Path Locks for XML Document Collaboration

Stijn Dekeyser and Jan Hidders
University of Antwerp (UIA)
Dept. Math and Computer Science
Universiteitsplein 1, B-2610 Antwerp, Belgium
{dekeyser, hidders}@uia.ua.ac.be

Abstract

The hierarchical and semistructured nature of XML data can cause complicated types of update-behavior. The updates are not limited to entire document trees, but can involve subtrees and even individual elements. These document parts correspond to e.g. sections in text documents or sub-diagrams in vector graphics files. Providing suitable locking mechanisms for semi-structured data can significantly improve collaboration systems that store their data as XML documents.
In this paper we show that concurrency control mechanisms in CVS, relational, and object oriented database systems are inadequate for collaboration systems based on semi-structured data. We therefore propose a new locking scheme of fine granularity based on path locks. *We also show that our proposed mechanism avoids conflicts by ensuring serializability, that it supports both top-down and bottom-up query evaluation, and that it is relatively efficient.*

1. Introduction

Largely due to its semi-structured nature, XML [4] is rapidly becoming the language of choice to publish data on the web. Apart from being of importance to the commercial world, XML has also attracted considerable interest from researchers. Most recent fundamental research has focussed on for example Tree Automata [21], while more applied research has dealt with XML query languages such as XQuery [3]. Possibly because the existence of the DOM [24] interface which enables the user to change XML data in a procedural way, relatively little research has been done regarding update languages for XML. Currently, Lexus [17] and XUpdate [18] are two proposals for such a language.

The research mentioned so far has been focussed on single-user environments, where documents are created, stored and altered by one user. Thus, if one wants to collaborate on an XML document, an existing collaboration system such as CVS [5] has to be used. These systems, however, have a very coarse granularity and do not make use of locking schemes — multiple users can check out and make changes only upon entire documents simultaneously. The CVS system will then attempt to resolve conflicts automatically, and if it fails to do so, solicit help from the user.

Relational database systems have long since dealt with update conflicts in a satisfactory manner. Consequently, one way to offer concurrent update access to an XML document would be to store the document in relational database tables, and then use the existing locking scheme from the RDBMS. Unfortunately, as we will show in Section 2, this method often causes locks that are too restrictive.
Several existing XML-enabled relational database systems, such as Oracle 9i and Microsoft SQL Server, do use either variants of the traditional relational locking mechanisms, or provide conflict detection based on optimistic concurrency control.

Object-Oriented databases [15] also offer concurrency control. A significant portion of research [2, 19], however, deals with supporting long-lived transactions, thereby relaxing the serializability requirement. Of more interest to our work is the research that looks at locking hierarchical objects [6, 15]. In such systems, locking can be based on a hierarchy of collections, extents, objects etc. Protocols similar to those of relational databases extended with *intentional locks* are used. The remarks made in Section 2, are therefore also valid for OODBMSs.

Finally, the *Lore* database management system for semi-structured data [1, 20] did not contain support for concurrency control that utilizes semantical information, although this was mentioned in “future work”. Instead, it used page-based strict two phase locking. The authors of Lore pointed out that “the semi-structured nature [would] require us to rethink some aspects of traditional solutions” to concurrency

control.

In conclusion to this introduction, the semi-structured and hierarchical nature of XML documents invites more sophisticated collaboration schemes than CVS, RDBMSs and OODBMSs can offer. In this paper, we use locking theory from relational databases as a basis to expand upon, introducing *path locks* as a relatively fine-grained locking scheme for XML. This scheme is relatively straightforward to implement; its efficiency is shown to be acceptable.

Practical Use It is clear that a native XML database must offer a locking scheme if it is to accept multiple simultaneous transactions. A special case of such a native XML database could be an extension to existing webservers such as Apache. The extension would enable the webserver to offer XML-files to clients connecting over the Web much as a relational database server would offer a network user a view over its tables. Several clients could then collaborate on the same XML document which itself can be a representation of a word processing file, or a large diagram created by a vector graphics program.

Organization The paper is organized as follows. Section 2 compares existing concurrency control mechanisms from relational and object-oriented databases to our approach. Section 3 will present the query language that transactions use to read (part of) the XML document. It will also present a new update language that is similar to Lexus. Section 4 introduces the locking scheme that offers a fine granularity and efficient algorithms both for requesting and placing locks. It is divided in two subsections, reflecting two different but equivalent locking methods.
Section 5 gives a proof that the locking scheme ensures serialization of schedules of transactions (*sufficiency*). Also, it shows that the conflict rules given for unordered trees are not too restrictive (*necessity*). Section 6 illustrates how references are treated in this model and how they can be used for efficient bottom-up query evaluation. Finally, Section 7 discusses future research opportunities, while Section 8 briefly lists our conclusions.

2. Comparison to Related Work

In the introduction we have described how current document collaboration systems such as CVS do not offer a locking scheme of adequate granularity. In this section, we show that storing XML data in a relational database and using the RDBMS's concurrency mechanisms is also insufficient.

Semi-structured data can be stored in traditional relational databases in many different ways [10, 12]. For all these different representations, the locking mechanisms of RDBMSs may cause locks that are too restrictive. Central in our proof of this is the fact that the parent-child relationship is typically modelled within one relation. Though this relationship can indeed be split up in several tables, in general one table will — because of XML's semi- structured nature (elements can exist at any nesting depth) — contain arbitrarily many tuples that model the parent-child relationship. For our discussion it is therefore appropriate to abstract this into one relation.

We will examine the following locking mechanisms utilized by relational database systems. First, we start with the simplest case where entire tables are locked to prevent *phantoms* [8] from occuring. Secondly, as an improvement to the first system, we investigate predicate locks. To conclude the comparison with RDBMSs, we look at intention locks and tree locking.

2.1. Table locking

In this approach the entire table representing the hierarchy will be locked in case of an update. This is to prevent phantoms from occuring; since this paper introduces a locking scheme that is shown to support serializability, phantoms are not permitted. Such a lock on the entire parent-child relationship table makes it impossible, for example, to add an element in a subtree that has not been read by any user.

```
<document id="0">
<person id="1", age="55">
   <name>Peter</name>
   <addr>Parklane 7</addr>}
   <child>
      <person id="3", age="22">
         <name>John</name>
         <addr>Unistreet 1</addr>
         <hobby>swimming</hobby>
         <hobby>cycling</hobby>
      </person>
   </child>
   <child>
      <person id="4", age="7">
         <name>David</name>
         <addr>Parklane 7</addr>
      </person>
   </child>
</person>
<person id="2", age="43">
   <name>Mary</name>
   <addr>Parklane 7</addr>
   <hobby>painting</hobby>
</person>
</document>
```

Figure 1. A fragment of an XML document D.

Example 1 *Consider that a user U has accessed document D of Figure 1 and has "seen" elements* `<hobby>` *appearing at any level under elements* `<child>` *which themselves can be found at any depth in the XML tree representing D. Stated differently, using the well-known XPath [7] syntax, user U has issued the query* `//child//hobby`*. As the answer to this query, user U has received all node-identifiers of elements* `<hobby>` *that appear under some element* `<child>`*. In this case, these hobbies correspond to hobbies* swimming *and* cycling*, but not to* painting *since that* `hobby` *does not appear under a* `child` *element.*
Using the Edge Table *approach proposed by [12] to convert semi-structured data to the relational model, we get the tuples shown in Table 1.*

source	ordinal	name	target
1	1	age	55
1	5	child	**3**
1	6	child	**4**
3	5	hobby	swimming
3	6	hobby	cycling
2	4	hobby	painting
...	...	...	...

Table 1. A fragment of the Edge Table for XML document D.

We are obliged — since we require serializability and thus do not allow phantoms that could be introduced if row-level locking was used — to lock the entire Edge Table for U's query.
Now consider a user V that wants to make changes to the `<hobby>` *element representing* painting *(which does not appear under an element* `<child>`*). Using an RDBMS locking scheme on this and other relational representations of D would prohibit V from doing this because the entire table is locked.*

2.2. Predicate Locks

Predicate locks [23] have been introduced to fix the problem mentioned in the previous section. There is no longer any need to lock an entire table; phantoms cannot occur because predicates are given that describe the tuples that have been selected in an INSERT, UPDATE or DELETE query. New or altered tuples that satisfy these predicates cannot be added to the table, thus eliminating the threat of phantoms.

Predicate locks have two problems in view of this work, however. First, they are rarely implemented in commercial relational databases systems because they are prohibitively expensive. Indeed, testing satisfiability for even simple predicates (i.e. consisting of Boolean combinations of comparisons between a field of a tuple and a constant) is NP-complete [14]. Thus, storing XML in existing RDBMSs and using the product's locking scheme will almost certainly not offer the benefits of predicate locks.

The second and most important problem is that while predicate locks come very close to capturing the expressiveness of our proposed locking scheme, they still fall short.

Lemma 1 *Using a relational database system with predicate locks causes locking behaviour that is too restrictive for semi-structured data.*

Proof. We proof the lemma by giving an example where unnecessarily restrictive predicate locks are set. Consider an XML document D' with only a root node `<doc>`; this root has the internal element identifier 0 such that in the relational representation of D' it is identified by `source='0'`. When a user poses the query `/A//B`, this has the informal meaning that an element `<A>` may be added directly 'under' `<doc>` as long as no `<B>` element is later added somewhere under `<A>`. To evaluate the query in a relational database that uses predicate locks, consider that the query processor starts by reading all the `<A>` children of the root node such that in a next phase it can recursively look for `<B>` nodes under the `<A>` nodes. This first read query will result in the predicate lock `source='0'` $\wedge$ `name='A'`. Clearly, this is not what we want, since this predicate lock means that no-one can insert a new `A` element under the root, regardless of whether a `<B>` element gets inserted under it in a later phase or not. ■

In our Technical Report [9] we show that we can fundamentally change the process by which predicate locks are set such that the new relational locking mechanism closely mimics the way our proposed system works. However, such extentions require the re-evaluation of each active query each time an update is performed. On top of that, the satisfiability problem remains. For all these reasons, predicate locks in relational database systems are unsuited for the kind of locking mechanism we would like to use for XML documents.

2.3. Hierarchical and Tree Locking Protocols

Hierarchical locking protocols [13], also known as "multi-granularity locking protocols", are used for data that can be thought of as nested hierarchical granules and where it is important that we can place locks on granules at different levels in the hierarchy. Usually such protocols allow shared and exclusive locks at different levels but with the restriction that if a granule is to be locked then corresponding intention locks (or stronger locks) must be acquired for all the

ancestors, i.e., the granules that directly or indirectly contain this granule. Additionally if a granule is to be extended with a new element then an exclusive lock on the granule must be acquired.
If such a protocol is applied to XML data then a query like `//A//B` will require shared locks on the whole document tree and therefore disallow any update on it by other transactions.

As mentioned in the introduction, object-oriented databases typically implement some version of the hierarchical locking protocol [15]. The granules in this case are then, for example, classes, extents, objects, etc.

Tree Locking Another type of protocol that is often used for hierarchical data are so- called *tree locking protocols* [22]. In these protocols locks do not hold for entire granules but ony for nodes, i.e., when a node is locked its descendants are not also locked. However, there is also the restriction that a lock can only be acquired for a node if an identical or stronger lock was already obtained for the parent of the node. As in hierarchical locking protocols we also need to acquire an (exclusive) lock on a node if we want to add or remove children [16].
Also for these protocols it holds that a query like `//A//B` will require shared locks on all the element nodes in the document tree and thereby block any update by other transactions.

For a comprehensive overview of Hierarchical Locking and Tree Locking protocols, see [2].

3. Query and Update Language

The data model we assume for XML documents is the standard XPath data model [11], including the Document Node. The user accesses the documents through XPath-like queries. The result of these queries is either a list of node identifiers or a list of strings. We use a limited version of the surface syntax of XPath which is described by the following grammar

$$\begin{aligned} \mathcal{P} &::= \mathcal{F} \mid \mathcal{F}/\mathcal{P} \mid \mathcal{F}//\mathcal{P} \\ \mathcal{F} &::= . \mid T \mid * \mid @A \mid @* \mid \tau \mid \sigma \end{aligned}$$

where T is the set of tag names, A is the set of attribute names, τ denotes the `text()` function that retrieves all the children that are text nodes, and σ denotes the `string-value()` function that retrieves the string value of an attribute or a text node.
The path expressions can be used for queries starting from the document node or from node identifiers that were previously retrieved in the same transaction. For this purpose we assume that during a transaction the user has a set of variables $\mathcal{X} = \{\mathtt{x}_1, \mathtt{x}_2, \ldots\}$ into which she can store the intermediate results of the queries. The contents of these variables may be manipulated by the user as long as they always contain either lists of strings or lists of node identifiers that were previously retrieved in the same transaction. A query statement is now defined as a statement of the form $\mathcal{X} := \mathcal{Q}$ where the set of queries $\mathcal{Q}$ is defined by the following grammar.

$$\mathcal{Q} ::= /\mathcal{P} \mid //\mathcal{P} \mid \mathcal{X}/\mathcal{P} \mid \mathcal{X}//\mathcal{P}$$

Next to query statements we also define update operators that can be called by the user to change the document. Update operators are grouped in three classes: attribute manipulation, element manipulation and text node manipulation. The following list describes all operators in an informal way.

- Attribute manipulation
  - `create-attribute(e-id, a-name, string)` Creates an attribute with name `a-name` and string `string` under the element with identifier `e-id`.
  - `delete-attribute(a-id)` Deletes an attribute with identifier `a-id`.
  - `update-attribute(a-id, string)` Updates an attribute with identifier `a-id` to string `string`.
- Element manipulation
  - `create-element-under(e-id, tagname)` Creates an (empty) element with tagname `tagname` immediately succeeding the last child (element or text node), if it exists, of the element with identifier `e-id`. If such a child does not exist, the new element becomes the only child of the element with said identifier. The newly created element always becomes a leaf node in the document tree.
  - `delete-leaf-element(e-id)` Deletes the element with identifier `e-id` if and only if this element is a leaf node in the document tree. Otherwise no action is taken.
- Text node manipulation
  - `create-text-under(e-id, string)` Creates a text node with string `string` immediately succeeding the last child (element or text node), if it exists, of the element with identifier `e-id`. If such a child does not exist, the new text node becomes the only child of the element with said identifier.

- `delete-text(t-id)` Deletes the text node with identifier `t-id`.
- `update-text(t-id, string)` Updates the string of the text node with identifier `t-id` to `string`.

Our update language additionally includes variants to all `create-*-under` operators, in which `under` is replaced by either `before` or `after`. However, since the locking behavior of the operators is similar to that of the `create-*-under` operators, we have omitted them from the above list. In the remainder of this paper, when we discuss an `*-under`, a `*-before`, or an `*-after` operator, it will act as as a prototype for the variants.

The parameters to our operators include node identifiers which were extracted from the result of the query statements that the user posed before requesting an update. Thus, writing always implies reading.

4. Locking Scheme

The locking scheme is defined by giving a set of read locks, a set of write locks, a description of which locks must be obtained for which operation and a compatibility matrix that defines which locks can be obtained when certain other locks are already obtained by other transactions. We will assume that once a transaction acquires certain locks it keeps them until it ends.

In this paper, we introduce two equivalent systems of setting locks: "Path Lock Satisfiability" and "Path Lock Propagation". The latter system causes a multitude of read locks to be placed but makes checking for conflicts trivial, as it only checks lock equality locally (i.e. within one node). The former system, in contrast, sets very few locks but requires more work when checking for conflicting locks.

4.1. Path Lock Satisfiability

4.1.1 Read Locks

We start with the definition of the read locks. A *read lock* is defined as a tuple (n, p) where n is the node identifier for which the lock holds and p is a path in $\mathcal{P}$. The informal meaning of such a lock is that the transaction has issued a query p starting from node n.
To determine the set of all read locks R_q that must be obtained for a certain query statement $\mathtt{x}_n := q$ we use the following rules:

- If $q = \mathtt{x}_m/p$ then $R_q = \{(x, p) | x \in \mathtt{x}_m\}$;
- If $q = \mathtt{x}_m//p$ then $R_q = \{(x, .//p) | x \in \mathtt{x}_m\}$;
- If $q = /p$ then $R_q = \{(r, p)\}$ where r is the document node;
- If $q = //p$ then $R_q = \{(r, .//p)\}$ where r is the document node.

We give a small example to illustrate the proces.

Example 2 *We shall use the same document D as in Example 1. We extend the original query statement to $q =$ `//child//hobby/`τ`/`σ, indicating that we want to read the text in* `hobby` *elements that occur somewhere under the element* `child`*. Thus the XPath document node (labled 'virtual-root' in the figure) recieves the read lock `.//child/hobby/`τ`/`σ, as shown in Figure 2.*

Note that this method of placing read locks does not favour either a top-down or a bottom-up query evaluation strategy. We will briefly revisit this issue in Section 6.

4.1.2 Write Locks

We proceed with the definition of the write locks. A *write lock* is defined as a tuple (n, f) where n is the node identifier for which the lock holds and f is an element of $\mathcal{F} \setminus \{.\}$. The informal meaning of such a lock is that if a node has a lock f then the list of children corresponding to f have been changed or if $f = \sigma$ the text value of the node has been updated.
The following list defines which write locks must be obtained for which update operator:

- Attribute manipulation
  - `create-attribute(`n`, `a`, `v`)` : A $@a$ lock on n.
  - `delete-attribute(`n`)` : A $@a$ lock on the parent of n where a is the attribute name of n.
  - `update-attribute(`n`, `v`)` : A σ-lock on n.
- Element manipulation
  - `create-element-under(`n`, `t`)` : A t lock on n.
  - `delete-leaf-element(`n`)` : A t lock on the parent of n where t is the tag name of n.
- Text node manipulation
  - `create-text-under(`n`, `v`)` : A τ lock on the parent of n.
  - `delete-text(`n`)` : A τ lock on the parent of n.
  - `update-text(`n`, `v`)` : A σ lock on n.

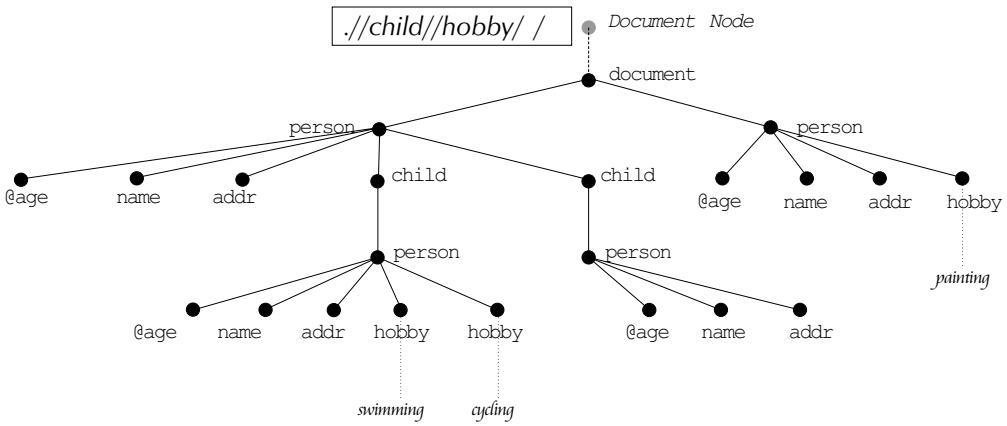

Figure 2. Document tree over D with read locks

4.1.3 Lock Compatibility

We have established which read locks are required by queries and which write locks are required by update operations. What remains to be defined is when exactly such locks can be obtained when other transactions have also acquired locks upon the same document. For this purpose we define the notion of *conflicting locks*. It is defined by the rules in the following definition.

Definition 1 *The following rules define which locks conflict.*

1. *Two read locks never conflict.*
2. *A read lock (n, p) conflicts with a write lock (n', f) if and only if n is an ancestor of n' and* path$(n, n')/f$ *satisfies p.*
3. *A write lock (n, f) always conflicts with a write lock (n, f').*

Informally, the *path* operator used in rule 2 denotes the pathname of the path from an ancestor to a descendant node; i.e., *path*(n, n') is the list of element names (separated by '/') encountered on the path from n to n'.
A pathname *satisfies* an XPath expression p if it is equal to one of the pathnames represented by p.

Whether a lock can be obtained by a transaction is determined as follows:

> A transaction can obtain a lock iff there is no other transaction that holds a conflicting lock.

Example 3 *Considering the query of Example 1, suppose that we want to insert a* `child` *element under the first* `person` *element in the tree. This results in the read lock (*`dr`, `.//child//hobby`*) and the write lock (*`p01`, `child`*) over document D. The literal* `p01` *is the identifier of the node that represents the first* `person` *child of the* `document` *element, while* `dr` *denotes the Document Node.*
To see whether there is a conflict, we need to check whether the path path$(\texttt{dr}, \texttt{p01})/\texttt{child}$ *satisfies* `.//child//hobby`*. Since* `./document/person/child` *is only a prefix of a direct path defined by* `.//child//hobby`*, the two locks do not conflict and the update (i.e., insertion of a* `child` *element under the first* `person`*) element will be performed.*

4.1.4 Complexity

Clearly the space complexity in this system is not an issue; very few locks need to be set. Relative to the size of the tree, the space complexity is $O(1)$. However, the time complexity is more important here than in the alternative method. Indeed, we need to check the satisfiability of a direct path by a more general XPath expression. Such a test can be likened to checking whether a string is accepted by a non-deterministic automaton. The XPath expression p can be written as an NFA while a direct path t can be seen as a string. If the string is accepted by the NFA, the direct path is equal to one of the direct paths represented by p. From Automata Theory, we know that this problem is of time complexity $O(n^4)$.

4.2. Path Lock Propagation

4.2.1 Read Locks

The set of read locks in this protocol is the same as in the previous path lock satisfiability protocol. Given a query statement $\mathtt{x}_n := q$ the protocol starts with placing the locks in R_q as was defined in the lock satisfiability protocol. We will call this set of locks in this protocol the *initial lock set*.

From the initial lock set we derive other read locks that must also be obtained by processes called *read-lock inference* and *read-lock propagation*. With read-lock inference we mean the derivation of locks that must also be obtained for the same node. This is done with the rules shown in Figure 3.

$$.//p \Rightarrow p$$
$$./p \Rightarrow p$$

Figure 3. Inference Rules for Read Locks.

The process of read-lock propagation causes read locks on a node to be propagated to nodes just below this node in the document tree. This is done with the rules shown in Figure 4.

parent lock	child type	child name	child lock
`.//`p	element	-	`.//`p
t`/`p	element	t	p
t`//`p	element	t	`.//`p
`*/`p	element	-	p
`*//`p	element	-	`.//`p
`@`a`/`p	attribute	a	p
`@*/`p	attribute	-	p
τ`/`p	text	-	p

Figure 4. Propagation Rules for Read Locks.

The processes of read-lock inference and read-lock propagations are applied to R_q until no more new locks are added. The result is R_q^*, the set of read locks that have to be acquired for the query q. Since R_q^* depends upon the document tree it has to be recomputed every time this tree is updated.

We give a small example to illustrate the processes.

Example 4 *We shall use the same document D as in Example 1. We extend the original query statement to $q =$* `//child//hobby/`τ`/`σ*, indicating that we want to read the text in* `hobby` *elements that occur somewhere under* `child`*. Note that this is a rather powerful query, in the sense that it will traverse the whole tree. Usually, queries will be less powerful, and fewer locks will have to be set.*

The initial read lock is `.//child//hobby/`τ`/`σ *for the document node. With the first inference rule we also derive the lock* `child//hobby/`τ`/`σ *for this node.*

The tree in Figure 5 depicts, in boxes, the locks associated to the nodes of D. These locks are abbreviated to make the figure readable; the full version is in the following table.

	lock	*inferred lock*
q	`.//child//hobby/`τ`/`σ	`child//hobby/`τ`/`σ
q_1	`.//hobby/`τ`/`σ	`hobby/`τ`/`σ
q_2	τ`/`σ	
q_3	σ	

Note that the full *path makes up a lock (hence the name Path Lock); they cannot be abbreviated. Indeed, consider that the* `document` *node would only have* `child` *as its lock. This would mean that another transaction cannot add a* `child` *node under* `document`*, which is of course not what we want. A* `child` *may be added under* `document` *as long as that* `child` *does not itself receive a* `hobby` *node as a child.*

Although the powerful query stated in the above example causes many locks to be set in the tree, these locks can be efficiently stored in a hash table. The hash table relates nodes in the tree with the correct locks.

The mechanisms of lock inference and lock propagation may seem a bit involved but they correspond closely to how the query may actually be computed, e.g., the nodes to which the locks are propagated are the same as the nodes that have to be visited to compute the query. Therefore they can be computed at very little extra cost. Moreover, the recomputation of R_q^* after an update of the document tree can be done by propagating only the locks of the parent that a node is created or deleted under.

Note that in the propagation system, the setting of locks is an atomic operation. Either all locks that are infered or propagated are obtained, or none are.

4.2.2 Write Locks

Write Locks are set in exactly the same manner as described in Section 4.1.2.

4.2.3 Lock Compatibility

We have established which read locks are required by queries and which write locks are required by update operations. What remains to be defined is when exactly such locks can be obtained when other transactions have also acquired locks upon the same document. For this purpose we define the notion of *conflicting locks*. It is defined by the rules in the following definition.

Definition 2 *The following rules define which locks conflict.*

1. *Two read locks never conflict.*
2. *A read lock (n, p) conflicts with a write lock (n, f) if and only if $p = f \vee p =$*.

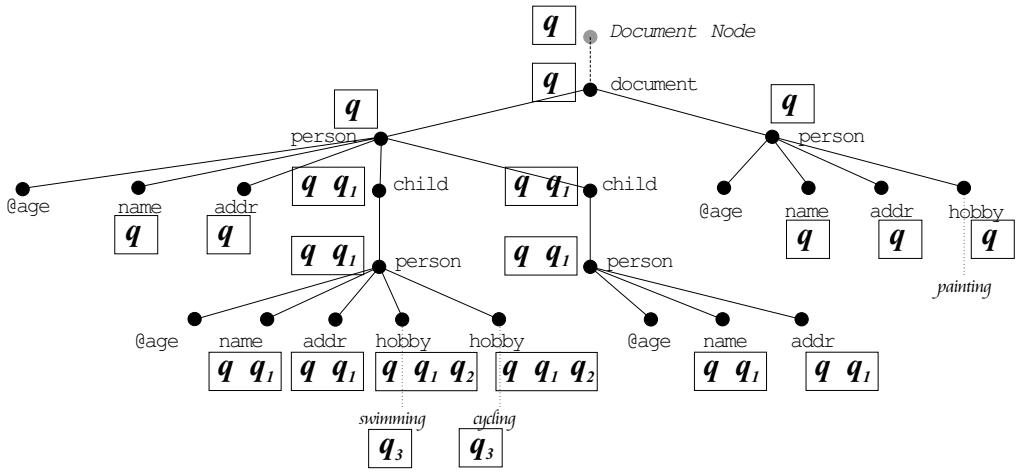

Figure 5. Document tree over D with propagated read locks

3. *A write lock (n, f) always conflicts with a write lock (n, f').*

Whether a lock can be obtained by a transaction is determined as follows:

> A transaction can obtain a lock iff there is no other transaction that holds a conflicting lock.

4.2.4 Complexity

Checking for a read- and write lock conflict clearly becomes trivial in this system. The two locks are present in the same node, and only the equality of the read lock to the write lock or to * needs to be checked. Thus, the time complexity of checking for conflicts in the propagation system is $O(1)$.
The setting of read locks can occur at the time of query evaluation; this causes only a small increase in execution time.
Thus, while the time complexity for this method is low w.r.t. the alternative method, the space complexity is a more serious issue. Specifically, the space complexity is $O(nm)$, where n is the size of the tree, and m is the length of the lock expression.

4.3. Equivalence

It is relatively easy to see that the "Path Lock Satisfiability" system and the "Path Lock Propagation" system are in some sense equivalent. With 'equivalent' we mean that the same transaction schedules will cause the same conflict behavior in the two alternative methods. As mentioned before, there is only a trade-off between the quantity of locks to be set (space) and the testing of lock conflicts (time).

5. Theoretical Results

The theoretical results given in this section make use of the "Path Lock Satisfiability" system. As the "Path Lock Propagation" system is equivalent, these results can also be obtained for the alternative system.

5.1. Serializability

The purpose of a locking scheme is to ensure serializability, i.e., if the schedule of a set of transactions is accepted by the locking scheme then a certain linear order of the transactions exists such that if the transaction were consecutively executed in that order then the answers to all the queries in transactions and the final database state should be the same as for the accepted schedule. We will give here only a sketch of the proof.

Theorem 1 *If two consecutive operations occuring in different transactions do not conflict then they commute.*

Proof. If two operations in different transactions do not conflict then they are either

- two read operations,
- a read and an update operation, or
- two update operations on different nodes.

Suppose that we have two read operations then these trivially commute. Next we consider the case of a read and an update operation. If the update operation is an insert then the new node is not in the result of the read operation if the corresponding locks do not conflict. Since after an insert operation the result of a query stays the same or is extended

with the new node, it follows that the result of the read operation will be the same before and after the insert operation. If the update operation is a delete operation then the deleted node is not in the result of the read operation since the corresponding locks conflict, as is the asumption. Since after a delete operation the result of a query stays the same or the deleted node is removed from the result, it follows that the result of the read operation is the same before and after the delete operation. Finally, we consider two update operations on different nodes. Since these two operations cannot influence each other and work on different parts of the tree it follows trivially that they commute. ∎

5.2. Necessity

The conflict rules given in Definition 1 are slightly too restrictive. Indeed, two transactions should be able to delete leaf elements no matter in which order they are executed since the end result will be the same. Such delete-delete combinations are, however, prevented by the rule which says that any write lock conflicts with any other write lock at the same node.
The rule is included, however, since this is the only case in which the rule is too restrictive; for the two other combinations of updates (insert-delete and insert-insert), the rule is exactly right.

We can adapt Definition 1 to the case of unordered trees. When the order of the children of a node is not important, more update combinations are allowed. For instance, two transactions that insert a child can commute.
The correct conflict rules for unordered trees are as follows: in Definition 1 the third rule is replaced by

3. A write lock (n, f) never conflicts with a write lock (n, f').

The observant reader will notice that this rule would allow an insert-delete combination to commute. However, since write locks always imply read locks, this combination will be correctly blocked.

We now prove the necessity of the conflict rules for unordered trees. This result shows that the adapted conflict rules for unordered trees are not too restrictive.

Theorem 2 *If two consecutive operations occuring in different transactions conflict according to the locking protocol then they cannot commute.*

Proof. In unordered trees, read-read operations and update-update operations never conflict. Thus, the only two types of operations that can conflict are read operations and update operations (i.e. insert and delete). If we look at a read operation and an insert operation that conflict, then the insert operation requires a t write lock on a node n and the query operation requires a read lock with path p on an ancestor n' of n such that the path path(n',n)/t satisfies the path expression of the query operation. If this is the case then the inserted node will be in the result of the query if the insert precedes the query and therefore the two operations do no commute.
If we look at a conflicting read operation and a delete operation, then the delete operation requires a t write lock on a node n and the query operation requires a read lock with path p on an ancestor n' of n such that the path path(n',n)/t satisfies the path expression of the query operation. If this is the case then the deleted node will be in the result of the query if the query precedes the delete and therefore the two operations do no commute. ∎

6. Using References as Indices

We have so far not discussed the use of references in XML documents. In this section we show a naive way to simulate such use. We then show how this method can be used to construct an index mechanism to speed up queries.

6.1. Naive Simulation of References

In an XPath expression we can use an XML reference to jump from one subtree to another. Our locking protocol currently does not handle such jumps efficiently. However, we offer a naive way to simulate the use of references. In future work, we will propose improvements.

To follow a reference in our locking mechanism, we propose to treat references as if they were simply attributes with no special meaning, querying their value just like those of other attributes. We can subsequently compare this value with the identifier of another node. An example will clarify this.

Example 5 *To follow an* `idref` *with the name* `child` *from a* `person` *element we write two queries. The result of query* $q_1 =$ `//person/@child` *is the value* v *of the idref which we use in the second query:* $q_2 =$ `//person[@id=`v`]`.

Note that, strictly speaking, we cannot express the equality within a query, as in q_2. Rather, the user program will have to compare the result of q_1 with the result of q_2. This issue will be addressed in future work.

The approach given here is naive in the sense that for both queries our concurrency control mechanism will insert locks. This is correct behaviour for the node that contains the reference, but it is too restrictive for the referenced node. Indeed, consider that the second query will place (weak)

locks on the ancestors of the referenced node. This implies that the user has read the path leading to the referenced node, instead of jumping there directly from the referring element.

6.2. Using References as Indices

References can be used to model indices that allow direct access to certain nodes within the document tree. A structure like a B-tree, for example, can be seen as a tree with references to another tree that contains the indexed information. Combined with the simulation of references as described before, this tells us how the locking protocol should proceed if the document is accessed through the index.

7. Further Research

The current approach to dereferencing needs to be replaced to ensure more concurrency. We are currently extending the query language to include conditional paths, which will make the use of references more efficient.
For more sophisticated collaborative applications, our locking mechanism should be extended such that the reordering of the chapters of a book by one user while other users are writing parts of these chapters, is allowed. Thus, this would imply an even more sophisticated, and possibly less efficient, locking scheme. We are currently ([9]) extending the update language to include a `move-tree` operator which will enable such updates and thus allow for more concurrency.
The query language needs to be extended with other XPath axes such as `parent`, `ancestor`, `following` and others, and also for other functions that are allowed in XPath.

8. Conclusion and Acknowledgements

We have presented a locking scheme for XML documents that allows the same document to be queried and updated by more than one user, whilst guaranteeing serializability. The locking scheme is fine-grained relative to existing mechanisms in relational and object-oriented databases, and allows the locking of subtrees of the document tree and more complicated subsets if these can be described by a certain subset of XPath queries.

We would like to thank the anonimous referees of Wise'02 for their insightful and valuable comments.

References

[1] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The LOREL query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, April 1997.

[2] N. Barghouti and G. Kaiser. Concurrency control in advanced database applications. *ACM Computing Surveys*, 23(3):269–317, 1991.

[3] S. Boag, D. Chamberlin, M. Fernández, et al. XQUERY 1.0: An XML query language. *W3C Working Draft*, 2001.

[4] T. Bray, J. Paoli, et al. Extensible markup language (XML) 1.0 (second edition). *W3C Recommendation*, October 2000.

[5] P. Cederqvist et al. Version management with CVS. *WWW Manual*, http://www.cvshome.org/docs/manual/, 1993.

[6] W. Cellary, E. Gelenbe, and T. Morzy. Concurrency control in distributed database systems. *Studies in Computer Science and Artifi cial Science (3)*, ISBN: 0-444-70409-4, North-Holland, 1988.

[7] J. Clark and S. DeRose. Xml path language (xpath). *W3C Recommendation*, November 1999.

[8] C. Date. *An Introduction to Database Systems, 7th edition*. ISBN: 0-201-38590-2, Addison Wesley Longman, 2000.

[9] S. Dekeyser and J. Hidders. A basic locking protocol for xml. *Technical Report 02-05, UIA*, ftp://win-ftp.uia.ac.be/pub/dekeyser/basiclocking.ps, 2002.

[10] A. Deutsch, M. Fernández, and D. Suciu. Storing semistructured data with STORED. *Proceedings ACM SIGMOD*, Philadephia, 1999.

[11] M. Fernández, J. Marsh, and M. Nagy. Xquery 1.0 and xpath 2.0 data model. *W3C Working Draft*, August 2002.

[12] D. Florescu and D. Kossmann. Storing and querying xml data using an rdmbs. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.

[13] J. Gray, A. Lorie, et al. Granularity of locks in a large shared data base. *Proceedings of the International Conference on Very Large Data Bases*, Framingham, Massachusetts, 1975.

[14] H. Hunt and D. Rosenkrantz. The complexity of testing predicate locks. *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, Boston, Massachusetts 1979.

[15] S. Khoshafi an. *Concurrency Control in Object-Oriented Databases*. John Wiley & Sons, ISBN 0-471-57058-3, 1993.

[16] V. Lanin and D. Shasha. Tree locking on changing trees. *FJCC*, pages 380–389, 1986.

[17] A. Laux. LEXUS: XML update language. *Working Draft 00.09.09*, Infozone Project, http://www.infozone-group.org/lexusDocs/html/wd-lexus.html, 1999.

[18] A. Laux and L. Martin. XUPDATE: XML update language. *XML:DB Initiative*, www.xmldb.org/xupdate/, 2000.

[19] M. Loomis. Object databases, the essentials. *ISBN: 0-201-56341-X, Addison Wesley*, 1995.

[20] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, 1997.

[21] F. Neven and T. Schwentick. On the power of tree-walking automata. *Information and Computation (to appear), extended abstract in ICALP*, 2000.

[22] A. Silberschatz and Z. Kedem. Consistency in hierarchical database systems. *Journal of the ACM*, 27(1):72–80, 1980.

[23] G. Weikum and G. Vossen. Transactional information systems. *ISBN: 1-55860-508-8, Morgan Kaufmann*, 2002.

[24] L. Wood et al. Document object model (dom) level 1 specifi cation. *W3C Recommendation*, October 1998.