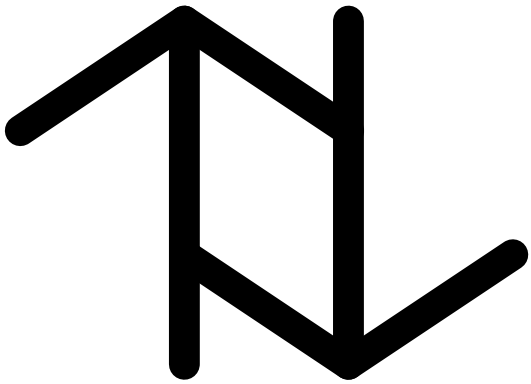

Using XML to Build Efficient Transaction-Time Temporal Database Systems on Relational Databases

Fusheng Wang Xin Zhou Carlo Zaniolo

March 2, 2005

TR-81

A TIMECENTER Technical Report

Title	Using XML to Build Efficient Transaction-Time Temporal Database Systems on Relational Databases
Author(s)	Fusheng Wang Xin Zhou Carlo Zaniolo
Publication History	March 2005. A TIMECENTER Technical Report.

TIMECENTER Participants

Aalborg University, Denmark
Christian S. Jensen (codirector), Simonas Šaltenis, Janne Skyt, Giedrius Slivinskas, Kristian Torp

University of Arizona, USA
Richard T. Snodgrass (codirector), Faiz A. Currim, Sabah A. Currim, Bongki Moon, Sudha Ram, Stanley Yao

Individual participants
Yun Ae Ahn, Chungbuk National University, Korea; Michael H. Böhlen, Free University of Bolzano, Italy; Curtis E. Dyreson, Washington State University, USA; Dengfeng Gao, Indiana University South Bend, USA; Fabio Grandi, University of Bologna, Italy; Heidi Gregersen, Aarhus School of Business, Denmark; Vijay Khatri, Indiana University, USA; Nick Kline, Microsoft, USA; Gerhard Knolmayer, University of Bern, Switzerland; Carme Martín, Technical University of Catalonia, Spain; Thomas Myrach, University of Bern, Switzerland; Kwang W. Nam, Chungbuk National University, Korea; Mario A. Nascimento, University of Alberta, Canada; John F. Roddick, Flinders University, Australia; Keun H. Ryu, Chungbuk National University, Korea; Dennis Shasha, New York University, USA; Michael D. Soo, amazon.com, USA; Andreas Steiner, TimeConsult, Switzerland; Paolo Terenziani, University of Torino, Italy; Vassilis Tsotras, University of California, Riverside, USA; Fusheng Wang, Siemens Corporate Research, USA; Jef Wijsen, University of Mons-Hainaut, Belgium; and Carlo Zaniolo, University of California, Los Angeles, USA

For additional information, see The TIMECENTER Homepage:
URL: <http://www.cs.aau.dk/TimeCenter>

The TIMECENTER icon on the cover combines two "arrows." These "arrows" are letters in the so-called *Rune* alphabet used one millennium ago by the Vikings, as well as by their precedessors and successors. The Rune alphabet (second phase) has 16 letters, all of which have angular shapes and lack horizontal lines because the primary storage medium was wood. Runes may also be found on jewelry, tools, and weapons and were perceived by many as having magic, hidden powers.

The two Rune arrows in the icon denote "T" and "C," respectively.

Abstract

Better support for temporal applications by database systems represents an important technical objective that is difficult to achieve since it requires an integrated solution for several problems, including (i) expressive temporal representations and data models, (ii) powerful languages for temporal queries and snapshot queries, (iii) indexing, clustering and query optimization techniques for managing temporal information efficiently, and (iv) architectures that bring together the different pieces of enabling technology into a robust system. In this paper, we present the ArchIS system that achieves these objectives by supporting a temporally grouped data model on top of RDBMS. ArchIS' architecture uses (a) XML to support temporally grouped (virtual) representations of the database history, (b) XQuery to express powerful temporal queries on such views, (c) temporal clustering and indexing techniques for managing the actual historical data in a RDBMS, and (d) SQL/XML for executing the queries on the XML views as equivalent queries on the relational DB. The performance studies presented in the paper show that ArchIS is quite effective at storing and retrieving under complex query conditions the transaction-time history of relational databases. By supporting database compression as an option, ArchIS also assures excellent storage efficiency for archived histories. This approach achieves full-functionality transaction-time databases without requiring temporal extensions in XML or database standards.

1 Introduction

The interest in and user demand for temporal databases have only increased with time [1]; unfortunately, DBMS vendors and standard groups have not moved aggressively to extend SQL with support for transaction-time or valid-time. Given the strong application demand and the significant research efforts spent on these problems [2], the lack of viable solutions suggests that (i) the technical challenges posed by the problem are many and severe, (ii) their severity is exacerbated by the inflexibility of the relational data model and the lack of extensibility of SQL, and (iii) major R&D costs are expected to add temporal support to RDBMS by directly extending SQL. In this paper, we instead introduce a novel low-cost solution, by showing how XML and its query languages can be used to overcome most of these difficulties, and propose transaction-time extensions for database systems that require no modification of existing standards. Indeed, unlike the relational model, XML provides excellent support for temporally grouped data models, which have long been advocated as the most natural and effective representations of temporal information [3]. Moreover, unlike SQL, XQuery [4] is Turing-complete and natively extensible [5, 6]. Thus many additional constructs needed for temporal queries can be defined in XQuery itself, without having to depend on difficult-to-obtain extensions by standard committees. Therefore, while the challenge of expressing and supporting complex temporal queries should never be underestimated, in this paper we will show that the additional complexity of going from standard queries into temporal ones is much less when starting from XML/XQuery than when starting from relational tables and SQL.

This situation creates the unique opportunity of bringing much needed temporal database support to the users, since database vendors, while torpid on temporal extensions for RDBMS, are moving feverishly to add support for XML and XQuery to their systems. For instance, most database systems support the viewing of the database through XML views that can be queried using XQuery and other languages. These queries are then supported by mapping them into equivalent queries on the underlying database [7, 8]. Database vendors and standard groups are adding these capabilities to SQL through the SQL/XML initiative [9, 10].

In this paper, we propose a very useful generalization of this idea, by showing that the evolution history of a relational database can also be viewed naturally using XML and queried effectively using XQuery. Moreover, the ArchIS system discussed in this paper demonstrates that the temporal data and temporal queries can be supported efficiently via the data-compression, clustering, indexing and query-mapping techniques discussed in the paper.

The paper is organized as follows. After a discussion of related work in the next section, in Section 3 we show that XML provides a natural vehicle for implementing a temporally grouped data model for

representing the evolution history of a relational database. In fact, in Section 4 we show that complex temporal and snapshot queries can be expressed on such views using XQuery. In Section 5 and 6, we focus on the efficient implementation of such queries on a RDBMS, where queries on the XML views are translated into SQL/XML queries on the relational tables, and various indexing/clustering techniques are used to make the execution of these queries efficient. Query performance study in Section 7 shows that ArchIS is quite effective, and in Section 8, we propose database compression as an option, and present a simple but effective technique for compressing archived databases.

2 Related Work

Time in XML

Some interesting research work has recently focused on the problem of representing historical information in XML. In [11], valid time on the Web is supported by proposing a new <**valid**> markup tag for XML/HTML documents, thus temporal visualization can be implemented on web browsers with XSL. In [12], a dimension-based method is proposed to manage changes in XML documents, however how to support queries is not discussed.

There are other approaches to support temporal XML documents through extending XML data models or query languages, such as extending XML data model or XPath to support temporal XML documents in [13], [14] and [15]. (In our approach, we instead support XPath/XQuery without any extension to XML data models or query languages.)

A τXQuery language is proposed in [16] to extend XQuery for temporal support, which has to provide new constructs for the language. An archiving technique for scientific data was presented in [17]. The scheme proposed here presents several similarities to that proposed in [17], but also provides full support for XML query languages.

Temporal Databases and Grouped Representations

There is a large number of temporal data models and query languages, including [18, 19]; thus the design space for the relational data model has been exhaustively explored [2]. Clifford et al. [3] classified them as two main categories: *temporally ungrouped* and *temporally grouped* data models, and they showed that the second representation has more expressive power and is more natural since it is history-oriented [3]. TSQL2 [20] tries to reconcile the two approaches [3] within the severe limitations of the relational tables. Our approach is based on a temporally grouped data model, which dovetails perfectly with the hierarchical structure of XML documents.

TimeDB [21] is a layered architecture that translates temporal queries into RDBMS, where temporal data is represented as tuples with intervals, thus temporally ungrouped. Recently Oracle implemented Flashback [22], an advanced recovery technology that allows users to rollback to old versions of tables in case of errors. However, Flashback only provides limited queries, and efficient support of temporal queries is not provided, where retrieval of historical data is through reading update logs.

The use of XML in publishing and querying database history was previously proposed in [23]. No system implementation was however discussed in [23], and neither were the key pieces of the enabling technology that make it run, including SQL/XML, temporal indexing, clustering and compression.

Temporal Clustering

A temporal clustering technique is discussed in [24] to efficiently retrieve the history of versioned XML documents. In this paper, we extend this technique to cluster temporal data in a RDBMS.

id	name	salary	title	deptno	start	end
1001	Bob	60000	Engineer	d01	1995-01-01	1995-05-31
1001	Bob	70000	Engineer	d01	1995-06-01	1995-09-30
1001	Bob	70000	Sr Engineer	d02	1995-10-01	1996-01-31
1001	Bob	70000	TechLeader	d02	1996-02-01	1996-12-31

Table 1: The snapshot history of employees

deptno	deptname	mgrno	start	end
d01	QA	2501	1994-01-01	1998-12-31
d02	RD	3402	1992-01-01	1996-12-31
d02	RD	1009	1997-01-01	1998-12-31
d03	Sales	4748	1993-01-01	1997-12-31

Table 2: The snapshot history of departments

3 Viewing Relation History in XML

Table 1 and Table 2 describe the history of employees and departments as they would be viewed in traditional transaction-time databases [2] using a temporally ungrouped representation, where **`id`** is the key of the table and remains invariant in the history. [1] With this approach, any change in an attribute value will lead to an new history tuple. The drawbacks for this representation is that, i) redundancy information is preserved between tuples, e.g., the name of Bob appeared the same but was stored in all the tuples; and ii) temporal queries need to frequently coalesce tuples. Temporal coalescing is a source of complications in temporal databases, which is complex and hard to scale in RDBMS. For instance, a temporal coalescing query can take more than 20 lines of SQL with SQL92, and the best performance of coalescing on RDBMS is quadratic [26].

These problems can be overcome using a representation where the timestamped history of each attribute is grouped under the attribute [3] (Figure 1 and Figure 2), i.e., value equivalent attribute histories are grouped if the intervals are adjacent or overlap. While the nested representations hard hard to be represented in flat tables, they can be naturally represented by XML-based hierarchical views shown in Figure 3 and Figure 4. We will call these *H-documents* (or *H-views* when these are virtual representations). The root element in an H-document represents the corresponding table's history (the creation and deletion of a table), and its child elements represent the grouped history of attribute values. Each element in an H-document is assigned two attributes **`tstart`** and **`tend`**, to represent the inclusive time-interval of the element. The value of **`tend`** can be set to *now*, to denote the ever-increasing current time. (This will be further discussed in Section 4.3.) Note that there is a *temporal covering constraint* that the interval of a parent node (table history) always covers that of its child nodes (attribute histories). The H-document also has a simple and well-defined schema.

Our H-documents use a temporally grouped data model [3]. Clifford, et al. [3, 27, 28] show that temporally-grouped models are more natural and powerful than temporally ungrouped ones. One benefit of our approach is that it greatly reduces the need for coalescing, since an attribute history is already grouped. Another significant benefit is the effectiveness of expressing complex temporal queries with XQuery, as discussed next.

[1] In the remainder of this paper, our granularity for time is a day; however, all the techniques we present are equally valid for any granularity used by the application. For finer granularity, techniques in [25] can be used. Furthermore, throughout this paper, we assume that relation keys remain invariant.

id	name	salary	title	deptno
1995-05-01	1995-05-01	1995-01-01	1995-01-01	1995-01-01
		60000	**Engineer**	**d01**
		1995-05-31	1995-09-30	1995-09-30
		1995-06-01	1995-10-01	1995-10-01
1001	**Bob**		**Sr Engineer**	
		70000	1996-01-31	**d02**
			1996-02-01	
			Tech Leader	
1996-12-31	1996-12-31	1996-12-31	1996-12-31	1996-12-31

Figure 1: Temporally grouped history of employees

deptno	deptname	mgrno
1994-01-01	1994-01-01	1994-01-01
d01	**QA**	**2501**
1998-12-31	1998-12-31	1998-12-31
1992-01-01	1992-01-01	1992-01-01
		3402
		1996-12-31
d02	**RD**	1997-01-01
		1009
1998-12-31	1998-12-31	1998-12-31
1993-01-01	1993-01-01	1993-01-01
d03	**Sales**	**4748**
1997-12-31	1997-12-31	1997-12-31

Figure 2: Temporally grouped history of departments

4 Temporal Queries using XQuery

The key advantage of our approach is that powerful temporal queries can be expressed in XQuery without requiring the introduction of new constructs in the language. We next show how to express the main classes of temporal queries as discussed in [18, 20]: *temporal projection, temporal snapshot, temporal slicing, temporal join, temporal aggregate,* and *restructuring*, on **`employees.xml`** document(Figure 3) and **`departments.xml`** document(Figure 4).

QUERY 1: Temporal Projection. Retrieve the title history of employee “Bob”:

```
element title_history{
for $t in doc("employees.xml")/employees/
    employee[name="Bob"]/title
return $t }
```

This query shows the benefit of removing coalescing in the query result. Since the history of titles is grouped, the projected result is already coalesced. While for temporally ungrouped data model, coalescing has to be performed on the results.

QUERY 2: Temporal Snapshot. Retrieve the managers on 1994-05-06:

```
<employees tstart="1995-01-01" tend="1996-12-31">
 <employee tstart="1995-01-01" tend="1996-12-31">
  <id tstart="1995-01-01" tend="1996-12-31">1001</id>
  <name tstart="1995-01-01" tend="1996-12-31">Bob</name>
  <salary tstart="1995-01-01" tend="1995-05-31">60000</salary>
  <salary tstart="1995-06-01" tend="1996-12-31">70000</salary>
  <title tstart="1995-01-01" tend="1995-09-30">Engineer</title>
  <title tstart="1995-10-01" tend="1996-01-31">Sr Engineer</title>
  <title tstart="1996-02-01" tend="1996-12-31">Tech Leader</title>
  <deptno tstart="1995-01-01" tend="1995-09-30">d01</deptno>
  <deptno tstart="1995-10-01" tend="1996-12-31">d02</deptno>
 </employee>
 <!-- ...  -->
</employees>
```

Figure 3: The history of the **employee** table is viewed as **employees.xml**

```
<depts tstart="1991-01-01" tend="1998-12-31">
 <dept tstart="1994-01-01" tend="1998-12-31">
  <deptno tstart="1994-01-01" tend="1998-12-31">d01</deptno>
  <deptname tstart="1994-01-01" tend="1998-12-31">QA</deptname>
  <mgrno tstart="1994-01-01" tend="1998-12-31">2501</mgrno>
  </dept>
 <dept tstart="1992-01-01" tend="1998-12-31">
  <deptno tstart="1992-01-01" tend="1998-12-31">d02</deptno>
  <deptname tstart="1992-01-01" tend="1998-12-31">RD</deptname>
  <mgrno tstart="1992-01-01" tend="1996-12-31">3402</mgrno>
  <mgrno tstart="1997-01-01" tend="1998-12-31">1009</mgrno>
 </dept>
 <!-- ...  -->
</depts>
```

Figure 4: The history of the **dept** table is viewed as **depts.xml**

```
for $m in doc("depts.xml")/depts/dept/mgrno
    [tstart(.)<=xs:date("1994-05-06") and
    tend(.) >= xs:date("1994-05-06")]
return $m
```

Note that **xs** is the namespace of XML Schema (the declaration of namespaces is ignored here). **tstart($e)** and **tend($e)** are user-defined functions (expressed in XQuery) that get the starting date and ending date of an element respectively, thus the implementation is transparent to users. This will be further discussed in Section 4.2.

QUERY 3: Temporal Slicing. Find employees who worked at any time between 1994-05-06 and 1995-05-06:

```
for $e in doc("employees.xml")/employees
    /employee[ toverlaps(.,
    telement( xs:date("1994-05-06"),
    xs:date("1995-05-06") ) ) ]
```

```
return $e/name
```

Here **toverlaps(**$**a,** $**b)** is a user-defined function that returns true if one node overlaps with another one, and false otherwise, and **telement(**$**a,** $**b)** constructs an element with **a** and **b** as its attributes.

QUERY 4: Temporal Join. Find the history of employees each manager manages:

```
element manages{
for $d in doc("depts.xml")/depts/dept
for $m in $d/mgrno
return
element manage {$d/deptno,  $m,
    element employees {
    for $e in doc("employees.xml")/
        employees/employee
    where  $e/deptno = $d/deptno and
        not(empty(overlapinterval($e, $m) ) )
    return($e/name, overlapinterval($e,$m))
}}}
```

This query will join **depts.xml** and **employees.xml** documents and generate a hierarchical XML document grouped by dept and manager. **overlapinterval(**$**a,** $**b)** is a user-defined function that returns an element **interval** with overlapped interval as attributes **tstart** and **tend**. If there is no overlap, the element is not returned which satisfies the XQuery built-in function **empty(**$**e)**.

QUERY 5: Temporal Aggregate. Retrieve the history of the average salary:

```
let $s := document("emp.xml")/employees/
employee/salary
return tavg($s)
```

Here **tavg(**$**s)** is a user-defined function that can directly computed with XQuery with a single scan. First, a list of salary-timestamp pairs are generated by adding and decreasing the salary value of each interval; then these salaries are sorted by the timestamp, and for each timestamp, all the changes are added up to get a delta sum. If the delta is different from zero, then the old interval is ended and a new one is started, where the new sum is the old one plus the delta.

Other temporal aggregates such as *RISING* or *moving window* aggregate can also be supported through user-defined functions.

QUERY 6: Restructuring. Find the maximum length of time during which Bob worked continuously without changing title or department:

```
for $e in doc("emp.xml")/employees/
    employee[name="Bob"]
let $d := $e/dept
let $t := $e/title
let $overlaps := restructure($d, $t)
```

```
return max($overlaps)
```

The user-defined function **restructure** takes two lists and returns all the overlapped intervals.

4.1 More Complex Queries

Here we discuss more advanced temporal queries, such as *until*, *since*, and *contain*, which are often used as a test for the expressive power of temporal languages [19]. For instance, the following is a *since* query:

QUERY 7: A Since B. Find the employee who has been a Senior Engineer in dept "d001" since he/she joined the dept:

```
for $e in doc("employees.xml")/employees/employee
let $m:= $e/title[.="Sr Engineer" and
    tend(.)=current-date()]
let $d:=$e/deptno[.="d001" and tcontains($m, .)]
where not empty($d) and not empty($m)
return <employee>
{$e/id, $e/name}</employee>
```

Here **tcontains($e)** is a user-defined function to check if one interval covers another.

QUERY 8: Period Containment. Find employees with the same employment history as employee "Bob", i.e., they worked in the same department(s) as employee "Bob" and exactly for the same periods:

```
for $e1 in doc("employees.xml")/employees
    /employee[name = "Bob"]
for $e2 in doc("employees.xml")/employees
    /employee[name != "Bob"]
where  every $d1 in $e1/deptno satisfies
some $d2 in $e2/deptno satisfies
(string($d1)=string($d2) and tequals($d2,$d1))
and every $d2 in $e2/deptno satisfies
some $d1 in $e1/deptno satisfies
(string($d2)=string( $d1) and tequals($d1,$d2))
return <employee>{$e2/name}</employee>
```

Here **tequals($d1,$d2)** is a user-defined function to check if two nodes have equal intervals.

4.2 Temporal Functions

The use of functions **tstart($e)** and **tend($e)** in temporal queries offers the advantage of divorcing the queries from the low-level details used in representing time, e.g., if the interval is closed at the end, or how *now* is represented. Other useful functions predefined in our system include:

Restructuring functions: **coalesce($l)** will coalesce a list of nodes, and **restructure($a,$b)** will return all the overlapped intervals on two set of nodes.

Interval functions: **toverlaps($a,$b), tprecedes($a,$b), tcontains($a,$b), tequals($a,$b), and tmeets($a,$b)** will return true or false according

to two interval positions; The **overlapinterval(**$**a,**$**b)** will return the overlapped interval if they overlap, and the result has the form:

<**interval tstart= "d1" tend="d2" /**>.

Duration and date/time functions:
timespan($**e)** returns the time span of a node;
tstart($**e)** returns the start time of a node;
tend($**e)** returns the end time of a node;
tinterval($**e)** returns the interval of a node;
telement($**Ts,** $**Te)** constructs an empty element **telement** with attributes **tstart** and **tend**;
rtend($**e)** recursively replaces all the occurrence of "9999-12-31" with the value of **current_date**;
externalnow($**e)** recursively replaces all the occurrence of "9999-12-31" with the string "now".

The details of the functions are described in the Appendix.

4.3 Support for 'now'

An important issue in temporal databases is how to handle *now* or *UC* (until changed) [29, 30]. In a transaction-time database, *now* means that the values in the tuple are still current at the time the query is asked. In our strategy, we replace the symbol "now" with the value **current_timestamp** (or **current_date**, depending on the used time granularity). Such instantiation is performed conservatively only when needed.

Internally, we use the "end-of-time" value (e.g., "9999-12-31" for date) to denote the "now" symbol. The user does not access this value directly, he/she will access it through built-in functions **tstart(**$**e)** and **tend(**$**e)**. While the function **tstart(**$**e)** returns the start of the interval, the **tend(**$**e)** function returns its end, if this is different from "9999-12-31" and **current_date** otherwise. This representation can assure that the current search techniques based on indexes and temporal ordering can be used without any change.

The (fragments of) XML documents returned in the output of queries such as QUERY 1, use the "9999-12-31" internal representation for *now*, so that they can be given as input of other temporal queries. However, for data returned to the end-user, two different representations are preferable. One is to return the **current_date** by applying function **rtend(**$**e)** that, recursively, replaces all the occurrence of "9999-12-31" with the value of **current_date**. The other is to return a special string, such as "now" or "until-changed" to be displayed on the end-user screen. As discussed in [29], this is often the more intuitive and appealing for users, and is supported by our built-in function **externalnow(**$**e)** that does that for the node **e** and its sub-nodes.

5 The ArchIS System

Two approaches are possible for storing and querying H-documents: one is to use a native XML DBMS such as Tamino XML Server [31]; the other is to use RDBMSs and provide mappings of queries and query results between the XML views and the underlying database systems. The query performance and the storage efficiency of the two approaches are compared in Section 7.

The main design issues that must be addressed for an efficient realization of the second approach include:

- how to map (shred) the XML views representing the H-documents into tables (which we call *H-tables*),
- how to translate queries from the XML views to the H-tables, and
- which indexing, clustering and query mapping techniques should be used for high performance.

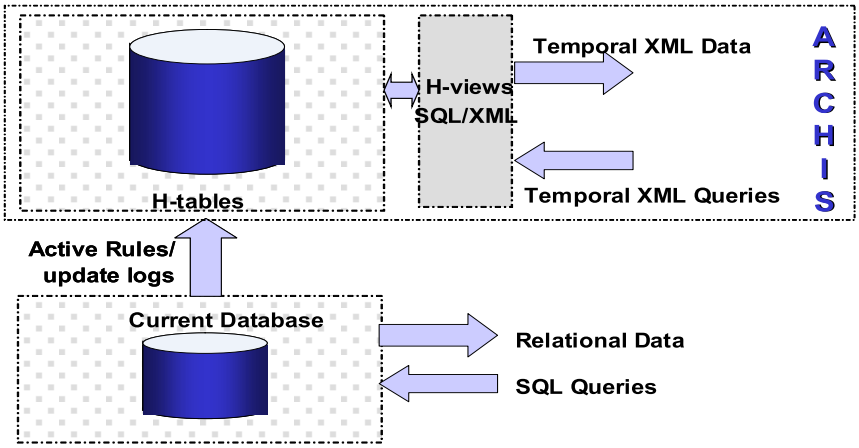

Figure 5: ArchIS: Archival Information System

We will next discuss the solutions of these problems used in our *Archival Information System—ArchIS*, which uses the RDBMS-based approach (*ArchIS-DB2* on DB2 and *ArchIS-ATLaS* on ATLaS [32]). The architecture of ArchIS is shown in Figure 5. In our implementation, the 'current database' and H-tables are implemented as tables in a same database, but the results are easily generalized to the situations where these two are separate, or even the case where the current database is a view containing the *now* snapshot of the H-tables.

5.1 H-tables

Each H-document is stored in the database as internal H-tables. For each table in the current relational database we store a key table and several attribute history tables. An attribute history table is built for each attribute to store the history of such attribute. A key table is built for the key. Each table will include two attributes **`tstart`** and **`tend`** to represent the valid interval of that tuple. Besides, a global relation table is used to record the history of relations.

For example, we have the following relation in the current database:

```
employee(id, name, salary, title, deptno)
```

where **`id`** is the key. The history of the table is viewed as an H-document, which is then decomposed as the following tables in ArchIS:

The Key Table:

```
employee_id(id, tstart, tend)
```

Since **`id`** will not change along the history, the interval **`(tstart, tend)`** in the key table also represents the valid interval of the employee.

For composite keys, for example, (`supplierno, itemno`), we build a key table as `lineitem_id(id, supplierno, itemno, tstart, tend)`, where `id` is a unique value generated from (`supplierno, itemno`). The use of keys is for easily joining of all attribute histories of an object such as an employee.

Attribute History Tables:

```
employee_name(id,name,tstart,tend)
...
employee_deptno(id, deptno, tstart,tend)
employee_salary(id, salary, tstart,tend)
employee_title(id,  title,  tstart,tend)
```

The values of **id**s in the above tables are the corresponding key values, thus indexes on such **id**s can efficiently join these relations.

A sample content of the **employee_salary** table is:

```
ID     SALARY    TSTART      TEND
================================
100022 40000  02/20/1988 02/19/1989
100022 42010  02/20/1989 02/04/1990
100022 42525  02/20/1990 02/04/1991
100022 42727  02/20/1991 02/19/1992
...
100023 43162  07/13/1988 07/13/1989
...
```

When a new tuple is inserted, the **TSTART** for the new tuple is set to the current timestamp, and **TEND** is set to *now*. When there is a delete on a current tuple, we simply change the **TEND** value in that tuple as current timestamp. An update can be viewed as a delete followed by an insert.

Global Relation Table:

relations(relationname, tstart, tend)

will record all the relations history in the database schema, i.e., the time spans covered by the various tables in the database. This corresponds to the root elements of H-documents.

Our design builds on the assumption that keys (e.g., **empno**) remain invariant in the history. Otherwise, a system-generated surrogate key can be used.

5.2 Updating Table Histories

Changes in the current database can be tracked with either update logs or triggers. For our testing on ArchIS-DB2, we build triggers that successfully track changes in the current database and archive them into H-tables. For ArchIS-ATLaS, for better performance, we use update logs to track and archive changes.

5.3 Query Mapping

Middleware such as XPERANTO [7] could be used to publish relational databases the underlying content of our H-tables into XML, and of queries on such tables. Very general translation mechanisms from XML documents to RDBMS have been studied in [33]. For the case at hand, however, we prefer to use optimized strategies that exploit the simple and well-defined mappings that relate the external H-documents (actually, H-views since they are virtual objects) with the underlying H-tables to achieve better performance.

The studies presented in [34] show that the best performance is obtained when the XML documents are constructed inside the relational engine. This high-performance approach can be achieved using SQL/XML [9,

10], a new standard widely supported by database vendors, where both tag-binding and structure construction is pushed inside the relational engine. Therefore, ArchIS implements XQuery on H-views, by translating them into equivalent SQL/XML expressions on H-tables. The expressions on H-tables use the SQL/XML constructs **`XMLElement`**, **`XMLAttributes`**, and **`XMLAgg`**, which are discussed next.

The **`XMLElement`** and **`XMLAttributes`** constructs are used to return elements and their attributes. **`XMLAgg`** is an aggregate function, which constructs an XML value from a collection of XML value expressions. For instance, to return an **`new_employees`** element containing all the employees hired after 02/04/2003, we can write the following SQL/XML query:

```
select XMLElement (Name "new_employees",
XMLAttributes ("02/04/2003" as "start"),
XMLAgg (XMLElement (Name "employee", e.name))
from employee_name as e
where e.tstart >= "02/04/2003"
```

Assuming that only Bob and Jack were hired after 02/04/2003, the previous query returns the following output:

```
<new_employees start = "02/04/2003">
<employee>Bob</employee>
<employee>Jack</employee>
</new_employees>
```

These SQL/XML constructs simplify the translation from queries expressed on H-views to equivalent queries on H-tables. For instance, the SQL/XML translation of QUERY 1 in Section 4 is shown below:

```
select XMLElement (Name "title_history",
XMLAgg (XMLElement (Name "title",
    XMLAttributes (T.tstart as "tstart",
    T.tend as "tend"), T.title)))
from employee_title as T, employee_name as N
where N.id = T.id and N.name = "Bob"
group by N.id
```

Notice that the **`N.id = T.id`** condition in the **`where`** clause is generated due to the **`[name="bob"]`** predicate in the XPath expression. A **`group by`** clause is also added to group all titles of an **`id`** into an element through the **`XMLAgg()`** function.

As another example, QUERY 3 will be translated to:

```
select XMLElement (Name "emp",
XMLElement (Name "id", XMLAttributes (
e.tstart as "tstart",e.tend as "tend"),e.id),
XMLElement (Name "name", XMLAttributes(
n.tstart as "tstart",n.tend as "tend"),n.name))
from employee_id e, employee_name n
where e.id = n.id  and toverlaps( e.tstart,
e.tend, "1994-5-06", "1995-5-06")
```

Here a join condition is needed to join **e.id** with **n.id**, which is implied in the XPath expression $**e/name**. The translation of UDF (user-defined function) **toverlaps** takes in the **tstart** and **tend** values, and returns true or false. More on built-in function translation is discussed in Section 5.4.

The mapping of queries on H-views to H-tables can be summarized as five main steps:

- Identification of variable range: For each variable defined by a **for** or **let** expression in the original query, we identify whether, in the underlying H-tables, this corresponds to (i) a tuple variable ranging over a key relation, or (ii) a tuple variable ranging over an attribute table, or (iii) an attribute variable such as **T.A** where **T** is a tuple variable over a key table or an attribute table, and **A** denotes an attribute in such tables. For each distinct tuple variable in the original query, a distinct tuple variable is created in the **from** clause of the SQL/XML query.

  For instance, QUERY 1 identifies two attribute variables, from tables **employee_title** and **employee_name**. Therefore, the **from** clause of the SQL/XML statement contains such two tuple variables with aliases **T** and **N**.

- Generation of join conditions: There is a join condition **T.id** and **N.id** for any pair of distinct tuple variables.

- Generation of the **where** conditions: these are the conditions that are contained in the **where** clause of the XQuery or specified in the path expression (e.g., **[name="Bob"]** in QUERY 1).

- Translation of built-in functions: Temporal functions (such as **toverlaps(**$**a,**$**b)**) are simply mapped into the corresponding built-ins we have implemented for ArchIS. We will have more discussion for function mapping in the next section.

- Output generation: This is achieved through the use of the **XMLElement** and the **XMLAgg** constructs previously described. Through expression of these constructs the ArchIS compiler supports simple expressions, such as **return** $**t** of QUERY 1, and more complex expressions such as QUERY 4. Meanwhile, users have the opiton to specify a **table** construct in the return clause to bypass the SQL/XML transformation, so the results can be returned as tables.

The algorithm is described in 1, and is implemented based on the code of Galax [35], an open source implementation of XQuery.

The translated SQL/XML queries on the H-tables often contain many natural joins such as **N.id = T.id**. These joins execute very fast (in linear time) since every table is already sorted on its **id** attribute.

Based on the simple mapping relationship between H-view and H-tables, and efficient execution of SQL/XML publishing functions inside SQL engine [34], our query translation is very efficient (see performance in Section 7).

5.4 Function Mapping

User-defined temporal functions discussed in Section 4.2 are implemented as equivalent functions in ArchIS. This simplifies the mapping from XQuery to SQL. There are two situations for function mapping, based on node types in H-views:

Leaf nodes: For leaf nodes such as **id** or **salary**, we can identify a tuple variable **T** over either a key table or an attribute history table. As a result, we can take the **T.tstart** and **T.tend** attribute variables as input to the UDF, and implement the functionality of the UDF.

Parent nodes of leaf nodes: For example, the **employee** node in our H-document is a parent node of leaf nodes. Since the **tstart** and **tend** attribute values are the same as those of their **id** child nodes, we can identify the tuple variable **T** over their key table, and pass **T.tstart** and **T.tend** as the input of the UDF.

Algorithm 1 Mapping XQuery to SQL/XML

1: **for** each variable $\$v_i$ in **`for`** and **`let`** clause of XQuery **do**
2: Find table T_i and Column A_i according to schema mapping
3: **end for**
4: **for** any variable v_j which is defined by a relative XPath from v_i, such as $v_j := v_i/salary$ **do**
5: Generate join condition $C_{ij-id} : V_i.id = V_j.id$
6: **end for**
7: **for** every condition V_i op V_j in **`where`** clause of XQuery **do**
8: Generate condition $C_{ij-where}$: $T_i.A_i$ op $T_j.A_j$
9: **end for**
10: **for** every function in XQuery $fn(\$v_i, \$v_j)$ **do**
11: Generate function fn_{ij}($T_i.A_i$, $T_i.tstart$, $T_i.tend$, $T_j.A_j$, $T_j.tstart$, $T_j.tend$)
12: **end for**
13: **for** every $\$v_i$ in **`return`** clause **do**
14: Generate E_i: XMLElement (Name v_i's element name, XMLAttributes (T_i.tstart as "tstart", T_i.tend as "tend"), $T_i.A_i$)
15: **end for**
16: **for** every parent-child relationship $v_i(v_j)$ in **`return`** clause **do**
17: Generate E_{ij}: XMLElement (Name v_i's element name, XMLAttributes (T_i.tstart as "tstart", T_i.tend as "$tend$"), E_j, $T_i.A_i$)
18: **end for**
19: Generate output SQL:
$SELECT\ (\cup_i E_i) \cup (\cup_{i,j} E_{ij})$
$From\ \cup_i T_i$
$WHERE\ (\cup_{i,j} C_{ij-id}) \cup (\cup_{i,j} C_{ij-where})$

Temporal aggregate functions such as **`tavg`** in QUERY 5 of Section 4 can be effectively mapped into SQL 2003 OLAP functions [36]. In addition, XQuery built-in functions [37] will also be supported in ArchIS in the future, based on the above mapping strategies.

6 Temporal Clustering and Indexing

In our current RDBMS-based archiving scheme, tuples are stored in a temporally grouped order (i.e., the salary history of an employee before that of the next employee). Performance on snapshot queries can be improved with a more effective temporal clustering scheme. Thus, we use a segment-based archiving scheme which has better temporal clustering, and will boost the performance of most temporal queries, and is also amendable to compression techniques.

6.1 Usefulness-Based Clustering

Assume that an attribute history is stored in a segment. For each segment, we can always define its usefulness as $U = N_{live}/N_{all}$, where N_{live} is the count of live(or current) tuples and N_{all} is the count of all tuples. U begins with 100% and decreases with updates. We also define a minimum tolerable usefulness U_{min}.

Initially all tuples in an attribute history table are archived in a live segment SEG_{live} with usefulness $U = 100\%$. Updates will be performed on the live segment, and when U drops below U_{min}, we perform

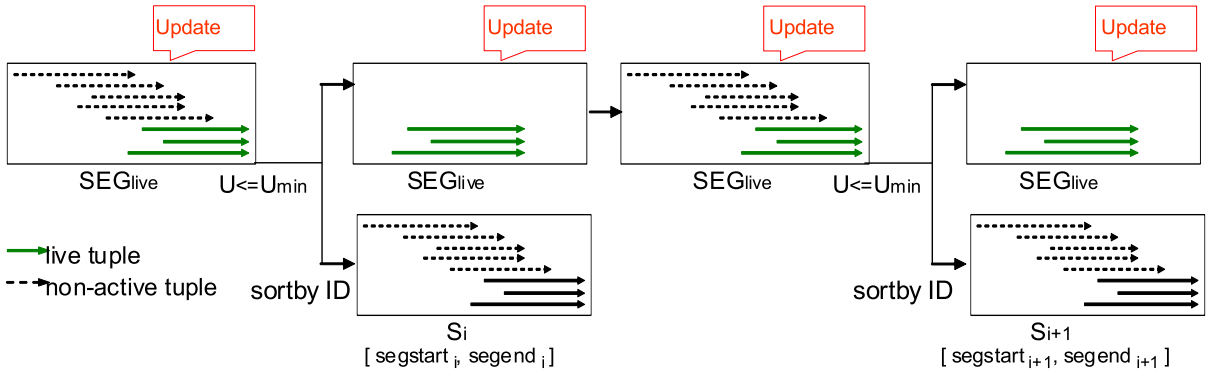

Figure 6: Segment-based clustering

the following operations:

1. A new segment S_i is allocated;
2. The interval of this segment is recorded in the table **`segment (segno, segstart, segend)`**, where **`segstart`** and **`segend`** record the starting and ending time for the segment respectively;
3. All tuples in SEG_{live} are copied into a new segment S_i, sorted by **`ID`**;
4. All live tuples in SEG_{live} are copied into a new live segment $SEG_{live'}$, and the old live segment is dropped.

After these operations are completed, segment $SEG_{live'}$ becomes the new starting segment for updates, and the process repeats. The process is illustrated in Figure 6.

As an example, the segment-based scheme for **`employee_salary`** table will be clustered on **`segno`**, as shown follows:

```
SegNo       ID     SALARY        TSTART      TEND
=============================================
001       100022   40000     02/20/1988 02/19/1989
001       100022   42010     02/20/1989 02/04/1990
001       100022   42525     02/20/1990 02/04/1991
001       100022   42727     02/20/1991 12/31/9999
002       100022   42727     02/20/1991 02/19/1992
```

And the content in **`segment`** table will be:

```
    SegNo      segstart      segend
=======================================
    001       01/01/1985   10/17/1991
    002       10/18/1991   07/08/1995
    ...
```

An important feature of this usefulness-based clustering is, the following two conditions are always satisfied for any tuple in a segment:

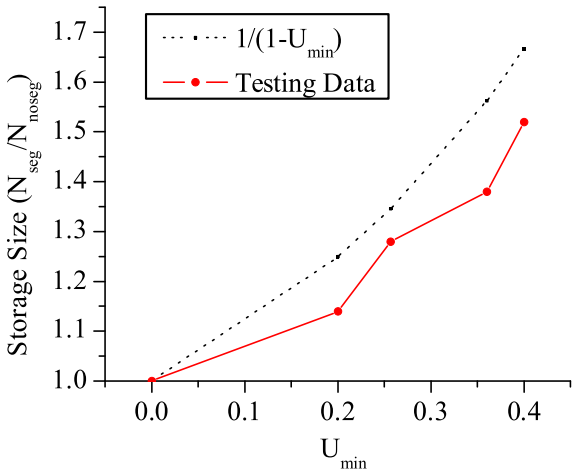

Figure 7: Storage sizes for different U_{min}

$$tstart_{tuple} \leq segend_{SEG} \tag{1}$$

$$tend_{tuple} \geq segstart_{SEG} \tag{2}$$

There are several advantages for segment-based clustering: First, the current live segment always has a high usefulness, which assures effective updates; second, records are globally temporally clustered on segments; third, for snapshot queries, only one segment is used, and for temporal slicing queries, only segments involved are used, thus such queries can be more efficient, as discussed in Section 7.1; and last, we have the flexibility to control the number of redundant tuples in segments by U_{min}, as discussed next.

6.2 Storage Usage

Assume all segments have usefulness U_{min}, thus the total number of invalid (non-current) tuples of all segments are (1 - U_{min}) $\times$ N_{seg}, where N_{seg} is the total number of tuples in archived segments. Assume for the worst case, all tuples (N_{noseg}) in the original relation (without segmentation) become invalid, then $N_{noseg} \geq (1 - U_{min}) \times N_{seg}$, or:

$$\frac{N_{seg}}{N_{noseg}} \leq \frac{1}{1 - U_{min}} \tag{3}$$

Figure 7 shows the ratio of storage size with different U_{min}, compared to that without segmentation.

When U_{min} increases, the number of segments increases, and the storage overhead increases as well. There are 3 segments when U_{min} = 0.2, 5 segments when U_{min} =0.26, 7 segments when U_{min}=0.36, and 9 segments when U_{min} = 0.4. Observe that the storage overhead for U_{min} =0.26 is about the same as for database without segmentation, since the average storage utilization is 75% in the situation where records are inserted into arbitrary pages in the file, rather than appended at the end.

The segment-based clustering can boost the performance of most temporal queries and is amendable for efficient compression, which will be discussed in the coming section.

The length of a segment T_{seg} is determined by U_{min} and also the update rates. Suppose the rates for insertion, deletion, and update are R_{ins}, R_{del}, and R_{upd} respectively, and the count of tuples at the beginning

of a segment is N_0 (with U_{min}=100%), and by estimating the count of live tuples at the end of the segment we get:

$$T_{seg} = \frac{N_0(1 - U_{min})}{U_{min}R_{upd} - (1 - U_{min})R_{ins} + R_{del}} \quad (4)$$

Thus higher update rate and/or deletion rate will lead to shorter segment, and higher insertion rate will lead to longer segment. By rewriting the equation, we can also find out that higher usefulness threshold will lead to shorter segment.

6.3 Query Mapping with Clustering

In Section 5.3, we have discussed the general mapping between XQuery and SQL/XML, whereby XQuery upon H-document is translated to SQL/XML upon H-tables. We can now modify our queries in such a way that, when the **`tstart`** and **`tend`** conditions are specified, we first find the segment number satisfying those conditions and then we use that to restrict the search to only segment(s) of the historical database involved in the query. This operation is made very efficient by the fact that all indexes are now augmented with a **`segno`** information.

For example, for snapshot query QUERY 2, first, the segment number **`sn`** of the segment which contains the timestamp **`1994-05-06`** is searched in the **`segment`** table, then the SQL query is modified by adding the segment number condition to shrink the search space:

```
select XMLElement(Name "mgrno", XMLAttributes(
m.tstart as"tstart", m.tend as "tend"), m.mgrno)
from dept_mgrno as m where m.segno = sn and
m.tstart<="1994-05-06" and m.tend>="1994-05-06"
```

Observe that, unless the number of segments becomes very large and exceeds the number of main-memory blocks available for sort-merge joins, joining H-tables remains a very efficient one-pass operation.

7 Performance Study

We investigate three systems for archiving: native XML database Tamino (Enterprise Edition V4.1); ArchIS-DB2 built on RDBMS DB2 (DB2 Enterprise Edition V7.2), and ArchIS-ATLaS built on ATLaS [32]. ATLaS is a compact RDBMS developed at UCLA that uses BerkeleyDB [38] as the storage manager and builds on top of it a SQL query engine. Both ArchIS-DB2 and ArchIS-ATLaS use the same same approach discussed in Section 5. The experiments are performed on a Pentium IV 2.4GHz PC with RedHat 8.0, with 256MB memory and an 80GB ATA hard drive.

We use the temporal employee data set [39] for our testing. The data set models the history of employees over 17 years, and simulates the increases of salaries, changes of titles, and changes of departments. The total size of the published XML documents from the history data is 334MB. To test the scalability of our system, we also use another data set of 2.28GB (7 times larger), as discussed later in this section.

To avoid OS caching and database buffer pool caching, we take two effective methods. To disable Linux OS caching, the hard drive with data is unmounted, which leads to invalidation of Linux pagecache [40]. Then the drive is remounted before running each query. To disable database buffer caching, databases are restarted for each query.

We also studied the performance of queries with variable memory sizes: 256MB, 512MB and 1GB. Since caching is effectively disabled, there is no difference on the query performance.

In the following performance study, each query is executed 7 times and the results are averaged.

Q1:	Snapshot(single object): find the salary of an employee 100002 on 05/16/1993;
Q2:	Snapshot: find the average salary of employees on 05/16/1993;
Q3:	History(single object): find the salary history of employee '100002';
Q4:	History: find the total number of salary changes;
Q5:	Temporal slicing: find the number of employees whose salary was more than 60K between 05/16/1993 and 05/16/1994;
Q6:	Temporal join: find the maximum salary increase over a two years period after 04/01/2001.

Table 3: Temporal queries on archived history

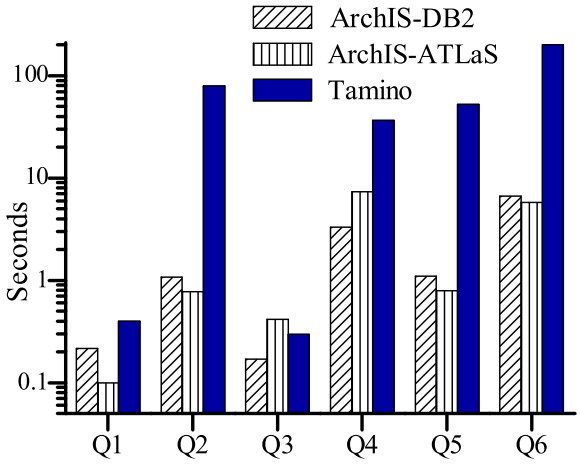

Figure 8: Query performance of segment-based archiving on RDBMS vs native XML DB

7.1 Query Performance

We investigate three systems to test the query performance: Tamino with H-documents, ArchIS-DB2 with segmented data, and ArchIS-ATLaS with segmented data (with U_{min} as 0.4 and 9 segments). On Tamino, the documents are automatically compressed for performance (data compression will be further discussed in Section 8). On ArchIS-DB2 and ArchIS-ATLaS, the data are stored as H-tables clustered on segments.

We prepare a set of typical temporal queries such as snapshot (on a single object and on all objects), temporal slicing(on a single object and on all objects), history, and temporal join, as shown in Table 3. In addition, a set of indexes are built for later query comparisons: indexes are created for all nodes/attributes which have values selected.

Figure 8 shows the query performance on the three systems. The results suggest that RDBMSs offer substantial performance advantage over a native XML DB for most queries. The difference of snapshot queries between RDBMS and native XML databases are more significant. For instance, snapshot query Q2 on ArchIS-ATLaS is 102 times faster than that on Tamino, and temporal slicing query Q5 is 66 times faster. History query Q4 on ArchIS-ATLaS is nearly 4 times faster, and temporal join Q6 is 35 times faster. Temporal aggregate queries were not compared given that they require recursive user-defined function in XQuery which are not yet supported in the current version of Tamino. However, we were able to support them efficiently on the RDBMSs using OLAP functions.

The better performance obtained from relational systems is partially due to the use of the segment-based archiving, while no segment-based archiving was used with Tamino. Actually, we experimented with the temporal clustering scheme in Tamino, but these failed to produce significant performance improvements. The problem of introducing effective temporal clustering and indexing schemes into native XML systems

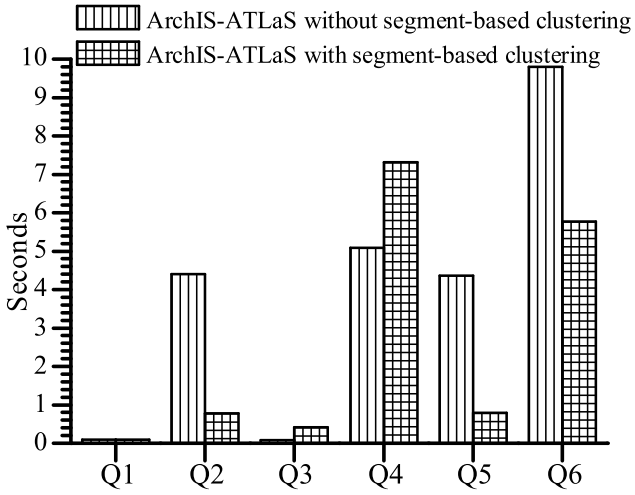

Figure 9: Query performance with and without segment-based clustering

is left for further research.

Query Translation Cost

Our query mapping is based on the simple relationship mapping between H-view and H-tables, and the translation is very efficient. For each of the 6 example queries in XQuery, the translation cost is less than 0.1ms.

The Effect of Segment-based Clustering

Figure 9 shows the performance of such queries on data with segment-based clustering (with U_{min} as 0.4 and 9 segments) versus without clustering on ArchIS-ATLaS. This shows that the segment-based clustering scheme significantly boosts the speed for snapshot and temporal slicing queries, e.g., snapshot query Q2 is 5.7 times faster on clustered data than non-clustered data, while temporal slicing query Q5 is 5.5 times faster. Temporal join Q6 is 1.7 times faster with segment-based clustering. The speeds of temporal queries (both Q1 and Q3) on a single object are close for clustering and without clustering due to the effectiveness of B+ tree index on object IDs. An exception is Q4, which is slower due to the scanning of the whole historical data, and the clustered scheme has a storage redundancy.

Performance on Snapshot

We also validate the performance of our clustering scheme by comparing the snapshot query Q2 with the one that directly executes on the current database: the former runs 27% slower than the latter. This is consistent with the storage overhead in archived segments caused by usefulness.

Scalability of ArchIS

To test the scalability of the performance on RDBMSs, we generate a new data set 7 times larger (2,338MB), and load it into RDBMS as clustered segments. Figure 10 show that the query execution time of most queries increases approximately linearly. For temporal queries on single object –Q1 and Q3, the time increase is even much less.

7.2 Storage Utilization

We also investigate the storage utilization on the three systems, and find that Tamino is very efficient in this respect, since Tamino automatically compresses documents with an algorithm similar to gzip.

Figure 11 shows the compression ratios (final storage size over H-document size) for the three systems.

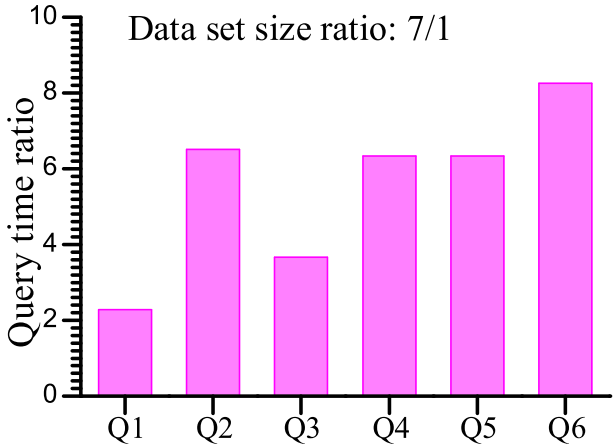

Figure 10: Query time comparison on ArchIS-DB2 between two data sets (with size ratio: 7/1)

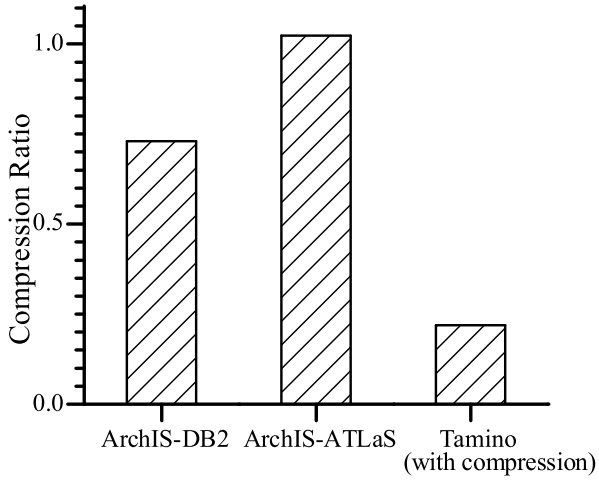

Figure 11: Compression ratios of H-document storage on different systems

The compression ratio on Tamino is 0.22. The storage size in H-tables is half of the H-document size. But with further segment-based clustering, there are redundant tuples among different segments, and the clustering index will take an additional overhead. As a result, ArchIS-DB2 has a compression ratio of 0.75, and ArchIS-ATLaS of 1.02 (ArchIS-ATLaS' storage manager BerkeleyDB uses clustered index which causes extra overhead on storage).

In the next section, we show that by compressing data in RDBMS as an option, we can reduce the storage significantly and the compression ratio in a RDBMS can reach that of Tamino, while we still maintain efficient query performance.

8 Database History Compression

RDBMS compression techniques have been investigated in the past[41, 42, 43], and most of the work focused on field level or row level compression. In [43], a page-based compression was proposed with vector quantization technique. Recently, Oracle introduces a dictionary-based compression [44], which is efficient on data warehouses. The disparity between CPU/memory and disk speeds is becoming larger and

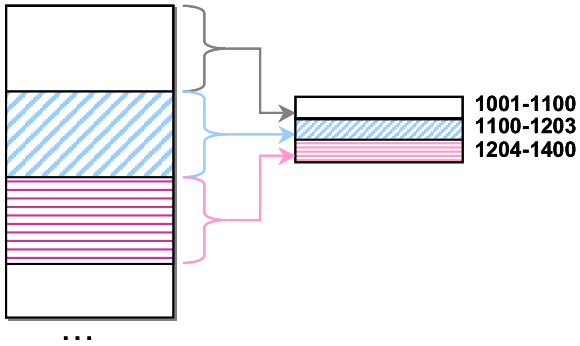

Figure 12: BlockZIP compresses data into blocks

larger. For example, our test shows that the average random reading time of one physical block from an IDE disk is about 14 ms, while the cost for uncompressing one block of gzipped data is 1.1 ms on a machine with P3 CPU at 500MHz, and only 0.26 ms on a machine with P4 CPU at 2.4GHz. With this inspiration, we propose a block-based compression technique for relational databases.

8.1 Block-based Compression: BlockZIP

Traditional data compression tools compress a file as a whole and can not be used for database applications. Here instead, we propose a block-based compression technique BlockZIP that supports block-based compression and uncompression.

BlockZIP (Figure 12) is based on zlib [45] (the library version of gzip). Zlib uses the deflation algorithm(an LZ77 variant) and Huffman encoding for data compression. The difference between BlockZIP and zlib is that instead of compressing the data as a whole, it compresses the data as block-sized blocks, and after compression with BlockZIP, the output consists of a set of block-sized compressed blocks concatenated together. Thus if we know which blocks to access, we only need to read and uncompress those specific blocks, so uncompressing of the whole file is not needed. The steps of BlockZIP compression are described in Algorithm 2. The uncompression of such blocks uses exactly the same zlib library functions.

BlockZIP facilitates uncompression at the granularity of a block, thus snapshot and temporal slicing queries can be efficient, since only a small number of blocks need to be uncompressed.

8.2 Storage Utilization with Compression

Compressed data blocks can be stored as BLOBs in a relational table, and user-defined uncompression table functions are used to extract records from each BLOB. We first generate a unique **`sid`** from **`(segno, id)`**, which is sorted in the order of **`segno`** and **`id`**. For a salary history table **`employee_salary (sid, salary, tstart, tend)`**, the content is BlockZIPed and each block is stored as a BLOB in table **`salary_blob(blockno, startsid, endsid, blockblob)`**, where **`startsid`** and **`endsid`** represent respectively the first **`sid`** and last **`sid`** in the compression block. A BLOB size of 4000 bytes is chosen for our experiments. An additional table **`salary_segrange(segno, startblock, endblock, segstart, segend)`** is used to keep the block range and interval for each segment. Note that the current segment has a high usefulness and is used for updates, thus not compressed.

We then compare the storage of the three systems with compression and without compression: Tamino(H-documents), ArchIS-DB2 and ArchIS-ATLaS (the latter two with segment clustered data). Figure 13 shows that with compression, the storage sizes of ArchIS-DB2 and ArchIS-ATLaS drop significantly and the compression ratio for ArchIS-DB2(0.23) and ArchIS-ATLaS(0.23) reach very closely to that of Tamino(0.22).

Algorithm 2 BlockZIP compression

```
1: BLOCKSIZE ← B bytes
2: DATASIZE ← D bytes
3: Sample the input data and get estimated compression factor f_0 and average record size R bytes
4: Estimate the number of characters to be compressed into one block: N ← B × f_0
5: BLOCKSTART ← 0
6: COMPRESSEDBLOCKS ← NULL
7: repeat
8:    Read N characters from position BLOCKSTART in the data stream { Data can be cached }
9:    Compress the N characters as block C with S bytes
10:   if (S < B) then
11:      Estimate the number of extra records to fit in the gap: R_+ ⇐ B - S
12:      if (R_+ < 1) then
13:         append R_+ blanks to C
14:         COMPRESSEDBLOCKS ← COMPRESSEDBLOCKS appends C
15:         BLOCKSTART ← BLOCKSTART + N
16:      else
17:         N ← N + R_+ × R
18:      end if
19:   else
20:      Estimate the number of records to be reduced: R_− ⇐ S - B
21:      N ← N - R_− × R
22:   end if
23: until (BLOCKSTART > D)
24: Output COMPRESSEDBLOCKS
```

Without compression, Tamino's compression ratio is 1.47, a 47% percent increase from original XML documents.

8.3 Query Performance with Compression

We then investigate the query performance on three sets of systems: a) ArchIS-DB2 and ArchIS-ATLaS with clustered data and with compression; b) ArchIS-DB2 and ArchIS-ATLaS with clustered data and without compression; and c) Tamino with non-clustered data and with compression. We use the same queries from Table 3.

Figure 14 shows that RDBMSs without compression have significant performance advantage over a native XML DB, and the benefit of RDBMSs remains for compressed data. With compression, ArchIS-ATLaS and ArchIS-DB2 run the snapshot queries much faster than Tamino, e.g., for snapshot query Q2, ArchIS-ATLaS is 67 times faster than Tamino, and ArchIS-DB2 is 37 times faster than Tamino. Temporal slicing queries are also much faster on both ArchIS-ATLaS and ArchIS-DB2: Q5 on ArchIS-ATLaS is 46 times faster than on Tamino, and ArchIS-DB2 is 26 times faster. All other historical queries are also faster on ArchIS-ATLaS than on Tamino. For temporal join Q6 on ArchIS-ATLaS, we effectively optimize the join through a user-defined aggregate [46] in one scan, and it takes only 6 seconds for compressed data.

On ArchIS-ATLaS, the performance with compression is very close to that without compression. We are also able to get better performance from ArchIS-ATLaS than ArchIS-DB2 for most queries on compressed data, since we used ArchIS-DB2 as a closed box, while for ArchIS-ATLaS we could control the internals of ArchIS-ATLaS for better query optimization. ArchIS-ATLaS' advantage increases on compressed data

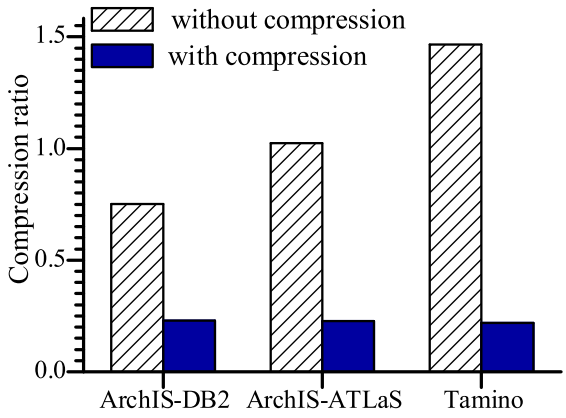

Figure 13: Compression ratios of historical XML storage on different systems

inasmuch as ArchIS-ATLaS' table functions performed better than those of ArchIS-DB2.

In summary, RDBMSs with temporal clustering show a significant performance advantage over a native XML database on most temporal queries. After introducing compression into RDBMSs, these still have a performance advantage while the native XML system has a marginal advantage in terms of storage efficiency.

8.4 Update Performance

When an update happens in the current database, it is tracked and ArchIS will update the live segment correspondingly, and all historical data archived in the history segments will not be touched. The situation is different for Tamino, where live data and historical data are mixed together, and insertions could cause page splits.

As an example, by updating the current salary of employee "Bob" by 10%, it takes 1.2 seconds on Tamino, and only 0.29 seconds for a segment-based clustering scheme on ArchIS-DB2.

As another example, for a simulated daily update, the cost is 15 seconds for Tamino, and 1.52 seconds for ArchIS-DB2. While normally updates are significant faster in ArchIS-ATLaS than in Tamino, with ArchIS-ATLaS we also have the occasional situations where the current segment's usefulness is below the threshold, and all current data are archived into a new segment. This takes 39 seconds, and if such segment is to be output and compressed, it takes an additional 36 seconds. However, the archiving of each segment only occurs once.

9 Conclusion and Future Work

The ArchIS system described in this paper demonstrates that the transaction time histories of relational databases can be stored and queried efficiently by using (i) XML to provide temporally-grouped representations of such histories, and (ii) SQL/XML to implement queries expressed against these representations. The paper elucidates the query mapping, indexing, clustering, and compression techniques used to achieve performance levels well above those of a native XML DBMS, as demonstrated by several experiments presented in the paper. The approach realized by ArchIS is general, and can be used to add a transaction-time capability to any existing RDBMS. The approach is also complete, since its realization does not require the invention of new techniques, nor costly extensions of existing standards.

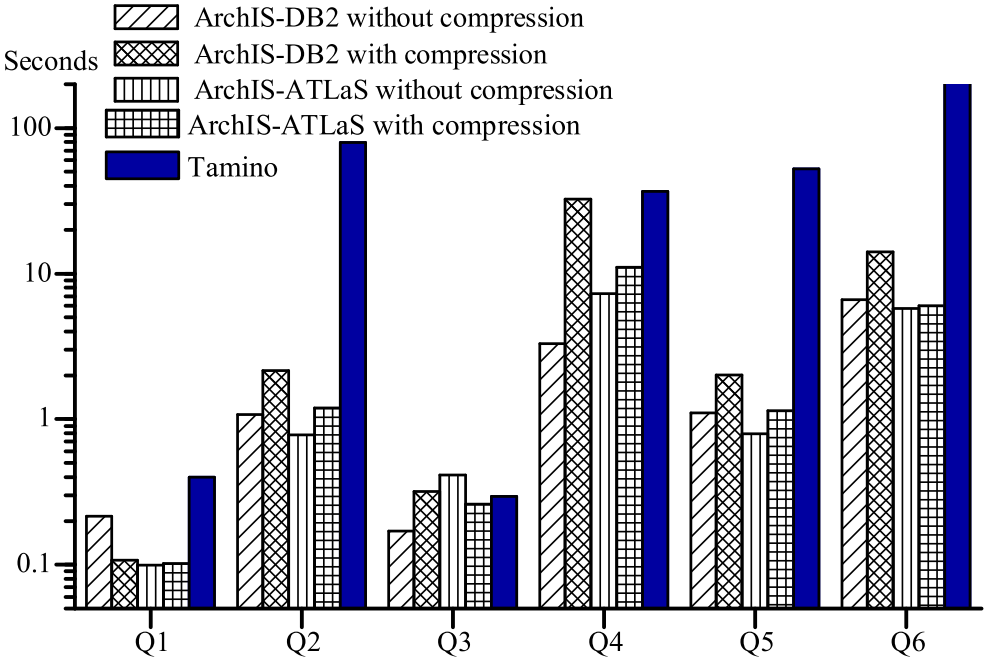

Figure 14: Query performance with compression

Several opportunities for further research and improvements have however emerged during our discussion. For instance, many end-users would prefer to interact with graphical user interfaces instead of XML/XQuery: the design of friendly interfaces based on temporally grouped models represents an interesting research problem.

At the physical level, many clustering and indexing techniques have been proposed for temporal databases [47] and deserve further investigations. Also other efficient data compression techniques proposed for XML data deserve further study [48].

Many interesting research questions also arise if we consider natural generalizations of our approach, and its possible applications to (i) valid-time databases and bitemporal databases, (ii) O-R DBMSs, and (iii) arbitrary XML documents. A related question was recently studied in [49], where it was concluded that temporally grouped models and XML remains effective, but complex indexing, clustering, and optimization techniques are needed to achieve high performance levels for valid time and bitemporal databases.

The second question involves the applicability of approaches similar to the one we have proposed to other data models, including object-oriented models and semistructured data models other than XML. Our intuition suggests that, not only the answer to these questions is largely positive, but, surprisingly enough, much of our approach to temporal information management is applicable to SQL itself. Indeed, the most recent SQL:2003 standards support nested relations [50] that can be used to support a temporally grouped data model. Simple temporal queries can be expressed in SQL itself, while more complex queries could require the use of a library of temporal functions and aggregates similar to those that we have developed for ArchIS. This suggests that database systems will be able to manage efficiently temporal information, and also give users a choice on whether to operate under XML standards or SQL standards—while their support is unified and optimized at the internal level.

The third research issue is perhaps the most important, since the preservation of digital artifacts represent a critical issue for the information age. The temporally grouped data model and timestamping scheme used here is also applicable to generic multi-version XML documents [51], to support evolution queries using XQuery. This scheme makes it possible to ask interesting temporal queries on the evolution of standards,

including e.g., the successive revision of XLink [52] standards, or, from the history of university catalogs, when a new course was first introduced. The XML-based approach here introduced represents a significant first step toward adding a historical information management and query capability to information systems.

References

[1] R. T. Snodgrass. Developing Time-Oriented Database Applications in SQL. *Morgan Kaufmann*, 1999.

[2] G. Ozsoyoglu and R.T. Snodgrass. Temporal and Real-Time Databases: A Survey. *TKDE*, 7(4):513–532, 1995.

[3] J. Clifford, A. Croker, F. Grandi, and A. Tuzhilin. On Temporal Grouping. In *Recent Advances in Temporal Databases*, pages 194–213. Springer Verlag, 1995.

[4] XQuery 1.0: An XML Query Language. http://www.w3.org/XML/Query.

[5] S. Kepser. A Simple Proof for the Turing-Completeness of XSLT and XQuery. In *Extreme Markup Languages*, 2004.

[6] M. Fernandez and J. Simon. Growing XQuery. In *ECOOP*, 2003.

[7] M. Carey, J. Kiernan, J. Shanmugasundaram, and et al. XPERANTO: A Middleware for Publishing Object-Relational Data as XML Documents. In *VLDB*, 2000.

[8] Oracle XML. http://otn.oracle.com/xml/.

[9] SQL/XML. http://www.sqlx.org.

[10] Information technology - Database languages - SQL Part 14: XML-Related Specifications (draft: 2003-07).

[11] F. Grandi and F. Mandreoli. The Valid Web: An XML/XSL Infrastructure for Temporal Management of Web Documents. In *ADVIS*, 2000.

[12] M. Gergatsoulis and Y. Stavrakas. Representing Changes in XML Documents using Dimensions. In *Xsym*, 2003.

[13] T. Amagasa, M. Yoshikawa, and S. Uemura. A Data Model for Temporal XML Documents. In *DEXA*, 2000.

[14] C.E. Dyreson. Observing Transaction-Time Semantics with TTXPath. In *WISE*, 2001.

[15] S. Zhang and C. Dyreson. Adding Valid Time to XPath. In *DNIS*, 2002.

[16] D. Gao and R. T. Snodgrass. Temporal Slicing in the Evaluation of XML Queries. In *VLDB*, 2003.

[17] P. Buneman, S. Khanna, K. Tajima, and W. Tan. Archiving scientific data. *ACM Trans. Database Syst.*, 29(1):2–42, 2004.

[18] R. T. Snodgrass. *The TSQL2 Temporal Query Language*. Kluwer, 1995.

[19] J. Chomicki, D. Toman, and M.H. Böhlen. Querying ATSQL Databases with Temporal Logic. *TODS*, 26(2):145–178, June 2001.

[20] C. Zaniolo, S. Ceri, C.Faloutsos, R.T. Snodgrass, V.S. Subrahmanian, and R. Zicari. *Advanced Database Systems*. Morgan Kaufmann Publishers, 1997.

[21] A. Steiner. *A Generalisation Approach to Temporal Data Models and Their Implementations*. PhD thesis, ETH Zurich, 1997.

[22] Oracle Flashback Technology. http://otn.oracle.com/deploy/availability /htdocs/flashback_overview.htm.

[23] F. Wang and C. Zaniolo. Publishing and Querying the Histories of Archived Relational Databases in XML. In *WISE*, 2003.

[24] S.Y. Chien, V.J. Tsotras, and C. Zaniolo. Version Management of XML Documents. In *WebDB*, 2000.

[25] C. S. Jensen and D. B. Lomet. Transaction Timestamping in Temporal Databases. In *VLDB*, 2001.

[26] M. H. Böhlen, R. T. Snodgrass, and M. D. Soo. Coalescing in Temporal Databases. In *VLDB*, 1996.

[27] J. Clifford. *Formal Semantics and Pragmatics for Natural Language Querying*. Cambridge University Press, 1990.

[28] J. Clifford, A. Croker, and A. Tuzhilin. On Completeness of Historical Relational Query Languages. *ACM Trans. Database Syst.*, 19(1):64–116, 1994.

[29] J. Clifford, C.E. Dyreson, T. Isakowitz, C.S. Jensen, and R.T. Snodgrass. On the Semantics of "Now" in Databases. *TODS*, 22(2):171–214, 1997.

[30] K. Torp, C. S. Jensen, and R. T. Snodgrass. Modification Semantics in Now-relative Databases. *Information Systems*, In Press.

[31] H. Schöning. Tamino - a DBMS Designed for XML. In *ICDE*, 2001.

[32] ATLaS. http://wis.cs.ucla.edu/atlas.

[33] D. DeHaan, D. Toman, M. P. Consens, and M. T. Ozsu. A Comprehensive XQuery to SQL Translation Using Dynamic Interval Encoding. In *SIGMOD*, 2003.

[34] J. Shanmugasundaram and et al. Efficiently Publishing Relational Data as XML Documents. In *VLDB*, 2000.

[35] Galax–an Open Source XQuery Implementation. http://www.galaxquery.org.

[36] N. Alur and P. Haas and D. Momiroska and et al. *DB2 UDB's High Function Business Intelligence in e-Business*. http://www.redbooks.ibm.com/, 2002.

[37] XQuery 1.0 and XPath 2.0 Functions and Operators. http://www.w3.org/tr/xquery/.

[38] BerkeleyDB. http://www.sleepycat.com.

[39] Employee Temporal Data Set. http://www.cs.auc.dk/TimeCenter/software.htm.

[40] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly, 2nd edition, 2002.

[41] D. Wilhite B. R. Iyer. Data Compression Support in Databases. In *VLDB*, 1994.

[42] U. Shaft J. Goldstein, R. Ramakrishnan. Compressing Relations and Indexes. In *ICDE*, 1998.

[43] C. V. Ravishankar W. K. Ng. Relational Database Compression Using Augmented Vector Quantization. In *ICDE*, 1995.

[44] Table Compression in Oracle9i Release2. http://otn.oracle.com/ oramag/webcolumns/2003/techarticles/poess_tablecomp.html.

[45] Zlib. http://www.gzip.org/zlib/.

[46] H. Wang and C. Zaniolo. Using SQL to Build New Aggregates and Extenders for Object-Relational Systems. In *VLDB*, 2000.

[47] B. Salzberg and V. J. Tsotras. Comparison of Access Methods for Time-evolving Data. *ACM Comput. Surv.*, 31(2):158–221, 1999.

[48] H. Liefke and D. Suciu. XMILL: An Efficient Compressor for XML Data. In *SIGMOD*, pages 153–164, 2000.

[49] F. Wang and C. Zaniolo. XBiT: An XML-based Bitemporal Data Model. In *ER*, 2004.

[50] Database Languages SQL, ISO/IEC 9075-*:2003.

[51] F. Wang and C. Zaniolo. Temporal Queries in XML Document Archives and Web Warehouses. In *TIME-ICTL*, 2003.

[52] XML Linking Language (XLink) Version 1.0. http://www.w3.org/TR/xlink/.

APPENDIX: Built-in Temporal Functions

The functions are written with the time granularity of date. It can easily be applied to other time granularity such as datetime and time. Dates comparisons are simplified as string comparisons, and can be implemented using date/time comparisons functions defined in XPath functions as well.

snapshot($e,$t):

```
define function snapshot( $e, $t ) {
  if((date($e/@tstart)<=date($t)) and
     (date($e/@tend) >= date($t)))
  then
    element{name($e)} {
     $e/text(), $e/@*[ string(name(.)) != "tstart" and
     string(name(.)) != "tend" ] ,
     for $child in $e/*
     return  snapshot($child, $t)
   }
  else ()
}
```

coalesce($e):

```
define function sortbytstart($e) {
     $e sortby (@tstart)
}

define function coalesce($e) {
 if (count($e) =1 ) then $e
 else
  if( $e[1]/text()!= coalesce(subsequence($e,2))[1]/text())
  then ($e[1], coalesce(subsequence($e,2)))
  else
   if(string($e[1]/@tend)<
      string(coalesce( subsequence($e,2))[1]/@tstart))
   then (  $e[1], coalesce(subsequence($e,2)) )
   else (
    element {name($e[1]) }
    {coalesce( subsequence($e,2)[1]/@tend ), $e[1]/@tstart,
    $e[1]/text()}, subsequence( coalesce(subsequence($e,2) ),2)
   )
}
```

toverlaps($a,$b):

```
define function toverlaps($a,$b){
 if($a/@tend< $b/@tstart or $b/@tend < $a/@tstart)
 then "false"
 else "true"
}
```

tmeets($a,$b):

```
define function tmeets($a,$b){
 if (subtract-dates($b/@tstart, $a/@tend)) = "1D" )
 then "true"
 else "false"
}
```

tprecedes($a,$b):

```
define function tprecedes($a,$b){
 if ( $b/@tstart > $a/@tend )
 then "true"
 else "false"
}
```

tequals($a,$b):

```
define function tequals($a,$b){
 if($a/@tstart = $b/@tstart and $a/@tend = $b/@tend)
 then "true"
 else "false"
}
```

tcontains($a,$b):

```
  define function tcontains($a, $b){
   if($a/@tstart<= $b/@tstart and $a/@tend >= $b/@tend)
   then true()
   else false()
  }
```

overlapinterval($e):

```
define function overlapinterval($a, $b){
 if($a/@tend< $b/@tstart or $b/@tend < $a/@tstart)
 then ()
 else element interval {
   attribute tstart {max(($b/@tstart,$a/@tsart))},
   attribute tend {min(($a/@tend,$b/@tend))}
 }
}
```

timespan($e):

```
define function timespan($e){
 subtract-dates($e/@tend,$e/@tstart)
}
```

tinterval($e):

```
define function tinterval($e){
 element interval {
   attribute tstart {$e/@tstart},
   attribute tend {$e/@tend}
 }
}
```

telement($e):

```
define function telement($ts,$te){
 element interval {
   attribute tstart {$ts},
   attribute tend {$te}
 }
}
```

rtend($e):

```
define function rtend($e){
   if  ($e/@tend = "9999-12-31")
  then
    element{name($e)} {
     $e/text(), $e/@*[ string(name(.)) != "tend"],
     attribute tend{current-date()},
     for $child in $e/*
     return  rtend($child)
   }
 else
   element{name($e)} {
     $e/text(), $e/@*,
      for $child in $e/*
     return  rtend($child)
   }
}
```

externalnow($e):

```
define function externalnow($e){
   if  ($e/@tend = "9999-12-31")
  then
    element{name($e)} {
     $e/text(), $e/@*[ string(name(.)) != "tend"],
     attribute tend{"now"},
     for $child in $e/*
     return  rtend($child)
   }
 else
   element{name($e)} {
     $e/text(), $e/@*,
```

```
      for $child in $e/*
     return  rtend($child)
   }
}
```