Publishing and Querying the Histories of Archived Relational Databases in XML

Fusheng Wang Carlo Zaniolo

Department of Computer Science
University of California, Los Angeles
wangfsh@cs.ucla.edu zaniolo@cs.ucla.edu

Abstract

There is much current interest in publishing and viewing databases as XML documents. The general benefits of this approach follow from the popularity of XML and the tool set available for visualizing and processing information encoded in this universal standard. In this paper, we explore the additional and unique benefits achieved by this approach on temporal database applications. We show that XML views combined with XQuery can provide surprisingly effective solutions to the problem of supporting historical queries on past content of database relations and their evolution. Indeed, using XML, the histories of database relations can be naturally represented by temporally grouped data models. Thus, we identify mappings from relations to XML that are most conducive to modeling and querying database histories, and show that temporal queries that would be very difficult to express in SQL can be easily expressed in standard XQuery.

Then, we turn to the problem of supporting efficiently the storage and the querying of relational table histories. We present an experimental study of the pros and cons of using native XML databases, versus using traditional databases, where the XML-represented histories are supported as views on the historical tables.

1 Introduction

Information systems have yet to address satisfactorily the problem of how to deal with the evolution in the structure and content of their underlying databases, and to support the query and retrieval of past information. This is a serious shortcoming for the web, where documents frequently change in structure and content, or disappear all together, causing serious problems, such as broken links and lack of accountability for sites of public interest. Database-centric information systems face similar problems, particularly since current database management systems (DBMSs) provide little help in that respect. Indeed DBMSs do not provide effective means for archiving and supporting historical queries on their past contents—to the point that, e.g., the current SQL standards lack the basic temporal extensions needed to express and support historical queries. Given the strong application demand and the significant research efforts spent on these problems [1], the lack of current solutions must be attributed, at least in part, to the technical difficulty of introducing temporal extensions into relational databases and object-oriented databases. Schema changes represent a particularly difficult and important problem for modern information systems, which need to be designed for evolution [2, 3, 4].

Meanwhile, there is much current interest in publishing and viewing database-resident data as XML documents. In fact, such XML views of the database can be easily visualized on web browsers and processed by web languages, including powerful query languages such as XQuery [5]. The definition of the mapping from the database tables to the XML view is in fact used to translate queries on these views into equivalent SQL queries on the underlying database [6, 7].

As the database is updated, its external XML view also evolves—and many users who are interested in viewing and querying the current database are also interested in viewing and querying its past snapshots and evolving history. In this paper, we show that the history of a relational database can be viewed naturally as yet another XML document. The various benefits of XML-published relational databases (browsers, web languages and tools, unification of database and web information, etc.) are now extended to XML-published relation histories. In fact, we show that we can define and query XML views that support a temporally grouped data model, which has long been recognized as very effective in representing temporal information [8], but could not be supported well using relational tables and SQL. Therefore, temporal queries that would be very difficult to express in SQL can now be easily expressed in standard XQuery.

We then turn to the problem of storing and querying

these historical views efficiently. A native XML database system can be used to store the history of database tables as XML documents. Alternatively, the document can be decomposed (shredded) back into tables, and stored in traditional DBMS; then queries on the external XML view are implemented as equivalent queries on the stored representation. We investigated the two approaches and identify clear tradeoffs of query versus storage performance presented by the two approaches.

2 Related Work

Time in XML. Some interesting research work has recently focused on the problem of representing historical information in XML. In [9] an annotation-based object model is proposed to manage historical semistructured data, and a special Chorel language is used to query changes. In [10] a new <valid> markup tag for XML/HTML documents is proposed to support valid time on the Web, thus temporal visualization can be implemented on web browsers with XSL.

In [11, 12], a data model is proposed for temporal XML documents. However, since a valid interval is represented as a mixed string, queries have to be supported by extending DOM APIs or XPath. Similarly, TTXPath [13] is another extension of XPath data model and query language to support transaction time semantics. (In our approach, we instead support XPath/XQuery without any extension to XML data models or query languages.)

An archiving technique for scientific data was presented in [14], based on an extension of the SCCS [15] scheme. This approach timestamps elements only when they are different from the parent elements, so the structure of the representation is not fixed; this makes it difficult to support queries in XPath/XQuery, which, in fact, is not discussed in [14]. The scheme we use here to publish the histories of relational tables present several similarities to that proposed in [14], but it also provides full support for XML query languages such as XPath and XQuery.

Temporal Databases and Grouped Representations. There is a large number of temporal data models proposed and the design space for the relational data model has been exhaustively explored [1]. Clifford et al. [8] classified them as two main categories: *temporally ungrouped* and *temporally grouped*. The second representation is said to have more expressive power and to be more natural since it is history-oriented [8]. TSQL2 [4] tries to reconcile the two approaches [8] within the severe limitations of the relational tables. Our approach is based on a temporally grouped data model, since this dovetails perfectly with XML documents' hierarchical structure.

Object-oriented temporal models are compared in [16], and a formal temporal object-oriented data model is proposed in [17]. The problem of version management in object-oriented and CAD databases has received a significant amount of attention [18, 19]. However, support for temporal queries is not discussed, although query issues relating to time multigranularity were discussed in [20].

Publishing Relational Databases in XML. There is much current interest in publishing relational databases in XML. One approach is to publish relational data at the application level, such as DB2's XML Extender [21], which uses user-defined functions and stored procedures to map between XML data and relational data. Another approach is a middleware based approach, such as in SilkRoute [22] and XPERANTO [6, 7], which define XML views on top of relational data for query support. For instance, XPERANTO can build a default view on the whole relational database, and new XML views and queries upon XML views can then be defined using XQuery. XQuery statements are then translated into SQL and executed on the RDBMS engine. This approach utilizes RDBMS technology and provides users with a unified general interface.

Several DBMS vendors are jointly working toward new SQL/XML standards [23]; the objective is to extend SQL to support XML, by defining mappings of data, schema, actions, etc., between SQL and XML. A new XML data type and a set of SQL XML publishing functions are also defined, and are partly supported in Oracle 9i [24].

3 History of DB Tables as XML Documents

Figure 1 and Figure 2 describe the history of employees and departments as they would be viewed in traditional transaction-time databases [1]. Instead of using this temporally ungrouped representation, we use temporally-grouped representations that exploit the expressive power of XML and its query languages. Thus, instead of the representation shown in Table 1 and Table 2 we will use the representation shown in Figure 1 and Figure 2. We will call these *H-documents*. Each element in an H-document is assigned the 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. Note that there is an intrinsic constraint that the interval of a parent node always covers that of its child nodes. The H-document also has a simple and well-defined schema.

For updates on a node, when there is a `delete`, the value of `tend` is updated to the current timestamp; when there is an `insert`, a new node is appended with `tend` set to *now*; and `update` can be implemented as a delete followed by an insert.

Table 1. The snapshot history of employees

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

Table 2. The snapshot history of departments

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

Our H-documents use a temporally grouped data model [8]. Clifford, et al. [8] show that temporally-grouped models are more natural and powerful than temporally-ungrouped ones. Temporal groups are however difficult to support in the framework of flat relations and SQL. Thus, many approaches proposed in the past instead timestamp the tuples of relational tables. These approaches incur into several problems, including the coalescing problem [4]. TSQL2's approach [4] attempts to achieve a compromise between these two [8], and is based on an implicit temporal model, which is not without its own problems [25].

An 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 temporal projections, snapshot queries, joins and historical queries on `employees` and `departments`. These queries were tested with Quip [26] (SoftwarAG's implementation of XQuery) and can be downloaded from `http://wis.cs.ucla.edu/~wangfsh/wise03/`.

3.1 Publishing Each Table as an XML Document with Columns as Elements

A natural way of publishing relational data is to publish each table as an XML document by converting relational columns into XML elements [27]. Figure 1 shows the history of the `employee` table and Figure 2 shows the history of the `dept` table. Thus the history of each relation is published as a separate H-document.

Based on the published documents, we can specify a variety of queries in XQuery:

QUERY 1: Temporal projection: retrieve the salary history of employee "Bob":

```
element salary_history{
  for $s in document("employees.xml")/employees/
     employee[name='Bob']/salary
  return $s }
```

QUERY 2: Snapshot queries: retrieve the departments on 1996-01-31:

```
for $d in document("depts.xml")/depts/dept
   [tstart(.)<="1996-01-31" and tend(.)>=
   "1996-01-31"]
let $n:=$d/deptname[tstart(.)<="1996-01-31" and
   tend(.)>="1996-01-31"]
let $m:=$d/mgrno[tstart(.)<="1996-01-31" and
   tend(.)>="1996-01-31"]
return( element dept{$n,$m } )
```

Note that `tstart()` and `tend()` are user-defined functions (in XQuery syntax) that get the starting time and ending time of an element respectively, thus the implementation is transparent to users. This will be further discussed in Section 4.2.

QUERY 3: Find employees history from 1995-05-01 to 1996-04-30:

```
for $e in document("employees.xml")/employees
    /employee
let $ol:=overlapinterval(
        $e, telement("1995-05-01","1996-4-30") )
where not (empty($ol))
return ( $e/name, $ol )
```

Here `overlapinterval($a, $b)` is a user-defined function that returns an element `interval` with overlapped interval as attributes`(tstart, tend)`. If there is no overlap, then no element is returned which satisfies the XQuery built-in function `empty()`. The next query is a containment query:

QUERY 4: Find employee(s) who worked in the "QA" department throughout the history of that department:

```
for $d in document("depts.xml")/depts/dept
   [deptname='QA']/deptno
for $e in document("employees.xml")/employees
    /employee[deptno=$d]
where tstart($e/deptno)=tstart($d) and tend($e)=
    tend($d)
return $e/name
```

QUERY 5: Find the manager's empno for each employee:

```
for $e in document("employees.xml")/employees
    /employee
for $d in document("depts.xml")/depts/dept
   [deptno=$e/deptno]
for $m in $d/mgrno
let $ol:=overlapinterval($m,$e)
where not (empty($ol))
return( $e/name, $m, $ol )
```

This query will join `employees.xml` and `depts.xml` by `dept`, and the `overlapinterval()` function will return only the intervals of managers that overlap with the employee with the overlapped timestamp intervals.

QUERY 6: Find the history of employees in each dept:

```
<employees tstart="1995-01-01" tend="1996-12-31">
  <employee tstart="1995-01-01" tend="1996-12-31">
    <empno tstart="1995-01-01" tend="1996-12-31">1001</empno>
    <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 1. The history of the `employee` table is published 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>
  <dept tstart="1993-01-01" tend="1997-12-31">
      <deptno tstart="1993-01-01" tend="1997-12-31">d03</deptno>
      <deptname tstart="1993-01-01" tend="1997-12-31">Sales</deptname>
      <mgrno tstart="1993-01-01" tend="1997-12-31">4748</mgrno>
  </dept>
  <!-- ... -->
</depts>
```

Figure 2. The history of the `dept` table is published as `depts.xml`

```
element depts{
 for $d in document("depts.xml")/depts/dept
  return
   element dept { $d/@*, $d/*,
    element employees {
     for $e in document("employees.xml")/
       employees/employee
     where  $e/deptno = $d/deptno and
        not(empty(overlapinterval($e, $d)))
     return($e/name, overlapinterval($e,$d))
}}}
```

This query will join depts and employees document and generate a hierarchical XML document grouped by dept.

Several alternative representations for table histories were studied in [28]. For instance, multiple tables can be first joined together and then represented by a single XML document. This approach offers no advantage compared to the one described above [28]. However IDs can be added to this representation to make some join queries easier [28]. Yet another approach consists of representing the joined tables by a hierarchically structured XML document. This approach simplifies some queries but complicate others [28]. The last approach is to represent the tuples of a relational table by the attribute values of the XML document. Then, the XML document reproduces the flat structure of tables with timestamped tuples, and the well-known problems of this temporally ungrouped representation [28]. In summary, publishing each table as a separate XML document with columns as elements was shown to be the approach of choice in [28].

4 Complex Temporal Queries

In the previous sections we have discussed temporal queries such as snapshot queries, interval queries, historical queries and temporal join queries. Next we show that we can specify more complex temporal queries. The following queries are based on the XML documents in Figure 1 and Figure 2, except that `name` element is replaced with two elements `firstname` and `lastname` to simulate real data.

In Q1 and Q2, we show that it's easy to support **since** and **until** connectives of first-order temporal logic [29].

`Q1. A Since B`. Find the employee who has been the

manager of the dept since he/she joined the dept “d007”:

```
for $e in document("employees.xml")/employees
    /employee
let $m:= $e/title[title="Manager" and
      tend(.)=current-date()]
let $d := $e/deptno[deptno ="d007" and
    tcontains($m, .) ]
where not empty($d) and not empty($m)
return <employee>
 {$e/empno, $e/firstname, $e/lastname}</employee>
```

Here `tcontains()` is a user-defined function to check if an interval is covers another interval.

`Q2. A Until B`. Find the employee who only worked in dept “d001” until he/she was appointed as the manager of dept “d007”:

```
for $e in document("employees.xml")/employees
    /employee
let $d1:=$e/deptno[1][deptno="d001"]
let $d2:=$e/deptno[deptno="d007" and
    tstart(.)=tend($d1)]
let $m:=$e/title[title="Manager" and tstart(.)<=
    tstart($d2) and tend(.)>=tstart($d2)]
where not empty($d1) and not empty($d2) and
    not empty($m)
return <employee>
 {$e/empno, $e/firstname, $e/lastname}</employee>
```

`Q3. Continuous Periods` . Find employees who worked in dept (deptno= “d007”) for 10 consecutive years:

```
for $e in document("employees.xml")/employees
  /employee[deptno="d007"]
let $duration := subtract-dates(tend($e/deptno),
 tstart($e/deptno) )
where dayTimeDuration-greater-than(
    $duration,"P3660D")
return
 <result>{$e/empno, $e/firstname, $e/lastname}
 </result>
```

Here “P3660” is a duration constant of 10 years in XQuery.

`Q4. Period Containment`. Find employees with same history as employee “10112”, i.e., they worked in the same department(s) as employee “10112” and exactly for the same periods:

```
for $e1 in document("employees.xml")/employees
    /employee[empno = '10112']
for $e2 in document("employees.xml")/employees
    /employee[empno != '10112']
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/empno}</employee>
```

where `tequals($d1,$d2)` is a user-defined function to check if two nodes have the same intervals, and it is equivalent to `tcontains($d1,$d2) and tcontains($d2,$d1)`.

4.1 User-Defined Temporal Functions

The user-defined functions `tstart` and `tend` in temporal queries offer 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:

History functions:

`history($e, Ts, Te)` will return the history of node e from time Ts to time Te;

`snapshot ($e, T)` will return the snapshot of node e at time T;

`diff($e, Ts, Te)` will return the difference of node e at time Ts and Te;

`invariance($e, Ts, Te)` is a complementary of diff and will return invariant parts of node e from Ts to Te.

Restructuring functions:

`coalese($l)` will coalese a list of nodes.

Interval functions:

`toverlaps, tprecedes, tcontains, tequals, tmeets` will return true or false according to two interval positions;

`overlapinterval($a,$b)` will return the overlapped interval if they overlap. The result has the form: `<interval tstart= "..." tend="..." />`.

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 node e;

`telement(Ts, Te)` constructs an empty element `telement`with attribute `tstart` and `tend` as the argument values respectively;

`getdbnow()` returns the database implementation of “now”;

`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”.

4.2 Support for ‘now’

This is an important topic that has received considerable attention in temporal databases [30]. For historical transaction time databases, ’now’ can only appear as the end of a period, and means ‘no changes until now.’ That is to say that the values in the tuple are still current at the time the query is asked. Therefore, a simple strategy to support this concept consists in replacing every occurrence of the symbol ‘now’ in the database with the value CURRENT_TIMESTAMP (or CURRENT_DATE, depending on the granularity used for time-stamping the tuples). This is basically the strategy we use in our implementation; of course, we do not perform

the actual instantiation on all the 'now' occurrences in the database for each query Q. Rather, we perform instantiations conservatively, as needed to to answer the query correctly.

Internally, we use "end-of-time" values to denote the 'now' symbol. For instance for dates we use "9999-12-31." The user does not access this value directly, he/she will access it through built-in functions. For instance, to refer to an employee on "2003-02-13", the user might write `tstart`($e) <="2003-02-13"<= `tend`($e). While the function `tstart()` returns the start of the interval, the `tend`() function returns its end, if this is different from '9999-12-31" and CURRENT_DATE otherwise.

Observe that while any other representation could have been used internally for 'now', 'end-of-time' values such as "9999-12-31" assure that the current search techniques based on indexes and temporal ordering can be used without any change.

The nodes returned in the output, such as `$s` in QUERY 1, use the "9999-12-31" representation used for internal data. However, for data returned to the end-user, two different representations are preferable. One is to return the CURRENT_DATE by applying function `rtend`() that, recursively, replaces all the occurrence of "9999-12-31" with the value of CURRENT_DATE. The other is to return a special string, such as *now* or *until-changed* to be displayed on the end-user screen. As discussed in [30], this is often the more intuitive and appealing for users, and is supported by our built-in function `externalnow`($e) that does that for the node `e` and its sub-nodes.

5 Representing and Querying Schema Histories in XML

An unexpected benefit of our approach is the ability of representing and querying the schema history of the underlining relations. In the past, a variety of temporal data models have been proposed [1] but few of them provide support for schema evolution or schema versioning [2]. Indeed, the flat structure of relational database makes it particularly difficult to support schema changes. In [31], we showed that by publishing the database history as XML documents (H-documents), we can also represent the schema history, with the help of the rich structure of XML.

The most basic schema evolutions in RDBMS are attribute evolution and relation evolution. Attribute evolution includes adding an attribute to the database and removing an attribute from the database. Relation evolution includes adding a relation, removing a relation, joining two relations into one, and decomposing a relation into two or more relations.

The attributes `tstart` and `tend` associated with each relation, e.g., `employees` (the root node) actually represent the time of creation and the time of deletion of the relation. By searching all the root nodes of the H-documents, we can have a complete history of all relations in the database.

Similarly, by coalescing the intervals of all `empno`, we get the interval of `empno` in the relation. Thus, the change history of every attribute of the relation is preserved. This is true under the assumption that there is no empty relation, and e.g., a relation begins when its first element is inserted. Indeed, content-free elements can be used to fill empty relations. For example, for the `salary` column, we can add the following element that specifies the period of existence of the salary attribute, independent of the existence of any actual salary value:

```
<salary tstart="2003-01-01" tend="now" />
```

The case of relations joined or decomposed can be handled by the usual view mechanism [31].

Schema History Queries. Since the schema history is incorporated in the H-documents, with powerful XML query languages such as XQuery, we can query the schema history directly without any extensions. The following are several queries on schema history:

Schema Q1: Schema Snapshot: find all the columns of `employees` relation on '1995-01-01':

```
<columns>{
for $e in distinct-values(collection("employees")
  /employees/employee/*[tstart(.)<="1995-01-01"
  and tend(.)="1995-01-01"]/local-name(.)  )
return <column>{$e}</column>
}</columns>
```

Schema Q2: Schema Change Detection: find when the address column was first added into the table:

```
let $a := collection("employees")/employees
    /employee/address
let $ts := min(  tstart($a) )
return $ts
```

More queries on schema evolution can be found in [31].

6 Historical Database Implementation

Two main implementation approaches are possible. One is to use a native XML database for storing and querying H-documents, the other is to shred the H-documents, to store them in relational tables, and map queries from XQuery to equivalent SQL statements. We compare the two approaches in terms of storage efficiency and query performance.

6.1 Native XML Databases

In this approach, the temporal XML documents (H-documents) are directly stored in the native XML database,

and XQuery queries can be directly specified upon the database. With update logs or active rules in the current database, changes can be tracked and archived in the historical database through updates on the H-documents.

We tested this approach on two leading native XML DBMSs in our testing, the Tamino XML Server [32] and the eXcelon XIS server [33]. Tamino has a text-based storage model, while XIS has an object-based storage model, which stores the persistent DOMs of XML document trees, backed by an OODBMS. Tamino automatically compresses large XML documents yielding a better storage efficiency, while a DOM tree is often much larger than the original XML document.

6.2 RDBMS-based Architecture

To decompose the historical XML documents, we make the following assumptions: each entity (e.g., `employee` or `dept`) or relation(e.g., `supplieritem`) in the current database has a unique key (e.g., `empno`) or composite keys(`supplierno, itemno`) to identify it, and the value of the key will not change along the history. Since all other attribute values (except the key) can be changed, one **attribute history table** is built for each attribute to store the history of such attribute. A **key table** is built for the keys. Each table will include two columns `tstart` and `tend` to represent the valid interval of that tuple. Besides, a **global relation table** is used to record all relations history, such as `employees` and `depts` relations.

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

```
employee(empno, firstname, lastname, sex,
DOB, deptno, salary, title)
```

where `empno` is the key. The history of the table is published as an H-document, which is then decomposed as the following tables:

Key table: `employee_id(id, tstart, tend)`
Since `id`(=`empno`) 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 follows:

```
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_lastname(id,lastname,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 (e.g., `empno`), thus indexes on such *id*s can efficiently join these relations.

The following shows a sample content of `employee_salary` table:

```
 ID     SALARY      TSTART     TEND
=======================================
100022  58805      02/04/1985 02/04/1986
100022  61118      02/05/1986 02/04/1987
100022  65103      02/05/1987 02/04/1988
100022  64712      02/05/1988 02/03/1989
100022  65245      02/04/1989 02/03/1990
100023  43162      07/13/1988 07/13/1989
...
```

Update on the history tables is similar to that for H-documents.

Global relation table:

```
relations(name, tstart, tend)
```

which records all the relations history in the database, such as `employees` and `departments`.

Publishing Table Histories. When the historical data are decomposed and stored into tables, we use a middleware to publishing data as XML. XTABLES [7](previously XPERANTO [6]) is a middleware-based approach to bridge XML queries and RDBMS, which creates XML views on relational data, supports XQuery on creating and querying XML views on relational data and pushes most computation down to the RDBMS engine. With XTABLES, a temporal XML view is reconstructed which is *exactly the same* as the original H-document representation.

For the history of employees, the default XML view is as follows:

```
<db>
 <employee_id>
  <row>
   <id>100022</id>
   <tstart>02/04/1985</tstart>
   <tend>12/31/9999</tend>
  </row>
  ...
 </employee_id>
 <employee_salary>
  <row>
   <id>100022</id> <salary>58805</salary>
   <tstart>02/04/1985</tstart>
   <tend>02/04/1986</tend>
  </row>
  ...
 </employee_salary>
 ...
 <relations>
  <row><name>employees</name>
   <tstart>01/01/1985</tstart>
   <tend>12/31/9999</tend>
  </row>
 </relations>
</db>
```

```
create view employees as(
let $rel := view("default")/relations/row[name="employees"]
return
  <employees tstart={$rel/tstart} tend={$rel/tend}>
  for $id in view("default")/employee_id/row
  return
    <employee tstart={$id/tstart} tend={$id/tend}>
      <empno tstart={$id/tstart} tend={$id/tend}>$id/id</empno>
      ...
      for $salary in view("default")/employee_salary/row[id=$id/id]
      return
        <salary tstart=$salary/tstart tend=$salary/tend>$salary/salary</salary>
        sortby(@tstart)
      ...
    </employee>
  </employees>
    )
```

Figure 3. User-defined XML view of employees that reconstructs the H-document view

Figure 3 shows the XQuery to reconstruct the H-document view of employees history, thus XML queries can be specified on such view through XTABLES.

Querying Table Histories. With the reconstructed H-document views through XTABLES, XQuery queries can be specified with XTABLES. The queries are translated into SQL queries and the query results are composed and returned as XML documents or fragments. Since user-defined functions are currently not supported in XTABLES, in the following experiments on DB2, we manually translate the XML queries as SQL queries.

Updating Table Histories. Changes in the current databases can be tracked with either update logs or triggers. For our testing, we built triggers (for update, deletion, and insertion) that successfully track and archive all changes in current tables updated into the historical tables.

7 Native XML DB vs RDBMS

Experimental Setup. We investigate three systems for archiving: native XML database-based: Tamino and XIS; and RDBMS-based: DB2. One of the criteria for archiving systems is storage utilization. To facilitate our testing, we generate a large amount of temporal data, representing as XML documents the history of 300,024 employees over 17 years.

The simulation models salary increases, changes of titles, and changes of departments. To accelerate queries for native XML databases, the single large XML document is cut into 600 small XML documents with 500KB each.

The testing machine is a PIII 733MHz PC with Windows 2000 Professional, with 512MB memory and a 80GB IDE hard drive. The databases we use are DB2 UDB Enterprise Edition V7.2, Tamino V3.1.1.4, and eXcelon XIS Server V3.1. The size of the temporal data represented in XML documents (H-documents) is 342,468KB.

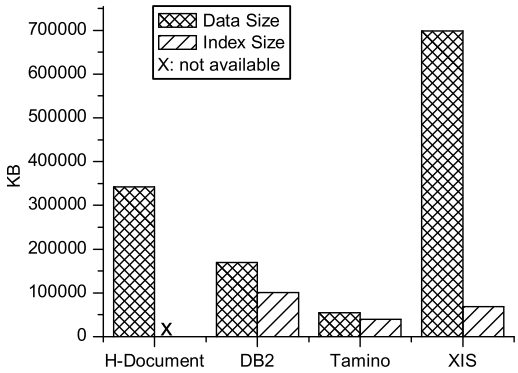

Figure 4. Comparison of Database Sizes for Archiving Temporal Data

Storage Utilization. Figure 4 shows that XIS has the largest storage since its storage model is based on DOM objects, which usually take much space for persistence. Tamino has the best storage efficiency, and the data size drops to 16% of the original XML documents. This is because Tamino automatically compresses large XML documents, while provides options to select compression or not for small XML documents. Without any compression, DB2 UDB reduces the size to 50% of original XML documents and lies in the middle. DB2 UDB doesn't provide compression options, while DB2 for OS/390 provides several data compression techniques [34] through either hardware or software, and the storage can be reduced to more than 50% for some data. Oracle 9i has recently announced a

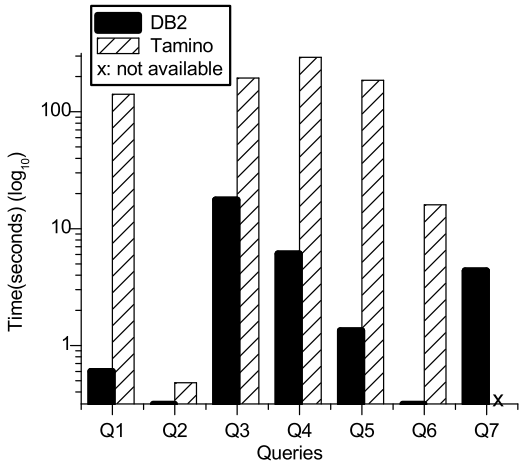

Figure 5. Query Performance of DB2 and Tamino

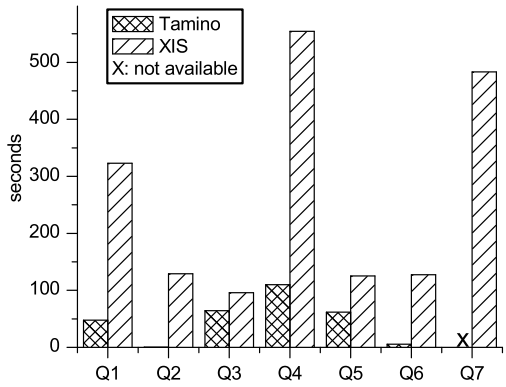

Figure 6. Query Performance of Tamino and XIS (with 1/3 of the Data)

table compression feature, with typical compression ratios of 3:1 [35]. Thus, while compression techniques working directly on XML documents can achieve remarkable compression rates [36, 32], it appears that competitive levels of storage efficiency can also be achieved by storing the DB histories in compressed relational tables. Moreover, the update overhead often incurred in dealing with compressed tables is of no significance for archived histories, and the query performance on uncompressed tables is typically competitive with that of uncompressed ones [35].

Besides data storage, a set of indexes are built for later query comparisons. DB2 has the largest number of indexes due to the number of history tables for each attribute, thus has the largest index size.

Query Performance. Query performance was tested for the following queries:

Q1. Find the total number of employees;
Q2. Retrieve the history of employee '110000';
Q3. Find all the employees throughout '1990-01-01' to '1991-01-01';
Q4. Find the average salary on '1995-01-01';
Q5. Find number of employees who worked in dept 'd005' throughout '1990-01-01' to '1999-01-01';
Q6. Find employees who joined the company on '1995-01-01';
Q7. Find the number of employees who has ever worked with employee '110000' in the same dept.

Among these, Q2 is a history query, Q4 and Q6 are snapshot queries, Q3 and Q5 are interval queries, Q1 is a scan of the database, and Q7 is a join query. The queries are written in SQL for DB2, and a combination of XPath and XQuery for XIS and Tamino. Each query is conducted 5 times and the average is selected.

Two sets of tests are conducted: DB2 and Tamino loaded with all the simulated temporal XML data (Figure 5), and Tamino and XIS loaded with 1/3 of the data (Figure 6) (since queries on XIS will run out of address space when loaded with all the data). Standard (B+ Tree) indexes are created for all nodes/attributes which have values selected.

DB2 offers significant performance advantage for all the queries, and Tamino comes next. The speed of the queries on DB2 is from 3 times to more than 100 times faster than that on Tamino. Moreover, our version of Tamino does not support join query Q7 directly in XPath, which runs very fast on DB2.

Among queries on DB2, selection queries Q2 and Q6 runs the fastest, while interval query Q3 takes the most time.

Among queries on Tamino, history query Q2 runs relatively fast, since XML-documents are clustered by employee history. However, Q2 runs 3 times slower than on DB2. Furthermore, it appears that queries requiring interval comparisons can run slowly on Tamino.

Queries on XIS always take longer than on other databases, especially for snapshot queries, join queries, or queries that need to scan the database.

Temporal Clustering. Performance on snapshot queries such as Q4 can be improved with more effective temporal clustering scheme. In our RDBMS-based archiving scheme, tuples in the historical tables are stored in the order of updates, thus neither temporally clustered nor clustered by objects. Traditional indexes such as B+ Tree will not help on snapshot queries, better temporal clustering is needed. We are currently developing a segment-based archiving scheme which has better temporal clustering, and will boost the performance of most temporal queries, and is also amendable to various compression techniques.

8 Conclusion and Future Work

In this paper, we propose a temporal-grouped data model that can represent the history of a relational database as XML documents (H-documents), without any extension to XML or XQuery. The XML-viewed history of database tables can be stored using a native XML database or using a RDBMS. Simple changes of the underlying relational schema can also be represented and queried with our XML-based approach.

We showed that RDBMS has significant query performance advantages compared to native XML database, which, however, can be more effective in terms of storage because of compression techniques.

To efficiently support temporal queries, we are currently developing a segment-based archiving scheme based on page usefulness, which have better temporal clustering and can significantly boost the performance on most temporal queries [37].

We are also working on the integration of these techniques with those used to manage and query the version history of XML documents [38].

Acknowledgement

The authors would like to thank Shu-Yao Chien and Vassilis Tsotras for many inspiring discussions. This research was in part supported by NSF grant IIS 0070135.

References

[1] G. Ozsoyoglu and R.T. Snodgrass. Temporal and real-time databases: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 7(4):513–532, 1995.

[2] J.F. Roddick. A model for schema versioning in temporal database systems. In *Proc. 19th. ACSC Conf.*, pages 446–452, 1996.

[3] J.F. Roddick. A survey of schema versioning issues for database systems. *Information and Software Technology*, 37(7):383–393, 1995.

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

[5] XQuery 1.0: An XML query language. http://www.w3.org/TR/xquery/.

[6] M. Carey, J. Kiernan, J. Shanmugasundaram, and et al. XPERANTO: A middleware for publishing object-relational data as XML documents. In *VLDB*, 2000.

[7] J.E. Funderburk, G. Kiernan, J. Shanmugasundaram, E. Shekita, and C. Wei. XTABLES: Bridging relational technology and XML. *IBM Systems Journal*, 41(4), 2002.

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

[9] S.S. Chawathe, S. Abiteboul, and J. Widom. Managing historical semistructured data. *Theory and Practice of Object Systems*, 24(4):1–20, 1999.

[10] F. Grandi and F. Mandreoli. The valid web: An XML/XSL infrastructure for temporal management of web documents. In *ADVIS*, 2000.

[11] T. Amagasa, M. Yoshikawa, and S. Uemura. A data model for temporal xml documents. In *DEXA*, 2000.

[12] T. Amagasa, M. Yoshikawa, and S. Uemura. Realizing temporal xml repositories using temporal relational databases. In *CODAS*, pages 63–68, 2001.

[13] C.E. Dyreson. Observing transaction-time semantics with TTXPath. In *WISE (1)*, 2001.

[14] P. Buneman, S. Khanna, K. Ajima, and W. Tan. Archiving scientific data. In *ACM SIGMOD*, 2002.

[15] M.J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, SE-1(4):364–370, 1975.

[16] R. Snodgrass. Temporal object-oriented databases: a critical comparision. *Modern Database Systems: The Object Model, Interoperability and Beyond. Addions-Wesley/ACM Press*, 1985.

[17] E. Bertino, E. Ferrai, and G. Guerrini. A formal temporal object-oriented data model. In *EDBT*, 1996.

[18] D. Beech and B. Mahbod. Generalized version control in an object-oriented database. In *ICDE*, pages 14–22, 1988.

[19] H. Chou and W. Kim. A unifying framework for version control in a cad environment. In *VLDB*, pages 336–344, 1986.

[20] G. Guerrini M. Mesiti E. Camossi, E. Bertino. Automatic evolution of multigranular temporal objects. In *TIME*, 2002.

[21] DB2 XML Extender. http://www-3.ibm.com/software/data/db2/extenders/xmlext/.

[22] M. Fernandez, W. Tan, and D. Suciu. Silkroute: Trading between relations and xml. In *8th Intl. WWW Conf.*, 1999.

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

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

[25] C. Chen and C. Zaniolo. Universal temporal extensions for database languages. In *ICDE*, 1999.

[26] Quip: Software AG's XQuery prototype. http://www.softwareag.com/tamino.

[27] J. Shanmugasundaram and et al. Efficiently publishing relational data as xml documents. In *VLDB*, 2000.

[28] Fusheng Wang and Carlo Zaniolo. Preserving and querying histories of xml-published relational databases. In *ECDM*, 2002.

[29] J. Chomicki, D. Toman, and M.H. Böhlen. Querying ATSQL databases with temporal logic. *TODS*, 26(2):145–178, June 2001.

[30] 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.

[31] Fusheng Wang and Carlo Zaniolo. Representing and querying the evolution of databases and their schemas in XML. In *Intl. Workshop on Web Engineering, SEKE*, 2003.

[32] H. Schning. Tamino - a DBMS designed for XML. In *ICDE*, 2001.

[33] XIS white papers. www.exln.com/products/whitepapers/.

[34] Paolo Bruni and Rama Naidoo. *DB2 for OS/390 and Data Compression*. IBM Redbooks, 1998.

[35] Meikel Poess and Hermann Baer. Decision speed: Table compression in action, http://otn.oracle.com/oramag/webcolumns/2003/techarticles/poess_tablecomp.html.

[36] Hartmut Liefke and Dan Suciu. XMill: an efficient compressor for XML data. pages 153–164, 2000.

[37] S.Y. Chien, V.J. Tsotras, and C. Zaniolo. Efficient schemes for managing multiversion xml documents. *VLDB Journal, Special Issue on XML Data Management*, 2003.

[38] Fusheng Wang and Carlo Zaniolo. Temporal queries in XML document archives and web warehouses. In *TIME-ICTL*, 2003.