Conflict Scheduling of Transactions on XML Documents

Stijn Dekeyser*

University of Southern Queensland
dekeyser@usq.edu.au

Jan Hidders

University of Antwerp
jan.hidders@ua.ac.be

Abstract

In the last few years an interest in native XML databases has surfaced. With other authors we argue that such databases need their own provisions for concurrency control since traditional methods are inadequate to capture the complicated update-behavior that is possible for XML documents. Ideally, updates should not be limited to entire document trees, but should involve subtrees and even individual elements. Providing a suitable scheduling algorithm for semistructured data can significantly improve collaborative systems that store their data — e.g. word processing documents or vector graphics — as XML documents.

In this paper we improve upon earlier work which presented two equivalent concurrency control schemes based on Path Locks, and a commit scheduler for these schemes. In contrast to the earlier work, we now introduce a *conflict* scheduler for XML databases which uses the same path lock conflict rules and the same basic query and update languages. This new scheduler has significantly different properties than the commit scheduler. We also give a comprehensive proof of serializability of schedules accepted by the new scheduler.

Keywords: XML, semistructured data, path locks, transaction model, serializability, concurrency control, conflict scheduler.

1 Introduction

Even though XML is not meant to replace traditional database systems, lately an interest in native XML databases has surfaced. Consequently, all features present in relational and object-oriented databases will be revisited in the context of semistructured data. One such feature is the necessity of a concurrency control mechanism in any type of database.

Concurrency control (Weikum & Vossen 2002) has been extensively studied in the context of traditional database management systems (Bernstein, Hadzilacos & Goodman 1987, Papadimitriou 1986). This literature has introduced important concepts such as locks, transactions, schedulers, etc. Protocols such as *two phase locking* have been proposed to ensure serializability (Eswaran, Gray, Lorie & Traiger 1976) — the central theoretical notion of correctness for concurrent database systems — of schedules of data manipulation language actions.

Also, locking mechanisms such as predicate locking (Eswaran et al. 1976), hierarchical locking (Gray, Putzolo & Traiger 1976) and tree based locking protocols (Silberschatz & Kedem 1980) have been introduced to suit special needs and to increase the level of concurrency that is allowed by schedulers. Another track of concurrency control research has focused on decidability of serializability and on the complexity of algorithms.

The result of this research has been the incorporation of efficient scheduling algorithms in DBMSs which ensure serializability and which allow a high degree of concurrency.

Clearly, it is possible to re-use these well-known results for providing concurrency control in semistructured databases. However, as has been shown in earlier work (Dekeyser & Hidders 2002, Grabs, Bohm & Schek 2002, Dekeyser, Hidders & Paredaens 2003*b*), the traditional solutions we mentioned above — while guaranteeing serializability — do not allow a sufficient degree of concurrency; i.e., they are too restrictive. We will come back to this issue in Section 2.

Looking at existing semistructured database systems, we found that the *Lore* system (Abiteboul, Quass, McHugh, Widom & Wiener 1997) 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 semistructured nature [would] require us to rethink some aspects of traditional solutions" to concurrency control. The Natix project proposed in (Fiebig, Helmer, Kanne, Moerkotte et al. 2002) recognizes that transaction management needs a different approach than traditionally thought; however, they focus mainly on recovery and isolation, and use standard hierarchical locking protocols introduced by Gray et al. in 1976.

As a consequence, the problem statement for our work is "what kind of conflict rules and scheduling algorithm for semistructured databases can guarantee both serializability and a high degree of concurrency?"

Apart from our own work, we have so far found only three papers and a technical report which attempt to solve this problem. These will be discussed and compared in Section 2.

Applicability Native XML databases (NXDs) are currently gaining popularity. Unfortunately, most

* Contact author's affiliation was the University of Antwerp, Belgium, at the time this research was conducted.

systems are document-based rather than offering concurrency at the level of individual elements. It is clear that much more collaboration could be achieved if different clients could connect to the same XML document and update it concurrently. This would allow multiple authors to work on the same word processing file, or on large vector graphics that are stored as an XML document.

Evidently, many other applications may benefit from the theory presented in this paper.

Contribution In previous work (Dekeyser & Hidders 2002, Dekeyser & Hidders 2003), we have investigated the use of path locks to solve the research problem mentioned in the introduction. We introduced two equivalent locking protocols: path locks satisfiability (PL-SAT) and path locks propagation (PL-PROP). For both systems, we introduced conflict rules and analyzed their complexity. We showed that the two systems are equivalent with respect to the conflicts they detect. We also indicated that the conflict rules are necessary. In addition, we introduced a *commit* scheduler that uses the PL-PROP system to guarantee serializability.

In this paper, however, we introduce the *conflict* scheduler which also makes use of the PL-PROP system. We indicate the properties which make this scheduler significantly different from the previous scheduler. Importantly, we outline aspects of recovery mechanisms using roll-back and abort operations. We also present a comprehensive proof of serializability, and show how the conflict scheduler can be extended to handle commit operations. Finally, we briefly report on on-going work to implement our transaction model for XML databases.

Organization The paper is organized as follows. Section 2 briefly restates from previous work that existing concurrency control mechanisms from relational and object-oriented databases are inadequate for our purposes. It also compares the few other papers related to our work. Section 3 formally introduces the data model, which captures most of the XPath data model features. The section also presents the query language and the update primitives that can be used to manipulate documents. Section 4 presents the two path lock schemes that can be used by schedulers to ensure serializability. We briefly sketch the advantages and disadvantages of both. Section 5 details the working of the conflict-scheduling algorithm, while section 6 proves that this scheduler guarantees serializability. Section 7 extends the conflict scheduler to include the use of a commit operation, and proves that this extension also guarantees correctness. Section 9 briefly outlines our ongoing efforts to implement the ideas described in this paper. Finally, Section 10 concludes the paper and mentions future research possibilities.

2 Related Work

2.1 Relational Methods

Semistructured data can be stored in traditional relational databases in many different ways (Deutsch, Fernández & Suciu 1999, Florescu & Kossmann 1999). 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 modeled within one relation. Though this relationship can indeed be split up in several tables, in general one table will — because of XML's semistructured 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. In table locking, the entire parent-child table will be locked upon an update, blocking other updates. In predicate locking, a predicate will lock without taking the hierarchy inherent in the document into account. Thus, this method will also make certain updates impossible.

2.2 Hierarchical Methods

In hierarchical and tree locking methods, an update at one level always requires locks to be set at higher levels in the tree. Furthermore, an exclusive lock is required on a node if one wants to add or remove children from that node. These requirements may be too strict in many cases.

Example 1 *Consider the XML document in Figure 1. Assume that a first user wants to add a* `hobby` *element to the person with* `id` $= 2$*, while a second, concurrent, user wants to add a* `child` *element to the same person. Hierarchical and tree-based locking protocols will typically prevent such concurrent action because the* `person` *element will be locked.*

More examples, for different situations and different locking protocols, can be found in the on-line version of the PhD thesis referred to in (Dekeyser 2003).

2.3 New Proposals

XPath-based Proposals. Several recent papers, including our own, which attempt to solve concurrency control for XML and semistructured databases are based on the observation that the data are usually accessed by means of XPath expressions. In this paper, as in our previous work, we propose to use a simplified form of XPath expressions as 'path locks' on the document, such that precisely all operations that change the result of the expression are no longer allowed.

A similar approach is taken in (Hye Choi & Kanai 2003) where conflicts with path locks are detected by accumulating updates in the XML tree and intelligently recomputing the results of the path expressions. As a result they can allow more complex path expressions, but conflict checking becomes more expensive.

Another related approach is presented in (Jea, Chen & Wang 2002) where locks are derived from the path expressions and a protocol for these locks is introduced that guarantees serializability.

DOM-based Proposals. Several locking protocols not based on path expressions but on DOM operations are introduced in (Helmer, Kanne & Moerkotte 2003). Here, there are locks that lock the whole document, locks that lock all the children of a certain node and locks that lock individual nodes or pointers between them. An interesting new aspect is here the possibility to use the DTD for conflict reduction and thus allowing more parallelism. Although these locking protocols seem very suitable in the case of DOM operations, it is not clear whether they will also perform well if most of the access is done by path expressions.

DataGuide-based Proposals. A potential problem with many of the previously mentioned protocols is that locks are associated with document nodes and so for large documents we may have large numbers of locks. A possible solution for this is presented in (Grabs et al. 2002), where the locks are associated with the nodes in a DataGuide, which is usually much

```
<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>
      </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.

smaller than the document. However, in general there is a trade-off between fewer locks in the DataGuide and the amount of concurrency allowed, as a lock on a node in the DG conceptually locks several nodes in the corresponding document tree. Much more serious, unfortunately, is that the proposed protocol does not guarantee serializability and allows phantoms. Moreover, the query language does not support the use of the *descendant-of* axis of XPath which is vital for querying semistructured data.

3 Data Model and DML

The data model we assume for XML documents is a simplification of the standard XPath data model and is similar to the data model classically used for semistructured data (Abiteboul, Buneman & Suciu 1999). Thus, we essentially consider node-labeled trees. However, for the purpose of locking we will allow more general acyclic graphs. We label nodes with a set of transaction identifiers to indicate that the node has been deleted by these transactions.

Definition 1 *The* instance graph (N, B, r, ν, δ) *is a rooted acyclic graph with vertices N, edges $B \subseteq N \times N$, the root r, nodes labeled with element names by $\nu : N \to E$ and with sets of transaction identifiers by $\delta : N \to 2^T$. The subgraph defined exactly by the nodes that are labeled by δ with the empty set is called the* actual instance *and is presumed to be always a tree with root r.*

In contrast to the XPath data model, in our model there is no distinction between elements, attributes, or text. Also, we do not consider an order between elements. These features can, however, be simulated in the instance graph model in a straightforward manner. For example, order can be simulated by using a skewed binary tree.

Example 2 *Figure 2 shows an instance graph representation d of document D given in Figure 1. Note that for clarity only a few labelings are shown; nodes are given a different coloring according to their type in D (elements are black, text is white, and attributes are gray). In our model, no distinction is made, however. The figure also shows that the document order is not preserved.*

Data Manipulation Language Now that we have defined the data model, we turn our attention to the data manipulation language. The query language is based on a subset of XPath expressions as defined by the following grammar:

$$\begin{aligned} \mathcal{P} &::= \mathcal{F} \mid \mathcal{P}/\mathcal{F} \mid \mathcal{P}//\mathcal{F} \\ \mathcal{F} &::= E \mid * \end{aligned}$$

where E is the universal set of strings representing the names of elements.

The path expressions can be used for queries starting from the root node or from nodes that were previously retrieved in the same transaction, since we assume that during a transaction the user has a set of variables into which he 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 sets of nodes that were previously retrieved in the same transaction.

We now turn to the semantics of path expressions. These will be useful in Section 4.2 in the description of conflict rules in the Path Lock Satisfiability scheme. In the following definition, the dot $\cdot$ denotes the concatenation of sets of strings. Also, we use the Kleene-star * to denote the zero or more recurrences of a substring.

Definition 2 *Let $\mathbf{L}(p)$ be the set of label paths selected by path expression p. We define $\mathbf{L}$ recursively*

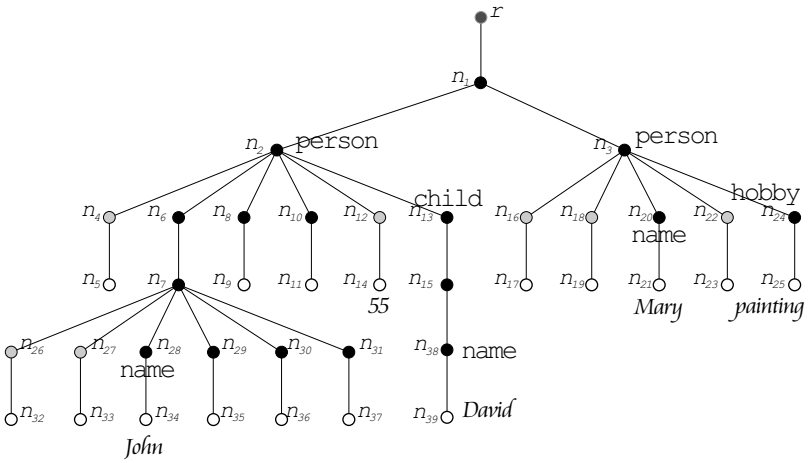

Figure 2: An instance graph representation of XML document D.

as follows.

$$\begin{aligned} \mathbf{L}(*) &= E \\ \mathbf{L}(e) &= \{e\} \text{ with } e \in E. \\ \mathbf{L}(p/f) &= \mathbf{L}(p) \cdot \{/\} \cdot \mathbf{L}(f) \\ \mathbf{L}(p//f) &= \mathbf{L}(p) \cdot \{/\} \cdot (E \cdot \{/\})^* \cdot \mathbf{L}(f) \end{aligned}$$

Thus, $\mathbf{L}(p)$ is the language of which the strings are the label paths represented by the path expression p.

The following definition enumerates the operations offered by the data manipulation language that can be used to alter a document.

Definition 3 *The following operations are defined on an instance graph.*

$\mathbf{A}(n, a)$ *This update operation, corresponding to an insert, adds a new edge starting from* n *and ending in a new node with label* a*. The new node is returned as the result of the operation. If in the new instance graph the actual instance is not a tree with root* r *then the operation fails*[1]*.*

$\mathbf{D}(n)$ *This update operation, which corresponds to a delete, adds the transaction identifier of the transaction that requests the operation to* $\delta(n)$*. This operation returns no result. If in the new instance graph the actual instance is not a tree with root* r *then the operation fails.*

$\mathbf{Q}(n, p)$ *This query operation returns as its result all nodes in the instance graph such that there is in the* actual instance *a path from* n *to this node that satisfies the path expression* p*.*

Note that the parameters to our operators include nodes which were extracted from the result of the query statements that the user posed before requesting an update. Thus, writing (i.e., updating the document) always implies reading.

Now that we have defined the operations of the data manipulation language, we turn to some traditional definitions from transaction management theory.

Definition 4 *An* action *is a pair* (o, t) *where* o *is one of the operations given in Definition 3 and* t *is a transaction identifier. A* transaction *is a finite list of actions having the same transaction identifier. A* schedule *is an interleaving of several transactions. A schedule is said to be* node-correct *if for every operation that uses a certain node there is an earlier action (containing an addition or a query) of the same transaction that had this node in its result.*

We give a short example next.

Example 3 *Consider the instance graph* d *given in Figure 2 which acts as the initial instance graph. Consider also the schedule*

$$\begin{aligned} S = \langle & a_1(\mathbf{Q}(r, //\texttt{person}), t_1), \\ & a_2(\mathbf{Q}(r, \texttt{document}), t_2), \\ & a_3(\mathbf{A}(n_3, \texttt{child}), t_1), \\ & a_4(\mathbf{D}(n_1), t_2)\rangle. \end{aligned}$$

This schedule[2] *is node-correct. The first three actions do not fail, while the last one does since the resulting document is not a tree (the graph becomes unconnected).*

Following tradition, two schedules are *equivalent* if (1) one is a permutation of the other, (2) the resulting actual instance is in both cases the same, and (3) all the queries in one schedule return the same result as the corresponding queries in the other schedule. A schedule is said to be *serializable* if it is equivalent with a serial schedule.

4 Path Locks

We now turn to the locking schemes that can be used by the scheduler to ensure serializability. In this paper we shall discuss both the path lock propagation (PL-PROP) scheme and the path lock satisfiability (PL-SAT) scheme. The PL-PROP scheme has the advantage that most proofs can be written more legibly; however, its clear disadvantage is that it requires more locks than PL-SAT. We refer to (Dekeyser 2003) and (Dekeyser et al. 2003*b*) for more details regarding the trade-off between PL-PROP and PL-SAT.

4.1 Path Lock Propagation Scheme

We start with the definition of the read locks. A *read lock* is defined as a tuple $\mathrm{rl}(t, n, p)$ where t is a transaction identifier, n is the node identifier in the instance graph for which the lock holds and p is a

[1] Failure means here that the scheduler does not execute the operation and reports this to the transaction that requested it.

[2] Note that technically the query in a_1 is not possible since it starts with //. However, this can be easily simulated.

path expression in $\mathcal{P}$. The informal meaning of such a lock is that the transaction has issued a query p starting from node n.

The *initial read lock* that must be obtained for a given query operation $\mathbf{Q}(n, p)$ that is issued by transaction t is simply $\mathrm{rl}(t, n, p)$. From the initial read lock we derive other read locks that must also be obtained by a process called *read-lock propagation*. The process of read-lock propagation causes read locks on a node to be propagated to nodes just below this node in the instance graph. This is done with the rules shown in Figure 3. The process of read-lock propagation is applied until no more new read locks are added; this process ends since the instance graph is both finite and acyclic.

The result is a set R_q^* of read locks that have to be acquired for the query $\mathbf{Q}(n, p)$. Since R_q^* depends upon the instance graph, it has to be recomputed every time the instance graph is updated. The recomputation of R_q^* after an update can be done by propagating only the locks of the parent that a node is created or deleted under. Thus, it can be done relatively efficiently.

We proceed with the definition of the write locks. A *write lock* is defined as a tuple $\mathrm{wl}(t, n, f)$ where t is a transaction identifier, n is the node identifier for which the lock holds and f is an expression over $\mathcal{F}$.

The following defines which write locks must be obtained for which update operator:

$\mathbf{A}(n, a)$: A write lock $\mathrm{wl}(t, n, a)$ on node n for transaction t.

$\mathbf{D}(n)$: Write locks $\mathrm{wl}(t, n, *)$ and $\mathrm{wl}(t, n', a)$ where n' is the parent of n in the instance graph and a is the label of n. If n or n' does not exist, then the corresponding write lock does not need to be obtained.

We now turn to an example to clarify the definitions in this section.

Example 4 *Consider the instance graph d given in Figure 2 and the first two actions of schedule S given next.* $S =$ $\langle a_1(\mathbf{Q}(r,$ `document/person/child/person/name`$), t_1),$ $a_2(\mathbf{D}(n_{25}), t_2), \ldots\rangle$.

The following table presents all the locks associated to d', which is the instance graph obtained after a_1 and a_2 are applied to instance graph d.

$rl(t_1, r,$ `document/person/child/person/name`$)$
$rl(t_1, n_1,$ `person/child/person/name`$)$
$rl(t_1, n_2,$ `child/person/name`$)$
$rl(t_1, n_3,$ `child/person/name`$)$
$rl(t_1, n_6,$ `person/name`$)$
$rl(t_1, n_{13},$ `person/name`$)$
$rl(t_1, n_7,$ `name`$)$
$rl(t_1, n_{15},$ `name`$)$
$wl(t_2, n_{24},$ `painting`$)$
$wl(t_2, n_{25}, *)$

It is clear that, while in many cases the PL-PROP scheme does not generate very many locks, in case the *descendant-of* axis of XPath is used, PL-PROP may require a large amount of locks. However, most of them do not cause conflicts (see the next paragraph), and they can also be stored efficiently. Even so, we will further on introduce the PL-SAT scheme to alleviate this problem.

To end this section, we need to define when locks *conflict*. A read lock $\mathrm{rl}(t, n, a)$ or $\mathrm{rl}(t, n, *)$ conflicts with a write lock $\mathrm{wl}(t', n, a)$ and a write lock $\mathrm{wl}(t', n, *)$ if $t \neq t'$. All other locks do not conflict.

Example 5 *The read lock $rl(t_1, n_{16},$* `name`*$)$ conflicts with the write lock $wl(t_2, n_{16},$* `*`*$)$. The instance graph d' of Example 4 did not contain conflicting locks.*

Note that two write locks do not conflict due to the node-correctness property of transactions. This property implies that consecutive additions and deletions always commute.

4.2 Path Lock Satisfiability Scheme

We now turn to an alternate path lock scheme. In contrast to the PL-PROP scheme, the satisfiability scheme requires fewer locks to be obtained but is more complex with regard to testing for conflicts. Thus, there is a trade-off between time and space complexity. Additionally, the PL-SAT scheme does not imply a top-down query evaluation strategy (McHugh & Widom 1999). For more details on these issues, we turn the reader to (Dekeyser 2003).

Read Locks Read locks in SAT are defined as in PROP. However, it is sufficient to obtain for a given query operation only the initial read lock. Thus, the lock propagation process is not applied in this case.

Write Locks Write locks in SAT are defined exactly as in PROP. Also, the update operations are to obtain the same write locks as defined earlier.

As in the path lock propagation scheme, we must yet define when path locks in the satisfiability scheme conflict. Read lock $\mathrm{rl}(t, n, p)$ conflicts with write lock $\mathrm{wl}(t', n', f)$ iff (1) $t \neq t'$, (2) n is an ancestor of n', and (3) $\overline{\lambda}(n, n')/f \subseteq \mathbf{L}(p)$. All other locks do not conflict.

Example 6 *Consider the same schedule S and instance graph d' as in Example 4. In the* SAT *scheme, the following are the only path locks that need to be obtained.*

$rl(t_1, r,$ `doc/person/child/person/name`$)$
$wl(t_2, n_{24},$ `painting`$)$
$wl(t_2, n_{25}, *)$

There are no conflicting locks, as $\overline{\lambda}_{d'}(r, n_{24})/$`painting` $\notin \mathbf{L}($`doc/person/child/person/name`$)$. *Likewise,* $\overline{\lambda}_{d'}(r, n_{25})/* \not\subseteq \mathbf{L}($`doc/person/child/person/name`$)$.

5 The Conflict Scheduler

In this section we detail the working of the conflict scheduler. In contrast to the commit scheduler proposed in (Dekeyser & Hidders 2003) which effectively lets transactions wait if their request cannot be processed, the conflict scheduler keeps accepting and processing actions until it fails (i.e., it detects a cycle in its dependency graph). Thus, it allows for more concurrent actions than the commit scheduler, but it may cause cascading roll-backs in case of recovery (see Section 8). Consequently, the choice between the commit and the conflict scheduler is highly dependent on the type of application.

Definition 5 *The* conflict scheduler *is the automaton whose state consists of a schedule S of actions that it has previously accepted and processed, a set of locks L, a dependency graph G which is a directed graph whose nodes are transaction identifiers, and an instance graph I. Its transition function γ maps S, L,*

1.	$\mathrm{rl}(t,n,a/p) \rightarrow \mathrm{rl}(t,n',p)$	if $(n,n') \in B$ and $\texttt{name}(n') = a$.
2.	$\mathrm{rl}(t,n,*/p) \rightarrow \mathrm{rl}(t,n',p)$	if $(n,n') \in B$.
3.	$\mathrm{rl}(t,n,a//p) \rightarrow \mathrm{rl}(t,n',p)$	if $(n,n') \in B$ and $\texttt{name}(n') = a$.
4.	$\mathrm{rl}(t,n,a//p) \rightarrow \mathrm{rl}(t,n',*//p)$	if $(n,n') \in B$ and $\texttt{name}(n') = a$.
5.	$\mathrm{rl}(t,n,*//p) \rightarrow \mathrm{rl}(t,n',p)$	if $(n,n') \in B$.
6.	$\mathrm{rl}(t,n,*//p) \rightarrow \mathrm{rl}(t,n',*//p)$	if $(n,n') \in B$.

Figure 3: Read lock propagation rules.

G, I and a newly requested action $a(o,t)$ to a schedule S', a set of locks L', a dependency graph G' and an instance graph I' as follows:

1. *The new instance graph I' is obtained by applying operation o to instance graph I. If the operation fails, then γ is not defined*[3].
2. *The new set of locks L' is obtained by adding to L those locks that are required by the operation o. If one of these locks conflicts with a lock in L of transaction t' then G' is equal to G plus the edge (t',t), otherwise G' is equal to G.*
3. *If G' contains cycles, then γ is not defined*[3].
4. *The schedule S' is S augmented with $a(o,t)$ provided that γ did not become undefined due to the previous points.*
5. *The sending process receives the result of o, if any.*

The execution of the conflict scheduler on a given instance graph I starts with the empty schedule S, the empty set of locks L, an empty graph G and the instance graph I. It receives the actions of S sequentially, and its result is either (1) the output schedule *S, the set of locks L, the dependency graph G and the instance graph I transformed according to each iteration of the conflict scheduler, or (2) undefined.*

A serial schedule equivalent to the output schedule is obtained by sorting the transactions according to their appearance in the topologically sorted dependency graph.

Example 7 *Consider the schedule S_{in} given next and the instance graph d given in Figure 2.*

$$\begin{aligned} S_{\text{in}} \;=\; & \langle a_1(\mathbf{Q}(r,\texttt{document/person//hobby}),t_1),\\ & a_2(\mathbf{Q}(r,\texttt{document}),t_2),\\ & a_3(\mathbf{A}(n_1,\texttt{person}),t_2),\\ & a_4(\mathbf{A}(n_{40},\texttt{hobby}),t_2),\\ & a_5(\mathbf{Q}(n_{24},\texttt{person}),t_1)\rangle \end{aligned}$$

Let the state of the conflict scheduler consist of the empty schedule S, the instance graph I_0 which is equal to d extended with the δ-function which labels all non-root nodes with the empty set, and the edgeless dependency graph G_0. The conflict scheduler accepts individual actions from S_{in} in the sequence given above.

After the first three actions, no conflicts have appeared and the dependency graph G_3 is still without edges. However, action a_4 causes a conflict between locks needed for a_1 and a_4. Thus, G_4 contains an edge from t_1 to t_2. After a_4, no further conflicts appear, so the conflict scheduler finishes and accepts S_{in} as its output schedule.

A serial schedule equivalent to S_{in} is the schedule obtained by first taking all actions of transaction t_1 and then all actions of transaction t_2.

[3]If γ is undefined, the sending process is notified that its action is not accepted, and the scheduler may wait for a new action. Thus deadlocks cannot occur. Livelock issues may be solved using traditional methods.

6 Serializability

In this section, we give a sketch of the serializability proof. The full proof can be found in (Dekeyser et al. 2003*b*). We will first give some preliminary definitions.

Definition 6 *A schedule is said to be* fail-free *if all its operations can be executed without any of them failing. A schedule is said to be a* legal schedule *if (1) it is node correct, (2) fail-free and (3) all sets of locks in the scheduler's state contain only non-conflicting locks.*

It is easy to see that the output schedule of the conflict scheduler is always node correct and fail-free, but is not always legal.

Theorem 1 *Every output schedule of the conflict scheduler is serializable.*

Sketch of the proof. We serialize the schedule by swapping consecutive operations. We assume some linear order on the transaction identifiers that respects the dependency graph at the end of the schedule, i.e., if the edge (t_j,t_i) is in this dependency graph then $t_j < t_i$. If there is a pair (o_i,t_i) and (o_{i+1},t_{i+1}) in the schedule S and $t_i > t_{i+1}$ then we swap them. Note that since $t_i > t_{i+1}$ it follows that the locks of the two operations do not conflict because if they would then the edge (t_i,t_{i+1}) would be in the final dependency graph and therefore $t_i < t_{i+1}$. So we show that if we swap these two operations then it holds for the resulting schedule S' that: (where L_i^S and G_i^S denote the set of locks L and the dependency graph in the state of the scheduler after processing the i-th step of S, and I_i^S denotes the resulting instance graph after this step)

1. the two swapped operations will not fail in S',
2. $G_{i+1}^{S'} \subseteq G_{i+1}^S$,
3. $I_{i+1}^{S'} = I_{i+1}^S$,
4. $L_{i+1}^{S'} = L_{i+1}^S$, and
5. if they exist the results of o_i and o_{i+1} remain the same, and
6. S' is node correct.

We will now prove each of these points.

1. *The two swapped operations will not fail in S'.* Since a query does not change the instance graph we only have to consider the cases that involve no query. For these pairs it holds that if they fail after the swap then one of the operations already failed before the swap or the locks of the two operations conflict.
2. *The dependency graph stays the same or decreases after the swap.* For all pairs that are swapped it holds that if after the swap the locks of the operations conflict with previous locks then also do so before the swap.

3. $I^S_{i+1} = I^{S'}_{i+1}$. Since a query does not change the instance graph we only need to consider four straightforward combinations. In these cases, the fail-free and node-correctness properties of schedules ensure that the operations commute.

4. $L^{S'}_{i+1} = L^S_{i+1}$. Since all operations always request the same locks and $I^S_{i+1} = I^{S'}_{i+1}$ it follows that after the two operations the same locks result from the propagation process.

5. *If they exist the results of o_i and o_{i+1} remain the same.* Without loss of generality we may assume that the additions always return the same result if they do not fail. For the queries it holds that if their results change then the locks it required will conflict with the locks of the other operation.

6. *S is node correct.* This follows trivially because the order of the actions within the same transaction is not changed and as was shown in the previous points the return results of the operations will not change. ∎

Note that this proof shows that the resulting *instance graphs* for S and S' are the same, not just the actual instances as required by the definition of equivalent schedules. Thus, this proof is stronger than strictly necessary.

7 Adding Commits

The serializability proof in the previous section works for schedulers that do not accept commit operations in transactions. This is clearly somewhat impractical since nothing would ever be removed from the instance graph, nor from the dependency graph, nor from the set of locks in the scheduler's state. In this section we solve this issue.

Consider the following description of the commit operation which we now require to appear at the very end of each transaction.

C() If it does not fail, the commit operation does the following things:

1. It removes all the locks in L that are owned by the committing transaction.
2. It deletes from the instance graph I nodes with a non-empty δ function if there are no locks in L' for that node.
3. It deletes from the dependency graph G the node for the committing transaction.

The commit operation fails if in G there is an edge that arrives in the node of the committing transaction.

Failure of the commit operation means that the transaction has to resubmit its commit operation until it succeeds. In practice this would be cumbersome but the scheduler can be redefined such that it remembers every commit and processes it when all transactions that logically appear before[4] the respective transaction have committed.

Theorem 2 *Every output schedule of the conflict scheduler enhanced with commits, is serializable.*

Proof. Let S_c be a schedule with commits and S the corresponding schedule without commits, and assume that S_c is accepted by the scheduler. We first show that if the scheduler accepts S_c then it also accepts S.

Since after every step in the two schedules the resulting actual instance is the same it holds that no operation in S will fail if no operation in S_c fails. If after an operation in S there is a conflict between the locks requested by this operation and those that are already in the instance graph and belong to a transaction that has not yet committed in S_c then these locks will still be present in the instance graph for schedule S_c and so the scheduler will add the same edge to the dependency graph. It follows that the dependency graphs for S are the same as those for S_c except that the nodes for transactions that already committed are removed. Since a commit fails if there is an edge in the dependency graph that leaves from the node for the transaction, it follows that there cannot be cycle in the dependency graphs for S if there isn't one in those for S_c.

Because the scheduler accepts S there is by Theorem 1 a serial schedule S' that is equivalent to S. It is easy to see that if we can extend S' with the commits of S_c to a serial schedule S'_c that is equivalent with S'. Since it also holds that S and S_c are equivalent it follows from the fact the equivalency relation is an equivalence relation, that S'_c and S_c are equivalent. Since S'_c is a serial schedule it follows that S_c is serializable. ∎

The above proof assumes that the conflict scheduler makes use of the PL-PROP path lock scheme. We conjecture that the theorem can also be proven making use of the PL-SAT mechanism, since both methods are equivalent with respect to the conflicts they detect (see (Dekeyser et al. 2003*b*)).

8 Recovery

In the definitions of the Commit (see (Dekeyser & Hidders 2003)) and the Conflict Schedulers we have explicitly mentioned a commit operation which must be used by a transaction to release its locks. We have not explicitly mentioned a *roll-back* or *abort* operation which is traditionally used for recovery from catastrophic events such as power outages. It is straightforward to extend our schedulers with such recovery methods. However, there are certain differences between the two.

Commit Scheduler Since the Commit Scheduler does not permit an action that causes a conflict to proceed, *dirty reads* cannot occur. A transaction is free to issue a roll-back, upon which the scheduler undoes the changes made to the instance graph by that transaction. The roll-back does not affect the other running transactions.

Conflict Scheduler The Conflict Scheduler can permit an action that causes a conflict to proceed, as long as there is no cycle in its dependency graph. Thus, dirty reads are possible and aborting a transaction can affect other transactions. The rule that states when a transaction may issue an abort operation or be rolled-back by the scheduler, is the same as for the commit operation in the Conflict Scheduler: a transaction may abort when all transactions occurring before it in the dependency graph have been rolled-back.

9 Implementation Efforts

We are currently working on the implementation of a Client-Server program that demonstrates our transaction model for XML databases. The Server is a Java application which at start-up reads and parses

[4] With respect to the topological sort of the dependency graph.

XML files and constructs an extended DOM representation. It uses RMI and synchronized methods to implement the path lock propagation scheme. Both the commit scheduler proposed in earlier work and the conflict scheduler proposed here are implemented and work well.

The Clients at this point allow for simple tree querying and manipulation. Each user 'sees' the entire XML document, but may only edit those nodes which he has queried or constructed. An automatic refresh function is used to update the surrounding context. We will extend the Client to become a text editor for LaTeX files encoded in XML documents; authors will then be able to use the software to collaborate on papers. We will also provide a rename and copy operation, which will be simulated using the basic operations provided in this paper.

Importantly, we are currently implementing the traditional hierarchical locking scheme and the new DGLOCK locking protocol proposed in (Grabs et al. 2002) on top of our document server and conflict scheduler. This will allow us to compare throughput of arbitrary sequences of transactions using the diverse locking protocols. The authors of DGLOCK have shown significant performance improvements of their scheme over the traditional flat transaction model. Based on the knowledge that our path locking model decides serializability using the complete instance as opposed to only the DataGuide, we expect path locks to outperform DGLOCK — in the area of degree of concurrency permitted — although this remains to be proven.

10 Conclusion and Future Work

We have introduced a conflict scheduler for XML databases which uses the path locks presented in earlier work. The scheduler works on instance graphs that are an extension of a simplified XPath data model that in effect retains a log of dirty read and write operations. We also presented a comprehensive proof that schedules accepted by the conflict scheduler are guaranteed to be serializable. Finally, we enhanced this result by extending the conflict scheduler to accept commit operations which allow it to become more efficient.

Regarding future theoretical work, we would like to investigate a model in which node identity plays a more important role, enabling operations such as *rename* and *move*. This more elaborate model should also extend the query language and allow for access of a document through an index.

Also, we are currently studying a more fundamental problem in which decidability of serializability of schedules proceeds in an instance independent manner. Preliminary results in this regard have been published in (Dekeyser, Hidders & Paredaens 2003*a*).

Finally, as mentioned in Section 9 we are implementing existing locking schemes on top of our conflict scheduler to obtain experimental results comparing throughput under different locking strategies.

Acknowledgements

We would like to thank University of Antwerp graduate student Joris Raeymaekers for his work on implementing our path lock scheme and scheduler. Our gratitude also goes to Jan Paredaens for some interesting discussions leading up to this work.

References

Abiteboul, S., Buneman, P. & Suciu, D. (1999), *Data on the Web: From Relations to Semistructured Data and XML*, Morgan-Kaufmann, San Francisco.

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

Bernstein, P., Hadzilacos, V. & Goodman, N. (1987), *Concurrency Control and Recovery in Database Systems*, Addison Wesley, Reading, Mass.

Dekeyser, S. (2003), Multiple Query Environment Problems, PhD thesis, University of Antwerp. `http://win-www.ruca.ua.ac.be/u/dekeyser/`.

Dekeyser, S. & Hidders, J. (2002), Path locks for XML document collaboration, *in* 'Proceedings of the Third Web Information Systems Conference (WISE 2002)', Singapore, pp. 105–114.

Dekeyser, S. & Hidders, J. (2003), A commit scheduler for XML databases, *in* 'Proceedings of the Fifth Asia Pacific Web Conference', Xi'an, China.

Dekeyser, S., Hidders, J. & Paredaens, J. (2003*a*), Instance independent concurrency control for semistructured databases, *in* 'Proceedings of the Eleventh Italian Symposium on Advanced Database Systems (SEBD)', Cetraro, Italy.

Dekeyser, S., Hidders, J. & Paredaens, J. (2003*b*), 'A transaction model for XML databases', *World Wide Web Journal* . To appear.

Deutsch, A., Fernández, M. & Suciu, D. (1999), Storing semistructured data with STORED, *in* 'Proceedings ACM SIGMOD', Philadephia.

Eswaran, K., Gray, J., Lorie, R. & Traiger, I. (1976), 'The notions of consistency and predicate locks in a database system', *Communications of the ACM* **19:11**, 624–633.

Fiebig, T., Helmer, S., Kanne, C., Moerkotte, G. et al. (2002), 'Anatomy of a native XML base management system', *VLDB Journal* **11(4)**.

Florescu, D. & Kossmann, D. (1999), 'Storing and querying XML data using an RDBMS', *IEEE Data Engineering Bulletin* **22(3)**, 27–34.

Grabs, T., Bohm, K. & Schek, H. (2002), XMLTM: Efficient transaction management for XML documents, *in* 'Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM 2002)', McLean, Virginia, pp. 142–152.

Gray, J., Putzolo, G. & Traiger, I. (1976), Granularity of locks and degrees of consistency in a shared data base, *in* 'Modeling in Data Base Management Systems', North Holland, Amsterdam.

Helmer, S., Kanne, C. & Moerkotte, G. (2003), Lock-based protocols for cooporation on XML documents, Technical Report 06/2003, University of Mannheim, Germany. Available at: http://pi3.informatik.uni-mannheim.de/publications/TR-03-006.ps.

Hye Choi, E. & Kanai, T. (2003), XPath-based concurrency control for XML data, *in* 'Proceedings of the 14th Data Engineering Workshop (DEWS'03)', Katayamazu, Japan.

Jea, K., Chen, S. & Wang, S. (2002), Concurrency control in XML document databases: XPath locking protocol, *in* 'Proceedings of the 9th International Conference on Parallel and Distributed Systems (ICPADS 2002)', pp. 551–56.

McHugh, J. & Widom, J. (1999), Query optimization for XML, *in* 'Proceedings of VLDB'99', Edinburg, Scotland, pp. 315–326.

Papadimitriou, C. (1986), *The Theory of Database Concurrency Control*, Computer Science Press, Rockville, MD.

Silberschatz, A. & Kedem, Z. (1980), 'Consistency in hierarchical database systems', *Journal of the ACM* **27(1)**, 72–80.

Weikum, G. & Vossen, G. (2002), *Transactional Information Systems*, Morgan Kaufmann. ISBN: 1-55860-508-8.