Algorithms for Temporal Query Operators in XML Databases

Kjetil Nørvåg*
Department of Computer and Information Science
Norwegian University of Science and Technology
7491 Trondheim, Norway
Kjetil.Norvag@idi.ntnu.no

Abstract

We have in a previous paper introduced the new query operators that are needed in order to support an XML query language that supports temporal operations. The query operators make it possible to query historical versions, retrieve documents valid at a certain time, query changes to documents, etc. In this paper, we describe algorithms for execution of the query operators, and we also discuss document content indexing for more efficient execution of the operators.

Keywords: XML, temporal databases, query processing, query execution

1 Introduction

Queries against the XML data can either be performed directly to the database storing the XML data (for example an object-relational database system), or to an XML data warehouse, created from XML data collected from the Web (for example Xyleme [21]).

The contents of an XML database or XML data warehouse is seldom static. New documents are created, documents are deleted, and more important: documents are updated. In many cases, we want to be able to search in historical (old) versions, retrieve documents that was valid at a certain time, query changes to documents, etc. In order to realize an efficient temporal XML database system, several issues have to be solved, including efficient storage of versioned XML documents, efficient indexing of temporal XML documents, and temporal XML query processing. In this paper, we concentrate on the issue of temporal XML query execution. Based on the query operators described in [13], we describe algorithms that can be used to realize these query operators.

The organization of the rest of this paper is as follows. In Section 2 we give an overview of related work. In Section 3 we give an overview of the work presented in [13] in order to put the new algorithms presented in this paper in context. In Section 4 we describe algorithms for executing the operators. In Section 5 we discuss some possible bottlenecks and scalability issues. Finally, in Section 6, we conclude the paper and outline issues for further research.

2 Related work

A model for representing changes in semistructured data (DOEM) and a language for querying changes (Chorel) was presented by Chawathe et al. in [4, 5]. Chorel queries were translated to Lorel (a language for querying semistructured data), and can therefore be viewed as a stratum approach. The work by Chawathe et al. has later been extended by Oliboni et al. [14].

*This work was done while the author was an ERCIM fellow at the VERSO group at INRIA, France.

Storage of versioned documents is studied by Marian et al. [12] and Chien et al. [6, 7, 15]. Chien et al. also consider access to previous versions, but only snapshot retrievals.

An approach that is orthogonal, but related to the work presented in this paper, is to introduce valid time features into XML documents, as presented by Grandi and Mandreoli [10, 11].

Other relevant work includes work on temporal databases. Examples are temporal document databases [3], temporal object query language [9], and temporal object database systems [16].

3 Temporal XML queries

In this section we give an introduction to temporal XML queries and query operators. These aspects have already been presented in [13] but are provided in order to make this paper self-containing.

3.1 Time and element identity

Two important issues that pose some additional difficulties in the context of XML, and in particular in the context of XML documents retrieved from the Web (in the case of an XML data warehouse), are time and identity. These issues will now be discussed in more detail.

3.1.1 Time in XML databases

As discussed in [13], we have different aspects of time in temporal databases. The two most common aspects are transaction time and valid time. In the context of XML database, we have two cases that from a query point of view are similar to transaction time: 1) local storage of documents (e.g., in a database system storing XML documents) where we have the exact timestamp for the documents, and 2) XML warehouses or other non-synchronized storage of copies of XML documents where we in general do not know the time of creation/storage of an XML document, only the time when the document was retrieved from the Web ("crawled"). Note that in the second case, the documents in the warehouse are not retrieved at the same point in time, so that the result is an inconsistent view of the documents. For example, a document can have references to a document not yet retrieved, or to a document that has already been deleted, and there might have been updates between the versions we have retrieved, i.e., we do not necessarily have all the versions of a particular document.

A third case, which has similarities to valid time, is document time, and is included in the documents themselves. This can for example be the time the document was written, or when it was posted.

In this paper, we concentrate on a transaction-time support. It should be noted that the techniques presented here are equally applicable to a valid-time context, but that additional operators should be introduced in that case, for example coalescing.

3.1.2 Identity of elements in versioned XML documents

XML documents have a quasi-persistent[1] identifier, the URL. However, in general the elements of a document *do not* have any identity of their own that persist from one version of a document to the next. This implies that many queries can be difficult to express, as well as expensive to execute. Two simple examples are 1) a query for the create time of elements, and 2) a query asking for the previous version of a certain element. Thus, although elements seen from "the outside" do not have persistent identifiers, we believe that the storage system should support this feature, in order to make it a part of the data model for the query language. One particular system that provides this functionality is Xyleme [12]. The persistent identifiers, in Xyleme called XIDs, identify an element in a particular document in a time independent manner, and will not be reused when an element is deleted. For convenience we will in this paper also use the acronym EID (Element ID), which is the concatenation of document ID and XID. Thus, an EID identifies uniquely a particular element in a particular document.

In a temporal XML database there will in general be more than one version of each element (different versions of an element have the same EID). In order to uniquely identify a particular version of an element,

[1] *Quasi*-persistent based on the observation that documents on the Web frequently are moved.

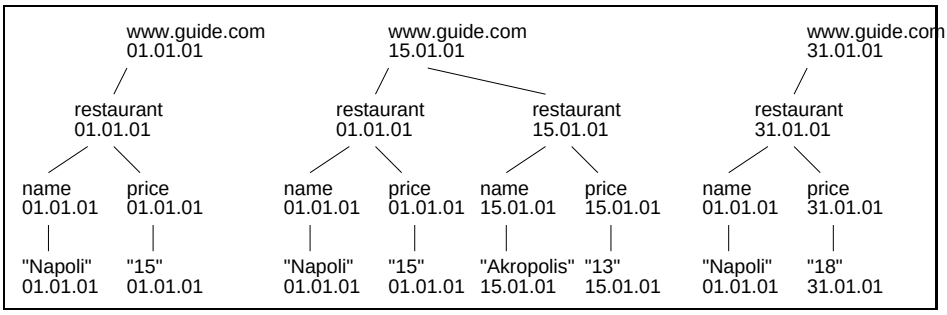

Figure 1: The restaurant list at guide.com as retrieved on January 1st, January 15th, and January 31st. The timestamps on each element is the time of update of the element or one of its children.

the timestamp can be used together with the EID. We denote the identifier of a particular version of an element TEID (temporal EID), i.e., the concatenation of EID and timestamp.

3.2 Assumed data model

A document in the database is viewed as a forest of trees. One of the advantages of this approach, is that a query on versions of a document (or several documents) is similar to a query on a general set of XML documents, which can also be viewed as a forest of trees. It is also similar to the forest of trees resulting from pre-filtering (i.e., returning subtrees of documents, possibly more than one tree for each document).

We assume that every element has a timestamp, and that every update of an element also implies update of the element it is contained in. Note that even if this logically has to be applied recursively up to the document to the root, *it does not have to be implemented in this way.* Note that the distinction between document timestamp and element timestamp is not significant for snapshot queries, only for change-oriented queries. An example of document versions can be seen in Figure 1. The document versions are versions of a restaurant guide database, as described in [5]. The restaurant guide will also be used in the examples below.

Note that in the *physical* storage model, it is unlikely that all versions of all documents are stored as complete versions. Instead, previous versions are stored as, e.g., delta versions. In order to reconstruct these previous versions, we might have to retrieve and process the complete last versions as well as a number of delta documents. This will be described in more detail in Section 4.1.

3.3 Examples of temporal XML queries

The main purpose of this section is not to create a new query language, but to describe what kind of queries can be expected in a temporal XML database. The query language is based on a mix of Lorel, the Xyleme query language[2] [2], and elements of XPath and XQuery [20] (note that the query operators and associated algorithms presented later in this paper are independent of which query language is actually used). For example, in order to retrieve documents valid at a particular time (snapshot query), a timestamp is given for the path in the `FROM` clause, filtering out only those element versions valid at the particular time:

```
SELECT R
FROM doc("http://guide.com/")/restaurant[26/01/2001] R
```

For more complex queries, or when we want more than one version to be selected, we use the keyword EVERY instead of timestamp. For example, in order to retrieve the price history of the restaurant named Napoli:

[2]Note that even if Xyleme has support for historical versions/deltas, there is no special support for these in the query language.

```
SELECT TIME(R), R/price
FROM doc("http://guide.com/")/restaurant[EVERY] R
WHERE R/name="Napoli"
```

where `TIME(R)` returns the timestamp of the element `R`. Further predicates on time can be included in the SELECT clause, including delete and create time of elements.

3.4 Algebra operators

Here we consider operators that will be needed in temporal XML query processing, and describe them in terms of input, output, and operation. In addition to the operators described here, we also assume the availability of traditional operators, for example projection and join, but we will not discuss them any further.

Two of the operators are extensions of the `PatternScan` operator described in [2]. The `PatternScan` operator takes as input a forest of trees (which can be a set of EIDs), which are XML documents or filtered elements (subtrees) from XML document, and a pattern tree which each tree shall be matched against. The pattern tree includes information on projection as well as *isParentOf* and *isAscendantOf* relationships. We can define the operator as `PatternScan(`F`, pattern)` where F is a forest of trees and `pattern` is the pattern tree. The algorithm for the original `PatternScan` operator can be informally described as:

1. For all words w_i in `pattern`, call P_i=`FTI_lookup(`w_i`)`, where `FTI_lookup` denote the function used to retrieve all postings of $word$ in the full-text index (e.g., retrieves document identifier and relationship information for all instances of `word` in all documents).
2. Execute `Join(`P_1`,...,`P_i`)` with join attributes:
   - document identifier
   - relationship (e.g., *isParentOf* or *isAscendantOf*)

For more details on the `PatternScan` operator we refer to [2].

3.4.1 Overview of the operators

The temporal query operators to be added are as follows:

- `TPatternScan(`F`, pattern, `T`)`

  This is a temporal snapshot `PatternScan` operator. `TPatternScan` is similar to the `PatternScan` operator in [2], except that it operates on the snapshot of documents valid at time T. The output of the operator is a set of TEIDs (see Section 3.1.2).

- `TPatternScanAll(`C`, pattern)`

  `TPatternScanAll` returns all matches for $pattern$ for all versions of the documents in a collection C. The output of the operator is a set of TEIDs.

- `DocHistory(document, `T_S`, `T_E`)`

  Returns all the versions of a certain document valid in the interval [T_S, T_E>, where [T_S, T_E> is short for the time interval from T_S to T_E, including T_S but not T_E (open-ended upper bound). The output of the operator is a set of TEIDs, where the TEIDs are roots of documents.

- `ElementHistory(EID, `T_S`, `T_E`)`

  Returns all versions of an element valid in the interval [T_S, T_E>. The output of the operator is a set of TEIDs (with the EID in the TEID equal to the EID in the input parameters).

- `CreTime(TEID)`

  Returns the create time of an element. This is useful for retrieving elements created before or after a particular time, e.g., as in:

  ```
  SELECT R FROM ...  WHERE CREATE_TIME(R)>=11/01/2001
  ```

  Note that EIDs are unique, so that when an element is deleted the EID will not be reused for a new element. Thus, we do not strictly need timestamps in the element identification for the `CreTime` and `DelTime` operators. However, as will be shown later in the paper when we describe the algorithms for the operators, the availability of timestamp can improve performance. The timestamps will in general be available in any case, so that no extra cost is involved by assuming its availability.

- `DelTime(TEID)`

  DelTime returns the delete time of an element.

- `PreviousTS(TEID)`
  `NextTS(TEID)`
  `CurrentTS(EID)`

  Returns the timestamp of the previous/next/current version of a given element version (note that timestamp is not needed for the current version, as this is given implicitly). The timestamp, together with the EID (i.e., the TEID), can be used for retrieving the version itself. These operators can be used in constructions like:

  ```
  SELECT DISTINCT CURRENT(R)/name FROM ...  WHERE ...
  ```

  which retrieves the current versions of elements (possibly generated from a temporal snapshot), and as in

  ```
  SELECT PREVIOUS(R) FROM ...  WHERE ...
  ```

  which retrieve the previous versions of elements.

- `Reconstruct(TEID)`

  Reconstructs the tree rooted at EID in TEID for a particular version. The timestamp T_{EID} in the TEID can for example be the result of `NextTS/PreviousTS/CurrentTS` operations. If previous versions of documents are stored as deltas this can imply accessing and processing a potentially large number of deltas in addition to one complete version (the exception is the current version which will normally be stored as a complete version). If previous versions of documents are stored as complete documents, only the actual version itself needs to be read.

- `Diff(E1, E2)`

  In some cases we want to query for the changes between different versions of elements. These changes can conveniently be returned (and eventually post-processed by the application or in a separate query) as `edit scripts`. Edit scripts describe changes between two versions, similar to for example the information in RCS files. In our context, the edit scripts are XML trees themselves. As described by Cobena et al. in [8], the change can be represented by a set of operations *delete subtree*, *insert subtree*, *update value of a text*, and *move node*. Each operation refer to positions in one of the two versions. Note that as long as an edit script is represented in XML this operator does not break closure properties of queries. E1 and E2 can be versions of the same element, but can also represent different documents or subtrees of elements. `Diff` is useful in constructions like:

  ```
  SELECT DIFF(R1,R2) FROM ...  WHERE ...
  ```

It is possible to support string equality and string containment queries with different operators and access structures. However, as will be described later in this paper the access methods for containment queries already exist, so that there is little to gain from providing additional access methods for string equality. Therefore we expect that there are no separate operators and access structures for equality queries, and that the general containment operators/access methods are used, followed by equality testing.

3.4.2 Example queries

In order to illustrate the use of the operators, we now give three example queries based on the restaurant database example, together with the corresponding query operators.

Q1: List all restaurants in the list as of 26/01/2001:

```
SELECT R
FROM doc("http://guide.com/")/restaurant[26/01/2001] R
```

This is a snapshot query, listing the name in all versions of restaurant elements valid at time 26/01/2001 (that is, versions created before or on 26/01/2001 that is not further updated or deleted).
Operators: `TPatternScan`, followed by `Reconstruct`.

Q2: Retrieve the number of restaurants at 26/01/2001:

```
SELECT SUM(R)
FROM doc("http://guide.com/")/restaurant[26/01/2001] R
```

Operators: `TPatternScan` followed by the traditional aggregate operator `Sum`. Note that reconstruction of the documents is *not* needed. This is important, and shows that in many cases the storage of only deltas of previous document versions does not create performance problems.

Q3: List the price history of the restaurant "Napoli":

```
SELECT TIME(R), R/price
FROM doc("http://guide.com/")/restaurant[EVERY] R
WHERE R/name="Napoli"
```

The use of `EVERY` instead of a particular timestamp retrieves all versions of restaurant. Note that the predicate in the WHERE clause acts on all versions, not only the current version of the elements. As a result, the price history of all restaurants through the history with the name Napoli will be listed.
Operator: `TPatternScanAll`.

4 Operator execution

In this section we describe how temporal XML queries can be executed. We first describe the assumed physical storage model, storage and management of delta documents, and document content indexing, before we give an overview of the query operator algorithms.

4.1 Physical storage model

We assume that document versions are stored as a complete current version and previous versions stored in a chain of completed deltas (completed deltas can be used both as forward and backward deltas). This is similar to the storage strategy used in for example Xyleme [12]. This makes queries on temporal characteristics based on current version possible (e.g., query on some characteristic of the restaurant *currently* named "Napoli"), but other queries can be more expensive, for example queries on a restaurant not currently listed. However, as will be explained shortly, this bottleneck is partly handled by the primary indexing techniques that are used.

Each delta will in fact be stored as a separate XML document. Although stored in the same way in the repository, we will in the following distinguish between a delta document and a complete XML document. A named XML document will consist of one complete current version, and zero or more delta documents. The delta documents are indexed in a delta index (which could be as simple as an array). Each version is numbered, so that we do not have to store the timestamps in the text indexes etc. For each numbered delta, we store the timestamp of the actual version in the delta index. Note that this is only a good solution if we expect the delta indexes to be resident in memory. If not, it is best to store timestamps in the index.

4.2 Document content indexing

In order to facilitate `PatternScan` and similar operators, all documents are indexed by an inverted-list-based free-text index (FTI). This index indexes all words in the documents, including element names. The postings (one for each word occurrence) include document identifier as well as information that can be used to determine hierarchical relationships between elements from the same document. We denote the function used to retrieve all postings of $word$ in the full-text index (e.g., retrieves document identifier and relationship information for all instances of `word` in all documents) as `FTI_lookup(word)`.

Extensions. When extending `PatternScan` to the temporal domain as will be described below, the FTI has to be extended in order to support temporal operations. There are several alternatives for indexing the contents of the versions:

- **Index the contents of the versions.** This implies indexing the contents of each version together with timestamps. This facilitates search for versions containing a particular element/word, but searches like "when was a particular element deleted from a document" will be more costly.
- **Index the contents of the delta objects.** This implies indexing the operations, e.g., update, move and delete information directly in the text index. This would for example facilitate search for the path "`delete/restaurant/name/Napoli`". There are, however, some serious problems with this approach:
  - It would result in extremely many instances of the delta keywords/operations (for example *delete subtree* and *move node*).
  - It is less efficient for other access patterns, e.g., query on snapshot contents.
- **Indexing both snapshot and delta information.** This is a combination of the two previously described alternatives. This approach could be efficient for both snapshot and change based queries, but will result in larger indexes and higher update costs.

Based on the observations above, we choose the first alternative, i.e., to index the contents of versions. It should be noted that this does not rule out the two other alternatives, studying the relative performance of the three alternatives is left as a topic for future research.

Operations in temporal FTI. The following basic temporal operations have to be supported:

- `FTI_lookup(word)`: Lookup of current version, i.e., retrieves postings for all occurrences of `word` in documents currently valid (i.e., last version of undeleted documents).
- `FTI_lookup_T(word,`T`)`: Consider snapshot at time T. Retrives postings for all occurrences of `word` in document versions valid at time T.
- `FTI_lookup_H(word)`: Consider all postings for the whole history (all times) for a word, i.e., retrieves all postings for `word` in the FTI without considering timestamps.

Additional notes on indexes. Some query types can be very costly even when the described access structures are available. The fact that only deltas are stored can make version reconstruction necessary for many typical temporal queries. This can be expensive. This problem is especially serious because deltas will in many cases be stored unclustered (i.e., the deltas from one particular document is not stored together). As a result each delta read will involve a disk seek in the worst case. For this reason, the proposed access structures should only be considered basic (or primary) indexes, additional index structures and access methods might be needed in order to achieve acceptable performance. The goal should be to have an index that reduce the number of delta reads, as well as try to do as much of the work directly on the deltas and avoid reconstruction of snapshots.

4.3 Operator algorithms

In this section, we describe the algorithms for the temporal operators, assuming the availability of a temporal full-text index as described above.

4.3.1 TPatternScan

As described previously, two of our operators are extensions of the `PatterScan` operator in Xyleme [1]. The algorithm for the original `PatternScan` operator can be informally described as:

1. For all words w_i in `pattern`, call P_i=`FTI_lookup(`w_i`)`.
2. Execute `Join(`P_1`,...,`P_i`)` with join attributes:
   - document identifier
   - relationship (e.g., *isParentOf* or *isAscendantOf*)

The `TPatternScan(`F`, pattern, `T`)` operator can be executed using a variant of the algorithm for the original `PatternScan` operator. The main difference is that only entries valid at time T should be considered. This is done by using `FTI_lookup_T()` instead of `FTI_lookup()` when accessing the FTI:

1. For all words w_i in `pattern`, call P_i=`FTI_lookup_T(`w_i`,`T`)`
2. Execute `Join(`P_1`,...,`P_i`)` with join attributes:
   - document identifier
   - relationship (e.g., *isParentOf* or *isAscendantOf*)

4.3.2 TPatternScanAll

`TPatternScanAll(`C`, pattern)` can be executed by retrieving all occurrences of one of the words, and testing them against the other for match on document, path, and temporal validity. It can be viewed as a temporal multiway join:

1. For all words w_i in `pattern`, call P_i=`FTI_lookup_H(word)`
2. Execute `Join(`P_1`,...,`P_i`)` with join attributes:
   - document identifier
   - relationship (e.g., *isParentOf* or *isAscendantOf*)
   - time (i.e., words in the pattern valid at same time, which actually implies that this is a temporal join)

4.3.3 Reconstruct

In order to reconstruct a particular version given by TEID, the deltas between T_{EID} (i.e., the timestamp in the TEID) and NOW are applied on the complete current version. This is done backwards, i.e., the most current deltas first. With many deltas this can be very expensive, but there is also the possibility of snapshot versions made between T_{EID} and NOW. If snapshot versions exists, processing start using the oldest snapshot with timestamp greater or equal to T_{EID}, and the intermediate snapshots, to the version valid at T_{EID}.

4.3.4 DocHistory

Algorithm for `DocHistory(document,` T_S`,` T_E`)`:

1. Reconstruct the version valid at T_E as described for the Reconstruct operator.
2. Reconstructed the versions between T_S to T_E in the same way, using snapshots when possible. Note that this algorithm will output the document history backwards, i.e., the most previous versions first.

4.3.5 ElementHistory

Algorithm for `ElementHistory(EID,` T_S`,` T_E`)`:

1. F`=DocHistory(document(EID),` T_S`,` T_E`)`, where F is a forest of trees (i.e., a set of documents).
2. For each document in F, filter out the appropriate subtree rooted by EID.

Note that even if it was possible to optimize this so that only the desired subtrees are reconstructed, the whole deltas would have to be read anyway.

4.3.6 CreTime and DelTime

In order to retrieve the create time of an element using `CreTime(TEID)`, the two most interesting strategies are:

- Traverse the deltas backwards from the version valid at T_{EID} in the TEID until we find the delta where the element is introduced (note that no reconstruction is necessary). Note that this is the reason for assuming the availability of timestamp (i.e., TEID instead of only EID). If we did not have the timestamp we would have to start the traversal from the most recent version containing the name of the element. In many cases this would be the current version, because many elements have the same name.
- Use an additional index that indexes EID and create/delete timestamps. Note that if there is one common create-time index for all documents, inserts will not in general be append-only, because new elements can be inserted into documents. However, when a new document is inserted, updates to the index will be append-only, and it can be expected that frequently more than one element is inserted into an document. As a result, the average cost of inserting items into a create-time index will not be very high.

Traversing the deltas is straightforward, but can easily become a bottleneck if `CreTime` is a frequently used operator. In this case the best alternative will be to use an additional index.

Retrieving the delete time of an element based on TEID using `DelTime(TEID)` can be performed in a similar way, by traversal or indexing:

- If the document is deleted, and the element existed in the last version of the document, the delete time of the document is the delete time of the element. If the element was not resident in the last version, we have to traverse the deltas forward starting at time T_{EID} contained in the TEID until we find the delta where it was deleted.
- Use an additional index as described above.

Alternative strategies for `CreTime` and `DelTime` exists, for example:

- Store the create and delete timestamps in the XID list for each version. This is *not* a good idea, because we in addition to using space for timestamps also loose the opportunity of representing the XID list as a list of XID ranges.
- If deltas instead of version contents were indexed in the FTI (as described in Section 4.2), the FTI could be used to retrieve the create and delete timestamps of an element.

However, because of the drawbacks of these strategies we do not consider them appropriate.

4.3.7 NextTS, PreviousTS, and CurrentTS

These operators can be evaluated by a lookup in the delta index (see Section 4.1) for a particular document. The EID gives the document identifier, and given a certain timestamp T_{EID} the previous, next, and current timestamps cane be found by a lookup in the delta index. Note that in order to access the data in a version given by an EID and timestamp, the actual document version has to be reconstructed.

4.3.8 Diff

In order to generate the difference between elements, an XML difference algorithm with the subtrees rooted at the elements as input can be used. One such algorithm is described by Cobena et al. in [8]). This algorithm takes as input two documents/document versions, and returns the changes between the versions decribed in XML (see Section 3.4.1).

4.4 Issues related to equality in data model and query languages

In many queries, different versions are compared or matched. This issue is more complicated than it might looks like, because of the possible semantics of equality operators. In the XML data model [19] (and therefore in the XML query algebra [18] as well) two equality operators are proposed (although the exact semantics is still not completely defined): "=" which compares contents, and "==" which compares the identity of nodes.

Regarding the semantics of "=", the same issues applies in the temporal and non-temporal contexts. This includes issues as possible automatic type conversion, and whether shallow or deep equivalence should be used.

In the temporal context, when identity of nodes between different versions are compared, more difficult issues are introduced (this is related to the problem of creating deltas between versions [12]).

Assume the following query, which lists all restaurants that have increased their prices since 10/01/2001:

```
SELECT R1/name
FROM doc("http://guide.com/")/restaurant[10/01/2001] R1,
     doc("http://guide.com/")/restaurant R2
WHERE R1/name=R2/name AND R1/price < R2/price
```

This illustrates an example of a general problem: how to know we are actually comparing the prices of *the same restaurant.* In a city, several restaurant might have the same name (by coincidence, or because they are member of the same chain). Several restaurants can also have same name, but at different times (many restaurants have a short lifetime). In order to know more exactly, we could for example also compare address as well as restaurant name. This is easier said than done. For example, an address can be stored as a value in one address element, or the different parts of an address (street, number, etc) can be stored in different elements.

Based on the discussion of equality above, we have two approaches to compare the restaurants:

1. Compare by using "R1/name=R2/name" and assuming comparison of elements have the semantics of deep equality, i.e., is true if the two subtrees match completely, both in elements and values etc. However, this can be *too strict* in practice, considering that this is XML data.

2. If we assume that elements have persistent IDs (EIDs), this comparison could be performed by utilizing persistent node identifiers, possibly with syntax "R1==R2". This would in many cases make sense, because we can consider the actual restaurant as the same "object", even if elements (analogous to attributes in object database systems) have been updated, or even added. There are some problems with this approach. Some of them are related to the general problems when making diffs (deciding which elements/subtrees are equal from version to version). Another problem occurs if for example an entry is accidentally deleted from a page, and reintroduced in the next version. When reintroduced, will be given a new EID. If only EIDs are compared, equality test will fail even if the entry for a restaurant is exactly the same in version V_i and V_{i+2}.

3. Introduce a similarity operator "$\sim$" in the style of the approach of Theobald and Weikum [17].

There is not any perfect solution to this problem, but we consider a combination of shallow equality and a similarity operator to be the most interesting solution.

5 Discussion

Temporal query processing is in general expensive, and in our context we note the following potential performance bottlenecks:

- Many queries will involve a costly temporal scan (followed by Select).
- Queries involving create/delete will necessitate "delta chasing" if the information is not indexed.
- Using the XML storage strategy, the access to the deltas can be a major bottleneck. This can in part be reduced by snapshots at regular intervals between deltas. Actually, in order to not complicate processing based on deltas, these snapshots can be stored in addition to, rather than instead of, actual deltas.

There are also some issues on scalability that should be kept in mind: keeping historical versions will significantly increase the database as well as index size. It is not realistic to keep it all in main memory. Several strategies that partially solve the problem can be used:

- Only version particular documents, sites, or categories.
- Only make a subset of versioned documents/sites/categories possible to query for ordinary users. The size of this subset should be small enough to allow relevant index structures to fit in main memory.
- When a new version has been retrieved and a delta with the previous version, the new delta is aggregated with previous delta, instead of creating a new delta version. Only periodically, a new delta version is created.

6 Summary

In order to achieve the desired performance in XML databases with support for temporal queries, efficient execution of query operators is needed, and we have in this paper described the algorithms for execution of temporal XML queries, based on the query operators introduced in [13].

Future work include developing techniques for further reducing the cost of executing the query operators. The main goal in this context would be to develop techniques that can reduce the number of delta versions that have to be retrieved. Two important strategies for achieving this goal are to develop new types of indexes and algebraic rewriting techniques. Other future work includes study on how to modify the framework and operator algorithms so that they easily can be used on top of an existing non-temporal database system.

Acknowledgments

I would like to thank Serge Abiteboul for suggesting this topic of research, and Vincent Aguilera and Benjamin Nguyen for useful discussions and constructive comments.

References

[1] V. Aguilera, F. Boiscuvier, and S. Cluet. Pattern tree queries in Xyleme. Technical Report 200, Verso/INRIA, 2001.

[2] V. Aguilera, S. Cluet, P. Veltri, D. Vodislav, and F. Wattez. Querying XML documents in Xyleme. Technical Report 182, Verso/INRIA, 2000.

[3] M. J. Aramburu-Cabo and R. B. Llavori. A temporal object-oriented model for digital libraries of documents. *Concurrency and Computation: Practice and Experience*, 13(11), 2001.

[4] S. S. Chawathe, S. Abiteboul, and J. Widom. Representing and querying changes in semistructured data. In *Proceedings of the Fourteenth International Conference on Data Engineering*. IEEE Computer Society, 1998.

[5] S. S. Chawathe, S. Abiteboul, and J. Widom. Managing historical semistructured data. *TAPOS*, 5(3), 1999.

[6] S.-Y. Chien, V. J. Tsotras, and C. Zaniolo. A comparative study of version management schemes for XML documents (short version published at WebDB 2000). Technical Report TR-51, TimeCenter, 2000.

[7] S.-Y. Chien, V. J. Tsotras, and C. Zaniolo. Version management of XML documents: Copy-based versus edit-based schemes. In *Proceedings of the 11th International Workshop on Research Issues on Data Engineering: Document management for data intensive business and scientific applications (RIDE-DM'2001)*, 2001.

[8] G. Cobena, S. Abiteboul, and A. Marian. Detecting changes in XML documents. In *Proceedings of the 18th International Conference on Data Engineering*, 2002.

[9] L. Fegaras and R. Elmasri. A temporal object query language. In *Proceedings of the Fifth International Workshop on Temporal Representation and Reasoning*, 1998.

[10] F. Grandi and F. Mandreoli. The valid web: it's time to go. Technical Report TR-46, TimeCenter, 1999.

[11] F. Grandi and F. Mandreoli. The valid web: An XML/XSL infrastructure for temporal management of web documents. In *Proceedings of Advances in Information Systems, First International Conference, ADVIS 2000*, 2000.

[12] A. Marian, S. Abiteboul, G. Cobena, and L. Mignet. Change-centric management of versions in an XML warehouse. In *Proceedings of VLDB 2001*, 2001.

[13] K. Nørvåg. Temporal query operators in XML databases. In *Proceedings of the 17th ACM Symposium on Applied Computing (SAC'2002)*, 2002.

[14] B. Oliboni, E. Quintarelli, and L. Tanca. Temporal aspects of semistructured data. In *Proceeding of TIME-01*, 2001.

[15] C. Z. Shu-Yao Chien, Vassilis J. Tsotras. Efficient management of multiversion documents by object referencing. In *Proceedings of VLDB 2001*, 2001.

[16] A. Steiner. *A Generalisation Approach to Temporal Data Models and their Implementations*. PhD thesis, Swiss Federal Institute of Technology, 1998.

[17] A. Theobald and G. Weikum. Adding relevance to XML. In *WebDB (Informal Proceedings)*, 2000.

[18] World-Wide Web Consortium. XML query algebra, working draft, February 2001 (most recent version available at http://www.w3.org/TR/query-algebra/).

[19] World-Wide Web Consortium. XML query data model, working draft, February 2001 (most recent version available at http://www.w3.org/TR/query-datamodel/).

[20] World-Wide Web Consortium. XQuery: A query language for XML, February 2001 (most recent version available at http://www.w3.org/TR/xquery/).

[21] L. Xyleme. A dynamic warehouse for XML data of the web. *IEEE Data Engineering Bulletin*, 24(2), 2001.