuide. To cope with hierarchical nature of XML documents we use IX and IS intention locks. Besides, we introduce node and tree locks for locking the DataGuide's node and DataGuide's subtree

respectively. Moreover, we use node locks to prevent document order conflicts during execution of concurrent insert operations. For instance, the document order conflict arises if one transaction inserts new node as the last child into a node, while another transaction also inserts new node as the last child into the same node.

To cope with phantoms appearance we use logical locks. They allow to lock node's name in the DataGuide's subtree. These locks are useful for such queries as *//file*. According to the DTD of document *FS*, *file* element could appear at any level in *FS* document. Therefore, *FS*'s DataGuide could potentially contain arbitrary number of the *file* nodes. The logical lock on the *file* name set on DataGuide's root denies other transactions from inserting of any element with the name *file*.

4.1 Node and Tree Locks

Below we describe a set of all node and tree locks, employed by our method.

- *SI (shared into), SA(shared after)* and *SB(shared before)* locks. These node shared locks are used by insert operations. It is set on the DataGuide's nodes defined by *path-expr* of insert operator. This lock prevents any modifications of the node and insertion of another nodes *into* the node by concurrent transactions. *SA* and *SB* locks are defined in a similar way.
- *X (exclusive)* lock. This node lock must be obtained for the node to be modified. Note, that the nested nodes of the locked node may be read by another transactions.
- *ST (shared tree)* lock. This tree lock sets on a DataGuide's subtree to protect the whole subtree from any updates. XPath queries require this kind of locks. Due to the semantics of XPath the results of the location path are the subtrees selected by the last location step. It implies the request of the *ST* lock for subtrees retrieved by location path.
- *XT (exclusive tree)* lock. This tree lock sets on a DataGuide's subtree to protect the subtree from reading and modifications.
- *IS (intention share)* lock. According to the granular locking protocol we have to obtain these lock on each ancestor of the node which is to be locked in a shared mode.
- *IX (intention exclusive)* lock. According to the granular locking protocol we have to obtain these locks on each ancestor of the node which is to be locked in an exclusive mode.

Figure 4 shows compatibility matrix for the lock modes defined above. A compatibility matrix indicates whether a lock of mode M1 may be granted to a transaction, while a lock of mode M2 is presently held by another transaction.

Note, that *IX* and *X* locks are compatible since *IX* lock on a node only implies the intention to lock the descendants of the node. But it does not imply the lock on the node itself. *SI* (*SA*, *SB*) lock is not compatible with *SI* (*SA*, *SB*) lock, which prevents concurrent insert-into (insert-after, insert-before) operations upon the same node.

	granted							
requested	SI	SA	SB	X	ST	XT	IS	IX
SI	-	+	+	-	+	-	+	+
SA	+	-	+	-	+	-	+	+
SB	+	+	-	-	+	-	+	+
X	-	-	-	-	-	-	+	+
ST	+	+	+	-	+	-	+	-
XT	-	-	-	-	-	-	-	-
IS	+	+	+	+	+	-	+	+
IX	+	+	+	+	-	-	+	+

Figure 4: Lock compatibility matrix

4.2 Predicates

To cope with the value-based constraints on the node's content (extracted from location path) each node lock and tree lock are annotated with *predicate*. In this case, the lock compatibility matrix does not contain strict incompatibilities. Two locks are compatible if one of the following condition hold: (1) the mode of the one lock is compatible with the mode of another lock due to the lock compatibility matrix, (2) the predicates of these locks do not conflict (i.e. the conjunction of predicates is not satisfiable). Thus, taking into consideration the predicates on the node's value allows to reduce the number of conflicts between transactions significantly thereby increasing concurrency.

We will regard that location step without predicate has the true predicate. Besides, *IS* and *IX* locks are always annotated with true predicate.

4.3 Logical Locks

Now we turn to the discussion of logical locks which are used to prevent phantoms appearance. For example the transaction which issued *//file* query may suffer from phantoms since another transaction may insert new *file* element at some deep level of the document *FS* (see *FS*'s DTD). Generally speaking, phantoms can appear when (a) insert operation extends the DataGuide (adds new path to DataGuide), (b) the insertion of a new node changes a target node of some operation performed by another transaction.

Thus, we introduce two locks. The first lock is logical (*L*) lock, which must be set on DataGuide's node to protect subtrees from a phantom appearance. A logical lock specifies a set of properties. *L* lock prohibits the insertion of new nodes which possesses these properties. The second lock is insert new node (*IN*) lock, which specifies the properties of new node. The *L* and *IN* locks are compatible if the properties of *L* lock differs from the properties of inserted node. The properties may includes the node name, node value, child name and child value. For example the transaction which issued *//file[.='ls.cpp']* query must obtain L lock

on the DataGuide's root with properties: *node-name='file'*, *node-value='ls.cpp'*.

4.4 Locking Rules

In this subsection we describe a list of locking rules. These rules define which locks must be obtained for which operations. Each operation contains at least one *path-expr* which defines the operation's target nodes. Then the operation is applied to the target nodes.
Let *DP* be a data path set of all label paths in DataGuide that lead to data queried or updated. Then we may compute a set *NP* of all nodes in DataGuide (and associate predicates with them) which match any label path from *DP*. Moreover, let *PH* be a set of pairs *(n, properties)*;here *n* defines the DataGuide's node where a phantom could appear, *properties* specifies the conditions on the nodes to be logically locked.

- Rule for XPath query. For each node from *NP* performs (1) obtain *(ST, p)* lock (*p* is the associated predicate) on the node, (2) obtain *IS* lock (with true predicate) for each ancestor of the node and all nodes traversed via preceding-sibling and following-sibling axes. We denote such nodes as ANCSIBL nodes.
- Rule for insert-into operator. For each node from *NP* performs (1) if the node matches target nodes of insert operator then obtain *(SI, p)* lock on the node and *IS* lock on its ANCSIBL nodes, (2) if the node matches additional branches of *path-expr* then obtain *(ST, p)* lock on the node and *IS* lock on its ANCSIBL nodes, (3) if the node matches the new inserted node then obtain *(X, p)* lock on the node and *IX* lock on its ANCSIBL nodes. The rules for insert-after and insert-before operators are analogous.
- Rule for delete operator. For each node from *NP* performs (1) if the node matches the target nodes of delete operator then obtain *(XT, p)* lock on the node and *IX* lock on its ANCSIBL nodes, (2) if the node matches the additional branches of *path-expr* then obtain *(ST, p)* lock on the node and *IS* lock on its ANCSIBL nodes.
- Rule for rename operator. For each node from *NP* performs (1) if the node matches the target nodes or the new subtree of rename operator then obtain *(XT, p)* lock on the node and *IX* lock on its ANCSIBL nodes, (2) if the node matches the additional branches of *path-expr* then obtain *(ST, p)* lock on the node and *IS* lock on its ANCSIBL nodes.
- Rule for move-into operator. For each node from *NP* performs (1) if the node matches the target nodes of *path-expr2* (see definition of move operator) then obtain *(SI, p)* lock on the node and *IS* lock on its ANCSIBL nodes, (2) if the node matches the target nodes of *path-expr1* (i.e. subtree to be deleted) or the new subtree (i.e. subtree to be inserted) then obtain *(XT, p)* lock on the node and *IX* lock on its ANCSIBL nodes,(3)if the node matches the additional branches of *path-expr1* or *path-expr2* then obtain *(ST, p)* lock on the node and *IS* lock on its ANCSIBL nodes. The rules for move-after and move-before operators are analogous.
- Rule for replace operator. For each node from *NP* performs (1) if the node matches the target nodes of *path-expr* then obtain *(XT, p)* lock on the node and *IX* lock on its ANCSIBL nodes, (2) if the node matches the additional branches of *path-expr* then obtain *(ST, p)* lock on the node and *IS* lock on its ANCSIBL nodes, (3) if the node matches the new inserted node (defined by constructor) then obtain *(X, p)* lock on the node and *IX* lock on its ANCSIBL nodes.
- Rule for phantoms prevention. For each node from *PH* the *(L, properties)* lock must be obtained. Besides each operation which extends the DataGuide must obtain *(IN, properties)* lock on ANCSIBL nodes of new node.

4.5 Examples

Now let us consider several examples to illustrate the locking rules. Let us return to example1. Now we will show that both transactions from the example can proceed with proposed locking protocol. According to it, transaction *T1* must obtain *IS* lock on nodes *n1*, *n2* and *(ST,#t)* lock on node *n5*; here *#t* is the true predicate. At the same time *T2* must obtain *IX* lock on *n1* and *(X,#t)* lock on *n2*. As all locks are compatible transactions *T1* and *T2* could be executed concurrently. This is illustrated in Figure 5.

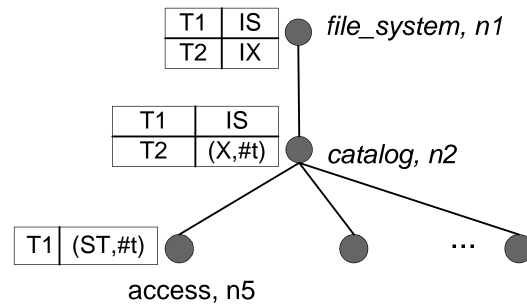

Figure 5: Locking rules for example1

Example 2 (conflict of two insert operations):
Let us suppose that transaction T1 inserts new *access* element: *INSERT <access/> INTO /file_system/catalog[name='home']/date/following-sibling::catalog,* while transaction T2 inserts new *date* element: *INSERT <date/> INTO /file_system/catalog/catalog.*
Figure 6 shows that transactions *T1* and *T2* cannot run concurrently since *SI* lock is not compatible with itself.
Example 3 (phantoms prevention):
Let us suppose that transaction *T1* retrieves all *file* elements found at any level inside *catalog* elements which can be found themselves inside *file_system* element. In XPath such a query looks like this: */file_system/catalog//file.* At the same time transaction

T2 inserts new *file* element into the *catalog* element by the following statement:
INSERT element{catalog}{} INTO /file_system/catalog/catalog

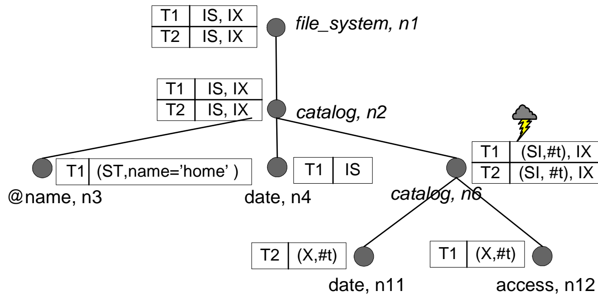

Figure 6: Incompatibility of insert operations

It is easy to see that the second transaction might add a phantom node for the first one. However, our locking rules prevent this situation. This is shown in the Figure 7. *(L, file)* lock is not compatible with *(IN, file)* lock. Thus, the insertion of the *file* element is denied.

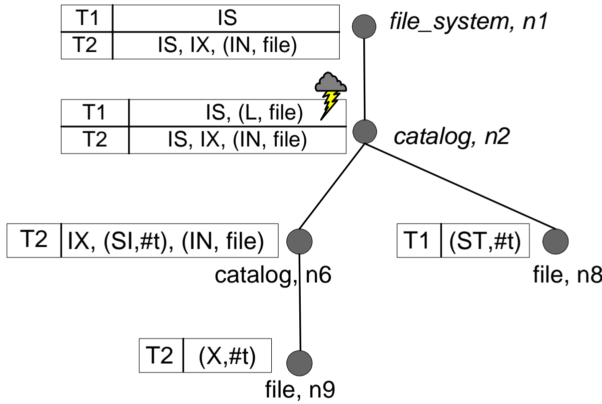

Figure 7: Logical locks

4.6 Unordered XML Documents

For some applications document order is not important. Our locking method could be easily modified to deal with unordered XML documents. In this case we do not need *SI*, *SA* and *SB* locks. Instead, conventional *S* lock is needed. It is a common shared lock. By definition *S* lock is compatible with itself. I.e. two insert operations, adding elements with different names into the same element do not conflict.

5 Related Work

There were proposed several locking schemes for synchronizing concurrent XML operations. Here is a brief overview of these methods.

Grabs et. al. [8] proposed a DGLOCK protocol, which is a combination of well-known granular and predicate locking on the DataGuide. This work has much in common with our one. But DGLOCK has several disadvantages: (1) as a consequence of granular locking we have a conflict in the example 1, (2) DGLOCK does not guarantee serializability and has no phantom prevention mechanism, (3) the query language does not support the *descendant* axis, which is very important for querying semistructured data.

In [11], the synchronization of concurrent transactions is considered in the context of DOM API. The authors present three types of locks: node locks, navigational locks and logical locks. Node and navigational locks are acquired for context nodes and virtual navigation edges respectively. In turn, logical locks are introduced to prevent phantoms. Authors offer variety options to enhance transaction concurrency. But synchronization of other APIs (e.g. XPath) is part of the future work.

There are a number of isolation protocols for the DOM API proposed in the work [12]. Unfortunately, these locking protocols were developed for DOM API only, and it is not clear whether they could do for XPath expressions.

Dekeyser et al. [4, 5] proposed the fine-grained (node-level) XPath-based locking protocol, which ensures serializability. But this method does not use the DataGuide. Instead all the locks are obtained on the document itself. Disadvantages of this approach have been already noted in this paper.

6 Conclusions

In this paper we have presented a new locking protocol for concurrent processing of XML data. Our method is based on the XDGL protocol proposed in our earlier work [13]. The method is not limited to native XML DBMS. It could be implemented on top of existing relational or object-oriented database system. Another important benefit of our protocol is the size of the locking structures. Unlike in most other locking protocols, the size of the XML document does not affect the number of locks needed for consistent execution of transaction directly. This happens because of the DataGuide structure properties. When a new node is added to a big document, the DataGuide usually does not change. The explanation of this feature is that the DataGuide provides only information for the kinds of paths. And usually, the set of different paths is rather small, since we insert nodes of the same type. In our method DataGuide is used for locking. As a consequence, the lock manager works with a relatively small structure, which is very likely to fit into the main memory even for the huge documents.

References

[1] P. Bernstein, V. Hadzilacos and N. Goodman, "Concurrency Control and Recovery in Database System", Addison-Wesley, 1987.

[2] N. Bray, J. Paoli. Extensible markup language (XML) 1.0 (second edition). W3C Recommendation, October 2000.

[3] J. Clark, S. DeRose, "XML path language (XPath) version 1.0", World Wide Web Consortium (W3C) Recommendation, Nov. 1999.

[4] S. Dekeyser, J. Hidders "Path Locks for XML Document Collaboration", In Proceedings of the Third WISE Conference, 2002.

[5] S. Dekeyser, J. Hidders, "A Commit Scheduler for XML Databases", In proceedings of the fifth Asia Pacific Web Conference, Xina, China, 2003.

[6] K. P. Eswaran, J. Gray, R. Lorie and I. Traiger, "The notions of consistency and predicate locks in a database systems", Comm of ACM, Vol. 19, No 11, pp. 624-633, November 1976.

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

[8] T. Grabs, K. Bohm and H. Schek, "XMLTM: efficient transaction management for XML documents", In Proceedings of the ACM International Conference on Information and Knowledge Management, McLean, Virginia, pp. 142-152.

[9] J. Gray, R. Lorie, "Granularity of locks in a large shared databases", Proceedings of the International Conference on Very Large Databases, 1975.

[10] J. Gray and A. Reuter, "Transaction processing: concepts and techniques", Morgan Kaufmann, 1993.

[11] M. P. Haustein, and Theo Haerder, "taDOM: a Tailored Synchronization Concept with Tunable Lock Granularity for the DOM API", In Proceedings of ADBIS Conference, LNCS 2798, Springer, 2003.

[12] S. Helmer, C.-C Kanne and G. Moerkotte, "Evaluating lock-based protocols for cooperation on XML documents", SIGMOD Record 33(1): 58-63, 2004.

[13] P. Pleshachkov, P. Chardin, S. Kuznetsov, "XDGL: XPath-Based Concurrency Control Protocol for XML Data", To appear in Proceedings of 22nd British National Conference on DataBases.

[14] A. Silberschatz and Z. Kedem, "Consistency in hierarchical database systems", Journal of the ACM, 27(1), pp. 72-80, 1980.