A Commit Scheduler for XML Databases

Stijn Dekeyser and Jan Hidders

University of Antwerp

Abstract. The hierarchical and semistructured nature of XML data may cause complicated update-behavior. Updates should not be limited to entire document trees, but should ideally involve subtrees and even individual elements. Providing a suitable scheduling algorithm for semistructured data can significantly improve collaboration systems that store their data — e.g. word processing documents or vector graphics — as XML documents. In this paper we improve upon earlier work (see [5]) which presented two equivalent concurrency control mechanisms based on Path Locks. In contrast to the earlier work, we now provide details regarding the workings of a commit scheduler for XML databases which uses the path lock conflict rules. We also give a comprehensive proof of serializability which enhances and clarifies the ideas in our previous work.

1 Introduction

Semistructured data [1] is an important topic in Information Systems research that has been studied extensively — especially regarding query languages — in the past and which has regained importance due to the popularity of XML. 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 [8] has been extensively studied in the context of traditional database management systems [2, 6, 7]. It is possible to re-use these results for providing concurrency control in semistructured databases. However, as we have shown in earlier work [5], the traditional solutions we mentioned — while guaranteeing serializability — do not allow a sufficient degree of concurrency; i.e., they are too restrictive.

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

In previous work, 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 and path locks propagation. For both systems, we introduced conflict rules and analyzed their complexity. We showed that the conflict rules were sufficient to ensure two actions from different transactions can be swapped if no conflict occurs.

In this paper, however, we introduce the scheduler that makes use of the path locks and the conflict rules. The second contribution of this paper is the inclusion of a comprehensive proof of serializability, which was lacking in earlier work. The proof also enhances and clarifies the ideas presented earlier.

2 Data Model and DML

The data model we assume for XML documents is a simplification of the standard XPath data model [3] and essentially node-labeled trees. However, for the purpose of locking we allow also acyclic graphs. We label nodes with a set of transaction identifiers to indicate that the node has been deleted by these transactions.

Definition 1 (Instance graph, actual instance). *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 $\nu : N \rightarrow E$ and with sets of transaction identifiers by $\delta : N \rightarrow 2^T$. 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.*

The query language is based on a subset of XPath expressions as defined by the following grammar.

$$\mathcal{P} ::= \mathcal{F} \mid \mathcal{P}/\mathcal{F} \mid \mathcal{P}//\mathcal{F}$$
$$\mathcal{F} ::= E \mid *$$

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

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

Definition 2 (Operations on the instance graph). *The following four operations are defined on an instance graph.*

$\mathbf{A}(n, a)$ *This update operation 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]*.*

D(n) *This update operation 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.*

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 p.*

C$()$ *This update operation removes all nodes n from the instance graph with $\delta(n)$ containing the identifier of the executing transaction. The operation returns no result. The operation fails if in the resulting instance graph the actual instance is not a tree*[1] *with root r.*

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

Definition 3 (Action, Transaction, and Schedule). *An* action *is a pair (t, o) where t is a transaction identifier and o is one of the operations given in Def. 2. A* transaction *is a finite list of actions having the same transaction identifier and in which there is exactly one commit operation (the last pair). 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.*

Following tradition, two schedules are equivalent if (1) one is a permutation of the other, (2) the resulting instance graph 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.

3 Path Locks

We now turn to the locking scheme that is used by the scheduler to ensure serializability.

We start with the definition of the read locks. A *read lock* is defined as a tuple $\text{rl}(t, n, p)$ where t is a transaction identifier, n is the node identifier

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

in the instance graph for which the lock holds and p is a 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 the next table. The process of read-lock propagation is applied until no more new read locks are added. For more information on the Path Lock Propagation mechanism, we refer the reader to [5].

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 $(n, n') \in B$.

Fig. 1. Read lock propagation rules.

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.

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. 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 The Commit Scheduler

In this section we detail the working of the commit scheduler. The term is based on the theoretical notion of *commit serializability* [8]. Thus, a commit scheduler guarantees that the schedules it accepts are serializable.

Definition 4 (Commit Scheduler). *The* commit 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 and an instance graph I. Its transition function γ maps S, I and a newly requested action $a(o,t)$ to a schedule S', a set of locks L' 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*[2]*.*
2. *For update and query operations, the set of locks L' is obtained by adding to L the locks required by the operation o. For the commit operation, L' is obtained by removing all locks from L which are owned by the transaction that commits, plus those locks on the nodes that are now deleted from the instance graph.*
   If L' contains conflicting locks, then γ is not defined[2]*.*
3. *The schedule S' is S augmented with $a(o,t)$ provided that γ did not become undefined due to the previous points.*
4. *The sending process receives the result of o, if any.*

The execution of the commit scheduler on a given instance graph I starts with the empty schedule S, the empty set of locks L, 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, and the instance graph I transformed according to each iteration of the commit scheduler, or (2) undefined.*

5 Serializability

In this section, we give a sketch of the serializability proof. The full proof can be found in the Technical Report [4]. We will first give some preliminary definitions.

Definition 5 (Legal and Fail-free Schedules). *A schedule is said to be* fail-free *if all its operations can be executed without any of them*

[2] If γ is undefined, the sending process is notified that its action is not accepted, and the scheduler waits for a new action. Thus deadlocks cannot occur.

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 compatible locks.*

It is easy to see that the output schedule of the scheduler is always a legal schedule.

Theorem 1. *Every legal schedule is serializable.*

Sketch of the proof. We presume some ordering on the transaction identifiers used in S such that $t_i < t_j$ if the commit of t_i preceeds the commit of t_j in S or there is a commit of t_i but not a commit of t_j in S. We serialize the schedule by repeatedly swapping two consecutive actions (t_i, o_i) and (t_{i+1}, o_{i+1}) if $t_i \neq t_j$ and $t_j < t_i$. It is easy to see that if there are no more such pairs then the schedule is serialized. It can also be shown that after a swap of such a pair the result will be an equivalent legal schedule if the schedule before the swap is a legal schedule. Assume that S is a legal schedule and we swap two consecutive actions (t_i, o_i) and (t_{i+1}, o_{i+1}) in S and $t_i \neq t_j$ and o_i is not a commit, then we prove that the following holds: (1) the two swapped operations will not fail in S', (2) all locks in $L_i^{S'}$ are compatible, (3) $I_{i+1}^{S'} = I_{i+1}^{S}$, (4) if they exist the results of o_i and o_{i+1} remain the same, (5) $L_{i+1}^{S'} \subseteq L_{i+1}^{S}$, and (6) S' is node correct. It follows from these points that S' is equivalent with S, fail-free and in all sets $L_j^{S'}$ there are no incompatible locks, i.e., S' is legal.

References

1. S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan-Kaufmann, San Francisco, 1999.
2. P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, Reading, Mass., 1987.
3. J. Clark and S. DeRose. XML Path Language (XPath). *W3C Recommendation*, November 1999.
4. S. Dekeyser and J. Hidders. A path-lock scheduler for XML databases. Technical Report 02-13, University of Antwerp, 2002. ftp://win-ftp.uia.ac.be/pub/dekeyser/scheduler.ps.
5. S. Dekeyser and J. Hidders. Path locks for XML document collaboration. In *Proceedings of the Second WISE Conference*, 2002.
6. J. Gray. Notes on database operating systems. In *Operating Systems: an Advanced Course*. Springer-Verlag, New York, 1978.
7. C. Papadimitriou. *The Theory of Database Concurrency Control*. Computer Science Press, Rockville, MD, 1986.
8. G. Weikum and G. Vossen. *Transactional Information Systems*. Morgan Kaufmann, 2002. ISBN: 1-55860-508-8.