Instance-Independent View Serializability for Semistructured Databases

Stijn Dekeyser
University of Southern Queensland
dekeyser@usq.edu.au

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

Jan Paredaens
University of Antwerp
jan.paredaens@ua.ac.be

Roel Vercammen*
University of Antwerp
roel.vercammen@ua.ac.be

February 1, 2008

Abstract

Semistructured databases require tailor-made concurrency control mechanisms since traditional solutions for the relational model have been shown to be inadequate. Such mechanisms need to take full advantage of the hierarchical structure of semistructured data, for instance allowing concurrent updates of subtrees of, or even individual elements in, XML documents. We present an approach for concurrency control which is document-independent in the sense that two schedules of semistructured transactions are considered equivalent if they are equivalent on all possible documents. We prove that it is decidable in polynomial time whether two given schedules in this framework are equivalent. This also solves the view serializability for semistructured schedules polynomially in the size of the schedule and exponentially in the number of transactions.

1 Introduction

In previous work [5, 6, 7] we have shown that traditional concurrency control [21] mechanisms for the relational model [2, 11, 19, 20] are inadequate to capture the complicated update behavior that is possible for semistructured databases. Indeed, when XML documents are stored in relational databases, their hierarchical structure becomes invisible to the locking strategy used by the database management system.

*Roel Vercammen is supported by IWT – Institute for the Encouragement of Innovation by Science and Technology Flanders, grant number 31581.

In general two actions, on two different nodes of a document tree, that are completely 'independent' from each other, cannot cause a conflict, even if they are updates. Changing the spelling of the name of one of the authors of a book and adding a chapter to the book cannot cause a conflict for instance. Most classical concurrency control mechanisms, when applied in a naive way to semistructured data, will not allow such concurrent updates. This consideration is the main reason why the classical approaches seem to be inadequate as a concurrency control mechanism for semistructured data.

Most of the work on concurrency control for XML and semistructured data is based on the observation that the data is usually accessed by means of XPath expressions. Therefore it is suggested in [5] to use a simplified form of XPath expressions as locks on the document such that precisely all operations that change the result of the expression are no longer allowed. Two alternatives for conflict-checking are proposed, one where path locks are propagated down the XML tree and one where updates are propagated up the tree, which both have their specific benefits. This approach is extended in [7] where a commit-scheduler is defined and it is proved that the schedules it generates are serializable. Finally in [9] an alternative conflict-scheduler is introduced that allows more schedules than the previously introduced commit-scheduler.

A similar approach is taken in [4] 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 [17] where locks are derived from the path expressions and a protocol for these locks is introduced that guarantees serializability.

Several locking protocols that are not based on path expressions but on DOM operations are introduced in [14, 15]. 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. A similar approach, but extended with the aspect of multi-granularity locking, is presented in [12, 13]. This approach seems more suitable for hierarchical data like semistructured data and XML. However, such mechanisms will often allow less concurrency than a path based locking protocol would.

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 [10] where the locks are associated with the nodes in a DataGuide, which is usually much smaller than the document. However, this protocol does not guarantee serializability and allows phantoms.

For all the approaches above it holds that the concurrency control mechanisms are somehow dependent upon the document. In most cases this means that if the document gets very large then the overhead may also become very large. This paper investigates the possibilities of a document-independent concurrency control mechanism. It extends the preliminary results on this subject that were presented in [8].

The total behavior of the processes that we consider in this paper is straightforward: each cooperating process produces a transaction of atomic actions that are queries or updates on the actual document. The transactions are interleaved by the scheduler and the resulting schedule has to be equivalent with a serial schedule. Two schedules on the same set of transactions are called equivalent iff **for each possible input document** they represent the same transformation and each query gives the same result in both schedules. This is a special definition of view equivalency, which we will use to decide view serializability [3] for a schedule.

Note that we consider view serializability, as opposed to conflict serializability. As we will show later on, conflict serializability, which might be more interesting from a computational point of view, will allow less schedules to be serialized and hence can be too restrictive.

The updates that we consider are very primitive: the addition of an edge of the document tree and the deletion of an edge. Semantically the addition is only defined if the added edge does not already exist in the document tree. Analogously the deletion is only defined if the deleted edge exists. A more general semantics, that does not include this constraint, can be easily simulated by adding first some queries.

There are some schedules for which the result is undefined for all document trees (e.g., a schedule consisting of two consecutive deletions of the same edge). These schedules are meaningless and are called inconsistent. Hence a schedule is consistent if there exists at least one document tree on which its application is defined. We prove that the consistenty of schedules is polynomially decidable.

In order to tackle the equivalence of schedules and transactions we first consider schedules without queries, and as such we have only to focus on the transformational behavior of the schedules. We will see that, contrary to the relational model, the swapping of the actions cannot help us in detecting the equivalence of two schedules. We prove that the equivalence of queryless schedules is also polynomially decidable, and that view serializability is exponentially decidable in the number of transactions and polynomially in the number of operations. Finally we generalize the results above for general schedules over the same set of transactions.

The paper contains a number of theoretical results on which the algorithms are based. The algorithms are a straightforward consequence of the given proofs or sketches. The complete proofs are given in [16].

The paper is structured as follows: Section 2 defines the data model, the operations and the semistructured schedules. Section 3 studies the consistency of schedules without queries. In Section 4 we study the equivalence and the view serializability problem for these queryless schedules. In Section 5 we generalize these results for consistent schedules.

2 Data Model and Operations

The data model we use is derived from the classical data model for semistructured data [1]. We consider directed, unordered trees in which the edges are labelled.

Consider a fixed universal set of nodes $\mathcal{N}$ and a fixed universal set of edge labels $\mathcal{L}$ not containing the symbol /.

Definition 1. *A* graph *is a tuple* (N, E) *with* $N \subseteq \mathcal{N}$ *and* $E \subseteq N \times \mathcal{L} \times N$. *A* document tree *(*DT*)* T *is a tuple* (N, E, r) *such that* (N, E) *is a graph that represents a tree with root* r. *The edges are directed from the parent to the child.*

```
<document id="0">
<person id="1",age="55",
 spouse="2">
  <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>
</person>
<person id="2", age="43">
  <name>Mary</name>
  <addr>Parklane 7</addr>
  <hobby>painting</hobby>
</person>
</document>
```

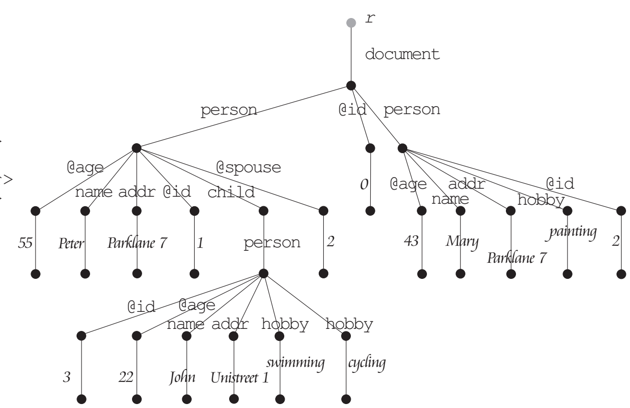

Figure 1: A fragment of an XML document and its DT representation.

Example 1. *Figure 1 shows a fragment of an XML document and its* DT *representation.*

This data model closely mimics the XML data model as illustrated in the next example. We remark, however, the following differences:

- **order:** Siblings are not ordered. This is not crucial, as an ordering can be simulated by using a skewed binary DT.
- **attributes:** Attributes, like elements, are represented by edges labeled by the name of the attributes (started with a @). The difference is that in this data model an element may contain several attributes of the same name.
- **labels:** Labels represent not only tag names and attribute names, but also values and text.
- **text:** Unlike in XML, it is possible for several text edges to be adjacent to each other.

A *label path* is a string of the form $l_1/\ldots/l_m$ with $m \geq 0$ and every l_i an edge label in $\mathcal{L}$. Given a path $p = ((n_1, l_1, n_2), \ldots, (n_m, l_m, n_{m+1}))$ in a graph G, the *label path of* p, denoted $\bar{\lambda}_T(p)$ (or $\bar{\lambda}(p)$ when T is subsumed) is the string $l_1/\ldots/l_m$.

Processes working on document trees do so in the context of a general programming language that includes an interface to a document server which manages transactions on documents. The process generates a list of operations that will access the document. In general there are three types of operations: the query, the addition and the deletion. The input to a query operation will be a node and a simple type of path expression, while the result of the invocation of a query operation will be a set of nodes. The programming language includes the concepts of sets, and has constructs to iterate over their entire contents. The input to an addition or a deletion will be an edge. The result of an addition or a deletion will be a simple transformation of the original tree into a new tree. If the result would not be a tree anymore it is not defined.

We now define the path expressions and the query operations, subsuming a given DT T.

The syntax of path expressions[1] is given by $\mathcal{P}$:

$$
\begin{array}{rcl}
\mathcal{P} & ::= & pe_\epsilon \mid \mathcal{P}^+ \\
\mathcal{P}^+ & ::= & \mathcal{F} \mid \mathcal{P}^+/\mathcal{F} \mid \mathcal{P}^+//\mathcal{F} \\
\mathcal{F} & ::= & * \mid \mathcal{L}
\end{array}
$$

The set $\mathbf{L}(pe)$ of label paths represented by a path expression pe is defined as follows:

$$
\begin{array}{rcl}
\mathbf{L}(pe_\epsilon) & = & \{\epsilon\} \\
\mathbf{L}(*) & = & \mathcal{L} \\
\mathbf{L}(l) & = & \{l\} \\
\mathbf{L}(pe/f) & = & \mathbf{L}(pe) \cdot \{/\} \cdot \mathbf{L}(f) \\
\mathbf{L}(pe//f) & = & \mathbf{L}(pe) \cdot \{/\} \cdot (\mathcal{L} \cdot \{/\})^* \cdot \mathbf{L}(f)
\end{array}
$$

Let n be an arbitrary node of T and pe a path expression. We now define the three kinds of operations: the query, the addition and the deletion.

Definition 2. *The query operation* $\text{query}(n, pe)$ *returns a set of nodes, and is defined as follows:*

- $\text{query}(n, pe)$ *with* $n \in \mathcal{N}$ *and* $pe \in \mathcal{P}$. *The result of a query on a* DT T *is defined as* $\text{query}(n, pe)[T] = \{n' \in N \mid \exists p$ a path in T from n to n' with $\bar{\lambda}(p) \in \mathbf{L}(pe)\}$.

The update operations $\text{add}(n, l, n')$ *and* $\text{del}(n, l, n')$ *return no value but transform a* DT $T = (N, E, r)$ *into a new* DT $T' = (N', E', r)$*:*

[1] Remark that path expressions form a subset of XPath expressions.

- $\mathrm{add}(n, l, n')$ *with* $n, n' \in \mathcal{N}$ *and* $l \in \mathcal{L}$*. The resulting* $T' = \mathrm{add}(n, l, n')[T]$ *is defined by* $E' = E \cup \{(n, l, n')\}$ *and* $N' = N \cup \{n'\}$*. If the resulting* T' *is not a document tree anymore or* (n, l, n') *was already in the document tree then the operation is undefined.*

- $\mathrm{del}(n, l, n')$ *with* $n, n' \in \mathcal{N}$ *and* $l \in \mathcal{L}$*. The resulting* $T' = \mathrm{del}(n, l, n')[T]$ *is defined by* $E' = E - \{(n, l, n')\}$ *and* $N' = N - \{n'\}$*. If the resulting* T' *is not a document tree anymore or* (n, l, n') *was not in the document tree then the operation is undefined.*

Note that the operations explicitly contain the nodes upon which they work. As we will explain in Section 4 this is justified by the fact that the scheduler decides at run time whether an operation is accepted or not.

We now give some straightforward definitions of schedules and their semantics.

Definition 3. *An* action *is a pair* (o, t)*, where* o *is one of the three operations* $\mathrm{query}(n, pe)$*,* $\mathrm{add}(n, l, n')$ *and* $\mathrm{del}(n, l, n')$ *and* t *is a transaction identifier. A* transaction *is a sequence of actions with the same transaction identifier. A* schedule *over a set of transactions is an interleaving of these transactions. The size* n_S *of a schedule* S *is the length of its straightforward encoding on a Turing tape*[2]*.*

We can apply *a schedule* S *on a* DT T*. The result of such an application is*

- *for each query in* S*, the result of this query.*

- *the* DT *that results from the sequential application of the actions of* S*; this* DT *is denoted by* $S[T]$

If some of these actions are undefined the application is undefined. Two schedules are equivalent iff they are defined on the same non-empty set of DT*s and on each of these* DT*s both schedules have the same result. The definition of serial and serializable schedules is straightforward.*

Since a transaction is a special case of a schedule all the definitions on schedules also apply on transactions.

Note that the equivalence of schedules and transactions is a document-independent definition. Let
$T_1 = (\{n_1, n_2\}, \{(n_1, l_2, n_2)\}, n_1)$,
$T_2 = (\{n_1, n_2\}, \{(n_1, l_1, n_2)\}, n_1)$,
$T_3 = (\{n_1\}, \emptyset, n_1)$ be three DTs and let
$S_1 = (\mathrm{add}(n_2, l_2, n_3), t_1), (\mathrm{query}(n_1, l_1/l_2), t_2)$,
$S_2 = (\mathrm{query}(n_1, l_1/l_2), t_2), (\mathrm{add}(n_2, l_2, n_3), t_1)$ be two schedules.
S_1 and S_2 are equivalent on T_1, they are not equivalent on T_2 and their application is undefined on T_3.
Let S_3 be the empty schedule and

[2] We assume that nodes can be encoded in $O(1)$-space

$S_4 = (\mathrm{add}(n_1, l_1, n_2), t_1), (\mathrm{del}(n_1, l_1, n_2), t_2)$.
S_3 and S_4 are not equivalent although they are equivalent on many DTs.

We will later on use the definition of equivalence to define serializability. In this paper we study view serializability, which is less restrictive than conflict serializability. We illustrate this claim by introducing informally a scheduling mechanism for generating conflict serializable schedules. A possible approach for this is to have a locking mechanism where operations can get locks, and in which a new operation of a certain process will only be allowed if it does not require locks that conflict with locks required by earlier operations. Because operations with non-conflicting locks can be commuted, any schedule that is allowed by such a scheduler can be serialized. The following example shows, however, that the reverse does not hold: Indeed, the next schedule

$$\begin{aligned} S = {} & (\mathrm{add}(r, l_1, n_1), t_1),\ (\mathrm{del}(r, l_1, n_1), t_2), \\ & (\mathrm{add}(r, l_2, n_2), t_2),\ (\mathrm{del}(r, l_2, n_2), t_2), \\ & (\mathrm{add}(r, l_2, n_2), t_1),\ (\mathrm{del}(r, l_2, n_2), t_1). \end{aligned}$$

is consistent since it is defined on $T = (\{r\}, \emptyset, r)$. Furthermore it is serializable, and the equivalent serial schedules are

$$\begin{aligned} S_1 = {} & (\mathrm{add}(r, l_1, n_1), t_1),\ (\mathrm{add}(r, l_2, n_2), t_1), \\ & (\mathrm{del}(r, l_2, n_2), t_1),\ (\mathrm{del}(r, l_1, n_1), t_2), \\ & (\mathrm{add}(r, l_2, n_2), t_2),\ (\mathrm{del}(r, l_2, n_2), t_2) \\ S_2 = {} & (\mathrm{del}(r, l_1, n_1), t_2), (\mathrm{add}(r, l_2, n_2), t_2), \\ & (\mathrm{del}(r, l_2, n_2), t_2),\ (\mathrm{add}(r, l_1, n_1), t_1), \\ & (\mathrm{add}(r, l_2, n_2), t_1),\ (\mathrm{del}(r, l_2, n_2), t_1). \end{aligned}$$

but we cannot go from S to S_1 nor to S_2 only by swapping with consistent intermediate schedules. This illustrated that an approach based on conflict serializability can be too strict.

3 Consistency of Queryless Schedules

A schedule is called *queryless* (QL) iff it contains no queries. Because of the way that operations can fail it is possible that the application of a certain transaction is not defined for any document tree. We are not interested in such transactions. We call a transaction t *consistent* iff there is at least one DT T with $t[T]$ defined.

Example 2. *The next transaction is consistent:*
$(\mathrm{add}(r, l_1, n_1), t_1)$, $(\mathrm{del}(r, l_1, n_1), t_1)$, $(\mathrm{add}(r, l_2, n_2), t_1)$,
$(\mathrm{del}(r, l_2, n_2), t_1)$, $(\mathrm{add}(r, l_2, n_2), t_1)$, $(\mathrm{del}(r, l_2, n_2), t_1)$.
Note, however, that there are DT*s on which this transaction is undefined. For example, if T contains an edge (r, l_3, n_1), then $t_1[T]$ is undefined, since the application of the first action of t_1 is undefined.*
The next transaction is inconsistent:
$(\mathrm{add}(n_1, l_1, n), t_1)$, $(\mathrm{add}(n_2, l_2, n), t_1)$.

We call a schedule S *consistent* iff there is at least one DT T with $S[T]$ defined. Remark that there are consistent schedules that cannot be serializable because they contain an inconsistent transaction. For instance, the consistent schedule $S = (\text{add}(r, l_1, n_1), t_1)$, $(\text{del}(r, l_1, n_1), t_2)$, $(\text{add}(r, l_1, n_1), t_1)$ is defined on $T = (\{r\}, \emptyset, r)$, and hence is not serializable, because every equivalent serial QL schedule would be undefined (since the transaction t_1 is not consistent). Transaction t_1 has the property that all QL schedules over a set of transactions that contain t_1 are non-serializable.

Note that the definition of consistent QL schedule is document-independent. It is clear that we are only interested in consistent transactions and schedules. Remark also that if two QL schedules are equivalent then they are both consistent. This equivalence relation is defined on the set of consistent QL schedules.

We will characterize the consistent QL schedules and prove that this property is decidable. For this purpose we will first attempt to characterize for which document trees a given consistent QL schedule S is defined, and what the properties are of the document trees that result from a QL schedule. We do this by defining the sets $N_I^{min}(S)$, $N_I^{max}(S)$, $E_I^{min}(S)$ and $E_I^{max}(S)$, whose informal meaning is respectively the set of nodes that are required in the input DTs on which S is defined, the set of nodes that are allowed, the set of edges that are required and the set of edges that are allowed. In the same way we define the sets $N_O^{min}(S)$, $N_O^{max}(S)$, $E_O^{min}(S)$ and $E_O^{max}(S)$ taking into account the output DTs.

Definition 4. *Let S be a QL schedule. $\phi_S(n, o)$ ($\phi_S((m, l, n), o)$) indicates that the first occurrence of the node n (the edge (m, l, n)) in the schedule S has the form of the operator o.* [3] *$\lambda_S(n, o)$ ($\lambda_S((m, l, n), o)$) indicates that the last occurrence of the node n (the edge (m, l, n)) in the QL schedule S has the form of the operation o. We define the sets $N_I^{min}(S)$, $N_I^{max}(S)$, $E_I^{min}(S)$ and $E_I^{max}(S)$, and the sets $N_O^{min}(S)$, $N_O^{max}(S)$, $E_O^{min}(S)$ and $E_O^{max}(S)$ as in Figure 2.*
A DT T is called a basic input tree (basic output tree) of S iff it contains all the nodes of $N_I^{min}(S)$ ($N_O^{min}(S)$), only nodes of $N_I^{max}(S)$ ($N_O^{max}(S)$), all the edges of $E_I^{min}(S)$ ($E_O^{min}(S)$) and only edges of $E_I^{max}(S)$ ($E_O^{max}(S)$).

Consider $S = (\text{add}(n_1, l_1, n_2), t_1)$, $(\text{del}(n_4, l_2, n_3), t_2)$, $(\text{del}(n_1, l_1, n_4), t_3)$ then

$$
\begin{aligned}
N_I^{min}(S) &= \{n_1, n_3, n_4\}\\
N_I^{max}(S) &= \mathcal{N} - \{n_2\}\\
E_I^{min}(S) &= \{(n_4, l_2, n_3), (n_1, l_1, n_4)\}\\
E_I^{max}(S) &= E_I^{min}(S) \cup \{(m, l, n) \in \\
&\qquad \mathcal{N} \times \mathcal{L} \times \mathcal{N} \mid m, n \neq n_2, n_3, n_4\}\\
N_O^{min}(S) &= \{n_1, n_2\}\\
N_O^{max}(S) &= \mathcal{N} - \{n_3, n_4\}\\
E_O^{min}(S) &= \{(n_1, l_1, n_2)\}\\
E_O^{max}(S) &= E_O^{min}(S) \cup \{(m, l, n) \in \\
&\qquad \mathcal{N} \times \mathcal{L} \times \mathcal{N} \mid m, n \neq n_2, n_3, n_4\}
\end{aligned}
$$

[3] For example, $\phi_S(n_2, \text{add}(r, l_2, n_2))$ holds in the consistent QL schedule in Example 2 above.

$$
\begin{aligned}
&N_I^{min}(S) = \{m \mid \phi_S(m, \text{add}(m,l,n))\} \cup \{m \mid \phi_S(m, \text{del}(m,l,n))\} \cup \{n \mid \phi_S(n, \text{del}(m,l,n))\}\\
&N_I^{max}(S) = \mathcal{N} - \{n \mid \phi_S(n, \text{add}(m,l,n))\}\\
&E_I^{min}(S) = \{(m,l,n) \mid \phi_S((m,l,n), \text{del}(m,l,n))\}\\
&E_I^{max}(S) = E_I^{min}(S) \cup \{(m,l,n) \mid \text{ no } (m_1,l_1,m) \text{ nor } (m_1,l_1,n) \text{ occurs in } S\}\\
&N_O^{min}(S) = \{m \mid \lambda_S(m, \text{del}(m,l,n))\} \cup \{m \mid \lambda_S(m, \text{add}(m,l,n))\} \cup \{n \mid \lambda_S(n, \text{add}(m,l,n))\}\\
&N_O^{max}(S) = \mathcal{N} - \{n \mid \lambda_S(n, \text{del}(m,l,n))\}\\
&E_O^{min}(S) = \{(m,l,n) \mid \lambda_S((m,l,n), \text{add}(m,l,n))\}\\
&E_O^{max}(S) = E_O^{min}(S) \cup \{(m,l,n) \mid \text{ no } (m_1,l_1,m) \text{ nor } (m_1,l_1,n) \text{ occurs in } S\}
\end{aligned}
$$

Figure 2: The Definition of the basic input and output sets.

We will prove in Theorem 1 that the application of a consistent schedule S is defined on each basic input tree of S.

Although $N_I^{max}(S)$, $E_I^{max}(S)$, $N_O^{max}(S)$ and $E_O^{max}(S)$ are in general infinite, they can be represented in a finite way: $N_I^{max}(S)$ by $\{n \mid \phi_S(n, \text{add}(m,l,n))\}$, $E_I^{max}(S)$ by $E_I^{min}(S) \cup \{n \mid \text{ there is a } (m_1,l_1,n) \text{ that occurs in } S\}$, $N_O^{max}(S)$ by $\{n \mid \lambda_S(n, \text{del}(m,l,n))\}$, $E_O^{max}(S)$ by $E_O^{min}(S) \cup \{n \mid \text{ there is a } (m_1,l_1,n) \text{ that occurs in } S\}$.

Lemma 1. *Let S be a schedule with size n_S. $N_I^{min}(S), N_I^{max}(S), E_I^{min}(S), E_I^{max}(S)$, $N_O^{min}(S), N_O^{max}(S), E_O^{min}(S)$ and $E_O^{max}(S)$ can be calculated in $O(n_S.log(n_S))$-time and in $O(n_S)$-space. For each of these sets and for any node or edge it is decidable in $O(n_S)$-time and $O(log(n_S))$-space whether the node or edge is in the set.*

Proof. (Sketch) We can decide whether a node or an edge is in one of the basic input or output sets by examining the actions of the schedule S. □

When a QL schedule is inconsistent this is always because two operations in the QL schedule interfere, as for example the two operations in the inconsistent transaction of Example 2: $(\text{add}(n_1,l_1,n),t_1)$ and $(\text{add}(n_2,l_2,n),t_1)$. If these two operations immediately follow each other then at least one of them will always fail. However, if between them we find the action $\text{del}(n_1,l_1,n)$ then this does no longer hold. The following definition attempts to identify such pairs of interfering operations and states which operations we should find between them to remove the interference.

Definition 5. *A QL schedule fulfills the* C-condition *iff*

1. *If* $\text{add}(n,l_1,n_1)$ *and* $\text{add}(n_2,l_2,n)$ *appear in that order in* S *then* $\text{del}(n,l_1,n_1)$ *appears between them.*
2. *If* $\text{add}(n_1,l_1,n)$ *and* $\text{add}(n_2,l_2,n)$ *appear in that order in* S *then* $\text{del}(n_1,l_1,n)$ *appears between them.*
3. *If* $\text{add}(n,l_1,n_1)$ *and* $\text{del}(n_2,l_2,n)$ *appear in that order in* S *then* $\text{del}(n,l_1,n_1)$ *appears between them.*

4. *If* $\mathrm{add}(n_1, l_1, n)$ *and* $\mathrm{del}(n, l_2, n_2)$ *appear in that order in* S *then* $\mathrm{add}(n, l_2, n_2)$ *appears between them.*

5. *If* $\mathrm{add}(n_1, l_1, n)$ *and* $\mathrm{del}(n_2, l_2, n)$ *appear in that order in* S *and* $(n_1, l_1) \neq (n_2, l_2)$ *then* $\mathrm{del}(n_1, l_1, n)$ *appears between them.*

6. *If* $\mathrm{del}(n, l_1, n_1)$ *and* $\mathrm{add}(n_2, l_2, n)$ *appear in that order in* S *then some* $\mathrm{del}(n_3, l_3, n)$ *appears between them.*

7. *If* $\mathrm{del}(n_1, l_1, n)$ *and* $\mathrm{add}(n, l_2, n_2)$ *appear in that order in* S *then some* $\mathrm{add}(n_3, l_3, n)$ *appears between them.*

8. *If* $\mathrm{del}(n_1, l_1, n)$ *and* $\mathrm{del}(n, l_2, n_2)$ *appear in that order in* S *then some* $\mathrm{add}(n_3, l_3, n)$ *appears between them.*

9. *If* $\mathrm{del}(n_1, l_1, n)$ *and* $\mathrm{del}(n_2, l_2, n)$ *appear in that order in* S *then* $\mathrm{add}(n_2, l_2, n)$ *appears between them.*

The following theorem establishes the relationship between consistency, basic input trees and the C-condition.

Theorem 1. *The following conditions are equivalent for a QL schedule* S*:*

1. *there is a basic input tree of* S *and the application of* S *is defined on each basic input tree of* S*.*

2. *there is a basic input tree of* S *on which the application of* S *is defined;*

3. S *is consistent;*

4. S *fulfills the C-condition;*

5. *there is a tree on which the application of* S *is defined and all trees on which the application of* S *is defined are basic input trees of* S*.*

Proof. (Sketch) Clearly $1 \to 2 \to 3 \to 4$ and $5 \to 3$. We prove that 4 implies 1. First we prove that there is a basic input tree for which S is defined. Then we prove that the application of S is defined on each basic input tree of S by induction on the length of S. Finally 3 implies 5. Indeed, let S be defined on T, where T is not a basic input tree of S. T does not satisfy one of the four conditions of Definition 4. In each case this yields a contradiction. □

Corollary 1. *It is decidable whether a QL schedule or a transaction is consistent in* $O(n_S^3)$*-time and* $O(n_S)$*-space.*

Proof. (Sketch) This follows from the decidability of the C-condition and Theorem 1. □

For the basic input and output sets we can derive the following property:

Property 1. *If* S *is a consistent QL schedule then* $E_I^{min}(S)$ *and* $E_O^{min}(S)$ *are forests.*

By $\mathrm{ADD}(S)$ we denote the set of edges that are added by the QL schedule S, i.e., they are added without being removed again afterwards, and by $\mathrm{DEL}(S)$ we denote the set of edges that are deleted by the QL schedule S, i.e., they are deleted without being added again afterwards.

Definition 6. *Let S be a consistent QL schedule. We denote*

$$\mathrm{ADD}(S) = \{(m,l,n) \mid \lambda_S((m,l,n), \mathrm{add}(m,l,n))\}$$
$$\mathrm{DEL}(S) = \{(m,l,n) \mid \lambda_S((m,l,n), \mathrm{del}(m,l,n))\}$$

We call $\mathrm{ADD}(S)$ *the addition set of S and* $\mathrm{DEL}(S)$ *its deletion set.*

Remark that two consistent QL schedules with the same ADD and DEL are not necessarily equivalent. Indeed $S_1 = (\mathrm{del}(n_1,l_1,n_2),t_2)$ and $S_2 = (\mathrm{add}(n_1,l_1,n_2),t_1)$, $(\mathrm{del}(n_1,l_1,n_2),t_2)$ are not equivalent although $\mathrm{ADD}(S_1) = \mathrm{ADD}(S_2)$ and $\mathrm{DEL}(S_1) = \mathrm{DEL}(S_2)$.

Lemma 2. *Let S be a consistent QL schedule and T be a basic input tree of S. $S[T] = T \cup \mathrm{ADD}(S) - \mathrm{DEL}(S)$ is a basic output tree*[4].

Proof. (Sketch) Clearly $T \cup \mathrm{ADD}(S) - \mathrm{DEL}(S)$ is the result of the application of S on T. We verify that $T \cup \mathrm{ADD}(S) - \mathrm{DEL}(S)$ is a basic output tree. □

The following lemma establishes the relationships between the addition and deletion sets, and the basic input and output sets.

Lemma 3. *Let S be a consistent QL schedule.*

$$\begin{aligned}
N_O^{min} =& (N_I^{min} - \{n \mid \exists (m,l,n) \in \mathrm{DEL}(S)\}) \cup \\
& \{n \mid \exists (m,l,n) \in \mathrm{ADD}(S)\} \\
N_O^{max} =& (N_I^{max} - \{n \mid \exists (m,l,n) \in \mathrm{DEL}(S)\}) \cup \\
& \{n \mid \exists (m,l,n) \in \mathrm{ADD}(S)\} \\
E_O^{min} =& (E_I^{min} - \mathrm{DEL}(S)) \cup \mathrm{ADD}(S) \\
E_O^{max} =& (E_I^{max} - \mathrm{DEL}(S)) \cup \mathrm{ADD}(S)
\end{aligned}$$

Proof. (Sketch) Results from Lemma 2. □

4 Equivalence and Serializability of QL Schedules

The purpose of a scheduler is to interleave requests by processes such that the resulting schedule is serializable. This can be done by deciding for each request whether the schedule extended with the requested operation is still serializable, without looking at the instance. In this section we discuss the problem of deciding whether two consistent QL schedules are equivalent, and whether a consistent QL schedule is serializable.

To begin with, it can be shown that the application two QL schedules over the same set of transactions on the same DT T result in the same DT, if they are both defined.

[4] We consider a graph as the set of its edges and vice versa.

Lemma 4. *Let S and S' be two QL schedules over the same set of transactions. $S[T] = S'[T]$ if $S[T]$ and $S'[T]$ are both defined.*

Proof. (Sketch) Considering a given edge, this edge is alternatively added and deleted in each of the applications. Since the two QL schedules are over the same set of transactions, the edge belongs to no result or to both results □

As a consequence the problem of deciding whether two consistent schedules over two given transactions are equivalent reduces to the problem of deciding whether their result is defined for the same DTs, which can be decided with the help of the basic input and output sets.

Theorem 2. *Two consistent QL schedules S_1, S_2 over the same set of transactions are equivalent iff they have the same set of basic input trees, ie. iff $N_I^{min}(S_1) = N_I^{min}(S_2)$, $N_I^{max}(S_1) = N_I^{max}(S_2)$, $E_I^{min}(S_1) = E_I^{min}(S_2)$ and $E_I^{max}(S_1) = E_I^{max}(S_2)$. Hence their equivalence is decidable in $O(n_S.log(n_S))$-time and $O(n_S)$-space.*

Proof. From Lemma 1 and Lemma 4. □

Note that this theorem does not hold for two arbitrary QL schedules. Indeed $S_1 = (\text{add}(m,l,n),t)$ and $S_2 = (\text{add}(m,l,n),t),(\text{del}(m,l,n),t)$ have the same basic input trees and are not equivalent.

We can use the basic input and output sets to decide whether one consistent schedule can directly follow another consistent schedule without resulting in an inconsistent schedule.

Lemma 5. *Let S_1 and S_2 be two consistent QL schedules. Let n_S be the size of $S_1.S_2$. $S_1.S_2$ is consistent iff $N_I^{min}(S_2) \subseteq N_O^{max}(S_1)$, $E_I^{min}(S_2) \subseteq E_O^{max}(S_1)$, $N_O^{min}(S_1) \subseteq N_I^{max}(S_2)$, $E_O^{min}(S_1) \subseteq E_I^{max}(S_2)$. The consistency of $S_1.S_2$ is decidable in $O(n_S.log(n_S))$-time and $O(n_S)$-space.*

Proof. (Sketch) A result of the C-conditions, Lemma 1 and Theorem 1.

□

The following lemma shows how the basic input and output sets can be computed for a concatenation of schedules if we know these sets for the concatenated schedules.

Lemma 6. *Let $S_1, S_2, ..., S_n$ and $S_1.S_2...S_n$ be $(n+1)$ consistent QL schedules. Then*

$$N_I^{min}(S_1...S_n) = \bigcup_{i=1}^{n}(N_I^{min}(S_i) \cap \bigcap_{k<i} N_I^{max}(S_k))$$
$$N_I^{max}(S_1...S_n) = \bigcap_{i=1}^{n}(N_I^{max}(S_i) \cup \bigcup_{k<i} N_I^{min}(S_k))$$
$$E_I^{min}(S_1...S_n) = \bigcup_{i=1}^{n}(E_I^{min}(S_i) \cap \bigcap_{k<i} E_I^{max}(S_k))$$
$$E_I^{max}(S_1...S_n) = \bigcap_{i=1}^{n}(E_I^{max}(S_i) \cup \bigcup_{k<i} E_I^{min}(S_k))$$

If n_S is the size of $S_1.S_2...S_n$ then these equalities can be verified in $O(n_S^3)$-time and $O(n_S)$-space.

Proof. By induction using Definition 4. □

Finally, the previous theorems can be used to show that serializability is decidable.

Theorem 3. *Given a QL schedule S of n_t transactions. It is decidable whether S is serializable in $O(n_t^{n_t}.n_S^3)$-time, and in $O(n_S^2)$-space.*

Proof. (Sketch) Indeed,

1. we verify whether each transaction is consistent, which is done in $O(n_S^3.n_t)$-time and in $O(n_S)$-space (Corollary 1);
2. we draw a graph that indicates which transactions can follow directly which other transactions (i.e. $T_i.T_j$ is defined), which is done in $O(n_t^2.n_S.log(n_S))$-time and in $O(n_t^2 + n_S)$-space (Lemma 5);
3. S is serializable iff there is a Hamilton path that is equivalent with S; to verify this:
   (a) we calculate the ordered N_I^{min}, N_I^{max},E_I^{min} and E_I^{max} of the transactions, which is done in $O(n_t.n_S.log(n_S))$-time and $O(n_t.n_S)$-space (Lemma 1);
   (b) there are $O(n_t^{n_t})$ Hamilton paths, for each of them:
      i. we verify its consistency, which is done in $O(n_S^3)$-time and $O(n_S)$-space (Corollary 1);
      ii. we calculate the ordered N_I^{min}, N_I^{max}, E_I^{min} and E_I^{max} of the Hamilton path, which is done in $O(n_S.log(n_S))$-time and $O(n_S)$-space (Lemma 1);
      iii. Lemma 6 and Theorem 2 are verified in $O(n_S^3)$-time and in $O(n_S)$-space.

□

5 Equivalence and Serializability of Schedules

In the previous section we only considered QL schedules, but in this section we consider all schedules. We start with generalizing the notions that were introduced for QL schedules.

Definition 7. *A schedule S is called consistent iff its corresponding QL schedule S' is consistent.* ADD(S) = ADD(S') *where S' is the QL schedule of S. Analogously for* DEL, E_I^{min}, E_I^{max}, E_O^{min}, E_O^{max}, N_I^{min}, N_I^{max}, N_O^{min}, N_O^{max}.

To verify whether two consistent schedules over the same set of transactions are equivalent, we first eliminate the queries and verify whether the resulting QL schedules are equivalent. (Cfr. Theorem 2). In this section we investigate the equivalence of two consistent schedules over the same set of transactions and whose QL schedules are equivalent. In the following examples it is shown that such schedules can be equivalent on all the DTs they are defined on, on only some of them or on none.

Example 3. *Let $l_1 \neq l_3$. Consider the following schedules:*

$$\begin{aligned}
S_1 = {} & (\mathrm{add}(n_2, l_2, n_3), t_1),\ (\mathrm{query}(n_1, l_1/l_2), t_2),\\
& (\mathrm{del}(n_2, l_2, n_3), t_1),\ (\mathrm{del}(n_1, l_3, n_2), t_1)\\
S_2 = {} & (\mathrm{query}(n_1, l_1/l_2), t_2),\ (\mathrm{add}(n_2, l_2, n_3), t_1),\\
& (\mathrm{del}(n_2, l_2, n_3), t_1),\ (\mathrm{del}(n_1, l_3, n_2), t_1)
\end{aligned}$$

S_1 and S_2 are correct and their corresponding QL schedules are equal. They are equivalent on all DT*s on which they are defined, hence they are equivalent.*

Consider the following schedules S_3 and S_4:

$$\begin{aligned}
S_3 = {} & (\mathrm{add}(n_2, l_2, n_3), t_1),\ (\mathrm{query}(n_1, l_1/l_2), t_2),\\
& (\mathrm{del}(n_2, l_2, n_3), t_1)\\
S_4 = {} & (\mathrm{query}(n_1, l_1/l_2), t_2),\ (\mathrm{add}(n_2, l_2, n_3), t_1),\\
& (\mathrm{del}(n_2, l_2, n_3), t_1)
\end{aligned}$$

S_3 and S_4 are consistent and their corresponding QL schedules are equal. They are equivalent on some DT*s on which they are defined and not equivalent on others.*

Finally, let S_5 and S_6 be the following schedules:

$$\begin{aligned}
S_5 = {} & (\mathrm{add}(n_2, l_2, n_3), t_1),\ (\mathrm{query}(n_1, l_1/l_2), t_2),\\
& (\mathrm{del}(n_2, l_2, n_3), t_1),\ (\mathrm{del}(n_1, l_1, n_2), t_1)\\
S_6 = {} & (\mathrm{query}(n_1, l_1/l_2), t_2),\ (\mathrm{add}(n_2, l_2, n_3), t_1),\\
& (\mathrm{del}(n_2, l_2, n_3), t_1),\ (\mathrm{del}(n_1, l_1, n_2), t_1)
\end{aligned}$$

S_5 and S_6 are consistent and their corresponding QL schedules are equal. They are, however, equivalent on no DT *on which they are defined.*

In order to prove the decidability of the equivalence of two schedules over the same set of transactions we first define the notion of SOP, Set Of Prefixes in Subsection 5.1, and some additional notation in Subsection 5.2.

5.1 SOP - Set Of Prefixes

Informally, the notion "Set Of Prefixes" (SOP) of a path expression pe for a label path lp, will allow us to find a set of path expressions pe', such that all path expressions pe'/lp together represent exactly these label paths of pe that end on lp. For example, consider the path expression $pe = b//*$ and the label path $lp = a$. Then b/a and $b//*/a$ represent the label paths of pe that end with label path a. Hence b and $b//*$ are a-prefixes of $b//*$.

We will now define the set of non-empty lp-prefixes in pe, denoted as $\mathrm{SOP}(pe)_{lp}$ as a set of path expressions that together represent the set of label paths pe' such that $pe'/lp \in \mathbf{L}(pe)$[5]. For instance $\mathrm{SOP}(b//*)_a = \{b, b//*\}$.

Definition 8. *Let pe be a path expression, lp be a label path and $l \in \mathcal{L}$. The set of non-empty lp-prefixes in pe, denoted as $\mathrm{SOP}(pe)_{lp}$ is defined by*

[5] We consider pe/ϵ to be equal to pe.

$$
\begin{array}{ll}
\mathrm{SOP}(pe)_\epsilon & = \{pe\} \\
\mathrm{SOP}(pe/*)_l & = \mathrm{SOP}(pe/l)_l \ = \{pe\} \\
\mathrm{SOP}(pe//*)_l & = \mathrm{SOP}(pe//l)_l = \{pe, pe//*\} \\
\mathrm{SOP}(pe/*)_{lp/l} & = \mathrm{SOP}(pe/l)_{lp/l} = \mathrm{SOP}(pe)_{lp} \\
\mathrm{SOP}(pe//*)_{lp/l} & = \mathrm{SOP}(pe//l)_{lp/l} \\
 & = \mathrm{SOP}(pe)_{lp} \cup \mathrm{SOP}(pe//*)_{lp} \\
\textit{Otherwise } \mathrm{SOP}(pe)_{lp} & = \emptyset.
\end{array}
$$

Furthermore we define $\mathbf{L}(\mathrm{SOP}(pe)_{lp}) = \bigcup_{pe_i \in \mathrm{SOP}(pe)_{lp}} \mathbf{L}(pe_i)$.

Lemma 7. $\mathbf{L}(\mathrm{SOP}(pe)_{lp}) = \{lp' \mid lp'/lp \in \mathbf{L}(pe)\}$.

Example 4.

- $\mathrm{SOP}(a/*/*/b)_{a/b} = \mathrm{SOP}(a/*/*)_a = \{a/*\}$
- $\mathrm{SOP}(a//*/c)_{a/b/c} = \mathrm{SOP}(a//*)_{a/b} = \mathrm{SOP}(a)_a \cup \mathrm{SOP}(a//*)_a = \{a, a//*\}$
- $\mathrm{SOP}(*//*)_{a/b/c} = \mathrm{SOP}(*)_{a/b} \cup \mathrm{SOP}(*//*)_{a/b} = \emptyset \cup \mathrm{SOP}(*)_a \cup \mathrm{SOP}(*//*)_a = \emptyset \cup \emptyset \cup \{*, *//*\} = \{*, *//*\}$
- $\mathrm{SOP}(a//b//d)_{b/c/d} = \{a, a//*, a//b, a//b//*\}$

Lemma 8. *Let pe be a path expression and lp be a label path.* $\mathrm{SOP}(pe)_{lp} = \{pe' \mid pe'$ *a prefix of pe and* $\mathbf{L}(pe'/lp) \subseteq \mathbf{L}(pe)\} \cup \{pe'//* \mid pe'$ *a prefix of pe and* $\mathbf{L}(pe'//*/lp) \subseteq \mathbf{L}(pe)\}$.

Lemma 9. *Let pe be a path expression of length n_{pe} and lp be a label path.* $\mathrm{SOP}(pe)_{lp}$ *is uniquely defined, finite and is computable in $O(n_{pe}^2.(n_{pe} + n_{lp}))$-time and $O(log(n_{pe} + n_{lp}))$-space.*

Proof. From Lemma 8 we know that we have to calculate the two sets: $\{pe' \mid pe'$ a prefix of pe and $\mathbf{L}(pe'/lp) \subseteq \mathbf{L}(pe)\}$ and $\{pe'//* \mid pe'$ a prefix of pe and $\mathbf{L}(pe'//*/lp) \subseteq \mathbf{L}(pe)\}$. Hence $\mathrm{SOP}(pe)_{lp}$ can be calculated in $O(n_{pe}^4)$-time [18], and in $O(n_{pe}^2)$-space. □

Lemma 10. *Let pe be a path expression and lp_1 and lp_2 be two label paths.* $\mathbf{L}(\mathrm{SOP}(pe)_{lp_1}) \subseteq \mathbf{L}(\mathrm{SOP}(pe)_{lp_2})$ *iff* $\forall pe_i \in \mathrm{SOP}(pe)_{lp_1}(\mathbf{L}(pe_i/lp_2) \subseteq \mathbf{L}(pe))$.

Proof. From Definition 8 and Lemma 7. □

Theorem 4. *Let pe be a path expression and lp_1 and lp_2 be two label paths. It is decidable in $O(n_{pe}^2.(n_{pe} + n_{lp}))$-time and in $O(n_{pe} + log(n_{pe} + n_{lp}))$-space whether* $\mathbf{L}(\mathrm{SOP}(pe)_{lp_1}) = \mathbf{L}(\mathrm{SOP}(pe)_{lp_2})$.

Proof. From Lemmas 9 and 10.

□

5.2 PQRN - Potential Query Result Nodes

The main concept that is introduced in this subsection is the set of Potential Query Result Nodes (PQRN) for a query Q in a schedule S. This set will contain all nodes n, that are added or deleted in S, and for which there exists a document tree T, such that n is in the result of the query Q when S is applied to T. [6]. For this puprose, we need to introduce some additional notations to characterize the trees on which a query Q in a schedule S will be executed. We will use these notations later on, and we also give some complexity results for calculating the value of these concepts.

Let S be a consistent schedule that contains the query $Q = \text{query}(n, pe)$.

- We denote by S^Q the actions of S that occur before Q; S^Q is called a subschedule of S;
- Let T be a basic input tree of S. We define $T^Q = S^Q[T]$ as the DT on which Q in S is evaluated; hence the result of the application of the query Q in S is $Q[T^Q]$;
- We denote by $E^{min}(S^Q)$ as the set that contains exactly those edges that **are required** in T^Q; This set is equal to $(E_I^{min}(S) - \text{DEL}(S^Q)) \cup \text{ADD}(S^Q)$ (Lemma 3);
- We denote by $E^{max}(S^Q)$ as the set that contains exactly those edges that **are allowed** in T^Q; This set is equal to $(E_I^{max}(S) - \text{DEL}(S^Q)) \cup \text{ADD}(S^Q)$ (Lemma 3).

$E^{min}(S^Q)$ is a forest (Property 1). As such every node m of $E^{min}(S^Q)$ has a unique ancestor without a parent in $E^{min}(S^Q)$; it is denoted by $ARoot(S^Q, m)$. The label of the path from $ARoot(S^Q, m)$ to m in $E^{min}(S^Q)$ is denoted by $ALabel(S^Q, m)$.

Lemma 11. *$Alabel(S^Q, m)$ and $ARoot(S^Q, m)$ can be computed in $O(n_S^2)$-time and $O(n_S)$-space.*

Proof. A consequence of Lemma 1.

□

If $\text{add}(m, l, n)$ or $\text{del}(m, l, n)$ are operations of S we say that n is a *non-building-node* of S. Otherwise n is called a *building-node* of S. Note that
$E^{max}(S^Q) = E^{min}(S^Q) \cup \{\text{edges that contain only building nodes}\}$ since
$E^{min}(S^Q) = (E_I^{min}(S) - \text{DEL}(S^Q)) \cup \text{ADD}(S^Q)$,
$E^{max}(S^Q) = (E_I^{max}(S) - \text{DEL}(S^Q)) \cup \text{ADD}(S^Q)$ and
$E_I^{max}(S) = E_I^{min}(S) \cup \{\text{edges that contain only building nodes}\}$.

We will now define the set of nodes $\text{PQRN}(S, Q)$. This set will contain all non-building-nodes that can be in the result of a query that starts with a node n that is not in $E^{min}(S^Q)$. After the formal definition we will show that this definition corresponds to this informal description. Finally we will show that this set is computable in polynomial time and space.

[6] This notion is only defined for a subset of queries, which will be specified later on.

Definition 9. *Let S be a consistent schedule that contains a query $Q = \text{query}(n, pe)$. We define the set* $\text{PQRN}(S, Q)$ *as:*
$\text{PQRN}(S, Q) = \{m|$

- *m a node in the graph $E^{min}(S^Q)$;*
- *m a non-building-node;*
- *$ARoot(S^Q, m)$ a building-node;*
- *$ARoot(S^Q, m) \neq n$;*
- $\mathbf{L}(\text{SOP}(pe)_{ALabel(S^Q,m)}) \neq \emptyset$

$\}$.

Lemma 12. *Let S be a consistent schedule, $Q = \text{query}(n, pe)$ a query that appears in S, and n a node that is not in the graph $E^{min}(S^Q)$. Then* $\text{PQRN}(S, Q)$ *is the set of non-building-nodes m, such that there exists a basic input tree T of S for which m is in the result of the query Q on the document tree $S^Q[T]$.*

Lemma 13. $\text{PQRN}(S, Q)$ *can be computed in $O(n_S^5)$-time and $O(n_S)$-space.*

Proof. From Theorem 4 and Lemma 11. ☐

5.3 Decidability of Equivalence

We will now establish the main result of this paper by proving that the equivalence of two schedules is decidable in our framework.

Lemma 14. *Given two consistent schedules S_1 and S_2 over the same set of transactions and whose QL schedules are equivalent. Let $Q = \text{query}(n, pe)$ be a query in these schedules and let n_a be the total number of actions in S_1 and S_2. It is decidable in $O(n_S^6)$-time and $O(n_S)$-space whether Q gives the same answer in S_1 as in S_2 for every possible basic input tree of S_1 and S_2.*

Proof. The next condition $CND(S_1, S_2, Q)$ detects when Q gives the same answer in S_1 as in S_2 for every possible basic input tree of S_1 and S_2 :
Definition of $CND(S_1, S_2, Q)$

1. $\{m \mid$ there is a path of $\mathbf{L}$(pe) from n to m in $E^{min}(S_1^Q)\} = \{m \mid$ there is a path of $\mathbf{L}$(pe) from n to m in $E^{min}(S_2^Q)\}$; this can be done in $O(n_S^3)$ time; this is a consequence of a result in [18]
2. furthermore, if n **is a building-node of** S_i:
   (a) $PQRN(S_1, Q) = PQRN(S_2, Q)$
   (b) for the nodes $m \in PQRN(S_1, Q)$ hold that
      i. $ARoot(S_1^Q, m) = ARoot(S_2^Q, m)$
      ii. $\mathbf{L}(\text{SOP}(pe)_{ALabel(S_1^Q,m)}) = \mathbf{L}(\text{SOP}(pe)_{ALabel(S_2^Q,m)})$

All this can be computed in $O(n_S^6)$-time and in $O(n_S)$-space.

□

The definition of the CND condition is illustrated in the following example.

Example 5. *In Example 3 we have*

- $E^{min}(S_1^Q) = \{(n_1, l_3, n_2), (n_2, l_2, n_3)\}$ *and* $E^{min}(S_2^Q) = \{(n_1, l_3, n_2)\}$*;* 1. *is fulfilled;* n_1 *is a building-node;* n_2 *and* n_3 *are non-building-nodes;* $\mathrm{PQRN}(S_1, Q) = \mathrm{PQRN}(S_2, Q) = \emptyset$*; hence* $CND(S_1, S_2, Q)$ *is fulfilled and* Q *gives the same answer in* S_1 *as in* S_2 *for every possible basic input tree of* S_1 *and* S_2*.*
- $E^{min}(S_3^Q) = \{(n_2, l_2, n_3)\}$ *and* $E^{min}(S_4^Q) = \emptyset$*;* 1. *is fulfilled;* n_2 *is a building-node;* n_3 *is a non-building-node;* $\mathrm{PQRN}(S_3, Q) = \{n_3\}$ *and* $\mathrm{PQRN}(S_4, Q) = \emptyset$*; hence* 2.$(a)$ *is not fulfilled and* Q *does not give the same answer in* S_3 *as in* S_4 *for every possible basic input tree of* S_3 *and* S_4*.*
- $E^{min}(S_5^Q) = \{(n_1, l_1, n_2), (n_2, l_2, n_3)\}$ *and* $E^{min}(S_6^Q) = \{(n_1, l_1, n_2)\}$*; hence* S_5 *and* S_6 *are not equivalent, since* 1. *is not fulfilled and* Q *does not give the same answer in* S_5 *as in* S_6 *for every possible basic input tree of* S_5 *and* S_6*.*

Theorem 5. *Given two consistent schedules* S_1 *and* S_2 *over the same set of transactions and whose QL schedules are equivalent. It is decidable in* $O(n_S^6)$*-time and* $O(n_S)$*-space whether they are equivalent.*

Proof. Consequence of Lemma 14. □

Finally, we can now combine the previous theorems to show that serializability is decidable in our framework.

Theorem 6. *Given a consistent schedule* S*. It is decidable in* $O(n_t^{n_t}.n_S^6)$*-time and* $O(n_S^2)$*-space whether* S *is serializable.*

Proof. From Theorem 3 and Theorem 5. □

6 Conclusion and Future Work

In this paper we have presented a concurrency control mechanism for semistructured databases. This mechanism is document-independent in the sense that two schedules of semistructured transactions are equivalent iff they are equivalent on all possible documents. This notion of equivalence is a special form of view equivalence. The transactions that we consider, consist of simple updates (inserting and deleting edges at the bottom of a tree) and queries (simple path expressions containing child and descendant steps). We have shown that equivalence of schedules can be decided efficiently (i.e., in polynomial time in the size of the schedule), and that the serializability can be decided in time polynomial in the size of the schedule and exponential in the number of transactions. Improving this complexity result is expected to be difficult, since it is generally known that deciding view serializability is NP-complete [21].

In future work, we will extend the results of this paper by defining the behaviour of currently undefined actions, and hence allowing more schedules to be serialized. For example, the addition of an edge which is already in the input tree is undefined in our current work, and hence the operation fails. However, we could also say that as a result of this addition, we obtain an output tree which is equal to the input tree, and a message which indicates that the edge was already present. In this approach the result of a schedule applied on a document tree would be an annotated version of the schedule and an output document tree. A schedule would then be serializable iff there exists a serial schedule with the same operations, which has, for each document, the same output document tree and the same message for each operation.

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] P. A. Bernstein and N. Goodman. A sophisticate's introduction to distributed concurrency control (invited paper). In *Eigth International Conference on Very Large Data Bases, September 8-10, 1982, Mexico City, Mexico, Proceedings*, pages 62–76. Morgan Kaufmann, 1982.

[4] E.-H. Choi and T. Kanai. XPath-based concurrency control for XML data. In *Proceedings of the 14th Data Engineering Workshop (DEWS 2003)*, Kaga city, Ishikawa, Japan, March 3-5 2003. Available at: `http://www.ieice.org/iss/de/DEWS/proc/2003/papers/6-C/6-C-04.pdf`.

[5] S. Dekeyser and J. Hidders. Path locks for XML document collaboration. In T. W. Ling, U. Dayal, E. Bertino, W. K. Ng, and A. Goh, editors, *Proceedings of the 3rd International Conference on Web Information Systems Engineering (WISE 2002)*, pages 105–114. IEEE Computer Society, 2002.

[6] S. Dekeyser and J. Hidders. A commit scheduler for XML databases. In *Proceedings of the Fifth Asia Pacific Web Conference*, Xi'an, China, 2003.

[7] S. Dekeyser and J. Hidders. A commit scheduler for XML databases. In X. Zhou, Y. Zhang, and M. E. Orlowska, editors, *Web Technologies and Applications, 5th Asian-Pacific Web Conference (APWeb 2003)*, volume 2642 of *Lecture Notes in Computer Science*, pages 83–88. Springer, 2003.

[8] S. Dekeyser, J. Hidders, and J. Paredaens. Instance indepedent concurrency control for semistructured databases. In *Proceedings of the Eleventh Italian Symp. on Advanced Database Systems (SEBD 2003)*, pages 323–334, 2003.

[9] S. Dekeyser, J. Hidders, and J. Paredaens. A transaction model for XML databases. *World Wide Web Journal (to appear)*, 2003.

[10] T. Grabs, K. Böhm, and H.-J. Schek. XMLTM: efficient transaction management for XML documents. In *Proceedings of the 2002 ACM CIKM International Conference on Information and Knowledge Management (CIKM 2002)*, pages 142–152, McLean, VA, USA, November 4-9 2002. ACM.

[11] J. Gray, G. Putzolo, and I. Traiger. *Modeling in Data Base Management Systems*, chapter Granularity of locks and degrees of consistency in a shared data base. North Holland, Amsterdam, 1976.

[12] M. P. Haustein and T. Härder. Adjustable transaction isolation in xml database management systems. In Z. Bellahsene, T. Milo, M. Rys, D. Suciu, and R. Unland, editors, *XSym*, volume 3186 of *Lecture Notes in Computer Science*, pages 173–188. Springer, 2004.

[13] M. P. Haustein and T. Härder. A lock manager for collaborative processing of natively stored xml documents. In S. Lifschitz, editor, *SBBD*, pages 230–244. UnB, 2004.

[14] S. Helmer, C.-C. Kanne, and G. Moerkotte. Lock-based protocols for cooperation on XML documents. Technical Report 06/2003, University of Mannheim, Germany, 2003. Available at: `http://pi3.informatik.uni-mannheim.de/publications/TR-03-006.ps`.

[15] S. Helmer, C.-C. Kanne, and G. Moerkotte. Evaluating lock-based protocols for cooperation on xml documents. *SIGMOD Rec.*, 33(1):58–63, 2004.

[16] J. Hidders, J. Paredaens, and R. Vercammen. Proof of serializability for semistructured databases. Technical Report UA 2004-01, University of Antwerp, 2004. Available at: `http://www.adrem.ua.ac.be/pub/TR2004-01.pdf`.

[17] K.-F. Jea, S.-Y. Chen, and S.-H. Wang. Concurrency control in XML document databases: XPath locking protocol. In *Proceedings of the 9th International Conference on Parallel and Distributed Systems (ICPADS 2002)*, pages 551–556. IEEE, December 17-20 2002.

[18] G. Miklau and D. Suciu. Containment and equivalence for an xpath fragment. In L. Popa, editor, *PODS*, pages 65–76. ACM, 2002.

[19] C. Papadimitriou. *The Theory of Database Concurrency Control*. Computer Science Press, Rockville, MD, 1986.

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

[21] G. Weikum and G. Vossen. *Transactional Information Systems*. Morgan Kaufmann, 2002. ISBN: 1-55860-508-8.